Andrew S. Tanenbaum–Albert S. Woodhull

Operációs rendszerek

Tervezés és implementáció

2. kiadás

Panem

A mű eredeti címe: Operating Systems. Design and Implementation. 3rd edition.
0131429388 by Tanenbaum, Andrew S.; Woodhull, Albert S.
Authorized translation from the English language edition
published by Pearson Education, Inc., publishing as Pearson Prentice Hall,

A kiadásért felel a Panem Könyvkiadó Kft. ügyvezetője, Budapest, 2007

Ez a könyv az Oktatási Minisztérium támogatásával, a Felsőoktatási Tankönyv- és Szakkönyv-támogatási Pályázat keretében jelent meg.


OKTATÁSI ÉS KULTURÁLIS MINISZTÉRIUM
OKM

ISBN 978-9-635454-76-1

Fordították: Alexin Zoltán, Bilicki Vilmos, Gombás Éva, Horváth Gyula,
Schrettner Lajos, Tanács Attila
Lektorálta: Kató Zoltán
Szerkesztette: Dávid Krisztina
Tördelte a Pipaszó Bt.
Készült a Dürer Nyomda Kft-ben, Gyulán
Felelős vezető Kovács János ügyvezető igazgató

panem@panem.hu
www.panem.hu

Tartalom

Előszó

Az operációs rendszerekről szóló kézikönyvek többsége az elméletet hangsúlyozza és kevésbé a gyakorlatot. Könyvünk megkísérli a kettő egyensúlyban tartását. Nagy részletességgel tárgyalja az összes alapvető fogalmat, melyek között megtalálhatók a processzusok, processzusok kommunikációi, szemaforok, monitorok, üzenetváltás, ütemezési algoritmusok, bemenet-kimenet, holtpont, eszközvezérlők, memóriakezelés, lapozási algoritmusok, fájlrendszerek, biztonság és védelem módszerei. Emellett részletesen ismertetünk egy Unix-kompatibilis konkrét rendszert is, a MINIX 3-at, melynek teljes forráskódját is rendelkezésre bocsátjuk tanulmányozás céljából. Ez a szerkezet biztosítja, hogy az olvasó ne csak a fogalmakat tanulja meg, hanem azt is, hogyan használhatók ezek valódi operációs rendszerekben.

Könyvünk első, 1987-es kiadása szinte forradalmasította az operációsrendszer-kurzusok oktatását. Addig a kurzusok java része csak elmélettel foglalkozott. A MINIX megjelenésével sok egyetem gyakorlati foglalkozásokat is bevezetett, ezeken a hallgatók egy valódi operációs rendszer belső működését vizsgálhatták. Ezt a törekvést nagyon kívánatosnak véljük, és reméljük, hogy ez a tendencia folytatódik.

A MINIX az első tíz évben sokat változott. Eredetileg a 256 K-s 8088-alapú IBM PC-re terveztük, csupán két hajlékonylemezzel, merevlemez nélkül. Ez a Unix 7-es verzióján alapult. Az idők során a MINIX több irányban is fejlődött: támogatta a 32 bites védett módú, nagy memóriával és nagy kapacitású merevlemezzel ellátott gépeket. A korábbi Unix 7-es verzió helyett a POSIX nemzetközi szabvány lett az alapja (IEEE 1003.1 és ISO 9945-1). Sok új tulajdonság is beépült, megítélésünk szerint túlságosan is sok, mások szerint viszont kevés. Ez vezetett végül a Linux megszületéséhez. A MINIX-et sok más platformra is átvitték, többek között fut Macintosh-, Amiga-, Atari- és SPARC-gépeken. Könyvünk előző, 1997-ben megjelent második kiadását, amely ezt a rendszert tárgyalta, széles körben használták az egyetemeken.

A MINIX népszerűsége töretlen, amit a Google kereső MINIX-találatainak száma is mutat.

A harmadik kiadás sok változtatáson ment keresztül. Az elméleti anyag java részét átdolgoztuk, és elég sok újdonság is bekerült. A legnagyobb változás azonban az új, MINIX 3 nevű rendszer tárgyalásában van, beleértve az új forráskód közre-

adását is. Bár a MINIX 3, ha nem is túl szorosan, a MINIX 2-re épül, több kulcsfontosságú kérdésben alapvetően különbözik tőle.

A MINIX 3 tervezését az a megfigyelés ösztönözte, hogy az operációs rendszerek egyre inkább túlméretezetté, lassúvá és megbízhatatlanná válnak. Más elektronikus eszközökkel, televízióval, mobiltelefonnal, DVD-lejátszóval összehasonlítva sokkal többször omlanak össze működés közben, és rengeteg olyan funkcióval és beállítási lehetőséggel rendelkeznek, amelyeket gyakorlatilag senki sem képes teljesen megismerni vagy jól kezelni. Mindemellett a számítógépes vírusok, férgek, kémprogramok, kéretlen reklámlevelek és a kártékony programok más formái „járványszerű" méreteket öltöttek.

Ezen problémák jó része nagymértékben a mai operációs rendszerek egy alapvető tervezési hibájából, a modularitás hiányából fakad. A teljes operációs rendszer egyetlen nagy, masszív kernel módban futó program, amelynek forráskódja tipikusan több millió C/C++ sorból áll. A több millió sorban akár egyetlen hiba is az egész rendszer hibás működését eredményezheti. A teljes kód hibamentességét elérni lehetetlen, különösen ha figyelembe vesszük, hogy ennek körülbelül 70%-a olyan meghajtóprogramokból áll, amelyeket külső cégek fejlesztenek, az operációs rendszer karbantartóinak hatáskörén kívül.

A MINIX 3-rendszerrel bemutatjuk, hogy ez a monolitikus tervezés nem az egyetlen lehetőség. A MINIX 3 magja csak egy körülbelül 4000 soros futtatható kódból áll, nem milliókból, mint a Windows, a Linux, a Mac OS X vagy a FreeBSD esetében. A rendszer többi része, beleértve a meghajtóprogramokat is (az óra meghajtóprogram kivételével), kicsi, moduláris felhasználói módban működő processzusok gyűjteménye, amelyek mindegyike esetében szigorúan korlátozott, hogy mit tehet és melyik másik processzusokkal kommunikálhat.

Bár a MINIX 3 fejlesztése jelenleg is zajlik, hiszünk abban, hogy a nagymértékben egységbe zárt, felhasználói módban futó processzusok összességeként felépülő operációsrendszer-modell megbízhatóbb operációs rendszerek megalkotását ígéri. A MINIX 3 különösen a kisebb PC-ket helyezi a középpontba, amilyenek általánosan megtalálhatók a harmadik világ országaiban, illetve beágyazott rendszerekben, ahol az erőforrások mindig korlátozottak. Mindenesetre ez a tervezési mód sokkal egyszerűbbé teszi a hallgatók számára az operációs rendszer működésének megértését, mintha egy nagy monolitikus rendszert kellene tanulmányozniuk.

A könyvhöz tartozó CD-ROM melléklet egy élő CD. A CD-ROM-meghajtóba helyezve és a gépet újraindítva pár másodperc után megjelenik a MINIX 3 bejelentkező parancssora. Rendszergazdaként (root) bejelentkezve kipróbálhatjuk a rendszert anélkül, hogy a merevlemezre telepítenénk. Természetesen a merevlemezre telepítés is lehetséges. A telepítés részletes leírása megtalálható a Függelékben.

Mint fentebb említettük, a MINIX 3 gyorsan fejlődik, új változatok gyakran jelennek meg. Az aktuális, CD lemezre írható képfájl letölthető a hivatalos honlapról: *www.minix.org*. Ezen a helyen találunk még nagy mennyiségű új szoftvert, dokumentációt és híreket a MINIX 3 fejlesztéséről. Rendelkezésre áll egy témacsoport a USENET-en, a *comp.os.minix*, ahol MINIX 3 témájú eszmecserére, illetve kérdésekre van lehetőség. Akiknek nincs témacsoport-olvasó szoftverük, a következő honlapon követhetik a témákat: *http://groups.google.com/group/comp.os.minix*.

A MINIX 3 merevlemezre telepítése helyett akár a manapság elérhető PC-szimulátorokon is futtathatjuk a rendszert. Néhány ilyen szimulátort a honlap főoldalán fel is soroltunk.

Rendkívül szerencsésnek mondhatjuk magunkat azért a sok segítségért, amit a munkánk során másoktól kaptunk. Mindenekelőtt Ben Gras és Jorrit Herder végezte az új változat programozási feladatainak nagy részét. Nagyszerű munkát végeztek a szoros határidő mellett, beleértve e-mailek megválaszolását sokszor jóval éjfél után is. Átolvasták a könyv kéziratát is, és sok hasznos megjegyzésük volt. Legmélyebb megbecsülésünk mindkettőjüknek.

Kees Bot szintén nagy segítséget nyújtott az előző változatokhoz, jó alapot biztosítva a további munkához. Kees sok kódrészt írt a 2.0.4 verzióig, hibákat javított, és számtalan kérdésre válaszolt. Philip Homburg írta a hálózati kód nagy részét, valamint számtalan egyéb hasznos módon segített, különösen a kézirathoz készített részletes véleményével.

Sokan, túl sokan ahhoz, hogy itt felsoroljuk őket, írtak kódrészeket már a legkorábbi változatokhoz is, ezzel segítve a MINIX elindulását. Oly sokan voltak és olyan sokféleképpen működtek közre, hogy itt még csak felsorolásuk sem lehetséges, ezért ezúton mondunk köszönetet mindannyiuknak.

Többen átolvasták a kézirat egyes részeit és javaslatokat fűztek hozzá. Gojko Babic, Michael Crowley, Joseph M. Kizza, Sam Kohn Alexander Manov és Du Zhang fogadják külön köszönetünket.

Végül a családjainknak szeretnénk köszönetet mondani. Suzanne tizenhatszor, Barbara tizenötször, Marvin tizennégyszer olvasta át eddig könyvünket. Bár ez már kezd gyakorlattá válni, a szeretet és a segítségnyújtás még inkább méltányolandó. (AST)

Al felesége, Barbara másodjára esik át czen. Támogatása, türelme és jó humora nélkülözhetetlen volt. Gordon türelmes hallgatóság volt. Még most is felemelő az érzés, hogy egy fiú érti és figyel azokra a dolgokra, amik az apját érdeklik. Végül, mostohaunokám, Zain születésnapja egybeesik a MINIX 3 kiadási dátumával. Egy nap ezt még értékelni fogja. (ASW)

Andrew S. Tanenbaum (AST)
Albert S. Woodhull (ASW)

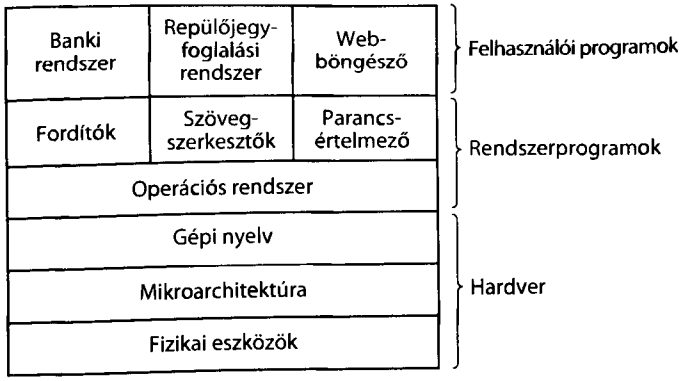
1. Bevezetés

Szoftver nélkül a számítógép valójában egy haszontalan vasdarab. Szoftvertámogatással azonban a számítógép képes információt tárolni, feldolgozni és visszakeresni; zenét és videót lejátszani; elektronikus levelet küldeni, az interneten kutatni; és a fenntartását számos más értékes tevékenységgel megszolgálni. A számítógépszoftverek durván két csoportra oszthatók: rendszerprogramok, amelyek a számítógép saját működését szervezik, és felhasználói programok, amelyek a felhasználó kívánságának megfelelő tényleges munkát végzik. A legalapvetőbb rendszerprogram az **operációs rendszer**, amely a számítógép erőforrásait kezeli és az alapot biztosítja a felhasználói programok írásához. Könyvünk témája az operációs rendszerek tárgyalása. Pontosabban megfogalmazva a MINIX 3 operációs rendszert használjuk modellként a tervezési alapelvek és a tényleges implementáció bemutatására.

Egy modern számítógépes rendszer egy vagy több processzorból, belső memóriából, lemezekből, nyomtatókból, hálózati csatolókból és más bemeneti-kimeneti (B/K – Input/Output, I/O) eszközökből áll. Azaz egy összetett rendszer. Olyan programot írni, amely mindezeket a komponenseket nyomon követi, helyesen, sőt optimálisan használja, rendkívül nehéz feladat. Ha minden programozónak azzal kellene foglalkoznia, hogy a lemezmeghajtók hogyan működnek, hány tucat dolog lehet sikertelen mialatt egy lemezblokkot olvas, akkor valószínűtlen, hogy sok programot egyáltalán megírtak volna.

Már sok évvel ezelőtt kétségtelenül világossá vált, hogy meg kell találni annak a módját, hogy megvédjük a programozókat a hardver bonyolultságától. A fokozatosan kifejlesztett módszer az, hogy a nyers hardver fölé egy szoftverréteget helyezünk, amely a teljes rendszert kezeli és a felhasználó számára egy olyan kapcsolódási felületet vagy **virtuális gépet** alkot, amelyet könnyebb megismerni és programozni. Ez a szoftverréteg az operációs rendszer.

Az operációs rendszer helyét az 1.1. ábra mutatja. Legalul van a hardver, amely sok esetben maga is két vagy több szintből (vagy rétegből) áll. Ez a legalsó réteg tartalmazza az integrált áramköri lapkákból épülő fizikai eszközöket, huzalozást, áramellátást, katódcsöveket és hasonló fizikai eszközöket. Ezek tervezése és működése azonban már a villamosmérnökök területe.



1.1. ábra. *Egy számítógépes rendszer hardverből, rendszerprogramokból és felhasználói programokból áll*

A következő a **mikroarchitektúra szint**, ahol a fizikai eszközöket működési egységekké csoportosítják. Ez a szint tipikusan tartalmaz belső CPU- (központi vezérlőegység) regisztereket és aritmetikai-logikai egységet magában foglaló adatútvonalat. Minden órajelciklusban egy vagy két operandus kerül betöltésre a regiszterekből, melyeket azután az aritmetikai-logikai egység dolgozza fel (például összeadás vagy logikai ÉS művelet). Az eredmény egy vagy több regiszterben tárolódik. Bizonyos gépeken az adatútvonal működését szoftver irányítja, melyet **mikroprogram**nak hívunk. Más gépeken ezt az irányítást közvetlenül a hardveráramkörök végzik.

Az adatútvonal célja utasítások egy halmazának a végrehajtása. Ezek közül néhány végrehajtható egy adatútvonal-ciklus alatt, mások több ciklust igénylenek. Ezek az utasítások regisztereket és más hardveres lehetőségeket használhatnak fel. Az assembly nyelven programozók számára elérhető hardver és az utasítások együttesen alkotják az **utasításkészlet-architektúrát** (**ISA**). Ezt a szintet gyakran **gépi nyelv**nek hívják.

A gépi nyelv általában 50 és 300 közötti utasítást tartalmaz, többségük a gépen belüli adatmozgatásokra, aritmetikai és összehasonlító műveletekre szolgál. Ezen a szinten a bemeneti-kimeneti eszközöket vezérlik oly módon, hogy speciális **eszközregiszter**eket értékekkel töltenek fel. Például a lemezolvasásra úgy utasíthatjuk a lemezvezérlőt, hogy regisztereit feltöltjük a lemezcím, belsőmemória-cím, bájtszám és irány (olvasás vagy írás) értékeivel. Ténylegesen még sok egyéb paraméter is szükséges a végrehajtáshoz, továbbá a művelet befejeztével visszaadott állapotjelző is bonyolult lehet. Sok I/O- (bemeneti/kimeneti) eszköz esetében még az időzítés is jelentős szerepet kap a programozásban.

Az operációs rendszer egyik fő feladata az összes ilyen bonyolultság elrejtése és a programozó számára egy kényelmesebb utasításkészlet biztosítása. Például a read block from file fogalmilag egyszerűbb, mint annak a részletein gyötrődni, hogy a lemezfejeket mozgassuk, várjuk, hogy azok a helyükre érjenek, és így tovább.

Az operációs rendszer felett van a rendszerszoftver maradék része. Itt találjuk a parancsértelmezőt (shell), az ablakkezelő rendszert, fordítókat, szövegszerkesztőket és a hasonló, alkalmazásoktól független programokat. Fontos tudnunk, hogy ezek a programok semmiképpen sem az operációs rendszer részei, bár általában a számítógépgyártótól származnak a gépre előre telepítve vagy az operációs rendszerrel egy csomagban, ha a telepítésre a vásárlás után kerül sor. Ez döntő, de kényes kérdés. Az operációs rendszer (általában) a szoftvernek az a része, amely **kernel mód**ban vagy **felügyelt mód**ban fut. Ezeket hardver védi a felhasználói kontárkodástól (eltekintve néhány öregebb mikroprocesszortól, amelyeknek hardvervédelme egyáltalán nincs). A fordítók, szövegszerkesztők **felhasználói mód**ban futnak. Ha egy felhasználónak nem tetszik egy adott fordító, nyugodtan megírhatja a sajátját, ha úgy dönt; de nem írhat saját óramegszakítás-kezelőt, amely az operációs rendszer része, és amelyet általában hardver véd a felhasználók módosítási kísérleteivel szemben.

Ez a különbség azonban elmosódik néhány beágyazott rendszer esetében (ahol nem érhető el kernel mód) vagy értelmezőprogrammal futtatott rendszerben (amilyenek például azok a Java-alapú rendszerek, ahol a komponensek szétválasztására értelmezőt használnak a hardver helyett). Hagyományos számítógépek esetében az operációs rendszer azonban mégis az, ami kernel módban fut.

Sok rendszer esetében azonban vannak olyan felhasználói módban futó programok is, amelyek az operációs rendszer működését segítik, vagy privilegizált feladatot hajtanak végre. Például gyakran elérhető olyan program, amely lehetővé teszi a felhasználó számára a jelszó megváltoztatását. Ez a program nem része az operációs rendszernek és nem kernel módban fut, de egyértelműen érzékeny műveletet hajt végre, amit speciális módon kell védeni.

Bizonyos rendszerekben, amilyen a MINIX 3 is, ez az ötlet a végletekig fokozódik, és olyan részek, amelyek hagyományosan az operációs rendszer részét képezik (amilyen a fájlrendszer), felhasználói szinten futnak. Ezekben a rendszerekben nehéz egyértelmű határvonalat húzni. Minden, ami kernel módban fut, nyilvánvalóan az operációs rendszer része, de néhány ezen kívül futó program is vitathatatlanul része, vagy legalábbis szorosan kapcsolódik hozzá. A MINIX 3 esetében például a fájlrendszer egyszerűen egy nagy, felhasználói módban futó C program.

Végül a rendszerprogramok fölé épülnek a felhasználói programok. Ezeket a programokat a felhasználók vásárolják vagy írják saját problémáik megoldására, ilyenek a szövegszerkesztés, táblázatkezelés, mérnöki számítások vagy információk tárolása adatbázisban.

1.1. Mi az az operációs rendszer?

A legtöbb számítógép-felhasználó szert tett már némi operációs rendszerekbeli tapasztalatra, mégis nehéz pontosan leszögezni, mi is az az operációs rendszer. A probléma egyik része az, hogy az operációs rendszer két, alapjában különböző feladatot, a gép kiterjesztését és az erőforrások kezelését látja el, és attól függően, hogy ki beszél róla, többnyire csak az egyik vagy a másik funkcióról hallunk. Nézzük most mindkettőt.

1.1.1. Az operációs rendszer mint kiterjesztett gép

Amint már korábban említettük, a legtöbb számítógép **architektúrája** (utasításkészlet, memóriaszervezés, I/O-rendszer, sínstruktúra) a gépi nyelv szintjén primitív és a programozása, különösen a bevitel/kivitel, kényelmetlen. Hogy ezt világossá tegyük, röviden tekintsük át, hogyan történik a hajlékonylemez I/O a NEC PD765-kompatibilis vezérlőlapkán (chipen), amelyet sok Intel-alapú személyi számítógépben használnak. (A „hajlékonylemez" és a „diszkett" kifejezéseket könyvünkben végig azonos értelemben használjuk.) A vezérlő 16 utasítással rendelkezik, mindegyikük 1 és 9 bájt közötti értéket tölt egy eszközregiszterbe. Az utasítások adatok olvasására, írására, a lemezfejek mozgatására, a pályák formázására, továbbá a vezérlő és a meghajtók inicializálására, érzékelésére, alaphelyzetbe állítására és bemérésére szolgálnak.

A két alapvető utasítás a read és write, mindkettőhöz 13 paraméter szükséges, melyek 9 bájton vannak elrendezve. Ezek a paraméterek adják meg az olyan mezőket, mint a beolvasandó lemez blokkcíme, a pályánkénti szektorok száma, a fizikai hordozón alkalmazott tárolási mód, a szektorok közötti hézag mérete, és hogy mi a teendő egy „törölt adat címe" jelzéssel. Ne bánkódjunk amiatt, ha nem értjük ezeket a mély értelmű dolgokat; éppen az a lényeg, hogy ez meglehetősen titokzatos. Amikor a művelet befejeződött, a vezérlő visszaad 23 állapot- és hibamezőt, 7 bájton elrendezve. Ha még ez sem lenne elég, a programozónak arra is állandóan ügyelnie kell, hogy a motor jár, vagy nem. Ha a motor ki van kapcsolva, be kell kapcsolni (hosszú felpörgésre való várakozással), mielőtt adatot olvashatunk vagy írhatunk. De a motort nem lehet túl sokáig bekapcsolva hagyni, mert a diszkett elkopik. Így a programozó arra kényszerül, hogy foglalkozzék a hosszú felpörgési idő és a diszkett kopása (és a rajta lévő adatok elvesztése) közötti választás dilemmájával.

Anélkül, hogy a *valódi* részletekbe mennénk, tisztában kell lennünk azzal, hogy egy átlagos programozó nem akar túl mélyen belemerülni a hajlékonylemez programozásába (sem a merevlemezébe, amelyik legalább olyan bonyolult és egészen más). Ehelyett a programozó egy egyszerű, magas szintű absztrakcióval akar foglalkozni. A lemez esetében egy szokásos absztrakció lehet az, hogy a lemez névvel ellátott állományok gyűjteményét tárolja. Minden állomány megnyitható olvasásra vagy írásra, ezután olvasható vagy írható, végül lezárandó. Az olyan részletek, hogy a tárolás vajon a módosított frekvenciamodulációt alkalmazza-e, és hogy a motor aktuális állapota milyen, nem jelenhetnek meg a felhasználó számára nyújtott absztrakcióban.

Az a program, amelyik a programozó elől elrejti a valódi hardvert, és egy egyszerű képet ad a névvel ellátott, olvasható és írható állományokról, természetesen az operációs rendszer. Ugyanúgy, ahogy az operációs rendszer a lemezhardvertől védi meg a programozót és nyújt egy egyszerű állományorientált kapcsolatot, képes elrejteni sok nemkívánatos foglalatosságot a megszakítások, az időzítések, a memóriaszervezés és más alacsony szintű tulajdonság kezelése kapcsán. Minden esetben az operációs rendszer által nyújtott absztrakció egyszerűbb és könnyebben használható, mint a mögötte lévő hardver.

Ebből a nézőpontból az operációs rendszer feladata az, hogy a felhasználónak egy olyan egyenértékű **kiterjesztett gépet** vagy **virtuális gépet** nyújtson, amelyiket egyszerűbb programozni, mint a mögöttes hardvert. Könyvünk annak részleteit tanulmányozza, hogy mindezt hogyan teljesíti az operációs rendszer. Dióhéjban összefoglalva az operációs rendszer különféle szolgáltatásokat nyújt, amelyeket a programok speciális, rendszerhívásoknak nevezett utasítások segítségével érhetnek el. Néhány gyakrabban használt rendszerhívást a fejezet további részében meg fogunk vizsgálni.

1.1.2. Az operációs rendszer mint erőforrás-kezelő

A „felülről lefelé" nézőpontból az operációs rendszer elsősorban egy kényelmes csatlakozási felület a felhasználók számára. A másik, „alulról felfelé" nézőpont azt tartja, hogy az operációs rendszer azért van, hogy az összetett rendszer minden egyes részét kezelje. A modern számítógépek processzorokból, memóriákból, órákból, lemezekből, egerekből, hálózati csatolókból, nyomtatókból és egyéb eszközök bő választékából állnak. Az utóbbi nézőpont szerint az operációs rendszer feladata az, hogy a különböző, a processzorokért, a memóriáért és az I/O-eszközökért versenyző programok számára szabályos és felügyelt módon biztosítsa ezeket.

Képzeljük el, mi történne, ha három, ugyanazon a számítógépen futó program egyidejűleg ugyanazon az egy nyomtatón próbálná kinyomtatni eredményeit. Az első néhány sor az 1. programtól, a következő néhány a 2. programtól, azután valamennyi a 3. programtól származna, és így tovább. Az eredmény káosz. Az operációs rendszer tud rendet teremteni az esetleges káoszban azzal, hogy a nyomtatóra irányított minden eredményt átmenetileg tárol a lemezen (spooling). Amikor egy program befejeződött, az operációs rendszer az eredményeit átmásolja a nyomtatóra abból a lemezállományból, ahol tárolta, miközben a többi program folytathatja saját eredményeinek előállítását, figyelmen kívül hagyva azt a tényt, hogy az eredmény valójában (még) nem a nyomtatóra megy.

Ha a számítógépen (vagy a hálózaton) több felhasználó van, még inkább szükség van a memória, az I/O-eszközök és más erőforrások kezelésére és védelmére, mert enélkül a felhasználók zavarhatnák egymást. Ezenfelül a felhasználók gyakran nemcsak a hardveren, de információkon (állományok, adatbázisok stb.) is osztoznak. Röviden ebből a nézőpontból az operációs rendszer elsőrendű feladata, hogy nyilvántartsa, ki melyik erőforrást használja, hogy teljesítse az erőforráskéréseket, hogy mérje a használatot, és hogy a különböző programok és felhasználók ellentmondásos kéréseit egyeztesse.

Az erőforrás-kezelés az erőforrások kétféle megosztását foglalja magában: az időalapút és a téralapút. Az időosztásos erőforrásokat a különböző programok vagy felhasználók felváltva használják. Először az egyikük kapja meg, majd egy másikuk, és így tovább. Például ha egy CPU-n egyszerre több program is fut, akkor azt az operációs rendszer először az egyik programnak osztja ki, majd mikor az már elég ideig futott, egy másik kapja meg, majd egy újabb, majd egyszer csak

újra az első. Az időosztás mikéntje, vagyis hogy ki következzen és mennyi ideig, az operációs rendszer feladata. Egy újabb időosztásos példa a nyomtató megosztása. Amikor több nyomtatási feladat is összegyűlik egy nyomtatóra, el kell dönteni, hogy melyik nyomtatása következzen.

A másik fajta megosztás a téralapú megosztás. A felváltott használat helyett itt mindenki kap egy részt az erőforrásból. Például a központi memória normál esetben fel van osztva számos futó program között, így mindegyikük egyszerre lehet rezidens (például a CPU felváltva történő felhasználásához). Feltételezve azt, hogy elegendő memória áll rendelkezésre több program számára is, sokkal hatékonyabb ezeket a memóriában tartani, mint a teljes memóriát egynek kiosztani, különösen ha annak csak kis részére van szüksége. Természetesen ez felveti a korrektség, a védelem és egyebek kérdéseit, amiket az operációs rendszernek kell megoldania. Egy másik térosztásos erőforrás a (merev)lemez. Sok rendszerben egyetlen lemez képes sok felhasználó állományait egyidejűleg tárolni. A lemezterület lefoglalása és annak nyilvántartása, hogy ki melyik lemezblokkot használja, az operációs rendszer egy tipikus erőforrás-kezelési feladata.

1.2. Az operációs rendszerek története

Az operációs rendszerek évek hosszú során alakultak ki. A következő szakaszokban röviden áttekintjük a fejlődés fontosabb mozzanatait. Mivel az operációs rendszerek történetileg szorosan kötődtek azon számítógépek architektúrájához, amelyen futottak, az egymást követő számítógép-generációkon keresztül mutatjuk meg, hogy milyenek voltak. Az operációs rendszerek generációinak és a számítógépek generációinak ez a kötődése laza, mégis ad némi áttekintést.

Az első valóban digitális számítógépet Charles Babbage (1792–1871) angol matematikus tervezte. Bár Babbage élete és vagyona javát „analitikai gépének" felépítési kísérleteire fordította, az soha nem működött megfelelően, mivel teljesen mechanikus volt, és korának technológiája nem tudta előállítani a szükséges kerekeket, fogaskerekeket, fogakat. Szükségtelen mondanunk, gépén nem volt operációs rendszer.

Érdekes történeti mellékkörülmény, hogy Babbage rájött, szüksége van szoftverre a gépéhez, ezért Ada Lovelace személyében a világ első programozójaként alkalmazott egy fiatal nőt, aki Lord Byron, a híres brit költő lánya volt. Az Ada® programozási nyelvet róla nevezték el.

1.2.1. Az első generáció (1945–1955): vákuumcsövek és kapcsolótáblák

A digitális számítógépek építésében Babbage sikertelen erőfeszítéseit követően nem sok előrehaladás történt a második világháborúig. Az 1940-es évek közepén – mások mellett – Howard Aiken (Harvard), Neumann János (Institute for

Advanced Study, Princeton), J. Presper Eckert és John William Mauchley (University of Pennsylvania), Konrad Zuse (Németország) számológépek építésében értek el sikereket. Az első gépek mechanikus reléket használtak, de túlságosan lassúak voltak, egy-egy ciklus másodpercekig tartott. A reléket később vákuumcsövek váltották fel. Ezek a gépek hatalmasak voltak, egész termeket betöltöttek több tízezer vákuumcsővel, de mégis több milliószor lassabbak voltak, mint a ma használatos legolcsóbb személyi számítógépek.

Ezekben a korai időkben minden egyes gépet egy külön csapat tervezett, épített, programozott, kezelt és végezte a karbantartását. Abszolút gépi nyelven folyt a programozás, gyakran az alapvető funkciók vezérlésére kapcsolótáblákat huzaloztak. A programozási nyelvek (még az assembly nyelv is) ismeretlenek voltak. Senki sem hallott még ekkor operációs rendszerekről. A programozók szokásos munkamódszere az volt, hogy feliratkoztak a falon függő beosztástáblán egy időintervallumra, azután lejöttek a gépterembe, behelyezték a saját kapcsolótáblájukat a számítógépbe, és a következő néhány órában azt remélték, hogy a 20000 körüli vákuumcső közül a futás alatt egy sem gyullad ki. A problémák többnyire egyszerű numerikus számítások, mint a sin-, cos- és logaritmustáblák kikínlódása voltak.

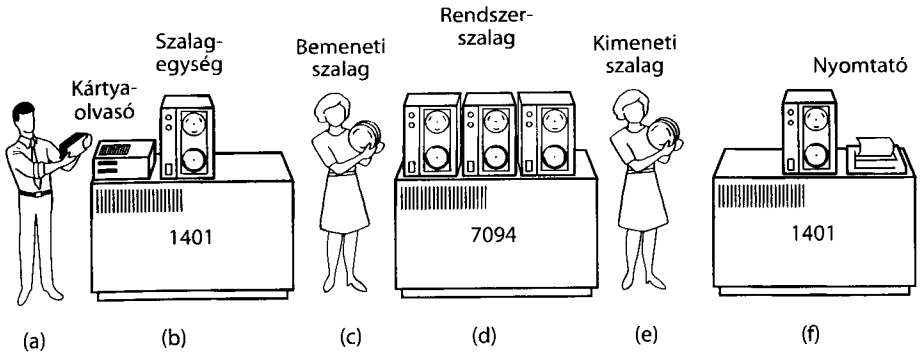
Az 1950-es évek elejére a módszer kicsit javult a lyukkártyák bevezetésével. Ekkor már kártyán lehetett programokat írni, és a kapcsolótáblák helyett ezek beolvasásával működtették a gépet; egyebekben a módszer nem változott.

1.2.2. A második generáció (1955–1965): tranzisztorok és kötegelt rendszerek

A tranzisztorok megjelenése az 1950-es évek közepén alaposan megváltoztatta a képet. A számítógépek eléggé megbízhatók lettek ahhoz, hogy gyárthatókká és eladhatókká váljanak annak a reményében, hogy elég hosszú ideig működőképesek maradnak és a vásárlóknak valami hasznos munkát végeznek. Itt különültek el először egymástól a tervezők, a gyártók, a kezelők, a programozók és a karbantartó személyzet.

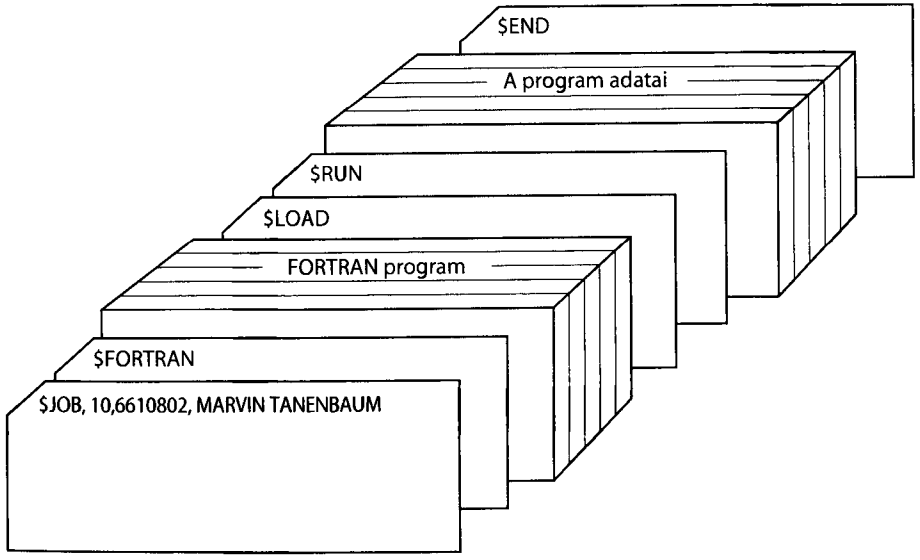
Ezek a gépek, amelyeket ma **nagyszámítógépek**nek (vagy mainframe-eknek) hívunk, különleges légkondicionált termekbe voltak zárva és szakképzett kezelők működtették őket. Csak nagy vállalatok, főbb kormányzati szervek vagy egyetemek tudták előteremteni a több millió dolláros árat. Ahhoz, hogy egy **feladat**ot (program vagy programok egy csoportja) futtatni lehessen, először a programozó papíron megírta a programot (FORTRAN vagy akár assembly nyelven), ezt kártyákra lyukasztották, a kártyacsomagot a beviteli terembe vitték és átadták az egyik kezelőnek, majd elmentek kávézni, míg az eredmény elkészült.

Amikor a számítógép végzett egy éppen futó feladattal, egy kezelő átment a nyomtatóhoz, letépte az eredményt és átvitte a kiviteli terembe, így a programozó később hozzájuthatott. Azután felvett egy új kártyacsomagot, amelyet a beviteli teremből hoztak, és beolvastatta. Ha a FORTRAN fordítóra volt szükség, a kezelő elhozta az állománytároló rekeszekből, és azt is beolvastatta. A gépidő java része azzal telt, hogy a kezelő a gépteremben menetelt.



1.2. ábra. *Egy korai kötegelt rendszer. (a) A programozók kártyáikat az 1401-eshez juttatják. (b) Az 1401-es szalagra olvassa a feladatköteget. (c) A gépkezelő átviszi a bemeneti szalagot a 7094-eshez. (d) A 7094-es végrehajtja a számolást. (e) A gépkezelő átviszi a kimeneti szalagot az 1401-eshez. (f) Az 1401-es kinyomtatja az eredményeket*

Nem csoda, ha a berendezés magas ára miatt hamarosan gondolkodni kezdtek azon, hogyan csökkentsék az elvesztegetett időt. Az általánosan elfogadott megoldás a **kötegelt rendszer** lett. Az ötlet az volt, hogy a bemeneti teremben gyűjtsünk össze egy kötegre való feladatot, ezeket olvastassuk mágnesszalagra egy (viszonylag) olcsó számítógéppel, mint például az IBM 1401-es, amely kitűnő volt kártyaolvasásban, szalagmásolásban, nyomtatásban, de nem jeleskedett numerikus számításokban. A másik, sokkal drágább gépet, mint például az IBM 7094-es, használjuk a tényleges számításokra. A megoldást az 1.2. ábra mutatja.



1.3. ábra. *Egy szokásos FMS-feladat*

A feladatköteg kb. egyórás összegyűjtése után a szalagot visszatekerték, átvitték a gépterembe és behelyezték a szalagolvasóba. A kezelő elindított egy speciális programot (a mai operációs rendszer elődjét), amely beolvasta az első feladatot és elindította a futtatását. Nyomtatás helyett az eredményeket egy másik szalagra írta. A feladatok befejeztével az operációs rendszer automatikusan beolvasta és elindította a szalagon következő feladatot. Amikor az egész köteggel végzett, a kezelő kivette a bemeneti és eredményszalagokat, a bemeneti szalagot kicserélte a következő köteggel, az eredményszalagot átvitte az 1401-esre **off-line** (azaz a főgéptől független) nyomtatásra.

Egy tipikus bemeneti feladat felépítését az 1.3. ábra mutatja. Egy $JOB kártyával kezdődik; ez specifikálja a maximális futási időt percekben, a terhelendő számlaszámot és a programozó nevét. Ezt követi a $FORTRAN kártya, jelezve az operációs rendszernek, hogy töltse be a FORTRAN fordítót a rendszerszalagról. Mögötte a lefordítandó program, majd egy $LOAD kártya, amely utasítja az operációs rendszert a most fordított célprogram betöltésére. (A lefordított programokat gyakran munkaszalagra írták és explicit kérésre töltötték be.) Ezután jön a $RUN kártya, amely a program futtatását kéri a mögötte található adatokkal. Végül a $END kártya jelzi a feladat végét. Ezek az egyszerű vezérlőkártyák voltak a mai feladatvezérlő nyelvek és parancsértelmezők előfutárai.

Az óriási második generációs számítógépeket többnyire tudományos és mérnöki számításokra használták, például fizikai és mérnöki feladatokban gyakran előforduló parciális differenciálegyenletek numerikus megoldására. Főként FORTRAN és assembly nyelven programoztak. Tipikus operációs rendszerek voltak az FMS (Fortran Monitor System) és az IBSYS, az IBM operációs rendszere a 7094-esen.

1.2.3. Harmadik generáció (1965–1980): integrált áramkörök és multiprogramozás

Az 1960-as évek elejéig a számítógépgyártók két, egymástól teljesen független és egymással nem kompatibilis termékvonallal rendelkeztek. Egyrészt voltak szóorientált, általános célú tudományos számítógépek, például a 7094-es, amelyeket tudományos és műszaki számításokra használtak. Másrészt voltak karakterorientált üzleti célú gépek, például az 1401-es; ezeket bankok és biztosítók széles köre használta szalagrendezésekre, nyomtatásra.

A két különböző termékvonal fejlesztése és fenntartása a gyártók számára költséges vállalkozás volt. Ráadásul a legtöbb új számítógép-vásárló először egy kis gépet igényelt, bár ezt később kinőtte, és egy olyan nagyobb gépet akart, amelyen az összes korábbi programjai futnak, de gyorsabban.

Az IBM egy tollvonással próbálta megoldani mindkét problémát: bevezette a System/360 rendszert. A 360-as sorozat az 1401-es méretűtől a 7094-eseknél sokkal erősebb, szoftverszinten kompatibilis gépekből állt. A gépek csak az árukban és a teljesítményükben (maximális memória, processzorsebesség, a megengedett I/O-eszközök száma stb.) különböztek. Mivel az összes gépnek ugyanaz volt a felépítése és utasításkészlete, elméletileg az egyikre írt program bármelyik másikon

is futhatott. Ráadásul a 360-ast mind tudományos (azaz numerikus), mind üzleti számítások végrehajtására is tervezték. Ezzel a gépek egyetlen családja minden vevő kívánságát teljesíteni tudta. A következő években az IBM kihozta a 360-as sorozat kompatibilis utódait, a 370, 4300, 3080, a 3090 és a Z jelzésű, modernebb technológiával készült családokat.

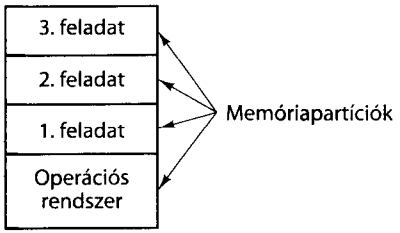
A 360-as volt az első olyan nagyobb sorozat, ahol a (kisméretű) integrált áramköröket (IC) alkalmazták, ezzel óriási ár-/teljesítményelőnyhöz jutottak a második generációs gépekhez képest, amelyek egyedi tranzisztorokból épültek. A siker azonnali volt, és a kompatibilis számítógépcsalád ötletét a nagyobb gyártók rövidesen átvették. Még mindig használják ezeknek a gépeknek a leszármazottait számítóközpontokban. Manapság hatalmas méretű adatbázisokat kezelnek (például repülőjegy-foglaláshoz), vagy olyan webszerverekként működnek, ahol másodpercenként több ezer kérést kell feldolgozni.

Az „egy család" ötlet volt a sorozat legnagyobb erőssége, egyben a legnagyobb gyengesége is. Az elképzelés szerint minden szoftver, beleértve az **OS/360** operációs rendszert is, minden modellen működőképes. A kis gépeken, amelyek éppen csak a kártyáról szalagra másolást végezték (mint az 1401-esek), és nagy rendszereken, amelyek időjárás-előrejelzési és más óriási számításokat készítettek (mint a 7094-esek). Egyaránt jónak kellett lennie kevés és sok külső egységgel rendelkező rendszeren. Működnie kellett üzleti és tudományos környezetben. És mindenekelőtt hatékonynak kellett lennie minden felhasználási területen.

Nem volt rá módszer, hogy az IBM (vagy bárki más) mindezen ellentmondásos követelményeknek megfelelő szoftveregységeket tudjon írni. Az eredmény egy hatalmas és rettenetesen bonyolult, az FMS-nél mintegy két-három nagyságrenddel nagyobb operációs rendszer. A programozók ezrei által írt, több millió soros assembly programból állt, ezerszám tartalmazott hibákat, melyek kiküszöbölésére folyamatosan áradtak a verziók. Minden új verzió néhány hibát javított, néhányat behozott, így az idők során a hibák száma valószínűleg konstans maradt.

Az OS/360 egyik tervezője, Fred Brooks írt egy szellemes és találó könyvet (Brooks, 1995) az OS/360-ról szerzett tapasztalatairól. Nem ismertethetjük itt a könyvet, legyen elég annyi, hogy a borítóján egy szurokcsapdába ragadt történelem előtti vadállatcsorda látható. Silberschatz és Galvin könyvének (Silberschatz et al., 2004) borítója hasonlóképpen dinoszauruszként ábrázolja az operációs rendszereket.

A hatalmas méret és a problémák ellenére az OS/360 és a többi számítógépgyártó harmadik generációs rokon operációs rendszerei elég jól teljesítették a vevők többségének igényeit. Elterjesztettek néhány olyan kulcsfontosságú módszert is, amelyek hiányoztak a második generációs operációs rendszerekből. Valószínűleg a **multiprogramozás** volt közülük a legfontosabb. Amikor az aktuális feladat várakozott a szalag vagy más I/O-művelet teljesítésére a 7094-esen, a processzor üresjáratra állt az I/O befejeztéig. Erősen CPU-intenzív tudományos számítások esetén az I/O ritka, így az elveszett idő jelentéktelen. Üzleti adatfeldolgozások esetében az I/O várakozási idő elérheti az összidő 80-90 százalékát is, ezért valamit tenni kellett a CPU üresjárati idejének a csökkentésére.



1.4. ábra. *Multiprogramozásos rendszer, három feladat van a memóriában*

Az a megoldás alakult ki, hogy a memóriát szeletekre particionálták, minden partícióhoz egy-egy feladatot rendeltek, ahogy az 1.4. ábra mutatja. Amíg egy feladat I/O teljesítésére várt, egy másik használhatta a CPU-t. Ha elég feladatot tudtak egyszerre tárolni a belső memóriában, a CPU-t az idő közel 100 százalékában is foglalkoztathatták. Az, hogy több feladat van egyidejűleg a memóriában, megkívánja azt a speciális hardvert, amely megvédi a feladatokat a többiek beavatkozásától és rongálásától; a 360-ast és a többi harmadik generációs gépet felszerelték ezzel a hardverrel.

A harmadik generációs operációs rendszerek egy másik jelentős tulajdonsága az a képesség volt, hogy a kártyákról a feladatokat a számítógépteremben való megjelenésükkor azonnal lemezre tudták olvasni. Valahányszor egy futó feladat befejeződött, az operációs rendszer egy új feladatot tudott a lemezről az immár üres partícióba tölteni és elindítani. A technikát **háttértárolás**nak (**spooling**, Simultaneous Peripheral Operation On Line) nevezik, és a kimenetre is alkalmazták. Háttértárolással az 1401-esre már nem volt szükség, a szalagok alkalmazásainak többsége eltűnt.

Bár a harmadik generációs operációs rendszereket jól felszerelték nagy tudományos számítások és komoly üzleti adatfeldolgozások végrehajtására, mégis megmaradtak kötegelt rendszereknek. Sok programozó visszasírta az első generációs időket, amikor órákon keresztül az egész gépet birtokolta, programjait gyorsan tesztelhette. A harmadik generációs rendszereken a feladat leadása és az eredmények visszakapása között több óra is eltelhetett, így egy félreütött vessző képes volt fordítási hibát okozni, amivel a programozó fél napja elveszett.

A gyors válaszidő iránti igény egyengette az **időosztás (timesharing)**, a multiprogramozás egy olyan variációjának útját, amikor is minden felhasználónak saját on-line terminálja van. Ha 20 felhasználó jelentkezett be egy időosztásos rendszerbe, és 17 közülük gondolkodik, beszélget vagy issza a kávéját, akkor a CPU a maradék három, kiszolgálást igénylő feladathoz rendelhető. Akik programokat tesztelnek, általában rövid parancsokat (például egy ötlapos eljárás* fordítását) adnak ki, nem pedig hosszúakat (például millió rekordos adathalmaz rendezését), így a számítógép nagyszámú felhasználót képes gyorsan, interaktív módon kiszolgálni, miközben esetleg nagy kötegelt feladatokon is dolgozik a háttérben, amikor a CPU üresjáratban volna. Az első igazi időosztásos rendszert (CTSS) az M.I.T.

* Könyvünkben az eljárás, a szubrutin és a függvény kifejezéseket felváltva, azonos értelemben használjuk.

fejlesztette ki egy speciálisan átalakított 7094-esen (Corbató et al., 1962). Az időosztás azonban csak akkor lett igazán népszerű, amikor a harmadik generációban széles körben elterjedt a hardvervédelem.

A CTSS sikerén felbuzdulva, az M.I.T, a Bell Labs és a General Electric (akkoriban jelentős számítógépgyártó) nekifogtak egy „számítógép-szolgáltató" fejlesztésének; ez egy gép lett volna, amely egyidejűleg több száz időosztásos felhasználót szolgál ki. Modelljük az elektromos hálózat volt – ha áramot akarunk, egyszerűen csatlakoztassunk egy falidugót, és remélhetjük, hogy ott annyi áramot kapunk, amennyi kell. A **MULTICS (MULTIplexed Information and Computing Service)** névre keresztelt rendszer tervezői egy hatalmas számítógépet álmodtak meg, amelyhez Bostonban minden lakos hozzáfér. Abban az időben csak álom volt az, hogy 30 év múlva az ő GE–645-ös típusuk teljesítményét messze meghaladó személyi számítógépek vásárolhatók majd ezer dollár alatti áron, amilyen álom manapság az óceán alatt futó, hangsebességnél gyorsabb transzatlanti vasút ötlete.

A MULTICS sikere vegyes volt. Felhasználók százainak kiszolgálására tervezték egy Intel 80386-alapú PC-nél alig nagyobb kapacitású gépre, bár az I/O-képessége jóval nagyobb volt annál. Ez nem annyira őrült ötlet, amennyire hangzik, lévén az emberek akkoriban tudták, hogyan kell kisméretű és hatékony programokat írni – ez a képesség a későbbiekben eltűnni látszik. Több ok is volt, ami miatt a MULTICS nem vette át a világuralmat; ezek közül nem elhanyagolható az a tény, hogy PL/I nyelven írták, a PL/I fordító pedig éveket késett és alig működött, amikor végül elkészült. Ráadásul a MULTICS roppantul nagyra törő volt a korához, akárcsak Charles Babbage analitikus gépe a XIX. században.

A MULTICS sok eredeti ötletet hozott a számítógépes szakirodalomba, de komoly termékké és sikeres üzletté válása sokkal nehezebb volt, mint azt bárki is gondolta. A Bell Labs kiszivárgott a projektből, a General Electric pedig teljesen kilépett a számítógépes üzletágból. Az M.I.T azonban kitartott és működésre bírta a MULTICS-ot. Kereskedelmi termékként végül az a cég árulta (Honeywell), amely a GE számítógépes üzletágát megvette, és nagyjából 80 jelentős cég és egyetem telepítette világszerte. Bár kevesen voltak, a MULTICS felhasználói rendkívüli módon lojálisak maradtak. Példaként a General Motors, a Ford és az Egyesült Államok Nemzetbiztonsági Hivatala csak az 1990-es évek vége felé állították le MULTICS rendszereiket. Az utolsó működő MULTICS-ot a Kanadai Védelmi Minisztériumban 2000 októberében állították le. Az üzleti siker elmaradása ellenére a MULTICS a későbbi rendszerekre óriási hatást gyakorolt. Bővebb információ található (Corbató et al., 1972; Corbató és Vyssotsky, 1965; Daley és Dennis, 1968; Organick, 1972; Saltzer, 1974). Van egy jelenleg is aktív honlapja, a *www.multicians.org*, ahol magáról a rendszerről, a tervezőiről és a felhasználóiról nagy mennyiségű információ érhető el.

A „számítógép-szolgáltató" kifejezés nem volt hallható többé, de az ötlet az elmúlt években új életre kelt. Legegyszerűbb formájában egy cég vagy egy osztályterem személyi számítógépei vagy **munkaállomásai** (a legkorszerűbb PC-k) **helyi hálózaton (Local Area Network, LAN)** keresztül egy **fájlkiszolgáló**hoz csatlakozhatnak, ahol az összes programot és adatot tárolják. A rendszergazdának így csak egyféle program- és adattelepítésről, illetve védelemről kell gondoskodnia.

Egy működésképtelen PC vagy munkaállomás esetében könnyen újratelepítheti a lokális szoftvereket anélkül, hogy aggódnia kellene a lokális adatok elérése vagy megőrzése miatt. Heterogénebb környezetben egy szoftverosztály, az úgynevezett **közvetítő szoftver (middleware)** alakult ki a lokális felhasználók és az általuk használt, távoli gépeken lévő fájlok, programok és adatbázisok közötti rés áthidalására. A közvetítő szoftver segítségével a felhasználó PC-je vagy munkaállomása lokálisnak érzékeli a hálózatba kapcsolt számítógépeket, és egységes felhasználói felületet biztosít annak ellenére, hogy sokféle különböző kiszolgáló, PC, valamint munkaállomás lehet használatban. Erre példa a világháló, a WWW (World Wide Web). Egy böngészőprogram egységes módon jeleníti meg a dokumentumokat, amelyek szöveges része egy kiszolgálóról, a képek egy másikról, a megjelenés módját definiáló stíluslap pedig akár egy harmadikról is származhat. Cégek és egyetemek általánosan használnak webes felületeket adatbázisok elérésére vagy programok futtatására olyan gépeken, amelyek egy másik épületben vagy akár másik városban találhatók. A közvetítő szoftver az **elosztott rendszerek** operációs rendszerének tűnik, bár igazából egyáltalán nem az, tárgyalása pedig túlmutat könyvünk keretein. Az elosztott rendszerekről bővebb információt Tanenbaum és Van Steen (Tanenbaum és Van Steen, 2002) könyvében találhatunk.

A harmadik generáció másik nagy fejlődési ága a miniszámítógépek csodálatosan gyors növekedése, elsőként 1961-ben a DEC PDP–1. A PDP–1 18 bites szavakból 4 K memóriával rendelkezett, ára viszont már csak 120000 dollár volt (a 7094-es árának kevesebb mint 5 százaléka), úgy vitték, mint a cukrot. Bizonyos nem numerikus számításokban majdnem olyan gyors volt, mint a 7094-es, és egy új iparág születését jelentette. Gyorsan követték a PDP más sorozatai (mind-mind inkompatibilisek, nem úgy, mint az IBM-család), a csúcs a PDP–11.

Ken Thompson, a Bell Labs MULTICS projektben dolgozó számítástudósa ráakadt egy használatlan kis PDP–7-re, és elkezdte a MULTICS lecsupaszított, egyfelhasználós változatának a megírását. Munkája a **Unix** operációs rendszer fejlesztésébe torkollott, amely a tudományos világ, kormányhivatalok és számos vállalat körében igen népszerűvé vált.

A Unix történetét mások mesélik el (például Salus, 1994). Mivel forráskódja hozzáférhető volt, minden szervezet kifejlesztette a saját (inkompatibilis) verzióját, ami káoszhoz vezetett. Két fő változat fejlődött ki: a **System V** rendszert az AT&T, a **BSD**-t **(Berkeley Software Distribution)** pedig a Kaliforniai Berkeley Egyetem készítette. Voltak kismértékben eltérő verziók is: ilyen a FreeBSD, az OpenBSD és a NetBSD. Az IEEE (*ejtsd* I triple E) **POSIX** néven kidolgozott egy szabványt, amelyet ma a legtöbb **Unix**-verzió betart. A POSIX egy minimális rendszerhíváskészletet definiál, amelyet a szabványos Unix-rendszereknek tartalmazniuk kell. Ma már néhány más operációs rendszer is tartalmazza a POSIX készletét. A POSIX szabványnak megfelelő szoftverek készítéséhez szükséges információ elérhető könyvekben (IEEE, 1990; Lewine, 1991), valamint a *www.unix.org* oldalon az Open Group „Single Unix Specification" címszó alatt. A fejezet további részében, amikor a Unixra hivatkozunk, akkor beleértjük az összes változatot, hacsak ezt külön nem jelezzük. Bár ezek belső felépítésükben különböznek, mindegyikük támogatja a POSIX szabványt, így a programozó számára elég hasonlók.

1.2.4. A negyedik generáció (1980-tól napjainkig): személyi számítógépek

Az LSI (Large Scale Integration – magas integráltságú) áramkörök fejlődésével, amelyek egy négyzetcentiméter szilikonon több ezer tranzisztort tartalmaznak, beköszöntött a **mikroprocesszor**-alapú személyi számítógépek kora. Az architektúra tekintetében a személyi számítógépek (kezdetben **mikroszámítógépek**nek hívták őket) kevéssé különböztek a PDP–11 osztály miniszámítógépeitől, de az áruk alaposan eltért. A saját miniszámítógép egy vállalati részleg vagy egy egyetemi tanszék számára tette elérhetővé a számítógépet. A mikroprocesszor megadta a lehetőséget arra, hogy bárkinek saját személyi számítógépe legyen.

A mikroszámítógépek számos családja létezett. Az Intel 1974-ben jelentette meg az első általános célú 8 bites mikroprocesszort, a 8080-ast. Több cég is gyártott 8080-ra vagy a vele kompatibilis Zilog Z80-ra épülő kész rendszereket, amelyeken a Digital Research cég **CP/M (Control Program for Microcomputers)** operációs rendszere volt széles körben használatos. A CP/M alá sok felhasználói programot készítettek, és körülbelül 5 évig uralta a személyi számítógépek világát.

A Motorola szintén megjelent egy 8 bites processzorral, a 6800-assal. Miután a cég elutasította a 6800 javítására szolgáló javaslataikat, a Motorola-mérnökök egy csoportja kivált megalapítva a MOS Technology céget, és nekikezdtek a 6502 CPU gyártásának. Ez volt a központi egysége számos korai rendszernek. Egyikük, az Apple II komoly vetélytársa lett a CP/M-rendszereknek az otthoni és az oktatási piacon. De a CP/M annyira népszerű volt, hogy sok Apple II tulajdonos vásárolt Z–80 társprocesszor bővítőkártyát a CP/M futtatásához, mivel a 6502 CPU nem volt kompatibilis a CP/M-rendszerrel. A CP/M-kártyákat egy kis cég, a Microsoft árulta, amely a CP/M-rendszert futtató mikroszámítógépekhez készített BASIC értelmező révén is rendelkezett piaci részesedéssel.

A mikroprocesszorok következő generációja 16 bites rendszer volt. Az Intel megjelentette a 8086-ot, majd az 1980-as évek elején az IBM megtervezte az IBM PC-t az Intel 8088-ra építve (ez egy 8 bites külső adatúttal rendelkező 8086-os volt). A Microsoft egy olyan csomagot ajánlott az IBM-nek, amely tartalmazta a Microsoft BASIC-et, valamint a **DOS (Disk Operating System)** operációs rendszert, amelyet ugyan egy másik cég készített, de a Microsoft felvásárolta a termékket, és szerződtette az eredeti szerzőt a további fejlesztésekhez. Az átdolgozott rendszer neve **MS-DOS (MicroSoft Disk Operating System)** lett, és gyorsan uralkodóvá vált az IBM PC-piacon.

A CP/M, az MS/DOS és az Apple DOS mind parancssoros rendszerek voltak: a felhasználók a billentyűzet segítségével gépelték be a parancsokat. Évekkel korábban Doug Engelbart a Stanford Research Institute-ban kitalálta az ablakokkal, ikonokkal, menükkel, egérrel rendelkező úgynevezett **grafikus felhasználói felület**et, a **GUI**-t **(Graphical User Interface)**. Az Apple-nél dolgozó Steve Jobs meglátta az igazán **felhasználóbarát** személyi számítógép lehetőségét (azok számára, akik nem tudtak semmit a számítógépekről, és nem akartak beletanulni), és az Apple Macintosht 1984 elején be is mutatták. Ez a Motorola 16 bites 68000 processzorát használta és 64 KB **ROM**-mal **(Read Only Memory – csak olvasható memória)** rendelkezett a grafikus felhasználói felület támogatására. A Macintosh tovább fejlődött az évek során. A következő Motorola-processzorok már igazi 32 bites rendszerek voltak, később az Apple átváltott az IBM 32 bites (majd 64 bites) RISC-architektúrájú PowerPC processzoraira. 2001-ben fontos váltás történt az operációs rendszer terén: megjelent a Berkeley Unixra épülő **Mac OS X**, a Macintosh grafikus felhasználói felületének új változatával. Végül 2005-ben az Apple bejelentette, hogy átvált Intel processzorokra.

A Microsoft kitalálta a Windowst, hogy a Macintoshsal versenyre tudjon kelni. Eredetileg a Windows csak egy grafikus környezet volt a 16 bites MS-DOS felett (vagyis inkább egy parancsértelmező, mintsem igazi operációs rendszer). A Windows aktuális verziói azonban már a Windows NT leszármazottjai, amely egy teljes 32 bites rendszer az alapoktól újraírva.

A másik nagy versenytárs a személyi számítógépek világában a Unix (és különböző leszármazottjai). A Unix a munkaállomásokon és a nagyszámítógépeken, mint amilyenek a hálózati szerverek, a legerősebb. Különösen népszerű a nagy teljesítményű RISC-alapú gépeken. Pentium-alapú gépeken a Linux kezd a Windows népszerű alternatívájává válni az egyetemi hallgatók és egyre növekvő számú vállalati felhasználó körében. (A könyvünkben végig a „Pentium" alatt a teljes Pentium-családot értjük, az alacsony teljesítményű Celeronoktól a csúcskategóriás Xeonig, valamint a kompatibilis AMD-processzorokat.)

Habár sok Unix-felhasználó, különösen a gyakorlott programozók a parancssori felületet részesítik előnyben a grafikus felülettel szemben, szinte minden Unix-rendszer támogatja az M.I.T.-n kifejlesztett, **X Window** néven ismert ablakos rendszert. Ez a rendszer kezeli az alapvető ablakműveleteket, lehetővé téve a felhasználó számára ablakok létrehozását, törlését, mozgatását és átméretezését az egér segítségével. Gyakran egy teljes grafikus felhasználói felület, amilyen például a **Motif**, is rendelkezésre áll az X Window-rendszer felett, amivel a Macintoshhoz vagy a Microsoft Windowshoz hasonló külsőt adhatnak a Unixnak azok a felhasználók, akik ilyenre vágynak.

Egy érdekes fejlődés vette kezdetét az 1980-as évek közepén: a személyi számítógép-hálózatok megnövekedtek, és megjelentek a **hálózati operációs rendszerek** és **osztott operációs rendszerek** (Tanenbaum és Van Steen, 2002). Egy hálózati operációs rendszeren a felhasználók számára több számítógép áll rendelkezésre, bejelentkezhetnek távoli gépekre, állományokat másolhatnak egyik gépről a másikra. Minden gépen saját lokális operációs rendszer fut, és mindegyiknek megvan a saját lokális felhasználója (vagy felhasználói). Alapjában véve a gépek függetlenek egymástól.

A hálózati operációs rendszerek lényegében nem különböznek az egyprocesszoros operációs rendszerektől. Nyilvánvalóan szükség van hálózati csatolókra és ezek alacsony szintű vezérlőszoftvereire, programokra a távoli bejelentkezések és az adatállományok távoli hozzáférésének kiszolgálására, de ez a többlet nem változtat az operációs rendszer lényegi struktúráján.

Nem így az osztott operációs rendszer, amelyik a felhasználói felé úgy mutatkozik, mint egy hagyományos egyprocesszoros rendszer, holott ténylegesen több processzorból áll. A felhasználóknak nem kell azzal törődniük, hogy hol futnak a

programjaik, hol tárolódnak az állományaik; ezt mind automatikusan és hatékonyan az operációs rendszer kezeli.

Nem elég az egyprocesszoros operációs rendszerhez egy kis programbővítés, hogy igazi osztott operációs rendszert kapjunk, mert alapvető módszerekben különböznek az osztott és a centralizált rendszerek. Az osztott rendszerekben például többnyire megengedett, hogy egy alkalmazás egy időben több processzoron fusson, ez pedig bonyolultabb processzorütemező algoritmust kíván ahhoz, hogy optimalizáljuk a párhuzamosítás mértékét.

A hálózaton keresztüli késleltetett kommunikáció miatt az ilyen vagy a hasonló algoritmusok hiányos, lejárt, sőt hamis információkkal kénytelenek dolgozni. Ez a helyzet nagyon különbözik az egyprocesszoros rendszerektől, ahol az operációs rendszernek teljes áttekintése van a rendszer állapotáról.

1.2.5. A MINIX 3 története

A Unix forráskódját a korai években (6. verzió) szabadon felhasználhatták az AT&T engedélye alapján, és gyakran tanulmányozták is. John Lions az ausztráliai New South Wales Egyetemen még egy kis füzetet is kiadott (Lions, 1996), amely sorról sorra kommentálja a kódot. Az AT&T hozzájárulásával ezt használta sok egyetem az operációs rendszerek tárgy tankönyveként.

Amikor az AT&T kibocsátotta a 7. verziót, kezdte észrevenni, hogy a Unix értékes üzleti áru, ezért az egyetemi kurzusokon nem engedélyezte forráskódját felhasználni, nyilván azért, hogy ne veszélyeztesse az üzleti titkot. Sok egyetem erre azzal reagált, hogy egyszerűen nem oktatták magát a Unixot, csak az elméletet.

Nem szerencsés a hallgatókkal csak az elméletet ismertetni, mert így az operációs rendszerek tényleges mibenlétéről csak hiányos áttekintésük lesz. Az operációs rendszerek elméleti kérdései, amelyeket nagy részletességgel tárgyaltak a kurzusokon és a könyvekben, nem minden esetben fontosak a gyakorlatban, például az ütemezési algoritmusok esetében sem. A gyakorlatban igazán érdekes kérdéseket, például az I/O-t és a fájlrendszereket mellőzték, mert kevés elméletük van.

A helyzet orvoslására könyvünk egyik szerzője (Tanenbaum) elhatározta, hogy teljesen az elejéről kezdve ír egy új operációs rendszert, amely felhasználói szempontból kompatibilis, felépítésében viszont teljesen különbözik a Unixtól. Egyetlen sort sem fog használni az AT&T kódjából, így nem sért szerzői jogokat, ha felhasználják csoportos vagy egyéni tanulásra. Az olvasó egy valódi operációs rendszert boncolgathat, hogy lássa, milyen belülről, ahogy a biológushallgató békát boncol. A neve **MINIX** lett, és 1987-ben jelent a meg teljes forráskóddal együtt, hogy bárki számára tanulmányozható és módosítható legyen. A MINIX név a mini-Unixból származik, mivel elég kicsi ahhoz, hogy kevésbé jártas hallgatók is átláthassák működését.

A jogi problémák kiküszöbölése mellett a Unixhoz képest a MINIX más előnyökkel is rendelkezik. Egy évtizeddel a Unix után készült, annál sokkal strukturáltabb. Például már a MINIX legelső megjelenése óta a fájlrendszer és a memóriakezelés nem az operációs rendszer része, hanem felhasználói programként

fut. A jelenlegi verzióban (MINIX 3) ez a strukturáltság ki lett terjesztve az I/O-meghajtóprogramokra is, amelyek (az óra-meghajtóprogram kivételével) szintén felhasználói módban futnak. A másik különbség, hogy a Unixot hatékonyra, míg a MINIX-et áttekinthetőre tervezték (ha egyáltalán lehet áttekinthetőségről beszélni több száz oldalas programok esetében). A MINIX-forráskód több ezer kommentárt (magyarázatot) is tartalmaz.

A MINIX eredetileg a Unix 7. verziójával kompatibilisra készült. A 7. verzió egyszerűsége és könnyedsége volt a minta. A 7. verzióról mondták sokszor, hogy nemcsak az elődein, hanem az utódain is sokat javított. A POSIX megjelenésekor a MINIX az új szabvány irányába fejlődött tovább, de megtartotta a korábbi programokkal való kompatibilitást is. Az ilyen fejlesztés általános a számítógépiparban, mert egyetlen gyártó sem szeretné, ha új gyártmányát a régi vevői csak nagy átállásokkal tudnák használni. A könyvünkben ismertetett MINIX-verzió, a MINIX 3, a POSIX szabványra épül.

A Unixhoz hasonlóan a MINIX is a C programozási nyelven készült, és a különböző számítógépek közötti hordozhatóság igen lényeges szempont volt. Az első implementáció IBM PC-re készült. Később számos más platformra is átkerült. A „kicsi a szép" filozófiáját követve, a MINIX futtatásához eredetileg még merevlemez sem kellett (az 1980-as évek közepén a merevlemez még költséges újdonság volt). Természetes, hogy a MINIX szolgáltatásainak és méretének növekedése következtében a futtatásához ma már szükséges merevlemez, de hála a MINIX-filozófiának, elegendő egy 200 megabájtos partíció (beágyazott alkalmazások esetében azonban nem szükséges a merevlemez). Ezzel szemben még a legkisebb Linux-rendszer is 500 megabájt lemezterületet igényel, és több gigabájtra van szükség az általánosan használt programok telepítéséhez.

Egy IBM PC-n futó MINIX felhasználója Unix-felhasználónak érezheti magát. Az alapvető programok megtalálhatók és ugyanazokat a funkciókat hajtják végre, mint a Unix-megfelelők (például a *cat, grep, ls, make* és a parancsértelmező). Ezeket a segédprogramokat, ugyanúgy, ahogy magát az operációs rendszert, a szerző és hallgatói, továbbá még néhányan az alapoktól teljes egészében újraírták, AT&T vagy más szabadalmaztatott programkód felhasználása nélkül. Sok egyéb szabadon terjeszthető program létezik manapság, és a legtöbb esetben ezek sikeresen átültethetők (lefordíthatók) MINIX alá.

A következő évtizedben a MINIX továbbfejlődött, és 1997-ben megjelent a MINIX 2, könyvünk második angol nyelvű kiadásával egyidejűleg, amely ezt az új változatot mutatta be. A két verzió közötti változtatások ha forradalmiak nem is, de alapvetők voltak (például a hajlékonylemezt és a 8088-as processzor 16 bites valós módját használó rendszerből merevlemezt és a 386-os processzor 32 bites védett módját használó rendszerré alakult).

A fejlesztés lassan, de szisztematikusan haladt 2004-ig, amikor Tanenbaum biztossá vált abban, hogy a szoftver túlságosan naggyá és megbízhatatlanná vált, és úgy döntött, hogy újra felveszi a kissé alvó MINIX-szálat. Az amszterdami Vrije Egyetemen hallgatóival és programozóival közösen elkészítette a MINIX 3-at a rendszer alapvető áttervezésével, nagymértékben átstrukturált kernellel, kisebb mérettel, valamint a modularitás és megbízhatóság hangsúlyozásával. Az új ver-

ziót mind PC-kre, mind beágyazott rendszerekre szánták, ahol a kompaktság, a modularitás és a megbízhatóság kulcsfontosságú. Míg a csoport néhány tagja teljesen új nevet szeretett volna, végül mégis a MINIX 3 lett a befutó, mivel a MINIX név már jól ismert volt. Hasonlóan, mint amikor az Apple felhagyott a saját operációs rendszerével, a Mac OS 9-cel, és lecserélte a Berkeley Unix egyik változatára; az új rendszer neve Mac OS X lett, és nem APPLIX vagy valami hasonló. A Windows-rendszerekben bekövetkezett alapvető változások után is megmaradt a Windows elnevezés.

A MINIX 3 magja csak egy körülbelül 4000 soros futtatható kódból áll, nem milliókból, mint a Windows, a Linux, a FreeBSD és más operációs rendszerek esetében. A kisméretű mag azért fontos, mert az ott előforduló hibák sokkal pusztítóbbak, mint a felhasználói módban futó programok esetében, és több kód több hibát jelent. Egy alapos vizsgálat megmutatta, hogy a *felderített* hibák száma 1000 végrehajtható soronként 6 és 16 között váltakozik (Basili és Perricone, 1984). A hibák tényleges száma valószínűleg sokkal magasabb lehet, mivel a kutatók csak a bejelentett hibákat tudták számolni, a bejelentetleneket nem. Egy másik vizsgálat (Ostrand et al., 2004) azt mutatja, hogy még egy tucatnál is több kiadott változat után is a fájlok 6%-a tartalmazott olyan hibát, amit később jelentettek, és egy bizonyos pont után a hibák száma stabilizálódik ahelyett, hogy aszimptotikusan közelítene a nullához. Ezt az eredményt alátámasztja az a tény is, hogy amikor nagyon egyszerű, automatikus modellellenőrzőt eresztettek a Linux és az OpenBSD stabil változataira (Chou et al., 2001; és Engler et al., 2001), az százszámra talált hibákat a magban, túlnyomórészt a meghajtóprogramokban. Ez az oka annak, hogy a meghajtóprogramok kikerültek a MINIX 3 magjából – felhasználói módban kisebb kárt képesek okozni.

Könyvünk a MINIX 3-at példaként ismerteti. A MINIX 3-rendszerhívásokra vonatkozó legtöbb hivatkozásunk azonban (az aktuális forráskódra vonatkozókkal ellentétben) érvényes más Unix-rendszerekre is. Ennek tudatában olvassuk ezt a könyvet.

Néhány olvasót érdekelhet a Linux és a MINIX viszonya, erről is ejtünk pár szót. Megjelenését követően egy USENET témacsoport (newsgroup), a *comp.os.minix* alakult a MINIX értékelésére. Néhány héten belül 40 000 előfizetője lett, majdnem mindegyikük rengeteg új szolgáltatást akart a MINIX-be építeni, ezzel nagyobbá, jobbá (de nagyobbá biztosan) akarták tenni. Naponta százak ajánlgatták javaslataikat, ötleteiket és programdarabkáikat. A MINIX szerzője éveken keresztül sikeresen hárította el ezeket a támadásokat, így maradt a MINIX elég tiszta ahhoz, hogy a hallgatók megérthessék, valamint elég kisméretű ahhoz, hogy hallgatók által is megfizethető gépeken fusson. Azon felhasználók számára, akik nem tartották sokra az MS-DOS-t, a MINIX jelenléte alternatívaként (forráskódjával együtt) okot adott arra, hogy végül vegyenek egy PC-t.

Egyikük egy finn egyetemi hallgató, Linus Torvalds volt. Torvalds telepítette a MINIX-et az új PC-jére, és alaposan tanulmányozta a forráskódot. Szerette volna a USENET-es hírcsoportokat (amilyen a *comp.os.minix* is) az otthoni gépén olvasni az egyetemi helyett, de ehhez néhány funkció hiányzott a MINIX-ből, ezért ennek megoldására írt egy programot. Hamar rájött, hogy egy másfajta terminál-

meghajtóra van szüksége, amelyet szintén elkészített. Ezután le akarta tölteni és elmenteni a hozzászólásokat, ehhez írt egy lemezmeghajtót, majd egy fájlrendszert. 1991 augusztusára elkészült egy nagyon egyszerű maggal, amelyet 1991. augusztus 25-én jelentett be a *comp.os.minix* hírcsoportban. Ez a bejelentés vonzotta az embereket, hogy segítsenek neki, aminek eredményeként 1994. március 13-án megjelent a Linux 1.0. Így született meg a Linux.

A Linux a **nyílt forráskód** mozgalom (amelyet a MINIX segített elindítani) egyik jelentős sikere lett. A Linux a Unix (és a Windows) vetélytársa több környezetben is, részben mivel a Linuxot futtatni képes PC-k teljesítménye a néhány Unix-implementáció által megkövetelt RISC-rendszerekkel vetekszik. Más nyílt forráskódú szoftverek, mindenekelőtt az Apache webkiszolgáló és a MySQL adatbázis-kezelő jól működnek Linuxon az üzleti világban. A Linux, az Apache, a MySQL és a nyílt forráskódú Perl és PHP programozási nyelvek gyakran használatosak webkiszolgálókon, időnként a **LAMP** betűszóval hivatkoznak rájuk. A Linux és a nyílt forráskódú szoftverek történetéről bővebben olvashatunk (DiNone et al., 1999; Moody, 2001; és Naughton, 2000).

1.3. Az operációs rendszer fogalmai

Az operációs rendszer és a felhasználói programok közötti kapcsolatot az a „kiterjesztett utasítás" készlet alkotja, amelyet az operációs rendszer biztosít. Hagyományosan ezeket a kiterjesztett utasításokat **rendszerhívások**nak nevezzük, bár többféle módon valósíthatók meg. Az operációs rendszer valódi működésének megértéséhez közelebbről kell megismerni ezt a kapcsolatrendszert. A kapcsolatrendszer által biztosított rendszerhívások operációs rendszerenként eltérők, de a mögöttük rejlő fogalmak egyre hasonlóbbá válnak.

Ez az oka annak, hogy választanunk kell a kissé homályos általánosságok („az operációs rendszerekben van rendszerhívás a fájlok olvasására") és a konkrét rendszerek („a MINIX 3-ban van egy read rendszerhívás három paraméterrel: egy a fájl azonosítására, egy megmondja hová helyezzük az adatokat, egy pedig a beolvasandó bájtok számát közli") között.

Mi az utóbbi megközelítést választottuk. Több munkával jár, de az operációs rendszer működésébe mélyebb betekintést nyújt. Az 1.4. alfejezetben részletesen vizsgáljuk a Unix (beleértve a többféle BSD-változatot is), a Linux és a MINIX 3 által biztosított rendszerhívásokat. Az egyszerűség kedvéért azonban általában csak a MINIX 3-ra fogunk hivatkozni, mivel a megfelelő Unix- és Linux-rendszerhívások többnyire POSIX-alapúak. Mielőtt belemerülnénk a konkrét rendszerhívások ismertetésébe, érdemes madártávlatból rápillantanunk a MINIX 3-ra, és egy általános benyomást szereznünk az operációs rendszerek mibenlétéről. Ez az áttekintés jól illik a Unixra és a Linuxra is.

A MINIX 3-rendszerhívások durván két nagy osztályba tartoznak: a processzusokat kezelő és az adatállományok rendszerét (fájlokat) kezelő osztályokba. Ennek megfelelően a kettőt külön-külön tanulmányozzuk.

1.3.1. Processzusok

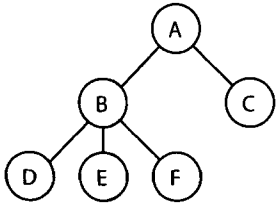
Kulcsfontosságú a MINIX 3-ban és minden operációs rendszerben a **processzus** fogalma. A processzus lényegében egy végrehajtás alatt lévő program. Minden processzushoz tartozik egy saját **címtartomány**, azaz a memória egy minimális (általában 0) és egy maximális című helye közötti szelet, amelyen belül a processzus olvashat és írhat. A címtartomány tartalmazza a végrehajtandó programot, annak adatait és a vermét. Minden processzushoz tartozik még egy regiszterkészlet, beleértve az utasításszámlálót, veremmutatót, egyéb hardverregisztereket és a program futásához szükséges egyéb információkat.

A 2. fejezetben részletesen vissza fogunk térni a processzus fogalmának tisztázására, most megfelelő, ha egy multiprogramozásos rendszerbeli processzusról kapunk képet. Időnként az operációs rendszer megszakítja egy processzus futását, és egy másikat kezd futtatni, mert például az előbbi processzusnak lejárt a megelőző másodpercre járó CPU-idő részesedése.

Ha az olyan esetekben, mint a fentiben is, ideiglenesen felfüggesztünk egy processzust, akkor később pontosan ugyanabban az állapotban kell folytatni, mint amelyben megszakítottuk a futást. Ezért a processzushoz tartozó minden információt a felfüggesztés időtartamára valahol tárolnunk kell. A processzus például olvasásra megnyithatott néhány fájlt. Minden ilyen fájlhoz hozzátartozik az aktuális pozícióra mutató pointer (azaz a legközelebb olvasandó bájt vagy rekord sorszáma). Ha egy ilyen processzust felfüggesztünk, az összes mutatót tárolnunk kell ahhoz, hogy az újraindítása utáni read hívások a megfelelő adatokat olvassák. A legtöbb operációs rendszerben a processzushoz tartozó minden, a saját címtartományának tartalmán kívüli információt az operációs rendszer **processzustáblázat**ában tárolnak. Ez az aktuálisan létező processzusokhoz tartozó struktúraelemekből álló vektor vagy láncolt lista.

Ennek megfelelően egy (felfüggesztett) processzus a címtartományának tartalmából, amelyet **memóriatérkép**nek szokás nevezni és a processzustáblázatbeli hozzá tartozó elemből áll; ez utóbbi tartalmazza egyebek között a regiszterértékeket.

Kulcsfontosságú processzuskezelő rendszerhívások a processzust létrehozó és megszüntető hívások. Nézzünk egy tipikus példát. A **parancsértelmező** vagy **shell** processzus parancsokat olvas egy terminálról. Egy program fordítását kérő parancs begépelését befejezte a felhasználó. A parancsértelmezőnek most létre kell hoznia



1.5. ábra. *Egy processzus-fastruktúra. Az* A *processzus két gyermekprocesszust, a* B*-t és a* C*-t hozta létre. A* B *processzus további három processzust, a* D*-t, az* E*-t és az* F*-et hozta létre*

egy új processzust, amely a fordítót hajtja végre. Amikor ez a processzus befejezte a fordítást, végrehajt egy rendszerhívást, amellyel megszünteti saját magát.

Windows és más grafikus felhasználói felületet használó operációs rendszer esetében a munkaasztalon található ikonokra történő (dupla) kattintással indíthatunk programokat, hasonló módon, mintha a program nevét egy parancssorba gépelnénk be. Bár a grafikus felhasználói felületeket nem igazán tárgyaljuk, lényegében azok is egyszerű parancsértelmezők.

Mivel egy processzus létrehozhat egy vagy több processzust (**gyermekprocesszusait**), és ezek újra saját gyermekprocesszusaikat, láthatjuk, hogy a processzusok az 1.5. ábra szerinti fastruktúrát alkotják. Az egymással olyan kapcsolatban lévő processzusok esetében, amikor egy közös feladat végrehajtásában együttműködnek, szükség van az egymással való kommunikációra és tevékenységük összehangolására. Ezt az ún. **processzusok közötti kommunikációt** a 2. fejezet tárgyalja részletesen.

További processzus-rendszerhívások alkalmasak memória kérésére (vagy már nem használt memória felszabadítására), gyermekprocesszus megszűnésére való várakozásra és programrészletek megosztott használatára.

Esetenként olyan processzus számára is szükség van információátadásra, amely éppen nem erre várakozik. Például egy processzus, amely egy másik gépen futó processzussal kommunikál, hálózaton keresztül küldi el az üzenetét. Annak kivédésére, hogy az üzenet vagy a válasz esetleg elveszhet, a küldő kérheti saját operációs rendszerét, hogy értesítse valahány másodperc múlva, amikor is a válasz hiányában az üzenetet újra elküldheti. Miután ezt az időzítést beállította, a processzus folytathatja saját munkáját.

Amikor a beállított másodpercek elteltek, az operációs rendszer egy **riasztás szignál**t (jelet vagy jelzést) küld a processzusnak. A szignál ideiglenesen felfüggeszti a processzus pillanatnyi tevékenységét, a regiszterértékeket eltárolja a verembe, és elindít egy speciális szignálkezelő eljárást, például a feltételezhetően elveszett üzenet újraküldésére. Amikor a szignálkezelő eljárás befejeződött, a processzus újraindítódik ugyanabban az állapotban, amilyenben a szignál érkezése előtt volt. A szignálok a hardvermegszakítások szoftvermegfelelői. Sokféle ok miatt generálódhatnak, nem csak időzítők lejártakor. A hardver által észlelt csapdák többsége, mint például az illegális utasítás-végrehajtás vagy érvénytelen memóriacím-használat, a „bűnös" processzus számára szignálként jelentkezik.

A rendszeradminisztrátor minden személynek, aki a MINIX 3 felhasználója lehet, ad egy **UID (User IDentification)** felhasználói azonosítót. A MINIX-ben indított minden processzus az indító személy UID-azonosítóját kapja. A gyermekprocesszus a szülő UID-azonosítóját kapja. A felhasználók különböző csoportok tagjai lehetnek, amelyek csoportazonosítóval **(GID – Group IDentification)** rendelkeznek.

Az ún. **szuperfelhasználó** UID-azonosítója speciális lehetőségeket kap, és sok védelmi szabályt áthághat. Nagy rendszerek esetében csak a rendszeradminisztrátor ismeri a szuperfelhasználó jelszavát, bár sok „rendes" felhasználó (rendszerint hallgató) jelentős erőfeszítést tesz olyan rendszerbeli hézagok felkutatására, amelyek jelszó nélkül is szuperfelhasználói jogokhoz juttatják.

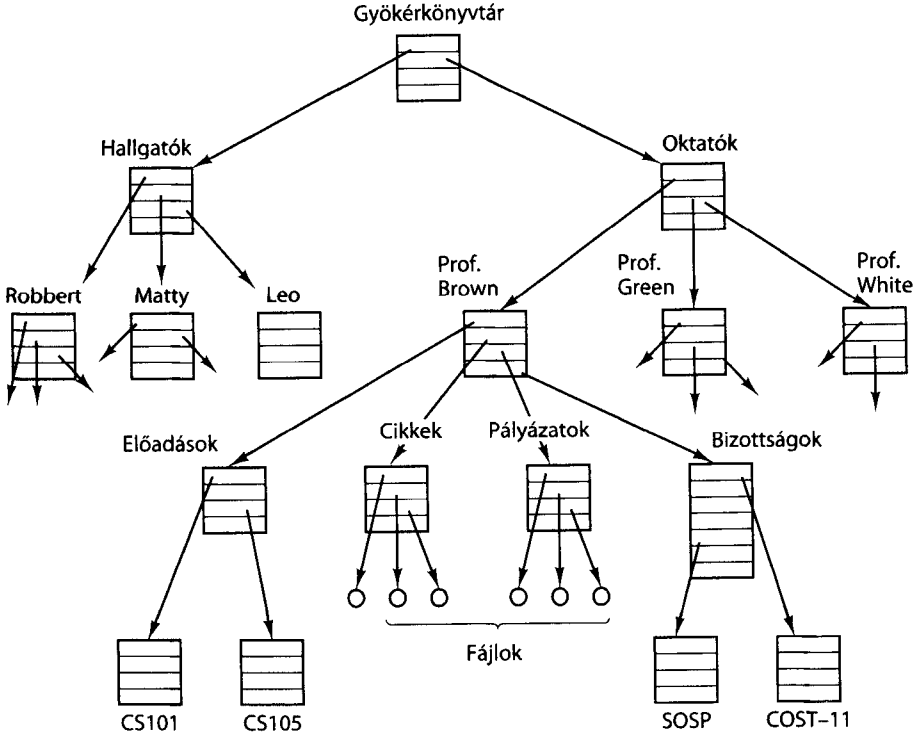
A processzusokat, a processzusok közötti kommunikációt és egyéb kapcsolódó témákat a 2. fejezetben tárgyaljuk.

1.3.2. Fájlok

A másik kiterjedt rendszerhívás osztály a fájlrendszerrel kapcsolatos. Már említettük, hogy az operációs rendszer egyik fő feladata a lemez és egyéb I/O-egységek különlegességeinek az elrejtése és a programozó számára egy világos, eszközfüggetlen fájlrendszer-absztrakció biztosítása. Nyilvánvalóan rendszerhívások szükségesek fájlok létrehozására, törlésére, olvasásukra és írásukra. Mielőtt egy fájlt olvashatunk, meg kell nyitni, olvasás után pedig le kell zárni, így ezekhez is rendszerhívások szükségesek.

A MINIX 3 a **könyvtár (directory)** fogalmát biztosítja a fájlok helyének nyilvántartására és a fájlok csoportosítására. Egy hallgatónak például lehet egy könyvtára a felvett kurzusairól (ebben tartja a kurzusok anyagait), egy másik az elektronikus levelezéséhez, egy harmadik a World Wide Web honlapjához. Rendszerhívások kellenek a könyvtárak létrehozására, törlésére. Ugyancsak hívások segítenek egy létező fájl valamelyik könyvtárba helyezésére vagy abból való törlésére. Könyvtárelemek fájlok vagy újabb könyvtárak lehetnek. Ez a modell szintén egy hierarchiához, a fájlrendszerhez vezet, amint az 1.6. ábra mutatja.

A processzusok és a fájlok hierarchiája egyaránt fastruktúrába szerveződik, de a hasonlóság itt be is fejeződik. A processzushierarchia általában nem túl mély (háromnál több szint szokatlan), míg a fájlhierarchia négy-öt, sőt többszintű. A processzushierarchia rövid életű, általában néhány perces, a könyvtár-hierarchia viszont évekig létezhet. A tulajdonosi és a védelmi rendszer úgyszintén különbözik a processzusok és a fájlok esetében. Egy gyermekprocesszus vezérlésére vagy akár elérésére általában csak a szülőprocesszus jogosult, míg a fájlok és könyvtárak olvasására majdnem mindig a tulajdonosnál bővebb csoport számára is van jogosultság.

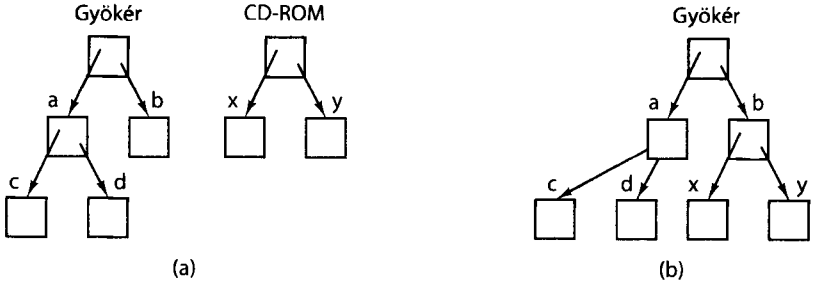


1.6. ábra. *Egy egyetemi tanszék fájlrendszere*

Bármely könyvtár-hierarchiabeli fájl azonosítható az **útvonal nevével**, kezdve a könyvtár-hierarchia tetejétől, az ún. **gyökérkönyvtár**tól. Az ilyen teljes útvonalnév azoknak a „/" jelekkel elválasztott könyvtárneveknek a listájából áll, amelyeket a gyökérkönyvtártól a fájlhoz vezető úton találunk. Az 1.6. ábrán a *CS101* fájl útvonalneve */Oktatók/Prof.Brown/Előadások/CS101*. A kezdő „/" azt jelenti, hogy teljes útvonalnévről van szó, azaz a gyökérkönyvtárból indulunk. A Windows esetében a „\" jel használatos az elválasztásra a „/" helyett, így a fenti elérési útvonal a következő lesz: *\Oktatók\Prof.Brown\Előadások\CS101*. A könyvben a Unix útvonal-megadási jelölésmódját fogjuk használni.

Minden processzus egy adott időpontban rendelkezik egy **munkakönyvtár**ral, ahol a nem „/" jellel kezdődő útvonalnevű fájlok keresése történik. Az 1.6. ábra példáján, ha a */Oktatók/Prof.Brown* a munkakönyvtár, akkor az *Előadások/CS101* útvonalnév ugyanazt a fájlt azonosítja, mint az előbbi teljes útvonalnév. A processzusok váltogathatják munkakönyvtáraikat, ha új munkakönyvtárat azonosító rendszerhívásokat hajtanak végre.

A MINIX 3 a fájlok és könyvtárak védelmére hozzájuk rendel egy 11 bites bináris védelmi kódot. A védelmi kód három 3 bites mezőből áll, rendre a tulajdonos, a tulajdonos csoportjának tagjai (a felhasználókat a rendszer-adminisztrátor csoportokba sorolja) és a többiek számára, a maradék 2 bitet később tárgyaljuk. A mezők bitjei az olvasási, írási, valamint végrehajtási jogokat jelzik. Egy ilyen 3 bites mezőt **rwx bitek**nek szokásos nevezni. Például az *rwxr-x--x* védelmi kód azt jelenti, hogy a tulajdonosnak olvasási, írási és végrehajtási, csoportja többi tagjának olvasási és végrehajtási (de írási nem), míg mindenki másnak csak végrehajtási joga van a fájlra. Könyvtárakra az *x* a keresési engedélyt jelzi. A „–" jel a megfelelő engedély hiányát jelzi (a megfelelő bit 0 értékű).



1.7. ábra. *(a) A CD-ROM-meghajtó fájljai a felcsatolás előtt nem elérhetők. (b) Felcsatolás után a fájlhierarchia részévé válnak*

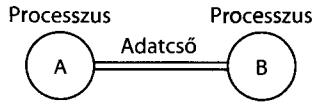
Egy fájlt olvasása vagy írása előtt meg kell nyitni, ekkor történik a jogosultságok ellenőrzése. Ha a hozzáférés engedélyezett, a rendszer visszaad egy kis egész értéket, az ún. **fájlleírót (deszkriptor)**, minden további műveletben ezt kell használni. Ha a hozzáférés tiltott, egy hibakódot (–1) kapunk vissza.

A MINIX 3 egy további fontos fogalma a fájlrendszerek felcsatolása. Majdnem minden személyi számítógépben van egy vagy több CD-ROM-meghajtó, ezekben cserélgethetjük a CD lemezeket. Az ilyen eltávolítható adathordozók (CD és DVD lemezek, hajlékonylemezek, Zip-meghajtók stb.) egyszerű kezelésére a MINIX 3 megengedi, hogy azok fájlrendszerét hozzácsatoljuk a fő könyvtár-hierarchiához. Nézzük az 1.7.(a) ábra esetét. A mount rendszerhívás előtt a merevlemezen található **gyökérfájlrendszer** és a CD-ROM-on található másik fájlrendszer egymástól elválasztott és független.

A CD-ROM fájlrendszere azonban nem használható, hiszen nem tudunk rá útvonalnevet mondani. A MINIX 3 nem engedi, hogy az útvonalnevek elé meghajtóneveket vagy számokat írjunk; ezzel ugyanis éppen eszközfüggőséget vezetne be, holott az operációs rendszernek illik az ilyet kiküszöbölni. Ehelyett a mount rendszerhívás a CD-ROM-meghajtó fájlrendszerét hozzácsatolja a gyökérfájlrendszerhez, megengedve a program kívánsága szerinti bármelyik helyre. Az 1.7.(b) ábrán a CD-ROM-meghajtó fájlrendszerét a *b* könyvtárra csatoltuk fel, ezzel a */b/x* és */b/y* fájlok elérhetők. Ha a *b* könyvtár tartalmaz fájlt, az mindaddig nem érhető el, amíg a CD-ROM-meghajtó oda van felcsatolva, hiszen a */b* hivatkozás a CD-ROM-meghajtó gyökérkönyvtárát jelenti. (Az ilyen fájlok elérhetetlensége nem tragédia, a fájlrendszereket üres könyvtárakra szokták felcsatolni.) Ha a rendszerben több merevlemez is elérhető, ezek szintén felcsatolhatók egyetlen fastruktúrába.

A MINIX 3 egy másik fontos fogalma a **specifikus fájl**. A specifikus fájlok segítségével lehet az I/O-eszközöket fájlokként kezelni. Ezzel a módszerrel ugyanazokkal a rendszerhívásokkal tudunk olvasni róluk és írni rájuk, amelyekkel a fájlokat olvassuk és írjuk. Kétféle specifikus fájl létezik: a **blokkspecifikus** és a **karakterspecifikus fájlok**. Blokkspecifikus fájlokat használunk normál esetben az olyan eszközök modellezésére, amelyek tetszőlegesen címezhető blokkok gyűjteményéből állnak, ilyenek például a lemezek. Ha megnyitunk egy blokkspecifikus fájlt és olvassuk például a 4. blokkot, akkor a program az eszköz 4. blokkját közvetlenül eléri, függetlenül attól, hogy az eszközön a fájlrendszer milyen struktúrájú. Hasonlóan karakterspecifikus fájlokat használunk nyomtatók, modemek és olyan eszközök modellezésére, amelyek karaktersorozatokat fogadnak vagy küldenek. A specifikus fájlok egyezményes helye a */dev* könyvtár. Például a */dev/lp* egy soros nyomtató lehet.

Még egy fogalomról ejtünk szót ebben az áttekintésben, amely mind a processzusokkal, mind a fájlokkal kapcsolatos, az adatcsőről. Az **adatcső** két processzus



1.8. ábra. *Két processzust egy adatcső köt össze*

összekapcsolására alkalmas egyfajta fájl, amelyet az 1.8. ábra is mutat. Ha az *A* és *B* processzusok adatcsövön keresztül szeretnének kommunikálni, akkor azt előtte fel kell építeniük. Amikor az *A* processzus adatot kíván küldeni a *B* processzusnak, akkor az adatcsőbe ír, mintha az kimeneti fájl lenne. A *B* processzus az adatokat az adatcsőből olvashatja, mintha az bemeneti fájl lenne. Ezzel a MINIX 3 processzusainak egymással való kommunikációja a közönséges fájlolvasás és -írás módszeréhez hasonlónak látszik. Sőt egy processzus csak speciális rendszerhívással képes megállapítani, hogy kimeneti fájlja nem valódi fájl, hanem adatcső.

1.3.3. A parancsértelmező

A rendszerhívásokat az operációs rendszer kódja hajtja végre. A szövegszerkesztők, fordítók, assemblerek, linkerek és a parancsértelmezők nem az operációs rendszer részei, de igen fontosak és hasznosak. Vállalva a dolgok egy kis keverését, ebben a szakaszban röviden ismertetjük a MINIX 3 parancsértelmezőjét, a **shell**t. Ez ugyan nem része az operációs rendszernek, de mivel intenzíven használja az operációs rendszer funkcióit, igen kitűnő példa arra, hogyan kell a rendszerhívásokkal bánni. A terminálja előtt ülő felhasználó és az operációs rendszer között is ez az elsődleges kapcsolati felület, hacsak nem grafikus felhasználói felületet használunk. Sokféle shell létezik, ilyen például a *csh,* a *ksh,* a *zsh* és a *bash* is. Mindegyikük támogatja a következőkben ismertetett funkciókat, amelyek az eredeti shellből származnak *(sh)*.

Minden felhasználó bejelentkezésekor egy parancsértelmező indul el. A parancsértelmező standard bemenete és kimenete a terminál. A parancsértelmező a **prompt**, például a dollárjel megjelenítésével indul, ezzel jelzi, hogy kész a felhasználó parancsának fogadására. Ha most a felhasználó például a

```
date
```

sort gépeli, akkor a parancsértelmező létrehoz egy gyermekprocesszust, amely a *date* programot futtatja. A gyermekprocesszus futása közben a parancsértelmező annak megszűntére vár. A gyermek megszűnésekor a parancsértelmező újra megjeleníti a promptot, és a következő bemeneti sor beolvasását kísérli meg.

A felhasználó a standard kimenetet átirányíthatja fájlba, ha begépeli például a

```
date >file
```

sort. Hasonlóan a standard bemenet is átirányítható, mint az alábbi parancsban

```
sort <file1 >file2
```

amely a *sort* programot indítja; ez bemenetét a *file1* fájlból veszi, eredményeit pedig a *file2* fájlba küldi.

Egy program kimenetét egy másik program bemeneteként használhatja, ha egy adatcsővel kötjük össze őket. A

```
cat file1 file2 file3 | sort >/dev/lp
```

parancsban a *cat* három fájlt konkatenál, eredményét átküldi a *sort* programnak, amely a sorokat alfabetikus sorrendbe rendezi. A *sort* kimenetét átirányítottuk a */dev/lp* fájlba, amely szokásosan a nyomtatót jelenti.

Ha a parancs végére egy „&" jelet gépel a felhasználó, akkor a parancsértelmező nem vár a parancs végrehajtásának befejeztére, hanem azonnal újra promptot ad. Így a

```
cat file1 file2 file3 | sort >/dev/lp &
```

parancs háttérfeladatként indítja el a *sort* programot, eközben a felhasználó folytathatja megszokott munkáját. Itt nincs hely arra, hogy a parancsértelmező számos egyéb érdekes tulajdonságát tárgyaljuk. A kezdők számára készült Unix-könyvek nagy része megfelelő azoknak, akik a MINIX 3 használatát szeretnék jobban megtanulni; ilyenek például (Ray és Ray, 2003; Herborth, 2005).

1.4. Rendszerhívások

Van már áttekintésünk arról, hogyan kezeli a MINIX 3 a processzusokat és a fájlokat, most nézzük az operációs rendszer és a felhasználói programok közötti kapcsolatot, azaz a rendszerhívások készletét. Tárgyalásmódunk POSIX- (International Standards 9945-1) alapú, így a MINIX 3-ra, a Unixra és a Linuxra is érvényes, de a legtöbb mai operációs rendszerben ugyanezeket a feladatokat végrehajtó rendszerhívások vannak, még ha részleteikben különböznek is. A rendszerhívások végrehajtásának esetenkénti módszere erősen gépfüggő, leginkább assembly nyelven fogalmazható meg, ezért egy eljáráskönyvtár áll rendelkezésünkre ahhoz, hogy C nyelvű programokból rendszerhívásokat hajthassunk végre.

Érdemes megjegyezni a következőt: bármelyik egyprocesszoros számítógép egy időben csak egy utasítást képes végrehajtani. Amennyiben a processzus egy felhasználói módban futó felhasználói programot futtat és rendszerhívásra van szüksége, egy csapdát vagy rendszerhívó utasítást kell végrehajtania, hogy a vezérlést átadja az operációs rendszernek. A paraméterek vizsgálatával az operációs rendszer eldönti, hogy a hívó processzus mit szeretne. Ezután végrehajtja a rendszerhívást, és visszaadja a vezérlést arra az utasításra, amely a rendszerhívást követi. Bizonyos értelemben egy rendszerhívás olyan, mint egy speciális eljáráshívás, csak a rendszerhívások a magba vagy más privilegizált operációsrendszer-komponensbe lépnek be, míg a hagyományos eljáráshívások nem.

Processzus-kezelés	pid = **fork**()	A szülővel azonos gyermekprocesszus létrehozása
	pid = **waitpid**(pid, &status, opts)	Gyermek megszűnésére várakozás
	s = **wait**(&status)	A waitpid elavult változata
	s = **execve**(name, argv, envp)	A processzus memóriatérképének felülírása
	exit(status)	A processzus végrehajtásának befejezése és az exit státus beállítása
	size = **brk**(addr)	Az adatszegmens méretének beállítása
	pid = **getpid**()	A hívó processzus pid azonosítójának visszaadása
	pid = **getpgrp**()	A hívó processzus csoportazonosítójának visszaadása
	pid = **setsid**()	Új szekció létrehozása és processzuscsoport gid visszaadása
	l = **ptrace**(req, pid, addr, data)	Tesztelésre használható
Szignálok	s = **sigaction**(sig, &act, &oldact)	Szignálokon végrehajtandó akciót definiál
	s = **sigreturn**(&context)	A szignál eljárásból való kilépés
	s = **sigprocmask**(how, &set, &old)	A szignál maszk vizsgálata vagy módosítása
	s = **sigpendIng**(set)	A blokkolt szignálhalmaz megkérése
	s = **sigsuspend**(sigmask)	A szignál maszk felülírása és a processzus felfüggesztése
	s = **kill**(pid, sig)	Szignál küldése egy processzusnak
	residual = **alarm**(seconds)	Az ébresztőóra beállítása
	s = **pause**()	A hívó felfüggesztése a következő szignál érkezéséig
Fájlkezelés	fd = **creat**(name, mode)	Új fájl létrehozásának elavult változata
	fd = **mknod**(name, mode, addr)	Reguláris, specifikus vagy könyvtár i-csomópont létrehozása
	fd = **open**(file, how, ...)	Fájl megnyitása olvasásra, írásra vagy mindkettőre
	s = **close**(fd)	Nyitott fájl lezárása
	n = **read**(fd, buffer, nbytes)	Adat olvasása fájltárolóba
	n = **write**(fd, buffer, nbytes)	Adat írása fájltárolóból fájlba
	pos = **lseek**(fd, offset, whence)	A fájlmutató mozgatása
	s = **stat**(name, &buf)	Fájl állapotinformációinak megkérése
	s = **fstat**(fd, &buf)	Fájl állapotinformációinak megkérése
	fd = **dup**(fd)	Nyitott fájl leírójának átmásolása
	s = **pipe**(&fd[0])	Adatcső létrehozása
	s = **ioctl**(fd, request, argp)	Fájlokon speciális műveletek végrehajtása
	s = **access**(name, amode)	Fájl elérhetőségének vizsgálata
	s = **rename**(old, new)	Fájl átnevezése
	s = **fcntl**(fd, cmd, ...)	Fájl zárolása és egyéb műveletek
Könyvtár- és fájlrendszer-kezelés	s = **mkdir**(name, mode)	Új könyvtár létrehozása
	s = **rmdir**(name)	Üres könyvtár megszüntetése
	s = **link**(name1, name2)	Egy új, a name1-re mutató name2 bejegyzés létrehozása
	s = **unlink**(name)	Egy könyvtárbejegyzés megszüntetése
	s = **mount**(special, name, flag)	Fájlrendszer felcsatolása
	s = **umount**(special)	Fájlrendszer lecsatolása
	s = **sync**()	A raktározott adatblokkok írása lemezre
	s = **chdlr**(dirname)	A munkakönyvtár változtatása
	s = **chroot**(dirname)	A gyökérkönyvtár változtatása
Védelem	s = **chmod**(name, mode)	A fájl védelmi bitjeinek változtatása
	uid = **getuid**()	A hívó uid azonosítójának megkérése
	gid = **getgid**()	A hívó gid csoportazonosítójának megkérése
	s = **setuid**(uid)	A hívó uid azonosítójának beállítása
	s = **setgid**(gid)	A hívó gid csoportazonosítójának beállítása
	s = **chown**(name, owner, group)	A fájl tulajdonosának és csoportjának változtatása
	oldmask = **umask**(complmode)	A módmaszk változtatása
Időkezelés	seconds = **time**(&seconds)	Az 1970. jan.1-jétől eltelt idő megkérése
	s = **stime**(tp)	Az 1970. jan.1-jétől eltelt idő beállítása
	s = **utime**(file, timep)	A fájlok utolsó hozzáférési idejének beállítása
	s = **times**(buffer)	Az elhasznált felhasználói és rendszeridő megkérése

1.9. ábra. *A MINIX fő rendszerhívásai. Az* fd *egy fájlleíró; az* n *egy bájtszámláló*

A rendszerhívások mechanizmusának megvilágítására nézzük a read esetét, melynek három paramétere van: a fájl, a tároló és a beolvasandó bájtok számának specifikációi. Egy C programból a read hívása így nézhet ki:

```
count = read(fd, buffer, nbytes);
```

A rendszerhívás (és a könyvtárbeli eljárás) a *count* változóban visszaadja a ténylegesen beolvasott bájtok számát. Ez az érték általában megegyezik az *nbytes* értékkel, de lehet kisebb, ha például az olvasás közben fájl végét észlelünk.

Ha a rendszerhívás nem hajtható végre paraméterhiba vagy lemezhiba miatt, a *count* értéke –1 lesz, továbbá az *errno* globális változóba egy hibakód kerül. Programjainkban mindig ellenőrizni kell a rendszerhívások által visszaadott értéket, hogy az esetleges hibát észrevegyük.

A MINIX 3-ban összesen 53 fő rendszerhívás van. Az 1.9. ábra (lásd előző oldal) sorolja fel ezeket áttekinthető csoportosításban, hat kategóriában. Van még néhány más hívás is, de mivel azok csak nagyon speciális célra használhatók, ezért kihagyjuk őket. A következő szakaszokban röviden megvizsgáljuk az 1.9. ábrán felsorolt hívások működését. Nagyrészt ezen rendszerhívások által biztosított tevékenységek határozzák meg az operációs rendszerek nyújtotta szolgáltatásokat. (Személyi számítógépeken az erőforrás-kezelés elhanyagolhatóan kis feladat a sokfelhasználós nagy rendszerekhez képest.)

Ez a megfelelő hely arra, hogy rámutassunk, a POSIX-eljáráshívások és a rendszerhívások egymáshoz való megfeleltetése nem feltétlenül egyértelmű. A POSIX szabvány felsorolja a rendszerek által biztosítandó eljárásokat, de nem írja elő, hogy ezek rendszerhívások, könyvtári eljárások vagy egyebek legyenek. Egyes esetekben a POSIX-eljárásokat a MINIX 3 könyvtári eljárásokkal valósítja meg. Más esetekben, ha a szabvány előírta eljárások egymástól csak kissé különböznek, akkor azokat egyetlen rendszerhívással valósítottuk meg.

1.4.1. Processzuskezelő rendszerhívások

Az első rendszerhíváscsoport a processzusok kezelésére szolgál. Az ismertetést a fork hívással kezdjük. A fork a processzusok létrehozásának egyetlen módja MINIX 3-ban. Az eredeti processzus pontos másolatát hozza létre, beleértve a fájlleírókat, regiszterértékeket, mindent. A fork után az eredeti és a másolat (a szülő- és a gyermek-) processzus végzi a maga külön-külön feladatát. A fork végrehajtásának pillanatában minden változójuk értéke azonos, mivel a szülő adatai másolódtak át a gyermek létrehozásakor. A későbbi módosítások azonban már függetlenek egymástól. (A program kódján a szülő és a gyermek osztozik, hiszen ez nem változtatható.) A fork által visszaadott érték a gyermek számára 0, a szülő számára a gyermek **PID** (processzusazonosító). A visszaadott PID-azonosító értéke alapján tudják a processzusok eldönteni, hogy melyikük a szülő- és melyikük a gyermekprocesszus.


```
#define TRUE 1
while (TRUE) {                                  /* ismételd örökké */
   type_prompt();                               /* prompt megjelenítése a képernyőn */
   read_command(utasitas, parameterek);         /* olvass bemenő adatokat a terminálról */

   if (fork() != 0) {                           /* indíts el egy gyermekeljárást */
      /* Szülő kódja */
      waitpid(-1, &status, 0);                  /* várj, amíg a gyermekeljárás kilép */
   } else {
      /* Gyermek kódja */
      execve(utasitas, parameterek, 0);         /* hajtsd végre az utasítást */
   }
}
```

1.10. ábra. *Egy lecsupaszított parancsértelmező. A* TRUE*-ról az egész könyvben feltételezzük, hogy 1-nek van definiálva*

A fork-ot követően a gyermeknek legtöbbször a szülőétől különböző programot kell végrehajtania. Nézzük a parancsértelmezőt. A terminálról olvas egy parancsot, fork-kal létrehoz egy gyermekprocesszust, várakozik arra, hogy a gyermek végrehajtsa a parancsot, ezt követően pedig olvassa a következő parancsot. A szülő a waitpid rendszerhívás végrehajtásával várja a gyermek megszűnését; ez csak a gyermek (vagy az egyik gyermek, ha több is van) megszűnéséig várakoztat. A waitpid végrehajtása után a második *(statloc)* paraméterében adott címen elhelyezi a gyermek exit státusát (normális vagy abnormális megszűnés, illetve exit státus). Különböző lehetőségek (opciók) is beállíthatók, amit a harmadik paraméter határoz meg. A waitpid helyettesíti a korábbi elavult wait hívást, amelyet csak a visszamenőleges kompatibilitás miatt soroltunk fel.

Most nézzük, hogyan használja a parancsértelmező a fork-ot. Ha egy parancsot begépeltünk, a parancsértelmező létrehozza az új processzust. Ez a gyermekprocesszus fogja végrehajtani a felhasználói parancsot. Ezt az execve rendszerhívás végrehajtásával teszi, amely a teljes memóriatérképét felülírja az első paraméterként adott fájl tartalmával. (Tulajdonképpen maga a rendszerhívás az exec, de ezt számos különböző könyvtári eljárás hívja más-más paraméterezéssel és alig különböző névvel.) Egy leegyszerűsített parancsértelmezőt illusztrál az 1.10. ábra a fork, a waitpid és az execve használatára.

Általános esetben az execve-nek három paramétere van: a végrehajtandó fájl neve, az argumentumvektorra mutató pointer és a környezeti változók vektorára mutató pointer. Ezeket most röviden ismertetjük. Néhány további könyvtári eljárás is rendelkezésünkre áll, például az *execl, execv, execle, execve,* amelyek megengedik bizonyos paraméterek elhagyását vagy más módon való megadását. Könyvünkben az exec nevet használjuk mint ezek reprezentatív rendszerhívását.

Nézzük a

```
cp file1 file2
```

parancsot, amely a *file1* tartalmát átmásolja a *file2* fájlba. Miután a parancsértelmező létrehozta, a gyermekprocesszus megkeresi és végrehajtja a *cp* fájlt a forrás- és a célfájl neveinek átadásával.

A *cp* (és többnyire minden más C nyelvű) program főprogramja a következő deklarációt tartalmazza:

```
main(argc, argv, envp)
```

ahol az *argc* a parancssorbeli elemek darabszáma, beleértve a program nevét is, példánkban az *argc* 3.

A második paraméter, az *argv* egy vektorra mutató pointer. A vektor *i*-edik eleme a parancssor *i*-edik karaktersorozatára mutató pointer, példánkban *argv*[0] a „cp", az *argv*[1] a „file1", míg az *argv*[2] a „file2" karaktersorozatra mutat.

A *main* harmadik paramétere, az *envp* a környezetre mutató pointer, amely egy *name=value* alakú értékadó karaktersorozatokat tartalmazó vektor. Ezekkel tudunk olyan információkat közölni a programokkal, mint például a terminál típusa vagy a home könyvtár neve. Az 1.10. ábrán a környezetet nem adjuk át a gyermekprocesszusnak, ezért 0 az *execve* harmadik paramétere.

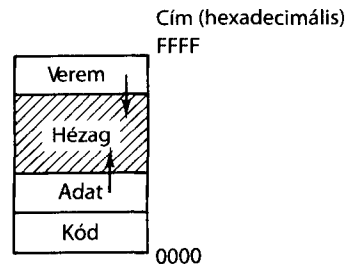
Ne essünk kétségbe, ha az exec-et bonyolultnak találjuk; ez a (szemantikusan) legösszetettebb rendszerhívás. Az összes többi sokkal egyszerűbb. Hogy lássunk egy egyszerűbbet is, nézzük az exit-et, amelyet minden programnak végrehajtása befejezésekor végre kell hajtania. Az exit státus (0–255) az egyetlen paramétere; ezt kapja vissza a szülő a waitpid rendszerhívás *statloc* paraméterében. A státus alacsonyabb helyi értékű bájtja a megszűnési státust tartalmazza; ez 0 normális megszűnéskor, egyéb értéke különböző hibafeltételeket jelent. A magasabb helyi értékű bájton a gyermek exit státusa van. Ha például a szülőprocesszus az

```
n = waitpid(-1, &statloc, options);
```

utasítást hajtja végre, akkor mindaddig felfüggesztődik, míg valamelyik gyermekprocesszusa meg nem szűnik. Ha egy gyermek például *exit*(4) végrehajtásával szűnik meg, akkor a szülő feléledésekor *n* értéke a gyermek *pid* azonosítója, a *statloc* értéke pedig 0x0400 lesz. (Könyvünkben a C írásmódnak megfelelő 0x prefixet használjuk hexadecimális számok jelölésére.)

A MINIX 3 processzusainak memóriáját három szegmensre tagoljuk: a **kódszegmens** (a program kódja), az **adatszegmens** (a változók) és a **veremszegmens**. Az 1.11. ábra mutatja, hogy az adatszegmens felfelé, míg a veremszegmens lefelé nő. Köztük található a nem használt címtartomány hézaga. A verem szükség szerint automatikusan terjeszkedik a hézagban, de az adatszegmens bővítéséhez a brk rendszerhívás explicit végrehajtása szükséges, amely azt a címet adja meg, hogy hol fejeződjék be az adatszegmens. Ez a cím lehet magasabb (az adatszegmens nő) vagy alacsonyabb (az adatszegmens csökken), mint az adatszegmens aktuális végcíme. Természetesen a veremmutató értékénél alacsonyabb paramétert kell használnunk, különben az adat- és veremszegmens átfednék egymást, ami megengedhetetlen.

A kényelmesebb programozás kedvéért egy *sbrk* könyvtári eljárás is rendelkezésünkre áll az adatszegmens méretének változtatására, ennek egyetlen paramétere



1.11. ábra. *A processzusok három szegmensből állnak: kód, adat és verem. Ebben a példában egy címtartomány van, de megengedett a külön-külön kód- és adatcímtartomány is*

az adatszegmenshez adandó bájtok számát tartalmazza (negatív érték csökkenti az adatszegmenst). Úgy működik, hogy lekérdezi az adatszegmens aktuális méretét; ez a brk visszaadott értéke, majd kiszámolja az új méretet, végül egy hívással erre állítja a méretet. A POSIX szabvány azonban nem definiál brk és sbrk hívásokat. Dinamikus helyfoglaláshoz a *malloc* könyvtári eljárás használatát javasolják a programozóknak, a *malloc* alapját képező megvalósítást ugyanis nem tartották alkalmasnak szabványosításra, mivel kevés programozó használja azt közvetlenül.

A getpid processzuskezelő rendszerhívás úgyszintén egyszerű. Visszaadja a hívó processzus PID-értékét. Emlékezzünk arra, hogy a fork végrehajtásakor csak a szülő kapja meg a gyermek PID-értékét. A gyermek a getpid végrehajtásával kaphatja meg saját PID-értékét. A getpgrp a hívó processzus csoportjának PID-értékét adja vissza. A setsid egy új szekciót hoz létre, ennek processzuscsoportja a hívó PID-értékét kapja. A szekció a POSIX-készlet opcionális **feladatvezérlés** fogalmával kapcsolatos; ez a MINIX 3-ban nincs implementálva, nem is foglalkozunk vele többet.

Utoljára említjük a ptrace processzuskezelő rendszerhívást, amelyet tesztelő programok használhatnak a tesztelt programok vezérlésére. Megengedi a tesztelt program memóriájának olvasását, írását és egyéb kezelési lehetőségeket ad.

1.4.2. Szignálkezelő rendszerhívások

Bár a processzusok közötti kommunikáció többnyire tervezett módon folyik, vannak esetek, amikor váratlan kapcsolatteremtésre van szükség. Például ha véletlenül egy nagyon hosszú fájl teljes tartalmának nyomtatását kérjük egy szövegszerkesztőtől, majd észrevesszük, hogy hibáztunk, valahogy meg kell állítanunk a szerkesztőt. A MINIX 3-ban a CTRL-C billentyű leütésével küldhetünk szignált a szerkesztőnek. A szerkesztő megkapja a szignált, és erre leállítja a nyomtatást. Szignálok alkalmasak a hardver által felderített csapdák jelzésére is: ilyenek az érvénytelen utasítás-végrehajtás vagy a lebegőpontos túlcsordulás. Az időintervallumok lejártát szintén szignálokkal implementálják.

Ha a szignált egy olyan processzus kapja, amely nem jelezte, hogy hajlandó ezt a szignált fogadni, a processzus minden további nélkül megszűnik. A processzus

a sigaction rendszerhívás végrehajtásával kerülheti el ezt a sorsot. Ezzel a hívással jelzi, hogy felkészült bizonyos típusú szignálok fogadására, továbbá megadja a szignálkezelő eljárásának a címét és egy változót, ahol az aktuális szignálkezelő eljárás címe elhelyezhető. A sigaction hívást követő, ennek megfelelő típusú szignál megjelenésekor (például a CTRL-C leütése) a processzus állapota saját vermébe mentődik, majd a szignálkezelő eljárás hívódik. Ez addig fut, amíg szükséges, sőt feladatától függően még további rendszerhívásokat is végrehajthat. A szignálkezelő eljárások a gyakorlatban leginkább igen rövidek. Ha a szignálkezelő eljárás befejeződött, a sigreturn hívást hajtja végre; ezzel a processzus ott folytatódik, ahol a szignál érkezésekor megszakadt. A sigaction váltotta fel a régebbi signal hívást; ez utóbbi könyvtári eljárásként még elérhető, de csak a visszafelé kompatibilitás megőrzése érdekében.

A szignálok blokkolhatók a MINIX 3-ban. Egy blokkolt szignál függő állapotban marad blokkolásának feloldásáig. Nem veszik el, de nem is küldjük el. A processzus a sigprocmask hívással definiálhatja a blokkolt szignálok halmazát oly módon, hogy egy bitvektort (maszkot) határoz meg. Lehetséges az is, hogy a processzus lekérdezze az aktuálisan függő állapotú, de blokkolt állapotuk miatt el nem küldhető szignálok halmazát. A sigpending hívás adja vissza ezt a halmazt bitvektor formájában. Végül a sigsuspend hívás szolgál arra, hogy a processzus egyenként állíthassa be a blokkolt szignálok bitvektorait és függeszthesse fel saját magát.

A szignálkezelő eljárás megadása helyett a processzus a SIG_IGN konstanst is megadhatja, ezzel a később megjelenő, adott típusú szignálok figyelmen kívül hagyását kéri, a SIG_DFL konstanssal pedig helyreállíthatja a szignálra vonatkozó alapeljárást. Az alapeljárás szignáltípusonként különböző, vagy a processzus megszüntetését, vagy a szignál figyelmen kívül hagyását jelenti. A SIG_IGN használatának bemutatására nézzük meg, mi történik, ha egy

```
command &
```

parancsra a parancsértelmező létrehoz egy háttérprocesszust. Nem volna kívánatos, ha a (CTRL-C lenyomásával keletkező) SIGINT szignál zavarná a háttérprocesszust, ezért a fork után, de még az exec előtt a parancsértelmező végrehajtja a

```
sigaction(SIGINT, SIG_IGN, NULL);
```

és

```
sigaction(SIGQUIT, SIG_IGN, NULL);
```

hívásokat; ezzel a SIGINT és a SIGQUIT szignálok letiltását kéri. (A SIGQUIT szignált a CTRL-\ generálja; ez ugyanaz, mint a SIGINT, azzal a különbséggel, hogy ha a processzus nem fogadja vagy nem kéri a figyelmen kívül hagyását, akkor a processzus megszűnik, és elkészül a memóriaképének fájlmásolata.) Ha nem háttérprocesszusról van szó, ezek a szignálok nem lesznek figyelmen kívül hagyva.

A CTRL-C leütése nem az egyetlen mód szignál küldésére. A kill rendszerhívással egy processzus egy másiknak tud szignált küldeni (ha közös az UID-azonosítójuk – független processzusok ugyanis nem küldhetnek szignált egymásnak). Tételezzük fel az előbbi példában, hogy a háttérprocesszus már elindult, de kiderül, hogy meg kell állítani. A SIGINT és a SIGQUIT szignálokat hatástalanítottuk, valami egyébre van szükség. A megoldás a *kill* parancs végrehajtása, amely a kill rendszerhívással bármelyik processzusnak képes szignált küldeni. Ha a 9-es (SIGKILL) szignált elküldjük bármelyik háttérprocesszusnak, akkor az megszűnik. A SIGKILL nem kezelhető le és nem hagyható figyelmen kívül.

A valós idejű alkalmazásokban gyakori eset, hogy a processzusok meghatározott időintervallumonként megszakítandók valamilyen tevékenység végrehajtására, például a megbízhatatlan kommunikációs vonalakon esetleg elveszett üzenetek újraküldésére. Erre való az alarm rendszerhívás. Paramétere másodpercekben megad egy időintervallumot, amelynek elteltével a processzus egy SIGALRM (ébresztőóra) szignált kap. Egy processzusnak egyszerre csak egy ébresztőóra-beállítása lehet. Így, ha egy alarm hívás történik 10 másodperces paraméterrel, majd 3 másodperccel később egy másik 20 másodperces paraméterrel, akkor csak egy szignált fogunk kapni, mégpedig a második hívás után 20 másodperccel. Az első hívást semmissé teszi a második alarm hívás. Ha az alarm paramétere 0, akkor minden korábbi ébresztőóra-beállítás megsemmisül. Ha egy ébresztőóra-szignált nem fogadunk, akkor az alapeljárás érvényesül, azaz a processzus megszűnik.

Az az eset is előfordul, hogy szignál érkezéséig nincs teendője a processzusnak. Nézzük például azt a számítógéppel segített oktatóprogramot, amely az olvasás sebességét és megértését teszteli. Megjeleníti a szöveget a képernyőn, majd hívja az alarm-ot 30 másodperces paraméterrel. Amíg a tanuló olvassa a szöveget, a programnak nincs teendője. Esetleg várakozhatna egy üres ciklusban, de ezzel a más processzusok vagy felhasználók számára hasznos CPU-időt vesztegetné el. Jobb ötlet a pause végrehajtása, amely a MINIX 3-mal azt közli, hogy a következő szignál érkezéséig függessze fel a processzust.

1.4.3. Fájlkezelő rendszerhívások

Igen sok rendszerhívás a fájlrendszerekkel kapcsolatos. Ebben a szakaszban az egyedi fájlokra vonatkozó hívásokkal foglalkozunk. A következő szakasz a könyvtárak vagy a fájlrendszerek egészére vonatkozó hívásokat fogja tárgyalni. Új fájl létrehozására a creat hívás szolgál. (Hogy miért creat és nem create, már az idő homályába veszett.) Paraméterei a fájl nevét és védelmi módját jelentik. Az

```
fd = creat("abc", 0751);
```

hívás egy *abc* nevű fájlt az oktális 0751 védelmi kóddal hoz létre. (C-ben a vezető 0 oktális konstanst jelez.) A 0751 utolsó 9 bitje jelöli az *rwx* biteket a tulajdonosra (a 7 olvasási-írási-végrehajtási jog), a csoportra (az 5 olvasási-végrehajtási jog) és a többiekre (az 1 csak végrehajtási jog) vonatkozóan.

A creat nemcsak létrehozza, hanem a módtól függetlenül, írásra meg is nyitja a fájlt. A visszaadott *fd* fájlleíró használható a fájl írásakor. Ha a creat már létező

fájlra hajtódik végre, ez 0 hosszúságúra rövidül, természetesen csak ha a jogosultságok rendben vannak. A creat hívás elavult, mivel az open is létrehozhat fájlokat, csak a visszafelé kompatibilitás miatt soroltuk fel.

Specifikus fájlokat nem a creat-tel, hanem az mknod-dal hozunk létre. Például az

```
fd = mknod("/dev/ttyc2", 020744, 0x0402);
```

hívás létrehozza a */dev/ttyc2* nevű fájlt (ez a 2. konzol szokásos neve) a 020744 oktális móddal (karakterspecifikus fájl az *rwxr--r--* védelmi bitekkel). A harmadik paraméter magasabb helyi értékű bájtja a főeszközt (4), alacsonyabb helyi értékű bájtja a mellékeszközt (2) jelöli. A főeszköz bármi lehet, de a */dev/ttyc2* nevű szokásosan a 2-es mellékeszköz szokott lenni. Az mknod hívás hibára vezet, ha nem a szuperfelhasználó hajtja végre.

Létező fájl olvasása vagy írása előtt a fájlt open-nel meg kell nyitni. A hívásban specifikáljuk a megnyitandó fájl teljes útvonalnevét vagy a munkakönyvtárhoz képest a relatív útvonalnevét, továbbá az *O_RDONLY*, az *O_WRONLY* és az *O_RDWR* kódok valamelyikét; ezzel az olvasásra, írásra vagy mindkettőre való megnyitást kérve. A visszakapott fájlleíró használható ezután az olvasásra, írásra. Végül a fájlt le kell zárni a close hívással; ezzel a fájlleíró további creat vagy open hívásokban újrafelhasználhatóvá válik.

Kétségkívül a read és a write hívásokat használjuk a legtöbbet. A read-ről már szóltunk korábban; a write-nak ugyanazok a paraméterei.

A programok többsége a fájlokat szekvenciálisan olvassa és írja, néhány felhasználói programnak azonban szüksége lehet valamely fájl véletlenszerűen kiválasztott tetszőleges részéhez való hozzáférésre. Minden fájlhoz tartozik egy pointer, amely a fájl aktuális pozíciójára mutat. Ha szekvenciálisan olvasunk (írunk), a pointer a legközelebb olvasandó (írandó) bájtra mutat. Az lseek hívás szolgál a pozíciómutató értékének a módosítására, így az ezt követő read és write hívások indíthatók a fájl tetszőleges helyéről, esetleg a végéről is.

Az lseek háromparaméteres: a fájlleíró, a kívánt fájlpozíció, a harmadik pedig azt közli, hogy a fájlpozíció a fájl elejéhez, az aktuális pozíciójához vagy a végéhez képest relatív. Az lseek visszaadott értéke a mutató változása utáni fájlbeli abszolút pozíció.

A MINIX 3 minden fájlra vonatkozóan megőrzi a fájl típusát (közönséges vagy specifikus fájl, könyvtár vagy egyéb), méretét, utolsó módosítási idejét és más információkat. A programok ezen információkat a stat és az fstat rendszerhívásokkal kérhetik meg. Csak abban különbözik a kettő, hogy a fájlt az előbbi a nevével, utóbbi pedig a fájlleírójával specifikálja. Az fstat használható megnyitott fájlokra, különösen ha a fájl nevét nem is ismerjük, például a standard bemeneti és kimeneti fájlok esetében. Mindkét hívás második paramétere egy struktúra címe, ahová az információk elhelyezését kérjük. Az 1.12. ábrán látható a struktúra.

Amikor fájlleírókkal dolgozunk, a dup hívás hasznos lehet. Például nézzük azt a programot, amelyik lezárja a standard kimenetét (fájlleírója 1), standard kimenetként egy másik fájlt határoz meg, meghív egy függvényt, amely az eredményeit


```
struct stat {
    short st_dev;           /* az i-csomóponthoz tartozó eszköz */
    unsigned short st_ino;  /* az i-csomópont száma */
    unsigned short st_mode; /* a mód */
    short st_nlink;         /* a linkek száma */
    short st_uid;           /* a felhasználó uid azonosítója */
    short st_gid;           /* a csoport gid azonosítója */
    short st_rdev;          /* fő- vagy mellékeszköz specifikus fájlokhoz */
    long st_size;           /* fájlméret */
    long st_atime;          /* az utolsó hozzáfordulás időpontja */
    long st_mtime;          /* az utolsó módosítás időpontja */
    long st_ctime;          /* az utolsó i-csomópont módosítás időpontja */
};
```

1.12. ábra. *A* stat *és* fstat *rendszerhívások által visszaadott információ struktúrája. A tényleges programban egyes típusokat szimbolikus nevek jelölnek*

a standard kimenetre írja, végül helyreállítja az eredeti állapotot. Az 1 értékű fájlleíró lezárása és egy új fájl megnyitása pontosan azt eredményezi, hogy az új fájl lesz a standard kimenet (feltételezve, hogy a standard bemenet 0 értékű fájlleírója használatban van), de az eredeti állapot ezek után már nem állítható helyre.

Megoldásként először hajtsuk végre az

```
fd = dup(1);
```

utasítást, az ebben szereplő dup rendszerhívás ad egy új *fd* fájlleírót, amelyik ugyanarra a standard kimeneti fájlra való hivatkozás. Most már a standard kimenet lezárható és az új fájl megnyitható, használható. Amikor az eredeti állapot helyreállításának itt az ideje, az 1 értékű fájlleírót lezárjuk, ezt követően az

```
n = dup(fd);
```

végrehajtásával újra megkapjuk a legalacsonyabb, nevezetesen az 1 értékű fájlleírót, amely az *fd*-vel azonos fájlra hivatkozik. Az *fd* lezárásával végül a kiindulási állapotba jutunk vissza.

A dup hívás egy variánsa azt teszi lehetővé, hogy tetszőleges használaton kívüli fájlleírót hozzárendeljünk egy már megnyitott fájlhoz. Ez a

```
dup2(fd, fd2);
```

hívással történik, ahol *fd* a megnyitott fájl leírója, *fd2* pedig egy használaton kívüli leíró, amely ezek után az *fd*-vel azonos fájlra fog hivatkozni. Ha *fd* a standard bemenetre hivatkozik (0 értékű fájlleíró) és *fd2* értéke 4, akkor a fenti hívás után a 0 és 4 értékű fájlleírók mindegyike a standard bemenetre hivatkozik.

Már említettük, hogy a MINIX 3 a processzusok közötti kommunikációra adatcsöveket használ. Ha a felhasználó a

```
cat file1 file2 | sort
```

parancsot begépeli, a parancsértelmező létrehoz egy adatcsövet, megszervezi azt, hogy az első processzus standard kimenetén az adatcsőbe írjon, míg a második processzus a standard bemenetén az adatcsőből olvasson. A pipe rendszerhívás létrehoz egy adatcsövet, továbbá két fájlleírót ad vissza: egyik írásra, a másik olvasásra alkalmas. A

```
pipe(&fd[0]);
```

hívásban az *fd* két egész értékből álló vektor, *fd*[0] az olvasásra, *fd*[1] pedig az írásra szolgáló fájlleíró lesz. Többnyire ezt a hívást egy fork követi, a szülő az olvasásra, a gyermek az írásra szolgáló fájlleírót lezárja (vagy fordítva), miáltal az egyik processzus írni tud, a másik pedig olvasni tud ugyanabból az adatcsőből.

Az 1.13. ábrán egy mintaprogramot olvashatunk, amely két processzust hoz létre, egyikük az eredményeit adatcsövön keresztül adja át a másiknak. (Valósághűbb

```
#define STD_INPUT 0                 /* a standard bemenet fájlleírója */
#define STD_OUTPUT 1                /* a standard kimenet fájlleírója */

pipeline(process1, process2)
char *process1, *process2;          /* a programnevekre mutató pointerek */
{
  int fd[2];

  pipe(&fd[0]);                     /* az adatcső létrehozása */
  if (fork() != 0) {
    /* Ezt a programrészletet a szülőprocesszus hajtja végre */
    close(fd[0]);                   /* az 1. processzusnak az adatcsőből nem kell olvasnia */
    close(STD_OUTPUT);              /* az új standard kimenet előkészítése */
    dup(fd[1]);                     /* a standard kimenetet az fd[1]-hez rendeljük */
    close(fd[1]);                   /* ez a fájlleíró többé nem kell */
    execl(process1, process1, 0);
  } else {
    /* Ezt a programrészletet a gyermekprocesszus hajtja végre */
    close(fd[1]);                   /* a 2. processzusnak az adatcsőbe nem kell írnia */
    close(STD_INPUT);               /* az új standard bemenet előkészítése */
    dup(fd[0]);                     /* a standard kimenetet az fd[0]-hoz rendeljük */
    close(fd[0]);                   /* ez a fájlleíró tőbbé nem kell */
    execl(process2, process2, 0);
  }
}
```

1.13. ábra. *Egy két processzust összekötő adatcső létrehozásának vázlata*

lenne egy hibát elemző és argumentumokat is kezelő példa.) Először az adatcső jön létre, majd a fork után a szülő lesz az adatcső egyik processzusa, a gyermek pedig a másik. A végrehajtandó *process1* és *process2* programfájlok nem tudják, hogy egy adatcső részei, ezért a fájlleírókat kell úgy beállítani, hogy az első processzus standard kimenete a második processzus standard bemenete legyen. A szülőprocesszus lezárja az adatcső olvasására szolgáló fájlleíróját. Ezt követően lezárja standard kimenetét, a dup hívással eléri, hogy az 1 értékű fájlleíróval az adatcsőbe írhasson. Jegyezzük meg, hogy a dup mindig a legalacsonyabb értékű, használaton kívüli fájlleírót adja vissza; esetünkben ez 1. Végül a szülő lezárja az adatcsőre hivatkozó másik fájlleírót.

Az exec hívással elindított program a 0 és a 2 értékű fájlleírókat érintetlenül megkapja, az 1 értékű fájlleíróval pedig írhat az adatcsőbe. A gyermek programja hasonló. Az *execl* hívás paramétere ismétlődik, lévén az első a végrehajtandó programfájl neve, a második pedig a program első argumentuma. Első argumentumként a legtöbb program a programfájl nevét várja.

Az ioctl rendszerhívás alkalmazható a specifikus fájlok többségére. Ezt használjuk az SCSI szalag és CD-ROM-eszközök és más hasonló blokkspecifikus eszközök vezérlésére. Elsődleges feladata azonban a karakterspecifikus fájlokkal, főként a terminálokkal kapcsolatos. Több, a POSIX által definiált függvényt a programkönyvtár ioctl hívásokkal implementál. A *tcgetattr* és a *tcsetattr* könyvtári függvények ioctl hívásokkal hajtják végre az olyan feladatokat, mint a gépelési hibákat javító karakterek beállítása, a **terminálmód** megváltoztatása, és így tovább.

Hagyományosan három terminálmód van: a feldolgozott, a nyers és a cbreak mód. A **feldolgozott mód** a terminálok módjának alaphelyzete. Ebben a módban a visszaléptető és törlő karakter működik, a CTRL-S és CTRL-Q a terminálra írást megállítja, illetve újraindítja, a CTRL-D fájl végét jelent, a CTRL-C megszakítás szignált generál, míg a CTRL-\ a kilépési szignált és ezzel memóriatérkép fájlmásolatának előállítását eredményezi.

Nyers módban a karakterek előbbi funkciói nem léteznek, minden karakter feldolgozatlanul jut el közvetlenül a programokhoz. Ebben a módban a terminálra vonatkozó minden read hívás bármilyen leütött karaktert átad a programnak, töredéksorokat is, és nem vár a sor teljes begépelésére, mint a feldolgozott módban. A képernyőt használó szövegszerkesztők gyakran ezt a módot használják.

A **Cbreak mód** a kettő közötti átmenet. A visszaléptető, törlő és a CTRL-D karakterek szerkesztésre nem használhatók, a CTRL-S, CTRL-Q, CTRL-C és CTRL-\ funkciói azonban a feldolgozott módnak megfelelők. A nyers módhoz hasonlóan, a programok töredéksorokat is megkapnak. (Sorközi tévedéseit a felhasználó nem tudja törlésekkel rendbe hozni, mint a feldolgozott módban. Ha a sorközbeni szerkeszthetőséget nem engedjük meg, akkor viszont a programoknak nem kell a teljes sor begépeléséig várakozniuk.)

A POSIX nem a feldolgozott, nyers és cbreak terminológiát használja. A POSIX **kanonikus mód** kifejezése a feldolgozott módnak felel meg. Ebben a módban 11 speciális karakternek van funkciója, és a bemenet soronként valósul meg. A **nem kanonikus módban** előírható a karakterkészlet minimális száma, továbbá megadható egy időintervallum tizedmásodperces egységekben, ami alatt egy read

hívásnak teljesülnie kell. A POSIX flexibilitásra törekszik; különböző indikátorok beállításával a nem kanonikus módból akár nyers, akár cbreak mód kialakítható. A régebbi terminológia kifejezőbb, informálisan ezt fogjuk használni.

Az ioctl hívás háromparaméteres; példaképpen a *tcsetattr* eljárásban a terminál paramétereit az

```
ioctl(fd, TCSETS, &termios);
```

rendszerhívás fogja beállítani. Az első paraméter a fájlt, a második a műveletet azonosítja, a harmadik pedig egy POSIX-struktúra címe, amely az indikátorokat és a vezérlő karakterek vektorát tartalmazza. További műveletek szolgálnak arra, hogy a változtatások hatásait a rendszer elhalassza a folyamatban lévő kimenet teljesítésének befejezéséig, hogy a be nem olvasott bemenetet megsemmisítsük és az aktuális beállítást lekérdezzük.

Az access rendszerhívással kérdezhetjük le, hogy a védelmi rendszer engedélyez-e egy bizonyos fájlhoz való hozzáférést. Erre szükség van, mert ugyanaz a program különböző felhasználói UID-azonosítóval is futtatható. A SETUID lehetőségeit később ismertetjük.

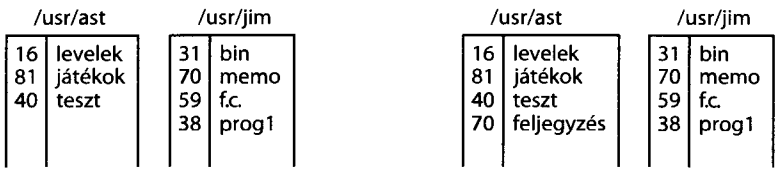
Egy fájlnak új nevet a rename rendszerhívással adhatunk. Paraméterei a régi és az új nevet specifikálják.

Végül az fcntl hívás az ioctl-hez kissé hasonlóan fájlok vezérlésére szolgál (mindkettőnek elég fáradságos a használata). Opciói közül a fájlzárolásokra szolgálók a legfontosabbak. A processzusok az fcntl hívást használhatják fájlok részeinek zárolására és felszabadítására vagy az egyes részek zároltságának vizsgálatára. Maga a rendszerhívás nem definiál zárolási mechanizmust, a programoknak kell egyezményes szemantikára épülniük.

1.4.4. Könyvtárkezelő rendszerhívások

Ebben a szakaszban azokról a rendszerhívásokról szólunk, amelyek inkább könyvtárakkal vagy a fájlrendszer egészével kapcsolatosak, mintsem az előző szakaszbeli egyedi fájlokra vonatkozó hívások. Az mkdir, illetve rmdir üres könyvtárakat hoz létre, illetve szüntet meg. A link hívás biztosítja azt, hogy egy fájl két vagy több néven is szerepelhessen, esetleg különböző könyvtárakban is. Bevált használata az, hogy egy programozócsoport tagjai egy közös fájlon osztoznak; mindegyikük a saját könyvtárában látja a fájlt, esetleg még különböző névvel is. Az, hogy a csoport tagjai osztoznak egy közös fájlon, nem azt jelenti, hogy mindegyikük kap egy saját másolatot, hanem azt, hogy ha bármelyikük módosít a tartalmán, akkor a többiek ezt azonnal észlelik, hiszen egyetlen fájlról van szó. Ha másolatokat készítenénk a fájlról, akkor az egyik másolaton végrehajtott későbbi módosítás a többire hatástalan lenne.

Az 1.14.(a) ábrán szemléltetjük a link működését. Ezen *ast* és *jim* két felhasználó, mindkettőjüknek van saját könyvtára, benne fájlokkal. Ha *ast* végrehajtja egy programjában a



1.14. ábra. *(a) Két könyvtár a* link *végrehajtása előtt. (b) Ugyanezek a végrehajtás után*

```
link("/usr/jim/memo", "/usr/ast/note");
```

rendszerhívást, akkor a *jim* könyvtárában lévő *memo* nevű fájl megjelenik az *ast* könyvtárában *note* néven. Ezután a */usr/jim/memo* és a */usr/ast/note* nevek ugyanarra a fájlra hivatkoznak.

A link működésének megértéséhez valószínűleg hozzájárul egy alaposabb vizsgálat. A Unixban minden fájlt egy egyedi szám, az i-szám azonosít. Fájlonként a fájl i-száma egy index az ún. **i-csomópont** (vagy i-csomó) táblázatban, ahol a fájl tulajdonosát, lemezblokkjainak helyét és egyebeket tárolunk. Egy könyvtár nem más, mint egy fájl, amelyben (i-szám, név) párokat tárolunk. A Unix első változataiban egy könyvtárbejegyzés 16 bájtból állt – 2 bájt az i-szám, 14 bájt a fájl neve számára. A hosszú fájlnevek támogatásához ennél összetettebb struktúra szükséges, de alapötletét tekintve a könyvtár továbbra is (i-szám és ASCII név) párok halmaza. Az 1.14. ábrán a *levelek* i-száma 16. A link egyszerűen készít egy új könyvtárbejegyzést (esetleg új névvel), ebben egy már létező fájl i-számát használja. Az 1.14.(b) ábrán két bejegyzésnek ugyanaz az i-száma (70); ezek ugyanarra a fájlra való hivatkozások. Ha az egyiket megszüntetjük az unlink rendszerhívással, a másik megmarad. Ha mindkét bejegyzést megszüntetjük, akkor a Unix észreveszi, hogy nincs a fájlra hivatkozó bejegyzés (az i-csomópontban egy mező őrzi az egy fájlra hivatkozó könyvtárbejegyzések számát), ezért törli a lemezről.

Már említettük korábban, hogy a mount rendszerhívás alkalmas két fájlrendszer egyesítésére. Általában van egy gyökérfájlrendszerünk a merevlemezen, ebben tároljuk a parancsaink bináris (végrehajtható) programjait és egyéb gyakran használt fájljainkat. A felhasználó ezenkívül például a CD-ROM-meghajtóba helyezheti a saját programjait tartalmazó lemezét.

A mount rendszerhívás végrehajtásával a CD-ROM-meghajtó fájlrendszere felcsatolható a gyökérfájlrendszerre; ezt mutatja az 1.15. ábra. Egy tipikus C programbeli utasítás a mount végrehajtására a

```
mount("/dev/cdrom0", "/mnt", 0);
```

hívás, ahol az első paraméter a CD-ROM-meghajtó blokkspecifikus fájljának a neve, a második a fában az a hely, ahová a felcsatolás történik, a harmadik pedig azt mondja meg, hogy a felcsatolandó fájlrendszer írható és olvasható, vagy csak olvasható.

1.15. ábra. *(a) Fájlrendszer felcsatolás előtt. (b) Fájlrendszer felcsatolás után*

A mount hívás után a CD-ROM-meghajtó bármelyik fájlja elérhető a gyökér- vagy a munkakönyvtártól induló útvonalnévvel, függetlenül attól, hogy melyik meghajtón van. Második, harmadik és további meghajtók is felcsatolhatók a fa tetszőleges pontjára. A mount adja a lehetőséget arra, hogy eltávolítható médiumokat egyetlen egységes hierarchiába rendezzünk, és ne kelljen azzal foglalkoznunk, hogy egy fájl melyik meghajtón van. Példánk a CD-ROM lemezről szólt, de merevlemezek vagy merevlemezek részei (**partíciók** vagy **mellékeszközök**) ugyanígy csatolhatók fel. Ha egy fájlrendszerre már nincs szükség, az umount rendszerhívással lecsatolhatjuk.

A MINIX 3 a memóriában egy **átmeneti tárolót (cache)** tart fenn a korábban használt blokkok őrzésére. Ezzel elkerüli a lemezről való újbóli olvasásukat, ha rövidesen újra szükségesek. Ha az átmeneti tárolóban egy blokk módosul (a fájlra vonatkozó write következtében), és a rendszer összeomlik, mielőtt a módosított blokk visszaíródna a lemezre, a fájlrendszer megsérül. Az esetleges sérülések csökkentésére időnként fontos az átmeneti tároló tartalmának visszaírása; ezzel az összeomlásokkal okozott adatvesztés mennyisége minimális maradhat. A sync rendszerhívás utasítja a MINIX 3-at, hogy az átmeneti tárolóban a beolvasásuk óta módosított blokkokat írja vissza a lemezre. A MINIX 3 indításakor egy *update* nevű háttérprogram is elindul, amely 30 másodpercenként kiad egy sync hívást az átmeneti tároló időnkénti visszaírására.

A könyvtárakkal kapcsolatos további két hívás a chdir és a chroot. Az első a munkakönyvtárat, a második a gyökérkönyvtárat változtatja. A

```
chdir("/usr/ast/test");
```

hívást követően egy *xyz* nevű fájlra vonatkozó megnyitás a */usr/ast/test/xyz* fájlt fogja megnyitni. A chroot hasonlóan működik. Ha egy processzus kérte a rendszertől a gyökérkönyvtár módosítását, akkor minden teljes (a „/" jellel kezdődő) útvonalnév az új gyökérkönyvtártól kezdődő útvonalnév lesz. Miért lehet erre szükség? Biztonsági okokból – az **FTP (File Transfer Protocol – fájlátviteli protokoll),** a **HTTP (HyperText Transfer Protocol – hiperszöveg-átviteli protokoll)** és hasonló protokollokhoz tartozó kiszolgálóprogramok így biztosítják azt, hogy a távoli felhasználók a fájlrendszernek csak az új gyökérkönyvtár alatt található részét érhessék el. Csak a szuperfelhasználó hajthatja végre a chroot-ot, de ő sem szokta gyakran.

1.4.5. A védelem rendszerhívásai

A MINIX 3-ban a védelemre minden fájlhoz egy 11 bites mód van rendelve. Ebből kilenc a tulajdonos, a csoport és a többiek olvasás-írás-végrehajtás bitjei. A fájl módját a chmod rendszerhívással változtathatjuk. Például, hogy egy fájlt csak olvashatóvá tegyünk mindenki, kivéve a tulajdonos számára, hajtsuk végre a

```
chmod("file", 0644);
```

hívást.

A két további védelmi bit, a 02000 és a 04000, a SETGID (csoportazonosító, GID-beállítás) és a SETUID (felhasználóazonosító, UID-beállítás) bit. Ha egy felhasználó úgy hajt végre egy programot, hogy a SETUID bit be van kapcsolva, akkor a végrehajtás ideje alatt a felhasználó UID-azonosítója a programfájl tulajdonosának azonosítója lesz. Ennek a funkciónak nagyon gyakori felhasználási módja a csak a szuperfelhasználó által végrehajtható tevékenységeket végző programok végrehajtásának engedélyezése más felhasználók számára is. Például a könyvtár létrehozása az mknod-dal történik; ez csak szuperfelhasználónak megengedett. Ha az *mkdir* program a szuperfelhasználó tulajdona a 04755 móddal, akkor minden felhasználó megkapja a jogot ennek végrehajtására, de csak nagyon korlátozott lehetőségekkel.

Ha egy processzus olyan fájlt hajt végre, amelynek SETUID és SETGID bitjei be vannak kapcsolva, akkor a tényleges UID- és GID-azonosítói különböznek a valódi azonosítóitól. Néha szükségünk van arra, hogy lekérdezzük a valódi és a tényleges UID- és GID-azonosítókat. A getuid és a getgid rendszerhívások erre szolgálnak. Ezek visszaadják a valódi és a tényleges azonosítókat, ezért négy könyvtári eljárás áll rendelkezésünkre: *getuid, getgid,* illetve *geteuid, getegid;* ezek rendre a valódi, illetve a tényleges UID-, GID-azonosítókat adják vissza.

Az átlagos felhasználó nem módosíthatja UID-azonosítóját, kivéve ha bekapcsolt SETUID bittel rendelkező programot hajt végre. A szuperfelhasználó viszont használhatja a setuid rendszerhívást, amely mind a valódi, mind a tényleges UID-azonosítókat beállítja. A setgid hasonló a GID-azonosítókhoz. A szuperfelhasználó a fájl tulajdonosát is módosíthatja a chown hívással. Most már láthatjuk, hogy a szuperfelhasználónak bőven van lehetősége a védelmi rendszer megsértésére (és az is világos, hogy miért fordítanak a hallgatók annyi időt a szuperfelhasználóvá válási kísérleteikre).

Ebben a kategóriában az utolsó két rendszerhívást közönséges felhasználói processzusok is végrehajthatják. Az umask egy bitvektort állít be, amely fájlok létrehozásakor azok védelmi bitjeire lesz befolyással. Az

```
umask(022);
```

végrehajtását követően a creat és az mknod védelmi paramétereiben a 022 bitek törlődni fognak. Ezért a

```
creat("file", 0777);
```

hívás a fájl védelmi módját 0755-re fogja beállítani. Az umask biteket a gyermekprocesszusok is öröklik, ezért ha a parancsértelmező a felhasználó bejelentkezésekor ezt a fenti értékre állítja, akkor véletlenül sem történhet meg, hogy a felhasználó processzusai olyan fájlokat hoznak létre, amelyeket mások is módosíthatnak.

Ha a program a szuperfelhasználó tulajdona és SETUID bitje be van kapcsolva, akkor joga van bármely fájl elérésére, hiszen a tényleges UID-azonosítója a szuperfelhasználóé. Gyakran szükség lehet arra, hogy megtudjuk, vajon az a személy, aki hívta a programot, rendelkezik-e az adott fájlhoz hozzáférési joggal. Ha megkíséreljük az ilyen fájlhoz a hozzáférést, az mindig sikerülni fog, ebből nem tudunk meg semmit.

Amire szükségünk van, az az, hogy a valódi UID-azonosítóval van-e hozzáférési jogunk. Erre szolgál az access rendszerhívás. A *mód* paraméter értéke 4 az olvasási, 2 az írási és 1 a végrehajtási jog lekérdezésére. A *mód* paraméter kombinálható is, ha például az értéke 6, akkor a visszaadott érték 0, amennyiben a valódi UID-azonosítóval az olvasási és az írási jog egyaránt megengedett; különben –1. A 0 értékű *mód* paraméterrel a fájl létezését és a hozzá tartozó útvonal kereshetőségét ellenőrizhetjük.

Annak ellenére, hogy a védelmi mechanizmusok a Unix-szerű operációs rendszerekben általában hasonlók, vannak különbségek és következetlenségek, amelyek biztonsági résekhez vezetnek. Ennek bővebb tárgyalását lásd Chen és társai könyvében (Chen et al., 2002).

1.4.6. Az időkezelés rendszerhívásai

A MINIX 3-ban négy hívás vonatkozik a rendszerórára. A time visszaadja a jelenlegi időt másodpercekben; 1970. január 1. éjfél a kezdő, 0 értékű időpont (a kezdő és nem a befejező éjfél). Természetesen a rendszerórát valamikor be is kell állítani, hogy később lekérdezhessük. Az stime hívással ezt a szuperfelhasználó teheti meg. A harmadik hívás az utime; ezzel a fájl tulajdonosa (vagy a szuperfelhasználó) tudja a fájl i-csomópontjában tárolt időt változtatni. Ennek alkalmazása eléggé behatárolt, de néhány programnak szüksége van rá. Ilyen például a *touch*, amely a fájl idejét a jelenlegi időre állítja.

Az utolsó a times hívás, amellyel a processzusról elszámolási információkat kaphatunk. Megmondja, hogy eddig közvetlenül mennyi CPU-időt használt a processzus, és azt is, hogy a rendszer mennyi időt fordított rá (rendszerhívásaink végrehajtásával). A processzus és összes gyermekei által felhasznált közvetlen és rendszeridőt is megadja.

1.5. Az operációs rendszer struktúrája

Eddig megismertük az operációs rendszert kívülről (azaz a programozói felületet), most nézzük meg belülről. A következő szakaszokban öt, már kipróbált struktúrát vizsgálunk, amellyel némi áttekintést nyerünk a lehetőségekről. Nem törekszünk teljességre, inkább ízelítőt adunk a gyakorlatban is kipróbált tervezési elvekből. Az öt struktúra a monolitikus rendszerek, a rétegelt rendszerek, a virtuális gépek, exokernelek és a kliens-szerver rendszerek.

1.5.1. Monolitikus rendszerek

Messzemenően ez a legelterjedtebb szervezési mód, de viselhetné akár a „Nagy összevisszaság" nevet is. Struktúrája a strukturálatlanság. Az operációs rendszer eljárások gyűjteménye, bármelyik hívhatja a másikat minden korlátozás nélkül. Ennél a módszernél a paraméterek és a visszaadott érték alapján minden eljárásnak jól definiált felülete van, és ha a programozó úgy gondolja, hogy eljárásában egy másik eljárás valami hasznosat nyújthat, akkor azt szabadon hívhatja.

Az operációs rendszer végrehajtható programjának előállításához először az eljárásokat, illetve az eljárások forráskódjait tartalmazó fájlokat lefordítjuk, azután a programszerkesztő segítségével az összeset egyetlen kóddá rakjuk össze. Az információelrejtés fogalma teljesen ismeretlen, minden eljárás látja az összes többit. (Ezzel ellentétes a modulokból vagy modulcsoportokból építkező tervezés, amikor is az információk döntő többségét a modulok belsejébe zárjuk, és csak az előre megtervezett belépési pontok hívhatók a modulon kívülről.)

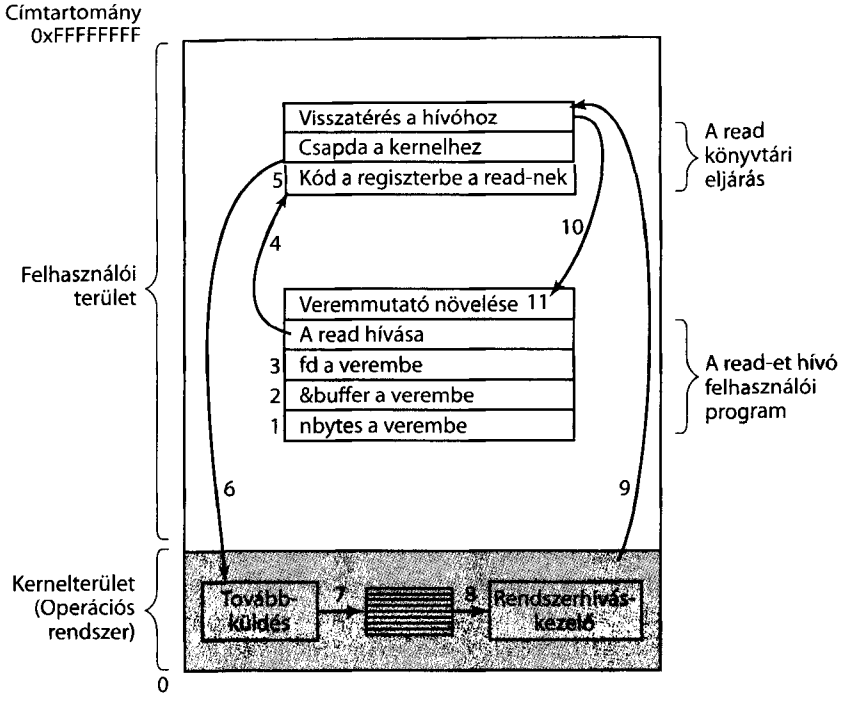
Ennek ellenére a monolitikus rendszereket is lehet kicsit strukturálni. Az operációs rendszer szolgáltatásait (a rendszerhívásokat) úgy kérjük, hogy először elhelyezzük a paramétereket egyezményes helyeken, például regiszterekben vagy a veremben, ezután pedig egy speciális, csapdázott ún. **kernelhívást** vagy **felügyelt hívást** hajtunk végre.

Ez az utasítás felhasználói módról kernel módra kapcsolja a gépet, és a vezérlést átadja az operációs rendszernek. (A legtöbb CPU két módban dolgozhat: kernel módban az operációs rendszernek dolgozik, és ekkor minden utasítás megengedett; felhasználói módban a felhasználói programoknak dolgozik, és az I/O, továbbá néhány más utasítás is tiltva van.)

Itt a jó alkalom, hogy megvizsgáljuk, hogyan hajtódnak végre a rendszerhívások. Emlékezzünk vissza, hogy a read hívást hogyan használjuk:

```
count = read(fd, buffer, nbytes);
```

A *read* könyvtári függvény hívásának előkészítéseként, amely a read rendszerhívást ténylegesen meghívja, a hívó program a paramétereket a verembe helyezi, ahogyan azt az 1.16. ábra 1–3. lépései mutatják. Történelmi okokból a C és C++ fordítók fordított sorrendben teszik a paramétereket a verembe (a magyarázat az, hogy a *printf* függvényhíváshoz az első paraméter, a formátumsztring legyen híváskor leg-

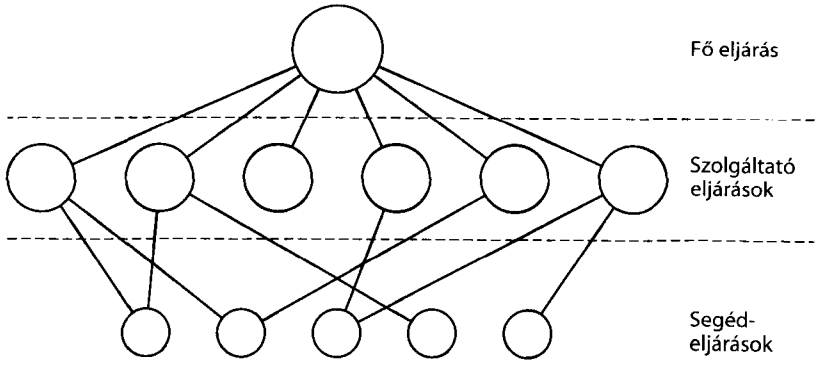


1.16. ábra. *A* read(fd, buffer, nbytes) *rendszerhívás 11 lépése*

felül). Az első és harmadik paraméterek érték szerint, a második pedig cím szerint adódik át; ez utóbbi azt jelenti, hogy a puffer címe (a & jelzi) és nem tartalma kerül átadásra. Ezután következik a tényleges könyvtári függvényhívás (4. lépés). Ez a megszokott eljáráshívó utasítás, amely tetszőleges eljárás hívására használható.

A valószínűleg assembly nyelven megírt könyvtári függvény a rendszerhívás számát rendszerint arra a helyre, például egy regiszterbe teszi, ahol az operációs rendszer azt elvárja (5. lépés). Majd a felhasználói módból kernel módba váltáshoz végrehajt egy trap utasítást, és egy kernelen belüli rögzített címtől elkezdi a végrehajtást (6. lépés). Az elinduló kernelkód megvizsgálja a rendszerhívás számát, és kiküldi a megfelelő rendszerhívás-kezelőnek, rendszerint a rendszerhívás számával indexelt, a rendszerhívás-kezelők címeit tartalmazó táblázat segítségével (7. lépés). Ekkor a rendszerhívás-kezelő lefut (8. lépés). Amikor a rendszerhívás-kezelő befejezte a munkát, a vezérlés visszaadódhat a felhasználói szinten található könyvtári eljárásnak a trap utasítást követő utasításra (9. lépés). Ez az eljárás a szokásos módon tér vissza a felhasználói programhoz (10. lépés).

A feladat befejezéséhez a felhasználói programnak helyre kell állítania a verem tartalmát, ahogyan bármelyik másik eljáráshívás után (11. lépés). Feltételezve, hogy a verem „lefelé" növekszik, ahogyan a legtöbbször történik, a lefordított kód pontosan annyival növeli a veremmutató értékét, hogy a *read* hívás előtt a verembe helyezett paraméterek eltűnjenek. A program ezután azt tehet, amit akar.



1.17. ábra. *Egy monolitikus rendszer egyszerű szerkezeti modellje*

Fentebb a 9. lépésnél jó okkal mondtuk azt, hogy „a vezérlés visszaadódhat a felhasználói szinten található könyvtári eljárásnak". A rendszerhívás ugyanis akár blokkolhatja is a hívót, ami megakadályozza a folytatását. Például ha a program a billentyűzetről vár adatot, de még nem történt gépelés, a hívót blokkolni kell. Ebben az esetben az operációs rendszer körülnéz, hogy van-e másik futtatható processzus. Később, amikor a kívánt bemeneti adat rendelkezésre áll, a processzszus visszakaphatja a vezérlést a rendszertől, és a 9–11. lépések végrehajtódnak.

Ez a szervezés utal az operációs rendszer alapstruktúrájára:

1. Főprogram; ez hívja a kívánt szolgáltató eljárásokat.
2. Szolgáltató eljárások készlete; ezek hajtják végre a rendszerhívásokat.
3. Segédeljárások a szolgáltató eljárások támogatására.

Ebben a modellben minden rendszerhíváshoz egy ezt kezelő szolgáltató eljárás tartozik. A segédprogramok a több szolgáltató eljárás által is igényelt feladatokat hajtják végre; ilyen például adatok átvétele a felhasználói programoktól. Az 1.17. ábra mutatja az eljárások így nyert háromszintű besorolását.

1.5.2. Rétegelt rendszerek

Az 1.17. ábrán látható megközelítés általánosításaként az operációs rendszer rétegekből álló hierarchia is lehet, ahol minden réteget az alatta lévőre építünk. Az első így építkező rendszert, a THE-t E. W. Dijkstra (1968) tervezte a hollandiai Technische Hogeschool Eindhoven egyetemen hallgatói közreműködésével. A THE kötegelt rendszer volt, és az Electrologica X8 holland gépen futott 27 bites szavakból álló 32 K memóriában (a bitek akkoriban költségesek voltak).

A rendszernek az 1.18. ábra szerinti 6 rétege volt. A 0. réteg végezte a processzor-hozzárendelést, a processzusok közötti átkapcsolást megszakítások jelentkezése vagy időintervallumok lejárta esetében. A 0. réteg fölött a rendszer olyan szekvenciális processzusokból állt, amelyeket már úgy lehetett programozni,

Réteg	Feladat
5	A gépkezelő
4	Felhasználói programok
3	Bemenet/kimenet kezelése
2	Gépkezelő processzus kommunikáció
1	Memória- és dobkezelés
0	Processzor-hozzárendelés és multiprogramozás

1.18. ábra. *A THE operációs rendszer struktúrája*

hogy nem kellett azzal törődni, hogy egyetlen processzoron több processzus is fut. Más szóval a 0. réteg biztosította a CPU multiprogramozhatóságát.

Az 1. réteg a memóriakezelést végezte. Lefoglalta a processzusok számára a belső memóriát és az 512 K szavas dobon a területeket. Ez utóbbin tárolták a processzusok olyan részeit (lapokat), amelyek számára nem volt hely a belső memóriában. Az 1. réteg felett a processzusoknak már nem kellett azzal törődniük, hogy memóriában vagy dobon vannak-e, mivel az 1. réteg biztosította számukra a visszatöltéshez szükséges lapokat, amikor erre szükség volt.

A 2. réteg kezelte a processzusok közötti kommunikációt és a gépkezelő konzolját. A magasabb rétegek processzusai már saját gépkezelői konzollal rendelkeztek. A 3. réteg felügyelte az I/O-eszközöket és a bejövő vagy kimenő adatfolyamok átmeneti tárolását. A 3. réteg feletti processzusok már absztrakt I/O-eszközöket érzékeltek, mentesültek a fizikai eszközök részleteinek kezelésétől. A 4. réteg a felhasználói programoké volt. Ezek mellőzhették a processzusokkal, memóriával, konzollal és I/O-eszközökkel való foglalkozást. A rendszer kezelőjének processzusa került az 5. rétegre.

A MULTICS-rendszer tovább általánosította a rétegelt koncepciót. Rétegek helyett a MULTICS koncentrikus gyűrűkbe szerveződött, a belsők több, a külsők kevesebb privilégiumot kaptak. Ha egy külső gyűrűbeli eljárás egy belső gyűrűbeli eljárást kívánt hívni, egy rendszerhívásnak megfelelő TRAP utasítást kellett végrehajtania. A TRAP paramétereinek a helyességét részletesen ellenőrizték a hívás teljesítésének engedélyezése előtt. Annak ellenére, hogy a MULTICS-ban a teljes operációs rendszer beletartozott minden felhasználói processzus címtartományába, a hardver képes volt minden egyes eljárást (ténylegesen a memóriaszegmenseket) védeni olvasás, írás vagy végrehajtás ellen.

Míg a THE-rendszer rétegelt volta valójában még csak egy tervezési segítség volt, hiszen a rendszer összes alkotóelemét végül is egyetlen végrehajtható programmá szerkesztették össze, addig a MULTICS gyűrűs szerkezete inkább futás közben alakult ki, nagyrészt hardvertámogatással. A gyűrűs szerkezet előnye, hogy könnyen kiterjeszthető a felhasználói alrendszerek strukturálására is. Például a tanár az n. gyűrűben futtatja a hallgatói programokat tesztelő és értékelő programját, míg a hallgatók programjai az $n + 1$. gyűrűben futnak, és nem tudják az érdemjegyeiket megváltoztatni. A Pentium hardvere támogatja a MULTICS gyűrűs szerkezetét, de ezt jelenleg nem használja ki egyetlen fontosabb operációs rendszer sem.

1.5.3. Virtuális gépek

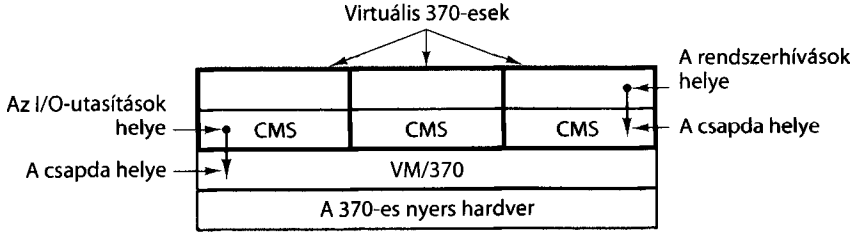
Az OS/360 első változatai szigorúan kötegelt rendszerek voltak. A 360-as sok felhasználója azonban időosztásra vágyott, így az IBM-en belüli és kívüli csoportok is úgy döntöttek, hogy megírják az időosztásos rendszert. A hivatalos IBM időosztásos rendszert (TSS/360) későn hozták ki, amikor pedig elkészült, olyan óriási és olyan lassú volt, hogy csak néhányan kezdték el használni. Abba is hagyták, miután már vagy 50 millió dollárt a fejlesztésére költöttek (Graham, 1970). De az IBM cambridge-i (Massachusetts) Scientific Centerének egyik csoportja előállt egy teljesen más rendszerrel, amelyet az IBM végül is elfogadott hivatalos termékének, és még ma is használnak a működő nagygépein.

Az eredetileg CP/CMS nevű, majd VM/370-re átkeresztelt rendszer (Seawright és MacKinnon, 1979) két jó ötletre épült: az időosztásos rendszerek egyrészt a multiprogramozást, másrészt a csupasz hardvernél sokkal kényelmesebb kapcsolatot adó kiterjesztett gépet biztosítanak. A VM/370 különlegessége, hogy ezt a két funkciót teljesen szétválasztja.

A **virtuális gép monitor** a lelke a rendszernek, amely a nyers hardveren fut, kezeli a multiprogramozást, és nem egy, hanem több virtuális gépet is szolgáltat, amint az az 1.19. ábrán látható. Ellentétben minden más operációs rendszerrel, ezek a virtuális gépek nem kiterjesztett gépek, a szokásos fájlokkal és jó tulajdonságokkal, hanem a hardver *pontos* másolatai, beleértve a felügyelt felhasználói módokat, I/O-t, megszakításokat, szóval a hardvert mindenestől.

Annak következtében, hogy mindegyik virtuális gép azonos a hardvergéppel, bármelyiken futtatható olyan tetszőleges operációs rendszer, amely a hardveren futni képes. A különböző virtuális gépeken különböző operációs rendszerek futtathatók, és igen gyakran futnak is. Az egyiken az OS/360 futhat kötegelt vagy adatfeldolgozási feladatkörrel, miközben a másikon a **CMS (Conversational Monitor System)** egyfelhasználós interaktív rendszer kiszolgálja az időosztásos felhasználókat.

Ha egy CMS-en futó program rendszerhívást hajt végre, akkor ezt a saját virtuális gépén futó operációs rendszer fogja kezelni, és nem a VM/370, mégpedig pontosan úgy, mintha valódi és nem virtuális gépen futna. A CMS-hardver I/O-utasításokat hajt végre lemez olvasására, írására, vagy mást, ami a hívás teljesítéséhez szükséges. A VM/370 ezeket az I/O-utasításokat csapdázza, és a valódi hardver szimulációjaként végrehajtja. A multiprogramozás és a kiterjesztett gép-



1.19. ábra. *A VM/370 és a CMS együttesének szerkezete*

funkciók teljes szétválasztása a rendszer egyes részeit egyszerűbbé, flexibilissé és könnyebben karbantarthatóvá tette.

Ma a virtuális gép fogalmát másra is használják, mégpedig a régi MS-DOS-programok futtatására Pentium processzoron. Amikor a Pentiumot és szoftverét tervezték, az Intel és a Microsoft rájött, hogy kénytelenek lesznek futtatni a régi szoftvereket az új hardveren. Az Intel ezért a Pentiumba beépített egy virtuális 8086 módot. Ebben a módban a gép 8086-ként viselkedik (szoftverszempontból ez megegyezik a 8088 processzorral), beleértve az 1 MB-on belüli 16 bites címzést is.

Ezt a módot használja a Windows és más operációs rendszerek MS-DOS-programok futtatására. A programok 8086 módban indulnak, és a hardveren futnak mindaddig, amíg közönséges utasításokat hajtanak végre. Ha viszont az operációs rendszerrel egy rendszerhívást kívánnak végrehajtatni, vagy felügyelt I/O-t kísérelnek meg közvetlenül, akkor a virtuális gép monitorának csapdájába esnek.

A monitor kétféleképpen reagálhat erre. Ha az MS-DOS maga is be van töltve a 8086 virtuális címtartományba, akkor a csapdázott eseményt egyszerűen továbbküldi az MS-DOS-nak, mintha a csapdázás valódi 8086-on történt volna. Ha majd később az MS-DOS kísérli meg az I/O végrehajtását, a műveletet a virtuális gép monitora elfogja és végrehajtja.

A másik lehetőség az, hogy a monitor az első csapdában elfogja az utasítást, és az I/O-t maga hajtja végre, hiszen pontosan tudja, hogy milyen rendszerhívások vannak az MS-DOS-ban és melyik csapdára mi a teendő. Ez nem olyan szép változat, mint az előbbi, mert csak az MS-DOS-t szimulálja pontosan, más operációs rendszereket nem. Másrészt viszont gyorsabb az előbbinél, mert nem kell az MS-DOS-t is elindítani az I/O kedvéért. Az MS-DOS virtuális 8086 módban való tényleges futtatásának van egy másik hátránya is. Az MS-DOS sokat játszadozik a megszakítások engedélyezésével/tiltásával, amelynek szimulációja költséges.

Megjegyezzük, hogy a fenti módszerek egyike sem azonos a VM/370-nel, ugyanis csak a 8086 és nem a teljes Pentium emulációjáról van szó. A VM/370-rendszeren magát a VM/370-et is lehet futtatni egy virtuális gépen. Mivel a Windows legkorábbi változatai is legalább 286-os processzort igényelnek, így nem futtathatók a virtuális 8086 gépen.

Számos virtuálisgép-megvalósítás vásárolható meg. Webhelyszolgáltatást nyújtó cégek számára gazdaságosabb lehet egy gyors (akár több processzorral rendelkező) szerveren több virtuális gépet futtatni, mint sok kis kapacitású gépet üzemeltetni, amelyek csak egy-egy weboldalt szolgálnak ki. Ennek megvalósításához a VMWare és a Microsoft Virtual PC programja érhető el a piacon. Ezek a programok nagyméretű fájlokat használnak a vendégrendszerek szimulált lemezeiként a gazdarendszeren. A hatékonyság biztosítása érdekében elemzik a vendégrendszer bináris programkódjait, és engedélyezik a biztonságos kódok futását közvetlenül a gazdahardveren, csapdát állítva azoknak az utasításoknak, amelyek operációsrendszer-hívásokat eszközölnek. Ilyen rendszerek az oktatásban is hasznosak. Például a MINIX 3 gyakorlati házi feladatokon dolgozó hallgatók a MINIX 3-at vendég operációs rendszerként használhatják VMWare-t futtató Windows-, Linux- vagy Unix-gazdarendszeren, más, a PC-re telepített program tönkretételének kockázata nélkül. A legtöbb, más tárgyat oktató professzor bizonyára nagyon ideges lenne, ha meg kellene osztania a számítógépes laborokat olyan operációsrendszer-kurzussal, ahol a hallgatók hibái tönkretehetnék vagy törölhetnék a lemez tartalmát.

Egy másik terület, ahol virtuális gépeket használnak, bár kissé más módon, a Java-programok futtatása. Amikor a Sun Microsystems kifejlesztette a Java programozási nyelvet, egy **JVM**-nek **(Java Virtual Machine)** nevezett virtuális gépet (vagyis egy számítógép-architektúrát) is megalkotott. A Java-fordító a JVM számára készít kódot, amelyet jellemzően egy szoftveres JVM-értelmező hajt végre. Ennek a megközelítésnek az előnye az, hogy a JVM-kódot az interneten keresztül tetszőleges gépre eljuttathatjuk, amely rendelkezik JVM-értelmezővel, és ott futtathatjuk. Ha a fordító például SPARC vagy Pentium bináris kódot készítene, akkor a tetszőleges gépre eljuttatás és futtatás nem menne ennyire könnyen. (Természetesen a Sun készíthetett volna olyan fordítót, amely SPARC bináris kódot állít elő, és terjeszthetett volna egy SPARC-értelmezőt, de a JVM egy sokkal egyszerűbben interpretálható rendszer.) A JVM használatának másik előnye az, hogy ha az értelmező megfelelően van megvalósítva, ami persze nem teljesen magától értetődő feladat, akkor a beérkező JVM-programok biztonságossága ellenőrizhető, és védett környezetben futtathatók, így nem tudnak adatot lopni vagy egyéb kárt okozni.

1.5.4. Exokernelek

A VM/370-en minden processzus megkapja a tényleges gép egy pontos másolatát. A Pentium virtuális 8086 módjában a felhasználói processzusok nem a tényleges gép, hanem egy másik pontos másolatát kapják. Az M.I.T. kutatói ennek az ötletnek a továbbfejlesztéseként felépítettek egy rendszert, amelyen a felhasználók a tényleges gép egy változatát kapják az erőforrások egy részével (Engler et al., 1995; és Leschke, 2004). Például az egyik virtuális gép a 0–1023, a másik az 1024–2047 lemezblokkokat kapja, és így tovább.

Az alsó rétegen kernel módban fut az ún. **exokernel** program. A feladata a virtuális gépek számára az erőforrások hozzárendelése, használat közben annak biztosítása, hogy a gépek egymás erőforrásait ne használhassák. A felhasználói virtuális gépeken saját operációs rendszer futhat (mint a VM/370-en vagy a Pentium virtuális 8086 módjában), a korlátozás csupán annyi, hogy csak a kért és a hozzá rendelt erőforrásokat használhatja.

Az exokernel szerkezet egyben meg is takarít egy ún. leképező réteget. Másfajta tervezés esetén a virtuális gépek azt gondolhatják, hogy saját, 0-tól egy maximális blokkszámig terjedő lemezzel rendelkeznek, ezért a virtuálisgép-monitornak kell táblázatok segítségével a blokkcímek (és egyéb erőforrások) leképezéseinek nyilvántartását vezetni. Az exokernel esetében erre a leképezésre nincs szükség, csak azt kell nyilvántartani, hogy mely erőforrásokat mely virtuális gépekhez rendelte. Ez a struktúra előnyösen el is választja a multiprogramozást (ezt az exokernel kezeli) és a felhasználók operációs rendszereinek kódját (ez a felhasználói szinten van), sőt ezt takarékosan valósítja meg, hiszen az exokernel csak a virtuális gépeket védi egymástól.

1.5.5. A kliens-szerver modell

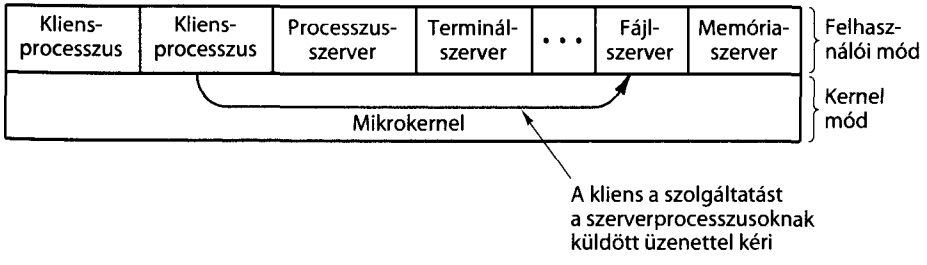
A VM/370 jelentősen egyszerűsödött azzal, hogy a hagyományos operációs rendszerek kódjának többségét áthelyezte egy magasabb rétegre (CMS). A VM/370 ennek ellenére azért elég bonyolult program maradt, hiszen nem olyan egyszerű több virtuális 370-et szimulálni (különösen ha még a hatékonyságot is szem előtt kell tartani).

A korszerű operációs rendszerek ennek az ötletnek a továbbfejlesztését mutatják, azaz egyre több és több programot tolnak magasabb rétegekbe. Ennek köszönhetően az operációs rendszerből végül egy minimális **kernel** marad. A módszer általában az, hogy az operációs rendszer több funkcióját felhasználói processzusokra bízzák. Ha egy szolgáltatást kérünk, például egy fájlblokk olvasását, akkor a felhasználói processzus (**kliensprocesszus**) egy kérést küld a **szerverprocesszus**nak, amely azután elvégzi a munkát és visszaküldi a választ.

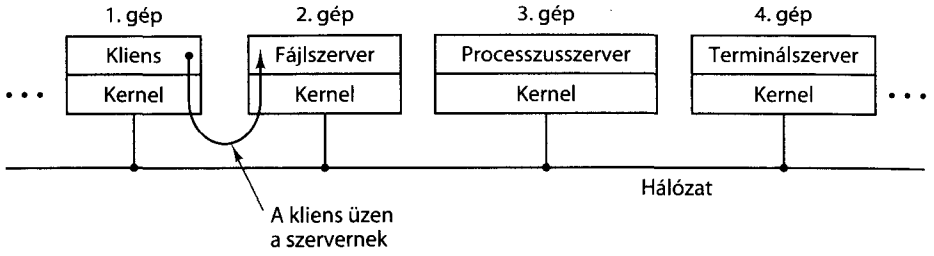
Ebben az 1.20. ábrán is látható modellben a kernelnek csak a kliens és a szerver közötti kommunikációt kell kezelnie. Az operációs rendszer darabokra vágása, ahol egy-egy rész csak a rendszer egy adott szeletével foglalkozik, mint fájl-, processzus-, terminál- vagy memóriagazdálkodás, végső soron az egyes részek leegyszerűsödését és jobb kezelhetőségét eredményezi. A szerverek felhasználói és nem kernel módban futnak, így a hardverhez nincs közvetlen hozzáférésük. Ezért ha a fájlszerver egy programhiba miatt összeomlik is, ez nem feltétlenül okozza a teljes gép leállását.

Osztott rendszerekben is megmutatkoznak a kliens-szerver modell használatának előnyei (1.21. ábra). Ha a kliens egy üzenet küldésével kommunikál a szerverrel, nem kell tudnia, hogy üzenetét lokálisan, a saját gépén fogadják vagy hálózaton keresztül átküldik egy távoli gépen futó szervernek. Mindkét esetben ugyanaz történik, és a kliensnek csak azzal kell törődnie, hogy a kérést elküldje és a választ fogadja.

A kernelről fentebb festett kép, mint csak üzenetközvetítő a kliensek és a szerverek között, nem teljesen realisztikus. Az operációs rendszer néhány funkcióját (például az I/O-eszközregiszterek feltöltését) felhasználói módban nehéz, esetleg lehetetlen végrehajtani. Az ilyen eseteket kétféleképpen kezelhetjük. Az egyik a speciális, kernel módban futó szerverprocesszusok (például I/O-eszközvezérlők) létrehozása, amelyek a teljes hardverhez hozzáférhetnek, de változatlanul az üze-



1.20. ábra. *A kliens-szerver modell*



1.21. ábra. *A kliens-szerver modell osztott rendszerekben*

netváltás módszerével kommunikálnak a többi processzussal. Ennek a mechanizmusnak egy változatát használtuk a MINIX korábbi változataiban, ahol is a meghajtóprogramok a kernel részei voltak, de külön processzusként futottak.

A másik lehetőség, hogy a kernelbe beépítünk egy **minimális kezelőkészlet**et, de ennek **használatá**ról a felhasználói módban futó szerverek gondoskodnak. Például a kernel felismerheti, hogy egy speciális címre küldött üzenet tartalmát valamelyik lemez I/O-eszközregisztereibe kell betölteni. Ebben a példában a kernel nem vizsgálja az üzenet tartalmát, annak érvényességét és jelentését, hanem vakon átmásolja a lemez eszközregisztereibe. (Természetesen emellett szükség van egy másik módszerre is, amely szerint az ilyen üzeneteket küldő processzusok jogosultságát ellenőrizzük.) Így működik a MINIX 3 is, a meghajtóprogramok a felhasználói szinten helyezkednek el, és speciális kernelhívások használatával kérhetik I/O-regiszterek olvasását és írását, valamint így férhetnek hozzá kernelinformációkhoz. Az elv, hogy a kezelőkészletet és annak használatát szétválasztjuk, fontos elv, amelyet az operációs rendszerekben változatos célokra gyakorta használnak.

1.6. Könyvünk további részeinek felépítése

Az operációs rendszerek négy fő részből állnak: processzuskezelés, I/O-eszközkezelés, memóriakezelés és fájlkezelés. A MINIX 3 is erre a négy részre tagolódik. A következő négy fejezet egyenként tárgyalja ezeket a részeket. A 6. fejezet ajánlott irodalmat és bibliográfiát tartalmaz.

A processzusokról, I/O-ról, memória- és fájlkezelésről szóló fejezetek általános felépítése azonos. Először a téma fő elveit mutatjuk be. Azután a MINIX 3-beli megfelelő elemekről adunk áttekintést (ezek a Unixra is érvényesek). Végül részletezzük a MINIX 3-implementációt. Ha az olvasót az elvek érdeklik, és a MINIX 3-megvalósítás nem, akkor az implementációs részeket átfuthatja, vagy ki is hagyhatja. Ha viszont érdekli, hogyan is működik egy valódi operációs rendszer (a MINIX 3), akkor mindent végig kell olvasnia.

1.7. Összefoglalás

Az operációs rendszereket kétféle nézőpontból vizsgálhatjuk: erőforrás-kezelők és kiterjesztett gépek. Mint erőforrás-kezelők, a feladatuk a rendszer különböző részeinek hatékony kezelése. Mint kiterjesztett gépek, a feladatuk a felhasználó számára olyan virtuális gép biztosítása, amelyet a felhasználó a valódi gépnél kényelmesebben tud használni.

Az operációs rendszerek története hosszú időszakot ölel fel; először csak a gépkezelőt helyettesítették, mára eljutottak a korszerű, multiprogramozható rendszerekig.

Minden operációs rendszer lelke a megvalósított rendszerhívások készlete. Ezek határozzák meg az operációs rendszer tényleges tevékenységeit. A MINIX 3 a hívások készletét hat csoportra tagolja. Az első csoport a processzusok létrehozására és megszüntetésére szolgál. A második a szignálkezelésre, a harmadik a fájlok írására és olvasására, a negyedik a könyvtárkezelésre, az ötödik az adatvédelemre, míg a hatodik az időkezelésre alkalmas csoport.

Az operációs rendszerek többféleképpen strukturálhatók. Legtöbbjük a monolitikus, a rétegelt, a virtuális gép, az exokernel, illetve a kliens-szerver modell valamelyikének struktúrájába sorolható.

Feladatok

1. Mi az operációs rendszer két fő feladata?
2. Mi a különbség a kernel mód és a felhasználói mód között? Miért fontos ez a különbség az operációs rendszerek szempontjából?
3. Mi a multiprogramozás?
4. Mi a háttértárolás? Mi a véleménye arról, hogy a nagyobb személyi számítógépekbe beépítik a kimenet átmeneti tárolását?
5. A régebbi számítógépeken minden bejövő és kimenő bájtot közvetlenül a CPU kezelt (nem volt DMA – Direct Memory Access). Milyen befolyással volt ez a multiprogramozásra?
6. A második generációs gépeken miért nem terjedt el az időosztás?
7. A következő utasítások közül melyek hajthatók végre csak kernel módban?
   (a) Megszakítás tiltása.
   (b) Az idő megkérése.
   (c) Az idő beállítása.
   (d) A memórialeképezés módosítása.
8. Soroljon fel a személyi számítógépek operációs rendszerei és a nagyszámítógépek operációs rendszerei közötti néhány különbséget.
9. Indokolja meg, miért lehet jobb minőségű egy zárt forráskódú, szabadalmazott operációs rendszer, amilyen a Windows is, mint egy nyílt forráskódú rendszer, például a Linux. Ezután indolja meg azt, miért lehet jobb minőségű egy nyílt forráskódú operációs rendszer, mint egy zárt forráskódú, szabadalmazott.

10. Egy MINIX-fájlra az *uid* = 12, a *gid* = 1 és a mód *rwxr-x---*. Az *uid* = 1 és *gid* = 1 azonosítókkal rendelkező felhasználó végre akarja hajtani ezt a fájlt. Mi történik?
11. A szuperfelhasználó puszta létezése egész sor védelmi problémát vet fel. Miért van mégis szükség szuperfelhasználóra?
12. A Unix minden változata támogatja a fájlok nevének abszolút elérési útvonallal (a gyökérkönyvtárhoz képesti hely) és a relatív elérési útvonallal (a munkakönyvtárhoz képesti hely) történő megadását. Lehetséges lenne az egyiktől megszabadulni és csak a másikat használni? Amennyiben igen, melyik lehetőséget tartaná meg?
13. Időosztásos rendszereken miért van szükség processzustáblázatra? Szükség van a táblázatra olyan személyi számítógépeken is, ahol egyszerre csak egy, az egész gépet használó processzus létezik?
14. Mi az alapvető különbség a blokkspecifikus és a karakterspecifikus fájlok között?
15. MINIX 3-ban az 1-es felhasználó tulajdonában lévő fájlt a 2-es felhasználó link hívással osztott használatúvá teszi, majd az 1-es felhasználó a fájlt törli. Mi történik, ha a 2-es felhasználó olvasni akarja a fájlt?
16. Nélkülözhetetlen funkcióval rendelkeznek az adatcsövek? Fontos működőképesség veszne el, ha nem lennének elérhetők?
17. Modern fogyasztói berendezések (például zenelejátszók, digitális fényképezőgépek) gyakran rendelkeznek képernyővel, ahol parancsokat vihetünk be, és ahol ezen parancsok bevitelének eredménye megjelenik. Ezek a berendezések belül gyakran rendelkeznek nagyon egyszerű operációs rendszerrel. A PC-k szoftverének mely részéhez hasonlítható ez a parancsbeviteli mód?
18. A Windows nem rendelkezik fork rendszerhívással, mégis képes új processzusok létrehozására. Próbálja kitalálni a szemantikáját annak a rendszerhívásnak, amelyet a Windows használ erre a célra.
19. Miért csak a szuperfelhasználó hajthatja végre a chroot hívást? (Emlékezzünk a védelmi problémákra.)
20. Vizsgáljuk meg a rendszerhívások listáját az 1.9. ábrán. Mit gondol, valószínűleg melyik hívás hajtódik végre leggyakrabban? Indokolja válaszát.
21. Tegyük fel, hogy egy számítógép másodpercenként 1 milliárd művelet végrehajtására képes, és hogy egy rendszerhíváshoz 1000 utasítás szükséges, beleértve a csapda és a kontextusváltás végrehajtását is. A számítógép másodpercenként hány rendszerhívást hajthat végre úgy, hogy a CPU kapacitásának a fele megmaradjon a felhasználói kód futtatására?
22. Az 1.16. ábrán láthatunk mknod rendszerhívást, de nincs rmnod. Jelenti-e ez azt, hogy nagyon-nagyon figyelmesen hozhatunk csak létre új csomókat ezzel a módszerrel, mert nincs lehetőségünk törölni őket?
23. Miért futtatja a MINIX 3 az *update* háttérprogramot állandóan?
24. Van értelme a SIGALRM szignál figyelmen kívül hagyásának?
25. Osztott rendszereken a kliens-szerver modell népszerű. Használható ez egygépes rendszeren is?

26. A Pentium kezdeti változatai nem támogatták a virtuális gép monitort. Milyen lényeges karakterisztika szükséges egy gép virtualizálásához?
27. Írjunk programot vagy programokat az összes MINIX 3-rendszerhívás tesztelésére. Használjuk a hívásokat különböző, esetleg hibás paraméterekkel és figyeljük meg, hogy a hibák kiderülnek-e.
28. Írjunk az 1.10. ábrán láthatóhoz hasonló parancsértelmezőt, amely már működőképes és tesztelésre is alkalmas. Bővíthetjük a bemenet és a kimenet átirányításával, adatcsövekkel, háttérfeladatok végrehajtásával és egyéb funkciókkal.

2. Processzusok

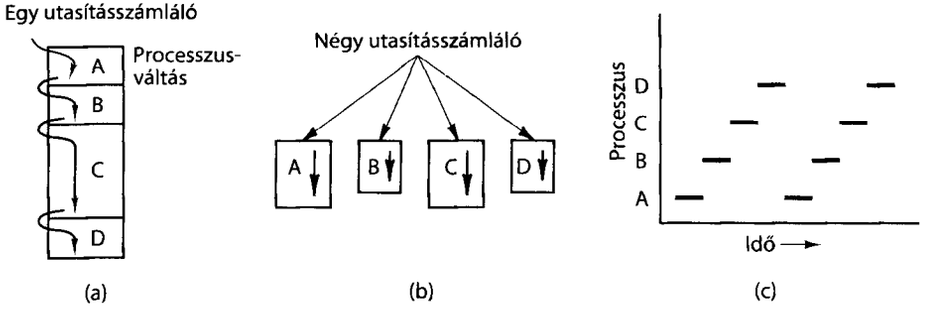
Ebben a fejezetben elkezdjük az operációs rendszerek tervezésének és megvalósításának részletes tanulmányozását mind általánosságban, mind pedig konkrétan a MINIX 3 esetében. Minden operációs rendszer központi fogalma a *processzus:* a futó program egy absztrakciója. Minden más ettől a fogalomtól függ, ezért fontos, hogy az operációs rendszer tervezője (és a hallgató) jól megértse, mit takar.

2.1. Bevezetés

Minden modern számítógép több dolgot képes egy időben elvégezni. Mialatt egy felhasználói program fut, a számítógép olvashat is egy lemezről, és szöveget is írhat a képernyőre vagy nyomtatóra. Egy multiprogramozható rendszerben a CPU is programról programra kapcsol, futtatva azokat néhány tíz vagy száz ezred másodpercig. Bár szigorúan véve a CPU minden időpillanatban csak egy programot futtat, egy másodperc leforgása alatt több programon is dolgozhat, és ezzel a párhuzamosság illúzióját kelti a felhasználóban. Néha **látszatpárhuzamosság**ról beszélünk ebben az összefüggésben, megkülönböztetve a **többprocesszoros** rendszerek (két vagy több CPU megosztva használja ugyanazt a fizikai memóriát) valódi hardverpárhuzamosságától. Az embernek nehéz követnie ezeket a többszörös, párhuzamos tevékenységeket. Ezért az operációs rendszerek tervezői az évek során egy olyan fogalmi modellt (szekvenciális processzusok) fejlesztettek ki, amely megkönnyíti a párhuzamosság kezelését. Ennck a modellnek a használata és néhány következménye lesz e fejezet tárgya.

2.1.1. A processzusmodell

Ebben a modellben minden, a számítógépen futtatható szoftver, gyakran beleértve magát az operációs rendszert is, **szekvenciális processzusok** vagy röviden **processzusok** sorozatává szerveződik. A processzus egyszerűen egy végrehajtás alatt álló program, beleértve az utasításszámláló, a regiszterek és a változók aktuális ér-



2.1. ábra. *(a) Négy program multiprogramozása. (b) Négy független, szekvenciális processzus elméleti modellje. (c) Minden időpillanatban csak egy program aktív*

tékét is. Elméletileg minden processzusnak saját virtuális CPU-ja van. A valóságban természetesen a valódi CPU kapcsolgat oda-vissza a processzusok között, de ahhoz, hogy a rendszert megértsük, könnyebb az (ál-)párhuzamosan futó processzusok együttesét elképzelni, mint megpróbálni követni, hogyan kapcsolgat a CPU programról programra. Ezt a gyors oda-vissza kapcsolást **multiprogramozás**nak nevezzük, ahogy azt már az 1. fejezetben láttuk.

A 2.1.(a) ábrán egy számítógépet látunk, amely multiprogramozást kezel négy programmal a memóriájában. A 2.1.(b) ábrán négy processzust látunk, mindegyiknek saját vezérlése (vagyis saját utasításszámlálója) van, és mindegyik a többitől függetlenül fut. Természetesen csak egyetlen fizikai utasításszámláló van, ezért az egyes processzusok futásakor annak logikai utasításszámlálója betöltődik az igazi utasításszámlálóba. Amikor futása egy időre befejeződik, a fizikai utasításszámláló elmentődik a processzus logikai utasításszámlálójába, amely a memóriában található. A 2.1.(c) ábrán látható, hogy elegendően hosszú időintervallumot tekintve minden processzus végrehajtódik, de minden adott időpillanatban aktuálisan csak egy processzus fut.

Azzal, hogy a CPU oda-vissza kapcsolgat a processzusok között, egy processzus nem állandó sebességgel halad előre a számításai végrehajtásában, és ez valószínűleg nem is reprodukálható, még akkor sem, ha ugyanazokat a processzusokat még egyszer futtatjuk. Ezért a processzusokba nem szabad belső időzítési feltételeket beépíteni. Tekintsünk például egy I/O-processzust, amely elindít egy szalagot, hogy kimentett fájlokat töltsön vissza, és végrehajt 10 000-szer egy üres ciklust, hogy a szalag felgyorsulhasson, majd kiad egy utasítást az első rekord beolvasására. Ha a CPU úgy dönt, hogy a várakozó ciklus alatt egy másik processzusra kapcsol át, lehet, hogy a szalagprocesszus csak azután fog újból futni, amikor az első rekord már elhaladt az olvasófej előtt. Amikor a processzusnak ehhez hasonló kritikus valós idejű követelményei vannak, azaz bizonyos eseményeknek be *kell* következni előre meghatározott néhány ezred másodpercen belül, speciális intézkedésekre van szükség, hogy biztosítsuk azok tényleges bekövetkezését. Rendszerint azonban a legtöbb processzust nem befolyásolja a CPU alapvető multiprogramozása vagy a különböző processzusok relatív sebessége.

A különbség egy processzus és egy program között hajszálnyi, de döntő. Egy analógia segíthet ezt világosabbá tenni. Tekintsünk egy konyhaművész számítógéptudóst, aki a lányának születésnapi tortát süt. Megvan a születésnapi torta receptje, és a szükséges nyersanyagok is rendelkezésre állnak a konyhában: van liszt, tojás, cukor, vaníliakivonat stb. Ebben az analógiában a recept a program (vagyis egy algoritmus, amelyet ideillő jelölésrendszerben írtak le), a számítógéptudós a processzor (CPU), és a sütemény hozzávalói a bemeneti adatok. A processzus a következő tevékenységekből áll: cukrászunk olvassa a receptet, veszi a hozzávalókat és süti a tortát.

Most képzeljük el, hogy a számítógéptudós fia sírva beszalad azzal, hogy megcsípte egy méhecske. A számítógéptudós megjegyzi, hol tart a receptben (az aktuális processzus állapotát elmenti), előveszi az elsősegélynyújtó könyvet, és elkezdi követni annak utasításait. Itt látjuk, hogy a processzort átkapcsolták az egyik processzusról (sütés) egy magasabb prioritású processzusra (elsősegélynyújtás), amelyek mindegyike külön programmal rendelkezik (recept, illetve elsősegélynyújtó könyv). A méhcsípés ellátása után a számítógéptudós visszatér a süteményéhez, ott folytatva, ahol abbahagyta.

Egy processzus, és ez itt a lényeg, egy bizonyosfajta tevékenység. Van programja, bemenő és kimenő adatai és állapota. Egyetlen processzort bizonyos ütemezési algoritmussal megoszthatunk több processzus között, amelyet arra használunk, hogy meghatározzuk, mikor fejezzük be a munkát az egyik processzuson és szolgáljunk ki egy másikat.

2.1.2. Processzusok létrehozása

Az operációs rendszernek valamilyen módon gondoskodnia kell az összes szükséges processzus létrehozásáról. Nagyon egyszerű rendszerekben, illetve olyan rendszerekben, amelyeket csak egy processzus futtatására terveztek (például egy eszköz valós idejű vezérlésére), megoldható, hogy minden processzus, amelyre valaha szükség lehet, a rendszer indulásakor elérhető legyen. Általános célú rendszerekben azonban szükség van processzusok létrehozására és megszüntetésére működés közben is. Most megvizsgáljuk a felmerülő kérdéseket.

Négy fő esemény okozhatja processzusok létrehozását:

1. A rendszer inicializálása.
2. A processzus által meghívott processzust létrehozó rendszerhívás végrehajtása.
3. A felhasználó egy processzus létrehozását kéri.
4. Kötegelt feladat kezdeményezése.

Egy operációs rendszer indulásakor gyakran számos processzus keletkezik. Sok közülük előtérben fut; ezek azok a processzusok, amelyek a felhasználókkal tartják a kapcsolatot, vagy számukra munkát végeznek. Mások háttérprocesszusok, amelyek nincsenek egy bizonyos felhasználóhoz hozzárendelve, ehelyett valami-

lyen sajátos feladatuk van. Például egy háttérprocesszust tervezhetünk a gépen tárolt weboldalak elérésére vonatkozó kérések fogadására, amely akkor aktivizálódik, amikor ilyen kiszolgálandó kérés érkezik. Azokat a processzusokat, amelyek a háttérben maradnak valamilyen tevékenység kezelésére (weboldalak, nyomtatás), **démonok**nak (**daemon**) hívjuk. Nagy rendszerek rendszerint tucatnyi ilyennel rendelkeznek. A MINIX 3-ban a *ps* programot használhatjuk a futó processzusok kilistázására.

A rendszerinduláskori processzuslétrehozás mellett később is létrehozhatók processzusok. Gyakran egy futó processzus ad ki rendszerhívást egy vagy több új processzus létrehozására, hogy munkáját segítsék. Új processzus létrehozása különösen hasznos, ha az elvégzendő munka könnyen megfogalmazható számos egymáshoz kapcsolódó, egymással együttműködő, de egyébként egymástól független processzussal. Például egy nagyméretű program fordításakor a *make* program meghívja a C fordítót a forráskód tárgykóddá alakításához, majd az *install* programot a program rendeltetési helyére másolásához, tulajdonosi információk, hozzáférési jogok és egyebek beállításához. A MINIX 3-ban a C fordító tulajdonképpen számos különböző program összessége, amelyek együtt dolgoznak. Ezek közé tartozik az előfeldolgozó, a C nyelvi elemző, az assembly kódgenerátor, az assembler és a tárgykódszerkesztő (linker).

Interaktív rendszerekben a felhasználók parancsok begépelésével indíthatnak programokat. A MINIX 3-ban virtuális konzolok segítségével a felhasználó programot, például egy fordítót indíthat, majd átválthat egy másik konzolra, ahol mondjuk indíthat egy szövegszerkesztőt a dokumentáció megírására, mialatt a fordító dolgozik.

Az utolsó olyan helyzet, ahol processzusok keletkezhetnek, csak a nagyszámítógépeken futó kötegelt rendszerekre érvényes. Itt a felhasználók kötegelt feladatokat küldhetnek a rendszernek (valószínűleg távolról). Amikor az operációs rendszer úgy dönt, hogy van elegendő rendelkezésre álló erőforrás egy új feladat futtatásához, készít egy új processzust, és ebben futtatja a bemeneti sorban található következő feladatot.

Technikai értelemben ezen esetek mindegyikében egy már létező processzus processzuslétrehozó rendszerhívás segítségével hoz létre új processzust. A kérdéses processzus lehet egy futó felhasználói processzus, egy billentyűzettel vagy egérrel elindított rendszerprocesszus, vagy egy kötegelt feladatkezelő processzus. Ez a processzus hajtja végre a rendszerhívást az új processzus létrehozásához. Ez a rendszerhívás mondja meg az operációs rendszernek, hogy egy új processzust hozzon létre, és közvetve vagy közvetlenül jelzi, hogy melyik programot kell ebben futtatnia.

A MINIX 3 egyetlen processzuslétrehozó rendszerhívással rendelkezik, a fork-kal. Ez a hívás az őt meghívó processzus tökéletes másolatát készíti el. A fork után a két processzus, a szülő és a gyermek ugyanazzal a memóriaképpel, környezeti sztringekkel és megnyitott fájlokkal fognak rendelkezni. Ennyi az egész. A gyermekprocesszus ezek után rendszerint végrehajt egy execve vagy hasonló rendszerhívást a memóriakép megváltoztatására és egy új program futtatására. Például amikor a felhasználó begépeli mondjuk a *sort* parancsot a parancsértelmezőben,

az elindít egy gyermekprocesszust, és a gyermek hajtja végre a *sort*-ot. Ennek a kétlépéses megoldásnak az oka az, hogy a gyermek képes legyen a fájlleírók manipulálására a fork után, de még az execve előtt a standard bemeneti, kimeneti és hibacsatornák átirányításához.

A MINIX 3-ban és Unixban egyaránt a processzus létrehozása után a szülő és a gyermek saját elkülönülő címtartománnyal rendelkezik. Ha bármelyikük megváltoztat egy szót a címtartományán belül, az a másik processzus számára nem lesz látható. A gyermek kezdeti címtartománya a szülőének másolata, de a két címtartomány elkülönül, írható memóriaterületen nem osztoznak (akárcsak néhány más Unix-megvalósítás, a MINIX 3 is képes a nem módosítható programkódrészt megosztani közöttük). Azonban az újonnan létrehozott processzus megteheti azt, hogy osztozik a létrehozójának néhány más erőforrásán, például a megnyitott fájlokon.

2.1.3. Processzusok befejezése

A processzus létrehozása után elkezd dolgozni, és teszi a dolgát. Azonban semmi sem tart örökké, még a processzusok sem. Előbb vagy utóbb az új processzus befejeződik, rendszerint a következő körülmények között:

1. Szabályos kilépés (önkéntes).
2. Kilépés hiba miatt (önkéntes).
3. Kilépés végzetes hiba miatt (önkéntelen).
4. Egy másik processzus megsemmisíti (önkéntelen).

A legtöbb processzus azért fejeződik be, mert végzett a feladatával. Amikor a fordítóprogram lefordította a neki adott programot, akkor végrehajt egy rendszerhívást, amivel közli az operációs rendszer felé, hogy elkészült. Ez az exit hívás a MINIX 3-ban. A képernyő-orientált programok is támogatják az önkéntes befejezést. Például a szövegszerkesztők mindig rendelkeznek olyan billentyűkombinációval, amellyel a felhasználó közölheti a processzussal, hogy mentse a munkafájlt, távolítsa el a megnyitott ideiglenes állományokat, és fejezze be a futását.

A befejezés második indoka az lehet, hogy a processzus végzetes hibát fedezett fel. Például ha a felhasználó begépeli a

```
cc foo.c
```

parancsot a *foo.c* program fordításához, és nem létezik ilyen nevű fájl, a fordítóprogram egyszerűen kilép.

A befejezés harmadik oka a processzus által okozott hiba, esetleg egy hibás programsor miatt. Erre példa többek között egy illegális utasítás végrehajtása, nem létező memóriacímre hivatkozás, vagy a nullával való osztás. A MINIX 3-ban a processzus az operációs rendszer tudtára hozhatja, hogy bizonyos hibákat maga kíván kezelni, amely esetben a processzus egy szignált kap (megszakítódik), ahelyett hogy befejeződne a hiba bekövetkeztekor.

A processzusbefejezés negyedik oka az, hogy egy processzus végrehajt egy olyan rendszerhívást, amely azt közli az operációs rendszerrel, hogy semmisítsen meg egy másik processzust. A MINIX 3-ban ez a kill hívás. Természetesen a megsemmisítőnek rendelkeznie kell a szükséges jogosultsággal a kérés végrehajtásához. Bizonyos rendszerekben egy processzus akár önkéntes, akár önkéntelen befejeződése maga után vonja az általa létrehozott valamennyi processzus azonnali megsemmisítését is. A MINIX 3 azonban nem így működik.

2.1.4. Processzushierarchiák

Bizonyos rendszerekben, amikor egy processzus egy másikat hoz létre, a szülő és a gyermek bizonyos értelemben kapcsolatban maradnak. A gyermek maga is több processzust hozhat létre, így egy processzushierarchiát kialakítva. A processzusok csak egy szülővel rendelkeznek (de lehet nulla, egy, kettő vagy több gyermekük).

A MINIX 3-ban egy processzus, a gyermekei és további leszármazottjai együttesen egy processzuscsoportot alkotnak. Amikor a felhasználó a billentyűzetről egy szignált küld, ez a szignál a billentyűzethez rendelt processzuscsoport összes tagja számára kézbesítődhet (rendszerint azoknak a processzusoknak, amelyek az aktuális ablakban jöttek létre). Ez a viselkedés szignálfüggő. Amennyiben a szignál egy csoportnak lett küldve, minden processzus elkaphatja, figyelmen kívül hagyhatja, vagy az alapértelmezett módon cselekedhet, ami a szignál általi befejezését jelenti.

A processzusfák használatának egyszerű példájaként nézzük meg, hogyan inicializálja magát a MINIX 3. Két speciális processzus, a **reinkarnációs szerver** és az **init**, megtalálható az indító memóriatartalomban (boot image). A reinkarnációs szerver feladata a meghajtóprogramok és a kiszolgálók (újra)indítása. Működését blokkolt állapotban kezdi, üzenetre várva, hogy mit hozzon létre.

Ezzel szemben az *init* végrehajtja az */etc/rc* szkriptet, aminek következtében arra utasítja a reinkarnációs szervert, hogy a kezdeti memóriatartalomban nem szereplő meghajtóprogramokat és kiszolgálókat indítsa el. Ez az eljárás a meghajtóprogramokat és a kiszolgálókat a reinkarnációs szerver gyermekeiként indítja el, így amennyiben bármelyikük befejeződik, a reinkarnációs szerver értesül erről és újraindíthatja (reinkarnálhatja). Ezzel a mechanizmussal képes a MINIX 3 lekezelni egy meghajtóprogram vagy kiszolgáló összeomlását, mivel egy másikat tud indítani automatikusan helyette. A gyakorlatban egy meghajtóprogram cseréje azonban jóval egyszerűbb, mint egy kiszolgálóé, mivel jóval kevesebb utóhatása van a rendszer más részeire nézve. (És nem mondjuk, hogy mindig tökéletesen működik; ez még egy folyamatban lévő munka.)

Amikor az *init* végzett ezzel, beolvassa az */etc/ttytab* konfigurációs fájlt, amely megadja, hogy mely terminálok és virtuális terminálok léteznek. Az *init* elindít a fork segítségével egy *getty* processzust mindegyikük számára, megjeleníti a bejelentkezési promptot, és vár a bemenetre. Amikor egy név begépelésre kerül, a *getty* az exec segítségével végrehajt egy *login* processzust a megadott névvel argumentumként. Amennyiben a felhasználó bejelentkezése sikeres, a *login* futtatja a

felhasználó parancsértelmezőjét (shell). Így a shell az *init* gyermekprocesszusa. A felhasználó parancsai a shell gyermekei és az *init* unokái lesznek. Az események ezen sorrendje példája a processzusfák működésének. A reinkarnációs szerver és az *init* kódja nincs a könyvünkben kilistázva, mint ahogy a parancsértelmezőé sem: valahol meg kell húzni a határt. Az alapötlet így is világos.

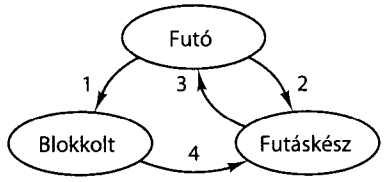
2.1.5. Processzusállapotok

Bár minden processzus egy önálló egység saját utasításszámlálóval, veremmel, nyitott fájlokkal, ébresztőkkel és egyéb belső állapottal, a processzusoknak gyakran szükségük van interakcióra, kommunikációra, szinkronizációra más processzusokkal. Egy processzus generálhat például olyan kimenetet, amelyet egy másik processzus bemenetként használ. A

```
cat chapter1 chapter2 chapter3 | grep tree
```

parancsértelmezőnek szóló utasításban az első processzus, a *cat* futtatása, három fájlt kapcsol össze. A második processzus, a *grep* futtatása, kiválaszt minden olyan sort, amely tartalmazza a „tree" szót. A két processzus relatív sebességétől függően (amely függ mind a programok relatív bonyolultságától, mind attól, hogy egy-egy processzus mennyi CPU-időt kapott) előfordulhat, hogy a *grep* futásra készen áll, de nincs feldolgozásra váró bemenet. Ekkor **blokkolódnia** kell, amíg nem áll rendelkezésre bemenő adat.

Egy processzus blokkolódik, ha logikailag nem lehet folytatni rendszerint azért, mert olyan bemenetre vár, amely még nem elérhető. Az is lehetséges, hogy egy processzust, amely elvileg kész és futásra képes, az operációs rendszer leállít, hogy a CPU-t egy kis időre egy másik processzusnak adja. Ez a két feltétel teljesen különböző. Az első esetben a felfüggesztés a probléma velejárója (nem tudjuk feldolgozni a felhasználói parancssort, amíg be nem gépelték). A második esetben ez a rendszer sajátossága (nincs elegendő CPU, hogy minden processzusnak saját külön processzora legyen). A 2.2. ábrán egy állapotdiagramot látunk, amely bemutatja azt a három állapotot, amelyben egy processzus lehet.



1. A processzus bemeneti adatra várva blokkol
2. Az ütemező másik processzust szemelt ki
3. Az ütemező ezt a processzust szemelte ki
4. A bemeneti adat elérhető

2.2. ábra. *A processzus lehet futó, blokkolt vagy futáskész állapotban. Láthatjuk az állapotok közötti átmeneteket*

1. Futó (az adott pillanatban éppen használja a CPU-t).
2. Futáskész (készen áll a futásra; ideiglenesen leállították, hogy egy másik processzus futhasson).
3. Blokkolt (bizonyos külső esemény bekövetkezéséig nem képes futni).

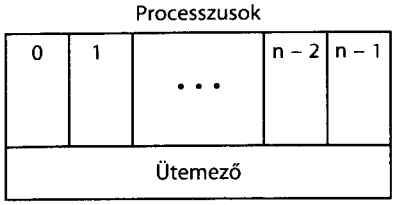
Az első két állapot logikailag hasonló. A processzus mindkét esetben futni akar, csak a második esetben ideiglenesen nincs számára elérhető CPU. A harmadik állapot az első kettőtől abban különbözik, hogy itt a processzus nem képes futni még akkor se, ha a CPU-nak nincs más dolga.

Mint a 2.2. ábrán látható, négy átmenet lehetséges a három állapot között. Az 1. átmenet akkor következik be, amikor egy processzus ráébred arra, hogy futását nem tudja folytatni. Néhány rendszerben ekkor a processzusnak egy rendszerhívást, a block-ot vagy a pause-t kell végrehajtania, hogy blokkolt állapotba kerüljön. Más rendszerekben, beleértve a MINIX 3-at is, amikor egy processzus adatcsőből vagy speciális fájlból (például terminálról) olvas, és nincs elérhető bemenet, a processzus automatikusan átkerül futó állapotból blokkoltba.

A 2. és 3. átmenetet a processzusütemező, mint az operációs rendszer része, váltja ki anélkül, hogy a processzus bármit is tudna erről. A 2. átmenet akkor következik be, amikor az ütemező úgy dönt, hogy a futó processzus már elég régóta fut, és itt az ideje, hogy egy másik processzus kapjon valamennyi CPU-időt. A 3. átmenet akkor fordul elő, amikor már minden más processzus megkapta a jogos részét, és itt az ideje, hogy az első processzus jusson a CPU-hoz, hogy ismét futhasson. Az ütemezés tárgya, vagyis annak eldöntése, hogy melyik processzus fusson, mikor és mennyi ideig, nagyon fontos dolog. Sok algoritmust találtak ki, hogy megpróbálják kiegyensúlyozni a két egymással versengő követelményt: a rendszernek mint egésznek a hatékonyságát és az egyes processzusok pártatlan kezelését. Magát az ütemezést és néhány ilyen algoritmust a fejezet későbbi részében tárgyaljuk.

A 4. átmenet akkor fordul elő, amikor az a külső esemény, amelyre a processzus várakozott, bekövetkezik (például megérkezik a bemenő adat). Ha egy processzus sem fut ebben a pillanatban, azonnal kiváltódik a 3. átmenet, és a processzus elkezd futni. Különben lehet, hogy várakoznia kell egy kicsit a *futáskész* állapotban, amíg a CPU elérhető lesz.

A processzusmodellt használva könnyebben el tudjuk képzelni, hogy mi történik a rendszer belsejében. Egyes processzusok olyan programokat futtatnak, amelyek a felhasználó által begépelt parancsokat hajtják végre. Más processzusok a rendszer részei, és olyan feladatokat látnak el, mint a fájlkiszolgáló felé érkező igények teljesítése vagy egy lemez vagy szalagegység résztevékenységeinek összehangolása. Amikor bekövetkezik egy lemezmegszakítás, a rendszer döntést hozhat az aktuális processzus futásának megállításáról és annak a lemezprocesszusnak a futtatásáról, amely eddig blokkolva volt, mert erre a megszakításra várt. Azért használtuk a feltételes módot, mert ez a futó processzus és a lemezmeghajtó processzus egymáshoz képesti prioritásaitól függ. De a lényeg az, hogy anélkül, hogy a megszakításokkal foglalkoznánk, úgy tekinthetünk a felhasználói processzusokra, a lemezprocesszusokra, a terminálprocesszusokra stb., mint amelyek



2.3. ábra. *A processzuselvű operációs rendszer alsó szintje kezeli a megszakításokat és az ütemezést. A felette lévő szinten vannak a szekvenciális processzusok*

blokkolódnak, amikor valaminek a bekövetkezésére várnak. Amikor a lemezblokkot beolvastuk vagy a karaktert leütöttük, az erre váró processzus blokkolása feloldódik, és ezzel készen áll, hogy újra fusson.

Ez a szemlélet eredményezi a 2.3. ábrán látható modellt. Itt az operációs rendszer legalsó szintje az ütemező, és felette helyezkedik el a többi processzus. Mind a megszakításkezelés, mind a processzusok tényleges indításának és megállításának részletei az ütemezőben vannak elrejtve, amely tulajdonképpen elég kicsi. Az operációs rendszer többi része könnyen beilleszthető a processzusmodellbe. A 2.3. ábra modelljét használja a MINIX 3. Természetesen az ütemezés nem az egyetlen dolog a legalsó szinten, a megszakításkezelés és a processzusok közötti kommunikáció támogatása is itt található. Mindazonáltal első közelítésként jól mutatja az alapvető szerkezetet.

2.1.6. Processzusok megvalósítása

A processzusmodell megvalósításához az operációs rendszer egy táblázatot (adatszerkezetek tömbjét) kezel, amelyet **processzustáblázat**nak nevezünk, processzusonként egy bejegyzéssel. (Néhány szerző ezeket a bejegyzéseket **processzusvezérlő blokk**oknak nevezi.) Ez a bejegyzés információt tartalmaz a processzus állapotáról, utasításszámlálójáról, veremmutatójáról, a lefoglalt memóriáról, a megnyitott fájljainak állapotáról, az elszámolási és ütemezési információjáról, ébresztőiről és egyéb szignáljairól, valamint minden egyéb, a processzusra vonatkozó olyan információról, amit azért kell elmenteni, amikor a processzust *futóról futáskész* állapotba kapcsoljuk, hogy később úgy indulhasson újra a processzus, mintha soha nem állítottuk volna le.

A MINIX 3-ban a processzusok közötti kommunikációt, a memóriakezelést és a fájlkezelést a rendszeren belül különálló modulok valósítják meg, így a processzustáblázat részekre van osztva, hogy minden modul karbantarthassa a számára szükséges mezőket. A 2.4. ábra a fontosabb mezők közül mutat be néhányat. Ehhez a fejezethez kizárólag az első oszlop mezői tartoznak. A másik két oszlop csak arra szolgál, hogy lássuk, a rendszer más részein milyen információkra van szükség.

Kernel	Processzuskezelés	Fájlkezelés
Regiszterek	Mutató a kódszegmensre	UMASK maszk
Utasításszámláló	Mutató az adatszegmensre	Gyökérkönyvtár
Programállapot szó	Mutató a bss szegmensre	Munkakönyvtár
Veremmutató	Kilépés állapota	Fájlleírók
Processzusállapot	Szignál állapota	Valós UID
Aktuális ütemezési prioritás	Processzusazonosító	Tényleges UID
Maximális ütemezési prioritás	Szülőprocesszus	Valós GID
Hátralévő ütemezett idő	Processzuscsoport	Tényleges GID
Időzítési egység mérete	Gyermekek CPU-ideje	Vezérlő tty
Felhasznált CPU-idő	Valódi UID	Mentési terület olvasáshoz/
Mutató az üzenetsorra	Tényleges UID	íráshoz
Függő szignálbitek	Valódi GID	Rendszerhívás paraméterei
Különböző jelzőbitek	Tényleges GID	Különböző jelzőbitek
Processzus neve	Fájlinformáció	
	megosztáshoz	
	Bittérkép a szignálokhoz	
	Különböző jelzőbitek	
	Processzus neve	

2.4. ábra. *A MINIX 3-processzustáblázat néhány mezője. A mezők a kernel, a processzuskezelés és a fájlrendszer szerint vannak felosztva*

Most, hogy megnéztük a processzustáblázatot, nézzük meg egy kicsit jobban, hogyan tartják fenn a többszörös szekvenciális processzusok illúzióját egy olyan gépen, amelynek egy CPU-ja és több I/O-eszköze van. Szigorúan véve most annak a leírása következik, hogyan dolgozik a MINIX 3-ban a 2.3. ábra „ütemezője", de a legtöbb modern operációs rendszer alapvetően ugyanígy működik. Minden egyes I/O-eszközosztályhoz (vagyis hajlékonylemez-egységekhez, merevlemezegységekhez, időmérőkhöz, terminálokhoz) tartozik egy tábla, amelyet **megszakításleíró táblának** nevezünk. A tábla egyes bejegyzéseinek legfontosabb része a **megszakításvektor**, amely a megszakítást kiszolgáló eljárás címét tartalmazza. Tegyük fel, hogy éppen a 23. felhasználói processzus fut, amikor egy lemezmegszakítás bekövetkezik. Az utasításszámlálót, a programállapot szót és esetleg egy vagy több regisztert a megszakításhardver az (aktuális) verembe teszi. A számítógép ezután a lemezmegszakítás-vektorban meghatározott címre ugrik. Ez minden, amit a hardver csinál. Innen kezdve a szoftveren a sor.

A megszakítást kiszolgáló eljárás azzal kezdi, hogy az összes regisztert elmenti az aktuális processzushoz tartozó processzustáblázat-bejegyzésbe. Az aktuális processzus száma és a bejegyzésére mutató pointer globális változóban marad, hogy gyorsan megtalálhassuk. Ezután a megszakítás által lerakott információk kikerülnek a veremből, és a veremmutató egy ideiglenes veremre állítódik, amelyet a processzuskezelő használ. Az olyan tevékenységek, mint a regiszterek elmentése vagy a veremmutató beállítása, nem fejezhetők ki magas szintű programozási nyelvekben, amilyen például a C, ezért ezeket kis assembly nyelvű rutinokkal valósítják meg. Amikor ez a rutin befejeződik, meghív egy C eljárást az adott megszakítástípushoz tartozó munka hátralévő részének elvégzésére.

1. A hardver verembe teszi az utasításszámlálót stb.
2. A hardver a megszakításvektorból betölti az új utasításszámlálót.
3. Az assembly nyelvű eljárás elmenti a regisztereket.
4. Az assembly nyelvű eljárás beállítja az új vermet.
5. A C megszakításkezelő üzenetet készít és küld.
6. Az ütemező futáskésznek jelöli a várakozó feladatot.
7. Az üzenetküldő kód az üzenetre várakozót futáskészre állítja.
8. A C eljárás visszatér az assembly kódba.
9. Az assembly nyelvű eljárás elindítja az új aktuális processzust.

2.5. ábra. *Vázlat az operációs rendszer legalsó szintjének tevékenységéről, amikor egy megszakítás bekövetkezik*

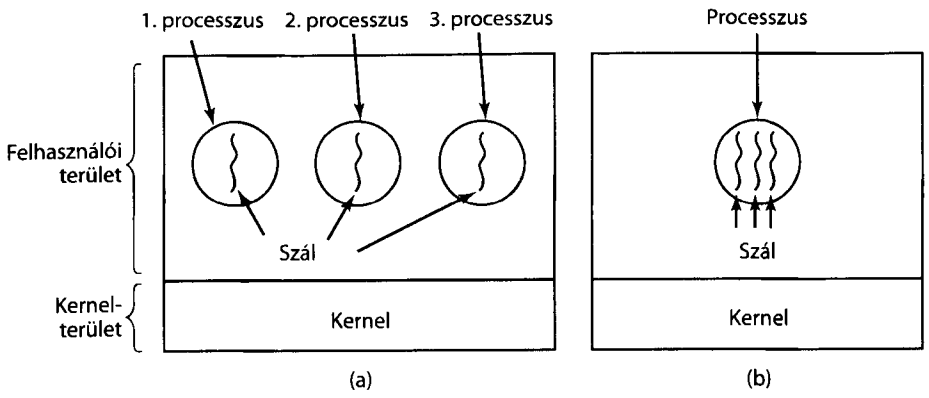
A processzusok kommunikációját a MINIX 3-ban üzenetekkel valósítjuk meg, vagyis a következő lépés, hogy összeállítsunk egy üzenetet, amelyet annak a lemezprocesszusnak küldünk, amelyik blokkolt állapotban erre vár. Az üzenet azt mondja, hogy megszakítás következett be, megkülönböztetve ezzel attól az üzenettől, amely felhasználói processzusoktól származik, és egy lemezblokk olvasását vagy valami hasonlót kér. A lemezprocesszus állapota *blokkoltról futáskészre* változik, és meghívásra kerül az ütemező. A MINIX 3-ban a különböző processzusoknak különböző prioritása van, hogy jobban ki tudjuk szolgálni például az I/O-eszközkezelőket, mint a felhasználói processzusokat. Ha most a lemezprocesszus a legmagasabb prioritású futtatható processzus, akkor az ütemező futásra kijelöli. Ha az a processzus, amelyet megszakítottunk, ugyanilyen fontos, vagy fontosabb, akkor az ütemező ismét azt választja ki futásra, és a lemezprocesszusnak kis ideig várnia kell.

Így vagy úgy az assembly nyelvű megszakítási kód által hívott C eljárás most visszatér, és az assembly nyelvű kód feltölti a regisztereket és a memóriatérképet a most aktuális processzus számára, majd elindítja futását. A 2.5. ábrán összefoglaljuk a megszakításkezelést és az ütemezést. Érdemes megjegyezni, hogy a részletek rendszerről rendszerre kissé eltérnek.

2.1.7. Szálak

Egy hagyományos operációs rendszerben minden egyes processzus saját címtartománnyal és egyetlen vezérlési szállal rendelkezik. Ez tulajdonképpen majdnem a teljes definíciója a processzusnak. Mindemellett gyakran fordul elő olyan helyzet, amikor kívánatos lenne több kvázi párhuzamosan futó vezérlési szál használata egy címtartományon belül, úgy, mintha különálló processzusok lennének (a közös címtartomány kivételével). Ezeket a vezérlési szálakat általában csak **szálaknak (thread)** hívjuk, vagy esetenként **könnyűsúlyú processzusoknak (lightweight process).**

A processzust tekinthetjük egymással összefüggő erőforrások egy csoportosítási módjának. A processzus címtartománya tartalmazza a program kódját, adatait és más erőforrásait. Ilyen erőforrások lehetnek megnyitott fájlok, gyermekprocesz-



2.6. ábra. *(a) Három processzus, mindegyik egy szállal. (b) Egy processzus három szállal*

szusok, függőben lévő ébresztők, szignálkezelők, elszámolási információk, egyebek. Ezek egyszerűbben kezelhetők, ha processzus formájában összerakjuk őket.

Egy újabb fogalom, amellyel a processzus rendelkezik, a végrehajtási szál, amit rendszerint **szál**ként rövidítenek. A szál rendelkezik utasításszámlálóval, amely nyilvántartja, hogy melyik utasítás végrehajtása következik. Regiszterei vannak, amelyek az aktuális munkaváltozóit tárolják. Rendelkezik veremmel, amely a végrehajtás eseményeit rögzíti, egy-egy kerettel minden meghívott eljáráshoz, amelyből nem tért még vissza a vezérlés. Bár a szálat egy processzuson belül kell végrehajtani, a szál és annak processzusa különböző fogalmak, így külön kezelhetők. A processzusok az erőforrások csoportosításai, a szálak pedig azok az egyedek, amelyeket CPU-n való végrehajtásra ütemeznek.

A szálak segítségével ugyanazon a processzuson belül több végrehajtást eszközölhetünk, amelyek egymástól nagymértékben függetlenek. A 2.6.(a) ábrán három hagyományos processzust látunk. Minden processzusnak saját címtartománya és egyetlen vezérlési szála van. Ezzel ellentétben a 2.6.(b) ábrán egy egyedülálló processzust látunk három vezérlési szállal. Bár mindkét esetben három vezérlési szálunk van, a 2.6.(a) ábrán mindegyik különböző címtartományban dolgozik, ugyanakkor a 2.6.(b) ábrán mindhárom ugyanazon a címtartományon osztozik.

Többszörös szálak használatára példaként tekintsünk egy webböngésző processzust. Sok weblap tartalmaz több kis képet. A böngészőnek a weblap minden képéért külön kapcsolatot kell kiépítenie a lapot tároló számítógéppel, és lekérnie a képet. Nagyon sok idő kárba vész az összes ilyen kapcsolat felépítése és lebontása miatt. A böngésző több szál egyidejű használatával több képet kérhet le egy időben, ezzel nagymértékben növelve a teljesítményt, mivel kis képek esetén a kapcsolat felépítésének ideje a korlátozó tényező és nem a sávszélesség.

Ha ugyanazon a címtartományon több szál van, a 2.4. ábra mezői közül néhány nem processzusonként, hanem szálanként értendő, vagyis egy különálló száltáblázatra van szükség szálankénti bejegyzésekkel. A szálankénti bejegyzések között lesz az utasításszámláló, a regiszterek és az állapot. Az utasításszámláló azért szükséges, mert a szálakat, hasonlóan a processzusokhoz, felfüggeszthetjük és folytat-

Processzushoz tartozó elemek	**Szálhoz tartozó elemek**
Címtartomány	Utasításszámláló
Globális változók	Regiszterek
Megnyitott fájlok	Verem
Gyermekprocesszusok	Állapot
Függőben lévő ébresztők	
Szignálok és szignálkezelők	
Elszámolási információ	

2.7. ábra. *Az első oszlop a processzushoz tartozó elemeket mutatja, amelyeken a szálak osztoznak. A második oszlop néhány, a szálakhoz egyenként tartozó elemet mutat*

hatjuk. A regiszterekre azért van szükség, mert amikor a szálakat felfüggesztjük, a regisztereiket el kell menteni. Végül a szálak, a processzusokhoz hasonlóan, *futó, futáskész* vagy *blokkolt* állapotban lehetnek. A 2.7. ábrán felsorolunk néhány processzushoz és szálhoz kapcsolódó elemet.

Vannak rendszerek, ahol az operációs rendszer nem vesz tudomást a szálakról. Más szóval azok teljes egészében a felhasználó hatáskörében vannak. Amikor például egy szál blokkolásra készül, kiválasztja és elindítja az utódját, mielőtt megáll. Különböző felhasználói szintű szálkezelő csomagok terjedtek el, mint a POSIX **P-szálak** és a Mach **C-szálak** csomagja.

Más rendszerekben az operációs rendszer tud arról, hogy a processzusnak több szála létezik. Ha egy szál blokkol, akkor az operációs rendszer választja ki a következőt futtatásra vagy ugyanabból a processzusból, vagy egy másikból. Ahhoz, hogy a kernel az ütemezést el tudja végezni, szüksége van egy száltáblázatra, amely a processzustáblázathoz hasonlóan a rendszerben lévő összes szálat nyilvántartja.

Bár lehet, hogy ez a két lehetőség egyformának látszik, teljesítményben jelentős mértékben különböznek. A szálak közötti kapcsolgatás sokkal gyorsabb, ha a szál kezelése a felhasználói szinten történik, mint amikor ehhez rendszerhívás szükséges. Ez az érv határozottan amellett szól, hogy a szálak kezelését a felhasználói szinten végezzük. Másrészről, amikor a szálakat teljes egészében a felhasználói szinten kezeljük, és egy szál blokkol (például I/O-ra vagy egy laphiba lekezelésére vár), a kernel az egész processzust blokkolja, hiszen nem tud más szálak létezéséről. Ez a tény másokkal együtt határozottan amellett érvel, hogy a szálak kezelését a kernelben kell elvégezni (Boehm, 2005). Következésképpen mindkét rendszert használják, és különféle hibrid megoldásokat is javasolnak (Anderson et al., 1992).

Mindegy, hogy a szálak kezelését a kernel vagy a felhasználói szint végzi, egy sor megoldandó probléma vetődik fel, amely a programozási modellt jelentősen megváltoztatja. Kezdetnek tekintsük a fork rendszerhívás hatásait. Ha a szülőprocesszusnak több szála van, létrehozzuk ezeket a gyermeknek is? Ha nem, lehet, hogy a processzus nem működik megfelelően, hiszen mindegyik szál lényeges lehet.

Mindamellett, ha a gyermekprocesszusnak ugyanannyi szála van, mint a szülőnek, mi fog történni, ha egy szál blokkolódik, mondjuk egy billentyűzetről történő read hívás miatt? Most két szál van blokkolva a billentyűzet miatt? Amikor egy

sort begépelünk, akkor mindkét szál kap ebből egy másolatot? Vagy csak a szülő? Vagy csak a gyermek? Hasonló problémák fordulnak elő nyitott hálózati kapcsolatoknál.

A problémák másik része ahhoz kapcsolódik, hogy a szálak adatszerkezeteken osztoznak. Mi történik, ha egy szál lezár egy fájlt, mialatt egy másik még olvas belőle? Tegyük fel, hogy egy szál észreveszi, hogy túl kevés a memória, és elkezd memóriát lefoglalni. Eközben egy szálváltás történik, és az új szál is észreveszi, hogy kevés a memória, és szintén elkezdi a memória lefoglalását. Egyszer vagy kétszer történik meg a foglalás? Majdnem minden rendszerben, amelyet nem arra terveztek, hogy szálakban gondolkodjon, a programkönyvtárak (mint például a memóriafoglaló eljárás) nem indíthatók újra és összeomlanak, ha másodszor is meghívjuk azokat, mialatt az első hívás még aktív.

Egy másik probléma a hibajelzéssel kapcsolatos. A Unixban egy rendszerhívás után a hívás állapota egy globális változóba, az *errno*-ba kerül. Mi történik, ha a szál végrehajt egy rendszerhívást, és mielőtt el tudná olvasni az *errno*-t, egy másik szál is végrehajt egy rendszerhívást, kitörölve az eredeti értéket?

Ezután tekintsük a szignálokat. A szignálok egy része logikailag szálspecifikus, míg mások nem. Például ha egy szál az alarm-ot hívja, az lenne a logikus, hogy az eredményszignál ahhoz a szálhoz jusson el, amelyik a hívást végrehajtotta. Ha a kernel tud a szálakról, akkor általában biztosak lehetünk abban, hogy a megfelelő szál kapja a szignált. Ha a kernel nem tud a szálakról, akkor a szálakat kezelő csomagnak magának kell valahogyan nyomon követnie a riasztásokat. További nehézség felhasználói szintű szálak kezelésénél, hogy (mint a Unixban is) egy processzusnak csak egy függőben lévő riasztása lehet, és mégis több szál is hívja az alarm-ot egymástól függetlenül.

Más szignálok, mint a billentyűzetről kezdeményezett *SIGINT*, nem szálspecifikusak. Ezeket kinek kell elkapnia? Egy kijelölt szálnak? Mindegyiknek? Egy újonnan létrehozott szálnak? Mindegyik megoldásban vannak problémák. Ráadásul mi történik, ha egy szál lecseréli a szignálkezelőket anélkül, hogy erről értesítené a többieket?

A szálak okozta utolsó probléma a veremkezelés. Sok rendszerben, amikor veremtúlcsordulás történik, a kernel egyszerűen automatikusan megnöveli a vermet. Ha a processzusnak több szála van, akkor lennie kell több vermének is. Ha a kernel nem tud az összes veremről, akkor nem képes azokat automatikusan megnövelni veremhiba esetén. Tény, hogy még csak fel sem ismeri, hogy a memóriahiba összefügg a verem növekedésével.

Ezek a problémák bizonyára nem leküzdhetetlenek, de megmutatják, hogy szálak bevezetése egy már meglévő rendszerbe, annak alapvető újratervezése nélkül, nem vezet működőképes rendszerhez. Legalább a rendszerhívások szemantikáját újra kell definiálni, és a könyvtárakat újra kell írni. Mindezeket úgy kell végrehajtani, hogy visszafelé kompatibilis maradjon azokkal a meglévő programokkal, amelyek a processzusokat az egyszálas esetre korlátozzák. A szálakról további információk állnak rendelkezésre (Hauser et al., 1993; és Marsh et al., 1991).

2.2. Processzusok kommunikációja

A processzusoknak gyakran szükségük van az egymással való kommunikációra. Például egy parancsértelmező adatcsőben, amikor az első processzus kimenő adatait át kell adni a második processzusnak, és sorban így tovább. Így szükség van a processzusok közötti kommunikációra, előnyben részesítve egy megszakítások nélküli, jól strukturált módot. A következő fejezetekben áttekintünk néhány, a **processzusok kommunikációjával**, az **IPC**-vel (**InterProcess Communication**) kapcsolatos témát.

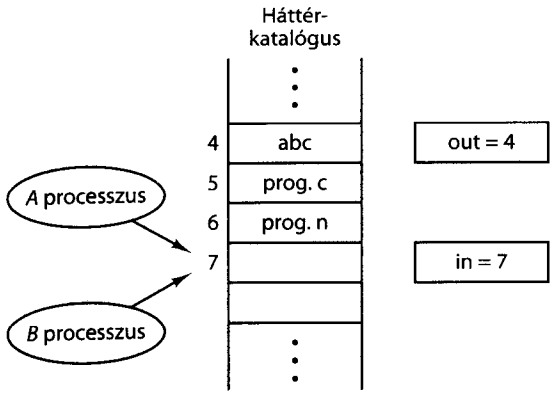
Itt három téma merül fel. Az első arról szól, hogyan tud egy processzus információt küldeni egy másiknak. A másodikban biztosítani kell, hogy kettő vagy több processzus ne tudja egymás útját keresztezni, amikor kritikus tevékenységbe kezdenek (tegyük fel, hogy két processzus mindegyike megpróbálja megszerezni az utolsó 1 MB memóriát). A harmadik függőség esetén a megfelelő sorrendbe állítással foglalkozik: ha az *A* processzus adatokat állít elő, és a *B* processzus kinyomtatja azt, akkor *B*-nek várnia kell, míg az *A* néhány adatot elkészít, és csak ezután kezdhet nyomtatni. Mindhárom témát meg fogjuk vizsgálni a következő szakaszban.

Érdemes megemlíteni, hogy két téma a szálakra is ugyanúgy vonatkozik. Az első – az információküldés – szálak esetében egyszerű, mivel közös címtartományon osztoznak (különböző címtartományban található szálak kommunikációja az egymással kommunikáló processzusok témához tartozik). A másik kettő azonban – egymást nem akadályozni és a megfelelő sorrendet kialakítani – a szálakra is vonatkozik. A problémák és a megoldásaik megegyeznek. A következőkben a problémát a processzusok keretén belül tárgyaljuk, de jegyezzük meg, hogy a problémák és a megoldások a szálak esetében is ugyanazok.

2.2.1. Versenyhelyzetek

Vannak operációs rendszerek, ahol az együtt dolgozó processzusok közös tárolóterületen osztozhatnak, amelyből mindegyik olvashat és amelybe mindegyik írhat. A megosztott tároló lehet a főmemóriában (valószínűleg egy kernel-adatstruktúrában), vagy lehet egy megosztott fájl; a megosztott tároló elhelyezkedése nem változtat a kommunikáció természetén vagy a felmerülő problémákon. Hogy lássuk, hogyan dolgozik a gyakorlatban a processzusok kommunikációja, tekintsünk egy egyszerű, de általános példát, a háttérnyomtatást. Ha egy processzus ki akar nyomtatni egy fájlt, akkor beteszi a fájl nevét egy speciális **háttérkatalógus**ba. Egy másik processzus, a **nyomtató démon**, rendszeresen ellenőrzi, hogy van-e nyomtatandó fájl, és ha van, akkor kinyomtatja és kitörli a nevét a katalógusból.

Képzeljük el, hogy a háttérkatalógusunknak nagyszámú rekesze van, amelyek sorszáma 0, 1, 2, ..., és mindegyik egy fájlnév tárolására alkalmas. Ezenkívül képzeljük el, hogy van két megosztott változónk, *out*, amely a következő nyomtatandó fájlra mutat, és *in*, amely a katalógus következő szabad rekeszére mutat. Ez a két változó jól tárolható egy kétszavas fájlban, amely minden processzus számára elér-



2.8. ábra. *Két processzus ugyanabban az időben akarja elérni a megosztott memóriát*

hető. Egy bizonyos pillanatban a 0–3-as rekeszek üresek (a fájlokat már kinyomtattuk), és a 4–6-os rekeszek teli vannak (a nyomtatásra sorban álló fájlok nevével). Többé-kevésbé egy időben az *A* és *B* processzusok elhatározzák, hogy besorolnak egy fájlt a nyomtatásra várók sorába. Ezt a helyzetet mutatja a 2.8. ábra.

Murphy törvényét (ami el tud romlani, az el is romlik) alkalmazva a következő történhet. Az *A* processzus elolvassa az *in* tartalmát, és a 7-es értéket tárolja a *következő_szabad_rekesz* lokális változóban. Éppen ekkor egy óramegszakítás történik, és a CPU elhatározza, hogy az *A* processzus már elég hosszú ideje fut, és átkapcsol a *B* processzusra. A *B* processzus szintén elolvassa az *in*-t, szintén 7-et kap, így a 7-es rekeszbe fogja a fájl nevét tárolni, és az *in*-t 8-ra frissíti. Ezután továbbhalad, és más dolgokat csinál.

Végül az *A* processzus ismét futni kezd, ott folytatva, ahol legutóbb abbahagyta. Megnézi a *következő_szabad_*rekesz-t, 7-et talál benne, és beírja a fájlnevet a 7-es rekeszbe, törölve azt a nevet, amelyet a *B* processzus éppen most tett oda. Ezután kiszámolja a *következő_szabad_rekesz* + 1 értéket, amely 8, és *in*-t 8-ra állítja. A háttérkatalógus most belsőleg konzisztens, tehát a nyomtatóprogram nem észlel hibát, de a *B* processzus sohasem kap kimenetet. A *B* felhasználó évekig várakozhat a nyomtatószobában, reménykedve várva a kimenetet, amely sohasem fog elkészülni. Az ehhez hasonló eseteket, ahol kettő vagy több processzus olvas vagy ír megosztott adatokat, és a végeredmény attól függ, hogy ki és pontosan mikor fut, **versenyhelyzeteknek** nevezzük. Egyáltalán nem szórakoztató nyomon követni versenyhelyzeteket tartalmazó programokat. A legtöbb teszt eredménye jó, de nagyon ritkán előfordulnak furcsa és megmagyarázhatatlan dolgok.

2.2.2. Kritikus szekciók

Hogyan kerüljük el a versenyhelyzeteket? A baj megelőzésének kulcsa – itt is és sok más esetben is, ahol megosztott memória, megosztott fájlok és bármi más megosztott dolog szerepel – az, hogy találjunk valamilyen módot annak megtiltására, hogy egy időben egynél több processzus olvassa és írja a megosztott adatokat. Más szavakkal, amire nekünk szükségünk van, az a **kölcsönös kizárás** – egy módszer, amely biztosítja, hogy ha egy processzus használ valamely megosztott változót vagy fájlt, akkor a többi processzus tartózkodjon ettől a tevékenységtől. A fenti probléma azért fordult elő, mert a *B* processzus azelőtt kezdte el használni a megosztott változók egyikét, mielőtt az *A* processzus végzett volna vele. Bármely operációs rendszer egyik fő tervezési szempontja, hogy megválasszuk az alkalmas primitív műveleteket a kölcsönös kizárás eléréséhez; a következőkben ezt a témát fogjuk részletesen megvizsgálni.

A versenyhelyzetek elkerülésének problémáját absztrakt módon is megfogalmazhatjuk. Az idő egy részében a processzus belső számolási és egyéb olyan tevékenységekkel van elfoglalva, amelyek nem vezetnek versenyhelyzetekhez. Azonban néha a processzus megosztott memóriához vagy fájlokhoz nyúl. A programnak azt a részét, amelyben a megosztott memóriát használja, **kritikus területnek** vagy **kritikus szekciónak** nevezzük. Ha úgy tudnánk rendezni a dolgokat, hogy soha ne legyen azonos időben két processzus a kritikus szekciójában, akkor elkerülhetnénk a versenyhelyzeteket.

Bár ez a követelmény megóv a versenyhelyzetektől, mégsem elegendő ahhoz, hogy korrekten együttműködő párhuzamos processzusaink legyenek, és azok hatékonyan használják a megosztott adatokat.

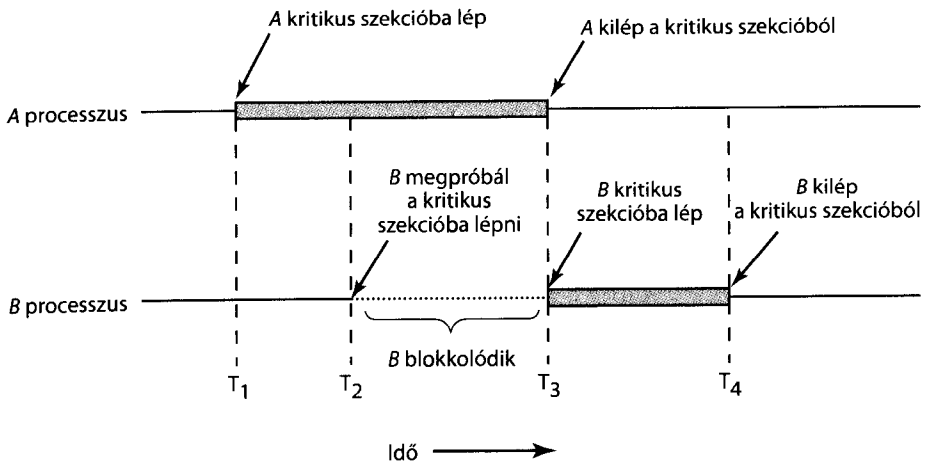
A jó megoldáshoz négy feltételt kell betartani:

1. Ne legyen két processzus egyszerre a saját kritikus szekciójában.
2. Semmilyen előfeltétel ne legyen a sebességekről vagy a CPU-k számáról.
3. Egyetlen, a kritikus szekcióján kívül futó processzus sem blokkolhat más processzusokat.
4. Egyetlen processzusnak se kelljen örökké arra várni, hogy belépjen a kritikus szekciójába.

A kívánt viselkedést a 2.9. ábra mutatja be. Itt az *A* processzus a T_1 időpillanatban lép be a kritikus területre. Egy kicsivel később, a T_2 időpillanatban a *B* processzus is megpróbál a kritikus területre lépni, de ez sikertelen lesz, mert egy másik processzus már belépett, és mi egyszerre csak egyet engedélyezünk. Ennek eredményeként a *B* processzus futása ideiglenesen felfüggesztődik a T_3 időpillanatig, amikor *A* elhagyja a kritikus területét, lehetővé téve ezzel, hogy *B* azonnal beléphessen. Egyszer csak (a T_4 időpillanatban) *B* is kilép a kritikus szekciójából, és így visszakerülünk a kiindulási helyzetbe, amikor nem volt processzus a kritikus szekciójában.

2.2.3. Kölcsönös kizárás tevékeny várakozással

Ebben az alfejezetben különféle lehetőségeket fogunk megvizsgálni a kölcsönös kizárás megvalósítására, azaz mialatt egy processzus azzal van elfoglalva, hogy a saját kritikus szekciójában a megosztott memóriát aktualizálja, ne legyen más olyan processzus, amely belép *saját* kritikus szekciójába és bajt okoz.



2.9. ábra. *Kölcsönös kizárás kezelése kritikus szekciókkal*

Megszakítások tiltása

A legegyszerűbb megoldás az, hogy minden processzus letiltja az összes megszakítást, mihelyt belép saját kritikus szekciójába, és újraengedélyezi, éppen mielőtt elhagyja azt. Azzal, hogy a megszakítások tiltva vannak, nem fordulhat elő óramegszakítás sem. Mivel a CPU csak órajelre vagy más megszakításra vált egyik processzusról a másikra, végül is a megszakítások kikapcsolásával a CPU nem fog másik processzusra váltani. Így ha egyszer egy processzus letiltotta a megszakításokat, megvizsgálhatja és módosíthatja a megosztott memóriát anélkül, hogy bármelyik más processzus beavatkozásától tartania kellene.

Ez a megközelítés azonban nem igazán vonzó, mert oktalanság a felhasználói processzusok kezébe adni a megszakítások kikapcsolásának lehetőségét. Tegyük fel, hogy egyikük megteszi ezt, és soha nem kapcsolja vissza. Ez a rendszer végét jelentheti. Továbbá egy többprocesszoros rendszerben, ahol kettő vagy több CPU van, a megszakítások tiltása csak arra a CPU-ra vonatkozik, amelyik a tiltó utasítást végrehajtotta. A többiek folytatják a futtatást, és hozzáférhetnek a megosztott memóriához.

Másrészt magának a kernelnek is gyakran hasznos a megszakítások tiltása néhány utasítás erejéig, amíg változókat vagy listákat aktualizál. Ha például megszakítás következne be, mialatt a futáskész állapotú processzusok listája inkonzisztens állapotban van, akkor versenyhelyzet fordulhatna elő. Ebből az következik, hogy a megszakítások tiltása gyakran hasznos technika magán az operációs rendszeren belül, de nem megfelelő a felhasználói processzusok számára mint általános kölcsönös kizárási mechanizmus.

Zárolásváltozók

Második lehetőségként keressünk egy szoftvermegoldást. Tekintsünk egy egyszerű, megosztott (zárolás-) változót, kezdetben 0 értékkel. Mielőtt egy processzus belépne a saját kritikus szekciójába, először ezt vizsgálja meg. Ha értéke 0, akkor a processzus 1-re állítja azt, és belép a kritikus szekcióba. Ha már 1, akkor a processzus addig vár, míg 0 lesz. Így a 0 azt jelenti, hogy egyetlen processzus sincs a saját kritikus szekciójában, és az 1 azt, hogy valamely processzus a saját kritikus szekciójában van.

Sajnos, ez az elgondolás pontosan ugyanazt a végzetes hibát rejti magában, mint amelyet már a háttérkatalógus esetében láttunk. Tegyük fel, hogy az egyik processzus elolvassa a zárolásváltozót, és látja, hogy értéke 0. Mielőtt be tudná állítani 1-re, egy másik processzus kerül ütemezésre, fut, és beállítja a változót 1-re. Amikor az első processzus ismét futni fog, megint beállítja 1-re a változót, és máris két processzus lesz egy időben a saját kritikus szekciójában.

Azt gondolhatnánk, hogy megkerülhetjük ezt a problémát azzal, hogy először kiolvassuk a zárolásváltozó értékét, majd ismét ellenőrizzük azt pontosan azelőtt, hogy írnánk bele, de ez valójában nem segít. A verseny akkor fog bekövetkezni, ha a második processzus éppen azután módosítja a változót, amikor az első processzus a második ellenőrzését befejezte.

Szigorú váltogatás

A kölcsönös kizárás problémájának harmadik megközelítését a 2.10. ábra mutatja. Ez a programtöredék, mint majdnem mindegyik ebben a könyvben, C-ben íródott. A C nyelvet azért választottuk, mert a valódi operációs rendszerek is általában C-ben íródnak (esetleg C++-ban), de szinte sohasem Javában vagy hasonló nyelven. A C nyelv erőteljes, hatékony és kiszámítható. Olyan jellemzők ezek, amelyek kritikusak operációs rendszerek írásához. A Java például nem kiszámítható, mert ha egy kritikus pillanatban fogy el a tárhely, akkor a szemétgyűjtőt a legkevésbé alkalmas pillanatban indítja cl. A C nyelv esetében ez nem fordulhat elő, mert nincs szemétgyűjtő mechanizmusa. A C, C++, Java és négy másik nyelv kvantitatív összehasonlítását megtalálhatjuk Prechelt (Prechelt, 2000) cikkében.

```
while (TRUE) {
    while (turn != 0)        /* ciklus */;
    critical_region();
    turn = 1;
    noncritical_region();
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* ciklus */;
    critical_region();
    turn = 0;
    noncritical_region();
}
```

(b)

2.10. ábra. *Egy javasolt megoldás a kritikus szekció problémára. (a) 0. processzus. (b) 1. processzus. Mindkét esetben figyeljünk arra, hogy a* while *utasítást pontosvessző (;) zárja le*

A 2.10. ábrán a 0 kezdőértékű *turn* egész változó követi nyomon, hogy ki lesz a következő, aki a kritikus szekcióba lép, és vizsgálja vagy aktualizálja a megosztott memóriát. Kezdetben a 0. processzus nézi meg a *turn* értékét, 0-nak találja, és belép a kritikus szekciójába. Az 1. processzus szintén 0-nak találja, és ezért belép egy rövid ciklusba, folyamatosan tesztelve a *turn* értékét, hogy lássa, mikor lesz 1. Azt, amikor folyamatosan tesztelünk egy változót egy bizonyos érték megjelenéséig, **tevékeny várakozás**nak nevezzük. Általában tartózkodni kellene ettől, mert pazarolja a CPU-időt. Csak akkor használjuk a tevékeny várakozást, ha ésszerűen elvárható, hogy a várakozás rövid lesz. A tevékeny várakozást használó zárolásokat **aktív várakozás**nak hívjuk.

Amikor a 0. processzus elhagyja a kritikus szekciót, beállítja a *turn*-t 1-re, hogy megengedje az 1. processzus kritikus szekciójába lépését. Tegyük fel, hogy az 1. processzus gyorsan befejezi a kritikus szekcióját, így mindkét processzus a saját nemkritikus területén van, a *turn* értéke pedig 0. Most a 0. processzus gyorsan végrehajtja a teljes ciklusát, elhagyja a kritikus szekcióját, beállítva a *turn*-t 1-re. Ezen a ponton *turn* értéke 1, és mindkét processzus a nemkritikus területen fut.

A 0. processzus hirtelen befejezi a nemkritikus szekcióját, és visszatér a ciklusának az elejére. Sajnos, nincs megengedve neki, hogy most belépjen a kritikus szekciójába, mert a *turn* értéke 1, és az 1. processzus a nemkritikus szekciójával van elfoglalva. Addig marad a while ciklusában, amíg az 1. processzus nem állítja *turn* értékét 0-ra. A váltogatás tehát nem túl jó ötlet, amikor az egyik processzus sokkal lassabb, mint a másik.

Ez a helyzet megsérti az előbb felállított 3. feltételt: a 0. processzust blokkolta egy olyan processzus, amely nem a kritikus szekciójában van. Visszatérve a nemrég tárgyalt háttérkatalógusra, ha most a háttérkatalógus olvasását és írását tekintjük kritikus szekciónak, akkor a 0. processzus nem nyomtathatna másik fájlt, mert az 1. valami mást csinál.

Valójában ez a megoldás megköveteli, hogy két a processzus egymást szigorúan váltogatva lépjen be saját kritikus szekciójába, például fájlokat tegyenek be a háttérkatalógusba. Egyiknek sincs megengedve, hogy egymás után kettőt adjon át. Bár ez az algoritmus elkerül minden versenyt, mégsem számít komoly jelöltnek a probléma megoldására, mert a 3. feltételt megsérti.

Peterson megoldása

Kombinálva az egymás váltogatásának ötletét a zárolásváltozók és a figyelmeztető változók ötletével, T. Dekker holland matematikus volt az első, aki kitalált egy olyan szoftvermegoldást a kölcsönös kizárás problémájára, amely nem igényel szigorú váltogatást. Dekker algoritmusát részletesen lásd (Dijkstra, 1965).

1981-ben G. L. Peterson talált egy egyszerűbb módot a kölcsönös kizárás megvalósítására, amely elavulttá tette Dekker megoldását. Peterson algoritmusát a 2.11. ábra mutatja. Ez az algoritmus két ANSI C-ben írt eljárásból áll, ami azt jelenti, hogy minden definiált és használt függvényhez függvényprototípusokat kell


```
#define FALSE  0
#define TRUE   1
#define N      2                                    /* a processzusok száma */

int turn;                                           /* ki következik? */
int interested[N];                                  /* kezdetben minden érték 0 (FALSE) */

void enter_region(int process);                     /* process vagy 0, vagy 1 */
{
   int other;                                       /* a másik processzus sorszáma */

   other = 1 - process;                             /* a process ellenkezője */
   interested[process] = TRUE;                      /* mutatja, hogy érdekeltek vagyunk */
   turn = process;                                  /* áll a zászló */
   while (turn == process && interested[other] == TRUE)    /* üres utasítás */ ;
}

void leave_region(int process)                      /* process: ki hagyja el */
{
   interested[process] = FALSE;                     /* mutatja a kritikus szekció elhagyását */
}
```

2.11. ábra. *Peterson megoldása a kölcsönös kizárás megvalósítására*

biztosítanunk. Helytakarékosságból azonban mi nem fogjuk megmutatni a prototípusokat ebben és a további példákban sem.

Mielőtt a megosztott változókat használná (vagyis mielőtt a kritikus szekcióba lépne), minden processzus meghívja az *enter_region*-t, paraméterként átadva a saját processzus sorszámát, 0-t vagy 1-et. Ez a hívás azt eredményezi, hogy ha szükséges, akkor a biztonságos belépésig várakozni fog. Miután végzett a megosztott változókkal, a processzus meghívja a *leave_region*-t, jelezve, hogy végzett, és megengedi a másik processzusnak, hogy belépjen, ha akar.

Nézzük, hogyan működik ez a megoldás. Kezdetben egyik processzus sincs a kritikus szekciójában. Most a 0. processzus meghívja az *enter_region*-t, amely jelzi az érdekeltségét a tömbelemének beállításával, majd a *turn* változót 0-ra állítja. Ha az 1. processzus nem érdekelt, az *enter_region* azonnal visszatér. Ha az 1. processzus most meghívja az *enter_region*-t, addig vár, míg az *interested*[0] *FALSE*-ra vált, ami csak akkor következik be, ha a 0. processzus meghívja a *leave_region*-t, hogy kilépjen a kritikus szekcióból.

Most tekintsük azt az esetet, amikor mindkét processzus majdnem egyszerre meghívja az *enter_region*-t. Mindkettő beírja a saját processzussorszámát a *turn*-be. Csak az utolsó beírás fog számítani; az első elvész. Tegyük fel, hogy az 1. processzus ír be utoljára, vagyis a *turn* értéke 1. Amikor mindkét processzus a while utasításhoz ér, a 0. processzus a várakozó ciklust nullaszor hajtja végre, és belép a kritikus szekciójába. Az 1. processzus ismételget, és nem lép be a kritikus szekciójába.

A TSL utasítás

Most lássunk egy olyan javaslatot, amelyik egy kis hardversegítséget igényel. Sok számítógépnek, különösen azoknak, amelyeket többprocesszorosnak terveztek, van egy

```
TSL RX,LOCK
```

utasítása (Test and Set Lock), amely a következőképpen dolgozik: beolvassa a *LOCK* memóriaszó tartalmát az RX regiszterbe, és ezután egy nem nulla értéket ír erre a memóriacímre. A szó kiolvasása és a tárolási művelet garantáltan nem választható szét – az utasítás befejezéséig más processzor nem érheti el a memóriaszót. A TSL utasítást végrehajtva a CPU zárolja a memóriasínt, a művelet befejezéséig megtiltva más CPU-knak a memória elérését.

A TSL utasítás alkalmazásához egy *LOCK* megosztott változót fogunk használni, hogy összehangoljuk a megosztott memória elérését. Amikor a *LOCK* 0, bármelyik processzus beállíthatja 1-re a TSL utasítás használatával, és ezután olvashatja vagy írhatja a megosztott memóriát. Amikor ezt megtette, a processzus visszaállítja a *LOCK* értékét 0-ra egy egyszerű MOVE utasítással.

Hogyan használhatjuk ezt az utasítást annak megakadályozására, hogy két processzus egyidejűleg lépjen be saját kritikus szekciójába? A megoldást a 2.12. ábra mutatja, ahol látunk egy négyutasításos szubrutint egy fiktív (de tipikus) assembly nyelven. Az első utasítás átmásolja a *LOCK* régi értékét a regiszterbe, majd a *LOCK*-ot 1-re állítja. Ezután a régi értéket összehasonlítja 0-val. Ha nem 0, a zárolás már megtörtént, így a program visszatér az elejére, és ismét tesztelni fogja. Előbb vagy utóbb 0 lesz (amikor a jelenleg a kritikus szekciójában lévő processzus végez kritikus szekciójával), és a szubrutin visszatér a zárolás beállításával. A zárolás feloldása egyszerű. A program csupán 0-t tárol a *LOCK*-ba. Nincs szükségünk speciális utasításra.

A kritikus szekció probléma egy megoldása most már egyszerű. Mielőtt a processzus belép a kritikus szekciójába, meghívja az *enter_region*-t, amely tevékenyen várakozik a zárolás feloldásáig; ezután zárol és visszatér. A kritikus szekció után a processzus meghívja a *leave_region*-t, amely 0-t tárol a *LOCK*-ba. Minden, a kritikus szekciókon alapuló megoldásnál a processzusoknak a megfelelő időben kell hívniuk az *enter_region*-t és a *leave_region*-t, hogy a módszer működjön. Ha egy processzus csal, a kölcsönös kizárás meghiúsul.

```
enter_region:
    TSL REGISTER,LOCK    | átmásolja a LOCK-ot a regiszterbe, és LOCK legyen 1
    CMP REGISTER,#0      | a LOCK nulla volt?
    JNE ENTER_REGION     | ha nem volt nulla, akkor be volt állítva, így ciklus
    RET                  | vissza a hívóhoz; beléptünk a kritikus szekcióba

leave_region:
    MOVE LOCK,#0         | LOCK legyen 0
    RET                  | visszatérés a hívóhoz
```

2.12. ábra. *A zárolás beállítása és törlése a* TSL *utasítással*

2.2.4. Alvás és ébredés

Mind Peterson megoldása, mind az a megoldás, amelyben a TSL utasítást használjuk, korrekt, de mindkettőnek megvan az a hibája, hogy tevékeny várakozást követel meg. Ezek a megoldások lényegében a következőképpen működnek: amikor egy processzus be akar lépni a kritikus szekciójába, ellenőrzi, hogy a belépés engedélyezett-e. Ha nem, akkor a processzus azonnal egy kis ciklusban marad az engedélyre várva.

Ez a megközelítés nemcsak a CPU-időt pazarolja, de még váratlan hatásai is lehetnek. Tekintsünk egy számítógépet két processzussal, a *H* legyen magas, az *L* pedig alacsony prioritású, amelyek egy kritikus szekción osztoznak. Az ütemezési szabályok olyanok, hogy valahányszor *H* futáskész állapotban van, futni fog. Egy bizonyos pillanatban, amikor *L* a kritikus szekciójában van, *H* futáskész állapotba kerül (például egy I/O-művelet befejeződik). *H* most tevékeny várakozásba kezd, de mivel *L*-re soha nem kerül sor, amikor *H* fut, így *L* soha nem kap esélyt, hogy elhagyja a kritikus szekcióját, ezért *H* a végtelenségig ismétel. Erre az esetre néha úgy szoktak hivatkozni, mint **fordított prioritás probléma**.

Most lássunk néhány olyan processzusok közötti kommunikációs primitívet, amelyek a CPU-idő pazarlása helyett blokkolnak, amikor nem megengedett, hogy a kritikus szekciójukba lépjenek. Az egyik legegyszerűbb a sleep és wakeup pár. A sleep egy rendszerhívás, amely a hívót blokkolja, vagyis fel lesz függesztve mindaddig, amíg egy másik processzus fel nem ébreszti. A wakeup hívásnak egy paramétere van, az a processzus, amelyet fel kell ébreszteni. Másik alternatíva az, hogy mind a sleep, mind a wakeup egy paraméterrel, egy memóriacímmel rendelkezik, amit a sleep-ek és a wakeup-ok összepárosítására használunk.

A gyártó-fogyasztó probléma

A primitívek használatára példaként tekintsük a **gyártó-fogyasztó problémát** (**korlátos tároló** problémának is nevezik). Két processzus osztozik egy közös, rögzített méretű tárolón. Az egyikük, a gyártó, adatokat helyez el benne, a másikuk, a fogyasztó, kiveszi azokat. (Általánosíthatjuk a problémát *m* gyártóra és *n* fogyasztóra is, de mi csak az egy gyártó és egy fogyasztó esetet tekintjük, mert ez a feltételezés egyszerűsíti a megoldásokat.)

A nehézség akkor jelentkezik, amikor a gyártó új elemet kíván a tárolóba tenni, de az már tele van. A megoldás a gyártó számára az, hogy elalszik, és felébresztik, amikor a fogyasztó egy vagy több elemet kivett. Hasonlóképpen, ha a fogyasztó szeretne egy elemet kivenni a tárolóból és látja, hogy a tároló üres, akkor elalszik, amíg a gyártó tesz valamit a tárolóba és felébreszti őt.

Ez a megközelítés elég egyszerűnek tűnik, bár ugyanolyan típusú versenyhelyzetekhez vezet, mint amilyet korábban láttunk a háttérkatalógusnál. Hogy nyomon követhessük az elemek számát a tárolóban, szükségünk lesz egy *count* változóra. Ha a tárolóban az elemek maximális száma *N*, akkor a gyártó programja először

```
#define N 100                                   /* a rekeszek száma a tárolóban */
int count = 0;                                  /* az elemek száma a tárolóban */

void producer(void)
{
   int item;

   while (TRUE) {                               /* végtelen ciklus */
      item = produce_item();                    /* a következő elem létrehozása */
      if (count = = N) sleep();                 /* ha a tároló tele van, megyünk aludni */
      insert_item(item);                        /* betesszük az elemet a tárolóba */
      count = count + 1;                        /* növeljük az elemek számát a tárolóban */
      if (count = = 1) wakeup(consumer);        /* üres volt a tároló? */
   }
}

void consumer(void)
{
   int item;

   while (TRUE) {                               /* végtelen ciklus */
      if (count = = 0) sleep();                 /* ha a tároló üres, megyünk aludni */
      item = remove_item();                     /* kiveszünk egy elemet a tárolóból */
      count = count - 1;                        /* csökken az elemek száma a tárolóban */
      if (count = = N - 1) wakeup(producer);    /* tele volt a tároló? */
      consume_item(item);                       /* kinyomtatjuk az elemet */
   }
}
```

2.13. ábra. *A gyártó-fogyasztó probléma egy végzetes versenyhelyzettel*

azt fogja vizsgálni, hogy a *count* értéke egyenlő-e *N*-nel. Ha igen, akkor a gyártó elalszik, ha nem, akkor hozzátesz egy elemet, és növeli a *count* értékét.

A fogyasztó programja hasonló, először vizsgálja a *count* értékét, hogy egyenlő-e nullával. Ha igen, akkor elalszik, ha nem nulla, akkor kivesz egy elemet, és csökkenti a számlálót. Mindegyik processzus teszteli azt is, hogy a másiknak aludnia kell-e, és ha nem, akkor felébreszti. A 2.13. ábrán látható mind a gyártó, mind a fogyasztó kódja.

Ahhoz, hogy a rendszerhívásokat, mint a sleep és a wakeup, kifejezhessük C-ben, könyvtári rutinok hívásaként fogjuk bemutatni. Ezek nem részei a standard C könyvtárnak, de feltételezhetően elérhetők mindazokban a rendszerekben, amelyekben megvannak ezek a rendszerhívások. Az *enter_item* és *remove_item* eljárások, amelyeket most nem mutatunk be, végzik az elemek tárolóba helyezését, illetve tárolóból való kivételét.

Térjünk most vissza a versenyhelyzethez. Ez előfordulhat, hiszen a *count* elérése nem korlátozott. A következő helyzet fordulhat elő: A tároló üres és a fogyasztó éppen most olvasta ki a *count* értékét, hogy megnézze, nulla-e. Ebben a pillanatban az ütemező elhatározza, hogy ideiglenesen megállítja a fogyasztó futását

és elindítja a gyártót. A gyártó betesz egy elemet a tárolóba, növeli a *count* értékét, és megállapítja, hogy az most 1. Ebből arra következtet, hogy a *count* éppen 0 volt, és így a fogyasztó bizonyára alszik. Tehát a gyártó hívja a *wakeup*-ot, hogy felébressze a fogyasztót.

Sajnos, a fogyasztó logikailag még nem alszik, így az ébresztő jelzés elvész. Amikor a fogyasztó ismét fut, megvizsgálja a *count* azon értékét, amelyet előzőleg beolvasott, 0-nak találja, és elalszik. Előbb-utóbb a gyártó megtölti a tárolót, és szintén elalszik. Mindkettő örökké aludni fog.

A probléma lényege az, hogy egy ébresztőjel, amelyet egy (még) nem alvó processzusnak küldtek, elveszett. Ha nem veszett volna el, akkor minden működne. Egy gyors javítás az, hogy módosítjuk a szabályokat egy **ébresztőt váró bit** hozzáadásával. Ha olyan processzusnak küldünk ébresztőt, amely még ébren van, akkor ez a bit beállítódik. Később, amikor a processzus megpróbál elaludni, de az ébresztőt váró bit be van kapcsolva, akkor kikapcsolja ezt a bitet, és ébren marad. Az ébresztőt váró bit egy másodlagos eszköz az ébresztő jel számára.

Míg az ébresztőt váró bit ebben az egyszerű példában megmenti a helyzetet, könnyű olyan példát konstruálni három vagy több processzussal, ahol egy ébresztőt váró bit nem elegendő. Készíthetünk újabb javítást is, és hozzávehetünk kettő vagy akár 8, vagy 32 ébresztőt váró bitet is, a probléma lényege megmarad.

2.2.5. Szemaforok

Ez volt a helyzet egészen addig, amíg 1965-ben E. W. Dijkstra (Dijkstra, 1965) azt nem javasolta, hogy egy egész változóban számoljuk az ébresztéseket későbbi felhasználás céljából. Javaslatában egy új változótípust vezetett be, amelyet **szemafor**nak hívunk. A szemafor értéke lehet 0, jelezve, hogy nincs elmentett ébresztés, vagy valamilyen pozitív érték, ha egy vagy több ébresztés függőben van.

Dijkstra azt javasolta, hogy két művelet legyen, a down és az up (rendre a sleep és a wakeup általánosításai). A down művelet megvizsgálja, hogy a szemafor értéke nagyobb-e, mint 0. Ha igen, csökkenti az értéket (vagyis felhasznál egy tárolt ébresztést), és azonnal folytatja. Ha az érték 0, akkor a processzust elaltatja, mielőtt a down befejeződne. Az érték ellenőrzése, cseréje és a lehetséges elalvás együtt egyetlen oszthatatlan **elemi művelet**ként hajtódik végre. Ez garantálja, hogy ha egy szemafor művelet elkezdődik, más processzus nem tudja elérni a szemafort mindaddig, amíg a művelet be nem fejeződik vagy nem blokkolódik. Az elemi művelet bevezetése nagyon lényeges a szinkronizációs problémák megoldásához és a versenyhelyzetek elkerüléséhez.

Az up művelet a megadott szemafor értékét növeli. Ha egy vagy több processzus aludna ezen a szemaforon, mivel képtelen volt befejezni egy korábbi down műveletet, akkor közülük az egyiket kiválasztja a rendszer (például véletlenszerűen), és megengedi neki, hogy befejezze a down műveletét. Így olyan szemaforon végrehajtva az up műveletet, amelyen processzusok aludtak, a szemafor még mindig 0 lesz, de eggyel kevesebb processzus fog rajta aludni. A szemafor növelésének és egy processzus felébresztésének művelete szintén nem választható szét. Egy up

műveletet végrehajtó processzus nem blokkolható, mint ahogy az előző modellben a wakeup végrehajtása sem volt az.

Mellesleg Dijkstra az eredeti cikkében a p és v neveket használta rendre a down és up helyett, de mivel ezek nehezen jegyezhetők meg azok számára, akik nem beszélik a holland nyelvet (és azok számára is csak részben, akik beszélik), mi ehelyett a down és up kifejezéseket használjuk. Ezeket először az Algol 68-ban vezették be.

A gyártó-fogyasztó probléma megoldása szemaforok segítségével

A 2.14. ábrán bemutatjuk, hogy a szemaforok megoldják az elveszett ébresztés problémáját. Fontos, hogy ezek oszthatatlan módon legyenek megvalósítva. Kézenfekvő, hogy az up és a down rendszerhívásként kerül megvalósításra, amelyben az operációs rendszer egyszerűen tilt minden megszakítást, mialatt vizsgálja és aktualizálja a szemafort, valamint elaltatja a processzust, ha kell. Mivel mindezen tevékenységek csak néhány utasításból állnak, nem okoz bajt a megszakítások tiltása. Ha több CPU-t használunk, minden szemafort védeni kell egy zárolásváltozóval, a TSL utasítást használva annak biztosítására, hogy egy időben csak egy CPU vizsgálja a szemafort. Biztosan érthető a különbség aközött, hogy TSL utasítást használva megvédjük a szemafort attól, hogy egy időben több CPU érje el, és aközött, hogy a gyártó vagy fogyasztó tevékeny várakozással a másikra vár, hogy az kiürítse vagy feltöltse a tárolót. A szemafor művelet mindössze néhány mikromásodpercig tart, míg a gyártónál vagy fogyasztónál tetszőlegesen sokáig tarthat.

Ez a megoldás három szemafort használ: a teli rekeszek számolására szolgál a *full,* az üres rekeszek számolására szolgál az *empty,* a *mutex* pedig azt biztosítja, hogy a gyártó és fogyasztó ne érje el a tárolót egy időben. Kezdetben a *full* 0, az *empty* a tárolóban lévő rekeszek számát tartalmazza, a *mutex* pedig 1. Az olyan szemaforokat, amelyeknek kezdőértéke 1, és arra szolgálnak, hogy biztosítsák, hogy kettő vagy több processzus közül egy időben csak egyikük léphessen be a kritikus szekciójába, **bináris szemaforok**nak nevezzük. Ha minden processzus pontosan azelőtt hajt végre egy down-t, mielőtt belép a kritikus szekciójába, és pontosan azután egy up-ot, miután kilép onnan, a kölcsönös kizárás biztosítva van.

Most, hogy rendelkezésünkre áll egy jó processzusok közötti kommunikációs primitív, térjünk ismét vissza a 2.5. ábrához, és vessünk egy pillantást a megszakítási tevékenységsorra. Egy szemaforokat használó rendszerben a megszakítások elrejtésének természetes módja, hogy egy 0 kezdőértékű szemafort rendelünk minden I/O-eszközhöz. Közvetlenül az I/O-eszköz indulása után a kezelő processzus végrehajt egy down-t a hozzárendelt szemaforon, így azonnal blokkolva magát. Amikor a megszakítás megérkezik, a megszakításkezelő végrehajt egy up-ot a hozzárendelt szemaforon, amely a megfelelő processzust újra futáskésszé teszi. Ebben a modellben a 2.5. ábra 6. lépése tartalmaz egy up-ot, amelyet az eszköz szemaforán hajtunk végre, és így a 7. lépésben az ütemező képes futtatni az eszközkezelőt. Természetesen, ha több processzus van futáskész állapotban, akkor az ütemező kiválaszthatja futásra a legfontosabb processzust. A fejezet későbbi részében azt is megnézzük, hogyan dolgozik az ütemező.


```
#define N 100                       /* a rekeszek száma a tárolóban */
typedef int semaphore;              /* a szemafor az int speciális fajtája */
semaphore mutex = 1;                /* felügyeli a kritikus szekció elérését */
semaphore empty = N;                /* a tároló üres rekeszeinek a száma */
semaphore full = 0;                 /* a tároló tele rekeszeinek a száma */

void producer(void)
{
   int item;

   while (TRUE) {                   /* a TRUE az 1 konstans */
      item = produce_item();        /* létrehoz valamit, amit a tárolóba lehet tenni */
      down(&empty);                 /* üres rekeszek száma csökken */
      down(&mutex);                 /* belépés a kritikus szekcióba */
      insert_item(item);            /* betesszük az új elemet a tárolóba */
      up(&mutex);                   /* elhagyjuk a kritikus szekciót */
      up(&full);                    /* tele rekeszek száma növekszik */
   }
}

void consumer(void)
{
   int item;

   while (TRUE) {                   /* végtelen ciklus */
      down(&full);                  /* tele rekeszek száma csökken */
      down(&mutex);                 /* belépés a kritikus szekcióba */
      item = remove_item();         /* kiveszünk egy elemet a tárolóból */
      up(&mutex);                   /* elhagyjuk a kritikus szekciót */
      up(&empty);                   /* üres rekeszek száma növekszik */
      consume_item(item);           /* csinálunk valamit az elemmel */
   }
}
```

2.14. ábra. *A gyártó-fogyasztó probléma szemaforok felhasználásával*

A 2.14. ábra példájában a szemaforokat valójában kétféle módon használjuk. A különbség elég fontos ahhoz, hogy jobban megvilágítsuk. A *mutex* szemafort a kölcsönös kizárásra használjuk. Ez biztosítja, hogy egy időben csak egy processzus olvassa vagy írja a tárolót és a hozzá kapcsolódó változókat. Ez a kölcsönös kizárás szükséges, hogy megelőzzük a káoszt. A kölcsönös kizárást és megvalósításának módját a következő alfejezetben tárgyaljuk bővebben.

A szemaforok másik felhasználási területe a **szinkronizáció**. A *full* és az *empty* szemaforok azért kellenek, hogy biztosítsák, hogy bizonyos eseménysorozatok bekövetkezzenek és bizonyosak ne. A mi esetünkben biztosítják, hogy a gyártó megállítsa a futását, ha a tároló tele van és a fogyasztó megállítsa a futását, ha a tároló üres. Ez a használat különbözik a kölcsönös kizárástól.

2.2.6. Mutexek

Amikor a szemafor számlálási képességére nincs szükség, akkor a szemafor egy egyszerűsített változata, a mutex kerülhet felhasználásra. A mutexek csak bizonyos erőforrások vagy kódrészek kölcsönös kizárásának kezelésére alkalmasak. Megvalósításuk könnyű és hatékony, ami miatt különösen hasznosak a teljes mértékben felhasználói szinten megvalósított szál (thread) csomagok számára.

A **mutex** egy olyan változó, amely kétféle állapotban lehet: nem zárolt vagy zárolt. Ennek következtében egyetlen bit is elegendő a reprezentálásához, de a gyakorlatban gyakran egy egész értéket használnak, ahol 0 jelenti a nem zárolt, és bármilyen más érték a zárolt állapotot. Két eljárás használatos a mutexek esetében. Amikor egy processzus (vagy szál) hozzá szeretne férni a kritikus szekcióhoz, meghívja a *mutex_lock* eljárást. Ha a mutex pillanatnyilag nem zárolt (ami azt jelenti, hogy a kritikus szekció elérhető), akkor a hívás sikeres, és a hívó szál szabadon beléphet a kritikus szekcióba.

Másrészről, ha a mutex már zárolt állapotban van, akkor a hívó blokkolódik, amíg a kritikus szekcióban lévő processzus nem végez, és meg nem hívja a *mutex_unlock* eljárást. Ekkor ha több processzus is blokkolódik a mutexen, közülük az egyik véletlenszerűen kiválasztott szerezheti meg a zárolást.

2.2.7. Monitorok

A szemaforokkal a processzusok kommunikációja könnyűnek tűnik, igaz? Felejtsük el. Lássuk közelebbről a 2.14. ábrán a down-ok sorrendjét, mielőtt beteszünk vagy kiveszünk egy elemet a tárolóból. Tegyük fel, hogy a két down-t a gyártó kódjában felcseréltük, vagyis a *mutex* csökkentése az *empty* előtt történik, és nem utána. Ha a tároló teljesen tele lenne, akkor a gyártó blokkolna, a *mutex* 0 lenne. Következésképpen a fogyasztó ezután megpróbálná elérni a tárolót, végrehajtana egy down-t a *mutex*-en, ami most 0, és szintén blokkolna. Mindkét processzus a végtelenségig blokkolt állapotban maradna, és soha sem folyna már semmilyen munka. Ezt a sajnálatos esetet **holtpont**nak hívják. A holtpontokkal a 3. fejezetben részletesen foglalkozunk.

Ez a probléma rámutatott arra, hogy óvatosnak kell lennünk, ha szemaforokat használunk. Egy szövevényes hiba, és minden egy nyomasztó megálláshoz vezet. Ez hasonló az assembly nyelvű programozáshoz, csak rosszabb, mert a hibák versenyhelyzetek, holtpontok és más megjósolhatatlan és reprodukálhatatlan viselkedési formák.

Hogy megkönnyítsék a helyes programok írását, Brinch Hansen (Hansen, 1973) és Hoare (Hoare, 1974) egy magasabb szintű szinkronizációs primitívet javasoltak, amelyet **monitor**nak neveztek el. A javaslataik kicsit különböztek egymástól, mint látni fogjuk. A monitor eljárások, változók és adatszerkezetek együttese, és mindezek egy speciális fajta modulba vagy csomagba vannak összegyűjtve. A processzusok bármikor hívhatják a monitorban lévő eljárásokat, de nem érhetik el közvetlenül a monitor belső adatszerkezeteit a monitoron kívül deklarált eljárásokból.


```
monitor example
   integer i;
   condition c;

   procedure producer(x);
   .
   .
   .
   end;

   procedure consumer(x);
   .
   .
   .
   end;
end monitor;
```

2.15. ábra. *Egy monitor*

Ez a szabály, amely általánosan használt a modern objektumorientált nyelvekben, amilyen a Java is, meglehetősen szokatlan volt a maga idejében, annak ellenére, hogy az objektumokat a Simula 67-ig vezethetjük vissza. A 2.15. ábra egy monitort mutat be, amelyet egy elképzelt nyelven, a Pidgin Pascal nyelven írtak.

A monitoroknak van egy kulcsfontosságú tulajdonsága, ez teszi használhatóvá a kölcsönös kizárás megvalósítására: minden időpillanatban csak egy processzus lehet aktív egy monitorban. A monitorok programozási nyelvi konstrukciók, ezért a fordítóprogram tudja, hogy ezek speciálisak, és képes a monitoreljárás-hívásokat másképpen kezelni, mint az egyéb eljáráshívásokat. Jellemzően, amikor egy processzus meghív egy monitoreljárást, akkor az eljárás első néhány utasítása ellenőrizni fogja, hogy más processzus jelenleg aktív-e a monitoron belül. Ha igen, akkor a hívó processzus felfüggesztésre kerül, amíg a másik processzus el nem hagyja a monitort. Ha nem használja másik processzus a monitort, akkor a hívó processzus beléphet.

A fordítóprogramtól függ, hogy hogyan valósítja meg a kölcsönös kizárást a monitor belépési pontjainál, de egy általános módszer a mutexek vagy bináris szemaforok használata. Mivel a fordítóprogram, és nem a programozó intézi el a kölcsönös kizárást, kisebb a valószínűsége, hogy valami elromlik. A monitort író személynek sosem kell tudnia, hogy hogyan intézi el a fordítóprogram a kölcsönös kizárást. Elegendő annyit tudni, hogy minden kritikus szekciót monitoreljárássá alakítva soha nem fogja két processzus egy időben a saját kritikus szekcióját végrehajtani.

Bár a monitorok egy könnyű módszert kínálnak a kölcsönös kizárás eléréséhez, ez nem elegendő, mint már fentebb láttuk. Szükségünk van egy olyan módszerre is, amellyel egy processzust blokkolhatunk, ha nem tud továbbhaladni. A gyártó-fogyasztó problémában elég könnyű az összes tároló-tele, tároló-üres vizsgálatokat monitoreljárásokra átalakítani, de hogyan kell blokkolni a gyártót, ha úgy találja, hogy a tároló tele van?

```
monitor ProducerConsumer
  condition full, empty;
  integer count;

  procedure insert(item: integer);
  begin
    if count = N then wait(full);
    insert_item(item);
    count := count + 1;
    if count = 1 then signal(empty)
  end;

  function remove: integer;
  begin
    if count = 0 then wait(empty);
    remove = remove_item;
    count := count - 1;
    if count = N - 1 then signal(full)
  end;

  count := 0;
end monitor;

procedure producer;
begin
  while true do
  begin
    item = produce_item;
    ProducerConsumer.insert(item)
  end
end;

procedure consumer;
begin
  while true do
  begin
    item = ProducerConsumer.remove;
    consume_item(item)
  end
end;
```

2.16. ábra. *A gyártó-fogyasztó probléma vázlata monitorokkal. Csak egy monitoreljárás aktív egy időben. A tárolónak N rekesze van*

A megoldás az **állapotváltozók** bevezetésében rejlik, két, rajtuk végezhető művelettel, a wait-tel és a signal-lal. Amikor egy monitoreljárás rájön, hogy nem tud tovább dolgozni (például a gyártó megállapítja, hogy a tároló tele van), végez egy wait-et egy állapotváltozón, mondjuk a *full*-on. Ez a tevékenység a hívó processzus blokkolását okozza. Ez azt is megengedi, hogy más, előzőleg a monitorba való belépéstől eltiltott processzus most belépjen.

Ez a másik processzus, például a fogyasztó, felébreszthetí alvó partnerét egy signal végrehajtásával azon az állapotváltozón, amelyre a partnere éppen vár. Annak megakadályozására, hogy két processzusunk legyen egy időben aktív a monitorban, szükségünk van egy szabályra, amely megmondja, hogy mi történjen a signal után. Hoare azt javasolta, hogy az újonnan felébresztett processzust hagyjuk futni, felfüggesztve a másikat. Brinch Hansen azt javasolta, hogy oldjuk meg a problémát azzal, hogy megköveteljük a signal-t végrehajtó processzustól, hogy *azonnal lépjen ki* a monitorból. Más szavakkal, a signal utasítás csak a monitoreljárás utolsó utasításaként fordulhat elő. Mi Brinch Hansen javaslatát fogjuk használni, mert ez koncepciójában egyszerűbb és megvalósítani is könnyebb. Ha egy signal lett végrehajtva egy olyan állapotváltozón, amelyre több processzus vár, ezek közül csak egy, az ütemező által meghatározott ébred fel.

Létezik egy harmadik megoldás is, amit sem Hoare, sem Brinch Hansen nem említ. Ez az lenne, hogy hagyjuk futni a szignált küldő processzust, és a várakozó processzus akkor léphessen be, ha a szignált küldő kilépett a monitorból.

Az állapotváltozók nem számlálók. Nem gyűjtenek össze szignálokat későbbi felhasználásra, mint ahogy azt a szemaforok teszik. Így, ha egy olyan állapotváltozó kap egy szignált, amelyre nem vár senki, akkor a szignál elvész. Más szavakkal, a wait-nek a signal előtt kell jönnie. Ez a szabály a megvalósítást egyszerűbbé teszi. A gyakorlatban ez nem probléma, mert változókkal könnyű nyomon követni minden processzus állapotát, ha szükséges. Egy processzus, amely egyébként végrehajtana egy signal-t, a változók vizsgálatával láthatja, hogy ez a művelet nem szükséges.

A 2.16. ábrán Pidgin Pascalban adjuk meg a gyártó-fogyasztó probléma vázlatát monitorokkal. A Pidgin Pascal használatának előnye itt az, hogy letisztult, egyszerű és pontosan követi a Hoare/Brinch Hansen-modellt.

Lehet, hogy az olvasóban felmerül, hogy a wait és signal műveleteknek is, hasonlóan a már korábban bemutatott sleep és wakeup műveletekhez, van végzetes versenyhelyzete. Tényleg nagyon *hasonlók,* de van egy döntő különbség: a sleep és wakeup használata azért fullad kudarcba, mert mialatt egy processzus próbál elaludni, egy másik próbálja őt felébreszteni. Monitorokkal ez nem fordulhat elő. Az automatikus kölcsönös kizárás a monitoreljárásban garantálja, hogy ha, mondjuk, a gyártó a monitoreljárás belsejében felfedezi, hogy a tároló tele van, képes lesz befejezni a wait műveletet anélkül, hogy tartania kellene attól a lehetőségtől, hogy az ütemező átkapcsolhat a fogyasztóra éppen a wait befejezése előtt.

Habár a Pidgin Pascal csak egy képzeletbeli nyelv, néhány valódi programozási nyelv is támogatja a monitorok használatát, még ha nem is mindig a Hoare és Brinch Hansen által megtervezett formában. Az egyik ilyen nyelv a Java. A Java objektumorientált nyelv, amely támogatja a felhasználói szintű szálakat, és megengedi a metódusok (eljárások) osztályokba szervezését. Egy eljárás deklarálásakor a synchronized kulcsszó megadásával a Java garantálja, hogy amint egy szál elkezdi végrehajtani az eljárást, egyetlen más szálnak sem engedi az osztály egyetlen másik synchronized-del megjelölt eljárásának megkezdését sem.

A Java szinkronizált eljárásai lényegesen különböznek a klasszikus monitoroktól: a Java nem rendelkezik állapotváltozókkal. Ehelyett két eljárást biztosít,

wait-et és a *notify*-t, amelyek megfelelnek a *sleep*-nek és a *wakeup*-nak, azzal a különbséggel, hogy ha szinkronizált eljárásokon belül használjuk őket, akkor nincsenek kitéve versenyhelyzeteknek.

Azzal, hogy a monitorok a kritikus szekciók kölcsönös kizárását automatikussá tették, a párhuzamos programozás kevésbé lett hibára hajlamos, mint a szemaforokkal. De még ezeknek is van néhány hátrányuk. Nem véletlen, hogy a 2.16. ábrát Pidgin Pascalban írtuk, és nem C-ben, ahogy a könyv többi példáját. Mint már előbb említettük, a monitor egy programozási nyelvi fogalom. A fordítóprogramnak kell ezeket felismernie és valahogy elintéznie a kölcsönös kizárást. A C, a Pascal és a legtöbb más programozási nyelvnek nincsenek monitorjai, tehát nem ésszerű elvárni a fordítóprogramjaiktól, hogy betartsanak valamilyen kölcsönös kizárási szabályt. Valójában mégis honnan tudná a fordítóprogram, hogy mely eljárások vannak a monitorban, és melyek nem?

Az ilyen nyelveknek szemaforuk sincs, bár szemaforokat hozzáadni könnyű: mindössze annyit kell tenni, hogy két kis assembly nyelvű rutint kell a könyvtárhoz hozzávenni, hogy kiadhassuk az up és down rendszerhívásokat. A fordítóprogramoknak még csak azt sem kell tudniuk, hogy ezek léteznek. Természetesen az operációs rendszernek tudnia kell a szemaforokról, de végül is, ha van egy szemaforalapú operációs rendszerünk, akkor már írhatunk hozzá felhasználói programokat C-ben vagy C++-ban (vagy akár FORTRAN-ban is, ha eléggé mazochisták vagyunk). A monitorok esetén azonban szükségünk van egy programozási nyelvre, amelybe ezek be vannak építve.

A másik probléma a monitorokkal és a szemaforokkal is az, hogy a kölcsönös kizárás problémájának megoldására tervezték ezeket egy vagy több CPU-ra, amelyek mindegyike elérheti a közös memóriát. A szemaforokat megosztott memóriába helyezve és megvédve azokat TSL utasításokkal, elkerülhetjük a versenyt. Amikor egy olyan osztott rendszerre térünk át, amelyik több, saját memóriával rendelkező CPU-ból áll, amelyek egy lokális hálózattal vannak összekötve, akkor ezek a primitívek alkalmazhatatlanokká válnak. A következtetés az, hogy a szemaforok túl alacsony szintűek, és a monitorok néhány programozási nyelvet kivéve nem használhatók. Ráadásul egyik primitív se szolgál a gépek közötti információcserére. Valami másra van szükségünk.

2.2.8. Üzenetküldés

Ez a valami más az **üzenetküldés**. A processzusok kommunikációjának ezen módszere két primitívet használ, a send-et és a receive-et, amelyek a szemaforokhoz hasonlóan – és nem úgy, mint a monitorok – inkább rendszerhívások, mint nyelvi konstrukciók. Mint ilyenek, könnyen beilleszthetők a könyvtári eljárások közé a következőképpen:

```
send(destination, &message);
```

és


```
receive(source, &message);
```

Az előbbi hívás egy üzenetet küld a célállomáshoz, az utóbbi egy üzenetet fogad egy adott forrástól (vagy *ANY*-től, ha a fogadót nem érdekli a forrás). Ha nincs elérhető üzenet, akkor a fogadó blokkolhatna, míg egy megérkezik. Vagy pedig azonnal visszatérhetne egy hibakóddal.

Tervezési szempontok az üzenetküldő rendszereknél

Az üzenetküldő rendszereknél sok kihívást jelentő probléma és tervezési szempont merül fel, amely nem került elő a szemaforoknál vagy monitoroknál, különösen ha a kommunikáló processzusok a hálózat különböző gépein vannak. Például az üzenetek el tudnak veszni a hálózaton. Hogy az üzenetek elvesztése ellen védekezzenek, a küldő és a fogadó megegyezhet, hogy amint egy üzenet megérkezik, a fogadó visszaküld egy speciális **nyugtázó** üzenetet. Ha a küldő nem kapja meg a nyugtát egy bizonyos időintervallumon belül, akkor újra elküldi az üzenetet.

Most gondoljuk át, mi történik akkor, ha maga az üzenet korrekten megérkezik, de a nyugta elvész. A küldő újból elküldi az üzenetet, így azt a fogadó kétszer kapja meg. Alapvető, hogy a fogadó meg tudjon különböztetni egy új üzenetet egy újraküldött régitől. Általában ez a probléma megoldódik, ha egymás utáni sorszámokkal látunk el minden eredeti üzenetet. Ha a fogadó egy olyan üzenetet kap, amelynek a sorszáma megegyezik az előzőével, akkor tudni fogja, hogy az üzenet ismétlés, amelyet figyelmen kívül hagyhat.

Az üzenetküldő rendszereknek azzal a kérdéssel is foglalkozniuk kell, hogy mi a neve a processzusoknak, azért, hogy a send és receive hívásban specifikált processzszus egyértelmű legyen. A **hitelesítés** szintén téma az üzenetküldő rendszerekben: hogyan tudja az ügyfél megmondani, hogy a valódi fájlszerverrel kommunikál, és nem egy szélhámossal?

A dolgok másik oldaláról nézve akkor is felmerülnek fontos tervezési kérdések, amikor a küldő és a fogadó ugyanazon a gépen vannak. Ezek egyike a hatékonyság. Mindig lassúbb egy processzustól egy másikhoz üzenetet másolni, mint egy szemafor műveletet elvégezni, vagy belépni egy monitorba. Rengeteget dolgoztak azon, hogy az üzenetküldést hatékonnyá tegyék. Cheriton (Cheriton, 1984) például azt javasolta, hogy korlátozzuk le az üzenetméretet akkorára, ami megfelel a gép regiszterének, és utána az üzenetek küldésére használjuk a regisztereket.

A gyártó-fogyasztó probléma üzenetküldéssel

Most nézzük meg, hogyan lehet megoldani a gyártó-fogyasztó problémát üzenetküldéssel, és nem megosztott memóriával. Ezt mutatja a 2.17. ábra. Feltételezzük, hogy minden üzenet egyforma hosszú, és ezeket a már elküldött, de még meg nem kapott üzeneteket az operációs rendszer automatikusan egy tárolóba teszi. Ennél a megoldásnál összesen N üzenetet használunk, hasonlóan a megosz-

```
#define N 100                                   /* a rekeszek száma a tárolóban */

void producer(void)
{
   int item;
   message m;                                   /* üzenetek tárolója */

   while (TRUE) {
      item = produce_item();                    /* létrehoz valamit, amit a tárolóba lehet tenni */
      receive(consumer, &m);                    /* várjuk, hogy egy üres megérkezzen */
      build_message(&m, item);                  /* elküldendő üzenet összeállítása */
      send(consumer, &m);                       /* küldünk egy elemet a fogyasztónak */
   }
}

void consumer(void)
{
   int item, i;
   message m;

   for (i = 0; i < N; i++) send(producer, &m);/* N üres elem elküldése */
   while (TRUE) {
      receive(producer, &m);                    /* fogadjuk az üzenetet, amiben az elem van */
      item = extract_item(&m);                  /* kibontjuk az elemet az üzenetből */
      send(producer, &m);                       /* visszaküldjük az üres elemet */
      consume_item(item);                       /* csinálunk valamit az elemmel */
   }
}
```

2.17. ábra. *A gyártó-fogyasztó probléma* N *üzenettel*

tott memóriában lévő tároló N rekeszéhez. A fogyasztó azzal kezdi, hogy elküld N üres üzenetet a gyártónak. Valahányszor a gyártónak van egy fogyasztóhoz küldendő eleme, vesz egy üres üzenetet, és visszaküld egy telit. Ily módon a rendszerben lévő üzenetek száma időben konstans marad, így azokat egy előre megadott méretű memóriaterületen lehet tárolni.

Ha a gyártó gyorsabban dolgozik, mint a fogyasztó, minden üzenet megtelik a fogyasztóra várva; a gyártó blokkolódik, arra várva, hogy egy üres jöjjön vissza. Ha a fogyasztó dolgozik gyorsabban, akkor az ellenkezője történik: minden üzenet üres lesz arra várva, hogy a gyártó feltöltse; ekkor a fogyasztó lesz blokkolva egy teli üzenetre várva.

Sok változat lehetséges üzenetek küldésére. A kezdők kedvéért nézzük meg, hogyan címezzük az üzeneteket. Egyik módszer az, hogy minden processzushoz hozzárendelünk egy egyedi címet, és az üzeneteket a processzusokhoz kell címezni. Egy másik módszer, hogy egy új adatszerkezetet találunk ki, amelyet **levelesládának** nevezünk. A levelesláda néhány, általában a levelesláda létrehozásakor specifikált számú üzenet ideiglenes tárolására szolgál. Amikor levelesládákat

használunk, a send és receive hívásokban a címparaméterek levelesládák és nem processzusok. Ha egy processzus olyan levelesládának próbál üzenetet küldeni, amely tele van, akkor addig felfüggesztődik, amíg abból a levelesládából ki nem vesznek egy üzenetet, ezáltal helyet biztosítva az újnak.

A gyártó-fogyasztó problémánál mind a gyártó, mind a fogyasztó N üzenet tárolására elegendő nagy levelesládát hozhat létre. A gyártó adatokat tartalmazó üzeneteket fog küldeni a fogyasztó levelesládájába, és a fogyasztó üres üzeneteket a gyártó levelesládájába. Ha levelesládákat használunk, az ideiglenes tárolási mechanizmus világos: a célállomás levelesládájában vannak mindazok az üzenetek, amelyeket már elküldtek neki, de még nem fogadta azokat.

A levelesládával kapcsolatos másik szélsőség az összes ideiglenes tárolás elhagyása. Amikor ezt a megközelítést követjük, és a receive előtt a send-et hajtjuk végre, akkor a küldő processzus blokkolódik, amíg a receive végrehajtódik, amikor is az üzenet közvetlenül, közbülső tárolás nélkül másolódhat a küldőtől a fogadóhoz. Hasonlóan, ha a receive hajtódik végre először, akkor a fogadó blokkolódik, amíg a send végrehajtódik. Ezt a stratégiát gyakran **randevú**nak hívják. Könnyebb megvalósítani, mint egy ideiglenesen tárolt üzenet tervét, de kevésbé rugalmas, mivel a küldő és a fogadó kénytelen szorosan egymáshoz igazodva futni.

A MINIX 3 operációs rendszert alkotó processzusok a randevúeljárást használják rögzített méretű üzenetekkel az egymással való kommunikációra. A felhasználói processzusok is ezt a módszert használják, amikor az operációs rendszer komponenseivel kommunikálnak, bár a programozó ezt nem látja, mivel könyvtári eljárásokon keresztül történnek a rendszerhívások. A felhasználói processzusok kommunikációja a MINIX 3-ban (és Unixban) adatcsöveken keresztül valósul meg, amelyek ténylegesen levelesládák. Az egyetlen valódi különbség a levelesládákkal történő üzenetküldés és az adatcső mechanizmusa között az, hogy az adatcsövek nem őrzik meg az üzenetek határait. Más szavakkal, ha egy processzus 10 darab 100 bájtos üzenetet ír egy adatcsőbe, és egy másik processzus 1000 bájtot olvas ebből az adatcsőből, akkor az olvasó egyszerre fogja megkapni mind a 10 üzenetet. Egy igazi üzenetküldő rendszerben minden read-nek csak egy üzenettel kellene visszatérnie. Természetesen, ha a processzusok megegyeznek abban, hogy az adatcsőből mindig azonos méretű üzeneteket olvasnak és írnak, vagy minden üzenet végét speciális karakterrel (például soremeléssel) zárják, akkor nem merül fel cz a probléma.

Az üzenetküldés általánosan használt technika párhuzamos programozású rendszerekben. Egy jól ismert üzenetküldő rendszer például az **MPI (Message-Passing Interface)**. Széles körben használják tudományos számításokhoz. Erről részletesebben lásd (Gropp et al., 1984; Snir et al., 1996) írtak.

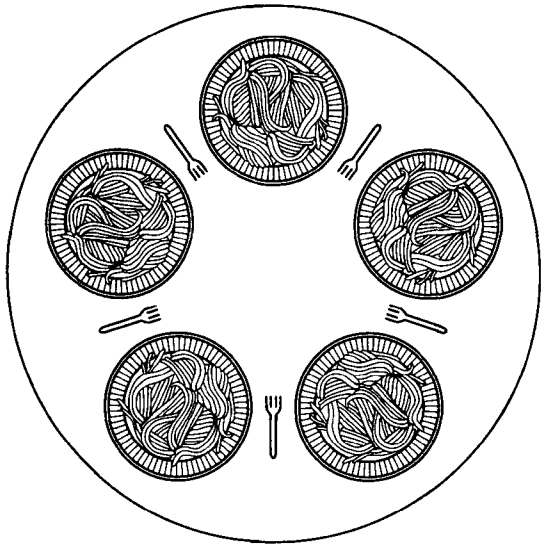
2.3. Klasszikus IPC-problémák

Az operációs rendszerek irodalma tele van olyan processzusok közötti kommunikációs problémákkal, amelyeket különféle szinkronizációs módszerek felhasználásával alaposan kielemeztek. A következő részekben két jól ismert problémát vizsgálunk meg.

2.3.1. Az étkező filozófusok probléma

1965-ben Dijkstra felvetett és megoldott egy szinkronizációs problémát, amelyet **étkező filozófusok problémának** nevezett el. Ettől kezdve mindenki, aki kitalált egy új szinkronizációs primitívet, ellenállhatatlan vágyat érzett arra, hogy az új primitív csodálatos voltát azzal bizonyítsa, hogy felhasználásával az étkező filozófusok problémára elegáns megoldást ad. A probléma egyszerűen a következő. Öt filozófus ül egy kerek asztal körül. Mindegyik filozófusnak van egy tányér spagettije. A spagetti olyan csúszós, hogy egy filozófusnak két villára van szüksége az evéshez. Minden egymás melletti két tányér között van egy villa. A 2.18. ábrán látjuk az asztal elrendezését.

A filozófusok élete egymást váltogató evési és gondolkodási periódusokból áll. (Ez most absztrakció, még akkor is, ha filozófusokról van szó. Nem fontos, hogy milyen egyéb tevékenységeket végeznek még.) Amikor egy filozófus éhes lesz, valamilyen sorrendben megpróbálja megszerezni a bal és jobb oldalán lévő villát is. Ha sikerült mindkét villát megszereznie, akkor eszik egy ideig, majd leteszi a villákat, és gondolkodással folytatja. A kulcskérdés az: tudunk-e olyan programot



2.18. ábra. *Ebédidő a filozófia tanszéken*


```
#define N 5                         /* a filozófusok száma */

void philosopher(int i)             /* i: a filozófus sorszáma, 0-tól 4-ig */
{
   while (TRUE) {
      think();                      /* a filozófus gondolkodik */
      take_fork(i);                 /* felveszi a bal villát */
      take_fork((i + 1) % N);       /* felveszi a jobb villát; % a moduló osztás művelet */
      eat();                        /* nyam-nyam, spagetti */
      put_fork(i);                  /* visszateszi az asztalra a bal villát */
      put_fork((i + 1) % N);        /* visszateszi az asztalra a jobb villát */
   }
}
```

2.19. ábra. *Az étkező filozófusok probléma egy hibás megoldása*

készíteni a filozófusok számára, amely az elvárások szerint működik, és soha nem akad el? (Egyesek szerint a két villa követelménye kicsit mesterkélt, lehet, hogy át kellene térni olaszról kínai ételre, rizzsel helyettesítve a spagettit és evőpálcikával a villát.)

A 2.19. ábra mutatja a nyilvánvaló megoldást. A *take_fork* eljárás megvárja, hogy a megadott villa elérhető legyen, és ekkor megszerzi. Sajnos a nyilvánvaló megoldás hibás. Tegyük fel, hogy mind az öt filozófus egyszerre szerzi meg a bal oldali villáját. Egyik se lesz képes a jobb oldalit megszerezni, és holtpont alakul ki.

Módosíthatjuk a programot úgy, hogy a program ellenőrizze, vajon a bal oldali villa megszerzése után a jobb oldali villa elérhető-e. Ha nem, akkor a filozófus tegye le a bal oldali villát, várjon egy kicsit, majd ismételje meg a teljes eljárást. Ez a javaslat sem vezet eredményre, bár más okból. Egy kis balszerencsével minden filozófus egyszerre kezdi az algoritmust végrehajtani, felveszi a bal oldali villáját, látja, hogy a jobb oldali villája nem elérhető, leteszi a bal oldali villáját, vár, majd megint felveszi a bal oldali villáját a többiekkel egy időben, és így tovább a végtelenségig. Az ilyen helyzetet, amelyben minden program korlátlan ideig folytatja a futást, de érdemben nem halad előre, **éhezés**nek nevezzük. (Annak ellenére éhezésnek nevezzük, hogy a probléma nem fordul elő egy olasz vagy kínai étteremben.)

Most azt gondolhatjuk, hogy ha a filozófusok véletlen ideig, és nem ugyanolyan hosszú ideig várakoznának a jobb oldali villa megszerzésének kísérlete után, akkor nagyon kicsi lenne az esélye annak, hogy az események ugyanabban az ütemben folytatódjanak egy órán keresztül. Ez az észrevétel helyes, és az alkalmazások többségében egy későbbi időpontban történő újrapróbálkozás nem is okoz gondot. Például egy Ethernetet használó helyi hálózatban a gépek csak akkor kezdenek csomagküldésbe, ha azt érzékelik, hogy éppen senki más nem küld. Mégis előfordulhat, hogy a vezeték távolabbi pontjain elhelyezkedő két gép a jelterjedési késleltetések miatt időben átfedve küldi a csomagokat – ezt nevezzük ütközésnek. Az ütközés felismerése után mindkét gép véletlen ideig vár, majd újra próbálkozik; a gyakorlatban ez a megoldás remekül bevált.

```
#define N                5                  /* a filozófusok száma */
#define LEFT             (i + N – 1) % N    /* az i bal szomszédjának a sorszáma */
#define RIGHT            (i + 1) % N        /* az i jobb szomszédjának a sorszáma */
#define THINKING         0                  /* a filozófus gondolkodik */
#define HUNGRY           1                  /* a filozófus megpróbál villát szerezni */
#define EATING           2                  /* a filozófus eszik */

typedef int semaphore;                      /* a szemafor az int speciális fajtája */
int state[N];                               /* tömb az állapotok nyomon követésére */
semaphore mutex = 1;                        /* a kritikus szekciók kölcsönös kizárásához */
semaphore s[N];                             /* filozófusonként egy szemafor */

void philosopher(int i)                     /* i: a filozófus sorszáma, 0-tól N – 1-ig */
{
   while (TRUE) {                           /* végtelen ciklus */
      think();                              /* a filozófus gondolkodik */
      take_forks(i);                        /* megszerzi mindkét villát, vagy blokkol */
      eat();                                /* nyam-nyam, spagetti */
      put_forks(i);                         /* mindkét villát visszateszi az asztalra */
   }
}
void take_forks(int i)                      /* i: a filozófus sorszáma, 0-tól N – 1-ig */
{
   down(&mutex);                            /* belépés a kritikus szekcióba*/
   state[i] = HUNGRY;                       /* rögzítjük, hogy az i filozófus éhes */
   test(i);                                 /* megpróbál 2 villát szerezni */
   up(&mutex);                              /* kilépés a kritikus szekcióból */
   down(&s[i]);                             /* blokkol, ha nem tudott villát szerezni */
}
void put_forks(i)                           /* i: a filozófus sorszáma, 0-tól N – 1-ig */
{
   down(&mutex);                            /* belépés a kritikus szekcióba */
   state[i] = THINKING;                     /* a filozófus befejezte az evést */
   test(LEFT);                              /* megnézi, hogy a bal szomszéd tud-e most enni */
   test(RIGHT);                             /* megnézi, hogy a jobb szomszéd tud-e most enni */
   up(&mutex);                              /* kilépés a kritikus szekcióból */
}

void test(i)                                /* i: a filozófus sorszáma, 0-tól N – 1-ig */
{
   if (state[i] = = HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
      state[i] = EATING;
      up(&s[i]);
   }
}
```

2.20. ábra. *Az étkező filozófusok probléma egyik megoldása*

Vannak azonban olyan alkalmazások, amikor előnyben részesítjük azokat a megoldásokat, amelyek mindig működnek, és még véletlen számok valószínűtlen sorozatának előfordulásakor sem hibázhatnak. Gondoljunk csak egy atomerőmű biztonsági berendezéseinek vezérlésére.

A 2.19. ábra programjának holtpont és éhezés nélküli javítása lehet az, ha egy bináris szemaforral *(mutex)* megvédjük a *think* hívást követő öt utasítást. Mielőtt egy filozófus megkezdi a villák megszerzését, egy down-t kellene végrehajtania a *mutex*-en. Miután a villákat visszatette, egy up-ot kellene végrehajtania a *mutex*-en. Elméleti szempontból ez a megoldás kielégítő. Gyakorlatban azonban van egy hatékonysági hibája: csak egy filozófus tud enni egy adott pillanatban. Mivel öt villánk van, meg kellene tudnunk engedni, hogy két filozófus egy időben egyen.

A 2.20. ábrán bemutatott megoldás holtpontmentes, és megengedi a maximális párhuzamosságot tetszőleges számú filozófus esetén. Használ egy *state* tömböt, hogy nyomon kövesse, hogy egy filozófus eszik, gondolkodik vagy éhes (próbálja megszerezni a villákat). Egy filozófus csak akkor mehet át evés állapotba, ha egyik szomszédja sem eszik. Az *i* filozófus szomszédait a *LEFT* és *RIGHT* makrók definiálják. Más szavakkal, ha *i* értéke 2, akkor *LEFT* 1 és *RIGHT* 3.

A program egy tömböt használ, amelyben filozófusonként egy szemafor van, így az éhes filozófusok blokkolódhatnak, ha a szükséges villák foglaltak. Megjegyezzük, hogy minden processzus a *philosopher* eljárást saját főprogramjaként futtatja, de más eljárások, mint a *take_forks, put_forks* és *test* közönséges eljárások és nem különálló processzusok.

2.3.2. Az olvasók és írók probléma

Az étkező filozófusok probléma hasznos, ha olyan processzusokat modellezünk, amelyek korlátozott számú erőforrásért, például I/O-eszközök kizárólagos eléréséért versenyeznek. Másik híres probléma az olvasók és írók probléma, amely egy adatbázis elérését modellezi (Courtois et al., 1971). Képzeljük el például egy légitársaság helyfoglalási rendszerét sok versengő processzussal, amelyek az adatbázist olvasni és írni szeretnék. Elfogadható, hogy több processzus egyidejűleg olvasson az adatbázisból, de ha egy processzus aktualizálja (írja) az adatbázist, akkor azt más processzusoknak nem szabad elérniük, még az olvasóknak sem. A kérdés az, hogy hogyan programozzuk az olvasókat és az írókat. A 2.21. ábrán láthatunk egy megoldást.

Ebben a megoldásban az első olvasó, aki hozzáfér az adatbázishoz, végrehajt egy down-t a *db* szemaforon. A következő olvasók csupán az *rc* számlálót növelik. Ha egy olvasó kilép, akkor csökkenti a számlálót, és az utolsó kilépő végrehajt egy up-ot a szemaforon, lehetővé téve egy blokkolt írónak, ha van ilyen, hogy belépjen.

Az itt bemutatott megoldásban van egy kis apróság, amire érdemes kitérni. Tegyük fel, hogy mialatt egy olvasó használja az adatbázist, egy másik olvasó érkezik. Mivel nem probléma, ha két olvasó van egy időben, ezért bebocsátják. A harmadik és további olvasókat is bebocsáthatják, ha érkeznek.

```
typedef int semaphore;          /* használjuk a fantáziánkat */
semaphore mutex = 1;            /* 'rc' elérését vezérli */
semaphore db = 1;               /* az adatbázis elérését vezérli */
int rc = 0;                     /* az olvasó vagy ezt akaró processzusok száma */

void reader(void)
{
   while (TRUE) {               /* végtelen ciklus */
      down(&mutex);             /* kizárólagos elérés beállítása 'rc'-hez */
      rc = rc + 1;              /* eggyel több olvasó van */
      if (rc = = 1) down(&db);  /* ha ez az első olvasó... */
      up(&mutex);               /* kizárólagos elérés elengedése 'rc'-hez */
      read_data_base();         /* az adatok elérése */
      down(&mutex);             /* kizárólagos elérés beállítása 'rc'-hez */
      rc = rc - 1;              /* eggyel kevesebb olvasó van */
      if (rc = = 0) up(&db);    /* ha ez az utolsó olvasó... */
      up(&mutex);               /* kizárólagos elérés elengedése 'rc'-hez */
      use_data_read();          /* nemkritikus szekció */
   }
}

void writer(void)
{
   while (TRUE) {               /* végtelen ciklus */
      think_up_data();          /* nemkritikus szekció */
      down(&db);                /* kizárólagos elérés beállítása */
      write_data_base();        /* az adatok aktualizálása */
      up(&db);                  /* kizárólagos elérés elengedése */
   }
}
```

2.21. ábra. *Az olvasók és írók probléma egy megoldása*

Most tegyük fel, hogy egy író érkezik. Az írót nem engedhetik be az adatbázisba, mert az íróknak kizárólagos hozzáférésre van szükségük, így az író felfüggesztődik. Később további olvasók jelennek meg. Amíg van legalább egy aktív olvasó, további olvasók bejöhetnek. Ennek a stratégiának az a következménye, hogy ha az olvasóknak folyamatos utánpótlása van, akkor azok megérkezésük után azonnal bejuthatnak. Az író mindaddig felfüggesztett állapotban marad, amíg az olvasók el nem fogynak. Ha mondjuk 2 másodpercenként jön egy új olvasó, és minden olvasónak 5 másodperces munkája van, az író soha nem kerül be.

Ennek a helyzetnek az elkerülésére a programot írhatjuk egy kicsit másképpen: amikor egy olvasó megérkezik, és egy író már vár, az olvasó felfüggesztődik az író mögött, ahelyett hogy rögtön beengednénk. Így az írónak csak azt kell megvárnia, hogy az előtte érkezett olvasók végezzenek, de nem kell megvárnia azokat az olvasókat, akik utána érkeztek. Ennek a megoldásnak az a hátránya, hogy kevesebb

párhuzamosságot enged meg, és így a hatékonyság csökken. Courtois és társai bemutatnak egy olyan megoldást, amely az íróknak ad prioritást. A részleteket lásd (Courtois et al., 1971).

2.4. Ütemezés

Az előző részek példáiban gyakran kerültünk abba a helyzetbe, hogy két vagy több processzus (például gyártó és fogyasztó) volt logikailag futásra képes. Multiprogramozott számítógépben gyakran előfordul, hogy több processzus verseng a CPU-ért. Amikor több processzus képes futni, de csak egy processzor áll rendelkezésre, akkor az operációs rendszernek el kell döntenie, hogy mely fusson először. Az operációs rendszer azon részét, amelyik ezt a döntést meghozza, **ütemezőnek (scheduler)** nevezzük; az erre a célra használt algoritmus pedig az **ütemezési algoritmus**.

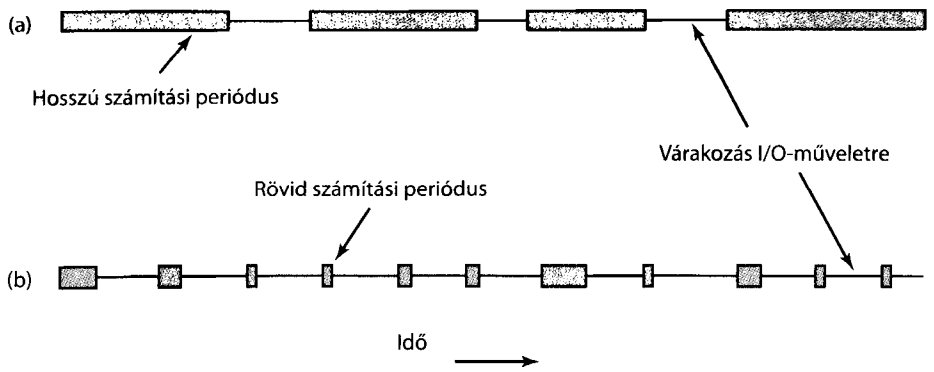
Az ütemezéssel kapcsolatos megfontolások nagy része egyaránt vonatkozik a processzusokra és a szálakra is. Először a processzusütemezéssel foglalkozunk, majd röviden áttekintjük a szálütemezéssel kapcsolatos speciális kérdéseket.

2.4.1. Bevezetés az ütemezésbe

A kötegelt rendszerek idejében, amikor a bemenő adatok mágnesszalagon voltak lyukkártya formátumban, az ütemezési algoritmus egyszerű volt: fusson a szalagon található következő feladat. Az időosztásos rendszerekben az ütemezési algoritmus bonyolultabb, hiszen gyakran több felhasználó vár a kiszolgálásra, és még kötegelt feladatsorok is lehetnek (például egy biztosítótársaságnál a követelések feldolgozására). Azt gondolhatnánk, hogy egy személyi számítógépen mindig csak egy aktív processzus van. Végül is nem túl valószínű, hogy a felhasználó szövegszerkesztés közben még programot is fordít a háttérben. Háttérfeladatok azonban gyakran előfordulnak, például az elektronikus leveleket kezelő háttérprocesszus küldhet vagy fogadhat e-maileket. Feltételezhetnénk azt is, hogy az utóbbi években a számítógépek annyival gyorsabbak lettek, hogy a CPU már olyan erőforrás lett, amelyből nincs hiány. Az új alkalmazások viszont rendszerint több erőforrást igényelnek. Példának felhozhatjuk a digitális fényképek feldolgozását vagy a valós idejű videolejátszást.

Processzusok viselkedése

Majdnem minden processzus váltogatva végez számításokat és I/O-műveleteket, ahogy a 2.22. ábrán látható. Az a tipikus, hogy a CPU dolgozik egy ideig folyamatosan, majd egy rendszerhíváshoz ér, hogy olvasson, vagy írjon egy fájlt. A rendszerhívás visszatérése után megint számol, amíg újra adatokra lesz szüksége, vagy



2.22. ábra. *Számítási és i/O várakozási periódusok váltakozása. (a) CPU-igényes processzus. (b) I/O-igényes processzus*

adatokat akar írni, és így tovább. Figyeljük meg, hogy némelyik I/O-tevékenység számításnak minősül. Például amikor képernyőfrissítéskor a CPU biteket másol a videomemóriába, akkor nem I/O-művelet történik, mert a CPU dolgozik. Ebben az értelemben I/O-művelet az, amikor a processzus blokkolt állapotba kerül, hogy megvárja, amíg egy külső eszköz befejezi a tevékenységét.

A 2.22. ábrán fontos észrevennünk, hogy némelyik processzus, mint például a 2.22.(a) ábrán látható is, ideje nagy részét számítások végzésével tölti, míg mások, mint például a 2.22.(b) ábrán látható, idejük nagy részét I/O-műveletekre való várakozással töltik. Előbbieket **számításigényes**nek (**CPU-igényes**nek), utóbbiakat **I/O-igényes**nek nevezzük. A számításigényes processzusokra hosszú számítási periódusok és ritka I/O-várakozások jellemzők, az I/O-igényesekre pedig rövid számítási periódusok és gyakori I/O-várakozások. Figyeljük meg, hogy a kulcstényező a számítási periódusok hossza, nem pedig az I/O-periódusok hossza. Az I/O-igényes processzusok azért azok, mert alig végeznek számítást az I/O-kérések között, nem pedig azért, mert különösen hosszú I/O-műveleteket végeznek. Ugyanannyi ideig tart beolvasni egy lemezblokkot, függetlenül attól, hogy mennyi ideig tart utána feldolgozni az adatokat.

Érdemes megjegyezni, hogy a CPU-k sebességének növekedésével a processzusok egyre inkább I/O-igényesekké válhatnak. Ez azért lehetséges, mert a CPU-k teljesítménye sokkal gyorsabban nő, mint a lemezeké. Ennek következtében az I/O-igényes processzusok ütemezése valószínűleg fontosabb témává válik a jövőben. Az alapötlet ezzel kapcsolatban az, hogy ha egy I/O-igényes processzus futni akar, akkor tehesse azt meg minél előbb, hogy kiadhassa a lemezparancsokat, és kihasználva tartsa a lemezt.

Mikor ütemezzünk?

Sok olyan helyzet van, amikor ütemezésre sor kerülhet. Először is, feltétlenül szükséges két esetben:

1. Amikor egy processzus befejeződik.
2. Amikor egy processzus blokkolódik I/O-művelet vagy szemafor miatt.

Mindkét esetben az éppen futó processzus nem tud tovább haladni, ezért másikat kell választani helyette.

Van további három eset, amikor rendszerint ütemezésre kerül sor, bár logikailag nem feltétlenül lenne szükséges:

1. Amikor új processzus jön létre.
2. Amikor I/O-megszakítás következik be.
3. Amikor időzítőmegszakítás következik be.

Új processzus létrehozása esetén van értelme újraértékelni a prioritásokat. Bizonyos esetekben a szülő kérhet a sajátjától különböző prioritást a gyermekének.

Egy I/O-megszakítás rendszerint azt jelenti, hogy valamelyik I/O-eszköz befejezte a feladatát, és lehet olyan processzus, amelyik éppen erre várt, és futtatható állapotba került.

Időzítőmegszakítás esetén lehetőség van annak eldöntésére, hogy az éppen futó processzus elég hosszú ideig futott-e már. Az ütemezési algoritmusok két csoportba sorolhatók az időzítőmegszakítások kezelésének vonatkozásában. **Nem megszakítható ütemezés** esetén az ütemező a kiválasztott processzust addig engedi futni, amíg az blokkolódik (I/O-művelet vagy másik processzusra várakozás miatt), vagy amíg önszántából le nem mond a processzorról. Ezzel ellentétben **megszakítható ütemezés** esetén a kiválasztott processzus csak legfeljebb egy előre meghatározott ideig futhat. Ha a kiszabott időintervallum végén még mindig fut, akkor felfüggesztésre kerül, és az ütemező egy másik processzust választ helyette (ha tud). Megszakítható ütemezés csak úgy valósítható meg, ha az időintervallum végén egy időzítőmegszakítást generál, ezáltal a CPU visszakerül az ütemezőhöz. Ha nincs időzítő, akkor a nem megszakítható ütemezés az egyedüli lehetőség.

Ütemezési algoritmusok csoportosítása

Nem meglepő módon, különböző környezetekben különböző ütemezési algoritmusokra van szükség. Ez azért fordulhat elő, mert különböző alkalmazási területek (és különböző fajta operációs rendszerek) esetén a célok is különböznek. Más szavakkal nem minden rendszerben ugyanazok az optimális ütemezési döntések. Három területet érdemes megkülönböztetni:

1. Kötegelt.
2. Interaktív.
3. Valós idejű.

Kötegelt rendszerekben nincsenek felhasználók, akik a terminálóknál türelmetlenül várják a választ. Emiatt nem megszakítható ütemezési algoritmusok, vagy minden processzus számára hosszú időintervallumokat engedélyező, megszakítható ütemezési algoritmusok használata gyakran elfogadható. Ezzel a megoldással csökken a processzusváltások száma, így nő a teljesítmény.

Interaktív rendszerekben az időnkénti megszakítás nélkülözhetetlen, nehogy valamelyik processzus kisajátítsa a CPU-t, és megakadályozza a többit a futásban. Még ha nem is szándékosan történik ilyen, programhiba miatt egy processzus kizárhatja az összes többit. A megszakításos ütemezésre van szükség ennek megakadályozására.

Valós idejű korlátokkal működő rendszerekben furcsa módon nem mindig van szükség megszakításos ütemezésre, mert a processzusok tudatában vannak, hogy nem futhatnak hosszú ideig, és rendszerint gyorsan elvégzik a feladatukat, majd blokkolódnak. Az interaktív rendszerektől eltérően a valós idejű rendszerekben csak a szóban forgó alkalmazás érdekeit szem előtt tartó programok futnak. Az interaktív rendszerek általános célúak, bármilyen program előfordulhat, még olyan is, amely nem működik együtt a többiekkel, vagy egyenesen ártó szándékú.

Ütemezési algoritmusok céljai

Ütemezési algoritmus tervezésekor jó tisztában lenni azzal, hogy az algoritmusnak mit kell elérnie. Bizonyos célok függnek a környezettől (kötegelt, interaktív vagy valós idejű), de vannak olyanok is, amelyek elérése mindig kívánatos. A 2.23. ábrán lehetséges célokat találunk, ezeket tárgyaljuk a következőkben.

A pártatlanság minden körülmények között fontos. Hasonló processzusoknak hasonló kiszolgálást kell kapniuk. Nem igazságos egyiknek sokkal több CPU-időt adni, mint egy hozzá hasonlónak. Természetesen processzusok különböző csoportjait szabad különbözőképpen kezelni. Gondoljunk csak egy atomerőmű biztonsági berendezéseinek vezérlésére és a dolgozók fizetési listájának elkészítésére az erőmű számítóközpontjában.

A pártatlansághoz némileg kapcsolódik a rendszer ütemezési elveinek betartatása. Ha a helyi ütemezési elv az, hogy a biztonsági vezérlőprocesszusok bármikor futhatnak, még akkor is, ha a fizetési lista 30 másodpercet késik, akkor az ütemezőnek ezt az elvet kell kikényszerítenie.

Egy másik általános cél a rendszer részeinek egységes terheltségét fenntartani, amikor csak lehetséges. Ha a CPU és az összes I/O-eszköz folyamatosan dolgozik, akkor időegységenként több feladat végezhető el, mint ha a rendszer egyes részei tétlenül állnak. Kötegelt rendszerekben például az ütemező döntheti el, hogy mely feladatokat tölti be a memóriába futtatásra. Jobb megoldás CPU-igényes és I/O-igényes feladatokat vegyesen futtatni, mint először az összes CPU-igényeset, majd ezek befejeződése után az I/O-igényeseket. Utóbbi stratégia esetén a CPU-

Minden rendszer
- Pártatlanság – minden processzusnak megfelelő hozzáférést biztosítani a CPU-hoz
- Elvek betartatása – a meghatározott elvek szerinti működés biztosítása
- Egyensúly – a rendszer minden részének egyenletes terhelése

Kötegelt rendszerek
- Áteresztőképesség – maximalizálni az időegységenként végrehajtott feladatok számát
- Áthaladási idő – minimalizálni a feladat-végrehajtás kezdeményezése és a befejeződés közt eltelt időt
- CPU-kihasználtság – a CPU soha nem állhat tétlenül

Interaktív rendszerek
- Válaszidő – a kérésekre gyors válasz biztosítása
- Arányosság – a felhasználók elvárásainak való megfelelés

Valós idejű rendszerek
- Határidők betartása – adatvesztés elkerülése
- Előrejelezhetőség – minőségromlás elkerülése multimédia-rendszerekben

2.23. ábra. *Az ütemezési algoritmus lehetséges céljai különböző körülmények között*

igényes feladatok futása alatt azok harcolnak egymással a CPU birtoklásáért, a lemez pedig kihasználatlanul áll. Később, amikor az I/O-igényes feladatok kerülnek sorra, harcolni fognak egymással a lemezért, a CPU pedig tétlenségre lesz ítélve. Gondosan kiválogatott feladatkeverékkel a folyamatos kihasználtság biztosítható.

A sok kötegelt feladatot (például biztosítási igények feldolgozását) végző vállalati számítóközpontok vezetői tipikusan három mérőszámot figyelnek, hogy megállapítsák, rendszerük mennyire működik jól. Ezek az **áteresztőképesség**, az **áthaladási idő** és a **CPU-kihasználtság**. Az áteresztőképesség a rendszerben időegységenként végrehajtott feladatok száma. Mindent figyelembe véve, másodpercenként 50 feladat jobb, mint másodpercenként 40 feladat. Az áthaladási idő a feladat-végrehajtás kezdeményezése és a befejeződés közt eltelt idő. Azt méri, hogy a felhasználóknak átlagosan mennyit kell várniuk arra, hogy megkapják az eredményt. A szabály itt az, hogy minél kisebb, annál jobb.

Egy olyan ütemezési algoritmus, amely maximalizálja az áteresztőképességet, nem feltétlenül tudja minimalizálni az áthaladási időt. Például ha rövid és hosszú feladatok vegyesen állnak sorban, akkor a rövid feladatok futtatásával kiváló áteresztőképesség mutatható fel (sok rövid feladat másodpercenként), de ennek az az ára, hogy a hosszú feladatok számára az áthaladási idő rettenetesen rossz lesz. Ha a rövid feladatok folyamatosan érkeznek, akkor a hosszú feladatok soha nem kerülnek sorra, az átlagos áthaladási idő a végtelenhez kezd közelíteni, miközben az átlagos áteresztőképesség magas marad.

A CPU-kihasználtság fontos szempont a kötegelt rendszerekben, mert a nagygépeken, ahol ezek a kötegelt rendszerek rendszerint futnak, a CPU teszi ki az ár tetemes részét. Emiatt a számítóközpontok vezetői bűntudatot éreznek, ha a CPU nem dolgozik folyamatosan. Valójában azonban ez nem valami jó mérőszám. Ami valóban számít, az az áteresztőképesség és az áthaladási idő. A CPU-kihasználtság

mérőszámként való felhasználása olyan, mintha az autókat a motor fordulatszáma alapján ítélnénk meg.

Az interaktív rendszerekre, különösen az időosztásos rendszerekre és a szerverekre más szabályok érvényesek. A legfontosabb cél a **válaszidő**, vagyis egy parancs kiadása és a válasz megérkezése között eltelt idő minimalizálása. Egy olyan személyi számítógépen, ahol háttérprocesszusok is futnak (például e-mail fogadás és küldés hálózatról), a háttértevékenységekkel szemben előnyt kell élveznie a felhasználó parancsainak, ha mondjuk el akar indítani egy programot, vagy meg akar nyitni egy fájlt. Az interaktív tevékenységek előnyben részesítését a felhasználók értékelni fogják.

Egy ehhez némileg kapcsolódó kérdés az, amit **arányosság**nak nevezhetnénk. Minden felhasználónak van (gyakran hibás) elképzelése arról, hogy minek mennyi ideig kellene tartania. Ha egy összetettnek tűnő kérés hosszú ideig tart, akkor azt elfogadják, de ha valami egyszerűnek tűnő tart sokáig, akkor bosszankodnak. Például ha egy olyan ikonra kattint valaki, amely egy analóg modemen keresztül teremt kapcsolatot az internetszolgáltatóval, és a kapcsolat felépítése 45 másodpercig tart, akkor valószínűleg a felhasználó elfogadja ezt, mint elkerülhetetlent. Másrészt ha olyan ikonra kattint, ami bontja a kapcsolatot, és ez tart 45 másodpercig, akkor ugyanez a felhasználó minden bizonnyal erősen átkozódni fog 30 másodperc elteltével, 45 másodpercnél pedig már habzik a szája. Mindez abból adódik, hogy az általános vélekedés szerint telefonhíváskor egy kapcsolat felépítése több ideig szokott tartani, mint amikor csak letesszük a kagylót. Bizonyos esetekben (mint az előzőben is) az ütemező semmit sem tehet a válaszidő tekintetében. Máskor viszont igen, különösen akkor, ha a késlekedés a processzusok sorrendjének helytelen megválasztásából adódott.

A valós idejű rendszereknek az interaktív rendszerektől eltérő tulajdonságaik vannak, így ütemezési céljaik is mások. Fő jellemzőjük a határidők megléte, amelyeket be kell tartani, ha lehet. Például ha a számítógép egy olyan eszközt vezérel, amely szabályos időközönként adatokat szolgáltat, akkor az adatgyűjtő processzus megkésett futtatása miatt adatvesztés léphet fel. Ezért a valós idejű rendszerekben a legfontosabb cél a határidők lehetőség szerinti betartása.

A legtöbb valós idejű rendszerben, főleg a multimédia-rendszerekben, fontos az **előrejelezhetőség**. Ha néha egy-egy határidőt nem sikerül betartani, az még nem végzetes, de ha a hanglejátszó processzus túlságosan szabálytalanul fut, akkor a hangminőség gyorsan romlik. A videolejátszás is hasonló problémákat vet fel, de a fülünk sokkal érzékenyebb az eltérésekre, mint a szemünk. Ezeket a jelenségeket csak pontosan előre jelezhető és szabályos ütemezéssel lehet kiküszöbölni.

2.4.2. Ütemezés kötegelt rendszerekben

Itt az ideje az általános ütemezési kérdésektől a konkrét algoritmusok felé fordulni. Ebben a részben kötegelt rendszerekben használt algoritmusokat tárgyalunk. A rá következőkben interaktív és valós idejű rendszereket vizsgálunk meg.

Érdemes rámutatni arra, hogy némelyik algoritmus kötegelt és interaktív rendszerekben is használatos, ezeket később tanulmányozzuk. Most csak azokra az algoritmusokra koncentrálunk, amelyek csak kötegelt rendszerekben használhatók.

Sorrendi ütemezés

Talán az ütemezési algoritmusok legegyszerűbbike a nem megszakítható **sorrendi ütemezés**. Ez az algoritmus olyan sorrendben osztja ki a CPU-t a processzusoknak, amilyen sorrendben azok kérik. Alapjában véve a futásra kész processzusok egyetlen várakozó soron állnak készenlétben. Amikor reggel az első feladat belép a rendszerbe, azonnal indulhat, és addig futhat, amíg akar. Ha további feladatok érkeznek, azok a várakozási sor végére kerülnek. Amikor egy futó processzus blokkolódik, a sor elején álló processzus futhat helyette. Amikor egy blokkolt processzus újra futásra kész lesz, akkor az újonnan érkezett feladatokhoz hasonlóan a sor végére kerül.

Ennek az algoritmusnak az a nagy erőssége, hogy könnyű megérteni és ugyanilyen könnyű beprogramozni. Pártatlan is abban az értelemben, ahogy pártatlan az, hogy a nehezen megszerezhető sport- vagy koncertjegyekhez az fér hozzá, aki hajlandó hajnali két órától sorban állni a pénztárnál. Ennél az algoritmusnál egyetlen láncolt lista tartja nyilván a futásra kész processzusokat. Egy processzus kiválasztása azt jelenti, hogy le kell venni a sor elejéről. Új feladat vagy blokkolás alól felszabadult processzus elhelyezése a lista végére történő beszúrást jelent. Mi lehetne ennél egyszerűbb?

Sajnos a sorrendi ütemezésnek van egy nagy hátránya is. Tegyük fel, hogy van egy számításigényes processzus, amely alkalmanként 1 másodpercig fut, és több I/O-igényes processzus, amelyek kevés processzoridőt használnak, de egyenként 1000 lemezolvasást kell elvégezniük futásuk alatt. A számításigényes processzus fut 1 másodpercig, majd olvas egy lemezblokkot. Ekkor az összes I/O-processzus fut, és mindegyik kezdeményez egy lemezolvasást. Amikor a számításigényes processzus megkapja a lemezblokkját, újból fut 1 másodpercig, majd ezt követik az I/O-processzusok gyors egymásutánban.

Összesítve az lesz az eredmény, hogy az I/O-igényes processzusok másodpercenként 1 blokkot tudnak olvasni, így 1000 másodperc alatt futnak le. Ha az ütemezési algoritmus 10 ezred másodpercenként megszakította volna a számításigényes processzust, akkor az I/O-igényes processzusok 1000 helyett 10 másodperc alatt végeztek volna, és még a számításigényes processzus sem lassult volna le nagyon.

A legrövidebb feladatot először

Most tekintsünk egy másik nem megszakítható kötegelt ütemezési algoritmust, amely feltételezi, hogy a futási idők előre ismertek. Egy biztosítótársaságnál például az ember képes elég pontosan megjósolni, hogy mennyi időt vesz igénybe

2.24. ábra. *Példa a legrövidebb feladatot először ütemezésre. (a) Négy feladat futtatása az eredeti sorrendben. (b) Négy feladat futtatása a legrövidebb feladatot először sorrendben*

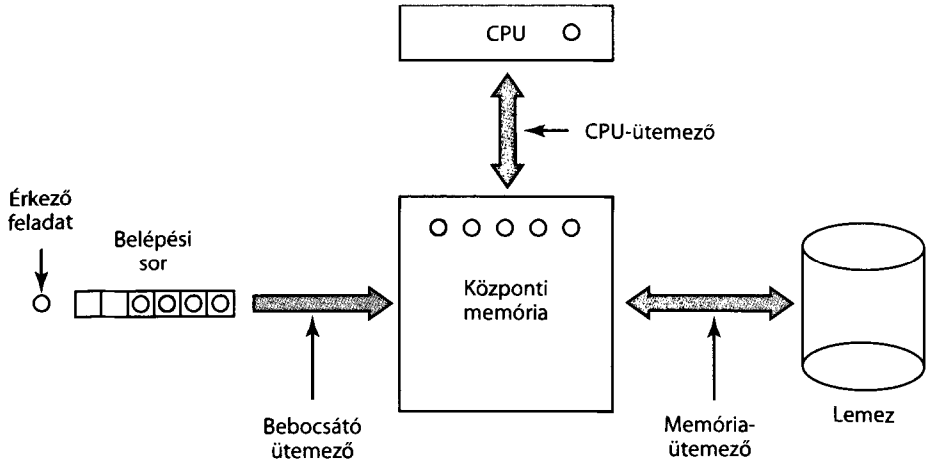
1000 kérelem kötegelt feldolgozása, mivel a hasonló munkák mindennaposak. Amikor több egyformán fontos feladat van a bemenő sorban futásra készen, akkor az ütemezőnek a **legrövidebb feladatot először** elvet kell használnia. Vessünk egy pillantást a 2.24. ábrára. Itt négy feladat van, *A*, *B*, *C* és *D*, amelyek futásideje rendre 8, 4, 4 és 4 perc. A megadott sorrendben futtatva ezeket, az áthaladási idő az *A* számára 8 perc, *B* számára 12 perc, *C* számára 16 perc, *D* számára pedig 20 perc, az átlag 14 perc.

Most nézzük ennek a négy feladatnak a futását a legrövidebb feladatot először elvet használva, mint azt a 2.24.(b) ábra mutatja. Az áthaladási idő most 4, 8, 12 és 20 perc, az átlag 11 perc. A legrövidebb feladatot először algoritmus bizonyíthatóan optimális. Tekintsük a négyfeladatos esetet, rendre *a*, *b*, *c és d* futásidőkkel. Az első befejezi *a* időpontban, a második $a + b$ időpontban, és így tovább. Az átlagos áthaladási idő $(4a + 3b + 2c + d)/4$. Világos, hogy *a* többszörösen járul hozzá az átlaghoz, mint a többi idő, ezért neki kell a legrövidebb feladatnak lennie, a *b* a következő, azután *c* és végül *d*, mint a leghosszabb, és így már csak a saját áthaladási idejére van hatással. Ugyanez az érvelés alkalmazható tetszőleges számú feladatra.

Érdemes megjegyezni, hogy a legrövidebb feladatot először csak akkor optimális, ha a feladatok egyszerre rendelkezésre állnak. Ellenpéldaként tekintsünk 5 feladatot, *A*-tól *E*-ig, amelyek futási ideje sorrendben 2, 4, 1, 1 és 1 perc. Az érkezési idejük 0, 0, 3, 3 és 3. Kezdetben csak *A* és *B* közül választhatunk, mivel a többi még nem érkezett meg. A legrövidebb feladatot először stratégia a feladatokat *A*, *B*, *C*, *D*, *E* sorrendben fogja futtatni, az átlagos várakozás 4,6 perc. Ha *B*, *C*, *D*, *E*, *A* sorrendben futtattuk volna őket, akkor viszont az átlagos várakozás csak 4,4 perc lett volna.

A legrövidebb maradék futási idejű következzen

A legrövidebb feladatot először algoritmus egy megszakítható változata a **legrövidebb maradék futási idejű következzen**. Ennél az algoritmusnál az ütemező mindig azt a processzust választja, amelynek a legkevesebb a befejeződésig még megmaradt ideje. Itt megint csak ismerni kell a futási időket előre. Amikor új feladat érkezik, a teljes ideje összehasonlításra kerül az éppen futó processzus még hátralévő idejével. Ha az új feladat kevesebb időt igényel, mint az aktuális processzus, akkor az aktuális processzust lecseréli az új feladatra. Ezzel a megoldással az új, rövid feladatok jó kiszolgálásban részesülnek.



2.25. ábra. *Háromszintű ütemezés*

Háromszintű ütemezés

Bizonyos szempontból a kötegelt rendszerekben az ütemezés három különböző szinten történhet, ahogy a 2.25. ábra mutatja. Az újonnan érkező feladatok először egy lemezen tárolt belépési várakozó sorba kerülnek. A **bebocsátó ütemező** dönti el, hogy mely feladatok léphetnek be a rendszerbe. A többi a belépési várakozó soron marad, amíg ki nem választják őket. A bebocsátást vezérlő tipikus algoritmus egy megfelelő számításigényes és I/O-igényes keverék előállítását kísérelheti meg. Másik lehetőség, hogy a rövid feladatokat hamar beengedi, a hosszabbaknak várniuk kell. A bebocsátó ütemező megteheti, hogy bizonyos feladatokat a belépési sorban tart, míg később érkezőket beenged, ha úgy látja jónak.

Ha egy feladat már belépett a rendszerbe, akkor létre lehet hozni számára egy processzust, és elkezdhet vetélkedni a CPU-ért. De az is megtörténhet, hogy olyan sok processzus van, hogy nem férnek el a memóriában. Ebben az esetben néhány processzust ki kell helyezni lemezre. Az ütemezés második szintje azt dönti el, hogy melyik processzus maradjon a memóriában, és melyik kerüljön ki lemezre. Azt a komponenst, amely ezt a döntést meghozza, **memóriaütemező**nek nevezzük.

A döntéseket sűrűn felül kell vizsgálni, hogy a lemezen tárolt feladatoknak is legyen esélyük a bekerülésre. Azonban, mivel egy feladat lemezről történő behozatala költséges művelet, a felülvizsgálatnak valószínűleg csak legfeljebb másodpercenként egyszer, esetleg ennél is ritkábban szabad megtörténnie. Ha a központ-

ti memóriát túl gyakran átszervezzük, akkor az nagy sávszélességet pazarol el a lemezegységnél, és lelassítja a fájlműveleteket.

A rendszer egészének teljesítményére tekintettel a memóriaütemezőnek körültekintően kell eljárnia, hogy meghatározza a memóriában tartott processzusok számát, amit a **multiprogramozás mértékének** is nevezünk, valamint azt, hogy milyen processzusok legyenek a memóriában. Ha van információja arról, hogy melyik processzus számításigényes, illetve melyik I/O-igényes, akkor megkísérelheti ezeknek valamilyen keverékét a memóriában tartani. Nagyon durva becsléssel, ha bizonyos típusú processzus idejének 20%-át számolással tölti, akkor ebből nagyjából öt elég ahhoz, hogy a CPU-t kihasználva tartsa.

Ezeknek a döntéseknek a meghozatalához a memóriaütemező rendszeres időközönként átnézi a lemezen tárolt processzusokat. A szempontjai között az alábbiak lehetnek:

1. Mennyi idő telt el a processzus lemezre vitele óta?
2. Mennyi CPU-időt használt fel a processzus nemrégiben?
3. Milyen nagy a processzus? (A kicsik nincsenek útban.)
4. Mennyire fontos a processzus?

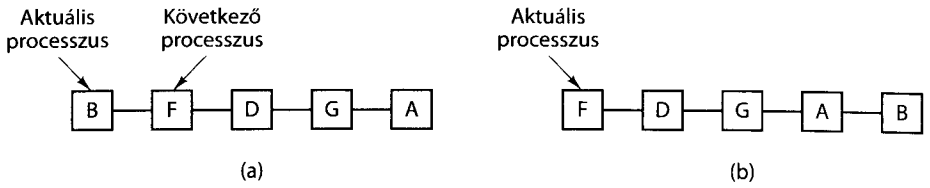
Az ütemezés harmadik szintje választja ki valójában, hogy a futásra kész processzusok közül melyik fusson következőnek. Ezt gyakran **CPU-ütemezőnek** nevezik, és az emberek általában erre gondolnak, amikor *az* ütemezőről beszélnek. Bármilyen megfelelő algoritmus használható itt, akár megszakítható, akár nem megszakítható. A fentebb említettek, és néhány, a következő részben tárgyalandó is ezek közé tartozik.

2.4.3. Ütemezés interaktív rendszerekben

Most olyan algoritmusokat fogunk megnézni, amelyek interaktív rendszerekben használhatók. Ezek mindegyike alkalmas CPU-ütemezőnek kötegelt rendszerben is. Bár a háromszintű ütemezés nem alkalmazható itt, kétszintű ütemezés (memóriaütemezés és CPU-ütemezés) lehetséges, sőt gyakori. A továbbiakban a CPU-ütemezőre és néhány gyakori ütemezési algoritmusra fogunk koncentrálni.

Round robin ütemezés

Most nézzünk meg néhány konkrét ütemező eljárást. Az egyik legrégebbi, legegyszerűbb, legpártatlanabb és legszélesebb körben használt algoritmus a **round robin**. Minden processzusnak ki van osztva egy időintervallum, amelyet **időszeletnek** nevezünk, és amely alatt engedélyezett a futása. Ha az időszelet végén a processzus még fut, akkor a CPU átadódik egy másik processzusnak. Ha a processzus blokkolódik vagy véget ér, mielőtt az időszelet letelik, akkor természetesen a CPU átadása a processzus blokkolásakor megtörténik. A round robin megvalósítása



2.26. ábra. *Round robin ütemezés. (a) A futtatandó processzusok listája. (b) A futtatandó processzusok listája azután, hogy* B *elhasználta az időszeletét*

egyszerű. Az ütemezőnek mindössze egy listát kell karbantartania a futtatandó processzusokról, amint ezt a 2.26.(a) ábrán látjuk. Amikor a processzus felhasználja az időszeletét, akkor a lista végére kerül, ahogyan a 2.26.(b) ábrán látható.

Csak egy érdekes kérdés van a round robin ütemezéssel kapcsolatban: az időszelet hossza. Egyik processzusról a másikra való átkapcsolás bizonyos adminisztrációs időt igényel – regiszterek és memóriatérképek mentése és betöltése, különböző táblázatok és listák aktualizálása, a gyorsítótár tartalmának visszaírása a memóriába, majd újratöltése stb. Tegyük fel, hogy ez a **processzusátkapcsolás** vagy **környezetátkapcsolás**, ahogy ezt sokszor nevezik, 1 ezred másodpercig tart a fenti tevékenységeket mind beleértve. Tegyük fel azt is, hogy az időszelet 4 ezred másodpercre van beállítva. Ezekkel a paraméterekkel a CPU 4 ezred másodperc hasznos munka után kénytelen 1 ezred másodpercet tölteni a processzusátkapcsolással. A CPU-idő 20%-a kárba vész az adminisztrációs költségekre. Ez nyilván túl sok.

A CPU hatékonyságának javítására beállíthatnánk az időszeletet mondjuk 100 ezred másodpercre. Ekkor az elpocsékolt idő csak 1% lenne. De fontoljuk meg, mi történik egy időosztásos rendszerben, ha 10 interaktív felhasználó közel egy időben leüti az ENTER billentyűt. Tíz processzus kerül fel a futtatható processzusok listájára. Ha a CPU éppen tétlen, az első azonnal elkezd futni, a második nem kezdhet el futni, csak 100 ezred másodperc múlva, és így tovább. Lehet, hogy a szerencsétlen utolsónak 1 másodpercet is várnia kell, mielőtt sorra kerül, feltéve, hogy a többiek kihasználják a teljes időszeletüket. A legtöbb felhasználó egy rövid parancsra adott 1 másodperces válaszidőt igencsak lomhának találna.

Egy másik szempont, hogy ha az időszelet az átlagos CPU használati periódusnál hosszabbra van állítva, akkor időszelet lejárta miatti megszakítás ritkán fog történni. Ehelyett a legtöbb processzus az időszelet lejárta előtt blokkolódik, ezzel processzusátkapcsolást okozva. Az időszelet lejárta miatti megszakítások kiiktatása javítja a teljesítményt, mert így váltások csak akkor történnek, amikor logikailag szükségesek, vagyis akkor, amikor a processzus úgysem tudná folytatni a futását, mert várakoznia kell valamire.

A megfogalmazható következtetések tehát: az időszelet túl kicsire állítása túl sok processzusátkapcsolást okoz, és csökken a CPU hatékonysága; de túl nagyra állítása rövid interaktív kérésekre gyenge válaszidőt eredményezhet. Az időszelet 20–50 ezred másodperc körüli értéke gyakran ésszerű kompromisszum.

Prioritásos ütemezés

A round robin ütemezés implicit módon feltételezi, hogy minden processzus egyformán fontos. A többfelhasználós számítógépeket használóknak és működtetőknek gyakran különböző véleményük van erről a témáról. Egy egyetemen a sorrend lehet: először a dékánok, azután a professzorok, a titkárok, a portások és végül a hallgatók. Külső tényezők figyelembevételének igénye a **prioritásos ütemezés**hez vezet. Az alapötlet egyszerű: minden processzushoz rendeljünk egy prioritást, és a legmagasabb prioritású futásra kész processzusnak engedjük meg, hogy fusson.

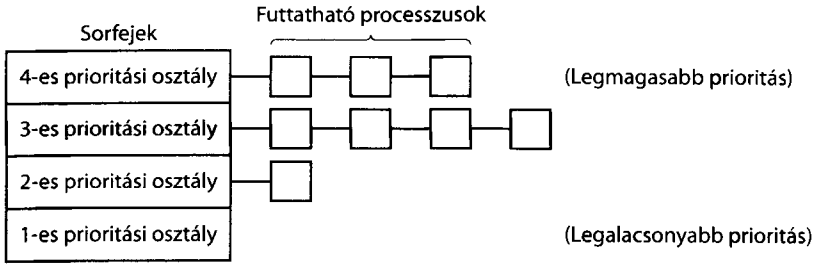
Még egy egyfelhasználós PC-n is lehet több processzus, amelyek közül némelyik fontosabb, mint a másik. Például az elektronikus leveleket a háttérben elküldő processzus kaphatna alacsonyabb prioritást, mint a képernyőn valós időben videofilmet megjelenítő processzus.

Annak megelőzésére, hogy a magas prioritású processzusok végtelen ideig fussanak, az ütemező minden óraütemben (vagyis minden órajel-megszakításnál) csökkentheti az éppen futó processzus prioritását. Amikor emiatt a prioritása a második legmagasabb prioritású processzusé alá csökken, akkor a processzusátkapcsolás megtörténik. Másik megoldás lehet, hogy minden processzushoz hozzárendelünk egy maximális időszeletet, ameddig futhat. Amikor ezt az időszeletet kihasználta, a következő legmagasabb prioritású processzus kap lehetőséget a futásra.

A prioritásokat a processzusokhoz statikusan vagy dinamikusan lehet hozzárendelni. Egy katonai számítógépben a tábornokok által indított processzus kezdődhet a 100-as prioritásnál, az ezredesek által indított processzusé 90-nél, az őrnagyoké 80-nál, a századosoké 70-nél, a hadnagyoké 60-nál, és így tovább. Egy fizetős számítóközpontban a magas prioritású feladatok ára lehet 100 dollár óránként, a közepes prioritásúaké 75 dollár óránként, az alacsony prioritásúaké 50 dollár óránként. A Unix-rendszerben van egy utasítás, a *nice* (udvarias), amely lehetővé teszi a felhasználónak, hogy önkéntesen csökkentse a processzusa prioritását azért, hogy a többi felhasználóval szemben udvarias legyen. Ezt soha senki nem használja.

A rendszer a prioritásokat dinamikusan is hozzárendelheti a processzusokhoz, hogy elérjen bizonyos rendszercélokat. Például van néhány erősen I/O-igényes processzus, és nagyon sok időt töltenek az I/O-műveletek befejezésére várva. Mindannyiszor, amikor egy ilyen processzus igényli a CPU-t, azonnal meg kellene kapnia, hogy lehetővé tegyük számára a következő I/O-kérés megkezdését, ami ezután már párhuzamosan hajtódhat végre egy másik processzus számításaival. Ha sokáig váratjuk az I/O-igényes processzust a CPU-ra, akkor az csak szükségtelenül hosszú ideig foglalja a memóriát. Egy egyszerű algoritmus, amely jó szolgáltatást nyújt az I/O-igényes processzusnak: a processzus prioritását állítsuk $1/f$-re, ahol f az utolsó időszeletből a processzus által felhasznált rész. Egy olyan processzusnak, amely csak 1 ezred másodpercet használt fel az 50 ezred másodperces időszeletéből, a prioritása 50 lesz, míg egy olyan processzusnak, amely 50 ezred másodpercig futott blokkolódás előtt, a prioritása 2 lesz, és annak, amelyik a teljes időszeletet kihasználta, a prioritása 1 lesz.

Gyakran kényelmes a processzusokat prioritási osztályokba sorolni, és prioritásos ütemezést használni az osztályok között, de round robin ütemezést egy



2.27. ábra. *Egy ütemezési algoritmus négy prioritási osztállyal*

osztályon belül. A 2.27. ábra egy rendszert mutat be négy prioritási osztállyal. Az ütemezési algoritmus a következő: amíg van futtatható processzus a 4-es prioritási osztályban, mindegyik egy időszeletig fog futni, round robin módon, és nem törődünk az alacsonyabb prioritási osztályokkal. Ha a 4-es prioritási osztály üres, akkor a 3-as prioritási osztály processzusai futnak round robin módon. Ha mind a 4-es, mind a 3-as osztály üres, akkor a 2-es osztály fut round robin módon, és így tovább. Ha a prioritások nincsenek időnként kiigazítva, akkor az alacsonyabb prioritási osztályok akár mind éhen is halhatnak.

A MINIX 3 a 2.27. ábrán látotthoz hasonló rendszert használ, habár alapértelmezés szerint 16 prioritási osztálya van. A MINIX 3-ban az operációs rendszer részei processzusként futnak. A MINIX 3 a taszkokat (I/O-meghajtókat) és a szervereket (processzuskezelő, fájlrendszer és hálózati szerver) a legmagasabb prioritási osztályokba teszi. Az egyes taszkok és szerverek kezdeti prioritása fordítási időben dől el; egy lassabb I/O-eszközhöz kisebb prioritás rendelhető, mint egy gyorsabb I/O-eszközhöz vagy akár egy szerverhez. A felhasználói processzusoknak általában kisebb a prioritása, mint a rendszerkomponenseknek, de az összes prioritási érték változhat a futás során.

Többszörös sorok

Az egyik legkorábbi prioritásos ütemező a CTSS-ben volt (Corbató et al., 1962). A CTSS-nek az volt a problémája, hogy a processzusváltás nagyon lassú volt, mert a 7094-es csak egy processzust tudott a memóriában tartani. Minden váltás abból állt, hogy az aktuális processzust ki kellett tenni a lemezre, és be kellett olvasni egy újat a lemezről. A CTSS tervezői gyorsan rájöttek, hogy hatékonyabb, ha a CPU-igényes processzusoknak időnként egy nagy időszeletet adnak, mintha gyakran adnak nekik kis időszeleteket (hogy csökkentsék a lapcserék számát). Másrészt, ha nagy időszeletet adnánk minden processzusnak, az gyenge válaszidőt jelentene, mint azt már láttuk. Megoldásuk prioritási osztályok felállítása volt. A legmagasabb osztályban lévő processzusok egy időszeletig futnak. A következő legmagasabb osztályban lévő processzusok két időszeletig futnak. A következő osztályban lévő processzusok négy időszeletig futnak, és így tovább. Valahányszor egy pro-

cesszus elhasználja az összes, számára biztosított időszeletet, egy osztállyal lejjebb kerül.

Példaként tekintsünk egy processzust, amelynek 100 időszeletnyi folyamatos számolási időre van szüksége. Induláskor egy időszeletet kap, ezután lecserélik. A következő alkalommal két időszeletet kap, mielőtt lecserélik. Az ezután következő futásokban 4, 8, 16, 32 és 64 időszeletet kap, bár az utolsó 64 időszeletből csak 37-et használ fel a munkája befejezéséhez. Csak 7 csere szükséges (beleértve a kezdeti betöltést is) a 100 helyett, amit egy tiszta round robin algoritmus végezne. Továbbá, ahogy egy processzus egyre mélyebbre süllyed a prioritási sorok között, egyre kisebb gyakorisággal fog futni, CPU-időt takarítva meg ezzel a rövid, interaktív processzusok számára.

A következő elv bevezetése azokat a processzusokat védi meg az örökös büntetéstől, amelyek induláskor hosszú futási időt igényeltek, de később interaktívvá váltak. Valahányszor ENTER-t ütöttek le egy terminálon, az ehhez a terminálhoz tartozó processzus a legmagasabb prioritási osztályba került, feltételezve róla, hogy interaktív lett. Egy szép napon egy felhasználó, akinek CPU-igényes processzusa volt, felfedezte, hogy ha csak ül a terminál előtt, és véletlenszerűen, néhány másodpercenként ENTER-t üt, akkor ez csodát tesz a válaszidejével. Ezt elmesélte a barátainak is. A történet tanulsága: sokkal nehezebb jól csinálni valamit a gyakorlatban, mint elméletben.

Sok más algoritmust is használtak már a processzusok prioritási osztályokba sorolására. Például a nagy hatású berkeley-i XDS 940 rendszernek (Lampson, 1968) négy prioritási osztálya volt; ezeket terminálnak, I/O-nak, rövid időszeletnek és hosszú időszeletnek neveztek. Ha egy processzus, amely terminálról érkező adatokra várt, végre megkapta az ébresztést, átment a legmagasabb prioritási (terminál) osztályba. Ha egy processzus lemezművelet befejezésére várt, átment a második osztályba. Ha egy processzus még futott, amikor elfogyott az időszelete, kezdetben a harmadik osztályba került. Azonban ha egy processzus egymás után túl sokszor használta el az időszeletét anélkül, hogy terminál vagy I/O miatt blokkolódott volna, lekerült a legalsó sorba. Sok más rendszer használ ehhez hasonlót, hogy kedvezzen az interaktív felhasználóknak és processzusoknak a háttérben futókkal szemben.

Legrövidebb processzus következzen

Mivel a legrövidebb feladatot először módszer mindig minimális átlagos válaszidőt ad kötegelt rendszerben, jó lenne, ha alkalmazhatnánk interaktív processzusokra is. Bizonyos mértékig alkalmazhatjuk. Az interaktív processzusok általában a következő sémát követik: várakozás utasításra, utasítás végrehajtása, várakozás utasításra, utasítás végrehajtása, és így tovább. Ha minden utasítás végrehajtását külön „feladatnak" tekintenénk, akkor minimalizálhatnánk az összválaszidőt, a legrövidebbet futtatva először. Az egyetlen probléma annak kitalálása, hogy a párhuzamosan futtatható processzusok közül melyik a legrövidebb.

Egy megközelítés az, hogy becsléseket végzünk a múltbeli viselkedés alapján, és a processzust a legkisebb becsült futásidő alapján futtatjuk. Tegyük fel, hogy

valamelyik terminálra az utasításonkénti becsült idő T_0. Tegyük fel, hogy a következő futást T_1-nek mérjük. A becslésünket aktualizálhatjuk ennek a két számnak a súlyozott átlagát véve, vagyis az $aT_0 + (1 - a)T_1$ képlettel. Az a megválasztásával meghatározhatjuk, hogy a processzus gyorsan elfelejtse-e a régi futásokat, vagy sokáig emlékezzen rájuk. Az $a = 1/2$ választással a következő egymás utáni becsléseket kapjuk:

$$T_0, \quad T_0/2 + T_1/2, \quad T_0/4 + T_1/4 + T_2/2, \quad T_0/8 + T_1/8 + T_2/4 + T_3/2.$$

Három új futás után a T_0 súlya az új becslésben 1/8-ra csökken.

Azt a technikát, hogy a sorozat következő elemét úgy becsüljük, hogy vesszük az éppen mért értéknek és az előző becslésnek a súlyozott átlagát, néha **öregedés**nek nevezzük. Ez sok olyan esetben alkalmazható, amikor a becslést az előző értékekre alapozva kell elvégezni. Az öregedést különösen könnyű megvalósítani, ha $a = 1/2$. Csupán annyit kell tenni, hogy az új értéket hozzáadjuk a jelenlegi becsléshez, és az összeget elosztjuk 2-vel (1 bittel jobbra léptetve azt).

Garantált ütemezés

Az ütemezés egy teljesen más megközelítése az, amikor ígéreteket teszünk a felhasználónak a teljesítménnyel kapcsolatban, és be is tartjuk. Egy reális ígéret, amelyet könnyű betartani, a következő: ha n felhasználó van bejelentkezve, amikor éppen dolgozol, akkor a CPU teljesítményének körülbelül $1/n$-ed részét fogod megkapni. Hasonlóan, egy egyfelhasználós rendszerben, amelyben n processzus fut, és minden szempontból egyenlők, akkor mindegyiknek meg kell kapnia a CPU-ciklusok $1/n$-ed részét.

Hogy betartsuk az ígéretet, a rendszernek nyomon kell követnie, hogy egy processzus mennyi CPU-időt kapott létrehozása óta. Ezután mindegyikhez kiszámítja a neki járó mennyiségét, nevezetesen a létrehozása óta eltelt időt osztja n-nel. Mivel minden processzusnak ismerjük az eddig elhasznált CPU-idejét, egyszerű kiszámolni az aktuálisan felhasznált CPU és a neki járó CPU arányát. A 0,5 arány azt jelenti, hogy a processzus csak a felét használta fel annak, mint amit felhasználhatott volna, a 2,0 arány pedig azt, hogy a processzus kétszer annyit használt fel, mint amennyi megillette volna. Az algoritmus ezután a legkisebb arányszámmal rendelkező processzust fogja futtatni, amíg az arányszám meg nem haladja a legközelebbi versenytársáét.

Sorsjáték-ütemezés

Ígéretet tenni a felhasználónak, és aztán betartani, remek ötlet ugyan, de nehéz megvalósítani. Egy másik algoritmust használva azonban hasonlóan megjósolható eredményt kapunk, de a megvalósítás sokkal könnyebb. Ezt **sorsjáték-ütemezés**nek nevezzük (Waldspurger és Weihl, 1994).

Az alapötlet az, hogy minden processzusnak sorsjegyet adunk a különböző rendszererőforrásokhoz, mint például a CPU-időhöz. Ha ütemezési döntést kell hozni, egy sorsjegyet véletlenszerűen kiválasztunk, és az a processzus kapja meg az erőforrást, amelynél a sorsjegy van. Amikor ezt CPU-ütemezésre használjuk, akkor a rendszer másodpercenként akár 50-szer is tarthat sorshúzást, a nyertes 20 ezred másodperc CPU-időt kap nyereményként.

George Orwell nyomán: „Minden processzus egyenlő, de néhány processzus egyenlőbb." A fontosabb processzusok többlet sorsjegyeket kaphatnak, hogy növeljék a nyerési esélyeiket. Ha 100 sorsjegyet adtunk ki, és egy processzusnak 20 van, akkor minden sorsolásnál 20% a nyerési esélye. Hosszú távon ez kb. 20% CPU-időt eredményez. A prioritásos ütemezéssel ellentétben, ahol nagyon nehéz megállapítani, hogy mit is jelent pontosan az, hogy a prioritás 40, itt a szabály világos: ha egy processzus a sorsjegyek f-ed részét birtokolja, akkor meg fogja kapni a kérdéses erőforrás kb. f-ed részét.

A sorsjáték-ütemezésnek számos érdekes tulajdonsága van. Például ha egy új processzus tűnik fel, és adunk neki néhány sorsjegyet, akkor a legközelebbi húzáson a sorsjegyei számának arányában van esélye a nyereményre. Más szavakkal a sorsjáték-ütemezéssel nagyon jó a válaszidő.

Együttműködő processzusok átadhatják egymásnak a sorsjegyeiket, ha akarják. Például ha egy kliens küld egy üzenetet a szervernek, és ezután blokkol, akkor átadhatja az összes sorsjegyét a szervernek, hogy megnövelje annak esélyeit a következő futásra. Amikor a szerver befejezte, visszaadja a sorsjegyeket a kliensnek, így az ismét futhat. Valójában kliensek hiányában a szervernek egyáltalán nincs szüksége sorsjegyekre.

A sorsjáték-ütemezéssel olyan problémákat is megoldhatunk, amelyeket más módszerekkel nehéz kezelni. Egy példa a videoszerver, amelyben több processzus videofolyamot állít elő kliensei részére, de különböző sebességgel. Tegyük fel, hogy a processzusoknak másodpercenként 10, 20 és 25 képkockára van szükségük. Azzal, hogy a processzusok létrehozásukkor 10, 20 és 25 sorsjegyet kapnak, automatikusan felosztják a CPU-időt a megfelelő, azaz 10 : 20 : 25 arányban.

Arányos ütemezés

Eddig feltételeztük, hogy minden processzust önmagában ütemezünk, tekintet nélkül arra, hogy ki a tulajdonosa. Ennek eredményeképpen, ha az 1-es felhasználó elindít 9 processzust, a 2-es felhasználó pedig 1 processzust, akkor round robin ütemezés és egyenlő prioritások esetén az 1-es felhasználó a CPU 90%-át kapja, a 2-es felhasználó pedig csak a 10%-át.

Ennek megakadályozására némelyik rendszer ütemezéskor figyelembe veszi, hogy ki a processzus tulajdonosa. Ebben a modellben minden felhasználó kap valamekkora hányadot a CPU-időből, és az ütemező úgy választja ki a folyamatokat, hogy ezt kikényszerítse. Tehát ha két felhasználónak a CPU 50-50%-a lett előirányozva, akkor ennyit fognak kapni attól függetlenül, hogy hány processzust futtatnak.

Például tekintsünk egy rendszert két felhasználóval, mindkettőnek 50% CPU-időt ígértünk. Az 1-es felhasználónak négy processzusa van, A, B, C és D, a 2-es felhasználónak csak egy, E. Round robin ütemezéssel egy lehetséges ütemezési sorozat, amelyik figyelembe veszi a keretfeltételeket:

A E B E C E D E A E B E C E D E...

Másrészről ha az 1-es felhasználónak kétszer annyi CPU-idő jár, mint a 2-esnek, akkor ezt kaphatjuk:

A B E C D E A B E C D E...

Természetesen sok más lehetőség van még, és ki is lehet használni attól függően, hogy mit tekintünk arányosnak.

2.4.4. Ütemezés valós idejű rendszerekben

Egy **valós idejű** rendszerben az idő alapvető szerepet játszik. Jellemzően egy vagy több külső fizikai eszköz ingert küld a számítógép felé, amire annak megfelelően reagálnia kell egy adott időn belül. Például egy CD-lejátszóban lévő számítógép biteket kap a meghajtótól, és ezeket nagyon rövid időn belül zenévé kell átalakítania. Ha a számolás túl sokáig tart, akkor a zene furcsán szól. Valós idejű rendszer lehet még a betegfigyelő rendszer egy kórház intenzív osztályán, a robotpilóta a repülőgépen vagy a robotvezérlő egy automatizált gyárban. Ezekben az esetekben egy túl későn kapott jó válasz gyakran ugyanolyan rossz, mintha egyáltalán nem kaptunk volna semmit.

A valós idejű rendszereket általában két kategóriába soroljuk: **szigorú valós idejű rendszerek**, ami azt jelenti, hogy abszolút határidők vannak, amelyeket kötelező betartani, és **toleráns valós idejű rendszerek**, ami azt jelenti, hogy néha egy-egy határidő elmulasztása nem kívánatos ugyan, de azért tolerálható. A valós idejű viselkedés mindkét esetben azzal érhető el, hogy a programot több processzusra osztjuk, mindegyiknek a viselkedése megjósolható és előre ismert. Ezek a processzusok általában rövid életűek, és jóval egy másodpercen belül befejeződhetnek. Külső esemény észlelésekor az ütemező feladata a processzusok olyan ütemezése, hogy minden határidő be legyen tartva.

Az események, amelyekre egy valós idejű rendszernek esetleg válaszolni kell, tovább csoportosíthatók: **periodikusak** (rendszeres intervallumonként fordulnak elő) és **aperiodikusak** (megjósolhatatlan az előfordulásuk). Lehet, hogy egy rendszernek több periodikus eseménysorozatot kell kezelnie. Attól függően, hogy mennyi idő szükséges az egyes események feldolgozásához, előfordulhat, hogy nem tudja mindet kezelni. Például ha m darab periodikus eseményünk van, és az i esemény periódusa P_i, kezelésére C_i másodperc CPU-időre van szükség, akkor csak abban az esetben kezelhetők a sorozatok, ha

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

Azokat a valós idejű rendszereket, amelyek ezt a feltételt teljesítik, **ütemezhetőnek** nevezzük.

Példaként tekintsünk egy toleráns valós idejű rendszert három periodikus eseménnyel, 100, 200 és 500 ezred másodperc periódussal. Ha ezek feldolgozása eseményenként rendre 50, 30 és 100 ezred másodperc CPU-időt igényel, akkor a rendszer ütemezhető, mert 0,5 + 0,15 + 0,2 < 1. Ha egy negyedik – 1 másodperces periódusú – eseményt hozzáveszünk, akkor a rendszer egészen addig ütemezhető marad, amíg ennek az eseménynek a feldolgozása nem igényel többet 150 ezred másodperc CPU-időnél. Ez a számítás közvetve feltételezi, hogy a környezetátkapcsolás költsége elhanyagolhatóan kicsi.

A valós idejű ütemezési algoritmusok dinamikusak vagy statikusak lehetnek. Az előbbi az ütemezési döntéseket futás közben hozza, az utóbbi a rendszer futásának megkezdése előtt. A statikus ütemezés csak akkor működik, ha előre teljes információnk van az elvégzendő feladatokról és a határidőkről is. A dinamikus algoritmusok esetében nincsenek ilyen korlátozások.

2.4.5. Elvek és megvalósítás

Mostanáig hallgatólagosan feltételeztük, hogy a rendszer minden processzusa különböző felhasználóhoz tartozik, és így verseng a CPU-ért. Bár ez sokszor igaz, néha mégis megtörténik, hogy egy processzusnak több gyermeke is van, amelyek a felügyelete alatt futnak. Például egy adatbázis-kezelő rendszer processzusnak több gyermeke lehet. A gyermekek dolgozhatnak különböző lekérdezéseken, vagy mindegyiknek lehet valamilyen speciális funkciója, amelyet végrehajt (lekérdezéselemzés, lemezelérés stb.). Lehetséges, hogy a főprocesszus pontosan tudja, hogy mely gyermekei a legfontosabbak (vagy időkritikusak), és melyek legkevésbé. Sajnos a korábban ismertetett ütemezők egyike sem fogad semmilyen információt a felhasználói processzusoktól az ütemezési döntésekhez. Ennek az az eredménye, hogy csak ritkán fordul elő, hogy az ütemező a legjobb döntést hozza meg.

A probléma megoldása az, hogy az **ütemezés megvalósítását** el kell választani az **ütemezési elvek**től. Ez azt jelenti, hogy az ütemezési algoritmust valamilyen módon paraméterezni kell, de a paramétereket a felhasználói processzusok tölthetik ki. Tekintsük ismét az adatbázis példát. Tegyük fel, hogy a kernel a prioritásos ütemezési algoritmust használja, de kínál egy rendszerhívást, amellyel a processzus beállíthatja (és megváltoztathatja) gyermekeinek prioritását. Így a szülő részletesen szabályozhatja gyermekei ütemezését, még akkor is, ha ő maga nem ütemez. Itt a megvalósítás a kernelben van, de az elveket a felhasználói processzus határozza meg.

2.4.6. Szálütemezés

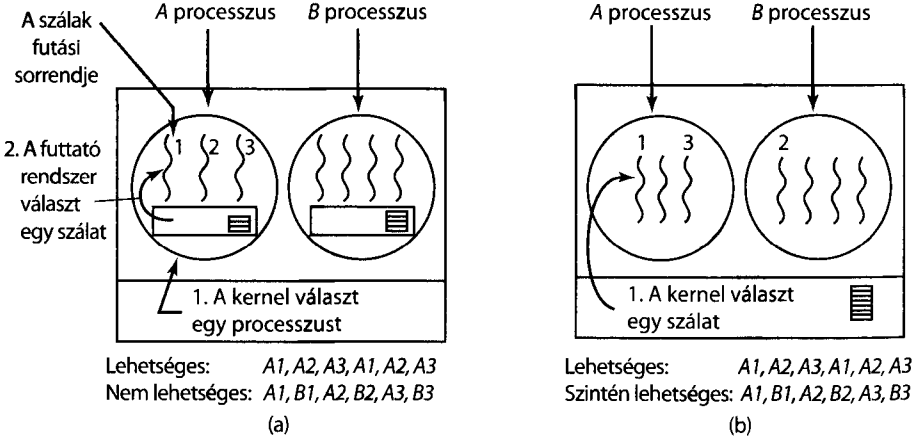
Amikor a processzusokon belül több végrehajtási szál van, akkor kétszintű párhuzamossággal állunk szemben: processzusok és szálak. Az ilyen rendszerekben az ütemezés alapvetően más attól függően, hogy felhasználói szintű vagy kernelszintű szálakat támogat-e a rendszer (esetleg mindkettőt).

Tekintsük először a felhasználói szintű szálakat. Mivel a kernel nem tud a szálak létezéséről, úgy működik, ahogy szokott. Kiválaszt egy processzust, mondjuk az *A*-t, és átadja neki a vezérlést egy időszelet erejéig. Az *A*-n belül a szálütemező dönti el, hogy melyik szál fusson, mondjuk az *A1*. Mivel a szálak párhuzamos futtatásához nincs időzítőmegszakítás, a kiválasztott szál addig futhat, ameddig akar. Ha elhasználja a processzus teljes időszeletét, akkor a kernel másik processzusra kapcsol át.

Amikor az *A* processzus végül újra futhat, akkor az *A1* szál fut tovább. Addig fogja használni az *A* összes idejét, amíg be nem fejeződik. Antiszociális viselkedése azonban nem érinti a többi processzust. Azok meg fogják kapni azt, amit az ütemező jogosnak tekint, függetlenül attól, hogy mi zajlik az *A* processzuson belül.

Most tekintsük azt az esetet, amikor az *A*-n belüli szálaknak viszonylag kevés tennivalójuk van, amikor rájuk kerül a sor, például 5 ezred másodpercnyi munka egy 50 ezred másodperces időszeletben. Emiatt mindegyik fut egy kicsit, majd visszaadja a vezérlést a szálütemezőnek. Ez például az *A1*, *A2*, *A3*, *A1*, *A2*, *A3*, *A1*, *A2*, *A3*, *A1* sorozathoz vezethet, mielőtt a kernel a *B* processzusra vált. Ez a helyzet a 2.28.(a) ábrán látható.

A futtató rendszer által használt ütemezési algoritmus a korábban tárgyaltak közül bármelyik lehet. A gyakorlatban a round robin ütemezés és a prioritásos



2.28. ábra. *(a) Felhasználói szintű szálak egy lehetséges ütemezése 50 ezred másodperces időszelet mellett, ha a szálak alkalmanként 5 ezred másodpercig futnak. (b) Kernelszintű szálak lehetséges ütemezése ugyanazon feltételek mellett*

ütemezés a leggyakoribb. Az egyedüli korlátozás az, hogy nincs időzítő, amellyel meg lehetne szakítani egy túl hosszan futó szálat.

Most tekintsük azt a helyzetet, amikor kernelszintű szálaink vannak. Ekkor a kernel választja ki, hogy melyik szál fusson. Nem kell figyelembe vennie, hogy a szál melyik processzushoz tartozik, de megteheti, ha akarja. A szál kap egy időszeletet, és erővel fel lesz függesztve, ha túllépné a kiszabott időt. Ha 50 ezred másodperces az időszelet, de a szálak blokkolódnak 5 ezred másodperc után, akkor a sorrend egy körülbelül 30 ezred másodperces időszakra lehet például *A1*, *B1*, *A2*, *B2*, *A3*, *B3*, ami viszont nem lehetséges ugyanezekkel a feltételekkel, ha felhasználói szintű szálak vannak. Ez a helyzet részlegesen a 2.28.(b) ábrán látható.

A felhasználói szintű és a kernelszintű szálak közötti nagy különbség a teljesítményben van. Felhasználói szinten egy szálváltás néhány gépi utasítással megoldható. Kernelszintű szálak esetén teljes környezetátkapcsolásra van szükség, memóriatérképet kell váltani, és érvényteleníteni a gyorsítótárat, ami nagyságrendekkel lassabb. Másrészről kernelszintű szálak esetében, ha egy szál I/O-művelet miatt blokkolódik, akkor nem blokkolja az egész processzust, mint felhasználói szintű szálak esetén.

Mivel a kernel tudja, hogy az *A* processzus egy száláról átváltani a *B* processzus egy szálára költségesebb, mint az *A* egy másik szálára (a memória térkép cseréje és a gyorsítótár tartalmának elvesztegetése miatt), ezt figyelembe veheti, amikor a döntéseit meghozza. Például ha van két szál, amelyek egyébként egyformán fontosak, de az egyik ugyanahhoz a processzushoz tartozik, mint az éppen blokkolódott szál, a másik pedig egy másik processzushoz, akkor az előző előnyt élvezhet.

Egy másik fontos szempont, hogy a felhasználói szintű szálak használhatnak alkalmazásfüggő szálütemezőt. Például tekintsünk egy webszervert egy diszpécserszállal, amely a beérkező kéréseket fogadja és kiosztja feldolgozószálaknak. Tegyük fel, hogy egy feldolgozószál éppen blokkolódott, és hogy a diszpécserszál és két további feldolgozószál kész a futásra. Melyik legyen a következő? A futtató rendszer ismeri az egyes szálak feladatait, és könnyen választhatja a diszpécsert, hogy az elindíthasson egy újabb feldolgozót. Ez a stratégia maximalizálja a párhuzamosságot olyan környezetben, ahol a feldolgozók gyakran blokkolódnak lemezműveletek miatt. Kernelszintű szálak esetén a kernel nem ismerheti a szálak tevékenységi körét (habár prioritást lehet hozzájuk rendelni). Általában azonban az alkalmazásfüggő szálütemezők jobban be tudják hangolni az alkalmazásokat, mint a kernel.

2.5. A MINIX 3-processzusok áttekintése

Miután tanulmányoztuk a processzuskezelés, a processzusok közötti kommunikáció és az ütemezés alapjait, áttekinthetjük, hogyan alkalmazzuk ezeket a MINIX 3 esetében. Míg a Unix magja monolitikus, azaz nincs modulokra bontva, addig a MINIX 3-processzusok együttese, melyek egymással és a felhasználói processzusokkal egyetlen kommunikációs alapmechanizmus segítségével tartanak kapcsolatot – üzenetküldéssel. Ez a felépítés sokkal modulárisabb és rugalmasabb szerkezetet eredményez, lehetővé teszi például, hogy akár az egész fájlrendszert lecseréljük egy másikra anélkül, hogy a kernelt újra kellene fordítani.

2.5.1. A MINIX 3 belső szerkezete

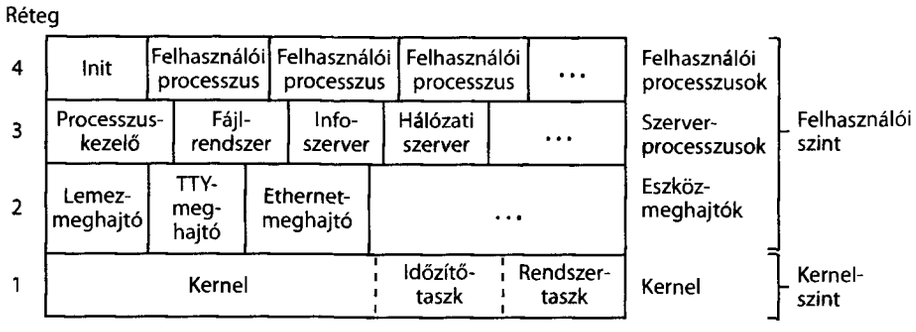
Kezdjük a MINIX 3 tanulmányozását a rendszer vázlatos áttekintésével. A MINIX 3 négy rétegből áll, ezek mindegyike egy jól meghatározott funkciót lát el. A négy réteg a 2.29. ábrán látható.

A **kernel** a legalsó rétegben ütemezi a processzusokat és kezeli a 2.2. ábrán látható futásra kész, futó és blokkolt állapotok közötti átmeneteket. Az üzenetkezeléshez szükséges a címzett érvényességének ellenőrzése, pufferek kialakítása a fizikai memóriában üzenetek küldése és fogadása esetén, valamint a bájtok átmásolása a küldőtől a címzetthez. A kernel része még az I/O-kapukhoz való hozzáférés és a megszakítások támogatása, amelyhez a modern mikroprocesszorokon az egyszerű processzusok számára nem elérhető, privilegizált **kernel módú** utasításokat kell használni.

A kernel mellett ez a réteg tartalmaz még két modult, amelyek az eszközmeghajtókhoz hasonlóan működnek. Az **időzítőtaszk** egy I/O-eszközmeghajtó abban az értelemben, hogy az időzítőjeleket generáló hardverrel áll kapcsolatban, de nem lehet hozzáférni úgy, mint egy lemez- vagy kommunikációs vonal meghajtójához – csak a kernelhez kapcsolódik.

Az 1-es réteg egyik legfőbb funkciója az, hogy elérhetővé teszi a **kernelhívások**at a fölötte elhelyezkedő eszközmeghajtóknak és szervereknek. Ezek között található az I/O-kapuk írása/olvasása, címtartományok közötti adatmozgatás stb. A hívások megvalósítását a **rendszertaszk** tartalmazza. Habár a rendszertaszk és az időzítőtaszk bele van fordítva a kernel címtartományába, külön processzusként ütemeződnek és saját hívási vermük van.

A kernel nagy része, az időzítő- és a rendszertaszk teljes egészében C-ben van megírva. A kernel egy kis része assembly kódban van. Az assembly kódú részek a



2.29. ábra. *A MINIX 3 négy rétegbe van rendezve. Csak a legalsó réteg processzusai használhatnak privilegizált (kernel módú) utasításokat*

megszakításkezeléssel, a processzusátkapcsolás alacsony szintű részleteivel (regiszterek mentése/visszatöltése és hasonlók) és az MMU- (Memory Management Unit – memóriakezelő egység) hardver alacsony szintű kezelésével foglalkoznak. Nagyjából az assembly kód a kernelnek azokat a részeit érinti, amelyek nagyon alacsony szinten közvetlenül a hardvert érik el, és nem lehet megírni C-ben. Ezeket a részeket újra kell írni, ha a MINIX 3-at új architektúrára akarjuk átvinni.

A kernel fölött elhelyezkedő három réteget egy rétegnek is tekinthetjük, mert alapvetően a kernel mindegyiket ugyanúgy kezeli. Mindegyik korlátozva van **felhasználói módú** utasításokra, és mindegyiket a kernel ütemezi. Egyik sem férhet hozzá közvetlenül az I/O-kapukhoz. Ezen túlmenően mindegyik csak a hozzárendelt memóriaszegmenseket érheti el.

Némelyik processzusnak azonban különleges kiváltságai vannak (például joga van kernelhívásokhoz). Ez a valódi különbség a 2-es, 3-as és 4-es rétegben elhelyezkedő processzusok között. A 2-es réteg processzusainak van a legtöbb kiváltsága, a 3-as rétegnek kevesebb, a 4-es rétegben lévőknek pedig egyáltalán nincs.

A 2-es rétegben lévő processzusok, az ún. **eszközmeghajtók (device driver),** kérhetik például a rendszertaszkot, hogy a nevükben adatot olvasson vagy küldjön valamelyik I/O-kapura. Minden eszköztípushoz külön meghajtóra van szükség, a lemezekhez, nyomtatókhoz, terminálokhoz és hálózati csatolókhoz. Ha más I/O-eszközök is vannak a rendszerben, akkor azokhoz is kell meghajtó. Az eszközmeghajtók más kernelhívásokat is igénybe vehetnek, például kérhetik, hogy az éppen beolvasott adatok egy másik processzus címtartományába másolódjanak át.

A 3-as réteg **szervereket** tartalmaz; ezek olyan processzusok, amelyek a felhasználói processzusok számára hasznos szolgáltatásokat nyújtanak. Két nélkülözhetetlen szerver van. A **processzuskezelő (process manager, PM)** hajtja végre mindazokat a MINIX 3-rendszerhívásokat, amelyek processzusok indításával/leállításával járnak, mint például a fork, exec és exit, illetve a szignálok kezelésével kapcsolatosakat, mint például az alarm és a kill, mert ezek megváltoztathatják a processzusok állapotát. A processzuskezelő a memória kezeléséért is felelős, például a brk rendszerhívás tartozik ide. A **fájlrendszer (file system, FS)** hajtja végre a fájlrendszerrel kapcsolatos összes rendszerhívást, mint a read, mount és chdir.

Fontos megérteni a kernelhívások és a POSIX-rendszerhívások közötti különbséget. A kernelhívások a rendszertaszk által nyújtott alacsony szintű funkciók, amelyek a taszkoknak és a szervereknek segítséget nyújtanak feladatuk elvégzésében. Egy hardver I/O-kapu olvasása tipikusan kernelhívást igénylő művelet. Ezzel szemben a POSIX read, fork és unlink magas szintű hívások, amelyeket a POSIX szabvány definiál; ezek a 4-es réteg felhasználói programjainak rendelkezésére állnak. A felhasználói programok sok POSIX-hívást tartalmaznak, de kernelhívást egyáltalán nem. Néha amikor nem fogalmazunk elég körültekintően, akkor kernelhívás helyett esetleg rendszerhívásról beszélünk. A kettőnek hasonló a mechanizmusa, és a kernelhívások a rendszerhívások speciális fajtájának tekinthetők.

A PM és FS mellett más szerverek is vannak a 3-as rétegben. Ezek MINIX 3 specifikus feladatokat végeznek. Azt kijelenthetjük, hogy processzuskezelő és fájlrendszer minden operációs rendszerben van. Az **információs szerver (IS)** nyomkövetési és állapotinformációt szolgáltat más meghajtókról és szerverekről. Ez fontosabb egy olyan operációs rendszerben, mint a MINIX 3, amely kísérletezésre készült, és kevésbé fontos egy kereskedelmi célú operációs rendszerben, amelyet a felhasználó nem módosíthat. A **reinkarnációs szerver (reincarnation server, RS)** elindít, és ha kell, újraindít olyan taszkokat, amelyek nem a kernellel együtt kerültek betöltésre. Konkrétan, a reinkarnációs szerver észreveszi, ha egy eszközmeghajtó hibásan működik, leállítja, ha még nem állt le, és egy új példányt indít, ezzel nagyban hibatűrővé téve a rendszert. A legtöbb operációs rendszerből hiányzik ez a funkcionalitás. Hálózatba kötött rendszerben az opcionális **hálózati szerver (inet)** is a 3-as rétegben helyezkedik el. A szerverek közvetlenül nem végezhetnek I/O-műveleteket, de kommunikálhatnak eszközmeghajtókkal, és I/O-műveleteket kérhetnek. A szerverek a kernellel is kommunikálhatnak a rendszertaszkon keresztül.

Ahogy az első fejezet elején említettük, az operációs rendszerek két dolgot tesznek: kezelik az erőforrásokat és kiterjesztik a gép funkcióit a rendszerhívások segítségével. A MINIX 3-ban az erőforrás-kezelést nagyrészt a 2-es rétegbeli meghajtók végzik, a kernelréteg besegít, amikor I/O-kapukhoz kell hozzáférni, vagy a megszakításrendszerre van szükség. A rendszerhívások értelmezését a processzuskezelő és a fájlrendszerszerverek végzik a 3-as rétegben. A fájlrendszer gondos tervezéssel „szerver"-ként lett kialakítva, ezért kis módosítással távoli gépre is áthelyezhető lenne.

Új szerver beillesztésekor a kernelt nem kell újrafordítani. A processzuskezelőt és a fájlrendszert ki lehet egészíteni egy hálózati szerverrel és más szerverekkel is, akár a MINIX 3 betöltésekor, vagy akár később is. Az eszközkezelők és a szerverek is közönséges végrehajtható állományként helyezkednek el a lemezen, de megfelelő módon indítva megkapják a működésükhöz szükséges privilégiumokat. Egy **szolgáltatás (service)** nevű felhasználói program nyújtja a felületet az ezt kezelő reinkarnációs szervernek. Jóllehet a meghajtók és a szerverek önálló processzusok, abban különböznek a felhasználói processzusoktól, hogy a rendszer működése alatt soha nem állnak le.

A 2-es és 3-as rétegben elhelyezkedő meghajtókat és szervereket együtt gyakran **rendszerprocesszus**ként fogjuk emlegetni. A rendszerprocesszusok nyilván az operációs rendszer részei. Nem tartoznak egyetlen felhasználóhoz sem, és jószerével mindegyik még azelőtt elindul, hogy az első felhasználó bejelentkezne. A felhasználói és rendszerprocesszusok közötti másik különbség, hogy a rendszerprocesszusok magasabb prioritással futnak. Valójában az eszközmeghajtóknak a szerverekénél magasabb prioritása van, de ez nem automatikus. A prioritás egyedileg kerül meghatározásra a MINIX 3-ban; lehetséges, hogy egy lassú eszközt kezelő meghajtó alacsonyabb prioritást kap, mint egy olyan szerver, amelynek gyorsan kell reagálnia.

Végül a 4-es réteg tartalmazza a felhasználói processzusokat – parancsértelmezőket, szövegszerkesztőket, fordítóprogramokat és a felhasználók által írt *a.out* programokat. Sok felhasználói processzus jön létre és szűnik meg, ahogy a felhasználók bejelentkeznek, dolgoznak és kijelentkeznek. A működő rendszer általában tartalmaz néhány olyan processzust, amelyek a rendszer indítása után

állandóan futnak. Ezek egyike az *init*, amelyet a következő részben fogunk leírni. Valószínűleg sok démon is futni fog. A **démon** egy olyan háttérprocesszus, amely periodikusan fut, vagy állandóan valamely esemény bekövetkezésére vár, például hálózati csomag érkezésére. Bizonyos értelemben a démon egy olyan szerver, amely önállóan indul, és felhasználói processzusként fut. A valódi, induláskor aktivizált szerverekhez hasonlóan a démonok konfigurálhatók úgy, hogy a közönséges felhasználói processzusokénál magasabb prioritást kapjanak.

A **taszk (task)** és az **eszközmeghajtó (device driver)** elnevezésekkel kapcsolatban egy megjegyzést kell tennünk. A MINIX régebbi verzióiban az összes meghajtó a kernellel együtt volt fordítva, ami elérhetővé tette számukra a kernel és egymás adatszerkezeteit. Az I/O-kapukat is közvetlenül elérhették. A nevük „taszk" volt, hogy meg lehessen különböztetni őket a sima felhasználói processzusoktól. A MINIX 3-ban az eszközmeghajtók teljesen a felhasználói területen lettek implementálva. Az egyedüli kivétel az időzítőtaszk, amely nyilván nem olyan eszközmeghajtó, mint azok, amelyeket a felhasználói processzusok eszközfájlokon keresztül elérhetnek. Arra törekedtünk, hogy a szövegben a „taszk" szó csak az időzítő- és a rendszertaszkkal összefüggésben forduljon elő, amelyek be vannak fordítva a kernelbe a megfelelő működés érdekében. Fáradságot nem kímélve lecseréltük a „taszk" előfordulásait „eszközmeghajtó"-ra, amikor felhasználói területen futó eszközmeghajtóról esik szó. A függvények és változók neveit, valamint a forráskódban található megjegyzéseket azonban nem írtuk át teljeskörűen. Így a MINIX 3 tanulmányozása közben előfordulhat, hogy a forráskódban a „task" szót találjuk ott, ahol „device driver"-nek kellene állni.

2.5.2. Processzuskezelés a MINIX 3-ban

A MINIX 3-processzusok megfelelnek annak az általános modellnek, amelyet e fejezet korábbi részében részletesen ismertettünk. A processzusok létrehozhatnak alprocesszusokat, ezek szintén létrehozhatnak újabb alprocesszusokat, így egy processzusfához jutunk. Valójában a rendszer minden felhasználói processzusa része annak a processzusfának, amelynek gyökere az *init* (lásd 2.29. ábra). A szerverek és az eszközmeghajtók természetesen speciális esetek, mert némelyiket az összes felhasználói processzus előtt kell elindítani, beleértve az *init*-et is.

A MINIX 3 elindulása

Hogyan indul egy operációs rendszer? A MINIX 3 indulási lépéseit a következő néhány oldalon összefoglaljuk. Néhány másik operációs rendszer indulásával kapcsolatban lásd (Dodge et al., 2005).

A legtöbb lemezegységgel ellátott számítógépen van egy **indítólemez- (boot disk)** hierarchia. Jellemzően, ha van lemez a hajlékonylemezes meghajtóban, az lesz az indítólemez. Ha nincs hajlékonylemez, de van CD a meghajtóban, akkor a CD lesz az indítólemez. Ha se hajlékonylemez, se CD nincs, akkor az első merevlemez lesz az indítólemez. A sorrend átállítható is lehet, ha bekapcsolás után belépünk a BIOS-ba. Más, különösen hordozható eszközöket is meg lehet adni.

Amikor a számítógépet bekapcsoljuk, és az indítólemez egy hajlékonylemez, akkor a hardver beolvassa a memóriába az indítólemez első sávjának első szektorát, és végrehajtja az ott található programot. Hajlékonylemez esetében a kérdéses szektor az **indító- (bootstrap)** programot tartalmazza. Ez egy nagyon rövid program, mivel el kell férnie egyetlen szektorban (512 bájt). A MINIX 3-indítóprogram betölti a nagyobb *boot* programot, amely aztán betölti magát az operációs rendszert.

Az előzőkkel ellentétben a merevlemezek egy közbenső lépést igényelnek. Egy merevlemez **partíciók**ra van osztva, az első szektor egy rövid programot és a lemez **partíciós táblá**ját tartalmazza. Ezeket együtt **elsődleges indítórekord**nak **(master boot record)** nevezzük. A programrész végrehajtásra kerül, ennek során kiolvassa a partíciós táblát és beállítja az **aktív partíció**t. Az aktív partíció első szektora tartalmaz egy indítóprogramot, amely betöltése és elindítása után megkeresi és futtatja a partíció *boot* programját ugyanúgy, mint a hajlékonylemez esetében.

A CD-ROM-ok később jelentek meg, mint a hajlékonylemezek és a merevlemezek. Ha CD-ről lehet indítani a rendszert, akkor több lehetőség van, mint egyetlen szektor betöltése. CD-ROM-ról induláskor a számítógép egy nagy adatblokkot képes azonnal a memóriába tölteni. Rendszerint a CD-ről egy indítólemez tartalmának pontos másolatát töltik a memóriába, ezt a területet **RAM-lemez**ként **(RAM disk)** használja a rendszer a továbbiakban. Az első lépés után a vezérlés a RAM-lemezre adódik át, és minden pontosan úgy történik, mintha fizikailag egy hajlékonylemez lenne az indítólemez. Olyan régebbi számítógépeken, amelyekben van ugyan CD-meghajtó, de nem lehet róluk rendszert indítani, a CD-ről az indítólemez tartalmát egy hajlékonylemezre kell másolni, amelyet aztán indítólemezként lehet használni. A CD-nek természetesen a meghajtóban kell lennie, mert a hajlékonylemezes indításkor szükség van rá.

Minden esetben a *boot* a hajlékonylemezen vagy a partícióban megkeres egy több részből álló állományt, és az egyes részeket a memória megfelelő részeibe tölti. Ez a **betöltési memóriakép (boot image).** A legfontosabb részek a kernel (ez tartalmazza az időzítőtaszkot és a rendszertaszkot), a processzuskezelő és a fájlrendszer. Ezenkívül még legalább egy lemezmeghajtót is be kell tölteni a memóriakép részeként. Sok más program is van még, közöttük a reinkarnációs szerver, a RAM-lemez, konzol, naplózó meghajtók és az *init*.

Hangsúlyozni kell, hogy a betöltési memóriakép részei különálló programok. Miután az alapvető kernel, processzuskezelő és fájlrendszer betöltődött, a rendszer további részeit egyenként be lehet tölteni. Kivétel a reinkarnációs szerver; ennek a memóriakép részének kell lennie. Ez ad az inicializáció után betöltött közönséges processzusoknak különleges jogokat, hogy rendszerprocesszusok lehessenek. A valamilyen hiba miatt működésképtelenné vált eszközmeghajtókat is újra tudja indítani, erre utal a neve is. Ahogy korábban említettük, legalább egy lemezmeghajtó mindenképpen szükséges. Ha a gyökérfájlrendszert RAM-lemezre akarjuk másolni, akkor a memóriameghajtóra is szükség van, egyébként később is betölthető. A *tty* és a *log* meghajtók opcionálisak a betöltési memóriaképben.

Komponens	Leírás	Betöltő
kernel	Kernel + időzítő- és rendszertaszk	(betöltési memóriaképben van)
pm	Processzuskezelő	(betöltési memóriaképben van)
fs	Fájlrendszer	(betöltési memóriaképben van)
rs	(Újra)indítja a szervereket és meghajtókat	(betöltési memóriaképben van)
memory	RAM-lemezmeghajtó	(betöltési memóriaképben van)
log	Puffereli a naplózási üzeneteket	(betöltési memóriaképben van)
tty	Konzol- és billentyűzetmeghajtó	(betöltési memóriaképben van)
driver	Lemez- (at, bios, hajlékonylemez) meghajtó	(betöltési memóriaképben van)
init	Az összes felhasználói processzus szülője	(betöltési memóriaképben van)
floppy	Hajlékonylemez-meghajtó	/etc/rc
is	Információs szerver (nyomkövetési listákhoz)	/etc/rc
cmos	A CMOS-órát olvassa az idő beállításához	/etc/rc
random	Véletlenszám-generátor	/etc/rc
printer	Nyomtatómeghajtó	/etc/rc

2.30. ábra. *Néhány fontos MINIX 3-rendszerkomponens. Egyéb komponensek, mint például Ethernet-meghajtó vagy az inet szerver is szerepelhetnek itt*

Azért töltődnek be korán, mert hasznos, ha üzeneteket tudunk megjeleníteni a konzolon, és naplózni a történéseket már a betöltés minél koraibb szakaszában. Az *init* később is betölthető lenne, de ez vezérli a rendszer kezdeti konfigurálását, és a betöltési memóriaképben volt a legegyszerűbb elhelyezni.

Az indítás nem triviális művelet. A *boot* programnak kell végrehajtania az egyébként a lemeztaszk és a fájlrendszer hatáskörébe tartozó műveleteket, mielőtt ez utóbbiak aktivizálódnak. Később még visszatérünk a MINIX 3 elindítására. Egyelőre elég annyi, hogy miután a betöltés befejeződött, a kernel elkezd futni.

Az inicializálás során a kernel elindítja a taszkokat, majd a processzuskezelőt, a fájlrendszert és minden más, a 3-as rétegben futó szervert. A processzuskezelő és a fájlrendszer ezután együttműködnek a betöltési memóriaképben lévő más szerverek és meghajtók elindításában. Ezek elindulnak és beállítják kezdeti állapotukat, majd blokkolnak, és várnak arra, hogy valamilyen tennivalójuk legyen. A MINIX 3-ütemezés prioritásokat rendel a processzusokhoz. Amikor az összes taszk és szerver blokkolt állapotban van, elindul az *init,* az első felhasználói processzus. A betöltési memóriaképben lévő és az inicializáció során elindított rendszerkomponenseket a 2.30. ábra mutatja.

A processzusfa inicializációja

Az **init** a legelső felhasználói processzus, de legutolsóként töltődik be a betöltési memóriaképből. Azt gondolhatnánk, hogy az 1.5. ábrán láthatóhoz hasonló processzusfa felépítése azonnal megkezdődik, ahogy az *init* elindul. Nem egészen így van. Igaz lenne egy hagyományos operációs rendszerben, de a MINIX 3 más. Először is, már jó néhány rendszerprocesszus fut, mire az *init* elindul. A *CLOCK*

(időzítő-) és a *SYSTEM* (rendszer-) taszkok különleges processzusok, a kernelen belül futnak, és kívülről nem látszanak. Nincs PID-jük, és nem részei semmilyen processzusfának. A processzuskezelő az első felhasználói processzus; a PID-je 0, és nem gyermeke és nem is szülője egyetlen más processzusnak sem. A reinkarnációs szerver lesz a szülője az összes többi, a betöltési memóriaképben található processzusnak (például szerverek, meghajtók). Emögött az a logika, hogy a reinkarnációs szervernek értesülnie kell róla, ha az előbbiek közül valamelyiket újra kell indítani.

Ahogy látni fogjuk, még az *init* indulása után is vannak különbségek a MINIX 3 és a hagyományos processzusfa-építési módszerek között. A Unix-szerű rendszerekben az *init* PID-je 1, és habár a MINIX 3-ban az *init* nem az első elindított processzus, a hagyományos 1-es PID-t megkapja. Mint az összes, a betöltési memóriaképben elhelyezkedő felhasználói processzus (a processzuskezelő kivételével), az *init* a reinkarnációs szerver egyik gyermeke lesz. A szabványos Unix-rendszerekhez hasonlóan az *init* először az **/etc/rc** parancsfájlt hajtja végre. Ez további eszközmeghajtókat és szervereket indít, amelyek nem részei a betöltési memóriaképnek. Az *rc* parancsfájl által indított összes program az *init* gyermeke lesz. Az elsők egyike a *service* nevű segédprogram. A *service* is az *init* gyermeke, ahogy az várható. De itt a dolgok megint a megszokottól eltérően alakulnak.

A *service* a reinkarnációs szerver felhasználói interfésze. A reinkarnációs szerver elindít egy közönséges programot, amelyet aztán rendszerprocesszussá változtat. Elindítja a *floppy*-t (ha betöltés közben nem volt rá szükség) a *cmos*-t (amelyre a valós idejű óra kiolvasásához van szükség), és az *is*-t, az információs szervert; ez azokat a nyomkövetési listákat kezeli, amelyeket a konzolbillentyűzet funkcióbillentyűinek (F1, F2 stb.) lenyomásakor kapunk. A reinkarnációs szerver egyik ténykedése, hogy a processzuskezelő kivételével gyermekeiként örökbe fogadja az összes rendszerprocesszust.

Miután a *cmos* eszközmeghajtó elindult, az *rc* parancsfájl inicializálhatja a valós idejű órát. Eddig a pontig a szükséges fájloknak azon az eszközön kell lenniük, ahol a gyökérfájlrendszer is van. A kezdetben szükséges meghajtók és a szerverek az */sbin* könyvtárban vannak; az indításhoz szükséges többi program a */bin*-ben. Az indítás első lépéseinek befejeződése után megtörténik a többi fájlrendszer felcsatolása. Az *rc* parancsfájl egyik fontos feladata annak ellenőrzése, hogy előző rendszerleállások nem okoztak-e esetleg fájlrendszerhibákat. Az ellenőrzés egyszerű – ha a rendszer rendben áll le a *shutdown* parancs végrehajtásával, akkor a */usr/adm/wtmp* bejelentkezési naplófájlba bekerül egy bejegyzés. A `shutdown -C` parancs ellenőrzi, hogy a *wtmp* utolsó bejegyzése megfelelő-e. Ha nem, akkor feltételezi, hogy nem szabályszerűen állt le a rendszer, és az *fsck* segédprogrammal ellenőrzi a fájlrendszereket. Az */etc/rc* utolsó feladata a démonok elindítása. Ez történhet kiegészítő parancsfájlokkal is. Ha megnézzük a `ps axl` parancs outputját, amelyen a PID-k és a szülő PID-k (PPID-k) is látszanak, akkor láthatjuk, hogy a démonok, mint például az *update* és a *usyslogd,* az elsők között vannak az állandó processzusok csoportjában, amelyek az *init* gyermekei.

Legvégül az *init* a potenciális termináleszközök listáját tartalmazó */etc/ttytab* állományt olvassa be. A bejelentkező terminálként szóba jöhető eszközök (a szabvá-

nyos disztribúcióban ez csak a fő konzol és még legfeljebb három virtuális konzol, de soros vonali és hálózati terminálokat is fel lehet venni) esetében az */etc/ttytab* tartalmaz bejegyzést a *getty* mezőben, és az *init* minden ilyen terminál számára elindít egy alprocesszust. Normál esetben minden ilyen alprocesszus a */usr/bin/getty* programot futtatja, amely egy üzenet kiírása után egy név begépeléséig várakozik. Ha valamelyik terminál speciális elbánást igényel (például telefonvonali összeköttetés esetén), akkor az */etc/ttytab* olyan programot (például */usr/bin/stty*) is megadhat, amely elvégzi a vonal kezdeti állapotának beállítását a *getty* futtatása előtt.

Amikor a felhasználó begépelte az azonosítóját, akkor ezzel a névvel mint argumentummal meghívódik a */usr/bin/login* program. A *login* eldönti, hogy kell-e jelszó, ha igen, akkor bekéri és ellenőrzi. Sikeres bejelentkezés után a *login* elindítja a felhasználó parancsértelmezőjét (ez alapértelmezésben a */bin/sh,* de az */etc/passwd* állományban mást is be lehet állítani). A parancsértelmező parancsok begépelésére várakozik, és elindít egy alprocesszust minden egyes parancs számára. Így minden parancsértelmező az *init* gyermeke, a felhasználói processzusok az *init* unokái, és az összes felhasználói processzus egyetlen processzusfában helyezkedik el.

Tulajdonképpen a kernelbe fordított taszkok és a processzuskezelő kivételével a processzusok, a rendszerprocesszusok és a felhasználói processzusok is, egyetlen fát alkotnak. De a hagyományos Unix-rendszer processzusfájától eltérően nem az *init* a fa gyökere, és a fa szerkezetéből nem lehet megállapítani, hogy a rendszerprocesszusok milyen sorrendben lettek elindítva.

A két legfontosabb, processzusok kezelésére szolgáló MINIX 3-rendszerhívás a fork és az exec. A fork az egyetlen módja annak, hogy új processzust hozzunk létre. Az exec segítségével egy processzus végrehajthat egy megadott programot. A program végrehajtása közben a programfájl fejlécében meghatározott mennyiségű memóriával rendelkezik. Ez a mennyiség futás közben végig változatlan, habár az adatszegmens, a veremszegmens és a szabad memória közötti arányok változhatnak.

A processzusokról minden információt a processzustábla tartalmaz, amelyet a kernel, a processzuskezelő és a fájlrendszer közösen használ, mindegyik a számára szükséges mezőket birtokolja. Egy új processzus létrejöttekor (fork hatására) vagy egy létező processzus befejeződésekor (exit vagy megszakítás hatására) a processzuskezelő először beállítja a saját mezőit a processzustáblában, majd üzenetet küld a fájlrendszernek és a kernelnek, hogy azok is tegyenek hasonlóképpen.

2.5.3. Processzusok közötti kommunikáció a MINIX 3-ban

Üzenetek küldésére és fogadására három elemi művelet áll rendelkezésre. Ezek az alábbi C könyvtári eljárásokkal hívhatók:

```
send(dest, &message);
```

üzenetet küld a *dest* processzusnak,

```
receive(source, &message);
```


üzenetet fogad a *source* processzustól (*ANY* esetén bármelyiktől), és

```
sendrec(src_dst, &message);
```

üzenetet küld egy processzusnak, majd várakozik, hogy ugyanaz a processzus választ küldjön. Mindhárom esetben a második paraméter az üzenet lokális címe. A kernelben lévő üzenetküldő mechanizmus az üzenetet a küldőtől a fogadóhoz másolja. A válasz (*sendrec* esetében) felülírja az eredeti üzenetet. Elvileg ezt a kernelmechanizmust le lehetne cserélni egy olyanra, amely hálózaton keresztül az üzeneteket eljuttatja egy másik géphez, osztott rendszert megvalósítva. Gyakorlatban ezt egy kicsit bonyolítaná az, hogy az üzenetek gyakran tartalmaznak nagyobb adatszerkezetekre mutató pointert, és egy osztott rendszernek az egész adatszerkezet átvitelét meg kellene oldania.

Minden taszk, eszközmeghajtó és szerverprocesszus csak bizonyos meghatározott processzusokkal válthat üzeneteket. Később tárgyaljuk annak részleteit, hogy ezt hogyan lehet kikényszeríteni. Az üzenetek általában a 2.29. ábra rétegeiben felülről lefelé haladnak, és egymásnak az egy rétegben vagy szomszédos rétegben lévő processzusok üzenhetnek. A felhasználói processzusok nem küldhetnek egymásnak üzenetet. A 4-es rétegben lévő felhasználói processzusok kezdeményezhetnek üzenetküldést a 3-as réteg szerverei felé, a 3-as réteg szerverei pedig kezdeményezhetnek üzenetküldést a 2-es réteg eszközmeghajtói felé.

Ha egy processzus üzenetet küld egy olyan processzusnak, amely éppen nem vár üzenetre, akkor a küldő blokkolódik, amíg a fogadó végre nem hajt egy receive hívást. Más szóval, a MINIX 3 a randevúeljárást használja abból a célból, hogy elejét vegye a már elküldött, de még nem fogadott üzenetek pufferelésében adódó problémáknak. Ennek a megközelítésnek az az előnye, hogy egyszerű, és nincs szükség pufferkezelésre (beleértve azt is, hogy nem fogyunk ki a pufferekből). Ezen túlmenően az üzenetek hossza kötött, fordítási időben kerül meghatározásra, ezért nem következhet be puffertúlírás sem, ami egyébként a programhibák gyakori oka.

Az üzenetváltásokra bevezetett korlátozások alapvető célja az, hogy ha az *A* processzusnak megengedjük, hogy send vagy sendrec hívást kezdeményezzen a *B* processzus felé, akkor a *B*-nek engedélyezhessünk receive hívást, amelyben *A*-t jelöli meg küldőként, de *B* ne küldhessen *A*-nak. Világos, hogy ha *A* blokkolódik, amikor küldeni próbál *B*-nek, és *B* is blokkolódik, amikor küldeni próbál *A*-nak, akkor holtpontba jutottunk. Az „erőforrás", amelyre mindkettőnek szüksége volna a művelet befejezéséhez, nem fizikai jellegű, mint egy I/O-eszköz, hanem a címzett egy receive hívása. A holtpontról bővebben a 3. fejezetben lesz szó.

Néha egy blokkoló üzenetküldés helyett másra van szükség. Van még egy fontos üzenetkezelő alapművelet, amelyet a

```
notify(dest);
```

C könyvtári függvénnyel hívhatunk, és arra használható, hogy a címzett figyelmét felhívjuk arra, hogy valamilyen fontos esemény bekövetkezett. A notify nem blokkoló, ami azt jelenti, hogy a küldő folytatja a futását függetlenül attól, hogy a címzett várakozik-e az értesítésre. Mivel nem blokkol, holtpontot sem okozhat.

Az értesítés az üzenetküldési mechanizmust felhasználva kerül kézbesítésre, de az átadható információ korlátozott. Általános esetben az ilyen üzenet csak a küldő azonosítóját és egy kernel által hozzáadott időbélyegzőt tartalmaz. Néha ez minden, amire szükség van. Például a billentyűzet notify-t használ, amikor valamelyik funkcióbillentyűt (F1-től F12-ig, illetve ugyanezek SHIFT-tel együtt) leütik. A MINIX 3-ban a funkcióbillentyűkkel nyomkövetési listák készítését lehet kezdeményezni. Az Ethernet-meghajtó olyan processzusra példa, amely csak egyfajta nyomkövetési listát generál, és más üzenetet soha nem is kell kapnia a konzolmeghajtótól. Így a DUMP-ETHERNET-STATS billentyű leütésekor a konzoltól az Ethernet-meghajtóhoz érkező értesítés félreérthetetlen. Más esetekben egy értesítés nem elég, de annak megérkezése után a címzett küldhet egy üzenetet az értesítés feladójának, további információt kérve.

Van oka annak, hogy az értesítések ilyen egyszerűek. Mivel a notify nem blokkol, akkor is használható, amikor a címzett még nem ért a receive-hez. De az egyszerűsége lehetővé teszi, hogy a nem kézbesíthető értesítés könnyen tárolható legyen, majd a címzettet azonnal értesíthetjük, amint a receive-et meghívta. Tulajdonképpen egyetlen bit elég. Az értesítéseket arra szántuk, hogy egymás között használják a rendszerprocesszusok, ezekből pedig aránylag kevés van. Minden rendszerprocesszusnak van egy bittérképe a kézbesítetlen értesítésekhez, 1 bit tartozik minden rendszerprocesszushoz. Így ha az *A* processzus olyankor akar értesítést küldeni a *B* processzusnak, amikor az nincs blokkolva egy receive-nél, akkor az üzenetkezelő rendszer *B* bittérképében beállítja az *A*-nak megfelelő bitet. Amikor a *B* később végrehajt egy receive-et, akkor az első teendő a kézbesítetlen értesítések bittérképének ellenőrzése. Ilyen módon több forrásból megkísérelt értesítésekről is tudomást szerezhet. Az egyetlen bit elegendő az értesítés információtartalmának előállításához. Meghatározza a küldő kilétét, a kernel üzenetkezelő kódja pedig a kézbesítéskor hozzáteszi az időbélyegzőt. Az időbélyegzők elsősorban időzítők lejártának ellenőrzésére használatosak, így nincs nagy jelentősége, ha az időbélyegző egy kicsit későbbi időpontot tartalmaz annál, mint amikor a küldő először próbálkozott az értesítés elküldésével.

Vannak még további részletei is az értesítési mechanizmusnak. Bizonyos esetekben az értesítési üzenet egy további mezője is használatos. Ha az értesítés egy megszakítás bekövetkeztéről informálja a címzettet, akkor az összes lehetséges megszakításforrás bittérképe is bekerül az üzenetbe. Ha pedig a küldő a rendszertaszk, akkor még az összes, a címzett részére még kézbesítetlen szignál bittérképe is része lesz az üzenetnek. Természetesen felmerül a kérdés, hogyan lehet mindezeket a kiegészítő információkat elküldeni egy olyan processzusnak, amely éppen nem akar üzenetet fogadni. A válasz az, hogy ezek az információk kernel-adatszerkezetekben vannak tárolva. Nem kell semmit másolni ahhoz, hogy megőrződjenek. Ha egy értesítést el kell halasztani, és egyetlen bitté kell zsugorítani, akkor abban az időpontban, amikor a címzett végrehajtja a receive utasítást, és az értesítést újra kell generálni, az értesítés forrása alapján lehet tudni, hogy milyen többletinformációt kell elhelyezni az üzenetben. A fogadó fél számára ugyancsak a küldő kiléte mondja meg, hogy számítson-e többletinformációra, illetve ha igen, akkor azt hogyan kell értelmezni.

Van még néhány, a processzusok közötti kommunikációhoz tartozó alapművelet. Ezekről a későbbiekben fogunk szót ejteni. Kevésbé fontosak, mint a send, a receive, a sendrec és a notify.

2.5.4. Processzusok ütemezése a MINIX 3-ban

Egy multiprogramozott operációs rendszert a megszakításrendszer tart életben. A bevitelt kérő processzusok blokkolódnak, így más processzusok is lehetőséget kapnak a futásra. Amikor a kért adat rendelkezésre áll, az éppen futó processzust megszakítja a lemez-, billentyűzet- vagy más hardver. Az időzítő szintén állít elő megszakításokat; ezek arra használatosak, hogy a bevitelt nem kérő felhasználói processzusok is átadják végül a CPU-t más processzusoknak. A MINIX 3 legalsó rétegének feladata a megszakítások elrejtése olyan módon, hogy üzenetekké alakítja őket. A processzusok (és taszkok) szempontjából az I/O-eszközök a műveletek befejezésekor üzenetet küldenek valamelyik processzusnak, felélesztve és futtathatóvá téve azt.

Megszakítások szoftverből is generálódnak, ebben az esetben ezeket gyakran **csapdá**nak nevezik. Az előzőkben leírt send és receive hívások a rendszerkönyvtár által **szoftvermegszakítás**sá alakulnak át; ezek hatása pontosan megegyezik a hardvermegszakításokéval – a szoftvermegszakítást végrehajtó processzus azonnal blokkolódik, a kernel pedig megkezdi a megszakítás feldolgozását. A felhasználói programok nem hivatkoznak közvetlenül a send-re és a receive-re, hanem valahányszor az 1.9. ábra valamelyik rendszerhívása aktivizálódik, akár közvetlenül, akár valamelyik könyvtári függvény útján, akkor a sendrec hívódik meg, és egy szoftvermegszakítás generálódik.

Valahányszor egy processzus futása közben megszakítás érkezik (akár egy I/O-eszköz, akár az időzítő miatt), vagy szoftvermegszakítás miatt felfüggesztődik, lehetőség adódik megvizsgálni, hogy melyik processzus érdemes leginkább a futásra. Ezt természetesen a processzusok befejeződésekor is meg kell tenni, de a MINIX 3-hoz hasonló rendszerekben az I/O-eszköz vagy időzítő miatti megszakítások sokkal gyakoribbak, mint a processzusbefejeződések.

A MINIX 3-ütemező egy többszintű sorban állásos rendszert alkalmaz. Várakozósorból 16 van, de újrafordítással ennél több vagy kevesebb is könnyen beállítható. A legalacsonyabb prioritású soron csak az *IDLE* processzus van, amely akkor fut, amikor nincs semmi más teendő. A felhasználói processzusok alapértelmezés szerint a legalsónál jó néhány sorral feljebb kerülnek.

A szerverek alapesetben a felhasználói processzusok számára engedélyezettnél magasabb prioritású sorokba kerülnek, az eszközmeghajtók a szervereknél magasabb, az időzítő- és a rendszertaszk pedig a legmagasabb prioritásúba. Nem valószínű, hogy bármikor is a 16 sor mindegyike egyszerre használatban legyen. Processzusok csak némelyikben indulnak. Egy processzust a rendszer vagy (bizonyos korlátokkal) a felhasználó a *nice* parancs használatával áthelyezhet másik prioritási sorba. A sok sor lehetőséget ad a kísérletezésre, illetve ahogy további eszközmeghajtók kerülnek be a MINIX 3-ba, az alapértelmezés szerinti beállítá-

sok hangolhatók a legjobb teljesítmény elérése érdekében. Például ha a hálózaton digitális audio- vagy videoadást szolgáltató szerverre lenne szükség, akkor egy ilyen szervernek a jelenlegieknél magasabb kezdeti prioritást lehetne adni, vagy egy jelenlegi szerver, vagy meghajtó kezdeti prioritását lehetne csökkenteni, hogy az új szerver jobb teljesítményt nyújthasson.

A processzus ütemezési sora által meghatározott prioritás mellett más mechanizmus is segít bizonyos processzusokat abban, hogy egy kis előnyre tegyenek szert a többivel szemben. Az időszelet, vagyis az egy processzus által egyszerre felhasználható időintervallum, nem ugyanakkora minden processzus esetén. A felhasználói processzusoknak viszonylag rövid az időszeletük. Az eszközmeghajtóknak és a szervereknek alapesetben addig kellene futniuk, amíg blokkolódnak. Az esetleges hibás működés elleni védekezésként azonban ezek is megszakíthatók, de hosszú időszeletet kapnak. Hosszú ideig futhatnak, de ha felhasználták a teljes időszeletüket, akkor felfüggesztődnek, nehogy lefagyasszák a rendszert. Ilyen esetben az időtúllépés miatt felfüggesztett processzust futásra késznek kell tekinteni, és a várakozósor végére lehet helyezni. Ha azonban kiderül, hogy az időt túllépő processzus ugyanaz, amely legutóbb futott, akkor ez annak a jele lehet, hogy beragadt egy ciklusba, és akadályozza az alacsonyabb prioritásúakat a futásban. Ebben az esetben a prioritása csökkentésre kerül úgy, hogy egy alacsonyabb prioritású sor végére kerül át. Ha a processzus megint túllépi az idejét, és másik processzus még mindig nem tudott futni, akkor megint lejjebb kerül. Előbb vagy utóbb el kell jutnunk addig a pontig, hogy valami más is futhat.

Egy prioritásban lejjebb léptetett processzusnak van lehetősége visszatérni magasabb prioritású sorba. Ha egy processzus felhasználja a teljes időszeletét, de nem akadályoz más processzusokat a futásban, akkor visszakerül a számára engedélyezett legmagasabb prioritáshoz tartozó sorba. Egy ilyen processzusnak láthatólag szüksége van a teljes időszeletére, és tekintettel van másokra is.

Máskülönben a processzusok ütemezése egy kismértékben módosított round robin módszerrel történik. Ha egy processzus nem használta fel a teljes időszeletét, akkor ezt úgy tekintjük, hogy I/O-művelet miatt blokkolódott, és amikor újra futásra kész állapotba kerül, akkor a sora elejére kerül, de úgy, hogy csak annyi időt használhat, amennyi az időszeletéből még megmaradt. Emögött az a szándék húzódik meg, hogy a felhasználói processzusok gyorsan tudjanak reagálni az I/O-eseményekre. Az időszeletét teljesen felhasználó processzus a sora végére kerül az eredeti round robin ütemezésnek megfelelően.

Mivel rendszerint a taszkoknak van a legmagasabb prioritása, majd az eszközmeghajtók, ezután a szerverek, végül a felhasználói processzusok következnek. A felhasználói processzusok csak akkor futhatnak, ha egyetlen rendszerprocesszusnak sincs dolga, és egy rendszerprocesszust nem akadályozhat meg a futásban egyetlen felhasználói processzus sem.

Új processzus kiválasztásakor az ütemező ellenőrzi, hogy van-e futtatható processzus a legmagasabb prioritású sorban. Ha van legalább egy ilyen, akkor a sor elején lévőt futtatja. Ha nincs ilyen, akkor egy sorral lejjebb végzi el ugyanezt az ellenőrzést, és így tovább. Mivel az eszközmeghajtók a szerverektől érkező kérésekre reagálnak, a szerverek pedig a felhasználói processzusoktól érkező kérések

hatására aktivizálódnak, idővel minden magas prioritású processzus be kell hogy fejezze a munkát, amit kértek tőle. Ezután blokkolódnak mindaddig, amíg a felhasználói processzusok megint lehetőséget kapnak a futásra, és további kéréseket generálnak. Ha nincs futtatható processzus, akkor az *IDLE* (tétlen) processzus kerül kiválasztásra. Ez alacsony fogyasztású üzemmódba kapcsolja a CPU-t a következő megszakítás beérkezéséig.

Valahányszor az időzítő megszakítást idéz elő, a rendszer megvizsgálja, hogy az éppen futó processzus olyan felhasználói processzus-e, amely már hosszabb ideje fut, mint a hozzárendelt időszelet. Ha igen, akkor az ütemező a sor végére teszi (lehet, hogy semmit nem kell csinálni vele, mert egyedül van a soron). Ezután a következő processzus kiválasztása a fent leírtaknak megfelelően történik. Az aktuális processzus csak akkor folytathatja a futását, ha egyedül van a során, és a magasabban lévő sorokban nincs egyetlen processzus sem. Máskülönben a legmagasabb prioritású, nem üres sor elején lévő processzus futhat. A nélkülözhetetlen eszközmeghajtók és szerverek olyan hosszú időszeletet kapnak, hogy alapesetben soha nem az időzítő miatt szakad meg a futásuk. Ha azonban valami elromlik, akkor a prioritásuk ideiglenesen csökkenhet, hogy ne fogják meg a rendszert teljesen. Valószínűleg semmi érdemlegeset nem lehet tenni, ha ilyesmi egy létfontosságú szerverrel történik meg, de legalább a rendszert szabályosan le lehet állítani, amivel meg lehet előzni az adatvesztést, és esetleg információt lehet gyűjteni annak kiderítéséhez, hogy mi okozta a problémát.

2.6. Processzusok megvalósítása MINIX 3-ban

Közeledünk a tényleges programkód megvizsgálásához, ezért érdemes néhány szót ejteni a használt jelölésekről. Az „eljárás", „függvény" és „rutin" kifejezéseket azonos értelemben fogjuk használni. A változók, eljárások és állományok nevei dőlt betűkkel szerepelnek, mint például *rw_flag*. Ezek a nevek valójában mind kisbetűsek, ha azonban a szövegben mondat elejére kerülnek, akkor nagybetűvel írjuk őket. Van néhány kivétel, a kernelbe fordított eszközmeghajtók nevei nagybetűsek, mint például *CLOCK*, *SYSTEM* és *IDLE*. A rendszerhívások Helvetica stílusú kisbetűvel fognak szerepelni, mint például read.

A MINIX 3 teljes verziója a mellékelt CD-ROM-on található. A MINIX 3 weboldaláról *(www.minix3.org)* letölthető az aktuális verzió, amelyben új funkciók, kiegészítő szoftver és dokumentáció is található.

2.6.1. A MINIX 3 forráskódjának szerkezete

Az ebben a könyvben bemutatott MINIX 3-implementáció IBM PC típusú, korszerű 32 bites processzorral (például 80386, 80486, Pentium, Pentium Pro, II, III, 4, M vagy D) ellátott számítógépre készült. Mindezekre Intel 32 bites processzorként fogunk hivatkozni. Egy szabványos Intel-alapú rendszerben a teljes C forrás-

kód a */usr/src/* könyvtárban van (az elérési utak végén szereplő „/" jelzi, hogy ezek könyvtárak). Más platformok esetén a forráskód könyvtárai máshol is lehetnek. A könyvben a MINIX 3 forráskódfájljaira mindvégig az *src/* alatt kezdődő elérési úttal hivatkozunk. Egy fontos alkönyvtár az *src/include/*, ahol a C definíciós fájlok mesterpéldányai találhatók. Erre a könyvtárra *include/*-ként hivatkozunk.

Minden könyvtár tartalmaz egy **Makefile** nevű fájlt, amely a szabványos Unix *make* segédprogram működését irányítja. A *Makefile* vezérli a saját könyvtárában található fájlok fordítását, és az is előfordulhat, hogy egyes alkönyvtáraiban is irányítja a fordítást. A *make* működése összetett, teljes leírása meghaladja ennek a szakasznak a kereteit, de összefoglalható úgy, hogy a *make* megoldja a több forrásfájlból álló programok hatékony lefordítását. A *make* biztosítja, hogy minden szükséges fájl lefordításra kerüljön. Ellenőrzi a korábban fordított modulokat, és újrafordítja azokat, amelyek forráskódjában változás történt az utolsó fordítás óta. Időt lehet megtakarítani azzal, hogy feleslegesen nem fordít le egyetlen fájlt sem. Végül, a *make* irányítja a külön lefordított modulok végrehajtható programmá szerkesztését, és esetleg az elkészült program telepítését is elvégzi.

Az *src/* könyvtárfa egésze vagy bármelyik része is áthelyezhető, mivel a könyvtárak *Makefile*-jai relatív elérési utakat használnak a többi C forráskönyvtárhoz. Például a gyors fordítás érdekében egy RAM-lemez gyökérkönyvtárában is el lehet helyezni *(/src/)*. Ha speciális változatot fejlesztünk, akkor az *src/* egy másolatát más néven is létrehozhatjuk.

A C definíciós fájlok elérési útja speciális eset. Fordítás közben minden *Makefile* feltételezi, hogy a definíciós fájlokat a */usr/include/* könyvtárban találja (vagy ennek megfelelő másikban, ha nem Intel-platformra történik a fordítás). Az */src/tools/Makefile* azonban azt feltételezi, hogy (Intel-platform esetén) a */usr/src/include/* könyvtárban megtalálja a definíciós fájlok mesterpéldányait. A teljes rendszer újrafordítása előtt azonban először a teljes */usr/include/* törlődik, majd a */usr/src/include/* átmásolódik a */usr/include/*-ba. Erre azért volt szükség, hogy a MINIX 3 fejlesztéséhez szükséges összes fájlt egy helyen lehessen tartani. Ez a megoldás azt is megkönnyíti, hogy a forráskódot és a definíciós fájlokat tároló könyvtárfákból több példányt tarthassunk egymás mellett, ha a MINIX 3-rendszer különböző beállításaival szeretnénk kísérletezni. Figyelni kell azonban arra, hogy ha szerkeszteni akarunk egy kísérlethez tartozó fájlt, akkor azt az */src/include/* alatt tegyük, és ne a */usr/include/* alatt.

Ez megfelelő alkalom, hogy a C nyelvben járatlanok figyelmébe ajánljuk a fájlnevek megadási módját az #include direktíván belül. Minden C fordítónak van egy alapértelmezés szerinti definíciós könyvtára, ahol a definíciós fájlokat keresi. Ez gyakran a */usr/include/*. Ha egy beilleszteni kívánt fájl neve kisebb és nagyobb jelek közé van zárva (<...>), akkor a fordítóprogram a fájlt az alapértelmezés szerinti könyvtárban vagy annak egy alkönyvtárában keresi, például az

```
#include <fájlnév>
```

a */usr/include/*-ból veszi a fájlt.

Sok programnak olyan definíciókra is szüksége van helyi definíciós fájlokban, amelyeket nem kell az egész rendszer számára elérhetővé tenni. Egy ilyen definíciós fájlnak a neve lehet ugyanaz, mint egy szabványos definíciós fájlnak, és elképzelhető, hogy éppen annak leváltására vagy kiegészítésére szánták. Amikor a fájl neve szokásos idézőjelek között van ("..."), akkor a fordítóprogram először abban a könyvtárban keresi, amelyikben a forrásfájl is van, majd ha ott nem találja, akkor az alapértelmezés szerinti könyvtárban. Így az

```
#include "fájlnév"
```

egy lokális fájlt illeszt be.

Az *include/* könyvtár számos szabványos POSIX definíciós fájlnak (POSIX header files) ad helyet. Ezen túl három alkönyvtárat tartalmaz:

sys/	– ez a könyvtár további szabványos POSIX definíciós fájlokat tartalmaz;
minix/	– a MINIX 3 operációs rendszer által használt definíciós fájlokat tartalmazza;
ibm/	– csak IBM PC esetén használt definíciós fájlokat tartalmaz.

Elősegítendő a MINIX 3-rendszer és a programok fejlesztését, további állományok és könyvtárak találhatók az *include/* könyvtárban, a CD-n, és elérhetők a MINIX 3 weboldalán is. Például az *include/arpa/*, az *include/net/* és ennek alkönyvtára, az *include/net/gen/* a hálózati kiterjesztéseket támogatja. Ezek nem szükségesek a MINIX 3-alaprendszer lefordításához.

Az *src/include/* mellett az *src/* könyvtár három fontos alkönyvtárban tartalmazza az operációs rendszer forráskódját:

kernel/	– 1-es réteg (ütemezés, üzenetek, időzítő- és rendszertaszk).
drivers/	– 2-es réteg (lemez, konzol, nyomtató stb. eszközmeghajtók).
servers/	– 3-as réteg (processzuskezelő, fájlrendszer, egyéb szerverek).

Van három további könyvtár, amelyek egy működő rendszerhez alapvető fontosságúak, de a könyvben nem tárgyaljuk:

src/lib/	– könyvtári eljárások forráskódja (például open, read).
src/tools/	– Makefile és parancsfájlok a MINIX 3 fordításához.
src/boot/	– a MINIX 3 betöltésére és telepítésére szolgáló programkód.

A MINIX 3 alapváltozata számos olyan további forrásfájlt tartalmaz, amelyeket a könyvben nem tárgyalunk. A processzuskezelő és fájlrendszer forráskódja mellett az */src/servers/* rendszerkönyvtár tartalmazza az *init* program és az *rs* reinkarnációs szerver forráskódját, mindkettő nélkülözhetetlen része a futó MINIX 3-rendszernek. A hálózati szerver forráskódja az */src/servers/inet/*-ben van.

Az */src/drivers/* a könyvben nem tárgyalt meghajtók forráskódját tartalmazza, többek között alternatív lemezmeghajtókhoz, hangkártyákhoz és hálózati kártyákhoz.

Mivel a MINIX 3 egy kísérleti operációs rendszer, az *src/test/* könyvtár olyan programokat tartalmaz, amelyek az operációs rendszert módosítás és újrafordítás után alaposan letesztelik. Egy operációs rendszer természetesen azért van, hogy parancsokat (programokat) futtassunk, itt van mindjárt a méretes *src/commands/* könyvtár, amely a kisegítőprogramok forráskódját tartalmazza (például *cat, cp, date, ls, pwd* és még több mint 200 másik). Néhány nagy, eredetileg a GNU- és a BSD-projektek keretében kifejlesztett nyílt forráskódú alkalmazás forráskódja is megtalálható itt.

A továbbiakban az egyszerűség kedvéért általában csak a fájlneveket fogjuk használni, ha a szövegkörnyezetből világos, hogy mi a teljes elérési út. Meg kell azonban jegyezni, hogy néhány fájlnév több könyvtárban is szerepel. Például több *const.h* nevű fájl is van. Az *src/kernel/const.h* a kernelben használt konstansokat definiálja, míg az *src/servers/pm/const.h* a processzuskezelő által használt konstansokat definiálja stb.

Az egy könyvtárba tartozó állományokat együtt tárgyaljuk, így nem kell félreértéstől tartani. A CD-n és a weboldalon a teljes forráskód megtalálható, és mindkét helyen a függvények, definíciók és globális változók megtalálását segítő tárgymutatót is elhelyeztünk.

A Függelék F.3. alfejezete tartalmazza a CD-n található fájlok neveit ábécésorrendben, definíciós fájlok, eszközmeghajtók, kernel, fájlrendszer és processzuskezelő részekkel. A Függelék ezen része, valamint a weboldal és a CD tárgymutatója a forráskódbeli sorszámmal hivatkozik az egyes tételekre.

Az 1-es réteg kódját az *src/kernel/* könyvtár tartalmazza. Ebben a könyvtárban olyan fájlok vannak, amelyek a processzusok kezelését támogatják; ezek a MINIX 3 legalsó rétegében helyezkednek el, ahogy a 2.29. ábrán is láthattuk. E réteg feladata a rendszerinicializálás, megszakításkezelés, üzenetküldés és processzusütemezés. Ezekhez szorosan kapcsolódik két olyan modul, amelyekkel fordítás után egy tárgymodulba kerülnek, de azok önálló folyamatként futnak. Egyik a rendszertaszk, amely a kernel szolgáltatásai és a felsőbb rétegek közötti interfészt biztosítja, a másik pedig az időzítőtaszk, amely időzítőjelekkel látja el a kernelt. A 3. fejezetben tanulmányozzuk az *src/drivers/* több alkönyvtárában található állományokat, amelyek a 2.29. ábra 2-es rétegének eszközmeghajtóit tartalmazzák. A 4. fejezetben a processzuskezelőhöz tartozó állományokat tekintjük át az *src/servers/pm/* könyvtárban, az 5. fejezetben pedig a fájlrendszer kerül terítékre az *src/servers/fs/* könyvtárban.

2.6.2. A MINIX 3 fordítása és futtatása

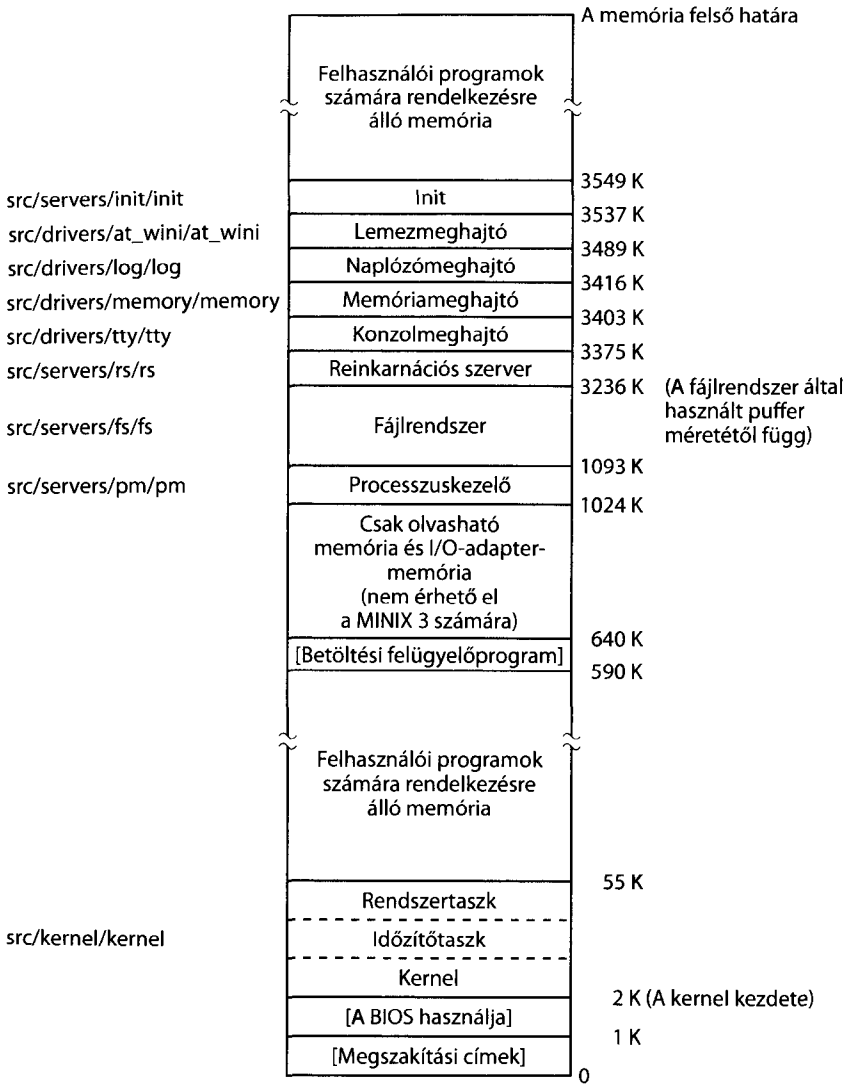
A MINIX 3 fordításához az *src/tools/*-ban kell indítani a make-et. Számos opció adható meg, ezek segítségével a MINIX 3 sokféleképpen telepíthető. Ha a make-et argumentumok nélkül indítjuk, akkor kilistázza a lehetőségeket. A legegyszerűbb módszer a make image.

A make image futtatásakor az *src/include/*-ból a definíciós fájlok friss példányai átmásolódnak az */usr/include/*-ba. Ezután az *src/kernel/*, valamint az *src/servers/* és az *src/drivers/* alkönyvtáraiban található forrásállományokból tárgykódot állítunk elő. Az *src/kernel/* tárgykódjait összeszerkesztve kapjuk a végrehajtható *kernel* programot. Az *src/servers/pm/* és *src/servers/fs/* tárgykódjaiból kapjuk a *pm,* valamint az *fs* programokat. A 2.30. ábrán a betöltési memóriakép részeként feltüntetett többi program is lefordítódik és összeszerkesztődik a saját könyvtárában. Ezek között van az *rs* és az *init* az *src/servers/* és *memory/* alkönyvtáraiban, a *log/,* valamint a *tty/* az *src/drivers/* alkönyvtáraiban. A 2.30. ábra „driver" sorában található komponens több lemezmeghajtó valamelyike lehet; most egy merevlemezről indítható MINIX 3-rendszert tárgyalunk, amely a szabványos *at_wini* eszközmeghajtót használja; ez az *src/drivers/at_wini/*-ben található. Más meghajtók is hozzáadhatók a rendszerhez, de a legtöbbjüket nem kell a betöltési memóriaképbe belefordítani. Ugyanez igaz a hálózati támogatásra is; az alap MINIX 3-rendszer fordítása szempontjából a hálózatos részek használata mellékes.

Egy betölthető MINIX 3-rendszer telepítéséhez az *installboot* program (ennek forrása az *src/boot/*-ban található) neveket ad a *kernel,* a *pm,* az *fs,* az *init* és a többi komponenshez, az állományokat a könnyebb betöltés érdekében kiegészíti úgy, hogy hosszuk a lemez szektorhosszának többszöröse legyen (hogy könnyebb legyen a részeket egymástól függetlenül betölteni), majd összefűzi őket egy közös állományba. Ez a fájl a betöltési memóriakép, ezt másolhatjuk rá egy hajlékonylemezre vagy egy merevlemez-partícióra a */boot/* vagy a */boot/image/* könyvtárba. Később a betöltési felügyelőprogram ezt be tudja tölteni, és a vezérlést át tudja adni az operációs rendszernek.

A 2.31. ábra mutatja a memória kiosztását, miután az egyes részek külön-külön betöltődtek. A kernel a memória alsó részébe töltődik, a betöltési memóriakép összes többi része 1 MB fölé. A felhasználói programok futtatásakor a kernel fölötti memória lesz elsőként felhasználva. Ha egy új program ott nem fér el, akkor a memória felső részébe, az *init* fölé kerül. A részletek természetesen a rendszer konfigurációjától függenek. A 2.31. ábra egy olyan MINIX 3-konfigurációt mutat, amelyben a fájlrendszer 512 darab 4 KB-os lemezblokk tárolására képes gyorsítótárral rendelkezik. Ez nem túl sok; nagyobb javasolt, ha van elég memória. Másrészt ha a lemezblokkok gyorsítótárát jóval kisebbre vennénk, akkor az egész rendszer elférne 640 K-ban, még néhány felhasználói processzusnak is maradna hely.

Világosan kell látnunk, hogy a MINIX 3 több teljesen független, egymással kizárólag üzenetküldéssel kommunikáló programból áll. Az *src/servers/fs/* és *src/servers/pm/* könyvtárban lévő *panic* eljárások nem ütköznek, mivel végül különböző végrehajtható állományba kerülnek. Csak néhány olyan könyvtári eljárás van az *src/lib/* könyvtárban, amelyet az operációs rendszer mindhárom része tartalmaz. Ez a moduláris szerkezet nagyon megkönnyíti, hogy például a fájlrendszert úgy módosítsuk, hogy az ne legyen kihatással a processzuskezelőre. Az is lehetséges, hogy a fájlrendszert teljes egészében eltávolítsuk, és egy másik gépen helyezzük el, ott fájlszerverként a kliensgépekkel hálózaton keresztül üzenetekkel kommunikálhat.



2.31. ábra. *A memória kiosztása, miután a MINIX 3 betöltődött a lemezről. A kernel, a szerverek és az eszközmeghajtók külön fordított és szerkesztett programok; ezeket a bal oldalon láthatjuk. A méretek megközelítők és nem arányosak*

A MINIX 3 modularitására másik példa, hogy a processzuskezelőt, a fájlrendszert és a kernelt egyáltalán nem érinti, hogy a rendszert hálózati támogatással vagy anélkül fordítjuk-e le. Egy Ethernet-meghajtó és az inet szerver is aktiválható a betöltés után; a 2.30. ábrán az */etc/rc* által indított processzusok között jelennének meg, és a 2.31. ábra valamelyik „felhasználói programok számára rendelkezésre álló memória" régiójába kerülnének. A hálózati funkciók engedélyezésével indított MINIX 3-rendszer távoli terminálként vagy ftp- és webszerverként is használható. A könyvben leírtak szerinti MINIX 3-rendszert csak akkor kell módosítani, ha meg akarjuk engedni a hálózaton keresztüli bejelentkezést: a módosítandó rész a *tty*, a konzol eszközmeghajtója, amelyet a távoli bejelentkezésekhez a virtuális terminálok használatát engedélyezve újra kell fordítani.

2.6.3. A közös definíciós fájlok

Az *include/* könyvtár és alkönyvtárai számos olyan állományt tartalmaznak, amelyek konstansok, makrók és típusok definícióit tartalmazzák. Ezen definíciók nagy részének meglétét és helyét *(includc/ és include/sys/)* a POSIX szabvány előírja. Ahogy a *.h* kiterjesztés jelzi, ezek az ún. **definíciós** fájlok (vagy állományok), és a C forrásprogramba az #include direktíva segítségével illeszthetők be. Ezek a direktívák a C nyelv beépített eszközei. A definíciós állományok megkönnyítik a nagy rendszerek karbantartását.

A felhasználói programok fordításához gyakran szükséges definíciós fájlok az *include/* könyvtárban, míg az elsődlegesen a rendszer részét képező programok fordításához használt definíciós fájlok az *include/sys/* könyvtárban helyezkednek el. Ez a megkülönböztetés nem olyan borzasztóan fontos, egy tipikus fordítás mindkét könyvtárt használja, legyen az akár egy felhasználói program vagy az operációs rendszer részének fordítása. Azokat az állományokat vesszük most sorra, először az *include/*, majd az *include/sys/* könyvtárból, amelyek az alap MINIX 3-rendszer fordításához szükségesek. A következő részben az *include/minix/* és az *include/ibm/* könyvtárakkal ismerkedünk meg. Ezek, mint a nevük is mutatja, kifejezetten a MINIX 3-rendszerhez, illetve annak IBM PC-n (valójában Intel-alapú PC-n) történő megvalósításához kötődnek.

Az elsők között néhány olyan általános célú definíciós fájl van, amelyeket közvetlenül a MINIX 3 egyetlen C nyelvű forrásprogramja sem használ. Ehelyett ezek más definíciós fájlokba kerülnek beillesztésre. A MINIX 3 mindegyik nagy kom-

```
#include <minix/config.h>      /* MUST be first – Ennek KELL lennie az elsőnek */
#include <ansi.h>              /* MUST be second – Ennek KELL lennie a másodiknak */
#include <limits.h>
#include <errno.h>
#include <sys/types.h>
#include <minix/const.h>
#include <minix/type.h>
#include <minix/syslib.h>
#include "const.h"
```

2.32. ábra. *Az elsődleges definíciós fájlokban megtalálható kódrészlet, amely biztosítja az összes C forrásprogram számára szükséges definíciós fájlok beillesztését. Figyeljük meg, hogy két* const.h *is szerepel, az egyik az* include/ *fából, a másik az aktuális könyvtárból kerül beillesztésre*

ponensének van egy elsődleges definíciós állománya; ezek az *src/kernel/kernel.h,* az *src/servers/pm/pm.h* és az *src/servers/fs/fs.h,* amelyek minden fordítás során beillesztésre kerülnek. Az eszközmeghajtók forráskódja is tartalmaz egy valamennyire hasonló fájlt; ez az *src/drivers/drivers.h*. Minden elsődleges definíciós fájl a megfelelő MINIX 3-komponenshez van összeállítva, de az első néhány soruk a 2.32. ábrán látható részlethez hasonló, és az ott látott fájlok többségét beilleszti. Az elsődleges definíciós fájlok később még előkerülnek. Ez a kis előretekintés csak azt szeretné hangsúlyozni, hogy különböző könyvtárakban található definíciós fájlokat együtt használunk. Ebben és a következő szakaszban a 2.32. ábrán látható állományok mindegyikét megemlítjük.

Az *include/* könyvtárban az első fájl az *ansi.h* (0000. sor).* A MINIX 3-rendszer bármelyik részének fordításakor ez a fájl kerül az *include/minix/config.h* után másodiknak beillesztésre. Az *ansi.h* célja az, hogy ellenőrizze, vajon a fordítóprogram megfelel-e a Nemzetközi Szabványügyi Hivatal C nyelvre vonatkozó szabványának. A szabványos C nyelvet néha ANSI C-nek is nevezik, mert a szabványt eredetileg az Amerikai Szabványügyi Hivatal (American National Standards Institute) készítette, majd később vált nemzetközileg elfogadottá. Egy szabványos C fordító számos olyan makrót definiál, amelyek a fordítás alatt álló programokban tesztelhetők. Az *__STDC__* egy ilyen makró, a szabványos fordítóprogram ennek 1 értéket ad, mintha az előfeldolgozó az alábbi sort olvasta volna

```
#define __STDC__ 1
```

A MINIX 3 jelenlegi változataiban található fordító megfelel a szabványnak, de korábbi változatait a szabvány elfogadása előtt fejlesztették ki, és lehetőség van arra, hogy a rendszert egy klasszikus (Kernighan & Ritchie) C fordítóval fordítsuk. Szándékunk szerint a MINIX 3-nak könnyen átvihetőnek kell lennie új gépekre, ennek az erőfeszítésnek része a régebbi fordítók támogatása is. A 0023. és 0025. sor közötti

```
#define _ANSI
```

definíció akkor kerül feldolgozásra, ha szabványos fordítót használunk. Az *ansi.h* számos makrót különbözőképpen definiál attól függően, hogy az *_ANSI* makró definiált-e, vagy sem. Ez egy példa **ellenőrző makró**ra.

Egy másik itt definiált ellenőrző makró a *_POSIX_SOURCE* (0065. sor). Ezt a POSIX követeli meg. Itt gondoskodunk a definiálásáról, ha valamelyik másik makró miatt a POSIX szabványnak meg kell felelni.

C programok fordításakor a függvények által fogadott argumentumok és a visszatérési értékek típusainak ismertnek kell lennie ahhoz, hogy az ilyen adatokra hivatkozó programkód lefordítható legyen. Egy összetett rendszerben nehéz a függvénydefiníciókat olyan sorrendben elhelyezni, hogy ennek a követelménynek

* Az itt megadott számok a CD mellékleten található MINIX 3-forráskód soraira vonatkoznak.

eleget tegyünk, ezért a C lehetővé teszi a **függvények prototípusá**nak megadását, vagyis a függvények argumentumainak és a visszatérési érték típusának **deklarálás**át azelőtt, hogy a függvényt **definiál**nánk. Ebben az állományban a legfontosabb makró a *_PROTOTYPE*, amely lehetővé teszi, hogy a függvények prototípusát

```
_PROTOTYPE (return-type function-name, (argument-type argument, ...))
```

formában írjuk, amit egy szabványos fordító előfeldolgozója

```
return-type function-name(argument-type argument, ...)
```

formára alakít, míg egy régi vágású (vagyis Kernighan–Ritchie-féle) fordítóprogram esetén

```
return-type function-name()
```

alakot kapunk.

Mielőtt az *ansi.h* tárgyalását befejezzük, térjünk ki még egy jellegzetességre. Az egész fájl (a kezdő megjegyzéseket kivéve)

```
#ifndef _ANSI_H
```

és

```
#endif /* _ANSI_H */
```

sorok közé van zárva.

Az #ifndef sor utáni sorban rögtön következik az _ANSI_H definíciója. Egy definíciós fájl minden fordítás során csak egyszer kerülhet beillesztésre; az előbbi konstrukció biztosítja, hogy a fájl tartalmát a fordító figyelmen kívül hagyja, amennyiben az többször is beillesztésre kerülne. Az *include/* könyvtárban található összes definíciós fájl is ilyen módon védett.

Két részletre érdemes kitérnünk ezzel kapcsolatban. Egyrészt az elsődleges definíciós állományok könyvtáraiban található fájlok elején az #ifndef ... #define sorozatban a fájl neve ki van egészítve egy aláhúzásjellel. Ugyanilyen nevű definíciós fájl lehet a C forráskód más könyvtáraiban is, és ugyanezt a módszert használtuk ott is, de aláhúzásjel nélkül. Az elsődleges definíciós fájl beillesztése nem akadályozza meg az ugyanolyan nevű definíciós fájl beillesztését egy lokális könyvtárból. Másrészt, figyeljük meg, hogy az #endif után az /* _ANSI_H */ megjegyzés nem kötelező. Ilyen megjegyzések használatával javíthatjuk az egymásba ágyazott #ifndef ... #endif és #ifdef ... #endif szakaszok átláthatóságát. De az ilyen megjegyzésekkel vigyázni kell: ha hibásak, akkor rosszabbak, mint ha egyáltalán nem is lennének.

Az *include/* könyvtárból a második, minden MINIX 3-forrásállományba közvetett módon beillesztett definíciós fájl a *limits.h* (0100. sor). Ez a fájl definiál alap-

vető C nyelvi méreteket, mint például a bitek száma egész mennyiségekben, és az operációs rendszerhez kapcsolódó korlátokat is, mint például a fájlnevek hossza.

Az *errno.h* (0200. sor) szintén szerepel majdnem az összes elsődleges definíciós fájlban. Ez tartalmazza azokat a hibakódokat, amelyeket a sikertelen rendszerhívások adnak vissza a felhasználói programoknak a globális *errno* változóban. Az *errno* néhány belső hiba azonosítására is szolgál, ilyen például, ha egy nem létező taszknak találnánk üzenetet küldeni. A rendszeren belül nem lenne hatékony, ha mindig meg kellene vizsgálni egy globális változót, valahányszor olyan függvényt hívunk meg, amely hibázhat, de a függvényeknek gyakran kell visszaadniuk egyéb egész értékeket, például I/O-művelet esetén az átvitt bájtok számát. A MINIX 3 megoldása erre a problémára az, hogy a hibakódokat negatív értékek reprezentálják a rendszeren belül, de a felhasználói programok számára pozitívvá alakítjuk őket. Erre azt a trükköt alkalmazzuk, hogy a hibakódokat az alábbihoz hasonló módon definiáljuk (0236. sor):

```
#define EPERM (_SIGN 1)
```

Az operációs rendszer részeinek elsődleges definíciós állományai definiálják a *_SYSTEM* makrót, de a felhasználói programok nem. Ha *_SYSTEM* definiált, akkor *_SIGN* értéke „–" lesz, különben nem kap értéket.

Az állományok következő csoportja nem kerül be minden elsődleges definíciós fájlba, de azért a MINIX 3-rendszer sok forrásállományában megtalálhatók. A legfontosabb az *unistd.h* (0400. sor), amely számos, a POSIX által megkövetelt konstans definícióját tartalmazza. Ezenkívül egy sor C függvény prototípusa is itt található, többek között a MINIX 3-rendszerhívások elérésére szolgáló függvényeké is. Egy másik sokat használt fájl a *string.h* (0600. sor), amely a karakterláncok kezelését végző C függvények prototípusainak ad helyet. A *signal.h* (0700. sor) tartalmazza a szabványos szignálok definícióját. Definiálásra került több olyan szignál is, amelyre csak a MINIX 3-nak van szüksége. Mivel monolitikus kernel helyett az operációs rendszer részfunkcióit egymástól független processzusok valósítják meg, ezért a rendszerkomponensek között szükség van egy, a szignálkezeléshez hasonló speciális kommunikációs mechanizmusra. A *signal.h* szignálok kezelésével kapcsolatos függvények prototípusait is tartalmazza. A későbbiekben ki fog derülni, hogy a szignálok kezelése a MINIX 3 minden részét érinti.

Az *fcntl.h* (0900. sor) számos, fájlművelethez szükséges szimbolikus nevet definiál. Például lehetővé teszi, hogy az *open* hívásakor az *O_RDONLY* makrót használjuk a 0 numerikus paraméter helyett. Habár erre az állományra a fájlrendszer hivatkozik a legtöbbször, a benne szereplő definíciók a kernel és a processzuskezelő számára is fontosak.

Ahogy a 3. fejezetben a taszkok tárgyalása során látni fogjuk, egy operációs rendszer konzol- és terminálcsatoló komponense meglehetősen összetett, mert az operációs rendszernek és a felhasználói programoknak sokféle hardvereszközzel kell együttműködniük szabványosított módon. A *termios.h* (1000. sor) a terminál jellegű I/O-eszközök kezeléséhez szükséges konstansokat, makrókat és függvényprototípusokat definiál. A legfontosabb a *termios* struktúra. Ez üzemmódjelző biteket,

adatátviteli sebességet meghatározó változókat és egy tömböt tartalmaz, utóbbi olyan speciális karakterek tárolására szolgál, mint az *INTR* vagy a *KILL*. Ez a struktúra számos makróval és függvényprototípussal együtt a POSIX szabvány része.

Egy mégoly teljességre törekvő szabvány, mint a POSIX, sem nyújt mindent, amire vágyunk, így a fájl utolsó része az 1140. sortól kezdve POSIX-kiterjesztéseket tartalmaz. Ezek egy részének haszna nyilvánvaló, mint például az 57 600 és ennél nagyobb baudértékek, valamint a terminálokon használható képernyőablakok támogatása. A POSIX szabvány nem tiltja a kiterjesztéseket, egy ésszerű szabvány úgysem lehet mindent magába foglaló. Ha azonban más környezetben is használható, hordozható programokat akarunk írni, akkor óvatosnak kell lennünk, és el kell kerülnünk a MINIX 3-kiterjesztések használatát. Ezt könnyen megtehetjük, mert a kiterjesztések definíciói minden állományban egy

```
#ifdef _MINIX
```

direktívával védettek. Ha a *_MINIX* nincs definiálva, akkor a fordító nem is látja a MINIX 3-kiterjesztéseket; teljesen figyelmen kívül hagyja őket.

A felügyeleti időzítők támogatására a *timers.h* (1300. sor) szolgál, amelyet a kernel elsődleges definíciós állománya is felhasznál. Definiál egy *struct timer* adattípust, valamint időzítők listáján működő függvények prototípusait. Az 1321. sorban találjuk a *tmr_func_t* típus definícióját *(typedef);* ez egy függvényre mutató pointer. Az 1332. sorban láthatjuk a felhasználását: egy *timer* struktúra időzítők listájának elemeként tartalmaz egy *tmr_func_t*-t, amely az időzítő lejártakor kerül meghívásra.

Megemlítünk még négy állományt az *include/* könyvtárból. Az *stdlib.h* olyan típusokat, makrókat és függvényprototípusokat definiál, amelyek a legegyszerűbb C programok kivételével szinte mindig szükségesek. A felhasználói programok fordítása során ez az egyik leggyakrabban használt definíciós fájl, a MINIX 3-rendszer forráskódjában azonban csak a kernelben hivatkozunk rá néhány helyen. Az *stdio.h* mindenkinek ismerős, aki a C nyelv tanulását a híres „Hello, World!" program megírásával kezdte. A rendszerfájlokban alig használjuk, de az *stdlib.h*-hoz hasonlóan szinte minden felhasználói programban megtalálható. Az *a.out.h* definiálja a lemezen tárolt végrehajtható programok fájlformátumát. Egy *exec* struktúra található benne, a processzuskezelő az ebben tárolt információ alapján tölti be a programokat exec hívás hatására. Végül, az *stddef.h* néhány gyakran használt makrót definiál.

Folytassuk az *include/sys/* könyvtárral. Ahogy a 2.32. ábrán látható, a MINIX 3 összes elsődleges definíciós állományában mindjárt az *ansi.h* után szerepel a *sys/types.h* (1400. sor), amelyben adattípusok vannak definiálva. Sok hiba származhat abból, ha félreértjük, hogy egy bizonyos helyen melyik alap adattípus szerepel. Ezeket kiküszöbölhetjük, ha az itt megadott definíciókat használjuk. A 2.33. ábra mutatja néhány típus bitekben mért hosszát 16 és 32 bites processzor esetén. Figyeljük meg, hogy minden típusnév végződése „_t". Ez nem egyszerűen csak konvenció; a POSIX szabvány írja elő. Ez egy **foglalt végződés**, nem használható olyan név esetén, amely *nem* típusnév.

Típus	16 bites MINIX	32 bites MINIX
gid_t	8	8
dev_t	16	16
pid_t	16	32
ino_t	16	32

2.33. ábra. *Néhány típus bitekben mért hossza 16 és 32 bites rendszerben*

A MINIX 3 jelenleg 32 bites mikroprocesszorokon fut, de a 64 bitesek egyre fontosabbak lesznek a jövőben. A hardver által nem támogatott típusok szintetizálhatók, ha szükséges. Az 1471. sorban az *u64_t* típus definíciója struct {u32_t[2]}. Erre a típusra nincs túl gyakran szükség a jelenlegi implementációban, de hasznos lehet – például minden lemez- és partícióadat (eltolási címek és méretek) 64 bites számként van tárolva, emiatt nagyon nagy lemezek is megengedettek.

A MINIX 3 sok olyan típusdefiníciót használ, amelyeket a fordítóprogram végül a kisszámú alaptípus egyikeként értelmez. Ezzel a kódot szerettük volna olvashatóbbá tenni; például egy *dev_t* típusú változóról azonnal látszik, hogy egy I/O-eszközt azonosító fő- és alárendelt eszközszám tárolására szánták. A fordítóprogram számára ezzel egyenértékű lenne, ha ezt a változót *short* típusúnak deklarálnánk. Megjegyezzük még, hogy sok itt definiált típusnak van nagybetűvel kezdődő alakja is, például *dev_t* és *Dev_t*. A fordítóprogram számára a nagybetűs változatok mind egyenértékűek az *int* típussal. Ezek a K&R fordítóprogramokhoz készült olyan függvényprototípusokban fordulnak elő, amelyekben az előforduló típusoknak kompatibilisnek kell lenniük az *int* típussal. A *types.h* megjegyzései további magyarázatokkal szolgálnak.

Említésre méltó még az a feltételes kódrészlet, amely az

```
#if _EM_WSIZE == 2
```

sorral kezdődik (1502–1516. sor). Ahogy korábban jeleztük, a legtöbb feltételes kódrészletet eltávolítottuk a könyv kedvéért. A fenti részlet azért maradt benne, hogy megmutassuk a feltételes definíciók egyik lehetséges felhasználási módját. Az itt használt *_EM_WSIZE* makró egy újabb példája a fordítóprogram által definiált ellenőrző makrónak. A célarchitektúra szóméretét tartalmazza bájtokban mérve. Az #if ... #else ... #endif sorozat használata egy módja annak, hogy bizonyos definíciók mindig helyesek legyenek, ezzel biztosítva, hogy az utána következő programrész helyesen forduljon le 16 és 32 bites rendszeren is.

Az *include/sys/* könyvtár sok más állománya is széles körben használt a MINIX 3-rendszerben. A *sys/sigcontext.h* (1600. sor) olyan adatszerkezeteket definiál, amelyeket a kernel és a processzuskezelő arra használnak, hogy a szignálkezelő rutinok előtt a rendszer állapotát elmentsék, utána pedig visszaállítsák. A *sys/stat.h* (1700. sor) definiálja az 1.12. ábrán már bemutatott struktúrát, amelyben a stat és fstat rendszerhívások adnak vissza információt. Itt található még a *stat, fstat* és más, az állományok jellemzőit manipuláló függvények prototípusa is. Erre az állományra a fájlrendszer és a processzuskezelő sok helyen hivatkozik.

Az ebben a részben utolsóként tárgyalt fájlok nem olyan gyakran használtak, mint az előzők. A *sys/dir.h* (1800. sor) definiálja egy MINIX 3-könyvtárbejegyzés szerkezetét. Csak egyszer hivatkozunk rá közvetlenül, de ezáltal egy olyan definíciós fájlba kerül be, amely a rendszerben széles körben használt. Egyebek mellett azért fontos, mert megadja, hogy egy fájlnév hány karaktert tartalmazhat (60-at). A *sys/wait.h* (1900. sor) a processzuskezelőben megvalósított wait és waitpid rendszerhívások által használt makrókat definiál.

Az *include/sys/* sok más állományát is megemlíthetnénk. A MINIX 3 támogatja a végrehajtható programok nyomkövetését és a hibás programok memóriatérképének elemzését egy nyomkövető programmal. Ehhez a *sys/ptrace.h* definiálja a ptrace rendszerhívás lehetséges műveleteit. A *sys/svrctl.h* az svrctl rendszerhívás által használt adatszerkezeteket és makrókat definiálja. Az svrctl igazából nem is rendszerhívás, de úgy használjuk, mintha az lenne. Az svrctl a szerverprocesszusok koordinálására használatos a rendszer indításakor. A select rendszerhívás segítségével több csatornán várakozhatunk bejövő adatra – például virtuális terminálok várakozhatnak hálózati kapcsolatra. A szükséges definíciókat a *sys/select.h* tartalmazza.

Szándékosan hagytuk a *sys/ioctl.h* és a kapcsolódó fájlok vizsgálatát a végére, mert nem érthetők meg teljesen, ha nem tekintjük át előbb a *minix/ioctl.h* állományt. Az ioctl rendszerhívással a különböző eszközöket vezérelhetjük. Egyre növekvő számú új, a modern számítógépes rendszerekhez illeszthető eszköz jelenik meg, ezek is valamiféle vezérlést igényelnek. A könyvben leírt MINIX 3-rendszer valójában abban különbözik más változatoktól, hogy aránylag kevés I/O-eszközt mutat be. Sok más eszköz, mint például hálózati csatolók, SCSI-vezérlők és hangkártyák beillesztésére van lehetőség.

A könnyebb kezelhetőség érdekében több kisebb állományt használtunk, mindegyikben egy definíciócsoport található. A *sys/ioctl.h* (2000. sor) mindegyiket beilleszti, ennyiben a 2.32. ábra elsődleges definíciós állományához hasonlít. Az egyik ilyen definíciós fájl a *sys/ioc_disk.h* (2100. sor). Ez és a *sys/ioctl.h* által beillesztett többi fájl az *include/sys/* könyvtárban van, mert a „nyilvános programozói felület" részének tekintjük, ami azt jelenti, hogy a programozók felhasználhatják a MINIX 3-környezetbe szánt programjaikban. Mindegyik függ azonban olyan makródefinícióktól, amelyek a *minix/ioctl.h* állományban (2200. sor) vannak elhelyezve, ezért ezt mindegyik be is illeszti. Programíráskor a *minix/ioctl.h*-t nem szabad önmagában használni, ezért van az *include/minix/*-ben, és nem az *include/sys/*-ben.

Az ezekben a fájlokban definiált makrók együtt azt határozzák meg, hogy a lehetséges funkciókhoz tartozó különböző adatokat hogyan csomagoljuk be az ioctl argumentumaként használt 32 bites egész számba. Például a lemezeknek öt műveletük van, ahogy a *sys/ioc_disk.h* 2110-től 2114-ig terjedő soraiban látható. A „d" karakter azt jelzi az ioctl-nek, hogy lemezműveletről van szó, 3-tól 7-ig terjedő egész szám kódolja a műveletet, a harmadik paraméter pedig olvasás vagy írás esetén megadja az adatátadásra használt struktúra méretét. A *minix/ioctl.h* 2225. és 2231. sorai között azt láthatjuk, hogy a karakterkód 8 bitje 8 hellyel balra tolódik, a struktúra méretének legkisebb helyi értékű 13 bitje 16 bittel balra tolódik, majd

ezek logikai VAGY műveletben egyesítődnek a műveleti kóddal. A 32 bites érték legnagyobb helyi értékű 3 bitjében kerül kódolásra a visszatérési érték típusa.

Bár sok munkának tűnik, mindez fordítási időben történik, futás közben pedig nagyon hatékony kapcsolódást ad a rendszerhívásnak, mivel a felhasznált argumentum a CPU legtermészetesebb adattípusa. Eszünkbe juttathat viszont egy híres megjegyzést, amelyet Ken Thompson írt a Unix egyik korai verziójának forráskódjába:

```
/* You are not expected to understand this */
```

(Többféleképpen is értelmezhető: „Ezt nem kell megértened", vagy „Ezt úgysem fogod megérteni".)

A *minix/ioctl.h* tartalmazza az ioctl rendszerhívás prototípusát is (2241. sor). Ezt általában a programozók nem hívják közvetlenül, mert az *include/termios.h* állományban definiált szabványos POSIX-függvények sok esetben kiváltják a régi *ioctl* könyvtári függvényt terminálok, konzolok és hasonló eszközök kezelése esetén. Ennek ellenére szükség van rá. Tulajdonképpen a terminálkezelő POSIX-függvények belül az ioctl rendszerhívást használják.

2.6.4. A MINIX 3 definíciós állományok

Az *include/minix/* és *include/ibm/* könyvtár MINIX 3-specifikus definíciós fájlokat tartalmaz. Az *include/minix/* állományai között vannak olyanok, amelyeknek architektúrától függő változatai vannak, de többségük minden megvalósításhoz szükséges. Egy ezek közül az *ioctl.h,* amit éppen az előbb láttunk. Az *include/ibm/* könyvtárban az IBM PC-n történő megvalósításhoz tartozó struktúrák és makrók találhatók.

Kezdjük a *minix/* könyvtárral. Az előző részben láttuk, hogy a *config.h* (2300. sor) a MINIX 3-rendszer minden részének elsődleges definíciós állományába bekerül, így ez a fordító által legelőször feldolgozott fájl. Ha módosítanunk kell hardvereltérések miatt, vagy mert az operációs rendszert másképpen akarjuk használni, akkor sok esetben mindössze ezt az állományt kell kijavítani és a rendszert újrafordítani. Azt javasoljuk, hogy ha módosítunk ebben a fájlban, akkor a 2303. sorban a megjegyzésben utaljunk a módosítás céljára.

A felhasználó által beállítható paraméterek mind a fájl első részében találhatók, de nem mindegyiket ajánlatos itt módosítani. A 2326. sorban egy másik definíciós fájl, a *minix/sys_config.h* kerül beillesztésre, és némelyik paraméter definíciója innen öröklődik. A programozók ezt így látták jónak, mert a rendszer néhány állományában szükség van a *sys_config.h* definícióira, de a többire a *config.h*-ból nincs. Sok olyan név van a *config.h*-ban, amelyek nem aláhúzásjellel kezdődnek (például *CHIP* vagy *INTEL*), és könnyen előfordulhatna, hogy ütköznek olyan általánosan használt nevekkel, amelyeket nagy valószínűséggel megtalálunk a MINIX 3 alá más operációs rendszerekből áthozott programokban. A *sys_config.h* nevei mind aláhúzásjellel kezdődnek, ezért a névütközés esélye kisebb.

A *MACHINE* értéke *_MACHINE_IBM_PC* lesz a *sys_config.h*-ban; a 2330. és 2334. sor között a lehetséges nevek rövid alternatíváit találjuk. A MINIX korábbi verziói Sun-, Atari- és Macintosh-platformokon is futottak, és a teljes forráskód tartalmaz alternatívákat ezekhez a hardverekhez is. A forráskód nagy része gépfüggetlen, de egy operációs rendszer mindig tartalmaz gépfüggő kódot is. Azt is meg kell jegyeznünk, hogy mivel a MINIX 3 meglehetősen új, a könyv megírásakor még további munkára van szükség ahhoz, hogy a MINIX 3-at nem Intel-platformokra is át lehessen vinni.

A *config.h* más definíciói segítségével a telepítés során egyéb igényeinket állíthatjuk be. Például a fájlrendszer által gyorsítótárnak használt pufferek száma általában a lehető legnagyobb kell hogy legyen, de sok pufferhez nagyon sok memória kell. A 2345. sorban beállított 128 blokk minimáhsnak tekinthető, és csak 16 MB-nál kevesebb RAM-mal rendelkező gépen kielégítő. Nagy memóriával rendelkező gépeken sokkal nagyobb számot érdemes ideírni. Ha modemet akarunk használni vagy hálózaton keresztül is be akarunk jelentkezni, akkor az *NR_RS_LINES* és az *NR_PTYS* definíciós sorokban kell az értékeket megnövelni és a rendszert újrafordítani. A *config.h* utolsó része olyan definíciókat tartalmaz, amelyek szükségesek, de nem változtathatók meg. Sok definíció csak alternatív neveket ad a *sys_config.h*-ban definiált konstansoknak.

A *sys_config.h* (2500. sor) olyan definíciókat tartalmaz, amelyek a rendszerprogramozók érdeklődésére tarthatnak számot, például ha valaki egy új eszközmeghajtót ír. Egyébként valószínűleg nem kell megváltoztatni semmit, talán kivételt képez az *NR_PROCS* konstans (2522. sor). Ez a processzustábla méretét adja meg. Ha a MINIX 3-rendszert hálózati szervernek akarjuk használni sok távoli felhasználóval vagy sok egyidejűleg futó szerverprocesszussal, akkor esetleg meg kell növelni ezt az értéket.

A következő fájl a *const.h* (2600. sor), amely a definíciós fájlok egy másik gyakori felhasználási módját illusztrálja. Számos olyan gyakran használt konstans definícióját találjuk itt, amelyet új kernel fordításakor valószínűleg nem változtatunk meg. Ezek egy helyre gyűjtése megelőzi azokat a nehezen felderíthető hibákat, amelyek több helyen megadott, egymásnak ellentmondó definíciókból származnának. Más *const.h* nevű állományok is vannak a MINIX 3 forráskódkönyvtáraiban, de ezek használata korlátozottabb. A csak a kernelben használt definíciókat az *src/kernel/const.h* tartalmazza, a csak a fájlrendszerben használt definíciókat pedig az *src/servers/fs/const.h.* A processzuskezelő lokális definíciói peidg az *src/servers/pm/const.h* állományban vannak. Az *include/minix/const.h* csak azoknak a definícióknak ad helyet, amelyeket a MINIX 3-rendszer több helyén is felhasználunk.

A *const.h* néhány definíciója figyelemre méltó. Az *EXTERN* egy makró, amelynek kifejtése *extern* (2608. sor). A definíciós fájlban deklarált, és több állományba is beillesztett globális változókat *EXTERN* előzi meg, mint például

```
EXTERN int who;
```

Ha a deklaráció csak

```
int who;
```

lenne, és több állományba is beillesztésre kerülne, akkor némelyik szerkesztőprogram többszörösen definiált változóra panaszkodna. Ezenkívül a C referencia kézikönyv (Kernighan és Ritchie, 1988) is egyértelműen megtiltja ezt a konstrukciót.

A hiba elkerülésének módja, hogy egyetlen hely kivételével

```
extern int who;
```

szerepeljen. Az *EXTERN* használata biztosítja ezt olyan módon, hogy a *const.h* minden beillesztése során *extern* helyettesítődik be, kivéve, ha valahol az üres karakterláncot adva értékül újradefiniáljuk. Ez úgy történik, hogy a globális definíciók a MINIX 3 minden részében egy speciális *glo.h* állományban vannak, mint például *src/kernel/glo.h,* amely közvetett módon minden fordításkor beillesztődik. Minden *glo.h* tartalmazza az

```
#ifdef _TABLE
#undef EXTERN
#define EXTERN
#endif
```

sorokat, és a MINIX 3 minden részében a *table.c* állományokban van egy

```
#define _TABLE
```

sor az #include szekció előtt. Így amikor a definíciós fájlok a *table.c* fordítása során beillesztésre és kifejtésre kerülnek, nem kerül *extern* az *EXTERN* helyére (mivel ez utóbbi értéke itt már az üres karakterlánc). Emiatt a globális változók részére memóriát csak egy helyen, a *table.o* tárgykódú állományban foglalunk le.

A C programozásban járatlan olvasót szeretnénk megnyugtatni, hogy nem baj, ha nem ért mindent ezekből a dolgokból, a részletek nem igazán fontosak. Ez Ken Thompson korábbi híres megjegyzésének egy udvarias formája. Néhány szerkesztőprogramnak problémát okozhat a definíciós fájlok többszöri beillesztése, mert ez bizonyos változók többszöri deklarációját eredményezheti. Ennek az egész *EXTERN* játéknak egyszerűen az a célja, hogy a MINIX 3 hordozhatóbb legyen, és olyan gépeken is használhassuk, ahol a szerkesztőprogram nem fogad el többszörösen definiált változókat.

A *PRIVATE* a *static* szinonimájaként van definiálva. A *PRIVATE* szerepel mindazoknak az eljárásoknak és adatoknak a deklarációjában, amelyekre nem hivatkozunk más állományokban, így ezek nevei nem is láthatók azon az állományon kívül, ahol deklaráltuk őket. Általános szabály, hogy minden változót és eljárást a lehető legkisebb hatáskörrel kell deklarálni. A *PUBLIC* definíciója az üres karakterlánc, így a

```
PUBLIC void lock_dequeue(rp)
```

deklarációt a C preprocesszor a


```
void lock_dequeue(rp)
```

alakra fejti ki, ami a C hatásköri szabályok szerint azt jelenti, hogy a lock_dequeue név exportálódik, és más, a programba beleszerkesztett állományokból is hívható, ebben az esetben bárhonnan a kernelből. Egy másik függvény ugyanabból a fájlból a

```
PRIVATE void dequeue(rp)
```

ami előfeldogozás után

```
static void dequeue(rp)
```

lesz. Ez a függvény csak ugyanabban a fájlban található programkódból hívható. A *PRIVATE* és a *PUBLIC* nem feltétlenül szükséges, csak egy próbálkozás arra, hogy a C hatásköri szabályok által okozott kárt enyhítse (alapértelmezés szerint a nevek exportálódnak; ennek éppen fordítva kellene lennie).

A *const.h* maradék részében a rendszerben sokat használt numerikus konstansok definíciói vannak. A *const.h* egy szekciója a gép- vagy konfigurációfüggő definícióknak van szentelve. Például a forráskód egészében a memóriafoglalás alapegysége a **memóriaszelet (click).** A memóriaszelet mérete függhet a processzortól, az Intel processzorokhoz 1024 az értéke. Az Intel, Motorola 68000 és Sun SPARC architektúrához tartozó értékek a 2673. és 2681. sor között találhatók. Ez a fájl tartalmazza a *MAX* és a *MIN* makrókat is, így a

```
z = MAX(x, y);
```

utasítással *z*-hez hozzárendelhetjük *x* és *y* közül a nagyobbikat.

Az elsődleges definíciós fájlok segítségével a *type.h* (2800. sor) is beillesztésre kerül minden fordítás során. Kulcsfontosságú típusokat és a hozzájuk tartozó numerikus értékeket tartalmaz.

Az első két struktúra két különböző memóriatérképet definiál, egyiket a lokális memóriarégiók számára (a processzus adatterületén belül), a másikat a távoli régiók, mint például a RAM-lemez számára (2828–2840. sor). Itt megemlíthetjük a memóriahivatkozások mögötti elveket. Ahogy az előbb említettük, a memóriaszelet a memória alapmértékegysége, MINIX 3-ban Intel processzorokra egy memóriaszelet 1024 bájt. A memória mérésének egyik egysége a **phys_clicks**, ezt a kernel használhatja, így bármelyik memóriaelemet meg tudja címezni a rendszerben. A másik lehetőség a **vir_clicks**, amelyet a processzusok használnak. Egy *vir_clicks* hivatkozás mindig egy processzushoz rendelt memóriaszegmens kezdőcíméhez viszonyított, és a kernelnek gyakran kell konvertálnia a virtuális (vagyis processzusalapú) és fizikai (RAM-alapú) címek között. A kényelmetlenséget ellensúlyozza, hogy a processzusok összes memóriahivatkozása *vir_clicks* egységben történhet.

Azt feltételezhetnénk, hogy ugyanaz az egység megfelel mindkét típusú hivatkozáshoz, de előnyösebb, ha a *vir_clicks* a processzusokhoz rendelt memória egy-

ségét jelenti, mert ebben az esetben ellenőrizhető, hogy a processzushoz rendelt memórián kívülre nem történik hivatkozás. Ez a modern Intel processzorok (például a Pentium család) **védett üzemmód**jának egy fontos szolgáltatása. Ennek hiánya a régebbi 8086-os és 8088-as processzorokon okozott némi fejfájást a korábbi MINIX-verziók tervezésekor.

Egy másik fontos itt definiált struktúra a *sigmsg* (2866–2872. sor). Amikor egy szignál érkezik, akkor a kernelnek úgy kell intézni a dolgokat, hogy a szignált kapó processzus a legközelebbi futásakor a szignálkezelőjét hajtsa végre, és ne ott folytatódjon, ahol előzőleg a futása megszakadt. A processzuskezelő elvégzi a szignálok kezelésével kapcsolatos legtöbb feladatot; egy ilyen struktúrát ad át a kernelnek, amikor szignált kap.

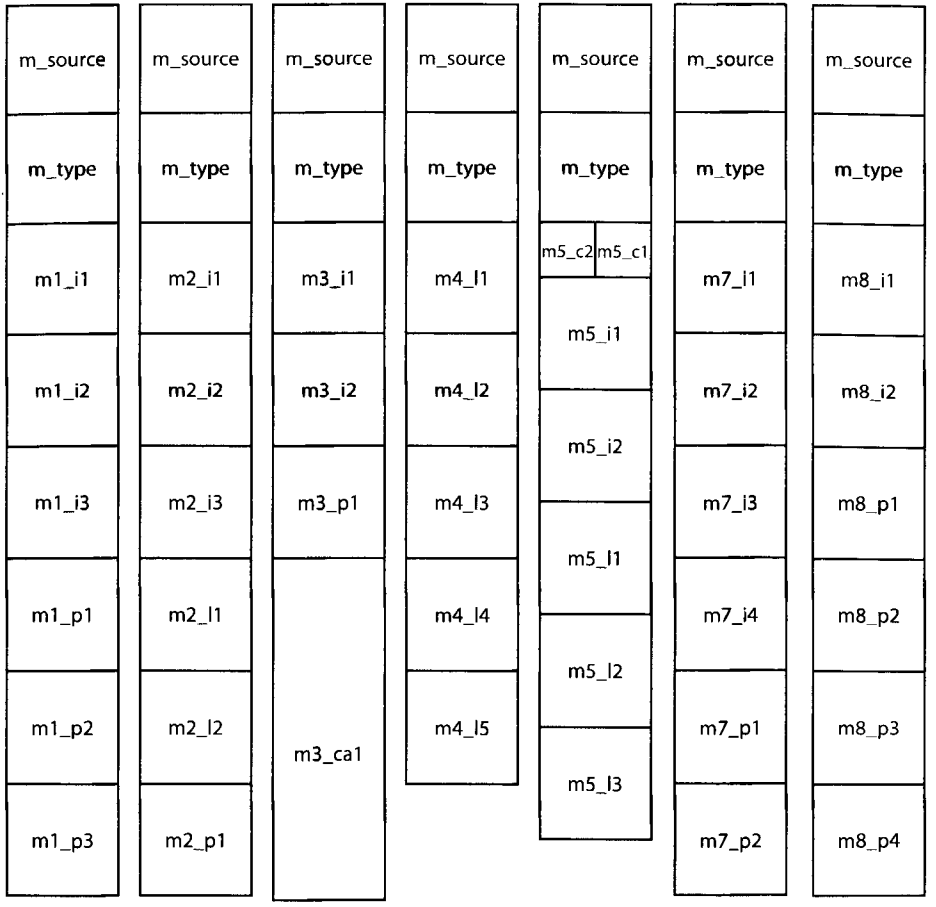
A *kinfo* struktúra (2875–2893. sor) arra szolgál, hogy információt adjon a kernelről a rendszer többi részének. A processzuskezelő ezt az információt használja, amikor a processzustábla rá eső részét beállítja.

Az *ipc.h* (3000. sor) adatszerkezeteket és függvényprototípusokat definiál a **processzusok közötti kommunikáció**hoz. A legfontosabb a *message* definíciója a 3020. és 3032. sor között. Definiálhattuk volna egy bizonyos hosszúságú bájtvektorként is, de helyesebb programozási gyakorlat az, ha egy olyan struktúrát használunk, amely tartalmazza a lehetséges üzenettípusok unióját. Hét üzenetformátumot definiálunk, *mess_1*-től *mess_8*-ig (a *mess_6* már elavult). Egy üzenetstruktúra tartalmazza a küldőt azonosító *m_source,* az üzenet típusát tartalmazó *m_type* mezőt (például *SYS_EXEC* a rendszertaszkhoz), valamint az adatmezőket.

A hét üzenettípus a 2.34. ábrán látható. Négy üzenettípus, az első kettő, illetve az utolsó kettő azonosnak látszik. Az adatelemek méretét tekintve ez így is van, de az elemek típusa különböző. Egy 32 bites Intel processzor esetén az *int, long* és pointer típusok is 32 bitesek, de nem feltétlenül ez a helyzet más hardver esetén. Hét típust használva könnyebb egy másik architektúrára áttérni.

Ha mondjuk három egészet és három mutatót (vagy három egészet és két mutatót) tartalmazó üzenetet kell küldeni, akkor a 2.34. ábrán látható első formátumot használjuk. Ugyanez érvényes a többi formátumra is. Hogyan rendelünk értéket az első egészhez az első formátumban? Tegyük fel, hogy az üzenetet *x*-szel jelöljük. Ekkor *x.m_u* az üzenetstruktúra unió részét jelöli. Az unió hét alternatívája közül az elsőt az *x.m_u.m_m1* jelöli ki. Végül az első egészet az *x.m_u.m_m1.m1i1* kifejezés azonosítja. Ez egész nagy falat, ezért az üzenetek után valamivel rövidebb mezőneveket definiálunk makróként. Így *x.m1_i1* használható *x.m_u.m_m1.m1i1* helyett. A rövid nevek mind „m" betűvel kezdődnek, majd a formátum száma és egy aláhúzás karakter áll. Ezután egy vagy két karakter jelzi, hogy a mező egész (integer), mutató (pointer), hosszú egész (long), karakter (char), karaktertömb (char array) vagy függvény (function). Végül egy sorszám következik, amely az egy üzeneten belül előforduló több azonos típus megkülönböztetésére szolgál.

Az üzenetformátumok tárgyalása közben remek alkalom nyílik megjegyezni, hogy az operációs rendszer és a fordítóprogram gyakran „tud" olyan dolgokról, mint a struktúrák szerkezete, és ez megkönnyítheti a rendszer készítőjének dolgát. A MINIX 3-ban az üzenetek *int* mezőit néha *unsigned* típusú értékek tárolására használjuk. Ez túlcsordulást okozhatna, de a programkód annak ismeretében készült, hogy a MINIX 3-fordító az *unsigned* és az *int* típusokat a bitminta megváltoztatása és túlcsordulás ellenőrzése nélkül másolja át egymásba. A pontosabb megoldás az lenne, ha minden *int* mezőt egy *int* és egy *unsigned* uniójára cserélnénk. Az előbbi érvényes az üzenetek *long* mezőire is; néha *unsigned long* adatok továbbítására használjuk őket. Hogy ez csalás? Mondhatjuk úgy is, de a MINIX 3 új architektúrára történő átvitele során az üzenetek pontos szerkezetének meghatározása nyilván nagy figyelmet igénylő munka; most pedig felhívtuk a figyelmet arra, hogy a fordítóprogram viselkedése is egy újabb figyelembe veendő szempont.



2.34. ábra. *A MINIX 3-ban használt hét üzenettípus. Az üzenetek részeinek mérete architektúránként változó; ez a diagram a 32 bites pointert használó gépen érvényes méreteket mutatja, ilyenek például a Pentium család tagjai*

Az *ipc.h* definiál még prototípusokat a korábban leírt üzenetkezelő alapműveletekhez (3095–3101. sor). A fontos send, receive, sendrec és notify mellett több másik is látható itt. Ezek nem sok helyen fordulnak elő, tulajdonképpen azt is mondhatnánk, hogy a MINIX 3 korábbi fejlesztési fázisaiból itt maradt relikviák.

A régi számítógépprogramokban remek régészeti ásatásokat lehet folytatni. Ezek eltűnhetnek a jövőbeni verziókban. Ennek ellenére, ha nem magyarázzuk el őket, akkor néhány olvasó biztosan aggódni fog miattuk. A nem blokkoló nb_send és nb_receive hívásait jórészt leváltotta a notify, amelyet később vezettünk be, mert jobb megoldásnak tekintettük a blokkolás nélküli küldés, illetve blokkolás nélküli üzenet-ellenőrzés problémájára. Az echo prototípusának nincs küldő és fogadó mezője. Nincs semmi haszna végleges programban, de fejlesztés közben hasznos volt az üzenetküldések és -fogadások időszükségletének meghatározására.

Van még egy, az elsődleges definíciós fájlok beillesztése útján általánosan használt fájl az *include/minix/* könyvtárban. Ez a *syslib.h* (3200. sor), amely a MINIX 3 szinte összes felhasználói szintű komponensébe azok elsődleges definíciós állományain keresztül kerül beillesztésre. Ez a fájl nem kerül be az *src/kernel/kernel.h*-ba, a kernel elsődleges definíciós állományába, mert a kernelnek nincs szüksége könyvtári függvényekre önmaga elérésére. A *syslib.h* olyan C könyvtári függvények prototípusait tartalmazza, amelyeket az operációs rendszer azért hív, hogy más operációsrendszer-szolgáltatásokat vegyen igénybe.

A C könyvtárakat nem tárgyaljuk részletesen a könyvben, de sok ezek közül szabványos, és minden C fordítóhoz rendelkezésre áll. A *syslib.h* függvényei azonban természetesen a MINIX 3-hoz kötődnek, és más fordítót használó gépre történő átvitel esetén ezeket újra kell írni. Szerencsére ez nem nehéz, mert a függvények mindössze annyit tesznek, hogy elhelyezik a paramétereket egy üzenetstruktúrában, majd elküldik az üzenetet, és a válasz alapján összeállítják az eredményt. Ezeknek a könyvtári függvényeknek a többsége tíz-egynéhány sorból álló C programmal van definiálva.

Említést érdemel még ebben a fájlban négy makró, amelyekkel bájt vagy szó hosszúságú adatokat tudunk I/O-kapukra küldeni, vagy onnan fogadni. Itt található a *sys_sdevio* függvény prototípusa is, erre mind a négy makró hivatkozik (3241–3250. sor). A MINIX 3-projekt lényeges része volt az eszközmeghajtók átvitele felhasználói szintre, ehhez pedig a kernelbe be kellett építeni olyan funkciókat, amelyek segítségével az eszközmeghajtók az I/O-kapuk írását és olvasását kezdeményezhetik.

A *sysutil.h*-ba (3400. sor) került néhány olyan függvény, amelyek a *syslib.h*-ban is lehetnének, ezért tárgykódjuk külön modult alkot. Két függvény, amelyek prototípusa itt van elhelyezve, további magyarázatot igényel. Az egyik a *printf* (3442. sor). Akinek van C programozási tapasztalata, az azonnal felismeri a szabványos *printf* könyvtári függvényt, amely szinte minden programban szerepel.

Ez viszont nem az a *printf* függvény. A szabványos könyvtári *printf* nem használható a rendszerkomponenseken belül. Többek között a szabványos *printf* a szabványos kimenetre akar írni, és képesnek kell lennie lebegőpontos számok formázására is. A szabványos kimenet használata azt jelentené, hogy át kell menni a fájlrendszeren, de amikor a rendszerkomponens valamilyen probléma miatt hibaüzenetet akar kiírni, akkor szerencsésebb, ha ezt meg tudja tenni más rendszerkomponensek segítsége nélkül. A szabványos verzió teljes formázási képességeinek beépítése is csak szükségtelenül növelné a kód méretét. Ezért a *printf* egyszerűsített verziója kerül befordításra a rendszerkönyvtárba, amely csak azt tudja, amire a komponenseknek szükségük van. A fordítóprogram ezt a platformtól függő helyen találja meg; 32 bites Intel-rendszereken ez az */usr/lib/i386/libsysutil.a*. Amikor a fájlrendszer, a processzuskezelő vagy az operációs rendszer más része szerkesztődik össze könyvtári függvényekkel, akkor a szerkesztő az egyszerűsített változatot találja meg, még mielőtt a szabványos könyvtárban keresne.

A következő sorban a *kputc* prototípusát találjuk. Ezt a *printf* rendszerszintű verziója hívja, hogy karaktereket jelenítsen meg a konzolon. Itt azonban trükkös dolgok történnek. A *kputc* több helyen is definiálva van. Van egy példány a rendszerkönyvtárban, alapesetben ez használatos, de a rendszer különböző részei saját verziót definiálnak. Látni fogunk egyet, amikor a következő fejezetben a konzol programozói felületet fogjuk tanulmányozni. A naplózó eszközmeghajtó (amit nem részletezünk) szintén definiálja a saját verzióját. Még a kernelben is van egy *kputc* verzió, de az egy speciális eset. A kernel nem használja a *printf*-et. Egy speciális kiíró függvény, a *kprintf* van definiálva erre a célra, a kernel ezt hívja, ha ki kell írnia valamit.

Ha egy processzus egy MINIX 3-rendszerhívást akar végrehajtani, akkor üzenetet küld a processzuskezelőnek vagy a fájlrendszernek. Minden üzenet tartalmazza az igényelt rendszerhívás számát. Ezek a számok a *callnr.h* állományban (3500. sor) vannak definiálva. Némelyik szám nem használt, ezek vagy később kerülnek csak implementálásra, vagy korábbi verziók olyan hívásait jelölik, amelyeket már könyvtári függvények kezelnek. A fájl vége felé olyan hívási számokat találunk, amelyek nem felelnek meg az 1.9. ábrán látott hívások egyikének sem. A (korábban említett) svrctl és a ksig, unpause, revive, valamint a task_reply csak az operációs rendszeren belül használatos. A rendszerhívás mechanizmus nagyon kényelmes ezek megvalósítására. Valójában mivel külső programok nem használhatják ezeket a „rendszerhívásokat", későbbi verziókban anélkül módosíthatók, hogy ez a felhasználói programok viselkedését befolyásolná.

A következő állomány a *com.h* (3600. sor). A név egyik értelmezése az lehetne, hogy „common", vagyis „általános", egy másik, hogy „kommunikáció". Ez a fájl szerverek és eszközmeghajtók közötti kommunikációhoz általános definíciókat tartalmaz. A 3623. és 3626. sor között taszksorszámok definíciói vannak. Negatívak, hogy meg lehessen különböztetni őket a processzusoktól. A 3633. és 3640. sor között processzusszámok definíciói találhatók, olyan processzusokéi, amelyek a betöltési memóriaképben találhatók. Figyeljünk arra, hogy ezek a számok a processzustábla indexei, nem szabad összekeverni őket a processzusazonosítókkal (PID).

A *com.h* következő része azt definiálja, hogy hogyan kell üzeneteket konstruálni notify művelethez. A processzusszámból egy olyan érték kerül előállításra, amelyet az üzenet *m_type* mezőjébe helyezünk. Az ebben a fájlban definiált értesítések és más üzenetek üzenettípusai úgy képződnek, hogy egy típusosztályt jelölő bázisértéket kombinálunk egy konkrét típust jelölő kis számmal. A fájl maradék része olyan makrók gyűjteménye, amelyek értelmes azonosítókat alakítanak át az üzenettípusokat és mezőneveket azonosító mágikus számokká.

Az */include/minix/* tartalmaz még néhány állományt. A *devio.h* (4100. sor) olyan típusokat és konstansokat definiál, amelyek az I/O-kapuk elérését teszik lehetővé felhasználói szintről, illetve néhány olyan makrót, amelyekkel egyszerűsödik a kapuk

és egyéb értékek megadása. A *dmap.h* (4200. sor) egy struktúrát és ennek tömbjét definiálja, mindkettő neve *dmap*. Ez a táblázat segít összerendelni a fő eszközszámokat a hozzájuk tartozó függvényekkel. A *memory* eszközmeghajtó fő- és alárendelt eszközszáma, illetve más fontos eszközök fő eszközszáma van még itt definiálva.

Az */include/minix/* tartalmaz olyan speciális állományokat is, amelyekre a rendszer lefordításához van szükség. Az egyik az *u64.h,* amelyben 64 bites aritmetikai műveletekhez találunk támogatást; ezekre a nagy kapacitású lemezeken végzett címszámításokhoz van szükség. Ezekről még nem is álmodtak akkor, amikor a UNIX, a C nyelv, a Pentium-processzorok vagy a MINIX ötlete megszületett. Elképzelhető, hogy a MINIX 3 jövőbeli verziói olyan nyelven lesznek írva, amelyekben van beépített támogatás 64 bites egész számok kezelésére 64 bites regiszterekkel ellátott CPU-kon. Addig az *u64.h* definícióival át lehet hidalni a problémát.

Három állományt említünk még. A *keymap.h* a különböző nyelvek által használt karakterkészletekhez speciális billentyűzetelrendezéseket megvalósító struktúrákat definiál. A fájl az ilyen táblázatokat előállító és betöltőprogramok számára is szükséges. A *bitmap.h* néhány olyan makrót definiál, amelyekkel bitek beállítása, törlése és ellenőrzése egyszerűbben megoldható. Végül, a *partition.h* definiálja a MINIX 3 számára a lemezpartíciók definiálásához szükséges információkat, akár abszolút lemezcím és méret alakban, akár cilinder, fej és szektorcím alakban. Az *u64_t* típust használjuk a cím és a méret megadásához, hogy nagy lemezek is kezelhetők legyenek. A partíciók szerkezetét nem ez a fájl írja le, hanem egy másik a következő könyvtárban.

Az utolsóként megvizsgált speciális *include/ibm/* könyvtár több olyan definíciós fájlnak ad helyet, amelyek az IBM PC-számítógépekhez tartozó definíciókat tartalmaznak. Mivel a C nyelv csak memóriacímeket ismer, és nincs felkészítve I/O-kapuk címének elérésére, ez a függvénykönyvtár olyan assembly nyelvű rutinokat tartalmaz, amelyek I/O-kapukat tudnak írni és olvasni. A különféle rutinokat az *ibm/portio.h* (4300. sor) definiálja. Bájtokat, egész számokat és hosszú egészeket lehet egyesével vagy adatsorozatként olvasni és írni is, *inb*-től (egy bájt beolvasása) *outsl*-ig (hosszú egészek sorozatának kiírása) terjedő rutinokkal. A kernel alacsony szintű rutinjainak a CPU-megszakítások letiltására és engedélyezésére is szükségük lehet, ezeket a C szintén nem tudja kezelni. A függvénykönyvtárban ehhez megvannak az assembly nyelvű rutinok, az *intr_disable* és az *intr_enable* a 4325. és a 4326. sorban van definiálva.

A következő fájl az *interrupt.h* (4400. sor), amely a PC-kompatibilis rendszerek megszakításvezérlője, és BIOS-a által használt kapu- és memóriacímeket definiál. Végül a *ports.h* (4500. sor) további kapukat definiál. Olyan címeket határoz meg, amelyek a billentyűzetinterfész és az időzítőlapka kezeléséhez szükségesek.

Több IBM-specifikus információt tartalmazó fájl van még az *include/ibm/* alatt. A *bios.h,* a *memory.h* és a *partition.h* bőségesen tartalmaz megjegyzéseket, és érdemes elolvasni annak, aki többet akar tudni a memóriakezelésről és a lemezek partíciós tábláiról. A *cmos.h,* a *cpu.h* és az *int86.h* további információkat tartalmaz a kapukkal, CPU-jelzőbitekkel, valamint a BIOS és a 16 bites módú DOS-szolgáltatások hívásával kapcsolatban. Végül a *diskparm.h* egy, a hajlékonylemezek formázásához szükséges adatszerkezetet tartalmaz.

2.6.5. Processzusok adatszerkezetei és definíciós állományai

Most pedig merüljünk bele az *src/kernel/* programkódjának tárgyalásába. Az előző két szakaszban egy tipikus definíciós fájlból vett néhány soros kivonat vezette a gondolatmenetünket; most nézzük a kernel valódi elsődleges definíciós állományát, ez a *kernel.h* (4600. sor). Három makró definíciójával kezdődik. Az első a *_POSIX_SOURCE*, amely a POSIX szabvány által definiált **ellenőrző makró (feature test macro).** Az ilyen makrók nevének mindig az aláhúzás karakterrel (_) kell kezdődnie. A makró definiálása biztosítja azt, hogy a szabvány által megkövetelt, valamint a nem kötelező, de kifejezetten megengedett makrók láthatók legyenek, míg a nem hivatalos kiterjesztésként definiált szimbólumok ne legyenek elérhetők. A következő két definíciót már említettük, a *_MINIX* makró felülbírálja a *_POSIX_SOURCE* hatását a MINIX 3 által definiált kiterjesztések használhatósága érdekében. A *_SYSTEM* makró segítségével pedig különbséget tehetünk a felhasználói kód és rendszerkód fordítása között, például a hibakódok előjelének megváltoztatása érdekében. A *kernel.h* más definíciós fájlokat is beilleszt az *include/* könyvtárból, illetve ennek *include/sys/, include/minix/* és *include/ibm/* alkönyvtáraiból, beleértve a 2.32. ábrán látható állományok mindegyikét. Ezeket az előző két szakaszban tárgyaltuk. Végül a lokális *src/kernel/* könyvtárból további hat definíciós fájl kerül beillesztésre, ezeknek a neve idézőjelek között található.

A *kernel.h* teszi lehetővé, hogy az

```
#include "kernel.h"
```

sor megadásával a nagyszámú fontos definíció az összes kernel-forrásállomány rendelkezésére áll. Mivel a beillesztések sorrendje néha fontos, a *kernel.h* ezt a sorrendet is egyszer s mindenkorra meghatározza. Ez magasabb szintre emeli a definíciós fájlok mögött rejlő koncepciót, amit úgy lehetne megfogalmazni, hogy „csináld meg egyszer jól, aztán felejtsd el a részleteket". Hasonló elsődleges definíciós fájlok találhatók a fájlrendszer és a processzuskezelő forrásállományai között is.

Most nézzük meg a *kernel.h* által használt lokális definíciós állományokat. Először egy újabb *config.h* nevű fájlt találunk, amelynek a rendszerszintű *include/minix/config.h*-hoz hasonlóan az összes többi #include előtt kell állnia. Ahogy az *include/minix/* közös definíciós könyvtárban, úgy az *src/kernel/* forráskönyvtárban is van *const.h* és *type.h* nevű fájl. Az *include/minix/* olyan állományokat tartalmaz, amelyek a rendszer sok eleme számára szükségesek, beleértve a rendszer felügyelete alatt futó programokat is. Az *src/kernel/* könyvtár állományai olyan definíciókat tartalmaznak, amelyek csak a kernel fordításához szükségesek. A fájlrendszer, a processzuskezelő és más rendszerszintű forráskönyvtárban is van *const.h* és *type.h* nevű fájl; ezek a rendszer megfelelő részéhez szükséges konstansokat és típusokat definiálnak. Az elsődleges definíciós fájl által beillesztett további két fájl a *proto.h* és a *glo.h;* ezeknek nincs megfelelőjük az *include/* könyvtárakban, de mint látni fogjuk, a fájlrendszer és a processzuskezelő esetében van. A *kernel.h*-ba utolsóként beillesztett definíciós állomány az *ipc.h*.

Mivel most először találkozunk ezzel a jelenséggel, figyeljük meg a *kernel/config.h* elején álló #ifndef ... #define sorozatot, amely kiküszöböli az esetleges többszöri beillesztésekből adódó problémákat. Az alapötletet láttuk már korábban is. De láthatjuk, hogy az itt definiált makró neve *CONFIG_H*, kezdő aláhúzásjel nélkül. Így ez különbözik az *include/minix/config.h*-ban definiált *_CONFIG_H* makrótól.

A kernel *config.h* verziója olyan definíciókat gyűjt egy helyre, amelyeket minden valószínűség szerint nem kell módosítani, ha a célunk egy operációs rendszer működésének megértése, vagy annak használata szokványos számítógépes környezetben. De tételezzük fel, hogy egy nagyon kisméretű MINIX 3-verziót akarunk készíteni egy tudományos műszer vagy házi készítésű mobiltelefon számára. A 4717. és a 4743. sor között egyenként letilthatjuk a kernelhívásokat. A szükségtelen funkciók kihagyása a memóriaigényt is csökkenti, mert az egyes kernelhívásokhoz szükséges programkód a 4717. és a 4743. sor közötti definíciókat felhasználó feltételes fordítási direktívák közé van helyezve. Ha valamelyik funkciót letiltjuk, akkor a végrehajtásához szükséges kód kimarad a tárgykódból. Például egy mobiltelefonban esetleg nincs szükség arra, hogy új processzusokat hozzunk létre, ezért az ezt megvalósító kód kimaradhat a végrehajtható programból, kisebb memóriafelhasználást eredményezve. A fájlban előforduló többi konstans közül a legtöbb alapparamétereket határoz meg. Például megszakításkezelés közben a rendszer egy *K_STACK_BYTES* méretű speciális vermet használ. Ez az érték a 4772. sorban kerül beállításra. A veremnek az *mpx386.s* nevű assembly nyelvű fájlban foglaljuk le a helyet.

A *const.h*-ban (4800. sor) található egy makró a 4814. sorban, amely a kernel memóriaterületéhez viszonyított virtuális címeket alakít fizikai címekké. A kernel kódjában máshol definiálva van egy *umap_local* nevű C függvény, amellyel a kernel végre tudja hajtani ugyanezt a konverziót más rendszerkomponensek számára, de a kernelen belül a makró hatékonyabb. Több más hasznos makró is definiálva van itt, többek között bittérképek manipulálására szolgálók. Az Intel hardverbe épített fontos biztonsági funkció aktivizálása is két makróval történik. A **processzor állapotszó (processor status word, PSW)** egy CPU-regiszter, ezen belül az **I/O védelmi szint (I/O Protection Level, IOPL)** bitek határozzák meg, hogy a megszakításrendszerhez és az I/O-kapukhoz engedélyezett-e a hozzáférés. A 4850. és a 4851. sorban különböző PSW-értékek vannak definiálva, amelyek normál és privilegizált processzusok számára meghatározzák a hozzáférést. Ezek az értékek új processzus indításának részeként a verembe kerülnek.

A *type.h* (4900. sor) olyan prototípusokat és struktúrákat definiál, amelyek minden MINIX 3-megvalósításban szükségesek. Például két struktúra is definiálva van, a *kmessages* a kernel diagnosztikai üzenetei számára, illetve a *randomness,* amelyet a véletlenszám-generátor használ.

A *type.h* tartalmaz számos gépfüggő típusdefiníciót is. A kód rövidebbé és olvashatóbbá tétele érdekében eltávolítottunk más CPU-kat érintő, feltételesen fordított részeket, de tudnunk kell, hogy az olyan definíciók, mint például a *stackframe_s* struktúra (4955–4974. sor), amely a regiszterek verembe mentését határozza meg, Intel-specifikusak. Más platformon a *stackframe_s* struktúra az ott használt CPU regisztereinek megfelelően épülne fel. Egy másik példa a *segdesc_s* struktúra (4979–4986. sor); ez része annak a védelmi mechanizmusnak, amely a processzusokat megakadályozza abban, hogy a hozzájuk rendelt memóriaterületen kívüli régiókhoz hozzáférjenek. Más CPU esetén a *segdesc_s* talán egyáltalán nem is létezne, ez attól függ, hogy a memóriavédelmet az hogyan oldja meg.

Ezekkel a struktúrákkal kapcsolatban még azt jegyezzük meg, hogy a szükséges adatok jelenléte még nem elegendő az optimális teljesítmény szempontjából. A *stackframe_s* kezelését assembly nyelvű programnak kell végeznie, ezért olyan formában kell definiálni, hogy minél gyorsabban lehessen írni és olvasni, ezáltal a processzusváltások ideje csökkenthető.

A következő fájl, a *proto.h* (5100. sor) tartalmazza az összes olyan függvény prototípusát, amelyeknek ismertnek kell lenniük azon az állományon kívül is, ahol definiálva vannak. Mindegyik az előző részben tárgyalt *_PROTOTYPE* makró segítségével van megadva, így a MINIX 3-kernel fordítható akár klasszikus C (Kernighan–Ritchie) fordítóval, mint amilyen az eredeti MINIX C fordító, vagy egy modern, szabványos ANSI C fordítóval, mint amilyen a MINIX 3-ban is található. Ezeknek a prototípusoknak egy része rendszerfüggő, mint például a megszakításkezelők, a kivételkezelők és az assembly nyelven írt függvények.

A *glo.h* (5300. sor) állományban a kernel globális változóit találjuk. Az *EXTERN* makró célját megismertük az *include/minix/const.h* leírásakor, rendes körülmények között behelyettesítéskor *extern* lesz az eredménye. Figyeljük meg, hogy a *glo.h* sok definícióját megelőzi ez a makró. Ha ezt az állományt a *table.c* fájl illeszti be, akkor az *EXTERN* üres értéket kap, mert ott a *_TABLE* makró definiálva van. Ezért az így definiált változók számára a tárolóhely lefoglalásra kerül, amikor a *table.c* fordításakor beilleszti a *glo.h*-t. A *glo.h* más kernelmodulok C forrásállományába történő beillesztése esetén a *table.c* változóit ott is ismertté teszi.

Az itt elhelyezett információs struktúrák közül néhányra szükség van indításkor. Az *aout* (5321. sor) a MINIX 3 összes rendszerkomponensének címét tartalmazó fejléc pointerét hordozza. Jegyezzük meg, hogy ezek **fizikai címek**, vagyis a processzor teljes címtartományának kezdetétől számítottak. Ahogy később látni fogjuk, a MINIX 3 indulásakor az *aout* fizikai címét a betöltési felügyelőprogram átadja a kernelnek, így a kernel inicializációs rutinjai az összes MINIX 3-komponens címéhez hozzájuthatnak a felügyelőprogram memóriájából. A *kinfo* (5322. sor) is fontos információt hordoz. Emlékezzünk vissza, hogy ez a struktúra az *include/minix/type.h*-ban van definiálva. Ahogy a betöltési felügyelőprogram az *aout* struktúrán keresztül ad át információt a processzusokról a kernelnek, a *kinfo* mezőit a kernel tölti fel, hogy magáról információt adjon a rendszer többi komponense számára.

A *glo.h* következő szakasza a processzusok vezérlésével és a kernel működésével kapcsolatos változókat tartalmaz. A *prev_ptr,* a *proc_ptr* és a *next_ptr* a processzustábla bejegyzéseire mutatnak, sorrendben az előzőleg futott, az éppen futó és a következőnek futó processzusra. A *bill_ptr* is egy processzustábla bejegyzésre mutat, azt jelzi, hogy melyik processzusnak számolja el a rendszer a felhasznált időt. Amikor egy felhasználói processzus meghívja a fájlrendszert, és a fájlrendszer fut, akkor a *proc_ptr* a fájlrendszer processzusra fog mutatni. A *bill_ptr* azonban a hívást kezdeményező felhasználóra fog mutatni, mert a fájlrendszer által

felhasznált időt rendszeridőként a hívóra kell terhelni. Nem hallottunk még olyan MINIX-rendszerről, amelynek a tulajdonosa megfizettette volna másokkal a felhasznált gépidőt, de meg lehetne tenni. A következő változó a *k_reenter,* és az egymásba ágyazott kernelhívások mélységét tartja nyilván. Ilyen például akkor történik, ha olyankor érkezik megszakítás, ha nem egy felhasználói processzus, hanem maga a kernel fut. Ez fontos, mert a felhasználói processzusok és a kernel közötti környezetváltások különböznek attól (és időigényesebbek), mintha újra belépnénk a kernelbe. Amikor egy megszakításkezelő befejeződik, akkor fontos megállapítani, hogy felhasználói processzust kell-e aktivizálni, vagy a kernelben kell maradni. Ezt a változót a megszakításokat engedélyező és letiltó függvények is vizsgálják, mint például a *lock_enqueue*. Ha egy ilyen függvény olyankor hajtódik végre, amikor a megszakítások már le vannak tiltva, akkor a végén nem biztos, hogy újra engedélyezni kell őket. Végül ebben a részben van egy elveszett órajeleket számláló változó. Az időzítőtaszk tárgyalása során térünk ki arra, hogy hogyan veszhet el ilyesmi, és mit lehet tenni, ha elveszett.

A *glo.h* utolsó változói azért kerültek ide, mert a kernelben mindenhol láthatónak kell lenniük, de deklarációjuk sorában az *EXTERN* helyett *extern* áll, mert ezek ún. **inicializált változók**. Ez a konstrukció a C nyelv része. Az *EXTERN* makró használata nem egyeztethető össze a C nyelvi inicializációval, mert minden változó kezdeti értéke csak egyszer állítható be.

A kernel területén futó taszkoknak, jelenleg az időzítőtaszknak és a rendszertaszknak, saját verem áll rendelkezésére a *t_stack* tömbben. Megszakításkezelés közben a kernel egy külön vermet használ, ez azonban nem itt van deklarálva, mert csak a megszakításkezelést végző assembly nyelvű rutin használja, így nem kell globálisnak lennie. A *kernel.h* által utolsóként beillesztett fájl, amely azért még mindig szerepel az összes fordításban, az *ipc.h* (5400. sor). A processzusok közötti kommunikáció során használt különböző konstansokat definiál. Később, a felhasználásuk helyén fogjuk ezeket tárgyalni, amikor a *kernel/proc.h* kerül sorra.

Több, széles körben használt kernel definíciós fájl van még; ezeket azonban a *kernel.h* nem illeszti be. Ezek közül az első a *proc.h* (5500. sor), amely egy processzustábla-bejegyzést definiál. Egy processzus állapotát a processzushoz tartozó memória tartalma és a processzustáblában róla tárolt információ együttese teljesen meghatározza. A CPU-regiszterek tartalma itt tárolódik, amikor a processzus nem fut, majd innen töltődnek fel, amikor a futása folytatódik. Ez teszi lehetővé annak az illúziónak a fenntartását, hogy több processzus fut egyszerre, és kölcsönhatásban van egymással, habár bármelyik időpillanatban egy CPU csak egyetlen processzus utasításaival tud foglalkozni. A CPU által a **környezetváltások** során a processzus állapotának mentésére és visszaállítására felhasznált idő szükséges, de nyilván ezalatt a processzusok tevékenysége fel van függesztve. Emiatt a struktúrák szerkezetét hatékonysági szempontból határozták meg. Ahogy a *proc.h* elején található megjegyzésből is kitűnik, sok assembly nyelvű rutin is hozzáfér ezekhez a struktúrákhoz, ezért egy másik definíciós fájl, az *sconst.h* olyan eltolási címeket definiál, amelyeket assembly programok használnak a processzustábla mezőinek használata közben. Emiatt ha a *proc.h*-ban változtatunk, akkor változtatásra kényszerülhetünk az *sconst.h*-ban is.

Mielőtt továbbmennénk, meg kell említenünk, hogy a MINIX 3 mikrokernel-architektúrája miatt a processzustáblához hasonló formában a processzuskezelő és a fájlrendszer is tart nyilván ezen komponensek működése szempontjából lényeges információt a processzusokról. Ez a három táblázat együtt megfelel a monolitikus operációs rendszerek processzustáblájának, de egyelőre, ha processzustábláról beszélünk, akkor a kernel processzustábláját értjük ezalatt. A többit később tárgyaljuk.

A processzustábla egy elemének típusát a *proc* struktúra (5516–5545. sor) adja. Minden ilyen táblaelem tartalmaz tárolóhelyet a regiszterek, a veremmutató, az állapotinformáció, a memóriatérkép, a maximális veremméret, a processzusazonosító, az elszámolási információ, a beállított időzítőadatok és az üzenetekkel kapcsolatos információk számára. Minden processzustábla-bejegyzés elején a *stackframe_s* struktúra található. Egy már a memóriában lévő processzust úgy hozunk futtatható állapotba, hogy a veremmutatójába betöltjük a hozzá tartozó processzustábla-bejegyzés címét, majd a CPU-regisztereket veremműveletekkel feltöltjük ebből a struktúrából.

Egy processzus állapotához azonban több minden tartozik, mint a regiszterek és a memória tartalma. A MINIX 3-ban minden processzus táblabejegyzésében van egy *priv* struktúrára mutató pointer (5522. sor). Ez a struktúra egyebek mellett engedélyeket tartalmaz arra nézve, hogy a processzus kinek küldhet és kitől fogadhat üzenetet. Ennek részleteit később tárgyaljuk. Egyelőre annyit jegyezzünk meg, hogy a rendszerprocesszusok mind egyedi engedélystruktúrával rendelkeznek, a felhasználói processzusok engedélyei viszont megegyeznek, ezért az ő pointereik mind ugyanarra az egyetlen struktúrára mutatnak. Van egy bitcsoportot összefogó bájt méretű mező, a *p_rts_flags* (5523. sor). E bitek jelentését alább megadjuk. Ha bármelyik bitbe 1 kerül, akkor a processzus nem futtatható, tehát a mező 0 értéke a futtathatóság feltétele.

A processzustábla bejegyzéseiben olyan információknak van hely fenntartva, amelyre a kernelnek szüksége lehet. Például a *p_max_priority* mező (5526. sor) azt határozza meg, hogy a processzus melyik ütemezési sorra kerüljön fel, amikor első alkalommal futásra kész állapotba kerül. Mivel a processzus prioritása csökkenhet, ha más processzusokat akadályoz, van egy *p_priority* mező is, amelynek kezdeti értéke megegyezik a *p_max_priority* értékével. Ténylegesen a *p_priority* határozza meg, hogy a processzus melyik sorra kerül, valahányszor futásra kész.

A processzusok által elhasznált időt két *clock_t* típusú változó tárolja (5532. és 5533. sor). Ezekhez a kernelnek hozzá kell férnie, és nem volna hatékony, ha a processzusok saját memóriaterületén tárolnánk, habár elvileg úgy is megoldható lenne. A *p_nextready* (5535. sor) segítségével vannak összeláncolva a processzusok az ütemezési sorokon.

A következő néhány mező a processzusok közötti üzenetekkel kapcsolatos információt tárol. Ha egy processzus nem tudja befejezni a send műveletet, mert a fogadó nem várakozik az adatátvitelre, akkor ez a küldő a címzett *p_caller_q* mutatója (5536. sor) által azonosított várakozó sorba kerül. Ilyen módon, amikor a célprocesszus végül belefog egy receive műveletbe, akkor könnyű megtalálni az

összes neki küldeni szándékozó processzust. A *p_q_link* mezőt (5537. sor) használjuk a várakozósor tagjainak összefűzésére.

A randevú jellegű üzenetátadást az 5538. és 5540. sor között lefoglalt tárolóhely teszi lehetővé. Ha egy processzus receive műveletet végezne, de nincs várakozó üzenet a számára, akkor a processzus blokkolódik, és annak a processzusnak a száma, amelytől üzenetet szeretne kapni, a *p_getfrom* mezőbe kerül. Hasonlóan, a *p_sendto* a címzett processzus számát tartalmazza, ha a processzus küldeni akar, de a címzett nem várakozik. Az üzenetpuffer címe a *p_messbuf* mezőben van tárolva. A processzustábla utolsó előtti bejegyzése a *p_pending* (5542. sor), egy bittérkép, amely azt tárolja, hogy mely beérkezett szignálokat nem dolgozta még fel a processzuskezelő (mert nem fogad éppen üzenetet).

Végül a processzustábla utolsó mezője egy karaktertömb, a *p_name,* amely a processzus nevét tárolja. A kernelnek a processzuskezeléshez nincs szüksége erre a mezőre. A MINIX 3 különböző **nyomkövetési listák**at tud készíteni, ha a konzolon speciális billentyűt ütünk le. Némelyik lista tartalmazhat információt az összes futó processzusról, ilyenkor egyéb adatok mellett a processzus neve is megjelenik. Ha minden processzusnak értelmes neve van, akkor a kernel működését könnyebb nyomon követni és megérteni.

A processzustábla-struktúra után a mezők értékeiként használható különböző konstansok következnek. A *p_rts_flags* bitjeinek beállításához használható értékek találhatók az 5548. és az 5555. sor között. Ha a bejegyzés nem használt, akkor *SLOT_FREE* van beállítva. Egy fork után a *NO_MAP* jelzi, hogy a gyermekprocesszus mindaddig nem futhat, amíg a memóriatérképe nincs beállítva. A *SENDING* és *RECEIVING* jelzi, hogy a processzus üzenet küldése vagy fogadása miatt blokkolva van. A *SIGNALED* és a *SIG_PENDING* azt jelzi, hogy valamilyen szignál érkezett, a *P_STOP* pedig a nyomkövetéshez nyújt segítséget. A *NO_PRIV* arra használható, hogy egy újonnan létrehozott rendszerprocesszus átmenetileg ne futhasson, amíg a beállítása teljeskörűen meg nem történt.

Ezt követően (5562–5567. sor) az ütemezési sorok száma és a *p_priority* mező megengedett értékei vannak definiálva. A fájl jelenlegi verziójában a felhasználói processzusok hozzáférhetnek a legnagyobb prioritású sorhoz is. Ez minden bizonnyal a meghajtók felhasználói módú tesztelésének kezdeti időszakából maradt vissza, a *MAX_USER_Q* értékét valószínűleg kisebb prioritásra (nagyobb számra) kell beállítani.

Ezután néhány makró következik, amelyek segítségével a processzustábla fontos részeinek címeit fordítási idejű konstansként lehet definiálni, hogy futás közben gyorsabb legyen a hozzáférés. Ezután pedig további, futási idejű számítások végzésére és tesztelésre alkalmas makrók következnek.

A *proc_addr* makró (5577. sor) azért kell, mert C-ben nem lehetséges negatív tömbindexeket használni. A *proc* tömb indexhatárai elvileg *-NR_TASKS* és *+NR_PROCS* lennének, de sajnos C-ben 0-val kell kezdődnie, így *proc*[0] a legnegatívabb taszkhoz tartozik, és így tovább. Az egymáshoz tartozó processzusok és bejegyzések megállapítását megkönnyítendő írhatjuk, hogy

```
rp = proc_addr(n);
```


így *rp* értékül kapja az *n*-edik processzushoz tartozó táblabejegyzés címét, legyen *n* akár pozitív, akár negatív.

Itt található a processzustábla definíciója is, *proc* struktúrák tömbjeként, mint *proc*[*NR_TASKS* + *NR_PROCS*] (5593. sor). Figyeljük meg, hogy az *NR_TASKS* definícióját az *include/minix/com.h* (3630. sor) tartalmazza, az *NR_PROCS* definícióját pedig az *include/minix/config.h* (2522. sor). Ezek együtt határozzák meg a kernel-processzustábla méretét. Az *NR_PROCS* megváltoztatásával olyan rendszert lehet létrehozni, amely több processzus kezelésére képes, ha szükséges (például egy nagyobb szerveren).

Végül a sebesség növelését célzó makrók definíciója került még ide. A processzustáblára gyakoriak a hivatkozások, egy tömbelem címének meghatározása pedig lassú szorzási művelet elvégzését igényli, ezért a processzustábla elemeire mutató pointerek *pproc_addr* tömbjét (5594. sor) hozzuk létre. A *rdy_head* és *rdy_tail* tömböket az ütemezési sorok tárolására használjuk. Például a felhasználói processzusok alapértelmezés szerinti sorának első elemére a *rdy_head*[*USER_Q*] mutat.

Ahogy a *proc.h* tárgyalásának elején említettük, van egy másik, *sconst.h* nevű fájl (5600. sor), amit a *proc.h*-val szinkronban kell tartani, ha a processzustáblában változtatunk. Az *sconst.h* assembly programok által használt konstansokat tartalmaz, az assembler által felhasználható formában. Ezek mind relatív címek a processzustábla *stackframe_s* struktúrájának kezdetéhez képest. Egyszerűbb ezeket a definíciókat külön állományban tartani, mert az assembly kódot a C fordító úgysem dolgozza fel. Másrészt ezek a definíciók mind gépfüggők is, ezért külön állományban tárolva egyszerűbb a MINIX 3 átvitele más gépre, mert egy másik processzorhoz úgyis másik *sconst.h* kell. Figyeljük meg, hogy sok relatív cím az előző plusz *W* alakban van megadva, ahol *W* a szóhosszal egyenlő (5601. sor). Ezzel a módszerrel ugyanaz a fájl használható a 16 és a 32 bites MINIX 3-verziókhoz is.

A többszörös definíciók okozhatnak problémát. A definíciós fájlokat azért alkalmazzuk, hogy egyetlen konzisztens definíciókészletünk legyen, amelyet aztán számos helyen használhatunk anélkül, hogy a részletekre különösebb figyelmet fordítanánk. A több helyen előforduló definíciók, mint például amelyeket a *proc.h* és az *sconst.h* tartalmaz, nyilván ellentmondanak ennek az alapelvnek. Ez természetesen egy speciális eset, de mint ilyen, speciális figyelmet igényel. Ha bármelyik fájl megváltozik, akkor ügyelnünk kell arra, hogy a kettő konzisztens maradjon.

A rendszerprivilégiumok struktúrája a *priv,* amelyet röviden említettünk a processzustábla ismertetése során, a *priv.h*-ban van definiálva (5718–5735. sor). Először van néhány jelzőbit, az *s_flags,* majd jönnek az *s_trap_mask, s_ipc_from, s_ipc_to* és *s_call_mask* mezők, amelyek meghatározzák, hogy mely rendszerhívások kezdeményezhetők, mely processzusokkal történhet üzenetátadás (-küldés vagy -fogadás), és mely kernelhívások megengedettek.

A *priv* struktúra nem része a processzustáblának, hanem minden processzustábla-bejegyzés tartalmaz egy ilyen típusú pointert. Csak a rendszerprocesszusoknak van saját példányuk; a felhasználói processzusok mind ugyanarra a példányra mutatnak. Így egy felhasználói processzus számára a fennmaradó bitek nem érde-

kesek, hiszen a megosztásnak nincs értelme. Ezek a mezők függőben lévő értesítések, hardvermegszakítások és szignálok bittérképei, illetve van egy időzítő is. A rendszerprocesszusok számára azonban van értelme ezeket definiálni. A felhasználói processzusok értesítéseit, szignáljait és időzítőit a processzuskezelő kezeli helyettük.

A *priv.h* szerkezete hasonló a *proc.h* szerkezetéhez. A *priv* struktúra definíciója után a jelzőbitekhez tartozó makródefiníciók jönnek, majd néhány fontos, fordítási időben ismert cím, végül futási idejű címszámítást végző makrók. Ezután egy *priv* struktúrákból álló táblázat, a *priv*[*NR_SYS_PROCS*] következik, amit egy pointerekből álló tömb követ, a *ppriv_addr*[*NR_SYS_PROCS*] (5762–5763. sor). A pointertömb gyors elérést biztosít, a processzustábla bejegyzéseinek elérését megkönnyítő pointertömbhöz hasonlóan. Az 5738. sorban definiált *STACK_GUARD* értéke egy könnyen felismerhető bitminta. A felhasználását később fogjuk látni. A kedves olvasót egy kis internetes keresésre bátorítjuk, hogy többet megtudjon ennek az értéknek a történetéről.

A *priv.h*-ban az utolsó tétel egy ellenőrzés, hogy az *NR_SYS_PROCS* értéke nagyobb-e, mint a betöltési memóriaképbe kerülő processzusok száma. Az #error direktíva hibaüzenetet ír ki, ha a feltétel igaz. Bár a különböző C fordítóprogramok eltérően viselkedhetnek, a szabványos MINIX 3 C fordítóprogram ennek hatására ki is lép.

Az F4 billentyű hatására egy olyan nyomkövetési listát kapunk, amely megmutat valamennyit a jogosultságtáblában tárolt információkból is. A 2.35. ábrán láthatjuk a tábla egy-két sorát néhány jellegzetes processzussal. A „flags" oszlop bejegyzéseinek magyarázata: P: időszelet lejárta miatt megszakítható (Preemptable); B: számlázási információ gyűjthető róla (Billable); S: rendszer (System). A „traps" bejegyzések magyarázata: E: echo; S: send; R: receive; B: mindkettő (Both); N: értesítés (Notification). A bittérképben az összes *NR_SYS_PROCS* darab (32) rendszerprocesszushoz tartozik egy bit, a sorrend az „id" mezőnek felel meg. (Az

--nr-	-id-	-name-	-flags-	-traps-	-ipc_to mask------
(-4)	(01)	IDLE	P-BS-	-----	00000000 00001111
[-3]	(02)	CLOCK	---S-	--R--	00000000 00001111
[-2]	(03)	SYSTEM	---S-	--R--	00000000 00001111
[-1]	(04)	KERNEL	---S-	-----	00000000 00001111
0	(05)	pm	P--S-	ESRBN	11111111 11111111
1	(06)	fs	P--S-	ESRBN	11111111 11111111
2	(07)	rs	P--S-	ESRBN	11111111 11111111
3	(09)	memory	P--S-	ESRBN	00110111 01101111
4	(10)	log	P--S-	ESRBN	11111111 11111111
5	(08)	tty	P--S-	ESRBN	11111111 11111111
6	(11)	driver	P--S-	ESRBN	11111111 11111111
7	(00)	init	P-B--	E--B-	00000111 00000000

2.35. ábra. *A jogosultságtábla nyomkövetési listájának részlete. Az időzítőtaszk, a fájlszerver, a tty és az init processzus jogosultságai jellemzők sorrendben a taszkokra, a szerverekre, az eszközmeghajtókra és a felhasználói processzusokra. A bittérkép 16 bitre lett rövidítve*

ábrán csak 16 bit látszik, hogy jobban elférjen a lapon.) Az összes felhasználói processzus osztozik a 0-s azonosítón, amely a bal szélső bitpozíció. A bittérkép azt mutatja, hogy a felhasználói processzusok, mint például az *init,* csak a processzuskezelőnek, a fájlszervernek és a reinkarnációs szervernek küldhetnek üzenetet, illetve csak a sendrec-et használhatják. Az ábrán látható szerverek és eszközmeghajtók bármelyik ipc alapműveletet használhatják, és a *memory* kivételével mindegyik küldhet bármelyik processzusnak.

Egy másik, sok helyre beillesztett definíciós fájl a *protect.h* (5800. sor). Ebben majdnem minden azoknak az Intel processzoroknak az architekturális részleteivel kapcsolatos, amelyek támogatják a védett üzemmódokat (80286, 80386, 80486 és a Pentium sorozat). Ezeknek a processzoroknak a részletes leírása túlmutat e könyv keretein. Elég annyi, hogy olyan belső regisztereket tartalmaznak, amelyek a memóriában elhelyezkedő **leírótáblák**ra mutatnak. A leírótáblák határozzák meg a rendszer erőforrásainak használatát, és megakadályozzák, hogy egyes processzusok mások memóriaterületéhez hozzáférjenek.

Ezenkívül a 32 bites Intel processzorok négy **jogosultsági szint** használatát teszik lehetővé, ezek közül a MINIX 3 hármat használ. Ezek szimbolikus definíciója az 5843. és 5845. sor között található. A kernel központi részei, amelyek a megszakításokat kezelik és a processzusok között váltanak, mindig *INTR_PRIVILEGE* jogosultsággal futnak. Nincs olyan CPU-regiszter vagy memóriarekesz, amely nem érhető el ezzel a jogosultsággal. A taszkok *TASK_PRIVILEGE* szinten futnak, ezzel elvégezhetik az I/O-műveleteket, de nem módosíthatnak például olyan speciális regisztereket, mint a leírótábla-mutatók. A szerverek és a felhasználói processzusok *USER_PRIVILEGE* jogosultsági szinten futnak. Ezek nem hajthatnak végre bizonyos utasításokat, ilyenek például az I/O-műveletek, memória-hozzárendelések vagy a jogosultsági szint megváltoztatása.

A jogosultsági szintek elve ismerős lesz a modern CPU-k felépítését ismerőknek, de nem biztos, hogy találkoztak már ilyen megszorításokkal azok, akik a számítógépek felépítését kis teljesítményű mikroprocesszorok assembly nyelvének tanulmányozásával ismerték meg.

A *kernel/* könyvtár egy definíciós állományáról még nem esett szó: ez a *system.h,* ennek tárgyalását elhalasztjuk a fejezet későbbi részére, ahol a rendszertaszkkal foglalkozunk. A rendszertaszk önálló processzusként fut annak ellenére, hogy a kernellel együtt fordítódik. Mostanra végigértünk a definíciós fájlokon, és készen állunk, hogy a **.c* végződésű C nyelvű forrásállományokat áttekintsük. Elsőként a *table.c* állományt nézzük meg (6000. sor). Lefordítása nem eredményez végrehajtható programot, de a tárgykód *(table.o)* tartalmazni fogja az összes kernel-adatszerkezetet. Ezen adatszerkezetek közül már soknak a definícióját láttuk a *glo.h* és más definíciós fájlok áttekintése során. A 6028. sorban, közvetlenül az #include direktívák előtt van a *_TABLE* makró definíciója. Ahogy korábban ismertettük, emiatt az *EXTERN* az üres értéket kapja, és minden deklaráció, amelyben ez szerepel, tárolóhelyet is lefoglal az adott változó számára.

A definíciós fájlokban található változódeklarációkon kívül van még két hely, ahol globális tárolóhely lefoglalása történik. Néhány definíciónak közvetlenül a *table.c* ad helyet. A 6037–6041. sorokban a rendszerkomponensek veremtárainak

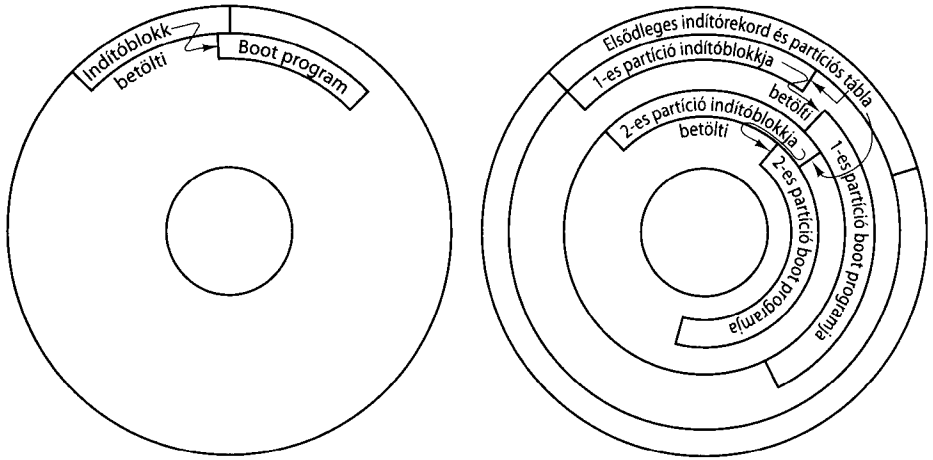
méreteit definiáljuk, és a taszkok számára szükséges teljes veremtárat lefoglaljuk a *t_stack*[*TOT_STACK_SPACE*] tömbbel (6045. sor).

A *table.c* fennmaradó része a processzusok tulajdonságaihoz kapcsolódó számos konstanst definiál, mint például jelzőbitek kombinációit, híváscsapdákat (call traps) és olyan bitmaszkokat, amelyekkel meghatározható, hogy ki kinek küldhet üzeneteket és értesítéseket (6048–6071. sor). Ilyet láttunk a 2.35. ábrán is. Ezt követően olyan maszkok következnek, amelyek definiálják a különféle processzszusoknak engedélyezett kernelhívásokat. A processzuskezelő és a fájlszerver is egyedi engedélyt kap. A reinkarnációs szerver minden kernelhívást elérhet, nem a saját használatára, hanem azért, mert mint a többi rendszerprocesszus szülője, gyermekeinek csak olyan jogosultságokat adhat, ami neki is megvan. Az eszközmeghajtók közös engedélykészletet kapnak a kernelhívásokhoz, kivétel ez alól a RAM-lemezmeghajtó, amelynek szokatlan memória-hozzáférésre van szüksége. (Figyeljük meg, hogy a 6075. sorban lévő megjegyzés „system services manager"-re hivatkozik, de helyette „reincarnation server" kellene – a név megváltozott fejlesztés közben, néhány megjegyzés még a régi nevet tartalmazza.)

Végül a 6095. és a 6109. sor között az *image* tábla definíciója található. Azért helyeztük el itt a definíciós fájl helyett, mert a többszörös deklarációt megakadályozó *EXTERN* trükk az inicializált változók esetében nem működik; vagyis nem írhatjuk bárhol azt, hogy

```
extern int x = 3;
```

Az *image* tábla tartalmazza azokat a részleteket, amelyek a betöltési memóriaképben tárolt processzusok inicializálásához szükségesek. A rendszer induláskor



2.36. ábra. *Indításhoz használt lemezek szerkezete. (a) Particionálás nélküli lemez. Az első szektor az indítóblokk. (b) Particionált lemez. Az első szektor az elsődleges indítórekord (masterboot)*

fogja használni. Példaként erre, tekintsük a 6096. sor megjegyzésében „qs"-nek nevezett mezőt. Ez az egyes processzusokhoz rendelt időszelet méretét mutatja. A közönséges felhasználói processzusok az *init* gyermekeiként 8 óraütemet kapnak a futásra. A CLOCK és a SYSTEM taszk 64 óraütemig futhatnak, ha szükséges. Igazából nem várható, hogy olyan sokáig fussanak blokkolódás nélkül, de a felhasználói szintű szerverektől és az eszközmeghajtóktól eltérően ezeket nem lehet egyre alacsonyabb prioritású sorokba áthelyezni, ha akadályoznak más processzusokat a futásban.

Ha új processzust akarunk tenni a betöltési memóriaképbe, akkor az *image* táblába fel kell venni egy új sort. Megengedhetetlen, hogy az *image* tábla mérete ne legyen összeegyeztethető más konstansok értékével. A *table.c* végén egy kis trükkel ellenőrizzük, hogy ez a hiba bekövetkezett-e. Kétszer is deklaráljuk a *dummy* tömböt, amely hiba esetén lehetetlen méretű lenne, így fordítási hibát okozna. Mivel e felesleges tömb előtt *extern* áll, így nem foglalunk neki helyet itt (és máshol sem). A programkódban sehol sem hivatkozunk rá; ez a fordítót nem fogja zavarni.

További globális helyfoglalás történik az assembly nyelvű *mpx386.s* nevű fájl végén. Bár ez később következik, mégis itt érdemes tárgyalnunk, mert a globális helyfoglalás a témánk. A 6822. sorban a .sect .rom assembly direktíva egy (a valódi MINIX 3-kernel azonosítására szolgáló) **mágikus szám**ot helyez el a kerneladatszegmens legelején. Egy .sect bss assembler direktíva és a .space pszeudoutasítás is található itt a kernel veremtárának lefoglalásához. A .comm pszeudoutasítás a verem tetején címkével lát el néhány memóriaszót, hogy közvetlenül is manipulálhatók legyenek. Az *mpx386.s*-hez visszatérünk még néhány oldallal később, miután a MINIX 3 elindulását áttekintettük.

2.6.6. A MINIX 3 indítása

Már majdnem itt az idő, hogy szemügyre vegyük a végrehajtható programot – de még nem egészen. Mielőtt ezt megtennénk, tekintsük át röviden, hogyan töltődik be a MINIX 3 a memóriába. A betöltés természetesen egy mágneslemezről történik, de a folyamat nem teljesen magától értetődő, és az események pontos sorrendje a lemez fajtájától függ. A 2.36. ábra mutatja egy hajlékonylemez és egy partíciókra osztott merevlemez szerkezetét.

Amikor a rendszer indul, a hardver (pontosabban egy ROM-ban lévő program) beolvassa az indítólemez első szektorát, és végrehajtja az ott található programot. Egy partíciók nélküli MINIX 3-hajlékonylemezen az első szektor egy indítóblokk, amely betölti a *boot* programot, ahogy az a 2.36.(a) ábrán látható. A merevlemezek particionáltak, és az első szektorban lévő program (a MINIX-rendszerekben *masterboot* a neve) először áthelyezi magát egy másik memóriaterületre, majd beolvassa a vele együtt az első szektorból betöltött partíciós táblát. Ezután betölti és végrehajtja az aktív partíció első szektorát, ahogy az a 2.36.(b) ábrán látható. (Rendszerint pontosan egy partíció van aktívként megjelölve.) Egy MINIX 3-partíciónak ugyanolyan a szerkezete, mint egy particionálás nél-

küli MINIX 3-hajlékonylemeznek, azaz van egy indítóblokkja, amely betölti a *boot* programot. A particionált és a nem particionált lemezek indítóblokkjának programkódja megegyezik. Mivel a *masterboot* program áthelyezi magát, az indítóblokk megírható úgy, hogy ugyanazon a memóriacímen kezdődjön, ahova a *masterboot* eredetileg betöltődik.

A valóságban a helyzet az ábrán láthatónál egy kicsit bonyolultabb is lehet, mert egy partíció alpartíciókat is tartalmazhat. Ebben az esetben a partíció első szektora az alpartíciók partíciós tábláját tartalmazó elsődleges indítórekord lesz. Végül azonban a vezérlés mindenképpen átadódik egy indítószektorra, egy olyan egység első szektorára, amely nincs tovább osztva. Egy hajlékonylemezen az első szektor mindig egy indítószektor. A MINIX 3 ekkor is megenged egy bizonyosfajta particionálást, de csak az első partíció lehet betölthető; nincs külön elsődleges indítórekord, és alpartíciók sem lehetségesek. Ez lehetővé teszi, hogy a particionált és particionálás nélküli lemezeket ugyanolyan módon lehessen csatolni. Egy particionált hajlékonylemez legfőbb haszna abban van, hogy az indítólemez feloszthatő egy RAM-lemezre másolható alaprészre és egy csatolható kiegészítő részre. Ez utóbbi leválasztható, ha már nincs rá tovább szükség, és így a lemezmeghajtó felszabadul a telepítés folytatásához.

A MINIX 3 indítószektorát egy *installboot* nevű speciális program hozza létre. Ez a lemezre íráskor az indítószektorba beleírja az (al)partícióján elhelyezkedő *boot* nevű program lemezcímét. A MINIX 3 *boot* programjának szabványos helye egy ugyanilyen nevű könyvtárban van, vagyis */boot/boot*. Bárhol lehet azonban – az *installboot* a lemezcímet helyezi el, ahonnan be kell tölteni. Ez azért szükséges, mert a *boot* betöltése előtt nem lehet könyvtárakat és fájlneveket használni egy állomány megtalálásához.

A *boot* a MINIX 3 másodlagos betöltője. Az operációs rendszer betöltésénél azonban egy kicsit többet tud, mert ez egy **felügyelőprogram** is egyben, amelynek segítségével a felhasználó megváltoztathat, beállíthat, és elmenthet különféle paramétereket. A *boot* a partíciója második szektorában keresi az érvényben lévő paramétereket. A MINIX 3 ugyanúgy, mint a szabványos UNIX, lefoglalja minden lemez első 1 K méretű blokkját **indítóblokk**nak, de a ROM-indító vagy az elsődleges indítórekord csak az első 512 bájtot tölti be, így 512 bájt rendelkezésre áll a paraméterek tárolására. Ezek vezérlik a betöltési műveletet, és maga az operációs rendszer is megkapja őket. Az alapértelmezés szerinti beállítás egyetlen menüpontot kínál fel, ez a MINIX 3 elindítása, de ki lehet bővíteni, így például más operációs rendszereket lehet indítani (más partíciók indítószektorának betöltése és elindítása révén), vagy a MINIX 3-at különféle beállításokkal is lehet indítani. Az alapértelmezés szerinti beállításokat úgy is megváltoztathatjuk, hogy a menü kihagyásával a MINIX 3 azonnal induljon.

A *boot* nem az operációs rendszer része, de elég okos ahhoz, hogy a fájlrendszer adatszerkezeteit felhasználva megtalálja az operációs rendszert a lemezen. A *boot* az *image*= indítóparaméter által megadott fájlt keresi, amely alapértelmezés szerint a */boot/image*. Ha ilyen névvel létezik egy közönséges fájl, akkor azt tölti be, ha azonban ez egy könyvtár, akkor abból a legújabb állományt választja ki. A legtöbb operációs rendszernél a betöltési memóriaképnek egy előre megadott nevű állományban kell lennie. A MINIX 3 azonban arra bátorítja a felhasználókat, hogy módosítsák és hozzanak létre új kísérleti verziókat. A felhasználók számára hasznos, hogy többféle változat közül választhatnak, így visszatérhetnek egy korábbi működő változathoz, ha netán egy kísérlet kudarcot vall.

Terjedelmi okokból nem tudunk részletesebben foglalkozni a betöltési felügyelőprogrammal. Ez egy összetett program, majdnem egy miniatűr operációs rendszer önmagában is. Együttműködik a MINIX 3-mal, és ez kapja vissza a vezérlést, amikor a MINIX 3-at szabályszerűen állítjuk le. Ha az olvasó többet akar tudni róla, a MINIX 3 honlapjáról elérhető a betöltési felügyelőprogram forráskódja részletes magyarázatokkal.

A MINIX 3 **betöltési memóriakép** (más néven **rendszer-memóriakép**) több programfájl egymáshoz illesztésével keletkezik: kernel, processzuskezelő, fájlrendszer, reinkarnációs szerver, eszközmeghajtók és az *init*, ahogy az a 2.30. ábrán látható. Megjegyezzük, hogy az itt ismertetett MINIX 3-konfiguráció csak egyetlen lemezes eszközmeghajtót tartalmaz a betöltési memóriaképben, de több is lehetne benne, amelyek közül egy címkével választható ki az aktív. Mint minden bináris program, a betöltési memóriaképben eltárolt fájlok is tartalmaznak egy rövid fejlécet, amely meghatározza, hogy betöltés után mennyi memóriát kell lefoglalni az adatoknak és a veremnek, így a következő program mindig a megfelelő címre kerülhet.

A hardvertől függ, hogy a memória mely területei állnak rendelkezésre a betöltési felügyelőprogram és a MINIX 3 komponensei számára. Ezenkívül némelyik architektúra esetén a végrehajtható programban lévő címeket ki kell igazítani a konkrét betöltési cím függvényében. Az Intel processzorok szegmentált architektúrája ezt szükségtelenné teszi.

A betöltés részletei a gép típusától függően változnak. Az a fontos, hogy így vagy úgy, az operációs rendszer betöltődik a memóriába. Miután a betöltés befejeződött, kisebb előkészületeket kell tenni a MINIX 3 elindításához. Először is, betöltés közben a *boot* kiolvas néhány bájtot a betöltési memóriaképből, amelyekből kiderül néhány fontos tulajdonsága, például az, hogy 16 vagy 32 bites módban kell-e futtatni. Ezután a kernel megkap néhány, a rendszer indításához szükséges paramétert. A MINIX 3 betöltési memóriakép komponenseinek *a.out* fejlécei átkerülnek a *boot* memóriaterületén elhelyezkedő tömbbe, amelynek báziscíme a kernelnek is átadódik. A MINIX 3 vissza tudja adni a vezérlést a betöltési felügyelőprogramnak, ezért a visszatérési címet is meg kell határozni. Ezek az elemek a verembe kerülnek, ahogy később látni fogjuk.

A betöltési felügyelőprogramnak még sok egyéb információt – az **indítóparamétereket** – is át kell adnia az operációs rendszernek. Ezek közül néhányra szüksége van a kernelnek, mások csak kiegészítő információt hordoznak, mint például a betöltési memóriakép neve. Ezek a paraméterek mind megadhatók *név=érték* alakban, táblázatuk címe a veremben kerül átadásra. A 2.37. ábra egy tipikus indítóparaméter-listát mutat abban a formában, ahogy a MINIX 3-parancssorból indítható sysenv parancsa megjeleníti.

Ebben a példában hamarosan újra felbukkan a fontos *memory* paraméter. Ez esetben azt jelzi, hogy a betöltési felügyelőprogram két memóriaszegmenst talált a

```
rootdev=904
ramimagedev=904
ramsize=0
processor=686
bus=at
video=vga
chrome=color
memory=800:92540,100000:3DF0000
label=AT
controller=c0
image=boot/image
```

2.37. ábra. *A kernelnek átadott indítóparaméterek egy tipikus MINIX 3-rendszerben*

MINIX 3 számára. Az egyik a hexadecimális 800 (decimális 2048) címen kezdődik, és mérete hexadecimális 0x92540 (decimálisan 599 360) bájt. A másik a 0x100000 (1 048 576) címen kezdődik, és 0x3df00000 (64 946 176) bájt méretű. Ez a legrégebbi PC-kompatibilis gépeket leszámítva tipikus. Az eredeti IBM PC-ben csak olvasható memória került a felhasználható memória felső végébe, amit 1 MB-ra korlátozott a 8088-as CPU. A mai PC-kompatibilis gépeknek az eredeti PC-nél több memóriájuk van, de kompatibilitási okokból a régi gépekkel megegyező címen ezeknek is csak olvasható memóriájuk van. Így az írható-olvasható memória nem folytonos, egy ROM-tartomány van az alsó 640 KB és az 1 MB feletti felső rész között. A betöltési felügyelőprogram az alsó memóriatartományba tölti a kernelt, de a szervereket, a meghajtókat és az *init*-et lehetőség szerint a ROM fölé. Ez elsődlegesen a fájlrendszer érdekében történik, mert az így nagy gyorsítótárat használhat a lemezblokkokhoz anélkül, hogy a ROM-részbe ütközne.

Meg kell jegyeznünk, hogy az operációs rendszerek nem mindig lokális lemezről töltődnek be. **Lemez nélküli munkaállomások** hálózaton keresztül, távoli lemezről is betölthetik az operációs rendszerüket. Ehhez természetesen ROM-ban elhelyezett hálózati szoftverre van szükség. Bár a részletekben lehetnek különbségek, ennek a folyamatnak a fázisai valószínűleg hasonlók az általunk elmondottakhoz. A ROM-programnak annyira kell okosnak lennie, hogy a hálózatról szerezzen egy végrehajtható fájlt, amely aztán behozza a teljes operációs rendszert. Ha a MINIX 3 ilyen módon töltődne be, nagyon kevés változtatásra lenne szükség az operációs rendszer kódjának memóriába töltése utáni inicializálási folyamatban. Szükség lenne természetesen egy hálózati szerverre és egy módosított fájlrendszerre, amely a hálózaton keresztül is el tud érni állományokat.

2.6.7. A rendszer inicializálása

A MINIX korábbi verziói 16 bites módban is lefordíthatók voltak, ha szükséges volt a régebbi processzorokkal való kompatibilitás, és a MINIX 3 is tartalmaz még valamennyit a 16 bites módú forráskódokból. Az itt ismertetett és a CD-n található verzió csak 80386-os, vagy annál jobb processzorral felszerelt 32 bites rend-


```
#include <minix/config.h>
#if _WORD_SIZE == 2
#include "mpx88.s"
#else
#include "mpx386.s"
#endif
```

2.38. ábra. *Alternatív assembly nyelvű forrásállományok közötti választás*

szereken használható. Nem működik 16 bites módban, és 16 bites verzió létrehozása bizonyos funkciók eltávolítását teheti szükségessé. Többek között a 32 bites tárgykódok nagyobbak a 16 biteseknél, és az egymástól független felhasználói szintű eszközmeghajtók nem használhatnak közösen programkódot olyan módon, ahogy egyetlen tárgykódú állományba fordítva tehetnék. Mindazonáltal ugyanazt a C forráskódot használhatjuk mindkét esetben, a kimenet attól függ, hogy a 16 vagy 32 bites verziójú fordítóprogrammal fordítunk-e. A fordítóprogram által definiált makró határozza meg az *include/minix/sys_config.h* állományban definiált *_WORD_SIZE* értékét is.

A MINIX 3 először végrehajtódó része assembly nyelven íródott, és különböző forrásállományokat kell használnunk a 16 bites, illetve a 32 bites fordító esetén. A kezdeti értékeket beállító 32 bites programkódot az *mpx386.s* fájl tartalmazza. A 16 bites alternatíva az *mpx88.s*. Mindkettő tartalmaz még assembly nyelvű támogatást más alacsony szintű kernelműveletekhez is. A választás automatikusan történik az *mpx.s* segítségével. Ez a fájl olyan rövid, hogy az egész elfér a 2.38. ábrán.

Az *mpx.s* a C előfeldolgozó #include direktívájának ritkán előforduló felhasználását illusztrálja. A szokásos #include direktívát használhatjuk állományok beillesztésére, de alternatív forráskódrészek közötti választásra is. Ha #if direktívákkal akarnánk ezt megtenni, akkor az egyenként is nagyméretű *mpx88.s* és *mpx386.s* fájl tartalmát egyetlen állományba kellene bezsúfolni. Ez nemcsak nehezen kezelhető lenne, de szükségtelenül nagy lemezterületet is foglalna, mert valószínűleg egy adott gépre történő telepítéskor csak az egyikre van szükség, a másikat törölni vagy archiválni lehet. A következőkben az *mpx386.s* 32 bites változatot használjuk.

Most fogunk először találkozni a végrehajtható programmal, ezért néhány szót szólunk arról, hogyan fogjuk ezeket bemutatni a könyvben. Egy nagy C program fordítása során használt sok forrásfájl nehezen átlátható. Általában egyszerre egy állományra fogunk koncentrálni. Az állományokon abban a sorrendben haladunk végig, ahogy a CD-n lévő forráskódban megjelennek. A MINIX 3-rendszer részeinek belépési pontjaival kezdjük, és a végrehajtás fő vonalát fogjuk követni. Ha egy kisegítő függvény hívásához érünk, akkor a hívás céljáról mondunk néhány szót, de a hívott függvény belső működését illetően általában nem megyünk bele a részletekbe, ezt majd a függvény definíciójánál tesszük meg. A fontos alárendelt függvények általában a magasabb szintű hívó függvények után ugyanabban az állományban vannak definiálva, ahol hívjuk őket, de kisebb vagy általános célú függvények néha külön állományba kerülnek összegyűjtve. Nem kíséreljük meg az összes függvény belső működését leírni. Próbáltuk a gépfüggő és gépfüggetlen részeket külön állományokba elhelyezni, ezzel is elősegítve a hordozhatóságot.

A mellékelt CD forráskönyvtáraiban és a MINIX 3 weboldalán az összes fájl teljes verziója rendelkezésre áll.

Nagy gondot fordítottunk arra, hogy a programkód emberi fogyasztásra alkalmas legyen. Ennek ellenére egy nagy programban sok elágazás van, egy függvény megértéséhez néha el kell olvasnunk az általa hívott függvényeket is, ezért az anyag általunk megadottól eltérő sorrendben történő tanulmányozása is segíthet a megértésben.

Miután lefektettük a programkód tanulmányozásának alapelveit, rögtön azzal kell kezdenünk, hogy miért tettünk kivételt egy esetben. A MINIX 3 elindulása során a vezérlés az *mpx386.s* assembly rutinjai, valamint a *start.c* és a *main.c* állományok C rutinjai között kalandozik. Ezeket a rutinokat a végrehajtás sorrendjében írjuk le, még akkor is, ha ezáltal ide-oda kell ugrálnunk az állományok között.

Amint az operációs rendszer betöltése befejeződött, a vezérlés a *MINIX* címkére adódik (*mpx386.s,* 6420. sor). Az első utasítás átugrik néhány adatbájtot; ezek között vannak a korábban említett betöltési felügyelőprogram jelzőbitjei (6423. sor). Mostanra ezek betöltötték funkciójukat, a felügyelőprogram beolvasta őket, amikor a kernelt betöltötte a memóriába. Azért vannak éppen itt, mert ez egy könnyen megadható memóriacím. Elsődlegesen a kernel jellegzetességeinek azonosítására szolgálnak, mindenekelőtt arra, hogy 16 bites vagy 32 bites rendszerrel állunk-e szemben. A betöltési felügyelőprogram mindig 16 bites módban indul, de szükség esetén átkapcsolja a CPU-t 32 bites módba. Ez még azelőtt történik, mielőtt a vezérlés a *MINIX* címkéhez ér.

Könnyebben megértjük a következő programrészeket, ha ismerjük a verem állapotát ezen a ponton. A felügyelőprogram számos paramétert ad át a MINIX 3-nak a vermen keresztül. A felügyelőprogram először az *aout* változó címét helyezi el a verembe, ez egy tömb címét tartalmazza, amelyben a betöltési memóriakép komponenseiből kinyert fejléc-információk találhatók. Ezt követően az indítóparaméterek mérete és címe kerül a verembe. Ezek mind 32 bites mennyiségek. Utána jön a felügyelőprogram kódszegmensének címe, és az a hely, ahol a végrehajtást folytatni kell a felügyelőprogramban, amikor a MINIX 3 kilép. Ezek mind 16 bites mennyiségek, mivel a felügyelőprogram 16 bites védett üzemmódban működik. Az *mpx386.s* első néhány utasítása a felügyelőprogram által használt 16 bites veremmutatót konvertálja a védett üzemmódban használatos 32 bitesre. Ezután a

```
mov ebp, esp
```

utasítás (6436. sor) a veremmutató értékét az ebp regiszterbe másolja, hogy így báziscímként használva a veremből ki lehessen olvasni a felügyelőprogram által ott elhelyezett értékeket, ahogy a 6464. és a 6467. sor között látható. Figyeljünk arra, hogy az Intel processzoroknál a verem lefele növekszik, ezért a 8(ebp) arra az értékre hivatkozik, amelyet a 12(ebp) által hivatkozott érték után tettünk a verembe.

Az assembly nyelvű programnak tekintélyes mennyiségű munkát kell elvégeznie, előkészíteni egy vermet, hogy a C fordító által lefordított program megfelelő környezetben futhasson, létre kell hoznia a processzor által a memóriaszegmensek definiálásához használt táblázatokat, valamint be kell állítania egyéb regisztereket. Amint ez megvan, az inicializáció a *cstart* C függvény (a *start.c*-ben van, ez következik) hívásával folytatódik (6481. sor). Figyeljünk arra, hogy az assembly programban erre *_cstart* néven hivatkozunk. Ez amiatt van, mert minden, a C fordító által fordított függvény neve elé egy aláhúzás karakter kerül a szimbólumtáblákban, és a szerkesztőprogram ilyen neveket keres, amikor a külön fordított modulokat összeszerkeszti. Mivel az assembler nem tesz aláhúzásjelet a nevek elé, ezt az assembly nyelvű program írójának kell megtennie, hogy a C fordító által létrehozott tárgykódú állományban a szerkesztőprogram megtalálja őket.

A *cstart* egy másik rutint hív, amely inicializálja a **globális leírótáblát (Global Descriptor Table)**, amely a 32 bites Intel processzorok memóriavédelmet felügyelő központi adatszerkezete, valamint a **megszakításleíró táblát (Interrupt Descriptor Table)**, amely a lehetséges megszakítások beérkezése esctén végrehajtandó programok kiválasztására használatos. A *cstart* végrehajtása után az lgdt és lidt utasítások (6487. és 6488. sor) feltöltik a megfelelő regisztereket a táblák címével, és ezáltal használatba veszik őket. A

```
jmpf CS_SELECTOR:csinit
```

utasítás úgy néz ki, mintha nem lenne semmi hatása, mert pontosan oda adja a vezérlést, ahova akkor kerülne, ha nop utasítások lennének a helyén. Ez azonban az inicializálás fontos része. Ez az ugrás kikényszeríti az előbb inicializált adatszerkezetek használatát. A processzor regisztereinek némi további manipulációja után, a 6503. sorban a *MINIX* egy ugrással (nem függvényhívással) a kernel *main* fő belépési pontjára (lásd *main.c*) adja a vezérlést. Ennél a pontnál az *mpx386.s* inicializáló programja teljesen lefutott. A fájl maradék része olyan kódrészleteket tartalmaz, mint például egy taszk vagy processzus indítása és újraindítása, megszakításkezelés és más kisegítő rutinok, amelyeket a hatékonyság érdekében assembly nyelven kellett írni. Ezekhez még visszatérünk a következő részben.

Most a legfelső szinten elhelyezkedő C inicializáló függvényeket tekintjük át. Az az általános alapelv, hogy amit csak lehet, magas szintű C programmal valósítsunk meg. Amint láttuk, már eddig is két *mpx* fájl van, így ha innen bármit is C programmal tudunk helyettesíteni, azzal két assembly kódrészletet küszöbölünk ki. A *cstart* (lásd *start.c,* 6920. sor) első dolgai között van a CPU védelmi rendszerének és megszakítástábláinak beállítása a *prot_init* hívásával. Ezután az indítóparamétereket átmásolja a kernel memóriájába, majd a *get_value* függvényt (6997. sor) felhasználva paraméternevekhez tartozó értékeket keres.

Ez a folyamat meghatározza a monitor típusát, a processzor típusát, a sín típusát, és ha 16 bites módban fut, akkor a processzor működési módját (valós vagy védett). Mindez az információ globális változókban kerül tárolásra abból a célból, hogy a kernel bármelyik részének rendelkezésére álljon, ha szükséges.

A *main* (lásd *main.c,* 7130. sor) befejezi az inicializálást, és megkezdi a rendszer normális működését. Az *intr_init* hívásával konfigurálja a megszakításvezérlő hardvert. Erre azért itt kerül sor, mert addig nem lehetséges, amíg a gép típusát nem ismerjük. (Az eljárás annyira gépfüggő, hogy külön állományba került, ez

hamarosan sorra kerül.) A hívás paramétere (1) azt jelenti az *intr_init* számára, hogy a MINIX 3 inicializálását kell elvégezni. A (0) paraméterrel híva a MINIX 3 leállásakor visszaállítja a hardvert az eredeti állapotra, hogy vissza lehessen térni a betöltési felügyelőprogramba. Az *intr_init* biztosítja, hogy az inicializáció befejezése előtt érkezett megszakításoknak ne legyen semmilyen hatása. Később elmagyarázzuk, hogyan teszi ezt.

A *main* legnagyobb része a processzustábla és a jogosultsági tábla felépítésének van szentelve, így amikor az első taszkok és processzusok beütemeződnek, a memóriatérképük, a regisztereik és a jogosultságaik megfelelően be lesznek állítva. A processzustábla minden bejegyzését szabadra állítjuk, és a gyors elérést biztosító *pproc_addr* tömböt is feltöltjük a 7150. és 7154. sor közötti ciklusban. A 7155. és a 7159. sor közötti ciklus törli a jogosultsági tablet, és feltölti a *ppriv_addr* tömböt, a processzustáblához és az elérését gyorsító tömbhöz hasonlóan. Mind a processzustáblában, mind a jogosultsági táblában elegendő az egyik mezőt egy meghatározott értékkel feltölteni ahhoz, hogy egy bejegyzés szabad voltát jelezzük. Azonban mindkét táblában minden bejegyzést inicializálni kell egy indexértékkel, akár szabad, akár nem.

Mellékesen egy apróság a C nyelvvel kapcsolatban: a 7153. sorban a

```
(pproc_addr + NR_TASKS)[i] = rp;
```

utasítást írhattuk volna

```
pproc_addr[i + NR_TASKS] = rp;
```

alakban is, mert a C nyelvben $a[i]$ csak egy alternatív jelölés $*(a + i)$ helyett. Így mindegy, hogy a-hoz vagy i-hez adunk hozzá egy állandót. Némelyik C fordítóprogram egy kicsit jobb kódot generál, ha az állandót a tömbhöz adjuk hozzá, és nem az indexhez. Hogy esetünkben van-e különbség, azt nem tudjuk megmondani.

Elérkeztünk a 7172. és 7242. sor között található hosszú ciklushoz, amely a betöltési memóriaképben elhelyezkedő processzusok futtatásához szükséges információkkal tölti fel a processzustáblát. (Figyeljük meg, hogy van egy idejétmúlt megjegyzés a 7161. sorban, amely csak taszkokat és szervereket említ.) Ezen processzusok mindegyikének jelen kell lennie induláskor, és normális működés közben nem is állnak le. A ciklus elején az *ip* értékül kapja egy bejegyzés címét a *table.c*-ben létrehozott *image* táblából (7173. sor). Mivel az *ip* egy struktúrára mutató pointer, a struktúra elemei az *ip->proc_nr* és ehhez hasonló jelölésekkel érhetők el, ahogy az a 7174. sorban is látható. Ezt a fajta jelölést kiterjedten használjuk a MINIX 3-forráskódban. Hasonlóan, az *rp* egy processzustábla-bejegyzésre mutató pointer, a *priv(rp)* pedig a jogosultsági tábla egy bejegyzésére mutat. A hosszú ciklusban található processzustábla- és jogosultságitábla-inicializáció nagy része abból áll, hogy az *image* táblából átírunk értékeket a processzustáblába és a jogosultsági táblába.

A 7185. sorban ellenőrizzük, hogy az aktuális processzus a kernel része-e, és ha igen, akkor a speciális *STACK_GUARD* bitminta kerül be a taszk vermének aljára. Ezt később megvizsgálva el lehet dönteni, hogy a verem túlcsordult-e. Ezután a taszkok kezdeti veremmutatói kerülnek beállításra. Minden taszknak külön veremmutató kell. Mivel a verem az alacsonyabb memóriacímek felé növekszik, a veremmutató kezdőértékét úgy lehet kiszámítani, hogy a báziscímhez hozzáadjuk a méretet (7190. és 7191. sor). Van egy kivétel: a *KERNEL* processzust (néhány helyen *HARDWARE* a neve) soha nem tekintjük futásra késznek, soha nem fut hagyományos processzusként, ezért nincs szüksége veremmutatóra.

A betöltési memóriakép komponenseinek tárgykódjai ugyanúgy fordítódnak, mint bármelyik másik MINIX 3-program, a fordító a fájlok elején egy fejlécet hoz létre, amelyet az *include/a.out.h* definiál. A betöltési felügyelőprogram a fejléceket a MINIX 3 indulása előtt a saját memóriaterületére másolja, majd amikor átadja a vezérlést a *MINIX* belépési pontra az *mpx386.s*-ben, akkor a fejlécek táblázatának fizikai címe a vermen keresztül az assembly programhoz kerül, ahogy azt már láttuk. A 7202. sorban a fejlécek egyike egy lokális *exec* típusú *e_hdr* struktúrába kerül, a *hdrindex*-et használva indexként a fejlécek táblázatában. Ezután az adatszegmens és a kódszegmens címek memóriaszelet egységekre konvertálása következik, hogy a processzus memóriatérképébe elhelyezhessük (7205–7214. sor).

Meg kell említenünk néhány dolgot, mielőtt továbbmennénk. Először is, a kernelprocesszusok esetén a *hrindex* mindig 0 értéket kap a 7194. sorban. Ezek a processzusok a kernellel egy fájlba fordítódnak, a veremszükségletükkel kapcsolatos információk pedig az *image* táblában vannak. Mivel egy kernelbe fordított taszk a kernel címterületén lévő bármilyen kódot meghívhat, illetve bármely adatot elérhet, ezért egy taszk méretéről nincs értelme beszélni. Így az *aout*-nak ugyanaz az eleme vonatkozik a kernelre és a taszkokra is, a taszkok méretmezői pedig a kernel adataival kerülnek feltöltésre. A taszkok a vereminformációkat az *image* táblából kapják, amely pedig fordításkor a *table.c*-ben inicializálódik. A kernelprocesszusok feldolgozása után a *hrindex* a ciklus minden lefutásakor növekszik eggyel (7196. sor), tehát a felhasználói szintű rendszerprocesszusok mindegyike a saját fejlécéből kapja a megfelelő adatokat.

Egy másik megemlítendő részlet, hogy az adatmásoló függvények nem feltétlenül konzisztensek a forrás és a célterület megadását illetően. E ciklus értelmezésekor próbáljuk meg elkerülni a félreértéseket. A szabványos C könyvtár *strncpy* függvényének első argumentuma a célterület: strncpy(to, from, count). Ez hasonló egy értékadó utasításhoz, amelyben a bal oldalon áll a változó, amelynek értéket akarunk adni, a jobb oldalon pedig a kifejezés, amely az értéket szolgáltatja. Ezt a függvényt a 7179. sorban arra használjuk, hogy a processzus nevét a processzustábla-bejegyzésbe másoljuk, nyomkövetési és egyéb célokra. Ezzel ellentétben a *phys_copy* függvény fordítva használja az argumentumokat: phys_copy(from, to, quantity). A *phys_copy* a 7202. sorban felhasználói processzusok programjainak fejléceit másolja.

Folytatva a processzustábla inicializációjának tárgyalását, a 7220. és a 7221. sorban az utasításmutató és a processzor állapotszó kezdeti értéke kerül beállításra. A taszkok állapotszava más, mint az eszközmeghajtóké és a szervereké, mert a taszkok magasabb jogosultsági szinten futnak, hogy az I/O-kapukat elérhessék. Ezt követően a veremmutató inicializálása következik, ha felhasználói szintű processzusról van szó.

A processzustáblának van egy eleme, amelyet nem kell (és nem is lehet) beütemezni. Ez a *HARDWARE* processzus, amely csak nyilvántartási célból létezik – a megszakítások kiszolgálása alatt eltelt időt neki számítja fel a rendszer. Az összes többi processzus a megfelelő ütemezési sorba kerül a 7234. és 7235. sorban. A *lock_enqueue* függvény letiltja a megszakításokat a sorok módosítása előtt, majd újra engedélyezi őket a sorok módosítása után. Erre nincs még szükség most, amikor semmi sem fut, de egyébként ez a szabályos eljárás, és nincs értelme külön kódot írni csak azért, hogy egyszer lefusson.

A processzustábla bejegyzéseinek inicializációjában az utolsó lépés az *alloc_segments* hívása a 7241. sorban. Ez egy gépfüggő rutin, amely a megfelelő mezőkbe tölti a processzusok által használt memóriaszegmensek helyét, méretét és hozzáférési jogait. A régebbi, védett üzemmódot nem támogató Intel processzorok esetén csak a szegmensek helyét definiálja. Más memóriakezelési módot használó processzortípus esetén újra kell írni.

Amikor a taszkok, a szerverek és az *init* processzustábla bejegyzései fel vannak töltve, a rendszer már majdnem működőképes. A *bill_ptr* pointer mutatja, hogy melyik processzusnak számlázzuk a processzoridőt; szükségünk van egy kezdeti értékre, ehhez az *IDLE* megfelelő választásnak tűnik (7250. sor). A kernel most már készen áll, hogy elkezdje azt, ami a feladata, a processzusok vezérlését és ütemezését, ahogy az a 2.2. ábrán látható.

A rendszer még nem minden része áll készen a szabályos működésre, de mindezek a részek független processzusként futnak, és futásra kész állapotban várakoznak. Inicializálni fogják magukat, amikor elindulnak. Csak annyi van hátra, hogy a kernel az *announce* meghívásával bejelentse, készen áll, majd meghívja a *restart*-ot (7251–7252. sor). Sok C programban a *main* egy ciklus, de a MINIX 3-kernelben csak az inicializációt kell elvégeznie. A 7252. sorban a *restart* hívása elindítja az első várakozó processzust. A *main* nem fogja visszakapni a vezérlést ezután.

A *_restart* az *mpx386.s* egy assembly nyelvű rutinja. Tulajdonképpen a *_restart* nem egy teljes függvény, csak egy nagyobb eljárás egyik közbenső belépési pontja. A következő részben részletesen fogjuk tárgyalni; most elég annyi, hogy a *_restart* egy processzusátkapcsolást fog kiváltani, így a *proc_ptr* által kijelölt processzus fog futni. Amikor a *_restart* először végrehajtódott, azt mondhatjuk, hogy a MINIX 3 már fut – egy processzust hajt végre. A *_restart* újra és újra végrehajtódik, ahogy a taszkok, szerverek és felhasználói processzusok lehetőséget kapnak a futásra, majd felfüggesztődnek üzenetre várakozás miatt, vagy mert át kell adniuk a processzort más processzusoknak.

A *_restart* első futásakor az inicializáció természetesen csak a kernel számára fejeződött be. Emlékezzünk vissza, hogy a MINIX 3-processzustábla három részből áll. Vajon hogyan futhat akár egyetlen processzus is addig, amíg a processzustábla fontos részei még nem is lettek feltöltve? A teljes választ későbbi fejezetekben fogjuk megkapni. A rövid válasz úgy hangzik, hogy a betöltési memóriakép processzusainak utasításmutatója kezdetben inicializációs programkódra mutat, és mindegyik elég hamar blokkolódik. Végül a processzuskezelő és a fájlrendszer is lehetőséget kap az inicializációs kód lefuttatására, amikor is a processzustábla náluk lévő része is beállításra kerül. Végül az *init* minden terminálhoz létrehoz egy

getty processzust. Ezek a processzusok blokkolódnak, amíg valamelyik terminálról input nem érkezik, amikor is az első felhasználó bejelentkezhet.

Végigkövettük a MINIX 3 elindulását három állományon keresztül, ebből kettő C-ben, egy pedig assembly nyelven volt írva. Az *mpx386.s* assembly nyelvű fájl további, megszakításokat kezelő programkódot is tartalmaz, ezt a következő szakaszban vizsgáljuk meg. Mielőtt azonban továbbmennénk, lássuk a két C fájl eddig nem említett rutinjainak rövid leírását. A *start.c* nem tárgyalt függvénye a *get_value* (6997. sor). Ez bejegyzéseket tud megkeresni a kernelkörnyezetben, amely az indítóparaméterek másolata. A függvény a szabványos könyvtári függvény egyszerűsített változata, amely azért szerepel itt, hogy a kernel egyszerűbb lehessen.

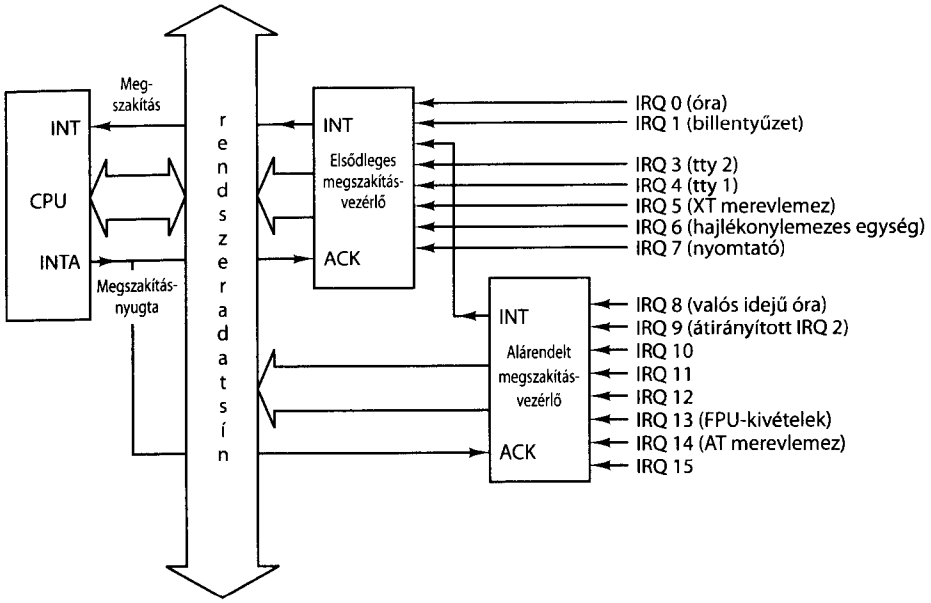
Van három további eljárás a *main.c*-ben. Az *announce* egy copyright-üzenetet jelenít meg, és közli azt is, hogy a MINIX 3 valós módban, avagy 16 bites vagy 32 bites védett üzemmódban fut-e, valahogy így:

```
MINIX 3.1 Copyright 2006, Vrije Universiteit, Amsterdam, The Netherlands
Executing in 32-bit protected mode
```

Amikor ez az üzenet megjelenik, akkor tudhatjuk, hogy a kernel inicializációja befejeződött. A *prepare_shutdown* (7273. sor) *SIGKSTOP* szignált küld az összes rendszerprocesszusnak (a rendszerprocesszusoknak nem lehet úgy szignált küldeni, ahogy a felhasználói processzusoknak). Ezután elindít egy időzítőt, hogy a rendszerprocesszusoknak legyen idejük leállni, mielőtt a végső *shutdown* eljárást meghívja. A *shutdown* alapesetben a MINIX 3 betöltési felügyelőprogramnak adja vissza a vezérlést. Ehhez a megszakításvezérlőket vissza kell állítani a BIOS szerinti helyzetbe az *intr_init(0)* meghívásával (7339. sor).

2.6.8. Megszakításkezelés a MINIX 3-ban

A megszakítások kezelésére szolgáló hardvereszközök nyilván gépfüggők, de minden rendszerben kell lennie olyan komponenseknek, amelyek az itt tárgyalt 32 bites Intel CPU-val felszerelt rendszerek elemeivel funkcionálisan megegyeznek. A hardvereszközök által generált megszakítások elektromos jelek, amelyeket első lépésben egy megszakításvezérlő kezel. Ez egy integrált áramkör, amely képes ilyen jeleket érzékelni és a processzor adatsínén az azoknak megfelelő adatmintát létrehozni. Erre azért van szükség, mert a processzornak csak egy bemenő vonala van az összes eszköz érzékelésére, így nem tudja megállapítani, hogy melyik eszközt kell kiszolgálnia. A 32 bites Intel processzorral felszerelt PC-k általában két ilyen vezérlő áramkörrel vannak ellátva. Mindkettő 8 bemenő jelet tud kezelni, de az egyik alá van rendelve a másiknak, azaz kimenő vonala a másik egy kiválasztott bemenő vonalára csatlakozik. Ezzel a kombinációval 15 különböző külső eszközt lehet érzékelni, ahogy a 2.39. ábrán látható. A 15 bemenet között vannak előre lefoglaltak. Az időzítőbemenet (IRQ 0) például nincs kivezetve csatlakozóhoz, ezért semmi mást nem rendelhetünk hozzá. Más bemenetekhez tartozik csatlakozó; ezeket bármilyen eszközhöz használhatjuk, amit bedugunk a csatlakozóba.



2.39. ábra. *32 bites Intel PC megszakításkezelő hardvere*

Az ábrán a megszakításjelek a jobb oldalon látható *IRQ n* vonalakon érkeznek. A CPU az INT vonalán kap értesítést a megszakítás bekövetkeztéről. A CPU-tól az INTA (interrupt acknowledge – megszakítás nyugtázása) vonalon érkező jel hatására a megszakításért felelős megszakításvezérlő olyan adatot helyez az adatsínre, amely a CPU számára azonosítja a végrehajtandó kezelőrutint. A megszakításvezérlőket az inicializálás során programozzuk fel, amikor a *main* meghívja az *intr_init* eljárást. Ez meghatározza, hogy a különböző megszakítások esetén a CPU milyen adatot kap, ezenkívül a vezérlő működését befolyásoló számos egyéb paramétert is beállít. Az adatsínre helyezett adat egy 8 bites szám, amelyet egy legfeljebb 256 elemű táblázat indexelésére használunk. A MINIX 3-táblázatnak 56 eleme van. Ezek közül 35-öt használunk ténylegesen; a többi a jövőben kifejlesztett Intel processzorokhoz és a MINIX 3 további fejlesztéséhez van fenntartva. A 32 bites Intel processzorokon ez a táblázat megszakítási kapuleírókat tartalmaz; ezek mindegyike egy 8 bájt hosszú, több mezőből álló struktúra.

Sokféle módon lehet reagálni a megszakításokra; a MINIX 3 esetében a bennünket leginkább érdeklő kapuleíró mezők a végrehajtandó kiszolgálórutin kódszegmensére, azon belül pedig a kezdőcímre mutatnak. A CPU a kiválasztott leíró által meghatározott rutint hajtja végre. Az eredmény ugyanaz, mint egy

```
int   <nnn>
```


assembly nyelvű utasítás végrehajtása. Az egyedüli különbség az, hogy hardvermegszakítás esetén az <nnn> a megszakításvezérlő áramkör egyik regiszteréből származik, és nem a programmemória egy utasításából.

A 32 bites Intel processzor bonyolult processzusváltási mechanizmust használ a megszakításokra adott válasz során, az utasításmutató beállítása egy másik függvény végrehajtásához csak egy része ennek. Amikor a CPU egy processzus futása alatt megszakítást érzékel, akkor létrehoz egy vermet, amelyet a megszakítás kiszolgálása közben használ. Ennek a veremnek a helyét a **folyamatállapot-szegmens (Task State Segment, TSS)** egyik bejegyzése határozza meg. Egyetlen ilyen struktúra van az egész rendszerben, ezt a *cstart* inicializálja a *prot_init* hívásakor, ami aztán minden processzus indításakor módosításra kerül. Ennek az a hatása, hogy a megszakítás esetén létrehozott verem mindig a megszakított processzus *stackframe_s* struktúrájának végén, a processzustábla-bejegyzésben jön létre. A CPU automatikusan betesz néhány kulcsfontosságú regisztert az új verembe, köztük azokat is, amelyek a megszakított processzus vermének és utasításmutatójának visszaállításához szükségesek. A megszakításkezelő programrész futása kezdetén ezt a processzustáblában elhelyezkedő területet használja veremnek, eddigre a megszakított processzushoz való visszatéréshez szükséges információ nagy része már elmentésre került. A megszakításkezelő a *stackframe_s* struktúrát kitöltve további regisztereket tesz ebbe a verembe, majd ezután átvált egy, a kernel által biztosított verem használatára, és megkezdi a megszakítás kiszolgálását.

A megszakítást kiszolgáló rutin befejeződésekor a kernelverem helyett újra egy processzustáblában lévő veremre váltunk át (nem szükségképpen ugyanarra, amely az utolsó megszakítás hatására jött létre), kiemeljük a pluszban elhelyezett regisztereket, majd végrehajtunk egy iretd (return from interrupt – visszatérés megszakításból) utasítást. Az iretd visszatér a megszakítás előtti állapothoz, visszaállítva a hardver által elmentett regisztereket és a megszakítás előtt használt vermet. Tehát egy megszakítás megállítja a futó processzust, a megszakítás kiszolgálásának befejeződése pedig újraindít egy processzust; ez esetleg különbözhet a legutoljára megállítottól. Az assembly nyelvű programozási könyvekben olvasható egyszerűbb megszakításkezelési mechanizmusoktól eltérően a megszakított processzus vermébe a megszakítás kezelése során semmit sem teszünk. Ezen túlmenően, mivel megszakítás után egy új verem jön létre egy (a TSS által meghatározott) ismert helyen, több processzus vezérlése lecgyszerűsödik. Egy másik processzus elindításához csak a veremmutatót kell a *stackframe_s* struktúrájára állítani, kiemelni az odatett regisztereket, és végrehajtani egy iretd utasítást.

A CPU megszakítás érzékelése esetén letiltja a megszakításokat. Ez biztosítja, hogy semmi nem történhet, ami a processzustáblában lévő verem túlcsordulását okozhatná. Ez automatikus, de vannak megszakítást letiltó és engedélyező assembly utasítások is. A megszakítások letiltva maradnak, míg a processzustáblán kívül elhelyezkedő kernelverem van használatban. Van egy olyan mechanizmus, amely lehetővé teszi, hogy kivételkezelő (a CPU által észlelt hibára adott válasz) fusson, amikor a kernelverem van használatban. A kivételek hasonlók a megszakításokhoz, de a kivételeket nem lehet letiltani. Így a kivételek miatt szükség van arra, hogy kezeljük ezt a helyzetet, amikor lényegében egymásba ágyazott meg-

szakításaink vannak. Ebben az esetben nem jön létre új verem. Ehelyett a CPU a megszakított program újraélesztéséhez szükséges létfontosságú regisztereket a már használatban lévő veremre helyezi. A kernel futása közben nem szabad kivételnek előfordulnia, ha mégis megtörténik, akkor „pánik" tör ki, azaz a kernel a *panic* eljárással kilép.

Amikor a kernel futása közben iredt utasításra kerül a sor, akkor egyszerűbb visszatérési mechanizmust alkalmazunk, mint felhasználói processzus megszakításakor. A processzor meg tudja állapítani, hogyan kezelje az iretd utasítást; ezt úgy éri el, hogy megvizsgálja az iretd végrehajtása során a veremből kikerülő kódszegmensszelektort.

A korábban említett jogosultsági szintek határozzák meg a megszakítások által kiváltott különböző reakciókat, amikor egy processzus fut, és amikor a kernel (ideértve a megszakításkezelő rutinokat is) fut. Az egyszerűbb mechanizmust használjuk, ha a megszakított program jogosultsága megegyezik a megszakítás hatására végrehajtott program jogosultságával. A gyakoribb eset azonban az, hogy a megszakított program alacsonyabb jogosultságú, mint a megszakítást kezelő program, ekkor a TSS-t és az új vermet alkalmazó bonyolultabb változatot kell alkalmaznunk. Egy kódszegmens jogosultsági szintje a kódszegmens szelektorában van tárolva, és mivel megszakítás esetén ez is a veremre kerül, a megszakítás kiszolgálása után megvizsgálhatjuk, hogy eldöntsük, mit kell tennie az iretd utasításnak.

A megszakítások kiszolgálása alatt használt verem létrehozásakor a hardver még egy szolgáltatást nyújt. A hardver megvizsgálja, hogy a verem elég nagy-e ahhoz, hogy legalább a minimálisan elhelyezésre kerülő adatok elférjenek benne. Ez megóvja a magasabb jogosultságú kernelt attól, hogy egy felhasználói processzus véletlenül (vagy rosszindulatúan) hibát okozzon azáltal, hogy egy rendszerhívást nem megfelelő veremmel kezdeményez. Ezeket a mechanizmusokat kifejezetten a több processzust egyszerre futtató operációs rendszerek megvalósításának elősegítésére építették be a processzorba.

Ez a viselkedés zavarba ejtő lehet, ha az olvasó nem ismeri a 32 bites Intel CPU-k belső működését. Rendesen nem mennénk ennyire a részletekbe, de egy megszakítás beérkezése és az iretd utasítás végrehajtásakor történő események megértése létfontosságú annak megértéséhez, hogy a kernel hogyan vezérli a 2.2. ábra „futó" állapotának átmeneteit. Nagyon megkönnyíti a programozó dolgát, hogy a hardver elvégzi a munka nagy részét, ezenkívül vélhetően az elkészült rendszer is hatékonyabb lesz. Mindez a hardversegítség azonban megnehezíti a program működésének megértését, ha csak a programszöveget olvassuk.

A megszakításkezelő rendszer leírása után most visszatérünk az *mpx386.s*-hez, és a MINIX 3-kernelnek azt a kicsiny részét nézzük meg, amelyik ténylegesen találkozik a hardvermegszakításokkal. Minden megszakításhoz tartozik egy belépési pont. A *_hwint00* és *_hwint07* közötti belépési pontoknál (6531–6560. sor) a *hwint_master* (6515. sor) hívása található, míg a *_hwint08* és *_hwint15* közötti belépési pontoknál a *hwint_slave* (6566. sor) hívása látható. Mindegyiknél van egy paraméter, amely a kiszolgálásra váró eszközt azonosítja. Ezek csak látszólag paraméteres eljáráshívások, valójában makróhívásokról van szó, így a *hwint_master* és a *hwint_slave* is nyolcszor kerül kifejtésre, mindig csak az *irq* paraméter változik. Ez pazarlásnak tűnhet, de a keletkező programkód nagyon tömör. A kifejtett makrókból keletkező tárgykód 40 bájtnál is kevesebb helyet foglal. A megszakítások kiszolgálásánál a sebesség nagyon fontos, és a fenti módszerrel kiküszöböljük a paraméter betöltéséből, a szubrutin meghívásából és a visszatérési érték átvételéből eredő időveszteséget.

A *hwint_master* tárgyalását úgy folytatjuk, mintha függvény lenne, és nem egy nyolc helyen kifejtett makró. Emlékezzünk arra, hogy mielőtt a *hwint_master* elkezd futni, a CPU létrehozott egy új vermet a megszakított processzus *stackframe_s* struktúrájában. Elmentett benne számos kulcsfontosságú regisztert, és a megszakítások le vannak tiltva. A *hwint_master* első dolga a *save* meghívása (6516. sor). Ez a szubrutin a megszakított processzus újraindításához szükséges összes többi regisztert elmenti. A sebesség növelése érdekében a *save* lehetne a makró része is, de ez körülbelül megkétszerezte volna a makró méretét, ezenkívül más függvény is hívja. Látni fogjuk, hogy a *save* trükközik a veremmel. Visszatérése után már a kernelverem van használatban, nem a processzustábla-bejegyzésben lévő.

A *glo.h*-ból két táblát használunk. Az *_irq_handlers* tartalmazza az átirányításhoz szükséges információt, beleértve a kezelő rutinok címeit is. A kiszolgálás alatt lévő megszakítás száma egy *_irq_handlers*-en belüli címmé alakul át. Ez a cím aztán a veremre kerül, hogy az *_intr_handle* paraméterként megkaphassa, majd az *_intr_handle* meg is hívódik. Az *_intr_handle* működését később vizsgáljuk meg. Egyelőre csak annyit mondunk, hogy nemcsak a kért megszakítási kiszolgálórutint hívja meg, hanem az *_irq_actids* tömbben is beállítja a kiszolgálási kísérlet sikerességét jelző bitet, illetve a várakozósor többi elemének is lehetőséget ad arra, hogy fussanak és lekerüljenek a sorról. Attól függően, hogy pontosan mi volt a kérés a kezelő felé, az *_intr_handle* visszatérése után a megszakítás kiszolgálása még nem feltétlenül fejeződött be. Ezt az *_irq_actids* megfelelő bejegyzésének megvizsgálásával lehet eldönteni.

Nemnulla érték az *_irq_actids*-ben azt jelzi, hogy az aktuális megszakítás kiszolgálása még nem fejeződött be. Ha ez a helyzet, akkor megtörténik a megszakításvezérlő beállítása úgy, hogy ugyanarról a vonalról ne fogadjon el újabb megszakítást (6522–6524. sor). Ez a művelet a megszakításvezérlő egy meghatározott bemenő vonalát tiltja le. A CPU megszakításbemenete már a megszakításjel beérkezésekor letiltódott, és még nem került újra engedélyezésre.

Az assembly nyelvet nem ismerők számára hasznos lehet néhány szót szólnunk az assembly nyelvű kódról. A 6521. sorban látható

```
jz 0f
```

utasítás nem az átugrandó bájtok számát adja meg. A 0f nem egy hexadecimális szám, és nem is egy normál címke. A közönséges címkék nem kezdődhetnek numerikus karakterrel. A MINIX 3-assembler ezt a jelölést használja **lokális címke** megadására; a 0f **előre (forward)** ugrást jelent a következő 0 címkéhez a 6525. sorban. A 6525. sorban kiírt bájt hatására a megszakításvezérlő visszatérhet normál üzemmódba, az aktuális megszakításvonal azonban esetleg letiltva maradhat.

Egy másik érdekes és esetleg zavaró tény, hogy a 6525. sor 0: címkéje újra előfordul ugyanabban az állományban, a *hwinit_slave*-ben a 6576. sorban. A helyzet még bonyolultabb, mint első ránézésre gondolnánk, mert ezek a címkék makrók belsejében vannak, a makrók pedig kifejtődnek, mielőtt az assembler látná a programot. Így tulajdonképpen 16 darab 0: címke van az assembler által fordított programban. Valójában a makrókban definiált címkék elburjánzásra való hajlama az oka annak, hogy az assembly nyelv lokális címkéket is megenged; lokális címke feloldásakor az assembler a megadott irányban legközelebb lévőt választja, a többi előfordulást figyelmen kívül hagyja.

Az *_intr_handle* hardverfüggő, kódjának részleteit akkor tárgyaljuk, amikor az *i8259.c* állományhoz érünk. Néhány szót azonban érdemes ejtenünk a működéséről. Az *_intr_handle* olyan struktúrák láncolt listáján halad végig, amelyek egyebek között az egyes eszközök által generált megszakításokat kezelő függvények címét és az eszközkezelők processzusszámát tartalmazzák. Azért láncolt lista, mert egy megszakításvonalon több eszköz is osztozhat. Az egyes eszközökhöz tartozó kezelőknek ellenőrizniük kell, hogy az ő eszközüket kell-e kiszolgálni éppen. Természetesen erre nem mindig van szükség, például az időzítőmegszakítások (IRQ 0) esetében sem, mert ez a vonal be van drótozva az időzítőjeleket előállító lapkához, itt más eszköz nem küldhet megszakításjelet.

A megszakításkezelők kódját úgy illik megírni, hogy hamar vissza tudjanak térni. Ha nincs semmilyen elvégzendő feladat, vagy ha a kiszolgálás azonnal befejeződött, akkor a visszatérési érték *TRUE*. A kezelő végrehajthat bizonyos műveleteket, például beolvashat az input eszközről egy adatbájtot, és azt elhelyezheti egy olyan pufferben, amelyet az eszközmeghajtó legközelebbi futásakor elér. A kezelő ezután kezdeményezheti üzenetek küldését az eszközmeghajtónak, amely ennek hatására majd közönséges processzusként futásra ütemeződik. Ha a kiszolgálás nem fejeződött be, akkor a kezelő a *FALSE* értékkel tér vissza. Az *_irq_actids* tömb minden eleme egy olyan bittérkép, amely a listán lévő kezelők visszatérési értékeit úgy összegzi, hogy az eredmény pontosan akkor lesz nulla, ha minden kezelő *TRUE* értékkel tért vissza. Ha nem ez a helyzet, akkor a 6522. és 6524. sor közötti programrész letiltja a vonalat, mielőtt magát a megszakításkezelőt a 6526. sor utasítása újra engedélyezi.

Ez a módszer biztosítja, hogy egy megszakításvonalhoz tartozó láncon lévő kezelők egyike sem aktiválódhat újra, amíg a megfelelő eszközmeghajtók el nem végezték a feladatukat. Világos, hogy szükség van a megszakítások más módon való újra engedélyezésére. Ezt a később tárgyalt *enable_irq* teszi lehetővé. Egyelőre elég annyi, hogy minden eszközmeghajtónak biztosítania kell, hogy feladata végeztével az *enable_irq* meghívódik. Az is egyértelmű, hogy az *enable_irq*-nak először az eszközmeghajtó bitjét törölnie kell az *_irq_actids* adott megszakításvonalhoz tartozó elemében, majd ellenőriznie kell, hogy minden bit nulla-e már. Csak ekkor szabad a megszakításvezérlő adott vonalát engedélyezni.

Az előbb leírtak a legegyszerűbb formában csak az időzítőmeghajtójára igazak, mert ez az egyetlen megszakításvezérelt eszköz, amelyik a kernelbe van fordítva. Egy másik folyamat megszakításkezelőjének címe nem értelmezhető a kernelen belül, és a kernel *enable_irq* függvényét nem hívhatja saját memóriaterületéről

egy processzus sem. A felhasználói szintű eszközmeghajtók részére, vagyis az időzítőmeghajtó kivételével az összes hardvermegszakításra aktiválódó eszközmeghajtó részére, a láncolt lista egy közös kezelő, a *generic_handler* címét tartalmazza. Ennek a függvénynek a forráskódja a rendszertaszk állományaiban van, de mivel a rendszertaszk a kernellel együtt fordítódik, és mivel ez a programrész megszakítás bekövetkezésekor hajtódik végre, nem igazán tekinthető a rendszertaszk részének. A láncolt lista elemeiben tárolt egyéb információk között van a megfelelő eszközmeghajtó processzusszáma. A *generic_handler* hívásakor üzenetet küld ennek az eszközmeghajtónak, aminek hatására az egy meghatározott kezelő függvényét hívja meg. A rendszertaszk a fent leírt eseménysorozat másik végét is támogatja. Amikor egy felhasználói szintű eszközmeghajtó befejezi a feladatát, akkor egy sys_irqctl rendszerhívást kezdeményez, amelynek hatására a rendszertaszk az eszközmeghajtó nevében meghívja az *enable_irq*-t, hogy az fogadhassa a további megszakításokat.

Visszatérve a *hwint_master*-hez, figyeljük meg, hogy egy ret utasítással fejeződik be (6527. sor). Nem azonnal szembetűnő, hogy itt valami trükk van. Ha egy processzust szakítottunk meg, akkor éppen a kernelvermet használjuk, és nem azt, amelyiket a hardver állított be a processzustáblában a *hwint_master* indulása előtt. Ebben az esetben a *save* veremmanipulációi a *_restart* címét hagyták a kernelveremben. Ez egy taszk, eszközmeghajtó, szerver vagy felhasználói processzus újbóli futtatását jelenti. Lehet, sőt igen valószínű, hogy ez nem ugyanaz a processzus lesz, amelyik a megszakítás beérkezésekor futott. Ez attól függ, hogy a megszakításkezelő rutin által létrehozott üzenet okozott-e változást az ütemezési sorokban. Hardvermegszakítás esetén szinte mindig ez fog történni. A megszakításkezelők általában üzenetet küldenek valamelyik eszközmeghajtónak, az eszközmeghajtók viszont általában magasabb prioritású ütemezési soron helyezkednek el, mint a felhasználói processzusok. Éppen ez annak a mechanizmusnak a lényege, amely a processzusok egyszerre történő végrehajtásának illúzióját létrehozza.

A teljesség kedvéért megemlítjük, hogy ha a kernel futása közben érkezik megszakítás, akkor a kernelverem van használatban, és a *save* a *restart1* címét hagyja benne. Ennek hatására a *hwint_master* végén lévő ret után a kernel folytatja azt, amit a megszakítás előtt csinált. Tehát a megszakítások egymásba lehetnek ágyazva, de az összes alacsony szintű kiszolgáló rutin befejeződése után a *_restart* fog végrehajtódni, és ezáltal a megszakított processzus helyett egy másik kaphat lehetőséget a futásra.

Egymásba ágyazott megszakítások nem fordulhatnak elő a MINIX 3-ban, mert a kernel futása közben a megszakítások le vannak tiltva. Azonban, ahogy korábban is említettük, szükség van erre a mechanizmusra is, mégpedig a kivételek kezelésénél. Amikor egy kivétel hatására az összes szükséges kernel rutin lefutott, akkor végül a *_restart*-ra kerül a vezérlés. A kernelkód végrehajtása közben egy kivétel hatása majdnem biztosan az lesz, hogy az utolsónak futott processzus helyett egy másik futhat. A kernelben keletkező kivétel hatása „pánik": a *panic* függvény megpróbálja a rendszert azonnal leállítani, hogy a lehető legkisebb kár keletkezzen.

A *hwint_slave* (6566. sor) majdnem ugyanolyan, mint a *hwint_master,* de neki mindkét megszakításvezérlőt újra kell engedélyeznie, mert mind a kettő letiltott állapotba kerül, amikor az alárendelt vezérlő megszakítást kap.

Most térjünk rá a *save* vizsgálatára (6622. sor), amelyet már többször is említettünk. Ahogy a neve is utal rá, egyik funkciója a megszakított processzus állapotának mentése a CPU által a processzustáblában előkészített verembe. A *save* a *_k_reenter* változót használja a megszakítások egymásba ágyazási mélységének megállapítására. Ha az aktuális megszakítás beérkezésekor egy processzus futott, akkor a 6635. sorban a

```
mov esp, k_stktop
```

utasítás átvált a kernelveremre, a következő utasítás pedig ráteszi a *_restart* címét. Ha olyankor érkezett megszakítás, amikor már úgyis a kernelvermet használtuk, akkor a *restart1* címét tesszük bele (6642. sor). Természetesen megszakítás nem következhet be itt, mindez a kivételek kezelése érdekében van. Akárhogy is jutunk el idáig, mostanra a belépéshez képest esetleg másik vermet használunk, és a hívó rutin visszatérési címe az éppen elmentett regiszterek alatt van eltemetve, tehát egy normál return utasítással nem tudunk visszatérni a hívóhoz. A 6638. és a 6643. sorban található a *save* két kilépési pontja; ezekben a

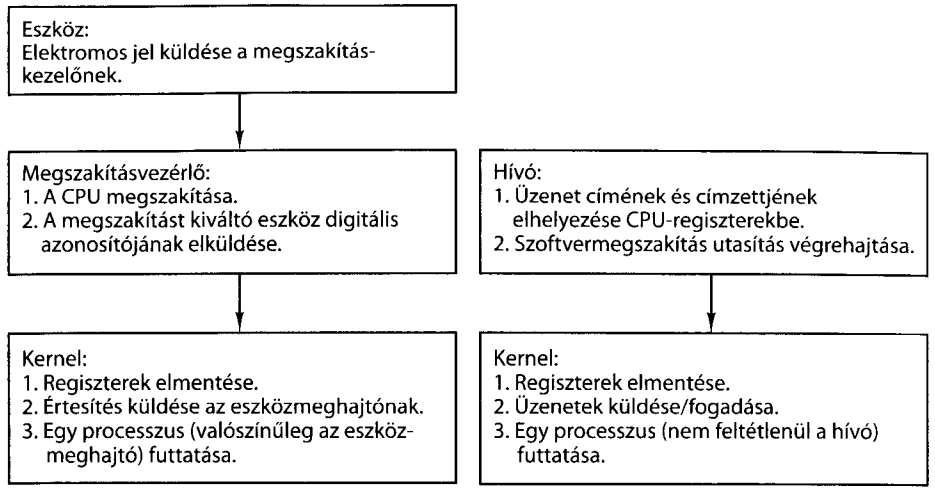
```
jmp RETADR-P_STACKBASE(eax)
```

utasítás a *save* meghívásakor elmentett címet használja.

Az újrabeléptethetőség (reentrancy) a kernelben sok problémát okoz, kiküszöbölése sok helyen egyszerűsítette a programkódot. A MINIX 3-ban a *_k_reenter* változónak van értelme, mert jóllehet közönséges megszakítások nem érkezhetnek be a kernel futása közben, de kivételek igen. Egyelőre azt kell megjegyeznünk, hogy a 6634. sorban található ugrás normál körülmények közben nem következhet be. De szükség van rá a kivételek miatt.

Mellékesen be kell vallanunk, hogy a MINIX 3 fejlesztésében az újrabeléptethetőség kiküszöbölése egy olyan eset, amikor a programozás leelőzte a dokumentálást. Bizonyos értelemben a dokumentálás nehezebb, mint a programozás – a fordítóprogram, vagy maga a program előbb-utóbb felszínre hozza a hibákat. A megjegyzések kijavítására nincs hasonló mechanizmus. Van egy elég hosszú megjegyzés az *mpx386.s* elején, ami sajnos pontatlan. A 6310. és a 6315. sor közötti résznek úgy kellene szólnia, hogy a kernelbe újrabelépés csak kivétel esetén történhet.

Az *mpx386.s* következő eljárása az *_s_call,* amely a 6649. sorban kezdődik. Mielőtt elmerülnénk a részletekbe, nézzük, hogyan fejeződik be. Nincs ret vagy jmp a végén. Valójában a *_restart* végrehajtásával folytatódik (6681. sor). Az *_s_call* a megszakításkezelő mechanizmus rendszerhívásos megfelelője. A vezérlés egy szoftvermegszakítás, vagyis egy int <nnn> utasítás végrehajtása után kerül ide. A szoftvermegszakításokat ugyanúgy kezeljük, mint a hardvermegszakításokat, természetesen ekkor a megszakításleíró tábla (Interrupt Descriptor Table) indexe



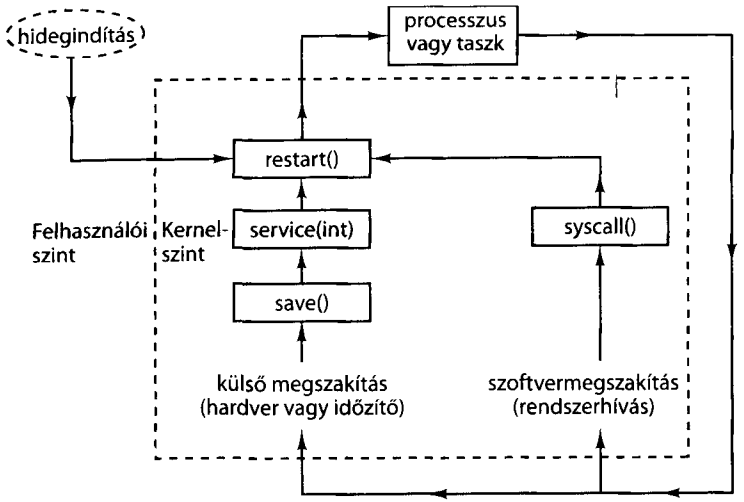
2.40. ábra. *(a) Egy hardvermegszakítás feldolgozása. (b) Egy rendszerhívás lefolyása*

az utasításban található nnn paraméter, és nem egy megszakításvezérlő áramkör szolgáltatja. Így amikor az *_s_call* elindul, akkor a CPU már átváltott egy (a folyamatállapot-szegmens által meghatározott) processzustáblában lévő veremre, és számos regisztert elhelyezett benne. Azáltal, hogy átfolyik a *_restart* elejére, az *_s_call* végül is egy iretd utasítással ér véget, így a hardvermegszakítással megegyező módon a *proc_ptr* által azonosított processzust indítja el. A 2.40. ábrán összehasonlítjuk egy hardvermegszakítást és egy szoftvermegszakítást használó rendszerhívás kezelését.

Most nézzük az *_s_call* néhány részletét. Az alternatív *_p_s_call* címke a 16 bites MINIX-verzió maradványa, amelynek külön rutinjai vannak a valós módú és a védett módú működéshez. A 32 bites verzióban mindkét címkéhez irányuló hívás ide kerül. MINIX 3-rendszerhívást használó programozó a programjába egy szokványos C függvényhívást ír, olyat, mint egy lokális függvény vagy könyvtárbeli rutin meghívásakor. A rendszerhívásban közreműködő könyvtári eljárás összeállít egy üzenetet, regiszterekbe tölti az üzenet címét és a címzett processzusazonosítóját, majd végrehajt egy int SYS386_VECTOR utasítást. Ahogy fentebb leírtuk, ennek hatására a vezérlés az *_s_call* elejére adódik át, miközben számos regiszter már a processzustáblában elhelyezkedő veremben van elmentve. A megszakítások is le vannak tiltva, ugyanúgy, mint hardvermegszakítás esetén.

Az *_s_call* első része úgy néz ki, mintha a *save* függvényt illesztettük volna be, és azokat a maradék regisztereket menti el, amelyeknek az értékét meg kell őrizni. Ahogy a *save* esetében is, a

```
mov esp, k_stktop
```



2.41. ábra. *A* restart *a rendszerindítás, a megszakítások és a rendszerhívások utáni közös pont. Az a processzus fog futni, amely a leginkább megérdemli (ez lehet különböző az utolsónak megszakított processzustól, gyakran az is). Ezen a diagramon nem látszanak a kernel futása közbeni megszakítások*

utasítás átvált a kernelveremre. (A szoftver- és hardvermegszakítások hasonlósága odáig terjed, hogy mindkettőben letiltjuk az összes megszakítást.) Ezt követően a *sys_call* hívása található (6672. sor); ezt a következő részben tárgyaljuk. Most csak annyit mondunk róla, hogy egy üzenet továbbítását váltja ki, az pedig az ütemező aktiválódását okozza. Így a *sys_call* visszatérésekor a *proc_ptr* valószínűleg nem a rendszerhívást kezdeményező processzusra fog mutatni. Ezután a vezérlés átcsorog a *restart* elejére.

Láttuk, hogy a *_restart* (6681. sor) többféle módon elérhető:

1. A rendszer indulásakor a *main* hívja.
2. Hardvermegszakítás után a *hwint_master* vagy a *hwint_slave* ide ugrik.
3. Rendszerhívás után az *_s_call* itt folytatódik.

A 2.41. ábrán egy egyszerűsített összefoglaló látható arról, hogy a *_restart*-on keresztül a vezérlés hogyan vándorol oda-vissza a processzusok és a kernel között.

A megszakításokat a *_restart* elérésekor minden esetben letiltjuk. Mire a 6690. sorhoz érünk, a következő futtatandó processzus már egyértelműen ki van választva, a letiltott megszakítások miatt már nem változhat meg. A processzustáblát körültekintően úgy terveztük, hogy a veremnek fenntartott résszel kezdődik, így az ebben a sorban található

```
mov esp, (_proc_ptr)
```


utasítás a CPU-veremmutató regiszterét erre a veremre állítja. Az

```
lldt  P_LDT_SEL(esp)
```

utasítás ezután feltölti a processzor lokális leírótáblájának (local descriptor table, LDT) regiszterét a veremből. Ez felkészíti a processzort arra, hogy a következő futtatandó processzus memóriaszegmenseit használja. Ezt követően egy utasítás beírja a következő megszakításkor használt verem címét a soron következő processzus processzustábla-bejegyzésébe, majd a következő utasítás ezt a címet a TSS-be írja.

A *restart* első részére nem lenne szükség, ha a kernel (beleértve a megszakítást kiszolgáló rutint is) futása közben érkezne megszakítás, mert ekkor a kernelverem van használatban, és a megszakításkezelő lefutása után a kernel folytatódhatna. De a MINIX 3-kernel valójában nem újrabeléptethető, és közönséges megszakítások nem érkezhetnek ilyen módon. A megszakítások letiltása azonban nem akadályozza meg a processzort abban, hogy kivételeket észleljen. A *restart1* címke (6694. sor) jelzi a helyet, ahol a végrehajtás folytatódik, ha kernelkód végrehajtása közben kivétel történik (reméljük azonban, hogy ilyesmi soha nem következik be). Ennél a pontnál a *k_reenter* változót csökkentjük annak jelzésére, hogy az esetleg egymásba ágyazott megszakítások egy szintjével végeztünk, a többi utasítás pedig visszaállítja a processzort abba az állapotba, amelyben a futtatandó processzus utolsó futásakor volt. Az utolsó előtti utasítás módosítja a veremmutatót, így a *save* hívásakor elmentett visszatérési címet figyelmen kívül hagyjuk. Ha az utolsó megszakítás egy processzus végrehajtása közben történt, akkor az utolsó iretd utasítás újraindítja a futásra most kiválasztott processzust, visszaállítva a maradék regisztereit a veremszegmenssel és a veremmutatóval együtt. Ha viszont az iretd-hez a *restart1* címkén keresztül jutottunk, akkor éppen nem egy processzusvermet, hanem a kernelvermet használjuk, és nem is egy processzushoz térünk vissza, hanem a megszakított kernel végrehajtása folytatódik. A CPU akkor érzékeli ezt, amikor az iretd végrehajtásakor a kódszegmensleírót kiveszi a veremből. Ekkor az utasítás hatása csak annyi, hogy a kernelverem marad használatban.

Ideje egy kicsit többet mondanunk a kivételekről. A **kivételek**et **(exception)** a CPU működése során fellépő különböző hibák okozzák. A kivételek nem feltétlenül rosszak. Használni lehet őket arra, hogy az operációs rendszert különféle szolgáltatások nyújtására ösztönözzük, például arra, hogy egy processzusnak több memóriát adjon, vagy behozzon egy pillanatnyilag lemezen lévő memórialapot, bár ezek a funkciók nincsenek benne a MINIX 3-alaprendszerben. Kivételeket programozási hibák is okozhatnak. A kernelben egy kivétel nagyon súlyos, és „pánik"-ra ad okot. Ha felhasználói programban keletkezik kivétel, akkor lehet, hogy azonnal le kell állítani, de az operációs rendszernek ettől még nem szabadna leállnia. A kivételeket a megszakításleíró tábla segítségével ugyanaz a mechanizmus kezeli, mint a megszakításokat. A tábla kivételekhez tartozó bejegyzései a 16 kivételkezelő belépési pontjaira mutatnak, az első a *_divide_error;* az utolsó a *_copr_error;* ezek az *mpx386.s* vége felé találhatók (6707–6769. sor). Ezek mind az *exception* (6774. sor) vagy az *errexception* (6785. sor) belépési pontokra ugranak

attól függően, hogy az adott körülmények között bekerül-e a verembe egy hibakód, vagy sem. Az assembly nyelvű program hasonló az eddig látottakhoz, a regiszterek a verembe kerülnek, majd a C nyelvű *_exception* (figyeljünk az aláhúzásra) rutint hívjuk az esemény kezelésére. A kivételek következményei különbözők. Némelyiket figyelmen kívül hagyjuk, mások „pánikot" okoznak, megint mások hatására esetleg szignált küldünk valamelyik processzusnak. Az *_exception* működésére később még visszatérünk.

Van még egy belépési pont, a *_level0_call* (6814. sor), amelyet úgy kezelünk, mintha megszakítás lenne. Akkor használjuk, amikor a kódnak a 0-s szintű, legmagasabb jogosultsági szinten kell futnia. Azért van az *mpx386.s* állományban a megszakítások és kivételek belépési pontjai között, mert maga is egy int <nnn> utasítás hatására hívódik meg. A kivételkezelő rutinokhoz hasonlóan meghívja a *save* eljárást, majd a vezérlés végül egy ret utasítás hatására a *_restart* belépési pontra kerül. Használatára később térünk ki, amikor olyan kódrészlethez érünk, amelynek normál körülmények között nem (néha még a kernel számára sem) elérhető jogosultságokkal kell futnia.

Végül egy adatok tárolására szolgáló területet foglalunk le az assembly fájl végén. Két adatszegmenst definiálunk itt. A

```
.sect .rom
```

deklaráció a 6822. sorban biztosítja, hogy a lefoglalt terület a kernel adatszegmensének legelejére kerül, valamint azt, hogy ez a rész csak olvasható terület lesz. A fordítóprogram egy mágikus számot helyez el itt; ezt megvizsgálva a *boot* el tudja dönteni, hogy valóban egy kernelt próbál-e betölteni. Teljes rendszer fordításakor különböző karakterlánckonstansok kerülnek ide. A másik adatterület a

```
.sect .bss
```

deklarációnál (6825. sor) a kernel inicializálatlan változói között helyet biztosít a kernelveremnek, afölött pedig a kivételkezelők változóinak. A szerverek és a közönséges proccsszusok vermei számára a végrehajtható fájlok összeszerkesztésekor kerül lefoglalásra tárolóhely; ezeknek végrehajtás előtt a kernel állítja be a veremszegmens-leírót és a veremmutatót. Ugyanezt a kernelnek saját maga számára is el kell végeznie.

2.6.9. Processzusok közötti kommunikáció a MINIX 3-ban

A MINIX 3-processzusok a randevú elv alapján üzenetekkel kommunikálnak egymással. Amikor egy processzus egy send műveletet hajt végre, akkor a kernel legalsó rétege ellenőrzi, hogy a címzett vár-e üzenetet a feladótól (esetleg hajlandó-e bármelyik processzustól fogadni). Ha igen, akkor az üzenetet a feladó pufferéből a fogadó pufferébe másolja, és mindkét processzust a futtathatók közé teszi. Ha a

címzett nem várakozik a küldő üzenetére, akkor a küldő blokkolódik, és a címzettnek küldeni szándékozó processzusok várakozási sorába kerül.

Amikor egy processzus egy receive műveletet hajt végre, akkor a kernel ellenőrzi, hogy van-e processzus a neki küldeni szándékozók sorában. Ha van, akkor az üzenetet átmásolja a blokkolt küldőtől a fogadóhoz, és mindkettőt futtathatóvá teszi. Ha nincs küldésre várakozó processzus, akkor a fogadó blokkolódik, amíg üzenet nem érkezik.

A MINIX 3-ban az operációs rendszer komponensei egymástól teljesen függetlenül futnak, és a randevúeljárás nem mindig teljesen megfelelő. Éppen ezekre a helyzetekre vezették be a notify alapműveletet, amely egy minimalista üzenetet, egy értesítést küld. A küldő nem blokkolódik, ha a címzett éppen nem vár üzenetre. Az értesítés azonban nem vész el. Amikor a címzett legközelebb receive-hez ér, akkor a várakozó értesítések a közönséges üzenetek előtt kézbesítődnek. Az értesítések olyan esetekben is használhatók, amikor a közönséges üzenetek holtponthoz vezethetnének. Korábban felhívtuk a figyelmet arra, hogy el kell kerülni az olyan helyzeteket, amikor az *A* processzus blokkolódik, mert *B*-nek akar üzenetet küldeni, és a *B* processzus is blokkolódik, mert *A*-nak akar üzenetet küldeni. Ha azonban az egyik üzenet egy nem blokkoló értesítés, akkor nincs semmi probléma.

A legtöbb esetben egy értesítés a feladóján kívül nem sokat árul el a címzettnek. Néha csak ennyire van szükség, de van két speciális eset, amikor az értesítés kiegészítő információt is hordoz. Mindenesetre a címzett küldhet üzenetet az értesítés feladójának, ha több információra van szüksége.

A kommunikáció magas szintű programkódja a *proc.c* állományban található. A kernelnek az a dolga, hogy a hardver- és a szoftvermegszakításokat üzenetekké konvertálja. Az előbbit hardvereszközök generálják, az utóbbiak pedig a rendszer szolgáltatásai iránti kérelmek, azaz rendszerhívások eredményeképpen jutnak el a kernelhez. A két eset elég hasonló ahhoz, hogy egy közös függvény kezelhetné őket, de hatékonyabb megoldás, ha specializált függvényeket hozunk létre.

A fájl elejéről egy megjegyzés és két makródefiníció említést érdemel. Listák kezelésénél pointerre mutató pointerek gyakran előfordulnak; ezek előnyeit és használatát taglalja a 7420. és a 7436. sor közötti megjegyzés. Két hasznos makró is definiálva van. Bár a neve általánosabbra utal, a *BuildMess* (7458–7471. sor) csak a notify számára hoz létre üzeneteket. Az egyedüli függvényhívás a *get_uptime,* amely az időzítőtaszk által kezelt változót olvas ki, hogy az értesítések időbélyegzőt is tartalmazhassanak. A *priv* függvény hívásának látszó részek egy *priv.h*-beli makrót takarnak:

```
#define priv(rp)  ((rp)->p_priv)
```

A másik makró a *CopyMess;* ez a *klib386.s*-ben lévő *cp_mess* assembly nyelvű rutin felhasználóbarát felülete.

A *BuildMess*-ről többet kellene mondanunk. A *priv* makró két speciális esetben használatos. Ha az értesítés forrása a *HARDWARE,* akkor rakománya is van, mégpedig a címzett processzus függőben lévő megszakítási kérelmeinek bittérképe. Ha a forrás a *SYSTEM,* akkor a rakomány a függőben lévő szignálok bittérképe.

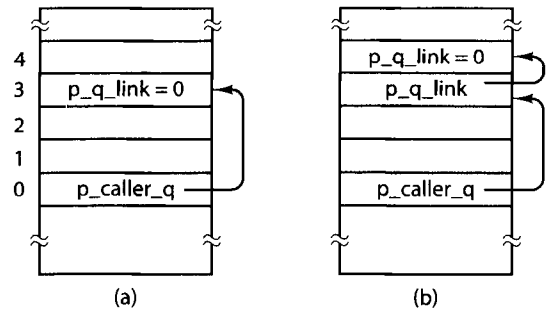
Mivel ezek a bittérképek rendelkezésre állnak a címzett processzustáblájának *priv* bejegyzésében, ezért bármikor hozzáférhetők. Az értesítések később is kézbesíthetők, ha az elküldésük pillanatában a címzett éppen nem várakozik üzenetre. Hagyományos üzenetek esetében ehhez valamilyen puffer kellene, amelyben a kézbesítetlen üzenetet eltárolhatnánk. Egy értesítés eltárolásához csak egy olyan bittérkép kell, amelyben minden olyan processzusnak van egy bitje, amelyik értesítést küldhet. Ha egy értesítés nem kézbesíthető, akkor a küldőnek megfelelő bit beállítódik a címzett bittérképében. Ha a receive meghívásakor a bittérkép ellenőrzése azt mutatja, hogy a küldő bitje be lett állítva, akkor az értesítés újragenerálható. A bit elárulja a küldő kilétét, ha pedig a küldő a *HARDWARE* vagy a *SYSTEM,* akkor a kiegészítő információk is hozzáadódnak. Ezenkívül csak az időbélyegzőre van szükség, ami szintén az újrageneráláskor adódik hozzá. Azokra a célokra, amikre felhasználják, az időbélyegzőnek nem szükséges azt az időt mutatnia, amikor az értesítést először elküldték, a kézbesítés ideje is megfelelő.

A *proc.c* első függvénye a *sys_call* (7480. sor), amely szoftvermegszakítást (ez a rendszerhívásokat kezdeményező int SYS386_VECTOR utasítás) alakít át üzenetté. A lehetséges küldők és a fogadók köre elég nagy, szükség lehet küldésre, fogadásra vagy küldésre és fogadásra is. Több mindent ellenőrizni kell. A 7490. és 7491. sorban a funkciókódot (*SEND, RECEIVE* stb.) és egyéb jelzőbiteket nyerünk ki az első argumentumból. Az első ellenőrzés arra irányul, hogy a hívó processzus jogosult-e a hívásra. A 7501. sorban felhasznált *iskerneln* egy makró, amely a *proc.h*-ban (5584. sor) van definiálva. A következő ellenőrzés során azt vizsgáljuk, hogy a megadott forrás vagy címzett érvényes processzus-e. Ezután még az üzenet mutatóját kell ellenőrizni, hogy a memória érvényes területére hivatkozik-e. A MINIX 3-ban a jogosultságok meghatározzák, hogy egy adott processzus mely másik processzusoknak küldhet üzenetet, a következő részben ez is ellenőrzésre kerül (7537–7541. sor). Végül még azt ellenőrizzük, hogy a címzett processzus még fut és nem kezdeményezett rendszerleállást (7543–7547. sor). Ha minden ellenőrzés rendben lezajlott, akkor a *mini_send, mini_receive* és a *mini_notify* közül hívjuk meg az egyiket, hogy az érdemi munkát végezze el. Ha a kért funkció *ECHO,* akkor a *CopyMess* makrót használjuk, megegyező forrással és címzettel. Ahogy azt korábban említettük, az *ECHO* csak tesztelési célokat szolgál.

A *sys_call* által ellenőrzött hibalehetőségek valószínűsége kicsi, de az ellenőrzések könnyen elvégezhetők, mert a fordítás során olyan programot kapunk, amely kis egész számok összehasonlítását végzi. Az operációs rendszer ezen legalsó szintjén tanácsos a legvalószínűtlenebb hibákat is ellenőrizni. Ez a programrész szinte biztosan sokszor végre fog hajtódni a rendszer futásának minden másodpercében.

A *mini_send, mini_receive* és *mini_notify* függvény a MINIX 3 normál üzeneteket kezelő mechanizmusának központi eleme, megérdemlik, hogy alaposan tanulmányozzuk őket.

A *mini_send* (7591. sor) három paramétert vár; ezek sorban: a hívó, a címzett processzus és a küldendő üzenetet tartalmazó puffer címe. A *sys_call* által elvégzett ellenőrzések után már csak egy szükséges, arra az esetre, ha küldők holtpontba kerülnének. A 7606. és 7610. sor között meggyőződünk arról, hogy a hívó és



2.42. ábra. *A 0-s processzusnak küldeni akaró processzusok várakozási sora*

a címzett éppen nem egymásnak próbálnak meg üzenetet küldeni.* A *mini_send* kulcsfontosságú ellenőrzése a 7615. és 7616. sorban van. Itt azt vizsgáljuk, hogy a címzett receive miatt blokkolva van-e, amit a processzustábla-bejegyzés *p_rts_flags* mezőjének *RECEIVING* bitje árul el. Ha éppen várakozik, akkor a kérdés már csak az, hogy kire. Ha a mostani küldőre vagy bárkire *(ANY),* akkor a *CopyMess* makró segítségével az üzenetet átmásoljuk, a fogadót pedig felszabadítjuk a blokkolás alól a *RECEIVING* bitjének törlésével. Az *enqueue*-t hívjuk, hogy a fogadó újból lehetőséget kapjon a futásra (7620. sor).

Ha azonban a címzett nincs blokkolva, vagy blokkolva van, de valaki mástól vár üzenetet, akkor a 7623. és 7632. sor közötti programrész hajtódik végre, ami blokkolja és a várakozási sorba teszi a küldőt. Egy adott címzettnek küldeni akaró processzusok egy láncolt listába vannak fűzve, a címzett *p_callerq* mezője a lista legelején álló processzushoz tartozó processzustábla-bejegyzésre mutat. A 2.42.(a) ábra mutatja, hogy mi történik, ha a 0-s processzus nem tudja fogadni a 3-as processzus üzenetét. Ha később a 0-s a 4-es üzenetét sem tudja fogadni, akkor a 2.42.(b) ábrán látható helyzet áll elő.

A *mini_receive* (7642. sor) eljárást a *sys_call* hívja, ha a *function* paramétere *RECEIVE* vagy *BOTH.* Korábban már említettük, hogy az értesítéseknek magasabb prioritásuk van, mint a közönséges üzeneteknek. Egy értesítés azonban soha nem megfelelő válasz egy send-re, ezért a kézbesítetlen értesítések bittérképe csak akkor kerül megvizsgálásra, ha a *SENDREC_BUSY* bit nincs beállítva. Ha talál értesítést a rendszer, akkor megjelöli, hogy már nem kézbesítetlen, és kézbesíti (7670–7685. sor). A kézbesítés során a *proc.c* első részében definiált *BuildMess* és *CopyMess* makró is felhasználásra kerül.

Mivel a notify üzenetek tartalmaznak időbélyegzőt is, azt gondolhatnánk, hogy ezáltal hasznos információhoz juthatunk, például ha a fogadó egy ideig nem tudta meghívni a receive-et, akkor az időbélyegző elárulja neki, hogy az értesítés mennyi ideig várakozott. Az értesítés azonban kézbesítéskor generálódik (és kerül bele az időbélyegző), nem pedig az elküldéskor. Van azonban célja annak, hogy az értesí-

* Amint a programlistából kiderül, valójában ennél többről van szó: azt ellenőrizzük, hogy a küldeni szándékozó processzusok között nem alakul-e ki körben várakozás. *(A fordító megjegyzése.)*

tés csak a kézbesítéskor generálódik. Nincs szükség olyan programkódra, amely az azonnal nem kézbesíthető értesítéseket eltárolja. Csak egy emlékeztető bit beállítása szükséges, hogy egy értesítést kell generálni, ha a kézbesítés lehetővé válik. Ennél gazdaságosabb tárolás elképzelhetetlen: minden kézbesítetlen értesítéshez 1 bitre van szükség.

Az is igaz, hogy általában az aktuális időre van szükség. Például a processzuskezelő értesítésként kapja a *SIG_ALARM* üzenetet, és ha az időbélyegző nem a kézbesítéskor lenne generálva, akkor a processzuskezelőnek le kellene kérnie azt a kerneltől, hogy az időzítő várakozási sorát ellenőrizni tudja.

Jegyezzük meg, hogy egyszerre csak egy értesítés kézbesítődik, a *mini_send* a 7684. sorban visszatér a kézbesítés után. A fogadó azonban nincs blokkolva, így megteheti, hogy újabb receive-et hajtson végre közvetlenül az értesítés kézhezvétele után. Ha nincs egy értesítés sem, akkor megnézzük, hogy a hívó soraiban van-e más típusú üzenet (7690–7699. sor). Ha van, akkor a *CopyMess* makró kézbesíti, majd a küldő blokkoltságát az *enqueue* hívásával megszüntetjük a 7694. sorban. Ebben az esetben a hívó nem blokkolódik.

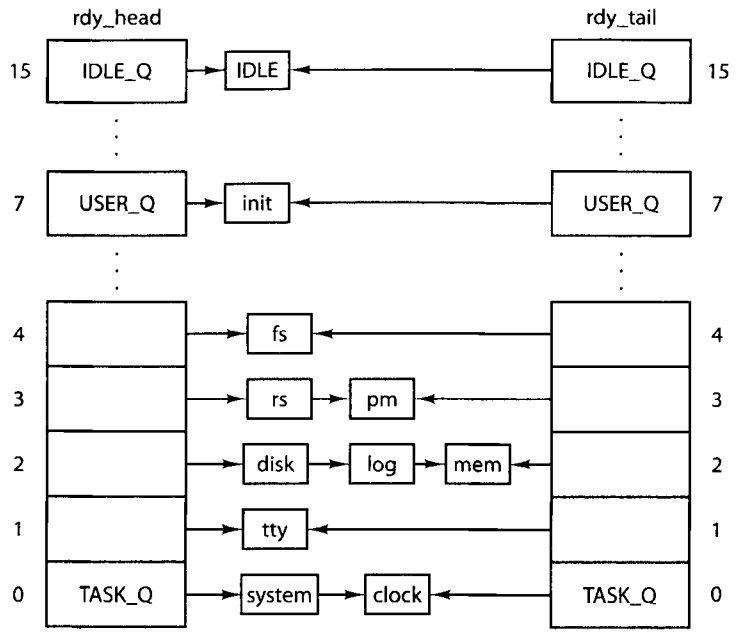
Ha sem értesítés, sem másfajta üzenet nem áll rendelkezésre, akkor a 7708. sorban a *dequeue* blokkolja a hívót.

A *mini_notify* (7719. sor) kézbesíti az értesítéseket. Hasonló a *mini_send*-hez, és röviden tárgyalható. Ha egy üzenet címzettje blokkolva van és kész a fogadásra, akkor a *BuildMess* legenerálja az értesítést, majd kézbesítődik. A címzett *RECEIVING* bitje törlődik, majd az *enqueue*-val az ütemezési sorra kerül (7738–7743. sor). Ha a címzett éppen nem várakozik, akkor az *s_notify_pending* bittérképében beállítódik egy bit, ami jelzi a kézbesítetlen értesítést, és azonosítja a küldőt. A küldő ezután folytatja munkáját. Ha az értesítés feldolgozása előtt ugyanannak a címzettnek újabb értesítést kell küldeni, akkor ugyanaz a bit játszik szerepet, mint az előbb. Tulajdonképpen az ugyanattól a feladótól származó több értesítés egyetlen értesítéssé egyesül. Ez a módszer szükségtelenné teszi a pufferelést, miközben aszinkron üzenetküldést tesz lehetővé.

Ha a *mini_notify* szoftvermegszakítás és az azt követő *sys_call* miatt hívódik meg, akkor ezen a ponton a megszakítások le lesznek tiltva. Elképzelhető azonban, hogy az időzítő-, a rendszertaszk vagy a MINIX 3-hoz egy a jövőben hozzáadandó taszk olyankor akar értesítést küldeni, amikor a megszakítások nincsenek letiltva. A *lock_notify* (7758. sor) egy biztonságos belépő a *mini_notify*-hoz. Ellenőrzi a *k_reenter* változót, hogy lássa, le vannak-e tiltva a megszakítások. Ha igen, akkor hívja a *mini_notify*-t azonnal. Ha a megszakítások engedélyezve vannak, akkor a *lock* hívásával letiltja őket, meghívja a *mini_notify*-t, majd az *unlock*-kal újra engedélyezi a megszakításokat.

2.6.10. Ütemezés a MINIX 3-ban

A MINIX 3 egy többszintű ütemezési algoritmust használ. A processzusok a 2.29. ábrán látható szerkezetnek megfelelő kezdeti prioritásokat kapnak, de most több réteg van, és a processzusok prioritása futás közben változhat is. A 2.29. ábra 1-es



2.43. ábra. *Az ütemező mind a tizenhat prioritási szinthez egy várakozási sort rendel. Itt a processzusok MINIX 3 indulása utáni elhelyezkedését láthatjuk*

rétegében elhelyezkedő időzítőtaszknak és rendszertaszknak van a legmagasabb prioritása. A 2-es réteg eszközmeghajtói alacsonyabb prioritást kapnak, de nem mindegyik ugyanakkorát. A szerverek a 3-as rétegben az eszközmeghajtóknál alacsonyabb prioritást kapnak, de megint csak van olyan, amelyik a többinél kisebbet. A felhasználói processzusok az összes rendszerprocesszusnál alacsonyabb, kezdetben ugyanakkora prioritással indulnak, de a *nice* parancs emelheti vagy csökkentheti egy felhasználói processzus prioritását.

Az ütemező a futtatható processzusokat 16 várakozási sorba osztja be, habár egy adott pillanatban nem biztos, hogy mindegyik használatban van. A 2.43. ábrán láthatók a sorok és azok a processzusok, amelyek már a helyükön vannak, amikor a kernel inicializációja befejeződik, és elkezd működni, vagyis a *main.c* 7252. sorában a *restart* hívásakor. Az *rdy_head* tömbnek minden eleme egy ütemezési sor első processzusára mutat. Hasonlóan az *rdy_tail* egy olyan tömb, amelynek elemei a sorok utolsó elemeire mutatnak. Ezek a *proc.h* 5595. és 5596. sorában vannak definiálva az *EXTERN* makró felhasználásával. A processzusok kezdeti elhelyezkedését a sorokon a *table.c*-beli *image* táblázat (6095–6019. sor) határozza meg.

Az ütemezés soronként round robin módszerrel történik. Ha egy futó processzus felhasználja a teljes időkeretét, akkor átkerül a sora végére, és egy új időszeletet kap. Ha azonban egy blokkolt processzust felélesztünk, akkor az a sora elejére kerül, amennyiben van még hátra valamennyi az időkeretéből. Nem kap azonban egy új, teljes időszeletet, csak annyi időt használhat fel, amennyi a blok-

koláskor maradt neki. Az *rdy_tail* tömb segítségével hatékonyan lehet processzusokat a sorok végére tenni. Valahányszor a futó processzus blokkolódik, vagy egy futtatható processzust meg kell szüntetni egy beérkező szignál miatt, akkor az érintett processzust az ütemező eltávolítja a sorból. Csak futtatható processzusok vannak a sorokon.

Az előbb leírt várakozási sorok felhasználásával az ütemezési algoritmus egyszerű: keressük meg a legnagyobb prioritású nem üres sort, és futtassuk a sor legelején álló processzust. Az *IDLE* processzus mindig futtatható, és a legalacsonyabb prioritású sorban van. Ha minden magasabb prioritású sor üres, akkor az *IDLE* fut.

Előzőleg láttunk már sok hivatkozást az *enqueue*-ra és a *dequeue*-ra. Most nézzük meg őket közelebbről. Az *enqueue* argumentuma egy processzustábla-bejegyzés pointere (7787. sor). Hív egy másik függvényt, a *sched*-et, olyan változókra mutató pointereket ad át neki, amelyek meghatározzák, hogy a processzusnak melyik sorra kell kerülnie, és hogy az elejére vagy a végére kell tenni. Három lehetőség van; ezek klasszikus adatszerkezetekkel kapcsolatos példák. Ha a kiválasztott sor üres, akkor hozzáadás után az *rdy_head* és az *rdy_tail* is a processzusra fog mutatni, a *p_nextready* kapcsolómező pedig a speciális *NIL_PROC* értéket fogja felvenni, ezzel jelezve, hogy nincs következő. Ha a processzus a sor elejére kerül, akkor a *p_nextready* mezőjébe átmásolódik az *rdy_head* aktuális értéke, az *rdy_head* pedig ettől kezdve az új processzusra fog mutatni. Ha a processzus a sor végére kerül, akkor a sor utolsó elemének *p_nextready* mezője az új processzusra fog irányulni, ahogy az *rdy_tail* is. Az új processzusban a *p_nextready* kapcsolómező a *NIL_PROC* értéket fogja kapni. Végül a *pick_proc* hívódik meg, amely eldönti, hogy melyik processzus fusson következőnek.

Amikor egy processzus futásra képtelenné válik, akkor a *dequeue*-t (7823. sor) kell hívni. Egy processzusnak futnia kell ahhoz, hogy blokkolódni tudjon, tehát az eltávolítandó processzus minden bizonnyal a sora elején lesz. Azonban szignált egy nem futó processzus is kaphat. Ezért a soron elindulva meg kell keresni az áldozatot, nagy valószínűséggel a legelső lesz az. Amikor megvan, akkor a szükséges pointereket megfelelően át kell állítani a sorról való eltávolításhoz. Ha éppen futott, akkor a *pick_proc*-ot is meg kell hívni.

Még egy érdekesség is van ebben a függvényben. Mivel a kernelben futó taszkoknak közös, hardver által meghatározott veremterülete van, ezeknek az integritását érdemes néha megvizsgálni. A *dequeue* elején van egy vizsgálat, ami megnézi, hogy a sorból eltávolított processzus a kernel területén működik-e. Ha igen, akkor le lehet ellenőrizni, hogy a hozzá tartozó verem végén elhelyezett megkülönböztető bitminta érintetlen-e, nem lett-e felülírva (7835–7838. sor).

A *sched*-hez értünk, amely meghatározza, hogy az újra futtathatóvá váló processzus melyik sorra, annak is melyik végére kerüljön. A processzustáblában minden processzushoz tárolva van az időszelet hossza, az időszeletéből aktuálisan megmaradt ideje, a prioritása és a megengedhető legnagyobb prioritása. A 7880. és a 7885. sor között ellenőrizzük, hogy a teljes időszeletét felhasználta-e. Ha nem, akkor folytatódhat még annyi ideig, amennyi maradt neki. Ha felhasználta az időkeretét, akkor ellenőrzésre kerül, hogy a processzus egymás után kétszer futott-e

anélkül, hogy másik processzusnak lehetősége lett volna futni. Ha ez történt, akkor úgy tekintjük, mintha végtelen, de legalábbis túlságosan hosszú ciklusban lenne, és a processzus kap +1 pont büntetést. Ha azonban úgy használta fel az időkeretét, hogy az előző futása óta más processzus is sorra került, akkor a büntetési tétel –1 lesz. Természetesen mindez nem segít, ha kettő vagy több processzus együtt fut ciklusban. Nyitott kérdés, hogy az ilyen helyzetet hogyan lehetne felismerni.

Ezután a sor kerül kiválasztásra. A 0-s sor a legmagasabb prioritású, a 15-ös a legkisebb prioritású. Lehetne érvelni amellett, hogy fordítva kellene lennie, de így konzisztens a hagyományos *nice* parancs paramétereként használt értékekkel, ahol egy pozitív érték azt jelenti, hogy a processzus alacsonyabb prioritáson fusson. A kernelprocesszusok (az időzítőtaszk és a rendszertaszk) immúnisak, de minden más processzus prioritása csökkenthető azáltal, hogy nagyobb sorszámú sorba helyezzük át büntetőpontok adásával. Alsó korlátja is van a prioritásnak, közönséges processzusok soha nem kerülhetnek az *IDLE* sorába.

Elérkeztünk a *pick_proc*-hoz (7910. sor). E függvény fő feladata a *next_ptr* beállítása. A *pick_proc*-ot meg kell hívni, ha a sorokban bármilyen változás következik be, ami befolyásolhatja, hogy melyik processzus fusson legközelebb. Amikor az aktuális processzus blokkolódik, a *pick_proc* meghívódik, hogy a CPU-t megkaphassa egy másik. Lényegében a *pick_proc* az ütemező.

A *pick_proc* egyszerű. Minden sort megvizsgál. Először a *TASK_Q* vizsgálata történik meg; ha nem üres, akkor a *proc_ptr* beállítódik a legelsőre, és a *pick_proc* azonnal visszatér. Egyébként a következő, eggyel alacsonyabb prioritású sor következik, egészen addig, amíg az *IDLE_Q*-hoz nem ér. A *bill_ptr* mutatót a kiválasztott felhasználói processzusra állítjuk be, hogy az általa elhasznált processzoridőt a számlájára tudjuk írni (7694. sor). Ez biztosítja, hogy az utolsónak futtatott felhasználói processzusnak számlázzuk a rendszer által az ő érdekében felhasznált időt.

A *proc.c* többi eljárása a *lock_send,* a *lock_enqueue* és a *lock_dequeue*. Ezek a megfelelő alapfüggvényhez nyújtanak hozzáférést, kiegészítve a *lock*-kal és az *unlock*-kal, ugyanúgy, ahogy a *lock_notify* esetében láttuk.

Összefoglalva, az ütemezési algoritmus több prioritási sort kezel. Mindig a legnagyobb prioritású sor legelső processzusát futtatjuk következőnek. Az időzítőtaszk felügyeli a processzusok által felhasznált időt. Ha egy felhasználói processzus elhasználja a teljes időszeletét, akkor a processzus a sora végére kerül, így egy egyszerű round robin ütemezést valósítunk meg a versengő felhasználói processzusok között. A taszkok, az eszközmeghajtók és a szerverek többnyire blokkolódásig futhatnak, mert hosszú időszeleteket kapnak, de ha túl sokáig futnának, akkor ezek is megszakíthatók. Ez valószínűleg nem gyakran történik meg, de így problémás magas prioritású processzusok sem tudják lefagyasztani a rendszert. Másokat a futásban akadályozó processzusok átmenetileg alacsonyabb prioritású sorba is kerülhetnek.

2.6.11. Hardverfüggő kernelkomponensek

Sok C függvény is függ a hardvertől. A MINIX 3 más rendszerekre történő átvitelének megkönnyítése érdekében ezeket a függvényeket nem az általuk támogatott magas szintű programrészekkel egy helyre, hanem az *exception.c,* az *i8259.c* és a *protect.c* állományokban helyeztük el, amelyeket ebben a szakaszban tárgyalunk.

Az *exception.c* a kivételkezelő *exception* rutint tartalmazza (8012. sor), amelyet a rutin assembly nyelvű része hív (mint *_exception*-t) az *mpx386.s* állományból. A felhasználói processzusoktól eredő kivételeket szignálokká konvertáljuk. A felhasználókról feltételezzük, hogy hibáznak programjaikban, de egy operációs rendszerből eredő kivétel valami komoly problémát jelez, és pánikot vált ki. Az *ex_data* tömb (8022–8040. sor) határozza meg pánik esetén a kiírandó üzenetet, vagy egyébként kivételenként a felhasználói processzushoz küldendő szignált. A korábbi Intel processzorok nem állították elő az összes kivételt, a bejegyzések harmadik mezője jelzi, hogy melyik az a legkorábbi modell, amely képes az adott kivétel előállítására. Ez a tömb érdekes összefoglalását nyújtja azon Intel processzorok fejlődésének, amelyekre a MINIX 3-implementációk készültek. A 8065. sorban egy alternatív üzenetet írunk ki, ha a kivételt az adott processzor elvileg nem generálhatta volna.

Hardverfüggő megszakításstámogatás

Az *i8259.c* három függvénye a rendszer inicializálása során az Intel 8259 megszakításvezérlő áramkörök beállítására szolgál. A 8119. sorban található makró egy haszontalan függvényt definiál, az igazira csak akkor van szükség, ha a MINIX 3-at 16 bites processzorra fordítjuk. Az *intr_init* (8124. sor) inicializálja a vezérlőket. Két lépésben biztosítjuk, hogy ne érkezhessen megszakítás mindaddig, amíg az inicializáció teljesen be nem fejeződött. Először az *intr_disable* hívódik meg a 8134. sorban. Ez egy C-ből hívott assembly nyelvű könyvtári függvény, amely egyetlen cli utasítást tartalmaz, ami megakadályozza, hogy a CPU reagáljon a megszakításokra. Ezután mindkét megszakításvezérlő regisztereibe egy sor bájtot írunk ki, aminek hatására a vezérlők nem veszik figyelembe a bemenő jeleket. A 8145. sorban kiírt bájt egy kivételével csupa 1-est tartalmaz, az alárendelt vezérlőtől az elsődleges vezérlőhöz futó bemeneti vonalat nem tiltjuk le (lásd 2.39. ábra). A 0 engedélyezi a bemenetet, az 1 letiltja. Az alárendelt vezérlőbe kiírt bájt csupa 1-est tartalmaz.

Az i8259-es megszakításvezérlő lapkán van egy táblázat, amely 8 bites indexeket generál; ezek alapján a CPU meg tudja határozni az egyes forrásokhoz (a 2.39. ábra jobb oldalán látható jelek) tartozó megszakítási kapuleírókat. Ezeket a BIOS inicializálja a számítógép bekapcsolásakor, és majdnem mindegyiket változtatás nélkül lehet hagyni. A megszakításokat igénylő eszközmeghajtók elindulásakor meg lehet változtatni, amit kell. Ezután minden meghajtó kérheti a megszakításvezérlőben a hozzá tartozó bit törlését, hogy a saját vonala engedélyezetté váljon. Az *intr_init* paramétere, a *mine* jelzi, hogy a MINIX 3 éppen indul, vagy leáll. Ezt a függvényt az induláskori inicializálásra és leálláskor a BIOS-értékek visszaállítására is lehet használni.

Miután a hardver beállítása befejeződött, az *intr_init* utolsó lépésként a BIOS-megszakításvektorokat átmásolja a MINIX 3-vektortáblába.

Az *i8259.c* második függvénye a *put_irq_handler* (8162. sor). Inicializáláskor a *put_irq_handler* minden olyan processzusra meghívódik, amelynek reagálnia kell valamilyen megszakításra. Ez a *glo.h*-ban *EXTERN*-ként definiált *irq_handlers* táblázatba elhelyezi a megszakításkezelő rutin címét. A modern számítógépekben 15 megszakításvonal nem mindig elég (mert 15-nél több I/O-eszköz is lehet), ezért előfordulhat, hogy két I/O-eszköz közösen használ egy vonalat. Ez a könyvben ismertetett MINIX 3 által támogatott alapvető eszközök esetében nem fordul elő, de amikor hálózati kártyákat, hangkártyákat vagy ezeknél különlegesebb I/O-kártyákat kell támogatni, akkor előfordulhat a vonalmegosztás. Ezt úgy teszszük lehetővé, hogy a megszakítástábla nem egyszerűen címeket tartalmaz. Az *irq_handlers*[*NR_IRQ_VECTORS*] egy olyan tömb, amely a *kernel/type.h*-ban definiált *irq_hook* struktúrákra mutató pointereket tartalmaz. Ezeknek a struktúráknak van egy ugyanilyen struktúra címét tartalmazó mezőjük, így az *irq_handlers* elemeiből kiinduló láncolt listák alakíthatók ki. A *put_irq_handler* egy elemet ad egy ilyen listához. A struktúrák legfontosabb tagja egy **megszakításkezelő** pointer, annak a függvénynek a címe, amelyet meg kell hívni megszakítás beérkezésekor, például ha egy I/O-művelet befejeződött.

A *put_irq_handler* néhány részletére érdemes kitérnünk. Figyeljük meg az *id* változót, amelyet 1-re állítunk, mielőtt a láncolt listát egy while ciklussal bejárjuk (8176–8180. sor). A ciklus törzsében minden lefutáskor az *id* értéke eggyel balra tolódik. A 8181. sorban található ellenőrzés a lista hosszát az *id* méretére korlátozza, vagyis 32-re egy 32 bites rendszerben. Alapesetben a lista bejárása során elérjük a végét, ahova egy új kezelőt felvehetünk. Amikor ez kész, az *id* is bekerül az új listaelem megegyező nevű mezőjébe. A *put_irq_handler* a globális *irq_use* egy bitjét is beállítja, ezzel jelzi, hogy az adott megszakításhoz létezik kezelő.

Ha az olvasó megértette a MINIX 3-nak azt a tervezési célját, hogy az eszközmeghajtókat a felhasználói területre vigyük át, akkor az előző leírás a megszakításkezelők aktivizálásáról egy kicsit zavarba ejtő lehetett. A struktúrákban tárolt megszakításkezelők címei csak akkor használhatók, ha a kernel címterületén elhelyezkedő függvényekre mutatnak. A kernel címterületén azonban csak egyetlen megszakítást használó eszközmeghajtó van, az időzítő. Mi a helyzet azokkal az eszközmeghajtókkal, amelyeknek saját címterülete van?

A válasz az, hogy azokat a rendszertaszk kezeli. Sőt a kernel- és a felhasználói processzusok közötti kommunikációra vonatkozó legtöbb kérdésre is ugyanez a válasz. Az a felhasználói területen elhelyezkedő eszközmeghajtó, amely megszakításokat akar kezelni, egy sys_irqctl rendszerhívással jelzi a rendszertaszknak, hogy megszakításkezelőt akar bejegyezni. A rendszertaszk ekkor meghívja a *put_irq_handler*-t, de az eszközmeghajtó címtartományában lévő függvény helyett a rendszertaszk területén elhelyezkedő *generic_handler* címe kerül a megszakításkezelő mezőbe. A *generic_handler* az *irq_hook* struktúrában található processzusszámot felhasználva előkeresi a meghajtó *priv* táblabejegyzését, és a meghajtó függőben lévő

megszakításait tároló bittérképben az aktuális megszakításhoz tartozó bitet beállítja. Ezután a *generic_handler* értesítést küld az eszközmeghajtónak. Az értesítés feladójaként *HARDWARE* szerepel, és a függőben lévő megszakítások bittérképe is bekerül az üzenetbe. A bittérkép miatt tulajdonképpen egyetlen értesítés az összes függőben lévő megszakításról informál. Az *irq_hook* struktúra tartalmaz még egy *policy* mezőt is, amely azt szabályozza, hogy a megszakítást azonnal újra engedélyezni kell, vagy letiltva kell maradnia. Utóbbi esetben az eszközmeghajtó dolga, hogy a megszakítás kiszolgálása után egy sys_irqenable rendszerhívással újra engedélyezze azt.

A MINIX 3 egyik tervezési célja az I/O-eszközök futási időben történő átkonfigurálásának támogatása. A következő függvény az *rm_irq_handler,* amely eltávolít egy megszakításkezelőt, amire szükség is van, ha egy eszközmeghajtót el akarunk távolítani és esetleg fel akarunk váltani egy másikkal. Ennek hatása éppen a *put_irq_handler* ellentéte.

A fájl utolsó függvénye az *intr_handle* (8221. sor), amelyet az *mpx386.s*-beli *hwint_master* és *hwint_slave* makró hív. A bittérképek *irq_actids* tömbjének a kiszolgálás alatt álló megszakításhoz tartozó elemét használja arra, hogy az egy listán lévő megszakításkezelők aktuális állapotát nyilvántartsa. Az *intr_handle* a lista minden függvényéhez beállítja a hozzá tartozó bitet az *irq_actids*-ben, majd meghívja a kezelőt. Ha a kezelőnek nem kell tennie semmit, vagy azonnal be tudja fejezni a munkát, akkor „true" értékkel tér vissza, és az *irq_actids* megfelelő bitje törlődik. A *hwint_master* és a *hwint_slave* vége felé az egész bittérkép mint egész szám ellenőrzésre kerül, hogy engedélyezni lehet-e az adott megszakítást, mielőtt a következő processzus futása elkezdődik.

Intel védett mód támogatás

A *protect.c* az Intel processzorok védett üzemmódjával kapcsolatos rutinokat tartalmaz. A **globális leírótábla (Global Descriptor Table, GDT), lokális leírótábla (Local Descriptor Table, LDT)** és a **megszakításleíró tábla (Interrupt Descriptor Table, IDT)** mindegyike a memóriában helyezkedik el, és a rendszer erőforrásainak védettségét biztosítja. A GDT és az IDT helyét egy speciális CPU-regiszter mutatja, a GDT bejegyzései LDT-kre mutatnak. A GDT minden processzusnak rendelkezésére áll, és az operációs rendszer által használt memóriaterületekhez szegmensleírókat tartalmaz. Rendes körülmények között minden processzushoz egy LDT tartozik, amely a processzushoz rendelt memóriaterületek szegmensleíróit tartalmazza. Egy leíró egy sok komponensből álló 8 bájtos struktúra, a szegmensleíró legfontosabb részei azok a mezők, amelyek egy memóriaterület báziscímét és felső korlátját határozzák meg. Az IDT is 8 bájtos leírókból áll, a legfontosabb része a megfelelő megszakítás beérkezésekor végrehajtandó programrész címe.

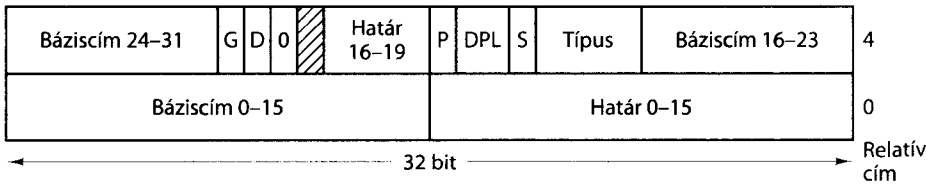
A *prot_init* (8368. sor) eljárás a GDT beállítását végzi a 8421. és 8438. sor között, a *start.c* állomány *cstart* függvénye hívja. Az IBM PC BIOS megköveteli, hogy egy előre megadott sorrendben legyenek a bejegyzések, a megfelelő indexek definícióit a *protect.h* fájl tartalmazza. Minden processzushoz a hozzá tartozó LDT a processzustáblában van elhelyezve. Ezek két leírót tartalmaznak, egy kódszegmens- és

egy adatszegmens-leírót – emlékezzünk arra, hogy most hardverszegmensekről beszélünk; ezek nem ugyanazok, mint az operációs rendszer által kezelt szegmensek. Az operációs rendszer a hardveradatszegmenseket tovább osztja adat- és veremszegmensre. A 8444. és 8450. sor között minden LDT-leírója bekerül a GDT megfelelő bejegyzésébe. Az *init_dataseg* és az *init_codeseg* függvény állítja elő ezeket a leírókat. Az egyes LDT leírók a processzusok memóriatérképének változásakor inicializálódnak (vagyis amikor egy exec rendszerhívás történik).

Egy másik, inicializálást igénylő processzor-adatszerkezet a **folyamatállapot-szegmens (Task State Segment, TSS)**. A struktúrát a fájl elején (8325–8354. sor) definiáljuk, a processzor regisztereinek és olyan egyéb információnak ad helyet, amelyeket processzusváltás esetén el kell menteni. A MINIX 3 csak azokat a mezőket használja, amelyek megszakítás esetén megadják az új verem helyét. Az *init_dataseg* hívása (8460. sor) biztosítja, hogy ezt a helyet a GDT segítségével meg lehet találni.

A MINIX 3 legalsó szintű működésének megértéséhez talán legfontosabb azt megértenünk, hogy a kivételek, hardvermegszakítások és int <nnn> utasítások hogyan vezetnek a kiszolgálásukra írt különféle programrészek végrehajtásához. Ez a **megszakítási kapuleíró (interrupt gate descriptor)** révén valósul meg. A *gate_array* (8383–8418. sor) tömböt a fordítóprogram feltölti a kivételek és a hardvermegszakítások kiszolgálórutinjainak címeivel, majd ez alapján a 8464. és 8468. sor között egy ciklus az *int_gate* függvény hívásaival beállítja a kapuleírókat.

Jó okai vannak annak, ahogy az adatokat a leírókban tároljuk. Ezek az okok a hardver részleteitől a fejlett processzorok és a 16 bites 286-os processzor közötti kompatibilitás megtartásának szükségességéig terjednek. Szerencsére ezeket a részleteket általában az Intel tervezőmérnökeire hagyhatjuk. Legnagyobb részben a C nyelv is lehetővé teszi a részletek figyelmen kívül hagyását. Egy igazi operációs rendszer megvalósítása során azonban előbb-utóbb szembekerülünk a részletekkel. A 2.44. ábrán egy bizonyosfajta szegmensleíró belső szerkezetét láthatjuk. Figyeljük meg, hogy a báziscím, amelyre a C programok egy egyszerű 32 bites egészként hivatkozhatnak, itt három részre van bontva; ezek közül kettő el van választva egymástól néhány 1, 2 és 4 bites értékkel. A szegmenshatár egy 20 bites mennyiség, amely egy 16 bites és egy 4 bites részből áll össze. A szegmenshatárt lehet értelmezni bájtok számaként vagy 4096 bájtos lapok számaként a *G* (granularity – szemcsézettség) bitértékétől függően. Más leírók, mint például a megszakítások kezelését meghatározók, szerkezete ettől eltérő, de ugyanilyen bonyolult. Ezeket a 4. fejezetben tárgyaljuk részletesebben.



2.44. ábra. *Egy Intel szegmensleíró szerkezete*

A *protect.c* többi függvényének nagy része a C változók és a leírók meglehetősen csúnya gépi ábrázolása (lásd 2.44. ábra) közötti konverziónak szentelt. Az *init_codeseg* (8477. sor) és az *init_dataseg* (8493. sor) feladata hasonló: a kapott paramétereket szegmensleírókká konvertálják. Feladatuk befejezéséhez mind a kettő hívja a soron következő *sdesc* függvényt (8508. sor). Ez az a hely, ahol a 2.44. ábrán látható rendetlen struktúra részleteivel foglalkozunk. Az *init_codeseg* és az *init_dataseg* nem csak a rendszer inicializálásakor kap szerepet. A rendszertaszk processzusok létrehozásakor is meghívja őket a megfelelő memóriaterületek lefoglalásához. A *seg2phys* (8533. sor) eljárást csak a *start.c* hívja, az *sdesc* által végrehajtott művelet fordítottját végzi el: egy szegmensleíróból kiveszi a báziscímet. A *phys2seg*-re (8556. sor) már nincs szükség, mert a sys_segctl rendszerhívás kezeli a távoli memóriaszegmensekhez történő hozzáférést, például a PC fenntartott területeihez 640 K és 1 M között. Az *int_gate* (8571. sor) funkciója hasonló az *init_codeseg* és az *init_dataseg* funkciójához: a megszakításleíró táblában tölt fel bejegyzéseket.

A *protect.c* függvénye, az *enable_iop* (8589. sor) egy elég piszkos trükköt képes elvégezni. Megváltoztatja az I/O-műveletek prioritási szintjét, így a futó processzusnak lehetősége nyílik az I/O-kapuk írását és olvasását végző utasítások végrehajtására. A függvény céljának magyarázata bonyolultabb, mint maga a függvény; ez csak beállít két bitet a hívó processzus vermében tárolt egyik szóban, amely a processzus legközelebbi futásakor a CPU állapotregiszterébe kerül. Nincs szükség az előbbi hatását megszüntető függvényre, mert az csak a hívó processzusra vonatkozik. Ezt a függvényt jelenleg nem használjuk, és felhasználói területen futó programnak nincs lehetősége rá, hogy meghívja.

A *protect.c* utolsó függvénye az *alloc_segments* (8603. sor), amelyet a *do_newmap* hív. A kernel *main* rutinja is meghívja inicializáláskor. Nagyon hardverfüggő. A processzustáblában található szegmens-hozzárendelések alapján beállítja azokat a regisztereket és leírókat, amelyeket a Pentium processzor a védett szegmensek hardverszintű támogatása során használ. A 8629. és 8633. sorban látható többszörös értékadás a C nyelv egyik jellemzője.

2.6.12. Kiegészítő eljárások és a kernelkönyvtár

Legvégül, a kernelnek van egy assembly nyelven írt segédfüggvényeket tartalmazó könyvtára, amelyet a *klib.s* lefordításával hozunk létre. Ezenkívül van néhány C-ben írt kiegészítő eljárás a *misc.c* állományban. Nézzük először az assembly fájlokat. A *klib.s* (8700. sor) egy rövid fájl, és ugyanazt a szerepet játssza, mint a korábban látott *mpx.s*: kiválasztja a megfelelő gépfüggő változatot a *WORD_SIZE* értéke alapján. Az általunk tárgyalt változat a *klib386.s* (8800. sor). Ez körülbelül két tucat olyan kisegítő rutint tartalmaz, amelyet assembly nyelven írtunk, egyrészt a hatékonyság miatt, másrészt azért, mert nem is lehet mindent C-ben megírni.

A *_monitor* (8844. sor) lehetővé teszi a betöltési felügyelőprogramhoz való visszatérést. A felügyelőprogram szempontjából a MINIX 3 csak egy szubrutin, a visszatérési címet a veremben hagyja, amikor a rendszert elindítja. A *_monitor*

visszaállítja a különböző szegmensszelektorokat és a MINIX 3 elindításakor elmentett veremmutatót, és visszatér mint egy normális szubrutin.

Az *int86* (8864. sor) BIOS-hívásokat támogat. A BIOS alternatív lemezmeghajtókat nyújt, amelyeket itt nem tárgyalunk. Az *int86* a betöltési felügyelőprogramhoz irányítja a hívást, az kezeli a védett mód és a valós mód közötti átmenetet, végrehajtja a BIOS-hívást, majd visszavált védett módba, és visszatér a 32 bites MINIX 3-ba. A betöltési felügyelőprogram a hívás során eltelt időt is visszaadja. Ennek felhasználását az időzítőtaszk tárgyalása során fogjuk látni.

Jóllehet az üzenetek másolására a *_phys_copy* (lásd alább) is alkalmas lett volna, mégis a gyorsabb, specializált *_cp_mess* eljárást (8952. sor) használjuk erre a célra. Hívása a

```
cp_mess(source, src_clicks, src_offset, dest_clicks, dest_offset);
```

formában történik, ahol a *source* a küldő processzus száma, amely a fogadó puffernek *m_source* mezőjébe másolódik. A címek, amelyek előírják, hogy az üzenetet honnan hova kell másolni, egy memóriaszelet sorszámmal (ez tipikusan az üzenetet tartalmazó szegmens eleje) és a memóriaszeleten belüli relatív címmel vannak megadva. Ez a megadási mód hatékonyabb, mint a *_phys_copy* által használt 32 bites címek.

Az *_exit*, *__exit* és *___exit* eljárások (9006–9008. sor) azért vannak definiálva, mert a MINIX 3 fordításakor esetleg olyan könyvtári rutinokat használunk, amelyek hívják az *exit* szabványos C függvényt. A kernelből való kilépésnek nincs értelme; nincs hova menni. Következésképpen itt a szabványos *exit* nem használható. A megoldás az, hogy engedélyezzük a megszakításokat, és belépünk egy végtelen ciklusba. Valamikor egy I/O-művelet vagy az időzítő megszakítást fog okozni, és a normális rendszerműködés helyreáll. A *___main* (9012. sor) belépési pont is kísérlet a fordítóprogram egy másik olyan akciójának hatástalanítására, amely felhasználói programok fordítása esetén értelmes lehet, de semmi haszna a kernelben. Egy assembly nyelvű ret (visszatérés szubrutinból) utasítás áll itt.

A *_phys_insw* (9022. sor), a *_phys_insb* (9047. sor), a *_phys_outsw* (9072. sor) és a *_phys_outsb* (9098. sor) lehetőséget adnak az I/O-kapuk elérésére; ezek az Intel-alapú gépek esetén a memóriától elkülönített címtartományban helyezkednek el, és külön műveletekkel lehet őket írni és olvasni. Az itt használt I/O-utasítások, az ins, insb, outs és outsb, úgy lettek megtervezve, hogy hatékonyak legyenek tömbök (sorozatok) használata esetén, valamint 16 bites szavak vagy 8 bites bájtok esetén is. A függvényekben a többi utasítás arra szolgál, hogy beállítsa azokat a paramétereket, amelyek szükségesek a megadott számú bájt vagy szó átmozgatására a fizikai címmel megadott puffer és egy I/O-kapu között. Ez a függvény képes a lemezek nagy sebességű kiszolgálására, ugyanezt bájtonkénti vagy szavankénti adatátvitellel nem lehetne elérni.

Egyetlen gépi utasítás elegendő a megszakítások letiltásához vagy engedélyezéséhez. Az *_enable_irq* (9126. sor) és a *_disable_irq* (9162. sor) ennél összetettebb. Utóbbiak a megszakításvezérlő áramköröket manipulálják, egyes hardvermegszakításokat engedélyeznek vagy tiltanak le.

A *_phys_copy* (9204. sor) eljárást C programból a

```
phys_copy(source_address, destination_address, bytes);
```

formában hívhatjuk, és a fizikai memória bármelyik részéből bármelyik másik részébe másol át egy adatblokkot. Mindkét cím abszolút, azaz a 0 cím valóban a címtartomány legelső bájtját jelenti, mind a három paraméter előjel nélküli hosszú egész (unsigned long).

Biztonsági okokból egy program által használni kívánt memóriaterületről törölni kell minden adatot, amelyet más programok hagytak ott. Ezt teszi a MINIX 3 exec rendszerhívása is, a *klib386.s* következő függvényét, a *phys_memset*-et (9248. sor) felhasználva.

A következő két rövid függvény speciálisan az Intel processzorokhoz készült. A *_mem_rdw* (9291. sor) függvény egy 16 bites szót ad vissza a memória bármelyik részéből. Az eredmény nullákkal kiegészítve a 32 bites *eax* regiszterbe kerül. A *_reset* függvény (9307. sor) alaphelyzetbe állítja a processzort. Ezt úgy éri el, hogy a processzor megszakításleíró tábla regiszterét a null mutatóval tölti fel, és végrehajt egy szoftvermegszakítást. Ennek ugyanaz a hatása, mint egy hardver-„reset"-nek.

Az *idle_task* (9318. sor) akkor hívódik meg, amikor nincs semmi más teendő. Végtelen cikluskénti van megírva, de nem egyszerűen egy aktív ciklus (aminek ugyanez lenne a hatása). Az *idle_task* kihasználja a hlt utasítást, amely energiatakarékos üzemmódba kapcsolja a processzort a következő megszakítás beérkezéséig. A hlt azonban egy privilegizált utasítás, 0-s szintű jogosultság hiányában végrehajtása kivételt okoz. Ezért az *idle_task* egy hlt utasítást tartalmazó szubrutin címét teszi a verembe, majd meghívja a *level0* függvényt (9322. sor). Ez a függvény kiveszi a *halt* szubrutin címét a veremből, és egy fenntartott területre másolja (a *glo.h*-ban van definiálva, és a *table.c*-ben lefoglalva).

A *_level0* a fenntartott területen elhelyezett címet egy megszakításkezelő címének tekinti, amelyet a legmagasabb, 0-s szintű jogosultsággal kell futtatni.

Az utolsó két függvény a *read_tsc* és a *read_flags*. Az első az rdtsc (read time stamp counter) utasítás végrehajtásával egy tsc nevű CPU-regiszter tartalmát olvassa ki. Ez a CPU-ciklusokat számlálja, és teljesítménymérésre, illetve nyomkövetéshez használható fel. Ezt az utasítást nem ismeri fel a MINIX 3-assembler, ezért a hexadecimális műveleti kódja segítségével illesztettük a programba. Végül a *read_flags* kiolvassa a processzus jelzőbitjeit, és C változóként adja vissza. A programozó fáradt volt, és a függvény célját leíró megjegyzés hibás.

Ebben a fejezetben utolsóként a *utility.c* nevű fájlt nézzük meg, amely három fontos függvényt tartalmaz. Amikor valami nagyon nagy baj történik a kernelben, akkor a *panic* (9429. sor) hívódik meg. Ez kiír egy üzenetet, és meghívja a *prepare_shutdown* függvényt. Amikor a kernel ki akar írni valamit, akkor nem használhatja a szabványos könyvtári *printf*-et, ezért egy speciális *kprintf* található itt (9450. sor). A könyvtári verzió összes formázó opciójára itt nincs szükség, de a funkcionalitás nagy része rendelkezésre áll. Mivel a kernel nem használhatja a fájlrendszert egy fájl vagy I/O-eszköz eléréséhez, ezért minden karaktert egy másik függvénynek, a *kputc*-nek (9525. sor) ad át, amely ezeket egy pufferbe helyezi. Később, amikor a *kputc* megkapja az *END_OF_KMESS* kódot, akkor értesíti a processzust, amely az ilyen üzeneteket kezeli. Ez az *include/minix/config.h*-ban van definiálva, és akár a naplózómeghajtó, akár a konzolmeghajtó is lehet. A naplózómeghajtó a konzolnak is átadja az üzenetet.

2.7. A MINIX 3-rendszertaszk

Annak a döntésnek, hogy nagy rendszerkomponenseket a kernelen kívül, önálló processzusként valósítunk meg, következményei vannak. Az egyik következmény, hogy ezek a komponensek nem végezhetnek konkrét I/O-műveleteket, nem manipulálhatják a kernel táblázatait, és nem tehetnek meg még egy sor olyan dolgot, amit az operációsrendszer-funkciók általában elvégeznek. Például a fork rendszerhívást a processzuskezelő intézi. Új processzus létrehozásáról a kernelnek is tudomást kell szereznie, hogy az ütemezésnél figyelembe tudja venni. Hogyan közli a processzuskezelő a kernellel?

A probléma megoldása az, hogy a kernel szolgáltatásokat kínál az eszközmeghajtóknak és a szervereknek. Ezek a szolgáltatások nem állnak a közönséges felhasználói processzusok rendelkezésére, de lehetővé teszik az eszközmeghajtók és a szerverek számára, hogy I/O-műveleteket végezzenek, hozzáférjenek a kernel táblázataihoz, és megvalósítsanak egyéb szükséges funkciókat anélkül, hogy a kernel részei lennének.

Ezeket a speciális szolgáltatásokat a **rendszertaszk (system task)** kezeli, amelyet a 2.29. ábra 1-es rétegében láthatunk. Bár a kernel tárgykódjába van fordítva, valójában egy külön processzusként ütemeződik. A rendszertaszk feladata az, hogy az eszközmeghajtóktól és a szerverektől kéréseket fogadjon, és azokat végrehajtsa. Mivel a rendszertaszk a kernel címtartományának része, ezért jogos, hogy itt tanulmányozzuk.

A fejezet korábbi részében láttunk példát a rendszertaszk által nyújtott szolgáltatásra. A megszakításkezelés tárgyalásakor leírtuk, hogy egy felhasználói szintű eszközmeghajtó a sys_irqctl rendszerhívást használva hogyan küld üzenetet a rendszertaszknak, amelyben kéri egy megszakításkezelő rutin regisztrálását. Egy felhasználói szintű eszközmeghajtó nem férhet hozzá azokhoz a táblázatokhoz, ahol a megszakításkezelő rutinok címei vannak, de a rendszertaszk igen. Továbbá, mivel a megszakításkezelő rutin címének a kernel címtartományában kell lennie, ezért a rendszertaszk a *generic_handler* függvény címét tárolja el. Ez úgy reagál egy megszakításra, hogy értesítést küld az eszközmeghajtónak.

Most jó alkalom nyílik a terminológia tisztázására. Hagyományos, monolitikus kernelű operációs rendszerben a **rendszerhívás (system call)** kifejezést minden olyan hívásra alkalmazzák, amikor a kernel valamilyen szolgáltatást nyújt. Modern, a Unixhoz hasonló operációs rendszerben a POSIX szabvány előírja, hogy milyen rendszerhívások álljanak a processzusok rendelkezésére. Természetesen lehetnek a szabványosakon kívül a POSIX-ot kibővítő hívások is, a programozók a rendszerhívásokra rendszerint valamilyen C könyvtári függvényen keresztül hivatkoz-

nak, mert ezek könnyen használható interfészt nyújtanak. Az is előfordulhat, hogy a programozó számára külön „rendszerhívás"-nak tűnő könyvtári függvények belül ugyanazzal a módszerrel fordulnak a kernelhez.

A MINIX 3-ban máshogy néznek ki a dolgok: az operációs rendszer komponensei felhasználói szinten futnak, habár rendszerprocesszusként speciális jogosultságaik vannak. A rendszerhívás kifejezést továbbra is használni fogjuk a POSIX által definiált rendszerhívásokra (és néhány MINIX-kiterjesztésre), amelyek az 1.9. ábrán láthatók, de a felhasználói processzusok nem kérnek közvetlenül a kerneltől szolgáltatást. A MINIX 3-ban a felhasználói processzusok rendszerhívásai a szerverprocesszusok felé irányuló üzenetekké alakulnak. A szerverprocesszusok üzeneteket használnak az egymással, az eszközmeghajtókkal és a kernellel történő kommunikációra. Ennek a résznek a témája a rendszertaszk, ez kapja meg az összes kernelszolgáltatásra irányuló kérést. Pontatlanul fogalmazva nevezhetnénk ezeket a kéréseket rendszerhívásoknak, de az egyértelműség kedvéért a **kernelhívás (kernel call)** kifejezést fogjuk használni. A felhasználói processzusok nem használhatják a kernelhívásokat. Sok esetben egy felhasználói processzus által kezdeményezett rendszerhívás egy szerver által kezdeményezett, hasonló nevű kernelhíváshoz vezet. Ez minden esetben azért történik így, mert a kérésnek olyan része van, amely kizárólag a kernel hatáskörébe tartozik.

Például egy felhasználói processzus által kezdeményezett fork rendszerhívás a processzuskezelőhöz kerül, amely a munka egy részét elvégzi. De az új processzus létrehozásához szükség van a processzustábla kernelben tárolt részének módosítására is, ezért a processzuskezelő egy sys_fork hívással jelez a rendszertaszknak, amely aztán a kernel táblázatait módosítani tudja. Nem minden kernelhívásnak van ilyen egyértelmű megfelelője a rendszerhívások között. Például van egy sys_devio kernelhívás az I/O-kapuk írására és olvasására. Ez a kernelhívás eszközmeghajtóktól érkezik. Az 1.9. ábrán felsorolt összes rendszerhívásnak több mint fele eredményezheti valamelyik eszközmeghajtó aktivizálását, és ezáltal egy vagy több sys_devio hívást.

Technikailag (a rendszerhívásokon és a kernelhívásokon kívül) beszélhetünk egy harmadik hívásfajtáról is. A processzusok közötti kommunikációt megvalósító send, receive és notify **üzenetkezelő alapművelet (message primitive)** tekinthető rendszerhívás jellegűnek. Valószínűleg hivatkoztunk is rájuk már így a könyv egyes részeiben – végül is a rendszert hívják. De igazából meg kellene különböztetni őket a rendszerhívásoktól és a kernelhívásoktól is. Használhatunk más kifejezést. Előfordul néha az **IPC alapművelet (IPC primitive)** vagy a **csapda (trap)** kifejezés, mindkettő megtalálható a forráskód megjegyzéseiben is. Az üzenetkezelő alapműveletre úgy kell gondolni, mintha egy rádiókommunikációs rendszer vivőhulláma lenne. Általában modulációra van szükség ahhoz, hogy a rádióhullámot hasznosítani lehessen; az üzenet típusa és egyéb komponensei teszik lehetővé, hogy a hívás információt hordozzon. Ritkán a modulálás nélküli hullám is hasznos lehet; például a repülőgépek leszállását segítő sugárnyaláb a repülőtereken. Ez a notify alapművelettel rokon, amely a feladótól eltekintve csak kevés információt hordoz.

2.7.1. A rendszertaszk áttekintése

A rendszertaszk 28 üzenetfajtát fogad, ahogy az a 2.45. ábrán látható. Ezek mindegyike tekinthető kernelhívásnak, bár látni fogjuk, hogy néha különböző névvel definiált makrók ugyanazt az ábrán látható üzenetfajtát eredményezik. Más esetekben pedig az ábra üzenetfajtái közül több is szerepel egy eljárásban, amely az adott feladatot elvégzi.

A rendszertaszk főprogramja a többi taszkhoz hasonló szerkezetű. A szükséges inicializálások után belép egy ciklusba. Ha kap egy üzenetet, átadja a megfelelő kiszolgáló eljárásnak, majd küldi a választ. Néhány általános kisegítő függvény ta-

Üzenettípus	Kitől	Jelentés
sys_fork	PM	Egy processzus kettévált
sys_exec	PM	Veremmutató beállítása EXEC után
sys_exit	PM	Egy processzus kilépett
sys_nice	PM	Ütemezési prioritás beállítása
sys_privctl	RS	Jogosultságok beállítása, módosítása
sys_trace	PM	PTRACE egy műveletének végrehajtatása
sys_kill	PM, FS, TTY	Szignál küldése processzusnak KILL hívása után
sys_getksig	PM	A PM a kezeletlen szignálokat ellenőrzi
sys_endksig	PM	A PM befejezte egy szignál feldolgozását
sys_sigsend	PM	Szignál küldése egy processzusnak
sys_sigreturn	PM	Szignál feldolgozása utáni takarítás
sys_irqctl	meghajtók	Megszakítás engedélyezése, tiltása és konfigurálása
sys_devio	meghajtók	I/O-kapu írása vagy olvasása
sys_sdevio	meghajtók	Adatsorozat írása vagy olvasása I/O-kapura/kapuról
sys_vdevio	meghajtók	I/O-kérések vektorának végrehajtása
sys_int86	meghajtók	Valós módú BIOS-hívás végrehajtása
sys_newmap	PM	Processzus memóriatérképének beállítása
sys_segctl	meghajtók	Szegmens hozzáadása és szelektor lekérése (távoli adatelérés)
sys_memset	PM	Memóriaterület feltöltése karakterrel
sys_umap	meghajtók	Virtuális cím konvertálása fizikai címre
sys_vircopy	FS, meghajtók	Másolás tisztán virtuális címzéssel
sys_physcopy	meghajtók	Másolás fizikai címzéssel
sys_virvcopy	bárki	VCOPY kérések vektora
sys_physvcopy	bárki	PHYSCOPY kérések vektora
sys_times	PM	Uptime és processzusidők lekérése
sys_setalarm	PM, FS, meghajtók	Szinkron riasztás beállítása
sys_abort	PM, TTY	Pánik: a MINIX nem tud tovább működni
sys_getinfo	bárki	Rendszer-információ lekérése

2.45. ábra. *A rendszertaszk által elfogadott üzenettípusok. A „bárki" azt jelenti, hogy bármelyik rendszerprocesszus; a felhasználói processzusok közvetlenül nem hívhatják a rendszertaszkot*

lálható a *system.c* nevű főállományban, de a főciklus a *kernel/system/* könyvtárban lévő külön fájlokban elhelyezett eljárásoknak adja át a feladatokat. A rendszertaszk megvalósításának tárgyalásakor látni fogjuk, hogy ez hogyan működik, és miért így van szervezve.

Először röviden leírjuk az egyes kernelhívások funkcióját. A 2.45. ábra üzenettípusai több csoportba sorolhatók. Az első néhány a processzuskezeléssel kapcsolatos. A sys_fork, sys_exec, sys_exit és a sys_trace nyilván szorosan kötődik a megfelelő szabványos POSIX-rendszerhíváshoz. Bár a *nice* nem része a POSIX szabványnak, ez a parancs végül a sys_nice kernelhíváshoz vezet, amivel a processzusok prioritását meg lehet változtatni. Ebből a csoportból egyedül talán a sys_privctl lehet ismeretlen. Ezt a reinkarnációs szerver (RS) használja, a MINIX 3-nak az a komponense, amely a közönséges felhasználói processzusként indított processzusok rendszerprocesszusokká történő konvertálásáért felelős. A sys_privctl megváltoztatja egy processzus jogosultságait, például azért, hogy kernelhívást kezdeményezzen. A sys_privctl-t akkor használjuk, amikor olyan eszközmeghajtót vagy szervert indítunk az */etc/rc* parancsfájlból, amely nem része a betöltési memóriaképnek. A MINIX 3-eszközmeghajtók bármikor indíthatók (vagy újraindíthatók); ilyen esetekben azonban szükség van a jogosultság módosítására.

A kernelhívások következő csoportja a szignálokhoz kapcsolódik. A sys_kill a felhasználók által elérhető (és félrekeresztelt) kill rendszerhíváshoz tartozik. A csoport többi tagja, a sys_getksig, sys_endksig, sys_sigsend és sys_sigreturn a processzuskezelőnek van fenntartva, hogy igénybe tudja venni a kernel segítségét a szignálok kezeléséhez.

A sys_irqctl, sys_devio, sys_sdevio és sys_vdevio kernelhívások csak a MINIX 3-ban találhatók meg. Ezek nyújtják a felhasználói szintű eszközmeghajtókhoz szükséges támogatást. A sys_irqctl-t már említettük a fejezet elején. Egyik funkciója egy hardver-megszakításkezelő beállítása, és a megszakítások engedélyezése felhasználói szintű eszközmeghajtóhoz. A sys_devio lehetővé teszi a felhasználói szintű eszközmeghajtóknak, hogy a rendszertaszkot I/O-kapuk írására és olvasására kérjék. Ez nyilván elengedhetetlen, és az is világos, hogy ez a módszer többletmunkával jár ahhoz képest, mintha a meghajtó a kernel része lenne. A következő két kernelhívás magasabb szintű I/O-eszköz támogatást nyújt. A sys_sdevio akkor használható, ha bájtok vagy szavak sorozatát kell kiírni egy I/O-címre, vagy beolvasni egy I/O-címről, például egy soros vonal esetében. A sys_vdevio arra használható, hogy egy vektorban tárolt I/O-kéréseket küldjünk a rendszertaszknak. Vektor alatt (kapu, érték) párok sorozatát kell érteni. A fejezet korábbi részében írtunk az *intr_init* függvényről, amely inicializálja az Intel i8259-es megszakításvezérlőket. A 8140. és a 8152. sor között egy utasítássorozat bájtok sorozatát írja ki. Mindkét i8259-es lapkának van egy módbeállító vezérlőkapuja, és egy másik, amely egy négybájtos sorozatot fogad az inicializáció során. Természetesen ez a programrész a kernelben hajtódik végre, és nem igényel közreműködést a rendszertaszk részéről. Ha azonban ugyanezt egy felhasználói processzus akarná végrehajtani, akkor egy 10 (kapu, érték) párból álló vektor címét tartalmazó üzenet sokkal hatékonyabb lenne, mint 10 üzenet, amelyek mindegyike egyetlen kapucímet és egy kiírandó értéket tartalmazna.

A 2.45. ábrán látható következő három kernelhívás különféle módokon a memóriával kapcsolatos. Az első a sys_newmap, amelyet a processzuskezelő hív, amikor változik valamelyik processzus által használt memória, így a kernelben lévő processzustábla rész is aktualizálható. A sys_segctl és a sys_memset biztonságos hozzáférést nyújtanak a processzusoknak a sajátjukon kívül eső memóriaterületekhez. A 0xa0000-tól 0xfffff-ig terjedő címtartomány az I/O-eszközök számára van fenntartva, ahogy azt már a MINIX 3 indulásával kapcsolatos részben említettük. Egyes eszközök ennek a memóriaterületnek egy részét I/O-műveletekre használják – például a videokártyák feltételezik, hogy a kártya memóriájának erre a területre leképezett részébe írt adatok megjelennek a képernyőn. A sys_segctl segítségével az eszközmeghajtók hozzájuthatnak olyan szegmensszelektorhoz, amellyel a hozzá tartozó memóriát megcímezhetik. A másik, a sys_memset akkor használható, amikor egy szerver olyan memóriaterületre akar adatokat írni, amely nem hozzá tartozik. A processzuskezelő arra használja, hogy új processzus indulásakor kinullázza a memóriát, ezzel megakadályozza, hogy az új processzus másik processzus által otthagyott adatokat olvashasson.

A kernelhívások következő csoportja memóriamásolásra való. A sys_umap virtuális címeket fizikai címekké konvertál. A sys_vircopy és a sys_physcopy memóriarégiókat másol virtuális, illetve fizikai címeket felhasználva. A következő két hívás, a sys_virvcopy és a sys_physvcopy az előzők vektoros verziói. Ugyanúgy, mint a vektoros I/O-kérések esetében, ezekkel egy másolássorozatot lehet kérni a rendszertaszktól.

A sys_times nyilvánvalóan az idővel kapcsolatos, a POSIX times rendszerhívásnak felel meg. A sys_setalarm a POSIX alarm rendszerhívással rokon, de csak távolról. A POSIX-hívást nagyrészt elintézi a processzuskezelő, amely a felhasználói processzusok számára karbantart egy időzítőkből álló sort. A processzuskezelő a sys_setalarm kernelhívást használja, amikor egy időzítő beállítására van szüksége. Ez csak akkor történik, amikor a processzuskezelő által karbantartott sor elején változás áll be, és nem feltétlenül akkor, amikor egy felhasználói processzus meghívja az alarm-ot.

A 2.45. ábra utolsó két kernelhívása rendszervezérlésre szolgál. A sys_abort kiindulhat a processzuskezelőből normál rendszerleállítási kérelmet követően vagy pánik után. A tty eszközmeghajtótól is eredhet, amennyiben a felhasználó lenyomta a CTRL-ALT-DEL billentyűkombinációt.

Végül a sys_getinfo egy nagyon általános információlekérő hívás. Ha végignézzük a MINIX 3 C forrásállományait, akkor valójában nagyon kevés hivatkozást találunk rá, de ha kiterjesztjük a keresést a definíciós könyvtárakra is, akkor nem kevesebb mint 13 makrót találunk az *include/minix/syslib.h*-ban, amelyek mind más nevet adnak a sys_getinfo-nak. Például a

```
sys_getkinfo(dst)  sys_getinfo(GET_KINFO, dst, 0, 0, 0)
```

a rendszerindításkor egy *kinfo* struktúrát (az *include/minix/type.h*-ban van definiálva a 2875. és 2893. sor között) ad vissza a processzuskezelőnek. Ugyanarra az információra többször is szükség lehet. Például a *ps* parancsnak ismernie kell a pro-

cesszustábla kernelben lévő részének helyét, hogy az összes processzus állapotáról információt tudjon adni. Megkérdezi a processzuskezelőt, amely a sys_getinfo fenti variánsát, a sys_getkinfo-t használja az információ megszerzéséhez.

Mielőtt befejezzük a kernelhívások típusainak áttekintését, meg kell említenünk, hogy a sys_getinfo nem az egyetlen kernelhívás, amelyet az *include/minix/syslib.h*-ban található makródefiníciók miatt több néven is el lehet érni. Például a sys_sdevio általában a sys_insb, sys_insw, sys_outsb vagy sys_outsw makrók által hívódik meg. A neveket úgy találtuk ki, hogy könnyen meg lehessen állapítani az adatátvitel irányát, illetve hogy bájtokkal vagy szavakkal dolgozunk-e. Hasonlóképpen a sys_irqctl-t is makrón keresztül hívjuk úgy, mint a sys_irqdisable-t és hasonlókat. Az ilyen makrók olvashatóbbá teszik a kódot, és a programozót is segítik azzal, hogy a konstans argumentumokat automatikusan generálják.

2.7.2. A rendszertaszk megvalósítása

A rendszertaszk a fordítás során a *kernel/* könyvtárban lévő *system.h* definíciós fájlból és a *system.c* C forrásállományból alakul ki. Ezenkívül van még egy speciális függvénykönyvtár a *kernel/system* könyvtárban. Megvan az oka ennek a szerkezetnek. Bár a MINIX 3-at itt úgy írjuk le, mint egy általános célú operációs rendszert, potenciálisan alkalmazható speciális célokra is, mint például valamilyen hordozható eszköz beágyazott rendszereként. Ilyen esetekben az operációs rendszer lecsupaszított verziója lehet a megfelelő. Például egy lemez nélküli eszköznek esetleg nincs szüksége a fájlrendszerre. A *kernel/config.h*-ban láttuk, hogy a kernelhívások fordítása egyenként engedélyezhető vagy letiltható. Könnyebbé teszi az egyedi rendszerek építését, ha az egyes kernelhívásokat megvalósító kódokat a fordítás utolsó fázisában a könyvtárból szerkesztjük be.

Az egyes kernelhívások kódjának külön állományba helyezése megkönnyíti a szoftver karbantartását. A *kernel/system/* könyvtár megtalálható a MINIX 3 weboldalán és a mellékelt CD-n is.

A *kernel/system.h* definíciós állománnyal (9600. sor) kezdjük. A 2.45. ábrán látható kernelhívások legtöbbjéhez tartozó függvények prototípusa megtalálható benne. Van még egy *do_unused* prototípus; ez a függvény nem támogatott kernelhívás esetén fut le. A 2.45. ábra néhány csoportja itt definiált makróknak felel meg. Ezek a 9625. és a 9630. sor között találhatók. Ezek olyan esetek, amikor egy függvény egynél több hívást is kezelni tud.

Mielőtt a *system.c* kódjára rátérnénk, figyeljük meg a *call_vec* hívási vektor deklarációját és a *map* makró definícióját a 9745. és a 9749. sor között. A *call_vec* függvényekre mutató pointerek tömbje, ami lehetővé teszi, hogy egy adott üzenet kiszolgálásához tartozó függvényt az üzenet típusa alapján találjunk meg. Ehhez az üzenet típusát, mint számot, indexként használjuk a tömbhöz. Ezt a módszert a MINIX 3 más részeiben is felfedezhetjük. A *map* makróval kényelmes lehet inicializálni ilyen tömböket. Úgy van definiálva, hogy ha érvénytelen argumentummal próbáljuk meg kifejteni, akkor egy negatív méretű tömböt deklarál. Ez természetesen lehetetlen, ezért fordítási hibát eredményez.

A rendszertaszk legfelső szintje a *sys_task* eljárás. Miután a függvénypointerek tömbjének inicializációja megtörtént, a *sys_task* belép egy ciklusba. Vár egy üzenetre, néhány lépésben ellenőrzi az érvényességét, majd a kérést átadja az üzenet típusának megfelelő függvénynek. Ha válaszolni kell, akkor a választ visszaküldi, és ezt a ciklust (9768–9796. sor) ismétli egészen addig, amíg a MINIX 3 fut. Az érvényesség ellenőrzése során a *priv* táblabejegyzés alapján megállapítja, hogy a hívónak engedélyezve van-e ez a típusú hívás, és hogy a hívás típusa egyáltalán érvényes-e. A munkát elvégző függvény hívása a 9783. sorban van. A *call_vec* tömb indexelésére a hívás számát használjuk, azt a függvényt hívjuk meg, amelyre a tömb megfelelő eleme mutat. A hívás argumentuma a kapott üzenetre mutató pointer, visszatérési értéke pedig egy állapotkód. Elképzelhető, hogy a függvény az *EDONTREPLY* kóddal tér vissza; ez azt jelenti, hogy nem kell válaszolni. Különben a 9792. sorban küldjük a választ.

Ahogy a 2.43. ábrán látható, a MINIX 3 indulásakor a rendszertaszk a legmagasabb prioritású sor elején van, ezért van értelme, hogy a megszakításkezelők adatszerkezetét és az időzítők listáját a rendszertaszk *initialize* függvénye inicializálja (9808–9815. sor). Mindenesetre ahogy korábban is megjegyeztük, a megszakításokra reagáló felhasználói szintű eszközmeghajtók számára a rendszertaszk engedélyezi a megszakításokat, ezért jogos, hogy ebben van a táblázat előkészítése. Hasonló okok miatt került ide a többi rendszerprocesszus által kért szinkron riasztásokhoz használt időzítőlisták beállítása is.

Folytatva az inicializációt, a 9822. és a 9824. sor között a *call_vec* tömb minden elemébe a *do_unused* függvény címe kerül, ezt hívjuk, ha nem támogatott kernelhívást kísérel meg valaki. A fájl maradék részében a 9827. és a 9867. sor között a *map* makró több kifejtése található, mindegyik egy függvény címét helyezi el a *call_vec* megfelelő indexű elemébe.

A *system.c* maradék része *PUBLIC*-ként deklarált függvényekből áll; ezeket a kernelhívásokat feldolgozó függvények is, de a kernel más részei is hívhatják. Például az első függvényt, a *get_priv*-et (9872. sor) a sys_privctl kernelhívást megvalósító *do_privctl* használja. Hívja még a kernel is, amikor a betöltési memóriakép processzusai számára processzustábla-bejegyzéseket hoz létre. A név talán egy kicsit félrevezető. A *get_priv* nem a már meglévő jogosultságokról ad vissza információt, hanem egy üres *priv* bejegyzést keres, és hozzárendeli a hívóhoz. Két eset van – a rendszerprocesszusok mindegyike saját bejegyzést kap a *priv* táblában. Ha nincs üres, akkor a processzus nem válhat rendszerprocesszussá. A felhasználói processzusok mind a tábla ugyanazon bejegyzésén osztoznak.

A *get_randomness* (9899. sor) kiindulási értékeket szolgáltat a véletlenszámgenerátorhoz, amely karakteres eszközként van implementálva a MINIX 3-ban. A legújabb Pentium osztályú processzorokban van egy belső ciklusszámláló, és annak egy utasítása, amellyel a számlálót ki lehet olvasni. Ezt használjuk, ha rendelkezésre áll, különben egy olyan függvényt hívunk meg, amely az időzítőlapka egyik regiszterét olvassa ki.

A *send_sig* értesítést küld egy rendszerprocesszusnak, miután az *s_sig_pending* bittérképben beállította a processzushoz tartozó bitet. A bitet a 9942. sorban állítja be. Figyeljük meg, hogy az *s_sig_pending* bittérkép a *priv* struktúra része, ezért

ezzel a módszerrel csak rendszerprocesszusoknak küldhető értesítés. Az összes felhasználói processzushoz egyetlen *priv* táblabejegyzés tartozik, az *s_sig_pending* mezőn viszont nem osztozhatnak, ezért egyáltalán nem használják. Még a *send_sig* hívása előtt megtörténik annak ellenőrzése, hogy a címzett egy rendszerprocesszus-e. A hívás vagy egy sys_kill kernelhívás eredménye, vagy a kerneltől jön, amikor a *kprintf* karaktersorozatot küld. Az első esetben a hívó állapítja meg, hogy a címzett rendszerprocesszus-e. A második esetben a kernel csak a beállított kimeneti processzusnak küld, amely vagy a konzol-, vagy a naplózómeghajtó lehet, de mindkettő rendszerprocesszus.

A következő függvény a *cause_sig* (9949. sor), azért hívjuk, hogy szignált küldjünk egy felhasználói processzusnak. Akkor van rá szükség, amikor a sys_kill kernelhívás célpontja egy felhasználói processzus. Azért van a *system.c*-ben, mert felhasználói processzusban keletkezett kivétel hatására a kernel közvetlenül is hívhatja. A *send_sig*-hez hasonlóan a fogadó kezeletlen szignálokat nyilvántartó bittérképében be kell állítani egy bitet, de a felhasználói processzusok esetében ez nem a *priv* táblában van, hanem a processzustáblában. A szóban forgó processzust a *lock_dequeue*-val ki is kell venni a futtathatók közül, valamint a (szintén a processzustáblában lévő) jelzőbitjeit aktualizálva rögzíteni kell, hogy szignált fog kapni. Ezután egy üzenet megy ki – de nem a célprocesszusnak. Az üzenetet a processzuskezelő kapja, amely aztán mindent elrendez a szignállal kapcsolatban, amit egy felhasználói szinten futó rendszerprocesszus csak elrendezhet.

Ezután három olyan függvény következik, amelyek mind a sys_umap kernelhívást támogatják. A processzusok rendszerint virtuális, vagyis egy bizonyos szegmens elejéhez viszonyított relatív címeket használnak. Néha azonban meg kell tudniuk egy memóriaterület abszolút (fizikai) címét, például ha két szegmens közötti adatmásolást készülnek kérni. Háromféleképpen lehet egy virtuális címet megadni. A processzusok alapesetben valamelyik hozzájuk rendelt és a processzustáblában bejegyzett program-, adat- vagy veremszegmens elejéhez képest értelmezett relatív címeket használnak. Az ilyen virtuális címek fizikaira történő konverzióját az *umap_local* függvény (9983. sor) végzi.

A második fajta memóriahivatkozás olyan régióra történhet, amely kívül van ugyan a processzushoz rendelt program-, adat- és veremszegmenseken, de amelyért valamilyen okból a processzus mégis felelős. Példa lehet erre egy video- vagy Ethernet-eszközmeghajtó, mert a hozzájuk tartozó videokártya vagy Ethernet-kártya memóriájának egy része az I/O-eszközök részére fenntartott 0xa0000-0xfffff sávba lehet leképezve. Egy másik példa lehet a memória eszközmeghajtó, amely a RAM-lemezt kezeli, de amely képes a memória bármely részéhez is hozzáférést biztosítani a */dev/mem* és a */dev/kmem* eszközökön keresztül. Ilyen memóriahivatkozások esetén a virtuálisról fizikaira történő konverzió az *umap_remote* feladata (10025. sor).

Végül hivatkozhatunk olyan memóriarészre is, amit a BIOS használ. Ehhez számítjuk a legalsó 2 KB-os részt, az alatt, ahova a MINIX 3 betöltődik, valamint a 0x90000-0xfffff részt, amelyhez a betöltött MINIX 3 felett tartozik valamennyi RAM, plusz az I/O-eszközöknek fenntartott terület. Az *umap_remote* ezt is tudja kezelni, de a harmadik függvény, az *umap_bios* (10047. sor) ellenőrzi is, hogy a megadott címek valóban ebbe a tartományba esnek.

A *system.c* utolsó függvénye a *virtual_copy* (10071. sor). Ennek a függvénynek legnagyobb része egy C switch utasítás, amely a fent említett három *umap_** közül választva a virtuális címeket átkonvertálja fizikaira. Ez megtörténik a forrásra és a célterületre is, majd a tényleges másolást a *klib386.s*-ben található assembly nyelvű *phys_copy* végzi (10121. sor).

2.7.3. A rendszerkönyvtár megvalósítása

Minden *do_xyz* alakú névvel ellátott függvény forráskódja egy alkönyvtárban, a *kernel/system/do_xyz.c*-ben van. A *kernel/* könyvtárban lévő *Makefile* tartalmaz egy

```
cd system && $(MAKE) -$(MAKEFLAGS) $@
```

sort, amelynek hatására a *kernel/system/* minden állománya lefordul, és előáll a *system.a* a *kernel/* könyvtárban. Amikor a vezérlés visszatér a fő kernelkönyvtárba, akkor a *Makefile* egy másik sorának hatására a kernel a tárgykódú fájlok szerkesztésekor először a lokális függvénykönyvtárból oldja fel a hivatkozásokat.

A *kernel/system/* könyvtárból itt két fájllal foglalkozunk. Azért ezeket választottuk, mert jól reprezentálják a rendszertaszk által nyújtott támogatások két osztályát. Az egyik támogatási kategória a kernel adatszerkezeteihez való hozzáférés olyan felhasználói szinten futó rendszerprocesszusok számára, amelyek igénylik ezeket az adatokat. Példaként a *system/do_setalarm.c* leírása fog szolgálni. A másik általános kategória az olyan rendszerhívások támogatása, amelyeknek nagyobb részét felhasználói szintű processzusok végrehajtják, de bizonyos műveleteket kernelszinten kell elvégezniük. Példaként a *system/do_exec.c*-t választottuk.

A sys_setalarm kernelhívás valamennyire hasonlít a sys_irqenable-re, amelyet említettünk, amikor a megszakításkezelést tárgyaltuk a kernelben. A sys_irqenable beállítja egy megszakításkezelő címét arra az esetre, ha egy adott vonalon megszakítás érkezne. A kezelő egy függvény a rendszertaszkon belül, a *generic_handler*. Ez egy értesítést generál annak az eszközmeghajtó processzusnak, amelynek reagálnia kell a megszakításra. A *system/do_setalarm.c* (10200. sor) olyan programkódot tartalmaz, amely az időzítőket a megszakításokhoz hasonlóan kezeli. Egy sys_setalarm kernelhívás inicializál egy időzítőt az olyan felhasználói processzusok számára, amelyeknek szinkron riasztásra van szükségük, és rendelkezésre bocsát egy függvényt, amelyet az időzítő lejártakor meg kell hívni, hogy a felhasználói processzus értesítést kapjon az eseményről. Egy korábban kért riasztás visszavonását is lehet kérni, ha lejárati időnek 0 értéket adunk meg az üzenetben. A művelet egyszerű – a 10230. és 10232. sor között az üzenetből kinyerjük az információt. A két legfontosabb tétel az időpont, amikor az időzítő lejár, illetve a processzus, amelynek erről tudomást kell szereznie. Minden rendszerprocesszusnak saját időzítő struktúrája van a *priv* táblában. A 10237. és a 10239. sor között meghatározzuk az időzítőstruktúra címét, majd a processzus számát, illetve a *cause_alarm* függvény címét elhelyezzük benne. Utóbbi függvényt kell végrehajtani, amikor az időzítő lejár.

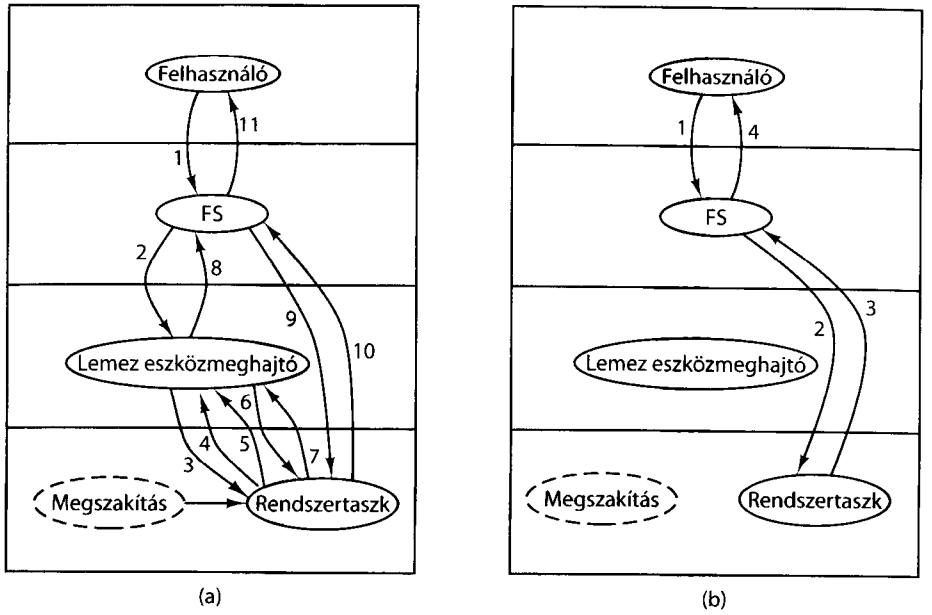
Ha az időzítő már aktív volt, akkor a sys_setalarm válaszüzenetében tudatja, hogy a riasztásig mennyi idő lett volna még hátra. A nulla visszatérési érték azt jelzi, hogy az időzítő nem aktív. Több lehetőséget kell figyelembe venni. Lehet, hogy az időzítő előzőleg le lett állítva – az időzítő inaktív voltát az *exp_time* mezőjében tárolt speciális *TMR_NEVER* érték jelzi. Ami a C programot illeti, ez csak egy nagy egész szám, ezért ezt az értéket explicit módon használjuk, amikor azt ellenőrizzük, hogy a lejárati idő elmúlt-e már. Az időzítő jelezhet olyan időpontot, amely már elmúlt. Ez nem valószínű, hogy megtörténik, de könnyű ellenőrizni. Az időzítő jelezhet jövőbeli időpontot is. Az első két esetben a visszatérési érték nulla, egyébként a hátralévő időt adja vissza (10242–10247. sor).

Végül az időzítő leállítása vagy beállítása következik. Ezen a szinten ez úgy történik, hogy a kívánt lejárati időt az időzítőstruktúra megfelelő mezőjébe írjuk, majd meghívunk egy függvényt, amely a konkrét munkát elvégzi. Természetesen az időzítő leállításához nem kell új értéket eltárolni. Hamarosan látni fogjuk a *reset* és a *set* függvényt, programkódjuk az időzítőtaszk forrásállományában van. Mivel a rendszertaszk és az időzítőtaszk is a kernelbe van fordítva, mindkét függvény deklarációja *PUBLIC,* ezért hozzáférhetők.

Van még egy függvény a *do_setalarm.c*-ben. Ez a *cause_alarm,* a felügyelőfüggvény, amelynek címe minden időzítőbe bekerül, így meg lehet hívni, ha a beállított idő letelik. A függvény maga az egyszerűség – értesítést (notify) küld annak a processzszusnak, amelynek száma szintén az időzítőstruktúrában van tárolva. Így a kernelbeli szinkron riasztás a kérelmező rendszerprocesszusnak küldött üzenetté alakul.

Mellékesen megjegyezzük, hogy amikor néhány oldallal ezelőtt (és ebben a szakaszban is) az időzítők inicializálásáról beszéltünk, akkor előkerült a rendszerprocesszusok által kért szinkron riasztás. Ha nem volt teljesen érthető, vagy a kedves olvasó azon tűnődik, hogy mi az a szinkron riasztás, esetleg mi a helyzet a nem-rendszerprocesszusok időzítőivel, akkor azt válaszolhatjuk, hogy ezekkel a kérdésekkel a következő szakaszban, az időzítőtaszk tárgyalásakor foglalkozunk. Olyan sok egymással összefüggő része van egy operációs rendszernek, hogy jószerivel lehetetlen a témákat úgy sorrendbe állítani, hogy időnként ne kelljen hivatkozni olyan részre, amely csak később következik. Ez különösen igaz akkor, ha a megvalósításról beszélünk. Ha nem egy valóságos operációs rendszerrel foglalkoznánk, akkor valószínűleg el tudnánk kerülni az ehhez hasonló zűrös részleteket. Ami azt illeti, az operációs rendszerek alapjainak teljesen elméleti tárgyalása során valószínűleg említést sem tennénk rendszertaszkról. Egy elméleti könyvben csak legyintünk, és figyelmen kívül hagyjuk azt a problémát, hogy miként lehet felhasználói szinten futó rendszerkomponenseknek korlátozott és szabályozott hozzáférést biztosítani olyan központosított erőforrásokhoz, mint például a megszakítások vagy az I/O-kapuk.

A *kernel/system/* könyvtárban a *do_exec.c* (10300. sor) az utolsó fájl, amelyet részletesen tárgyalunk. Az exec rendszerhívással kapcsolatos teendők nagy részét a processzuskezelő elvégzi. A processzuskezelő előkészít egy vermet az új programnak, amely tartalmazza az argumentumokat és a környezetet. A verem címét átadja a kernelnek egy sys_exec kernelhívás keretében. Ezt a hívást a *do_exec* (10318. sor) kezeli. A veremmutató a processzustábla kernelben tárolt részében kerül beállításra, és ha az éppen létrehozott processzus más memóriaszegmenst is



2.46. ábra. *(a) Egy blokk olvasása legrosszabb esetben 11 üzenetet igényel. (b) Egy blokk olvasása legjobb esetben is 4 üzenetet igényel*

használni fog, akkor a *klib386.s*-ben definiált *phys_memset* függvény törli azokat az adatokat, amelyek korábbról maradhattak azon a memóriaterületen (10330. sor).

Az exec hívás okoz egy kis anomáliát. A hívást kezdeményező processzus üzenetet küld a processzuskezelőnek, és blokkolódik. Más rendszerhívások esetén a válaszüzenet feloldaná a blokkolást. Az exec esetében azonban nincs válasz, mert az éppen betöltött futtatható program nem várja a választ. Ezért a *do_exec* a 10333. sorban magát a processzust szabadítja fel a blokkolás alól. A következő sor futásra kész állapotba helyezi a programot a *lock_enqueue* függvénnyel, amely zárolással kivédi a lehetséges versenyhelyzeteket. Végül a parancs neve elmentésre kerül, hogy a felhasználó azonosítani tudja a processzust, ha kiadja a *ps* parancsot, vagy lenyom egy olyan funkcióbillentyűt, amelynek hatására a processzustábla adataiba kaphat betekintést.

A rendszertaszk tárgyalásának végén megnézzük, hogy milyen szerepet játszik egy tipikus operációsrendszer-szolgáltatás kezelésében, konkrétan egy read rendszerhívás kiszolgálásában. Amikor a felhasználó hívja a read-et, akkor először a fájlrendszer ellenőrzi, hogy a gyorsítótárában megvan-e a kért blokk. Ha nincs, akkor üzenetet küld a megfelelő lemezhez tartozó eszközmeghajtónak, hogy töltse be a gyorsítótárba. Ezután a fájlrendszer üzenetet küld a rendszertaszknak, kérve a blokk átmásolását a felhasználói processzushoz. A legrosszabb esetben 11 üzenetre van szükség egy blokk beolvasásához, a legjobb esetben pedig csak 4-re. A 2.46. ábra mindkét esetet bemutatja. A 2.46.(a) ábrán a 3-as üzenet kéri a rendszertaszkot az I/O-műveletek elvégzésére, a 4-es üzenet a nyugta. Amikor

hardvermegszakítás történik, akkor a rendszertaszk az 5-ös üzenetben tájékoztatja erről a várakozó eszközmeghajtót. A 6-os üzenet kéri az adatoknak a fájlrendszer gyorsítótárába másolását, a 7-es üzenet az erre küldött válasz. A 8-as üzenet tudatja a fájlrendszerrel, hogy az adatok készen állnak, a 9-es és a 10-es pedig az adatok átmásolásának kérelme a felhasználóhoz, illetve a válasz. Végül a 11-es a felhasználónak küldött válasz. A 2.46.(b) ábrán a kért adat már a gyorsítótárban van, a 2-es és a 3-as üzenet az adatok átmásolásának kérelme a felhasználóhoz, illetve a válasz. Ezek az üzenetek többletmunkát jelentenek a MINIX 3-ban, ez az ára a moduláris felépítésnek.

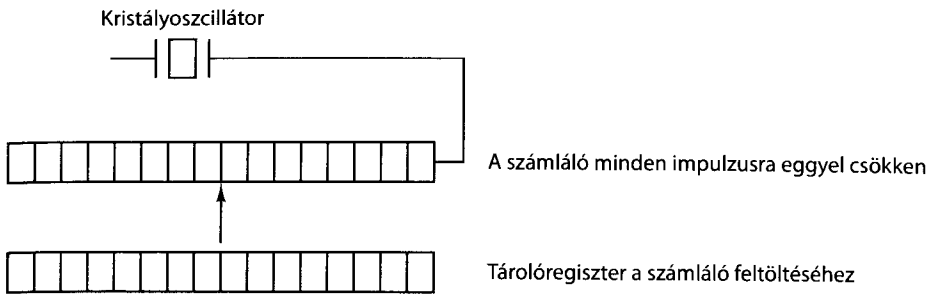
Valószínűleg az adatmásolást kérő kernelhívások a leggyakoribbak a MINIX 3-ban. Már láttuk a rendszertaszknak azt a részét, amelyik a konkrét másolást elvégzi; ez volt a *virtual_copy* függvény. Az üzenetküldési mechanizmus nem megfelelő hatékonyságának egyik ellenszere az, ha több kérést helyezünk el egyetlen üzenetben. A sys_virvcopy és a sys_physvcopy kernelhívások ezt teszik. A hívásokat kiváltó üzenetben van egy olyan vektorra mutató pointer, amely több átmásolandó memóriablokk leírását tartalmazza. Mindkettő a *do_vcopy*-t hívja, amely egy ciklusban először meghatározza a forrás- és a célterület címét, majd a *phys_copy* segítségével végrehajthatja az átmásolást, amíg az összes blokk sorra nem kerül. A következő fejezetben látni fogjuk, hogy a lemezegységek hasonlóképpen tudnak egy kérésben több átviteli kérelmet kezelni.

2.8. A MINIX 3-időzítőtaszk

Az **időzítők** (vagy **órák**) sok okból alapvető fontosságúak az időosztásos operációs rendszerek működésében. Például nyilvántartják a valós időt, és megakadályozzák, hogy valamelyik processzus kisajátítsa magának a CPU-t. A MINIX 3-időzítőtaszkja némileg hasonlít egy eszközmeghajtóra, mert egy hardvereszköz által generált megszakítások irányítják a működését. Az időzítő azonban nem blokkos eszköz, mint egy lemezegység, és nem is karakteres eszköz, mint mondjuk egy terminál. Valójában a MINIX 3-ban az időzítőhöz nem férhetünk hozzá a */dev/* könyvtár egyetlen állományán keresztül sem. Továbbá az időzítőtaszk a kernel címtartományában működik, és a felhasználói processzusok nem érhetik el közvetlenül. A kernel minden függvényéhez és adatához hozzáfér, de a felhasználói processzusok csak a rendszertaszkon keresztül érhetik el. Ebben az alfejezetben először az időzítőhardvert és -szoftvert tekintjük át általánosságban, majd megnézzük, hogy az elveket hogyan alkalmaztuk a MINIX 3 esetében.

2.8.1. Időzítőhardver

A számítógépekben kétfajta órát használnak; mindkettő lényegesen különbözik azoktól az óráktól, amiket az emberek használnak. Az egyszerűbbek a 110 vagy 220 voltos elektromos hálózatra csatlakoznak, és az 50 vagy 60 Hz-es frekvenciának megfelelően minden ciklusban megszakítást okoznak. Ezek jószerivel nem is fordulnak elő a modern személyi számítógépekben.



2.47. ábra. *Programozható óra (időzítő)*

A másik fajta órának három komponense van: egy kristályoszcillátor, egy számláló és egy tárolóregiszter, ahogy a 2.47. ábrán látható. Ha egy kvarckristályt megfelelően darabolunk és mechanikai feszültséget keltünk benne, akkor nagyon nagy pontosságú elektromos jelet állíttathatunk elő vele, kristálytól függően tipikusan az 5–200 MHz tartományban. Rendszerint minden számítógépben van legalább egy ilyen áramkör, amely a többi áramkörnek szinkronizáló jelet állít elő. Az előállított jel a számlálóhoz kerül, amely ennek hatására folyamatosan számlál lefelé. Amikor eléri a nullát, akkor megszakítást idéz elő. A 200 MHz-nél nagyobb frekvenciájúnak mondott számítógépekben rendszerint ennél lassabb óra és többszöröző áramkör van.

A programozható óráknak általában több üzemmódja van. Az **egyszeri módban (one-shot mode)**, az óra indulásakor a tárolóregiszter tartalma átmásolódik a számlálóregiszterbe, és a kristály minden egyes impulzusának hatására a számláló tartalma eggyel csökken. Amikor a számláló értéke eléri a nullát, akkor ez megszakítást okoz, és az óra megáll addig, amíg a szoftver újra nem indítja. **Ismétlődő módban (square-wave mode)**, miután a számlálóregiszter elérte a nullát és kiváltotta a megszakítást, a tárolóregiszter tartalma automatikusan újra bemásolódik a számlálóba, és az egész folyamat ismétlődik a végtelenségig. Ezeket a periodikus megszakításokat **órajeleknek (clock ticks)** nevezzük.

A programozható órák előnye, hogy a megszakítások frekvenciáját szoftver segítségével be tudjuk állítani. Ha egy 1 MHz-es kristályt használunk, akkor a számláló minden milliomod másodpercben impulzust kap. 16 bites regisztereket használva, a megszakítások közötti intervallum programozható módon 1 milliomod másodperc és 65 536 milliomod másodperc közé eshet. A programozható óralapkák általában két vagy három egymástól függetlenül programozható órát tartalmaznak, és egy sor egyéb funkciójuk lehet (például a lefelé számlálás helyett felfelé számlálhatnak, letilthatók a megszakítások, és így tovább).

Annak megakadályozására, hogy a számítógép kikapcsoláskor elfelejtse a pontos időt, elemmel működtetett másodlagos órát használnak, amely olyan alacsony fogyasztású áramkörökből áll, mint a digitális karórák. Ennek az elemmel mű-

ködtetett órának a tartalma rendszerindításkor kiolvasható. Ha ilyen másodlagos óra nincs a számítógépben, akkor a szoftver megkérdezheti a dátumot és a pontos időt a felhasználótól. Hálózati rendszerekben rendelkezésre áll egy szabványos protokoll arra, hogy a dátumot és a pontos időt egy távoli gépről kérdezzük le. Akárhogy is jutunk hozzá a pontos időhöz, azt át kell számítani valamely bázisidőponttól eltelt másodpercekre. A bázisidőpont lehet a Unix és MINIX által is használt **Universal Coordinated Time (UTC)** – régebben greenwichi középidő – szerinti 1970. január 1., 0 óra, vagy valami más. Az órajeleket a futó rendszer számolja, és minden eltelt másodperc után a valós idő számlálóját eggyel növeli. A MINIX 3 (és a legtöbb Unix-rendszer) nem veszi figyelembe a szökőmásodperceket, amelyekből 23 volt 1970 óta. Ezt nem tekintik súlyos hibának. Általában rendelkezésre állnak olyan segédprogramok, amelyekkel be lehet állítani a rendszer óráját és a háttérórát, illetve szinkronizálni lehet ezeket.

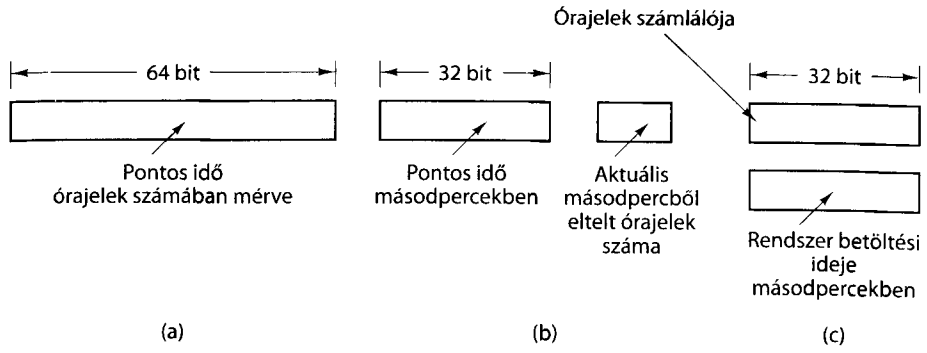
Meg kell említenünk, hogy a legkorábbi IBM-kompatibilis rendszerek kivételével minden számítógépeknek külön óraáramköre van a CPU, a belső adatsínek és más komponensek időzítőjeleinek előállítására. Ez az az óra, amire az emberek gondolnak, amikor CPU-órajelekről és -sebességről beszélnek. Ennek kezdetben megahertz volt az egysége, újabban a gigahertz tartományba esnek. Ezeknek az áramköre ugyanazokból az alapelemekből épül fel, de a velük szemben támasztott követelmények annyira eltérők, hogy a modern számítógépeknek külön áramkörük van a CPU vezérlésére és az idő nyilvántartására.

2.8.2. Időzítőszoftver

Az időzítő hardvere mindössze annyit tesz, hogy ismert időközönként megszakításokat generál. Az idővel kapcsolatos minden egyebet a szoftvernek, az időzítőmeghajtónak kell elvégeznie. Az időzítőmeghajtó pontos teendői változhatnak a különböző operációs rendszerekben, de általában a következőket tartalmazzák:

1. A pontos idő karbantartása.
2. Annak megakadályozása, hogy egy processzus tovább fusson, mint ami számára engedélyezett.
3. A felhasznált CPU-idő könyvelése.
4. A felhasználói processzusok által kezdeményezett alarm rendszerhívás lekezelése.
5. Felügyeleti időzítők (watchdog timer) nyújtása a rendszer többi része felé.
6. Futásidő-elemzés (profiling), monitorozás és statisztikai adatok gyűjtése.

Az első funkció, a pontos idő (másképpen valós idő – real time) karbantartása nem nehéz. Ehhez csak egy számlálót kell növelni minden egyes órajelre, mint ahogy ezt már említettük. Az egyetlen dolog, amire érdemes odafigyelni, az a pontos időt nyilvántartó regiszter bitjeinek száma. 60 Hz-es óra esetén egy 32 bites számláló nem sokkal két év után túlcsordul. Vagyis a rendszer nyilván nem tárolhatja a valós időt 32 biten az 1970. január 1-je óta eltelt órajelek számával.



2.48. ábra. *A pontos idő nyilvántartásának három módja*

Három megközelítés lehetséges az előbbi probléma megoldására. Az első megközelítés szerint 64 bites számlálót kell használni, bár így a számláló karbantartása sokkal időigényesebb, hiszen másodpercenként igen sokszor kell ezt elvégezni. A második megközelítés szerint a számlálóban a másodpercek számát tartjuk nyilván az órajelek száma helyett, egy kiegészítő számlálóban számoljuk az órajeleket addig, amíg egy teljes másodperc el nem telik. A 2^{32} másodperc több mint 136 év, így ez a módszer egészen jól fog működni a XXII. század elejéig.

A harmadik megközelítés szerint az órajeleket számoljuk, de nem egy előre meghatározott időponthoz, hanem a rendszerindítás időpontjához viszonyítva. Amikor a háttérórát olvassuk, vagy a felhasználó beállítja a valós (pontos) időt, akkor a rendszerindítás idejét ki kell számolni az így nyert pontos idő alapján, és ezt az értéket el kell tárolni a memóriában valamilyen megfelelő formában. Később, amikor szükség van a pontos időre, akkor ezt az eltárolt idő és a számláló összeadásával meg lehet határozni. A 2.48. ábra mind a három megközelítést bemutatja.

A második órafunkció annak megakadályozása, hogy egy processzus túl sokáig fusson. Amikor egy processzus elindul, akkor az ütemezőnek be kell állítania egy számlálót arra az értékre, amennyi időt a processzus órajelekben megadva futhat. Az időzítő által generált minden egyes megszakításkor az időzítőmeghajtója a számlálóban levő értéket eggyel csökkenti. Amikor a számláló eléri a nullát, akkor az időzítőmeghajtó meghívja az ütemezőt, hogy másik processzust indítson el.

A harmadik órafunkció a CPU használatának könyvelése. Ennek legpontosabb módja az, hogy a rendszer időzítőjétől független második időzítőt indítunk minden olyan alkalommal, amikor egy processzus elindul. Amikor a processzus megáll, akkor az időzítőből kiolvasható, hogy mennyi ideig futott. A helyes eredményhez az időzítő értékét el kell mentenünk minden egyes megszakítás beérkezésekor, és vissza kell állítanunk a megszakítás befejezésekor.

Egy kevésbé pontos, ám sokkal egyszerűbb módja a könyvelésnek, hogy egy globális változóban egy mutatót tárolunk az éppen futó processzus processzustáblabeli bejegyzésére. Minden egyes órajelre a processzus bejegyzésének egyik mezőjét eggyel növeljük. Így minden egyes órajelet „ráterhelünk" arra a processzusra,

amely az órajel érkezésekor éppen futott. Ennek a stratégiának gyenge pontja, hogy ha a processzus futása alatt sok megszakítás keletkezik, akkor az ezekben töltött idő teljes egészében a processzust terheli, bár maga a processzus nem túl sok mindent tudott elvégezni. A megszakítások alatti CPU-használat pontos könyvelése túl költséges, ezért ritkán teszik meg.

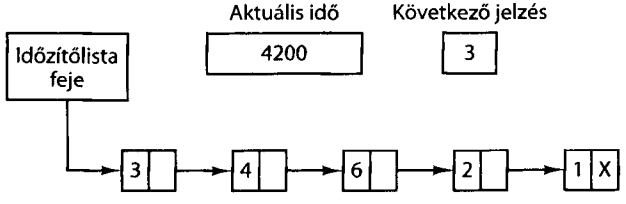
A MINIX 3-ban és sok más operációs rendszerben a processzusok kérhetik az operációs rendszert, hogy bizonyos időintervallum elteltével küldjön nekik figyelmeztetést. Ez a figyelmeztetés általában egy szignál, megszakítás, üzenet, vagy valami hasonló. Például egy hálózati alkalmazás kérhet ilyen figyelmeztetést, mert ha egy elküldött csomagra megadott időn belül nem érkezik nyugtázás, akkor a csomagot újra kell küldeni. Egy másik alkalmazás a számítógéppel segített oktatás, amikor a tanulónak meg kell mondani a választ, ha nem válaszol egy megadott időn belül.

Ha az időzítőmeghajtónak elég óra áll rendelkezésére, akkor minden egyes kéréshez más-más órát használhat. Ha ez nincs így, akkor egy fizikailag létező órával több virtuális órát kell szimulálnia. Ennek egyik módja egy olyan táblázat karbantartása, amely az összes függőben lévő riasztáshoz tartozó időpontokat tartalmazza, valamint egy változó, amely az időben legközelebbi idejét tárolja. Valahányszor a pontos idő frissítésre kerül, a meghajtó megvizsgálja, hogy a legközelebbi riasztási időpont elérkezett-e. Ha igen, akkor a táblázatból kikeresi a következő legközelebbi időpontot.

Ha sokan várakoznak riasztásra, akkor több órát hatékonyabban lehet úgy szimulálni, hogy idő szerint rendezve egy láncolt listába fűzzük a várakozó riasztási igényeket, ahogy a 2.49. ábra mutatja. A lista minden egyes eleme megadja, hogy a megelőzőhöz képest hány órajelet kell várni a jelzésig. Ebben a példában a riasztások a 4203., 4207., 4213., 4215. és 4216. órajelnél következnek be.

A 2.49. ábra szerint egy időzítő éppen lejárt. A következő megszakítás 3 órajel múlva következik be, a 3 éppen be lett töltve a számlálóba. Minden órajelre a *Következő jelzés* eggyel csökken. Amikor eléri a nullát, akkor létrejön a lista első elemének megfelelő jelzés, az elemet pedig eltávolítjuk a lista elejéről. Ezután a *Következő jelzés* értékét a lista új első elemének értéke szerint állítjuk be, a példa szerint 4-re. Sok esetben relatív időknél sokkal kényelmesebb abszolút időket használni, ezért a MINIX 3-ban is ezt a megközelítést választottuk.

Figyeljük meg, hogy egy óramegszakítás alatt az időzítőmeghajtójának sok a dolga. Többek között növelnie kell a valós időt, csökkentenie kell az időszeletből hátralévő időt, és ellenőriznie kell annak 0 voltát, el kell végeznie a CPU-idő



2.49. ábra. *Több időzítő szimulálása egyetlen órával*

könyvelését, és csökkentenie kell a riasztás számlálóját. Ezek az utasítások azonban igen nagy körültekintéssel lettek összeállítva, hogy nagyon gyorsak legyenek, hiszen minden egyes másodpercben igen sokszor végrehajtásra kerülnek.

Az operációs rendszer egyes részei is igényelnek időzítőket. Ezeket **felügyeleti időzítőknek (watchdog timer)** nevezzük. A merevlemezes egység meghajtóprogramjának vizsgálatakor látni fogjuk, hogy a lemezvezérlőnek küldött paranccsal egy időben egy ébresztéses időzítő is elindul, hogy akkor is lehetőség legyen a helyreállításra, ha a parancs végrehajtása teljesen sikertelen. A hajlékonylemezes meghajtók időzítő segítségével várakoznak arra, hogy a lemezegység motorja elérje a megfelelő sebességet, illetve hasonló módon állítják le a motort, ha bizonyos ideig nem történik semmilyen tevékenység. Néhány mozgatható nyomtatófejes nyomtató 120 karakter/másodperces sebességgel tud nyomtatni (ez 8,3 ezred másodperc karakterenként), de a nyomtatófejet nem tudja a bal margóra visszavinni 8,3 ezred másodperc alatt, ezért a meghajtóprogramnak várnia kell a kocsi vissza karakter kiírása után.

Az időzítőmeghajtó a felügyeleti időzítőket pontosan úgy kezeli, mint a felhasználói riasztásokat. Az egyetlen különbség, hogy az időzítő lejártakor nem egy szignál keletkezik, hanem a meghajtó meghív egy eljárást, amelyet a hívó bocsátott rendelkezésére. Ez az eljárás a hívó kódjának része. A MINIX 3 tervezésekor ez problémát jelentett, mert az egyik cél a meghajtóprogramok eltávolítása volt a kernel címteréről. Röviden úgy lehet összefoglalni, hogy a kernelszinten lévő rendszertaszk beállíthat riasztást felhasználói processzusok számára, majd az idő lejártakor értesítést küld nekik. Később még jobban kifejtjük ezt a témát.

Az utolsó tétel a listánkon a futásidő-elemzés (profiling). Néhány operációs rendszer lehetőséget ad arra, hogy a felhasználói program futásáról az utasításmutató értékeit tartalmazó hisztogramot készítsünk, ezáltal láthatóvá téve azt, hogy a program egyes részein mennyi időt töltött futás közben. Amikor a futásidő-elemzés lehetősége adott, akkor minden egyes órajelnél az időzítőmeghajtó megnézi, hogy az aktuális processzus kérte-e az adatgyűjtést. Ha igen, akkor megállapítja, hogy utasításmutató melyik címtartományban van, majd a címtartományhoz tartozó számlálóértéket eggyel növeli. Ezt az eljárást természetesen fel lehet használni a rendszer futásidő-elemzésére is.

2.8.3. A MINIX 3-időzítőmeghajtó áttekintése

A MINIX 3-időzítőmeghajtó a *kernel/clock.c* fájlban található. Három funkcionális részre különíthető el. Először is a következő fejezet eszközmeghajtóihoz hasonlóan van egy taszkfunkciója, vagyis egy ciklust futtat, amelyben kérésekre várakozik, majd a beérkező üzeneteket továbbítja feldolgozó szubrutinokhoz. Ez a szerkezet azonban majdhogynem csökevényes az időzítőtaszkban. Az üzenetek költségesek, mert mindig környezetátkapcsolással járnak. Ezért az időzítőtaszk esetében ezt csak akkor használjuk, ha tekintélyes mennyiségű munkát kell elvégezni. Csak egyetlen fajta üzenetet fogad el, emiatt csak egy kiszolgáló szubrutinja van, munkája végeztével pedig nem küld válaszüzenetet.

Az időzítőszoftver második nagyobb része a másodpercenként 60-szor aktivizálódó megszakításkezelő. A legszükségesebb nyilvántartási feladatokat végzi el, növeli egy változó értékét, ami a betöltés óta eltelt órajeleket számlálja. Ezt összehasonlítja a következő riasztási idővel. Frissíti azokat a számlálókat is, amelyek az aktuális processzus időszeletéből elhasznált időt, illetve az általa felhasznált összes időt tárolják. Ha a megszakításkezelő azt veszi észre, hogy egy processzus elhasználta az időszeletét, vagy lejárt egy időzítő, akkor üzenetet generál, amely a taszkciklushoz kerül. Egyébként nem küld üzenetet. Az alapelv az, hogy az órajelek hatására a kezelő minél kevesebbet, minél gyorsabban végezzen el. A költséges főtaszk csak akkor aktivizálódik, ha tekintélyes mennyiségű munkát kell elvégezni.

Az időzítőmeghajtó harmadik általános része egy szubrutingyűjtemény, amelynek tagjai általános támogatást nyújtanak, de a megszakítások során sem a megszakításkezelő, sem a főtaszk nem hívja meg őket. Az egyik szubrutin *PRIVATE,* és a taszk hívja, mielőtt belép a főciklusba. Ez inicializálja az időzítőt, vagyis az időzítőlapkára olyan adatokat ír, hogy az a kívánt időközönként megszakításokat generáljon. Az inicializációs rutin a megszakításkezelő címét is megfelelően elhelyezi, hogy a megszakításvezérlő megtalálja, amikor az időzítőlapka jelet küld neki az IRQ 8-as vonalon, végül engedélyezi a vonalat.

A *clock.c* többi szubrutinja *PUBLIC,* ezeket a kernelen belülről bárhonnan meg lehet hívni. Ami azt illeti, a *clock.c*-ből egyiket sem hívjuk. Többnyire a rendszertaszk hívja őket az idővel kapcsolatos rendszerhívások kiszolgálása közben. A szubrutinok között van például olyan, amelyik kiolvassa a betöltés óta eltelt időt nyilvántartó számlálót, egy másik az órajel-felbontású időzítéshez használható, vagy egy olyan, amelyik egyenesen az időzítőlapka egyik regiszterét olvassa ki, hogy ezred másodperces felbontású időzítés is lehetséges legyen. Időzítők beállítására és leállítására is vannak szubrutinok. Végül van egy olyan, amelyet a MINIX 3 leállításakor kell meghívni. Ez visszaállítja a hardveridőzítési értékeket a BIOS igényei szerint.

Az időzítőtaszk

Az időzítőtaszk csak egy fajta üzenetet fogad el, ez a *HARD_INT,* amelyik a megszakításkezelőtől érkezik. Minden más hibát jelent. Továbbá ezt az üzenetet nem kapja meg minden órajel után, annak ellenére, hogy az üzenet vétele után meghívott szubrutin neve *do_clocktick*. Csak akkor érkezik üzenet, és ezért a *do_clocktick* is csak akkor fut le, ha processzusütemezésre van szükség, vagy lejárt egy időzítő.

Az időzítő megszakításkezelője

A megszakításkezelő akkor aktivizálódik, amikor az időzítőlapka elszámol nulláig és megszakítást generál. Itt történik a legszükségesebb nyilvántartási feladatok elvégzése. A MINIX 3-ban az idő nyilvántartása a 2.48.(c) ábra szerint történik. A *clock.c*-ben azonban csak a betöltés óta eltelt órajelek számlálóját

tartjuk karban, a betöltéskori idő máshol van feljegyezve. Az időzítőszoftver csak az órajelszámláló aktuális értékével járul hozzá a valós idővel kapcsolatos rendszerhívásokhoz, további feldolgozást az egyik szerver végez. Ez megfelel annak a törekvésnek, hogy a MINIX 3-ban minél több funkcionahtást felhasználói szintű processzusokba helyezzünk át.

A megszakításkezelőben a lokális számlálót minden megszakításkor aktualizáljuk. Az órajelek elvesznek, amikor a megszakítások le vannak tiltva. Bizonyos esetekben ezt az effektust lehet korrigálni. Van egy globális változó, amely az elveszett órajeleket számlálja, ennek az értékét minden megszakításkor hozzáadjuk a főszámlálóhoz, majd lenullázzuk. A megvalósítás leírásánál fogunk látni egy példát erre.

A megszakításkezelő a processzustábla változóit is befolyásolja, elszámolási és vezérlési szempontból is. Az időzítőtaszk csak akkor kap üzenetet, ha az aktuális idő meghaladja a következőnek beütemezett riasztás idejét, vagy ha az aktuális processzus időszelete letelt. A megszakításkezelőben minden egyszerű egész műveletekkel megoldható – alapműveletek, összehasonlítás, logikai ÉS/VAGY, értékadások. Ezeket a C fordítóprogram könnyűszerrel hatékony gépi utasításokká alakítja. A legrosszabb esetben 5 összeadás/kivonás, 6 összehasonlítás, valamint néhány logikai művelet és értékadás elvégzésére van szükség egy megszakítás kiszolgálásához. Szubrutinhívásra egyáltalán nincs szükség.

Felügyeleti időzítők

Néhány oldallal ezelőtt függőben hagytuk azt a kérdést, hogy a felhasználói processzusoknak hogyan biztosíthatunk felügyeleti időzítőket, amelyeket rendszerint úgy képzelünk el, mint a felhasználó által megadott olyan eljárásokat, amelyek a felhasználó programjának részei, és egy időzítő lejártakor hajtódnak végre. Az világos, hogy ilyet a MINIX 3-ban nem lehet. Ellenben **szinkron riasztás** lehetséges, ezzel áthidalhatjuk a kernelszint és a felhasználói szint közötti szakadékot.

Most van itt az ideje, hogy elmagyarázzuk, mit értünk szinkron riasztás alatt. Egy szignál bármikor érkezhet, illetve egy hagyományos felügyeleti időzítő bármikor aktivizálódhat, függetlenül attól, hogy a program melyik részén van éppen a vezérlés. Ezért mondjuk, hogy ezek **aszinkron** módon működnek. Egy szinkron riasztás üzenetben érkezik, ezért csak akkor jut el a címzetthez, amikor az egy receive-et hajt végre. Azért nevezzük szinkronnak, mert csak akkor kerül a címzetthez, amikor az számít rá. Ha az értesítéses (notify) módszert alkalmazzuk a riasztásra, akkor a küldőnek nem kell blokkolódnia, a fogadónak pedig nem aggódnia azért, hogy lemarad a riasztásról. A notify üzenetet a rendszer eltárolja, ha a címzett éppen nem arra várakozik. Egy bittérképet használ erre a célra, amelyben minden bit egy lehetséges forrást jelöl.

A felügyeleti időzítők a *priv* tábla minden egyes elemében megtalálható *timer_t* típusú *s_alarm_timer* mezőt használják. Minden rendszerprocesszushoz tartozik egy bejegyzés a *priv* táblában. Egy felhasználói szinten futó rendszerprocesszus a sys_setalarm hívással állítja be az időzítőt; a hívást a rendszertaszk kezeli. A rend-

szertaszk a kernelbe van fordítva, ezért végre tudja hajtani az időzítő inicializálását a hívó számára. Az inicializáció abból áll, hogy az időzítő lejártakor meghívandó eljárás címét a megfelelő mezőben eltároljuk, majd az időzítőt felfűzzük az időzítők sorába, ahogy a 2.49. ábrán látható.

A végrehajtandó eljárásnak természetesen a kernel címtartományában kell lennie. Ez nem jelent problémát. A rendszertaszkban van egy függvény erre a célra; ez a *cause_alarm,* amely lefutásakor egy értesítést generál, ezzel szinkron riasztást ad a felhasználónak. Ez a riasztás kiválthatja a felhasználói eljárás meghívását. A kernelen belül ez egy valódi felügyeleti időzítő eljárás, de a riasztást kérő processzus számára egy szinkron riasztás. Ez nem ugyanaz, mintha az időzítő hívna meg egy eljárást a kérő címtartományában. Egy kis többletmunkával jár, de egyszerűbb, mint egy megszakítás.

Az előbb leírt módszer nem működik minden felhasználói szintű processzusra, csak a rendszerprocesszusokra. Minden rendszerprocesszusnak van egy saját példánya a *priv* struktúrából, de a felhasználói processzusok egyetlen példányon osztoznak. A struktúra nem megosztható részét, így a függőben lévő értesítések bittérképét és az időzítőt a felhasználói processzusok nem használhatják. A megoldás a következő: a processzuskezelő kezeli a felhasználói processzusok időzítőit ahhoz hasonlóan, ahogy a rendszertaszk teszi ezt a rendszerprocesszus számára. Minden processzusnak van egy *timer_t* típusú mezője a processzustábla azon részében, amelyet a processzuskezelő tart nyilván.

Amikor egy felhasználói processzus az alarm rendszerhívással kezdeményezi egy időzítő beállítását, akkor a processzuskezelő fog eljárni az érdekében, beállít egy időzítőt, és elhelyezi a saját listájába. A processzuskezelő megkéri a rendszertaszkot, hogy küldjön neki értesítést, amikor a listában lévő első időzítő lejárt. A processzuskezelőnek csak akkor kell segítséget kérnie, amikor az időzítőlistájának legelső eleme megváltozik. Ez előfordulhat amiatt, mert a legelső időzítő lejárt vagy törölték, vagy azért, mert egy új riasztási kérelem miatt új elemet kell beszúrni az addigi első elé. A szabványos POSIX-rendszerhívást ezzel a módszerrel támogatja a MINIX 3. A végrehajtandó eljárás a processzuskezelő címtartományában van. Amikor aktivizálódik, akkor a riasztást kérő felhasználói processzusnak egy szignált küld, nem pedig értesítést.

Ezred másodperces időzítés

A *clock.c*-ben van egy eljárás, amely milliomod másodperc felbontású időzítést tesz lehetővé. Különböző I/O-eszközöknél szükség lehet akár néhány milliomod másodperc rövidségű késleltetésekre is. Ezt a gyakorlatban a riasztások vagy az üzenetküldési felület használatával nem lehet megoldani. Az időzítőmegszakítások generálásához használt számlálót közvetlenül is ki lehet olvasni. Megközelítőleg 0,8 milliomod másodpercenként csökken eggyel, másodpercenként 60-szor fut le nulláig, vagyis 16,67 ezred másodpercenként. Ahhoz, hogy I/O-időzítéshez felhasználható legyen, egy kernelszinten futó eljárásnak folyamatosan figyelnie kellene, de sokat dolgoztunk azon, hogy az eszközmeghajtókat eltávolítsuk a kernelszintről. Jelenleg ezt a függvényt csak arra használjuk, hogy a véletlenszámgenerátor kezdőértéke is véletlenszerű legyen. Nagyon gyors számítógépen több mindenre is lehetne használni, de ez egy jövőbeli projekt.

Az időzítőszolgáltatások összefoglalása

A 2.50. ábra összefoglalja a *clock.c* által közvetlenül vagy közvetetten nyújtott különféle szolgáltatásokat. Sok *PUBLIC*-ként deklarált függvény van, amelyet a kernel vagy a rendszertaszk közvetlenül hívhat. A többi szolgáltatás csak közvetetten áll rendelkezésre, rendszerhívások révén, amelyeket végül a rendszertaszk hajt végre. A többi rendszerprocesszus fordulhat a rendszertaszkhoz közvetlenül, de a felhasználói processzusoknak kérést kell küldeniük a processzuskezelőhöz, amely szintén a rendszertaszkra támaszkodik.

A kernel és a rendszertaszk lekérdezheti az indulás óta eltelt időt, vagy beállíthat/leállíthat időzítőt az üzenetküldéssel járó többletmunka nélkül. A kernel és a rendszertaszk hívhatja a *read_clock*-ot is, amely az időzítőlapka számlálóját olvassa ki; ezzel megközelítőleg 0,8 milliomod másodperces egységekben lehet az időt mérni. A *clock_stop* függvényt a MINIX 3 leállásakor kell hívni, visszaállítja a BIOS-időzítőértékeket. Egy rendszerprocesszus, akár eszközmeghajtó, akár szerver, kérhet szinkron riasztást, aminek hatására egy kernelterületen lévő eljárás aktivizálódik, és értesítést küld a kérelmező processzusnak. POSIX-riasztást a felhasználói processzusok a processzuskezelőtől kérnek, amely továbbítja a kérést a rendszertaszknak. Az időzítő lejártakor a rendszertaszk értesíti a processzuskezelőt, az pedig szignált küld a felhasználói processzusnak.

Szolgáltatás	Elérés módja	Válasz	Kliensek
get_uptime	Függvényhívás	Órajelek száma	Kernel vagy rendszertaszk
set_timer	Függvényhívás	Nincs	Kernel vagy rendszertaszk
reset_timer	Függvényhívás	Nincs	Kernel vagy rendszertaszk
read_clock	Függvényhívás	Számláló	Kernel vagy rendszertaszk
clock_stop	Függvényhívás	Nincs	Kernel vagy rendszertaszk
Szinkron riasztás	Rendszerhívás	Értesítés	Szerver vagy eszközmeghajtó, a rendszertaszkon keresztül
POSIX-riasztás	Rendszerhívás	Szignál	Felhasználói processzus, a processzuskezelőn keresztül
Idő	Rendszerhívás	Üzenet	Bármelyik processzus, a processzuskezelőn keresztül

2.50. ábra. *Az időzítőmeghajtó által nyújtott, idővel kapcsolatos szolgáltatások*

2.8.4. A MINIX 3-időzítőmeghajtó megvalósítása

Az időzítőtaszk nem használ nagyobb adatstruktúrákat, ehelyett néhány változóban követi nyomon az idő változását. A *realtime* változó (10462. sor) alapvető – ez számlálja az órajeleket. Egy globális változó, a *lost_ticks,* a *glo.h*-ban van definiálva (5333. sor). Ezt a változót minden olyan kernelszinten futó függvény használhatja, amely elég sokáig letiltja a megszakításokat ahhoz, hogy egy vagy több órajel elvesszen. Ezt a változót jelenleg csak az *int86* függvény használja a *klib386.s*-ből. Az *int86* a betöltési felügyelőprogramot használja arra, hogy a vezérlést átadja a BIOS-nak, a felügyelőprogram pedig az ecx regiszterben visszaadja, hogy a BIOS-hívás lefutásához hány órajelre volt szükség. Ez azért lehetséges, mert bár a MINIX 3-időzítő megszakításkezelője nem működik a BIOS-hívás alatt, a betöltési felügyelőprogram nyilván tudja tartani az órajeleket a BIOS segítségével.

Az időzítőmeghajtó több más globális változót is használ, többek között a *proc_ptr, prev_ptr* és a *bill_ptr* mutatókat, hogy hozzáférjen az általuk azonosított processzusok processzustábla-bejegyzéseihez. Ezekben a bejegyzésekben több mezőhöz is hozzányúl; ezek között van a *p_user_time* és a *p_sys_time* az elszámoláshoz, valamint a *p_tlcks_left* az időszeletből megmaradt idő nyilvántartásához.

A MINIX 3 indulásakor minden eszközmeghajtó meghívásra kerül. Ezek legtöbbje végrehajt némi inicializációt, majd üzenetre várva blokkolódik. Az időzítőmeghajtó, a *clock_task* (10468. sor) is így tesz. Először meghívja az *init_clock*-ot, hogy 60 Hz-re inicializálja a programozható óra frekvenciáját. Amikor üzenetet kap, akkor meghívja a *do_clocktick* függvényt, ha az üzenet *HARD_INT* (10486. sor) volt. Másfajta üzenetre nincs felkészülve, azokat hibaként kezeli. A *do_clocktick* (10497. sor) nem hívódik meg minden egyes órajelre, tehát a neve nem adja pontos leírását a funkciójának. Csak akkor kerül meghívásra, ha a megszakításkezelő szerint valamilyen fontos tevékenységet kell elvégezni.

A *do_clocktick* (10497. sor) futását eredményezheti az, ha az éppen futó processzusnak letelt az időszelete. Ha a processzus megszakítható (a rendszertaszk és az időzítőtaszk nem az), akkor először meghívja a *lock_dequeue*-t, majd rögtön utána a *lock_enqueue*-t (10510–10512. sor), aminek az a hatása, hogy a processzust leveszi az ütemezési sor elejéről, majd futtathatóvá teszi és újraütemezi. A *do_clocktick* aktiválását eredményezi még az is, ha letelik egy felügyeleti időzítő. Időzítők és időzítők láncolt listái olyan sokszor fordulnak elő a MINIX 3-ban, hogy kisegítőfüggvényeikből egy könyvtárat hoztunk létre. Ezek közül a 10517. sorban hívott *tmrs_exptimers* meghívja a lejárt időzítőkhöz tartozó eljárásokat és deaktiválja őket.

Az *init_clock* (10529. sor) csak egyszer hívódik meg, amikor az időzítőtaszk elindul. Sok olyan pont van, amire rámutathatunk, és azt mondhatjuk: „Itt kezd el futni a MINIX 3." Ez a pont is esélyes jelölt; hiszen az időzítő alapvető fontosságú egy megszakítható ütemezéses többfeladatos rendszerben. Az *init_clock* három bájtot ír ki az időzítőlapkára, amivel beállítja az üzemmódot és a megfelelő értéket az elsődleges regiszterbe. Ezután bejegyezteti a processzusszámát, IRQ-számát és a kezelő címét, hogy a megszakítások a megfelelő helyre érkezzenek. Végül engedélyezi a megszakításvezérlő lapkát, és elkezdi fogadni a megszakításokat.

A *clock_stop* visszaállítja az időzítőlapkát az inicializálás előtti állapotba. Deklarációja *PUBLIC,* és a *clock.c*-ből sehonnan sem hívjuk meg. A *clock_init*-tel való egyértelmű kapcsolata miatt került ide. Csak a MINIX 3 leállásakor hívja meg a rendszertaszk, mielőtt a betöltési felügyelőprogramnak visszaadná a vezérlést.

Ahogy (pontosabban 16,67 ezred másodperccel azután, hogy) az *init_clock* lefutott, beérkezik az első időzítőmegszakítás, majd ezek másodpercenként 60-szor ismétlődnek egészen addig, amíg a MINIX 3 fut. A *clock_handler* (10556. sor) programja valószínűleg többször fut, mint a MINIX 3-rendszer bármelyik másik része. Következésképpen a *clock_handler* esetében a sebesség áll mindenek felett. Egyedül a 10586. sorban vannak szubrutinhívások. Csak akkor van rájuk szükség, ha egy elavult IBM PS/2-es rendszeren futunk. Az (órajelekben mért) aktuális idő frissítése a 10589. és a 10591. sor között található. Ezután a felhasználói és a számlázási idők kerülnek frissítésre.

A kezelő tervezésekor megkérdőjelezhető döntések születtek. A 10610. sorban két ellenőrzést végzünk, és ha bármelyik feltétel teljesül, akkor az időzítőtaszknak értesítést küldünk. Az időzítőtaszk által hívott *do_clockticks* függvény megismétli mindkét ellenőrzést, hogy eldöntse, mit kell tennie. Erre azért van szükség, mert a kezelő által használt notify híváson keresztül nem adhatunk át semmilyen többletinformációt, ami alapján a két esetet meg lehetne különböztetni. Az olvasót arra biztatjuk, hogy fontoljon meg egyéb alternatívákat és ezek kiértékelésének lehetséges módját.

A *clock.c* maradék része a már említett kisegítőfüggvényeket tartalmazza. A *get_uptime* (10620. sor) egyszerűen a *realtime* értékét adja vissza, ami csak a *clock.c* függvényei számára látható. A *set_timer* és a *reset_timer* a könyvtár más függvényeit használja, amelyek elrejtik az időzítőlisták kezelésének részleteit. Végül a *read_clock* kiolvassa és visszaadja az időzítőlapka számlálóregiszterében lévő aktuális értéket.

2.9. Összefoglalás

A megszakítások elfedésére az operációs rendszerek a párhuzamosan futó szekvenciális processzusok modelljét alkalmazzák. A processzusok különféle processzusok közötti kommunikációs mechanizmusok segítségével kommunikálhatnak egymással, mint például szemaforok, monitorok vagy üzenetek. Ezeket arra használjuk, hogy két processzus soha ne legyen egyszerre a kritikus szakaszában. Egy processzus lehet futó, futásra kész vagy blokkolt állapotban, és akkor kerülhet át egyik állapotból a másikba, ha önmaga vagy egy másik processzus végrehajt egy kommunikációs alapműveletet.

A processzusok közötti kommunikációs alapműveletek olyan problémák megoldására használhatók, mint például a gyártó-fogyasztó, az étkező filozófusok vagy az olvasók és írók. Még ezek használata esetén is óvatosnak kell lennünk, hogy elkerüljük a hibákat és holtponti helyzeteket. Számos ütemezési algoritmus ismert,

például round robin, prioritásos, többszintű várakozósoros, illetve elv által vezérelt ütemezők.

A MINIX 3 támogatja a processzusmodellt, és üzeneteket használ a processzusok közötti kommunikációra. Az üzeneteket nem puffereljük, így a send csak akkor sikeres, ha a fogadó már várakozik az üzenetre. Hasonlóan, a receive csak akkor sikeres, ha egy üzenet már rendelkezésre áll. Ha bármelyik művelet nem sikeres, akkor a hívó processzus blokkolódik. A MINIX 3 az üzenetek mellett a nem blokkoló notify alapműveletet is támogatja. Ha olyan címzettnek akarunk értesítést küldeni, amely éppen nem várakozik üzenetre, akkor ez egy bit beállítását eredményezi, aminek hatására a legközelebbi receive az értesítést kézbesíti a fogadónak.

Az üzenetek kezelésének illusztrálására tekintsünk egy felhasználót, aki read műveletet akar végezni. A felhasználói processzus üzenetet küld a fájlrendszernek, kérve az adatokat. Ha az adatok nincsenek a fájlrendszer gyorsítótárában, akkor a fájlrendszer kérést küld a lemezt kezelő eszközmeghajtónak, hogy olvassa be a lemezről. Ezután a fájlrendszer blokkolódik és vár az adatokra. A lemezmegszakítás beérkezésekor a rendszertaszk értesítést kap, így válaszolni tud a lemezegység eszközmeghajtójának, az pedig válaszol a fájlrendszernek. Ezen a ponton a fájlrendszer kéri a rendszertaszkot, hogy a gyorsítótárából az újonnan beolvasott adatblokkot másolja át a felhasználóhoz. Ezeket a lépéseket a 2.46. ábrán szemléltettük.

Megszakítás után processzusváltás következhet be. Ha egy processzus futása megszakad, akkor egy verem jön létre a hozzá tartozó processzustábla-bejegyzésben, és a folytatásához szükséges minden információ ebbe a verembe kerül. Bármelyik processzust újra futtatható állapotba lehet hozni, ha a veremmutatót a processzustábla bejegyzésére állítjuk, újratöltjük a regisztereket, majd végrehajtunk egy iretd utasítást. Azt az ütemező dönti el, hogy melyik processzustábla-bejegyzés címe kerül a veremmutatóba.

A kernel futása közben nem történhetnek megszakítások. A kernel futása közben bekövetkezett kivétel a processzustáblában lévő helyett a kernelvermet használja. Megszakítás kiszolgálása után egy processzus futhat tovább.

A MINIX 3 ütemezési algoritmusa prioritásos várakozási sorokat használ. A rendszerprocesszusok általában a legmagasabb prioritású sorokban futnak, a felhasználói processzusok pedig az alacsonyabb prioritású sorokban, de a prioritások egyedileg kerülnek kiosztásra. Egy hosszú ciklusban bennragadt processzus prioritása ideiglenesen csökkenhet, de visszakerülhet az eredeti helyére, ha más processzusok is lehetőséget kaptak a futásra. A *nice* paranccsal bizonyos keretek között megváltoztatható a processzusok prioritása. A processzusok round robin módszerrel követik egymást, alkalmanként mindegyik egy időszeletet kap, amelynek hossza függhet a processzustól. Ha egy processzus blokkolódás után újra futtatható állapotba kerül, akkor az időszeletének maradék részével a sora elejére kerül vissza. Ennek az a célja, hogy az I/O-műveleteket végző processzusok gyorsabban tudjanak reagálni. Az eszközmeghajtók és a szerverek hosszú időszeletet kapnak, mert várhatóan úgyis blokkolódásig futnak. Azonban még a rendszerprocesszusok is megszakíthatók, ha túl hosszú ideig futnak.

A kernel tárgykódú állománya tartalmaz egy rendszertaszkot, amely lehetővé teszi, hogy a felhasználói szintű processzusok kommunikáljanak a kernellel. Támogatja az eszközmeghajtókat és a szervereket azáltal, hogy számukra privilegizált műveleteket hajt végre. A MINIX 3-ban az időzítőtaszk is a kernelbe van fordítva. Nem a hagyományos értelemben vett eszközmeghajtó. A felhasználói szintű processzusok nem érhetik el az időzítőt eszközként.

Feladatok

1. A multiprogramozás miért központi jelentőségű egy modern operációs rendszer működéséhez?
2. Melyik három fő állapotban lehet egy processzus? Írja körül röviden mindegyiket.
3. Tételezzük fel, hogy egy olyan fejlett számítógép-architektúrát kell tervezni, amelyben a processzusváltást megszakítások helyett a hardver végezné. Milyen információra lenne szüksége a CPU-nak? Vázolja fel, hogyan működhetne a hardver-processzusváltás.
4. A megszakításkezelő rutinok minden mai számítógépen legalább részben assembly nyelven vannak írva. Miért?
5. Rajzolja újra a 2.2. ábrát úgy, hogy hozzáad két új állapotot: Új és Befejezett. Egy processzus létrehozásakor az Új állapotba kerül. Amikor kilép, akkor a Befejezett állapotba kerül.
6. A könyvben azt állítottuk, hogy a 2.6.(a) ábrán látható modell nem megfelelő gyorsítótárral ellátott fájlszerver számára. Miért nem? Lehetne minden processzusnak saját gyorsítótára?
7. Mi az alapvető különbség processzus és szál között?
8. Egy szálakat használó rendszerben a szálaknak is külön vermük van, vagy csak a processzusoknak? Indokolja meg a választ.
9. Mi az a versenyhelyzet?
10. Adjon példát versenyhelyzetre, ami akkor léphet fel, ha két együtt utazó ember számára akarunk repülőjegyet venni.
11. Írjon egy olyan parancsértelmező programot, amely egy számsort tartalmazó állomány utolsó elemét kiolvassa, hozzáad egyet, majd az így kapott számot az állomány végéhez hozzáfűzi. Futtassa a program egy példányát a háttérben, egyet pedig az előtérben, miközben mind a kettő ugyanazt az állományt próbálja módosítani. Mennyi idő kell ahhoz, hogy versenyhelyzet jelei mutatkozzanak? Hol van a kritikus szakasz? Módosítsa a programot úgy, hogy megszűnjön a versenyhelyzet. (Tanács: használja az

```
ln file file.lock
```

utasítást az adatállomány zárolásához.)

12. Hatékony módja-e a zárolásnak az

    In file file.lock

    módszer az előző példában szereplő programhoz hasonló helyzetekben? Miért, vagy miért nem?
13. Működik-e a *turn* változót használó tevékeny várakozásos megoldás (lásd 2.10. ábra), ha a két processzus egy közös memóriás többprocesszoros rendszeren fut, azaz két CPU van, és közös memória.
14. Tekintsünk egy olyan gépet, amelynek nincs TEST AND SET LOCK utasítása, de van egy olyan utasítása, amely egy oszthatatlan művelettel megcseréli egy regiszter és egy memóriaszó tartalmát. Használható-e ez a 2.12. ábrán látható *enter_region* rutin megvalósítására?
15. Vázolja fel, hogyan tudná a szemaforokat megvalósítani egy, a megszakításokat letiltani képes operációs rendszer.
16. Mutassa meg, hogyan lehet megvalósítani bináris szemaforok és közönséges gépi utasítások felhasználásával az egész értékű szemaforokat (vagyis olyan szemaforokat, amelyek tetszőlegesen nagy értéket tárolnak).
17. A 2.2.4. alfejezetben leírtunk egy szituációt, amelyben egy magas prioritású H processzus és egy alacsony prioritású L processzus szerepelt, és az események oda vezettek, hogy H végtelen ciklusban tartotta a processzort. Ugyanebbe a problémába ütköznénk-e, ha a prioritásos helyett round robin ütemezést használnánk? Indokolja meg a választ.
18. A monitorokban az állapotváltozók és két speciális művelet, a WAIT és a SIGNAL szolgál a szinkronizáció megvalósítására. Általánosabb megoldás lenne egyetlen WAITUNTIL művelet bevezetése, amelynek tetszőleges logikai kifejezés lehetne a paramétere. Így például írhatnánk, hogy

    WAITUNTIL $x < 0$ **or** $y + z < n$

    A SIGNAL műveletre sem lenne szükség. Ez a séma egyértelműen általánosabb, mint a Hoare és Brinch Hansen-féle megoldás, de mégse használják. Miért nem? (Tanács: gondoljon a megvalósításra.)
19. Egy gyorsétkezdének négyféle alkalmazottja van: (1) rendelésfelvevő, aki a vendégek rendelését felveszi; (2) szakács, aki elkészíti az ételeket; (3) csomagoló szakember, aki az ételeket becsomagolja és (4) pénztáros, aki a csomagokat a vendégeknek adja és beszedi a pénzt. Minden alkalmazott egy kommunikáló szekvenciális processzusnak tekinthető. Milyen kommunikációt alkalmaznak ezek az alkalmazottak? Hasonlítsa össze ezt a modellt a MINIX 3 processzusaival.
20. Tegyük fel, hogy van egy üzenetküldéses, postaládákat használó rendszerünk. A processzusok nem blokkolódnak, ha egy tele ládába küldenek, vagy egy üresből akarnak üzenetet kivenni. Ehelyett egy hibakódot kapnak vissza. A processzusok a hibakódra úgy reagálnak, hogy újra és újra próbálkoznak, amíg a művelet sikeres nem lesz. Vezethet-e ez a séma versenyhelyzetekhez?

21. Miért rendeljük az étkező filozófusok megoldásában (lásd 2.20. ábra) a *take_forks* eljárásban az állapotváltozóhoz a *HUNGRY* értéket?
22. Tekintsük a 2.20. ábra *put_forks* eljárását. Tegyük fel, hogy a *state*[*i*] változó a *THINKING* értéket a két *test* hívás *után* kapja meg, nem *előtte*. Hogy változtatná ez meg a megoldást 3 filozófus esetén? 100 filozófus esetén?
23. Az olvasók és írók probléma sokféleképpen megfogalmazható attól függően, hogy melyik processzuscsoport mikor indítható. Írja le körültekintően a probléma három változatát, mindegyikben kitüntetett (vagy alárendelt) szerepet szánva az egyik csoportnak. Minden variáns esetén adja meg, hogy mi történik, amikor egy olvasó vagy egy író kész hozzáférni az adatbázishoz, illetve amikor nem akarja tovább használni az adatbázist.
24. A CDC 6600 számítógép egyszerre maximum 10 I/O-processzust volt képes kezelni egy érdekes, ún. **processzormegosztásos** round robin ütemezéssel. Minden utasítás után processzusváltás történt, így az első végrehajtott utasítás az első processzusé volt, a második utasítás a második processzusé, és így tovább. A processzusváltást speciális hardver végezte, ez nem jelentett többletterhelést. Ha egy processzusnak T másodpercre volna szüksége a lefutáshoz versenytársak hiányában, mennyi időre volna szüksége processzormegosztást alkalmazva, n processzus jelenlétével?
25. A round robin ütemezők általában egy listában tartják nyilván a futtatható processzusokat; minden processzus pontosan egyszer fordul elő a listában. Mi történne, ha egy processzus kétszer is benne lenne a listában? El tud képzelni olyan okot, amiért ezt érdemes lenne megengedni?
26. Egy bizonyos rendszerben végzett mérések azt mutatták, hogy egy átlagos processzus T ideig fut, mielőtt egy I/O-művelet miatt blokkolódna. Egy processzusváltás S időegységet igényel; ez tulajdonképpen veszteség. Egy Q időegységet használó round robin ütemezés esetén adja meg képlettel a CPU hatékonyságát az alábbi esetekben:
    (a) $Q = \infty$
    (b) $Q > T$
    (c) $S < Q < T$
    (d) $Q = S$
    (e) Q majdnem 0
27. Öt program vár futásra. Becsült futásidejük 9, 6, 3, 5 és X. Milyen sorrendben kell őket végrehajtani, hogy az átlagos válaszidő a legkisebb legyen? (A válasz függ X-től.)
28. Öt kötegelt program (A-tól E-ig) érkezik a számítóközpontba szinte teljesen egy időben. Becsült futásidejük 10, 6, 2, 4 és 8 perc. Az előre megállapított prioritásaik sorrendben 3, 5, 2, 1 és 4. A legnagyobb prioritás az 5. Az alábbi ütemezési algoritmusok mindegyikére állapítsa meg az átlagos áthaladási időt.
    (a) round robin
    (b) prioritásos
    (c) érkezési sorrendben (most 10, 6, 2, 4, 8)
    (d) legrövidebb feladat először

Az (a) esetben tételezzük fel, hogy a rendszer multiprogramozott, és minden program egyenlően részesedik a CPU idejéből. A (b), (c) és (d) esetben tételezzük fel, hogy egyszerre csak egy program fut, de az addig, amíg befejeződik. Minden program csak a CPU-t használja.

29. Egy CTSS-en futó processzusnak 30 időszeletre van szüksége a lefutáshoz. Hányszor kell behozni lemezről, beleszámítva a legelsőt (mielőtt még elindult volna)?

30. Az $a = 1/2$ paraméteres öregedési algoritmust használjuk a futási idők megbecsülésére. Az előző négy futás, a legrégebbivel kezdve 40, 20, 40 és 15 ms. Mennyi a becslés a következő futásra?

31. A 2.25. ábrán láthattuk, hogy a háromszintű ütemezés hogyan működik kötegelt rendszerben. Lehetne ezt az ötletet alkalmazni olyan interaktív rendszerben, ahol nem érkeznek új feladatok? Hogyan?

32. Tegyük fel, hogy a 2.28.(a) ábrán látható szálak ebben a sorrendben futnak: egy az *A*-ból, egy a *B*-ből, egy az *A*-ból, egy a *B*-ből stb. Hány lehetséges szálsorozat van az első négy ütemezési eseményig?

33. Egy toleráns valós idejű rendszernek négy periodikus eseményt kell kezelnie, a periódusuk 35, 20, 10 és x ms CPU-idő. Mekkora az x legnagyobb értéke, amelyre a rendszer még beütemezhető?

34. Futás közben a MINIX 3 egy változót *(proc_ptr)* használ arra, hogy a futó processzushoz tartozó processzustábla-bejegyzésre mutasson. Miért?

35. A MINIX 3 nem puffereli az üzeneteket. Magyarázza meg, milyen problémákat okoz ez a tervezési elv az időzítő- és a billentyűzetmegszakítások esetén.

36. Amikor a MINIX 3-ban egy üzenetet küldünk egy alvó processzusnak, a *ready* eljárás a processzust a megfelelő ütemezési sorba teszi. Ez az eljárás a megszakítások letiltásával kezdődik. Miért?

37. A *mini_rec* MINIX 3-eljárás egy ciklust tartalmaz. Mire szolgál ez?

38. A MINIX 3 alapjában véve a 2.43. ábrán látható ütemezési módszert alkalmazza úgy, hogy az egyes osztályoknak különböző a prioritása. A legalsó osztály (felhasználói processzusok) round robin ütemezést használ, de a taszkok és a szerverek blokkolásig futhatnak. Lehetséges-e, hogy a legalsó osztály processzusai éhezzenek? Miért, vagy miért nem?

39. Alkalmas-e a MINIX 3 valós idejű alkalmazásokhoz, mint például az adatgyűjtés. Ha nem, hogyan lehetne erre alkalmassá tenni?

40. Tegyük fel, hogy van egy szemaforokat támogató operációs rendszerünk. Valósítson meg egy üzenetküldéses rendszert. Írjon az üzenetek küldésére és fogadására szolgáló eljárásokat.

41. Egy hallgató, akinek főszaka az antropológia, mellékszaka pedig a számítógép-tudomány, olyan kutatásba kezdett, amely azt hivatott vizsgálni, vajon az afrikai páviánoknak meg lehet-e tanítani a holtpont fogalmát. Keres egy mély kanyont, és kifeszít fölé egy kötelet, így a páviánok függeszkedve át tudnak jutni. Egyszerre több pávián is át tud menni, feltéve, hogy ugyanabba az irányba mennek. Ha kelet felé haladó és nyugat felé haladó páviánok kerülnek egyszerre a kötélre, akkor holtpont alakul ki (a páviánok beragadnak középen), mert nem tud egyik a másikon átjutni, miközben lógnak a kanyon fölött. Ha

egy pávián át akar kelni, körül kell néznie, hogy nem jön-e szembe egy társa. Írjon egy olyan, szemaforokat használó programot, amely elkerüli a holtponti helyzeteket. Ne törődjön azzal, hogy kelet felé haladó páviánok akármeddig feltarthatják a nyugat felé menőket.

42. Az előző feladat megoldását egészítse ki az éhezés elkerülésével. Ha egy kelet felé haladó pávián lát egy nyugat felé haladót a kötélen, akkor megvárja, míg az átér, de újabb nyugat felé haladó pávián nem kezdhet el mászni, amíg legalább egy át nem jött az ellenkező irányba.

43. Oldja meg az étkező filozófusok problémát szemaforok helyett monitorokkal.

44. Egészítse ki a MINIX 3-kernelt egy olyan programrésszel, amely számolja, hogy hány üzenetet küld az i processzus (vagy taszk) a j processzusnak (vagy taszknak). Az F4 billentyű lenyomására nyomtassa ki ezt a mátrixot.

45. Módosítsa a MINIX 3-ütemezőt úgy, hogy az tartsa nyilván, melyik felhasználói processzus mennyi CPU-időt kapott legutóbb. Amikor nincs futtatható taszk vagy szerver, válassza azt a felhasználói processzust, amely legutóbb a legkevesebb időt kapta.

46. Módosítsa a MINIX 3-at úgy, hogy minden processzus be tudja állítani a gyermekei prioritását egy új rendszerhívással. A rendszerhívás neve legyen setpriority, két paramétere pedig a gyermekprocesszus azonosítója és kívánt prioritása.

47. Módosítsa a *hwint_master* és *hwint_slave* makrókat az *mpx386.s* állományban úgy, hogy a *save* függvény utasításait a hívás helyére soronként beilleszti. Mennyivel növekedett meg a programkód mérete? Ki tud-e mutatni teljesítménynövekedést?

48. Futtassa le a sysenv parancsot a saját MINIX 3-rendszerén, és magyarázza el, hogy mit jelentenek az egyes tételek. Ha nincs hozzáférése MINIX 3-rendszerhez, akkor a 2.37. ábra tételeit magyarázza el.

49. A processzustábla inicializálásának tárgyalása közben említettük, hogy egyes C fordítóprogramok valamivel jobb kódot generálnak akkor, ha egy konstanst a tömbhöz adunk hozzá, nem az indexhez. Írjon néhány rövid C programot ennek a hipotézisnek az ellenőrzésére.

50. Módosítsa a MINIX 3-at úgy, hogy statisztikákat gyűjtsön arról, hogy ki kinek küld üzenetet. Írjon egy programot, amely összegyűjti és használható formában kinyomtatja a statisztikákat.

3. Bevitel/kivitel

Az operációs rendszer fő feladatai közül az egyik az I/O- (Input/Output – bevitel/kivitel) eszközök vezérlése. Az eszközök felé parancsokat kell kiadnia, kezelnie kell a megszakítási kéréseket és a hibákat. Továbbá gondoskodnia kell az eszközök és a rendszer többi része közötti kapcsolódási felületről, amelynek egyszerűnek és könnyen használhatónak kell lennie. A lehetőségek bővítése miatt, az eszközök közötti kapcsolatok megvalósítására egységes módszert kell biztosítania (eszközfüggetlen kapcsolat). A teljes operációs rendszernek egy jelentős hányadát az I/O-kód teszi ki. Az operációs rendszer megértéséhez szükséges az I/O működésének megismerése. Ennek a fejezetnek a tárgya, hogy miként támogatja az operációs rendszer az I/O-feladatok megoldását.

A fejezet felépítése a következő. Először néhány olyan elvet tekintünk át, amelyek szerint az I/O-hardver szervezve van. Ezt követően az I/O-szoftvert tárgyaljuk általánosan. Az I/O-szoftver rétegekre bontható, amelyek mindegyike egy-egy jól definiált feladatot lát el. Ezeket a rétegeket abból a szempontból vizsgáljuk, hogy mit tesznek, és hogyan illeszkednek egymáshoz.

A következő rész a holtpont problémával foglalkozik. Ebben definiáljuk a holtpont pontos fogalmát, megmutatjuk, hogyan állhat elő ez a probléma, adunk két módszert elemzésére, és megvizsgálunk néhány algoritmust, amelyek alkalmazásával megelőzhetők ennek a problémának az előfordulásai.

Ezután rátérünk a MINIX 3 vizsgálatára. Majd madártávlatból vetünk egy pillantást a MINIX 3 I/O részére, beleértve a megszakításokat, az eszközvezérlőket, az eszközfüggő és az eszközfüggetlen I/O-t. A bevezetésnek megfelelően számos I/O-eszközzel részletesen foglalkozunk: lemezek, billentyűzetek, megjelenítők. Mindegyik eszköznél megnézzük a hardvert és a szoftvert.

3.1. Az I/O-hardver alapjai

Az egyes emberek más-más módon nézik az I/O-hardvert. A villamosmérnökök a hardvert a fizikailag megtestesítő lapkák, vezetékek, erőforrások, motorok és egyéb fizikai komponensek szempontjából látják.

A programozók nézik a szoftver kapcsolódási felületeit – a parancsokat, amelyeket a hardver elfogad, a függvényeket, amelyeket végrehajt, és a hibákat, amelyeket visszajelezhet. Ebben a könyvben az I/O-eszközök programozásával foglalkozunk, nem a tervezésükkel, felépítésükkel vagy karbantartásukkal, vagyis érdeklődésünk a hardver programozására terjed ki, és nem érdekel bennünket a belső működésük. Mindazonáltal, sok I/O-eszköz programozása gyakran burkoltan kapcsolódik a belső működéséhez. A következő három szakaszban röviden kitérünk az I/O-hardver általános hátterére, mivel ez kapcsolódik a programozásukhoz.

3.1.1. I/O-eszközök

Az I/O-eszközöket nagyjából két csoportba sorolhatjuk: **blokkos eszközök** és **karakteres eszközök**. Blokkos eszközön olyan eszközt értünk, amely az információt adott méretű blokkokban tárolja, mindegyiket saját címmel. A szokásos blokkméret tartománya 512 bájt és 32 768 bájt között van. A blokkos eszközök lényeges tulajdonsága, hogy az egyes blokkok írhatók és olvashatók az összes többi blokktól függetlenül. A leggyakrabban használt blokkos eszköz a lemez.

Ha alaposabban megnézzük a határt a blokkonként címezhető, és az így nem címezhető eszközök között, akkor láthatjuk, hogy ez nem egy jól definiált fogalom. Mindenki egyetért abban, hogy a lemez egy blokkonként címezhető eszköz, mivel nem jelent problémát az olvasófej pillanatnyi helyzete, mindig lehetséges egy másik cilinder keresése, majd annak kivárása, hogy a kért blokk a fej alá forduljon. Most tekintsünk egy szalagmeghajtó egységet, amellyel biztonsági másolatot készíthetünk a lemezről. A szalagok blokkok sorozatából állnak. Ha a szalagmeghajtó egység kap egy olvasási parancsot az *N*-edik blokkra vonatkozóan, akkor először mindig visszacsévéli a szalagot az elejére, és utána mozgatja előre az *N*-edik blokkig. Ez a művelet a lemezen való kereséssel analóg, de annál sokkal hosszabb időt igényel. Szintén kérdéses, hogy egy szalag belső blokkjának újraírására van-e lehetőség, vagy sem. Még ha lehetséges lenne is a szalagot véletlen elérésű blokkos eszközként használni, az iménti probléma megoldása ennél többet kíván: hagyományosan nem használható ily módon.

Az I/O-eszközök másik típusát a karakteres eszközök alkotják. Egy karakteres eszköz vagy kibocsátja, vagy fogadja a karakterek sorozatát anélkül, hogy figyelembe venne bármilyen blokkszerkezetet. Az ilyen eszköz nem címezhető, és a keresési műveletet sem tudja végrehajtani. Nyomtatók, hálózati csatlakozási felületek, egerek (rámutatásra), digitalizáló táblák (pszichológiai laborkísérleteknél) és a legtöbb egyéb eszköz, amely nem lemez jellegű, karakteres eszköznek tekinthető.

Ez az osztályozási séma nem igazán jó. Bizonyos eszközök nem illeszkednek egyik osztályba sem. Például az órák nem címezhetők blokkonként. Nem generálnak, nem fogadnak karaktersorozatot. Mindössze annyit tesznek, hogy jól meghatározott időintervallumonként megszakításokat hoznak létre. Mégis, a blokkos és karakteres eszközök modellje elég általános ahhoz, hogy az operációs rendszer I/O-eszközfüggetlen szoftverjei közül néhánynak alapul szolgáljon. Például a fájl-

Eszköz	Adatátviteli sebesség
Billentyűzet	10 bájt/s
Egér	100 bájt/s
56 K modem	7 KB/s
Szkenner	400 KB/s
Digitális videokamera	4 MB/s
52x CD-ROM	8 MB/s
FireWire (IEEE 1394)	50 MB/s
USB 2.0	60 MB/s
XGA monitor	60 MB/s
SONET OC-12 hálózat	78 MB/s
Gigabit Ethernet	125 MB/s
Soros ATA merevlemez	200 MB/s
SCSI Ultrawide 4 merevlemez	320 MB/s
PCI-sín	528 MB/s

3.1. ábra. *Néhány gyakori eszköz, hálózat és sín adatátviteli sebessége*

rendszer absztrakt blokkos eszközökkel foglalkozik, és az eszközfüggő részt alacsonyabb szintű szoftverekre hagyja, az ún. **eszközmeghajtók**ra **(device driver).**

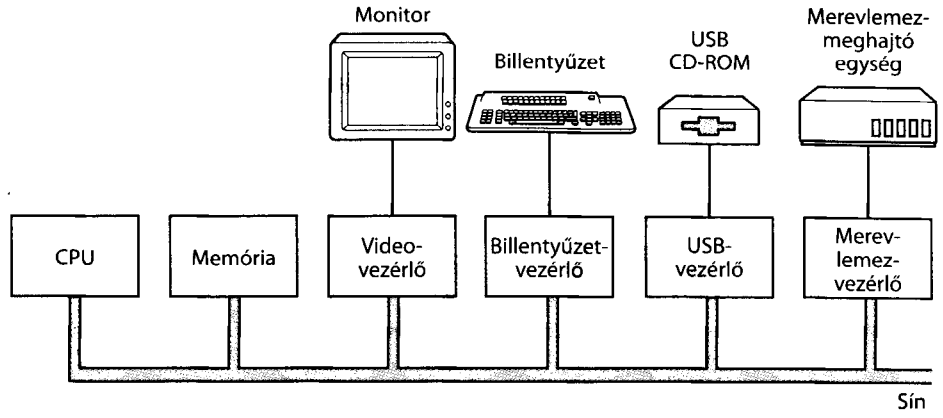
Az I/O-eszközök sebességi tartománya meglehetősen nagy, ami a szoftverek számára tekintélyes nehézséget okoz, mivel jelentősen különböző adatátviteli sebességnél kell jól működniük. A 3.1. ábra néhány közismert eszköz adatátviteli sebességét mutatja. Az idő múlásával az eszközök legtöbbje egyre gyorsabb.

3.1.2. Eszközvezérlők

Az I/O-egységek tipikusan mechanikus és elektromos összetevőkből épülnek fel. Gyakran lehetőség van arra, hogy a két részt különválasszuk, és ezzel egy moduláris-abb és általánosabb jellegű leírást adjunk. Az elektromos komponenst, **eszközvezérlő**nek vagy **adapter**nek nevezik. A személyi számítógépeknél ez gyakran egy nyomtatott áramköri lap alakjában jelenik meg, amely behelyezhető a bővítő nyílásba. A mechanikus komponens maga az eszköz. A 3.2. ábra mutatja ezt az elrendezést.

A vezérlőkártyán rendszerint van egy csatlakozó, amely kábellel felcsatolható az eszközhöz. Sok vezérlő kettő, négy vagy akár nyolc azonos eszközt is képes irányítani. Ha a vezérlő és az eszköz között szabványos kapcsolódási felület van, akár egy hivatalos ANSI, IEEE vagy ISO szabvány, akár valamilyen más, akkor a gyártó cégek a kapcsolódási felülethez illeszkedő vezérlőket és eszközöket tudnak készíteni. Például sok társaság gyárt lemezmeghajtó egységeket, amelyek az **IDE (Integrated Drive Electronics – integrált meghajtóelektronika)** vagy az **SCSI (Small Computer System Interface)** kapcsolódási felülethez illeszthetők.

Megjegyezzük, hogy a vezérlő és az eszköz közötti megkülönböztetést azért tesszük, mivel az operációs rendszer majdnem mindig a vezérlővel foglalkozik, és nem az eszközzel. A legtöbb személyi számítógép és szerver a 3.2. ábrán látható



3.2. ábra. *A CPU, a memória, a vezérlők és az I/O-eszközök összekapcsolásának egy modellje*

sín modellt alkalmazza a CPU és a vezérlők közötti kapcsolat megvalósításánál. A nagygépek gyakran más modellt használnak, speciális I/O-számítógépekkel; ezeket **I/O-csatornák**nak nevezik, amelyek átveszik a központi CPU-tól az átvitellel kapcsolatos feladatok egy részét.

A vezérlő és az eszköz közötti kapcsolódási felület gyakran alacsony szintű. Egy lemezt például lehet olyan formátumúra hozni, amelyben az 512 bájtos pálya 16 szektorra bontott. Az, amit valójában a meghajtó egység létrehoz, egy bitsorozat, amely egy **fejléc**cel kezdődik, utána egy 4096 bites szektor van, és végül egy ellenőrző összeg, amit **hibajavító kód**nak **(Error-Correcting Code, ECC)** neveznek. A fejléc a lemez formázása során kap értéket: tartalmazza a pályák és szektorok számát, a szektor méretét és hasonló adatokat.

A vezérlő feladata, hogy a bitsorozatot bájtokból álló blokkokba konvertálja és a szükséges hibajavításokat is elvégezze. Egy bájtokból álló blokk rendszerint először bitenként össze van gyűjtve a vezérlő egy belső pufferében. A memóriába másolás csak akkor hajtódik végre, ha a vezérlő már megvizsgálta a hibajavító kód helyességét, és a kijelölt blokk is hibátlan.

Egy monitorvezérlő úgy dolgozik, mint egy meglehetősen alacsony szintű bitszervezésű eszköz. A memóriából olvassa a megjelenítendő karaktereket tartalmazó bájtokat, és jeleket generál, amelyek módosítják a CRT sugárnyalábját. A vezérlő olyan jelzést is létrehoz, amely a CRT sugárnyalábját úgy irányítja, hogy egy teljes sor beolvasása után horizontálisan visszatér, és hasonlóan egy teljes képernyő beolvasása után a sugár vertikális visszatérését idézi elő. Az LCD képernyőnél ezek a jelek képpontokat választanak ki, és szabályozzák a fényerőt, így szimulálva a CRT elektron-sugárnyaláb hatását. Ha ez nem így lenne a videovezérlőnél, akkor az operációs rendszer programozójának kellene programoznia a tényleges képletapogatást. Az operációs rendszer néhány paraméterrel beállításokat hajt végre a vezérlővel kapcsolatosan, olyanokat, mint a képernyő soraiban a karakterek vagy a képpontok számának és a képernyő sorai számának beállítása, de a vezérlő látja el a megjelenítő irányítását.

3.1.3. Memórialeképezésű I/O

Minden vezérlő rendelkezik néhány regiszterrel, amelyet a CPU-val történő kapcsolat lebonyolítására használ. Ezekbe a regiszterekbe történő írással az operációs rendszer képes utasítani az eszközt, hogy adatot szállítson, adatot fogadjon, kapcsolja be vagy ki magát, vagy bizonyos más feladatot hajtson végre. Ezeknek a regisztereknek az olvasása révén az operációs rendszer megtudja, hogy milyen az eszköz állapota, vajon felkészült-e egy új parancs fogadására stb.

Ráadásul a vezérlőregiszterek mellett számos eszköz rendelkezik adatpufferrel, amelyet az operációs rendszer írni és olvasni tud. Például a számítógépek szokásosan videó RAM-ot használnak a képpontok képernyőn való megjelenítéséhez, ami ténylegesen egy adatpuffer, amelybe írhatnak a programok vagy az operációs rendszer.

Különbség tehető aközött, ahogyan a CPU kommunikál a vezérlőregiszterekkel és az eszköz adatpuffereivel. Két változat ismert. Az első megközelítés szerint mindegyik vezérlőregiszterhez hozzá van rendelve egy **I/O-kapu** szám, ami egy 8 vagy 16 bites egész szám. Egy speciális I/O-utasítás hatására, mint az

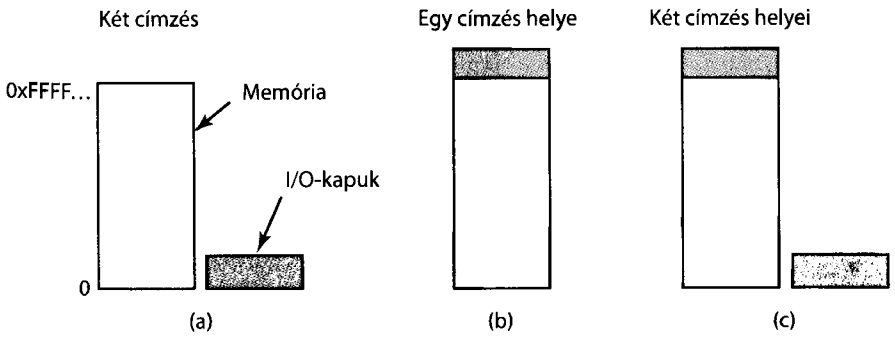
```
IN REG,PORT
```

a CPU kiolvassa a PORT vezérlőregiszter tartalmát, és tárolja az eredményt a CPU REG regiszterében. Hasonlóan, az

```
OUT PORT,REG
```

utasítás esetén a CPU a REG tartalmát beírja egy vezérlőregiszterbe. A korai számítógépek legtöbbje, beleértve majdnem az összes nagygépet, mint az IBM 360 és az összes utódja, ilyen módon működtek.

Ebben a rendszerben a memória és az I/O-cím helyek különbözők: ezt mutatja a 3.3.(a) ábra.



3.3. ábra. *(a) Külön I/O- és memóriahelyek. (b) Memórialeképezésű I/O. (c) Hibrid*

Más számítógépeknél az I/O-regiszterek a szokásos memóriacímtár egy részén találhatók, ez látható a 3.3.(b) ábrán. Ezt az elrendezést nevezik **memórialeképezésű I/O**-nak, amit a PDP–11 miniszámítógépnél vezettek be. Mindegyik vezérlőregiszter a memória egy olyan adott helyén van, ami csak erre használható. Szokásosan a kijelölt címek a címtartomány tetején helyezkednek el. A 3.3.(c) ábra egy hibrid rendszert mutat memórialeképezésű I/O-adatpufferrel és külön a vezérlőregiszterek számára I/O-kapukkal. A Pentium alkalmazza ezt az architektúrát: 640 K-tól 1 M-ig a címek az IBM PC-vel kompatibilis eszközök adatpufferei számára vannak fenntartva, továbbá vannak 0-tól 64 K-os I/O-kapuk is.

Hogyan működik ez a rendszer? Minden esetben, amikor a CPU egy szót olvasni akar vagy a memóriából, vagy egy I/O-kapuból, a sín címvonalára elhelyezi a címet, és utána kiad egy READ jelet a sínvezérlő vonalán. Egy második jelvonalat is használ, amely megmondja, vajon I/O-hely vagy memóriahely van megadva. Ha memóriahely, akkor a memória a felelős a kérésért, ha egy I/O-hely, akkor az I/O-eszköz a felelős. Ha csak memóriahely van – mint a 3.3.(b) ábra esetén –, akkor minden memóriamodul és minden I/O-eszköz összehasonlítja a címvonalat azzal a címtartománnyal, ami hozzátartozik. Ha a cím beleesik a tartományába, akkor ő a felelős a kérésért. Mivel egyetlen cím sincs memóriához és I/O-eszközhöz egyaránt hozzárendelve, így félreértés és konfliktus nem léphet fel.

3.1.4. Megszakítások

Rendszerint a vezérlőregisztereknek egy vagy több **állapotbitjük** van, aminek tesztelésével eldönthető, vajon egy kiviteli művelet hajtódik végre, vagy egy új adat érhető el egy beviteli eszközről. A CPU képes egy olyan ciklust futtatni, amely egy állapotbitet addig tesztel, amíg az eszköz készen áll a fogadásra vagy az új adat szolgáltatására. Ezt nevezik **tevékeny várakozás**nak. Már találkoztunk ezzel a fogalommal a 2.2.3. alfejezetben, mint a kritikus szekciók kezelésének lehetséges módszerével, de abban az összefüggésben el lett vetve mint a legtöbb esetben kerülendő technika. Nagy mennyiségű I/O esetén lehetséges, hogy nagyon hosszú ideig kell várakoznia valakinek az adat elfogadására vagy feldolgozására, ami nem elfogadható, kivéve azt a kevés számú rendszert, amelyek párhuzamos processzusokat nem futtatnak.

Az állapotbit mellett sok vezérlő alkalmaz megszakításokat, amely megmondja a CPU-nak, hogy mikor írhatja vagy olvashatja a regisztereket. A 2.1.6. alfejezetben már láttuk, hogyan kezeli a CPU a megszakításokat. Az I/O-val összefüggésben tudni kell, hogy a legtöbb interfészeszköz egy outputot ad ki, amely logikailag azonos a „művelet elvégezve" vagy „az adat készen áll" regiszter állapotbittel, de amely vezérli a rendszersín IRQ (Interrupt ReQuest) megszakításkérés-vonalak egyikét. Így amikor egy megszakítás művelet végbemegy, ez megszakítja a CPU-t, és elindítja a megszakításkezelő programot. Ez a kód tájékoztatja az operációs rendszert, hogy az I/O készen van. Majd az operációs rendszer ellenőrizheti az állapotbiteken keresztül, hogy minden jól működött, és vagy begyűjti az eredményezett adatot, vagy elindít egy újabb próbálkozást.

A megszakításvezérlő inputjainak a száma korlátozott lehet; a pentiumos PC-knél csak 15 áll az I/O-eszközök rendelkezésére. Néhány vezérlő a rendszer alaplapján mereven behuzalozott, mint például a billentyűzetvezérlő az IBM PC-nél. Régebbi rendszereknél az eszköz által használt IRQ megszakításkérés egy kapcsolóval vagy billenőkapcsolóval volt a vezérlőhöz kötve. Ha egy felhasználó egy új behelyezhető lapot vett, akkor kézzel kellett beállítania az IRQ-t úgy, hogy a létező IRQ-val ne ütközzön. Kevés felhasználó tudta ezt helyesen elvégezni; ez vezetett ahhoz, hogy az ipar kifejlesztette a **Plug 'n Play (beilleszt és működik)** rendszert, amelyben a BIOS automatikusan hozzárendeli az IRQ megszakításkéréseket az eszközökhöz az indítási időben, elkerülve így a konfliktusokat.

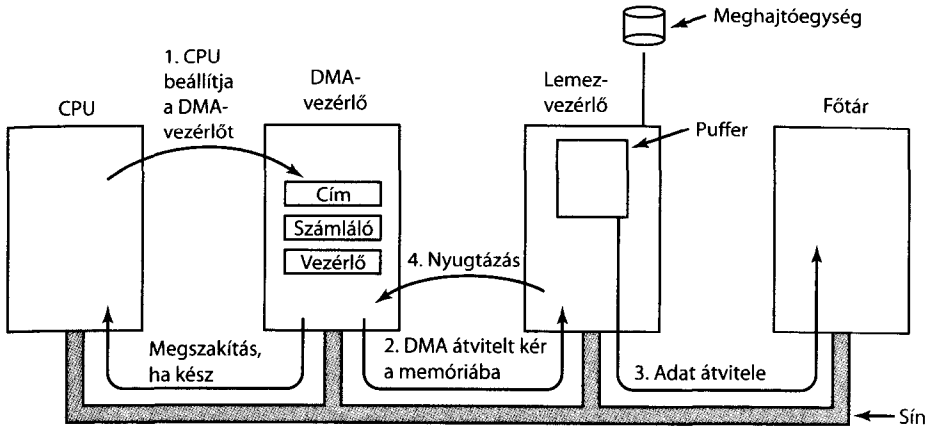
3.1.5. Közvetlen memóriaelérés (DMA)

Egy rendszer akár memórialeképezésű I/O-val rendelkezik, akár nem, a CPU számára szükséges az eszközvezérlők címzése, hogy adatot cseréljen velük. A CPU bájtonként is kérheti az adatokat az I/O-vezérlőtől, akkor viszont olyan eszköznél, mint egy nagy blokkokat tartalmazó lemez, a CPU sok időt pazarolna. Ez indokolja, hogy gyakran egy másik módszert alkalmaznak, az ún. **közvetlen memóriaelérést** vagy **DMA-t (Direct Memory Access)**. Az operációs rendszertaszk akkor alkalmazhatja a DMA-t, ha a hardver rendelkezik egy DMA-vezérlővel, ami a legtöbb esetben megvan. Néha ez a vezérlő be van ágyazva a lemezvezérlőkbe vagy más vezérlőkbe, de az ilyen esetekben külön DMA-vezérlő szükséges minden eszköz számára. Általánosabb, hogy egyetlen DMA-vezérlő van (például az alaplapon) több eszköz számára az átvitelek szabályozásához, gyakran egyidejű módon.

A DMA-vezérlő, akárhol is van fizikailag, hozzáfér a CPU-tól független rendszersínhez; ezt szemlélteti a 3.4. ábra. Számos regisztert tartalmaz, amelyeket a CPU olvasni és írni tud. Tartalmaz egy memóriacím-regisztert, egy bájtszámláló regisztert és egy vagy több vezérlőregisztert. A vezérlőregiszter határozza meg a használandó I/O-kaput, az átvitel irányát (olvasás az I/O-eszközről vagy írás az I/O-eszközre), az átvitel egységét (bájtonként vagy szavanként) és az átvitel egy ütemében továbbítandó bájtok számát.

Ahhoz, hogy elmagyarázzuk a DMA működését, először nézzük meg, hogyan zajlana le a lemezes olvasás, ha nincs DMA. A vezérlő először beolvassa a meghajtóegységből a blokkot (egy vagy több szektort), amit úgy végez, hogy bitenként folytatólagosan olvas a meghajtóegységből a saját belső pufferébe, amíg egy teljes blokk kialakul. Ezután kiszámolja a hibajavító kódot, amellyel ellenőrzi, hogy nem fordult elő olvasási hiba. Ezután egy megszakítást idéz elő. Amikor az operációs rendszer beindítja a futtatást, akkor a lemez egy blokkját a vezérlő pufferéből olvassa be egy ciklus végrehajtásával, amelyben minden egyes iterációs lépésben egy bájtot vagy szót olvas a vezérlő regiszteréből, és azt a memóriában tárolja, növelve a memóriacímet és csökkentve az egységszámlálót, addig olvas, amíg ennek értéke nulla lesz.

DMA alkalmazásánál a folyamat másként zajlik. Először is a CPU beprogramozza a DMA-vezérlőt, beállítva a regisztereit, így ez tudja, hogy milyen átvitelre



3.4. ábra. *Egy DMA átvitelművelet végrehajtása*

van szükség (3.4. ábrán az 1. lépés). Kiad egy parancsot a lemezvezérlőnek, hogy olvasson be adatot a lemezről a belső pufferbe, és ellenőrizze a hibajavító kódot. Amikor az adat helyes a lemezvezérlő pufferében, akkor indulhat a DMA.

A DMA-vezérlő elindítja az átvitelt, kiadva egy olvasási kérést a lemezvezérlőnek a sínen keresztül (2. lépés). Ez az olvasási kérés olyan, mint bármely másik olvasási kérés, és a lemezvezérlő nem is tudja, illetve nem figyel arra, hogy a kérés vajon a CPU-tól vagy a DMA-vezérlőtől érkezik. Szokásosan, a sín címsorában van az a memóriacím, ahová a lemezvezérlőnek a belső pufferéből a következő szót kell vinni. A memóriába írás egy szabványos sínciklus (3. lépés). Mikor a memóriába írás készen van, a lemezvezérlő küld egy visszaigazoló jelet a DMA-nak a sínen keresztül (4. lépés). A DMA-vezérlő ezután növeli a memóriacímet és csökkenti a bájtszámlálót. A 2–4. lépéseket mindaddig ismétli, amíg a bájtszámláló értéke el nem éri a 0 értéket. Ez az a pont, amikor a vezérlő egy megszakítást hoz létre. Amikor az operációs rendszer feldolgozza a megszakítást, nem kell a blokkot a memóriába másolnia, mert az már ott van.

Érdekes lehet, hogy miért nem tárolja a vezérlő a bájtokat azonnal a főtárban, amint a lemezről megkapja. Más szavakkal, miért van szükség egy belső pufferre? Ennek két oka van. Az egyik ok a belső puffer alkalmazására, hogy mielőtt az átvitel megtörténne, a lemezvezérlő ellenőrizheti a hibajavító kódot. Ha ez hibás, akkor hibajelzést generál, és nem történik meg az átvitel a memóriába. A másik ok, hogy ha egyszer egy lemezes adatátvitel elkezdődött, akkor a bitek állandó átviteli sebességgel folyamatosan jönnek a lemezről, függetlenül attól, hogy a vezérlő készen áll-e fogadásukra, vagy sem. Ha a vezérlő az adatot közvetlenül a memóriába próbálná írni, akkor minden egyes szó átviteléhez a rendszersínt kellene használnia. Ha viszont a sín foglalt, mert más eszköz használja éppen, akkor a vezérlőnek várakoznia kellene. Ha azelőtt érkezik a lemezről a következő szó, hogy a vezérlő az előzőt továbbadta, akkor a vezérlőnek azt valahol tárolnia kell. Ha a sín nagyon elfoglalt, akkor a vezérlő valószínűleg csak kevés szót képes tárolni, és sok adminisztrációt kell készítenie. Amikor a blokk belsőleg van pufferelve, akkor az adat-

csatornára egészen addig nincs szükség, amíg a DMA el nem indul, így a vezérlő szerkezete sokkal egyszerűbb, hiszen a DMA-s memóriába való átvitel nem jelent kritikus időt.

Nem minden számítógép alkalmaz DMA-t. Az ellenérv az, hogy a központi CPU gyakran jóval gyorsabb, mint a DMA-vezérlő, és sokkal gyorsabban tudja a munkát elvégezni (amikor az I/O-eszköz gyorsasága nem jelent korlátozó feltételt). Ha nincs más feladata, amivel foglalkozzon, akkor értelmetlen lenne, ha a (gyors) CPU várakozna arra, hogy a (lassú) DMA-vezérlő befejezze a munkát. Továbbá, pénz takarítható meg, és jelentőséggel bír a beágyazott számítógépeknél, ha a DMA-vezérlőtől megszabadulunk, és minden munkát szoftverúton a CPU-val végeztetünk el.

3.2. Az I/O-szoftver alapelvei

Hagyjuk most az I/O-hardvert, és nézzük az I/O-szoftvert. Először áttekintjük az I/O-szoftver céljait, majd megnézzük, hogy az operációs rendszer szempontjából a különböző módszereket az I/O hogyan képes elvégezni.

3.2.1. Az I/O-szoftver céljai

A kulcsfogalom az I/O-szoftver tervezésénél az úgynevezett **eszközfüggetlenség**. Ez azt jelenti, hogy lehetőség legyen olyan program írására, amely hozzáfér bármelyik I/O-eszközhöz anélkül, hogy előzetesen az eszközt meg kellene adni. Például egy programnak képesnek kell lennie arra, hogy egy állományt inputként be tudjon olvasni egy hajlékonylemezről, egy merevlemezről, egy CD-ROM-ból anélkül, hogy a programot módosítani kellene a különböző típusú eszközök szerint. Lehetőség legyen olyan parancs írására, mint

```
sort<input>output
```

és ez működjön akár egy hajlékonylemezről, egy IDE-lemezről, egy SCSI-lemezről vagy a billentyűzetről jövő bemenettel és tetszőleges típusú lemezre vagy a képernyőre kerülő kimenettel. Az operációs rendszer feladata azon problémáknak a megoldása, amelyek abból a tényből erednek, hogy ezek az eszközök valóban különbözők, és nagyon eltérő parancssorozatokat igényelnek az olvasáshoz és az íráshoz.

Az eszközfüggetlenséghez szorosan kapcsolódik az **egységes névhasználat**. Egy állomány vagy egy eszköz nevének egyszerűen egy jelsorozatnak vagy egy egész számnak kellene lennie, függetlenül az eszköztől. A Unixnál és a MINIX 3-nál az összes lemez egy fájlrendszerben hierarchikusan rendezett valamilyen módon, így a felhasználónak nem kell tudni arról, hogy melyik név melyik eszközhöz tartozik. Például egy hajlékonylemez **logikailag csatolható** a */usr/ast/backup* könyvtárhoz,

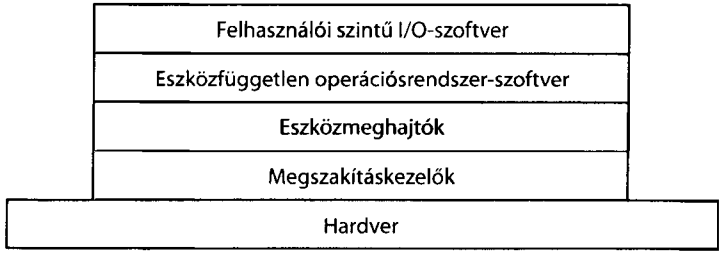
így egy állománynak a könyvtárba másolása esetén az állomány a hajlékonylemezre másolódik. Így az összes állomány és eszköz azonos módon címezhető: egy elérési út nevével.

Egy másik fontos kérdés az I/O-szoftverrel kapcsolatban a **hibakezelés**. A hibákat amennyire csak lehet, a hardverhez közeli szinten kellene kezelni. Ha a vezérlő olvasási hibát észlel, akkor megpróbálja a hibát kijavítani. Ha erre nem képes, akkor az eszközmeghajtónak kell kezelnie a hibát, esetleg a blokk olvasásának megismétlésével. Sok hiba átmeneti jellegű, mint például egy porszemcse előfordulása az olvasófejen, ami megszűnhet a művelet megismétlésével. A magasabb szoftverrétegeknek csak akkor kellene tudomást szerezniük a hibákról, ha azok az alacsonyabb rétegekben nem kezelhetők. Sok esetben a hiba orvoslása tökéletesen elvégezhető alacsony szinten anélkül, hogy a magasabb szintek bármit is észlelnének belőle.

További fontos kérdésként jelentkeznek a **szinkron** (blokkolásos) és **aszinkron** (megszakításvezérelt) **átviteli** módszerek. A fizikai I/O-k legtöbbje aszinkron jellegű – a CPU beindítja az átvitelt, és más feladatot lát el a megszakítás érkezéséig. A felhasználói programokat sokkal egyszerűbb úgy írni, ha az I/O-műveletek blokkoltak – a program egy receive rendszerhívás után automatikusan felfüggesztődik egészen addig, amíg az adat a pufferben elérhetővé válik. Az operációs rendszernek rendelkeznie kell olyan funkciókkal, amelyeken keresztül megszakításvezérelten kezelheti a blokkolt felhasználói programokat.

Az I/O-szoftverekkel kapcsolatban jelentkezik a **pufferezés** fogalom. Gyakran egy eszközről jövő adat nem tárolható közvetlenül a végső célhelyen. Például amikor csomag érkezik a hálózatról, az operációs rendszer addig nem tudja, hogy hová helyezze, amíg nem tárolja és vizsgálja meg valahol a csomagot. Továbbá a szigorúan valós válaszidejű eszközöknél (például a digitális audioeszközök) az adatot előre el kell helyezni egy kiviteli pufferbe, hogy ne jelentkezzen egy üres puffer feltöltésének ideje, ebből a célból kerülni kell a puffer túlcsordulását. A pufferezés tekintélyes mennyiségű másolással jár, és gyakran jelentősen befolyásolja az I/O teljesítményét.

Az utolsó fogalom, amellyel itt foglalkozunk, az eszközök megosztott és az ezzel ellentétes monopol módú használata. Néhány I/O-eszközt, például a lemezeket, sok felhasználó használhatja ugyanabban az időben. Nem okoz problémát, ha azonos időben több felhasználó is megnyit állományokat ugyanazon a lemezen. Más eszközök, például a szalagmeghajtó egységek, egyszerre csak egy felhasználó szá-



3.5. ábra. *Az I/O-szoftverrendszer rétegei*

mára elérhetők. Utána egy másik felhasználó birtokolhatja a szalagmeghajtó egységet. Ha két vagy több felhasználó véletlenszerű összevisszaságban írogatna blokkokat ugyanarra a szalagra, akkor használhatatlan dolog keletkezne. A monopol módú eszközök bevezetése ugyanakkor számos problémát okoz (például holtpontok). Ismét az operációs rendszerre marad, hogy kezelje mind a megosztva, mind a monopol módon használt eszközöket úgy, hogy a problémákat is elkerülje.

Az I/O-szoftver gyakran négy rétegbe van szervezve, miként a 3.5. ábra mutatja. A következő alfejezetekben valamennyit megnézzük, kezdve az alapoknál. Ebben a fejezetben a hangsúly az eszközvezérlőkön (2. réteg) lesz, de összefoglalást adunk az I/O-szoftver többi részéről is, megmutatva, hogyan illeszkednek egymáshoz az I/O-rendszer darabjai.

3.2.2. Megszakításkezelők

A megszakítások az élet örömtelen dolgai; jóllehet nem kerülhetők el. El kell rejteni őket az operációs rendszer belső részébe olyan mélyen, amennyire lehetséges, hogy az operációs rendszernek csak kicsi része tudjon róla. A legjobb módja az elrejtésnek, ha egy I/O-műveletet elkezdő meghajtó blokkolódik, amíg az I/O végbemegy, és egy megszakításkérés megjelenik. A meghajtó blokkolni tudja magát, például egy szemafor down, egy feltételváltozó wait vagy egy üzenet receive műveletével, vagy más hasonló eszközzel.

Megszakítás esetén a megszakításeljárás kezeli a megszakítást. Ezután képes a meghajtó blokkoltságát megszüntetni és elindítani. Bizonyos esetekben ez a szemafor up helyzetbe állítását jelenti. Másoknál a monitor feltételváltozója signal-ra változik. Vannak rendszerek, amelyeknél egy üzenet indul a blokkolt meghajtóhoz. Minden esetben a megszakítás egyértelmű hatása, hogy az előzőleg blokkolt meghajtó most folytatódhat. Ez a modell akkor működik a legjobban, ha a meghajtók független processzusokba strukturálhatók, saját állapotokkal, vermekkel és programszámlálókkal rendelkeznek.

3.2.3. Eszközmeghajtók

Ebben a fejezetben korábban már láttuk, hogy mindegyik eszközvezérlőnek vannak regiszterei, amelyen keresztül parancs adható, vagy kiolvasható az állapota, vagy mindkettő. A regiszterek száma és a parancsok jellege eszközönként nagyon különböző. Például az egérmeghajtó információt fogad az egértől, amely megadja az elmozdulását és azt, hogy melyik gombja lett lenyomva. Ezzel ellentétben a lemezvezérlőnek ismernie kell a szektorokat, sávokat, cilindereket, olvasófejeket, a kar mozgását, a motorikus meghajtóegységet, az olvasófej-beállítási időket és egyéb mechanikákat, amelyek révén a lemez valójában használható. Természetesen ezek a meghajtók nagyon különbözők lesznek.

Így a számítógéphez csatolható valamennyi I/O-eszköz vezérlésénél bizonyos eszközspecifikus kódra van szükség. Ezt a kódot nevezik **eszközmeghajtónak**, amelyet általában az eszköz gyártója ír, és az eszközzel együtt szállít egy CD-ROM-on. Mivel minden operációs rendszernek a saját meghajtóira van szüksége, ezért az eszközök gyártói több népszerű operációs rendszer meghajtójával is ellátják az eszközeiket.

Minden egyes eszközmeghajtó adott típusú eszközöket vagy legalábbis közeli rokonságban levő eszközök egy osztályát kezeli. Valószínű, hogy kellemes lenne, ha csak egyetlen egérmeghajtó kellene, még akkor is, ha a rendszer számos különböző fajtájú egeret támogat. Egy másik példaként vegyünk egy lemezmeghajtót, amely rendszerint számos eltérő méretű és eltérő sebességű lemez, talán még a CD-ROM kezelésére is képes. Másrészt, az egér és egy lemez annyira eltérők, hogy különböző meghajtóra van szükségük.

Annak érdekében, hogy az eszköz hardvere, a vezérlő regiszterei elérhetők legyenek, az eszközmeghajtó hagyományosan a kernel része. Ez a megközelítés a legjobb teljesítményt adja, és a legrosszabb megbízhatóságot, mivel a rendszer teljes összeomlását idézheti elő bármelyik eszközmeghajtóban egy hiba. A MINIX 3 ebből a modellből indul ki, hogy erősítse a megbízhatóságot. Mint látni fogjuk, a MINIX 3-nál minden eszközmeghajtó külön felhasználói szintű processzusként jelenik meg.

Mint már korábban megjegyeztük, az operációs rendszer annak alapján sorolja be a meghajtókat, hogy **blokkos eszközök** (mint például a lemez) vagy **karakteres eszközök** (mint például a billentyűzet és a nyomtató). A legtöbb operációs rendszer definiál egy olyan szabványos interfészt, amelyet az összes blokkos meghajtónak támogatnia kell, és egy olyan másik interfészt, amelyet az összes karakteres meghajtónak kell támogatnia. Ezek az interfészek olyan eljárásokból állnak, amelyet az operációs rendszer meghív, mikor a meghajtót dolgoztatni akarja.

Általános értelemben egy eszközmeghajtó feladata az eszközfüggetlen szoftvertől érkező absztrakt kérések fogadása és annak biztosítása, hogy a kérés ki legyen elégítve. Egy tipikus kérés a lemezmeghajtótól, hogy az n-edik blokkot olvassa be. Ha a meghajtó éppen üresjáratban van a kérés beérkezésének időpontjában, akkor azonnal elkezdi a kérés végrehajtását. Ha viszont már egy kérés miatt foglalt, akkor az új kérést elhelyezi a még megválaszolatlan kérések sorába, és amint lehetséges, foglalkozik vele.

Egy I/O-kérés végrehajtásakor az első lépés ellenőrizni, hogy az inputparaméterek helyesek-e, és hiba esetén jelezni. Ha a kérés érvényes, akkor a következő lépés az absztrakt alaknak konkrét formára transzformálása. Egy lemezmeghajtó esetében ez azt jelenti, hogy meghatározza a kért blokk tényleges lemezen való helyét, ellenőrzi, vajon a meghajtóegység motorja működik-e, megvizsgálja, hogy az olvasófej a megfelelő pályára van-e állítva, és így tovább. Röviden, a meghajtónak el kell döntenie, hogy milyen vezérlő műveletekre van szükség, és azok milyen sorrendben hajtódnak végre.

Valahányszor a meghajtó eldöntötte, hogy mely parancsokat kell kiadni a vezérlő számára, elkezdi azok beírását a vezérlő eszközregisztereibe. Egyszerű vezérlők egy időben csak egy parancsot tudnak kezelni. Intelligensebb vezérlők készek parancsok egy láncolt listájának az elfogadására is, amelyeket aztán végrehajtanak minden további, az operációs rendszertől jövő segítség nélkül.

A parancs vagy parancsok kiadása után kétféle helyzet közül az egyik lesz alkalmazható. Sok esetben az eszközmeghajtónak várakoznia kell, amíg a vezérlő bizonyos munkát elvégez számára, vagyis blokkolja magát a megszakítási kérés beérkezéséig, aminek hatására a blokkoltsága megszűnik. Más esetekben a művelet késedelem nélkül végrehajtódik, így a meghajtónak nem szükséges blokkolni. Az utóbbi esetre példa a képernyő görgetése, ami bizonyos grafikus kártyáknál csak néhány bájtnak a vezérlőregiszterekbe való beírását igényli. Semmiféle mechanikus mozgás nem szükséges, így a teljes művelet elvégezhető néhány mikroszekundum alatt.

Az előbbi esetben a blokkolt meghajtót a megszakítási kérés ébreszti fel. Az utóbbi esetben a meghajtó soha nem alszik. A művelet elvégzése után mindkét esetben hibaellenőrzés szükséges. Ha minden rendben van, akkor a meghajtó átadhatja az adatot az eszközfüggetlen szoftvernek (azaz egy blokkot beolvas). Végül a hibákkal kapcsolatos néhány állapotinformációt ad vissza a hívójának. Ha bármi kérés van a sorban, akkor azok egyikét kiválasztja, és elkezdi a teljesítését. Ha a sor üres, akkor a meghajtó blokkolódik, várakozván a következő kérésre.

Egy meghajtónak a fő feladata az olvasási és írási kérésekkel való foglalkozás, de lehetnek más kérések is. Például lehet, hogy a meghajtónak inicializálni kell az eszközt a rendszer indításakor vagy az eszköz első használatakor. Szintén lehetnek feladatok a tápegységgel, a Plug 'n Playjel vagy bejelentkezési eseményekkel kapcsolatban.

3.2.4. Eszközfüggetlen I/O-szoftver

Jóllehet bizonyos I/O-szoftverek eszközfüggők, nagy hányaduk eszközfüggetlen. A meghajtók és az eszközfüggetlen szoftverek közötti pontos határ függ a rendszertől, mivel bizonyos tevékenységek, amelyeket eszközfüggetlen szoftverrel kellene megoldani, adott esetben a meghajtóval hatékonyabban elvégeztethetők. A 3.6. ábrán felsorolt tevékenységek tipikusan az eszközfüggetlen szoftver részei. MINIX 3-ban az eszközfüggetlen szoftver a fájlrendszer része. Bár az 5. fejezetben tanulmányozni fogjuk a fájlrendszert, itt gyors áttekintést adunk az eszközfüggetlen szoftverről.

Az eszközfüggetlen szoftver alapvető feladata azoknak az I/O-tevékenységeknek a végrehajtása, amelyek minden eszköznél közösek, és egy szabványos kapcsolódási felület biztosítása a felhasználói szintű szoftver részére. A továbbiakban részletesebben megnézzük a fenti tevékenységeket.

Egységes kapcsolódási felület az eszközmeghajtóknál
Pufferezés
Hibakezelés
Monopol módú eszközök lefoglalása és elengedése
Eszközfüggetlen blokkméret biztosítása

3.6. ábra. *Az eszközfüggetlen I/O-szoftver tevékenységei*

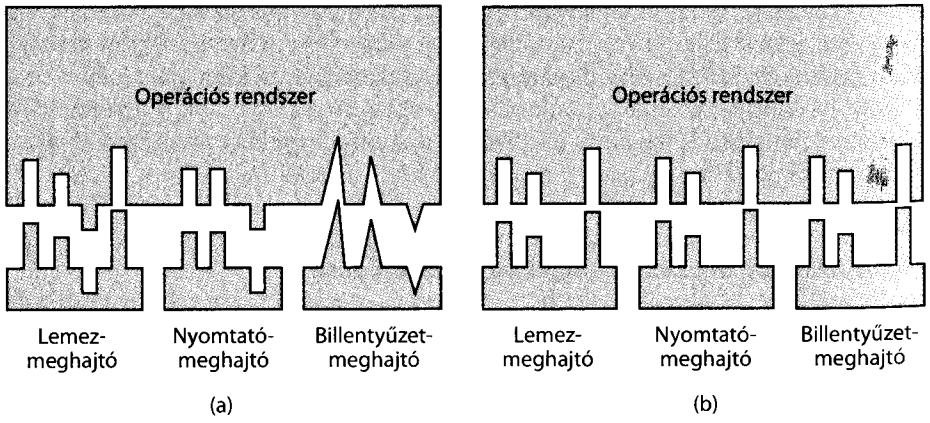
Egységes kapcsolódási felület az eszközmeghajtóknál

Fontos kérdés egy operációs rendszernél, hogy miként tekintheti az összes I/O-eszközt és -meghajtót többé-kevésbé azonosnak. Amennyiben a lemezek, nyomtatók, billentyűzetek stb. eltérő módon kapcsolódnak, akkor valahányszor egy új perifériás eszköz jelenik meg, az operációs rendszert az új eszköz szerint módosítani kellene. A 3.7.(a) ábra egy ilyen helyzetet szemléltet, vagyis ekkor minden eszközmeghajtónak különböző kapcsolódási felülete van az operációs rendszerhez. Ezzel ellentétben a 3.7.(b) egy olyan tervezést mutat, amelyben minden meghajtó azonos kapcsolódási felületű.

Egy szabványos kapcsolódási felületnél sokkal egyszerűbb egy új meghajtó bekötése, feltéve, hogy illeszkedik a meghajtó kapcsolódási felülethez. Ez azt is jelenti, hogy a meghajtóíróknak ismerniük kell az elvárásokat (milyen funkciókról kell gondoskodni és milyen kernelfüggvények hívhatók). A gyakorlatban az összes eszköz nem azonos, de rendszerint csak kisszámú eszköztípus van, és még ezek is általában majdnem azonosak. Például még a blokkos és karakteres eszközöknek is sok funkciója közös.

Az egységes kapcsolódási felület esetén egy másik szempont, hogy miként történjen az I/O-eszközök elnevezése. Az eszközfüggetlen szoftver gondoskodik a szimbolikus eszközneveknek a valós meghajtókhoz való hozzárendeléséről. Például a Unixban és a MINIX 3-ban egy olyan eszköznév, mint */dev/tty00,* egyértelműen kijelöl egy adott fájlhoz tartozó i-csomópontot, és ebben az i-csomópontban van egy **főeszközszám,** amely a megfelelő meghajtóra utal. Az i-csomópont tartalmaz egy **mellékeszközszám**ot is, amely paraméterként adódik át a meghajtónak, jelezve az egységet az íráshoz vagy olvasáshoz. Minden eszköznek van fő- és mellékeszközszáma, és az összes meghajtó elérhető a kiválasztott meghajtó főeszközszámával.

Az elnevezéshez szorosan kapcsolódik a védelem. Hogyan előzi meg a rendszer azt, hogy a felhasználók ne tudjanak elérni olyan eszközöket, amelyekhez nincs



3.7. ábra. *(a) Szabványos meghajtó interfész nélkül. (b) Szabványos meghajtó interfésszel*

hozzáférési joguk? A Unixnál, MINIX 3-nál és a Windows későbbi verzióinál, mint amilyen a Windows 2000 és az XP, az eszközök a fájlrendszerben jelennek meg, mint objektumnevek, ami azt jelenti, hogy a szokásos fájlvédelmi szabályokat alkalmazzák az I/O-eszközökre is. Minden eszköz esetében a rendszergazda állítja be a valós engedélyeket (például a Unixnál az *rwx* biteket).

Pufferezés

A pufferezés szintén egy megoldandó probléma mind a blokkos, mind a karakteres eszközök esetében. A blokkos eszközöknél a hardver általában az írást és olvasást teljesen blokkonként hajtja végre, míg a felhasználói processzusok szabadságot kívánnak abban, hogy az írás és olvasás milyen egységekben történjen. Ha egy felhasználói processzus egy fél blokkot ír, akkor az operációs rendszer általában mindaddig belsőleg tárolja ezt az adatot, amíg az adat további része is beolvasásra kerül, és ekkor teszi ki a blokkot a lemezre. Karakteres eszközöknél a felhasználó írhatja gyorsabban az adatokat, mint ahogy azok kivitelre kerülnek, ami szükségessé teszi a pufferezést. Szintén pufferezést igényel a felhasználás előtt billentyűzetről érkező input.

Hibakezelés

A hibák jóval gyakoribbak az I/O-val összefüggésben, mint más esetekben. Az operációs rendszernek az előforduló hibákat a legjobb tudása szerint kell kezelni. Sok hiba nagymértékben eszközfüggő, így csak a meghajtó tudja, hogy mit tegyen (újra próbálkozik, figyelmen kívül hagyja, pánikba esik). Egy tipikus hiba áll elő a lemez olyan blokkjánál, amely megsérült és többé már nem olvasható. Miután a meghajtó egy bizonyos számú alkalommal megpróbálja a blokk ismételt olvasását, feladja a további próbálkozást, és informálja az eszközfüggetlen szoftvert. A hiba kezelése innentől eszközfüggetlen. Ha a hiba egy felhasználói állomány olvasásakor jelentkezik, akkor valószínű, hogy elegendő a hibát közölni a hívóval. Ha azonban egy lényeges rendszeradat olvasásakor lép fel a hiba, például a bittérképet tartalmazó blokknál, ami a szabad blokkokat mutatja, akkor az operációs rendszernek valószínűleg nincs más választása, mint hogy kinyomtat egy hibaüzenetet, és befejeződik.

Monopol módú eszközök lefoglalása és elengedése

Néhány eszközt, mint például a CD-ROM-írókat, egyszerre csak egyetlen processzus használhat. Az operációs rendszer dolga, hogy a lemez használhatóságára vonatkozó kérést megvizsgálja, és azt elfogadja vagy elutasítsa, attól függően, hogy a lemez elérhető vagy foglalt. A kérések kezelésének egyszerű módja, ha maguktól a processzusoktól kérik, hogy az eszközökért közvetlenül hajtsanak végre

egy open-t a specifikus fájlokon. Ha az eszköz nem hozzáférhető, akkor az open sikertelen lesz. Az ilyen monopol módú eszközöket azután a bezárásuk teszi újra elérhetővé.

Eszközfüggetlen blokkméret

Nem minden lemez szektormérete azonos. Az eszközfüggetlen szoftverre hárul az a feladat, hogy ezt a tulajdonságot elrejtse, és egységes blokkméretet kínáljon a magasabb rétegek felé, például úgy, hogy több szektort tekint egyetlen logikai blokknak. Így a magasabb rétegek csak olyan absztrakt eszközökkel foglalkoznak, amelyek azonos logikai blokkméretet használnak a fizikai szektor méretétől függetlenül. Hasonló a helyzet néhány karakteres eszköznél, amelyek az adatokat bájtonként szállítják (ilyenek a modemek), míg mások nagyobb egységekben végzik ezt (ilyenek a hálózati kártyák). Ezt a különbözőséget is el kell rejteni.

3.2.5. A felhasználói szintű I/O-szoftver

Bár a legtöbb I/O-szoftver az operációs rendszerben van, kis része könyvtárakból áll, összekapcsolódva felhasználói programokkal, sőt olyan teljes programokkal, amelyek a kernelen kívül futnak. A rendszerhívásokat, beleértve az I/O-rendszerhívásokat is, rendszerint könyvtári eljárások végzik. Amikor egy C programban a

```
count = write(fd, buffer, nbytes);
```

utasítás szerepel, akkor a *write* könyvtári eljárás a programhoz kapcsolódik, és futáskor a memóriában levő bináris program tartalmazza. Az összes ilyen könyvtári eljárás gyűjteménye nyilvánvalóan része az I/O-rendszernek.

Míg ezek az eljárások szinte csak azt teszik, hogy a rendszerhívás számára a paramétereket elhelyezik a megfelelő helyre, addig vannak olyan I/O-eljárások, amelyek ténylegesen munkát végeznek. Például a formázott bevitelt és kivitelt könyvtári eljárások végzik. C-ből crre példa a *printf,* amely egy formázott sztringet és lehetőleg néhány változót vesz, mint inputot, felépíti az ASCII formátumú sztringet, és meghívja a write eljárást a sztring kiíratásához. A *printf*-re példaként nézzük a

```
printf("The square of %3d is %6d\n", i, i*i);
```

Ez kialakít egy sztringet, amely a 14 karakteres „The square of" kódból, az ezt követő *i*-nek az értékéből, ami 3 karakteres sztring, a 4 karakteres „is", a 6 karakteres i^2, és végül a sorvégjelből tevődik össze.

A bevitellel kapcsolatban egy hasonló eljárásra példa a *scanf,* amely olvassa a bemenő adatokat, és a formátumot leíró jelsorozatnak megfelelően – amely azonos szintaxisú, mint a *printf*-nél – értelmezi és tárolja azt a változókban. A szabvá-

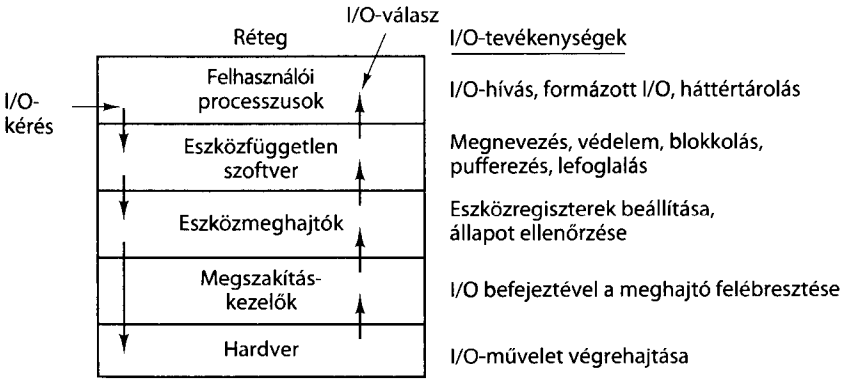
nyos I/O-könyvtárban számos olyan eljárás van, amelyek az I/O-t végzik, és mindegyik úgy fut, mint a felhasználói program része.

Nem minden felhasználói szintű I/O-szoftver tartalmaz könyvtári eljárásokat. Egy másik fontos fogalom a háttértárolás módszer. A **háttértárolás (spooling)** a multiprogramozású rendszerekben a monopol módon használható I/O-eszközök egyik kezelési módja. Tekintsünk egy tipikusan háttértárolással működő eszközt, a nyomtatót. Bár technikailag könnyen megengedhető, hogy bármely felhasználói processzus megnyithassa a nyomtató speciális, karakteres állományát, de tételezzük fel, hogy egy processzus a megnyitás után órákig nem tesz semmit. Ez idő alatt más processzus sem tudna semmit sem nyomtatni.

Az előbb említett probléma elkerülésére készült egy **démon**nak nevezett speciális processzus, és létrehoztak egy ún. **háttérkönyvtár**at **(spooling directory)**. Egy állomány nyomtatásakor a processzus először a teljes nyomtatandó állományt létrehozza, és a háttérkönyvtárba teszi. A démon az egyetlen processzus, amely rendelkezik a nyomtató speciális fájljához való hozzáféréssel, hogy a könyvtárban levő állományokat kinyomtassa. A speciális fájlnak a közvetlen felhasználói használattól való megvédésével elkerülhető az a probléma, hogy valaki az állományt a megnyitva szükségtelenül hosszú ideig lefoglalja.

A háttértárolás módszerét nemcsak a nyomtatók esetében alkalmazzák, hanem más helyzetekben is. Például az elektronikus levelezés rendszerint egy démont használ. Mikor egy üzenetet küldünk, az üzenet egy levelező háttértárba kerül. Később a levelező démon próbálja azt elküldeni. Lehetséges, hogy egy adott időpontban a megadott célállomás nem érhető el, ekkor a démon az üzenetet a háttértárban hagyja, megjelölve azzal a státuszinformációval, hogy ismételten próbálkozni kell az elküldéssel. A démon küldhet üzenetet a feladónak is, tájékoztatva a küldés késéséről, vagy arról, hogy adott idő elteltével sem sikerült a levelet a próbálkozások ellenére elszállítania. Ezek a tevékenységek nem az operációs rendszerre tartoznak.

A 3.8. ábra összefoglalja az I/O-rendszert, felsorolja a rétegeket fő feladataikkal együtt. Alulról kezdve a rétegek: hardver, megszakításkezelők, eszközmeghajtók, eszközfüggetlen szoftverek és végül felhasználói processzusok.



3.8. ábra. *Az I/O-rendszer rétegei és az egyes rétegek fő feladatai*

A 3.8. ábrán a nyilak a vezérlés irányát mutatják. Például amikor egy felhasználói program állományból blokkot akar olvasni, akkor az operációs rendszer segítségére van szükség a hívás végrehajtásához. Az eszközfüggetlen szoftver például szétnéz a blokkok raktárában. Ha a kívánt blokk nincs ott, akkor hívja az eszközmeghajtót, hogy az adjon ki egy kérést a hardver felé, hogy a lemezről olvassa be. A processzus egész addig blokkolva marad, amíg a lemezművelet végrehajtódik.

Mikor a lemezművelet befejeződött, a hardver egy megszakítási kérést generál. A megszakításkezelő kideríti, hogy mi történt, melyik eszközzel kell most éppen foglalkozni. Ezután kiveszi az eszközből az állapotot, és felébreszti az alvó processzust, hogy fejezze be az I/O-kérést, és engedje a felhasználói processzust folytatódni.

3.3. Holtpontok

A számítógéprendszerek erőforrásai között sok olyan van, amelyet egy időben csak egy processzus használhat. Ilyenek a nyomtatók, a szalagmeghajtó egységek és a rendszer belső táblázatának bejegyzései. Ha két processzus egyszerre írna a nyomtatóra, zagyvaság lenne az eredmény. Ha két processzus azonos fájlrendszertáblahelyet használna, akkor hibás fájlrendszer állna elő, és valószínűleg összeomlana a rendszer. Következésképpen, minden operációs rendszer olyan engedélyezési hatáskörrel rendelkezik, amely alapján egy processzus kizárólagos hozzáférést kap bizonyos erőforrásokhoz, mind a hardverhez, mind a szoftverhez.

Sok esetben egy processzus egynél több erőforrás esetében is kizárólagos hozzáférést igényel. Tegyük fel például, hogy két processzus mindegyike egy szkennelt dokumentumot akar CD-re rögzíteni. Az *A* engedélyt kér a szkenner használatára, és lefoglalja azt. A *B* processzus másként van programozva, és először a CD-írót kéri és foglalja le. Most az *A* processzus kéri a CD-írót, de a kérés mindaddig el van utasítva, amíg a *B* el nem engedi azt. Sajnos ahelyett, hogy a *B* elengedné a CD-írót, kéri a szkenner elérését. Ebben a pillanatban mindkét processzus blokkolt, és az is marad örökre. Ezt a helyzetet **holtpont**nak nevezik.

Holtpont sok olyan helyzetben is kialakulhat, amely nem a monopol módon használható I/O-eszközökre vonatkozó kérések miatt áll elő. Például egy adatbázisrendszerben egy program valószínűleg több rekordot is zárol, amelyekkel dolgozik, azért, hogy elkerülje a versenyt. Ha az *A* processzus zárolja az *R1* rekordot, a *B* processzus az *R2* rekordot, majd mindegyik processzus megpróbálja zárolni a másik rekordját, holtponthelyzet áll elő. Vagyis holtpont előfordulhat a hardver- és a szoftvererőforrásokkal kapcsolatosan egyaránt.

Ebben a fejezetben a holtpont problémával foglalkozunk, közelebbről megnézzük kialakulásának mikéntjét, és tanulunk néhány módszert a probléma megelőzésére vagy elkerülésére. Bár ehelyütt az operációs rendszerhez kapcsolódó holtponttal foglalkozunk, de a probléma előfordulhat adatbázisrendszereknél és sok más számítástudományi területen, így ez az anyag a párhuzamos rendszerek széles körében alkalmazható.

3.3.1. Erőforrások

Holtpont jöhet létre, amikor megengedett, hogy a processzusok kizárólagosan érhessenek el eszközöket, állományokat stb. A holtpontot, amennyire lehetséges, általánosan tárgyaljuk, és az engedélyezett objektumokra mint **erőforrások**ra hivatkozunk. Egy erőforrás lehet egy hardvereszköz (például egy szalagmeghajtó egység) vagy egy információ (például egy adatbázisban egy zárolt rekord). Egy számítógép általában sok különböző erőforrással rendelkezik, amelyek megszerezhetők. Bizonyos erőforrásokból több azonos példány is lehet, például három szalagmeghajtó egység. **Felcserélhető erőforrások**ról* beszélünk, ha egy erőforrásnak több egymással felcserélhető példánya is van, és az erőforrással kapcsolatos kérések kielégítésére a példányok bármelyike használható. Röviden: erőforráson azt a valamit értjük, amit ugyanabban az időben csak egy processzus használhat.

Az erőforrások két fajtája: a megszakíthatatlanok (monopol módú) és a megszakíthatók. Egy erőforrás **megszakítható**, ha elvehető az azt birtokló processzustól bármiféle hiba bekövetkezte nélkül. A memória ilyen megszakítható erőforrás. Tekintsük például azt a rendszert, amely egy 64 MB-os felhasználói memóriából, két 64 MB-os processzusból, valamint egy nyomtatóból áll, és mindegyik processzus nyomtatni kíván valamit. Az *A* processzus kéri és kapja a nyomtatót, majd hozzálát a nyomtatandó értékek kiszámolásához. Mielőtt azonban a számolással végezne, felhasználja a rendelkezésére álló időegységet, és így a memóriából kikerül.

A *B* processzus most fut, és sikertelen kísérletet tesz a nyomtató megszerzésére. Potenciálisan a rendszer holtponthelyzetbe került, mivel *A*-hoz a nyomtató és *B*-hez a memória van kapcsolva, és egyik sem tud folytatódni a másik által lefoglalt erőforrás nélkül. Szerencsére *B*-től elvehető a memória, és *A* betehető oda. Most *A* futhat, elvégezheti a nyomtatást, és utána a nyomtatót elengedi. Holtpont nem állt elő.

Ezzel szemben egy **megszakíthatatlan erőforrás**nak az elvétele a pillanatnyi tulajdonostól, számolási hibát eredményez. Ha egy processzus elkezdett egy írási folyamatot egy CD-ROM-ra, és hirtelen elveszik tőle a CD-írót, majd azt egy másik processzusnak adják, akkor egy kusza eredmény alakul ki a CD-n. A CD-írókat nem lehet megszakítani tetszőleges pillanatban.

Általában a holtpont probléma a megszakíthatatlan erőforrásokhoz kapcsolható. A megszakítható erőforrásokból álló lehetséges holtpontok rendszerint feloldhatók az erőforrásoknak az eljárások közötti újrakiosztásával. Épp ezért vizsgálatunk középpontjába a megszakíthatatlan erőforrások kerülnek.

Egy erőforrás használatával kapcsolatos tevékenységek absztrakt formái a következők:

1. Az erőforrás kérése.
2. Az erőforrás használata.
3. Az erőforrás elengedése.

* Ez egy jogi és pénzügyi fogalom. Az arany cserélhető: az egyik gramm arany ugyanolyan jó, mint a másik.

Ha az erőforrás a kéréskor nem hozzáférhető, akkor a kérő processzus kénytelen várakozni. Bizonyos operációs rendszerekben az erőforrás sikertelen kérésekor a processzus automatikusan blokkolódik, és felébred, mikor a kérés kielégíthető. Más rendszerekben a sikertelen kéréskor egy hibakód generálódik, és a hívó eljárás dolga az, hogy várakozzon egy kis ideig, majd ismét próbálkozzon.

3.3.2. A holtpont alapelvei

A holtpont formálisan a következőképpen definiálható:

> *Egy processzusokból álló halmaz holtpontban van, ha mindegyik halmazbeli processzus olyan eseményre várakozik, amelyet csak egy másik halmazbeli processzus okozhat.*

Mivel mindegyik processzus várakozik, így egyikük sem okozhat bármi olyan eseményt, ami felébreszthetné a halmaz bármely más tagját, vagyis az összes processzus folyamatosan várakozik mindvégig. Ennél a modellnél feltesszük, hogy a processzusok egyszálúak, és nem jelentkezik megszakítás, ami felébresztene egy blokkolt processzust. A megszakítás jelentkezését tiltó feltétel egy másféle holtpontprocesszust előz meg, ami a felébresztésből adódna, mondjuk egy váratlan esemény alkalmával a halmaz többi processzusa el lenne engedve.

A legtöbb esetben a processzusok arra az eseményre várnak, hogy más halmazbeli elem az általa pillanatnyilag lefoglalt néhány erőforrást elengedje. Más szavakkal: a holtpontos processzusok halmazának mindegyik tagja egy olyan erőforrásra vár, amelyet egy holtpontos processzus lefoglalva tart. Egyik processzus sem tud futni, egyik sem tud semmiféle erőforrást elengedni, és egyik sem tud felébredni. A processzusok száma, a lefoglalt és kért erőforrások száma és fajtája lényegtelen dolgok. Ez az eredmény érvényes mind hardver-, mind szoftvererőforrásokra.

A holtpont feltételei

Coffman és társai (Coffman et al., 1971) megmutatták, hogy négy feltétel megléte szükséges a holtpont kialakulásához:

1. Kölcsönös kizárás feltétel. Minden egyes erőforrás vagy aktuálisan hozzá van rendelve pontosan egy processzushoz, vagy szabad.
2. Birtoklás és várakozás feltétel. A már korábban kapott erőforrásokat birtokló processzusok kérhetnek újabb erőforrásokat.
3. Megszakíthatatlanság feltétel. Egy processzustól az előzőleg engedélyezett erőforrások nem vehetők el semmi módon. Az erőforrásokat csak az őket birtokló processzusok engedhetik.
4. Ciklikus várakozás feltétel. Két vagy több processzusból összetevődő ciklikus láncnak kell kialakulnia, amelynek mindegyik processzusa olyan erőforrásra várakozik, amelyet a láncban következő processzus tart fogva.

A feltételek mindegyikének teljesülnie kell egy holtpontban. Ha egy vagy több feltétel nem teljesül, akkor holtpont sem alakulhat ki.

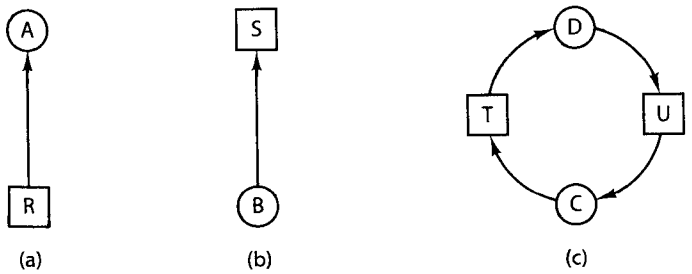
Levine cikksorozatában (Levine, 2003a; 2003b; 2005) rámutat arra, hogy az irodalomban változatos helyzeteket neveznek holtpontnak, és a Coffman és mások által megfogalmazott feltételeket kielégítő helyzeteket valójában **erőforrásholtpont**nak kellene hívni. Az irodalomban vannak olyan „holtpont" példák, amelyekben az összes fenti feltétel nem teljesül. Például ha egy kereszteződésben négy jármű találkozik, és megpróbálnak a szabályoknak engedelmeskedni, elsőbbséget adni a jobbra levő járműnek, akkor egyik sem haladhat tovább, de ez olyan eset, amelynél erőforrás-birtoklás nincs. Ez a probléma inkább „ütemezési holtpont", amely megoldható egy rendőrtől kívülről érkező elsőbbséget definiáló paranccsal.

Érdemes megjegyezni, hogy mindegyik feltétel egy adott szabállyal kapcsolatos, amellyel a rendszer vagy rendelkezik, vagy nem. Lehet-e egy adott erőforrást egynél több processzushoz hozzárendelni egyszerre? Egy processzus foglalhat-e egy erőforrást, és kérhet-e egy másikat? Megszakítható-e az erőforrás? Léteznek ciklikus várakozások? A későbbiekben megnézzük, hogyan támadható meg a holtpont bizonyos feltételek negálásával.

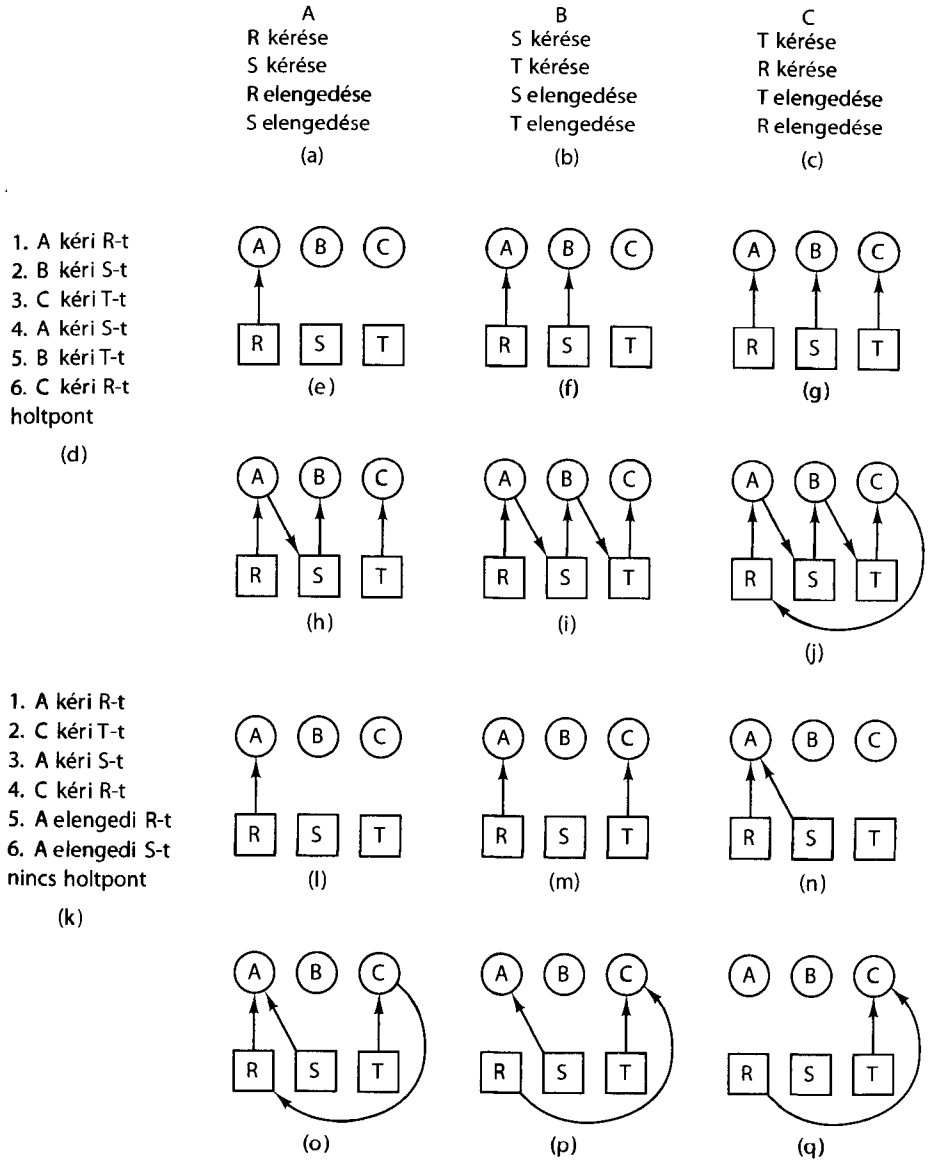
A holtpont modellje

Holt megmutatta, hogy miként modellezhető irányított gráfokkal ez a négy feltétel (Holt, 1972). A gráfoknak kétféle csúcsa van: processzus, körrel jelölve, és erőforrás, négyzettel jelölve. Egy erőforráscsúcsból (négyzet) egy processzuscsúcsba (kör) mutató él azt jelenti, hogy az erőforrást a processzus előzőleg már kérte, engedélyezték számára, és jelenleg birtokolja. A 3.9.(a) ábra szerint az *R* erőforrás az *A* processzushoz van rendelve.

Egy processzusból egy erőforrásba mutató él azt jelenti, hogy a processzus pillanatnyilag blokkolt és várakozik az erőforrásra. A 3.9.(b) ábra alapján a *B* processzus várakozik az *S* erőforrásra. A 3.9.(c) ábra egy holtpontot mutat: a *C* processzus a *T* erőforrásra várakozik, amelyet aktuálisan a *D* processzus birtokol. A *D* processzus nincs abban a helyzetben, hogy elengedje a *T* erőforrást, mivel éppen az *U* erőforrásra várakozik, amelyet a *C* birtokol. Mindkét processzus a végtelenségig



3.9. ábra. *Az erőforrás-lefoglalások gráfjai. (a) Az erőforrás birtoklása. (b) Egy erőforrás kérése. (c) Holtpont*



3.10. ábra. *Példa holtpont előfordulására és ennek elkerülésére*

fog várakozni. Egy gráfbeli kör azt jelzi, hogy holtpont van; ezt a kört alkotó processzusok és erőforrások hozzák létre. Ebben a példában a *C–T–D–U–C* egy kör.

Most nézzük hogyan használhatók az erőforrásgráfok. Képzeljük el, hogy van három processzusunk, *A*, *B* és *C*, és három erőforrásunk, *R*, *S* és *T*. A három processzus erőforrásigényeit és -elengedéseit a 3.10.(a)–(c) ábra mutatja. Az operációs rendszer bármelyik blokkolatlan processzust bármikor futtathatja, dönthet

úgy, hogy az *A* processzust futtatja egész addig, amíg minden munkáját elvégzi, ezután a *B* processzust futtatja végig, és végül a *C*-t.

Ez a sorrend nem vezet semmiféle holtponthoz (mivel nincs az erőforrásokért verseny), de párhuzamosítást sem tartalmaz. Az erőforrásigényeken és -elengedéseken túl a processzusok számolnak és I/O-t végeznek. Amikor a processzusok egymás után futnak, akkor nincs lehetőség arra, hogy mialatt egy processzus I/O-ra várakozik, egy másik használhassa a CPU-t. A processzusok ilyen szigorúan szekvenciálisan való futtatása valószínűleg nem optimális. Másrészt, ha a processzusok egyike sem végez I/O-t, akkor a legrövidebb feladatot először ütemező módszert jobb választani, mint a ciklikus ütemezés (round robin) módszerét, vagyis bizonyos körülmények között a processzusok szekvenciális futtatása a legjobb mód.

Most tegyük fel, hogy a processzusok mind I/O-t, mind számolást végeznek, és így a ciklikus ütemezés egy ésszerű ütemezési algoritmus. Az erőforráskérések a 3.10. (d) ábrán szereplő sorrend szerint érkeznek. Ha ez a hat kérés az adott sorrendben teljesül, akkor az ennek megfelelő hat erőforrásgráf a 3.10.(e)–(j) ábrákon látható. A 4. kérés teljesítése után az *A* blokkolódik, az *S*-re várakozva, amit a 3.10.(h) ábra mutat. A következő két lépés során a *B* és *C* is blokkolódik, ami végül is körhöz és holtponthoz vezet; ezt a 3.10.(j) ábra szemlélteti. Ennél a pontnál a rendszer lefagy.

Mint ahogy már megjegyeztük, az operációs rendszernek nem kötelező a processzusokat valami speciális sorrendben futtatnia. Különösen akkor, ha egyes erőforrások használata holtponthoz vezetne, akkor az operációs rendszer egyszerűen felfüggesztheti a processzust a kérés teljesítése nélkül (azaz, nem ütemezi a processzust) addig, amíg biztonságossá nem válik. A 3.10. ábrán előforduló esetben, ha az operációs rendszer tudott volna a küszöbönálló holtpontról, akkor ahelyett, hogy az *S*-t engedélyezte, felfüggeszthette volna a *B*-t. Ha csak az *A* és *C* fut, akkor a 3.10.(d) ábra helyett a 3.10.(k) ábrán látható kéréseket és elengedéseket kapnánk. Ehhez a sorozathoz a 3.10.(l)–(q) ábrákon szereplő erőforrásgráfok tartoznak, és ez nem vezet holtponthoz.

A (q) lépés után a *B*-nek megengedhető az *S*, mivel az *A* befejeződik, és *C* is birtokol már mindent, amire szüksége van. Ha esetleg *B*-nek ténylegesen blokkolódnia kell, amikor a *T*-t kéri, akkor sem lesz holtpont, ugyanis *B* csak addig várakozik, amíg *C* befejeződik.

Később ebben a fejezetben egy algoritmus vizsgálatára is sor kerül, amellyel olyan lefoglalási döntések hozhatók, amelyek nem vezetnek holtponthoz. Most már érthető, hogy az erőforrásgráfok eszközként szolgálnak ahhoz, hogy megnézhessük, vajon egy adott igény/elengedés sorozat holtponthoz vezet-e. Csak ki kell elégíteni lépésenként az igényeket és elengedéseket, minden egyes lépés után ellenőrizni kell a gráfot, hogy tartalmaz-e kört. Ha igen, akkor holtpont van; ha nem, akkor nincs holtpont. Bár az erőforrásgráfokat olyan esetben használtuk, amelyekben minden egyes erőforrástípusból csak egyetlen erőforrás volt, az erőforrásgráfok általánosíthatók az olyan esetekre is, amikor több azonos típusú erőforrás van (Holt, 1972). Viszont Levine rámutatott arra, hogy felcserélhető erőforrásokkal ez valójában bonyolultabbá válik (Levine, 2003a; 2003b). Ha van a gráfnak egy ciklusba nem tartozó ága, egy holtpontba nem tartozó processzus, amely az erőforrások egyikének egy példányát birtokolja, akkor nincs holtpont.

Általában négy stratégiát használnak a holtpontokkal kapcsolatban.

1. A probléma teljesen figyelmen kívül hagyása. Lehet, hogy ennek az lesz a hatása, hogy az is figyelmen kívül hagy minket.
2. Felismerés és helyreállítás. Engedjük a holtpontot megjelenni, vegyük észre és cselekedjünk.
3. Dinamikus elkerülés az erőforrások körültekintő lefoglalásával.
4. Megelőzés, strukturálisan meghiúsítva a négy szükséges feltétel egyikét, amelyek szükségesek a holtpont kialakulásához.

A következő négy alfejezetben valamennyi stratégia vizsgálatára sor kerül.

3.3.3. A strucc algoritmus

A strucc algoritmus a legegyszerűbb megközelítés: dugjuk a fejünket a homokba, és tegyünk úgy, mintha egyáltalán semmi probléma nem létezne.* Erre a stratégiára a különböző emberek különböző módon reagálnak. A matematikusok szerint ez teljesen elfogadhatatlan, és véleményük szerint a holtpontot mindenáron meg kell előzni. A mérnökök megkérdezik, hogy milyen gyakran várható a probléma megjelenése, milyen gyakorisággal omlik össze a rendszer más okok miatt, mennyire veszélyes egy holtpont. Ha a holtpontok átlagban ötévente egyszer fordulnak elő, míg a hardverhibákból, a fordító hibáiból és a rendszernek az operációs rendszer tévedéseiből származó összeomlása hetente egyszer előfordul, akkor a legtöbb mérnök nem óhajt nagy árat fizetni a holtpontok megelőzéséért a rendszer teljesítményének csökkenésével vagy kényelmi szempontok feladásával.

Tegyük ezt az ellentétet konkrétabbá, és nézzük a Unix és a MINIX 3 operációs rendszert, ahol felléphetnek holtpontok, amelyeket észre sem vesz a rendszer, nemhogy automatikusan feloldaná azokat. A rendszerbeli processzusok teljes számát a processzustáblázat bejegyzéseinek a száma határozza meg. Így a processzustáblázat szabad helyei véges számú erőforrások. Ha egy fork sikertelen, mivel a táblázat tele van, akkor a probléma ésszerű megközelítése, hogy a fork-ot végrehajtó processzus valameddig várakozik, és utána ismét próbálkozik.

Tegyük most fel, hogy a MINIX 3-rendszerben 100 processzus számára van hely. Tíz program fut, amelyek mindegyikének 12 (rész) processzust kell létrehoznia. Miután mindegyik processzus már 9 processzust létrehozott, a 10 eredeti processzus és a 90 új processzus kitölti a táblázatot. Az eredeti 10 processzus most egy-egy végtelenségig ismétlődő sikertelen klónozással küszködik – holtponthelyzet. Ezen helyzet bekövetkeztének valószínűsége nagyon kicsi, de megtörténhet. Mondjunk le a processzusokról és a fork hívásról, hogy a probléma megszűnjön?

* Valójában ez egy ostoba szólás. A struccok 60 km/h sebességgel képesek futni, rúgásuk pedig elég erős ahhoz, hogy akár egy vacsoráról ábrándozó oroszlánt is meg tudjanak ölni vele.

A nyitott állományok maximális száma is korlátozva van, mégpedig az i-csomópont táblázat méretével, így a táblázat beteltével – az előzőhöz hasonló – probléma áll elő. Egy másik korlátozott erőforrás a csere lemezterülete. Valójában az operációs rendszer majdnem minden táblázata egy véges erőforrás. Ezeket mind el kellene dobnunk, mert megtörténhet, hogy egy n processzusból álló halmazban a processzusok mindegyike – miután a véges erőforrások $1/n$-ed részét már kérte – csak próbálkozna a fennmaradó igénye kielégítésével?

A legtöbb operációs rendszer, így a Unix, a MINIX 3 és a Windows megközelítése, hogy ne foglalkozzunk a problémával, és ez azon alapul, hogy a felhasználók többsége inkább elfogadja azt, hogy alkalmanként holtpont keletkezzen, mint hogy egy olyan szabályhoz alkalmazkodjon, amely minden felhasználónak csak egy processzust, egy nyitott állományt és mindenből csak egyet engedélyezne. Ha a holtpont felszámolása semmibe se kerülne, akkor erről a problémáról nem is kellene beszélni. A probléma az, hogy az ár magas, ami főleg a processzusokat sújtaná a kényelmetlen szigorításokkal. Így szembe kell nézni azzal a nem túl örömteli helyzettel, amelyben választani kell a kényelem és a megbízhatóság között, és elemezni kell, hogy melyik a fontosabb és kinek. Ilyen feltételek mellett általános megoldást nehéz találni.

3.3.4. Felismerés és helyreállítás

A második stratégia a felismerés és helyreállítás. Ennek a technikának az alkalmazásakor a rendszer semmi egyebet nem tesz, mint figyeli az erőforrásigényeket és -elengedéseket. Minden egyes erőforráskéréskor vagy -elengedéskor az erőforrásgráfot módosítja, és ellenőrzi, hogy van-e benne kör. Ha kör keletkezik, akkor az abban levő processzusok egyikét megszünteti. Ha így sem sikerült megszüntetnie a holtpontot, akkor vesz egy másik processzust és megszünteti azt is, és így tesz mindaddig, amíg a kör meg nem szűnik.

Egy valamivel durvább módszer az, amely még az erőforrásgráffal sem foglalkozik, hanem periodikusan ellenőrzi, hogy vannak-e olyan processzusok, amelyek mondjuk már 1 óránál hosszabb ideje folyamatosan blokkoltak. Az ilyen processzusokat megszünteti.

A felismerés és helyreállítás stratégia a nagygépeknél gyakran használt módszer, különösen kötegelt rendszerekben, amelyekben egy processzus megszüntetése és újraindítása elfogadott dolog. A gondot ekkor az jelenti, hogy gondoskodni kell az eredeti állapothoz képest módosult állományok helyreállításáról, továbbá minden más mellékhatást ki kell küszöbölni, ami előfordulhatott.

3.3.5. A holtpont megelőzése

A harmadik holtpont stratégia alkalmas megszorításokat ír elő a processzusoknak úgy, hogy a holtpont eleve lehetetlen legyen. A Coffman és társai által megfogalmazott négy feltétel (Coffman et al., 1971) kulcsként szolgál néhány lehetséges megoldáshoz.

Elsőként vegyük a kölcsönös kizárás feltételt. Ha egyetlen erőforrás sincs soha kizárólagosan egy processzushoz rendelve, akkor holtpont sem lehet. De nyilvánvaló, hogy ha két processzusnak egy időben megengedett, hogy az eredményeit a nyomtatóra kiírja, akkor ebből káosz lesz. A nyomtatóra küldött állományok háttértárban történő tárolásával több processzus is tud egy időben kimenetet előállítani. Ebben a modellben csak egyetlen processzus kéri ténylegesen a fizikai nyomtatót, és ez a nyomtató démonja. Mivel a démonnak nincs semmi más erőforrásigénye, így a nyomtatóval kapcsolatos holtpont megszűnik.

Sajnos nem minden eszköznél alkalmazható a háttértárolásos módszer (a processzustáblázat esetében ez a módszer nem vezetne jóra). De holtpont állhat be a háttértárként szolgáló lemezhelyekért folyó versenyben is. Mi történik akkor, ha van két processzus, amelyek mindegyike az elérhető belső tárolóhelyek felét már felhasználta az output számára, és ekkor még az output folyamat nem fejeződik be? Ha a démon úgy van programozva, hogy a nyomtatást elkezdi, mielőtt a teljes kimenet kialakulna a háttértárban, akkor a nyomtatónak esetleg tétlenül kell várakoznia a kimenet első részének kiírása után, mert a kiviteli processzusra kell várnia. Ezért a démonok általában úgy vannak programozva, hogy csak azt követően nyomtatnak, miután a teljes kimeneti állomány elérhető. Esetünkben van két processzus, amelyek outputjának egy része befejeződött, de nem teljesen és nem tudnak folytatódni. Egyik processzus sem fejeződik be, és a lemezen holtpont jön létre.

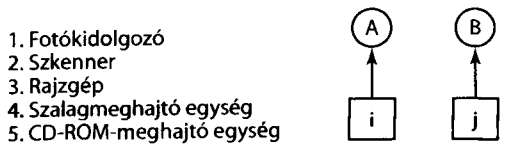
A Coffman és társai által adott második feltétel ígéretesebbnek néz ki. Ha sikerül megelőzni olyan helyzeteket, amelyekben erőforrásokat birtokló processzusok várakoznak további erőforrásokra, akkor nem lesz holtpont. Ezen cél elérésének egyik módja az, ha minden processzus még a futása előtt az összes erőforrásigényét közölné. Ha mind elérhető lenne, akkor a processzus lefoglalhatna bármit, ami kell, és végig futhatna. Ha egy vagy több erőforrás foglalt lenne, akkor semmit sem foglalna le, hanem várakoznia kellene.

Az első gond ezzel a megközelítéssel kapcsolatban az, hogy sok processzus a kezdetekor nem tudja, hogy mennyi erőforrásra lesz szüksége. Egy másik probléma, hogy ezzel a módszerrel az erőforrások nincsenek optimálisan kihasználva. Vegyük például azt az esetet, amikor egy processzus adatokat olvas egy bemeneti szalagról, egy óráig elemzi azokat, majd ír egy kimeneti szalagot, valamint kirajzolja az eredményt. Ha az összes erőforrást előre kell kérni, akkor ez a processzus egy órán keresztül foglalja a kimeneti szalagmeghajtó egységet és a rajzolót.

A „birtokol és várakozik" feltétel megtörésének egy kissé másik módja, amikor egy erőforrás igénylésekor a processzusnak először el kell engednie az összes, pillanatnyilag birtokolt erőforrását. Ezután megpróbálhatja mindegyiket megszerezni, ha egyáltalán kell neki.

A harmadik feltétel (megszakíthatatlanság) elkerülése még kevésbé biztató, mint a másodiké. Ha egy processzus már birtokolja a nyomtatót, és eredményei nyomtatása közben elveszik a nyomtatót, mert egy kért rajzgép nem elérhető, az hihetetlenül nagy zagyvasághoz vezetne.

Már csak egy feltétel maradt. A ciklikus várakozás többféle módon is megszüntethető. Egy egyszerű mód, ha alkalmazzuk azt a szabályt, hogy egy processzus bármely pillanatban csak egyetlen erőforrást foglalhat. Ha igénye van másodikra is,



3.11. ábra. *(a) Numerikusan rendezett erőforrások. (b) Egy erőforrásgráf*

akkor el kell engednie az első erőforrást. Egy olyan processzusnál, amely egy hatalmas állományt kíván egy szalagról átmásolni egy nyomtatóra, ez a megszorítás elfogadhatatlan.

A ciklikus várakozás elkerülésének másik módja az összes erőforrás megszámozásával történik; ez látható a 3.11.(a) ábrán. Most a szabály a következő: a processzusok bármikor igényelhetik az erőforrásokat, de csak a megadott számsorrendben. Egy processzus kérheti először a szkennert és utána egy szalagmeghajtó egységet, de nem kérhet először egy rajzgépet és utána egy szkennert.

Ennek a szabálynak az érvényesítése mellett az erőforrások foglaltsági gráfjában sohasem lesz kör. Nézzük, miért igaz ez a 3.11.(b) ábrában szereplő két processzus esetében. Holtpont csak akkor lenne, ha A kérné a j erőforrást és B kérné az i erőforrást. Tételezzük fel, hogy az i és a j különböző erőforrások, ekkor a sorszámaik különbözők. Ha $i > j$, akkor A-nak nincs megengedve a j kérése, mivel az alacsonyabb, mint amivel már rendelkezik. Ha $i < j$, akkor B-nek nincs megengedve az i kérése, mivel az alacsonyabb, mint amivel már rendelkezik. A holtpont mindegyik esetben lehetetlen.

Több processzus esetében teljesen hasonló meggondolások érvényesek. Mindig van a kiosztott erőforrások között egy legnagyobb számú. Ezt az erőforrást birtokló processzus soha nem kérhet már kiosztott erőforrást. Vagy befejeződik, vagy rosszabb esetben még magasabb számú erőforrásokat kér, amelyek mindegyike hozzáférhető. Végül befejeződik, és felszabadítja az erőforrásait. Ekkor valamely másik processzus foglalja a legnagyobb számú erőforrást, és így ez is befejeződhet. Röviden szólva, létezik egy forgatókönyv, amely szerint az összes processzus lefut, így holtpont nem alakulhat ki.

Ezt az algoritmust csak kissé változtatja meg, ha az erőforrások megszerzésének szigorúan növekvő sorrendben való követelményét kidobjuk, és csak ahhoz ragaszkodunk, hogy a processzusok ne kérhessenek az éppen birtokolt erőforrásaiknál kisebb számút. Ha egy processzus kezdetben a 9-et és a 10-et kérte, és utána mindkettőt elengedte, akkor valójában mindent kezdhet elölről, így nincs semmi ok arra, hogy tilos legyen számára akár az 1-es erőforrást kérni.

Bár az erőforrások számozott sorrendjével a holtpont kiküszöbölhető, de szinte lehetetlen egy mindenkit kielégítő sorrendet találni. Mikor az erőforrások között vannak processzustáblázati szabad helyek, zárolt adatbázisrekordok és más absztrakt erőforrások, akkor a lehetséges erőforrások és a különböző használatok száma olyan nagy lehet, hogy nem rendezhetők olyan sorrendbe, amely alapján dolgozhatnánk. Miként Levine rámutat (Levine, 2005), az erőforrások rendezése

Feltétel	Megközelítés
Kölcsönös kizárás	Háttértárolás
Birtokol és várakozik	Az összes erőforrásigény előzetes kérése
Megszakíthatatlanság	Az erőforrás elvétele
Ciklikus várakozás	Erőforrások numerikus rendezése

3.12. ábra. *A holtpontmegelőzés lehetőségeinek összefoglalása*

nem alkalmas a felcserélhető rendszernél – egy teljesen jó és elérhető erőforráspéldány nem lenne elérhető egy ilyen szabálynál.

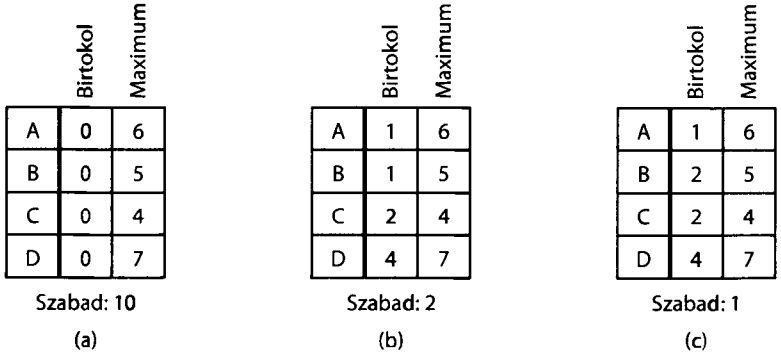
A 3.12. ábrán a holtpontmegelőzés különböző megközelítéseinek összefoglalása látható.

3.3.6. A holtpont elkerülése

A 3.10. ábrán szereplő példánál láttuk, hogy a holtpont elkerülése nem valamilyen szabálynak a processzusokra való rákényszerítésével történt, hanem minden egyes erőforráskérés biztonságos voltának gondos elemzésével. A kérdés most az: van-e olyan algoritmus, amellyel mindig elkerülhető a holtpont? A válasz egy feltételes igen – elkerülhető a holtpont, ha bizonyos információ előre elérhető. Ebben a részben a holtpontelkerülés olyan módjait vizsgáljuk, amelyek alapja a körültekintő erőforrás-lefoglalás.

A bankár algoritmus egyetlen erőforrásra

Dijkstrától származik az egyik holtpontelkerülő ütemezési algoritmus, amely úgy ismert, mint a **bankár algoritmus** (Dijkstra, 1965). Ez egy kisvárosi bankár munkájának modellje, aki ügyfelek egy csoportjával foglalkozik, akiknek engedélyez



3.13. ábra. *Három erőforrás-lefoglalási állapot. (a) Biztonságos. (b) Biztonságos. (c) Bizonytalan*

bizonyos nagyságú hitelt. A bankárnak nincs szükségszerűen annyi készpénze, ami elegendő lenne az összes ügyfél teljes hiteligényének egy időben való kielégítésére. A 3.13.(a) ábra négy ügyfelet tüntet fel; ezeket jelölje *A*, *B*, *C* és *D*, a számukra engedélyezett hitelegységek (például 1 egység 1 K dollár) számával. A bankár tudja, hogy nem lesz szüksége mindegyik ügyfélnek azonnal a maximális hitelre, így a kiszolgálásukra csak 10 egységet foglal le a 22 helyett. Továbbá bízik abban is, hogy minden ügyfél a teljes hitel felvételét követően amint lehet, azonnal visszafizeti a kölcsönt. (Itt az ügyfelek játsszák a processzusok szerepét, a szalagmeghajtó egységek az egységek szerepét, a bankár pedig az operációs rendszer szerepét.)

Az ábra mindegyik része az erőforrás-lefoglalások szerinti **rendszerállapot**ot mutatja, vagyis ügyfelenként mutatja a már kölcsönvett pénzt (a már birtokolt szalagmeghajtó egységek) és a maximálisan elérhető hitelt (a szalagmeghajtó egységek maximális száma, amennyire egyszer később szükség lesz). Egy állapot **biztonságos**, ha létezik ezzel kezdődő olyan állapotsorozat, amelynek eredményeként mindegyik ügyfél felvehet összesen annyi kölcsönt, amennyit a hitel lehetősége enged (minden processzus megkapja az összes erőforrását, és befejeződik).

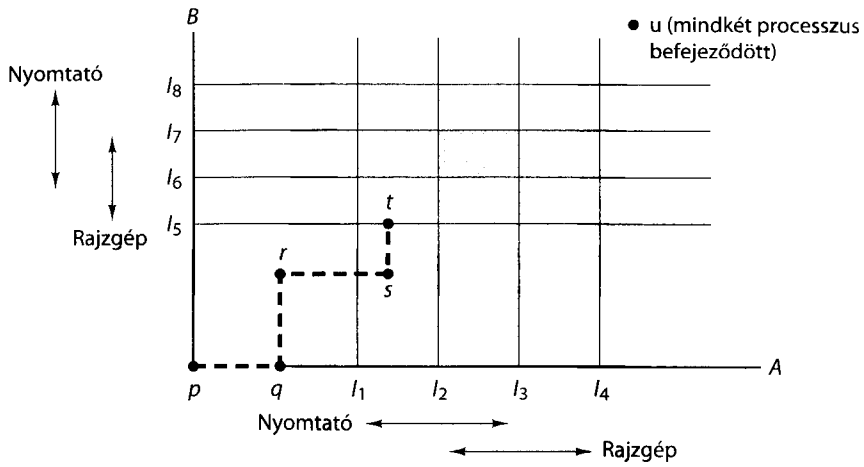
Az ügyfelek intézik a saját ügyeiket, időről időre kölcsönt (vagyis erőforrást) kérnek. Egy adott pillanatbeli helyzetet a 3.13.(b) ábra szemléltet. Ez az állapot biztonságos, mivel két egység megmaradt, és a bankár késleltetheti a kéréseket, kivéve a *C*-t, így engedi a *C* befejeződését, és elengedi a négy erőforrását. A bankár négy egységgel a kezében megengedheti *D*-nek vagy *B*-nek, hogy megkapja a kívánt egységeket, és így tovább.

Nézzük, mi lenne, ha történetesen *B* egy egységgel többet kérne, mint azt tette a 3.13.(b) ábra szerint. Akkor a 3.13.(c) ábrán lévő helyzet állna elő, amelyik bizonytalan állapotú. Ha mindegyik ügyfél a számára maximálisan engedélyezett kölcsönt akarná, akkor a bankár egyikőjüket sem tudná kielégíteni, és holtpont keletkezne. Egy bizonytalan állapot nem feltétlenül vezet holtponthoz, mivel az ügyfelek valószínűleg nem igénylik a teljes lehetséges hitelt, azonban a bankár nem számolhat ezzel a viselkedéssel.

A bankár algoritmus minden kérés megjelenésekor megnézi, hogy vajon engedélyezése biztonságos állapothoz vezet-e. Ha igen, akkor a kérést jóváhagyja, egyébként későbbre halasztja. Egy állapot biztonságos voltának eldöntésekor a bankár azt ellenőrzi, hogy van-e elegendő erőforrása ahhoz, hogy kielégítsen néhány ügyfelet. Ha ezt megteheti, akkor feltevés szerint ezek a kölcsönök visszafizetődnek, és következhet annak az ügyfélnek a vizsgálata, aki legközelebb van a hitellimitjéhez, és így tovább. Ha minden kölcsönt végül visszafizetnek, akkor az állapot biztonságos, és a kezdeti kérést ki lehet elégíteni.

Erőforrás-pályagörbék

Az előző algoritmus egyetlen erőforrásosztályra volt kitalálva (például csak szalagmeghajtó egységekre vagy csak nyomtatókra, de nem mindegyikből néhányra). A 3.14. ábrán egy olyan modell látható, amely két processzussal és két erőforrással foglalkozik, például egy nyomtatóval és egy rajzgéppel. A vízszintes tengelyen az



3.14. ábra. *Kétprocesszusú erőforrás-pályagörbék*

A processzus által végrehajtott parancsok számát ábrázoljuk. A függőleges tengelyen jelöljük a *B* processzus által végrehajtott parancsok számát. I_1-nél az *A* kér egy nyomtatót; I_2-nél kér egy rajzgépet. A nyomtatót I_3-nál, a rajzgépet I_4-nél engedi el. A *B* processzusnak I_5-től I_7-ig egy rajzgépre és I_6-tól I_8-ig egy nyomtatóra van szüksége.

A diagram minden pontja a két processzus közös állapotát jelöli. A kezdeti állapot a *p*, amelynél egyik processzus sem hajtott végre még parancsot. Ha az ütemező először az *A*-t futtatja, akkor a görbe a *q* pont felé mozdul, ekkorra *A* már néhány parancsot elvégez, de *B* még nem tett semmit. A *q* pontnál a görbe függőlegessé válik, jelezve, hogy az ütemező áttért a *B* futtatására. Egyetlen processzor esetén minden pályaszakasz vagy vízszintes, vagy függőleges, de sosem ferde. Továbbá a haladási irány mindig felfelé vagy jobbra, de soha nem lefelé vagy balra (a processzusok nem futtathatók visszafelé).

Amikor *A* keresztezi az I_1 vonalat az *r*-ből *s* felé tartó szakaszon, akkor kéri és megkapja a nyomtatót. Amikor *B* eléri a *t* pontot, akkor kéri a rajzgépet.

A bevonalkázott területek különösen érdekesek. A délnyugat–északkelet irányú vonalakkal satírozott terület jelöli azt, hogy mindkét processzusnak szüksége van a nyomtatóra. A kölcsönös kizárási szabály lehetetlenné teszi az ilyen területre történő belépést. A másik módon satírozott terület azt jelenti, hogy mindkét processzus foglalja a rajzgépet, ami szintén lehetetlen. Egyik feltétel esetén sem léphet a rendszer a bevonalkázott területekre.

Ha a rendszer belépne az I_1-gyel és I_2-vel, mint oldalakkal, az I_5-tel felülről és az I_6-tal alulról határolt területre, akkor végül is holtpontba jutna, mikor az I_2 és I_6 metszését elérné. Ennél a pontnál *A* kéri a rajzgépet és *B* kéri a nyomtatót, amelyek mindegyike már foglalt. Az egész terület állapota bizonytalan, vagyis nem szabad belelépni. A *t* pontnál csak egyetlen biztonságos dolog tehető: az *A* processzust kell futtatni az I_4-ig. Ezután már bármely görbe *u*-ba vezet.

Fontos látni, hogy a t pontnál a B kér egy erőforrást. A rendszernek el kell döntenie, hogy ezt elfogadja, vagy sem. Ha elfogadja, akkor a rendszer egy bizonytalan tartományba lép, és végül holtpontba kerül. A holtpont elkerüléséhez a B-t addig fel kell függeszteni, amíg az A kéri és elengedi a rajzgépet.

A bankár algoritmus többpéldányos erőforrástípusok esetén

A grafikus modell nehezen alkalmazható az olyan általános esetekben, ahol tetszőleges a processzusok száma és tetszőleges az erőforrások osztályainak száma is, sőt több példány lehet mindegyikből (például két rajzgép, három szalagmeghajtó egység). A bankár algoritmus általánosítható ilyen helyzetekre. A 3.15. ábra mutatja, hogyan dolgozik ilyenkor.

A 3.15. ábrán két mátrix van. A bal oldali mutatja, hogy az öt processzus jelenleg külön-külön hány erőforrást foglal le az egyes erőforrástípusokból. A jobb oldali mátrix azt mutatja, hogy mennyi erőforrásra van még igényük a befejeződéshez. Ahogy egyetlen erőforrás esetében, úgy itt is a processzusoknak a végrehajtásuk előtt közölniük kell a teljes erőforrásigényüket, hogy a rendszer képes legyen a jobb oldali mátrix lépésenkénti meghatározására.

Az ábra jobb oldalán három vektor van, E mutatja a meglévő erőforrásokat, P a foglalt, A pedig a szabad erőforrásokat. E-ből látható, hogy a rendszerhez hat szalagmeghajtó egység, három rajzgép, négy nyomtató és két CD-ROM-meghajtó egység tartozik. Közülük jelenleg öt szalagmeghajtó egység, három rajzgép, kettő nyomtató és kettő CD-ROM-meghajtó egység foglalt. Ugyanezt kapjuk a bal oldali mátrix négy erőforrásoszlopának összegzésével. A szabad erőforrásvektor egyszerűen a rendszerben lévő és a jelenleg használt erőforrások különbsége.

Egy állapot biztonságos voltának ellenőrzésére most a következő algoritmus alkalmazható.

Processzus	Szalagmeghajtó egységek	Rajzgépek	Nyomtatók	CD-ROM-ok
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Lefoglalt erőforrások

Processzus	Szalagmeghajtó egységek	Rajzgépek	Nyomtatók	CD-ROM-ok
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

További erőforrásigények

E = (6342)
P = (5322)
A = (1020)

3.15. ábra. *A bankár algoritmus többpéldányos erőforrástípusok esetén*

1. Megkeres egy olyan R sort a jobb oldali mátrixban, amely A-nál kisebb vagy egyenlő. Ha ilyet nem talál, akkor a rendszer végül holtpontba kerül, mivel egyik processzus sem tud végigfutni.
2. Tegyük fel, hogy a kiválasztott sorhoz tartozó processzus kéri az összes erőforrás-szükségletét (ez garantáltan teljesíthető) és befejeződik. Jelöljük ezt a processzust, mint lefutottat, és az összes erőforrását adjuk az A vektorhoz.
3. Ismételjük az 1 és 2 lépéseket, amíg vagy mindegyik processzus befejeződik, és ekkor a kezdő állapot biztonságos volt, vagy egy holtpont keletkezéséig, és ekkor az állapot bizonytalan.

Ha több processzus is megfelel az 1. lépés választásának, akkor azok bármelyike választható: az erőforráskészlet vagy bővül, vagy legrosszabb esetben változatlan marad.

Térjünk most vissza a 3.15. ábra példájához. A jelenlegi állapot biztonságos. Tegyük fel, hogy a B processzus most egy nyomtatót kér. Ez az igény kielégíthető, mert az eredményezett állapot továbbra is biztonságos (a D processzus befejeződhet, utána az A vagy E processzus, majd a többiek).

Most képzeljük el, hogy miután B megkapta a maradék két nyomtató egyikét, az E is kér egy nyomtatót, az utolsót. Ennek az igénynek a kielégítésekor a szabad erőforrások vektora (1000) lenne, ami holtponthoz vezet. Világos, hogy E igényét nem szabad azonnal kielégíteni, hanem kis időre el kell halasztani.

A bankár algoritmust először Dijkstra publikálta 1965-ben. Azóta szinte valamennyi operációs rendszerekkel foglalkozó könyvben részletesen le van írva. Számtalan cikket írtak különböző aspektusairól. Néhány szerző vakmerően rámutatott arra, hogy jóllehet elméletileg az algoritmus csodálatos, a gyakorlatban lényegében használhatatlan, mivel a processzusok csak ritkán tudják előre a maximális erőforrásigényüket. Továbbá a processzusok száma van amikor nem állandó, hanem dinamikusan változik, mivel új felhasználók léphetnek be és ki. Sőt erőforrások, amelyekre mint elérhetőkre számít a rendszer, hirtelen eltűnhetnek (a szalagmeghajtó egység meghibásodik). Így a gyakorlatban csak kevés olyan rendszer van, ha van egyáltalán, amely a bankár algoritmust alkalmazza a holtpont elkerülésére.

Összefoglalva, a „megelőzésként" korábban leírt séma túlságosan megszorító, az „elkerülés" nevű, itt leírt algoritmus pedig olyan információt kér, amely általában nem adható meg. Ha az olvasónak eszébe jutna egy – a gyakorlatban és az elméletben egyaránt jó – általános célú algoritmus, akkor írja le és küldje el a helyi számítástudományi folyóirathoz.

Jóllehet általános esetben sem az elkerülés, sem a megelőzés nem túl ígéretes, viszont különleges alkalmazásokra sok speciális célú algoritmus ismert. Erre példa az alábbi. Sok adatbázisrendszerben van olyan művelet, amely gyakran előfordul; ilyen azoknak a rekordoknak a zárolási kérése, amelyeket azután megváltoztat. Amikor egy időben több processzor fut, akkor a holtpontnak valódi veszélye van. A probléma megszüntetésére speciális technikákat alkalmaznak.

Az egyik gyakran használt megközelítés **kétfázisú zárolás**ként ismert. Az első fázisban a processzus megpróbálja az összes szükséges rekordot egyesével zárolni.

Ha sikerül, akkor beindul a második fázis, elvégzi a módosításait, és elengedi a zároltakat. Az első fázisban ténylegesen nincs munkavégzés.

Ha az első fázisban olyan rekordra van szükség, amely már zárolt, akkor a processzus elengedi az összes általa lefoglaltat, és mindent kezd elölről. Bizonyos értelemben ez a megközelítés hasonló az összes szükséges erőforrásigény előre – vagy legalábbis azt megelőzőleg – való kéréséhez, mielőtt valami visszavonhatatlan történne. A kétfázisú zárolás néhány változatánál nincs elengedés és újrakezdés, ha az első fázisban egy zárolás jelenik meg.

Akárhogyan is, de ez a stratégia általában nem alkalmazható. A valós idejű rendszerekben és a processzusirányító rendszerekben például nem fogadható el az, hogy egy processzus befejeződjön, mert egy erőforrás nem elérhető, és mindent kezdjen újra. Egy olyan processzusnak az újraindítása sem fogadható el, amely már olvasott vagy írt üzenetet a hálózatra, állományokat módosított vagy bármi olyat tett, ami biztonságosan nem ismételhető meg. Az algoritmus csak azokban a helyzetekben dolgozik jól, ahol a programozó nagy alapossággal megoldott olyan kérdéseket, hogy a program az első fázis alatt tetszőleges pontban megállítható és újraindítható legyen. Sok alkalmazás azonban nem strukturálható így.

3.4. A MINIX 3 I/O áttekintése

A MINIX 3 I/O szerkezeti felépítését a 3.8. ábra szemlélteti. Az ott szereplő felső négy réteg megfelel a 2.29. ábrán levő MINIX 3 négyrétegű szerkezetének. A következő részekben röviden megnézzük mindegyik réteget, hangsúlyt helyezve az eszközmeghajtókra. A megszakítási kérések kezelésével a 2. fejezet foglalkozott, az eszközfüggetlen I/O-t pedig az 5. fejezetben a fájlrendszerekkel kapcsolatban tárgyaljuk.

3.4.1. Megszakításkezelők és I/O-elérés a MINIX 3-ban

Sok eszközmeghajtó elindít néhány I/O-eszközt, és utána blokkolódik, várakozva egy bejövő üzenetre. Ezt az üzenetet általában a megszakításkezelő generálja az eszköznek. Más eszközmeghajtók nem indítanak el semmi fizikai I/O-t (például egy RAM-lemezről olvasás és írás egy tárcímleképezéses megjelenítőre), nem használnak megszakításokat, és nem várakoznak I/O-eszköztől jövő üzenetre. Az előző fejezetben részletesen ismertettük azt a kernelbeli módszert, amellyel a megszakítások üzenetet generálnak és taszkváltást idéznek elő, így itt nem foglalkozunk ezzel. Általánosan megnézzük a megszakításokat és az I/O-t az eszközmeghajtóknál. A részletekre pedig akkor térünk vissza, amikor a különböző eszközök kódjait vizsgáljuk.

Lemezeszközöknél a bevitel és kivitel általában egyrészt parancsadás egy eszköznek, hogy hajtsa végre a műveletét, másrészt a művelet befejezéséig való várakozás. A lemezvezérlő végzi a munka nagy részét, és csak kevés hárul a megszakításkezelőre. Egyszerű lenne az élet, ha minden megszakítás ilyen könnyen kezelhető lenne.

Néha az alacsony szintű kezelőnek van több munkája. Az üzenetküldéses módszernek ára van. Amikor egy megszakítás gyakran előfordulhat, de megszakításonként az I/O mennyisége kevés, akkor lehet, hogy kifizetődőbb a kezelőt többet dolgoztatni, elhalasztani a meghajtónak menő üzenetküldést több megszakításkérés beérkezéséig, amikor már több munkája lenne a meghajtónak. A MINIX 3-ban a legtöbb I/O esetében ez nem lehetséges, mivel a kernelbeli alacsony szintű kezelőként alkalmazva majdnem minden eszköz esetén egy általános célú eljárás van.

Az utolsó fejezetben láttuk, hogy az óra kivételt jelent. Mivel a kernelbe van fordítva, az órának van saját kezelője az extra munkákhoz. Sok óraütéskor nagyon kevés a tennivaló az idő kezelésén kívül. Ez viszont elvégezhető az időzítőtaszkhoz üzenetküldés nélkül is. Az óra-megszakításkezelő növeli a találóan *realtime* nevű változó értékét, lehetőleg egy BIOS-hívás alatti ütések számával. A kezelő néhány járulékos, nagyon egyszerű számolást végez – növeli a felhasználói idő és a számlázási idő számlálóját, csökkenti az aktuális processzus *ticks_left* számlálóját, teszteli, vajon van-e lejárt időzítő. Üzenetet csak akkor küld az időzítőtaszknak, ha azt észleli, hogy az aktuális processzus felhasználta az időegységét, vagy egy időzítés lejárt.

Az óra-megszakításkezelő a MINIX 3-nál egyedi, mivel az óra az egyedüli megszakítással vezérelt eszköz, amely a kernelben fut. Az órahardver be van építve a PC-be – valójában az óramegszakítás-vonal nincs összekapcsolva egyik bővítő I/O-vezérlő beillesztési pontjával sem –, így nem lehetséges egy órát módosító csomag telepítése, helyettesítve az órahardvert és a gyártó által készített meghajtót. Ennek az az oka, hogy az órameghajtó fordításának a kernelben kell lennie, és a kernelben levő bármely változóból elérhetőnek kell lennie. A MINIX 3 egyik tervezési szempontja volt, hogy a többi eszközmeghajtó számára az elérésnek ez a típusa ne legyen szükségszerű.

A felhasználói szinten futó eszközmeghajtók nem tudják közvetlenül elérni a kernel helyét vagy az I/O-kapukat. Jóllehet lehetséges, de megsértené a MINIX 3 tervezési elveit, ha egy megszakítást kiszolgáló eljárás olyan távoli hívást tudna tenni, amelynek hatására végrehajtódna egy felhasználói processzus kódrészében levő szolgáltatóeljárás. Ez még annál is veszélyesebb lenne, mintha megengednénk, hogy felhasználói szintű processzus hívjon egy kernelben levő függvényt. Ebben az esetben legalább biztosak lehetnénk abban, hogy a függvény írója egy hozzáértő, az operációs rendszer biztonságára figyelő tervező, vagy lehetőleg olyan valaki, aki ezt a könyvet olvasta. A kernel kódjai nem lennének megbízhatók, ha a felhasználói programok hozzáférnének.

Az I/O-elérésnek számos olyan különböző szintje van, amely egy felhasználói szintű eszközmeghajtó számára szükséges lehet.

1. Egy meghajtónak a szokásos adathelyen kívül eső memóriaelérésre lehet szüksége. A csak ilyen típusú elérést igénylő meghajtóra példa a RAM-lemezt kezelő memóriameghajtó.

2. Egy meghajtó olvashatja és írhat az I/O-kapura. Az ilyen műveletekhez a gépi szintű parancsok csak kernel módban alkalmazhatók. Ahogy hamarosan látni fogjuk, a merevlemez-meghajtónak szüksége van ilyen elérésre.
3. Egy meghajtónak lehet, hogy előre látható megszakításokat kell megválaszolnia. Például a merevlemez-meghajtó a lemezvezérlőnek ír parancsokat, amelyek megszakítást okoznak a tervezett művelet befejeződésekor.
4. Egy meghajtónak lehet, hogy előre nem látható megszakításokat kell megválaszolnia. A billentyűzetmeghajtó ebbe a kategóriába sorolható. Az előző eset egy részosztályának tekinthető azzal, hogy kiszámíthatatlanul bonyolítja a dolgokat.

Az összes esetet a kernelhívások támogatják, és a rendszertaszk kezeli.

Az első eset az extra memóriaszegmens-elérés, kihasználja az Intel processzorok által biztosított hardverszegmentációt. Bár egy szokásos processzus csak a saját kód-, adat- és veremszegmenseihez fér hozzá, a rendszertaszk megengedi, hogy más szegmensek definiálhatók és elérhetők legyenek a felhasználói szintű processzusokból. Így a memóriameghajtó a RAM-lemezen levő memóriaterületet eléri, valamint más területeket más speciális célból. A konzolmeghajtó hasonlóan eléri a videomegjelenítő adapternek a memóriáját.

A második esetre a MINIX 3 kernelhívásokat biztosít az I/O-parancsok használatához. A rendszertaszk végzi az aktuális I/O-t egy kevésbé privilegizált processzus helyett. A fejezetben később megnézzük, hogyan használja ezt a szolgáltatást a merevlemez-meghajtó. Itt most csak a téma bevezetésére kerül sor. A lemezmeghajtónak lehet, hogy írnia kell egy egyszerű kiviteli kapura, hogy kiválasszon egy lemezt, ezután olvasnia kell egy másik kapuról azért, hogy az eszköz készenlétéről megbizonyosodjon. Amennyiben normálisan nagyon gyors válaszadás várható, akkor ez lekérdezéssel végezhető. Vannak kernelhívások, amelyek specifikálnak egy kaput és a kiírandó adatot vagy a beolvasott adat számára egy helyet. Ezek megkövetelik, hogy egy kaput olvasó hívás ne blokkolódjon, vagyis ténylegesen a kernelhívások ne blokkoljanak.

Némi biztosítás hasznos lehet az eszközhibákkal szemben. Egy lekérdező ciklusban lehet olyan számláló, amely befejezi a ciklust, ha az eszköz nem áll készen bizonyos számú iteráció után. Ez általában nem jó elképzelés, mivel a ciklus futási ideje függeni fog a CPU gyorsaságától. Egy lehetőség ezzel kapcsolatban a számlálónak CPU-időhöz kötődő értékkel való indítása, lehetőleg egy globális változó alkalmazásával, amelynek inicializálása a rendszer indításakor történik. A MINIX 3-rendszerkönyvtár jobb módszert nyújt a *getuptime* függvény révén. Ez egy kernelhívással hozzáfér a rendszerindítás óta eltelt órajelek számát tartalmazó számlálóhoz, amelyet az időzítőtaszk kezel. Ennek az információnak azonban ára van, a ciklusban eltelt időt nyomon kell követni, ami minden egyes iterációs lépésben egy további kernelhívást jelent. Egy másik lehetőség megkérni a rendszertaszkot egy figyelőóra beállítására. Azonban egy receive blokkoló művelet szükséges ahhoz, hogy egy órától információt lehessen elérni. Amennyiben gyors válaszidőre van szükség, akkor ez nem jó megoldás.

A merevlemez szintén változatos kernelhívásokat alkalmaz az I/O-hoz, ami lehetővé teszi, hogy kapuk, kiírandó adatok és módosítandó változók listáját küldje a rendszertaszknak. Hasznos tudni, hogy a merevlemez-meghajtó egy bájtsorozat írását kéri a hét kimeneti kapura, hogy elindítson egy műveletet. Az utolsó bájtja a sorozatnak egy parancs, és a lemezvezérlő, amikor egy parancsot befejez, egy megszakítást generál. Mindezt egy egyszerű kernelhívás kíséri, nagyban csökkentve a szükséges üzenetek számát.

Ezzel rátérünk a lista harmadik pontjára: egy váratlan megszakítás megválaszolása. Miként a rendszertaszk vizsgálatánál megjegyeztük, amikor egy felhasználói szintű program (egy sys-irqctl kernelhívást alkalmazva) érdekében egy megszakítás elkezdődik, akkor a megszakításkezelő rutinja mindig a *generic_handler,* a rendszertaszk részeként definiált függvény. Ez a rutin a megszakítást egy értesítő üzenetté alakítja át a megszakítást kiváltó processzus számára. Az eszközmeghajtónak egy receive műveletet kell elindítania a kernelhívás után, amely kiadja a parancsot a vezérlőnek. Amikor az értesítés megérkezik, az eszközmeghajtó végre tudja hajtani a megszakítás kiszolgálását.

Ebben az esetben egy megszakítás körültekintően védve van, ha valami balul ütne ki. Arra, hogy egy várt megszakítás nem következik be, egy processzus úgy készül fel, hogy a rendszertaszkot egy időzítő beállítására kéri meg. Az időzítők szintén értesítő üzeneteket generálnak, és így a receive művelet értesítést kaphat, ha megszakítás fordult elő vagy egy időzítő lejárt. Bár az értesítés nem szállít sok információt, mégsem okoz problémát, mert az értesítő üzenet jelzi az eredetét. Bár mindkét értesítést a rendszertaszk generálja, egy megszakítás értesítőjében benne lesz, hogy a *HARDWARE*-ből jön, az idő lejártáról pedig az értesítőben megjelenik, hogy a *CLOCK*-ból jön.

Egy másik probléma is van. Ha egy megszakítás időzített módon történik, és az időzítő beállított, akkor egyszer a jövőben az időtartam elhasználását egy másik receive művelet észleli, valószínűleg a meghajtó főciklusában. Egyik megoldás egy kernelhívással megbénítani az órát, amikor a *HARDWARE*-ből az értesítés megjön. Vagy ha valószínűleg a következő a receive művelet lesz, ahol a *CLOCK*-ból nem vár üzenetet, akkor ki kellene hagyni, és ismét a receive-et hívni. Bár kevésbé valószínű, de elképzelhető, hogy a lemezművelet egy váratlanul hosszú késés után következik be, egy megszakítást generálva az időzítő lejárta után. Az előálló helyzet többféleképpen kezelhető. Amikor időtúllépés lép fel, egy kernelhívás megbénítja a megszakítást, vagy egy olyan receive művelet indul, amely nem vár megszakítást, és figyelmen kívül hagyja a *HARDWARE*-ből jövő üzenetet.

Alkalmas itt megjegyezni, hogy amikor egy megszakítás először engedélyezett, egy kernelhívás beállítja a megszakítással kapcsolatos „szabályokat". A szabály egyszerűen egy állapotjelzővel valósul meg, amely meghatározza, hogy a megszakítás automatikusan újra engedélyezhető-e vagy addig nem lehetséges, amíg az eszközmeghajtó, amelyet kiszolgál, egy kernelhívással újra nem engedélyezi. Lehetséges, hogy egy megszakítást követően az eszközmeghajtónak tekintélyes mennyiségű munkát kell elvégeznie, és így lehet, hogy legjobb a megszakítást addig nem engedni, amíg az összes adat másolása el nem készül.

A listánk negyedik esete a legproblémásabb. A billentyűzettámogatás a tty meghajtó része, amely egyaránt ellátja a kivitelt és a bevitelt is, továbbá párhuzamos eszközöket is képes támogatni. Így az input jöhet egy lokális billentyűzetről, de jöhet egy soros vonalon csatolt vagy hálózati csatolású távoli felhasználótól is. Számos olyan processzus futása is lehetséges, amelyek mindegyike különböző helyi vagy távoli terminálra produkál kimenetet. Mikor nem ismert, hogy egy megszakítás mikor fordul elő, ha egyáltalán előfordul, képtelenség egy receive hívás blokkolása, hogy egy egyszerű erőforrásból az input elérhető legyen, ha ugyanannak a processzusnak kell esetleg válaszolnia más beviteli vagy kiviteli erőforrásoknak is.

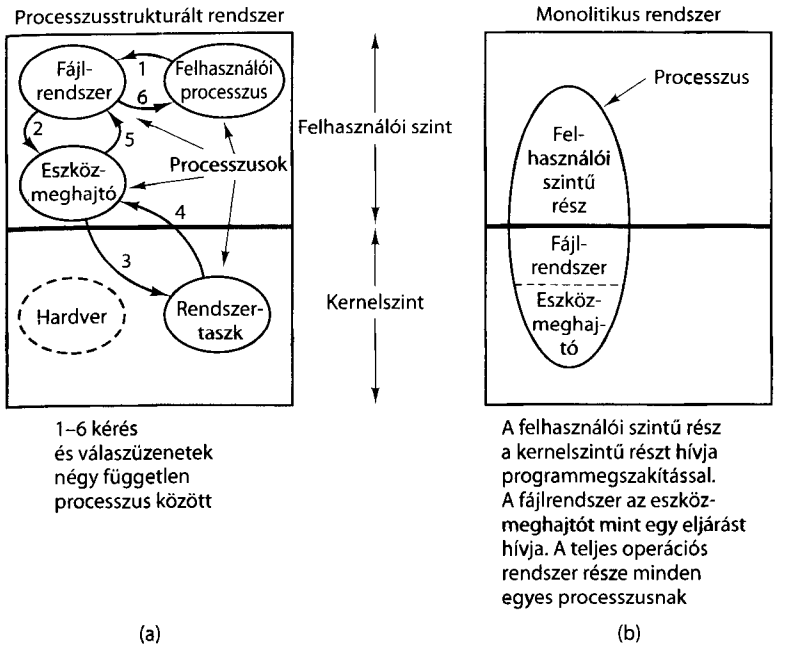
A MINIX 3 ennek a problémának a kezelésére számos technikát alkalmaz. A terminálmeghajtók által használt alapvető technika a billentyűzetbemenetnél megszakításkezeléssel történik, amilyen gyorsan csak lehetséges, így a karakterek nem vesznek el. Egy karakternek a billentyűzethardverből egy pufferbe való írása a lehető legkevesebb munkával történik. Továbbá amikor a billentyűzetről az adat a megszakításnak eleget téve átkerül, amint az adat pufferezve van, a billentyűzetről ismét olvas, még mielőtt azt megelőzően, hogy a megszakításból visszatérne. A megszakítások olyan értesítő üzeneteket generálnak, amelyek a küldőt nem blokkolják; ez segít megelőzni az input elvesztését. A nem blokkoló receive művelet szintén alkalmazható, bár ezt csak rendszerösszeomlás során használják az üzenetek kezelésére. A jelzőórákat olyan eljárások aktiválására is használják, amelyek a billentyűzetet ellenőrzik.

3.4.2. A MINIX 3 eszközmeghajtói

Egy MINIX 3-rendszerben meglévő I/O-eszközök mindegyik osztályához egy különálló I/O-eszközmeghajtó tartozik. Ezek a meghajtók teljesen önálló processzusok saját hellyel, regiszterekkel, veremmel, és így tovább. Az eszközmeghajtók, a MINIX 3-processzusok által használt standard üzenetküldéses módszerrel érintkeznek a fájlrendszerrel. Az egyszerű eszközmeghajtót valószínűleg önálló forrásfájlokként írták. A RAM-lemez, a merevlemez és a hajlékonylemez meghajtóinál minden eszköztípushoz van egy forrásfájl, és a *driver.c*-ben és a *drvlib.c*-ben vannak a közös rutinok az összes blokkos eszköztípus támogatására. A szoftver, hardverfüggő és hardverfüggetlen részekre történő felosztása révén, könnyen alkalmazható a különböző hardverkonfigurációknál. Bár néhány közös forráskódot is használnak, mégis mindegyik lemeztípus meghajtója önálló processzusként fut, hogy gyors adatmozgatást tudjon megvalósítani, és elkülönítse a meghajtókat egymástól.

A terminálmeghajtó forráskódja hasonló módon szervezett, a *tty.c* állományban van a hardverfüggetlen kódja és külön állományban a különböző eszközöket támogató forráskód – úgymint a tárra leképező konzolok, a billentyűzet, a soros vonal és az egyéb végberendezések. Ebben az esetben azonban egyetlen processzus támogatja az összes különböző eszköztípust.

Az olyan eszközcsoportoknál, mint a lemezes eszközök és a terminálok, amelyekhez különféle forrásfájl tartozik, vannak definíciós fájlok. A *driver.h* az összes



3.16. ábra. *A felhasználó–rendszer közötti kommunikáció strukturálásának két módja*

blokkos eszközmeghajtót támogatja. A *tty.h* az összes terminéleszköznek a közös definícióját nyújtja.

A MINIX 3-ban a teljesen elkülönülő processzusok tervezési elve az operációs rendszer komponenseinek futtatásával kapcsolatban erősen modulált és mérsékelten hatékony. Ebben a MINIX 3 és a Unix lényegesen különbözik. A MINIX 3-ban egy processzus úgy olvas egy állományt, hogy üzenetet küld a fájlrendszerprocesszusnak. A fájlrendszer ekkor üzenetet küld a lemezmeghajtónak, amelyben kéri, hogy az igényelt blokkot olvassa be. A lemezmeghajtó kernelhívással kéri a rendszertaszkot, hogy végezze el az aktuális I/O-t, és másolja az adatot a processzusok között. A 3.16.(a) ábra ezt a folyamatot szemlélteti (a valóság némi egyszerűsítésével). Mivel üzeneteken keresztül mennek végbe ezek a belső tevékenységek, a rendszer különböző részei rákényszerülnek arra, hogy szabványos módon kapcsolódjanak a többi részhez.

A Unixban minden processzusnak két része van: egyik a felhasználói memóriarészben, a másik a kernel memóriaterületén, ahogy ez a 3.14.(b) ábrán látható. Egy rendszerhíváskor az operációs rendszer valami bűvös módon átkapcsol a felhasználói szintű részből a kernelszintűre. Ez a szerkezet a MULTICS-tervezés maradványa, amelyben a kapcsolás csupán szokásos eljáráshívás volt, nem pedig megszakítás, amelyet a felhasználói rész állapotának mentése követ, ahogy az a Unixban van.

Az eszközmeghajtók a Unixban egyszerű kerneleljárások, amelyek a processzszus kernelszintű részéből hívódnak. Amikor a processzusnak egy meghajtóra van

szüksége, akkor egy kerneleljárást hív, ami többek között alvó helyzetbe teszi, és addig alszik, amíg egy megszakításkezelő felébreszti. Megjegyezzük, hogy csak a felhasználói processzus kerül alvó állapotba; a kernelszintű és a felhasználói szintű részek a processzusnak ténylegesen különböző részei.

Az operációs rendszer tervezői szűnni nem akaró vitát folytatnak a monolitikus rendszerek, mint a Unix, és a processzusstrukturált rendszerek, mint a MINIX 3, érdemeiről. A MINIX 3-megközelítés jobban strukturált (modulárisabb), világosabb a részek közötti kapcsolat, és könnyen átvihető osztott rendszerekre, amelyekben a processzusok különböző számítógépen futnak. A Unix megközelítése hatékonyabb, mivel az eljáráshívások sokkal gyorsabbak az üzenetküldéseknél. Sok esetben a MINIX 3-at elvetették, de úgy gondoljuk, hogy mivel növekszik a hatékony személyi számítógépek elérhetősége, így az áttekinthetőbb szoftverszerkezet is egyre fontosabb, még akkor is, ha a rendszer lassúbbá válik általa. A teljesítményveszteség 5–10%-a tulajdonítható a felhasználói szintű operációs rendszer futtatásának. Természetesen sok operációsrendszer-tervező nem osztja azt a nézetet, hogy érdemes áldozni egy kis sebességet nagyobb modularitásért és jobb rendszer-megbízhatóságért.

Ebben a fejezetben a RAM-lemez-, a merevlemez-, az óra- és a terminálmeghajtókat tárgyaljuk. Egy szabványos MINIX 3-konfigurációhoz hozzátartoznak a hajlékonylemez- és nyomtatómeghajtók, de ezekkel nem foglalkozunk részletesen. Vannak MINIX 3-szoftverek egyéb eszközmeghajtók számára is, ilyenek az RS–232 soros vonal, a CD-ROM, az Ethernet-adapter és a hangkártyák. Ezek a MINIX újrafordításával ágyazhatók be, külön fordíthatók és bármikor indíthatók.

Ezeknek a meghajtóknak a MINIX 3-rendszer más részeivel való kapcsolata azonos módon valósul meg: a kérésüzenetek a meghajtókhoz vannak küldve. Az üzenetek több mezőt is tartalmaznak, rendszerint a művelet kódját (például READ vagy WRITE) és ezek paramétereit. A meghajtó megkísérli a kérés teljesítését, és utána visszaküld egy választ.

Kérések		
Mező	**Típus**	**Jelentés**
m.m_type	int	A kért művelet
m.DEVICE	int	A használandó másodlagos eszköz
m.PROC_NR	int	Az I/O-t kérő processzus
m.COUNT	int	A bájtok száma vagy az IOCTL kód
m.POSITION	long	Hely az eszközön
m.ADDRESS	char*	Cím a kérő processzusban

Válaszok		
Mezők	**Típus**	**Jelentés**
m.m_type	int	Mindig DRIVER_REPLY
m.REP_PROC_NR	int	Ugyanaz, mint a kérésben a PROC_NR
m.REP_STATUS	int	A mozgatott bájtok száma vagy hibaszám

3.17. ábra. *A fájlrendszer által a blokkos eszközmeghajtókhoz küldött üzenetek és a visszaküldött válaszok mezői*

Blokkos eszközöknél a kérés- és válaszüzenetek mezőit a 3.17. ábra mutatja. A kérésüzenet tartalmazza annak a pufferterületnek a címét, ahonnan az adat kikerül, vagy ahová a beérkező adat elhelyezésre kerül a kérésnek megfelelően. A válaszban van állapotinformáció, ezt használva a kérő processzus ellenőrizni tudja, hogy a kérése valóban teljesült-e, vagy nem. A karakteres eszközök mezői alapvetően hasonlók, de meghajtónként kissé különbözhetnek egymástól. A terminálmeghajtóhoz jövő üzenetek tartalmazhatják egy adatszerkezet címét, ami specifikálja a terminál összes kialakítható működési módját, például azokat a karaktereket, amelyeket soron belüli karaktertörlésre, sortörlésre használ.

Mindegyik meghajtónak feladata a többi processzustól, normálisan a fájlrendszertől jövő kérések fogadása és azok végrehajtása. A blokkos eszközök meghajtóit úgy írták meg, hogy azok üzenetet kapnak, végrehajtják azt, és választ küldenek. Ez a strukturális döntés többek között azt jelenti, hogy ezek a meghajtók szigorúan szekvenciálisak, nem végeznek semmiféle belső multiprogramozást, hogy megőrizzék egyszerűségüket. Egy hardverkérés kiadásakor a meghajtó végez egy receive műveletet, amivel jelzi, hogy csak megszakítási kérések fogadásával foglalkozik, nem érdekli semmiféle új munkára vonatkozó kérés. Valamennyi új kérőüzenetnek várnia kell, amíg a jelenlegi munkáját be nem fejezi (randevú elv). A terminálmeghajtó némileg más, mivel itt egyetlen, számos más eszközt kiszolgáló meghajtóról van szó. Így lehetséges, hogy elfogad egy új kérést, amely billentyűzetről kér bevitelt, mialatt még nem fejezte be a soros vonalról való olvasási kérés teljesítését. De az eszközöknek be kell fejezniük egy kérést, mielőtt egy új kérést elkezdenének teljesíteni.

```
message mess;                                   /* üzenetpuffer */

void io_driver() {
   initialize()                                 /* csak a kezdeti beállításkor végzi el */
   while (TRUE) {
      receive(ANY, &mess);                      /* várakozik egy feldolgozandó kérésre */
      caller = mess.source;                     /* a processzus, ahonnan az üzenet jött */
      switch(mess.type) {
         case READ:     rcode = dev_read(&mess); break;
         case WRITE:    rcode = dev_write(&mess); break;
         /* egyébként; az OPEN, a CLOSE és az IOCTL esetében is */
          default:      rcode = ERROR;
      }
      mess.type = DRIVER_REPLY;
      mess.status = rcode;                      /* eredmény kódja */
      send(caller, &mess);                      /* válaszüzenetet küld a hívónak */
   }
}
```

3.18. ábra. *Egy I/O-eszközmeghajtó főeljárásának vázlata*

A blokkos eszközök főprogramja szerkezetileg azonos; ez a 3.18. ábrán van vázolva. A rendszer első elindításakor valamennyi meghajtó elindítására sor kerül, hogy a belső táblázatok és hasonló dolgok kezdeti beállításai megtörténjenek. Ezután mindegyik eszközmeghajtó blokkolódik, míg egy üzenetet nem kap. Egy üzenet beérkezésekor a hívó azonosítóját elmenti, és meghív egy eljárást a munka elvégzésére; más-más eljárást hív az egyes műveletek elvégzésére. A munka elvégzése után egy választ küld a hívónak, majd a meghajtó visszatér a ciklus elejéhez, vagyis várakozik a következő kérésre.

A *dev_XXX* eljárások mindegyike a meghajtó által nyújtott műveletek valamelyikét kezeli. A történteket tükröző állapotkódot adnak vissza. Ez a kód a válaszüzenet *REP_STATUS* mezőjébe kerül, és értéke az átmozgatott bájtok száma (nulla vagy pozitív), ha minden jól ment; hiba esetén pedig a hibakód (negatív). Az érték különbözhet a kért bájtok számától. Egy állomány végénél a kért mennyiségnél kevesebb is lehet az elérhető bájtok száma. A terminálóknál legfeljebb egy sor adódik vissza (kivéve a nyers módot), még ha a kért mennyiség nagyobb is.

3.4.3. Eszközfüggetlen I/O-szoftver a MINIX 3-ban

A MINIX 3-ban az összes eszközfüggetlen I/O-kódot a fájlrendszerprocesszus tartalmazza. Az I/O-rendszer annyira szorosan kapcsolódik a fájlrendszerhez, hogy egyetlen összefésült processzusban vannak. A fájlrendszer tevékenységeit a 3.6. ábrán soroltuk fel, kivéve a monopol módú eszközök kérését és elengedését, amelyek a MINIX 3-ban nem lehetségesek a jelenlegi szerkezetben. Ezek azonban könnyen hozzáadhatók az aktuális eszközmeghajtókhoz, ha erre a jövőben igény merülne fel.

A meghajtókkal, pufferezéssel és blokklefoglalásokkal kapcsolatos kapcsolódási felületek kezelésén túl a fájlrendszer foglalkozik az i-csomópontok, könyvtárak és logikailag felcsatolt fájlrendszerek védelmével és kezelésével. Részletesen erre az 5. fejezet tér ki.

3.4.4. Felhasználói szintű I/O-szoftver a MINIX 3-ban

Ebben a fejezetben a korábban már felvázolt általános modellt alkalmazzuk. Könyvtári eljárások vannak rendszerhívásokhoz és az összes olyan C függvényhez, amelyeket a POSIX szabvány megkövetel, ilyenek a formázott beviteli és kiviteli függvények, a *printf* és *scanf*. A standard MINIX 3-konfigurációban van egy háttértáras démon, az lpd, amely az lp paranccsal átadott állományokat tárolja és nyomtatja. A szabványos MINIX 3-szoftvertermékek tartalmaznak olyan démonokat is, amelyek számos hálózati funkciót támogatnak. Ebben a könyvben leírt MINIX 3-konfiguráció támogatja a legtöbb hálózati műveletet, az összes olyat, ami ahhoz kell, hogy indításkor elérhetők legyenek a hálózati szerverek és meghajtók az Ethernet-adapterek számára. A terminálmeghajtó újrafordítása pszeudo-terminálokkal és soros vonalak alkalmazásával bővíti a távoli terminálokról és hálózatról soros vonalakon (beleértve a modemeket) történő bejelentkezéseket. A hálózati műveletek igényelnek némi operációs rendszertől jövő segítséget, ami az ebben a könyvben leírt konfigurációjú MINIX 3-nak nem része, de a MINIX 3-ba a hálózati szerver könnyen befordítható. A hálózati szerver azonos prioritással fut a tárkezelővel és a fájlrendszerrel, és azokhoz hasonlóan úgy fut, mint egy felhasználói processzus.

3.4.5. Holtpontkezelés a MINIX 3-ban

Hagyományainak megfelelően a MINIX 3 ugyanazt az utat követi, mint a Unix a holtponttal kapcsolatban, vagyis figyelmen kívül hagyja ezt a problémát.

Rendszerint a MINIX 3-ban nincsenek monopol módú I/O-eszközök, de ha valaki egy ipari szabványú DAT szalagmeghajtó egységet akarna a PC-hez csatolni, az ehhez szükséges szoftver elkészítése nem okozna különösebb problémát. Röviden, a holtpont egyedül csak az impliciten megosztott erőforrásokkal kapcsolatban fordul elő, ilyen erőforrások a processzustáblázat szabad helyei, az i-csomópont táblázat szabad helyei stb. Az ismert holtpont algoritmusok nem tudják az ilyen erőforrásokat kezelni, vagyis azokat, amelyek kérése nem explicit.

Valójában, a fentiek nem feltétlenül igazak. Az egy dolog, hogy a felhasználói processzusoknál elfogadjuk a holtpont-kialakulás kockázatát, sajnos azonban magában az operációs rendszerben is van néhány hely, ahol tekintélyes gondot okoz az elkerülése. Fő előfordulási helye a processzusok közötti belső üzenetátadásnál van. Például a felhasználói processzusoknak csak a sendrec üzeneti módszer alkalmazása engedett, így egy felhasználói processzusnak soha nem kellene zárolva lennie, mivel ez egy receive művelet, amikor nincs hozzá üzenetet küldő processzus. A szerverek csak a send vagy a sendrec műveletet használják az eszközmeghajtóval való kommunikálásra, és az eszközmeghajtók csak a send vagy a sendrec műveletet használják a rendszertaszkkal a kernelrétegben való kommunikáláshoz. Ritka eset az, ahol a szervereknek egymás között kell kommunikálniuk, olyan, mint váltások a processzuskezelő és a fájlrendszer között, amikor beállítják a saját processzustáblarészeiket, ilyenkor a kommunikáció rendje nagyon körültekintően megtervezett a holtpont elkerülése miatt. Az üzenetátadásnak szintén a legalacsonyabb szintjénél megy végbe annak a biztosítása, hogy ha egy processzus valamit küld, akkor a célprocesszus ne tehesse ugyanazt.

A fenti megszorításokon túl a MINIX 3-ban egy új notify üzenetprimitív látja el azoknak a helyzeteknek a kezelését, amelyekben egy üzenetet kell küldeni „felfelé" (upstream) irányban. A notify blokkolódik, és az értesítés tárolódik, amikor a címzett közvetlenül nem elérhető. Ebben a fejezetben a MINIX 3-eszközmeghajtók vizsgálatánál látni fogjuk, hogy a notify használata jelentős.

A másik módszer, amivel a holtpont megelőzhető, a zárolás. Az eszközök és állományok zárolása az operációs rendszer támogatása nélkül is lehetséges. Egy állománynév úgy viselkedik, mint egy helyes globális változó, amelynek a jelenlétét vagy hiányát az összes többi processzus érzékelheti. A MINIX 3-rendszerben és a legtöbb Unix-rendszerben is általában van egy */usr/spool/locks/* nevű speciá-

lis könyvtár, amelybe a processzusok elteszik a **zárolási állományok**at, megjelölve így minden használatban levő erőforrást. A MINIX 3-fájlrendszer a POSIX-féle állományzárolást is támogatja. De ezeknek a mechanizmusoknak egyike sem kényszeríthető ki. Csak a processzusok helyes viselkedésében lehet bízni, ugyanis nincs semmi megelőző védelem arra, hogy egy program megpróbáljon használni egy másik processzus által zárolt erőforrást. Ez a módszer nem azonos az erőforrások megszakított használatával, mivel nem gátolja meg az első processzust, hogy folytassa az erőforrás használatát. Más szavakkal, nem érvényesül a kölcsönös kizárás feltétel. Egy helytelenül viselkedő processzus ilyen ténykedésének eredménye valószínűleg valami zagyvaság, de mindenesetre nem lesz holtpont.

3.5. Blokkos eszközök a MINIX 3-ban

A MINIX 3 különféle blokkos eszközöket támogat, így először ezek közös jellemzőit vesszük szemügyre. Ezt követi a RAM-lemez, a merevlemez és a hajlékonylemez vizsgálata. Mindegyik más okból érdekes. Vizsgálódásainkhoz példaként nagyon jó a RAM-lemez, mivel rendelkezik a blokkos eszközök minden tulajdonságával általában, a tényleges I/O-t kivéve – ugyanis ez a „lemez" valójában a memória egy része. Egyszerűsége alkalmassá teszi, hogy vele kezdjünk. A merevlemeznél látható, hogy milyen egy valódi lemezmeghajtó. Azt várhatnánk, hogy a hajlékonylemez támogatása könnyebb, mint a merevlemezé, ez azonban nem így van. A hajlékonylemez minden részletének vizsgálatára nem térünk ki, de rámutatunk számos, a hajlékonylemez-meghajtóban lévő bonyodalomra.

Előretekintve, a blokkos meghajtók vizsgálata után foglalkozunk a terminál- (billentyűzet + megjelenítő) meghajtóval, ami fontos tartozéka minden rendszernek és jó példa a karakteres eszközmeghajtókra.

Az egyes alfejezetekben a megfelelő hardvert és a meghajtó mögötti szoftverelveket írjuk le, áttekintést adva a megvalósításról és magukról a kódokról. Ez a tárgyalásmód az alfejezeteket hasznos olvasmánnyá teszi még azon olvasók számára is, akiket a kódok részletei nem érdekelnek.

3.5.1. Blokkos eszközmeghajtók áttekintése a MINIX 3-ban

Már korábban említettük, hogy az összes I/O-meghajtó főeljárásai hasonló szerkezetűek. A MINIX 3-ban mindig legalább két darab blokkos eszközmeghajtó a rendszerbe van fordítva: a RAM-lemezmeghajtó és/vagy a különféle merevlemez-meghajtók közül egy, vagy pedig egy hajlékonylemez-meghajtó. Rendszerint három darab blokkos eszköz van – mind a hajlékonylemez-meghajtó, mind pedig egy **IDE (Integrated Drive Electronics – integrált meghajtóelektronika)** merevlemez-meghajtó is jelen van. Mindegyik blokkos eszköz meghajtója függetlenül van fordítva, de a forráskód egy közös könyvtára megosztott számukra.

A régebbi MINIX-verzióknál a CD-ROM-meghajtó még elkülönülve volt jelen, amivel szükség esetén bővíteni lehetett. Ma már elképzelhetetlen különálló CD-ROM-meghajtó. Különböző meghajtócégek szabadalmazott interfészeinek támogatása szükséges, de a modern CD-ROM-meghajtók rendszerint felcsatolhatók az IDE-vezérlőhöz, bár a notebook számítógépeken néhány CD-ROM USB csatlakozású. A MINIX 3 teljes verziója támogatja a merevlemez-eszközmeghajtót, beleértve a CD-ROM-ot, viszont ebben a könyvben mi kivettük a CD-ROM-támogatást a meghajtóból.

Természetesen minden blokkos eszközmeghajtónak végre kell hajtania néhány beállítást. A RAM-lemezmeghajtónak bizonyos mennyiségű memóriát le kell foglalnia, a merevlemez-meghajtónak meg kell határoznia a merevlemezhardver paramétereit, és így tovább. A hardverspecifikus beállítások miatt valamennyi lemezmeghajtó egyedenként van meghíva. Miután végrehajtották a szükséges beállításokat, mindegyik meghajtó meghívja a főciklusát tartalmazó függvényt. Ez a ciklus vég nélkül fut; a hívóhoz nincs visszatérés. Amikor a főcikluson belül egy üzenet érkezik, akkor egy olyan függvényhívás történik, amely függvény elvégzi az üzenetben kívánt műveletet, és egy válaszüzenetet generál.

A mindegyik lemezmeghajtó processzus által hívott közös főciklus akkor kerül lefordításra, amikor a *drivers/libdriver/driver.c* és a többi állomány a könyvtárában le van fordítva, és a *driver.o* tárgykód egy másolata utána van kapcsolva mindegyik lemezmeghajtó végrehajtható állományában. Valamennyi meghajtó azt a technikát követi, hogy átad a főciklusnak egy paramétert, amely egy mutatót tartalmaz azon függvények címeinek egy táblázatára, amelyeket a meghajtó az egyes műveletekhez használ, és ezután közvetve hívja ezeket a függvényeket.

Ha a meghajtók egyetlen futtatható állományba fordítódnának le, akkor a főciklusnak csak egyetlen másolatára lenne szükség. A MINIX korai változatában olyan kód volt írva, amelyben az összes meghajtó együtt volt fordítva. A MINIX 3-ban hangsúlyt helyeznek az egyedi, amennyire lehetséges, független operációsrendszer-komponensekre, bár az elkülönülő programok számára a közös forráskód használata a megbízhatóság növelésének jó módja. Feltéve, hogy ha egyszer valamit rendbe hozunk, az jó lesz minden meghajtó számára. Vagy ha az egyik alkalmazásban találnak egy hibát, akkor más alkalmazásokban is létezhet észrevétlenül. Épp ezért az osztott forráskódot nagyon alaposan tesztelni kell.

A többszörös eszközmeghajtóknál ténylegesen hasznos egyéb függvényeknek egy halmaza a *drivers/libdriver/drvlib.c*-ben van definiálva, és a *drvlib.o* teszi ezeket elérhetővé. Az összes szolgáltatott tevékenységről feltehető, hogy egyetlen állományban van, de ezek nem mindegyike szükséges minden eszközmeghajtónak. Például a *memory* meghajtó, amely egyszerűbb, mint a többi meghajtó, csak a *drive.o*-hoz kapcsolódik. Az *at_wini* meghajtó mind a *driver.o*-, mind a *drvlib.o*-hoz kapcsolódik.

A 3.19. ábrán a főciklusnak egy vázlata látható; formailag a 3.18. ábrán lévőhöz hasonló. Az olyan utasítások, mint a

```
code = (*entry_points->dev_read)(&mess);
```

közvetett függvényhívások. Mindegyik meghajtó esetén más-más *dev_read* függvényhívás történik, jóllehet mindegyik meghajtó egy főciklust hajt végre, amely ugyanabból a forrásfájlból van fordítva. Azonban néhány művelet, mint például a close annyira egyszerű, hogy egynél több eszköz is ugyanazt a függvényt hívja.

Van hat olyan lehetséges művelet, amelyet mindegyik eszközmeghajtótól megkövetelhetünk. Ezek megfelelnek azoknak a lehetséges értékeknek, amelyek a 3.17. ábrán látható üzenet *m.m_type* mezőjében előfordulhatnak. A műveletek a következők:

1. OPEN
2. CLOSE
3. READ
4. WRITE
5. IOCTL
6. SCATTERED_IO

Ezek legtöbbjét valószínűleg az olvasók programozási tapasztalatukból már jól ismerik. Az eszközmeghajtók szintjén a legtöbb művelet az azonos nevű rendszerhívásokhoz kapcsolódik. Például vegyük a *READ* és a *WRITE* műveleteket, amelyek jelentése: olvasás és írás. Ezeknél a műveleteknél egy blokknyi adat átmozgatása történik az eszközről a hívást beindító processzushoz tartozó memóriahelyre, vagy fordítva. Egy *READ* művelet normálisan addig nem ad vissza semmi eredményt a hívónak, amíg az adatok átmozgatása tart, azonban az operációs rendszer mozgathatja az adatokat pufferelve egy *WRITE* esetében, azért, hogy későbbre halassza a tényleges átmozgatást, és azonnal visszatér a hívóhoz. Ez nagyszerű, amíg a hívó is be van ebbe avatva; később újra használható az a puffer, amelyből az operációs

```
message mess;                                   /* üzenetpuffer */

void shared_io_driver(struct driver_table *entry_points) {
/* a hívás előtt mindegyik meghajtó kezdeti beállítása megtörténik */
   while (TRUE) {
      receive(ANY, &mess);
      caller = mess.source;
      switch(mess.type) {
         case READ:      rcode = (*entry_points->dev_read)(&mess); break;
         case WRITE:     rcode = (*entry_points->dev_write)(&mess); break;
         /* egyébként, beleértve az OPEN, a CLOSE és IOCTL esetét is */
         default:        rcode = ERROR;
      }
      mess.type = DRIVER_REPLY;
      mess.status = rcode;                      /* eredmény kódja */
      send(caller, &mess);
   }
}
```

3.19. ábra. *Egy I/O-meghajtó főeljárása közvetett hívásokkal*

rendszer az adatokat kimásolta a kiíráshoz. Az *OPEN* és a *CLOSE* jelentése az eszközöknél hasonló az állományműveletként alkalmazott open és close rendszerhívásokéhoz: az OPEN műveletnek ellenőriznie kell az eszköz hozzáférhetőségét, vagy ellenkező esetben egy hibaüzenetet kell visszaadnia; egy CLOSE-nak garantálnia kell, hogy bármely olyan pufferezett adat, amelyet a hívó ki akart írni, teljesen átmozgatásra kerüljön az eszközön lévő végső helyre.

Lehet, hogy az *IOCTL* művelet nem olyan ismerős. Sok I/O-eszköz rendelkezik műveletparaméterekkel, amelyeket alkalmanként vizsgálni és természetesen változtatni kell. Az *IOCTL* ilyen. Egy szokványos feladat az átviteli sebesség vagy egy kommunikációs vonal paritásának megváltoztatása. Blokkos eszközöknél az *IOCTL* műveletek kevésbé megszokottak. A MINIX 3-ban az *IOCTL* művelettel történik a particionált lemezes eszközök vizsgálata és megváltoztatása (bár ez csupán egy blokknyi adat írását és olvasását jelenti).

Nem kétséges, hogy a *SCATTERED_IO* művelet a legkevésbé ismert ezek között. A kimondottan gyors lemezes eszközökkel (például RAM-lemez) nehéz elérni megfelelő lemez I/O-teljesítményt, ha minden lemezes kérés egyszerre csak egy blokkra vonatkozik. Egy *SCATTERED_IO* megengedi a fájlrendszernek, hogy egy kérésre több blokk írását vagy olvasását elvégezze. Egy *READ* művelet esetében lehet, hogy a járulékos blokkokat nem kéri az a processzus, amelynek a nevében a hívás történt; az operációs rendszer megkísérel elébe menni a jövőbeni adatkéréseknek. Ilyen kéréseknél az eszközmeghajtó nem fogadja el szükségszerűen az összes átviteli kérést. Az egyes blokkokra vonatkozó kérést egy jelzőbittel lehet módosítani, ami jelzi az eszközmeghajtónak, hogy a kérés elhagyható. Valójában a fájlrendszer mondhatja: „Szép dolog lenne mindezeket az adatokat birtokolni, de ebben a pillanatban nincs szükségem rájuk." Az eszköz azt tehet, ami a legjobb neki. A hajlékonylemez-meghajtó például visszaadja az egyetlen pályáról kiolvasható összes blokkot, és hatásosan azt mondja: „Ezeket neked adom, de túl sokáig tart egy másik pályára való átmozdulás; később ismét kérj meg a fennmaradottak átadására."

Amikor adatot kell írni, akkor nem kérdéses, hogy minden körülmények között írni kell. Azonban az operációs rendszer több írási kérést pufferelhet abban a reményben, hogy több blokk írása hatékonyabban végezhető el, mintha az egyes kéréseket a beérkezésükkor kezelné. Egy *SCATTERED_IO* kérésben, akár írás vagy olvasás esetében, a kért blokkok listájának rendezésével a művelet hatékonyabban elvégezhető, mintha a kérésekkel azonnal foglalkozna. Továbbá azáltal, hogy a meghajtó egyetlen hívásra több blokkot mozgat át, a MINIX 3-ban csökken az elküldött üzenetek száma.

3.5.2. Közös blokkos eszközmeghajtó szoftver

Az összes blokkos eszközmeghajtó számára szükséges definíciókat a *drivers/libdriver/driver.h*-ban helyezték el. Az állomány legfontosabb része a 10829–10845. sorokban elhelyezkedő *driver* adatszerkezet, amelyet bizonyos meghajtók arra használnak, hogy a munkájukat alkotó egyes részek elvégzésére használt függ-

vények címeinek listáját átadják. Szintén itt van definiálva a *device* adatszerkezet (10856–10859. sor), amiben a partíciókra, a kezdőcímekre és a méretre vonatkozó legfontosabb információ kerül bájtegységekben megadásra. Ezt az elhelyezési módot azért választották, hogy ne legyen szükség semmi átalakításra, amikor a munkavégzés memóriabeli eszközzel történik, maximális válaszadási sebességnél. A valóságos lemezek esetében oly sok más ok késlelteti az elérhetőséget, hogy a szektorokba való átalakítás nem jelent lényeges hátrányt.

A főciklus és az összes blokkos eszközmeghajtó közös függvényeinek forráskódja a *driver.c*-ben van. Az esetleg szükséges hardverspecifikus beállítások elvégzése után mindegyik meghajtó a *driver_task*-ot hívja, és hívásparaméterként átad egy *driver* adatszerkezetet. A DMA-műveletekhez használt puffer címének kijelölése után a főciklusba (11071–11120. sor) való belépés következik. Ez a ciklus vég nélkül fut; a hívóhoz nincs visszatérés.

A főciklus switch műveletében az első öt üzenettípus, a *DEV_OPEN, DEV_CLOSE, DEV_IOCTL, DEV_CANCEL, DEV_SELECT* esetében közvetett hívásokra kerül sor, amelyek a *driver* adatszerkezetben átadott címeket használják. A *DEV_READ* és *DEV_WRITE* üzenetek mindegyikénél a *do_rdwt* közvetlen hívása történik; *DEV_GATHER* és *DEV_SCATTER* üzenetek mindegyikénél a *do_vrdwt* közvetlen hívása történik. Azonban a switch részből való akár közvetett, akár közvetlen hívásokkal paraméterként átadódik a *driver* adatszerkezet, és így azt minden hívott rutin használni tudja, ha szüksége van rá. A *do_rdwt* és a *do_vrdwt* némi előzetes feldolgozást végez, viszont ezután ezek is indirekt hívásokat végeznek az eszközspecifikus eljárásokhoz.

A többi eset, a *HARD_INT*, *SYS_SIG* és a *SYN_ALARM* mind felelősek a nyugtázásokért. Ezek szintén közvetett hívásokban jelentkeznek, de mindegyik befejeződése után egy continue utasítás van. Ennek hatására a vezérlés visszatér a ciklus elejére, kikerülve a tisztogatást és a válaszüzenet-lépéseket.

Bármi is volt az üzenetben levő kérés, annak végrehajtása után az eszköz természetétől függően szükségessé válhat bizonyos tisztogatás. Egy hajlékonylemez esetében például valószínű, hogy egy időmérő elindítására kerül sor, és ha egy adott rövid időn belül további kérés nem érkezik, akkor a lemezmeghajtó motorja kikapcsolódik. Egy közvetett hívással történik ez is. A tisztogatást követően egy válaszüzenet kialakítására és a hívóhoz való küldésére kerül sor (11113–11119. sor). Lehetséges, hogy egy eljárás az üzenettípusok egyikét kiszolgálja, és értékül egy *EDONTREPLY*-t eredményez, hogy elfojtsa a válaszüzenetet, de az aktuális meghajtók egyike sem használja ezt az opciót.

Mindegyik taszk első dolga a főciklusba lépés után, hogy hívja az *init_buffer*-t (11126. sor), ami kijelöl egy puffert a DMA-műveletekhez. Az, hogy ez a kezdeti beállítás mégis szükséges, az eredeti IBM PC-hardverének azon tulajdonságából származik, amely megköveteli, hogy a DMA puffer ne lépje át a 64 KB-os határt. Azaz egy 1 KB-os DMA-puffer kezdődhet 64510-nél, de nem 64514-nél, mivel az a puffer, amely az utóbbi címnél kezdődne, a 65536-nál meghaladná a 64 KB határt.

Ez a zavaró szabály azért kell, mert az IBM PC egy régi DMA lapkát, az Intel 8237A-t használja, amelynek 16 bites a számlálója. Egy nagyobb számláló kellene, mert a DMA abszolút címeket használ, nem szegmensregiszterhez relatív címeket. Régebbi gépeken, amelyek a memóriát csak 1 MB-ig tudják címezni, a DMA-cím alacsonyabb helyi értékű 16 bitje a 8237A-ba kerül, míg a magasabb helyi értékű 4 bit egy speciális 4 bites tárolóelembe (latch) kerül. Az újabb gépek 8 bites speciális tárolóelemet használnak, és 16 MB-ot tudnak címezni. Amikor a 8237A a 0xFFFF-ről a 0x0000-re vált, akkor nem jön létre semmi átvitel a speciális tárolóegységben, és így a DMA-cím hirtelen a memóriának 64 KB-tal kisebb című részére ugrik.

Egy hordozható C program nem képes egy adatstruktúrához abszolút címeket meghatározni, így nincs mód annak megelőzésére, hogy a fordító a puffert használhatatlan helyre tegye. Ennek kiküszöböléséhez ki kell jelölni bájtoknak kétszer olyan nagy vektorát, mint amekkora a *buffer*-nél van (11044. sor), és e vektor aktuális elérésére egy *tmp_buf* (11045. sor) mutatót kell lefoglalni. Az *init_buffer* a *tmp_buf*-nak a *buffer* elejére történő próba jellegű beállítását végzi, és megnézi, hogy van-e elegendő hely a 64 KB-os határig. Ha a próbabeállítás nem biztosít elegendő helyet, akkor a *tmp_buf*-t növeli az aktuálisan kért bájtok számával. Így bizonyos mennyiségű hely mindig kárba vész a *buffer*-hez lefoglalt hely egyik vagy másik végénél, de nem következik be hiba amiatt, hogy a puffer nem veszi figyelembe a 64 KB-os határt.

Az IBM PC család újabb számítógépei már jobb DMA-vezérlőkkel rendelkeznek, így a kód egyszerűsíthető, és a memória egy kis része is visszavehető, ha valaki egészen biztos abban, hogy a gépében az iménti probléma nem állhat elő. Viszont ha mégis tévedne, akkor szembe kell néznie a jelentkező hibákkal. Ha egy 1 KB-os DMA-pufferre van igény, akkor 1 a 64-hez a valószínűsége annak, hogy egy régi DMA-lapkás gépen a probléma megjelenik. Minden alkalommal, amikor a kernelszintű forráskódjában módosítás történik úgy, hogy a lefordított kernelbeli méret változik, akkor van esély arra, hogy a probléma előforduljon. A legvalószínűbb, hogy amikor a hiba a következő hónapban vagy a következő évben jelentkezik, akkor ezt annak a kódnak tulajdonítják majd, amit utoljára módosítottak. Az ehhez hasonló, váratlan hardversajátosságok okozhatnak olyan rendkívül homályos hibákat, amelyek megtalálása heteket vehet igénybe (ráadásul a felhasználói kézikönyvek egy szóval sem tesznek említést ezekről).

A következő függvény a *driver.c*-ben a *do_rdwt* (11148. sor). Ez sorjában hívhat kettő eszközfüggetlen függvényt, amelyekre a *driver* adatszerkezet *dr_prepare*, *dr_transfer* mezői mutatnak. Itt és a következőkben a C nyelvi *(*function_pointer)* jelölést használjuk arra a függvényre, amelyre a *function_pointer* mutat.

A kérésben levő bájtmennyiség pozitív voltának vizsgálata után a *do_rdwt* meghívja a *(*dr_prepare)*-t. Ez a művelet betölti a *device* struktúrában elérhető lemezek, partíciók vagy részpartíciók báziscímét és méretét. A memóriameghajtó esetében, mivel ez nem támogatja a partíciókat, csak a mellékeszközszám érvényességét ellenőrzi. A merevlemeznél a mellékeszközszám használható a mellékeszközszámmal megjelölt partíció vagy részpartíció méretének kiszámításához. Ennek sikerülnie kell, mert a **dr_prepare* csak akkor sikertelen, ha egy open műveletben érvénytelen eszközt adtak meg. Következő egy *iovec_t* struktúra (amely a 2856–2859. sorban van definiálva az *include/minix/type.h*-ban), az *iovec1* feltöltése. Ez a struktúra megadja a lokális puffer virtuális címét és méretét, amelybe vagy

amelyből az adatot a rendszertaszk másolja. Azonos struktúra van használva, mint a kérésvektor egy eleme a több-blokkos hívásnál. Egy változó címe és az azonos típusú változó egy tömbje első elemének a címe pontosan azonos módon kezelhető. Ezután egy másik közvetett hívás, a *(*dr-transfer)* jön, amely végrehajtja az adatmásolást és a kért I/O-műveleteket. Az átvitelkezelő eljárások mindegyike elvárja, hogy megkapja a kérések egy tömbjét. A *do-rdwt*-ben az utolsó argumentum a hívásnál az 1, ami egyelemű tömböt ad meg.

A következő részben a lemezhardver tárgyalásánál látni fogjuk, hogy a kérések beérkezési sorrendben való megválaszolása nem hatékony módszer, így ez a rutin megengedi az egyes eszközöknek, hogy az eszköz tulajdonságából adódó legkedvezőbb módon kezeljék a kéréseket. Az egyes eszközökre jellemző módok a közvetett hívásokon keresztül maszkolódnak. A RAM-lemeznél a *dr_transfer* egy olyan eljárásra mutat, amely egy kernelhívással a rendszertaszkot arra kéri, hogy másolja az adatot a fizikai memória egyik részéből a másikra, ha a mellékeszközszám elérhető a */dev/ram,* a */dev/mem,* a */dev/kmem,* a */dev/boot* vagy a */dev/zero* valamelyikében. (Természetesen semmi másolás nincs a */dev/null* elérésénél.) Egy valódi lemeznél a *dr_transfer*-re mutató kódnak szintén a rendszertaszkot kell kérnie egy adatátvitelre. De a másolási művelet előtt (egy olvasásnál) vagy utána (egy írásnál) egy kernelhívással kell kérni a rendszertaszkot, hogy az aktuális I/O-t végezze el; a lemezvezérlő részben levő regiszterekbe megfelelő bájtok írásával válassza ki a lemezen a helyet; és határozza meg az átvitel méretét és irányát.

Az átviteli eljárás az *iovec1* struktúrában levő *iov_size* számlálót módosítja; hibakódot (egy negatív számot) ad vissza, ha volt hiba, vagy egy pozitív számot, amely az átmozgatott bájtok számát jelzi. Nem feltétlenül hiba, ha egyetlen bájt átmozgatása sem történik meg; ez azt jelzi, hogy elérte az eszköz végét. A főciklushoz visszatérve a válaszüzenet *REP_STATUS* mezejében hibakód vagy bájtszám van visszaadva a *driver_task*-ból.

A következő függvény a *do_vrdwt* (11182. sor) kezeli az összes széttagolt I/O-kérést. Egy üzenet széttagolt I/O-kéréssel az *ADDRESS* mezőt használja az *iovec_t* adatszerkezetek egyik tömbjének az elérésére, amelyek mindegyike egy puffercímet és az átmozgatandó bájtok számát tartalmazza. A MINIX 3-ban ilyen kérés csak a lemezen levő összefüggő blokkokra vonatkozhat; az üzenctben benne van a kiindulási pozíció az eszközön, és hogy írás vagy olvasás a művelet. Így egy kérésben levő összes művelet vagy olvasás, vagy írás lesz, az eszközön levő blokkok sorrendjébe rendezve. A 11198. sorban található ellenőrzés megnézi, hogy ez a hívás egy kernelszintű I/O-taszk érdekében van-e; ez egy maradvány a MINIX 3 egy korai fejlesztéséből, mielőtt az összes lemezmeghajtó felhasználói szintűre lett átírva.

Alapvetően ennek a műveletnek a kódja nagyon hasonlít ahhoz, amit egyszerű olvasáskor vagy íráskor végrehajt. Ugyanazok a közvetett hívások vannak az eszközfüggetlen *(*dr_prepare)*-re és a *(*dr_transfer)*-re. A többszörös kéréseket a ciklus a *(*dr_transfer)* által mutatott függvénnyel belsőleg kezeli. Az utolsó argumentum ez esetben az 1; ez az *iovec_t* elemek tömbmérete. A ciklus befejeződése után visszamásolódik a kérések vektora oda, ahonnan jött. A vektor mindegyik komponensének *io_size* mezője mutatja a kérésnek megfelelően átmozgatott bájtok számát, és bár a *driver_task* által kialakított válaszüzenetben közvetlenül nincs visszaadva az egyes kérések teljes bájtigénye, ehhez a hívó hozzáfér a vektorból.

A *driver.c* következő néhány rutinja segíti a fenti műveleteket. A *(*dr_name)* hívás visszaadja az eszköz nevét. Ha az eszköz neve nincs meghatározva, a *no_name* függvény visszaadja a „noname" sztringet. Vannak olyan eszközök, amelyek egy adott szolgáltatást nem kérnek, például a RAM-lemez nem igényel semmiféle speciális szolgáltatást a *DEV_CLOSE* kérésnél. A *do_nop* a kérés típusától függően különféle kódokat ad vissza. További függvények a *nop_signal,* a *nop_alarm,* a *nop_prepare,* a *nop_cleanup* és a *nop_cancel* függvények látszólagos rutinok olyan eszközök számára, amelyek nem igénylik ezeket a szolgáltatásokat.

Végül a *do_diocntl* (11216. sor) elvégzi a *DEV_IOCTL* kéréseket egy blokkos eszköznél. Egy *DEV_IOCTL* művelet partíciós információra vonatkozó olvasási *(DIOCGETP)* vagy írási *(DIOCSETP)* kérés lehet, egyébként hiba keletkezik. A *do_diocntl* hívja az eszköz *(*dr_prepare)* függvényét, amely ellenőrzi az eszköz érvényességét, és egy mutatót ráállít a *device* adatszerkezetre, amely tartalmazza a partíció kezdetét és bájtokban a méretet. Olvasási kéréskor meghívja az eszköz *(*dr_geometry)* függvényét, hogy megkapja a partíció cilinder-, fej- és szektorinformációját. Minden esetben egy sys_datacopy kernelhívással kérhető a rendszertaszktól, hogy az adatot másolja a meghajtó és a kérő processzus memóriahelyei között.

3.5.3. A meghajtó könyvtára

A *drvlib.h* és a *drvlib.c* állományokban az IBM PC-kompatibilis számítógépek lemezeinek particionálását támogató, rendszerfüggő kódok vannak.

A particionálással egyctlen tárolóeszköz szétbontható részeszközökké. A legismertebb a merevlemezek esete, de a MINIX 3 támogatja a hajlékonylemezeknél is a particionálást. Néhány szempont, amiért a lemezes eszközöket particionálják:

1. Nagy lemezekben olcsóbb az egységnyi lemezkapacitás. Két vagy több operációs rendszernek különböző fájlrendszerekkel való használata során sokkal gazdaságosabb egyetlen nagy lemezt particionálni, mint az egyes operációs rendszerek számára több kisebb lemezt üzembe helyezni.
2. Az operációs rendszereknek az általuk kezelhető eszközök méretében korlátozottak a lehetőségeik. A jelenlegi vizsgálatainkban szereplő MINIX 3-változat például 4 GB-os fájlrendszert kezelhet, de a régebbi változatoknál ez a határ 256 MB. Így az e feletti lemezterület kárba vész.
3. Egy operációs rendszer használhat két vagy több különböző fájlrendszert. Például egy szabványos fájlrendszer használható a szokásos állományokhoz és egy másként felépített fájlrendszer a memóriacsere (swap) helyeként.
4. Kényelmes megoldás lehet, ha a rendszer állományainak egy része külön logikai eszközre kerül. A MINIX 3 gyökérfájlrendszerének egy kis eszközre való helyezésével könnyebbé válik a mentés, és elősegíti indításkor a RAM-lemezre való másolást.

A lemezparticionálás támogatása megegyezés kérdése. Ez a tulajdonság nem kötődik a hardverhez. A particionálás támogatása eszközfüggetlen. Azonban, ha egynél több operációs rendszer fut egy adott hardverkiépítettség mellett, akkor meg kell egyezni a partíciós táblázat formátumában. IBM PC-ken a szabványt az MS-DOS *fdisk* parancsa állítja be, és a többi OS-rendszer, mint a MINIX 3, a Windows és a Linux, ezt a formátumot használva együtt tudnak létezni az MS-DOS-szal. Amikor egy másik géptípushoz kapcsolódik a MINIX 3, akkor értelemszerűen egy olyan partíciós táblázatformátumot használ, amely kompatibilis az új hardveren futó másik operációs rendszer által használttal. Így az IBM számítógépeken a particionálást támogató MINIX 3-forráskód *driver_c* helyett a *drvlib.c*-be került, két okból. Először is, nem az összes lemeztípus támogatja a particionálást. Mint korábban már említettük, a memóriameghajtó a *driver.o*-hoz kapcsolódik, nem használja a *drvlib.o*-ba fordított függvényeket. Másrészt ez könnyebbé teszi a MINIX 3 kapcsolását a különböző hardverekhez. Könnyebb egy kis állományt elhelyezni, mint sok szekcióval egy nagyot szerkeszteni és a különböző környezetekben fordítani.

Az *include/ibm/partition.h*-ban definiált alap-adatstruktúra a ROM-beli szoftverek (firmware) tervezőitől örökölt, és az `#include` utasítással ágyazható be a *drvlib.h*-ba (10900. sor). Ez információt tárol a partíciók cilinder-fej-szektor geometriájáról, valamint kódokat a partíciók fájlrendszertípusainak azonosítására és egy jelzőt a betölthetőség jelzésére. Ezen információk legtöbbjére nincs szüksége a MINIX 3-nak, miután egyszer a fájlrendszer már ellenőrzött.

A *partition* függvény (*drvlib.c*-ben a 11426. sor) meghívására akkor kerül sor első alkalommal, mikor egy blokkos eszközt először nyitnak meg. Az argumentumok között ott van a *driver* adatszerkezet, így eszközfüggő függvényeket is hívhat, továbbá egy kezdeti mellékeszközszám és egy paraméter, jelezve, hogy a particionálás fajtája hajlékonylemez, elsődleges partíció vagy részpartíció. Ez hívja a *(*dr_prepare)* eszközfüggő függvényt, hogy ellenőrizze az eszköz érvényességét, és beletegye a kezdőcímet és a méretet a *device* adatszerkezetbe, amelyről az előző szakaszban már szóltunk. Ezután hívja a *get_part_table* függvényt, hogy megállapítsa, vajon egy partíciós tábla jelen van-e, és ha igen, akkor beolvassa. Ha nincs semmi partíciós tábla, akkor a munkát elvégezte. Egyébként kiszámítja az első partíció mellékeszközszámát, felhasználva az eredeti hívásban specifikált particionálási fajtára alkalmazott alárendelt eszközök számozási szabályait. Az elsődleges partíciók esetében a partíciós tábla úgy van rendezve, hogy a partíciók rendezése megfelel a más operációs rendszereknél használtakkal.

Ennél a pontnál a *(*dr_prepare)* függvény egy másik hívása is megtörténik; ez alkalommal az első partíció újra kiszámított eszközszámának felhasználásával. Ha a részeszköz érvényes, akkor egy ciklus hajtódik végre a tábla minden belépési pontjára, ebben megvizsgálja az eszközön levő táblából beolvasott értékeket, hogy kívül esnek-e azon a tartományon, ami korábbról fennáll a kezdőcímre és a teljes eszköz méretére. Ha eltérés mutatkozik, akkor a tárban a táblázat kiigazítására kerül sor úgy, hogy illeszkedjen. Ez meglepőnek tűnhet, de mivel a táblázatokat különböző operációs rendszerek írhatják, és egy másik operációs rendszert használó programozó esetleg megpróbálja valami szokatlanra használni a partíciós

táblázatot, vagy valami más okból adódóan a lemezen a táblázatban szemét is lehet. Mi a bizalmunkat a MINIX 3 szerint kiszámított számokba helyezzük. Jobb a biztonság, mint az utólagos sajnálkozás.

Még a cikluson belül, az eszközön az összes partíció esetében, ha a partíció MINIX 3-partícióként azonosított, akkor a *partition* függvény hívódik rekurzívan, hogy a részpartíciók információit összegyűjtse. Ha a partíció egy kiterjesztett partíció, akkor a következő függvény, az *extpartition* hívása történik.

Az *extpartition* (11501. sor) függvény valójában a MINIX 3 operációs rendszer esetében nem tesz semmit, így ennek részleteivel itt nem foglalkozunk. Vannak operációs rendszerek (például Windows), amelyek **kiterjesztett partíciók**at használnak. Kapcsolt listákat alkalmaznak a rögzített méretű tömbök helyett a részpartíciók támogatásához. A MINIX 3 az egyszerűség miatt a részpartícióknál ugyanazt a módszert használja, mint az elsődleges partícióknál. Azonban feltehető olyan, a kiterjesztett partíciókra vonatkozó minimális támogatás, amely lehetővé teszi, hogy a MINIX 3 parancsai más operációs rendszerek állományait és könyvtárait olvashassák és írhassák. A műveletek könnyűek; sokkal bonyolultabbá válna, ha feltételeznénk a teljes támogatását annak, hogy a kiterjesztett partíciók felcsatolása és egyéb használata ugyanolyan módon történne, mint az elsődleges partícióknál.

A *get_part_table* (11549. sor) meghívja a *do_rdwt* függvényt, hogy megkapja az eszközön (vagy részeszközön) azt a szektort, ahol egy partíciós tábla van. A relatív elhelyezkedés argumentuma nulla, ha a hívás egy elsődleges partíció megszerzésére van kiadva, és nem nulla egy részpartíció esetében. Ez ellenőrzi a bűvös számot (0xaa55), és igaz vagy hamis állapotot ad vissza, jelezve az érvényes partíciós tábla megtalálását. Ha a tábla létezik, akkor az argumentumként átadott táblacímre bemásolja azt.

Végül a *sort* (11582. sor) rendezi a belépési pontokat egy partíciós táblában a legalacsonyabb szektor szerint. Azok a belépési pontok, amelyekhez nem tartozik partíció, a rendezésben nem vesznek részt, és a rendezettek végére kerülnek, még ha nulla érték van is az alacsony szektorú mezőjükben. A rendezés egyszerű buborékrendezéssel történik; semmi szükség hatékonyabb algoritmus alkalmazására egy csupán négy tételből álló rendezésnél.

3.6. RAM-lemezek

Most visszatérünk az egyes blokkos eszközmeghajtókhoz, és közülük többet részletesen tanulmányozunk. Az első, amit megnézünk, a memóriameghajtó. Ezzel a memória bármely része elérhető. Fő haszna, hogy megengedi a memória egy részének lefoglalását és annak egy hagyományos lemezhez hasonló használatát; erre itt mint RAM-lemezmeghajtóra fogunk hivatkozni. Nem nyújt állandó jellegű tárolást, de az erre a területre bemásolt állományok rendkívül gyorsan elérhetők.

Egy RAM-lemez szintén hasznos az olyan számítógép operációs rendszerének első telepítésekor, amelynek csak egy cserélhető tárolóeszköze van, például egy

hajlékonylemez, CD-ROM vagy valamilyen hasonló eszköz. A gyökéreszköznek a RAM-lemezen való elhelyezésével a cserélhető tárolóeszközök fel- és lecsatolhatók aszerint, hogy szükségesek-e a merevlemezre való adatátvitelhez. A gyökéreszközt a hajlékonylemezre helyezve lehetetlenné válik állományok hajlékonylemezekre való mentése, mivel a gyökéreszköz (az egyetlen hajlékonylemez) nem lehet lecsatolva. A RAM-lemezeket „élő" CD-ROM-ként is szokták használni, ami lehetővé teszi tesztelési vagy demonstrációs céllal az operációs rendszer futtatását bármely állománynak a merevlemezre másolása nélkül. Ezen túlmenően a RAM-ban elhelyezkedő gyökéreszköz a rendszert nagyon rugalmassá teszi: hajlékonylemezek és merevlemezek tetszőleges kombinációja felcsatolhatók. Ilyen élő CD-ROM-on forgalmazzák a MINIX 3-at és sok más operációs rendszert.

Majd látni fogjuk, hogy a memóriameghajtó számos egyéb tevékenységet is támogat a RAM-lemezzel kapcsolatban. A közvetlen memória elérése támogatott, akár bájtonként, akár más egységenként. Ez a módszer inkább egy karakteres eszközé, mint egy blokkos eszközé. Egyéb memóriameghajtó által támogatott karakteres eszköz a */dev/zero* és a */dev/null,* amelyek széles körben ismertek.

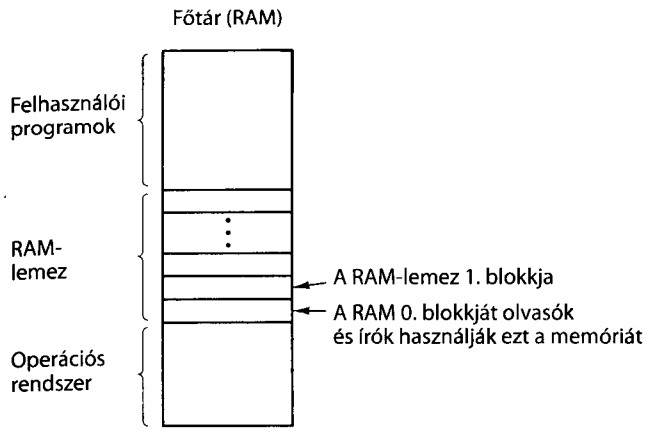
3.6.1. Hardver és szoftver a RAM-lemeznél

A RAM-lemez hátterében megbúvó ötlet nagyon egyszerű. A blokkos eszköz egy tároló két paranccsal: egy blokk olvasása és egy blokk írása. Szokásosan ezek a blokkok ciklikus elérésű tárolókon jelennek meg, például hajlékonylemezen vagy merevlemezen. A RAM-lemez ezeknél egyszerűbb: a főtár előre lefoglalt részét használja a blokkok elhelyezésére. Egy RAM-lemez az azonnali elérés (semmi keresés vagy forgási késleltetés) előnyével rendelkezik, így alkalmas olyan programok és adatok tárolására, amelyek elérésére gyakran van szükség.

Itt érdemes röviden rámutatni azon rendszerek közötti különbségre, amelyek támogatják a fájlrendszerek csatolását, és amelyek nem (például az MS-DOS és a Windows). A felcsatolt fájlrendszereknél a gyökéreszköz mindig jelen van egy állandó helyen, és a cserélhető fájlrendszerek (például lemezek) felcsatolhatóak az állományfájlba, és így kialakul egy egyesített fájlrendszer. Ha egyszer már mindennek megtörtént a felcsatolása, akkor a felhasználó nem lesz gondban amiatt, hogy melyik eszközön van egy állomány.

Ezzel ellentétben, az MS-DOS-os rendszereknél a felhasználónak valamennyi állomány helyét meg kell határoznia, vagy expliciten – például *B:\DIR\FILE* alakban –, vagy bizonyos alapbeállítások felhasználásával (aktuális eszköz, aktuális könyvtár és így tovább). Egy vagy két hajlékonylemez esetében ez a teher még kezelhető, de lemezek tucatjaival rendelkező nagy számítógéprendszer esetén az eszközök nyomon követése egész idő alatt elviselhetetlen lenne. Jusson eszünkbe, hogy Unix operációs rendszerek futnak a kis otthoni és hivatali számítógépektől a szuperszámítógépekig, köztük az IBM Blue Gene/L szuperszámítógépen is, amely a könyv írásakor a leggyorsabb gép a világon; az MS-DOS pedig csak kis rendszereken fut.

A 3.20. ábra felvázolja a RAM-lemez alapgondolatát. A RAM-lemez *n* darab blokkra van osztva, és *n* attól függ, hogy mennyi memória van a lemez számára le-



3.20. ábra. *Egy RAM-lemez*

foglalva. Minden egyes blokknak ugyanakkora a mérete, mint a valódi lemezeken levő blokkméret. Amikor a meghajtó kap egy blokk írásával vagy olvasásával kapcsolatos üzenetet, akkor éppen csak ki kell számítani a kért blokk helyét a RAM-lemez memóriájában, és onnan máris lehet olvasni vagy írni a blokkot, hajlékonylemez vagy merevlemez alkalmazása helyett. Végül is a hívott rendszertaszk végzi az átvitelt. A kernelben levő *phys_copy* assembly nyelvű eljárás dolgozik, másol a felhasználói programba vagy onnan, a hardver képessége szerinti maximális sebességgel.

Egy RAM-lemezmeghajtó támogathatja a memória több területének RAM-lemezként való használatát, közöttük a hozzájuk rendelt különböző mellékeszközszámmal téve különbséget. Általában ezek a területek eltérők, de néhány esetben kényelmi okok miatt átlapoltak is lehetnek, mint ahogy a következő részben látható lesz.

3.6.2. A RAM-lemezmeghajtó áttekintése a MINIX 3-ban

A MINIX 3 RAM-lemez valójában hat, egymáshoz szorosan kapcsolódó meghajtó. Mindegyik beérkező üzenet meghatároz egy alárendelt eszközt; ezek a következők:

0:/dev/ram	2:/dev/kmem	4:/dev/boot
1:/dev/mem	3:/dev/null	5:/dev/zero

A fenti felsorolásban az első speciális fájl, a */dev/ram* egy igazi RAM-lemez. Sem a mérete, sem a kezdőpontja nincs a meghajtóba beépítve. Ezeket a MINIX 3 indításakor a fájlrendszer határozza meg. Ha az indítóparaméter specifikálja, hogy a gyökérfájlrendszer a RAM-lemezen van, de nem definiálják a RAM-lemez méretét, akkor a gyökérfájlrendszerkép eszközének méretével azonos méretű RAM-

lemez van létrehozva. A gyökérfájlrendszer méretétől eltérő RAM-lemezméret egy indítóparaméterrel adható meg; ez lehet nagyobb, de lehet tetszőleges olyan érték, amely elfér a memóriában, és elegendő helyet hagy a rendszer tevékenységéhez, ha a RAM-ba a gyökérfájlrendszert nem kell bemásolni. A méret ismeretében egy megfelelő nagyságú memóriablokk-kérés történik, amelyet a processzszusvezérlő kivesz az inicializálás alatt a memóriából. Ezzel a módszerrel lehetővé válik a RAM-lemezként jelentkező mennyiség növelése vagy csökkentése anélkül, hogy az operációs rendszert újra kellene fordítani.

A következő két alárendelt eszköz olvassa és írja a fizikai memóriát, illetve a kernel memóriaterületét. A megnyitott és olvasó */dev/mem* kiveszi a fizikai memóriahelyek tartalmát, kezdve a nulla abszolút címtől (a valós módú megszakításvektorok). A szokásos felhasználói programok soha nem teszik ezt, de egy hibakereső rendszerrel rendelkező rendszerprogramnak szüksége lehet erre a lehetőségre. Megnyitva a */dev/mem*-et és írva arra, a megszakításvektorok fognak megváltozni. Szükségtelen is mondanunk, hogy mindezt csak azoknak a gyakorlattal rendelkező felhasználóknak szabad tenniük, akik tudják, hogy pontosan mit tesznek, és nekik is a legnagyobb körültekintéssel kell dolgozniuk.

A */dev/kmem* speciális fájl, hasonló a */dev/mem* állományhoz, kivéve, hogy ennek az állománynak a nulla sorszámú bájtja most a kernel-adatmemória nulla sorszámú bájtja, amely helynek az abszolút címe a MINIX 3-kernelbeli kódszegmens méretétől függően változik. Ezt is elsősorban a hibakereső és más, nagyon speciális programok használják. Vegyük észre, hogy ez a két alárendelt eszköz átlapoltan fedi le a RAM-lemez területét. Ha például tudjuk pontosan a memóriában a kernel elhelyezkedését, akkor a */dev/mem* állományt megnyitva megkereshetjük a kernel-adatterület kezdetét, és ugyanazokat a dolgokat láthatjuk, mint a */dev/kmem* kezdetének olvasásakor. Azonban ha a kernelt újrafordítjuk, megváltoztatva a méretét, vagy ha a MINIX 3 egy leszűkített változatában a kernel a memória valamely más helyére került, akkor a */dev/mem*-ben különböző mennyiségű helyet kell átkutatni, hogy láthassuk ugyanazt a dolgot, mint amit a */dev/kmem* elején láthatunk. Mindkét speciális fájlt védeni kell: meg kell előzni, hogy a rendszergazdán kívül más is használja.

A csoport utolsó állománya, a */dev/null,* egy olyan speciális fájl, amely átvesz adatot és eldobja azt. A parancsértelmező ezt használja, amikor a program olyan kimenetet állít elő, amelyre nincs szüksége. Például:

```
a.out>/dev/null
```

futtatja az *a.out* programot, de eldobja a kimenetét. A RAM-lemezmeghajtó hatásosan kezeli ezt az alárendelt eszközt, mivel a mérete nulla, és így semmi adat nincs oda bemásolva vagy onnan kivéve. Ilyet olvasva egy azonnali EOF-t (End of File, fájlvége) kapunk.

Ha valaki látta ezeknek az állományoknak a könyvtári bejegyzéseit a */dev/*-ben, akkor észrevehette, hogy csak a */dev/ram* a blokkos fájl. Az összes többi karakteres eszköz. A memóriameghajtó még egy blokkos eszközt támogat, ez a */dev/boot*. Az eszközmeghajtó nézőpontjából egy másik blokkos eszköz a RAM-ban van implementálva; ez a */dev/ram*. Bárhogyan is, ez azt jelenti, hogy a kezdeti beállítás a betöltési memóriaképhez csatolt fájl memóriába másolásával történik az *init* után, ahelyett hogy a memória egy üres blokkjával kezdődne, mint a */dev/ram* esetében. Feltételezhető, hogy ilyen eszköztámogatásra csak a későbbiekben lesz szükség, így a MINIX 3 nem alkalmazza.

Végül a memóriameghajtó által támogatott utolsó eszköz, egy másik karakteres fájl, a */dev/zero*. Néha kényelmes, ha egy zero forrással rendelkezünk. A */dev/zero*-ra írás olyan, mint a */dev/null*-ra írás: eldobja az adatot. De a */dev/zero* olvasása nullát ad bármilyen kívánt mennyiségben, akár egyetlen karaktert, vagy egy teljes lemezt feltöltve.

A meghajtószinten a */dev/ram, /dev/mem, /dev/kmem,* a */dev/boot* kezelők kódja azonos. Az egyedüli különbség közöttük, hogy mindegyik a memória más-más területének felel meg, ez a *ram-origin* és *ram-limit* tömbökkel van jelezve, és mindegyik a mellékeszközszámmal indexelve. A fájlrendszer az eszközöket egy magasabb szinten kezeli, és azokat vagy karakteresnek, vagy blokkosnak ismeri fel, és ennek megfelelően felcsatolja a */dev/ram*-ot és a */dev/boot*-ot, és kezeli az eszközökön levő könyvtárakat és állományokat. Karaktereskénti definiált eszköznél a fájlrendszer csak folytonos adatot tud olvasni vagy írni (bár egy folytonos olvasás a */dev/null*-ból csak EOF-ot ad).

3.6.3. A RAM-lemezmeghajtó megvalósítása a MINIX 3-ban

Más lemezmeghajtókhoz hasonlóan a RAM-lemezmeghajtó főciklusa is a *driver.c* állományban van. A *memory.c*-ben a memóriaeszközök eszközfüggő támogatása valósul meg. A memóriameghajtó fordításakor a *drivers/libdriver/driver.o* nevű tárgyfájlnak egy másolata elkészül a *drivers/libdriver/driver.c* fordítással, és ez van összeszerkesztve a *drivers/memory/memory.o* tárgyfájllal, ami a *drivers/memory/memory.c* fordítás eredménye.

Érdemes megnézni, hogyan történik a főciklus fordítása. A *driver.h*-ban (10829–10845. sor) levő *driver* struktúra deklarációja definiálja az adatstruktúrát, de nem hozza azt létre. Az *m_dtab* deklaráció (11645–11660. sor) létrehoz egy példányt, feltölti a struktúra minden egyes részét egy függvénymutatóval. Néhány függvény általános kódjának fordítása a *driver.c* fordításakor történik, ilyen például a *nop* összes függvénye. Vegyük észre, hogy a memóriameghajtó belépési pontjai közül héthez vagy keveset csináló, vagy semmit sem csináló eljárás tartozik, az utolsó kettő pedig *NULL*-ként van definiálva (ez azt jelenti, hogy ezek a függvények soha nem lesznek meghíva, még egy *do_nop* sem szükséges). Mindez azt bizonyítja, hogy a RAM-lemez műveletei nem különösebben bonyolultak.

A memóriaeszköz nem követeli meg nagyszámú adatstruktúrának vagy másnak a definícióját. Az *m_geom*[*NR_DEVS*] (11627. sor) tárolja a báziscímét és 64 bites egészekként a hat memóriaeszköz méretét, így a MINIX 3-nál nem fordul elő, hogy ne rendelkezne elég nagy RAM-lemezzel. A következő sor más meghajtókban nem látható, érdekes struktúrát definiál. Az *m_seg*[*NR_DEVS*] látszólag egy egészekből álló tömb, de ezek az egészek azt jelzik, hogy egy szegmensleírás meg-

található-e. A memória eszközmeghajtó a felhasználói szintű processzusok között szokatlan, mivel lehetővé tesz memóriaterület-elérést a szokásos kód-, adat- és veremszegmenseken kívül, amelyeket mindegyik processzus birtokol. Ez a tömb tárolja azt az információt, amely a kijelölt memóriabővítési terület elérését megengedi. Az *m_device* változó tárolja az éppen aktív alárendelt eszköz ebben a vektorban levő indexét.

A */dev/ram*-nak mint gyökéreszköznek a használatához a memóriameghajtót nagyon korán, a MINIX 3 indításakor inicializálni kell. A következőként definiált *kinfo* és *machine* struktúrák az indítás alatt a kernelből kinyert adatot tárolják, ami a memóriameghajtó inicializálásához szükséges.

Mielőtt a végrehajtható kód elkezdődik, egy másik adatstruktúra definiálása is megtörténik. Ez a *dev_zero,* egy 1024 bájtos tömb, amely a */dev/zero*-ra vonatkozó read hívásnál gondoskodik az adatról.

A *main* főeljárás (11672. sor) egy függvényt hív a lokális inicializálás elvégzésére. Ezután hívja a főciklust, amely üzeneteket kap, továbbküldi azokat a megfelelő eljárásokhoz, és visszaküldi a válaszokat. A befejezéskor a *main*-hez nem tér vissza.

A következő függvény, az *m-name* triviális. Ez a „memory" karaktersorozatot adja vissza, amikor hívják.

Olvasáskor vagy íráskor a főciklus három hívást kezdeményez: egyet, ami az eszközt előkészíti, egyet az aktuális adatátvitel elvégzésére és egyet a helyreállításra. Ezek közül egy memóriaeszköz esetében az első hívás *m_prepare*. Ez ellenőrzi, hogy a kérés érvényes mellékeszközre vonatkozik-e, és utána visszaadja a kért RAM-terület kezdőcímét és méretét tároló adatszerkezet címét. A második hívás *m_transfer* (11706. sor). Ez végzi az összes munkát. Ahogyan láttuk a *driver.c*-ben, az adatolvasásra vagy -írásra vonatkozó összes hívás átalakítódik többszörös, folytonos adatblokkokat olvasó vagy író hívásokká – ha csak egy blokkra van szükség, a kérés akkor is többszörös blokkra vonatkozik egyértékű számlálóval. Így az átvitelnek csak két fajtája jut a meghajtóhoz, a *DEV_GATHER*-hez egy vagy több blokk olvasásakor, és a *DEV_SCATTER*-hez egy vagy több blokk írásásakor. Így a mellékeszközszám ismeretében *m_transfer* egy ciklust indít el, amelyet a kért átvitelek számának megfelelően ismétel. A ciklusban van egy kapcsoló az eszköz típusára.

A */dev/null* esetében egy *DEV_GATHER* kérés vagy egy *DEV_SCATTER* kérés eredménytelen, és a tevékenység közvetlenül a kapcsoló végére kerül. Ez az átmozgatott bájtok számát (bár ez a szám a */dev/null*-nál nulla) úgy adja vissza, mintha egy write műveletet végrehajtott volna.

Az összes valós memóriahelyre hivatkozó eszköztípusnál hasonló a tevékenység. A kért hely ellenőrzése az eszköz mérete alapján történik, megvizsgálva, hogy vajon a kérés az eszközhöz lefoglalt memóriakorlátok közé esik-e. Ezután egy kernelhívás a hívó memóriájába vagy abból másolja az adatot. Két kódmaradvány is van, ami ezt eredményezi. A */dev/ram,* a */dev/kmem* és a */dev/boot* esetén virtuális címek használhatók; ezekhez szükséges a memóriaterület szegmenscíme, amely elérhető az *m_seg* tömbből egy sys_vircopy kernelhívással (11640–11652. sor). A */dev/mem* egy fizikai címet használ és a sys_physcopy hívását.

A megmaradó művelet egy olvasás vagy írás a */dev/zero*-ra. Az adat olvasása a *dev_zero* tömbből történik, mint már korábban említettük. Megkérdezheti valaki, hogy miért nem generálódnak a nulla értékek, ahelyett hogy mindegyik egy pufferből másolódik be. Mivel egy adat másolása egy adott helyre kernelhívással történik, egy ilyen módszer vagy bájtok hatástalan másolását igényelné a memóriameghajtótól a rendszertaszkokhoz, vagy a rendszertaszkban nullákat generáló kódnak kellene lennie. Az utóbbi megközelítés növelné a kernelszintű kód bonyolultságát, amit a MINIX 3 szeretne elkerülni.

A memóriacszköznél egy írás vagy olvasás művelet befejezéséhez nincs szükség a harmadik lépésre, így a megfelelő *m_dtab*-ban levő belépési pontoknál egy *nop_finish* hívás van.

Egy memóriaeszköz megnyitását az *m_do_open* (11801. sor) végzi. A munka főrésze az *m_prepare* hívásával hajtódik végre, amely megvizsgálja a hivatkozott eszköz érvényességet. Az itt levő kódnál azonban érdekesebb a MINIX régebbi verzióiban itt található kódra vonatkozó megjegyzés. Előzőleg egy sajátosság volt itt elrejtve. Egy felhasználói processzus általi hívás, amely megnyitja a */dev/mem*-et vagy */dev/kmem*-et, varázslatos képességet adna a hívónak olyan parancsok végrehajtására, amelyek elérik az I/O-kapukat. A Pentium osztályú CPU-k négy jogosultsági szintet valósítanak meg, és a felhasználói processzusok szokásosan a legalacsonyabb jogosultsági szintnél futnak. A CPU egy általános védekezési módot generál, amikor egy processzus megpróbál egy parancsot nem a neki megengedett jogosultsági szinten végrehajtani. Ezt a módszert biztonságosnak tekintették, mivel a memóriaeszközöket csak gyökér jogosultságú felhasználó érhette el. Ez a potenciálisan kockázatos megoldás a MINIX 3-ból hiányzik, mivel azok a kernelhívások, amelyek engedik az I/O-elérést a rendszertaszkon keresztül, most rendelkezésre állnak. Ha a MINIX 3 van kötve a hardverhez, amely memórialeképezésű I/O-t alkalmaz, akkor szükséges lehet egy ilyen konstrukció újrabevezetése. Az ezt végző függvény, az *enable_iop* bennmarad a kernelkódban – hogy mutassa, hogyan volt ez kezelve –, bár most árvának érzi magát.

A következő függvény az *m_init* (11817. sor) hívására csak a *mem_task* első meghívásakor kerül sor. Ez a rutin számos kernelhívást használ, és érdemes tanulmányozni, hogy lássuk, ahogyan a meghajtók egymásra hatnak a kernelszinten keresztül a MINIX 3-ban, felhasználva a rendszertaszk-szolgáltatásokat. Először egy sys_getkinfo kernelhívás történik, hogy a kernelhívás *kinfo* adatának egy másolata elérhető legyen. Ebből az adatból a */dev/kmem* báziscímét és -méretét az *m_geom* adatstruktúra megfelelő mezőibe másolja. Egy másik kernelhívás, a sys_segctl átalakítja a */dev/kmem* fizikai címét és méretét a szegmensleíró információba, ami ahhoz szükséges, hogy a kernelmemória virtuális memóriahelyként legyen kezelhető. Ha egy betöltési eszköz memóriaképének a fordítása a rendszer betöltési memóriaképében van, akkor a */dev/boot* báziscímének a mezője nem lesz nulla. Ha ez így van, akkor az eszköz számára a memóriaterület eléréséhez az információ pontosan ugyanazon a módon készül, mint ahogyan a */dev/kmem* esetében. Nullákkal töltődik fel az a tömb, amely az adatot biztosítja a */dev/zero* elérésekor. Ez valószínűleg szükségtelen; a C fordítóról feltehető, hogy az újként létrehozott statikus változókat nullákkal inicializálja.

Végül az *m_init* a sys_getmachine kernelhívással kapja meg a kernelből az adatok egy másik halmazát, a *machine* struktúrát, amely a különféle lehetséges hardveralternatívákat jelzi. Ez esetben szükséges az az információ, hogy a CPU védett módú műveletekre alkalmas, vagy nem. Ez alapján a */dev/mem* mérete vagy 1 MB-ra, vagy 4 GB – 1-re állítódik be, attól függően, hogy a MINIX 3 a 8088-as vagy a 80386-os módban fut. Ez a MINIX 3 által támogatott maximális méret, nem tehető semmi annak érdekében, hogy több RAM legyen a gépen telepítve. Csak az eszköz mérete kerül beállításra, a fordítóra van bízva a báziscím nullára állítása. Mivel a */dev/mem* mint fizikai (nem virtuális) memória érhető el, ezért nincs szükség egy sys_segctl kernelhívásra a szegmensleíró beállításához.

Mielőtt elhagynánk az *m_init*-et, meg kell említeni egy másik kernelhívás alkalmazást, bár ez a kódban nem szembetűnő. A memóriameghajtó inicializálása során alkalmazott tevékenységek közül sok a MINIX 3-ra jellemző funkciókhoz alapvető, így számos teszt is végbemegy, és egy teszt sikertelensége esetén a *panic* hívására kerül sor. A *panic* esetében egy könyvtári eljárás végül is a sys_exit kernelhívást eredményezi. A kernel, a processzuskezelő és a fájlrendszer saját *panic* eljárással rendelkeznek. Az eszközmeghajtók és más kisebb rendszerkomponensek számára könyvtári eljárások állnak rendelkezésre.

Meglepő, hogy a most vizsgált *m_init* függvény nem inicializálja a */dev/ram* fontos memóriaeszközt. Erről a következő függvény, az *m_ioctl* (11863. sor) gondoskodik. Valójában csak egy ioctl művelet van definiálva a RAM-lemez számára; ez a *MIOCRAMSIZE,* amelyet a fájlrendszer használ a RAM-lemez méretének beállításához. A legtöbb munka során nincs szükség a kernel semmiféle szolgáltatására. A 11887. sorban az allocmem hívás rendszerhívás, de nem kernelhívás. Ezt a processzuskezelő kezeli, amely karbantartja az összes olyan információt, ami a memória egy használható területének megkereséséhez szükséges. Bárhogyan is, egy kernelhívás kell a végén. A 11894. sorban a sys_segctl hívás állítja elő a fizikai címet és a méretet az allocmem által, visszaadva a szegmensinformációban azt, ami a további eléréshez szükséges.

A *memory.c*-ben definiált utolsó függvény az *m_geometry*. Ez egy utánzat. Nyilvánvalóan a cilinderek, fejek és szektorok egy félvezető memória címzésében lényegtelenek, de ha egy kérés egy memóriaeszköz ilyen információjára vonatkozik, ez a függvény szimulálja a 64 fejes és sávonként 32 szektoros helyzetet, és kiszámolja a méretből a cilinderek számát.

3.7. Lemezek

Minden moderm számítógépnek, kivéve a beágyazott gépeket, vannak lemezmeghajtó egységei. Ezért most tanulmányozni fogjuk ezeket. A hardverrel kezdjük, majd a lemezszoftvereknek néhány általános jellemzőjéről beszélünk. Ezután a MINIX 3 lemezvezérlésének fogunk a mélyére ásni.

3.7.1. Lemezhardver

Minden valódi lemez cilinderekbe szervezett, minden egyes cilinder annyi pályavonalból áll, ahány függőlegesen elhelyezkedő olvasófej tartozik a lemezhez. A pályavonalak szektorokba vannak osztva, a szektorok száma körvonalanként a hajlékonylemezeken 8 és 32 közötti érték, míg néhány merevlemezen ez több száz is lehet. A legegyszerűbb konstrukciókban mindegyik pályavonalon azonos számú szektor van. Valamennyi szektor azonos számú bájtból áll, ami egy kicsit elgondolkodtató tulajdonság, hiszen a lemez pereméhez közel levő pályavonalak fizikailag hosszabbak, mint amelyek a középponthoz vannak közel. Akárhogyan van is, a szektorok írása és olvasása azonos időt igényel. Nyilvánvalóan a bentebb levő pályavonalakon a jelek sűrűsége nagyobb, és néhány lemezkonstrukciónál a bentebbi pályavonalaknál szükséges a meghajtóban az olvasófejek korrekciója. Ezt azonban a lemezt vezérlő hardver útján kezeli, és a felhasználó nem lát belőle semmit (de még az operációs rendszer készítője sem).

A bentebb és kintebb levő pályavonalakra jellemző jelsűrűségek közötti különbözőség a kapacitásban jelent áldozatot, és kifinomultabb rendszerek meglétét kívánja. Azok a hajlékonylemez-szerkezetek, amelyek nagyobb sebességgel forognak, amikor a fejek kintebbi pályavonalak fölött vannak, már kipróbáltak. Valójában több szektor is elférne az egyes pályavonalakon, növelve ezzel a lemez kapacitását. Azonban a MINIX 3-at jelenleg használó rendszerek egyike sem támogatja az ilyen lemezeket. A modern nagy merevlemezmeghajtó egységeknél a kintebbi pályákon a szektorok száma nagyobb, mint a bentebbi pályavonalakon. Ezek az **IDE- (Integrated Drive Electronics – integrált meghajtóelektronika)** meghajtószerkezetek a bonyolultabb feldolgozást a meghajtószerkezetbe beépített elektronikával hajtják végre. Az operációs rendszer számára azonban mindez úgy látszik, mintha minden pályavonalon azonos számú szektor lenne.

A meghajtószerkezet és a vezérlőelektronika ugyanolyan fontosak, mint a mechanikai hardver. A lemezvezérlő fő eleme egy speciális integrált áramkör, valójában egy kis mikroszámítógép. Valamikor ez a számítógép hátlapjába bedugott kártyán volt, de a modern rendszereknél a lemezvezérlő az alaplapra van integrálva.

A lemezvezérlő áramköre a merevlemeznél egyszerűbb lehet, mint a hajlékonylemeznél, aminek az oka, hogy a merevlemez-meghajtó egységbe be van építve egy hatékony elektronikus vezérlő. Az eszköz sajátossága, ami a lemezmeghajtóra jelentősen hat, hogy a vezérlő rendelkezik azzal a lehetőséggel, hogy egyszerre kettő vagy több meghajtóegységen is keressen. Ezt hívják **átlapolt keresés**nek. Mialatt a vezérlő és a szoftver vár az egyik meghajtóegységen való keresés befejezésére, a vezérlő elindíthat egy keresést egy másik meghajtóegységen. Sok vezérlő képes olvasni vagy írni az egyik egységen, mialatt keres egy vagy több más meghajtóegységen, azonban egy hajlékonylemez-vezérlő nem tud egyszerre két meghajtóegységen olvasni és írni. (Az olvasás vagy írás alkalmával a vezérlőnek a biteket mikroszekundum nagyságrendű idő alatt kell mozgatnia, így egy átvitel során felhasználja számolási erejének nagy részét.) Más a helyzet az integrált vezérlőkkel rendelkező merevlemezeknél és egy olyan rendszerben, amely egynél több merevlemez-meghajtó egységgel rendelkezik, ugyanis ekkor ezek egyidejűleg

működhetnek, legalábbis a lemez és a vezérlő puffermemóriája közötti átvitel során. A vezérlő és a rendszermemória között egyszerre csak egy átvitel lehetséges. Két vagy több művelet egy időben való végrehajtásának lehetősége tekintélyesen csökkentheti az átlagos hozzáférési időt.

A modern merevlemezek adatait megnézve láthatjuk, hogy a meghajtószoftver által használt geometriai szerkezet és a fizikai szerkezet majdnem mindig különböző. Valójában, ha valaki kikeresné egy nagy merevlemez számára „ajánlott betöltési paraméterek"-et, valószínű specifikációként 16 383 cilindert, 16 fejet és sávonként 63 szektort találna, függetlenül a lemez valódi méretétől. Ezek a számok egy 8 GB méretű lemeznek felelnek meg, de szokásosak az ilyen vagy nagyobb lemezeknél is. Az eredeti IBM PC tervezői egy 6 bites mezőt szántak a BIOS ROM szektorszámlálójának, 4 bitet a fej megadására és 14 bitet a cilinder kiválasztására. 512 bájtos szektoroknál ez 8 GB lesz. Így ha valaki egy nagyon régi számítógépbe próbál egy nagy lemezmeghajtót telepíteni, akkor azt találja, hogy csak 8 GB-ot képes elérni akkor is, ha nagyobb lemezzel rendelkezik. Ilyen korlátozás esetén rendszerint **logikai blokkcímzést** alkalmaznak, ami a lemez szektorait egymás után, a nullával kezdve számozza, anélkül hogy a lemez geometriáját figyelné.

Egy modern lemez geometriája csupán kitaláció. Egy modern lemezen a felület 20 vagy több zónára osztott. A lemez középpontjához közelebb levő zónák kevesebb szektort tartalmaznak sávonként, mint a széléhez közelebbi zónák. Így a szektorok hossza közelítőleg azonos, függetlenül a lemezen való elhelyezkedéstől, ami a lemezfelület hatékonyabb felhasználását eredményezi. Belsőleg, az integrált vezérlő a zónákat, cilindereket, fejeket és szektorokat kiszámolva címezi a lemezt. De ezt a felhasználó nem látja, és a részletek ritkán találhatók meg a publikált specifikációkban. Végül is semmi nem mutat cilinder, fej, szektor használatára a lemez címzésénél, ha csak valaki nem egy régi számítógéppel dolgozik, ami nem támogatja a logikai blokkcímzést. Nem érdemes egy új 400 GB-os meghajtót vásárolni egy 1983-ban vásárolt PC-XT-hez; 8 GB-nál többet úgysem lehetne elérni.

Itt az alkalom, hogy megemlítsünk egy zavaró dolgot a lemezkapacitás-specifikációkkal kapcsolatban. A számítógépes szakmában a 2 hatványok használata a megszokott – egy kilobájt (KB) $2^{10} = 1024$ bájt, egy megabájt (MB) $2^{20} = 1024^2$ bájt stb. – a memóriacszközök méretének kifejezésére. Egy gigabájtnak (GB) eszerint 1024^3 vagy 2^{30} bájtnak kellene lennie. De a lemezgyártó cégek úgy vették át a „gigabájt" terminológiát, hogy annak jelentése 10^9, ami (papíron) azonnal növeli a termékük méretét. Így a fent említett 8 GB-os korlát egy 8,4 GB-os lemez a lemezforgalmazók nyelvén. Mostanában elmozdulás van a gibibájt (GiB) alkalmazása felé; ennek jelentése 2^{30}. A szerzők ebben a könyvben a maguk módján tiltakoznak a hagyomány felrúgása ellen, és olyan fogalmakat, mint megabájt és gigabájt továbbra is úgy használnak, ahogy eddig.

3.7.2. RAID

Jóllehet a modern lemezek sokkal gyorsabbak, mint a régiek, a CPU teljesítményének javulása nagyban túlhaladta a lemezekét. Évek során sokaknak eszébe jutott, hogy a párhuzamos lemez I/O segíthet a helyzeten. Így alakult ki az I/O-eszközök új osztálya, a **RAID**, a **Redundant Array of Independent Disks** (független lemezek redundáns tömbje) angol elnevezésből. Eredetileg a RAID tervezői (Berkeley) a RAID szót mint a „Redundant Array of Inexpensive Disks" rövidítését használták, szemben a **SLED**-del (**Single Large Expensive Disk**). Amikor azonban kereskedelmileg népszerűvé vált a RAID, a lemezgyártó cégek megváltoztatták a RAID mozaikszó jelentését, mivel nem lett volna jó olcsó (inexpensive) névvel egy drága (expensive) terméket árulni. A RAID alapötlete egy lemezekkel teli egység telepítése a számítógéphez, általában egy nagy szerverhez úgy, hogy egy RAID-vezérlővel van kicserélve a lemezvezérlő kártya, az adatok elérése a RAID-en keresztül történik, a többi művelet pedig a szokásos módon megy végbe.

A független lemezek számos módon használhatók együtt. Ezek mindegyikének kimerítő leírására nincs módunk, és a MINIX 3 nem is támogatja (még) a RAID-et, de egy operációs rendszerekkel foglalkozó könyvben legalább néhány lehetőséget meg kell említenünk. A RAID használatával gyorsabb a lemezelérés és az adattárolás is biztonságosabb.

Példaként nézzünk egy kétmeghajtóegységes egyszerű RAID-et. Mikor több szektornyi adatot kell írni a „lemez"-re, a RAID-vezérlő a 0, 2, 4 stb. szektorokat az első meghajtóegységhez küldi, az 1, 3, 5 stb. szektorokat pedig a második meghajtóegységhez. A vezérlő szétosztja az adatot, és a két lemeznek az írása egyidejűleg történik, ezzel megduplázva az írási sebességet. Olvasáskor mindkét meghajtóegység párhuzamosan olvas, míg a vezérlő a valódi sorrendben összerakja az adatot; ezzel az olvasás sebessége kétszer olyan gyorssá válik. Ez **sávos módszer**ként ismert, amely a 0-s szintű RAID-nek egy egyszerű példája. A gyakorlatban négy vagy több meghajtóegységet szoktak alkalmazni. Ezek akkor dolgoznak a legjobban, amikor nagy blokkokban történik az olvasás vagy az írás. Természetesen semmi haszna, ha a tipikus eszközkérés egyszerre csak egyetlen szektorra vonatkozik.

Az előző példa rámutat, hogy a többszörös meghajtóegységek növelik a sebességet. És a biztonság? Az 1-es szintű RAID a 0-s szintű RAID-hez hasonlóan dolgozik, kivéve, hogy az adatokat duplikálja. Két meghajtóegység nagyon egyszerű tömbjét alkalmazva minden adat mindegyikre ráíródik. Ez nem okoz gyorsulást, viszont 100%-os redundanciát eredményez. Olvasási hiba észlelésekor nincs szükség újrapróbálkozásra, ha a másik meghajtóegység az adatot helyesen olvassa. A vezérlőnek csak azt kell biztosítania, hogy a helyes adat át legyen adva a rendszernek. Írás közbeni hiba észlelésekor valószínűleg nem lenne jó ötlet kihagyni az újrapróbálkozást. Ha a hibák elég gyakoriak, az ismételt végrehajtás átugrása észrevehetően gyorsabbá tenné az olvasást, ekkor viszont talán itt az idő bejelenteni: a teljes kudarc elkerülhetetlen. A RAID-ben alkalmazott meghajtóegységek jellemzően „melegen"cserélhetők, vagyis kicserélhetők a rendszer leállítása nélkül.

Többszörös lemezek bonyolultabb tömbjeiel mind a sebesség, mind a megbízhatóság növelhető. Példaként tekintsünk egy 7 lemezes tömböt. A bájtok 4 bites szavakba lennének osztva úgy, hogy az egyes bitek a négy meghajtóegység egyikén lennének rögzítve, a másik három meghajtóegység pedig arra lenne használva, hogy egy hárombites hibajavító kódot rögzítsenek. Ha elromlik egy meghajtóegység, és egy újjal kell „melegen"cserélni, akkor a hiányzó meghajtóegység egyenértékű a rossz bitessel, így a rendszer a karbantartás alatt is futhat. Hét meghajtóegység áráért megbízható teljesítmény érhető el, ami négyszer gyorsabb az egyetlen meghajtóegységnél, és nincs állásidő.

3.7.3. Lemezszoftver

Ebben a részben áttekintünk néhány kérdést, amelyek általában kapcsolódnak a lemezmeghajtókhoz. Először nézzük, mennyi időbe telik egy lemezes blokk olvasása vagy írása. A kért idő az alábbi három tényezőből határozható meg.

1. A keresési idő (az olvasófejnek a megfelelő cilinderhez történő mozgatásához szükséges idő).
2. A fordulási késés (a megfelelő szektornak az olvasófej alá fordulásához szükséges idő).
3. Az adatmozgatás tényleges ideje.

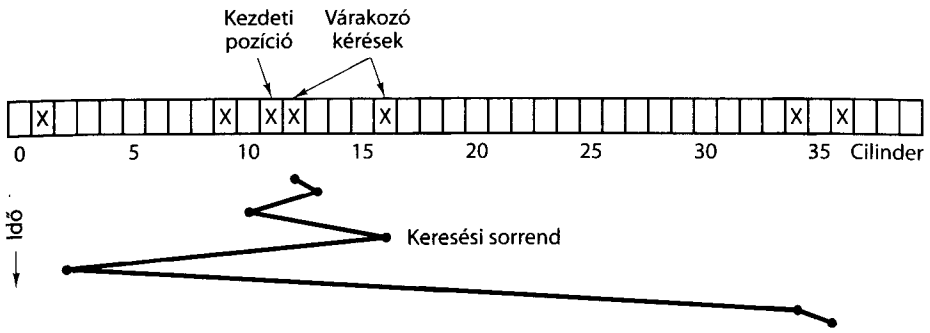
A legtöbb lemeznél a keresési idő lényegesen nagyobb a másik kettőnél, így a keresési idő csökkentésével a rendszer teljesítménye lényegesen javítható.

A lemezes eszközök hajlamosak a meghibásodásra. Vannak olyan hibaellenőrzések, amelyek egy ellenőrző összeget jegyeznek be a lemez minden szektorába. A szektorok címei, amelyek a lemez formázásakor íródnak fel, szintén rendelkeznek ellenőrző adattal. A hajlékonylemeznél a vezérlőhardver tudathatja a hiba észlelését, de a szoftver dönti el, hogy mit tegyen vele. A merevlemez-vezérlők gyakran átvállalják ennek a tehernek nagy részét.

A merevlemezeknél sajátságos, hogy az átviteli idő az egymást követő szektoroknál egy pályavonalon belül nagyon rövid. Így a kértnél több adat beolvasásával és memóriába helyezésükkel a lemez hozzáférési sebessége nagymértékben növelhető.

Lemezfej-ütemező algoritmusok

Ha a lemezmeghajtó a kéréseket egyesével fogadja, és végrehajtja azokat a beérkezés sorrendje szerint, vagyis **első-bejövő, első-kiszolgált (First-Come, First-Served, FCFS)** módszerrel, akkor keveset tehet a keresési idő optimalizálásáért. De egy másik stratégiára is lehetőség van, amikor a lemez erősen betöltött. Valószínű, hogy mialatt az olvasófej egy kérésnek megfelelően keres, közben más processzusok más lemezkérésekkel állnak elő. Sok lemezmeghajtó egy táblázatot kezel, amely a cilinderszámokkal van indexelve, és minden egyes cilinderszámhoz



3.21. ábra. *A legközelebbit keres elsőként (SSF) lemezütemező algoritmus*

létrehozza a még várakozó, az adott cilinderre vonatkozó kérések egy láncolt listáját, amelynek kezdete a táblázat cilinder indexű komponense.

Ha van egy ilyen adatszerkezet, akkor javíthatjuk az első-bejövő, első-kiszolgált ütemező algoritmust. Nézzünk például egy 40 cilinderes lemezt. Bejön egy kérés a 11. cilinderen levő egyik blokk olvasására. Mialatt a 11. cilinder keresése folyamatban van, újabb kérések érkeznek a cilinderekre, érkezési sorrendben az 1., 36., 16., 34., 9. és 12. cilinderre. Ezek most bekerülnek a várakozó kérések táblázatába, mégpedig mindegyik a cilinderszámnak megfelelő láncolt listába. A 3.21. ábra mutatja a kéréseket.

Amikor az aktuális kérés (11. cilinderre) befejeződik, akkor a lemezmeghajtó választás elé kerül, hogy melyik kérést kezelje következőként. Ha az FCFS stratégiát választja, akkor a következő az 1. cilinderre vonatkozó kérés lenne, utána a 36. cilinderre, és így tovább. Ez az algoritmus megkövetelné, hogy az olvasófej egymást követően 10, 35, 20, 18, 25 és 3 cilinderrel mozduljon el, ami összesen 111 cilinderrel való elmozdulást jelent.

A másik lehetőség, hogy a következő kérésnek a cilinderszámban legközelebbit választja, így minimalizálva a keresési időt. A 3.21. ábrán adott példa esetében a kérések most említett módon való ütemezésénél a sorrend 12, 9, 16, 1, 34, 36, amit az ábra alsó részében levő törtvonal szemléltet. E sorozatnál az olvasófej elmozdulása 1, 3, 7, 15, 33 és 2, ami összesen 61 cilindernyi. Ez az algoritmus a **legközelebbit keres elsőként (Shortest Seek First, SSF)** módszer, amely az FCFS-hez képest az összes olvasófej mozgatások mennyiségét majdnem a felére csökkenti.

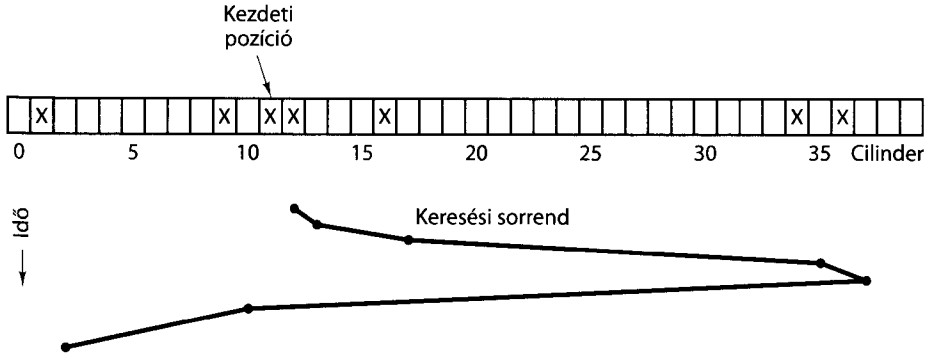
Sajnos az SSF esetén is van probléma. Tegyük fel, hogy több kérés érkezik folyamatosan, mialatt a 3.21. ábrán levő kérések lezajlanak. Például ha a 16. cilinderre elmozdul az olvasófej, egy új kérés jelenik meg a 8. cilinderre, akkor ez a kérés előnyt élvez az 1. cilinderrel szemben. Ha ezután egy kérés a 13. cilinderre jön be, akkor az olvasófej következő elmozdulási célja a 13. lesz az 1. helyett. Egy nagyon sűrűn betöltött lemez esetén az olvasófej várhatóan az idő legnagyobb részében a lemez közepén marad, így az ettől a helytől távolabb levő kéréseknek addig kell várniuk, míg az a helyzet áll elő, hogy nincs több középre vonatkozó kérés. A középtől távoli kérések kiszolgálása így nagyon lassúvá válhat. Összeütközésbe kerül két cél: a minimális válaszidő és a méltányosság.

Magas épületekben szintén foglalkozni kell ezzel az összefüggéssel. Egy magas épületben a lift ütemezésének problémája hasonló egy lemez olvasófejének az ütemezéséhez. A liftet hívó kérések véletlenszerűen folytonosan jönnek az emeletekről (cilinderek). A liftet működtető mikroprocesszornak folyamatosan nyomon kellene követnie azt a sorrendet, amelyben az ügyfelek a hívógombot megnyomták, ha az FCFS stratégiát használja. De használhatná az SSF módszert is.

Azonban a legtöbb lift ezektől különböző algoritmust alkalmaz, hogy áthidalja a hatékonysági és a méltányossági célok között feszülő konfliktust. Folyamatosan ugyanabba az irányba mozog, ameddig van abban az irányban el nem intézett kérés, ekkor irányt változtat. Ez az algoritmus mind a lemezek, mind a liftek világában úgy ismert, mint a **liftes algoritmus**, és megkívánja a szoftvertől, hogy kezeljen egy bitet: az aktuális irányt jelző bitet, amelynek értéke *UP* vagy *DOWN*. Mikor egy kérés befejeződik, akkor a lemez vagy a lift meghajtója ezt a bitet vizsgálja meg. Ha az értéke *UP*, akkor az olvasófej vagy a liftszekrény a legközelebbi magasabb várakozó kérés felé mozdul. Ha nincs várakozó kérés a magasabb pozícióknál, akkor az irányt jelző bitet ellenkezőre változtatja. Amikor a bit *DOWN*-ra van beállítva, akkor a mozgás a legközelebbi alacsonyabb kérés helyére történik, ha van ilyen.

A 3.22. ábra a liftes algoritmust mutatja, ugyanarra a hét kérésre alkalmazva, mint a 3.21. ábrán, feltéve, hogy az irányt jelző bit kiinduláskor *UP* értékű. A sorrend, amely szerint a cilinderek kiszolgálása megtörténik, 12, 16, 34, 36, 9 és 1, és az ezekhez szükséges olvasófej-mozgatások 1, 4, 18, 2, 27 és 18, vagyis az összes mozgatás 60 cilindernyi. Ennél a példánál a liftes algoritmus egy kicsivel jobb az SSF-nél, bár általában rosszabb. A liftes algoritmusnak van egy kellemes tulajdonsága, miszerint a kérések tetszőleges halmazára rögzített összes mozgatások számára van felső korlátja: ez a cilinderek számának a kétszerese.

A válaszidőkben kisebb szórás (méltányossági cél) érhető el az előző algoritmus kisebb módosításával (Teory, 1972): eszerint mindig azonos irányban történik a mozgatás. Amikor a megválaszolatlan kérésekkel rendelkező, legnagyobb számú cilinder kiszolgálása megtörténik, az olvasófej ahhoz a legkisebb számú cilinderhez mozdul el, amelyhez van megválaszolatlan kérés, és utána folyamatosan felfelé mozog. Itt a legkisebb számú cilindert úgy kezeljük, mintha közvetlenül a legnagyobb számú cilinder fölött lenne.



3.22. ábra. *A lemezkérések liftes algoritmus szerinti ütemezése*

Néhány lemezvezérlő módot ad arra, hogy szoftvereszközökkel vizsgálható legyen az olvasófej alatti szektorszám. Az ilyen vezérlők esetében egy másik optimalizálás is lehetséges. Ha egy cilinderre vonatkozóan két vagy több megválaszolatlan kérés is van, akkor a meghajtó arra a szektorra vonatkozó kérést adhatja ki, amely a következőkben az olvasófej alatt elhalad. Vegyük észre, ha a cilinder több pályavonalból áll, akkor az egymást követő kérések büntetlenül vonatkozhatnak a különböző pályavonalakra (egy cilinderen belül). A vezérlő az olvasófejek közül bármelyiket azonnal kiválaszthatja, mivel az olvasófej kiválasztása nem igényel semmilyen fejmozgatást vagy fordulási késleltetést.

Az adatátvitel lebonyolítási sebességében a modern merevlemez és hajlékonylemez között olyan nagymértékű az eltérés az első javára, hogy a gyorsítótárazás (caching) valamilyen fajtája szükséges. Egy szektor olvasására vonatkozó kérés tipikusan a szektornak és a pályavonalán levő, ezt követő, bizonyos számú szektoroknak a vezérlő gyorsítótárába való elhelyezését eredményezi, az átmozgatott szektorok száma a vezérlő gyorsítótárában levő hozzáférhető helyek mennyiségétől függ. A mai gyorsítótárak gyakran 8 MB-osak, vagy ennél is nagyobbak.

Több meghajtóegység esetén a rájuk vonatkozó megválaszolatlan kérések számára külön-külön táblázatot kell kezelni. Minden egységnek egy keresési kérést kell kiadni, hogy mozdítsa az olvasófejet arra a cilinderre, ahonnan a következő kérést majd teljesítheti (feltéve, hogy a vezérlő megengedi az átlapolt kereséseket). Amikor az aktuális átvitel befejeződik, akkor egy vizsgálatot kell végeznie, amelynek során megnézi, hogy mely egységek pozicionáltak már a helyes cilinderre. Ha egy vagy több ilyen van, akkor a következő átvitel elkezdhető az egyik, már jó cilinderre pozicionált egységen. Ha egyik olvasófej sincs a megfelelő helyen, akkor a meghajtónak ki kell adnia egy új keresést arra a meghajtóegységre, amely éppen befejezte az átvitelt, és várakozik a következő megszakításig, amikor is megnézi, hogy melyik olvasófej érte el elsőként a célhelyet.

Hibakezelés

A RAM-lemezeknek nem kell törődniük a keresés és fordulási késleltetés optimalizálásával: mindegyik blokk azonnal olvasható vagy írható bármiféle fizikai mozgatás nélkül. Egy másik terület a hibakezelés, amiben a RAM-lemezek egyszerűbbek, mint a valódi lemezek. A RAM-lemezek mindig dolgoznak, míg a valódi lemezek nem mindig. A valódi lemezeknél sokféle hiba lehetséges. A közismertebb hibák közül néhány:

1. Programozási hiba (például nem létező szektor kérése).
2. A hibajavító kód átmeneti hibája (például az olvasófejen levő porszem is okozhatja).
3. A hibajavító kód tartós hibája (például a lemez blokkja fizikailag károsodott).

4. Keresési hiba (például a 6. cilinderhez küldött olvasófej a 7. cilinderhez mozdul).
5. A vezérlő hibája (például a vezérlő elutasítja a parancsok elfogadását).

A lemezmeghajtó dolga, hogy a lehető legjobb módon kezelje ezeket a hibákat.

Programozási hibák fordulnak elő, amikor a meghajtó azt mondja a vezérlőnek, hogy keressen egy nem létező cilindert, olvasson egy nem létező szektorból, használjon egy nem létező olvasófejet, vagy hajtson végre átvitelt egy nem létező memóriából. A legtöbb vezérlő ellenőrzi a számára átadott paramétereket, és ha azok érvénytelenek, akkor panaszkodik. Elvileg ezeknek a hibáknak soha nem kellene előfordulniuk, de mit tegyen a meghajtó, ha a vezérlő jelzi valamelyik hiba bekövetkeztét? Egy házilagosan kifejlesztett rendszernél a legjobb, ami tehető, a megállás és egy üzenet kinyomtatása, mondjuk, „Hívd a programozót!", így a hiba nyomon követhető és kijavítható. Egy kereskedelmi szoftvertermék esetében, amelyet az egész világon tömegesen használnak, az előbbi megközelítés már kevésbé vonzó. Valószínűleg az egyetlen dolog, amit tehetünk, a pillanatnyi lemezkérés hibával való befejezése, remélve, hogy a hiba túl gyakran nem ismétlődik meg.

A hibajavító kód ideiglenes hibáját okozhatják a levegőben levő porszemcsék azzal, hogy az olvasófej és a lemez felülete közé jutnak. A művelet néhányszori megismétlésével a hiba a legtöbb esetben eltűnik. Ha a hiba továbbra sem szűnik meg, akkor a blokkot **hibás blokk**ként kell megjelölni, és a továbbiakban kerülni kell a használatát.

A hibás blokkok elkerülésének egyik módja egy nagyon speciális program írása, amely bemenetként veszi a hibás blokkokat, és létrehoz egy állományt, amely az összes hibás blokkot tartalmazza. Ha egyszer ez az állomány megvan, akkor a további helyek lefoglalásakor ezek a blokkok már lefoglaltakként jelennek meg, így újabb lefoglalásuk nem lehetséges. Egész addig nem fordul elő probléma, amíg valaki meg nem próbálja olvasni a hibás blokkok állományát.

Ne olvassuk a hibás blokkok állományát – ezt könnyebb mondani, mint megvalósítani. Sok lemezről készül úgy biztonsági mentés, hogy a tartalmát pályavonalanként egy háttérszalagra vagy lemezegységre írják. Ezt az eljárást követve a hibás blokkok problémát okoznak. A lemez tartalmának állományonként való mentése lassabb folyamat, de a probléma elkerülhető vele, ha a mentőprogram ismeri a hibás blokkok állományának nevét, és tartózkodik azok másolásától.

Egy másik probléma, ami nem oldható meg a hibás blokkokat tartalmazó állományokkal, ha egy hibás blokk a fájlrendszer egy olyan adatszerkezetében van, amelynek helye rögzített. Majdnem minden fájlrendszernek van legalább egy rögzített helyű adatszerkezete, mert ez könnyen megtalálható. Egy particionált fájlrendszernél ilyenkor lehetőség van az ismételt particionálásra, kihagyva a hibás pályavonalakat, de ha az állandó hiba egy hajlékonylemez vagy egy merevlemez első néhány szektorában van, az általában a lemez használhatatlanságát jelenti.

Intelligens vezérlők néhány pályavonalat, amelyek a felhasználói programokból normális esetben nem érhetők el, tartalékként foglalnak le. Mikor egy lemezegység formázása történik, akkor a vezérlő feltárja a hibás blokkokat, és a hibásat automatikusan helyettesíti egy tartalék pályavonallal. A vezérlő belső memóriájában és a lemezen tárolásra kerül az a táblázat, amely a hibás pályavonalaknak a tartalékokba való leképezését tartalmazza. Ez a helyettesítés közvetlenül nem érinti (láthatatlan a táblázat) a meghajtót, kivéve, hogy a gondosan kimunkált liftes algoritmus nagyon szegényessé válik, ha például a vezérlő titkon a 800. cilindert használja a 3. cilinder helyett. A gyári lemezek felületénél alkalmazott technológiák sokat javultak, de még nem hibátlanok. Azonban a tökéletlenség felhasználók előli elrejtésének technológiája szintén javult. Sok vezérlő a használat során kialakuló új hibákat is kezeli, a helyrehozhatatlan hibák észlelésekor állandóan kiutalva helyettesítési blokkokat. Az ilyen lemezeknél a meghajtószoftver ritkán látja bármi jelét a hibás blokkok létezésének.

A keresési hibákat az olvasófej mechanikus problémái okozzák. A vezérlő nyomon követi az olvasófej helyzetét. Egy keresés végrehajtásánál impulzusok sorozatát adja az olvasófej motorjának, cilinderenként egy impulzust, hogy az olvasófejet az új cilinderhez mozdítsa. Amikor az olvasófej eléri a célhelyét, a vezérlő kiolvassa az aktuális cilinderszámot (az eszköz formázásakor íródik fel). Ha az olvasófej rossz helyen van, akkor keresési hiba jelentkezik, és javítási tevékenységre kerül sor.

A legtöbb merevlemez vezérlője automatikusan kijavítja a keresési hibákat, de sok hajlékonylemez-vezérlő (idetartozik az IBM PC is) csupán a hibát jelző bitet állítja be, és a többit a meghajtóra hagyja. A meghajtó ezt a hibát egy recalibrate parancs kiadásával kezeli; ez elmozdítja az olvasófejet, amennyivel csak lehet, és beállítja az aktuális cilinder vezérlő által kezelt belső jelzését a 0 értékre. Ez a módszer rendszerint megoldja a problémát. Ha mégsem, akkor a meghajtóegységet meg kell javítani.

Az eddigiekből látható, hogy a vezérlő valójában egy speciális, kis számítógép szoftverrel, változókkal, pufferekkel és esetenként hibákkal. Váratlan események sorozata néha előidézheti, hogy a vezérlő ciklusba kerül, vagy elveszít egy pályavonalat, amelyen éppen dolgozik. Ilyen hibát válthat ki egy megszakítási kérés az egyik eszközön, ami egy másik eszközre alkalmazott recalibrate paranccsal egyidejűleg fordul elő. A vezérlők tervezői általában gondolnak a legrosszabbra is, így a lapkán egy olyan áramköri elemet alakítottak ki, ami működése esetén eléri, hogy a vezérlő felejtse el azt, amit eddig tett, és állítsa be újra magát. Ha minden sikertelen, akkor a lemezmeghajtó beállíthat egy bitet, amelynek hatására segítségül hívódik az előbb említett áramköri elem, és újraindítja a vezérlőt. Ha ez sem segít, akkor a meghajtó csak annyit tehet, hogy kinyomtat egy üzenetet és feladja.

Pályavonalankénti raktározás

Egy új cilinder megkereséséhez szükséges idő rendszerint sokkal nagyobb, mint az olvasófej alá való fordulási késés, és mindig sokkal nagyobb, mint egy szektor átviteli ideje. Más szavakkal, ha a meghajtó az olvasófej valamely helyre való elmozgatásának nehézségén túljutott, akkor már szinte egyre megy, hogy csak egy szektort vagy egy teljes pályavonalat olvas be. Ez a megállapítás még inkább igaz, ha a vezérlő érzékeli a fordulást, és így a meghajtó láthatja az olvasófej alatt levő

szektort, és kiadhat egy kérést a következő szektorra. Ezzel a módszerrel egyetlen körülfordulási idő alatt lehetséges egy teljes pályavonal olvasása. (Normálisan egy fél fordulatot tesz, és csak egyetlen szektort olvas átlagban.)

Néhány lemezmeghajtó kihasználja ezt a tulajdonságot egy rejtett, pályavonalankénti gyorsítótárba raktározás kezelésével. Nincs szükség semmi lemezes átvitelre, amikor egy gyorsítótárban levő szektorra van igény. A pályavonalankénti gyorsítótárba raktározás egyik hátránya (túl azon, hogy növeli a szoftver bonyolultságát és pufferhelyet igényel), hogy a raktár és a hívóprogram közötti átvitelt CPU használatával kell végrehajtani, egy beprogramozott ciklust alkalmazva, ahelyett hogy a munkát a DMA-hardverrel végezné el.

Bizonyos vezérlők ezt a folyamatot egy lépéssel továbbviszik, és a pályavonalankénti raktárnak a saját belső tárukat használják, amely láthatatlan a meghajtó számára, így az átvitelt a vezérlő és a memória között a DMA bonyolítja le. Az ilyen módon dolgozó vezérlőknél a lemezmeghajtóra a munka nagyon kicsi része marad. Megjegyezzük, hogy a vezérlő és a meghajtó is abban a jó helyzetben van, hogy egyetlen parancsra egy teljes pályavonalat olvas vagy ír, míg az eszközfüggetlen szoftver nem képes erre, mivel a lemezt blokkok lineáris sorozataként tekinti, a lemez pályavonalakba és cilinderekbe való osztottsága nélkül. Csak a vezérlő ismeri a valódi geometriát.

3.7.4. A MINIX 3 merevlemez-meghajtója

A merevlemez-meghajtó a MINIX 3 első olyan része, amellyel kapcsolatban már láttuk, hogy a hardverelemek különböző típusú széles körét kell kezelnie. Mielőtt a meghajtót tárgyalnánk, röviden megnézünk néhány kérdést, amelyek a hardverek különbözőségéből erednek.

A „PC" valójában különböző számítógépeknek egy családja. Nemcsak különböző processzorokat használnak a család különböző tagjaiban, hanem néhány alapvető különbség is van az alaphardverben. A MINIX 3-at a Pentium osztályú CPU-val rendelkező legújabb rendszerekre fejlesztették ki, jóllehet ezek között is vannak különbségek. Például a legrégebbi Pentium-rendszerek 16 bites AT-síneket használnak, amelyek eredetileg a 80286-os processzorra voltak tervezve. Az AT-sínt okosan tervezték, mert a régebbi 8 bites perifériák is használhatók maradtak. A későbbi rendszereket 32 bites PCI-sínnel bővítették a perifériák számára, mialatt az AT-sínt is megőrizték. A legújabb tervekből már kidobták az AT-sínt és csak egy PCI-sínt biztosítanak. Viszont indokolt elvárása a régebbi számítógéppel rendelkezőknek, hogy akár 8, 16 vagy 32 bites perifériákkal is használhassák a MINIX 3-at.

Minden sínhez az **I/O-adapterek**nek más-más családja tartozik. Régebbi rendszereken külön áramköri lapok vannak, amelyek az alaplapba vannak csatlakoztatva. Az újabb rendszereknél sok szabványos adapter, különösen a lemezvezérlők az alaplap integrált komponensei. Ez nem jelent korlátozást a programozók számára, mivel az integrált adapterek szokásosan szoftverinterfésszel rendelkeznek a cserélhető eszközök számára. Sőt az integrált vezérlők rendszerint megbéníthatók. Így a beépített eszköz helyett lehetővé válik olyan továbbfejlesztett bővítő

eszközök használata is, mint például az SCSI-vezérlő. Ennek a rugalmasságnak az az előnye, hogy az operációs rendszert nem kell korlátozni csupán egyetlen adaptertípus használatára.

Az IBM PC családban, mint ahogy a legtöbb számítógéprendszernél, mindegyik adatcsatornát az alap I/O-rendszer csak olvasható memóriájában (Basic Input/Output System Read-Only Memory, BIOS ROM) levő szoftver (firmware) kezeli, amely tulajdonképpen a híd szerepét tölti be az operációs rendszer és a hardver sajátosságai között. Néhány perifériás eszközhöz a ROM-beli BIOS-on túl még perifériás kártyák is tartozhatnak. Az operációs rendszer tervezőjének azzal a nehézséggel kell szembenéznie, hogy az IBM típusú számítógépeknél a BIOS-t konkrét operációs rendszerhez tervezték, mégpedig az MS-DOS-hoz, amely nem támogatja a multiprogramozást, és 16 bites valós módban fut, amely a legalacsonyabb szintű közös módja a 80x86-os CPU családból rendelkezésre álló különféle műveleti módoknak.

Ha valaki az IBM PC-hez egy új operációs rendszert készít, akkor több lehetősége van. Egyik ezek közül, hogy a perifériákhoz meghajtóként a BIOS-beli szoftvereket használja, vagy ír új meghajtókat, a gyors előmemóriába elhelyezve. Ez nem volt nehéz választás a MINIX korai verziói esetén, mivel a BIOS sok esetben nem alkalmas az igényeikhez. Természetesen a MINIX 3 indításánál a betöltési felügyelőprogram a BIOS-t használja a rendszer kezdeti betöltéséhez akár merevlemezről, akár CD-ROM-ról, akár hajlékonylemezről – ennek nincs praktikus alternatívája. Ha viszont a rendszer már be van töltve a saját I/O-meghajtóival együtt, akkor a BIOS-nál sokkal jobban is megoldhatjuk.

Ekkor a második választási lehetőség jelenik meg: hogyan illeszthető a meghajtónk BIOS nélkül a hardver változatos fajtáihoz a különböző rendszereknél? Mivel a modern 32 bites Pentium-rendszeren, amelyre a MINIX 3-at tervezték, a merevlemez-meghajtóknak két alapvető típusa van, ezért a probléma konkrétabb vizsgálatához tekintsük az integrált IDE-vezérlőt és a PCI-síneknél alkalmazható SCSI-vezérlőket. Ha valaki szeretné a régebbi hardverét használni és a MINIX 3-at alkalmazni, hogy a korábbi MINIX-verzióknak megfelelő hardveren dolgozzon, akkor négy merevlemez-vezérlő típust kell megfontolnia: az eredeti 8 bites XT típusú vezérlő, a 16 bites AT típusú vezérlő és két különböző típusú vezérlő az IBM PS/2 soros számítógépekhez. Az összes ilyen helyzet kezelésére számos lehetőség van:

1. Mindegyik merevlemez-vezérlő esetében, amit illeszteni akarunk, újrafordítjuk az operációs rendszer megfelelő változatát.
2. A betöltési memóriaképbe fordítjuk a különböző merevlemez-meghajtókat, és a rendszer automatikusan eldönti az induláskor, hogy melyiket használja.
3. A betöltési memóriaképbe fordítjuk a különböző merevlemez-meghajtókat, és a felhasználóra bízzuk, hogy eldöntse, melyiket használja közülük.

Mint látjuk majd, ezek az esetek nem zárják ki egymást.

Hosszú távon valójában a legjobb mód az első. Egy konkrét üzembe helyezésnél felesleges lenne sosem használt meghajtók kódjainak lemezen vagy memóriá-

ban való tárolása. Ez azonban a szoftverforgalmazók számára rémálom lenne. Ekkor ugyanis a felhasználóknak négy különböző indítólemezt kellene adniuk, és egy használati útmutatót, ami drága és nehézkes módszer. Így legalábbis a kezdeti üzembe helyezésnél másik alternatívát tanácsos választani.

A második módszernél az operációs rendszernek a kártyák azonosításához le kell tapogatni a perifériákat a kártyákon levő ROM-ot olvasva, vagy az I/O-kapukat írva és olvasva. Ez lehetséges (és az új IBM típusú rendszereknél jobb is, mint a régebbieken), viszont a nem szabványos I/O-eszközöknél nem alkalmazható. Egy eszköz azonosításánál az I/O-kapuk letapogatása néha működésbe hozhat egy másik eszközt, amely megszerezheti a vezérlést és megbéníthatja a rendszert. Ezzel a módszerrel bonyolult az eszközök kódjának kialakítása, és ráadásul nem is működik jól. Az ilyen módszert alkalmazó operációs rendszereknek általában rendelkezniük kell bizonyos, megsemmisítő mechanizmussal, tipikusan ilyen, amit a MINIX 3 használ.

A MINIX 3-ben használatos a harmadik módszer is, ami engedi, hogy számos meghajtó a betöltési memóriaképben legyen. A MINIX 3 betöltési felügyelőprogram az elindításkor több **indítóparaméter** beolvasását engedi. Ezek beadhatók kézi úton vagy egy lemezen vannak állandó jelleggel tárolva. Beindításkor, ha egy betöltési paraméter alakja

```
label = AT
```

akkor az IDE-lemezvezérlőt *(at_wini)* jelöli ki, mint ami a MINIX 3 indításakor használható. Ez függ attól, hogy a címkéhez az *at_wini* vezérlő mit jelöl ki. A címkék hozzárendelése a betöltési memóriakép fordításakor történik.

A többszörös merevlemez-meghajtókkal kapcsolatos problémák minimalizálására a MINIX 3 két másik dolgot is tesz. Az egyik az, hogy van még egy meghajtója, ami összeilleszti a MINIX 3-at és a ROM BIOS merevlemezes támogatást. Ez a meghajtó majdnem garantáltan működik minden rendszerben és az alábbi indítóparaméterrel alkalmazható:

```
label = BIOS
```

Ezt azonban csak végszükség esetén szabad használni. A MINIX 3, amint már említettük, csak védett módban fut 80386-os vagy ennél jobb processzorú rendszereken, de a BIOS-kód mindig valós (8086) módban fut. A védett mód kikapcsolása, majd visszakapcsolása minden BIOS-beli rutin meghívásakor, nagyon lassú módszer.

A másik stratégia, amelyet a MINIX 3 a meghajtóknál alkalmaz, hogy a legutolsó pillanatig elhalasztja a beállítást. Így ha bizonyos hardverkiépítettség mellett a merevlemez-meghajtó egyáltalán nem működik, még akkor is elindítható a MINIX 3 egy hajlékonylemezről, és megfelelően dolgozhat. Nem lesz semmi problémája a MINIX 3-nak, míg kísérletet nem tesz egy merevlemez elérésére. Lehet, hogy ez a felhasználó szempontjából nem tűnik valami óriási áttörésnek, de nézzük a következő szituációt: ha megpróbálnánk beindítani az összes meghajtót már a rendszer indításakor, a rendszer teljesen megbénulhatna olyan eszközök helytelen konfigurációja miatt, amelyekre esetleg nincs is szükség. Ha az egyes meghajtók indítása késleltethető a használatukig, akkor a rendszer a működőképesekkel dolgozhat, és közben a felhasználó kideríthetí és megszüntetheti a hibát.

Egy kis kitérőként elmondhatjuk, hogy a lecke nagyon kemény volt: a MINIX korábbi változatai megpróbálták a merevlemez beállítását, amint a rendszer indítása megtörtént. Ha nem volt jelen merevlemez, akkor a rendszer felfüggesztődött. Ez a viselkedésmód különösen szerencsétlen volt, mivel a MINIX tökéletesen futhat egy merevlemez nélküli rendszeren is, persze korlátozott tárolókapacitás mellett és kisebb teljesítménnyel.

Ebben és a következő szakaszban a vizsgálataink során modellként az AT típusú merevlemez-meghajtót vesszük, ami a szabványos MINIX 3 felépítésében az alapértelmezett meghajtó. Ez egy sokoldalú meghajtó, amely a merevlemez-vezérlőket kezeli a legkorábbi 80286-os rendszerben használtaktól a modern **EIDE (Extended Integrated Drive Electronics – kiterjesztett integrált meghajtóelektronika)** gigabájt kapacitású vezérlőkig. A modern IDE-vezérlők támogatják a szabványos CD-ROM-meghajtót. A merevlemez-művelet általános szempontjai, amelyeket ebben a részben tárgyalunk, alkalmazhatók a többi támogatott meghajtóra is.

A merevlemez taszkjának főciklusa ugyanaz a közös kód, amelyet már tanulmányoztunk, és a kérések szabványos kilenc típusát támogatja. A *DEV_OPEN* kérés tekintélyes mennyiségű munkával járhat, mivel a merevlemezen mindig vannak partíciók és lehetnek részpartíciók is. Egy eszköz megnyitásakor (vagyis az első elérésekor) ezeket be kell olvasni. A CD-ROM-támogatásnál a *DEV_OPEN*-nek ellenőriznie kell az eszköz jelenlétét, mivel az cserélhető. A CD-ROM esetében a *DEV_CLOSE* műveletnek szintén van értelme: az ajtót ki kell nyitni, és a CD-ROM-ot ki kell adni. A hajlékonylemez-meghajtó egységeknél még inkább jelentkeznek a cserélhető eszközökkel kapcsolatos bonyodalmak; ezeket egy későbbi szakaszban tanulmányozzuk. A CD-ROM-nál a *DEV_IOCTL* művelet beállít egy jelzőt annak jelzésére, hogy a *DEV_CLOSE* műveletnek ki kell-e dobnia a lemezt a meghajtóból. A *DEV_IOCTL* művelet a partíciós táblázat írására és olvasására használható.

A *DEV_READ, DEV_WRITE, DEV_GATHER* és a *DEV_SCATTER* kérések mindegyikének kezelése, ahogy azt az előzőkben mutattuk, két fázisban történik: előkészítés és átvitel. A merevlemeznél a *DEV_CANCEL* és a *DEV_SELECT* hívások nem jelennek meg.

A merevlemez-meghajtó egyáltalán nem ütemez, ezt a fájlrendszer teszi, amely összerak egy vektort az összegyűjt/leoszt I/O-kérésekből. A kérések a fájlrendszer háttértárából jönnek mint többszörös blokkokra vonatkozó *DEV_GATHER* vagy *DEV_SCATTER* kérések (a MINIX 3 alapértelmezésű konfigurációjában 4 KB), viszont a merevlemez-meghajtó képes egy szektor (512 bájt) tetszőleges egészszám-szorosának megfelelő kéréseket kezelni. Mint ahogy láttuk, minden esetben az összes meghajtó főciklusa transzformálja az egyszerű blokkokra vonatkozó kéréseket a kérések vektorának egy elemébe.

Egy kérésvektorban az olvasási és írási kérések nem fordulhatnak elő sem vegyesen, sem opcionálisként jelölve. Egy kérésvektor elemei egymás utáni lemezszek-

torokra vonatkoznak, és a fájlrendszer rendezi a vektort, mielőtt átadja az eszközmeghajtónak, így elegendő megadni a kiindulási lemez helyét a kérések számára.

A meghajtótól folytonos olvasást vagy írást várunk el, legalábbis a kérésvektor első kérésénél, és hibás kérés esetén azonnali visszatérést. A fájlrendszerre van bízva, hogy mit tesz; a fájlrendszer megpróbál végrehajtani egy írás műveletet, de csak akkor tér vissza a hívó processzushoz, amikor egy olvasás adatait elérte.

A tagolt I/O-t használó fájlrendszer képes megvalósítani a liftes algoritmus Teory-változatához hasonló valamit – emlékeztetünk, hogy egy tagolt I/O-kérésben a kérések listája a blokkszám szerint rendezett. Az ütemezés második lépése a modern merevlemeznél a vezérlőben történik. Az ilyen vezérlők okosak és nagy mennyiségű adat pufferezésére képesek, belsőleg beprogramozott algoritmusokat használnak az adatok leghatékonyabb sorrendbe való rendezésére, figyelmen kívül hagyva a kérések beérkezési rendjét.

3.7.5. A merevlemez-meghajtó megvalósítása MINIX 3-ban

A mikroszámítógépekben használt kis merevlemezeket néha „winchester" lemezeknek hívják. Ez egy IBM-kódnév volt arra a feladatra, hogy kifejlesszenek egy olyan lemeztechnológiát, amelyben az író/olvasó fej egy vékony légpárnán röpül, és leszáll a rögzítőeszközre, amikor a forgás abbamarad. A név magyarázata, hogy a korai modell két adatmodullal rendelkezett, egy 30 megabájtos rögzített és egy 30 megabájtos cserélhető modullal, és ez feltehetően a fejlesztőket a Winchester 30-30 lőfegyverre emlékeztette, amely az Egyesült Államok nyugati határterületeinek legendás fegyvere volt. Bármi is legyen a név eredete, az alapvető technológia azonos, bár napjaink tipikus PC-lemezei sokkal kisebbek, és a kapacitásuk sokkal nagyobb, mint a 14 hüvelykes lemezeké, amelyek az 1970-es évek elején voltak tipikusak, amikor a winchester-technológiát kifejlesztették.

A MINIX 3 AT típusú merevlemez-meghajtója az *at_wini.c*-ben van (12100. sor). Ez egy intelligens eszköz bonyolult meghajtója; a vezérlő regisztereit, állapotbitjeit és parancsait, az adatszerkezeteket és típusokat leíró makródefiníciók több oldalt tesznek ki. Mint más blokkoseszköz-meghajtóknál, a *driver* adatszerkezet *w_dtab* (12316–12331. sor) kezdeti beállítása az aktuálisan dolgozó függvényekre vonatkozó mutatókkal történik. Ezek legtöbbjének definíciója az *at_wini.c*-ben van, de mint a merevlemez esetében, itt sem szükséges a tisztogató művelet, így a *dr_cleanup* belépési pontja a *driver.c*-ben levő, más meghajtókkal megosztva használt, közös *nop_cleanup*-ra mutat. További olyan függvények is vannak, amelyek lényegtelenek a meghajtó számára; ezek szintén *nop_* függvényekkel indulnak. Az *at_winchester_task* indító függvénye (12336. sor) egy eljárást hív, amely elvégzi a hardverfüggő kezdeti beállításokat, és utána meghívja a *driver.c*-ben levő főciklust, átadva a *w_dtab* címét. A *libdriver/driver.c*-ben levő *driver_task* főciklus vég nélkül fut, függvényeket hívogatva a *driver* táblázat segítségével.

Mivel most a tényleges elektromechanikus tárolóeszközökkel foglalkozunk, a munka jelentős részét a merevlemez-meghajtó kezdeti beállításának az *init_params*-szal való (12347. sor) elvégzése teszi ki. A merevlemez különféle paramétereit a

wini vektor tartja nyilván; ennek definiálása a 12254–12276. sorban van, amelyekben a *MAX_DRIVES* (8) meghajtóegységek mindegyikének van egy eleme, vagyis négy hagyományos IDE-meghajtónak és négy PCI-sínen levő meghajtóegységnek, amelyek lehetnek akár IDE-csatolású vezérlők, akár **SATA- (Serial AT Attachment – soros AT-csatolás)** vezérlők.

A kezdeti beállítás lépéseinek késleltetési szabálya szerint akkor kerül sor ezekre a lépésekre, amikor igazán szükségessé válnak az első alkalommal, ezt megelőzően hibához vezethetnének; az *init_params* semmi olyat nem tesz, amihez a lemezvezérlőt el kellene érni. A fő feladat, hogy bizonyos, a merevlemez logikai konfigurációjával kapcsolatos információt bemásoljon a *wini* vektorba. Egy pentiumos számítógépen a ROM BIOS az alapkonfigurációs információt a CMOS memóriából nyeri ki, amely az adatok megőrzésére használatos. A BIOS ezt akkor hajtja végre, amikor a számítógépet először bekapcsolják, mielőtt a MINIX 3 betöltőprocesszusa elkezdődik. A 12366–12392. sorban levő információ a BIOS-ból van másolva. Sok olyan konstanst használ, mint például az *NR_HD_DRIVES_ANDR*, amelynek a definíciója az *include/ibm/bios.h*-ban van; ez a MINIX 3 CD-ROM-on megtalálható. Így ezek az információk nem kereshetők vissza. Egy korszerű lemeznél az információ közvetlenül a lemezről is megismerhető az első eléréskor. A BIOS adatbejegyzését folytatva, mindegyik meghajtóegység számára az *init_drive* függvény hívásával további lemezinformációt tölt be.

Az IDE-vezérlésű régebbi rendszereken, még az AT típusú periféria kártyás esetében is, a lemezfüggvények az alaplapra integráltak. A modern meghajtóegységvezérlők rendszerint a PCI-eszközökhöz hasonlóan működnek, leginkább 32 bites adatúttal a CPU-hoz, nem pedig 16 bites AT-sínnel. Szerencsére az inicializálást követően az interfész a lemezvezérlők mindkét generációjánál azonos módon jelenik meg a programozó számára. A munka elvégzéséhez az *init_params_pci* (12437. sor) hívására kerül sor, ha szükségesek a PCI-eszközök paraméterei. Ennek az eljárásnak a részleteire nem térünk ki, de néhány dolgot meg kell említenünk. Először is a 12361. sorban a *w_instance* változó értékének beállítása az *ata_instance* indítóparaméter alapján történik. Ha az indítóparaméter nincs beállítva, akkor az érték nulla lesz. Ha nullánál nagyobbra van állítva, akkor a 12365. sorban levő vizsgálat eredménye a BIOS lekérdezése és a szabványos IDE-meghajtó inicializálás átugrása. Ebben az esetben csak a PCI-sínen talált meghajtóegységek lesznek bejegyezve.

Továbbá egy PCI-sínen található vezérlő azonosítása *c0d4–c0d7* közötti vezérlőeszközként történik. Ha a *w_instance* nem nulla, akkor nem játszanak szerepet a *c0d0–c0d3* közötti meghajtóegység-azonosítók, kivéve, ha egy PCI-sínvezérlő „kompatibilisként", azonosítja magát. Egy kompatibilis PCI-sínvezérlővel irányított meghajtóegységek azonosítója *c0d0–c0d3* között van. A legtöbb MINIX 3-felhasználó számára ezek a dolgok valószínűleg nem érdekesek. Kevesebb mint négy meghajtóegységgel (beleértve a CD-ROM-meghajtót) rendelkező számítógép valószínűleg úgy jelenik meg a felhasználó számára, mint aminek egy klasszikus konfigurációja van, *c0d0*-tól *c0d3*-mal jelölt meghajtóegységekkel, akár IDE-, akár PCI-vezérlőkhöz csatolva, és akár a klasszikus 40 tűs párhuzamos csatlakozót, akár az újabb soros csatlakozót használják. Ennek az illúziónak a fenntartása azonban bonyolult programozási háttérrel valósul meg.

A közös főciklus meghívása után egy rövid ideig semmi sem történik, amíg a merevlemez elérésére kísérlet nem történik. A merevlemezre vonatkozó első elérési kísérletkor egy üzenet kéri a *DEV_OPEN* műveletet, amit a főciklus kap, és közvetve meghívja a *w_do_open*-t (12521. sor). Ezután a *w_do_open* meghívja a *w_prepare*-t, hogy eldöntse az eszköz vonatkozó kérés jogosságát, utána meghívja a *w_identify*-t, hogy azonosítsa az eszköz típusát, és hogy végrehajtsa a *wini* vektorban néhány további paraméter kezdeti beállítását. Végül a *wini* vektorban egy számláló vizsgálatával eldönti, hogy a MINIX 3 elindítása óta ez-e az első alkalom az eszköz megnyitására. A vizsgálat után a számláló értékét megnöveli. Ha ez az első *DEV_OPEN* művelet, akkor meghívja a *partition* függvényt (*drvlib.c*-ben).

A következő függvény, a *w_prepare* (12577. sor) átvesz egy *device* egész típusú argumentumot, ami a használandó meghajtóegység vagy partíció mellékeszközszáma, és visszaad a *device* adatszerkezetre egy mutatót; ebben van az eszköz kezdőcíme és mérete. A C nyelvben nincs akadálya annak, hogy ugyanazzal a névvel azonosítsunk egy adatszerkezetet és egy változót. A mellékeszközszámból eldönthető, hogy az eszköz egy meghajtóegység, egy partíció vagy egy részpartíció. Ha egyszer a *w_prepare* elvégezte a munkáját, akkor a többi lemezt író vagy olvasó függvény közül egyiknek sem kell foglalkoznia a particionálással. Mint láttuk, egy *DEV_OPEN* kérés teljesítésekor a *w_prepare* meghívására kerül sor; ez azonban egy előkészítés/átvitel ciklusnak is az egyik (első) része, amely ciklusra minden adatátviteli kéréskor szükség van.

A szoftverkompatibilis AT típusú lemezek már meglehetősen hosszú ideje használatosak, így szükséges, hogy a *w_identify* (12603. sor) különbséget tudjon tenni az évek során bevezetett különböző konstrukciók között. Az első lépésben ellenőrzi, hogy van-e egy olvasható és írható I/O-kapu, ha a családba tartozó összes lemezvezérlőnél ez kell. Eddig ez az első olyan példa, ahol felhasználói szintről I/O-kapu elérés van, és ez a tény megérdemli a figyelmet. A lemezeszköz I/O-ok számára a 12201-től 12208-ig lévő sorokban definiált *command* struktúra bájtértékek sorozatával van feltöltve. Később egy kicsit részletesebben is megvizsgáljuk ezt; itt azt jegyezzük meg, hogy a struktúra két bájtja feltöltött, egyik az *ATA_IDENTIFY* értékkel, ami úgy értelmezhető, mint egy parancs, amely **ATA- (AT Attachment – AT-csatolás)** meghajtóegységként kéri az azonosítását, és egy másik, amely kiválasztja a meghajtóegységet. Majd a *com_simple* meghívása történik.

Ez a függvény elrejti a hét I/O-kapu azon vektorának a megkonstruálását, amelybe a címek és a bájtok írása történik, ennek az információnak a rendszertaszkhoz küldését, egy megszakításra várakozást és a visszaadott állapot ellenőrzését. Ez ellenőrzi a meghajtóegység működőképességét, és a 12629. sorban levő sys_insw kernelhívással egy 16 bites karaktersorozat írását engedi meg. Ennek az információnak a visszafejtése egy szövevényes processzus, a részleteire nem térünk ki. Ennek sikeres megtörténte után tekintélyes mennyiségű információ válik elérhetővé, beleértve azt a karaktersorozatot is, ami azonosítja a lemez modelljét és az eszközhöz tartozó fizikai cilinder-, fej- és szektorparamétereket. (Vegyük észre, hogy előfordulhat az, hogy a „fizikai" konfigurációként közöltek nem azonosak az igazi fizikai konfigurációval, de nincs más alternatíva számunkra, mint a lemezmeghajtó egység követeléseinek elfogadása.) A lemezinformáció azt is jelzi, hogy vajon alkalmazható-e rá a **lineáris blokkcímzés (Linear Block Addressing, LBA)**. Ha igen, akkor a meghajtó figyelmen kívül hagyja a cilinder-, fej- és szektorparamétereket, és abszolút szektorszámokat használ a címzésre, ami sokkal egyszerűbb.

Mint ahogy korábban megjegyeztük, lehetséges, hogy az *init_params* nem tudja a BIOS-táblázatokból kideríteni a logikai lemezkonfigurációt. Ha ez történik, akkor a 12666–12674. sorban található kód megpróbál készíteni egy megfelelő paraméterhalmazt az alapján, amit magáról a lemezről olvas. Az ötlet az, hogy a maximális cilinder-, fej- és szektorszámok 1023, 255 és 63 lehetnek, ami az eredeti BIOS-adatszerkezetben ezeknél a mezőknél megengedett bitek számából származtatható.

Ha az *ATA_IDENTIFY* parancs sikertelen, ez egyszerűen jelentheti azt, hogy a lemez egy olyan régi modell, amely nem támogatja a parancsot. Ez esetben az *init_params* által kiolvasott logikai konfiguráció értékeihez juthatunk csak hozzá. Ha érvényesek, akkor a *wini* fizikai paraméter mezőibe bemásolódnak, egyébként egy hiba adódik vissza, és a lemez nem elérhető.

Végül a MINIX 3 a címek bájtokban való számlálásához egy *u32_t* változót használ. A partíciók mérete 4 GB-ra van korlátozva. Viszont a *device* struktúra, ami a partíciók báziscímét és méretét rögzíti (a *drivers/libdriver/driver.h*-ban van definiálva a 10856–10858. sorban), az *u64_t* használja, és egy 64 bites szorzást alkalmaz a meghajtóegység méretének kiszámításához (12688. sor). A teljes meghajtóegység kezdőcíme és mérete ezután a *wini* vektorba kerül, meghívódik a *w_specify*, ha szükséges kétszer, hogy paramétereket adjon vissza a lemezvezérlőnek (12691. sor). Végül több kernelhívás következik: egy sys_irqsetpolicy hívás biztosítja, hogy amikor egy lemezvezérlő-megszakítás előfordul, és a megszakítás kezelése megtörténik, automatikusan újra engedélyezve a megszakításokat. Ezután egy sys-irgenable hívás teszi lehetővé a megszakítást.

A *w_name* (12711. sor) visszaad egy mutatót, az eszköz nevét tartalmazó karaktersorozatra; ez lehet: AT-D0, AT-D1, AT-D2 vagy AT-D3. Mikor egy hibaüzenetet kell megjeleníteni, ez a függvény mondja meg, hogy az melyik eszköz generálta azt.

Lehetséges, hogy egy meghajtóegység kikapcsolásra kerül, mert valamilyen szempontból a MINIX 3-mal nem kompatibilis. A *w_io_test* (12723. sor) gondoskodik minden egyes meghajtócgység teszteléséről az első megnyitási kísérletkor. A rutin a meghajtóegység első blokkját próbálja olvasni rövidebb várakozási értékkel, mint egy rendes műveletnél szokás. Ha a tesztelés sikertelen, akkor a meghajtóegységet állandóan elérhetetlennek jelöli.

A *w_specify* (12775. sor) átadja a paramétereket a vezérlőnek, szintén újra beállítja a meghajtóegységet (ha az egy régebbi modell), végrehajtva egy keresést a nulla sorszámú cilinderre.

A *do_transfer* (12814. sor), mint a nevéből következik, összeállítja a *command* struktúrát az összes olyan bájt értékéből, amik egy nagy mennyiségű adat (255 lemezszektor mennyiség lehet) átvitelének kéréséhez szükségesek, és utána meghívja a *com_out*-ot, amely a lemezvezérlőhöz küldi a parancsot. Az adatnak a lemez címzésétől függően formázottnak kell lennie, vagy cilinder-, fej-, szektorcímzés, vagy LBA címzés szerint. Belsőleg a MINIX 3 a lemez blokkjait lineárisan címzi, így LBA címzésnél az első hárombájtnyi mezőt feltölti a szektorszámláló megfele-

lő számú bittel való jobbra léptetésével, majd maszkolással 8 bites értékeket kap. A szektorszámláló egy 28 bites szám, így az utolsó maszkolás művelet egy 4 bites maszkot használ (12830. sor). Ha a lemez nem támogatja az LBA módszert, akkor a cilinder-, fej- és szektorértékeket kiszámolja az alkalmazott lemez paraméterei alapján (12833–12835. sor).

A kód egy utalást tartalmaz egy jövőbeni fejlesztési irányra vonatkozóan. A 28 bites szektorszámlálóval dolgozó LBA címzés a MINIX 3-at 128 GB-os vagy kisebb méretű lemezek teljes kihasználására korlátozza. (Nagyobb lemez is használható, de a MINIX 3 csak az első 128 GB-ot éri el.) A programozók gondolkoztak egy újabb **LBA48** módszeren, de még nem implementálták, amely 48 bitet használ a lemez blokkjainak címzésére. A 12824. sorban levő teszt megvizsgálja, hogy van-e lehetőség erre. A teszt a MINIX 3-nál mindig sikertelen lesz a leírtak miatt. Ez jó, mivel nem kell kódot biztosítani a sikeres teszt esetére. Nem szabad elfelejteni, hogy amennyiben a MINIX 3-at módosítanánk, hogy alkalmazza az LBA48-at, akkor annál többet kell tennünk, mint hogy bizonyos kóddal bővítünk. Sok helyen változtatásokat kellene tenni, hogy a 48 bites címek kezelhetők legyenek. Valószínű könnyebbnek tűnik megvárni, hogy a MINIX 3 egy 64 bites processzorra is átvihető legyen. Ha a 128 GB-os lemez nem elég nagy valakinek, az LBA48 módszer 128 PB (petabájt) elérést nyújt.

Most röviden megnézzük, hogyan történik egy magasabb szinten az adatátvitel. A korábban már vizsgált *w_prepare* van meghíva elsőként. Ha az átviteli kérés többszörös blokkra vonatkozik (vagyis egy *DEV_GATHER* vagy *DEV_SCATTER* kérés), akkor közvetlenül ezután a 12848. sorban levő *w_transfer* hívása történik. Ha az átvitel egyszerű blokkra vonatkozik (egy *DEV_READ* vagy egy *DEV_WRITE* kérés), akkor egy leoszt/összegyűjt vektor kialakítására kerül sor, majd ezután meghívódik a *w_transfer*. A *w_transfer*-t úgy írták, hogy egy *iovec_t* vektorba várja a kéréseket. A kérésvektor mindegyik eleme tartalmaz egy puffercímet és a puffer méretét, ami a lemez szektorméretének többszöröse lehet csak. Az összes többi szükséges információ a hívás argumentumaként van átadva, és ez a teljes kérésvektorra is vonatkozik.

Először egy egyszerű tesztelés következik; ez megnézi, hogy az átvitel kezdetét megadó lemezcím a szektorkorlátnak megfelel-e (12863. sor). Ezután a függvény külső ciklusa indul. A kérésvektor minden elemére ismétlődik ez a ciklus. Mint ahogyan korábban már többször láttuk, a ciklusban, mielőtt a függvény a valódi munkát elvégezné, bizonyos számú teszt van. Először a kért bájtok számának kiszámolása történik a kérésvektor elemeinek *iov_size* mezőinek összeadásával. Ellenőrzésre kerül, hogy ez a szám egy szektorméretnek pontosan a többszöröse-e. Egy másik teszt azt ellenőrzi, hogy a kezdési cím az eszköz végénél vagy azon túl van-e, és ha a kérés túlnyúlna az eszköz végén, akkor a kérés méretét megcsonkítja. Az eddigi számítások bájtokban voltak, míg a 12876. sorban levő számolás 64 bites aritmetikát használ a lemezen a blokk helyének kiszámítására. Vegyük észre, hogy bár az alkalmazott változó neve *block*, ez a lemez blokkjainak egy száma, azaz 512 bájtos szektoroknak, és nem a 4096 bájtos MINIX 3 által belsőleg használt „block"-nak. Ezután még egy beállítás van. Minden meghajtóegységnél definiált az egyszerre kérhető bájtok maximális száma, és ha szükséges, akkor ennek alapján a kérés csökkentésére kerül sor. Miután ellenőrzi, hogy a lemez inicializálva van-e, és ha szükséges volt a kérés csonkítása, akkor egy csonka adatra ad egy kérést a *do_transfer* (12887. sor) meghívásával.

Egy átviteli kérés elkészülte után a belső ciklus indul, amely mindegyik szektorra ismétlődik. Egy olvasás vagy írás műveletnél valamennyi szektor számára egy megszakítás generálódik. Az olvasáskor a megszakítás jelzi, hogy az adat készen van, és átmozgatható. A 12913. sorban levő *sys_insw* kernelhívás kéri a rendszertaszkot, hogy olvassa ismételve a jelzett I/O-kaput, mozgassa át az adatot a jelzett processzus adathelyében egy virtuális címre. Az írás műveletnél a sorrend fordított. A *sys_outsw* meghívás néhány sorral lentebb, a vezérlőre ír egy adatsorozatot, és a lemezvezérlőtől jön a megszakítás, amikor a lemezre az átvitel befejeződik. Akár olvasás, akár írás történik, az *at_intr_wait* meghívással kapható meg a megszakítás, például a 12920. sorban az írási műveletet követően. Bár a megszakítás elvárt, ez a függvény a várakozás megszüntetésére is módot ad, ha egy hibás működés lép fel és megszakítás nem jönne. Az *at_intr_wait* a lemezvezérlő állapotregiszterét olvassa és visszaad különféle kódokat. A 12933. sorban van ennek a tesztelése. Írásnál vagy olvasásnál bekövetkező hiba esetén van egy break, ami átugorja azt a szekciót, ahova az eredmény rögzítve van, és a mutatókat és számlálókat a következő szektor számára állítja be, így a belső ciklusban a következő alkalommal ismét lesz próbálkozás ugyanarra a szektorra, ha megengedett további próba. Ha a lemezvezérlő egy hibás szektorról ad jelzést, akkor a *w_transfer* azonnal befejeződik. Más hibák esetén egy számláló értéke nő, és addig lehet a függvényt folytatni, amíg a *max_errors* értékét nem éri el.

A következő függvény, amelyet megvizsgálunk a *com_out;* ez a lemezvezérlőnek küldi a parancsot, de mielőtt a kódját megnéznénk, először nézzük a vezérlőt szoftverszempontból. A lemezvezérlő irányítása egy regiszterhalmazon keresztül történik, amely regiszterek néhány rendszernél memórialeképezésűek, de az IBM-kompatibilis rendszereken I/O-kapukként jelennek meg. Meg fogjuk nézni ezeket a regisztereket (és általánosan az I/O-vezérlőregisztereket), és megvizsgáljuk a használatuk néhány vonatkozását. A MINIX 3-ban további bonyodalom, hogy a vezérlők felhasználói szinten futnak, és nem tudnak regisztert olvasó és író parancsot végrehajtani. Alkalom nyílik a kernelhívások ilyen megszorítások melletti használatának megnézésére is.

Egy szabványos IBM-AT típusú merevlemez-vezérlő regisztereit mutatja a 3.23. ábra.

Már számos alkalommal foglalkoztunk az I/O-kapukhoz kapcsolódó olvasással és írással, hallgatólagosan úgy kezelve ezeket, mint memóriacímeket. Valójában az I/O-kapuk eltérően viselkedhetnek a memóriacímektől. Általában azok a beviteli és kiviteli regiszterek, amelyek történetesen ugyanazzal az I/O-kapu címmel rendelkeznek, nem azonos regiszterek. Így beírva egy adatot egy konkrét címre, a rá következő olvasási művelet nem feltétlenül tudja azt visszakeresni. Például a 3.23. ábrán az utolsó regiszter olvasásakor a lemezvezérlő állapotát mutatja, míg beírni ide a vezérlőnek szánt parancsot lehet. Szintén általános, hogy egy I/O-eszköz regiszterének olvasása vagy írása előidézi egy tevékenység előfordulását, függetlenül az átmozgatott adat részleteitől. Ez az AT-lemezvezérlő parancsregiszterére

Regiszter	Olvasási tevékenység	Írási tevékenység
0	Adat	Adat
1	Hiba	Kompenzációírás
2	Szektorszámláló	Szektorszámláló
3	Szektorszám (0–7)	Szektorszám (0–7)
4	Cilinder alacsony bitek (8–15)	Cilinder alacsony bitek (8–15)
5	Cilinder magas bitek (16–23)	Cilinder magas bitek (16–23)
6	Meghajtóegység/fej kiválasztása (24–27)	Meghajtóegység/fej kiválasztása (24–27)
7	Állapot	Parancs

(a)

7	6	5	4	3	2	1	0
1	LBA	1	D	HS3	HS2	HS1	HS0

LBA: 0 = Cilinder/fej/szektor mód
1 = Logikai blokkcímzési mód
D: 0 = Fő meghajtóegység
1 = Alárendelt meghajtóegység
HSn: CHS mód: Fejkiválasztás cilinder/fej/szektor módban
LBA mód: Blokk-kiválasztó bitek 24–27

(b)

3.23. ábra. *(a) Egy IDE merevlemez-vezérlő vezérlőregiszterei. A zárójelben levő számok a logikai blokkcím bitjei minden regiszterre LBA módban. (b) A meghajtóegység/fej kiválasztás regiszterének mezői*

is érvényes. Szokásosan az alacsonyabb számozású regiszterekbe olyan adat kerül, amely alapján kiválasztódik egy lemezcím, ahonnan olvas, illetve ahová ír, és utoljára a parancsregiszterbe a műveletkód helyeződik el. A parancsregiszterbe írt adat határozza meg, hogy mi lesz a művelet. A művelet kódjának a parancsregiszterbe beírása eredményezi a művelet elkezdését.

Előfordul az is, hogy néhány regiszternek vagy a regiszterek mezőinek a használata nagyban eltérhet a műveletek különböző módjainál. Az ábrán szereplő példánál maradva, attól függően, hogy a 6-os regiszter 6. bitjébe (az LBA bitbe) 0 vagy 1 kerül beírásra, vagy a CHS (Cilinder-Head-Sector, cilinder-fej-szektor), vagy az LBA mód kerül kiválasztásra. A 3., 4., 5. regiszterbe és a 6. regiszter alsó 4. bitjébe írt vagy onnan olvasott adatok értelmezése az LBA bit beállítása szerint különböző.

Most nézzük, hogyan történik egy parancsnak a vezérlőhöz küldése a *com_out* (12947. sor) meghívásával. A függvény meghívására a *cmd* struktúra beállítása után kerül sor (a *do_transfer* által, mint már korábban láttuk). Mielőtt bármelyik regiszter megváltozna, az állapotregiszter olvasásával eldönthető, hogy a vezérlő foglalt-e. Ez a *STATUS_BSY* bit tesztelésével történik. Itt fontos a gyorsaság, de normális esetben a lemezvezérlő készen áll, vagy kész lesz rövid időn belül, ezért tevékeny várakozást használunk. A *STATUS_BSY* ellenőrzésére a 12960. sorban

a *w_waitfor* hívódik meg. A *w_waitfor* egy kernelhívást alkalmaz, amellyel kéri a rendszertaszkot, hogy olvasson egy I/O-kaput, így a *w_waitfor* az állapotregiszter egy bitjét tudja tesztelni. Ez ciklikusan ismétlődik, míg a bit elkészül vagy időtúllépés áll elő. A ciklusból a visszatérés gyors, ha a lemez készenléti állapotban van. Így ha a vezérlő készenléti állapotban van, akkor minimális késéssel visszaad egy igaz értéket, de igaz érték lehet egy késés után is, ha azonnal nem elérhető a vezérlő, illetve hamis érték keletkezik, ha a vezérlő nem kerül készenléti állapotba a megszabott időn belül. Az időtúllépésről bővebben a *w_waitfor* tárgyalásánál szólunk.

Egy vezérlő egynél több meghajtóegységet is működtethet, így valahányszor a vezérlő készenléti állapotba kerül, egy bájt írása történik, amely alapján kiválasztódik a meghajtóegység, az olvasófej és a műveleti kód (12966. sor), és ezután ismét a *waitfor* meghívása következik. Egy lemezmeghajtó egység néha sikertelenül hajtja végre a parancsot, vagy helytelenül adja át a hibakódot – végül is egy mechanikai eszközről van szó, amely megakadhat, beszorulhat vagy eltörhet benne valami –, így a biztonság kedvéért egy sys_setalarm kernelhívást ad ki a rendszertaszknak, hogy ütemezzen be egy ébresztő eljárást. Ezt követően az a parancs lesz kiadva, amelynek hatására először az összes paraméter beíródik a különböző regiszterekbe, és végül bekerül a parancsregiszterbe a parancs kódja. Ez a sys_voutb kernelhívással történik, amely egy *(value, address)* párokból álló vektort küld a rendszertaszkhoz. A rendszertaszk sorban mindegyik értéket *(value)* az *address* által definiált I/O-kapura írja. A sys_voutb híváshoz az adatvektor a *pv_set* makró alkalmazásával van konstruálva, amely az *include/minix/devio.h*-ban van definiálva. A műveleti kódnak a parancsregiszterbe való írása indítja el a műveletet. Mikor ez befejeződik, egy megszakítás generálódik, és egy nyugtázó üzenet is elküldésre kerül. Ha a parancs túllépi az időt, akkor a jelző lejár, és egy egyidejű figyelmeztető értesítés a lemezvezérlőt felébreszti.

A következő függvények viszonylag rövidek. A *w_need_reset* (12999. sor) meghívására kerül sor, amikor időtúllépés következik be, mialatt várakozik a lemezmegszakításra vagy készenléti állapot kialakulására. A *w_need_reset*-nek csupán az a feladata, hogy figyelemmel kísérje mindegyik meghajtóegység *state* változóját a *wini* vektorban, és kierőszakoljon egy kezdeti beállítást a következő hozzáféréshez.

A *w_do_close*-nak (13016. sor) kevés dolga van a hagyományos merevlemezekkel. A CD-ROM támogatásához bővítő kód szükséges.

A *com_simple* segítségével olyan vezérlőparancsok adhatók ki, amelyek bármilyen adatátvitel nélkül befejeződnek. Ebbe a kategóriába tartoznak azok a parancsok, amelyek visszakeresik egy lemez azonosítóját, beállítanak bizonyos paramétereket, a lemezt ismételten beállítják. Ennek a használatára a *w_identify*-ban már láttunk egy példát. A hívása előtt a *command* struktúrának már hibátlanul inicializálva kell lennie. Vegyük észre, hogy közvetlenül a *com_out* meghívása után egy *at_intr_wait* hívás történik. Ez lényegében egy receive-et hajt végre, ami blokkol, amíg egy értesítés érkezik, jelezve egy megszakítás bekövetkeztét.

Már láttuk, hogy a *com_out* egy sys_setalarm kernelhívást kezdeményez, mielőtt a rendszertaszkot kérné, hogy a regisztereket írja, ami beállít és végrehajt egy parancsot. Ahogyan az áttekintést adó fejezetben írtuk, a következő receive művelet rendszerint egy megszakítási értesítést kaphat. Ha egy figyelmeztető jelzés be-

következik, és megszakítás nem fordul elő, akkor a következő üzenet egy *SYN_ALARM* lesz. Ez esetben a *w_timeout* hívása történik a 13046. sorban. Az, hogy mit kell tenni, a *w_command*-ban levő parancstól függ. Az időtúllépés egy előző műveletből valószínűleg el van halasztva, és a *w_command*-ba belekerül a *CMD_IDLE,* ami azt jelenti, hogy a lemez a műveletét végrehajtotta. Ebben az esetben semmi nem történik. Ha a parancs nincs teljesítve, és a művelet egy olvasás vagy egy írás, akkor lehet, hogy az I/O-kérések méretének csökkentése segíthet. Ez két lépésben történik, először a maximálisan kérhető szektorok száma 8-ra, majd ezt követően 1-re csökken. Minden időtúllépésnél kinyomtatódik egy üzenet, a *w_need_reset* meghívásával az összes meghajtóegység a következő eléréseknek megfelelően újból beállításra kerül.

Amikor alapállapotba hozásra van szükség, a *w_reset* hívódik meg (13076. sor). Ez a függvény egy *tickdelay* függvényt használ, amely beállít egy jelzőórát, és azután ennek lejártára vár. Egy kezdeti késlekedés után, ami ahhoz kell, hogy a meghajtóegység az előző műveletből magához térjen, a lemezvezérlő vezérlőregiszterében egy bit be van billentve egy időre **(kapuzójel)** – azaz, egy adott időintervallumig az 1 szintre van állítva, utána visszatér a 0 logikai szintre. A művelet után a *waitfor* meghívására kerül sor, ami a meghajtóegységnek egy ésszerű időintervallumot ad a készenléti állapot jelzésére. Az újraindítás sikertelensége esetén kinyomtatódik egy üzenet, és egy hibaállapottal visszatér.

Az adatátvitelt magukba foglaló lemezparancsok normálisan egy megszakítás előállításával fejeződnek be, ami egy üzenetet küld vissza a lemezmeghajtónak. Valójában minden egyes szektor írásakor vagy olvasásakor egy megszakítás jön létre. A *w_inrt_wait* függvény (13123. sor) egy ciklusban a *receive*-t hívja, és ha egy *SYS_ALARM* üzenet érkezik, akkor a *w_timeout* meghívására kerül sor. Az egyetlen más üzenetfajta, amelyet a függvény láthat, az a *HARD_INT*. Mikor ezt kapja, akkor kiolvassa az állapotregisztert, és meghívja az *act_args*-t újrainicializálni a megszakítást.

A *w_intr_wait* meghívása nem közvetlen; mikor egy megszakítás érkezését várjuk, akkor az *at_intr_wait* (13152. sor) függvény meghívása történik. Miután az *at_intr_wait*-hez megszakítás érkezik, gyors ellenőrzés történik a meghajtóegység állapotbitjeire. Minden rendben van, ha a tevékenységnek megfelelő bitek hibát írnak, és a hiba teljesen világos. Máskülönben közelebbről kell megnézni. Ha a regiszter egyáltalán nem olvasható, akkor pánik lép fel. Ha a probléma egy hibás szektor, akkor egy speciális hibakód adódik vissza, a többi probléma egy általános hibakódot eredményez. Minden esetben a *STATUS_ADMBSY* bit van beállítva és később visszaállítva a hívó által.

Már számos helyen láttuk a *w_waitfor* (13177. sor) meghívását, ami a lemezvezérlő állapotregiszterében egy bitet tevékeny várakozással figyel. Ez olyan helyzetben van alkalmazva, ahol elvárt, hogy a bit valószínűleg az első tesztnél hibátlan, és egy gyors teszt a kívánatos. A gyorsaság miatt a MINIX korábbi változatainál egy makrót használtak, amely közvetlenül olvasta az I/O-kaput – természetesen ez nem engedhető egy felhasználói szintű meghajtónál a MINIX 3-ban. Itt a megoldás egy do...while ciklus egy minimum költséggel az első teszt elvégzése előtt. Ha a tesztelt bit hibátlan, akkor azonnal visszatér a ciklusba. Az időtúllépés miatti hibalehetőség kezelése a ciklusban van implementálva, az óraütések figyelésével. Ha időtúllépés fordul elő, akkor kerül meghívásra a *w_need_reset*.

A *w_waitfor* függvény által használt *timeout* paramétert a *DEF_TIMEOUT_TICKS* definiál a 12228. sorban 300 ütésre vagy 5 másodpercre. Egy hasonló paraméter, a *WAKEUP* (12216. sor) az időzítőtaszkból az ütemező ébresztését végzi, és 31 másodpercre van beállítva. Ha azt nézzük, hogy egy szokványos processzus csak 100 ms időt kap futásra, akkor a foglaltsági várakozásra pazarolható időperiódus nagyon hosszúnak tűnik. Azonban ezek a számok az AT osztályú számítógépekhez illeszkedő lemezes eszközöknél előírt szabványnak tesznek eleget, amely szerint 31 másodpercet kell adni egy lemeznek, hogy a megfelelő sebességre felgyorsuljon. Az igazság természetesen az, hogy ez a legrosszabb eset, és a legtöbb rendszernél a felpörgetés csak bekapcsoláskor fordul elő, vagy esetleg egy hosszú tétlen periódus után, legalábbis a merevlemeznél. A CD-ROM-oknál vagy más olyan eszközöknél, amelyeknek gyakran fel kell pörögniük, ez valószínűleg lényegesebb kérdés.

Néhány további függvény található az *at_wini.c*-ben. A *w_geometry* a kiválasztott merevlemezeszköz logikailag maximális cilinder-, fej- és szektorszámát adja vissza. Ebben az esetben valós számokról van szó, nem úgy, mint a RAM-lemezmeghajtónál volt. A *w_other* ismeretlen parancsokat és ioctl-eket kap el. A MINIX 3 jelenlegi változata valójában nem alkalmazza. A *w_hw_int* meghívására akkor kerül sor, amikor váratlanul egy hardvermegszakítás érkezik. Az áttekintő részben már láttuk, hogy ez akkor történik, amikor egy időtúllépés a várt megszakítás előfordulása előtt lejár. Ez megválaszol egy receive műveletet, amely blokkolva várakozik a megszakításra, viszont a megszakítás nyugtáját egy rá következő receive találhatja meg. Az egyetlen, amit tesz, hogy újra engedélyezi a megszakítást az *ack_irqs* függvény meghívásával (13297. sor). Ez keresztülmegy az összes ismert meghajtóegységen, és a sys_irqenable kernelhívás alkalmazásával biztosítja az összes megszakítás lehetőségét. Végül az *at_wini.c* végénél két furcsa, kicsi függvény található, az *strstatus* és az *strerr*. Ezek makrókat használnak (a definíció megtalálható a 13313. és a 13314. sorban) a hibakódok sztringbe való összefűzéséhez. Ezek a függvények, mint írtuk, a MINIX 3-ban nincsenek használva.

3.7.6. Hajlékonylemezek kezelése

A hajlékonylemez-meghajtó hosszadalmasabb és bonyolultabb a merevlemezmeghajtónál. Ez ellentmondásnak tűnhet, mivel a hajlékonylemez szerkezete egyszerűbbnek látszik a merevlemezénél, azonban az egyszerűbb szerkezet egyszerűbb vezérlővel párosul, és így több odafigyelést igényel az operációs rendszertől. További bonyodalmak forrása az a tény, hogy ezek az adathordozó eszközök cserélhetők. Ebben az alfejezetben néhány olyan dolgot tárgyalunk, amelyeket az implementálónak figyelembe kell vennie a hajlékonylemezekkel kapcsolatban, de a MINIX 3 hajlékonylemez-meghajtó kódjának részleteibe nem megyünk bele. Legfontosabb részei hasonlók a merevlemez megfelelő részeihez.

A hajlékonylemez-meghajtóval kapcsolatban egy dolog miatt nincs ok aggodalomra, ez a különféle típusú vezérlők támogatása, amivel foglalkozni kellett a me-

revlemez-meghajtó esetében. Bár az eredeti IBM PC-konstrukció nem támogatta a napjainkban használatos nagy jelsűrűségű hajlékonylemezeket, az IBM PC család minden számítógépének hajlékonylemez-vezérlője egyetlen szoftvermeghajtóval működik. A merevlemezzel ellentétes helyzet valószínűleg abból adódik, hogy a hajlékonylemeznél vélhetően hiányzik a motiváció a teljesítmény növelésére. A hajlékonylemezeket igen ritkán használják a számítógéprendszer működése során munkatárolóként; sebességük és tárolókapacitásuk is meglehetősen korlátozott a merevlemezekéhez képest. A hajlékonylemezek valamikor fontosak voltak az új szoftverek terjesztésében és a biztonsági másolatok tárolásánál, de mióta a hálózatok és a nagy kapacitású cserélhető tárolók váltak általánossá, a PC-khez újabb hajlékonylemez-meghajtó egység szabványokat már nem terveznek.

A hajlékonylemez-meghajtók sem az SSF stratégiát, sem a liftes algoritmust nem alkalmazzák. Szigorúan egymás után fogadják a kéréseket, és végrehajtják azokat, mielőtt egy másik kérést elfogadnának. Ez az elv jelent meg az eredeti MINIX-tervben, mivel a MINIX-et személyi számítógépeken való használatra tervezték, ahol az idő túlnyomó részében csak egy processzus aktív. Így annak valószínűsége, hogy egy lemezcserére kérés érkezzék, mialatt egy másik végrehajtása tart, nagyon kicsi. Éppen ezért kevés haszon származna egy olyan, meglehetősen bonyolult szoftverből, amire szükség lenne a kérések sorba állításánál. De azért sem lenne érdemes bonyolítani, mert manapság a hajlékonylemezeket ritkán használják másra, mint egy merevlemezes rendszerből vagy abba történő adatmozgatásra.

Az elmondható, hogy más blokkos eszközökhöz hasonlóan, a hajlékonylemez-meghajtó is tud osztott I/O-kérést kezelni. Azonban a hajlékonylemez-meghajtó esetében a kérések vektora kisebb, mint a merevlemeznél, mégpedig a felső korlátot a lemez pályavonalán elhelyezkedő szektorok maximális száma jelenti.

A hajlékonylemez hardverének egyszerűsége a felelős a néhány esetben bonyolult hajlékonylemez-meghajtó szoftverért. Az olcsó, lassú, kis kapacitású hajlékonylemez-meghajtó egységek nem rendelkeznek olyan kifinomult integrált vezérlőkkel, amelyek részét képezik a modern merevlemez-meghajtó egységeknek, így a lemezműveletek azon aspektusaival, amelyek egy merevlemez-meghajtó egység esetében a műveletbe rejtve vannak, a hajlékonylemez-meghajtó szoftvernek expliciten kell foglalkoznia. A hajlékonylemez-meghajtó egység egyszerűségéből adódó komplikációra példa a *SEEK* művelet, amely az olvasófejet adott pályavonalra mozgatja. Egyik merevlemez sem követeli meg a meghajtószoftvertől, hogy expliciten meghívjon egy *SEEK* műveletet. Lehet hogy a merevlemeznél a programozó számára látható cilinder-, fej-, szektorgeometria nem egyezik meg a lemez fizikai geometriájával. Valójában ez utóbbi bonyolultabb is lehet. Szokásosan többszörös zónák (cilinderek csoportja) vannak, több szektorral sávonként a kintebbi zónákon, mint a bentebbieken. Azonban ez nem látható a felhasználó számára. A merevlemezek elfogadják a lemez abszolút szektorszámmal történő címzését, vagyis a logikai blokkos címzést (LBA), mint a cilinder-, fej- és szektorcímzés alternatíváját. A cilinder, fej, szektor hármassal történő címzésnél bármely olyan geometria használható, amely nem létező szektorokat nem címez, mivel a lemez integrált vezérlője kiszámítja az olvasófej elmozdulásának helyét, és a pályavonalon belüli szektort is megkeresi, amikor szükséges.

Hajlékonylemeznél a *SEEK* műveletet expliciten programozni kell. Sikertelen *SEEK* esetén egy *RECALIBRATE* műveletet végrehajtó rutinra van szükség, ami kikényszeríti a fej elmozdulását a 0. cilinderre. Ezzel a vezérlőnek lehetősége nyílik arra, hogy a tervezett pályavonalra előremozdítsa a fejeket, azokat az ismert számszor léptetve. Természetesen a merevlemeznél is hasonló műveletre van szükség, de a meghajtóegység vezérlője anélkül vezérli a fejeket, hogy az eszközmeghajtó szoftvertől részletes útmutatót kapna.

A hajlékonylemez-meghajtó egység néhány olyan sajátossága, ami bonyolítja a meghajtót:

1. A hajlékonylemez cserélhető eszköz.
2. Többféle lemezformátum használatos.
3. Motorvezérlés.

Néhány merevlemez-vezérlő lehetőséget ad cserélhető eszközök használatára, ilyen például egy CD-ROM-meghajtó egység, de általában a meghajtóvezérlő bármilyen komplikált esetet képes kezelni anélkül, hogy támogatás lenne az eszközmeghajtó szoftverben. Egy hajlékonylemeznél azonban nincs beépített támogatás, pedig egyre inkább kellene. A hajlékonylemezek leggyakoribb alkalmazásai közül bizonyosaknál – új szoftver telepítése vagy állományok biztonsági tárolása – valószínűleg szükséges a lemezek cseréje a meghajtóegységben. Hiba léphet fel, ha az egyik hajlékonylemezre szánt adat egy másik lemezre kerül rá. Az eszközmeghajtónak kellene valamit tennie ennek megelőzésére. Bár ez nem mindig lehetséges, mivel nem minden hajlékonylemez-meghajtó egység hardver teszi lehetővé annak meghatározását, hogy vajon a legutolsó hozzáférés óta az egység ajtaja nyitva van-e. Egy másik probléma a cserélhető eszközökkel kapcsolatban, hogy egy rendszer felfüggesztett állapotba kerülhet, amikor kísérletet tesz egy hajlékonylemez-meghajtó egység elérésére, amelyben nincs lemez. Egy nyitott ajtó érzékelésével ez a probléma megoldható lenne, de mivel erre nincs mindig lehetőség, így néhány intézkedést kell tenni az időtúllépéssel és a hibavisszaadással kapcsolatban, ha a hajlékonylemezes művelet nem fejeződik be ésszerű időn belül.

Cserélhető eszköz helyettesíthető másik eszközzel, és a hajlékonylemezeknél sok különböző lehetséges formátum jöhet szóba. Az IBM-kompatibilis hardver mind a 3,5 hüvelykes, mind az 5,25 hüvelykes lemezmeghajtó egységeket támogatja, ezek pedig változatos módon formázhatók 360 KB-tól 1,2 MB-ig (5,25 hüvelykes lemeznél) vagy 1,44 MB-ig (3,5 hüvelykes lemeznél).

A MINIX 3 hét különböző hajlékonylemez-formátum használatát támogatja. Az ebből adódó probléma megoldására két lehetőség is van. Az egyik módszer szerint minden lehetséges formátumra mint különböző meghajtóegységekre hivatkozik, és gondoskodik több alárendelt eszközről. A MINIX régebbi változatai ezt a módszert alkalmazták. Tizennégy különböző eszközt definiáltak, felsorakoztatva ezeket az első meghajtóegységbeli 360 KB-os 5,25 hüvelykes lemeznek megfelelő */dev/pc0*-tól a */dev/PS1*-ig, ami az 1,44 MB-os 3,5 hüvelykes második meghajtóegységbeli lemezhez tartozik. Ez kényelmetlen megoldás volt. A MINIX 3 egy másik módszert alkalmaz: amikor az első hajlékonylemez-meghajtó egység

/dev/fd0-ként vagy a második */dev/fd1*-ként van címezve, akkor a lemezmeghajtó a meghajtóegységet elérve megvizsgálja az abban pillanatnyilag található lemez formátumát. Néhány formátum több cilindert vagy több szektort tartalmaz pályavonalanként, mint más formátumok. Egy hajlékonylemez formátumának meghatározása a magasabb számozású szektorok és pályavonalak olvasása útján történik. Egy kizárásos módszerrel a formátum meghatározható. Ehhez idő kell, de a modern számítógépeken valószínűleg csak 1,44 MB-os 3,5 hüvelykes lemezek vannak, és elsőként ez a formátum van ellenőrizve. Másik lehetséges probléma, ha egy lemez hibás szektorokat tartalmaz, és esetleg nem ismerhető fel. Egy segédprogramot alkalmaznak a lemez azonosításához; az operációs rendszerben ezt túlságosan lassú lenne automatikusan végrehajtani.

A hajlékonylemez-meghajtóval kapcsolatos, utoljára maradt nehézség a motor vezérlése. A lemezek csak akkor olvashatók vagy írhatók, ha forgásban vannak. A merevlemezeket úgy tervezik, hogy több ezer órát futhatnak elhasználódás nélkül, de a motor állandó működése miatt a hajlékonylemez-meghajtó egység és a lemezek gyorsan kopnának. Ha a motor még nincs bekapcsolva a meghajtóegység elérésekor, akkor egy parancs kiadása szükséges, amely az egységet elindítja, és utána fél másodpercet vár, mielőtt az adat olvasását vagy írását megkísérelné. A motor kikapcsolása és bekapcsolása lassú, ezért a MINIX 3 egy meghajtóegység használata után még néhány másodpercig az egység motorját bekapcsolva hagyja. Ha ezen intervallumon belül ismét felhasználásra kerül a meghajtóegység, akkor az idő további néhány másodperccel meghosszabbodik. Ha a meghajtóegységet nem használják az adott intervallumon belül, akkor a motor kikapcsolódik.

3.8. A terminálok

Évtizedeken keresztül a felhasználók olyan készülékekkel kommunikáltak a számítógépekkel, amelyek a felhasználó inputját közvetítő billentyűzetből és a számítógép válaszait megjelenítő egységből álltak. Hosszú évekig ezeket az önálló készülékeket **terminálok**nak nevezték, és vezetékkel kapcsolódtak a számítógéphez. A pénzügyi szektorban és az utazási irodákban a nagyszámítógépekhez még mindig gyakran használják ezeket a terminálokat, amelyek rendszerint modemen keresztül kapcsolódnak a nagyszámítógéphez, különösen ha távol helyezkednek el tőle. A személyi számítógépek megjelenésével azonban a billentyűzet és a megjelenítő egység inkább különálló periféria lett, mint egyetlen készülék, azonban annyira szorosan összefüggenek egymással, hogy együtt tekintjük át őket a továbbiakban a „terminál" elnevezés alatt.

Történetileg a terminálóknak sok formája alakult ki. A terminálmeghajtó program feladata, hogy ezeket a különbségeket elrejtse, így az operációs rendszer és a felhasználói programok eszközfüggetlen részét nem szükséges átírni minden egyes termináltípushoz. A következő alfejezetekben a már megszokott módon közelítjük meg témánkat, először a terminálok hardveréről és szoftveréről lesz szó általánosságban, majd pedig a MINIX 3-szoftverről.

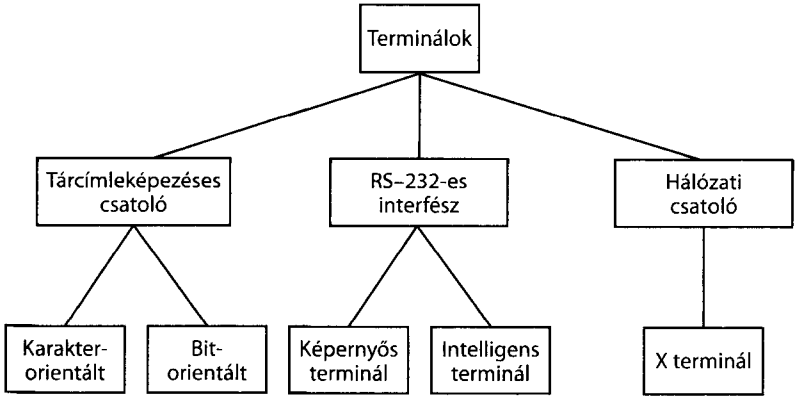
3.8.1. A terminálhardver

Az operációs rendszer szempontjából a terminálokat három nagy csoportba sorolhatjuk aszerint, ahogyan az operációs rendszer információt cserél velük, illetve alapvető elektronikai tulajdonságaik alapján. Az első csoport a tárcímleképezéses terminálok csoportja, amelyek egy billentyűzetből és egy képernyős megjelenítőből állnak, mindkettő elektronikailag hozzá van csatlakoztatva a számítógéphez. Ezt a típust használják az összes személyi számítógépben a billentyűzethez és a monitorhoz. A második csoportba tartoznak azok, amelyek egy RS–232 szabványt használó soros vonalon keresztül csatlakoznak, leggyakrabban egy modemen keresztül. Ezt a modellt használják néhány nagyszámítógépen, de a PC-knek ugyancsak van soros vonali csatolójuk. A harmadik csoport olyan terminálokból áll, amelyek hálózaton keresztül kapcsolódnak a számítógéphez. Ez a csoportosítás látható a 3.24. ábrán.

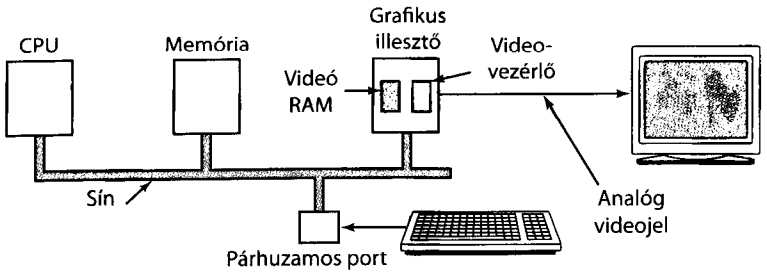
Tárcímleképezéses terminálok

A terminálok első nagy csoportját, ahogyan a 3.24. ábra is mutatja, a tárcímleképezéses terminálok alkotják. Ezek integrált részei magának a számítógépnek, különösen a személyi számítógépeknek. Egy képernyős megjelenítőből és egy billentyűzetből állnak. A tárcímleképezéses megjelenítők egy speciális memórián, az ún. **videó RAM**-on keresztül csatlakoznak a számítógéphez, ez a tárterület része a számítógépben található memória címtartományának, és a CPU ugyanúgy címzi, mint a memória többi részét (lásd 3.25. ábra).

A videó RAM kártyán található egy olyan lapka (chip) is, amelyet **videovezérlő**nek neveznek. Ez a lapka veszi elő a karakterkódokat a videó RAM-ból, és állítja elő a képernyős megjelenítőt vezérlő videojelet. A képernyős megjelenítőknek két fajtája van: a katódsugárcsöves monitorok és a lapos megjelenítők. A **katódsugárcsöves**



3.24. ábra. *Terminháltípusok*



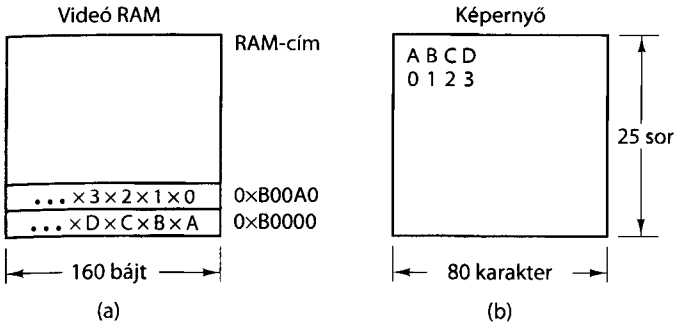
3.25. ábra. *A tárcímleképezéses terminálok közvetlenül a videó RAM-ba írnak*

monitor (Cathode Ray Tube, CRT) elektron-sugárnyalábot generál, amely vízszintesen pásztázza a képernyőt, és sorokat rajzol ki rá. A képernyő sorainak száma tipikusan 480 és 1200 között változik a képernyő tetejétől az aljáig, míg egy sor 640–1920 képpontból áll. Ezeket a képpontokat nevezik **pixel**nek. A videovezérlőjel modulálja az elektron-sugárnyaláb intenzitását, meghatározva, hogy egy adott pixel világos vagy sötét legyen. A színes monitorokban három sugárnyaláb van: a piros, a zöld és a kék képpontokhoz, amelyeket egymástól függetlenül modulálnak.

A lapos megjelenítők teljesen másként működnek, de egy CRT-kompatibilis lapos megjelenítő ugyanolyan szinkron- és videojeleket fogad, mint egy CRT monitor, és ezeket arra használja, hogy egy folyadékkristály cellát vezéreljen velük minden pixelpozícióban.

Egy egyszerű monokróm képernyő esetén egy karakter (beleértve a karakterek közötti helyközöket is) egy 9 pixel széles és 14 pixel magas dobozba fér el, és a képernyő 25 sort, soronként 80 karaktert jelenít meg. A képernyőnek így 350 egyenként 720 pixelt tartalmazó rasztersora van. Minden egyes képet másodpercenként 45–70 alkalommal frissítenek. A videovezérlő megtervezhető úgy, hogy kiolvassa az első 80 karaktert a videó RAM-ból, előállít 14 rasztersort, majd kiolvassa a következő 80 karaktert a videó RAM-ból, és előállítja a következő 14 rasztersort, és így tovább. Valójában a karaktereket legtöbbször minden egyes rasztersor előállítása során kiolvassák a memóriából, hogy a videovezérlőben ne legyen szükség pufferelésre. A 9-szer 14 bites karakterminták a videovezérlő által használt ROM-ban vannak. (RAM is használható a felhasználók által definiálható betűkészletek támogatásához.) A ROM-ot 12 bites címmel címzik meg, amelyből 8 bit a karakter kódja, 4 bit a rasztersor azonosítása. A ROM minden egyes bájtjának 8 bitje 8 képpontot vezérel; a 9. – a karakterek közötti képpont – mindig üres. Így 14 × 80 = 1120 videó RAM-beli memóriahivatkozásra van szükség a szöveg egy sorának a képernyőn való megjelenítéséhez. Ugyanennyi hivatkozás történik a karakterképeket tároló ROM-ra is.

Az eredeti IBM PC gépeknek több képernyő üzemmódja volt. A legegyszerűbb esetben a konzol egy karakteres megjelenítő volt. A 3.26.(a) ábrán a videó RAM egy részletét láthatjuk. Minden egyes képernyőn lévő karakter – a 3.26.(b) ábrán látunk egy példát – két bájtot foglal el a RAM-ban, az alacsony helyi értéken van a megjelenítendő karakter ASCII kódja. A magas helyi érték egy attribútumbájt,



3.26. ábra. *(a) A videó RAM tartalma az IBM monokróm megjelenítő esetén. × jelöli az attribútumbájtokat. (b) A megfelelő képernyőkép*

amely meghatározza a színt, az inverz módot, villogó módot és így tovább. Ebben a módban a 25 × 80 karaktert tartalmazó teljes képernyőhöz 4000 bájt videó RAM szükséges. Minden modern megjelenítő a mai napig támogatja ezt az üzemmódot.

A jelenlegi grafikus terminálok ugyanezt az elvet használják, azonban ekkor minden egyes képpontot a képernyőn önállóan vezérelnek. A legegyszerűbb konfigurációban, monokróm képernyő esetén, minden képpontnak egy bit felel meg a videó RAM-ban. Egy másik szélsőséges esetben minden képpontot egy 24 bites szám ábrázol, 8-8 bit a piros, a zöld és a kék színt. Egy 768 × 1024 pixel felbontású színes képernyő képének tárolásához 24 bites pixelekkel 2 MB RAM szükséges.

A tárcímleképezéses megjelenítők megjelenésével a billentyűzet teljesen elvált a képernyőtől. A billentyűzet soros vagy párhuzamos porton keresztül csatlakoztatható. Minden egyes billentyű leütésekor a CPU-hoz egy megszakításkérés érkezik, és a billentyűzetmeghajtó egy I/O-kapu kiolvasásával kapja meg a leütött karaktert.

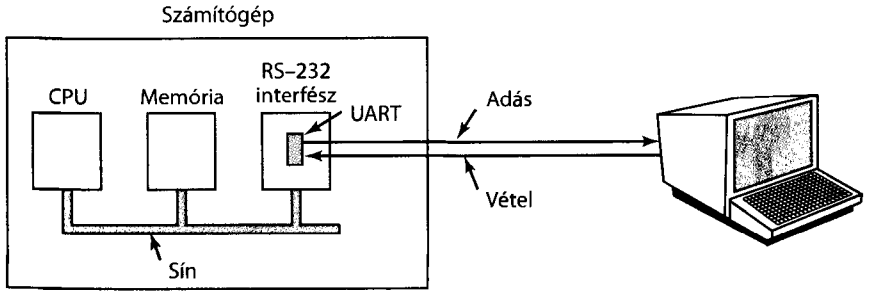
Egy PC-n a billentyűzet egy beépített mikroprocesszort tartalmaz, amely egy speciális soros porton keresztül kommunikál az alaplapon lévő vezérlővel. Megszakításkérés keletkezik, valahányszor egy billentyűt lenyomnak, és akkor is, amikor egyet felengednek. Továbbá a billentyűzethardver csupán a megnyomott gomb billentyűkódját (scan code), és nem az ASCII (American Standard Code for Information Interchange) kódját továbbítja. Amikor az *A* gombot lenyomják, a 30-as billentyűkód kerül az I/O-regiszterbe. A meghajtóprogram feladata, hogy eldöntse róla, hogy kisbetűs, nagybetűs, CTRL-A, ALT-A, CTRL-ALT-A vagy valamilyen más billentyűkombináció. Mivel a meghajtóprogram tudja, hogy mely billentyűket nyomták le, de még nem engedtek fel (például SHIFT), így elég információval rendelkezik e feladat végrehajtásához. Bár ez a fajta billentyűzetillesztés minden terhet a szoftverre helyez, ugyanakkor rendkívül rugalmas. Például a felhasználói programok megtudhatják, hogy a lenyomott számjegy a billentyűzet felső sorából vagy a jobb oldali numerikus billentyűzetről érkezett. Elvileg a meghajtóprogram ezt az információt meg tudja adni.

RS–232-es terminálok

Az RS–232-es terminálok olyan eszközök, amelyek egy billentyűzetből és egy megjelenítőből állnak, amelyek egy soros interfészen keresztül bitenként kommunikálnak (lásd 3.27. ábra). Ezek a terminálok egy 9 vagy 25 tűs csatlakozóval rendelkeznek, amelyből egy tű az adatküldésé, egy tű az adatfogadásé és egy tű a föld. A többi tű a különféle vezérlőjeleké, amelyek nagyobb részét nem használják. Ahhoz, hogy egy karaktert egy RS–232-es terminálhoz eljuttassunk, a számítógépnek bitenként el kell küldeni a karakter kódját, amelyet megelőz egy start bit, és 1 vagy 2, a karaktereket elválasztó stop bit követ. A stop bitek elé egy paritás bit is kerülhet, amely alapszintű hibafelderítést tesz lehetővé, bár ez csak nagyszámítógépes rendszer esetén szokásos követelmény. Az általánosan elterjedt átviteli sebességek 14400 és 56000 bit/s, az előbbi a fax, az utóbbi adatok átvitelére. Az RS–232-es terminálokat általában arra használják, hogy egy távoli számítógéppel teremtsenek kapcsolatot egy modem és egy telefonvonal segítségével.

Minthogy a számítógép és a terminál is karakterekkel dolgozik, de a soros vonalon bitenként kell kommunikálniuk, ezért kifejlesztettek olyan mikroáramköröket, amelyek megvalósítják a karakter-soros átvitel, illetve a soros átvitel-karakter konverziókat. Ezeket **UART**-nak **(Universal Asynchronous Receiver Transmitters)**, univerzális aszinkron adóvevőnek hívják. Az UART-okat úgy kapcsolják a számítógépekhez, hogy egy RS–232-es illesztőkártyát csatlakoztatnak a rendszersínhez, ahogyan a 3.27. ábra mutatja. A modern számítógépekben az UART és az RS–232 interfész gyakran része az alaplap áramköreinek. Az alaplapon lévő UART-ot le lehet tiltani, és ezzel lehetővé lehet tenni egy modemillesztő kártya csatlakoztatását a sínhez vagy mindkettő működését. Egy modemben ugyancsak van egy UART (amelyet esetleg más funkciókkal együtt integrálnak egy multifunkciós lapkára), a kommunikációs csatorna azonban inkább egy telefonvonal, mint soros kábel. Azonban a számítógép számára az UART ugyanúgy néz ki, függetlenül attól, hogy a médium egy dedikált soros kábel vagy egy telefonvonal.

Az RS–232-es terminálok kihalófélben vannak, PC-k foglalják el a helyüket, bár még mindig előfordulnak régebbi nagyszámítógépes rendszerekben, különösen a



3.27. ábra. *Egy RS–232-es terminál egy adatvonalon keresztül, bitenkénti átvitellel kommunikál a számítógéppel. A számítógép és a terminál teljesen függetlenek egymástól*

banki, repülőjegy-foglalási és hasonló alkalmazásoknál. Az olyan terminálprogramokat azonban, amelyek megengedik, hogy egy távoli számítógép szimuláljon egy terminált, még mindig széles körben használják.

Egy karakter megjelenítéséhez a terminálmeghajtó program beírja a karaktert az illesztőkártyába, ahol az tárolódik, majd az UART bitenként továbbítja a soros vonalra. Még 56000 bit/s esetén is 140 μs-ra van szükség egy karakter továbbításához. Ennek az alacsony átviteli sebességnek az eredménye, hogy a meghajtóprogram általában elküld egy karaktert az RS–232-es illesztőkártyának, majd blokkolt állapotba kerül, és várja az illesztő által generált megszakítást, amikor a karaktert továbbította, és az UART képes egy újabb karakter fogadására. Az UART, mint az a nevében is benne van, képes a karaktereket egyidejűleg fogadni és küldeni. Megszakításkérés keletkezik egy karakter beérkezésekor is, és általában lehetőség van kevés számú beérkező karakter pufferelésére is. Egy megszakítás kiszolgálásakor a terminálmeghajtó programnak egy regisztert kell megvizsgálnia, hogy meghatározza a megszakítás okát. Vannak olyan illesztőkártyák, amelyek CPU-t és memóriát is tartalmaznak, és több vonalat is tudnak kezelni, átvéve ezzel a CPU-tól az I/O terhének nagy részét.

Ahogy már említettük, az RS–232-es terminálok alosztályokba sorolhatók. A legegyszerűbbek a papírra író (hardcopy) terminálok voltak. Ezek a billentyűzeten leütött karaktereket továbbították a számítógépbe, a számítógép által küldött karaktereket pedig papírra írták. Ezek a terminálok mára elavultak, és igen ritkán láthatók.

A „buta" CRT-terminálok ugyanígy működnek, csak papír helyett képernyőt használnak. Ezeket „üvegre író"-nak (glass tty) is szokás hívni, mivel ugyanazt a tevékenységet végzik, mint a papírra író tty terminálok. (A „tty" kifejezés – a Teletype® rövidítése –, ami egy korábbi, a számítógépterminál-piacon úttörő munkát végző társaság neve; a „tty" elfogadottá vált tetszőleges fajtájú terminál jelölésére.) A „buta" terminálok (glass tty-k) mára szintén elavultak.

Az intelligens CRT-terminálok valójában kicsinyített, speciális célú számítógépek. Van bennük CPU és memória, és egy programot is tartalmaznak, általában ROM-ban. Az operációs rendszer szempontjából a „buta" terminálok és az intelligens terminálok között a legfőbb különbség az, hogy az utóbbiak megértenek bizonyos vezérlőszekvenciákat. Például elküldve az ASCII ESC karaktert (033), majd bizonyos másik karaktereket, a kurzor tetszőleges helyre mozgatható a képernyőn, beszúrhatunk szöveget a képernyő közepén, és így tovább.

3.8.2. A terminálszoftver

A billentyűzet és a képernyő majdnem független eszközök, így különválasztva fogjuk kezelni őket a továbbiakban. (Nem teljesen függetlenek, hiszen a leütött billentyűket meg kell jeleníteni a képernyőn.) A MINIX 3-ban a billentyűzet- és a képernyőmeghajtó ugyanannak a processzusnak a részei; más rendszerekben szétválhatnak különálló meghajtókká.

A beolvasást kezelő szoftver

A billentyűzetmeghajtó alapfeladata, hogy összegyűjtse a billentyűzetről érkező adatokat, és továbbítsa a felhasználói programoknak, amikor azok a terminálról olvasnak. Két lehetséges filozófiát fogadhatunk el a meghajtóprogram szempontjából. Az első szerint a meghajtó feladata csak az, hogy fogadja az érkező adatokat és továbbítsa felfelé módosítás nélkül. A felhasználói program, amikor a terminálról olvas, ASCII kódok nyers sorozatát kapja. (Billentyűsorszámot továbbítani túl primitív és túlságosan gépfüggő lenne.)

Ez a szemlélet különösen megfelel az olyan fejlett képernyőszerkesztők igényeinek, mint az *emacs,* amelyek lehetővé teszik a felhasználóknak, hogy tetszőleges műveletet rendeljenek bármely karakterhez vagy karaktersorozathoz. Ez azonban azt jelenti, hogy ha a felhasználó a *date* szó helyett azt gépeli, hogy *dste,* majd kijavítja a hibát három visszatörlés (backspace), az *ate* és a kocsi vissza karakterek begépelésével, a felhasználói program megkapja mind a 11 begépelt ASCII kódot.

A programok többsége nem kíván ennyi részletet. Csak a javított bemenő adatra van szükségük, és nem arra, hogy pontosan milyen karaktersorozattal állították elő. Ez a megfigyelés vezet el a második filozófiához: a meghajtó kezeli a soron belüli szerkesztési műveleteket, és csak a kijavított sorokat továbbítja a felhasználói programoknak. Az első filozófia karakterorientált, a második pedig sororientált. Eredetileg a **nyers mód (raw mode)** és a **feldolgozott mód (cooked mode)** kifejezésekkel hivatkoztak rájuk. A POSIX szabvány a kevésbé képszerű **kanonikus mód** kifejezést használja a sororientált módra. A legtöbb rendszerben a kanonikus mód egy jól definiált konfigurációra vonatkozik. A **nemkanonikus mód** a nyers móddal azonos, bár a terminál viselkedésének sok részlete eltérő lehet. A POSIX-kompatibilis rendszerek különféle könyvtári függvényeket biztosítanak, amelyek támogatják bármelyik mód választását, és lehetőséget adnak a terminálkonfiguráció sok tulajdonságának változtatására. A MINIX 3-rendszerben ezt a feladatot az ioctl rendszerhívás támogatja.

A billentyűzetmeghajtó első számú feladata a karakterek összegyűjtése. Ha minden billentyűleütés megszakítást okoz, a meghajtó a karaktert a megszakítás kiszolgálása során szerezheti meg. Ha a megszakítást az alacsony szintű szoftver átfordítja üzenetté, lehetővé válik, hogy az újonnan megszerzett karakter bekerüljön az üzenetbe. Másik lehetőség, hogy a karakter a memóriában lévő kis pufferbe kerül, és egy üzenet értesíti a meghajtót, hogy valami érkezett. Ez utóbbi megoldás biztonságosabb akkor, ha üzenetet csak várakozó processzus kaphat, és van valamekkora esély arra, hogy a billentyűzetmeghajtó még elfoglalt az előző karakter feldolgozásával.

Amint a billentyűzetmeghajtó megkapott egy karaktert, azonnal el kell kezdenie a feldolgozását. Ha a billentyűzet billentyűsorszámot küld, és nem az alkalmazói szoftver által használt karakterkódot, akkor a meghajtónak egy táblázat alapján kell a kódok között a konvertálást végrehajtania. Nem minden IBM-„kompatibilis" konfiguráció használja a szabványos billentyűzetsorszámokat, így ha a meghajtó ezeket is támogatni akarja, akkor a különböző billentyűzetekhez különböző táblázatokat kell hozzárendelnie. Egy egyszerű megközelítés, hogy

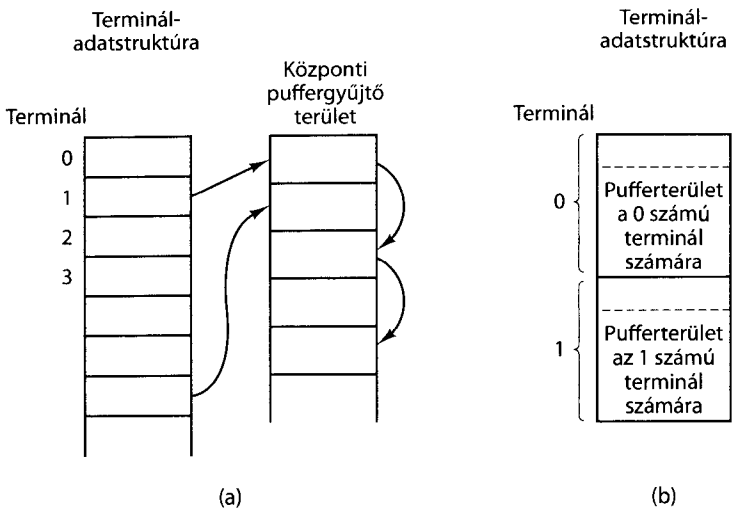
a billentyűzet által szolgáltatott kódokat ASCII (American Standard Code for Information Interchange) kódokra átalakító táblázatot befordítják a billentyűzetmeghajtóba, azonban ez nem felel meg a nem angol nyelvet használóknak. A billentyűzet elrendezése különbözik az egyes országokban, az ASCII karakterkészlet még a Föld nyugati részén élő emberek többségének sem felel meg, ahol a spanyol, portugál és francia nyelvekben olyan ékezetes karakterekre és írásjelekre van szükség, amelyeket az angolban nem használnak. Annak érdekében, hogy a különböző nyelvek számára szükséges rugalmas billentyűzetelrendezés iránti igényt kielégítsék, sok operációs rendszer gondoskodik betölthető **billentyűzet-térképek**ről vagy **kódlapok**ról, amelyek lehetővé teszik, hogy megválaszthassuk a billentyűkód és az alkalmazásnak átadott kód közötti hozzárendelést rendszerbetöltéskor vagy később.

Ha a terminál kanonikus (feldolgozott) módban van, a karaktereket tárolni kell, amíg egy teljes sor össze nem gyűlik, mivel a felhasználó később dönthet úgy, hogy a sor egy részét törli. Akkor is szükség van pufferelésre, ha a terminál nyers módban van, hisz előfordul, hogy a program még nem kérte az inputot, ezért a karaktereket átmenetileg tárolni kell, megengedve az előre gépelést. (Azok a rendszertervezők, akik az előre gépelést nem engedik meg a felhasználóknak, megérdemelnék, hogy kátrányba és tollba hempergessék őket, vagy ami még rosszabb, hogy kényszerítsék őket arra, hogy a saját rendszerüket használják.)

Két közismert módja van a karakterpufferelésnek. Az első esetben a meghajtó egy közös puffergyűjtő területet alkalmaz, ahol minden egyes puffer tartalmazhat, mondjuk, 10 karaktert. Minden terminálhoz egy adatszerkezet kapcsolódik, mely több más adat mellett egy mutatót is tartalmaz az olyan pufferekből álló láncra, amelyek az adott terminálról érkező adatokat tárolják. Ahogy érkeznek a karakterek, újabb puffereket foglalnak le és fűznek hozzá a lánchoz. Amikor a karaktereket továbbították a felhasználói programnak, a puffert kiveszik a láncból, és visszateszik a közös puffergyűjtő területre.

A másik megközelítésben a terminálhoz tartozó adatszerkezet közvetlenül tartalmazza a puffert, közös pufferterület nincs. Minthogy a felhasználók számára megszokott, hogy begépelnek egy parancsot, amelynek végrehajtása egy kis időbe kerül (mondjuk egy fordítás), majd előre gépelnek néhány sort. Hogy ez biztonságosan működhessen, a meghajtónak terminálonként úgy 200 karaktert kell lefoglalnia. Egy 100 terminálos nagyméretű osztott rendszerben az előre gépelés biztosításához 20 K memóriát állandóan lefoglalni nyilvánvalóan túlságosan sok, míg egy közös puffergyűjtő területtel valószínűleg 5 K tárterület is elegendő. Másrészről viszont a terminálonkénti pufferterület-foglalás egyszerűbbé teszi a meghajtót (nincs láncoltlista-feldolgozás), ezért ezt használják inkább a csak egykét terminált kiszolgáló személyi számítógépek esetén. A 3.28. ábra mutatja be a két módszer közti különbséget.

Jóllehet a billentyűzet és a képernyő logikailag különálló eszközök, sok felhasználó ahhoz szokott hozzá, hogy a begépelt karakterek rögtön megjelennek a képernyőn. Néhány (régebbi) terminál kénytelen (hardver által) automatikusan megjeleníteni, amit éppen begépelnek, ami nemcsak kellemetlen, ha jelszót gépelnek be, de meglehetősen korlátozza a fejlettebb szövegszerkesztők és egyéb prog-



3.28. ábra. *(a) Központi puffergyűjtő terület. (b) Terminálonként saját pufferterület*

ramok rugalmasságát. Szerencsére a PC-k billentyűzete nem jelenít meg semmit billentyűleütéskor, így a szoftveren múlik a bemenő adatok megjelenítése. Ezt a folyamatot **echózás**nak hívják.

Az echózást bonyolítja az a tény, hogy a program is írhat a képernyőre, amíg a felhasználó gépel. Az a minimum, hogy a billentyűzetmeghajtónak kell kiszámítania, hogy hová írja ki az újonnan beérkező adatokat úgy, hogy azt a program kimenő adatai ne írják felül.

Bonyolítja az echózást az is, ha egy 80 karakter sorhosszúságú terminálon 80-nál több karaktert gépelnek be. Az alkalmazástól függően megfelelő lehet a sor megtörése és új sor kezdése. Néhány meghajtóprogram levágja a sort 80 karakterre, és eldobja a 80. oszlop után következő összes karaktert.

Egy másik probléma a tabulátor (TAB) kezelése. Minden terminál rendelkezik TAB billentyűvel, de a megjelenítők a tabulátort a kimeneten tudják kezelni. A meghajtó feladata, hogy kiszámítsa a kurzor aktuális helyét, figyelembe véve mind a programok kimenetét, mind az echózás által adott kimenetet, és kiszámítani a szóközök megfelelő számát, amit echózni kell.

Ezzel elérkeztünk az eszközök ekvivalenciájának problémájához. Egy szövegsor végén logikailag szükség van egy kocsi vissza karakterre, hogy a kurzor visszakerüljön az 1. oszlopba és egy soremelésre, hogy a következő sorra álljon. Nem lenne szerencsés, ha a felhasználótól megkövetelnénk mindkét karakter leütését minden egyes sor végén (bár néhány régi terminálnak van olyan billentyűje, amely előállítja mindkettőt, 50 százalék esélye van rá, hogy ezt abban a sorrendben teszi, ahogy a szoftver kéri őket). A meghajtón múlt (és múlik ma is), hogy tetszőleges bejövő adatot az operációs rendszer által használt szabványos belső formára konvertáljon.

Ha a szabványos forma csak egy soremelés tárolása (ez a Unix és minden utódjának a konvenciója), akkor a kocsi vissza karaktereket soremeléssé kell átalakítani. Ha a belső forma szerint mindkettőt tárolni kell, akkor a meghajtónak be kell szúrnia egy soremelést, ha kocsi vissza érkezik, és egy kocsi visszát, ha soremelést kap. Akármi is a belső szabvány, előfordulhat, hogy a terminálnak kocsi visszát és a soremelést is kell kapnia az echózáshoz, hogy a képernyőt megfelelően frissítse. Minthogy egy nagyszámítógéphez igen sokféle terminál csatlakozhat, a billentyűzetmeghajtón múlik, hogy az összes különböző kocsi vissza/soremelés kombinációkat a belső rendszer szabványára konvertálja, és úgy rendezze, hogy az összes echózás megfelelően történjen.

Ehhez kapcsolódó probléma a kocsi vissza és soremelés időzítése. Néhány terminálon a kocsi vissza vagy soremelés megjelenítése hosszabb ideig tarthat, mint egy betűé vagy számé. Ha a terminálba épített mikroprocesszornak egy nagy szövegblokkot kell másolnia, hogy görgesse a képet, akkor a soremelés lassú lehet. Ha egy mechanikus írófejnek kell a papír bal margójához visszatérnie, a kocsi vissza lehet lassú. Mindkét esetben a meghajtóra vár, hogy **kitöltő karaktereket** (közömbös null karakterek) szúrjon be a kimenő folyamba, vagy megállítsa a kiírást arra az időre, amíg a terminál utoléri magát. A várakozáshoz szükséges idő mennyisége gyakran a terminál sebességével van kapcsolatban, például 4800 bit/s esetén nincs szükség várakozásra, de 9600 bit/s esetén egy kitöltő karakterre lehet szükség. A hardveres TAB-bal rendelkező termináloknak, főleg a papírra író fajtáknak, szüksége lehet várakozásra TAB után is.

Kanonikus módú működés esetén számos bejövő karakternek speciális jelentése van. A 3.29. ábra bemutatja az összes POSIX által megkívánt speciális karaktert és azokat a továbbiakat, amelyeket a MINIX 3 ismer fel. Az alapértelmezés szerint kiosztott karakterek mindegyike vezérlő karakter, amely nem ütközhet a szöveges be-

Karakter	POSIX-név	Megjegyzés
CTRL-D	EOF	Fájlvége
	EOL	Sorvége (nem definiált)
CTRL-H	ERASE	Egy karakter visszatörlése (backspace)
CTRL-C	INTR	Processzus megszakítása (SIGINT)
CTRL-U	KILL	Az egész megkezdett sor törlése
CTRL-\	QUIT	Processzus memóriakép nyomtatás kiváltása (SIGQUIT)
CTRL-Z	SUSP	Felfüggesztés (a MINIX figyelmen kívül hagyja)
CTRL-Q	START	Kiírás elkezdése
CTRL-S	STOP	Kiírás megállítása
CTRL-R	REPRINT	Bemenet újra megjelenítése (MINIX-kiterjesztés)
CTRL-V	LNEXT	Betű következik (MINIX-kiterjesztés)
CTRL-O	DISCARD	Kiírás mellőzése (MINIX-kiterjesztés)
CTRL-M	CR	Kocsi vissza (Nem változtatható)
CTRL-J	NL	Soremelés (Nem változtatható)

3.29. ábra. *A speciálisan kezelt karakterek kanonikus mód esetén*

menettel vagy a programok által használt kódokkal, azonban az utolsó kettő kivételével mindegyik megváltoztatható az *stty* paranccsal, ha szükséges. A Unix régebbi változataiban ezek közül többnek is más volt az alapértelmezés szerinti értéke.

Az *ERASE* karakter lehetővé teszi az éppen begépelt karakter törlését. A MINIX 3-ban ez a visszatörlés (backspace, CTRL-H). Ez nem kerül be a bejövő karakterek sorába, hanem kitörli az őt megelőző karaktert a sorból. Három karakterből álló sorozat formájában echózható, úgymint visszatörlés, szóköz, visszatörlés, hogy az előző karakter eltűnjön a képernyőről. Ha az előző karakter egy TAB volt, kitörléséhez szükséges annak nyomon követése, hogy a kurzor hol volt a TAB előtt. A rendszerek többségében visszatörléssel csak az aktuális soron belül törölhetünk karaktereket, a kocsi vissza karakter nem törölhető ki, és nem lehet az előző sorhoz visszatérni.

Amikor a felhasználó a begépelt sor elején vesz észre egy hibát, gyakran kényelmesebb az egész sort törölni és elölről kezdeni. A *KILL* karakter (a MINIX 3-ban CTRL-U) törli a teljes sort. MINIX 3-ban a törölt sor el is tűnik a képernyőről, néhány rendszer azonban echózza egy kocsi vissza és soremelés karakterrel a végén, mivel néhány felhasználó szereti látni a régi sort. Következésképp ízlés dolga a *KILL* echózásának módja. Ugyanúgy, mint az *ERASE* esetén, általában nincs lehetőség arra, hogy az aktuális sor elé visszamenjünk. Ha egy karaktercsoportot törölnek, kérdés, hogy megéri-e a fáradságot a meghajtónak, vagy nem, hogy visszaadja a puffert a közösbe.

Előfordul, hogy az *ERASE* és *KILL* karaktereket ugyanúgy kell bevinni, mint a közönséges adatokat. Az *LNEXT* úgy működik, mint egy **váltó karakter (escape karakter, ESC).** A MINIX 3-ban CTRL-V az alapértelmezett értéke. Például a régebbi Unix-rendszerek a @ jelet használták *KILL* jelként, de az internetes levelezőrendszerek *linda@cs.washington.edu* formájú levélcímeket használnak. Ha valaki jobban kedveli a régebbi konvenciókat, újradefiniálja a *KILL* jelet mint @, de a levél címzéséhez a @ jelet mint szöveget kell bevinnie. Ezt teheti meg a CTRL-V és a @ begépelésével. A CTRL-V maga is bevihető szövegként a CTRL-V CTRL-V begépelésével. Egy CTRL-V karaktert látva a meghajtó beállít egy jelzőt, mely azt mondja meg, hogy a következő karakter mentesül a speciális feldolgozás alól. Az *LNEXT* karakter maga nem kerül be a beérkező karakterek sorába.

A felhasználó számára vezérlő kódok állnak rendelkezésre, hogy megakadályozza a képernyőkép kigördülését a látótérből, hogy befagyassza a képet, majd később továbbindítsa. A MINIX 3-ban ezek a *STOP* (CTRL-S) és a *START* (CTRL-Q) kódok. Nem tárolódnak, hanem csak egy jelzőt állítanak be vagy törölnek ki a terminál adatszerkezetében. Valahányszor kiírásra történik kísérlet, ezt a jelzőt megvizsgálják. Ha a jelző be van állítva, nem történik kiírás. Általában a program kiírása mellett az echózás is le van tiltva.

Gyakran van szükség nyomkövetés közben egy éppen futó program megszüntetésére. Az *INTR* (CTRL-C) és a *QUIT* (CTRL-\) karakterek használhatók erre a célra. A MINIX 3-ban a CTRL-C az összes processzusnak, amelyet a terminálról indítottak, elküldi a SIGINT szignált. A CTRL-C megvalósítása elég érdekes. A nehéz része a feladatnak a szignál eljuttatása a meghajtótól a rendszer azon részéhez, amely a szignálokat kezeli, amikor az valójában nem is kérte ezt az információt. A CTRL-\ hasonló a CTRL-C-hez, kivéve azt, hogy SIGQUIT szignált küldi el, amely egy memóriakép kiírását váltja ki, hacsak nem fogják el vagy nem hagyják figyelmen kívül.

Amikor e billentyűk bármelyikét leütik, a meghajtónak a kocsi vissza, soremelés karaktereket kell echóznia, továbbá az összes összegyűjtött bemenő adatot ki kell törölnie a tiszta lappal történő indítás érdekében. Történetileg a DEL-t széles körben használták az *INTR* kezdeti értékeként sok Unix-rendszerben. Mivel sok program használja szövegszerkesztésre a DEL-t felcserélhetően a visszatörléssel (backspace), ezért ma inkább a CTRL-C-t részesítik előnyben.

Egy másik speciális karakter az *EOF* (CTRL-D), amelynek hatása a MINIX 3-ban az, hogy minden, a terminálhoz érkezett függőben lévő olvasási kérelem kielégítésre kerül, bármi is található a pufferben, még akkor is, ha a puffer üres. A sor elején CTRL-D karaktert gépelve a program 0 bájt hosszúságú bemenő adatot kap, amit úgy szokás értelmezni, mint a fájl végét, és erre a legtöbb program ugyanúgy viselkedik, mintha egy bemenő adatállomány végjelét látná.

Néhány terminálmeghajtó sokkal ügyesebb soron belüli szerkesztést tesz lehetővé, mint amit itt vázoltunk. Speciális vezérlő karaktereik vannak egy szó törlésére, karakterenkénti vagy szavankénti előre- vagy hátralépésre, az épp gépelt sor elejére vagy végére ugrásra, és így tovább. Ezeknek a funkcióknak a hozzáadása a meghajtó méretét jócskán megnöveli, és felesleges is, amikor a fejlett képernyőszerkesztő programok valójában nyers módban dolgoznak.

A POSIX szabvány megköveteli, hogy a szabványos eljáráskönyvtár tartalmazzon néhány olyan függvényt, amelyek segítségével a programok a terminálok paramétereit felügyelhetik, és amelyek közül a legfontosabbak a *tcgetattr* és a *tcsetattr*. A *tcgetattr* a 3.30. ábrán látható *termios* struktúra másolatát kéri le, amely az összes olyan információt tartalmazza, ami a speciális karakterek megváltoztatásához, a működési mód beállításához és a terminál más jellemzőinek módosításához szükséges. A program meg tudja vizsgálni a pillanatnyi beállításokat, és az igényeknek megfelelően módosíthatja azokat. A *tcsetattr* ezután visszaírja az adatstruktúrát a terminálmeghajtóba.

```
struct termios {
    tcflag_t c_iflag;       /* bemeneti módok */
    tcflag_t c_oflag;       /* kimeneti módok */
    tcflag_t c_cflag;       /* vezérlő módok */
    tcflag_t c_lflag;       /* helyi üzemmódok */
    speed_t c_ispeed;       /* bemeneti sebesség */
    speed_t c_ospeed;       /* kimeneti sebesség */
    cc_t c_cc[NCCS];        /* vezérlőkarakterek */
};
```

3.30. ábra. *A termios struktúra. A MINIX 3-ban a* tcflag_t *egy short, a* speed_t *egy int és a* cc_t *egy char típus*

A POSIX szabvány nem határozza meg, hogy ezeket a követelményeket könyvtári függvényhívással vagy rendszerhívással kell-e megvalósítani. A MINIX 3 egy rendszerhívást szolgáltat, amelynek neve ioctl, hívásának módja pedig:

```
ioctl(file_descriptor, request, argp);
```

feladata, hogy megvizsgálja és módosítsa sok I/O-eszköz konfigurációját. Ezt a hívást használják a *tcgetattr* és *tcsetattr* függvények megvalósítására. A *request* változó értéke határozza meg, hogy a *termios* struktúrát olvasni vagy írni kell-e, és az utóbbi esetben, hogy a kérést azonnal figyelembe kell-e venni, vagy elhalasztódjon mindaddig, amíg a jelenleg sorban álló kiírások befejeződnek. Az *argp* változó egy mutató a hívóprogramban lévő *termios* struktúrára. A program és a meghajtó közötti kommunikációnak ezt a módját nem szépsége, hanem a Unix-kompatibilitás miatt választották.

Következzen néhány megjegyzés a *termios* struktúrához. A négy, jelzőket tartalmazó szó nagyfokú rugalmasságot biztosít. A *c_iflag* egyes bitjei vezérlik azt a sokféle módot, ahogyan a beolvasást kezelik. Például az *ICRNL* bit okozza a *CR* (kocsi vissza) karakterek *NL* (soremelés) karakterekké konvertálását a bemenő adatokban. Ez a jelző a MINIX 3-ban alapértelmezés szerint be van állítva. A *c_oflag* olyan biteket tartalmaz, amelyek a kiírás folyamatára vannak hatással, például az *OPOST* bit engedélyezi a kiírás végrehajtását. Ez és az *ONCLR* bit eredményezi, hogy a kiírásnál az *NL* karakterek *CR NL* szekvenciává konvertálódjanak; ez a MINIX 3-ban szintén alaphelyzetben be van állítva. A *c_cflag* egy vezérlőjelzőket tartalmazó szó. A MINIX 3 alapértelmezett beállítása egy 8 bites karaktereket fogadó vonalat engedélyez, valamint a modem bontja a kapcsolatot, ha a felhasználó a vonalon kijelentkezik. A *c_lflag* a lokális beállítások jelzőit tartalmazza. Egyik bitje, az *ECHO* az echózást engedélyezi (ez kikapcsolható bejelentkezéskor, a jelszó begépelésének biztonsága érdekében). A legfontosabb bitje az *ICANON*, amely a kanonikus mód használatát engedélyezi. Ha ez ki van kapcsolva, számos dologra nyílik lehetőség. Ha az összes többi beállítást az alapértelmezés szerint hagyjuk, a hagyományos **cbreak módd**al (karakterenkénti küldési móddal) egyenértékű üzemmódba kerülünk. Ebben a módban a karakterek anélkül kerülnek a programhoz, hogy egy teljes sor bevitelét megvárnák, de az *INTR, QUIT, START* és *STOP* karakterek megőrzik hatásukat. Mindez azonban letiltható a jelzők bitjeinek törlésével, hogy a hagyományos nyers móddal egyenértékű módhoz jussunk.

A sokféle speciális karaktert, amelyek megváltoztathatók, a MINIX 3-kiterjesztéseket is beleértve, a *c_cc* tömb tárolja. Ez a tömb tartalmaz két paramétert, amelyeket nemkanonikus mód esetén használnak. A *MIN* érték, amely *c_cc*[*VMIN*]-ben tárolódik, meghatározza azt a minimális karakterszámot, amelynek beérkezése szükséges ahhoz, hogy kielégíthető legyen egy read hívás. A *c_cc*[*VTIME*]-ban tárolt *TIME* érték egy időkorlátot állapít meg az ilyen hívásokra. A *MIN* és *TIME* befolyásolja egymást, ahogyan az a 3.31. ábrán látható, ahol egy *N* bájtot kérő hívást mutatunk be. A *TIME* = 0 és *MIN* = 1 esetén a szokásos nyers módhoz hasonlóan viselkedik a rendszer.

	TIME = 0	**TIME > 0**
MIN = 0	Azonnal visszatér azzal, ami hozzáférhető, 0 és *N* közötti számú karakterrel	Azonnal elindul az időzítő. Az első bevitt karaktert, vagy ha letelt az idő, akkor a 0 karaktert adja vissza
MIN > 0	Legalább *MIN* és legfeljebb *N* bájt visszaadása. Meghatározatlan ideig tartó blokkolt állapot lehetséges	A bájtok közti időzítő az első bájt után indul. *N* bájtot ad vissza, ha az idő lejártáig ennyi megérkezik, de legalább 1 bájtot. Meghatározatlan ideig tartó blokkolt állapot lehetséges

3.31. ábra. *Nemkanonikus módban a* MIN *és a* TIME *értékei meghatározzák, hogy egy olvasási kérés mikor tér vissza.* N *a kért bájtok száma*

A kiírást kezelő szoftver

A kiírás egyszerűbb, mint a beolvasás, bár az RS–232-es terminálok meghajtói nagyon különböznek a tárcímleképezéses terminálok meghajtóitól. A szokásos módszer az RS–232-es terminálok esetén, hogy minden terminálhoz kapcsolódik egy kimeneti puffer. A pufferek tartozhatnak ahhoz a közös pufferterülethez, ahol a bemeneti pufferek is találhatók, vagy egy terminálhoz rendelhetik őket úgy, ahogy a beolvasásnál láttuk. Amikor a programok a terminálra írnak, a kimenő adatok először a pufferekbe kerülnek. Hasonlóan az echózott adatok is a pufferekbe másolódnak. Miután az összes adat a pufferekbe került (vagy a pufferek megteltek), az első karakter kiíródik, és a meghajtóprogram várakozó állapotba kerül. Amikor a megszakítás megérkezik, a következő karakter kerül kiírásra, és így tovább.

A tárcímleképezéses terminálóknál egy egyszerűbb megoldás lehetséges. A kiírandó karaktereket egyesével kiolvassák a felhasználói területről, és közvetlenül a videó RAM-ba írják. Az RS–232-es terminálok esetén a kiírandó karaktereket csupán a terminálhoz vezető vonalra kell küldeni. A tárcímleképezéses terminálok esetén vannak karakterek, amelyek speciális kezelést igényelnek, többek között a visszatörlés, a kocsi vissza, a soremelés és a csengetés (CTRL-G). A tárcímleképezéses terminálok meghajtójának szoftveresen kell nyomon követnie a videó RAM-ban az aktuális pozíciót, ahová a kiírandó karakterek kerülhetnek, és az aktuális pozíciót tovább kell mozgatnia. A visszatörlés, a kocsi vissza és a soremelés mind az aktuális pozíció megfelelő módosítását igényli. A TAB-ok ugyancsak speciális kezelést igényelnek.

Különösen akkor, ha a képernyő legalsó sorában egy soremelést írunk ki, és a képernyő tartalmát gördíteni kell. Hogy lássuk, hogyan történik a görgetés, tekintsük a 3.26. ábrát. Ha a videovezérlő a RAM olvasását mindig a memória 0xB0000 címénél kezdi, karakter módban a képernyő görgetésének egyetlen módja az lenne, hogy 24 × 80 karaktert (minden karakterhez 2 bájt szükséges) a 0xB00A0 címről a 0xB0000 címre másolunk; ez bizony időigényes elképzelés. Bittérképes üzemmódban ez még rosszabb.

Szerencsére a hardver általában nyújt némi segítséget. A legtöbb videovezérlőnek van egy regisztere, amely meghatározza, hogy a videó RAM mely címétől kezdve kell a karaktereket a képernyő legfelső sorához kiolvasni. Ezt a regisztert a 0xB00A0 címre állítva a 0xB0000 helyett az előzőleg még másodikként megjelenő sor legfelülre kerül, és az egész képernyő feljebb gördül egy sorral. A meghajtónak így már csak az a dolga, hogy ami szükséges, az új legalsó sorba másolja. Amikor

Vezérlőszekvencia	Jelentés
ESC [*n*A	Felfelé mozgás *n* sorral
ESC [*n*B	Lefelé mozgás *n* sorral
ESC [*n*C	Jobbra mozgás *n* hellyel
ESC [*n*D	Balra mozgás *n* hellyel
ESC [*m*; *n*H	Kurzor mozgatása (*m*, *n*) pozícióba (y = *m*, x = *n*)
ESC [*s*J	Képernyő törlése a kurzortól (0 a végéig, 1 az elejétől, 2 az egész)
ESC [*s*K	Sortörlés a kurzortól (0 a végéig, 1 az elejétől, 2 az egész sor)
ESC [*n*L	*N* sor beszúrása a kurzor pozíciójában
ESC [*n*M	*N* sor törlése a kurzor pozíciójában
ESC [*n*P	*N* karakter törlése a kurzor pozíciójában
ESC [*n*@	*N* karakter beszúrása a kurzor pozíciójában
ESC [*n*m	Kiemelés engedélyezése *n* értékétől függően (0 = normál, 4 = félkövér, 5 = villogó, 7 = inverz)
ESC M	Visszafelé görgetés, ha a kurzor a felső sorban van

3.32. ábra. *A terminálmeghajtó által kiírás közben elfogadott ANSI-vezérlőszekvenciák. ESC jelzi az ASCII escape karaktert (0x1B),* n, m *és* s *opcionális numerikus paraméterek*

a videovezérlő elérkezik a RAM legfelső címéhez, átfordul, és szépen folytatja a bájtok kiolvasását a legalacsonyabb címtől kezdve.

Egy másik feladat, amivel a tárcímleképezéses terminálok meghajtójának foglalkoznia kell, a kurzorpozicionálás. Általában ennek megoldásához is nyújt a hardver némi segítséget egy regiszter formájában, amely meghatározza, hogy a kurzornak hová kell lépnie. Végül itt van a csengetés problémája. A csengetés megszólaltatásához a hangszóróra egy szinuszos vagy négyszögjelet kell küldeni, amely a videó RAM-tól meglehetősen elkülönült része a számítógépnek.

A képernyőszerkesztő programoknak és sok más bonyolult programnak szüksége van arra, hogy a képernyőt sokkal összetettebb módon frissítsék, mint csupán a szöveg görgetése a képernyő alján. Hozzájuk alkalmazkodva több terminálmeghajtó támogatja a különféle vezérlőszekvenciákat (escape sequence). Bár néhány terminál egyéni vezérlőszekvencia-készletet támogat, előnyös, ha van egy szabvány, amely megkönnyíti a szoftver adaptálását egyik rendszerből egy másikba. Az Amerikai Nemzeti Szabványügyi Intézet (American National Standards Institute, ANSI) definiált egy szabványos vezérlőszekvencia-készletet, amelyből a MINIX 3 támogatja az ANSI-vezérlőszekvenciáknak egy a 3.32. ábrán látható részhalmazát. Ezek elegendők a gyakori műveletek végrehajtásához. Amikor a meghajtó meglátja azt a karaktert, amellyel a vezérlőszekvencia kezdődik, beállít egy jelzőt, és addig vár, amíg a vezérlőszekvencia hátralévő része is megérkezik. Amikor minden megérkezett, a meghajtónak szoftveresen végre kell azt hajtania. Szöveg beszúrásához, törléséhez szövegblokkokat kell mozgatni a videó RAM-on belül. A hardver ebben már nem segít, kivéve a görgetést és a kurzor megjelenítését.

3.8.3. A MINIX 3 terminálmeghajtójának áttekintése

A terminálmeghajtó forrása négy C fájlban található (hatban, ha az RS–232-es és a pszeudoterminál támogatást is megengedjük), és ezek együtt messze a legnagyobb meghajtót alkotják a MINIX 3-ban. A terminálmeghajtó program méretét részben megmagyarázza az az észrevétel, hogy a meghajtó mind a billentyűzetet, mind a képernyőt kezeli, amelyek mindegyike önmagában is elég bonyolult eszköz, csakúgy, mint a két másik opcionális termináltípus. Mégis meglepetést okoz a legtöbb ember számára, hogy a terminál I/O harmincszor akkora mennyiségű kódot igényel, mint az ütemező. (Ezt az érzést erősíti számos operációs rendszerekről szóló könyv, amely harmincszor akkora teret szentel az ütemezésnek, mint az összes I/O-nak.)

A terminálmeghajtó több mint egy tucat üzenettípust fogad. A legfontosabbak a következők:

1. Olvasás a terminálról (az FS-től egy felhasználói processzus kezdeményezésére).
2. Írás a terminálra (az FS-től egy felhasználói processzus kezdeményezésére).
3. Terminálparaméterek beállítása ioctl számára (az FS-től egy felhasználói processzus kezdeményezésére).
4. Egy billentyűzetmegszakítás következett be (billentyűt nyomtak le vagy engedtek fel).
5. Az előző kérés visszavonása (a fájlrendszertől, amikor egy szignál érkezik).
6. Egy eszköz megnyitása.
7. Egy eszköz lezárása.

Más üzenettípusokat speciális célokra használnak, például diagnosztikai képernyők előállítására, amikor funkcióbillentyűket nyomnak le, vagy memóriaképmentés kiváltására.

Az íráshoz és az olvasáshoz használt üzenetek formátuma azonos, ahogy a 3.17. ábra mutatja, kivéve, hogy a *POSITION* mező nem szükséges. Egy mágneslemez esetén a programnak meg kell adnia, hogy melyik blokkot akarja olvasni. Egy billentyűzet esetén nincs választási lehetőség, a program mindig a következő begépelt karaktert kapja. A billentyűzetek nem támogatják a *seek* (keresés) műveletet.

A *tcgetattr* és *tcsetattr* POSIX-függvényeket, amelyek a terminálok attribútumainak (tulajdonságainak) lekérdezésére és módosítására használnak, az ioctl rendszerhívás támogatja. A helyes programozási gyakorlat az, ha ezeket és a többi, az *include/termios.h*-ban szereplő függvényt használjuk, és a C könyvtárra hagyjuk a könyvtárhívások ioctl rendszerhívásokká konvertálását. A MINIX 3-ban azonban szükség van néhány olyan vezérlő műveletre, amellyel a POSIX nem rendelkezik, például alternatív billentyűzettérkép betöltése, és ezekhez a programozónak explicit módon az ioctl-t kell használnia.

Az ioctl rendszerhívás által a meghajtónak küldött üzenet egy kéréskódot és egy mutatót tartalmaz. A *tcsetattr* függvény esetén az ioctl hívás a *TCSETS, TCSETSW* vagy *TCSETSF* kéréstípusból és egy *termios* struktúra címét tartalmazó mutató-

ból áll, amely olyan, mint a 3.30. ábrán látható struktúra. Minden ilyen hívás az éppen aktuális beállításokat újakkal cseréli le, a különbség annyi, hogy a *TCSETS* kéréstípus esetén az új beállítás azonnal hatályba lép, a *TCSETSW* típusú kérelem addig nem lép érvénybe, amíg az összes kiírandó adat átvitele meg nem történt, a *TCSETSF* pedig megvárja a kiírás végét, és a még be nem olvasott bemenő adatokat törli. A *tcgetattr* függvényhívás egy *TCGETS* kéréstípusú ioctl rendszerhívásra fordítódik át, és a kitöltött *termios* struktúrát adja vissza a hívónak, így az eszköz pillanatnyi állapota megvizsgálható. Azok az ioctl hívások, amelyek nem felelnek meg a POSIX által definiált függvényeknek, mint például *KIOCSMAP* kérés, amelyet új billentyűzettérkép betöltésére használnak, másfajta adatszerkezet címét tartalmazó mutatókat adnak át, ebben az esetben a mutató egy *keymap_t* címét tartalmazza, amely egy 1536 bájtos struktúra (128 billentyű × 6 módosítóbillentyű 16 bites kódjai). A 3.39. ábra foglalja össze, hogyan történik a szabvány POSIX-hívás ioctl rendszerhívássá konvertálása.

A terminálmeghajtó egy *tty_table* nevű fő adatszerkezetet használ; ez az egyes terminálokhoz rendelt *tty* struktúrákból álló tömb. Egy szokásos PC-nek egy billentyűzete és egy képernyője van, de a MINIX 3 képes maximum nyolc virtuális terminál támogatására, a képernyőadapter-kártyán lévő memória méretétől függően. Így a konzolnál ülő személy több példányban bejelentkezhet, és lehetősége van arra, hogy a képernyőre írást és billentyűzetről olvasást egyik „felhasználótól" a másikhoz kapcsolja. Két virtuális konzol esetén az ALT-F2 választja ki a második konzolt, az ALT-F1 pedig visszatér az elsőhöz. ALT és a nyílbillentyűk szintén használhatók. Ráadásul a soros vonalak két távoli felhasználó csatlakozását is támogatják RS–232-es kábelen vagy modemen keresztül, a **pszeudoterminálok** pedig a felhasználók hálózaton keresztüli csatlakozását teszik lehetővé. A meghajtóprogram írásakor ügyeltek arra, hogy könnyű legyen újabb terminálok hozzáadása. A szabványos konfiguráció, két virtuális konzolt tartalmaz, és a soros vonalak és a pszeudoterminálok le vannak tiltva.

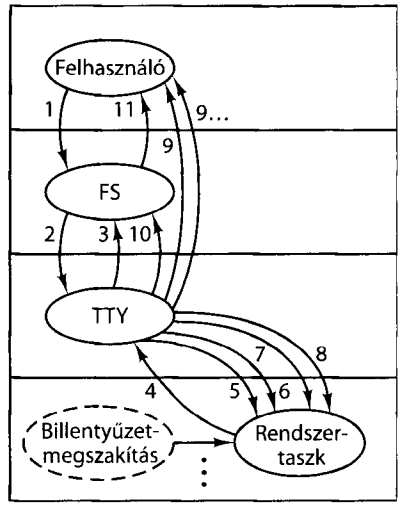
Minden egyes *tty* struktúra a *tty_table* tömbben nyomon követi mind a beolvasást, mind a kiírást. A beolvasáshoz egy várakozási sort tart fenn, amelyben a már begépelt, de a program által még be nem olvasott karakterek vannak, információkat tárol azokról a karakterbeolvasási kérésekről, amelyek még nem érkeztek be, és egy időkorlát, amely biztosítja, hogy az olvasási kérelmet úgy lehessen végrehajtani, hogy a meghajtó ne blokkoljon folyamatosan, ha nem gépeltek be egyetlen karaktert sem. A kiíráshoz a még be nem fejezett kiírási kérelmek paramétereit tárolja. Más mezők egyéb általános változókat tartalmaznak, mint például az előbb tárgyalt *termios* struktúra, amely mind a beolvasás, mind pedig a kiírás számos tulajdonságát befolyásolja. A *tty* struktúra tartalmaz egy olyan mezőt is, amely olyan információkra mutat, amelyek csak bizonyos csoportokba tartozó eszközök számára, de nem az összes *tty_table*-ben található összes eszköz számára szükségesek. Például a konzolmeghajtó hardverfüggő részének szükséges az aktuális képernyő-pozíció, videó RAM pozíció és a megjelenítéshez az aktuális szín- (attribútum-) bájt, de ez az információ nem szükséges egy RS–232-es vonal támogatásához. Az egyes eszköztípusokhoz tartozó egyedi adatszerkezetek

ugyanott találhatók, mint ahol azok a pufferek, amelyek a megszakításkezelő rutinoktól kapott bemeneteket fogadják. Az olyan lassú eszközök, mint a billentyűzet, nem igényelnek olyan nagy puffert, mint amekkorát a gyors eszközök.

Terminálbemenet

Hogy jobban megértsük, hogyan működik a meghajtóprogram, nézzük meg, hogy a terminálon begépelt karakterek milyen mechanizmuson keresztül találnak utat a rendszeren át ahhoz a programhoz, amely kéri őket. Bár ez a fejezet csupán áttekintés kíván lenni, mégis megadjuk a forrásprogram megfelelő sorainak (CD melléklet) számát, hogy segítsünk az olvasónak megtalálni minden egyes felhasznált függvényt. (Bár kalandos útnak tűnhet a bemenettel kapcsolatos gyakorlatokhoz a kódot a *tty.c*, a *keyboard.c* és a *console.c* fájlokból venni, amelyek mindegyike terjedelmes fájl.)

Amikor egy felhasználó bejelentkezik a rendszer konzolján, akkor egy parancsértelmező (shell) indul el, amelyben a szabványbemenet, -kimenet és a szabvány hibacsatorna a */dev/console* eszközre van irányítva. A parancsértelmező elindul és olvasni próbál a szabványbemenetről a *read* könyvtári eljárás meghívásával. Ez az eljárás egy olyan üzenetet küld, amely tartalmazza a fájlleírót, a puffercímet és a helyszámlálót a fájlrendszerhez. Ez az (1) jelű üzenet a 3.33. ábrán. Az üzenet elküldése után a parancsértelmező blokkolt állapotba kerül, és vár a válaszra. (A fel-



3.33. ábra. *A terminálhoz érkező olvasási kérés, amikor épp nincs várakozó bejövő karakter. FS a fájlrendszer, TTY a terminálmeghajtó. A terminálmeghajtó egy üzenetet kap minden billentyű lenyomásakor, és tárolja a leütött karakter billentyűkódját a várakozási sorban. Később ezeket értelmezi, és egy ASCII kódokat tartalmazó pufferben összeállítja, amelyet azután a felhasználói processzus területére átmásol*

használói processzusok csak a sendrec primitívet hajthatják végre, ami kombinálja a send-et egy receive-vel attól a processzustól, amelyhez elküldték az üzenetet.)

A fájlrendszer fogadja az üzenetet, és megkeresi azt az i-csomópontot (i-node), amely a megadott fájlleíróhoz tartozik. Ez az i-csomópont a */dev/console* karakterspecifikus fájlhoz tartozik és tartalmazza a terminál fő- és mellékeszközszámát. A terminálok főeszközszáma 4, a konzol mellékeszközszáma 0.

A fájlrendszer a *dmap* (device map) nevű eszköztérkép alapján keresi meg a TTY terminálmeghajtó számát. Ezután küld egy üzenetet a terminálmeghajtónak; ezt mutatja a (2) nyíl a 3.33. ábrán. Általában a felhasználó eddig az ideig még semmit sem gépelt be, így a terminálmeghajtó nem tudja a kérést kielégíteni. Azonnal visszaküld egy választ a fájlrendszernek, hogy felszabadítsa a blokkolt állapotból, és közli, hogy nincs felhasználható karakter; ezt mutatja a (3) nyíl. A fájlrendszer feljegyzi, hogy a processzus terminálbemenetre (azaz billentyűzetre) vár a konzolhoz tartozó *tty_table* struktúrában, és halad tovább, hogy a következő kérést feldolgozza. Természetesen a felhasználói parancsértelmező blokkolt marad, amíg a kért karakterek meg nem érkeznek.

Mikor végül leütik a karaktert a billentyűzeten, ez két megszakítást is okoz, egyet, amikor lenyomják, és egyet, amikor elengedik. Fontos tudni, hogy a PC-billentyűzet nem ASCII kódokat állít elő; minden egyes billentyű egy billentyűkódot (scan code) állít elő, amikor lenyomják, és egy másikat, amikor felengedik. Az alsó 7 bit a „lenyomás" és a „felengedés" billentyűkódjában azonos. A különbség a legmagasabb helyi értékű bitben van, ami 0, amikor a billentyűt lenyomták, és 1, amikor felengedték. Ugyanez érvényes a CTRL és SHIFT módosítóbillentyűkre. Bár, végül is ezek a billentyűk nem eredményeznek ASCII kódokat, amelyeket a felhasználói processzusnak vissza lehetne adni, hanem billentyűkódokat generálnak, amelyek megmutatják, hogy melyik billentyűt nyomták le (a meghajtó képes különbséget tenni a bal és a jobb oldali SHIFT billentyűk között, ha szükséges), és ezenkívül még billentyűkként két megszakítást okoznak.

A billentyűzetmegszakítás az IRQ 1. A megszakításvezeték nem érhető el a rendszersínen keresztül, és nem osztható meg semmilyen más I/O-kártyával. Amikor a *_hwint01* (6535. sor) meghívja az *intr_handle* (8221. sor) eljárást, nem kell ciklusok hosszú sorát bejárnia, hogy kitalálja, hogy a TTY-t kell értesíteni. A 3.33. ábrán bemutatjuk, hogy a rendszertaszk egy figyelmeztető üzenetet (4) küld, mivel a *system/do_irqctl.c* (nem szerepel a források között) fájlban lévő *generic_handler* generált egyet, de ezt az eljárást az alacsony szintű megszakításkezelő eljárások közvetlenül hívták. A rendszertaszkprocesszus nem aktiválódik. Amikor a *tty_task* (13740. sor) aktivizálja *kbd_interrupt*-ot (15335. sor), ami viszont a *scan_keyboard*-ot (15800. sor) hívja meg. A *scan_keyboard* végzi el az (5, 6, 7) kernelhívásokat, amelynek eredményeképpen a rendszertaszk olvas és ír néhány I/O-kapura, ami végül visszaadja a billentyűkódot, amelyet azután egy körkörös pufferben tárolnak. A *tty_events* jelzőt beállítják, amely mutatja, hogy a puffer karaktereket tartalmaz és nem üres.

Ettől a ponttól kezdve már nincs szükség üzenetre. Minden alkalommal, amikor a *tty_task* egy újabb ciklust kezd, megvizsgálja a *tty_events* jelzőt minden egyes terminálra, és minden olyan eszköz esetén, amelynek a jelzője be van állítva,

meghívja *handle_events*-et (14538. sor). A *tty_events* többféle tevékenységet is jelezhet (bár a beolvasás a legvalószínűbb), ezért *handle_events* mindig meghívja az eszközspecifikus függvényeket, mind a beolvasásra, mind a kiírásra. Billentyűzetről történő bevitel esetén ez a *kb_read* (15360. sor) meghívását eredményezi, amely nyomon követi azokat a billentyűkódokat, amelyek a CTRL, SHIFT és ALT gombok lenyomását és felengedését mutatják, és átkonvertálja a billentyűkódokat ASCII kódokká. A *kb_read* pedig hívja *in_process*-t (14486. sor), amely feldolgozza az ASCII kódokat, figyelembe véve a speciális karaktereket, az esetleg beállított különböző jelzőket, beleértve azt is, hogy a kanonikus mód érvényben van-e vagy nincs. Ennek hatása legtöbbször az, hogy a *tty_table*-ben szereplő konzol bemenő sorához hozzáadja a karaktereket, bár néhány kódnak, mint például a backspace, más hatása van. Alaphelyzetben szintén az *in_process* kezdeményezi az ASCII kódok echózását a képernyőre.

Amikor elegendő karakter érkezett be, a terminálmeghajtó egy újabb kernelhívást (8) hajt végre, amellyel megkéri a rendszertaszkot, hogy az adatokat a parancsértelmező által kért címre átmásolja. Ez a művelet nem üzenetküldés, ezért szaggatott vonalak jelzik (9) a 3.33. ábrán. Több mint egy ilyen vonal látható, mivel több ilyen művelet is történhet, amíg a felhasználó kérését teljesen kielégítik. Amikor a művelet végül befejeződik, a terminálmeghajtó egy üzenetet küld a fájlrendszernek, hogy a munka elkészült (10), és a fájlrendszer erre az üzenetre úgy reagál, hogy visszaküld egy üzenetet a parancsértelmezőnek, hogy feloldja annak blokkoltságát (11).

Annak definíciója, hogy mikor érkezett be elegendő számú karakter, a terminálmódtól függ. Kanonikus mód esetén a kérést akkor tekintik befejezettnek, amikor egy soremelés (linefeed), sorvége (end-of-line) vagy fájlvége (end-of-file) kód beérkezett, és hogy a bemenetet megfelelően feldolgozhassák, a beolvasott sor hossza nem haladhatja meg a bemeneti sor méretét. Nemkanonikus mód esetén a beolvasás sokkal nagyobb mennyiségű karaktert is kérhet, és az *in_process*-nek esetleg többször is kell karaktereket átvinnie, mielőtt egy üzenet érkezik visszza a fájlrendszerhez, ami jelzi, hogy a művelet befejeződött.

Jegyezzük meg, hogy a rendszertaszk az aktuális karaktereket a TTY címterületéről közvetlenül a parancsértelmezőére másolja át. Azok nem mennek keresztül a fájlrendszeren. Blokkos I/O esetén az adat keresztülhalad a fájlrendszeren, lehetővé téve a legutoljára használt blokkok egy gyorsítótárának létrehozását. Ha a kért blokk történetesen éppen a gyorsítótárban van, a kérést a rendszer azonnal ki tudja elégíteni, anélkül hogy lemez I/O történne.

Terminál I/O esetén a gyorsítótárnak nincs jelentősége. Továbbá a fájlrendszertől a lemezmeghajtóhoz érkező kérés minden esetben kielégíthető legfeljebb néhány száz ms alatt, így nem okoz bajt, ha a fájlrendszernek várnia kell. A terminál I/O befejezése akár órákat is igényelhet, vagy sosem ér véget (kanonikus módban a terminálmeghajtó egy teljes sorra várakozik, nemkanonikus módban szintén hosszú ideig várakozhat a *MIN* és *TIME* beállításától függően). Emiatt elfogadhatatlan lenne a fájlrendszer blokkolása addig, amíg egy terminál beolvasási kérést nem elégítettek ki.

Később előfordulhat, hogy a felhasználó már előre gépelt, és hogy a karakterek már rendelkezésre állnak, mielőtt kérték volna őket a korábbi megszakításokban vagy a 4 esemény bekövetkezésekor. Ebben az esetben az 1, 2 és az 5–11 események a beolvasási kérést követően gyors egymásutánban zajlanak le, a 3 esemény pedig egyáltalán nem fordul elő.

A MINIX-rendszer korábbi verzióit ismerő olvasók emlékezhetnek arra, hogy ezekben a verziókban a TTY (és minden más) meghajtót a kernellel együtt fordították le. Minden meghajtónak saját megszakításkezelője volt a kernel területén. A billentyűzetmeghajtó esetében a megszakításkezelő maga is tudott pufferelni néhány billentyűkódot, és el tudott végezni bizonyos előfeldolgozásokat (a legtöbb billentyűelengedés kódját ki lehet dobni, csak az olyan módosítóbillentyű, mint a SHIFT billentyűkódját kell megőrizni). Maga a megszakításkezelő nem küldött üzeneteket a TTY meghajtónak, mivel nagy a valószínűsége annak, hogy a TTY nem kerülne blokkolt állapotba egy receive hatására, és képes üzeneteket fogadni bármely adott időpontban. Ehelyett az órajelmegszakítás-kezelő ébresztette fel a TTY meghajtót periodikusan. Ezt a technikát alkalmazták, hogy elkerüljék a billentyűzetbemenet elvesztését.

Korábban bizonyos mértékben különbséget tettünk a várt megszakítások kezelése, mint amilyeneket a lemezvezérlő generál, és az olyan megjósolhatatlan megszakításkérések között, mint amilyenek a billentyűzetről érkeznek. Azonban a MINIX 3-rendszerben nincs különösebb teendő a megjósolhatatlan megszakításokkal kapcsolatban. Hogyan lehetséges ez? Egy fontos dologról nem szabad elfeledkeznünk, óriási teljesítménykülönbség van azok között a számítógépek között, amelyekre a MINIX-rendszer korábbi verzióit írták és a jelenlegi típusok között. A CPU órajelének sebessége megnőtt, míg az egy utasítás végrehajtásához szükséges órajelek száma lecsökkent. A MINIX 3-hoz legalább egy 80386-os processzor ajánlott. Egy lassú 80386-os nagyjából hússzor gyorsabban fogja végrehajtani az utasításokat, mint az eredeti IBM PC. Egy 100 MHz-es Pentium talán 25-ször olyan gyors, mint egy lassú 81386. Ezért talán a CPU-sebesség elegendő.

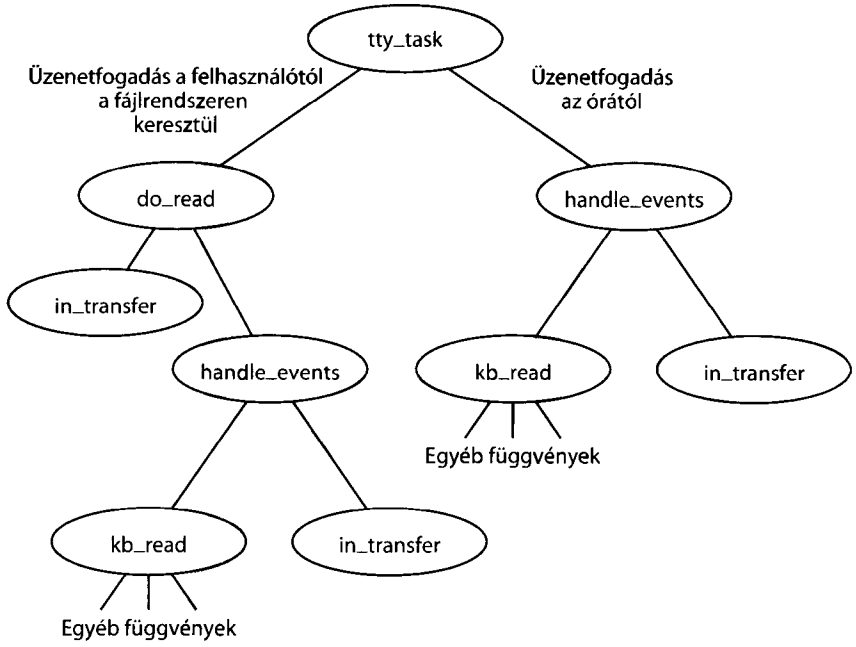
Egy másik dolog, amit figyelembe kell venni az, hogy a billentyűzetbemenet számítógépes szemmel nézve nagyon lassú. Percenként 100 szó sebességnél a gépelő kevesebb mint 10 karaktert üt le másodpercenként. Még a leggyorsabb gépelő esetén is a terminálmeghajtóhoz valószínűleg egy megszakításüzenet érkezik minden billentyűzeten leütött karakter esetén. Azonban más adatbevitcli eszközök esetén nagyobb adatsebesség valószínű – egy soros portra csatlakoztatott 56 000 bit/s sebességű modemről akár 1000-szerese vagy még nagyobb, mint a gépelőé. Ennél a sebességnél körülbelül 120 karaktert fogadhat a modem órajelenként, de hogy lehetővé váljon a modemes kapcsolaton az adattömörítés, a modemhez kapcsolt soros portnak legalább kétszer ennyit kell tudnia kezelni.

Még egy dolgot a soros porttal kapcsolatban figyelembe kell venni: karaktereket és nem billentyűkódokat továbbítanak, így még egy régi UART-tal is, amely nem végez pufferelést, csupán egy megszakítás történik billentyűleütésenként kettő helyett. Sőt az újabb PC-k már olyan UART-okkal vannak ellátva, amelyek 16, sőt akár 128 karaktert is pufferelnek. Így nincs szükség karakterenként egy megszakításra. Például egy 16 karakteres pufferrel rendelkező UART konfigurálható

lenne úgy, hogy csak akkor kérjen megszakítást, ha már 14 karakter van a pufferében. Az Ethernet-alapú hálózatok sokkal nagyobb sebességgel tudnak karaktereket továbbítani, mint egy soros vonal, de az Ethernet-kártyák egész csomagokat pufferelnek, és csak egy megszakításra van szükség csomagonként.

A terminálbemenetről szóló áttekintésünket azzal fejezzük be, hogy összegezzük azokat az eseményeket, amelyek akkor történnek, amikor a terminálmeghajtó először aktiválódik egy olvasási kérés eredményeként, és amikor ismét aktiválódik azután, hogy megkapta a billentyűzetről a bemenetet (lásd 3.34. ábra). Az első esetben, amikor egy üzenet érkezik a terminálmeghajtóhoz, amely a billentyűzetről kér karaktereket, a *tty_task* nevű főeljárás (13740. sor) hívja a *do_read*-et (13953. sor), hogy szolgálja ki a kérést. A *do_read* eltárolja a hívás paramétereit a *tty_table* billentyűzethez tartozó bejegyzései között, abban az esetben, ha a pufferben nincs kellő számú karakter a kérés kielégítésére.

Azután hívja az *in_transfer*-t (14416. sor), hogy megkaphassa a már várakozó bemenő adatokat, majd pedig a *handle_events*-et (14358. sor), amely viszont hívja [a *(*tp->tty_devread)* függvénymutatón keresztül] a *kb_read*-et (15360. sor) és újra az *in_transfer*-t azért, hogy megpróbáljon a bemenő folyamból még néhány karaktert kinyerni. A *kb_read* még számos más eljárást is hív, amelyek nem szerepelnek a 3.34. ábrán, hogy végrehajtsa a feladatát. Ennek az eredménye az, hogy



3.34. ábra. *A beolvasás kezelése a terminálmeghajtóban. A fa bal oldali ágát hajtják végre egy karakterolvasási kérés feldolgozásakor. A jobb oldali ágát pedig akkor, ha egy billentyűzetüzenet érkezik a meghajtóhoz azelőtt, hogy a felhasználó karaktereket kért volna*

bármi, ami azonnal rendelkezésre áll, azt átmásolják a felhasználóhoz. Ha semmi sem áll rendelkezésre, semmit sem másolnak. Ha az olvasási művelet véget ér vagy az *in_transfer,* vagy a *handle_events* hívásával, egy üzenetet továbbítanak a fájlrendszernek, amikor az összes karakter átvitele megtörtént; így a fájlrendszer felszabadíthatja a hívót a blokkolás alól. Ha az olvasási művelet nem fejeződött be (nincs vagy nincs elég karakter), a *do_read* visszajelez a fájlrendszernek, hogy fel kell-e függeszteni az eredeti hívót, vagy ha egy nem blokkoló olvasási kérésről van szó, akkor megszakítani az olvasást.

A 3.34. ábra jobb oldala összegzi azokat az eseményeket, amelyek akkor következnek be, ha a billentyűzetről érkező megszakítás következtében a terminálmeghajtó felébred. Amikor egy karaktert begépeltek, a *kbd_interrupt* (15335. sor) megszakítás-„kezelő" meghívja a *scan_keyboard*-ot, amely a rendszertaszkot hívja, hogy elvégezze az I/O-műveletet. (A „kezelőt" azért tettük idézőjelek közé, mert ez nem igazi megszakításkezelő, amelyet megszakításkérés esetén hívnak meg, ezt az eljárást egy üzenettel aktiválják, amelyet a *tty_task*-nak küldenek a rendszertaszkban lévő *generic_handler*-ből.) Ezután a *kbd_interrupt* beteszi a kapott karakterkódot az *ibuf* billentyűzetpufferbe, és beállít egy jelzőt, hogy a konzoleszközzel kapcsolatos esemény történt. Amikor a *kbd_interrupt* visszaadja a vezérlést a *tty_task*-nak, egy continue utasítás eredményeként a főciklus újabb iterációba kezd. Ez az üzenet beállítja, hogy az összes termináleszközt ellenőrizzék, és minden egyes megjelölt termináleszközre meghívódik a *handle_events*. A billentyűzet esetében a *handle_events* meghívja a *kb_read* és az *in_transfer* eljárásokat, éppen úgy, ahogyan ez az eredeti olvasási kérésnél történt. Az ábra jobb oldalán bemutatott események többször bekövetkezhetnek, amíg elegendő számú karakter nem érkezik, hogy a *do_read* által fogadott kérést ki lehessen elégíteni a fájlrendszertől kapott első üzenet után. Ha a fájlrendszer további karaktereket próbál meg kérni ugyanarról az eszközről, mielőtt az előző kérés befejeződött volna, hibaüzenetet kap. Természetesen minden eszköz független; egy távoli terminál felhasználója részéről érkező olvasási kérést elkülönítetten dolgozzák fel a konzolnál ülő felhasználóétól.

A *kb_read* által hívott, de a 3.34. ábrán be nem mutatott függvények között van a *map_key* (15303. sor), amely a hardver által generált billentyűkódokat (scan kódokat) ASCII kóddá konvertálja, a *make_break* (15431. sor), amely a módosítóbillentyűk (mint például a SHIFT) helyzetét követi nyomon, valamint az *in_process* (14486. sor), amely az olyan komplikációkat kezeli, mint amikor a tévedésből bevitt adatra a felhasználó visszatörléssel (backspace) megpróbál visszamenni, továbbá kezeli a többi speciális karaktert, valamint a különféle beviteli módokban hozzáférhető lehetőségeket. Az *in_process* hívja a *tty_echo*-t (14647. sor) is, így a begépelt karakterek a képernyőn is megjelennek.

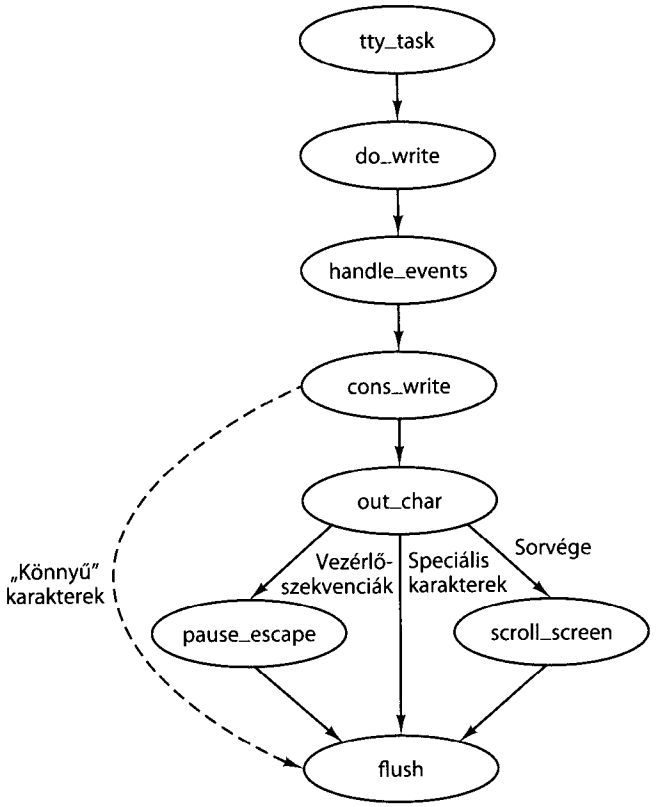
Terminálkimenet

Általában a konzolkimenet egyszerűbb, mint a terminálról történő adatfogadás, mert az operációs rendszer irányítja, és nem kell foglalkoznia kényelmetlen időpontokban érkező kiírási kérelmekkel. Minthogy a MINIX 3 konzolja egy tár-

címleképezéses képernyő, a kiírás a konzolra különösen egyszerű. Nem kellenek megszakítások, az alapművelet-adatok átmásolása egyik memóriaterületről a másikra. Másrészről a képernyő kezelésének összes részlete, beleértve a vezérlőszekvenciák feldolgozását is, a meghajtóprogramra vár. Ahogyan az előző részben a billentyűzetbemenet esetén tettük, most is végigkövetjük azokat a lépéseket, amelyek során a karakterek a konzol képernyőjére kerülnek. A példában feltételezzük, hogy az aktív képernyőre történik a kiírás, a virtuális konzolok okozta kisebb problémákat később tárgyaljuk.

Amikor egy processzus ki szeretne valamit írni, meghívja *printf*-et. A *printf* a write-ot hívja, hogy egy üzenetet küldjön a fájlrendszernek. Az üzenet egy mutatót tartalmaz a kiírandó karakterekre (nem magukat a karaktereket tartalmazza). A fájlrendszer egy üzenetet küld a terminálmeghajtónak, amely kiolvassa és bemásolja őket a videó RAM-ba. A 3.35. ábra mutatja a kiírásban részt vevő fontosabb eljárásokat.

Amikor egy üzenet érkezik a terminálmeghajtóhoz, hogy írjon a képernyőre, a *do_write*-ot (14029. sor) hívja meg, hogy tárolja el a paramétereket a konzolhoz



3.35. ábra. *Terminálkimenetnél használt főbb eljárások. Szaggatott vonal jelöli azt, amikor a karaktereket a* cons_write *egyenesen a* ramqueue*-ba másolja*

tartozó *tty* struktúrában, a *tty_table* tömb egyik elemében. Ezután a *handle_events* hívása következik (ugyanennek a függvénynek a hívása történik, ha a *tty_events* jelző be van állítva). Ez a függvény minden esetben meghívja a paraméterében kiválasztott eszköz kiíró és beolvasó rutinjait is. A konzol képernyője esetén ez azt jelenti, hogy ha van várakozó bemenő adat a billentyűzetről, akkor először azt dolgozzák fel. Ha van várakozó bemenő adat, az echózandó karakterek hozzáadódnak az esetleg már régebb óta kiírásra várakozó adatokhoz. Ezután következik a *cons_write* hívása (16036. sor), ami a tárcímleképezéses képernyő kiíró eljárása. Ez az eljárás a *phys_copy*-t használja, hogy karakterblokkokat másoljon a felhasználói processzus területéről egy lokális pufferbe, ezt és az ezt követő néhány lépést esetleg néhányszor megismétli, mivel a lokális puffer csak 64 bájtot tárol. Amikor a lokális puffer megtelik, minden egyes 8 bites bájt egy másik pufferbe, a *ramqueue*-ba kerül. Ez egy 16 bites szavakból álló tömb. A szavak másik bájtjába a képernyőattribútum-bájt aktuális értéke kerül, amely meghatározza a háttér- és előtérszíneket és más jellemzőket. Amikor lehetséges, a karakterek közvetlenül a *ramqueue*-ba kerülnek, de bizonyos karakterek, mint például a vezérlőkarakterek vagy a vezérlőszekvenciákat alkotó karakterek speciális kezelést igényelnek. Speciális kezelésre akkor is szükség van, amikor a karakter képernyő-pozíciója meghaladja a képernyő szélességét, vagy ha a *ramqueue* betelik. Ezekben az esetekben az *out_char* (16119. sor) kerül meghívásra, hogy továbbítsa a karaktereket, és végrehajtsa a szükséges egyéb tevékenységeket. Például a *scroll_screen*-t (16025. sor) hívja akkor, ha a képernyő utolsó sorában való kiírás közben soremelés karakter érkezik, és a *parse_escape* dolgozza fel a vezérlőszekvenciák karaktereit. Rendszerint az *out_char* hívja a *flush*-t (16259. sor), amely a *ramqueue* tartalmát átmásolja a videokártya-memóriába, az assembly nyelven írt *mem_vid_copy* rutint felhasználva. A *flush* hívódik meg az utolsó karakter a *ramqueue*-ba való átvitele után is, hogy biztosan megjelenjen minden kiírás. A *flush* tevékenységének utolsó eredménye egy parancs kiadása a 6845-ös videovezérlő áramkörnek, hogy a kurzort állítsa a helyes pozícióba.

Logikailag megfelelő lenne a felhasználói processzus területéről elhozott bájtokat egy ciklusban egyesével beírni a videó RAM-ba. Azonban a karakterek összegyűjtése a *ramqueue*-ba és egy egész blokk átmásolása a *mem_vid_copy* hívásával hatékonyabb a Pentium osztályú processzorok védett (protected) memóriájú környezetében. Érdekes módon ez a technika már a MINIX 3 korai változatainál alkalmazásra került, amelyek régebbi típusú, védett memória nélküli processzorokon futottak. A *mem_vid_copy* előfutára egy ütemezési problémával küszködött – a régebbi típusú képernyők esetén a videomemóriába akkor kellett írni, amikor a képernyő üresjáratban volt, azaz amíg a CRT sugárnyaláb függőleges irányban alulról a legfelső sorba futott vissza, hogy elkerülhető legyen mindenféle szemét megjelenése a képernyőn. A MINIX 3 már nem gondoskodik ilyen elavult berendezések támogatásáról, mivel a hatékonyságot nagymértékben rontotta volna, de a MINIX 3 modern változata is hasznot húz más módon a *ramqueue* blokként történő másolásából.

A konzol számára elérhető videó RAM memória határait a *console* struktúrában szereplő *c_start* és *c_limit* határozzák meg. Az aktuális kurzorpozíciót a

c_column és *c_row* mezők tárolják. A (0, 0) koordináta a képernyő bal felső sarka, ahonnan a hardver elkezdi a képernyő feltöltését. A videomegjelenítés a *c_org* által megadott címtől kezdődik és 80 × 25 karakteren keresztül (4000 bájt) folytatódik. Más szavakkal a 6845-ös mikroáramkör kiolvassa a videó RAM *c_org* által meghatározott szavát, és megjeleníti a karakterbájtot a képernyő bal felső sarkában, felhasználva az attribútumbájtot a színek, a villogás stb. vezérlésére. Ezután kiolvassa a következő szót, és megjeleníti a karaktert az (1, 0) pozícióban. Ez folytatódik, amíg elér a (79, 0) pozícióba, amikor elkezdi a képernyő második sorát a (0, 1) pozícióban.

Amikor a számítógépet először elindítják, a képernyő törlődik, a kimenő adatok a *c_start* címtől kezdve kerülnek be a videó RAM-ba, a *c_org*-hoz pedig ugyanaz az érték rendelődik, mint a *c_start*-hoz. Az első sor így a képernyő legtetején jelenik meg. Amikor a kiírást új sorban kell folytatni, akár azért, mert az első sor megtelt, akár azért, mert az *out_char* soremelés karaktert érzékelt, a kiírandó adatok a *c_start* plusz 80 címre kerülnek. Végül mind a 25 sor betelik, és a képernyőt **görgetni** kell. Néhány program, például a szövegszerkesztők, a lefelé görgetést is megkívánják, amikor a kurzor a képernyő felső sorában van, és a szövegben előre kell haladni.

A képernyőgörgetést két módon lehet megvalósítani. **Szoftveres görgetés** esetén a (0, 0) pozícióra kerülő karakter mindig a videomemória elején található, a *c_start* által mutatott címhez képest a 0 címen, és a videovezérlő áramkör úgy van programozva, hogy erről a címről vegye az első adatot olyan módon, hogy a *c_org* értéke is ugyanezt a címet tartalmazza. Amikor a képernyőt görgetni kell, a videó RAM 80-as relatív című helyének tartalmát, azaz a második sor elejét a 0-s relatív címre másolják át, a 81-es című szót az 1-es című helyre, és így tovább. A megjelenítés rendje változatlan, a 0-s címre tett adat a képernyő (0, 0) pozícióján jelenik meg, és a képernyőn megjelenő kép látszólag egy sorral feljebb mozdul. Ennek az ára az, hogy a CPU 80 × 24 = 1920 szót mozgatott. **Hardveres görgetés** esetén az adatokat nem mozgatják a memóriában, ehelyett a videovezérlő lapkát utasítják arra, hogy máshonnan kezdje a képernyő megjelenítését, például a 80-as címen tárolt szótól. Ennek lekezelése úgy történik, hogy *c_org* tartalmához 80-at hozzáadnak, ezt az értéket elmentik a későbbi felhasználás érdekében, és beírják a videovezérlő mikroáramkör megfelelő regiszterébe is. Ehhez vagy az szükséges, hogy a vezérlő elég okos legyen ahhoz, hogy a RAM-ot körkörösen kezelje, azaz vegye ismét az adatokat a videomemória elejéről (a *c_start*-ban szereplő címről), amikor elér a RAM végére (a *c_limit*-ben lévő címre), vagy az, hogy a videó RAM nagyobb kapacitású legyen, mint 80 × 25 = 2000 szó, ami egy képernyőnyi adat tárolásához elegendő.

A régebbi képernyőillesztők általában kisebb memóriával rendelkeztek, de képesek voltak a RAM-ot körkörösen kezelni, így hardveres görgetést végezni. Az újabb illesztők rendszerint jóval nagyobb memóriával rendelkeznek, mint ami egy képernyőnyi szöveg megjelenítéséhez szükséges, de vezérlőjük nem képes a RAM körkörös kezelésére. Így egy 32 768 bájt képernyőmemóriával rendelkező képernyőillesztő 204 teljes, 160 bájtos sort tárolhat, így 179 hardveres görgetésre képes, mielőtt a körbefordulásra való képtelensége problémát okozna. De azután szük-

Mezőnév	Tartalom
c_start	A konzol videomemóriabeli kezdőcíme
c_limit	A konzol videomemóriabeli végcíme
c_column	Aktuális oszlop (0–79), 0 a bal oldalon
c_row	Aktuális sor (0–24), 0 a legfelső sor
c_cur	Kurzor videó RAM-beli offset címe (eltolás)
c_org	RAM-beli hely, amelyre a 6845-ös bázisregisztere mutat

3.36. ábra. *A* console *struktúra mezői, amelyek az aktuális képernyő-pozíciókra vonatkoznak*

ség lesz az utolsó 24 sor adatának átmásolására vissza a videomemória 0-s címére. Bármelyik módszert használják, egy üres sort is kell a videó RAM-ba másolni, amely biztosítja azt, hogy a képernyő alján lévő új sor üres legyen.

Amikor virtuális konzolok használata lehetséges, a videoillesztő memóriáját azonos méretű részekre osztják a kívánt számú konzol számára, és minden egyes konzolnál megfelelően beállítják a *c_start* és a *c_limit* változókat. Ennek a görgetésre is van hatása. Egy virtuális konzolokat is támogatni képes, elég nagy memóriával rendelkező illesztő esetén, szoftveres görgetés történik időről időre, bár hivatalosan a hardveres görgetés van érvényben. Minél kisebb az a memória, amely az egyes konzolok képernyője számára használható, annál gyakoribb, hogy szoftveres görgetést kell használni. A felső határt akkor érik el, amikor a maximális számú lehetséges virtuális konzolt konfigurálják. Ekkor minden görgető művelet szoftveres görgetés lesz.

A kurzornak a videó RAM kezdetéhez viszonyított relatív pozíciója a *c_column* és a *c_row* alapján határozható meg, de gyorsabb, ha közvetlenül tároljuk (a *c_cur* mezőben). Amikor egy karaktert ki kell írni, akkor azt a videó RAM a *c_cur* címére helyezik el, majd a *c_cur*-t ugyanúgy, mint a *c_column*-t, módosítják. A 3.36. ábra összefoglalja a *console* struktúra mezőit, amelyek befolyásolják a kurzor pozícióját és a képernyő memóriabeli kezdőcímét.

Azoknak a karaktereknek a feldolgozása, amelyek a kurzor pozíciójára hatással vannak (például soremelés, visszatörlés), a *c_column,* a *c_row* és a *c_cur* értékének megfelelő beállításával történik. Ez a *flush* végén történik a *set_6845* meghívásával, amely beállítja a videovezérlő lapka regisztereit.

A terminálmeghajtó támogatja a vezérlőszekvenciákat, amivel lehetővé teszi, hogy a szövegszerkesztő és más interaktív programok rugalmasan frissítsék a képernyőt. A támogatott szekvenciák, amelyek az ANSI szabvány egy részhalmazát alkotják, így megfelelnek annak az igénynek, hogy a más hardver és operációs rendszer számára fejlesztett programok könnyen átvihetők legyenek a MINIX 3-ba. Két csoportja van a vezérlőszekvenciáknak; az egyikbe azok tartoznak, amelyek sohasem tartalmaznak változó paramétert, a másikba pedig azok, amelyek tartalmazhatnak ilyet. Az első típusból csak egyet támogat a MINIX 3: ez az ESC M, amely visszafelé járja be a képernyőt, azaz feljebb viszi a kurzort egy sorral, és lefelé görgeti a képernyőt, ha a kurzor már a legfelső sorban van. A második csoportba tartozóknak egy vagy két numerikus paramétere lehet. Az ebbe

a csoportba tartozó szekvenciák mind az ESC [karakterekkel kezdődnek. A „[" a **vezérlősorozat-bevezető** jel. A 3.32. ábrán bemutattuk azokat az ANSI szabvány alapján definiált szekvenciákat, amelyeket a MINIX 3 felismer.

A vezérlőszekvenciák elemzése nem egyszerű dolog. A MINIX 3-ban az érvényes vezérlőszekvenciák lehetnek csupán 2 karakteresek, mint az ESC M, vagy elérhetik a 8 karakteres hosszúságot, abban az esetben, ha a szekvenciának két numerikus paramétere van, és mindegyik értéke egy-egy kétjegyű szám lehet, mint például az ESC [20;60H, amely a kurzort a 20. sor 60. oszlopába viszi. Egy olyan szekvencia esetén, amelynek egy paramétere van, a paraméter elhagyható; vagy ha két paramétere van, elhagyható bármelyik vagy mindkettő. Ha egy paramétert elhagytak, vagy olyan értéket adtak meg, amely az érvényes tartományon kívül esik, akkor egy alapérték kerül a helyére. Az alapérték a legkisebb megengedett érték.

Tekintsük a következő módszereket arra, ahogyan egy program összeállíthat egy olyan szekvenciát, amely a kurzort a képernyő bal felső sarkába mozgatja.

1. ESC [H elfogadható, hisz nincsenek paraméterek, így a legkisebb érvényes értékekkel hajtódik végre.
2. ESC [1;1H a kurzort az 1. sor 1. oszlopába küldi. (Az ANSI szabványban a sorok és oszlopok számozása 1-től kezdődik.)
3. Mind az ESC [1;H, mind az ESC [;1H szekvenciában hiányzik az egyik paraméter, amelynek 1 a feltételezett értéke, ahogy az első példában láttuk.
4. ESC [0;0H hatása ugyanaz, hiszen mindkét paraméter értéke kisebb, mint a legkisebb érvényes érték, így az kerül a helyükre.

Ezek a példák nem azt kívánják elősegíteni, hogy az ember szabadon használhat helytelen paramétereket, hanem azt, hogy az a kód, amely ezeket a szekvenciákat elemzi, egyáltalán nem egyszerű.

A MINIX 3 ezt az elemzést egy véges állapotú automatával valósítja meg. A *console* struktúrában lévő *c_esc_state* változó normális körülmények között 0 értékű. Amikor az *out_char* felfedez egy ESC karaktert, a *c_esc_state* értékét 1-re változtatja, és a következő karaktereket a *parse_escape* (16293. sor) dolgozza fel. Ha a következő karakter a vezérlősorozat bevezető jele, akkor 2-re állítja be az állapotot *(c_esc_state)*, egyébként a szekvenciát befejezettnek tekinti, és meghívja a *do_escape*-et (16352. sor). A 2-es állapotban, amíg numerikus karakterek érkeznek, a paraméter előző értékét (kezdetben 0) 10-zel szorozza, és hozzáadja az aktuális karakter numerikus értékét. A paraméterértékek egy tömbben tárolódnak, és amikor egy pontosvesszőt érzékel, a feldolgozás a tömb következő elemére lép. (Jóllehet, a MINIX 3-ban a tömbnek csak két eleme van, de az elv ugyanez.) Amikor egy nem numerikus karaktert talál, ami nem is a pontosvessző, a szekvenciát befejezettnek tekinti, és szintén a *do_escape*-et hívja. A *do_escape*-be lépéskor az aktuális karaktert használják arra, hogy kiválasszák pontosan, milyen tevékenységet kell végezni, és hogy miként értelmezzék a kapott paramétereket, akár az alapértékeket, akár azokat, amelyek a karakterfolyamban érkeztek. Ezt mutatja a 3.44. ábra.

Betölthető billentyűzettérképek

Az IBM-billentyűzet nem közvetlenül ASCII kódokat állít elő. A billentyűket egy számmal azonosítják, kezdve azokkal a billentyűkkel, amelyek az eredeti PC-billentyűzet bal felső részén voltak – az 1-es az ESC billentyű, a 2-es az „1" (szám), és így tovább. Minden billentyűhöz egy szám tartozik, beleértve a módosító billentyűket is, mint a bal és a jobb SHIFT billentyűk, a 42 és az 54 számok. Amikor egy billentyűt lenyomnak, a MINIX 3 megkapja a billentyű sorszámát, mint billentyűkódot (scan kód). Egy billentyűkód generálódik akkor is, amikor egy billentyűt felengednek, de a felengedéskor előállított kódban a legmagasabb helyi értékű bit 1-es lesz (ami ugyanaz, mintha 128-at adnánk a billentyű sorszámához). Így a billentyű lenyomása és felengedése megkülönböztethető. Annak nyomon követésével, hogy mely módosítóbillentyűket nyomtak le, de még nem engedtek fel, nagyszámú billentyűkombináció lehetséges. Általános célokra természetesen az olyan kétujjas kombinációk a legmegfelelőbbek a két kézzel gépelőknek, mint a SHIFT-A vagy a CTRL-D, de speciális alkalmakra a három (vagy több) billentyűből álló kombinációk is elképzelhetők, mint például a CTRL-SHIFT-A vagy a jól ismert CTRL-ALT-DEL, amelyet a PC-felhasználók arról ismernek, hogy ezzel a rendszer alapállapotba hozható és újratölthető.

A PC-billentyűzet összetettsége nagyfokú rugalmasságot biztosít abban, hogy miként használják. Egy szabványbillentyűzeten 47 közönséges billentyű található (26 alfabetikus, 10 numerikus, 11 írásjel). Ha a három módosítóbillentyűs kombinációt is használni akarjuk, mint a CTRL-ALT-SHIFT, akkor egy 376 (8 × 47) karakterből álló készletet tudunk támogatni. És ez még nem a lehetőségek határa, de egyelőre tegyük fel, hogy nem akarjuk megkülönböztetni a jobb és bal oldali módosítóbillentyűket, vagy a numerikus billentyűzetet, vagy a funkcióbillentyűk valamelyikét használni. Valójában nem csak a CTRL, ALT és SHIFT használható módosítóbillentyűként, nyugdíjba küldhetünk néhányat a közönséges billentyűk közül, és módosítóbillentyűkként használhatjuk őket, ha úgy döntünk, hogy egy ilyen rendszert támogató meghajtót írunk.

Az ilyen billentyűzettel rendelkező operációs rendszerek egy **billentyűzettérképet (keymap)** használnak, hogy meghatározzák, milyen kódot kell továbbküldeniük egy programnak, attól függően, hogy melyik billentyűt nyomták meg, és hogy milyen módosítók vannak érvényben. A MINIX 3 billentyűzettérképe logikailag egy 128 sorból, amelyek a lehetséges billentyűkódokat képviselik, és 6 oszlopból álló tömb (ezt a méretet úgy választották meg, hogy alkalmazkodjon a japán billentyűzet méretéhez; az USA és más európai billentyűzetek nem tartalmaznak ilyen sok billentyűt). Az oszlopok képviselik a módosító nélküli, a SHIFT, a CTRL, a bal ALT, a jobb ALT és az ALT-SHIFT kombinációk valamelyikét. Ezzel a sémával 732 [(128–6) × 6] karakterkód állítható elő, megfelelő billentyűzet esetén. Ez megköveteli, hogy a táblázatban szereplő adatok 16 bites értékek legyenek. Az amerikai billentyűzet számára az ALT és ALT2 oszlopok megegyeznek. Az ALT2-t ALT GR-nek hívják más nyelvek billentyűzetein, és ezek a billentyűzettérképek olyan billentyűket is támogatnak, amelyeken 3 jel van; ezeknél az ALT GR-t használják módosítóbillentyűként.

A szabvány-billentyűzettérkép – amelyet a következő sor jelöl ki:


```
#include keymaps/us-std.src
```

a *keyboard.c* állományban –, fordításkor kerül a MINIX 3-kernelbe, de egy

```
ioctl(0, KIOCSMAP, keymap)
```

hívás használható arra, hogy egy másik billentyűzettérképet töltsenek be a kernelbe, a *keymap* címre. Egy teljes billentyűzettérkép 1536 bájtot foglal el (128 × 6 × 2). A kiegészítő billentyűzettérképek tárolása tömörített formában történik. A *genmap* programmal készíthető egy új tömörített billentyűzettérkép. Fordításkor a *genmap* beilleszti az adott billentyűzethez tartozó *keymap.src* kódot, így a térkép a *genmap*-be belefordítódik. Normális esetben fordítás után a *genmap*-et azonnal lefuttatják, az egy állományba kiírja a tömörített billentyűzettérképet, majd a bináris *genmap*-et törlik. A *loadkeys* parancs beolvas egy tömörített billentyűzettérképet, a belsejében kicsomagolja, majd meghívja az ioctl-t, hogy az áthelyezze a kernel memóriájába. A MINIX 3 indításkor automatikusan végrehajthatja a *loadkeys* parancsot, de a programot a felhasználó is bármikor lefuttathatja.

A billentyűzettérkép forráskódja egy nagy, kezdőértékekkel feltöltött tömb. A 3.37. ábra az *src/kernel/keymaps/us-std.src* állományból mutat be néhány sort táblázatos elrendezésben, ami bemutatja a térkép néhány adottságát. Az IBM PC-billentyűzeten nincs olyan billentyű, amely a 0 billentyűkódot generálja. Az 1-es kód, az ESC billentyű sorában szereplő adatok azt mutatják, hogy a visszaadott érték változatlan a SHIFT vagy a CTRL billentyű esetén, de más a kód az ALT és az ESC együttes lenyomásakor. A fordításkor az egyes oszlopokba kerülő értékeket az *include/minix/keymap.h* állományban definiált makrók határozzák meg:

```
#define C(c)    ((c) & 0x1F)        /* Kontrollkód maszkja */
#define A(c)    ((c) | 0x80)        /* Nyolcadik bit beállítása (ALT) */
#define CA(c)   A(C(c))             /* CTRL-ALT */
#define L(c)    ((c) | HASCAPS)     /* A „Caps Lock befolyásolja" attribútum */
```

Billentyűkód	Karakter	Normál	SHIFT	ALT1	ALT2	ALT-SHIFT	CTRL
100	Nincs	0	0	0	0	0	0
101	esc	C('[')	C('[')	CA('[')	CA('[')	CA('[')	C('[')
102	'1'	'1'	'!'	A('1')	A('1')	A('!')	C('A')
113	'='	'='	'+'	A('=')	A('=')	A('+')	C('@')
116	'q'	L('q')	'Q'	A('q')	A('q')	A('Q')	C('Q')
128	CR/LF	C('M')	C('M')	CA('M')	CA('M')	CA('M')	C('J')
129	CTRL	CTRL	CTRL	CTRL	CTRL	CTRL	CTRL
159	F1	F1	SF1	AF1	AF1	ASF1	CF1
127	???	0	0	0	0	0	0

3.37. ábra. *A billentyűzettérképet tartalmazó forráskód néhány adata*

Az első három makró ezek közül a megadott karakter kódjának a bitjeit manipulálja, hogy előállítsa az alkalmazás számára szükséges kódot. Az utolsó makró a HASCAPS bitet állítja 1-re a 16 bites érték magas helyi értékű bájtjában. Ez egy jelző, amely azt mutatja, hogy ellenőrizni kell a *capslock* változó állapotát, és valószínűleg módosítani kell a kódot annak megfelelően a visszatérés előtt. Az ábrában a 2, 13 és 16 billentyűkódokhoz tartozó sorok azt mutatják be, hogy egy tipikus numerikus, írásjel és alfabetikus billentyű kezelése hogyan történik. A 28-as kód esetén egy speciális tulajdonság látható – az ENTER billentyű alaphelyzetben a CR kódot (0x0D) állítja elő, amelyet itt a C('M') reprezentál. Mivel azonban a Unix-állományokban a soremelés karakter az LF (0x0A) kód, és esetenként szükség lehet ennek közvetlen bevitelére, a billentyűzettérkép erre a CTRL-ENTER kombinációt biztosítja, ami előállítja ezt a kódot, C('J').

A 29-es billentyűkód az egyik módosítóbillentyűt jelenti, és úgy kell felismerni, hogy nem számít, milyen másik billentyű van lenyomva, így tehát a CTRL értéket adja vissza, figyelmen kívül hagyva bármely más lenyomott billentyűt. A funkcióbillentyűk nem eredményeznek rendes ASCII értékeket, és a táblázatnak az 59-es billentyűkódhoz tartozó sora mutatja szimbolikusan azokat az értékeket (az *include/minix/keymap.h* állományban vannak definiálva), amelyeket az F1 billentyűnek különböző módosítóbillentyűkkel kombinált lenyomása állít elő. Ezek a következők: F1: 0x0110, SF1: 0x1010, AF1: 0x0810 ASF1: 0x0C10 és CF1: 0x0210. Az ábra utolsó, a 127-es billentyűkódhoz tartozó sora tipikusan egy a táblázat végének közelében található sorok közül. Sok billentyűzet számára – legtöbbjüket Európában és az amerikai kontinenseken használják – nincs elég billentyű, hogy az összes lehetséges billentyűzetkód előállítható legyen, ezért a táblázatban a hiányzó billentyűkhöz tartozó bejegyzések nullákkal vannak feltöltve.

Betölthető betűkészletek

A régebbi PC-típusok a képernyőre kerülő karakterek generáláshoz a mintákat csupán egy ROM-ban tárolták, de a modern rendszerekben használt megjelenítő egy RAM-ot biztosít a videokártyán, amelybe egyedileg konfigurálható karaktermináták tölthetők be. Ezt a MINIX 3-ban az

```
ioctl(0, TIOCSFON, font)
```

ioctl művelet végzi. A MINIX 3 egy 80 oszlop × 25 soros videó módot támogat, a betűkészlet-állományok 4096 bájtot tartalmaznak. Minden bájt egy 8 képpontból (pixel) álló sort reprezentál. A képpont világít, ha a bit értéke 1, és 16 ilyen sor kell minden egyes karakter leírásához. Azonban a videoillesztő 32 bájtot használ minden egyes karakter leírásához, hogy nagyobb felbontást biztosítson azokban a videomódokban, amelyeket a MINIX 3 jelenleg nem támogat. A *loadfont* utasítás szolgál arra, hogy konvertálja ezeket a fájlokat abba a 8192 bájtos *font* struktúrába, amelyre az ioctl hívás hivatkozik, és ezt az eljáráshívást használja a betűkészlet betöltésére. Azonban minden videoillesztőnek van egy alapértelmezés szerinti

betűkészlete, amely a ROM-jába van beégetve, ami alaphelyzetben rendelkezésre áll. Nem szükséges a MINIX 3-ba magába befordítani egy betűkészletet, az egyetlen betűkészlet-támogatás, amelyre a kernelben szükség van, az az a kód, amely végrehajtja a *TIOCSFON* ioctl műveletet.

3.8.4. Az eszközfüggetlen terminálmeghajtó implementációja

Ebben a fejezetben a terminálmeghajtó program forráskódját fogjuk részletesen elemezni. Amikor a blokkos I/O-eszközöket tanulmányoztuk, láttuk, hogy több, néhány eszközt támogató különböző meghajtóprogram alapulhat egy közös szoftveren. A terminálleszközök esetében a helyzet hasonló, azzal a különbséggel, hogy egy terminálmeghajtó van, amely támogat több különféle terminálleszközt. Az eszközfüggetlen kóddal fogjuk kezdeni. A későbbi fejezetekben pedig a billentyűzet és a tárcímleképezéses konzolmegjelenítő eszközfüggő kódját fogjuk áttekinteni.

A terminálmeghajtó adatstruktúrája

A *tty.h* állomány olyan definíciókat tartalmaz, amelyeket a terminálmeghajtót implementáló C fájlok használnak. Mivel ez a meghajtó több különböző eszközt is támogat, ezért a mellékeszközszámot kell arra használni, hogy megkülönböztessék az egyes eljáráshívásokban melyik támogatott eszköz fordul elő; ezek a 13405 és a 13409 sorok között vannak definiálva.

A *tty.h* állományban található *O_NOCTTY* és *O_NONBLOCK* jelzők definíciói (ezek az open hívás opcionális argumentumai), amelyek az *include/fcntl.h* fájlban található definíciók duplikációi, megismétlésük azért történt, hogy ne legyen szükség egy másik fájl beillesztésére. A *devfun_t* és a *devfunarg_t* típusokat (13423. és 13424. sor) általában függvények címét tartalmazó mutatókként definiálják, hogy lehetőséget adjanak egy olyan indirekt hívási mechanizmusra, amely hasonlít ahhoz, amit a lemezmeghajtók főciklusának kódjában láttunk.

Az ebben a fájlban deklarált változók közül soknak az azonosítója a *tty_* prefixszel kezdődik. A legfontosabb definíció a *tty.h* állományban a *tty* struktúra (13426–13488. sor). Minden egyes terminálleszközre van egy ilyen adatstruktúra (a konzolmegjelenítő és a billentyűzet egyetlen terminálnak számít). A *tty* struktúrában az első változó a *tty_events;* ez egy jelző, amely akkor van beállítva, ha egy megszakítás olyan változást okoz, ami igényli, hogy a terminálmeghajtó odafigyeljen erre az eszközre.

A *tty* struktúra többi része úgy van szervezve, hogy egy csoportba gyűjti azokat a változókat, amelyek a beolvasással, kiírással, az eszköz állapotával és a befejezetlen műveletekkel kapcsolatosak. A beolvasással foglalkozó részben a *tty_inhead* és a *tty_intail* definiálja azt a sort, ahol a beérkezett karaktereket gyűjtik. A *tty_incount* számlálja az ebben a sorban lévő karakterek számát, a *tty_eotct* pedig karaktereket vagy sorokat számlál a következők szerint. Minden eszközspecifikus hívás indirekten történik, azokat a rutinokat kivéve, amelyek inicializálják a terminált. Ezek ál-

lítják be az indirekt hívásokhoz használt mutatókat. A *tty_devread* és a *tty_cancel* mezők olyan eszközspecifikus eljárásokra mutatnak, amelyek az olvasás és a bevitel műveleteket hajtják végre. A *tty_min* változót a *tty_eotct*-vel történő összehasonlításokban használják. Amikor az utóbbi egyenlővé válik az elsővel, egy olvasási művelet véget ér. Kanonikus beolvasás esetén a *tty_min* 1-re van állítva, és a *tty_eotct* a bevitt sorokat számlálja. Nemkanonikus beolvasás alatt a *tty_eotct* a karaktereket számlálja, a *tty_min* pedig a *termios* struktúra *MIN* mezőjének az értékét veszi fel. A két változó összehasonlítása így megadja, hogy mikor van készen egy sor, vagy mikor értük el a minimális karakterszámot az aktuális módtól függően. A *tty_tmr* egy időzítő ehhez a *tty* struktúrához, a *termios TIME* mezőjéhez használják.

Mivel a kimeneti sort eszközspecifikus kód kezeli, a *tty* struktúra kimenettel foglalkozó része nem tartalmaz változókat, kizárólag mutatókból áll, amelyek az olyan eszközspecifikus függvényekre mutatnak, amelyek kiírnak, echóznak, megszakítójelet küldenek és megszakítják a kiírást. Az állapotrészben a *tty_reprint*, a *tty_escaped* és a *tty_inhibited* jelzők találhatók, amelyek azt jelzik, ha az utolsó beolvasott karakter speciális jelentéssel bír, például amikor egy CTRL-V (LNEXT) karakter érkezett be, a *tty_escaped* értékét 1-re állítják, jelezve, hogy a következő karakternek bármilyen speciális jelentését figyelmen kívül kell hagyni.

Az adatszerkezet következő része a folyamatban lévő *DEV_READ, DEV_WRITE* és *DEV_IOCTL* műveletekről tartalmaz adatokat. E műveletek mindegyikében két processzus vesz részt. A rendszerhívásokat kezelő kiszolgáló azonosítóját (ez általában az FS), a *tty_incaller* (13458. sor) tartalmazza. A kiszolgáló egy másik processzus nevében hívja meg a terminálmeghajtót, amelynek egy I/O-műveletre van szüksége; ennek a kliensnek az azonosítója található a *tty_inproc*-ban (13459. sor). Ahogyan a 3.33. ábra mutatja, egy read során a karakterek a terminálmeghajtó területéről egyenesen az eredeti hívó memóriaterületén lévő pufferbe kerülnek. A *tty_inproc* és *tty_in_vir* határozzák meg a puffer helyét. A következő két változó, a *tty_inleft* és a *tty_incum* számon tartják a még szükséges és a már átvitt karakterek számát. Hasonló változók szükségesek a write rendszerhíváshoz is. Az ioctl számára lehetséges azonnali adatátvitel a kérő processzus és a meghajtó között, ezért szükség van egy virtuális címre, viszont nincs szükség olyan változókra, amelyek egy művelet folyamatát követik. Egy ioctl kérést el lehet halasztani, például amíg a folyamatban lévő kiírás befejeződik, de amikor megfelelő az időpont, a kérést egyetlen művelettel hajtják végre.

Végül a *tty* struktúra tartalmaz néhány olyan változót, amely nem tartozik egyik kategóriába sem, idetartoznak az olyan függvények címét tartalmazó mutatók, amelyek a *DEV_IOCTL* és *DEV_CLOSE* műveleteket eszközszinten kezelik, a POSIX stílusú *termios* struktúra és a *winsize* struktúra, amely támogatást nyújt ablakorientált képernyős megjelenítésekhez. Az adatszerkezet utolsó része tárhelyet biztosít magának a bemenő adatokat tartalmazó sornak a *tty_inbuf* tömbben. Figyeljük meg, hogy ez a tömb *u16_t* típusú elemekből áll, nem pedig 8 bites *char* karakterekből. Bár az alkalmazások és az eszközök 8 bites kódot használnak a karakterekhez, a C nyelv megköveteli, hogy a *getchar* beolvasó függvény egy nagyobb adattípuson legyen definiálva, így egy szimbolikus *EOF* értéket adhat vissza az összes 256 lehetséges bájtértéken felül.

A *tty_table,* amely egy *tty* struktúrákból álló tömb, *extern*-ként van deklarálva (13491. sor). Minden egyes, az *include/minix/config.h* található *NR_CONS, NR_RS_LINES* és *NR_PTYS* definíciókkal engedélyezett terminálhoz tartozik egy tömbelem. A könyvben tárgyalt konfiguráció két konzolt engedélyez, de a MINIX 3 újrafordításával további virtuális konzolokat, egy vagy két duál soros vonalat és maximum 64 pszeudoterminált adhatunk a rendszerhez.

Van egy másik *extern* definíció is a *tty.h* állományban. A *tty_timers* (13516. sor) egy az időzítő által használt mutató, a *timer_t* mezőkből álló csatolt lista fejének tárolására. A *tty.h* definíciós fájlra több fájl is hivatkozik, a *tty_table* és a *tty_timelist* számára a hely a *tty.c* fordítása közben kerül lefoglalásra.

Két makró, a *buflen* és a *bufend* definíciója található a 13520. és a 13521. sorokban. Ezeket gyakran használják a terminálmeghajtó kódjában, amely sokszor másol adatokat a pufferekbe vagy a pufferekből.

Az eszközfüggetlen terminálmeghajtó

A fő terminálmeghajtó és az eszközfüggetlen segédfüggvények mind a *tty.c* állományban találhatók. Ezt számos makródefiníció követi. Ha egy eszköz inicializálása még nem történt meg, az eszközspecifikus függvények mutatói nullákat fognak tartalmazni, a C fordító tölti fel őket. Ez teszi lehetővé a *tty_active* makró definiálását (13687. sor), amely *FALSE* eredményt ad vissza, ha egy nullát tartalmazó mutatót talál. Természetesen egy eszközt inicializáló kód nem érhető el indirekt módon, hiszen éppen ennek a feladata többek között, hogy inicializálja a mutatókat, amelyek az indirekt elérést lehetővé teszik. A 13690–13696. sorok feltételes makródefiníciókat tartalmaznak, amelyek az RS–232-es vagy a pszeudoterminálok inicializálását egy nulla című függvény hívásával teszik egyenlővé, amikor ezek az eszközök nincsenek konfigurálva. A *do_pty* ugyanígy tiltható le ebben a részben. Ez lehetővé teszi, hogy az ezekhez az eszközökhöz tartozó kódot teljes egészében kihagyják, ha nincs rá szükség.

Mivel minden egyes terminálnak sok konfigurálandó paramétere lehet, és egy hálózatos rendszerben jó néhány terminál fordulhat elő, ezért egy *termios_default* struktúrát deklaráltak és töltöttek fel kezdőértékekkel a 13720–13727. sorokban (ezek mindegyike az *include/termios.h* állományban van definiálva). Ezt a struktúrát átmásolják a *tty_table* tömb egy terminálhoz tartozó elemébe, amikor inicializálni vagy újrainicializálni kell. A speciális karakterek alapértelmezett értékeit a 3.29. ábra mutatta be. A 3.38. ábra a felhasznált különböző jelzők alapértelmezett

Mező	Alapértelmezett érték
c_iflag	BRKINT ICRNL IXON IXANY
c_oflag	OPOST ONLCR
c_cflag	CREAD CS8 HUPCL
c_lflag	ISIG IEXTEN ICANON ECHO ECHOE

3.38. ábra. *A termios jelzők alapértelmezés szerinti értékei*

értékeit sorolja fel. A kód következő sora a *winsize_defaults* struktúrát deklarálja hasonló módon. A C fordítóra hagyják, hogy mindent nulla kezdőértékkel töltsön fel. Ez pontosan a megfelelő alapértelmezés szerinti tevékenységet jelenti: „az ablak mérete ismeretlen, használja az */etc/termcap*-ot" .

A terminálmeghajtó belépési pontja a *tty_task* (13740. sor). Mielőtt a főciklusba belépne, meghívja *tty_init*-et (a 13752. sor). A billentyűzet és a konzol inicializálásához szükséges információkat a gazda számítógépről a sys_getmachine kernelhívással szerzik meg, ezután a billentyűzethardvert inicializálják. A *kb_init_once* rutint hívják meg ennek érdekében. Azért nevezik így, hogy megkülönböztessék egy másik inicializáló eljárástól, amelyet minden egyes virtuális konzol inicializálásának részeként hívnak meg később. Végül egyetlen 0 kerül kiírásra, hogy kipróbálja a kimeneti rendszert, és működésbe hozzon mindent, ami az első használatig nem kerül inicializálásra. A forráskódban a *printf* egy hívása látható, de ez nem ugyanaz a *printf,* mint amit a felhasználói programok használnak, hanem egy speciális változat, amely a konzolmeghajtóban található egyik lokális függvényt hívja, amelynek a neve *putk.*

A főciklus, amely a 13764–13876. sorokban található, alapvetően ugyanúgy működik, mint bármely más meghajtó főciklusa – üzenetet kap, egy switch utasítást hajt végre az üzenet típusának megfelelően, hogy meghívja a megfelelő függvényt, majd egy válaszüzenetet állít elő. Vannak azonban bizonyos komplikációk. Az első, hogy az utolsó megszakítás óta további karaktereket olvashattak be, vagy a kimeneti eszközre kiírandó karakterek várakozhatnak készen. Mielőtt a főciklus megpróbálkozik egy üzenet fogadásával, mindig megvizsgálja a *tp->tty_events* jelzőket minden terminálra, és ha szükséges, meghívja a *handle_events*-et a befejezetlen ügyek kezelésére. Csak amikor már semmi sem követel azonnali figyelmet, kerül sor az üzenetet fogadó hívás végrehajtására.

Az üzenettípusokat bemutató diagram, amely a *tty.c* állomány elejéhez közeli megjegyzésekben található, tartalmazza a leggyakrabban használt típusokat. Sok üzenettípus, amely speciális szolgáltatásokat kíván a terminálmeghajtótól, nincs felsorolva. Ezek nem egy bizonyos eszközre jellemzők. A *tty_task* főciklusa ellenőrzi, hogy vannak-e ilyenek, és kiszolgálja őket, mielőtt az eszközspecifikus üzeneteket ellenőrizné. Először ellenőrzi, hogy van-e *SYN_ALARM* üzenet, és ha az üzenettípus ez, akkor meghívódik az *expire_timers,* ami kiváltja egy felügyeleti rutin végrehajtását. Ezután következik a continue utasítás. Valójában a következő néhány eset mindegyike után egy continue következik. Hamarosan bővebben is szót ejtünk erről.

A következő ellenőrzött üzenettípus a *HARD_INT.* Ez legvalószínűbben a helyi billentyűzeten egy billentyű lenyomásának vagy felengedésének a következménye. Azt is jelentheti, ha a soros portok engedélyezve vannak, hogy bájtokat fogadott az egyik soros port – abban a konfigurációban, amelyet tanulmányozunk, nincsenek, de benne hagytuk a fájlban a feltételesen leforduló kódot, hogy bemutassuk, hogy a soros port bemenetét hogyan kezelnék. Egy bitmezőt az üzenetben arra használnak, hogy a megszakítás forrását megadja.

Ezt követi a *SYS_SIG* ellenőrzése. A rendszerprocesszusoktól (a meghajtóktól és a kiszolgálóktól) megkövetelik, hogy blokkolt állapotba kerüljenek, ha üzenetre várakoznak. A közönséges szignálokat csak az aktív processzusok kapják meg,

így a szabványos Unix üzenetküldési módszer nem működik a rendszerprocesszusoknál. Egy *SYS_SIG* üzenetet használnak egy rendszerprocesszus értesítésére. Egy szignál a terminálmeghajtó számára jelentheti azt, hogy a kernel leáll *(SIGKSTOP),* a terminálmeghajtó leáll *(SIGTERM),* vagy a kernel egy üzenetet kíván a konzolra nyomtatni *(SIGKMESS),* és az ezeknek megfelelő rutinokat meghívják.

A nem eszközspecifikus üzenetek utolsó csoportja a *PANIC_DUMP, DIAGNOSTICS* és az *FKEY_CONTROL.* Ezekről akkor fogunk többet mondani, ha elérkezünk azokhoz a függvényekhez, amelyek kiszolgálják őket.

Most a continue utasításról: a C nyelvben egy continue utasítás rövidre zár egy ciklust, és visszaadja a vezérlést a ciklus elejére. Így ha bármelyiket az eddig említett üzenettípusokból detektálják, amint kiszolgálták, a vezérlés visszakerül a főciklus elejére a 13764. sorban, az események ellenőrzését megismétlik, és ismét meghívják a receive-et, és várják az új üzenetet. Különösen a bemenet esetén fontos, hogy készek legyünk válaszolni olyan gyorsan, amilyen gyorsan lehet. Ugyancsak, ha az üzenettípustesztek bármelyike a ciklus első részében sikeres volt, akkor nincs értelme, hogy elvégezzük bármelyik tesztet az első switch után.

Korábban említettünk komplikációkat, amelyekkel a terminálmeghajtónak foglalkoznia kell. A második komplikáció az, hogy ez a meghajtó több eszközt szolgál ki. Ha a megszakítás nem egy hardvermegszakítás, az üzenetben a *TTY_LINE* mezőt használják arra, hogy megadja, melyik eszköznek kell válaszolnia az üzenetre. A mellékeszközszám dekódolása összehasonlítások sorozatával történik, amelynek eredményeként a *tp*-t a megfelelő *tty_table* elemre pozicionálják (13834–13847. sor). Ha az eszköz egy pszeudoterminál, a *do_pty* (a *pty.c* állományban van) hívása következik, majd a főciklust újrakezdik. Ebben az esetben a *do_pty* generálja a válaszüzenetet. Természetesen, ha a pszeudoterminálok nincsenek engedélyezve, a *do_pty* hívás a már korábban definiált üres makrót használja. Remélhetően nem fordulhat elő az, hogy nem létező eszközök elérésére tesznek kísérletet, de mindig egyszerűbb egy újabb ellenőrzést beiktatni, mint végigellenőrizni, hogy nincs-e másutt hiba a rendszerben. Abban az esetben, ha az eszköz nem létezik, vagy nincs konfigurálva egy *ENXIO* hibaüzenetet tartalmazó válaszüzenet keletkezik, és a vezérlés ismét visszakerül a ciklus elejére.

A meghajtó további része hasonlít ahhoz, amit a többi meghajtó főciklusánál láttunk: az üzenettípus szerinti switch (13862–13875. sor). A kérés típusának megfelelő függvény, a *do_read,* a *do_write* és a többi függvényt hívják meg. Mindegyik esetben inkább a meghívott függvény állítja elő a válaszüzenetet, mint hogy a válasz konstruálásához szükséges információt visszajuttassa a főciklushoz. A főciklus végén kizárólag akkor állítanak elő válaszüzenetet, ha nem érkezett érvényes típusú üzenet, ebben az esetben egy *EINVAL* típusú hibaüzenetet küldenek vissza. Mivel a válaszüzeneteket sok különböző helyről küldik a terminálmeghajtón belül, egy közös rutint, a *tty_reply*-t hívják, amely a válaszüzenetek megszerkesztésének részleteit kezelje.

Ha a *tty_task*-hoz érkezett üzenet érvényes típusú üzenet, nem egy megszakítás eredménye és nem egy pszeudoterminálról érkezett, a főciklus végén lévő switch kézbesíti a *do_read, do_write, do_ioctl, do_open, do_close, do_select* vagy

a *do_cancel* függvények egyikéhez. Mindegyik hívás argumentuma a *tp* mutató, amely egy *tty* struktúrára mutat, és az üzenet címe. Mielőtt részletesen szemügyre vesszük mindegyiket, megemlítünk néhány általános szempontot. Mivel a *tty_task* több terminálleszközt szolgálhat ki, ezeknek a függvényeknek gyorsan kell visszatérniük, hogy a főciklus folytatódhasson.

Bár a *do_read, do_write* és *do_ioctl* lehet, hogy nem képes azonnal elvégezni az összes kért munkát, annak érdekében, hogy az FS feldolgozhasson más hívásokat is, sürgős válaszadásra van szükség. Ha a kérést nem lehet azonnal teljesíteni, a válaszüzenet státusmezőjébe a *SUSPEND* (felfüggesztve) kód kerül. Ez megfelel a 3.33. ábrán a (3)-mal jelzett üzenetnek, és ez felfüggeszti azt a processzust, amely kezdeményezte a hívást, és feloldja fájlrendszer blokkolt állapotát. A (10) és (11)-nek megfelelő üzeneteket később küldik el, amikor a művelet elvégezhető. Ha a kérést maradéktalanul kielégítették, vagy egy hiba történt, akkor az átvitt bájtok száma vagy a hiba kódja kerül be a fájlrendszernek szóló válaszüzenetstátusz mezőjébe. Ebben az esetben egy üzenetet küldenek azonnal a fájlrendszertől az eredeti hívást kezdeményező processzushoz, hogy felébresszék.

A terminálról olvasás alapvetően különbözik egy lemezeszközről történő olvasástól. A lemezmeghajtó kiad egy parancsot a lemez hardverének, és egy idő múlva visszaérkeznek az adatok; ezt csak egy mechanikus vagy elektronikus hiba akadályozhatja meg. A számítógép megjeleníthet egy válaszadást igénylő kérdést (prompt) a képernyőn, de semmilyen módon nem kényszerítheti a billentyűzet előtt ülő személyt, hogy kezdjen el gépelni. Még arra sincs garancia, hogy egyáltalán fog ülni valaki ott. Hogy a kívánt gyors visszatérést meg lehessen valósítani, a *do_read* (13953. sor) azzal kezdi, hogy tárolja azt az információt, amely lehetővé teszi, hogy a kérést később teljesítsék, amikor a bemenő adat megérkezik. Először néhány hibaellenőrzést kell végezni. Hiba, ha az eszköz még mindig az előző kérés teljesítéséhez szükséges bemenetet várja, vagy ha az üzenetben kapott paraméterek hibásak (13964–13972. sor). Ha ezek a tesztek lefutottak, a kérésre vonatkozó információkat átmásolják az eszközhöz tartozó *tp->tty_table* struktúra megfelelő területére (13975–13979. sor). Az utolsó fontos lépés, a *tp->tty_inleft* beállítása a kért karakterek számára. Ezt a változót használják arra, hogy meghatározzák hogy az olvasási kérés teljesítettnek tekinthető-e. Kanonikus módban a *tp->tty_inleft* minden egyes visszaadott karakter után eggyel csökken, amíg csak egy sorvége jel nem érkezik, akkor egyből nullára csökken. Nemkanonikus módban másképpen használják, de bármelyik esetben nullára csökken az értéke, ha a kérést teljesítették, akár úgy, hogy lejárt a várakozási idő, akár úgy, hogy legalább a minimális számú kért bájtot fogadták. Amikor a *tp->tty_inleft* nullára csökken, egy válaszüzenetet küldenek. Amint látni fogjuk, válaszüzenetet több helyen is előállíthatnak. Néha szükség van annak ellenőrzésére, hogy egy olvasó processzus vár-e még a válaszra; a *tp->tty_inleft* nullától különböző értéke alkalmas jelző lehet erre a célra.

Kanonikus módban a terminál addig várja a bemenő adatokat, amíg vagy a kérésben megadott számú karaktert megkapja, vagy egy sor végét vagy a fájl végét elérte. A 13981. sorban tesztelik a *termios* struktúra *ICANON* bitjét, hogy a kanonikus mód érvényes-e a terminálra. Ha ez nincs beállítva, a *termios MIN* és *TIME* értékeit vizsgálják meg, hogy meghatározzák a további teendőket.

A 3.31. ábrán láttuk, hogy *MIN* és *TIME* befolyásolják egymást, így különféle lehetőségeket nyújtanak arra, hogy egy olvasási kérés hogyan viselkedjen. A *TIME* vizsgálata a 13983. sorban történik. A nulla érték megfelel a 3.31. ábra bal oldali oszlopának, és ebben az esetben további vizsgálatra nincs szükség ezen a ponton. Ha *TIME* nem nulla, *MIN*-t kell megvizsgálni. Ha ez nulla, a *settimer*-t hívják, hogy elindítsák az időzítőt, amely egy várakozási idő letelte után megállítja a *DEV_READ* kérést akkor is, ha egyetlen bájtot sem fogadtak. A *tp->tty_min* értékét ekkor 1-re állítják, így a hívás azonnal véget ér, ha egy vagy több bájt érkezik az időkorlát letelte előtt. Ezen a ponton még semmilyen ellenőrzést sem végeztek a lehetséges bemenetre, így lehet, hogy már egynél több karakter is várakozik, amivel teljesíteni lehetne a kérést. Ebben az esetben annyi karaktert, amennyi várakozik, de maximum annyit, amennyit a read kérésben megadtak, visszaadnak, amint a bemenetet megtalálják. Ha mind a *TIME,* mind a *MIN* nullától különböző értékűek, az időzítőnek más a szerepe. Az időzítő ilyenkor bájtok közötti időzítőként működik. Csak akkor indul el, ha az első karakter megérkezett, és minden következő karakter után újraindul. A *tp->tty_eotct* számlálja a karaktereket nemkanonikus mód esetén, és ha ez nulla a 13993. sorban, akkor még egyetlen karakter sem érkezett, és a bájtok közötti időzítő le van tiltva.

Bármely esetről van is szó, a 14001. sorban egy *in_transfer* hívás történik, amely a már a bemenő adatok sorában lévő bájtokat átmásolja közvetlenül az olvasást kérő processzusba. Ezután egy *handle_events* hívás következik, amely újabb adatokat tehet a bemenő adatok sorába, és amely újból meghívja az *in_transfer*-t. Ez a nyilvánvaló hívásismétlés némi magyarázatot igényel. Bár az eddigi megbeszélés a billentyűzetbemenet alapján folyt, a *do_read* kódja a kód eszközfüggetlen részében található, és kiszolgálja a soros vonallal csatlakozó távoli terminálokról történő bemenetet. Lehetséges, hogy egy előző beolvasás olyan mértékben megtöltötte az RS–232-es bemenő adatokat tartalmazó pufferét, hogy a beolvasás fel lett függesztve. Az *in_transfer* első meghívása nem indítja el a folyamot ismét, de a *handle_events* meghívásának már lehet ilyen hatása. Azt, hogy ez az *in_transfer* egy második hívását is eredményezi, tekintsük ajándéknak. A fontos az, hogy biztosítsuk: a távoli terminál ismét küldhessen adatokat. Bármelyik hívás eredményezheti azt, hogy a kérés teljesült, és válaszüzenetet küldjenek az FS számára. A *tp->tty_inleft* jelzőként működik, hogy lássák, megtörtént-e a válasz visszaküldése; ha az értéke még nem nulla a 14004. sorban, a *do_read* önmaga generálja és küldi el a válaszüzenetet. Ez a 14013–14021. sorokban történik. (Feltesszük, hogy itt most nem egy select rendszerhívás történt, és ezért nem fog meghívódni a *select_retry* a 14006. sorban.)

Ha az eredeti kérés egy blokkolt állapotot nem okozó olvasási kérés volt, a fájlrendszernek üzenetet küldenek, hogy egy *EAGAIN* hibakódot küldjön vissza az eredeti hívónak. Ha a hívás egy szokványos blokkolt állapottal járó olvasás, az FS egy *SUSPEND* kódot kap, amely a fájlrendszert felszabadítja a blokkolt állapotból, de azt jelenti a számára, hogy az eredeti hívót továbbra is blokkolt állapotban hagyja. A terminálhoz tartozó *tp->tty_inrepcode* mezőt a *REVIVE* értékre állítják be ebben az esetben. Ha a read kérést teljesítették, ezt a kódot helyezik el a fájlrendszernek küldött üzenetbe, ami azt jelenti, hogy az eredeti hívót felfüggesztették, és most fel kell éleszteni.

A *do_write* (14029. sor) hasonló a *do_read*-hez, de egyszerűbb nála, mivel a write rendszerhívás kezelésére kevesebb opció vonatkozik. Az ellenőrzések hasonlók azokhoz, amelyeket a *do_read* végez annak érdekében, hogy kiderítse, hogy egy előző kiírás nincs-e még mindig folyamatban, és hogy az üzenet paraméterei érvényesek-e, ezután pedig a kérés paramétereit átmásolják a *tty* struktúrába. A *handle_events*-et ezt követően hívják meg, és ellenőrzik a *tp->tty_outleft*-et, hogy befejeződött-e a feladat (14058–14060. sor). Ha igen, akkor egy válaszüzenetet már elküldött a *handle_events,* így nem maradt más tennivaló. Ha nem, egy válaszüzenetet állítanak elő olyan üzenetparaméterekkel, amelyek attól függnek, hogy vajon az eredeti write hívás blokkolt állapotot idézett-e elő.

A következő függvény, a *do_ioctl* (14079. sor) kódja hosszú, de nem nehéz megérteni. A *do_ioctl* törzse két switch utasítás. Az első meghatározza annak a paraméternek a hosszát, amelyre a kérésüzenetben található egyik mutató mutat (14094–14125. sor). Ha a méret nem nulla, a paraméter érvényességét vizsgálják meg. A tartalmat itt nem tudják ellenőrizni, azt megvizsgálhatják, hogy vajon egy megkövetelt méretű struktúra a megadott címen található-e azon a szegmensen belül, amelyben lennie kell. A függvény további része egy másik switch utasítás a kért ioctl művelet szerint (14128–14225. sor).

Sajnos a POSIX megkövetelte műveletek ioctl hívással történő támogatása azt jelentette, hogy ki kellett találni elnevezéseket az ioctl művelet számára, amelyek sugallják, de nem ismétlik meg a POSIX által megkövetelt elnevezéseket. A 3.39. ábra mutatja be a POSIX-kérések elnevezései és a MINIX 3 ioctl hívásában használt elnevezések közötti kapcsolatot. Egy *TCGETS* művelet egy felhasználói *tcgetattr* hívást szolgál ki, és egyszerűen visszaadja a terminálhoz tartozó *tp->tty_termios* struktúra másolatát. A következő négy kéréstípusnak ugyanaz a kódja. A *TCSETSW, TCSETSF* és *TCSETS* kéréstípusok megfelelnek a POSIX-ban definiált *tcsetattr* függvény felhasználói hívásainak, és mindegyik ugyanazt az alapvető tevékenységet végzi: egy új *termios* struktúrát másolnak át a terminálhoz tartozó *tty* struktúrába. A másolás a *TCSETS* híváskor azonnal végrehajtódik, míg a *TCSETSW* és *TCSETSF* hívások esetén, ha a kiírás befejeződött, egy *sys_vircopy* kernelhívással szerzik meg az adatokat a felhasználótól, amit a *setattr* hívása követ (14153–14156. sor). Ha a *tcsetattr* függvényt olyan módosítással hívták meg, hogy halassza el a tevékenységét az éppen folyamatban lévő kiírás végéig, akkor a kérésben szereplő paramétereket elhelyezik a terminálhoz tartozó *tty* struktúrába a későbbi feldolgozás számára, ha a 14139. sorban a *tp->tty_outleft* vizsgálata kideríti, hogy a kiírás még nem fejeződött be. A *tcdrain* a kiírás befejeződéséig felfüggeszt egy programot, amíg a kiírás nem fejeződik be, és egy *TCDRAIN* típusú ioctl hívássá fordítják át. Ha a kiírást már befejezték, nincs több teendő. Ha a kiírást még nem fejezték be, akkor az információt a *tty* struktúrában kell hagyni.

POSIX-függvény	POSIX-művelet	IOCTL-típus	IOCTL-paraméter
tcdrain	(nincs)	TCDRAIN	(nincs)
tcflow	TCOOFF	TCFLOW	int=TCOOFF
tcflow	TCOON	TCFLOW	int=TCOON
tcflow	TCIOFF	TCFLOW	int=TCIOFF
tcflow	TCION	TCFLOW	int=TCION
tcflush	TCIFLUSH	TCFLSH	int=TCIFLUSH
tcflush	TCOFLUSH	TCFLSH	int=TCOFLUSH
tcflush	TCIOFLUSH	TCFLSH	int=TCIOFLUSH
tcgetattr	(nincs)	TCGETS	termios
tcsetattr	TCSANOW	TCSETS	termios
tcsetattr	TCSADRAIN	TCSETSW	termios
tcsetattr	TCSAFLUSH	TCSETSF	termios
tcsendbreak	(nincs)	TCSBRK	int=időtartam

3.39. ábra. *POSIX-hívások és a nekik megfelelő IOCTL-műveletek*

A POSIX *tcflush* függvénye törli a még beolvasatlan bemenő adatokat és/vagy az el nem küldött kimenő adatokat az argumentumainak megfelelően. Az ioctl hívássá történő átfordítása egyértelmű, egy *tty_icancel* függvény hívásából áll, amely az összes terminált kiszolgálja, és/vagy a *tp->tty_ocancel* által mutatott eszközfüggő függvény meghívásából (14159–14167. sor). A *tcflow* hasonlóan egyértelműen fordítódik át egy ioctl hívássá. A kiírás felfüggesztésére vagy újraindítására a *tp->tty_inhibited* értékét *TRUE*-ra vagy *FALSE*-ra állítják, majd beállítják a *tp->tty_events* jelzőt. A kiírás felfüggesztésére vagy újraindítására a megfelelő *STOP* (alaphelyzetben CTRL-S) vagy a *START* (CTRL-Q) kódot küldi el a távoli terminálnak, az eszközfüggő echo rutint használva, amelyre *tp->tty_echo* mutat (14181–14186. sor).

A további műveletek legtöbbjét a *do_ioctl* kezeli, egyetlen kódsorban, amely meghívja a megfelelő függvényt. A *KIOCSMAP* (billentyűzettérkép betöltése) és a *TIOCSFON* (betűkészlet betöltése) műveletek esetében megvizsgálják, hogy az eszköz biztosan egy konzol-e, mivel ezek a műveletek más terminálokra nem alkalmazhatók. Ha virtuális terminálokat használnak, akkor ugyanazt a billentyűzettérképet és betűkészletet alkalmazzák mindegyik konzolhoz, mivel a hardver nem nyújt semmilyen más egyszerű módot ennek megtételére. Az ablakméretező műveletek egy *winsize* struktúrát másolnak a felhasználói processzus és a terminálmeghajtó között. Figyeljük meg a *TIOCSWINSZ* művelet kódja alatti megjegyzéseket. Amikor egy processzus megváltoztatja ablakának a méretét, néhány Unix-verzióban a kernelnek egy *SIGWINCH* szignált kell küldenie a processzus csoportjához. Ezt a szignált a POSIX szabvány nem írja elő, és a MINIX 3-ban nincs megvalósítva. Azonban ha valaki úgy dönt, hogy használni akarja ezeket a struktúrákat, akkor megfontolhatja olyan kódnak a hozzáírását, amely kezdeményezi ezt a szignált.

Az utolsó két eset a *do_ioctl*-ben a POSIX megkövetelte *tcgetpgrp* és *tcsetpgrp* függvényeket támogatja. Ezekhez az esetekhez semmilyen tevékenység nem kapcsolódik, mindig egy hibajelzést adnak. Semmi rossz nincs ebben – ezek a függvények **feladatvezérlést (job control)** támogatnak, ami azt a képességet jelenti, hogy a billentyűzetről felfüggeszthessenek és újraindíthassanak egy processzust. A feladatvezérlést nem követeli meg a POSIX, és a MINIX 3 sem támogatja. Azonban a POSIX előírja ezeket a függvényeket akkor is, ha a feladatvezérlést nem támogatják, hogy biztosítsa a programok hordozhatóságát.

A *do_open*-nek (14234. sor) egy egyszerű alapműveletet kell végrehajtania – megnöveli az eszközhöz tartozó *tp->tty_openct* változó értékét, hogy meggyőződ-

hessenek arról: a megnyitás megtörtént. Azonban vannak bizonyos ellenőrzések, amelyeket először végre kell hajtani. A POSIX specifikációja szerint közönséges terminálok esetén az első processzus, amely egy terminált megnyit, lesz a **munkavezető (session leader)**, és amikor a munkavezető meghal, akkor a terminál elérésének jogát a processzus csoportjában lévő többi processzustól is megvonják. A démonoknak szükségük lehet rá, hogy hibaüzeneteket írhassanak ki. Ha a hibakimenetük nincs átirányítva egy fájlba, akkor egy olyan képernyőn kell megjelenjen, amelyet nem lehet lezárni.

A MINIX 3-ban erre a célra egy */dev/log* nevű eszköz szolgál. Ez fizikailag megegyezik a */dev/console* eszközzel, de egy saját mellékeszközszámmal címzik meg, és másként is kezelik. Ez egy csak írható eszköz, így a *do_open* egy *EACCESS* hibával tér vissza, ha megkísérelnék olvasásra megnyitni (14246. sor). A *do_open* által végzett másik teszt az *O_NOCTTY* jelzőt vizsgálja. Ha ez nincs beállítva, és az eszköz nem a */dev/log*, akkor a terminál a processzuscsoport vezérlőtermináljává válik. Ez úgy történik, hogy a hívó processzus számát beteszik a *tty_table* táblázat megfelelő elemének *tp->tty_pgrp* mezőjébe. Ezt követően növelik meg a *tp->tty_openct* változó értékét és küldik el a válaszüzenetet.

Egy terminálleszköz megnyitható többször is, ezért a következő, *do_close* nevű függvénynek (14260. sor) nincs más teendője, mint csökkenteni a *tp->tty_openct* értékét. A 14266. sorban lévő ellenőrzés meghiúsítja az eszköz lezárására tett kísérletet, ha az éppen a */dev/log*. Ha ez a művelet az utolsó lezárás, a beolvasást megszakítják a *tp->tty_icancel* meghívásával. A *tp->tty_ocancel* és *tp->tty_close* mutatókkal hivatkozott eszközfüggő rutinokat ugyancsak meghívják. Ezután az eszközhöz tartozó *tty* struktúra különböző mezőit visszaállítják alapértelmezés szerinti értékeikre, és elküldik a válaszüzenetet.

Az utolsó általunk tekintett üzenettípus kezelője a *do_cancel* (14281. sor). Ezt hívják meg, amikor egy szignál érkezik egy olyan processzus számára, amely blokkolt állapotba került, amikor olvasni vagy írni próbált. Három állapotot kell megvizsgálni:

1. A processzus olvashatott, miközben megszüntették.
2. A processzus írhatott, miközben megszüntették.
3. A processzust egy *tcdrain* felfüggeszthette, amíg a kiírás be nem fejeződik.

Mindegyik esetet megvizsgálják a függvényben, és az általános *tp->tty_icancel*-t vagy pedig a *tp->tty_ocancel* által mutatott eszközfüggő rutint hívják meg, ha szükséges. Az utolsó esetben az egyetlen megkövetelt tevékenység a *tp->tty_ioreq* jelző törlése, hogy ezzel jelezzék: az ioctl művelet mostanra befejeződött. Végül a *tp->tty_events* jelzőt beállítják, és egy válaszüzenetet küldenek.

A terminálmeghajtót támogató kód

Most, miután megnéztük a *tty_task* főciklusa által hívott legfelső szintű függvényeket, ideje, hogy megnézzük az ezeket támogató kódot is. A *handle_events*-cel (14358. sor) fogjuk kezdeni. Ahogy a korábbiakban már említettük, a terminál-

meghajtó főciklusának minden egyes lefutásakor minden egyes terminaleszközre megvizsgálják a *tp->tty_events* jelzőt, és ha ez azt mutatja, hogy egy adott terminál figyelmet igényel, meghívják a *handle_events*-et. A *do_read* és a *do_write* ugyancsak meghívják a *handle_events*-et. Ennek a rutinnak gyorsan kell dolgoznia. Törli a *tp->tty_events* jelzőt, majd meghívja az eszközspecifikus rutinokat, hogy olvassanak és írjanak a függvények címeit tartalmazó *tp->tty_devread* és *tp->tty_devwrite* mutatók felhasználásával (14382–14385. sor).

Ezeket a függvényeket feltétel nélkül hívják meg, mivel semmilyen mód nincs arra, hogy megállapítsák: egy olvasási vagy egy írási kérés állította-e be a jelzőt. Tervezéskor választottak így, mivel két jelzőnek az ellenőrzése minden egyes eszközre költségesebb lehet, mint két függvényhívás elvégzése minden olyan esetben, amikor az eszköz aktív volt. Valamint a legtöbb esetben, amikor egy karakter érkezik egy terminálról, akkor azt echózni kell, így mindkét hívásra szükség van. Amint azt a *do_ioctl* által hívott *tcsetattr* kezelésének tárgyalásakor megjegyeztük, a POSIX elhalaszthatja a vezérlőműveletek végrehajtását az eszközökön, amíg az aktuális kiírás be nem fejeződött, ezért az eszközfüggő *tty_devwrite* meghívása utáni időpont éppen megfelelő az *ioctl* műveletek elvégzéséhez. Ez a 14388. sorban történik, ahol a *dev_ioctl*-t akkor hívják meg, ha van függőben lévő vezérlésiművelet-igény.

Mivel a *tp->tty_events* jelzőt a megszakítások állítják be, és egy gyors eszközről a karakterek gyors ütemben érkezhetnek egymás után, ezért van esély arra, hogy mialatt az eszközfüggő olvasó és író, valamint a *dev_ioctl* rutinok a munkájukat elvégzik, egy újabb megszakítás ismételten beállítsa a jelzőt. Ezért magas a prioritása a bemenő karakterek továbbmásolásának abból a pufferből, ahova a megszakításrutin először elhelyezte őket. Így a *handle_events* egészen addig ismétli az eszközfüggő rutinok hívását, amíg a *tp->tty_events* jelzőt beállított állapotban levőnek találja a ciklus végén (14389. sor). Amikor a bevitel folyama megáll (lehet ez a kiírás folyama is, bár az adatbevitel esetében sokkal valószínűbbek az ilyen megismételt kérések), meghívják az *in_transfer*-t, hogy átmásolja a karaktereket a bemeneti sorból egy pufferbe, amely azon a processzuson belül van, amely a read műveletet meghívta. Az *in_transfer* maga küld egy válaszüzenetet, akár úgy, hogy a maximális számú kért karakter átvitelével teljesítette a kérést, vagy elérte a sor végét (kanonikus módnál). Ha ez így van, a *tp->tty_left* nulla lesz a *handle_events*-be történő visszatéréskor. Ezen a ponton egy további vizsgálat következik, és egy válaszüzenetet küldenek, ha az átvitt karakterek száma elérte a kért minimális karakterszámot. A *tp->tty_inleft* ellenőrzése megakadályozza, hogy duplikált üzenetet küldjenek.

A következő rutin, amelyet megvizsgálunk, az *in_transfer* (14416. sor), azért felelős, hogy a meghajtó memóriaterületén levő bemeneti sorból az adatokat átvigye annak a felhasználói processzusnak a pufferébe, amely a beolvasást kérte. Egy egyszerű blokkmásolás azonban itt nem lehetséges. A bemenő sor egy ciklikus puffer, és a karaktereket ellenőrizni kell, hogy a fájl végét nem érték-e el, illetve ha kanonikus mód érvényes, az adatátvitel csak a sor végéig tart. Ugyancsak a bemenő sor 16 bites mennyiségekből áll, a fogadó fél puffere azonban egy 8 bites karaktereket tartalmazó tömb. Ezért egy közbülső, lokális puffer használatára

van szükség. A karaktereket egyesével megvizsgálják, amikor a lokális pufferbe helyezik őket, és amikor az megtelik, vagy a bemenő sort kiürítették, meghívják a *sys_vircopy* rutint, amely átmásolja a lokális puffer tartalmát a fogadó processzus pufferébe (14432–14459. sor).

A *tty* struktúra három változóját, a *tp->tty_inleft*-et, a *tp->tty_eotct*-t, valamint a *tp->tty_min*-t használják arra, hogy eldöntsék, az *in_transfer*-nek van-e bármilyen teendője, és az első kettő ezek közül a változók közül vezérli a főciklust. Ahogyan azt már említettük, a *tp->tty_inleft* kezdetben megkapja a read hívásban kért karakterek számát. Általában ezt eggyel csökkentik, valahányszor egy karakter átvitelre került, de hirtelen nullává válhat, ha egy feltétel jelzi a bemenet végének elérését. Valahányszor ez a változó nullává válik, egy válaszüzenetet küldenek az olvasást kérőnek, így ez egy jelzőként is működik, amely mutatja, hogy küldtek-e már üzenetet, vagy sem. Így ha a 14429. sorban az ellenőrzés azt mutatja, hogy a *tp->tty_inleft* már nulla, akkor ez elég ok arra, hogy megszakítsák az *in_transfer* végrehajtását üzenet küldése nélkül.

A vizsgálat következő részében a *tp->tty_eotct*-t és a *tp->tty_min*-t hasonlítják össze. Kanonikus módban mindkét változó a bemenet teljes sorainak számára, míg nemkanonikus módban a karakterek számára utal. A *tp->tty_eotct* eggyel nő, ha a bemeneti sorba egy sortörés vagy egy bájt kerül, és eggyel csökkenti az *in_transfer*, amikor egy sort vagy egy karaktert eltávolít a sorból. Más szavakkal, azoknak a soroknak vagy bájtoknak a számát tárolja, amelyeket a terminálmeghajtó fogadott, de még nem továbbított az olvasó processzusnak. A *tp->tty_min* tárolja a sorok minimális számát (kanonikus módban), illetve a karakterekét (nemkanonikus módban), amennyit át kell vinni az olvasási kérés teljesítéséhez. Az értéke mindig 1 kanonikus módban, illetve 0 és *MAXINPUT* (MINIX 3-ban ez 255) közé esik nemkanonikus módban. A 14429. sor ellenőrzésének második fele azt eredményezi, hogy az *in_transfer* azonnal visszatér kanonikus módban, ha egy teljes sor még nem érkezett meg. Az átvitelt nem hajtják végre, amíg egy sor teljessé nem válik, így a sor tartalma módosítható, például egy ERASE vagy KILL begépelésével, mielőtt a felhasználó az ENTER billentyűt leüti. Nemkanonikus módban akkor történik azonnali visszatérés, ha a minimális számú beérkezett karakter még nem áll rendelkezésre.

Néhány sorral később a *tp->tty_inleft* és a *tp->tty_eotct* vezérli az *in_transfer* főciklusát. Kanonikus módban az átvitel addig folytatódik, amíg a bemeneti sorban már egyetlen teljes sor sem marad. Nemkanonikus módban a *tp->tty_eotct* a függőben levő karakterek számát tartalmazza. A *tp->tty_min* meghatározza azt, hogy a ciklusba beléptek-e, de azt nem, hogy mikor kell megállni. Amint beléptünk a ciklusba, vagy az összes rendelkezésre álló karaktert, vagy az eredeti hívásban kért számú karaktert átviszik, aszerint, hogy melyik a kisebb.

A bemenő sorban levő karakterek 16 bitesek. A felhasználói processzus számára ténylegesen továbbítandó karakterkód az alsó 8 biten található. A 3.40. ábra bemutatja, hogy a felső biteket hogyan használják. Három közülük annak jelzésére szolgál, hogy a karakter előtt leütötték-e az escape billentyűt (CTRL-V), a fájl végét jelzi-e vagy egyike a sor végét jelző karaktereknek. Négy biten azt tároljuk, hogy hány karakterhelyre van szükség a képernyőn, ha az adott karaktert echózzák. A 14435. sorban levő ellenőrzésben megvizsgálják, hogy az *IN_EOF* bit (a 3.40. ábrán a *D*) be van-e állítva. Ezt rögtön a belső ciklus elején megvizsgálják, mert a fájlvége karaktert (CTRL-D) magát nem továbbítják az olvasó processzusnak, és nem számít bele a karakterek számába sem. Minden egyes karakter átvitelekor egy maszkot alkalmaznak, hogy nullázzák a felső 8 bitet, és csak az alsó 8 biten található ASCII kódot viszik át a lokális pufferbe (14437. sor).

V: IN_ESC, az LNEXT lenyomásával (CTRL-V)
D: IN_EOF, fájlvége (CTRL-D karakter)
N: IN_EOT, sortörés (soremelés és egyebek)
cccc: az echózott karakterek száma
7: a 7. bit kinullázásra kerül, ha az ISTRIP be van állítva
6-0: az ASCII kód

3.40. ábra. *A bemeneti sorban levő karaktereket felépítő mezők*

Több lehetőség is van annak jelzésére, hogy a bemenő adatok végét jelezzük, de az eszközfüggő bemeneti rutin feladata annak meghatározása, hogy az érkezett karakter egy soremelés, CTRL-D vagy más ilyen karakter, és hogy megjelöljön minden ilyen karaktert. Így az *in_transfer*-nek csak ezt a jelzést, az *IN_EOT* bitet (a 3.40. ábrán az *N*) kell vizsgálnia a 14454. sorban. Ha ilyet találtak, akkor a *tp->tty_eotct*-t eggyel csökkentik. Nemkanonikus módban minden egyes karakter ilyennek számít, amikor beteszik őket a bemeneti sorba, és ekkor minden karaktert megjelölnek az *IN_EOT* bittel is, így a *tp->tty_eotct* azon karakterek számát fogja tartalmazni, amelyeket még nem távolítottak el a bemeneti sorból. Az egyetlen különbség az *in_transfer* főciklusának működésében a két mód esetén a 14457. sorban található. Itt a *tp->tty_inleft*-et nullázzák, ha olyan karaktert találnak, amely sortörésként van megjelölve, de csak akkor, ha a kanonikus mód van érvényben. Így amikor a vezérlés visszatér a ciklus elejére, akkor a ciklus megfelelően véget ér egy sortörés után kanonikus módban, de nemkanonikus módban a sortöréseket figyelmen kívül hagyják.

Amikor a ciklus befejeződik, van általában egy részlegesen megtelt, átvitelre váró lokális puffer (14461–14468. sor). Ekkor ha a *tp->tty_inleft* elérte a nullát, egy válaszüzenet küldenek. Kanonikus módban ez mindig így van, de ha nemkanonikus mód van érvényben, és az átvitt karakterek száma kisebb, mint a teljes kérésé, akkor nem küldenek választ. Ha elég jó memóriánk van a részletek megjegyzésére, emlékezhetünk rá, hogy ahol az *in_transfer* hívásait láttuk (a *do_read* és a *handle_events* függvényekben) az *in_transfer* hívása után következő kód akkor küld válaszüzenetet, ha az *in_transfer* úgy tér vissza, hogy a *tp->tty_min*-ben szereplő mennyiségnél több karaktert vitt át, ami természetesen éppen ez az eset. Annak az okát, hogy miért nem az *in_transfer*-ből küldik a válaszüzenetet feltétel nélkül, akkor fogjuk látni, amikor megtárgyaljuk a következő függvényt, amely az *in_transfer*-t más körülmények között hívja.

A következő függvény az *in_process* (14486. sor). Ezt a függvényt az eszközspecifikus szoftver hívja olyan közös feldolgozás elvégzésére, amelyeket minden be-

menetre végre kell hajtani. Paraméterei: egy mutató a forráseszköz *tty* struktúrájára, egy mutató a feldolgozásra váró karakterek 8 bites tömbjére és egy számláló. A számláló értékét visszaadják a hívónak. Az *in_process* hosszú függvény, de a tevékenységei nem bonyolultak. A bemeneti sorhoz 16 bites karaktereket ad hozzá, amit később az *in_transfer* dolgoz fel.

Az *in_transfer* a karakterek kezelésére a következő kategóriákat biztosítja.

1. A közönséges karakterek 16 bitre kiterjesztés után kerülnek be a bemeneti sorba.
2. A későbbi feldolgozást befolyásoló karakterek módosítják a jelzőket, hogy jelezzék a hatást, de nem kerülnek be a bemeneti sorba.
3. Azok a karakterek, amelyek az echózást vezérlik, azonnali hatást váltanak ki, de nem kerülnek be a bemeneti sorba.
4. A különleges jelentőséggel rendelkező karakterek kódokat kapnak, mint például az *EOT* bit, amit a felső bájtjuk tárol, amikor bekerülnek a bemeneti sorba.

Először nézzünk meg egy teljesen normális esetet: egy közönséges karaktert, mint például az „x" (ASCII kódja 0x78) leütnek egy rövid sor közepén, nincs semmilyen vezérlőszekvencia érvényben, egy olyan terminálon, amelyet a szabványos MINIX 3 alapértelmezés szerinti tulajdonságaival állítottak üzembe. Amikor a bemeneti eszközről ez a karakter megérkezik, a 3.40. ábrán jelzett 0–7 biteken fog elhelyezkedni. A 14504. sorban a legfelső bitjét, a 7. bitet kinullázhatják, ha az *ISTRIP* jelző be van állítva, de a MINIX 3 alapbeállítása, hogy nem vágják le a legfelső bitet, megengedve teljes 8 bites kódok bevitelét. Ez egyébként nem befolyásolja az „x" karakterünk bevitelét. A MINIX 3 alapértelmezése megengedi a bemenet bővített kezelését, ennek következtében a *tp->tty_termios.c_lflag IEXTEN* bitjének vizsgálata (4507. sor) igazat ad, de a következő ellenőrzés már hamis eredményt szolgáltat az általunk feltett feltételek mellett: nincs karakterválasztás (escape) érvényben (14510. sor), a bemenet maga nem egy escape karakter (14517. sor), és ez a bemenet nem a *REPRINT* karakter (14524. sor).

A következő néhány sorban található ellenőrzések szerint a bemenő karakter nem a speciális *_POSIX_VDISABLE* karakter, sem a *CR* vagy az *NL*. Végül egy pozitív eredmény: kanonikus mód van érvényben; ez a normál alapértelmezés (14324. sor). Az „x" karakterünk nem az *ERASE* karakter, sem a *KILL*, *EOF* (CTRL-D), *NL* vagy *EOL* valamelyike, így a 14576. sorig még semmi sem fog történni vele. Itt azt találjuk, hogy az *IXON* bit alapértelmezés szerint be van állítva, azaz a *STOP* (CTRL-S) és *START* (CTRL-Q) karakterek használata engedélyezett, de az ezután következő vizsgálatok ezekre nem találnak egyezést. A 14597. sorban azt találjuk, hogy az *ISIG* bit, ami alapértelmezés szerint be van állítva, megengedi az *INTR* és a *QUIT* karakterek használatát, de ismét nem található egyezés.

Valójában az első érdekes dolog, amely egy közönséges karakterrel megtörténhet, a 14610. sorban fordul elő, ahol ellenőrzik, hogy a bemeneti sor megtelt-e már. Ha erről lenne szó, akkor a karaktert nem vennék figyelembe ennél a pontnál, hiszen kanonikus mód van érvényben, és a felhasználó nem látná a

képernyőn echózva a karaktert. (A continue utasítás hagyja figyelmen kívül a karaktert, mivel az a külső ciklus újrakezdését váltja ki.) Azonban teljesen normális feltételeket tételeztünk fel ehhez a példához, ezért tegyük fel, hogy a puffer még nincs tele. A következő ellenőrzés (14616. sor), amely azt vizsgálja, hogy speciális nemkanonikus módú feldolgozásra szükség van-e, nem teljesül, ami egy előreugrást eredményez a 14629. sorra. Itt hívják meg az *echo*-t, hogy a felhasználó számára megjelenítsék a karaktert, mivel a *tp->tty_termios.c_lflag ECHO* bitje alapértelmezés szerint be van állítva.

Végül a 14632-től 14636-ig terjedő sorokban „túladunk" a karakteren, beteszszük a bemeneti sorba. Ekkor a *tp->tty_incount*-ot megnöveljük eggyel, de mivel ez egy közönséges karakter, így nincs megjelölve az *EOT* bittel, a *tp->tty_eotct* nem változik.

A ciklus utolsó sora meghívja az *in_transfer*-t, ha a bemeneti sorba most átvitt karakter éppen megtölti azt. Azonban normál feltételek mellett, amit feltételeztünk ebben a példában, az *in_transfer* semmit sem fog tenni, akkor sem, ha meghívják, mivel (feltettük, hogy a bemenő sort normális módon szolgálják ki és a megelőző bemenetet fogadták, amikor az előző bemeneti sor teljessé vált) a *tp->tty_eotct* nulla, *tp->tty_min* értéke 1, és az *in_transfer* elején levő ellenőrzés (14429. sor) azonnali visszatérést eredményez.

Most, hogy áthaladtunk az *in_process*-en egy közönséges karakterrel szokásos feltételek mellett, menjünk vissza az *in_process* kezdetére, és vizsgáljuk meg, mi történik kevésbé szokásos feltételek esetén. Először nézzük meg a karakterválasztást (escape), amely lehetővé teszi, hogy a felhasználói processzusnak egy olyan karaktert továbbítsanak, amelynek általában speciális hatása van. Ha egy karakterválasztás érvényben van, akkor a *tp->tty_escaped* jelző be van állítva, és amikor ezt észlelik (14510. sor), a jelzőt azonnal törlik, és beállítják az aktuális karakternek a 3.40. ábrán *V*-vel jelölt *IN_ESC* bitjét. Ez speciális feldolgozást követel meg, amikor a karaktert echózzák – a vezérlőkaraktereket „^" plusz vezérlőkarakter formában jelenítik meg, hogy láthatók legyenek. Az *IN_ESC* bit megakadályozza azt is, hogy az adott karaktert a következő vizsgálatok speciális karakterként értelmezzék.

A következő néhány sor magát az escape karaktert, az *LNEXT* karaktert (CTRL-V alapértelmezés szerint) dolgozza fel. Amikor az *LNEXT* kódot észlelik, a *tp->tty_escaped* jelzőt beállítják, és kétszer meghívják a *rawecho*-t egy „^" és egy visszatörlés (backspace) karakter kiírására. Ez arra emlékezteti a billentyűzetnél ülő felhasználót, hogy egy karakterválasztás van érvényben, és amikor a következő karaktert echózzák, az felülírja a „^"-ot. Az *LNEXT* karakter egy olyan karakterre példa, amely hatással van a következő karakterekre (ebben az esetben csak a közvetlenül utána következő karakterre). Ez a karakter nem kerül be a sorba, és két *rawecho* hívás után a ciklus újrakezdődik. Az előbbi két feltételvizsgálat sorrendje lényeges, mert lehetővé teszi azt, hogy az *LNEXT* karaktert kétszer egymás után lenyomják annak érdekében, hogy a másodikat egy processzus számára átadják, mint aktuális adatot.

A következő *in_process* által feldolgozott speciális karakter a *REPRINT* karakter (CTRL-R). Amikor egy ilyen karaktert találnak, akkor ez a *reprint* meghívását

okozza (14525. sor) a következőkben, amely az aktuális echózott kimenetet újra megjeleníti. Ezután a *REPRINT*-et magát eldobják anélkül, hogy hatással lenne a bemeneti sorra.

Minden speciális karakter kezelését részletesen megvizsgálni unalmas lenne, és az *in_process* forráskódja is világos. Csak néhány további dolgot említünk meg. Az egyik az, hogy a bemeneti sorba elhelyezett 16 bites értékek felső bájtjának speciális bitjeivel könnyű hasonló hatású karakterek egy osztályát azonosítani. Így az *EOT* (CTRL-D), az *LF* és a váltakozó *EOL* (alaphelyzetben nem definiált) mindegyikét a 3.40. ábrán *D*-vel jelölt *EOT* bittel megjelölik (14566–14573. sor), hogy a későbbi felismerésüket megkönnyítsék.

Végül megindokoljuk az *in_transfer* korábban említett furcsa viselkedését. Válaszüzenetet nem minden esetben generálnak, amikor véget ér, bár a korábban látott *in_transfer* hívásokban úgy tűnt, mintha visszatéréskor minden esetben generálnának válaszüzenetet. Felidézve az *in_transfer* meghívásának, amit az *in_process* hajt végre, amikor a bemeneti sor megtelt (14639. sor), nincs semmilyen hatása, ha kanonikus mód van érvényben. De ha nemkanonikus módú feldolgozás lenne kívánatos, minden egyes karakter *EOT* bitjét beállítják a 14618. sorban, így minden egyes karaktert megszámol a *tp->tty_eotct* a 14636. sorban. Ezután ez okozza azt, hogy belépünk az *in_transfer* főciklusába annak meghívásakor, a megtelt bemeneti sor miatt nemkanonikus módban. Ilyen esetekben nincs szükség az *in_transfer* befejeződésekor üzenetküldésre, hiszen valószínűleg lesz még több beolvasni való karakter az *in_process*-be való visszatérés után. Valójában bár a kanonikus mód esetén a read által beolvasható karakterek számát korlátozza a bemeneti sor mérete (MINIX 3-ban 255 karakter), nemkanonikus módban egy read hívásnak képesnek kell lennie a POSIX által megkövetelt *_POSIX_SSIZE_MAX* számú karaktert átvinni. Ennek értéke MINIX 3-ban 32767.

A következő néhány függvény *tty*.c-ben a karakterbemenetet támogatja. A *tty_echo* (14647. sor) néhány karaktert speciálisan kezel, de többnyire csak megjeleníti őket ugyanannak az adatbevitelre használt eszköznek a kimeneti oldalán. Egy processzustól származó kimenet is továbbítódhat egy eszközhöz a bemenet echózásával egy időben, ami összezavarja a dolgokat, ha a billentyűzet előtt ülő felhasználó megpróbálja megnyomni a visszatörlést (backspace). Hogy ezt megoldják, a *tp->tty_reprint* jelzőt mindig *TRUE* értékre állítják az eszközfüggő kiíró rutinok esetén normál kimenetnél, így egy visszatörlés kezelésére meghívott függvény azt jelezheti, hogy kevert kimenetet állítottak elő. Mivel a *tty_echo* is az eszközfüggő kimeneti rutinokat használja, a *tp->tty_reprint* aktuális értékét meg kell őrizni echózás alatt; erre szolgál az *rp* lokális változó (14668–14701. sor). Azonban amikor épp egy új bemeneti sor kezdődik el, az *rp*-t *FALSE* értékre állítják ahelyett, hogy a régi értékét vennék, így biztosítják, hogy a *tp->tty_reprint* törlésre kerüljön, amikor az *echo* véget ér.

Észrevehettük, hogy a *tty_echo* visszaad egy értéket, például az *in_process* 14629. sorában lévő hívásnál:

```
ch = tty_echo(tp, ch)
```


Az *echo* által visszaadott érték tartalmazza a képernyő-pozíciók számát, amelyet az echózás megjelenítésére használtak fel, ez maximum 8 lehet, *TAB* karakter esetén. Ezt a számlálót helyezték el a 3.40. ábrán a *cccc*-vel jelölt mezőbe. A közönséges karakterek egy helyet foglalnak el a képernyőn, de ha egy vezérlőkaraktert [nem a *TAB*, *NL*, *CR* vagy egy *DEL* (0x7F)] echóznak, az egy „^” és egy nyomtatható ASCII karakter formájában jelenik meg, és két helyet foglal el a képernyőn. Másrészt egy *NL* vagy egy *CR* nem foglal el egyetlen helyet sem. A tényleges echózást természetesen egy eszközfüggő rutin meghívásával kell elvégezni, azaz valahányszor egy karaktert ki kell juttatni az eszközre, egy indirekt függvényhívást kell elvégezni a *tp->tty_echo* felhasználásával, ahogyan az például a 14696. sorban történik közönséges karakterek esetén.

A következő függvényt, a *rawecho*-t arra használjuk, hogy kikerüljük az *echo* által végrehajtott speciális tevékenységeket. Ez ellenőrzi, hogy az *ECHO* jelző be van-e állítva, és ha igen, akkor elküldi a karaktert az eszközfüggő *tp->tty_echo* rutinon keresztül mindenféle speciális feldolgozás nélkül. Itt egy *rp* nevű lokális változó szolgál arra, hogy megakadályozza, a *rawecho* saját maga által hívott kimeneti rutinja megváltoztassa a *tp->tty_reprint* értékét.

Amikor az *in_process* egy visszatörlést talált, a következő függvényt, a *backover*-t (14721. sor) hívják meg. Ez úgy módosítja a bemenő sort, hogy eltávolítja a sor korábbi fejét, ha a visszalépés lehetséges – ha a sor üres, vagy az utolsó karakter egy sortörés, akkor visszalépés nem lehetséges. Ekkor megvizsgálják az *echo*-nál és a *rawecho*-nál említett *tp->tty_reprint* jelzőt. Ha ez *TRUE,* akkor a *reprint* rutint (14732. sor) hívják meg, hogy a kimeneti sor egy nyers másolatát megjelenítsék a képernyőn. Ezután az utolsó kiírt karakter *len* mezőjét vizsgálják meg (ez a *cccc* mező a 3.40. ábrán), hogy meghatározzák, hány karaktert kell törölni a képernyőről, és minden egyes karakterre egy-egy visszatörlés-szóköz-visszatörlés sorozatot küldenek a *rawecho* segítségével, hogy eltávolítsák a nemkívánatos karaktereket a képernyőről, és szóközzel (space) helyettesítsék.

A következő függvény a *reprint*. A *backover*-en kívül a felhasználó is meghívhatja, ha megnyomja a *REPRINT* (CTRL-R) billentyűt. A 14764-től 14769-ig terjedő ciklus a bemeneti sorban visszafelé haladva megkeresi az utolsó sortörést. Ha az utolsó betöltött helyen találja meg, akkor nincs teendő, és a *reprint* visszatér. Egyébként kiírja a képernyőre a CTRL-R-t, amely a képernyőn mint a „^ R” kétkarakteres sorozat jelenik meg, majd a következő sor elejére áll, és ismételten kiírja a bemeneti sort az utolsó sortöréstől a végéig.

Most érkeztünk el az *out_process*-hez (14789. sor). Az *in_process*-hez hasonlóan ezt is az eszközfüggő rutinok hívják, de ez egyszerűbb. Ezt csak az RS–232 és pszeudoterminálok eszközspecifikus rutinjai hívják, a konzol rutinja nem. Az *out_process* a bájtok ciklikus pufferén dolgozik, de nem távolítja el őket a pufferből. Az egyetlen változtatás, amelyet a tömbön végrehajt az, hogy beilleszt egy *CR* karaktert a pufferben levő *NL* karakter elé, ha mind az *OPOST* bit (kimenő adatok feldolgozása engedélyezett), mind az *ONLCR* (NL átalakítása CR-NL-re) a *tp->tty_termios.oflag*-ben be vannak állítva. A MINIX 3 esetében mindkét bit alapértelmezés szerint be van állítva. A feladata még az, hogy az eszköz *tty* struk-

túrájában levő *tp->tty_position* változó értékét karbantartsa. A tabulátor és visszatörlés karakterek bonyolulttá teszik az életet.

A következő rutin a *dev_ioctl* (14874. sor). Ez a *do_ioctl*-t támogatja a *tcdrain* és a *tcsetattr* függvény végrehajtásában, amikor akár a *TCSADRAIN,* vagy a *TCSAFLUSH* opciókkal hívták meg. Ezekben az esetekben a *do_ioctl* nem tudja befejezni a tevékenységét azonnal, ha a kimenet még nem kész, így a kérésre vonatkozó információt eltárolják a *tty* struktúrának a késleltetett ioctl műveletek számára fenntartott részében. Amikor a *handle_events* fut, az eszközfüggő kiíró rutin meghívása után először ellenőrzi a *tp->tty_ioreq* mezőt, és meghívja a *dev_ioctl*-t, ha van függőben lévő művelet. A *dev_ioctl* megvizsgálja a *tp->tty_outleft*-et, hogy megállapítsa, befejeződött-e a kiírás, ha igen, akkor elvégzi ugyanazokat a tevékenységeket, mint amiket azonnal végrehajtott volna, ha nem lett volna késleltetés. A *tcdrain* kiszolgálásánál az egyetlen tevékenység a *tp->tty_ioreq* mező törlése, és egy válaszüzenet küldése a fájlrendszernek, hogy ébressze fel azt a processzszust, amely az eredeti hívást kezdeményezte. A *tcsetattr* hívás *TCSAFLUSH* változata meghívja a *tp->tty_icancel*-t a bemenet megszakítására. A *tcsetattr* mindkét változatánál lemásolják a *termios* struktúrát, amelynek címét az eredeti ioctl híváskor átadják az eszköz *tp->tty_termios* struktúrájába. Ezután meghívják a *setattr* függvényt, amit hasonlóan a *tcdrain*-hez, egy válaszüzenet elküldése követ, a blokkolt eredeti hívó processzus felébresztésére.

A *setattr* (14899. sor) a következő eljárás. Amint már láttuk, ezt a *do_ioctl* vagy a *dev_ioctl* hívja a terminálleszköz tulajdonságainak megváltoztatására, valamint a *do_close,* hogy visszaállítsa a tulajdonságokat az alapértelmezett értékekre. A *setattr* függvényt mindig meghívják azután, hogy egy új *termios* struktúrát másoltak át az eszköz *tty* struktúrájába, hiszen pusztán a paraméterek bemásolása nem elég. Ha a vezérelni kívánt eszköz most nemkanonikus módba kerül, az első teendő, hogy a jelenleg a bemeneti sorban levő minden karakter *IN_EOT* bitjét be kell állítani, mintha ezeket a karaktereket eredetileg is nemkanonikus módban helyezték volna a bemeneti sorba, ha akkor nemkanonikus mód lett volna érvényben. Egyszerűbb végigmenni és ezt tenni minden karakterre (14913–14919. sor), mint megvizsgálni, hogy a karaktereknél ez a bit be volt-e már állítva. Semmilyen módszer nincs annak meghatározására, hogy mely tulajdonságok változtak meg éppen most, és melyek tartották meg előző értéküket.

A következő tevékenység a *MIN* és *TIME* értékek ellenőrzése. Kanonikus módban a *tp->tty_min* mindig 1; ezt a 14926. sorban állítják be. Nemkanonikus módban a két érték kombinációi négy különböző működési módot tesznek lehetővé, amint az a 3.31. ábrán látható. A 14931-tól 14933-ig terjedő sorokban először a *tp->tty_min* felveszi a *tp->tty_termios.c_cc*[*VMIN*]-ben átadott értéket, amelyet ezután módosítanak, ha az értéke nulla, és a *tp->tty_termios.c_cc*[*VTIME*] nem nulla.

Végül a *setattr* gondoskodik arról, hogy a kiírás ne álljon meg, ha az XON/XOFF vezérlés le van tiltva, elküld egy *SIGHUP* szignált, ha a kiírás sebességét nullára állították, és végrehajt egy indirekt hívást a *tp->tty_ioctl* által mutatott eszközfüggő rutinra, hogy végrehajtsa azt, amit csak az eszközszinten lehet végrehajtani.

A következő függvény, a *tty_reply* (14952. sor) már sokszor előkerült az előzők során. A feladata teljes mértékben világos: összeállít és elküld egy üzenetet. Ha valamilyen okból a válaszküldés meghiúsul, akkor egy pánik állapot következik. A következő függvények ugyanilyen egyszerűek. A *sigchar* (14973. sor) megkéri a PM-et, hogy küldjön egy szignált. Ha a *NOFLSH* jelző nincs beállítva, akkor a sorban tárolt bemenet törlődik – a beérkezett karakterek vagy sorok számát nullázzák, és a bemeneti sor elejének és végének a címét tartalmazó mutatókat egyenlővé teszik. Ez az alapértelmezés szerinti tevékenység. Amikor a *SIGHUP* szignált venni kívánják, akkor a *NOFLSH*-t be lehet állítani, lehetővé téve azt, hogy az input és az output folytatódhasson a szignál vétele után. A *tty_icancel* (15000. sor) függvény feltétel nélkül kiüríti a várakozó bemenetet a *sigchar*-nál leírt módon, és ezenfelül meghívja a *tp->tty_icancel* által mutatott eszközfüggő függvényt, hogy az törölje a bemenő adatokat, amelyek magában az eszközben lehetnek, vagy az alacsony szintű kód puffereiben.

A *tty_init*-et (15013. sor) akkor hívják meg, amikor a *tty_task* először elindul. Sorban végigmegy a lehetséges terminálokon, és beállítja az alapértelmezés szerinti tulajdonságaikat. Kezdetben egy üres, semmilyen tevékenységet nem végző függvénynek a *tty_devnop*-nak címe kerül be a *tp->tty_icancel,* a *tp->tty_ocancel,* a *tp->tty_ioctl* és a *tp->tty_close* változókba. Ezután a *tty_init* meghívja a terminál kategóriájának (konzol, soros vonali vagy pszeudoterminál) megfelelő eszközfüggő inicializáló függvényeket. Ezek beállítják az indirekt módon hívott eszközfüggő függvények mutatóit valós mutatókra. Emlékezzünk arra, hogy ha egy kategórián belül egyáltalán nincs eszköz konfigurálva, akkor egy olyan makrót készítenek, amely azonnal visszatér, így nincs arra szükség, hogy egy nem konfigurált eszközhöz tartozó kód egyetlen részét is lefordítsuk. Az *scr_init* hívása inicializálja a konzol meghajtóját, és meghívja a billentyűzet inicializálását végző rutint is.

A következő három függvény az időzítőket támogatja. Egy felügyeleti időzítőt egy függvény címét tartalmazó mutatóval inicializálnak, hogy meghívja, amikor az idő lejár. A *tty_timed_out* az a függvény, amelyet a legtöbb időzítőhöz beállít a terminálmeghajtó. Beállítja az eseményjelzőket, hogy kikényszerítse mind a bemenet, mind pedig a kimenet feldolgozását. Az *expire_timers* kezeli a terminálmeghajtó időzítőket tartalmazó sorát. Emlékezzünk vissza, hogy cz az a függvény, amelyet a *tty_task* főciklusából hívnak meg, amikor egy *SYN_ALARM* üzenet érkezik. Egy könyvtári rutint, a *tmrs_exptimers*-t használják arra, hogy bejárja az időzítők csatolt listáját, lejárttá tegye őket, és meghívja a felügyeleti függvényét mindegyiknek, amelyik lejárt. Visszatérve a könyvtári függvényből, ha a bemeneti sor még aktív, egy sys_setalarm kernelhívást kezdeményeznek, ami egy másik *SYN_ALARM* üzenetet kér. Végül a *settimer* (15089. sor) beállítja az időzítőket arra, hogy nemkanonikus módban egy read hívásból mikor kell visszatérni. Ezt paraméterekkel hívják: a *tty_ptr* egy mutató egy *tty* struktúrára, az *enable* egy egész szám, amely *TRUE* vagy *FALSE* lehet. A *tmrs_settimer* és a *tmrs_clrtimer* könyvtári rutinokat arra, hogy letiltsanak vagy engedélyezzenek egy időzítőt, az *enable* paraméter szerint. Ha egy időzítő engedélyezett, akkor a felügyeleti függvény mindig a korábban leírt *tty_timed_out*.

A *tty_devnop* leírása (15125. sor) szükségszerűen hosszabb, mint a végrehajtható kód, mivel neki nincs ilyen. Ez egy tevékenység nélküli (no-operation) függvény, amelyet indirekt módon hívnak, ha egy eszköz nem igényel egy kiszolgálást. Láttuk, hogy a *tty_devnop*-ot a *tty_init* használja mint alapértelmezett értékeket különböző függvénymutatókhoz, mielőtt egy eszköz inicializáló rutinját meghívja.

A *tty.c*-ben lévő utolsó elem némi magyarázatot igényel. A select egy rendszerhívás, amelyet akkor használnak, amikor több I/O-eszköz kér kiszolgálást váratlan időpontokban egy processzustól. A klasszikus példa egy kommunikációs program, amelynek figyelemmel kell kísérnie a lokális billentyűzetet és egy távoli rendszert, amely talán modemen keresztül csatlakozik. A select hívás megengedi több eszközfájl megnyitását és figyelemmel kísérését, hogy mikor lehet róluk olvasni, vagy mikor lehet rájuk írni blokkolás nélkül. A select nélkül két processzus használatára lenne szükség a kétirányú kommunikáció kezelésére: egyik lenne a mester, és felügyelné az egyik irányban történő kommunikációt, a másik pedig a szolga, aki felügyelné a másik irányú kommunikációt. A select egy olyan tulajdonságra példa, amely nagyon jó hogy van, de amely lényegesen bonyolítja a rendszert. A MINIX 3 egyik tervezési célja az volt, hogy olyan egyszerű legyen, hogy elvárható időn belül, elvárható erőfeszítéssel megérthessék, ezért bizonyos határokat kellett szabnunk. Itt nem fogjuk tárgyalni a *do_select* (15135. sor) és a *select_try* (14313. sor), a *select_retry* (14348. sor) kiszolgáló rutinokat.

3.8.5. A billentyűzetmeghajtó megvalósítása

A következőkben figyelmünket a MINIX 3-konzolt működtető eszközspecifikus kódra fordítjuk, amely egy IBM PC-billentyűzetből és egy tárcímleképezéses megjelenítőből áll. Az ezt alkotó fizikai eszközök teljesen különállók: egy szokványos asztali számítógépben a megjelenítő egy illesztőkártyát használ (ebből legalább fél tucat különböző alaptípus létezik), amely a hátfalba van bedugva, míg a billentyűzetet az alaplapba épített áramkör szolgálja ki, amely a billentyűzetben levő 8 bites egylapkás célszámítógéppel tartja a kapcsolatot. A két aleszköz teljesen különböző szoftvertámogatást igényel, amely a *keyboard.c* és a *console.c* fájlokban található.

Az operációs rendszer a billentyűzetet és a konzolt ugyanazon eszköz, a */dev/console* részeinek látja. Ha elég nagy memóriával rendelkezik a képernyőillesztő, akkor a **virtuális konzol** támogatását végző kódot is lefordíthatjuk, és a */dev/console* mellett további logikai eszközök is lehetnek, a */dev/ttyc1, /dev/ttyc2* és így tovább. Bármely időben csak az egyikről érkező kimenet jelenik meg a képernyőn, és csak egy billentyűzetet használhatnak adatbevitelre, bármelyik konzol az aktív. Logikailag a billentyűzet a konzol alá van rendelve, de ez csak két, viszonylag kis dologban nyilvánul meg. Az első, hogy a *tty_table* táblázatban, amely a konzolhoz tartozó *tty* struktúrát tartalmazza, és ahol külön mezők állnak rendelkezésre a beolvasás és a kiírás számára, például a *tty_devread* és *tty_devwrite* mezők, a *keyboard.c*-ben, illetve a *console.c*-ben lévő függvénymutatókat bekapcsoláskor töltik fel. Azonban csak egy *tty_priv* mező van, és ez csak a konzol adatstruktúráira mutat. A második az, hogy a *tty_task* mielőtt belépne a főciklusába, meghív minden egyes eszközt egyszer, hogy inicializálja. A */dev/console* esetében meghívott kód a *console.c*-ben van, és a billentyűzetet inicializáló kód innen kerül meghívásra. Ezt az implicit hierarchiát akár meg is lehetne fordítani. A korábbiakban előbb mindig az adatbevitelt vizsgáltuk a kiírás előtt, amikor az I/O-eszközöket vizsgáltuk, és ezt a mintát fogjuk követni, amikor a *keyboard.c*-t tárgyaljuk ebben a fejezetben, a *console.c* tárgyalását pedig a következő fejezetre hagyjuk.

A *keyboard.c* ugyanúgy kezdődik, mint a legtöbb látott forrásfájl, jó néhány #include utasítással. Ezek közül azonban az egyik nem szokásos. A *keymaps/us-std.src* (a 15128. sorban illesztik be) nem egy szokásos definíciós fájl; ez egy C forrásfájl, amelynek eredményeként az alapértelmezés szerinti billentyűzettérképet egy kezdőértékekkel feltöltött tömbként összeállítják a *keyboard.o*-ban. A billentyűzettérkép forrásfájl néhány jellegzetes bejegyzése látható a 3.37. ábrán. Az #include utasítások után néhány makró következik, amelyek különféle konstansokat definiálnak. Az első csoportot a billentyűzetvezérlővel történő alacsony szintű kommunikációban használják. Közülük sok I/O-kapuk címe vagy olyan bitkombinációk, amelyeknek jelentősége van ezekben a kapcsolatokban. A következő csoport a speciális billentyűk szimbolikus neveit tartalmazza. A 15249. sorban a körkörös billentyűzetpuffer méretét definiálják szimbolikusan mint *KB_IN_BYTES*, amelynek az értéke 32, majd a következőkben magát a puffert és a kezelését végző változókat. Mivel csak egyetlenegy van ebből a pufferből, figyelmet kell fordítani arra, hogy a teljes tartalmát feldolgozzák, mielőtt a virtuális konzolt váltják.

A változók következő csoportját arra használják, hogy különböző állapotinformációkat tároljanak bennük, amelyeket meg kell jegyezni ahhoz, hogy egy billentyűlenyomást megfelelő módon értelmezzenek. Ezeket különböző módon használják fel. Például a *caps_down* jelző (15266. sor) a *TRUE* és a *FALSE* (igaz és hamis) értékek között váltakozik, valahányszor a CAPS LOCK billentyűt megnyomják. A *shift* jelzőt (15264. sor) *TRUE*-ra állítják, amikor valamelyik SHIFT billentyűt lenyomják, és *FALSE*-ra, amikor mindkét SHIFT billentyűt felengedik. Az *esc* változót akkor állítják be, ha egy billentyűkód-választás érkezik. A következő karakter megérkezése minden esetben törli az értékét.

A *map_key0* függvény (15297. sor) makróként van megvalósítva. Visszaadja a billentyűkódnak megfelelő ASCII kódot, figyelmen kívül hagyva a módosítókat. Ez megfelel a billentyűzettérkép-tömb első (SHIFT nélküli, normál) oszlopának. Ennek „nagy testvére" a *map_key* (15303. sor), amely elvégzi a billentyűkód teljes leképezését egy ASCII kódra, figyelembe véve a (több) lenyomott módosító billentyűt, a közönséges billentyűk mellett.

A billentyűzetmegszakítás-kezelő rutin, a *kbd_interrupt* (15335. sor) kerül meghívásra valahányszor egy billentyűt lenyomnak vagy felengednek. Ez meghívja az *scode*-ot, hogy elkérje a billentyűkódot a billentyűzetvezérlő lapkából. A billentyűkód legmagasabb helyi értékű bitje be van állítva, ha egy billentyű felengedése okozta a megszakítást, az ilyen kódokat figyelmen kívül lehet hagyni, hacsak nem az egyik módosítóbillentyűről van szó. Azonban annak érdekében, hogy a legkevesebb munkát végezzük azért, hogy a megszakításkérést a lehető leggyorsabban szolgáljuk ki, minden nyers billentyűkódot a körkörös pufferbe tesznek, és az aktuális konzol *tp->tty_events* jelzőjét beállítják (15350. sor). Céljaink érdekében

42	35	163	170	18	146	38	166	38	166	24	152	57	185
L+	h+	h–	L–	e+	e–	l+	l–	l+	l–	o+	o–	SP+	SP–

54	17	145	182	24	152	19	147	38	166	32	160	28	156
R+	w+	w–	R–	o+	o–	r+	r–	l+	l–	d+	d–	CR+	CR–

3.41. ábra. *A billentyűkódok a bemenő pufferben, a megfelelő leütésekkel, amelyek a billentyűzetről bevitt sor karaktereinek felelnek meg. Az* L *és* R *a bal, illetve jobb oldali* SHIFT *billentyűt jelentik, + és – a billentyű lenyomását, illetve felengedését. A billentyű felengedésének billentyűkódja 128-cal nagyobb, mint ugyanezen billentyű lenyomásának kódja*

feltesszük, ahogyan korábban is tettük, hogy select hívások nem történnek, és hogy a *kbd_interrupt* ezután azonnal visszatér. A 3.41. ábra a pufferben levő billentyűkódokat mutatja egy rövid bemeneti sor esetében, amely két nagybetűs karaktert tartalmaz, mindegyiket megelőzi a SHIFT lenyomásának billentyűkódja, és követi a SHIFT felengedésének billentyűkódja. Kezdetben mindkét billentyűlenyomás és -felengedés kódját tárolják.

Amikor egy *HARD_INT* megszakításkérés érkezik a billentyűzettől a *tty_task*-hoz, a teljes főciklus nem fut le. Egy continue utasítás a 13795. sorban a főciklus egy újabb futását váltja ki azonnal a 13764. sorban. (Ez kissé leegyszerűsített, a forráskód listában benne hagytunk némi feltételes kódot, hogy bemutassuk: ha a soros vonali meghajtó is engedélyezve van, akkor annak a felhasználó oldalán lévő megszakításkezelőjét is meg kell hívni.) Amikor a vezérlés a ciklus elejére kerül, és most a konzoleszköz *tp->tty_events* jelzője be van állítva, akkor meghívják a *kb_read* (15360. sor) eszközfüggő rutint a konzol *tty* struktúrájának *tp->tty_devread* mezőjében levő mutató segítségével.

Billentyű	Scan kód	„ASCII"	Vezérlőszekvencia
HOME (kezdet)	71	0x101	ESC [H
Up Arrow (felfelé nyíl)	72	0x103	ESC [A
PGUP (lapozás felfelé)	73	0x107	ESC [V
–	74	0x10A	ESC [S
Left Arrow (balra nyíl)	75	0x105	ESC [D
5	76	0x109	ESC [G
Right Arrow (jobbra nyíl)	77	0x106	ESC [C
+	78	0x10B	ESC [T
END (vége)	79	0x102	ESC [Y
Down Arrow (lefelé nyíl)	80	0x104	ESC [B
PGDN (lapozás lefelé)	81	0x108	ESC [U
INS (beszúrás)	82	0x10C	ESC [@

3.42. ábra. *A numerikus billentyűzet által generált vezérlőszekvenciák. Amikor a közönséges billentyűk billentyűkódját ASCII kódokká konvertálják, a speciális billentyűk „pszeudo-ASCII" kódokat kapnak, amelyeknek az értéke 0xFF-nél nagyobb*

A *kb_read* kiveszi a billentyűkódokat a billentyűzet körkörös pufferéből, és ASCII kódokat helyez el a saját lokális pufferébe, amely elég nagy ahhoz, hogy elférjenek benne azok a vezérlőszekvenciák is, amelyeket a numerikus billentyűzetről érkező billentyűkódok esetén kell generálni. Ezután meghívja a hardverfüggetlen *in_process*-t, hogy az a karaktereket a bemeneti sorba tegye. A 15379. sorban az *icount* értékét eggyel csökkentik. A *make_brake* az ASCII kódot egy egész számként adja vissza. Néhány speciális billentyűnek, mint például a numerikus vagy a funkcióbillentyűknek, itt még 0xFF-nél nagyobb kódja van. A *HOME* és az *INSRT* (0x101 és 0x10C, az *include/minix/keymap.h*-ban van definiálva) közé eső kódokat, amelyeket a numerikus billentyűzeten levő billentyűk lenyomása eredményez, háromkarakteres vezérlőszekvenciákká konvertálják a 3.42. ábrán látható *numpad_map* tömb segítségével.

Ezután a szekvenciát átadják az *in_process*-nek (15392–15397. sor). Nagyobb kódokat nem adnak át az *in_process*-nek. Helyette egy ellenőrzést végeznek az ALT-LEFT-ARROW, ALT-RIGHT-ARROW, valamint ALT-F1–F12 közé eső kódokra, és ha egyiket megtalálják, meghívják a *select_console*-t a virtuális konzol váltására. A CTRL-F1–F12 billentyűknek ugyanilyen módon adnak speciális jelentést. A CTRL-F1 megmutatja a funkcióbillentyűk leképezéseit (lásd később). A CTRL-F3 vált a hardveres és a szoftveres görgetés között a konzol megjelenítőjén. A CTRL-F7, CTRL-F8 és a CTRL-F9 szignálokat generál, amelyeknek ugyanaz a hatása, mint a CTRL-\, a CTRL-C és a CTRL-U-nak ebben a sorrendben, kivéve, hogy ez a hozzárendelés nem változtatható meg az stty paranccsal.

A *make_break* (15431. sor) a billentyűkódokat ASCII kódokká alakítja, majd beállítja azoknak a változóknak az értékét, amelyek a módosítóbillentyűk állapotát követik. Először azonban ellenőrzi a „mágikus" CTRL-ALT-DEL kombinációt, amelyet minden PC-felhasználó az MS-DOS alatt a rendszer újraindításának egyik módjaként ismer. Emlékezzünk arra a megjegyzésre, hogy ezt egy alacsonyabb szinten jobb volna kezelni. Azonban a MINIX 3 megszakításkezelésének egyszerűsége a kernel területén lehetetlenné teszi a CTRL-ALT-DEL észlelését ott, amikor a megszakításkérést elküldik, a billentyűkódot még nem olvasták be.

Mivel szabályos rendszerleállítást szeretnénk, ezért a PC BIOS-rutinjainak meghívása helyett a sys_kill kernelhívást hajtjuk végre, ami kezdeményezi a *SIGKILL* szignál küldését az *init*-nek, az összes processzus szülő processzusának (15448. sor). Az *init* fogadja ezt a szignált, és úgy értelmezi, mint egy szabályos rendszerleállítás megkezdésére vonatkozó parancsot, még mielőtt visszatérne a betöltési felügyelőprogramhoz, ahonnan a rendszer teljes újraindítása vagy a MINIX 3 újratöltése végrehajtható.

Természetesen nem reális azt elvárni, hogy ez minden esetben működik. A legtöbb felhasználó tisztában van a hirtelen leállás veszélyeivel, és nem nyomja meg a CTRL-ALT-DEL-t egészen addig, amíg valami tényleg el nem romlik, és a rendszer normális irányítása lehetetlenné válik. Ilyenkor elképzelhető, hogy a rendszer már annyira összezavarodott, hogy egy másik processzus számára szignált küldeni már nem lehetséges. Ezért van egy *static* változó, a *CAD_count* a *make_break*-ben. A legtöbb rendszerhiba esetén a megszakításrendszer még működik, így a billentyűzetről még érkezhet bemenő adat és a terminálmeghajtó működésben maradhat.

A MINIX 3 kihasználja a számítógép-felhasználók viselkedését, akik előszeretettel csapkodják a billentyűket, ha valami nem akar megfelelően működni (lehet, hogy ez az egyik bizonyítéka annak, hogy az őseink a majmok voltak). Ha az *init* leállítása nem sikerül, és a felhasználó még kétszer megnyomja a CTRL-ALT-DEL kombinációt, akkor egy sys_abort kernelhívás történik, amelynek következtében a betöltési felügyelőprogramhoz kerül vissza a vezérlés, az *init* hívását megkerülve.

A *make_break* fő része könnyen érthető. A *make* változó tárolja azt, hogy a billentyűkódot a billentyű lenyomása vagy felengedése váltotta ki, és ezután a *map_key* meghívása visszaadja az ASCII kódot a *ch*-ba. A következő utasítás egy, a *ch* értéke szerinti switch (15460–15499. sor). Tekintsünk két esetet, egy közönséges és egy speciális billentyű leütését. Egy közönséges billentyű esetén a swtich-ben egyik eset sem teljesül, így a default esetben (15498. sor) visszaadják a billentyű kódját, ha a *make* változó értéke igaz. Ha valahogyan egy közönséges billentyűt billentyűfelengedésként fogadtak, akkor a –1 értéket adják vissza, és ezt a hívó *kb_read* figyelmen kívül hagyja. Egy speciális billentyűt, mint például a *CTRL,* a switch egy megfelelő helyén azonosítanak, jelen esetben a 15461. sorban. A megfelelő változó, ebben az esetben a *ctrl,* megjegyzi a *make* állapotát és a visszaadásra kerülő karakterkód helyére a –1-et helyettesíti (amelyet majd figyelmen kívül hagynak). Az *ALT, CALOCK, NLOCK* és az *SLOCK* billentyűk kezelése bonyolultabb, de mindegyik speciális billentyűre az eredmény hasonló: egy változó felveszi vagy a billentyű állapotát (azoknál, amelyeknek csak akkor vannak hatása, amíg lenyomva tarják őket), vagy az ellenkezőjére változtatja az állapotát (a lock billentyűknél).

Még egy vizsgálatra érdemes eset van: az *EXTKEY* kód és az *esc* változó. Ez utóbbi nem tévesztendő össze a billentyűzeten levő ESC billentyűvel, amely a 0x1B ASCII kódot adja vissza. Nincs mód rá, hogy az *EXTKEY* kódot a billentyűzeten levő egy vagy több billentyű lenyomásának kombinációjával előállítsuk; ez a PC billentyűzetének **kiterjesztett billentyűelőtagja (prefix),** a kétbájtos billentyűkódnak az első bájtja jelzi, hogy egy billentyű nem volt része az eredeti PC-billentyűzethez tartozó billentyűknek, de ugyanaz a billentyűkódja, mint annak, amit megnyomtak. Sok esetben a szoftverek a két kódot egyformán kezelik. Például majdnem minden esetben ez történik a normál „/" és a numerikus billentyűzeten levő „/" esetén. Más esetekben szükség lehet a két billentyű megkülönböztetésére. Például sok, az angol nyelvűtől eltérő billentyűzettérkép eltérően kezeli a bal és a jobb oldali ALT billentyűket, így megenged olyan billentyűket, amelyeknek három különböző karakterkódot kell előállítaniuk. Mindkét ALT ugyanazt a billentyűkódot (56) generálja, de a jobb oldali lenyomásakor ezt megelőzi az *EXTKEY* kód. Amikor az *EXTKEY* kódot adják vissza, az *esc* jelző beállításra kerül. Ebben az esetben a *make_break* kilép a switch utasításból, kihagyva ezzel az utolsó lépést a normál visszatérés előtt, amely minden más esetben törli az *esc* jelzőt (15458. sor). Ez azt eredményezi, hogy az *esc* csak a közvetlenül ez után érkező kód esetén lesz érvényben. Ha tisztában van a PC billentyűzetének működési elveivel, akkor ez egyszerre ismerős, de mégis egy kicsit furcsa, mivel a PC BIOS nem engedi meg, hogy valaki olvassa egy ALT billentyű kódját, és kiterjesztett kód esetén más értéket ad vissza, mint a MINIX 3.

A *set_leds* (15508. sor) kapcsolja be és ki a világító diódákat, amelyek jelzik, hogy lenyomták-e a NUM LOCK, CAPS LOCK vagy SCROLL LOCK billentyűket. Egy vezérlő bájt, a *LED_CODE* kerül egy kimeneti portra, jelezve a billentyűzetnek, hogy a következő bájt, amely megjelenik, a fényeket fogja vezérelni, és a három világító dióda állapota a következő bájt három bitjén lesz kódolva. Természetesen ezeket a műveleteket kernelhívások hajtják végre, amelyek kérik a rendszertaszkot, hogy írjon a kimeneti portokra. A következő két függvény támogatja ezt a műveletet. A *kb_wait* (15530. sor) hívását arra használják, hogy megállapítsák, a billentyűzet kész-e parancssorozatok fogadására, és a *kb_ack* (15552. sor) hívásával ellenőrzik, hogy a parancsot nyugtázták-e. Mindkét parancs tevékenyen várakozik, folyamatosan olvas a kívánt kód megérkezéséig. Ez egy nem ajánlott technika a legtöbb I/O-művelet esetén, de a billentyűzeten levő LED-ek be- és kikapcsolását nem kell túl gyakran végezni, így ha ezt nem hatékonyan végezzük, nem veszítünk sok időt. Figyeljük meg, hogy mind a *kb_wait*, mind a *kb_ack* meghiúsulhat, és a visszatérési kódjukból megállapítható, ha ez történt. Az időkorlátokat úgy kezelik, hogy korlátozzák az ismétlések számát a ciklusban egy számlálóval. Azonban a billentyűzeten levő LED-ek kapcsolgatása nem annyira fontos, hogy ezekkel a visszatérési értékekkel bajlódjunk, így a *set_leds* vakon megy tovább.

Mivel a billentyűzet a konzol részét képezi, így az inicializáló rutinja, a *kb_init* (15572. sor) a *console.c*-ben levő *scr_init*-ből hívódik meg, nem közvetlenül a *tty.c*-ben levő *tty_init*-ből. Ha a virtuális konzolokat engedélyezték (azaz az *NR_CONS* nagyobb, mint 1 az *include/minix/config.h*-ban), akkor a *kb_init*-et minden egyes logikai konzolra meghívják. A következő függvényt a *kb_init_once*-t csak egyszer hívják meg (15583. sor), amint a nevéből következik. Ez állítja be a világító diódákat a billentyűzeten, és ellenőrzi a billentyűzetet, hogy ne legyen elmaradt, beolvasásra váró billentyűleütés. Majd két tömböt tölt fel kezdőértékekkel, az *fkey_obs*-t és az *sfkey_obs*-t, amelyeket arra használnak, hogy a funkcióbillentyűket olyan processzusokhoz kapcsolják, amelyeknek reagálni kell rájuk. Amikor mindez készen van, két kernelhívást hajt végre, meghívja a sys_irqsetpolicy és a sys_irqenable funkciókat, amelyekkel beállítja a billentyűzetmegszakítás kezelését, automatikus újraengedélyezést állít be, így minden esetben egy figyelmeztető üzenetet küldenek a *tty_task* számára, amikor egy billentyűt lenyomnak vagy felengednek.

Bár a következőkben több lehetőségünk lesz arra, hogy megvitassuk, hogyan működnek a funkcióbillentyűk, itt a jó alkalom az *fkey_obs* és az *sfkey_obs* tömbök leírására. Mindegyiknek 12 eleme van, mivel a modern PC-billentyűzeteken 12 F billentyű van. Az első tömböt a módosító billentyűk nélküli F billentyűk, a másodikat a SHIFT-tel módosított F billentyűk esetén használják. Mindegyikük *obs_t* típusú elemekből áll, ami olyan struktúra, amely egy processzusszámot és egy egész számot tárol. Ez a struktúra és ezek a tömbök a *keyboard.c* fájlban vannak deklarálva a 15279–15281. sorokban. Az inicializálás egy speciális, szimbolikus *NONE*-nal jelölt értéket helyez el a struktúra *proc_nr* komponensében, amivel jelzi, hogy az nincs használatban. A *NONE* olyan érték, amely kívül esik az érvényes processzusszámok tartományán. Jegyezzük meg, hogy a processzusszám nem egy *pid,* egy bejegyzést azonosít a processzustáblában. Ez az elnevezés félreérthető lehet. Azonban üzenet küldésére egy processzusszámot alkalmaznak inkább, mint

egy *pid*-et, mert a processzusszámok indexelik a *priv* táblát, amely megadja, hogy a processzus számára meg van-e engedve, hogy üzeneteket fogadjon. Az *events* egész szám egy számláló, amelyet kezdetben nullára állítanak. Arra fogják használni, hogy eseményeket számláljanak vele.

A következő három függvény mind elég egyszerű. A *kbd_loadmap* (15610. sor) majdnem triviális. Ezt a *tty*.c-ben levő *do_ioctl* hívja meg akkor, amikor a felhasználói adatterületről egy billentyűzettérképet át kell másolni, és felül kell írni az alapértelmezés szerinti billentyűzettérképet. Az alapértelmezett térkép a *keyboard.c* fájl elejére beillesztett térképforrásfájl fordításával jön létre.

A MINIX első megjelenése óta mindig biztosította különböző rendszer-információk mentését vagy más speciális tevékenységeket a rendszerkonzolon az F1, F2 stb. billentyűk lenyomására. Más operációs rendszerek általában nem nyújtanak ilyen szolgáltatást, de a MINIX mindig is oktatási eszköz kívánt lenni. A felhasználók bátran barkácsolhatnak vele; ez azt jelenti, hogy a felhasználóknak segítségre lehet szükségük a hibakereséshez. Sok esetben az F billentyűkkel előállított kimenet akkor is hozzáférhető, ha a rendszer már összeomlott. A 3.43. ábra összefoglalja ezeket a billentyűket és a funkcióikat.

Ezeknek a billentyűknek két csoportja van. Ahogyan korábban megjegyeztük a CTRL-F1–F12 billentyűket a *kb_read* detektálja. Ezek olyan eseményeket váltanak ki, amelyeket a terminálmeghajtó kezelni tud. Ezek az események nem szükségképpen jelenítenek meg memóriaképet. Valójában csak a CTRL-F1 ad egy információs képernyőt; megjeleníti a funkcióbillentyűkhöz rendelt tevékenységeket. A CTRL-F3 váltja a hardveres és a szoftveres görgetést a konzolképernyőn, a többi pedig szignálokat küld.

Billentyű	Célja
F1	A kernel processzustáblájának megjelenítése
F2	A processzus memóriahasználatának megjelenítése
F3	Betöltési memóriakép
F4	A processzus privilégiumai
F5	Betöltési felügyelőprogram paraméterei
F6	Megszakításkezelők és rendszabályok
F7	Kernelüzenetek
F10	Kernelparaméterek
F11	Időzítési adatok (ha engedélyezve van)
F12	Ütemezési sorok
SF1	A processzuskezelő processzustáblája
SF2	Szignálok
SF3	A fájlrendszer processzustáblája
SF4	Eszköz-/meghajtó-hozzárendelés
SF5	Nyomtatóbillentyű-hozzárendelések
SF9	Ethernet-statisztika (csak RTL8139 esetén)
CF1	Billentyűzettérkép megjelenítése
CF3	Hardveres/szoftveres görgetés váltás a konzolon
CF7	SIGQUIT küldése; ugyanaz, mint a CTRL-\
CF8	SIGINT küldése; ugyanaz, mint a CTRL-C
CF9	SIGKILL küldése; ugyanaz, mint a CTRL-U

3.43. ábra. *A* func_key() *által felismert funkcióbillentyűk*

A funkcióbillentyűk magukban vagy a SHIFT billentyűvel együtt lenyomva olyan eseményeket váltanak ki, amelyeket a terminálmeghajtó nem tud kezelni. Eredményezhetnek figyelmeztető üzeneteket egy kiszolgálónak vagy egy meghajtónak. Mivel a szerverek és a meghajtók betölthetők, engedélyezhetők és letilthatók, miután a MINIX 3 már elindult, a billentyűk fix hozzárendelése a fordítás alkalmával nem lenne kielégítő. A futás alatti változtatás engedélyezéséhez a *tty_task* *FKEY_CONTROL* típusú üzeneteket fogad. A *do_fkey_ctl* függvény (15624. sor) szolgálja ki az ilyen kéréseket. A kérések típusai: *FKEY_MAP*, *FKEY_UNMAP* vagy *FKEY_EVENTS*. Az első kettő beregisztrál vagy töröl egy processzust a megadott kulccsal az üzenetben megadott térképbe, a harmadik üzenettípus pedig visszaad egy a hívóhoz tartozó térképet a lenyomott billentyűkhöz, és nullázza az ezekhez a billentyűkhöz tartozó *events* mezőt. Egy kiszolgáló processzus, az **információs szerver (information server, IS)** inicializálja a betöltési memóriaképben található processzusok beállításait, és közreműködik a válaszok generálásában. De egyedi meghajtók ugyancsak beregisztrálhatják magukat, hogy válaszolhassanak egy funkcióbillentyűre. A hálózati meghajtók általában ezt teszik, mivel a csomagstatisztikákat bemutató táblázat hasznos lehet a hálózati problémák megoldásában.

A *func_key*-t (15715. sor) a *kb_read* hívja annak meghatározására, hogy egy olyan speciális billentyűt nyomtak-e le, amely helyi feldolgozást kíván. Ezt minden kapott billentyűkódra megvizsgálják. Ha az nem funkcióbillentyű, akkor még legfeljebb három összehasonlítást tesznek, mielőtt a vezérlés visszakerül a *kb_read*-hez. Ha a funkcióbillentyű regisztrált, akkor a megfelelő processzusnak egy figyelmeztető üzenetet küldenek. Ha a processzus olyan, hogy csak egyetlen billentyűt regisztrált be, akkor a figyelmeztetés önmagában is elegendő a processzus számára, hogy tudja, mit kell tennie. Ha processzus az információs kiszolgáló vagy olyan, ami több billentyűt regisztrált be, akkor egy párbeszédre van szükség – a processzusnak egy *FKEY_EVENTS* kérést kell küldenie a terminálmeghajtónak, hogy dolgozza fel a *do_fkey_ctl*-lel, ez pedig tájékoztatja a hívót, hogy mely billentyűk az aktívak. A hívó azután már eljuthat minden lenyomott billentyűhöz rendelt rutinhoz.

A *scan_keyboard* (15800. sor) a hardverkapcsolat szintjén működik, olvassa és írja az I/O-kapukat. A billentyűzetvezérlő a 15809–15810. sorokban értesül arról, hogy egy karakter beérkezett, beolvas egy bájtot, majd visszaírja úgy, hogy a legfelső helyi értékű bitjét 1-re állította, és ezután még egyszer újraírja úgy, hogy ugyanezt a bitet kinullázta. Ez megakadályozza azt, hogy ugyanazt az adatot kétszer beolvassuk egy következő olvasásnál. Nincs állapot-ellenőrzés a billentyűzet olvasásánál, de ez semmikor sem okozhat problémát, mivel a *scan_keyboard*-ot csak egy megszakítás eredményeként hívják meg.

A *keyboard.c*-ben levő utolsó függvény a *do_panic_dumps* (15819. sor). Ha egy rendszerpánik eredményeként kerül végrehajtásra, akkor a felhasználó számára biztosítja azt a lehetőséget, hogy a funkcióbillentyűket használva nyomkövetési

információkat jelenítsen meg a képernyőn. A 15830–15854. sorokban levő ciklus ismét a tevékeny várakozásra példa. A billentyűzetet olvassuk addig, amíg egy ESC-t meg nem nyomnak. Természetesen senki sem panaszkodhat, hogy a rendszer összeomlása után hatékonyabb technikára lenne szükség, amíg az újraindítási parancsra vár. A cikluson belül egy ritkán használt, blokkolással nem járó olvasási műveletet, az nb_receive-t használnak, hogy lehetőség legyen más üzenetek fogadására is, ha vannak ilyenek, és a billentyűzet ellenőrzésére: van-e bemenet, amelynek várhatóan a következő üzenetben ajánlott lehetőségek egyikének kellene lennie. Az üzenet akkor jelenik meg a képernyőn, ha belépünk ebbe a függvénybe.

```
Nyomjon ESC-t az újratöltéshez, DEL-t a leállításhoz, F billentyűket a rendszer
nyomkövetéséhez.
```

A következő alfejezetben láthatjuk azt a kódot, amely megvalósítja a *do_newkmess* és a *do_diagnostics* függvényeket.

3.8.6. A képernyőmeghajtó megvalósítása

Az IBM PC megjelenítőjét virtuális terminálokként konfigurálhatjuk, ha elegendő memória áll rendelkezésre. Ebben a szakaszban a konzol eszközfüggő kódját fogjuk megvizsgálni. Ezenkívül megnézzük azokat a nyomkövetési (debug) kiíró rutinokat, amelyek a billentyűzet és a képernyő alacsony szintű szolgáltatásait használják. Ezek akkor is lehetőségeket adnak a konzolnál ülő felhasználóval történő korlátozott kapcsolatra, amikor MINIX 3 más részei már nem működnek, és hasznos információkkal szolgálnak még egy majdnem teljes rendszerösszeomlás esetén is.

A konzolkiírás hardverfüggő megvalósítása a PC tárcímleképezéses képernyőjére a *console.c*-ben van. A *console* struktúrát a 15981–15998. sorokban definiálták. Bizonyos értelemben ez a struktúra a *tty.c*-ben definiált *tty* struktúra egy kiterjesztése. Inicializáláskor a konzol *tty* struktúrájának *tp->tty_priv* mezője egy olyan mutató lesz, amely a saját *console* struktúrájára mutat. A *console* struktúra első eleme egy mutató, amely a megfelelő *tty* struktúrára mutat vissza. A *console* struktúra komponensei pontosan azok, amik egy képernyős megjelenítő esetén várhatók: a kurzor sor- és oszlopértékét rögzítő változók, a képernyő-memória kezdőcíme és a megjelenítéshez használható memória méretkorlátja, az a memóriacím, amely a vezérlő mikroáramkör báziscímmutatójába kerül, valamint a kurzor aktuális címe. A többi változót a vezérlőszekvenciák kezeléséhez használják. Miután a karakterek 8 bites bájtokként érkeznek, és össze kell őket kapcsolni az attribútumbájtokkal, és ezeket a 16 bites szavakat kell átvinni a videomemóriába, az átvitelre kerülő blokkot a *c_ramqueue*-ban építjük fel, amely elég nagy tömb ahhoz, hogy tárolja egy teljes 80 karakteres sor 16 bites karakter-attribútum párjait. Minden virtuális konzolnak szüksége van egy *console* struktúrára, és ezek számára a *cons_table* tömbben (16001. sor) foglalunk helyet. Amint azt a *tty* és más struktúrák esetén is tettük, a *console* struktúra elemeire rendszerint egy mutatóval fogunk hivatkozni, például *cons->c_tty*.

Az a függvény, amelynek a címe minden egyes konzol *tp->devwrite* elemében található, a *cons_write* (16036. sor). Ezt csak egy helyről hívják, a *tty.c*-ben található *handle_events* függvényéből. A *console.c*-ben lévő legtöbb függvény ennek a támogatására szolgál. Amikor ezt a függvény először hívják meg, miután a kliens processzus egy write hívást kezdeményezett, a kiírásra kerülő adat a kliens pufferében található, amelyet a *tty* struktúra *tp->tty_outproc* és a *tp->out_vir* mezőinek segítségével találunk meg. A *tp->outleft* azt mondja meg, hogy hány karaktert kell átvinnünk, és a *tp->tty_outcum* mező kezdetben nulla, jelezve azt, hogy még egyet sem vittünk át. Ez a szokásos helyzet a *cons_write*-ba történő belépéskor, mert normális körülmények között, ha egyszer meghívásra került, akkor az eredeti hívásban kért összes adatot átviszi. Azonban ha a felhasználó le akarja lassítani az adatok képernyőn való megjelenítését, leütheti a *STOP* (CTRL-S) karaktert a billentyűzeten, amelynek az eredménye az lesz, hogy beállítódik a *tp->inhibited* jelző. A *cons_write* azonnal visszatér, ha ez a jelző be van állítva, még akkor is, ha a write még nem fejeződött be. Ilyen esetben a *handle_events* folytatni fogja a *cons_write* meghívását, és amikor a *tp->tty_inhibited* jelző végre törlődik azzal, hogy a felhasználó leüti a *START* (CTRL-Q) billentyűt, a *cons_write* folytatni tudja a megszakított adatátvitelt.

A *cons_write* egyetlen paramétere egy mutató az adott konzol *tty* struktúrájára, tehát az első teendő a *cons* változó inicializálása úgy, hogy az az adott konzol *console* struktúrájára mutasson (16049. sor). Eztán, mivel a *handle_events* hívja a *cons_write*-ot, amikor elindul, az első tevékenység annak eldöntése, hogy van-e egyáltalán elvégzendő feladat. Ha nincs, akkor egy gyors visszatérés történik (16056. sor). Ezt követően a 16061–16089. sorok közötti főciklusba érkezünk. Ez a ciklus felépítésében hasonló a *tty.c*-ben levő *in_transfer* főciklusához. Egy 64 karaktert tárolni képes lokális puffert töltenek meg a *sys_vircopy* kernelhívással, hogy átvegyék az adatokat a kliens pufferéből. Ezt követően a forrásterület mutatóját és a számlálókat aktualizáljuk, majd a lokális pufferből minden egyes karaktert átviszünk a *cons->c_ramqueue* tömbbe a megfelelő attribútumbájttal együtt, hogy később a képernyőre kerülhessenek a *flush* segítségével.

Amint azt a 3.35. ábra mutatja, a karakterátvitelt a *cons->ramqueue*-ba többféle módon is megtehetjük. Minden egyes karakterre meghívhatjuk az *out_char*-t, de megjósolható, hogy az *out_char* egyik speciális szolgáltatására sem lesz szükség, ha a karakter egy látható karakter, vezérlőszekvencia nincs folyamatban, a képernyő szélességét nem haladtuk meg, és a *cons->c_ramqueue* nincs tele. Ha az *out_char* teljes szolgáltatáskészletére nincs szükség, a karaktert közvetlenül beleteszik a *cons->c_ramqueue*-ba az attribútumbájtjával együtt (amelyet a *cons->c_attr*-ból kaphatunk meg) és a *cons->c_rwords*-öt (a kimeneti sor indexe), a *cons->c_column*-ot (a képernyőn a kurzor oszlop mutatója) és a *tbuf*-ot (egy mutató a pufferbe) mind megnövelik. A karakter ilyen direkt módon történő behelyezése a *cons->ramqueue*-ba megfelel a 3.35. ábra bal oldalán látható szaggatott vonalnak. Ha szükséges, akkor az *out_char*-t (16082. sor) hívják meg. Ez elvégzi az összes adminisztrációt, továbbá meghívja a *flush*-t, amely elvégzi a végső átvitelt a képernyő-memóriába, ha szükséges.

Az átvitel a felhasználói pufferből a lokális pufferbe, majd a kimeneti sorba mindaddig ismétlődik, amíg a *tp->tty_outleft* azt mutatja, hogy van még átvien-

dő karakter, és a *tp->tty_inhibited* jelző nincs beállítva. Amikor az átvitel megáll, akár azért, mert a write művelet befejeződött, vagy azért, mert a *tp->tty_inhibited* beállítódott, ismét meghívják a *flush*-t, hogy a sorban levő utolsó karakterek is átvitelre kerüljenek a képernyő-memóriába. Ha a művelet befejeződött (amelyet úgy ellenőriznek, hogy a *tp->tty_outleft* nulla), egy válaszüzenetet küldenek a *tty_reply* meghívásával (16096–16097. sor).

A *handle_events*-ből történő *cons_write* hívásokon kívül kiírandó karaktereket küldhetnek a konzolra az *echo* és *rawecho* függvények is a terminálmeghajtó eszközfüggetlen részében. Ha a konzol az aktuális kimeneti eszköz, akkor a *tp->tty_echo* mutatón keresztül történő hívásokat a következő, *cons_echo* (16105. sor) függvényhez irányítják. A *cons_echo* minden feladatát az *out_char*-t, majd a *flush* meghívásával végzi. A billentyűzetről a bemenet karakterenként érkezik, és a gépelő személy az echózást észrevehető késleltetés nélkül szeretné látni, ezért a karakterek kimeneti sorba helyezése nem lenne kielégítő.

Az *out_char* (16119. sor) egy ellenőrzést végez, hogy egy vezérlőszekvencia van-e folyamatban, meghívja a *parse_escape*-t, és azonnal visszatér, ha igen (16124–16126. sor). Egyébként egy switch utasítással megvizsgálja a speciális eseteket: null, visszatörlés, csengetés karakter és így tovább. Legtöbbjük kezelését egyszerű megérteni. A soremelés és a tabulátor kezelése a legbonyolultabb, hiszen ezek eredményezhetnek bonyolult változást a képernyőkurzor pozíciójában és igényelhetnek görgetést. Az utolsó ellenőrzés az *ESC* kódra történik. Ha ez érkezett, akkor a *cons->c_esc_state* jelzőt beállítják (16181. sor), és így az *out_char* további hívásai a *parse_escape*-re terelődnek át, amíg a szekvencia be nem fejeződik. Végül az alapértelmezés szerinti tevékenységet hajtják végre a nyomtatható (látható) karakterekre. Ha a képernyő szélességét meghaladtuk, akkor szükség lehet a képernyő görgetésére és a *flush* meghívására. Mielőtt egy karaktert a kimeneti sorba tesznek, megvizsgálják, hogy a sor nem telt-e meg, és ha igen, meghívják a *flusht*-t. Egy karakternek a sorba helyezése ugyanazokat az adminisztrációs feladatokat követeli meg, mint amilyeneket korábban a *cons_write*-ban láttunk.

A következő függvény a *scroll_screen* (16205. sor). A *scroll_screen* kezeli mind a felfelé görgetést – azt a normál esetet, amelyet akkor kell végrehajtani, ha a képernyő alján levő sor megtelt –, mind a lefelé görgetést, ami akkor fordul elő, amikor a kurzormozgató parancsok megkísérlik a kurzort a képernyő legfelső sora fölé mozgatni. Mindkét irányú görgetés megvalósításra három lehetséges módszer ígérkezik. Ezeket a különböző videokártyáknak támogatniuk kell.

Vizsgáljuk meg a felfelé görgetés esetét. Kezdetben a *chars* értéke a képernyőn levő karakterek száma, levonva belőle 1 sornak megfelelő számú karaktert. A szoftveres görgetést a *vid_vid_copy* egyetlen olyan hívásával végezzük, amely *chars* számú karaktert mozgat a memóriában lejjebb, annyival, ahány karakter van egy sorban. A *vid_vid_copy* kezeli az átfordulást, azaz, ha olyan memóriablokk mozgatását kérik, amely túlcsordul a videomegjelenítő memóriájának felső végén, akkor veszi a túlcsorduló részt a memóriablokk alsó címéről, és egy olyan címre mozgatja, amely magasabb címen található, mint az a rész, amelyet lejjebb mozgatunk, azaz az egész blokkot körkörös tömbként kezeli. A hívás egyszerűsége azonban egy elég lassú műveletet takar, még akkor is, ha a *vid_vid_copy* egy assembly nyelvű rutin (amely a *drivers/tty/vidcopy.s*-ben van definiálva). Ez a hívás 3840 bájt mozgatását kívánja a CPU-tól, ami nagy munka, még assembly nyelven is.

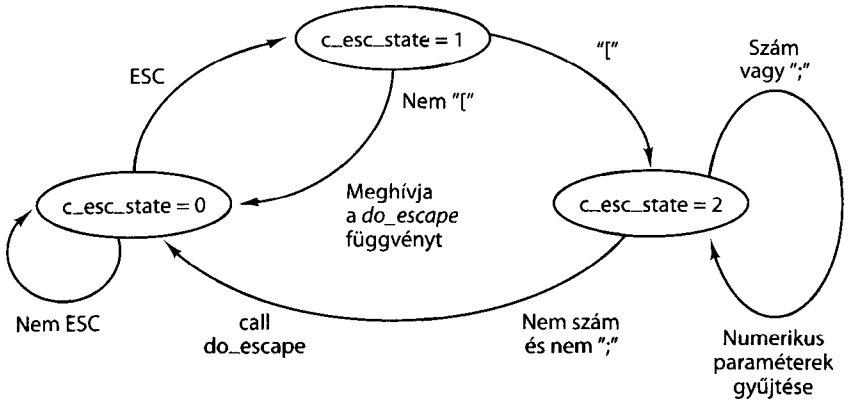
A szoftveres görgetési módszer soha nem az alapértelmezett: az operátornak kell ezt kiválasztania, ha a hardveres görgetés nem működik vagy nem kívánatos valamilyen ok miatt. Egyik ok lehet az, ha a *screendump* utasítást kívánjuk használni akár azért, hogy a képernyő-memória tartalmát el tudjuk menteni egy fájlba, vagy a főkonzol képernyőjét szeretnénk látni, amikor egy távoli terminálról dolgozunk. Ha a hardveres görgetés van érvényben, akkor a *screendump* valószínűleg váratlan eredményt ad, mivel a képernyő-memória kezdete valószínűleg nem fog egybeesni a látható képernyő kezdőcímével a képernyő-memóriában.

A 16226. sorban a *wrap* változót ellenőrizzük, egy összetett vizsgálat első részeként. A *wrap* igaz értéket ad az olyan régebbi megjelenítők esetén, amelyek támogatták a hardveres görgetést, és ha az ellenőrzés sikertelen, egy egyszerű hardveres görgetés történik a 16230. sorban, ahol a videovezérlő lapka által használt *cons->c_org* mutatót, amely a képernyő kezdetét mutatja a videovezérlő mikroáramkörben, úgy állítják be, hogy a képernyő bal felső sarkában megjelenítendő legelső karakterre mutasson. Ha a *wrap* értéke hamis *(FALSE)*, akkor az összetett vizsgálat azzal folytatódik, hogy a görgetés során felfelé mozgatandó blokk túlcsordul-e az adott konzol számára megadott memóriablokk határán. Ha igen, akkor ismét meghívjuk a *vid_vid_copy*-t, hogy a blokkot mozgassa a konzol memóriaterületének elejére, és a kezdetének a mutatóját aktualizálják. Ha nincs átlapolás, akkor a vezérlés egy egyszerű hardveres görgetési módszerre kerül, amelyet a régebbi videovezérlők mindig használtak. Ez a *cons->c_org* igazításából és az új kezdőpozíció értékének a videovezérlő lapka megfelelő regiszterébe történő beírásából áll. Ennek meghívását később hajtják végre, miután a képernyőn levő legalsó sort törölték, hogy a „görgetés" hatását keltsék.

A lefelé görgetés kódja nagyon hasonlít a felfelé görgetés kódjához. Végül a *mem_vid_copy*-t hívják meg, hogy töröljék a *new_line* által megcímzett legalsó (legfelső) sort. Ezután a *set_6845* hívásával beírjuk az új kezdeti pozíciót, a *cons->c_org* értékét a megfelelő regiszterekbe, a *flush* pedig biztosítja, hogy az összes változás láthatóvá váljék a képernyőn.

A *flush*-t (16259. sor) már többször említettük. A *flush* átviszi a kimeneti sorban lévő karaktereket a videomemóriába a *mem_vid_copy* segítségével, karbantart néhány változót, biztosítja, hogy a sor- és oszlopszámok értelmesek legyenek, kiigazítja őket, ha például egy vezérlőszekvencia megpróbálja a kurzort negatív oszlopértékre állítani. Végül annak kiszámítása következik, hogy hol kell lennie a kurzornak, és ezt összehasonlítják a *cons->c_cur*-ral. Ha nem egyeznek, és a most kezelt videomemória megegyezik az aktuális virtuális konzoléval, akkor *set_6845* hívásával beállítja a megfelelő értéket a videovezérlő kurzor regiszterébe.

A 3.44. ábra bemutatja, hogy a vezérlőszekvenciák kezelését hogyan lehetséges egy véges automatával reprezentálni. Megvalósítása a *parse_escape* (16293. sor) függvénnyel történik, amelyet az *out_char* elején hívunk meg, ha a *cons->c_esc_state* nem nulla. Az ESC-et magát az *out_char* ismeri fel, és beállítja a *cons->c_esc_state*-et 1-re. Amikor a következő karakter megérkezik, a *parse_escape* felkészül a további feldolgozásra úgy, hogy egy „\0" karaktert tesz a *cons_c_esc_intro*-ba, a



3.44. ábra. *A vezérlőszekvenciákat feldolgozó véges automata*

paraméterek tömbjének kezdetét jelző mutatót, a *cons->c_esc_paramv*[0] címét beteszi a *cons->c_esc_parmp*-be, és kinullázza magát a paramétertömböt. Ezután az ESC-et követő első karaktert vizsgálják – csupán a „[" és az „M" a megengedett értékek. Az első esetben a „["-t bemásolják a *cons->c_esc_intro*-ba, és az automata a 2-es állapotba kerül. A második esetben a *do_escape* függvényt hívják meg, hogy végrehajtsák a tevékenységet, és az escape állapotjelzőt nullára állítják. Ha az ESC-et követő karakter nem a megengedettek közül való, akkor figyelmen kívül hagyják, és a rá következő karakterek ismét normálisan jelennek meg.

Ha egy ESC [szekvencia érkezik, a következő karaktert a 2-es escape állapotban vizsgálják meg. Ennél a pontnál három lehetőség adódik. Ha a karakter egy numerikus karakter, a számértékét hozzáadjuk a *cons->c_esc_parmp* által mutatott érték 10-szereséhez, amely mutató kezdetben a *cons->c_esc_parmv*[0]-ra mutat (és nulla a kezdőértéke). Az escape állapot nem változik. Ez lehetővé teszi számjegyek sorozatának bevitelét és egy nagy numerikus paraméter előállítását, bár a MINIX 3 által jelenleg értelmezett legnagyobb érték 80, amelyet a kurzor tetszőleges pozícióra való mozgatásához használt vezérlőszekvenciában használnak (16335–16337. sor). Ha a következő karakter egy pontosvessző, akkor van egy másik numerikus paraméter, a paraméterre mutató változót továbbállítják, így lehetővé teszik, hogy a következő numerikus értéket összegyűjtsék a második paraméterben (16339–16341. sor). Ha a *MAX_ESC_PARMS* értékét megnövelnénk, hogy több paraméter számára foglaljon helyet, akkor a fenti kódot nem kellene megváltoztatni ahhoz, hogy további numerikus értékeket gyűjtsön össze, újabb paraméterek begépelése után. Végül ha a karakter nem számjegy és nem pontosvessző, akkor meghívják a *do_escape*-et.

A *do_escape* (16352. sor) egyike a MINIX 3-rendszer forráskódjában található hosszabb függvényeknek, még akkor is, ha a MINIX 3 vezérlőszekvencia-készlete viszonylag szerény. Azonban teljes hosszában a benne levő kód könnyen érthető. Egy kezdeti *flush* hívás után, amellyel biztosítjuk azt, hogy a képernyő teljesen naprakész legyen, egy egyszerű if elágazás van, aszerint hogy az ESC-et közvetlenül követő karakter egy speciális vezérlőszekvencia kezdő karaktere-e, vagy sem. Ha nem, akkor csak egy érvényes tevékenység lehet, a kurzor mozgatása egy sorral feljebb, ha a szekvencia ESC M volt. Figyeljük meg, hogy az „M" ellenőrzése egy default ággal rendelkező switch utasításban történik, érvényességi vizsgálattal, és előre gondolva arra a lehetőségre, hogy olyan vezérlőszekvenciákat is hozzáadhassunk a rendszerhez, amelyek nem az ESC [formát használják. Sok vezérlőszekvenciánál a tipikus tevékenység a következő: megvizsgálják a *cons->c_row* változót, hogy szükség van-e görgetésre. Ha a kurzor már a 0. sorban van, akkor a *scroll_screen*-t hívják meg a *SCROLL_DOWN* paraméterrel; egyébként a kurzort egy sorral feljebb viszik. Az utóbbit egyszerűen a *cons->c_row* csökkentésével és a *flush* meghívásával érik el. Ha egy vezérlőszekvencia-bevezető karaktert találtak, akkor az 16377. sorban levő else utáni kód fog végrehajtódni. Megvizsgálják, hogy egy „[" érkezett-e, az egyetlen, amit a MINIX 3 vezérlőszekvencia-bevezető karakterként felismer. Ha a szekvencia érvényes, akkor a vezérlőszekvencia első paramétere (vagy 0, ha nem volt numerikus paraméter) bekerül a *value* (16380. sor) változóba. Ha a szekvencia nem érvényes, akkor semmi sem történik, kivéve azt, hogy a következő nagy switch (16381–16586. sor) utasítás kimarad, és az escape állapotot nullára állítják a *do_escape*-ből való visszatérés előtt. Az érdekesebb esetben, amikor a szekvencia érvényes, végrehajtják a switch utasítást. Nem fogjuk az összes esetet megtárgyalni, csak megemlítjük azt a néhányat, amely jellemző a vezérlőszekvenciák egyes tevékenységtípusaira.

Az első öt, numerikus argumentumok nélküli, vezérlőszekvenciát az IMB PC-billentyűzetén levő négy nyíl és a HOME billentyű generálják. Az első kettő, az ESC [A és ESC [B hasonlók az ESC M-hez, azzal a különbséggel, hogy fogadhatnak numerikus paramétert, és így több mint egy sorral is képesek fel és le mozogni, de nem görgetik a képernyőt, ha a megadott paraméter olyan mozgást határozna meg, amely a képernyő határain kívül esik. Ezekben az esetekben a *flush* lekezeli a határokon túli mozgást, és korlátozza a mozgást a legalsó vagy legfelső sorig. A következő két szekvencia, az ESC [C és az ESC [D, amelyek jobbra és balra mozgatják a kurzort, hatását hasonlóan korlátozza a *flush*. Amikor a nyílbillentyűk generálják a szekvenciákat, nincs numerikus argumentumuk, így az alapértelmezés szerinti egy sor vagy oszlop mértékű mozgás történik.

Az ESC [H két paraméterrel rendelkezhet, például ESC [20;60H. A paraméterek abszolút és nem az aktuális kurzorpozícióhoz relatív pozíciókat határoznak meg, és az 1-gyel induló számokat 0-val kezdőkké alakítják a megfelelő értelmezéshez. A HOME billentyű az alapértelmezés szerinti (nincs paraméter) szekvenciát generálja, amely az (1, 1) pozícióba viszi a kurzort.

Az ESC [sJ és az ESC [sK törli a teljes képernyő vagy az aktuális sor egy részét, a megadott paraméter függvényében. Mindkét esetben kiszámolják a karakterek számát. Például az ESC [1J esetén a *count* értéke a képernyő kezdetétől a kurzor pozíciójáig terjedő karakterek száma lesz. A karakterek száma, valamint egy pozíció-paraméter a *dst,* ami lehet akár a képernyő kezdete, *cons->c_org,* akár az éppen aktuális kurzorpozíció, *cons->c_cur;* ezek szoktak lenni azok a paraméterek, amelyekkel a *mem_vid_copy*-t meghívják. Ezt az eljárást egy olyan paraméterrel hívják meg, amelynek hatására a megadott területet az aktuális háttérszínnel tölti fel.

Regiszterek	Funkció
10–11	A kurzor mérete
12–13	A megjelenítendő képernyő kezdőcíme
14–15	A kurzor pozíciója

3.45. ábra. *A 6845-ös néhány regisztere*

A következő négy szekvencia beszúr és töröl sorokat és szóköz karaktereket a kurzor pozícióban. Működésük nem igényel részletes magyarázatot. Az utolsó eset, az ESC [*n*m (figyeljünk arra, hogy a „n" egy numerikus paramétert jelöl, míg az „m" egy karakterliterál) a *cons–>c_attr*-ra van hatással, arra az attribútumbájtra, amely a karakterkódok közé ékelődik, amikor azok kiíródnak a videomemóriába.

A következő függvényt, a *set_6845*-öt (16594. sor) akkor használják, amikor a videovezérlő lapkát kell programozni. A 6845-ös lapkának 16 bites belső regiszterei vannak, amelyek 8 bites egységekben programozhatók, minden egyes regiszter írása négy I/O-kapu írásával jár. Úgy hajtják őket végre, hogy egy tömböt állítanak össze (vektort) portcím és értékpárokból, majd elvégzik a sys_voutb kernelhívást, amely elvégezteti a rendszertaszkkal a kért I/O-feladatot. A 6845 típusú videovezérlő mikroáramkör néhány regisztere a 3.45. ábrán látható.

A következő függvény a *get_6845* (16613. sor), amely a videovezérlő áramkör regisztereiben található értékeket adja vissza. Ez ugyancsak a rendszertaszkot használja a feladatának elvégeztetésére. A jelenlegi MINIX 3-kódban sehol sem hívják meg, azonban hasznos lehet a további bővítések során, például grafikus támogatás hozzáadásánál.

A *beep* (csengetés) függvényt (16629. sor) akkor hívjuk meg, amikor egy CTRL-G karaktert kell kiírni. Ez kihasználja a PC-nek azt a képességét, hogy hangot lehet előállítani úgy, hogy a hangszórónak négyszöghullámjelet küldünk. A hang megszólaltatása az I/O-kapuk egyfajta varázslatos manipulációjával kezdődik, amelyet csak az assembly programozók szerethetnek igazán. A kód egy érdekesebb része az időzítőt használja a hang kikapcsolására. Mint rendszerprivilégiumokkal rendelkező processzus (azaz egy bejegyzés a *priv* táblában) a terminálmeghajtó üzembe állíthat egy időzítőt a *tmrs_settimers* könyvtári függvény használatával. A 16655. sorban ezt a következő, *stop_beep* függvénnyel végzik el, egy olyan függvénnyel, amelyet akkor kell meghívni, ha az időzítő lejárt. Ezt az időzítőt a terminálmeghajtó saját időzítő sorában helyezik el. A sys_setalarm kernelhívás következik ezután, amely arra kéri a rendszertaszkot, hogy egy időzítőt indítson el a kernelben. Amikor az lejár, egy SYN_ALARM üzenetet kap a terminálmeghajtó főciklusa, a *tty_task*, amely meghívja az *expire_timers* függvényt, hogy lássa el a terminálmeghajtóhoz tartozó összes időzítőt, amelyek közül az egyik az, amelyiket a *beep* állított be.

Az *scr_init*-et (16679. sor) a *tty_init* hívja meg annyiszor, amennyi az *NR_CONS* értéke. Minden esetben egy *tty* struktúra címét tartalmazó mutató a paramétere, amely a *tty_table* egy elemére mutat. A 16693–16694. sorokban a *line* változót használják indexként a *cons_table* tömbhöz; kiszámítják az értékét, ellenőrzik, és

ha érvényes, akkor ezt használjuk a *cons* inicializálására, amely az aktuális konzolbejegyzésre mutat. Ezen a ponton lehet inicializálni a *cons–>c_tty* mezőt az eszköz fő *tty* struktúrájának címét tartalmazó mutatóval, és ezután már a *tp–>tty_priv* az eszköz *console_t* struktúrájára mutathat. Ezután a *kb_init*-et hívják meg, hogy inicializálja a billentyűzetet, és ezután az eszközfüggő rutinok címét tartalmazó mutatókat beállítják, hogy a *tp–>tty_devwrite* a *cons_write*-ra, míg a *tp–>tty_echo* a *cons_echo*-ra mutasson. A képernyővezérlő bázis regiszterének I/O-címét a BIOS-ból nyerik, míg a videomemória kezdőcímét és méretét a 16708–16731. sorokban határozzák meg, és a *wrap* jelzőt (amely azt határozza meg, hogyan kell görgetni) annak megfelelően állítják be, hogy milyen videovezérlőt használnak. A 16735. sorban a videomemória szegmensleíróját inicializálja a rendszertaszk a globális leírótáblában (GDT).

Ezután következik a virtuális konzolok inicializálása. Minden alkalommal, amikor az *scr_init*-et meghívják, más-más *tp* értéket adnak át paraméterként, és így más *line* és *cons* értékeket használunk a 16750–16753. sorokban, így biztosítanak minden egyes virtuális konzol számára egy saját részt a rendelkezésre álló videomemóriából. Ezután minden egyes képernyőt letörölnek, indulásra kész állapotba hoznak, és végül kiválasztják a 0-s című konzolt, hogy ez legyen az, amit először aktiválnak.

Néhány rutin a terminálmeghajtónak magának, a kernelnek és más rendszerkomponensnek a nevében jelenít meg kimenetet. Az első, a *kputc* (16775. sor) egyszerűen a *putk*-t hívja, amely egy szöveget ír ki bájtonként a következőkben leírt módon. Ez a rutin azért van itt, mert az a könyvtári rutin, amely a *printf* függvényt adja, olyan rendszerkomponensekben használt, amelyek úgy vannak megírva, hogy hivatkoznak egy ilyen nevű karakternyomtató rutinra, de más függvények a terminálmeghajtóban egy *putk* nevűt várnak.

A *do_new_kmess* (16784. sor) a kerneltől érkező üzenetek kinyomtatására szolgál. Az „üzenet" nem éppen a legjobb szó itt, mivel most nem a processzusok közötti kommunikációban használt üzenetekről van szó. Ez a függvény arra szolgál, hogy szöveget jelenítsen meg a konzolon, információkat, figyelmeztetéseket vagy hibajelzéseket adjon a felhasználónak.

A kernelnek speciális mechanizmusra van szüksége, hogy információt jelenítsen meg. Robusztusnak kell lennie, hogy használható legyen rendszerindításkor, mielőtt a MINIX 3 minden komponense elindulna, vagy pánik során, a másik olyan alkalommal, amikor a rendszer főbb részei lehet, hogy nem elérhetők. A kernel a szöveget egy körkörös pufferbe írja, ez része egy struktúrának, amely mutatókat is tartalmaz a következő kiírandó bájtra és a még fel nem dolgozott szöveg méretét. A kernel egy *SYS_SIG* üzenetet küld a terminálmeghajtónak, amikor van új szöveg, és az a *do_new_kmess* függvényt hívja, amikor a *tty_task* főciklusa fut. Ha a dolgok nem mennek simán (azaz a rendszer összeomlott), akkor a *SYS_SIG* üzenetet egy olyan ciklusban észlelik, amelyben szerepel egy blokkolással nem járó olvasási művelet a *do_panic_dumps*-ban, amit a *keyboard.c*-ben láttunk, és a *do_new_kmess*-t is innen hívják meg. Bármelyik esetben a sys_getkmessages rendszerhívás lekéri a kernelstruktúra másolatát, és a bájtokat egyesével megjelenítik, átadják őket a *putk*-nak, végül egy utolsó *putk* hívás egy nulla bájttal kikényszeríti

a kiírást a képernyőre. Egy lokális statikus változót használnak arra, hogy kövesse a pufferben a pozíciót az egyes üzenetek között.

A *do_diagnostics*-nak (16823. sor) hasonló a feladata, mint a *do_new_kmess*-nek, de a *do_diagnostics*-ot arra használják, hogy inkább rendszertaszkok üzeneteit jelenítse meg, ne pedig a kernelét. A *DIAGNOSTICS* üzenetet akár a *tty_task* főciklusa, akár a *do_panic_dumps* ciklusa fogadhatja, és bármelyik esetben a *do_diagnostics*-ot hívják meg. Az üzenet tartalmaz egy mutatót egy pufferre a hívó processzusban és egy számlálót az üzenet méretével. Nem használnak lokális puffert; ehelyett ismételt sys_vircopy kernelhívásokat hajtanak végre, hogy megszerezzék a szöveget bájtonként, egyesével. Ez védi a terminálmeghajtót: ha valami elromlik, és egy processzus túlságosan nagyméretű kimenetet kezd el generálni, nem fordulhat elő puffertúlcsordulás. A karaktereket egyesével írják ki a *putk* meghívásával, amit egy nulla bájt követ.

A *putk* (16850. sor) karaktereket írhat ki a képernyőre bármely kód részéről, amelyet a terminálmeghajtóhoz hozzászerkesztettek, és azok a függvények használják, amelyeket éppen most tárgyaltunk, hogy szöveget írjanak ki a kernel vagy más rendszerkomponens nevében. Csak meghívja az *out_char* függvényt minden egyes nem nulla megkapott karakterre, és meghívja a *flush*-t a nulla bájtra a szöveg végén.

A *console.c*-ben levő többi megmaradt rutin rövid és egyszerű, így gyorsan átnézzük őket. A *toggle_scroll* (16869. sor) azt teszi, amit a neve is mutat, váltogatja azt a jelzőt, amely meghatározza, hogy hardveres vagy szoftveres görgetést használnak. Egy üzenetet is kiír az aktuális kurzor pozícióra, és megadja a kiválasztott módot. A *cons_stop* (16881. sor) újrainicializálja a konzolt, olyan állapotba hozza, amelyet a betöltési felügyelőprogram vár a rendszer leállítása vagy újraindítása előtt. A *cons_org0*-t (16893. sor) csak akkor használják, amikor az F3 billentyűvel megváltoztatták a görgetési módot, vagy előkészülnek a rendszer leállítására. A *select_console* (16917. sor) kiválaszt egy virtuális konzolt. Az új sorszámmal hívják meg, és kétszer meghívja a *set_6845*-öt, hogy a videovezérlő a videomemória megfelelő részét jelenítse meg a képernyőn.

Az utolsó két rutin erősen hardverfüggő. A *con_loadfont* (16931. sor) egy betűkészletet (fontot) tölt le a grafikus illesztőbe, az ioctl *TIOCSFON* műveletének támogatására. Meghívja a *ga_program*-ot (16971. sor), hogy egy I/O-kapura elküldött mágikus írások sorozatának eredményeként láthatóvá váljon a videovezérlő betűkészlet memóriája, amelyet normál körülmények között nem címezhet meg a CPU. Ezután a *phys_copy*-t hívják, hogy a betűkészlet adatait átmásolják erre a memóriaterületre, és egy másik mágikus írássorozattal visszaállítjuk a grafikus vezérlő normális működési módját.

Az utolsó függvény a *cons_ioctl* (16987. sor). Egyetlen funkciója van, beállítja a képernyő méretét, csak az *scr_init* hívja a BIOS-ból nyert értékekkel. Ha szükség lenne egy valódi ioctl hívásra, hogy megváltoztassuk a sizeMINIX 3screen kódját, hogy új méreteket adhassunk meg, akkor azt meg kellene írni.

3.9. Összefoglalás

A bevitel/kivitel (I/O) fontos kérdés, amelyet gyakran elhanyagolnak. Bármelyik operációs rendszernek egy tekintélyes része az I/O-val foglalkozik. Ennek ellenére az I/O-eszközmeghajtók gyakran felelősek az operációs rendszer problémáiért. Az eszközmeghajtókat gyakran olyan programozók írják, akik az eszközöket készítő gyáraknak dolgoznak. A hagyományos operációs rendszerek felépítése általában megkívánja, hogy az eszközmeghajtók számára megengedjék, hogy hozzáférhessenek az olyan kritikus erőforrásokhoz, mint a megszakítások, az I/O-kapuk és a más processzusokhoz tartozó memória. A MINIX 3 kialakítása olyan, hogy a meghajtókat független, korlátozott privilégiumokkal rendelkező processzusokként elkülöníti egymástól, így a meghajtóban fellépő hibától nem omlik össze az egész rendszer.

Vizsgálódásunkat azzal kezdtük, hogy megnéztük az I/O-hardvert, az I/O-eszközök és az I/O-vezérlők kapcsolatát, amelyekkel a szoftvernek foglalkoznia kell. Ezután áttekintettük az I/O-szoftver négy szintjét: a megszakításkezelő rutinokat, az eszközmeghajtókat, az eszközfüggetlen I/O-szoftvert és azokat az I/O-könyvtárakat és háttértárolókat (spooler), amelyek a felhasználói címterületen futnak.

A következőkben tárgyaltuk a holtpont problémáját és azt, hogy miként lehet vele megbirkózni. Holtpont akkor keletkezik, amikor processzusok egy csoportjában minden egyes processzus kizárólagos hozzáférést kapott bizonyos erőforrásokhoz, és mindegyik még egy másik erőforrást is meg akar kapni, amelyet a csoportban lévő valamelyik másik processzus már lefoglalt. Ilyenkor valamennyi blokkolt állapotba kerül, és egyik sem lesz képes futni a későbbiek során. A holtpontot úgy lehet elkerülni, ha a rendszert úgy alakítjuk át, hogy ez soha ne jöhessen létre, például egy processzusnak csak egy erőforrás birtoklását engedjük meg egy adott időpillanatban. Úgy is elkerülhető a holtpont, hogy minden egyes erőforráskéréskor megvizsgáljuk, hogy vezethet-e olyan helyzethez, amelyben holtpont (nem biztonságos állapot) lehetséges, és megtagadjuk vagy késleltetjük azokat a kéréseket, amelyek problémához vezethetnek.

A MINIX 3-ban az eszközmeghajtók független processzusokként vannak megvalósítva, amelyek a felhasználói címterületen futnak. Megnéztük a RAM-lemezmeghajtót, a merevlemez-meghajtót és a terminálmeghajtót. Mindegyik meghajtó rendelkezik egy főciklussal, amely kéréseket kap, feldolgozza ezeket, esetenként visszaküldve egy jelzést arról, ami történt. A főciklus, valamint a RAM-lemezmeghajtó, a merevlemezegység és a hajlékonylemez-meghajtók közös funkcióinak forráskódja egy közös könyvtárban található, azonban minden egyes meghajtó fordítása és szerkesztése a könyvtár egy saját lemásolt példányával történik. Mindegyik eszközmeghajtó saját címterében fut. A számos terminált, amelyek a rendszerkonzolt, a soros vonali kapcsolatokat és a hálózati kapcsolatokat használják, mind egyetlen terminálmeghajtó processzus szolgálja ki.

Az eszközmeghajtóknak változatos a kapcsolata a megszakításrendszerrel. Azok az eszközök, amelyek gyorsan képesek elvégezni munkájukat, mint a RAM-lemezek és a tárcímleképezéses megjelenítők, egyáltalán nem használnak megszakításokat. A lemezmeghajtó munkájának nagy részét a saját kódjával végzi, és a megszakításkezelők csak állapotinformációkat adnak számára. A megszakítások

minden esetben szükségesek, és egy receive művelettel lehet megvárni az egyik bekövetkezését. Billentyűzetmegszakítás bármikor bekövetkezhet. Minden megszakítás által generált és a terminálmeghajtónak küldött üzenetet a meghajtó főciklusában fogadnak és dolgoznak fel. Amikor egy billentyűzetmegszakítás bekövetkezik, a feldolgozás első fázisa, hogy a beolvasást a leggyorsabban végezzük el, hogy a rendszer készen állhasson a következő megszakítás fogadására.

A MINIX 3-meghajtóknak korlátozott privilégiumaik vannak, és saját maguk nem tudnak megszakításokat kezelni vagy I/O-kapukhoz hozzáférni. A megszakításokat a rendszertaszk kezeli, amely egy üzenet küldésével értesíti a meghajtót, amikor egy megszakítás bekövetkezik. Az I/O-kapukhoz történő hozzáférés szintén a rendszertaszk közvetítésével történik. A meghajtók nem tudják közvetlenül írni vagy olvasni az I/O-kapukat.

Feladatok

1. Egy 1x sebességű DVD-olvasó 1,32 MB/s sebességgel tud adatot küldeni. Hányszoros sebességű az a DVD-meghajtó egység, amely egy USB 2.0 kapcsolaton keresztül még meghajtható adatvesztés nélkül?
2. Sok lemez tartalmaz hibajavító kódot (Error Correction Code, ECC) minden egyes szektor végén. Ha a hibajavító kód rossz, milyen tevékenységeket tud elképzelni, és ezeket a szoftver vagy a hardver melyik része hajtja végre?
3. Mi a memórialeképezésű I/O? Miért használják időnként?
4. Magyarázza el, hogy mi a közvetlen memóriaelérés (Direct Memory Access, DMA), és miért használják.
5. Bár a DMA nem használja a CPU-t, a maximális átviteli sebesség mégis korlátozott. Tekintsük egy blokk beolvasását a lemezről. Nevezzen meg három olyan tényezőt, amelyek alapvetően meghatározhatják az átviteli sebességet.
6. A CD-minőségű zene megköveteli, hogy másodpercenként 44 100-szor vegyünk mintát a hangjelből. Tegyük fel, hogy egy időzítő megszakításokat generál ezzel a sebességgel, és minden egyes megszakítás kezelése 1 μs ideig tart egy 1 GHz-es CPU-nak. Mi az a legalacsonyabb CPU-sebesség, amelyet még használhatunk anélkül, hogy adatot vesztenénk? Tegyük fel, hogy a megszakításkor végrehajtandó műveletek száma azonos, tehát ha megfelezzük az órajel-frekvenciát, akkor a megszakítások kezelési ideje megduplázódik.
7. A megszakítások egyik alternatívája a folyamatos lekérdezés (polling). Vannak-e olyan esetek, amikor úgy gondolja, hogy a lekérdezés a jobb megoldás?
8. A lemezvezérlőknek belső puffereik vannak, és ezek egyre nagyobbak minden egyes új modellnél. Miért?
9. Minden eszközmeghajtónak két interfésze van az operációs rendszerrel. Az egyik interfész olyan függvényhívások halmaza, amelyeket az operációs rendszer hajt végre a meghajtón. A másik rendszerhívások egy olyan halmaza, amelyeket a meghajtó kezdeményez az operációs rendszer felé. Nevezzen meg egy-egy valószínű hívást mindegyik interfészből.

10. Miért törekednek arra az operációsrendszer-tervezők, hogy eszközfüggetlen I/O-szolgáltatásokat biztosítsanak, ahol csak lehetséges?
11. Az I/O-szoftverének négy rétege közül melyikben történik a következő?
    (a) A sáv-, szektor- és fejcím kiszámítása egy lemezolvasáshoz.
    (b) Az utóbbi időben használt blokkok gyorsítótárának kezelése.
    (c) Parancsok írása az eszközregiszterekbe.
    (d) Annak ellenőrzése, hogy a felhasználó jogosult-e az eszköz használatára.
    (e) Bináris egész számok ASCII karakterekké konvertálása a nyomtatáshoz.
12. A nyomtatásra várakozó kimeneti fájlokat miért gyűjtik össze általában egy lemezen, mielőtt kinyomtatásra kerülnek (spooling)?
13. Adjon példát olyan holtpontra, amely a fizikai világban fordulhat elő.
14. Tekintsük a 3.10. ábrát. Tegyük fel, hogy az (o) lépésben a *C* nem az *R*-et, hanem az *S*-et igényli. Vajon ez holtponthoz vezet? Tegyük fel, hogy *S*-et és *R*-et is igényli. Ilyenkor mi a válasz?
15. Nézzük meg figyelmesen a 3.13.(b). ábrát. Ha *D* még egy egységet kér, akkor ez biztonságos vagy nem biztonságos állapothoz vezet? Mi a helyzet akkor, ha az igény nem *D*-től, hanem *C*-től érkezik?
16. A 3.14. ábrán levő pályák vagy horizontálisak, vagy vertikálisak. El tud képzelni olyan helyzetet, amelyben diagonális pályák is létezhetnek?
17. Tegyük fel, hogy a 3.15. ábra szerinti *A* processzus a legutolsó szalagegységet igényli. Holtponthoz vezet-e ez az esemény?
18. Egy számítógépnek hat szalagos egysége van, amelyért *n* processzus versenyez. Minden egyes processzusnak két meghajtóegységre van szüksége. Milyen *n* esetén lesz a rendszer holtpontmentes?
19. Lehetséges-e, hogy egy rendszer olyan állapotba kerül, amely nem holtpont, és nem biztonságos? Ha igen, akkor adjunk rá példát. Ha nem, bizonyítsuk, hogy minden állapot vagy holtpont, vagy biztonságos.
20. Egy elosztott rendszernek, amely levelesládákat használ, két IPC primitívje van, a send és a receive. Az utóbbi primitív megad egy processzust, ahonnan fogadni akarunk, és blokkolt állapotba kerül abban az esetben, ha ettől a megadott processzustól nem kap üzenetet, még akkor is, ha van várakozó üzenet más processzusoktól. Nincsenek megosztott erőforrások, de a proccsszusoknak más okokból kifolyóan gyakran kell kommunikálniuk. Lehetséges a holtpont?
21. Egy elektronikus pénzátutalási rendszerben azonos processzusok százai működnek a következőképpen. Minden egyes processzus elolvas egy bemenő sort, amely megmondja a pénzmennyiséget, a terhelendő és a jóváírandó számlaszámot. Ezután zárolja mindkét számlát, átviszi a pénzt, majd feloldja a zárolást, amikor kész. Amikor sok processzus fut párhuzamosan, akkor valós veszély van a holtpont kialakulására: az *x* számla zárolása után könnyen előfordulhat, hogy nem sikerül az *y* számlát zárolni, mert az *y* számlát már zárolta egy olyan processzus, amely most éppen az *x* számlára vár. Tervezzen egy olyan sémát, amely megelőzi a holtpontokat. Csak akkor szabad elengedni a számlákat, ha már befejeztük a tranzakciót. (Más szóval olyan megoldást nem engedünk meg, amely zárol egy számlát, de rögtön elengedi, ha a másik számla zárolva van.)

22. A bankár algoritmust futtatjuk egy olyan rendszerben, ahol m erőforrás osztály és n processzus van. Nagy m és n esetén egy állapot biztonságossága eldöntésének műveletigényére a korlát $m^a n^b$-vel arányos. Mik az a és b értékei?
23. Tekintsük a 3.15. ábrán látható bankár algoritmust. Tegyük fel, hogy az A és a D processzusok megváltoztatják igényüket egy újabb (1, 2, 1, 0) és (1, 2, 1, 0) elemre. Kielégíthetők-e ezek az igények, és biztonságos állapotban marad-e a rendszer?
24. Hamupipőke és a herceg válófélben vannak. A közös tulajdonuk megosztása érdekében a következő algoritmusban állapodtak meg. Minden reggel mindegyikük egy levelet küld a másik ügyvédjének, amelyben egy dolgot kér a közös tulajdonból. Miután a levelek kézbesítése egy napot vesz igénybe, megállapodnak abban, hogy ha felfedezik, hogy mindketten ugyanazt a dolgot kérték, akkor a következő napon egy olyan levelet küldenek, amellyel visszamondják a kérést. A tulajdonuk között található a kutyájuk, Woofer, Woofer kutyaólja, a kanárijuk, Tweeter és Tweeter kalitkája. Az állatok ragaszkodnak a házaikhoz, így aztán megállapodtak abban, hogy az olyan megosztás, amely valamely állatot elszakítaná házától, érvénytelen, és az egész megosztást teljesen elölről kell kezdeni. Mind Hamupipőke, mind a herceg kétségbeesetten akarja Woofert. Így aztán nyaralni mentek (persze külön), és mindkét házastárs beprogramozott egy személyi számítógépet a tárgyalás lefolytatására. Amikor visszaérkeztek a vakációról, a számítógépek még mindig tárgyaltak. Miért? Lehetséges, hogy holtpont alakult ki? Lehet, hogy örökké fognak várni? Indokolja a válaszát.
25. Tekintsünk egy lemezegységet, amelyen 1000 db 512 bájtos szektor található sávonként, 8 sáv cilinderenként, és 10 000 cilinder 10 ms körülfordulási idővel. A sávról sávra történő keresési idő 1 ms. Mi a maximálisan elérhető gyors írási sebesség? Mennyi ideig tarthat egy ilyen írás?
26. Egy lokális hálózatot a következőképpen használnak. A felhasználó rendszerhívásokat ad ki ahhoz, hogy csomagokat írjon a hálózatra. Az operációs rendszer átmásolja az adatokat a saját pufferébe. Ezután átmásolja az adatokat a hálózati vezérlő kártyára. Amikor minden bájt biztonságosan megérkezett a vezérlő belsejébe, kiküldik őket a hálózatra 10 Mbit/s sebességgel. A fogadó hálózati vezérlő minden egyes bitet 1 μs-mal később tárol, mint azt elküldték. Amikor az utolsó bit is megérkezett, a fogadó CPU egy megszakításkérést ad ki, majd az operációs rendszere átmásolja az újonnan érkezett csomagot a pufferébe, hogy megvizsgálja. Amikor kiderítette, hogy a csomag melyik felhasználó számára érkezett, átmásolja az adott felhasználó területére. Ha feltételezzük, hogy minden egyes megszakítás és a hozzá tartozó feldolgozás 1 ms, és hogy a csomagok 1024 bájtosak (nem számítva a fejlécet), valamint hogy egy bájt másolása 1 μs, mi a maximális sebesség, amellyel egyik processzus adatokat tud továbbítani a másiknak? Tegyük fel, hogy a küldő blokkolt állapotba kerül, amíg a fogadó oldalon a munka nem ér véget, és egy nyugtázás nem érkezik vissza. Az egyszerűség kedvéért tegyük fel, hogy a nyugtázás visszaérkezésének ideje elhanyagolhatóan kicsi.
27. A 3.17. ábrán bemutatott üzenetformátumot a blokkeszközök meghajtóinál használjuk a kérés elküldésére. Lehet-e mezőket elhagyni karakteres eszközök esetében? Melyeket?

28. A lemezmeghajtóhoz a következő cilinderértékekkel érkeznek kérések: 10, 22, 20, 2, 40, 6 és 38, ebben a sorrendben. Egy keresés cilinderenként 6 ms. Mekkora keresési (seek) idő szükséges, ha
    (a) az elsőként érkezettet szolgáljuk ki elsőként;
    (b) a legközelebbi cilindert szolgáljuk ki következőnek;
    (c) a liftes algoritmust használjuk (először felfelé mozgunk)?
    Minden esetben feltehetjük, hogy induláskor a fej a 20. cilinderen áll.
29. Egy személyi számítógépeket árusító ügynök egy délnyugat-amszterdami egyetemen azt állította a termékbemutatón, hogy cége jelentős erőfeszítéseket tett annak érdekében, hogy Unix-verziójuk nagyon gyors legyen. Példaképp elmondta, hogy a lemezmeghajtó a liftes algoritmust használja, és szektorsorrendben sorba gyűjti az egy cilinderre érkező kéréseket. Egy hallgató, Harry Hacker, akire az előadás nagy hatást gyakorolt, vett is egy ilyen gépet. Hazavitte, és írt egy programot, amely 10 000 véletlenszerűen kiválasztott blokkot olvasott a lemezről. A legnagyobb meglepetésére a mért teljesítmény azonos volt azzal, mint amit az elsőként érkezett, elsőként lesz kiszolgálva (FIFO) algoritmustól vártunk volna. Hazudott-e az ügynök?
30. Egy Unix-processzus két részből áll: egy felhasználói és egy kernelrészből. A kernelrész mire hasonlít inkább: egy szubrutinra vagy egy korutinra?
31. Egy bizonyos számítógépóra-megszakítás kezelője 2 ms időt igényel (beleértve a taszkváltás idejét is) órajelenként. Az óra 60 Hz-es sebességgel működik. A CPU-idő hány százalékát fordítják az óra kiszolgálására?
32. A könyvben két példát is adtunk a felügyeleti időzítőkre: a hajlékonylemez motorjának felpörgetésére és a nyomtatóterminálok kocsi vissza kezelésére használtuk ezeket. Adjon meg egy harmadik példát is.
33. Miért kezelik az RS–232-es terminálokat megszakításokkal, míg a tárcímleképezéses terminálok nem megszakításvezéreltek?
34. Gondoljuk végig, hogyan működik egy terminál. A meghajtó kiír egy karaktert, majd blokkolt állapotba kerül. Amikor a karakter már kiíródott, egy megszakítás történik, és üzenetet kap a blokkolt meghajtó, amelynek hatására a következő karakter kerül kiírásra, majd ismét blokkolódik. Vajon jól működik-e ez a módszer 110 baudos vonalak esetén, ha az üzenetküldés, a karakterkiírás és a blokkolás ideje 4 ms? Mi a helyzet 4800 baudos vonalak esetén?
35. Egy bittérképes terminál 1200 × 800 képpontot tartalmaz. Egy ablak görgetéséhez a CPU-nak (vagy a vezérlőnek) a sor feljebb viteléhez az összes bitet (amely az ablak része) át kell másolnia a videomemória egyik részéből a másikba. Mennyi ideig tart annak az ablaknak a görgetése, amely 66 sor magas és 80 karakter széles (5280 karakter összesen), ha minden karakter 8 pixel széles és 12 pixel magas, és egy bájt átmásolása 500 ns ideig tart? Mennyi a terminálhoz tartozó baudráta, ha minden sor 80 karakteres? Egy karakter képernyőn történő megjelenítésének ideje 50 μs. Számoljuk ki a baudrátát, ha ugyanez a terminál színes, 4 bit/pixeles felbontással. (Ekkor egy karakter képernyőn történő megjelenítésének ideje 200 μs.)
36. Miért gondoskodnak az operációs rendszerek olyan escape karakterekről, mint a CTRL-V a MINIX-ben?

37. A CTRL-C (SIGINT) karakter érkezése után a MINIX-meghajtó törli az összes terminálhoz tartozó kimenő sorban levő karaktert. Miért?
38. Sok RS–232-es terminálnak van olyan vezérlőszekvenciája, amely kitörli az aktuális sort, és az alatta levő sorokat egy sorral feljebb mozgatja. Mit gondol, hogyan van ez megvalósítva a terminálban?
39. Az eredeti IBM PC színes képernyőjénél a videó RAM-ba történő írás „havazást" váltott ki (csúnya pöttyök jelentek meg az egész képernyőn), ha az írás nem éppen a CRT sugárnyaláb vertikális visszatérésénél történt. A képernyő képe 80 × 25 karakter, minden egyes karakter 8 × 8 pixel méretű. Minden egyes sor 640 pixele a sugárnyaláb egy horizontális letapogatásával rajzolódik ki, amely 63,6 μs ideig tart, beleértve a horizontális visszafutási időt is. A képernyőt másodpercenként 60-szor rajzoljuk ki, amely megköveteli a sugárnyaláb vertikális visszatérítését a képernyő tetejére. Az idő hányad részében áll rendelkezésre a videó RAM, hogy beleírjanak?
40. Írjon egy grafikus meghajtót az IBM színes displayhez vagy más hasonló bittérképes megjelenítőhöz. A meghajtónak tartalmaznia kell parancsokat egyedi képpontok beállítására és törlésére, téglalapok képernyőn való mozgatására és más olyan lehetőségekre, amelyeket érdekesnek tart. A felhasználói programok a meghajtóval úgy érintkezzenek, hogy megnyitják a */dev/graphics* fájlt és erre írják a parancsaikat.
41. Módosítsa a MINIX hajlékonylemez-meghajtót úgy, hogy egysávos gyorsítótárral rendelkezzen.
42. Valósítson meg egy olyan hajlékonylemez-meghajtót, amely inkább karakteres eszközként, mint blokkeszközként működik, kikerülve a fájlrendszer gyorsítótárát. Egy ilyen megoldással a felhasználók nagy adatterületeket olvashatnak egyszerre a lemezről, amelyek DMA-val közvetlenül a felhasználói címterületre vihetők, nagyban növelve így a hatékonyságot. Ezt a meghajtót főképp azok a programok használhatnák, amelyeknek a fájlrendszertől függetlenül kell a lemez bitjeit olvasni. A fájlrendszert ellenőrző programok például ebbe a kategóriába tartoznak.
43. Valósítsa meg a Unix PROFIL rendszerhívását, amely hiányzik a MINIX-ből.
44. Módosítsa a terminálmeghajtót úgy, hogy az előző karakter törlésére szolgáló speciális karakter mellett egy másik speciális karaktere is legyen az előző szó törlésére.
45. Egy új, cserélhető lemezes merevlemez típust adunk a MINIX 3-rendszerhez. Az eszközt fel kell pörgetni, valahányszor a lemezt kicserélik, és a felpörgetés sokáig tart. Arra számíthatunk, hogy a rendszer futása alatt gyakran fogják cserélgetni a lemezeket. Ezért az *at_wini.c*-ben levő *waitfor* rutin nem lesz megfelelő. Tervezzen egy új *waitfor* rutint, amelyben ha a várt bitminta nem érkezik meg egy másodperces tevékeny várakozás után, akkor a rendszer olyan állapotba kerül, amelyben a lemezmeghajtó egy másodpercig alszik, majd ismét teszteli a portot, majd ismét alszik egy másodpercig, egészen addig, amíg a keresett minta nem érkezik meg, vagy amíg egy előre beállított *TIMEOUT* idő le nem telik.

4. Memóriagazdálkodás

A memória fontos erőforrás, így gondosan kell vele bánni. Bár napjainkban az átlagos otthoni számítógépek kétezerszer akkora memóriát tartalmaznak, mint a hatvanas évek kezdetének legnagyobb számítógépe, az IBM 7094, a programok mérete éppen olyan gyorsan nőtt, mint a memória. Parkinson törvényének paragrafusa szerint: „a programok és adataik mindig kitöltik a rendelkezésre álló memóriát." Ebben a fejezetben bemutatjuk, hogyan kezelik az operációs rendszerek a memóriát.

Ideális esetben, amit minden programozó szeretne, a memória végtelen nagy és gyors, valamint nem felejtő, azaz áramszünet esetén is megőrzi tartalmát. Ezenkívül még azt is szeretnénk, ha olcsó lenne. Sajnos a technika nem képes ilyen álmokat megvalósítani. Így hát a legtöbb számítógép **memóriája hierarchikusan** szerveződik az alábbi rétegekből: kicsi, nagyon gyors, drága és felejtő gyorsítómemória; több száz megabájt, közepes sebességű és árú, felejtő központi memória (RAM); valamint több tíz vagy néhány száz gigabájt, lassú, olcsó és nem felejtő lemezes tároló. Az operációs rendszer feladata az, hogy megszervezze e memóriafajták használatát.

Az operációs rendszernek azt a részét, amely a hierarchikus memória kezelését végzi, **memóriakezelő**nek nevezzük. A feladata az, hogy nyilvántartsa mely memóriarészek szabadok vagy foglaltak, memóriát foglaljon a processzusoknak, amikor szükséges, illetve felszabadítsa, ha már nincs szükség rá, valamint vezérelje a cserét a központi memória és a lemez között, ha a központi tár túl kicsi a processzusok befogadásához. A legtöbb rendszerben (de a MINIX 3-ban nem) ezt a feladatot a kernel látja el.

Ebben a fejezetben számos egyszerű vagy bonyolult memóriakezelő algoritmust vizsgálunk. A kezdetektől indulva először a legegyszerűbb memóriakezelő rendszert tárgyaljuk, majd fokozatosan haladunk az egyre bonyolultabbak felé.

Amint már az első fejezetben is rámutattunk, a számítástechnika világában a történelem megismétli önmagát: a miniszámítógépek szoftvere olyan volt, mint a mainframe gépeké, a személyi számítógépek szoftvere pedig a miniszámítógép szoftverére hasonlított. A folyamat most megismétlődik a kézi számítógépekkel, a PDA-kal és a beágyazott rendszerekkel. Ezekben a rendszerekben még mindig egyszerű memóriakezelő megoldásokat használnak. Ez az oka annak, hogy ezeket a módszereket még érdemes tanulmányozni.

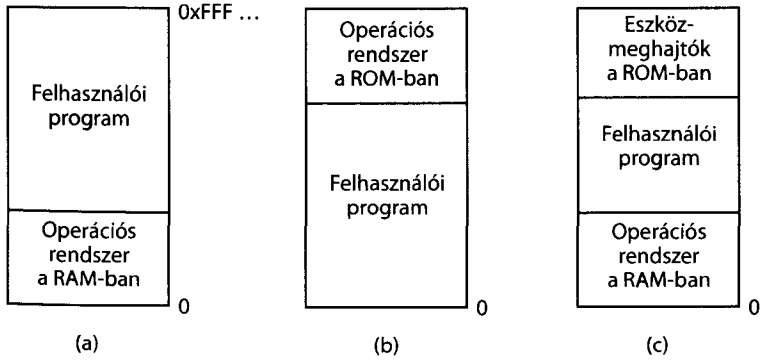
4.1. Alapvető memóriakezelés

A memóriakezelő algoritmusokat két csoportra oszthatjuk: azok, amelyek a végrehajtás közben mozgatják a processzusokat a központi tár és a lemez között (csere, lapozás), és azok, amelyek nem. Az utóbbiak az egyszerűbbek, így először ezeket tárgyaljuk. Ezután a cserét és a lapozást vizsgáljuk. A fejezet olvasása közben tartsuk észben, hogy a csere és a lapozás a memória pótlására szolgál azért, hogy a rendszer egy időben az összes programot futtatni tudja. Amennyiben valaha is elérjük azt az állapotot, amikor a főmemória mérete elegendő lesz, abban az esetben a különböző memóriakezelő megoldások közötti vita idejét múlttá válhat.

Más szemszögből vizsgálva, mint azt már említettük, úgy tűnik, hogy a szoftver mérete legalább olyan gyorsan növekszik, mint a memóriáé, így lehetséges, hogy mindig szükség lesz hatékony memóriakezelő megoldásokra. Az 1980-as években sok egyetemen működött időosztásos rendszer több tucat (többé-kevésbé elégedett) felhasználóval, mindössze 4 MB memóriával rendelkező VAX szerveren. Ma az egyfelhasználós Windows XP részére is legalább 128 MB memóriát ajánl a Microsoft. A multimédiás alkalmazások irányába mutató trend egyre több memóriaigényt vetít előre, így nagy valószínűséggel a következő évtizedekben is szükség lesz a hatékony memóriakezelő eljárásokra.

4.1.1. Monoprogramozás csere és lapozás nélkül

A legegyszerűbb memóriakezelési módszer az, hogy egy időben csak egy programot futtatunk, a memóriát megosztva az operációs rendszer és a program között. E módszer három változata látható a 4.1. ábrán. A 4.1.(a) eset szerint az operációs rendszer a memória (RAM) alacsony címein található, vagy a felső címeken a ROM-ban, amint a 4.1.(b) ábra mutatja. A 4.1.(c) esetben az eszközmeghajtók a memória magas címein a ROM-ban találhatók, az operációs rendszer maradék része az alacsony címeken a RAM-ban. Az első modellt régen a nagyszámítógépes



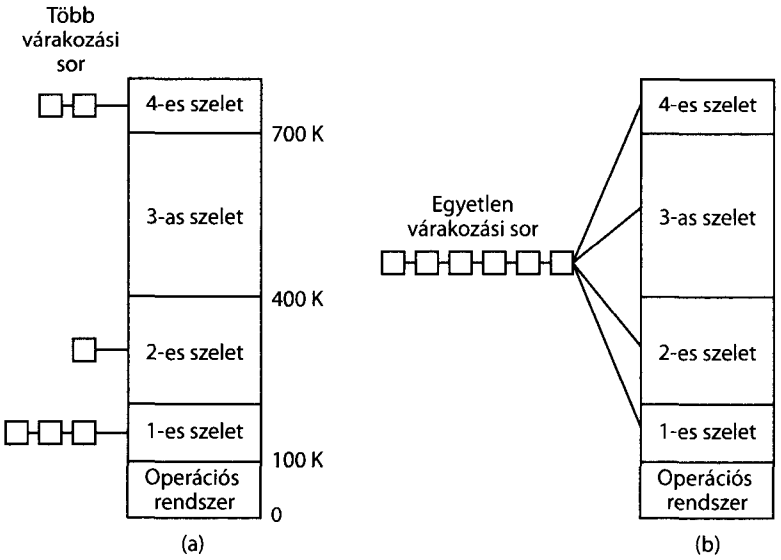
4.1. ábra. *A memória szervezésének három egyszerű módja operációs rendszer és egyetlen felhasználói processzus esetén. Más lehetőségek is léteznek*

rendszerekben és miniszámítógépeken használták, ma már ritkán találkozunk vele. A második modellt néhány kézi számítógép és beágyazott rendszer használja. A harmadik modellel a korai személyi számítógépeknél találkozhatunk (például MS-DOS-rendszerben); itt a rendszer ROM-ba égetett részét **BIOS**-nak **(Basic Input Output System)** nevezik.

Ha egy rendszer ilyen, akkor egy időben csak egy processzus futhat. Amint a felhasználó begépeli a parancsot, az operációs rendszer betölti a kért programot a lemezről, és végrehajtja. Miután a processzus befejeződik, az operációs rendszer újabb parancsra várakozik. Amikor megkapja a parancsot, betölti az új programot a memóriába, felülírva az előzőt.

4.1.2. Multiprogramozás rögzített méretű partíciókkal

A nagyon egyszerű beágyazott rendszereket kivéve, ma már ritkán találkozhatunk monoprogramozással. A legtöbb mai rendszer lehetővé teszi több processzus egy időben történő futtatását. Az időosztásos rendszerekben egyszerre több processzus tartózkodik a memóriában, amikor az egyik egy I/O-művelet befejezésére vár, addig egy másik processzus használhatja a központi egységet. Így a multiprogramozás növeli a CPU kihasználtságát. A hálózati szerverek mindig rendelkeznek több processzus (több kliens számára) egy időben történő futtatásához szükséges képességekkel, manapság már a legtöbb kliensgép (például asztali számítógép) is rendelkezik ezzel a képességgel.



4.2. ábra. *(a) Rögzített memóriaszeletek külön-külön várakozási sorral. (b) Rögzített memóriaszeletek egyetlen várakozási sorral*

A multiprogramozás megvalósításának legegyszerűbb módja az, hogy felosztjuk a memóriát n (lehetőleg nem egyenlő méretű) szeletre. Ezt a particionálást például manuálisan elvégezhetjük a rendszerindításnál.

Amikor egy munka beérkezik, a rendszer berakja annak a partíciónak a várakozási sorába, amely a legkisebb azok közül, amelyekbe belefér a program. Mivel a partíciók mérete rögzített, amíg a munka fut, elvész a partíciónak az a része, amit nem használ fel. A 4.2.(a) ábrán láthatjuk, hogy néz ki a fix partíciók rendszere partíciónként egy-egy külön várakozási sorral.

Ennek a módszernek az a hátránya, hogy előfordulhat olyan eset, hogy a nagy partíció várakozási sora üres, de a kisebb partíciókra sok munka várakozik, mint például a 4.2.(a) ábra 1. és 3. partíciója. Itt a kis munkának várnia kell a memóriába kerülésre, annak ellenére, hogy sok üres memória van. E módszer egyik lehetséges változata az, hogy csak egyetlen várakozási sor van, mint a 4.2.(b) ábra is mutatja. Amikor egy partíció kiürül, akkor az a munka töltődik be, amelyik belefér és a sorban a legelső. Mivel a nagy partíciókat nem érdemes kis munkákra pazarolni, egy másik stratégia az, hogy az egész várakozási sorból kiválasztjuk a legnagyobb munkát, amelyik belefér az üres partícióba. Megjegyzendő, hogy az utóbbi algoritmus hátrányosan különbözteti meg a kis munkákat, mint méltatlanokat arra, hogy egy egész partíciót birtokoljanak, holott ezek a kis munkák (gyakran interaktív munkák) a legjobb kiszolgálást érdemelnék meg, nem a legrosszabbat.

Az egyik megoldási mód, hogy legalább egy kis partíciónk legyen, ebben futhatnak a kis munkák, így nem kell egy nagy partíciót lefoglalni.

A másik megoldás az a szabály, hogy egyetlen munka sem mellőzhető k-nál többször a futásra kiválasztáskor. Minden olyan esetben, amikor nem a munkát választottuk, az kap egy pontot. Ha már k pontja van, akkor nem szabad elhanyagolni.

Ezt az operátor által reggel beállított, rögzített partíciókkal bíró rendszert sok évig használták az OS/360-ban az IBM-nagygépeken. A neve **MFT** volt (Multiprogramozás fix számú feladattal, OS/MFT). Egyszerűen megérthető és megvalósítható: a bejövő munkák egy sorban várakoznak, amíg a megfelelő partíció fel nem szabadul, ezután a munka betöltődik és lefut. Napjainkban azonban kevés operációs rendszer – ha egyáltalán van ilyen – támogatja ezt a modellt (még a kötegelt nagygépes rendszerekben sem).

4.1.3. Relokáció és védelem

A multiprogramozás két megoldandó problémát vet fel: a relokáció és a védelem kérdését. Mint a 4.2. ábrából is látszik, a különböző munkák különböző címeken futnak. Amikor a programot szerkesztik (azaz a főprogramot, az eljárásokat és a könyvtári függvényeket összerakják egyetlen címtérbe), a szerkesztőprogramnak tudnia kell, hogy a program a memória melyik címén fog kezdődni.

Példaként tegyük fel, hogy az első utasítás egy eljáráshívás a szerkesztő által készített bináris program 100-as címére. Ha ez a program az első partícióba töltődik be (a 100 K címre), akkor ez az utasítás a 100-as abszolút címre ugrik, amely az operációs rendszer területén belül van. Ami nekünk kell, az a 100 K + 100-as címre ugrás. Ha a program a második partícióba töltődik, akkor a 200 K + 100-as címen levő eljárást kell végrehajtani, és így tovább. Ezt a problémát **relokációs** problémának nevezzük.

Az egyik lehetséges megoldás az, hogy a program betöltésekor az operációs rendszer módosítja az utasításokat. Ha az első partícióba töltjük a programot, akkor 100 KB-ot, ha a másodikba, akkor 200 KB-ot kell minden címhez hozzáadni. Ahhoz, hogy a relokációt ilyen módon a betöltésnél hajtsuk végre, a szerkesztőnek bele kell raknia a programba egy térképet, hogy a programban mely szavak a relokálandó címek és melyek az utasításkódok, konstansok vagy egyéb nem relokálandó elemek. Az OS/MFT működött ilyen módon.

A betöltés alatti relokáció nem oldja meg a védelem kérdését. Egy rosszindulatú program mindig konstruálhat új utasításokat, és rájuk adhatja a vezérlést. Mivel ebben a rendszerben a programok abszolút memóriacímeket használnak relatív címek helyett, nincs mód arra, hogy megakadályozzuk a programokat olyan utasítások létrehozásában és végrehajtásában, amelyek tetszőleges memóriacímet olvasnak vagy írnak. A többfelhasználós rendszerekben nagyon nem kívánatos, hogy a programok más felhasználóhoz tartozó memóriát írjanak vagy olvassanak.

Az IBM által a 360-asban alkalmazott védelmi megoldás az, hogy 2 KB-os blokkokra osztják a memóriát, és minden blokkhoz egy négybites védelmi kódot rendelnek. A PSW (programállapotszó) tartalmaz egy négybites kulcsot. A 360-as hardverszinten elkap minden kísérletet, amikor egy processzus egy olyan memóriablokkot akar elérni, amelynek védelmi kódja különbözik a kulcstól. Mivel csak az operációs rendszer tudja megváltoztatni a védelmi kódokat és a kulcsot, a felhasználói processzusok védettek a többi processzustól és az operációs rendszertől.

A relokáció és a védelem együttes problémájára alternatív megoldás az, hogy a gépnek két speciális regisztere van, a **bázis-** és a **határ**regiszter. Amikor egy program végrehajtódik, a bázisregiszterbe betöltődik a partíció kezdőcíme, a határregiszterbe pedig a partíció hossza. Minden memóriacím automatikusan generálódik a bázisregiszter és a hivatkozott memóriacím összeadásával. Például ha a bázisregiszter értéke 100 K, egy CALL 100 utasítás valójában egy CALL 100 K + 100 utasítás végrehajtását jelenti a programkód módosítása nélkül. A címeket a határregiszterrel ellenőrzi a rendszer, hogy ne történhessen kísérlet a partíción kívüli memóriacímek elérésére. A felhasználói programok nem módosíthatják a bázis- és a határregiszter értékét.

A megoldás egy hátránya, hogy minden egyes memóriahivatkozásnál szükség van egy összehasonlításra és egy összeadásra. Az összehasonlítás ugyan gyors, de az összeadás speciális hardver nélkül lassú az átviendő terjedési ideje miatt.

Ezt a modellt alkalmazta a világ első szuperszámítógépe, a CDC 6600. Az eredeti IBM PC-ben használt Intel 8088-as processzor ennek a modellnek a határregiszter nélküli gyengébb változatával dolgozott. Ma már kevés számítógép használja ezt a modellt.

4.2. Csere

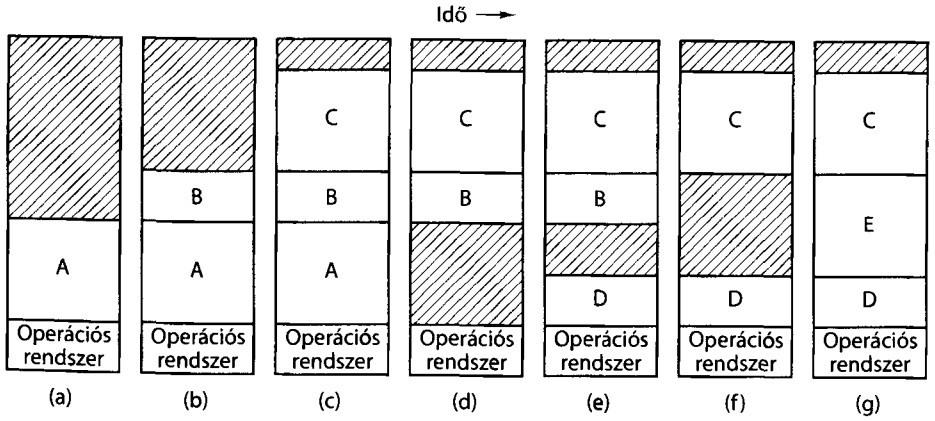
A kötegelt feldolgozású rendszereknél a memória rögzített méretű partíciókra osztása egyszerű és hatékony módszer. Minden munka betöltődik a megfelelő partícióba, amikor a sor elejére kerül, aztán a befejeződéséig a memóriában marad. Amíg a munkák elegendő memóriát és processzoridőt kapnak, nincs ok bonyolultabb modell alkalmazására.

Más a helyzet az időosztásos vagy a grafikus felületű rendszereknél. Gyakran nincs elég memória az összes aktív processzus befogadásához, így a felesleges processzusokat a lemezen kell tartani, és dinamikusan kell betölteni futtatásra.

A memóriakezelésnek két általános, (részben) az adott hardvertől függő megközelítése van. Az egyszerűbb stratégia a **csere (swapping)**, a processzusokat teljes egészükben mozgatja a memória és a lemez között. A másik stratégia neve **virtuális memória**; ez akkor is engedi a programokat futni, ha csak egy részük van a központi memóriában. Először a cserét, majd a 4.3. alfejezetben a virtuális memóriát vizsgáljuk.

A cserélő rendszerek működése a 4.3. ábrán látható. Kezdetben csak az *A* processzus van a memóriában, ezután behozzuk a lemezről a *B* és a *C* processzust. A 4.3.(d) ábrában az *A* processzust kitettük a lemezre, ezután bejön a *D*, és a *B*-t kivisszük a lemezre, végül *A* ismét bejön. Mivel *A* most más helyen van, az általa tartalmazott címeket relokálni kell vagy a szoftver által, amikor betöltődik, vagy a hardver által (ez a valószínűbb) a program futása közben.

A rögzített partíciójú rendszerek (lásd 4.2. ábra) és a változó partíciójú rendszerek között (lásd 4.3. ábra) az a fő különbség, hogy az utóbbiban a partíciók száma, helye és mérete dinamikusan változik, ahogy a processzusokat mozgatjuk a központi memória és a lemez között. A változó partíciójú rendszerekben rejlő rugalmasság jobb memóriakihasználtságot eredményez, de bonyolultabbá teszi a lefoglalást és felszabadítást.



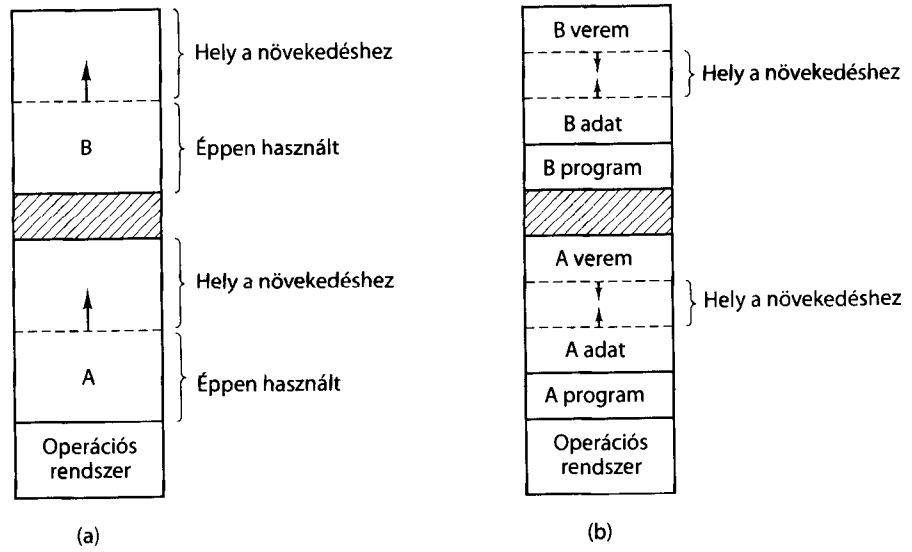
4.3. ábra. *A memória változása a processzusok indulásakor és leállásánál. A vonalkázott terület a szabad memóriát jelzi*

Amikor a csere sok lyukat hoz létre a memóriában, a processzusok mozgatásával ezeket egy nagy lyukká lehetne összeolvasztani. Ezt a technikát **memóriatömörítésnek** nevezik. Nagy processzorigénye miatt általában nem használják. Például egy 1 GB-os gépen, amely 2 GB/s (0,5 ns/bájt) sebességgel tud másolni, 0,5 másodpercig tart az egész memória tömörítése. Ez nem tűnik túl soknak, de zavaró lehet egy videót néző felhasználónak.

Amikor az operációs rendszer létrehoz vagy behoz egy processzust, döntést kell hoznia, hogy mennyi memóriát foglaljon le számára. Ha a processzust egy rögzített mérettel hozza létre, és ez a méret nem változik, akkor egyszerű a dolog, az operációs rendszernek pontosan ennyi memóriát kell lefoglalnia, se többet, se kevesebbet.

Azonban probléma adódik, ha a processzus adatszegmense növekszik, például dinamikusan foglal memóriát, mint sok programnyelvben. Ha a processzus mellett egy lyuk van a memóriában, akkor ezt lefoglalhatja és hozzáadhatja a processzus memóriaterületéhez. Ha a processzus mellett egy másik processzus van a memóriában, akkor az operációs rendszer elmozgathatja a növekvő processzust egy nagyobb lyukba, vagy egy esetleg több processzust kirakhat a lemezre, hogy elegendő helyet csináljon. Ha a processzus már nem tud tovább nőni a memóriában, és a lemezen kijelölt rész is tele van, akkor a processzusnak várnia kell vagy le kell állnia.

Ha várható, hogy a legtöbb processzus növekszik futás közben, akkor jó ötletnek látszik egy külön memóriarészt foglalni, amikor egy processzust behozunk vagy áthelyezünk, hogy csökkentsük a mozgatások számát. Amikor egy processzust kiviszünk a lemezre, akkor elég csak a ténylegesen használt memóriaterüle-



4.4. ábra. *(a) Növekvő adatszegmens számára lefoglalt memóriaterület. (b) Növekvő adat- és veremszegmens számára lefoglalt memóriaterület*

tet kimásolni, az extramemóriát felesleges. A 4.4.(a) ábrán látható egy memóriakép, amelyben két processzus van, mindegyikhez a növekedéshez foglalt egy-egy külön extra memóriarész.

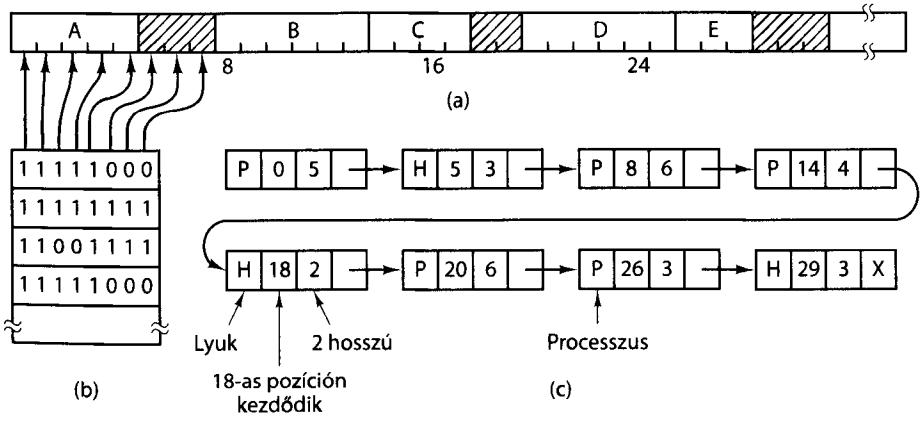
Ha a processzusnak két növekvő szegmense van, például az adatszegmens (heap) a dinamikusan létrehozott és megszüntetett változóknak, és a veremszegmens a lokális változóknak és a visszatérési címeknek, akkor egy jó megoldás látható a 4.4.(b) ábrán. Minden processzusnak van egy veremszegmense, amely a processzusnak lefoglalt memória tetejétől lefelé terjeszkedik, és egy adatszegmense, amely a programkód után következik, és felfelé növekszik. A két szegmens között levő memória egyszerre mindkettőhöz tartozik. Ha túlcsordul, akkor a processzust egy nagyobb lyukba kell áthelyezni, vagy ki kell vinni a lemezre, vagy le kell állítani.

4.2.1. Memóriakezelés bittérképpel

Amikor dinamikusan foglalunk memóriát, akkor azt az operációs rendszernek kezelnie kell. Két módszer van a memóriahasználat nyilvántartására: a bittérkép és a szabadlista. Ebben és a következő alfejezetben ezt a két módszert tárgyaljuk.

A bittérképes módszernél a memória néhány szónyi vagy kilobájtnyi allokációs egységekre osztott. Minden allokációs egységhez tartozik egy bit a bittérképen, ami 0, ha az egység szabad, és 1, ha foglalt (esetleg fordítva). A 4.5. ábra a memória egy részletét és a hozzá tartozó bittérképet mutatja.

Az allokációs egység mérete fontos tervezési szempont. Ha az egység kicsi, akkor a bittérkép nagy. Ha az egység 4 bájt, akkor 32 bitnyi memóriához egy bit szükséges a térképen, $32n$ bit memória n bitet használ a térképből, így a memória 1/33-ad része lesz a térkép. Ha az allokációs egység nagy, akkor a bittérkép kicsi,



4.5. ábra. *(a) A memória egy részlete öt processzussal és három lyukkal. A beosztások a foglalási egységeket mutatják. A vonalkázott terület (a bittérképen 0) szabad. (b) A megfelelő bittérkép. (c) Ugyanezek az adatok listában ábrázolva*

de jelentős mennyiségű memória megy veszendőbe a processzusok utolsó egységéből, ha a processzusok mérete nem pontos többszöröse az allokációs egységnek.

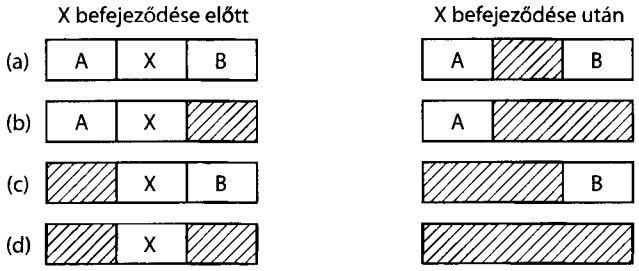
A bittérkép a memóriakezelés egyszerű módja rögzített mennyiségű memória adminisztrációra való felhasználásával, mert a bittérkép mérete csak a memória és az allokációs egység méretétől függ. A fő probléma az, ha egy k méretű processzust akarunk a memóriába rakni, akkor a memóriakezelőnek k darab egybefüggő 0 bitet kell keresnie a térképen. A térképen egy adott hosszúságú sorozat keresése lassú művelet, mert a keresett átnyúlhat a szóhatárokon. Ez egy ellenérv a bittérképes módszerrel szemben.

4.2.2. Memóriakezelés láncolt listákkal

A memóriakezelés másik módja az, hogy láncolt listába fűzzük a szabad és a foglalt szegmenseket; itt szegmens alatt egyaránt értjük a processzusokat és a lyukakat két processzus között. A 4.5.(a) ábrán látható memóriarészlet láncolt listás reprezentációját a 4.5.(c) ábra mutatja. A listának minden eleme a hozzá tartozó lyuk (H) vagy processzus (P) kezdőcímét, hosszát és a következő listaelem címét tartalmazza.

Ebben a példában a szegmensek listája kezdőcím szerint rendezett. Ennek a sorrendnek az az előnye, hogy ha egy processzus befejeződött vagy kitettük a lemezre, akkor a lista egyszerűen karbantartható. Egy befejeződött processzusnak normális esetben két szomszédja van (kivéve, ha a memória elején vagy végén található); ezek egyaránt lehetnek processzusok által foglalt területek vagy lyukak; a négy lehetőség a 4.6. ábrán látható. Az (a) esetben az eddig foglalt processzusterületet egy lyukkal helyettesítjük, a (b) és a (c) esetben két listaelemet egyesítünk egyetlen lyukká (a lista egy elemmel rövidebb lesz), a (d) esetben pedig három elemből készítünk egy lyukat, és a lista két elemmel lesz rövidebb. Mivel a processzustábla bejegyzése magára a befejeződött processzus listaelemére mutat, a listát célszerű kétszeresen láncolni. Így könnyebb ellenőrizni, hogy a keletkezett lyuk összeolvasztható-e valamelyik szomszédjával.

Kezdőcím szerint rendezett listában számos algoritmus használatos a memória lefoglalására az új (vagy a lemezről éppen behozott) processzusok számára. Feltesszük, hogy a memóriakezelő tudja, hogy mennyi memóriát kell lefoglalni.



4.6. ábra. *A befejeződő X processzus négy szomszédsági kombinációja*

A legegyszerűbb algoritmus a **first fit**. A processzuskezelő addig keres a szegmensek listájában, amíg meg nem találja az első megfelelő méretű lyukat. Ezt a lyukat két részre vágja, az egyik lesz a processzusé, a másik megmarad szabadnak, kivéve azt a statisztikailag valószínűtlen esetet, amikor a lyuk pontosan akkora, mint a processzus. A first fit a leggyorsabb algoritmus, mert a lehető legkevesebbet keres.

A first fit egy változata a **next fit**. A first fithez hasonlóan működik, kivéve azt, hogy megjegyzi, hol találta az előző megfelelő lyukat, innen indul a következő keresés, nem pedig a lista elejétől, mint a first fitnél. Bays (Bays, 1977) szimulációi megmutatták, hogy a next fit valamivel rosszabb teljesítményű, mint a first fit.

Egy másik jól ismert algoritmus a **bestfit**. A best fit az egész listát végigkeresi, és a legkisebb alkalmas lyukat adja meg. Ahelyett, hogy megállna egy nagy lyuknál, amire később szükség lehet, a best fit megpróbál a szükségeshez közelebbi méretű lyukat keresni.

Példaként a first fit és a best fit algoritmusokra tekintsük újból a 4.5. ábrát. Ha egy 2 méretű blokk szükséges, akkor ezt a first fit az 5-ös címen foglalja le, a best fit pedig a 18-as címen.

A best fit lassabb, mint a first fit, mert minden alkalommal átnézi az egész listát. Valamelyest meglepő, hogy a first fitnél és a next fitnél több memóriát veszteget el, ugyanis hajlamos arra, hogy kicsi, használhatatlan lyukakat csináljon a memóriában. A first fit átlagban nagyobb lyukakat produkál.

A kis lyukak méretét vizsgálva jó ötletnek tűnik a **worst fit** algoritmus, amely a legnagyobb lyukat választja ki, így a szabadon maradt lyukdarab elég nagy ahhoz, hogy használható legyen. A szimulációk azonban megmutatták, hogy a worst fit nem olyan jó, mint a többi eljárás.

Mind a négy algoritmus felgyorsítható, ha a processzusok és a lyukak számára különálló listákat készítünk. Ilyenkor a lyukak, és nem a processzusok vizsgálatának szentelhetünk minden energiát. Ennek az árát a memória lefoglalásánál és felszabadításánál fizetjük meg, mert a felszabadult szegmens helyét be kell illeszteni a helyére a lyukak listájába, és el kell távolítani a processzuslistából.

Ha külön processzus- és lyuklistát használunk, akkor a lyukak listáját a méret szerint is rendezhetjük, így a best fit gyorsabb lesz. Ha a best fit a legkisebb lyuktól a legnagyobb felé haladva keres, akkor az első megfelelő méretű lyuk jó lesz, ezért nem szükséges tovább keresni. Ha a lyuklista méret szerint rendezett, akkor a first fit és a best fit cgyformák, a next fit pedig értelmetlen.

Ha különálló listákat alkalmazunk, akkor lehetőség nyílik egy kis javításra. A külön felépített lyuklista [lásd 4.5.(c) ábra] helyett használhatjuk magukat a lyukakat listaelemként. Minden lyuk első szava a lyuk mérete, a második pedig a következő listaelemre mutat. Így nincs szükség a külön listára és a foglaltságot jelző bitre.

A következő algoritmus a **quick fit**, amely a leggyakrabban kért méretekhez külön lyuklistákat épít. Például egy *n* elemű táblázatban az első elem mutathat a 4 KB-os, a második a 8 KB-os, a harmadik pedig a 12 KB-os lyukak listájára, és így tovább. Egy 21 KB-os lyukat egyaránt berakhatunk a 20 KB-os listába vagy a páratlan méretű lyukak külön listájába. A quick fit nagyon gyorsan megtalálja a megfelelő méretű lyukat, de megvan ugyanaz a hátránya, mint a többi méret szerint rendezett listát használó algoritmusnak, miszerint költséges ellenőrizni, hogy egy felszabadult szegmenst a szomszédokkal össze tudjuk-e vonni. Ha nem működik a lyukak összevonása, akkor a memória sok kis lyukra darabolódik fel, amelyekbe nem férnek bele a processzusok.

4.3. Virtuális memória

Már sok évvel ezelőtt felmerült az a probléma, hogy a programok túl nagyok voltak, és nem fértek bele a rendelkezésre álló memóriába. Gyakran alkalmazott megoldás volt a program rétegekre darabolása, az **overlays**. Először a 0-s sorszámú réteg kezd futni, amikor befejeződik, akkor hívja a következő réteget. Néhány bonyolultabb rendszer megengedi, hogy egyidejűleg több réteg is a memóriában legyen. Az operációs rendszer a rétegeket a lemezen tartja, és szükség szerint cserélgeti a memória és a lemez között.

Bár a program futása alatt a rétegek mozgatása a lemez és a memória között az operációs rendszer feladata, a döntést, hogy miként darabolja rétegekre a programot, a programozónak kell meghoznia. Egy nagy program kis részekre bontása időrabló és unalmas munka. Nem tartott sokáig, amíg valaki kitalálta, hogyan lehetne az egész munkát az operációs rendszerre bízni.

Az erre kifejlesztett módszert **virtuális memória**nak nevezik (Fotheringham, 1961). A virtuális memória lényege, hogy a program, az adat és a verem együttes mérete meghaladhatja a fizikai memória mennyiségét. Az operációs rendszer csak a program éppen használt részét tartja a memóriában, a többi a lemezen van. Például egy 256 MB-os memóriában is futhat egy 512 MB-os program, ha az operációs rendszer mindig a megfelelő 256 MB-os részét tartja bent a memóriában.

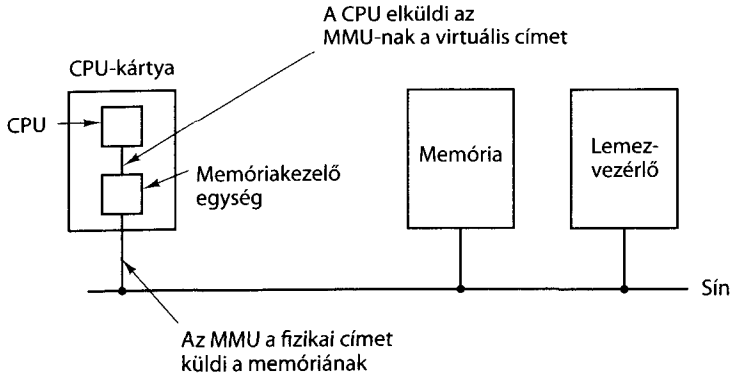
A virtuális memória a multiprogramozást támogató rendszereknél is működik, ekkor több program darabjai vannak a memóriában. Ha egy program nem futhat, mert saját darabjának behozatalára, vagyis I/O-műveletre várakozik, akkor ugyanúgy, mint más multiprogramozást támogató rendszereknél, a CPU egy másik programnak adható.

4.3.1. Lapozás

A virtuális memóriát használó rendszerekben leggyakrabban a **lapozás** technikáját alkalmazzák. Minden számítógépen van egy memóriacímhalmaz, amelyet a programok elő tudnak állítani. Amikor a program egy olyan utasítást hajt végre, mint a

```
MOV REG,1000
```

akkor az a REG-regiszterbe másolja az 1000-es memóriarekesz tartalmát (vagy a géptől függően fordítva). A címek indexeléssel, bázis- vagy szegmensregiszterből, esetleg egyéb módon generálódhatnak.



4.7. ábra. *Az MMU helye és szerepe. Itt az MMU a CPU részeként van ábrázolva, mivel napjainkban ez az általános. Logikailag azonban különálló integrált áramkör is lehetne, mint ahogy az az elmúlt években volt*

Ezeket a program által generált címeket **virtuális címek**nek, ezek halmazát pedig **virtuális címtartomány**nak nevezik. Egy virtuális memória nélküli gépben a virtuális címek közvetlenül a memóriasínre kerülnek, és a fizikai memóriából a címmel megegyező memóriaszót olvassák vagy írják. Ha virtuális memóriát használunk, akkor a virtuális címek nem kerülnek közvetlenül a memóriasínre, ehelyett a memóriakezelő egységbe – **MMU (Memory Management Unit)** – kerülnek, amely a virtuális címeket képezi le a fizikai címekre (lásd 4.7. ábra).

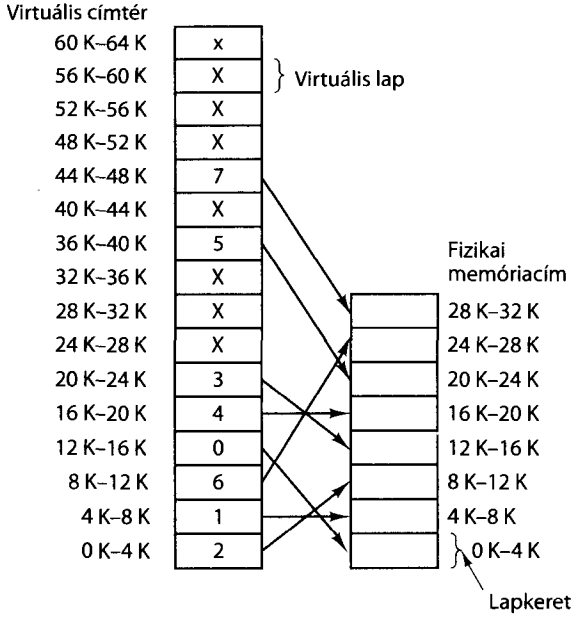
A 4.8. ábra erre a leképezésre mutat egy egyszerű példát. Példánkban a számítógép 16 bites címeket tud generálni 0-tól 64 KB-ig. Ezek a virtuális címek. A gépnek azonban csak 32 KB fizikai memóriája van, így bár írhatunk 64 KB-os programot, azt nem lehet egészben betölteni a memóriába és futtatni. A program teljes egészében a lemezen van, és csak azok a részek töltődnek be, amelyekre szükség van.

A virtuális címteret **lap**nak **(page)** nevezett egységekre osztják, ennek megfelelő egység a fizikai memóriában a **lapkeret**. A lapok és a lapkeretek mindig pontosan egyforma méretűek. Ebben a példában ez 4 KB, a létező rendszerekben a lapméret 512 bájt és 1 MB közé esik. 64 KB-os virtuális címtér és 32 KB fizikai memória esetén 16 virtuális lap és 8 lapkeret van. A memória és a lemez közötti átvitel laponként történik.

Amikor például a program

```
MOVE REG,0
```

utasítással a 0-s címet éri el, akkor az MMU megkapja a 0-s virtuális címet. Ez a virtuális cím a 0-s lapra esik (0–4095), az a lap a 2-es lapkeretben (8192–12287) található. Így az MMU a kapott virtuális címet a 8192-es fizikai címre képezi le, és ezt a címet küldi ki a memóriasínre. A memória semmit sem tud az MMU-ról, csak megkapja és teljesíti a 8192-es cím írására vagy olvasására vonatkozó kérést. Ehhez hasonlóan az MMU a 0 és 4095 közti virtuális címeket a 8192 és 12287 közötti fizikai címtartományra képezi le.



4.8. ábra. *A laptábla által megadott kapcsolat a virtuális és a fizikai címek között*

Hasonlóan a

```
MOVE REG,8192
```

utasítás a

```
MOVE REG,24576
```

alakra transzformálódik, mert a 8192-es virtuális cím a 2-es lapra esik, ez a lap a 6-os lapkeretbe esik (24576–28671-es fizikai címek). Harmadik példa a 20500-as virtuális cím, ez 20 bájtra van az 5-ös lap kezdetétől, így a 12288 + 20 = 12308-as fizikai címre képződik le.

Egyedül az MMU leképezése azonban nem oldja meg azt a problémát, hogy a virtuális címtér nagyobb, mint a fizikai memória. Mivel csak nyolc fizikai lapkeret van, az MMU a 4.8. ábrán csak nyolc lapot képez le a fizikai memóriára. A nem leképezhető lapokat kereszt jelzi az ábrán. A hardverben egy **jelenlét/hiány bit** követi nyomon, hogy mely lapok vannak jelenleg fizikailag jelen a memóriában.

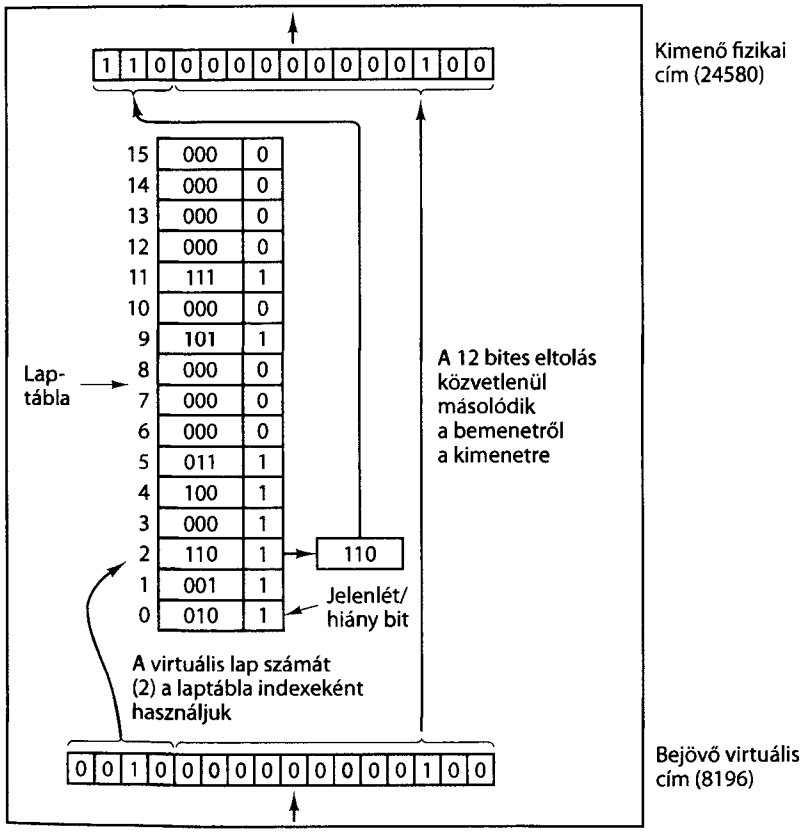
Mi történik, ha a program egy olyan lapra hivatkozik, ami nincs a fizikai memóriában? Például a

```
MOVE REG,32780
```

utasítás a 8-as virtuális lap 12. bájtjára hivatkozik. Az MMU észleli, hogy a lap nincs a memóriában, és egy **laphiba** megszakítással jelez az operációs rendszernek. Az operációs rendszer vesz egy lapkeretet, a tartalmát kiírja a lemezre, behozza ide a hivatkozott lapot a lemezről, módosítja a laptérképet, majd a megszakítást okozó utasítástól folytatja a program végrehajtását.

Például ha az operációs rendszer az 1-es lapkeret kidobása mellett dönt, akkor a 8-as virtuális lapot a 4096-os fizikai címre tölti be, és két helyen megváltoztatja a laptáblát. Elsőként az 1-es virtuális lapot meg kell jelölni, hogy nincs bent a memóriában, így elkap minden 4–8 KB közötti virtuális címre történő hivatkozást. Ezután a laptáblában a 8-as virtuális lapnál levő keresztet egy 1-esre cseréli, így a megszakítást okozó utasítást újra végrehajtva a 32780-as virtuális cím a 4108-as fizikai címre képződik le.

Most vizsgáljuk meg, hogyan működik az MMU, és hogy miért használhatunk kettőhatvány méretű lapokat. A 4.9. ábrán láthatjuk, hogyan képezi le az MMU a 8196-os (binárisan 0010000000000100) virtuális címet a 4.8. ábra laptérképe alapján. Az MMU a bejövő 16 bites virtuális címet egy 4 bites lapszámra és egy 12 bites



4.9. ábra. *Az MMU belső működése 16 darab 4 KB-os lap esetén*

offsetre vágja szét. A 4 bites lapszámokkal a 16 lapot, a 12 bites offsetekkel pedig egy lapon belül mind a 4096 címet reprezentálhatjuk.

A lapszámot a **laptábla** indexeként használjuk, amely az adott virtuális laphoz tartozó lapkeretet mutatja meg. Ha a *jelenlét/hiány* bit 0, akkor megszakítást okoz az operációs rendszer felé. Ha a bit 1, akkor a lapkeret számát a kimeneti regiszter felső három bitjébe másolja, a többi 12 bit a bejövő cím offset-része lesz. Ez a 15 bit együtt a fizikai cím, ezt küldi a memóriasínre a fizikai memória címzéséhez.

4.3.2. Laptáblák

A legegyszerűbb esetben a virtuális címek fizikai címekre történő leképezése úgy történik, ahogyan az előbb leírtuk. A virtuális címeket szétvágjuk virtuális lapszámra (felső bitek) és offset-részre (alsó bitek). Például egy 16 bites címmel és 4 KB-os lapmérettel a felső 4 bit jelezhetné a lehetséges 16 lap egyikét, míg az alsó 12 bit tartalmazná a címet (0–4095) a kiválasztott lapon belül. Más elosztás. például 3 vagy 5 bit felhasználása a lapra szintén lehetséges. A különböző vágások különböző méretű lapokat eredményeznek.

A lapsorszámot a laptábla indexeként használjuk, hogy megtaláljuk a lap bejegyzését, amelyben a lapkeret száma található, ha a lap a memóriában van. A virtuális címben a lapsorszámot a lapkeret számára cseréljük, és az így kapott fizikai címet küldjük a memóriához.

A laptábla célja az, hogy a virtuális lapokat lapkeretekre képezzük le. Matematikai értelemben a laptábla egy függvény, ahol a virtuális lapszám az argumentum, a lapkeret száma pedig az eredmény. Ennek a függvénynek az eredményét helyettesítjük a virtuális címben a lapszám helyére, és így kapjuk a fizikai címet.

Az egyszerű leírás ellenére két fő problémával kell szembenéznünk:

1. A laptábla nagyon nagy lehet.
2. A leképezésnek gyorsnak kell lennie.

Az első pont abból következik, hogy a modern számítógépek legalább 32 bites virtuális címeket használnak. Például 4 KB-os lapméretnél a 32 bites címtér több mint egymillió lapból áll, a 64 bites címtér pedig ennél is jóval több lapot tartalmaz. Egymillió laphoz egymillió bejegyzés kell a laptáblában. Ne felejtsük el azt sem, hogy minden processzushoz egy saját laptábla tartozik, mivel saját virtuális címtere van.

A gyorsaság azért szükséges, mert a leképezést minden egyes memóriahivatkozásnál végre kell hajtani. Egy tipikus utasításnak utasításszava és gyakran memóriaoperandusa is van, így utasításonként egy, kettő vagy több memóriahivatkozás történik. Ha egy utasítás végrehajtása mondjuk 1 ns, akkor a leképezést 250 ps alatt el kell végezni, mert különben a leképezés szűk keresztmetszet lesz.

A gyors lapleképezés fontos szempont a számítógépek tervezésénél. Bár ez a probléma a csúcsgépeknél, amelyeknek nagyon gyorsaknak kell lenniük, a legfontosabb, a kisgépeknél is számít, ahol fontos az ár/teljesítmény arány. Ebben

és a következő alfejezetben megvizsgáljuk a jelenlegi számítógépekben lapozásra használt hardvermegoldásokat.

A legegyszerűbb ötlet az, mint a 4.9. ábrán is látható, hogy legyen egy gyors hardverregiszterekből álló tömb, a lap sorszáma szerint indexelve. Amikor egy processzus elindul, a rendszer betölti a memóriából a processzus laptábláját a regiszterekbe. Ennek a megoldásnak az az előnye, hogy a leképezés alatt nem kell a memóriához nyúlni, mert a laptábla a regiszterekben van. A hátránya az ára (különösen ha a laptábla nagy). Továbbá minden processzusváltásnál a laptábla regiszterekbe töltése a teljesítményt is csökkenti.

A másik véglet, amikor a laptábla teljes egészében a központi memóriában van, és egy regiszter mutat a laptábla elejére. Ilyenkor csak egy regiszter értékét kell átírni processzusváltásnál. Természetesen hátrány, hogy minden utasításnál egy vagy több memóriahivatkozás kell a laptábla eléréséhez. Ezen okok miatt ezeket a megoldásokat ritkán használják tiszta formában. A következőkben néhány nagyobb teljesítményű változatot tanulmányozunk.

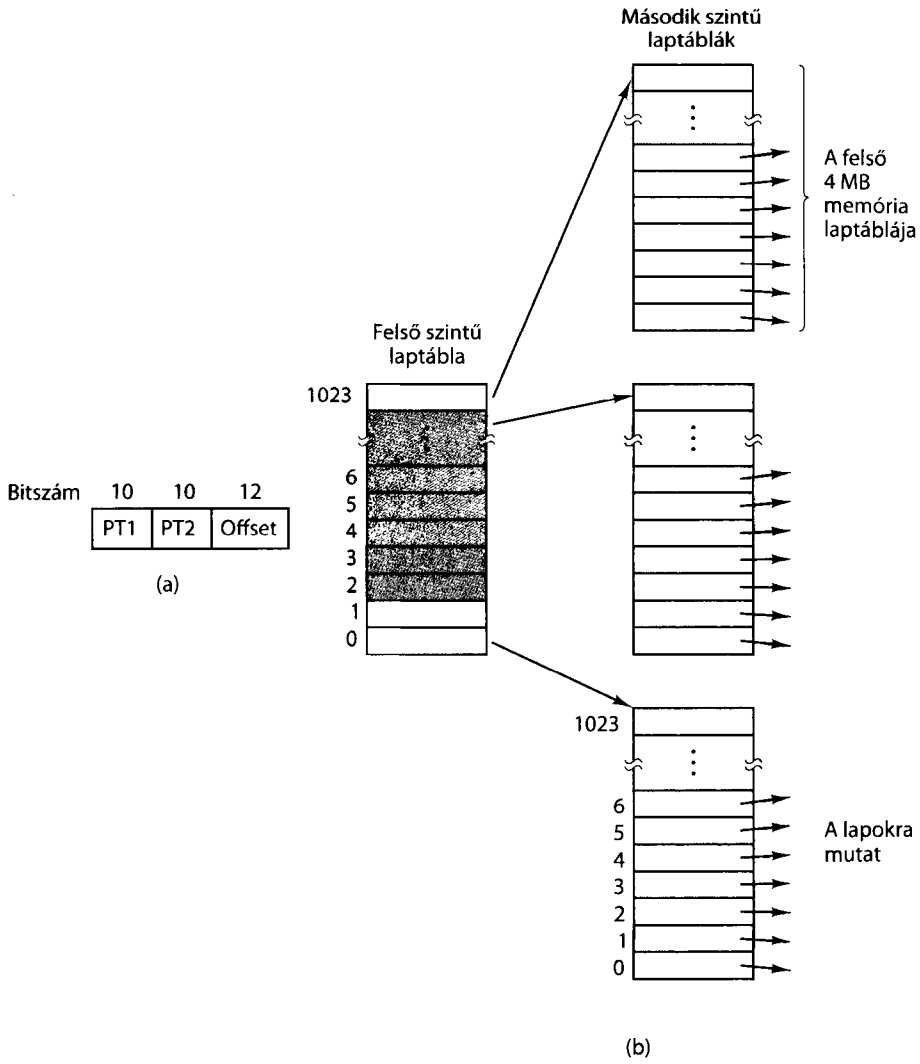
Többszintű laptáblák

Ahelyett, hogy egy nagy laptáblát állandóan a memóriában tároljunk, sok rendszerben többszintű laptáblát alkalmaznak. Egy egyszerű példa látható a 4.10. ábrán, a 32 bites virtuális címeket egy 10 bites *PT1,* egy 10 bites *PT2* mezőre és egy 12 bites *offset*-re osztjuk. Mivel az offset 12 bites, a lapméret 4 KB, és összesen 2^{20} darab virtuális lap van.

A többszintű laptáblás módszer titka az, hogy nem tart bent minden laptáblát egyszerre a memóriában, főként azokat nem, amelyekre nincs éppen szükség. Például egy 12 MB-os program esetén az első 4 MB a programkód, a második az adat, a felső 4 MB a verem, ilyenkor a verem alja és az adatok vége között egy nagy kihasználatlan lyuk van.

A 4.10.(b) ábrán láthatjuk, hogyan működik a példában a kétszintű laptábla. A bal oldalon van a felső szintű laptábla a *PT1* mezőhöz tartozó 1024 darab bejegyzéssel. Az 1024 bejegyzés mindegyike 4 MB-ot reprezentál a teljes 4 GB-os (azaz 32 bites) virtuális címtérből. Amikor az MMU egy virtuális címet kap, akkor először a *PT1* mezőjét veszi, majd ezt a felső szintű laptábla indexeként használja. A kapott bejegyzés egy második szintű laptábla lapkeretére mutat. A felső szintű laptábla 0-s bejegyzése a programkód, az 1-es az adat, az 1023-as a verem laptáblájára mutat. A többi bejegyzést nem használjuk. A *PT2* mező a kiválasztott második szintű laptábla indexelésére szolgál, az itt talált bejegyzés már a lapot tartalmazó lapkeret számát adja meg.

Példaként tekintsük a 0x00403004-es (decimálisan 4206596) virtuális címet; ez az adatszegmens 12292-es bájtja. Ennél a virtuális címnél a *PT1* = 1, a *PT2* = 3 és az *offset* = 4. Az MMU először a *PT1* alapján a felső szintű laptábla 1-es bejegyzését kapja; ehhez a 4–8 MB-ig terjedő címek tartoznak. Ezután a *PT2*-vel indexelve a második szintű laptáblában megkapjuk a 3-as bejegyzést, amelyhez a 12288–16383-as címek tartoznak a 4 MB-os szeleten belül (azaz a 4206592–4210687-es abszolút



4.10. ábra. *(a) Egy 32 bites cím két laptáblamezővel. (b) Kétszintű laptábla*

címek). Ez a bejegyzés tartalmazza a lapkeret számát, amiben a 0x00403004-es virtuális cím lapja van. Ha ez a lap nincs a memóriában, a laptáblabejegyzés *jelenlét/hiány* bitjének nulla értéke jelzi a laphibát. Ha a lap a memóriában van, akkor a második szintű tábla lapkeretszáma és az offset (4) kombinációjából megkapjuk a fizikai címet. Ezt a címet küldjük aztán a memóriasínre.

Bár a 4.10. ábrán a címtér több mint egymillió lapot tartalmaz, csak négy laptáblára van szükség: a felső szintű laptáblára, valamint a 0–4 MB-hoz, a 4–8 MB-hoz és a felső 4 MB-hoz tartozó második szintű laptáblákra. A felső szintű laptábla fennmaradó 1021 bejegyzésének *jelenlét/hiány* bitje 0, hogy az esetleges hivatko-

záskor laphibát okozzon. Ebben az esetben az operációs rendszer észreveszi, hogy a processzus egy olyan címre hivatkozott, amely nem hozzá tartozik, és egy szignált küld a processzusnak, vagy leállítja azt. Példánkban a különféle méretekre kerek számokat választottunk, és a *PT1* és *PT2* egyforma, de a gyakorlatban természetesen más értékek is használhatók.

A 4.10. ábrán látható kétszintű laptábla bővíthető három-, négy- vagy többszintűre is. Több szint nagyobb hajlékonyságot jelent, de kérdéses, hogy két szint felett megéri-e a nagyobb bonyolultság miatt.

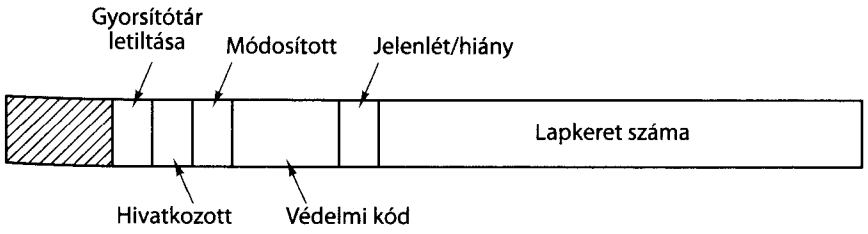
A laptáblabejegyzés struktúrája

Most pedig a laptáblák egésze helyett vizsgáljuk meg a tábla egyetlen bejegyzését! A bejegyzés pontos szerkezete erősen gépfüggő, de minden gépen nagyjából ugyanazokat az információkat tárolják. Egy példát láthatunk a 4.11. ábrán. A bejegyzés mérete gépenként változik, de általában 32 bites. A legfontosabb információ a *lapkeretszám,* az algoritmus célja ennek az értéknek az előállítása. A következő adat a *jelenlét/hiány* bit. Ha ez a bit 1, akkor a lap bent van a memóriában az előbb megadott lapkeretben, ha 0, akkor a lap nincs a memóriában, és a lapkeretszám nem tartalmaz értékes adatot. Egy olyan bejegyzés elérése, amelynek ez a bitje 0, laphibát okoz.

A *védelmi* bitek adják meg, hogy milyen elérés megengedett. A legegyszerűbb esetben ez a mező csak egy bitet tartalmaz, amely írható-olvasható lap esetén 0, csak olvasható lapra pedig 1. Bonyolultabb esetben három független bit van, egy az olvasás, egy az írás és egy a végrehajtás engedélyezésére minden egyes lapra külön-külön.

A *módosítás* és a *hivatkozás* bit a lap használatát követi nyomon. Amikor a lapra írnak, akkor a hardver automatikusan beállítja a *módosítás* bitet. A bit értéke akkor számít, amikor az operációs rendszer visszaveszi a lapkeretet. Ha a lap módosított (dirty, azaz piszkos), akkor vissza kell írni a lemezre, egyébként pedig nem (a lap clean, azaz tiszta), mert ilyenkor a lemezen levő másolat érvényes. Ezt a bitet gyakran **dirty bit**nek hívják, mert a lap állapotát adja meg.

A *hivatkozás* bit a lap hivatkozásakor (olvasásnál és írásnál egyaránt) 1 lesz. Ez az érték az operációs rendszert segíti annak eldöntésében, hogy laphibánál melyik lapot dobja ki. A nem használt lapot célszerűbb kidobni, mint a használtak. Ez a



4.11. ábra. *Egy jellegzetes laptáblabejegyzés*

bit fontos szerepet játszik számos, a fejezetben később ismertetendő lapcserélési algoritmusnál.

Végül az utolsó bit a gyorsítótár használatát tiltja le a lapra. Ez olyan lapoknál hasznos, amelyek eszközregisztereket tartalmaznak. Ha az operációs rendszer egy ciklusban egy I/O-eszköz válaszára várakozik, akkor fontos, hogy az adatot mindig az eszköztől vegye a gép, és ne a gyorsítótárban levő régi értéket használja. Ezzel a bittel a gyorsítótárazás letiltható. Azoknál a gépeknél, amelyeknek memóriába ágyazott I/O helyett külön I/O-területük van, nincs szükség erre a bitre.

Megjegyzendő, hogy a lemezcím, ahol a lap van, amikor nincs a memóriában, nem része a laptáblának. Ennek az oka az, hogy a laptábla csak a virtuális címről fizikai címre való leképezéshez szükséges információkat tárolja. A laphibák kezeléséhez szükséges adatok az operációs rendszeren belül egy szoftvertáblában vannak. A hardvernek nincs szüksége rá.

4.3.3. TLB – címfordítási gyorsítótár

A legtöbb lapozási modellnél a laptáblák nagy méretük miatt a memóriában vannak. Ez a tervezés nagy hatással van a teljesítményre. Például vizsgáljunk egy utasítást, amely egy regisztert egy másikba másol. Lapozás nélkül ez az utasítás csak egy memóriahivatkozást okoz, amikor az utasításkódot beolvassuk. Lapozásnál a laptábla kezeléséhez további memóriahivatkozások szükségesek. A végrehajtás sebességét általában az határozza meg, hogy a központi egység milyen gyorsan kapja az utasításokat és az adatokat a memóriából, két laptábla-hivatkozás egy memóriahivatkozás mellett 2/3-ával csökkenti a teljesítményt. Ilyen feltételek mellett senki sem használná ezt a módszert.

A számítógép-tervezők éveket gondolkodtak ezen, és rájöttek a megoldásra. A megoldás azon az észrevételen alapszik, hogy a programok legtöbb hivatkozása a lapok egy kis részhalmazára történik, így az ezekhez tartozó laptáblabejegyzéseket gyakran, a többit ritkábban használjuk. Ez a **hivatkozáslokalitás** egyik példája; ezzel a koncepcióval még foglalkozunk a későbbiekben.

A megoldás az, hogy egy kis hardvereszközzel szerelték fel a gépeket, amely a laptábla megkerülésével képezi le a logikai címeket a fizikai címekre. Az eszközt

Érvényes	Virtuális lap	Módosított	Védelmi kód	Lapkeret
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

4.12. ábra. *Egy TLB a lapozás felgyorsítására*

TLB-nek (**Translation Lookaside Buffer – címfordítási gyorsítótár**) vagy **asszociatív memóriának** hívják. A TLB az MMU-ban található és néhány bejegyzést tartalmaz, a 4.12. ábrán például 8-at, de ritkán többet 64-nél. Mindegyik bejegyzés egy lap adatait tartalmazza: a virtuális lapszám, a módosítási bit, a védelmi kód (olvasási/írási/futtatási jogok) és a valós lapkeret száma, ahol a lap van. Ezek a mezők a laptáblabeli bejegyzés mezőinek felelnek meg. Egy további bit jelzi, hogy a bejegyzés érvényes (azaz használatban van), vagy nem.

A 4.12. ábrán látható TLB-t például egy olyan processzus generálhatta, amely egy 19, 20, 21-es virtuális lapon levő ciklust futtat, mert ezek a lapok a védelmi kód szerint olvashatók és végrehajthatók. Az adatok (mondjuk egy feldolgozott tömb) a 129-es és a 130-as lapon vannak, a 140-es lap tartalmazza a tömbszámításnál használt indexeket. Végül a verem a 860-as és a 861-es lapon van.

Vizsgáljuk meg, hogyan működik a TLB. Amikor az MMU egy virtuális címet kap, akkor először megvizsgálja, hogy a lap száma benne van-e a TLB-ben. A vizsgálat minden bejegyzésre egyidejűleg (párhuzamosan) zajlik. Ha megtalálta a lapot, és az elérés nem sérti a védelmi kódot, akkor a lapkeret számát közvetlenül a TLB-ből veszi, nem kell a laptáblához fordulni. Ha a lap száma benne van a TLB-ben, de az utasítás egy olyan lapra próbál írni, amely csak olvasható, akkor ugyanúgy védelmi hiba generálódik, mintha a laptáblát használtuk volna.

Érdekes, hogy mi történik akkor, ha a virtuális lapszám nincs a TLB-ben. Ilyenkor az MMU a laptáblához fordul, majd kidob egy bejegyzést a TLB-ből, és az éppen hivatkozott lap adataival helyettesíti. Ha erre a lapra újra szükség lesz, akkor megtalálja a TLB-ben. Ha egy bejegyzést kidobunk a TLB-ből, akkor a módosítás bitjét vissza kell másolni a memóriában levő laptáblabejegyzésbe. A többi érték ugyanolyan. Ha a laptáblából töltünk egy bejegyzést a TLB-be, akkor minden mezőt a memóriából kell venni.

Szoftveres TLB-kezelés

Mostanáig feltettük, hogy minden lapozásos virtuális memóriájú gépnek hardveres laptáblája és TLB-je van. A TLB vezérlése és hibáinak kezelése teljes egészében az MMU dolga. Az operációs rendszer csak akkor kap megszakítást, ha a lap nincs a memóriában.

A múltban helytálltak ezek a feltételezések, azonban a modern RISC-gépeknél, mint a SPARC MIPS, Alpha, HP PA és PowerPC, majdnem a teljes lapkezelés szoftveres. Ezeknél a gépeknél a TLB-bejegyzéseket is az operációs rendszer tölti fel. Ha egy keresett lap nincs a TLB-ben, akkor a laptáblához fordulás helyett egy TLB-hiba generálódik, és az operációs rendszer kapja meg a vezérlést. Az operációs rendszernek kell megkeresnie a lapot, kivennie egy bejegyzést a TLB-ből és beraknia az újat, majd a laphibát okozó utasítástól újra kell indítania a processzszust. Ezt természetesen nagyon kevés utasítással kell megtennie, mert TLB-hiba sokkal gyakrabban fordul elő, mint laphiba.

Ha a TLB elég nagy ahhoz (mondjuk 64 bejegyzés), hogy csökkentse a hibaarányt, akkor a szoftveres TLB-kezelés elég hatékony lesz. A fő nyereség az egyszerűbb MMU, amely által hely szabadul fel a CPU-lapkán a gyorsítótárnak vagy egyéb teljesítménynövelő dolognak. A szoftveres TLB-kezelést Uhlig és mások (Uhlig et al., 1994) alaposan tárgyalják.

Számos stratégiát dolgoztak ki a szoftveres TLB-kezelés teljesítményének javítására. Bala és mások (Bala et al., 1994) megközelítése egyaránt csökkenti a TLB-hibák számát és a hiba kezelésének költségét. A TLB-hibák elkerüléséhez az operációs rendszer kiszámíthatja, hogy a következőkben mely lapok használata a legvalószínűbb, és az ezekhez tartozó bejegyzéseket előre betöltheti a TLB-be. Például ha egy kliensprocesszus egy távoli eljáráshívást végez egy ugyanazon a gépen futó szerverprocesszushoz, akkor valószínű, hogy a szervernek nemsokára futnia kell. Ezt tudván, amikor a rendszer a távoli eljáráshívást dolgozza fel a küldéshez, akkor megkeresheti, hogy hol a szerver kódja, adata és verme, és betöltheti ezen lapok bejegyzéseit a TLB-be.

A TLB-hibák kezelésének hardveres és szoftveres esetben egyaránt az a módja, hogy a laptáblából indexeléssel kiválasztjuk a hivatkozott lapot. A szoftveres kezelésnél az a probléma, hogy a laptáblát tartalmazó lap indexe nem biztos, hogy a TLB-ben van, így ez újabb TLB-hibát okozhat a feldolgozás alatt. Ezeket a hibákat kiküszöbölhetjük egy nagy (például 4 KB-os vagy nagyobb) szoftveres gyorsítótár használatával, amely TLB-bejegyzéseket tartalmaz, és lapja mindig a TLB-ben van. Ha először ezt a gyorsítótárat ellenőrzi, akkor az operációs rendszer jelentősen csökkenteni tudja a TLB-hibák számát.

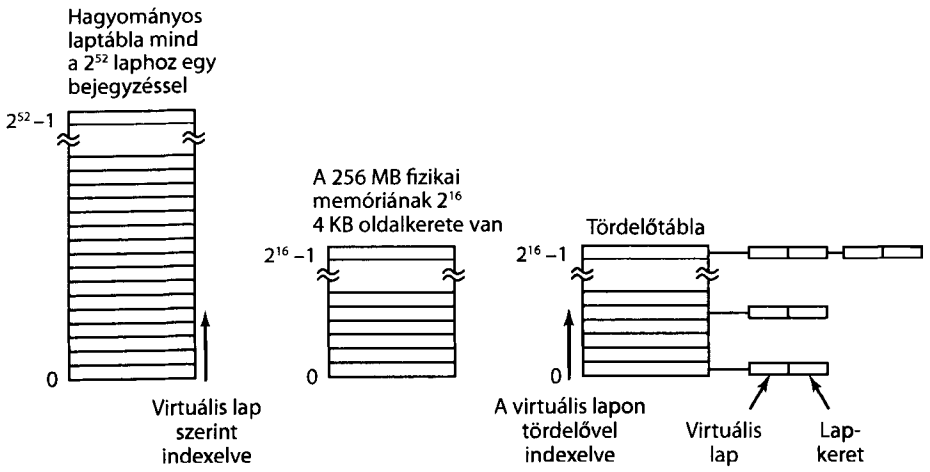
4.3.4. Invertált laptáblák

A hagyományos laptábla egy tömb, amelyben minden laphoz pontosan egy bejegyzés tartozik, és a virtuális lapszámmal indexelhető. Ha a címtér 2^{32} bájtos és a lapméret 4 KB, akkor több mint egymillió laptáblabejegyzés szükséges. Egy ilyen laptábla minimum 4 MB-ot foglal el. Nagyobb rendszerekben ez a méret valószínűleg elfogadható.

Azonban a 64 bites számítógépek elterjedésével drasztikusan megváltozik a helyzet. Ha a címtér 2^{64} bites, 4 KB-os lapméretnél több mint 10^{52} bejegyzést tartalmazó táblára van szükségünk. Amennyiben minden bejegyzés 8 bájtos, a tábla több mint 30 millió gigabájtos. Nem lehetséges most sem, és valószínűleg a következő évtizedekben sem lesz az, hogy csak a laptáblának több mint harmincmillió gigabájt legyen legfoglalva. Ezért valami más megoldást kell találni a 64 bites virtuális címtér lapozásához.

Az egyik megoldás az invertált laptábla módszere. Ennél a valós tár minden lapkeretéhez tartozik egy bejegyzés ahelyett, hogy a virtuális címtér minden lapjához lenne egy. Például 64 bites virtuális címtér, 4 KB-os lapméret és 256 MB RAM esetén az invertált laptábla csak 65 536 bejegyzést tartalmaz. Egy bejegyzés azt tartalmazza, hogy az adott lapkeretet melyik processzus melyik lapja használja.

Bár az invertált laptábla sok helyet takarít meg, van egy nagy hátránya: nehezebb a virtuálisról fizikaira történő címfordítás. Amikor az n processzus a p lapra hivatkozik, a rendszer nem találja meg a fizikai lapot a laptábla p-vel való indexe-



4.13. ábra. *A tradicionális és az invertált laptábla összehasonlítása*

lésével. Ehelyett az invertált laptáblában meg kell keresni az (n, p) pároshoz tartozó bejegyzést. Sőt ezt a keresést minden hivatkozáskor meg kell csinálni, nemcsak laphiba esetén. Egy 64 KB-os táblázatban való keresés minden memóriahivatkozásnál nem teszi a számítógépet villámgyorssá.

A kiutat ebből a helyzetből a TLB használata jelenti. Ha a TLB az összes gyakran használt lapot tartalmazza, akkor a címfordítás éppen olyan gyors, mint a rendes laptábla alkalmazásával. TLB-hiba esetén azonban az invertált laptáblában szoftveresen kell keresni. Ezt a keresést például egy virtuális címek kivonatait tartalmazó hasító- (hash) tábla alkalmazásával valósíthatjuk meg. Az olyan virtuális lapokból, amelyek a memóriában vannak és a kivonatuk azonos, egy-egy láncolt listát képezünk (lásd 4.13. ábra). Amennyiben a kivonattáblának annyi helye van, amennyi fizikai lapja a gépnek, az átlagos lánchossz egy lesz, nagyban gyorsítva a leképezést. Amikor a lapkeret számát megtaláltuk, az új párt (fizikai, virtuális) beszúrhatjuk a TLB-táblába, és a hibás műveletet újraindíthatjuk.

Jelenleg az IBM-, Sun- és a Hewlett-Packard-munkaállomásokon használnak invertált laptáblát, de a 64 bites rendszerekkel az invertált laptábla is terjed. Az invertált táblák létfontosságúak ezeken a gépeken. Egyéb megoldások nagyméretű virtuális memória kezelésére lásd (Huck és Hays, 1993; Talluri és Hill, 1994; Talluri et al., 1995). További információ a virtuális memóriával kapcsolatos hardverkérdésekről (Jacob és Mudge, 1998) művében található.

4.4. Lapcserélési algoritmusok

Amikor laphiba keletkezik, akkor az operációs rendszernek ki kell választania egy lapot, amelyet kidob a memóriából, hogy helyet csináljon a behozandó lapnak. Ha a kidobandó lapot módosították, akkor a tartalmát vissza kell írni a lemezre. Ha a lapot nem módosították (például a lap programkódot tartalmaz), akkor a lemezen lévő példány ugyanolyan, így nincs szükség visszaírásra. A behozott lap egyszerűen csak felülírja a kidobott lapot a lapkeretben.

Bár a kidobandó lapot véletlenszerűen is ki lehetne választani, a rendszer teljesítménye sokkal jobb lesz, ha egy ritkán használt lapot dobunk ki. Ha egy gyakran használt lapot távolítunk el, akkor valószínű, hogy rövidesen vissza kell hozni, és ez extraterhelést okoz. A lapcserélési algoritmusokkal több elméleti és gyakorlati munka foglalkozik, a következőkben a legfontosabb algoritmusokból mutatunk be néhányat.

Érdemes megjegyezni, hogy a „lapcsere"-probléma a számítógép-tervezés más területein is felmerül. A legtöbb számítógép rendelkezik egy vagy több memóriagyorsítótárral, amelyek a legutoljára használt 32 vagy 64 bájtos memóriablokkokat tartalmazzák. Amikor egy gyorsítótár megtelik, néhány blokkot el kell távolítani belőle. Ez a probléma ugyanaz, mint a lapcsere, a különbség csak a rövidebb időskála (néhány nanoszekundum és nem néhány milliszekundum alatt kell végrehajtani). A rövidebb időskála magyarázata egyszerű: a hiányzó gyorsítótárblokkot a memóriában keresik, ennek viszont nincs keresési ideje és nincs rotációs késleltetése.

Egy másik példa a webböngészőben található. A böngésző eltárolja a merevlemezen lévő gyorsítótárában az előzőleg látogatott weboldalakat. Mivel a gyorsítótár mérete gyakran előre meghatározott, így amennyiben a felhasználó sokat használja a böngészőt, a gyorsítótár nagy valószínűséggel betelik. Amikor egy weboldalt be kell tölteni, a böngésző ellenőrzi, hogy az adott oldal megvan-e a gyorsítótárban, illetve ha megvan, akkor frissebb-e az aktuális oldal, mint az eltárolt. Amennyiben az oldal nincs a gyorsítótárban, vagy újabb, mint a gyorsítótárban lévő, a böngésző letölti. Amennyiben újabb, akkor lecseréli vele a régi verziót a gyorsítótárban. Ha a gyorsítótár betelik, dönteni kell arról, hogy melyik weboldalt törölje ki. A megfontolás hasonló, mint a virtuális memória esetében, azzal a lényeges különbséggel, hogy itt a gyorsítótárban lévő lapokat nem írják, és ezek a szerverre sem másolják vissza őket. A virtuális memóriarendszerben a gyorsítótárban lévő lapok lehetnek tiszták vagy piszkosak.

4.4.1. Az optimális lapcserélési algoritmus

A legjobb lapcserélési algoritmust egyszerű leírni, de lehetetlen megvalósítani. Az algoritmus a következőképpen működik: laphiba esetén a memóriában van a lapoknak egy részhalmaza. Ezek közül a lapok közül egyre hivatkozunk a következő utasításban (ez a lap tartalmazza az utasítást). A többi lapra esetleg csak tíz, száz vagy ezer utasítás múlva lesz szükség. Minden lap megcímkézhető azzal a számmal, ahány utasítás végrehajtódik, mielőtt először hivatkozunk rá.

Az optimális lapcserélési algoritmus szerint a legnagyobb számmal jelzett lapot kell eltávolítani. Ha egy lapra nem hivatkozunk a következő 8 millió, egy másikra pedig a következő 6 millió utasításban, akkor az előbbit eltávolítva a következő laphiba időpontját későbbre tolhatjuk ki. A számítógépek, akárcsak az emberek, megpróbálják a kellemetlen eseményeket minél későbbre halasztani.

Az egyetlen probléma ezzel az algoritmussal az, hogy nem lehet megvalósítani. Egy laphiba pillanatában az operációs rendszer nem tudja, hogy a következőkben milyen lapokra történik hivatkozás. (Korábban egy hasonló helyzetet láttunk az SJF ütemező algoritmusnál – honnan tudja a rendszer, hogy melyik a legrövidebb munka?) A program szimulátoron való *első* futtatásával nyomon követhetjük a hivatkozásokat, így a *második* futtatásnál az így kapott hivatkozási adatok alapján már működhet az optimális algoritmus.

Ezen a módon összehasonlítható az optimális értékkel a megvalósítható algoritmusok teljesítménye. Ha egy operációs rendszer például csak 1%-kal rosszabb az optimálisnál, akkor a jobb algoritmus keresése legfeljebb 1%-os javulást eredményezhet.

A további zűrzavar elkerülése érdekében világossá kell tenni, hogy a laphivatkozásoknak ez a naplózása csak az éppen vizsgált programról és csak egy specifikus esetről ad információt, az ebből származtatott lapcserélési algoritmusok csak erre a programra és az adott esetben bevitt adatokra lesznek jók. Bár ez a módszer jó a lapcserélési algoritmusok kiértékelésére, a gyakorlatban létező rendszereknél nem használják. Az alábbiakban olyan algoritmusokkal foglalkozunk, amelyek a *valódi* rendszerekben is használhatók.

4.4.2. Az NRU lapcserélési algoritmus

Abból a célból, hogy az operációs rendszer hasznos statisztikákat készíthessen a lapok használatáról, a virtuális memóriával rendelkező számítógépek többségében minden laphoz két állapotbit tartozik. Az *R* bit minden hivatkozáskor (olvasáskor és íráskor egyaránt) 1-re állítódik. Az *M* bit íráskor (azaz módosításnál) lesz 1. Ezek a bitek a 4.11. ábrán látható módon a laptáblabejegyzésben találhatók. Ezeket a biteket minden memóriahivatkozásnál karban kell tartani, ezért lényeges, hogy ezeket a beállításokat a hardver végezze el. Ha egy bit egyszer egyesre vált, akkor az is marad, amíg az operációs rendszer szoftveresen át nem állítja nullára.

Ha a hardvernek nincsenek ilyen bitjei, akkor ezt a következőképpen szimulálhatjuk szoftveresen. Ha egy processzus elindul, akkor minden laptáblabejegyzését megjelöljük, hogy nincs bent a memóriában. Amint egy lapra hivatkozunk, laphiba lép fel. Az operációs rendszer beállítja az *R* bitet, és megváltoztatja a laptábla bejegyzését, hogy az adott lapra mutasson *CSAK OLVASHATÓ* módban, majd a hibát okozó utasítástól újraindítja a processzust. Ha később a lapra írunk, akkor még egy laphiba lép fel, erre az operációs rendszer beállítja az *M* bitet, és átváltja a lap módját *ÍRHATÓ-OLVASHATÓ*-ra.

Az *R* és az *M* bitet a következő egyszerű lapcserélési algoritmusban használhatjuk fel: amikor egy processzus elindul, az operációs rendszer a processzus

minden lapjának mindkét bitjét nullára állítja. Időnként (például minden óramegszakításnál) nullázzuk az *R* bitjet, így megkülönböztetjük azokat a lapokat, amelyekre az utóbbi időben nem volt hivatkozás.

Laphiba esetén az operációs rendszer az *R* és *M* bitek alapján négy csoportba osztja a lapokat:

0. osztály: nem hivatkozott, nem módosított;
1. osztály: nem hivatkozott, módosított;
2. osztály: hivatkozott, nem módosított;
3. osztály: hivatkozott, módosított.

Első pillantásra az 1. osztály lehetetlennek tűnik, de mégis előfordulhat, ha a 3. osztály valamely elemének *R* bitje nullázódik az óramegszakítás során. Az óramegszakítás nem nullázhatja le az *M* bitet, mert ezt az információt használjuk fel annak eldöntésére, hogy a lap tartalmát vissza kell-e írni a lemezre. *R* törlése *M* törlése nélkül, 1-es osztályú lapot eredményez.

NRU (Not Recently Used – nem mostanában használt) algoritmus véletlenszerűen kiválaszt egy lapot a legkisebb sorszámú nemüres osztályból, és ezt a lapot dobja ki a memóriából. Az algoritmus azon a felismerésen alapszik, hogy jobb egy olyan módosított lapot kidobni, amelyet azt utolsó óraütemben (általában húsz ezred másodperc) nem használtak, mint egy tiszta lapot, amelyet gyakran használnak. Az NRU algoritmus fő vonzereje az egyszerűség, a közepesen hatékony implementálhatóság és az, hogy gyakran megfelelő, bár nem optimális a kapott teljesítmény.

4.4.3. A FIFO lapcserélési algoritmus

Egy másik egyszerű algoritmus a **FIFO (First-In, First-Out – elsőként be, elsőként ki).** Hogy megértsük, hogyan működik ez az algoritmus, képzeljünk el egy boltot, ahol pontosan *k* termék tárolására elegendő polc van. Egy nap egy cég bevezet egy új terméket. Mivel az új termék gyorsan sikeres lett, a boltnak meg kell szabadulnia egyik régi termékétől, hogy az újnak legyen helye.

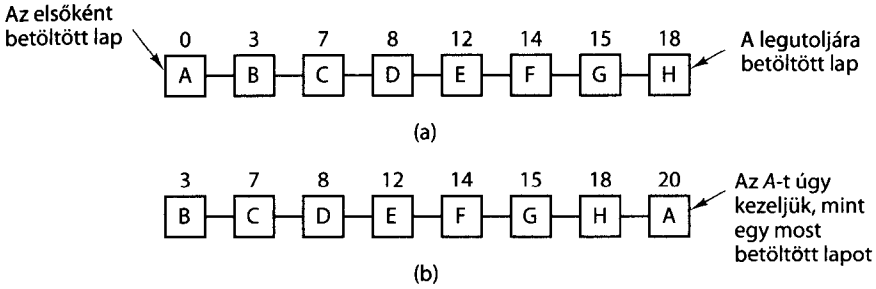
Az egyik lehetőség, hogy megkeresik azt a terméket, amelyet a legrégebben raktároznak (például 120 éve kezdték árulni), és ettől szabadulnak meg. A boltnak egy láncolt listát kell kezelnie, az új terméket a lista végére teszik, és a lista elejéről dobják el a régit.

Ezt az ötletet lapcserélési algoritmusként is használhatjuk. Az operációs rendszer egy listába fűzi a memóriában levő lapokat, a lista elején van a legrégebbi lap, a legutoljára beérkezett lap a lista végén található. Laphiba esetén az első lapot dobja ki, és a lista végére fűzi az új lapot. A bolti példánál maradva a FIFO kidobhatja a bajuszkenőcsöt, de ugyanúgy kidobhatja a lisztet, a sót és a vajat is. Számítógépeknél is előfordul ilyen probléma, emiatt a FIFO-t tiszta formájában ritkán alkalmazzák.

4.4.4. A második lehetőség lapcserélési algoritmus

A FIFO egyszerű módosítása a gyakran használt lapok kidobásának elkerülésére az, hogy megvizsgáljuk a legrégebbi lap *R* bitjét. Ha ez a bit nulla, akkor a lap régi és nem használt, így eldobható. Ha ez a bit egyes értékű, akkor töröljük a bitet, és a lapot a lista végére tesszük, a betöltési idejét pedig úgy módosítjuk, mintha most jött volna be a memóriába. Ezután folytatjuk a keresést.

Ezt az algoritmust **második lehetőségnek (second chance)** hívják, egy példa a 4.14. ábrán látható. A 4.14.(a) ábrán az *A–H* lapokat látjuk egy láncolt listában abban a sorrendben, ahogy a memóriába kerültek.



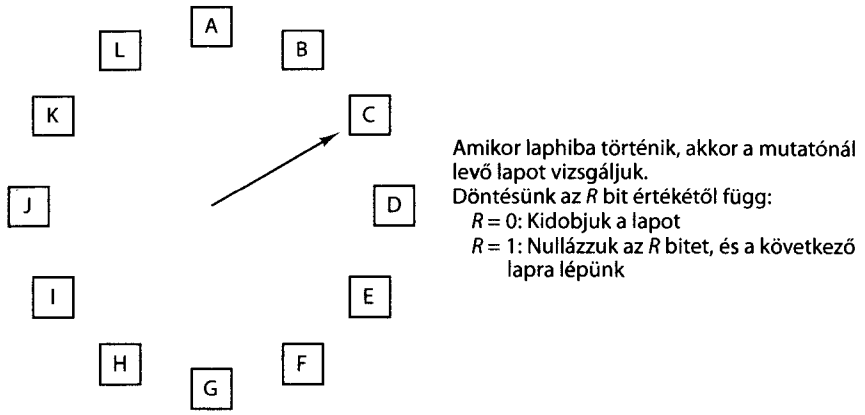
4.14. ábra. *A második lehetőség működése. (a) A lapok FIFO sorrendben rendezettek. (b) A laplista, ha a 20. időegységben laphiba történt, és az* A *lap* R *bitje 1. A lapok fölötti számok a betöltési időpontjukat mutatják*

Tegyük fel, hogy a 20. időegységben egy laphiba történt. A legrégebbi lap az *A*, amely a processzus indulásakor, a nullás időpontban került be a memóriába. Ha az *A* lap *R* bitje nulla, akkor ki lehet dobni a memóriából, az *M* bit értékétől függően esetleg vissza kell írni a lemezre. Másrészt, ha az *R* bit 1, akkor a lapot a lista végére kell tenni, *R* bitjét törölni, és betöltési idejét az aktuális időpontra (20) változtatni. A kidobandó lap keresése a *B* lapnál folytatódik.

Az algoritmus egy olyan régi lapot keres, amelyre nem volt hivatkozás az utolsó időintervallumban. Ha minden lapra hivatkoztak, akkor úgy működik, mint a FIFO. Például képzeljük el, hogy a 4.14.(a) ábrán látható lapok *R* bitje mind egyes. Az operációs rendszer egyenként átrakja a lista végére a lapokat, és nullára állítja a biteket. Amikor visszaér az *A* laphoz, a lap *R* bitje már nulla, így az *A* lapot dobja ki, ezért az algoritmus mindig befejeződik.

4.4.5. Az óra lapcserélési algoritmus

Bár a második lehetőség egy ésszerű algoritmus, kevésbé hatékony, mert állandóan mozgatja a lapokat körbe-körbe a listában. Hatékonyabb megoldás, ha a lapokat egy órához hasonló körkörös listában tároljuk, ahogy az a 4.15. ábrán látható. A mutató mindig a legrégebbi lapra mutat.



4.15. ábra. *Az óra lapcserélési algoritmus*

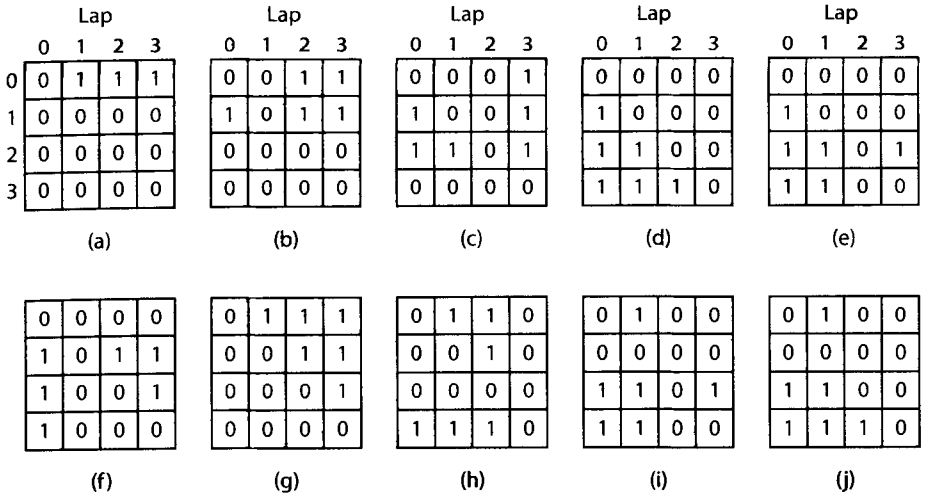
Amikor laphiba történik, akkor a mutatónál levő lapot vizsgáljuk. Ha az *R* bitje nulla, akkor ezt a lapot kidobjuk, a helyére rakjuk az újat, és eggyel továbbléptetjük a mutatót. Ha az *R* bit egy, akkor nullázzuk a bitet, és eggyel továbbléptetjük a mutatót. A keresés addig megy, amíg egy olyan lapot nem találunk, amelynek *R* bitje nulla. Ezt **óra (clock)** lapcserélési algoritmusnak hívják, és csak az implementációjában, nem a kiválasztott lapban különbözik a második algoritmustól.

4.4.6. Az LRU lapcserélési algoritmus

Az optimális algoritmus egyik jó közelítése azon a megfigyelésen alapszik, hogy az utolsó néhány utasításnál gyakran használt lapokra valószínűleg újra szükség lesz a következő néhány utasításban. Hasonlóan, a régen nem használt lapokat valószínűleg ezután sem használják. Ez az ötlet a következő algoritmust adja: amikor laphiba történik, akkor a legrégebben nem használt lapot dobjuk ki. Ezt a stratégiát **LRU (Least Recently Used – legrégebben használt)** lapcserélésnek nevezik.

Az LRU ugyan elméletileg megvalósítható, de nem túl olcsón. Az LRU implementálásához a memóriában levő lapokból egy láncolt listát kell készíteni, a lista elején van a legutoljára használt lap, a végén pedig a legrégebben használt lap. A nehézség az, hogy a listát minden memóriahivatkozásnál karban kell tartani. Megkeresni a lapot a listában, kivenni és a lista elejére fűzni még hardveresen is nagyon időigényes művelet (feltéve, ha építhetünk ilyen hardvert).

Az LRU speciális hardverrel történő implementálásának több más módja is van. Először nézzük a legegyszerűbb eljárást. Szükség van egy 64 bites számlálóra (c), amely minden memóriahivatkozásnál automatikusan eggyel nő. Továbbá minden laptáblabejegyzésben kell lennie egy mezőnek, amely ezt a számlálóértéket tárolja. Minden memóriahivatkozás után a *C* aktuális értéke beíródik a hivatkozott lap bejegyzésébe. Laphiba esetén az operációs rendszer megkeresi a legkisebb számlálóértékkel rendelkező lapot; ez lesz a legrégebben használt lap.



4.16. ábra. *Az LRU-bitmátrix használatával, amikor a lapokra az alábbi sorrendben hivatkozunk: 0, 1, 2, 3, 2, 1, 0, 3, 2, 3*

Most vizsgáljunk meg egy másik hardveres LRU-algoritmust. Egy n lapkeretes géphez a hardver egy $n \times n$-es bitmátrixot kezel, ami kezdetben csupa nulla. Amikor a k lapkeretre hivatkozunk, a hardver a mátrix k-adik sorát csupa egyesre, majd a k-adik oszlopot csupa nullára állítja. A legkisebb bináris értékű sor tartozik a legrégebben használt laphoz, a második legkisebb a második legrégebben használthoz, és így tovább. Az algoritmus működése a 4.16. ábrán látható, ahol négy lapkeret van, a hivatkozások rendre

0 1 2 3 2 1 0 3 2 3

A 0-s hivatkozás után kapjuk a 4.16.(a) ábrán látható állapotot. Miután az első lapra hivatkozunk, a 4.16.(b) ábrán látható szituációt kapjuk, és így tovább.

4.4.7. Az LRU szoftveres szimulációja

Bár mindkét előző LRU-algoritmus megvalósítható, kevés gép rendelkezik a szükséges hardvertámogatással. Ezért olyan megoldásra van szükség, amelyet szoftveresen lehet implementálni. Az egyik ilyen lehetőség az **NFU (Not Frequently Used – nem gyakran használt)** algoritmus. Ehhez minden laphoz szükség van egy szoftveres számlálóra, ami kezdetben nulla. Az operációs rendszer minden óramegszakításnál megvizsgálja a memóriában levő lapokat, és minden egyes lap R bitjét (amely 0 vagy 1 lehet) hozzáadja a számlálóhoz. Valójában a számláló azt adja meg, hogy milyen gyakran használták a lapot. Laphiba esetén a legkisebb számlálóértékű lapot dobja ki a memóriából.

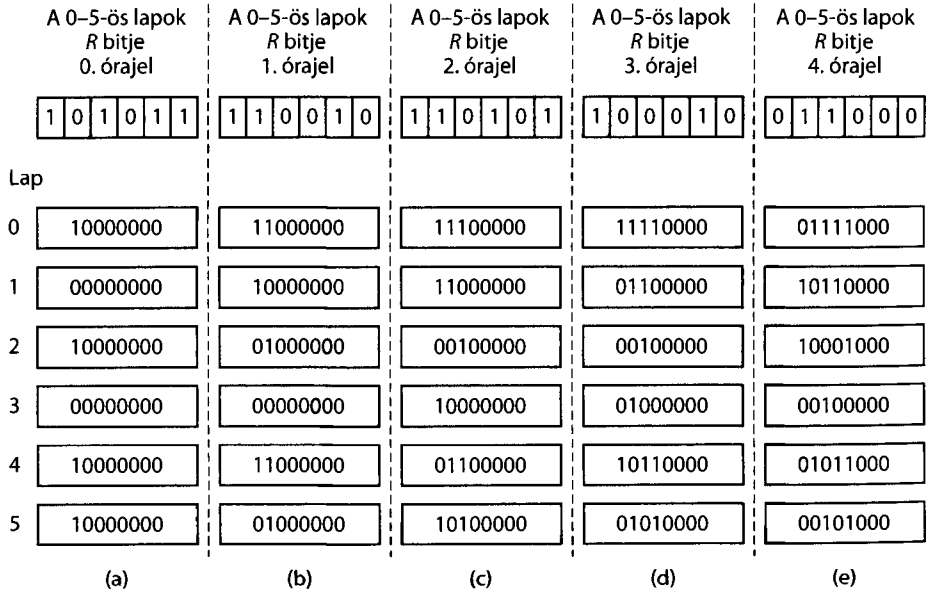
Az NFU-val az a fő probléma, hogy nem felejt el semmit. Például egy többmenetes fordítóprogram esetén az első menetben gyakran használt lapok a többi menetben is nagy számlálóértékkel rendelkeznek. Ha az első menet hosszabb, mint a többi, akkor a kódját tartalmazó lapoknak mindig nagyobb lesz a számlálója, mint a többi menet lapjainak. Így hát az operációs rendszer a hasznos lapokat fogja eltávolítani azon lapok helyett, amelyekre már nincs szükség.

Szerencsére az NFU kis módosítással jól szimulálja az LRU-t. A módosítás két részből áll. Először is a számlálót egy bittel jobbra toljuk, mielőtt az R-et hozzáadnánk. Másrészt az R-et nem a jobb oldali, hanem a bal oldali bithez adjuk hozzá.

A 4.17. ábra mutatja, hogyan működik az **öregítő (aging)** néven ismert, módosított algoritmus. Tegyük fel, hogy az első időegység végén a 0–5-ös lapok R bitje rendre 1, 0, 1, 0, 1 és 1. Más szóval az első időegységben a 0, 2, 4 és 5-ös lapokra hivatkoztunk, így R bitjük egyes lett, a többié nulla maradt. A hat számláló eltolása és R-rel való növelése után a 4.17.(a) ábrán látható állapotot kapjuk. A további négy oszlop a következő négy időegység után mutatja a hat számláló értékét.

Laphiba esetén a legkisebb számlálóértékű lapot dobjuk ki a memóriából. Világos, hogy a legutóbbi négy időegységben nem hivatkozott egyik lap számlálója négy darab nullás bittel kezdődik, így kisebb, mint egy olyan lapé, amelyre a legutóbbi három időegységben hivatkoztunk.

Ez az algoritmus két dologban különbözik az LRU-tól. Tekintsük a hármas és az ötös lapot a 4.17.(e) ábrán. Egyikre sem hivatkoztunk két időegység óta, de mindkettőt használtuk három időegységgel ezelőtt. Az LRU szerint ebből a kettőből kell kiválasztani egy lapot kidobásra. Az a baj, hogy nem tudjuk, a kettő közül



4.17. ábra. *Az öregítő algoritmus szoftveresen szimulálja az LRU-t. Tekintsük a fenti hat lapot öt időegységen át. Az öt időegységet az (a)–(e) ábra mutatja*

melyik lapot használtuk utoljára ebben az időegységben. Időegységenként egyetlen bit használatával nem tudjuk megkülönböztetni az időegységen belüli korábbi hivatkozásokat a későbbiektől. Így a hármas lapot dobjuk ki, mert nem volt rá hivatkozás, az ötöst viszont két időegységgel azelőtt is használtuk már.

A másik különbség az LRU-hoz képest az, hogy az öregítő algoritmus véges sok bites (a példánkban 8 bites) számlálót használ. Tegyük fel, hogy két lap számlálója egyaránt nulla, ezekből véletlenszerűen dobjuk ki az egyiket. Lehet, hogy az egyiket csak 9, a másikat pedig 1000 időegység óta nem használtuk, ezt nem tudjuk a számlálóból. A gyakorlatban egy 8 bites számláló általában elég egy 20 ezred másodperc körüli időegységhez. Ha egy lapra már 160 ezred másodperce nem hivatkoztunk, az már valószínűleg nem fontos a továbbiakban.

4.5. A lapozásos rendszerek tervezési szempontjai

Az előző alfejezetben elmagyaráztuk a lapozás működését, és megadtunk, modelleztünk néhány alapvető lapcserélési algoritmust. A puszta működés ismerete nem elég, egy rendszer tervezéséhez többet kell tudni, hogy jól működjön. A különbség olyan, mint a sakkban ismerni a figurák lépéseit vagy jól játszani. A következő alfejezetben néhány további szempontot vizsgálunk, amit a lapozásos rendszerek tervezőinek figyelembe kell venniük a jó teljesítmény érdekében.

4.5.1. A munkahalmaz modell

A lapozás legegyszerűbb eseténél egy processzus úgy indul, hogy még egy lapja sincs a memóriában. Amint a CPU a legelső utasítást próbálja olvasni, laphiba történik, erre az operációs rendszer behozza az első utasítást tartalmazó lapot. Gyorsan következik néhány további laphiba a verem és a globális változók miatt. Egy idő múlva a processzus legtöbb szükséges lapja a memóriában lesz, és a processzus viszonylag kevés laphibával fog futni. Ezt a stratégiát **igény szerinti lapozás**nak hívják, mert egy lap csak akkor töltődik be, amikor szükség van rá, előbb nem.

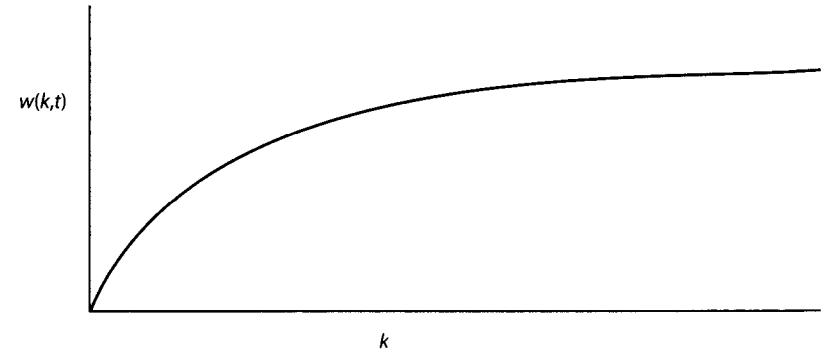
Természetesen könnyű egy olyan tesztprogramot írni, amely szisztematikusan olvassa a teljes címtér összes lapját, és olyan sok laphibát okoz, hogy nem lesz elég a memória a processzus rendes futásához. Szerencsére a legtöbb processzus nem így működik. **Lokális hivatkozás**nak nevezik, ha egy processzus a végrehajtásának egy fázisában csak a lapjainak egy kis részhalmazára hivatkozik. Például egy többmenetes fordító egy menetében csak néhány lapra hivatkozik, más menetekben pedig más lapokra. A hivatkozás lokalitásának koncepciója széles körben alkalmazható a számítástudományban, történetéről részletesebben olvashatunk (Denning, 2005).

A processzus által éppen használt lapok halmazát **munkahalmaz**nak nevezik (Denning, 1968a; 1980). Ha az egész munkahalmaz a memóriában van, akkor a processzus laphiba nélkül fut egész addig, amíg egy másik végrehajtási fázisba nem ér (például a fordító következő menete). Ha a memória túl kicsi ahhoz, hogy befogadja az egész munkahalmazt, a processzus sok laphibát okoz, és lassabban fut, mert amíg egy utasítás végrehajtása néhány nanomásodpercig tart, addig egy lap beolvasása a lemezről néhány 10 ezred másodperc is lehet. Egy vagy két utasítás végrehajtásával 10 ezred másodpercenként a folyamat évekig is eltarthat. Ha egy processzus gyakran okoz laphibát, azt **vergődés**nek nevezik (Denning, 1968b).

A multiprogramozást támogató rendszerekben a processzusokat gyakran kirakják a lemezre (azaz minden lapjukat eltávolítják a memóriából), hogy más processzusok is processzorhoz jussanak. A kérdés az, hogy mit kell tenni, amikor egy processzus újból végrehajtódik. Tulajdonképpen semmit sem kell tenni, a processzus egész addig laphibákat okoz, amíg az egész munkahalmaza be nem töltődik. A probléma az, hogy húsz, száz vagy akár ezer laphiba előfordulása, valahányszor egy processzus betöltődik, lassú és sok processzoridőt pazarol el, mivel az operációs rendszernek néhány ezred másodpercet el kell töltenie a laphibák feldolgozásával, a nagyszámú lemez I/O-műveletet nem is említve.

Ezért sok lapozásos rendszer megpróbálja nyomon követni a processzusok munkahalmazát, és a program indítása előtt betölti a szükséges lapokat. Ezt a megoldást **munkahalmaz modell**nek hívják (Denning, 1970). Ez a tervezési mód jelentősen csökkenti a laphibák számát. A programindítás *előtti* lapbetöltést **előlapozás**nak nevezik. Megjegyezzük, hogy a munkahalmaz idővel változik.

Régóta ismert, hogy legtöbb program hivatkozásai nemlineárisan oszlanak el a saját címterében. A hivatkozások többsége egy viszonylag kicsi laphalmazban csoportosul. A memóriahivatkozás betölthet egy utasítást vagy adatot, illetve elmenthet egy adatot. Minden t időpillanatban létezik egy olyan halmaz, amely minden, a k legújabb hivatkozásokat tartalmazó lapot tartalmazza. Ez a $w(k, t)$ halmaz a munkahalmaz. Mivel a nagy k érték távolabbi múltat reprezentál, a hozzá tartozó lapok száma nem csökkenhet k növelésével. Ennek következtében $w(k, t)$ monoton nem csökkenő függvénye k-nak. Mivel egy program nem hivatkozhat több lapra, mint amennyit a címtere tartalmaz, és kevés program használ minden egyes lapot, ezért $w(k, t)$ felső határa véges. A 4.18. ábra szemlélteti a munkahalmaz méretének és k-nak a viszonyát.



4.18. ábra. *A munkahalmaz a lapok egy halmaza, amelyekre a* k *legújabb memóriahivatkozás mutat. A* w(k, t) *függvény a munkahalmaz méretét mutatja be egy* t *időpillanatban*

Tény, hogy a legtöbb program véletlenszerűen használja a lapok egy kis halmazát, de ez a halmaz az idő előrehaladtával lassan változik. Ez magyarázza a görbe gyorsan emelkedő első részét és a nagyobb k értékekhez tartozó lassú emelkedését. Például egy program, amely egy két lapot elfoglaló, négy lapon található adatot manipuláló ciklust hajt végre, mind a hat lapot hivatkozhatja, mind az ezer utasításban, de egy más lap legújabb hivatkozása lehetett akár több millió utasítással is ez előtt az inicializáló részben. Emiatt az aszimptotikus viselkedés miatt a munkahalmaz tartalma nem függ k választott értékétől. Más szavakkal, létezik a k értékeknek egy széles tartománya, amelyre a munkahalmaz változatlan marad. Mivel a munkahalmaz az idő múlásával lassan változik, a legutóbbi leállításkor használt munkahalmaz alapján jó tippeket adhatunk a program újraindításához szükséges lapokra. Az előlapozás ezen lapok betöltését jelenti, mielőtt még a processzus újra elindulna.

A munkahalmaz modell megvalósításához szükséges, hogy az operációs rendszer nyilvántartsa, mely lapok tartoznak a munkahalmazba. Az egyik módszer ezen információk megszerzésére az előbb ismertetett öregítő algoritmus. Ha egy lap számlálójának felső n bitje tartalmaz egyest, akkor a lap a munkahalmaz eleme. Ha egy lapra az utolsó n időegységben nem hivatkoztak, azt kidobjuk a munkahalmazból. Az n paraméter értéke kísérleti úton állapítható meg, de a rendszer teljesítménye nem különösen érzékeny a pontos értékre.

A munkahalmazt fel lehet használni az óra algoritmus teljesítményének növelésére is. Eredetileg, ha a mutató egy nullás R bittel rendelkező lapra mutatott, akkor ezt a lapot dobtuk ki. A javítás az, hogy megvizsgáljuk, hogy a lap eleme-e az aktív processzus munkahalmazának. Ha eleme, akkor a lapot nem dobjuk ki, hanem folytatjuk a kidobandó lap keresését. Ezt az algoritmust **wsclock**nak (munkahalmaz óra algoritmus) hívják.

4.5.2. Lokális vagy globális helyfoglalás

Az előzőkben számos algoritmust vizsgáltunk a laphiba esetén kidobandó lap kiválasztására. A választással kapcsolatos fő szempont (amit eddig a szőnyeg alá söpörtünk) a memória megosztása a párhuzamosan futó processzusok között.

Tekintsük a 4.19.(a) ábrát. Három futtatható processzusunk van: A, B és C. Tegyük fel, hogy az A processzus laphibát okozott. Vajon a lapcserélési algoritmusnak az A-hoz tartozó hat lap vagy a memóriában levő összes lap közül kellene kiválasztania a legrégebben használt lapot? Ha csak az A lapjai közül választunk, akkor a kiszemelt lap az $A5$ lesz, és a 4.19.(b) ábrán látható állapotot kapjuk.

Másrészt, ha az összes lap közül választunk, tekintet nélkül arra, hogy a lap kihez tartozik, akkor a $B3$ lapot kapjuk, és a 4.19.(c) ábrán látható helyzetbe jutunk. Az előbbit **lokális**, az utóbbit **globális** lapcserélési algoritmusnak nevezzük. A lokális algoritmusok a memóriának egy rögzített méretű szeletét foglalják le egy-egy processzus részére. A globális algoritmusok dinamikusan osztják el a memória lapkereteit a futó processzusok között, ilyenkor az egy processzushoz tartozó lapkeretek száma időben változik.

	Kor
A0	10
A1	7
A2	5
A3	4
A4	6
A5	3
B0	9
B1	4
B2	6
B3	2
B4	5
B5	6
B6	12
C1	3
C2	5
C3	6

(a)

A0
A1
A2
A3
A4
A6
B0
B1
B2
B3
B4
B5
B6
C1
C2
C3

(b)

A0
A1
A2
A3
A4
A5
B0
B1
B2
A6
B4
B5
B6
C1
C2
C3

(c)

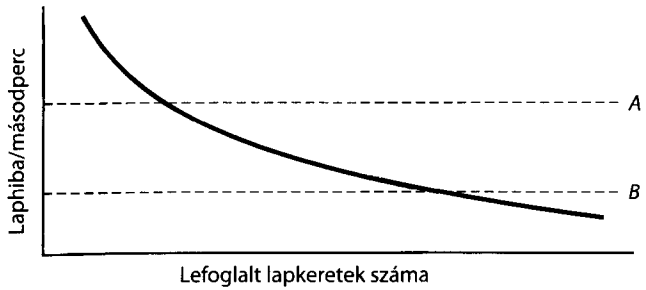
4.19. ábra. *A lokális és globális lapcsere összehasonlítása. (a) Eredeti állapot. (b) Lokális lapcsere. (c) Globális lapcsere*

Általában a globális algoritmusok működnek jobban, mert a munkahalmaz mérete a program futása során változik. Ha lokális algoritmust használunk, és a munkahalmaz nő, akkor vergődés lesz az eredmény, hacsak nincsen sok szabad lapkeretünk. Ha a munkahalmaz csökken, akkor a lokális algoritmusok memóriát pazarolnak el. Ha globális algoritmust használunk, akkor a rendszernek mindig döntenie kell, hogy mennyi lapkeretet adjon egy-egy processzusnak. Az egyik mód erre, hogy az öregedés bitekkel figyeljük a munkahalmaz méretét, de ez a módszer nem előzi meg a vergődést. A munkahalmaz mérete milliomod másodpercek alatt változhat, de az öregedés bitek értéke egy durva mérték, sok időegység kell a beállásához.

A másik megközelítés az, ha egy megfelelő algoritmus rendeli a lapkereteket a processzusokhoz. Az egyik módszer, hogy periodikusan meghatározzuk a futó processzusok számát, és egyenlő arányban elosztjuk közöttük a lapkereteket. Például ha 12 416 rendelkezésre álló (azaz nem az operációs rendszerhez tartozó) lapkeretünk és 10 processzusunk van, akkor minden processzus 1241 lapkeretet kap. A fennmaradó 5 lapkeretet tartaléknak használjuk laphibák esetére.

Bár ez a módszer jónak látszik, azonos mennyiségű memóriát ad egy 10 KB-os és egy 300 KB-os processzusnak ahelyett, hogy a processzusméretek arányában osztaná el a memóriát, azaz a 300 KB-os processzus harmincszor akkora helyet kapna, mint a 10 KB-os. Bölcs dolog lenne minden processzushoz megadni egy minimumszámot, amivel még futni tud. Néhány gépen egy egyszerű kétoperandusú utasítás végrehajtásához akár hat lapra is szükség lehet, mert az utasításkód, a forrás- és a céloperandus mind-mind eshet laphatárra. Ha csak öt lapot foglalunk le, akkor az ilyen utasításokat tartalmazó programokat nem lehet lefuttatni.

Amennyiben egy globális algoritmust használunk, minden processzust elláthatunk a méretével arányos számú lappal, de a foglalást a processzusok futása során



4.20. ábra. *Laphibaarány a lefoglalt lapkeretek számának függvényében*

dinamikusan frissíteni kell. A közvetlen megoldások egyike a **laphiba-gyakoriság (Page Fault Frequency, PFF)** algoritmus. Ez az algoritmus megmondja, hogy növeljük, vagy csökkentsük a processzuslap foglalását, de arról semmit sem mond, hogy melyik lapot kell kicserélni hiba esetén, csak a lefoglalt laphalmaz méretét szabályozza.

A lapcserélési algoritmusok jó részére (az LRU-ra is) igaz, hogy a laphibák aránya csökken, ha több lapkeretet foglalunk le. Ezt a tulajdonságot a 4.20. ábra illusztrálja.

A laphiba-gyakoriság, amelyen a PFF algoritmus alapszik, egyszerűen mérhető: számolnunk kell a hibák számát másodpercenként, esetleg célszerű egy átlagot számítani az eltelt másodpercekre. Egy egyszerű megoldásként hozzáadhatjuk a jelen hiba gyakoriságot az előzőhöz, és eloszthatjuk kettővel. Az *A* jelű szaggatott vonal mutatja az elfogadhatatlanul magas laphibaarányt, ilyenkor a processzusnak több lapkeretet kell adni, hogy csökkenjen a laphibák száma. A *B* jelű vonal mutatja azt az alacsony értéket, amiből az következik, hogy a processzusnak túl sok lapkerete van. Ebben az esetben a processzustól el lehet venni lapkereteket. Ilyen módon a PFF algoritmus a laphibák számát az elfogadható határok között igyekszik tartani.

Ha úgy találjuk, hogy olyan sok processzus van a memóriában, hogy lehetetlen mindet az *A* szint alatt tartani, akkor néhány processzust eltávolíthatunk a memóriából, és lapkereteiket szétoszthatjuk a többi processzus között, vagy felhasználhatjuk tartalékként a további laphibák kezelésére. A döntés, hogy melyik processzust vigyük ki, a **terheléselosztás** egy formája. Azt mutatja, hogy a lapozás mellett a csere is szükséges, csak most a cserét a lehetséges memóriaigény csökkentésére használjuk, és nem a blokkok visszavételére. Az, hogy a memóriaterhelés csökkentésére processzusokat mozgatunk ki, egy kétszintű ütemezésre emlékeztet, amelynek keretében néhány processzust a merevlemezre mozgatunk, és egy rövid idejű ütemező ütemezi a megmaradó processzusokat. A két ötlet kombinálható azzal, hogy csak annyi processzust mozgatunk ki lemezre, hogy a laphibaarány elfogadható legyen.

4.5.3. Lapméret

A lapméret gyakran választható értékű paraméter. Ha a hardver lapmérete 512 bájt, akkor az operációs rendszer könnyen tekintheti a 0 és 1, 2 és 3 stb. lapokat 1 KB-osaknak, mindig két egymás melletti 512 bájtos lapkeretet lefoglalva a számukra.

Az optimális lapméret meghatározásához számos különböző tényezőt kell egyensúlyba hoznunk. Ennek következtében nincs minden tényezőre vonatkozó optimum. Kezdésként: nézzük azt a két tényezőt, amelyek a kicsi lapméretet indokolják. A véletlenszerűen kiválasztott kód-, adat- és veremszegmensek nem foglalnak el egész számú lapot. Az utolsó lapnak átlagosan a fele üres marad, a lapnak ezt a részét elvesztegetjük. Ezt a veszteséget **belső töredezettség**nek hívják. Ha n darab szegmens van a memóriában, és a lapméret p bájt, akkor átlagosan $np/2$ bájtot vesztünk el ilyen módon. Ebből az következik, hogy a lapméretet érdemes kicsire választani.

Egy másik érv a kis lapméret mellett az, hogy ha van egy program nyolc egymás utáni 4 KB-os menettel, akkor 32 KB-os lapméret esetén mindig 32 KB-ot foglal le, 16 KB esetén mindig 16 KB-ot, 4 KB-os vagy kisebb lapméretnél pedig csak 4 KB-ot. Általában nagy lapméretnél több kihasználatlan rész van a memóriában, mint kis lapméretnél.

Másrészt kis lapméret esetén a program több lapból áll, így nagyobb laptábla szükséges. Egy 32 KB-os programnak csak négy darab 8 KB-os, viszont 64 darab 512 bájtos lap kell. Az átvitel a lemezre és vissza laponként történik; ez leginkább a lemez keresési és a felpörgetési idejétől függ, így egy nagy lap átvitele majdnem ugyanannyi ideig tart, mint egy kis lapé. Például a 64 512 bájtos lap betöltése 64×10 ms, míg a négy darab 8 KB-os lapé $4 \times 10{,}1$ ms.

Néhány gépnél a laptáblát külön regiszterekbe kell tölteni, amikor a központi egység átkapcsol az egyik processzusról a másikra. Az ilyen gépeken kisebb lapméretet használva, a laptábla betöltése annyiszor hosszabb ideig tart, mint amennyiszer a lapméret kisebb. Sőt a laptábla által elfoglalt hely úgy nő, ahogy a lapméret csökken.

Az utolsó pontot matematikailag is elemezhetjük. Legyen az átlagos processzusméret s, a lapméret p bájt. Továbbá legyen egy laptáblabejegyzés mérete e bájt. Egy processzushoz körülbelül s/p lap kell; ez a laptáblában se/p bájtot foglal el. A belső töredezettség miatt az utolsó lapon elvesztett memória átlagosan $p/2$ bájt. Így a teljes költség a laptábla és a belső töredezettség miatt a következő kifejezéssel jellemezhető:

$$\text{költség} = se/p + p/2$$

Az első tag (a laptábla mérete) akkor nagy, ha a lapméret kicsi. A második tag (belső töredezettség) akkor nagy, ha a lapméret is nagy. Az optimum valahol középen van. A p szerinti első deriváltat véve, az optimumot az alábbi egyenletből számoljuk:

$$-se/p^2 + 1/2 = 0$$

Ebből a formulából azt kapjuk, hogy az optimális lapméret:

$$p = \sqrt{2se}$$

Például s = 1 MB és e = 8 bájt esetén az optimális lapméret 4 KB. A kereskedelemben kapható számítógépek 512 bájt és 1 MB közötti lapméretet használnak. Egy tipikus érték az 1 KB, de manapság a 4 és a 8 KB a legelterjedtebb. A memória növekedtével a lapméret is növekszik, ez a növekedés azonban nemlineáris. Négyszer nagyobb memória ritkán duplázza meg a lapméretet.

4.5.4. Virtuális memória interfész

Mostanáig úgy kezeltük a virtuális memóriát, hogy átlátszó a processzusok és a programozók számára, azaz csak egy nagy virtuális címteret látnak a kisebb fizikai memóriával rendelkező számítógépen. Sok rendszerben ez igaz is, de néhány fejlettebb rendszerben a programozó vezérelheti a memóriahasználatot, és a program javítása érdekében, a hagyományostól eltérő módon is használhatja a memóriát. Ezekről lesz szó röviden ebben az alfejezetben.

Az egyik ok, hogy a memória kezelését a programozó kezébe adjuk, az, hogy megoszthasson egy memóriarészt kettő vagy több processzus között. Ha a programozó elnevezheti a memóriarészeket, akkor a processzus átadhatja ezt a nevet egy másik processzusnak, így a másik processzus is el tudja érni ezt a memóriarészt. Ha kettő vagy több processzus osztozik ugyanazon a lapon, akkor nagy sávszélességű kapcsolat jöhet létre, mert amit az egyik processzus a közös memóriába ír, azt a másik elolvashatja.

A közös lapok a nagy teljesítményű üzenetküldő rendszerek implementálásánál használhatók fel. Normális esetben, ha egy üzenetet küldünk, az jelentékeny költséggel átmásolódik az egyik processzus címteréből a másikba. Ha a processzusok vezérelhetik a laptérképüket, akkor a küldő processzus kiszedi a lapjai közül az üzenetet tartalmazó lapot (lapokat), a fogadó processzus pedig belapozza azt. Ilyenkor az adatok helyett csak a lap nevét másoljuk át.

Egy újabb fejlett memóriakezelő technika az **elosztott közös memória** (Feeley et al., 1995; Li és Hudak, 1989; Zekauskas et al., 1994). Az ötlet az, hogy hálózatban futó processzusok között megosszunk lapokat, lehetőleg (de nem feltétlenül) egyetlen közös lineáris címtérrel. Ha egy processzus egy olyan lapra hivatkozik, amely nincs belapozva, akkor laphibát okoz. A laphibakezelő, amely egyaránt lehet a kernel- vagy a felhasználótérben, megkeresi a lapot tároló gépet, küld egy üzenetet, hogy a kért lapot lapozzák ki, és küldjék el a hálózaton. Amikor a lap megérkezik, belapozza, és újraindítja a hibát okozó utasítást.

4.6. Szegmentálás

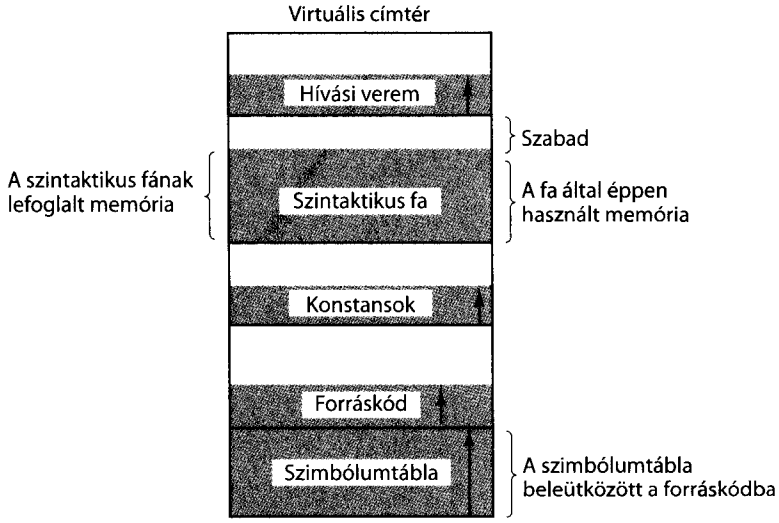
Eddig a virtuális memóriát egydimenziós tömbként tárgyaltuk, mert a virtuális címek nullától egy maximális címig terjedtek. Sok problémánál azonban jobb kettő vagy több különálló virtuális címteret használni, mint egyetlenegyet. Például egy fordítóprogram több táblázatot is felépít a fordítás alatt, például az alábbiakat:

1. A forráskód a lista nyomtatásához (kötegelt rendszerekben).
2. A szimbólumtábla a változók nevével és attribútumaival.
3. Az egész és lebegőpontos konstansok táblája.
4. A program szintaktikai fája.
5. A fordítóprogram saját verme.

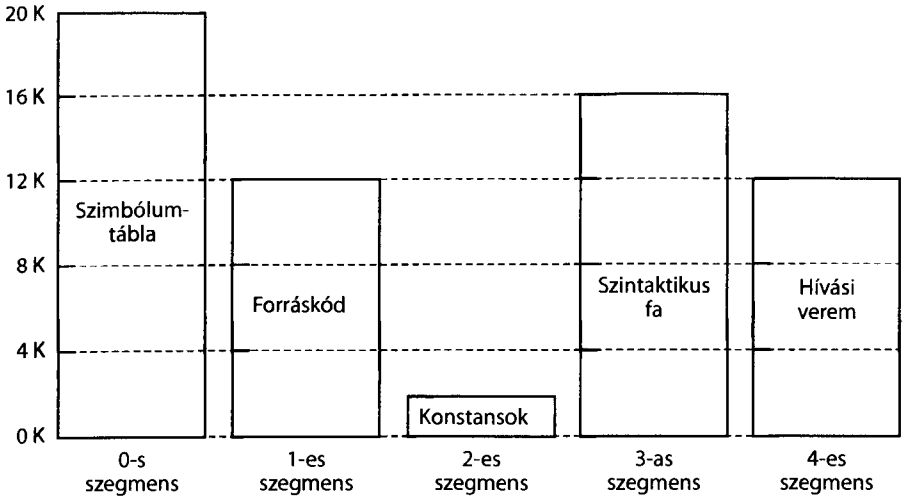
Az első négy tábla a fordítás alatt állandóan növekszik, az utolsó előre nem látható módon nő és csökken. Egy egydimenziós memóriában a 4.21. ábrán látható módon öt darab nagy, egybefüggő részre kell osztani a virtuális címteret.

Mi történik, ha egy egyébként minden tekintetben hagyományos forrásprogramban túl sok változó van? A szimbólumtáblának lefoglalt memóriaszelet betelik, és beleütközik a következő tábla területébe. A fordító egy üzenetet küld, hogy túl sok változó van, és abbahagyja a fordítást, pedig lehet, hogy a többi táblánál még maradt kihasználatlan memória.

A másik lehetőség az, hogy a fordító Robin Hoodot játszik, helyet vesz el azoktól a tábláktól, amelyeknek sok szabad helyük van, és a helyszűkében levő tábláknak adja. Ez a módszer működik, de hasonló ahhoz, ha valaki saját maga kezeli programrétegeit, legjobb esetben csak kellemetlenség, legrosszabb esetben hosszú és unalmas munka.



4.21. ábra. *Egydimenziós címtér növekvő táblákkal, az egyik tábla beleütközhet a másikba*



4.22. ábra. *A szegmentált memória lehetővé teszi, hogy mindegyik tábla a többitől függetlenül növekedjen vagy csökkenjen*

Mi lehet a valódi módja annak, hogy a programozót megszabadítsuk a táblák memóriakezelésének gondjától, hasonlóan ahhoz, ahogy a virtuális memória felmentette a programozót a program rétegekre szabdalása alól?

A legáltalánosabb megoldás az, hogy lehetővé tesszük több, egymástól teljesen független címtér létrehozását. A címtereket **szegmens**eknek hívják. Minden szegmens egy lineáris címtér nullától valamilyen maximális címig. Egy szegmens hosszúsága nulla és egy meghatározott maximum közötti tetszőleges szám lehet. A különböző szegmensek általában eltérő hosszúak, sőt a szegmens mérete változhat a program végrehajtása során. Például a veremszegmens mérete megnő, ha valamit belerakunk a verembe, és csökken, ha kiveszünk onnan valamit.

Mivel mindegyik szegmens külön címteret alkot, a szegmensek egymástól függetlenül nőhetnek vagy csökkenhetnek. Ha például a veremnek több hely kell, akkor megnőhet, mert nincs semmi más a címterében, aminek nekiütközne. Természetesen egy szegmens megtelhet, de a szegmensméret általában olyan nagy, hogy ez az eset ritkán következik be. A szegmentált vagy kétdimenziós memóriában a címeknek két részből kell állniuk, a szegmens számából és a szegmensen belüli címből. A 4.22. ábra az előbb látott fordítóprogram tábláit mutatja szegmentált memóriában. Az ábrán öt független szegmens látható.

Kiemeljük, hogy a szegmens a legegyszerűbb formájában egy logikai egység, a programozó is így használja. Egy szegmens egy vagy több eljárást, tömböt, vermet vagy skalárváltozókat tartalmazhat, de általában nem keverik össze egy szegmensbe a különböző típusokat.

A szegmentált memóriának más előnye is van amellett, hogy a változó méretű adatszerkezetek egyszerűen kezelhetők. Ha minden eljárás egy külön szegmens nullás címén van, akkor a program összeszerkesztése egészen egyszerű. Az *n.*

szegmensben levő eljárás meghívásánál az (*n*, 0) címet kell használni; a 0 az eljárás belépési pontja.

Ha az *n.* szegmensben levő eljárást módosítjuk és újrafordítjuk, akkor a többi eljárást nem kell megváltoztatni, mert a belépési címek nem változtak. Egydimenziós memóriában az eljárások szorosan egymás mögé vannak pakolva. Így ha egy eljárás mérete megváltozik, az eltolja az azt követő eljárások belépési pontját. Emiatt az összes olyan eljárást módosítani kell, amely egy elmozdult eljárást hívott, hogy az új címet használja. Ha egy programban több száz eljárás van, akkor ez a művelet költséges lehet.

A szegmentálás lehetővé teszi adatok vagy eljárások megosztását processzusok között. Ennek leggyakoribb példája az **osztott könyvtár**. Az ablakozó rendszerrel működő modern munkaállomásoknál majdnem minden programhoz hatalmas grafikus könyvtárat kellene hozzáfordítani. Szegmentálás esetén a grafikus könyvtárakat a processzusok között megosztott közös szegmensbe lehetne tenni és megosztani több processzus között, így ezeket a közös könyvtárakat nem kellene minden processzus címterének külön-külön tartalmaznia. Ha egy tiszta lapozásos rendszerben akarunk osztott könyvtárat használni, akkor bonyolultabb dolgunk van, az ilyen rendszerek valójában a szegmentálás szimulálásával teszik ezt.

Mivel a szegmens egy logikai egység, így az eljárásoknak és az adatoknak különböző védelmi szintjük lehet. Az eljárásokat tartalmazó szegmens csak végrehajtható, nem lehet olvasni vagy módosítani. Egy lebegőpontos tömböt tartalmazó szegmens csak olvasható és írható, de nem végrehajtható, így hibát okoz a kísérlet, hogy átadjuk rá a vezérlést. Ez a védelem segít megtalálni a programhibákat.

A szegmentált memória teszi lehetővé a védelmet, de a védelemnek nincs értelme az egydimenziós memória esetében. A szegmentált memóriánál a programozó

Szempont	Lapozás	Szegmentálás
Tudnia kell-e a programozónak, hogy ezt a technikát használja?	Nem	Igen
Hány lineáris címtér használható?	1	Sok
Lehet-e a teljes címtér nagyobb, mint a fizikai memória mérete?	Igen	Igen
Meg lehet-e különböztetni és külön-külön védeni az adatokat és az eljárásokat?	Nem	Igen
Egyszerű-e a változó méretű táblázatok elhelyezése?	Nem	Igen
Segíti-e az eljárások felhasználók közötti megosztását?	Nem	Igen
Miért fejlesztették ki ezt a technikát?	Nagy lineáris címteret kapjunk több fizikai memória vásárlása nélkül	Lehetővé teszi, hogy a programot és az adatokat logikai egységekre vágjuk szét, és segíti ezek védelmét és megosztását

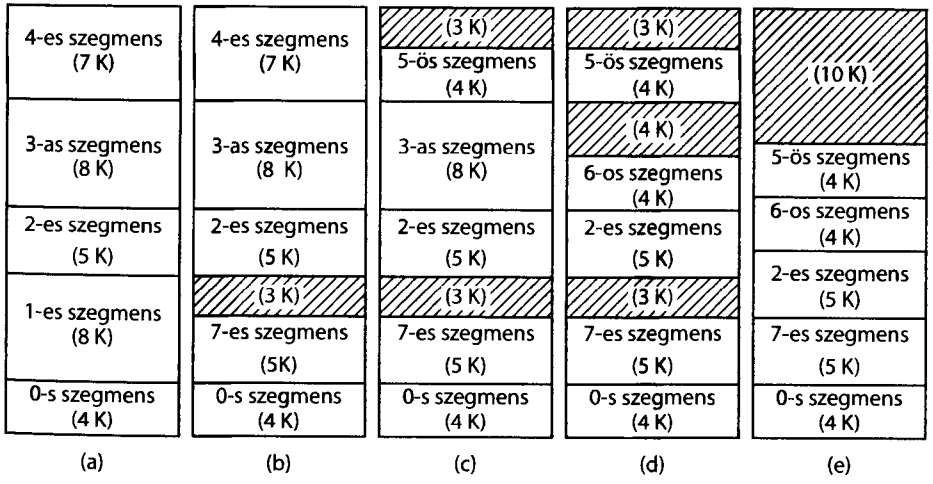
4.23. ábra. *A lapozás és a szegmentálás összehasonlítása*

tudja, hogy mi van egy szegmensben. Mivel minden szegmens csak egyféle dolgot tartalmaz, a szegmens erre alkalmas védelmi kódot használhat. A lapozást és a szegmentálást a 4.23. ábrán hasonlítjuk össze.

Egy lap tartalma általában véletlenszerű. A programozónak nem kell tudnia, hogy a programja lapozással fut. Bár néhány bit hozzáadásával a laptáblabejegyzésekhez meghatározhatunk védelmi szinteket, de ahhoz, hogy a programozó ezt használni tudja, nyomon kell követnie, hogy hol vannak a laphatárok a processzus címterében. Mivel a szegmentált memória felhasználója úgy érzi, hogy mindig a központi memóriában van minden szegmens (vagyis úgy címezheti azokat, mintha ott lennének), minden szegmenst külön védhet anélkül, hogy az adminisztrációval kelljen foglalkoznia.

4.6.1. A tiszta szegmentálás implementációja

A szegmentálás implementációja egy lényeges dologban különbözik a lapozástól: a lapok fix méretűek, a szegmensek nem. A 4.24.(a) ábra egy olyan fizikai memóriát mutat, amelyben kezdetben öt szegmens van. Nézzük meg, hogy mi történik, ha kivisszük az egyes szegmenst, és helyébe a hetes számú, kisebb szegmenst hozzuk be. Az eredmény a 4.24.(b) ábrán látható memóriakép lesz. A hetes és a kettes szegmens között van egy kihasználatlan terület; ez egy lyuk. Ezután a négyes szegmenst az ötössel helyettesítjük a 4.24.(c) ábrán látható módon, majd a hármas helyett a hatost hozzuk be, ahogy az a 4.24.(d) ábrán látható. Ha a rendszer tovább fut, akkor a memória részekre darabolódik, egyes részek szegmenst, mások lyukat tartalmaznak, a kis lyukak összesen sok memóriát vesztegetnek el. Ezt a jelenséget **külső töredezettségnek** (fragmentáció) hívják. A 4.24.(e) ábrán látható módon tömörítéssel javítható.



4.24. ábra. *(a)–(d) A külső töredezettség kialakulása. (e) A töredezettség megszüntetése tömörítéssel*

4.6.2. Szegmentálás lapozással: Intel Pentium

A Pentium processzor legfeljebb 16 000, egyenként 2^{32} bájtos virtuális címterű szegmenst támogat. A processzor beállítható (az operációs rendszer által) csak szegmentáció, csak lapozás vagy mindkettő használatára. A legtöbb operációs rendszer a Windows XP-t és a legtöbb Unixot is beleértve, az egyszerű lapozásos modellt használja; ez esetben minden processzusnak egy 2^{32} bájtos virtuális címterű szegmense lehet. Mivel a Pentium processzor képes sokkal nagyobb címterek kezelésére is, és van egy operációs rendszer (OS/2), amely kihasználja a címzés minden képességét, ezért a következőkben nagy vonalakban megismerkedünk a Pentium processzorok virtuális memóriakezelésével.

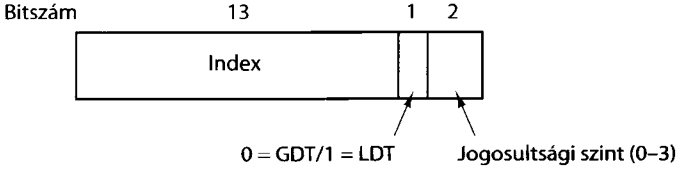
A Pentium virtuális memóriájának szíve két táblázatból áll: a **lokális leírótábla (Local Descriptor Table, LDT)** és a **globális leírótábla (Global Descriptor Table, GDT)**. Minden programnak saját LDT-je van, de GDT csak egy van, ezt közösen használja a gépen futó összes program. Az LDT tartalmazza a program saját lokális szegmenseit (kód, verem, adat stb.), a GDT-ben pedig a rendszerszegmensek találhatók, beleértve az operációs rendszer szegmenseit is.

Egy szegmens eléréséhez a Pentium először betölti a szegmens szelektorát a gép hat szegmensregiszterének egyikébe. A futás során a CS-regiszter a kód-, a DS az adatszegmens szelektorát tartalmazza. A többi szegmensregiszter kevésbé fontos. A szelektor egy 16 bites szám, felépítése a 4.25. ábrán látható.

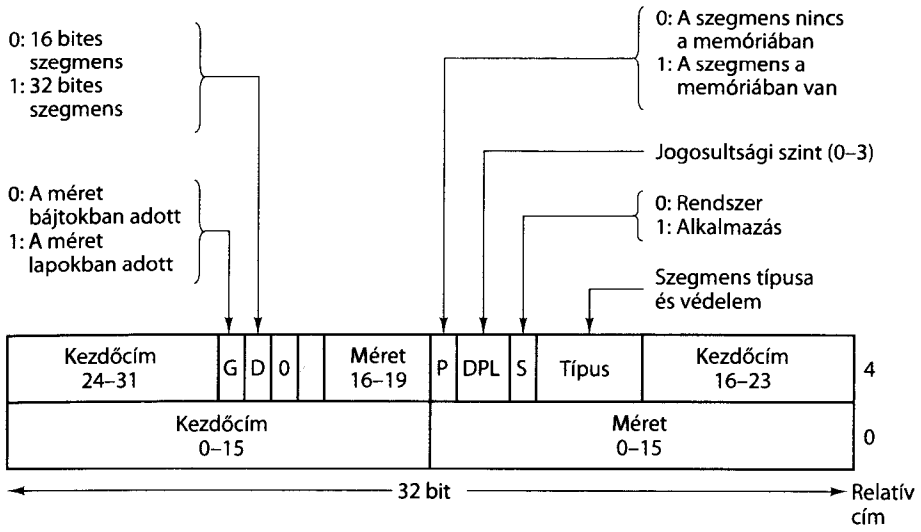
Egy bit azt mutatja, hogy a szegmens lokális vagy globális (azaz az LDT-ben vagy a GDT-ben van). 13 bit adja meg a GDT- vagy az LDT-beli bejegyzés sorszámát, így ezek a táblák egyenként 8192 darab szegmensleírót tartalmazhatnak. A fennmaradó 2 bit a védelmi kód, ezt a későbbiekben pontosabban leírjuk. A nullás leíró tiltott. A szegmensregiszterbe töltésével biztonságosan jelezhetjük, hogy a szegmensregiszter jelenleg nem érhető el. Amennyiben használjuk, akkor megszakítást okoz.

Amikor egy szelektort a szegmensregiszterbe töltünk, akkor a hozzá tartozó leíró az LDT-ből vagy a GDT-ből a mikroprogram-regiszterekbe töltődik, így gyorsabban elérhető lesz. A 4.26. ábrán látható leíró 8 bájtos, tartalmazza a szegmens báziscímét, hosszát és egyéb információkat.

A szelektorból egyszerűen meghatározhatjuk a deszkriptor helyét. Először a szelektor második bitje alapján kiválasztjuk az LDT-t vagy a GDT-t. Ezután a szelektort egy belső regiszterbe másoljuk, és alsó három bitjét nullázzuk. Végül hozzáadjuk az LDT vagy GDT tábla címét, így megkapjuk a leíró címét. Például a 72-es szelektor a GDT 9-es bejegyzésére mutat, ami a GDT + 72-es címen található.



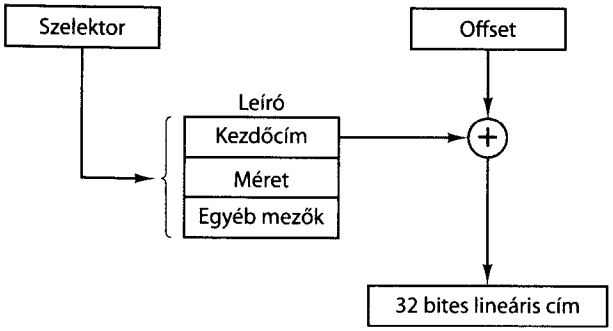
4.25. ábra. *A Pentium szelektora*



4.26. ábra. *A Pentium kódszegmensének leírója. Az adatszegmens kissé különbözik ettől*

Most nézzük meg, hogyan konvertálódik egy (szelektor, offset) páros fizikai címmé. Mivel a mikroprogram tudja, hogy melyek a használt szegmensregiszterek, a hozzájuk tartozó leírókat a belső regiszterekben tárolhatja. Ha a szegmens nem létezik (szelektora 0), vagy ki van lapozva, akkor megszakítást okoz.

Ezután ellenőrzi, hogy az offset a szegmensen belül van-e, ha nem, akkor megszakítást okoz. Logikailag 32 bit kellene a szegmens méretének megadásához, de a leíróban erre a célra csak 20 bit van, így más eljárást használnak. Ha a leíró *gbit*-je (szemcsézettség) nulla, akkor a *határmező (limit)* a szegmens pontos méretét adja meg (legfeljebb egy megabájt). Ha ez a bit 1, akkor a *határmező* bájtok helyett lapokban tartalmazza a szegmens méretét. A Pentium lapmérete 4 KB, így a 20 bites méret 2^{32} bites szegmensekhez elég.



4.27. ábra. *A (szelektor, offset) pár átalakítása lineáris címmé*

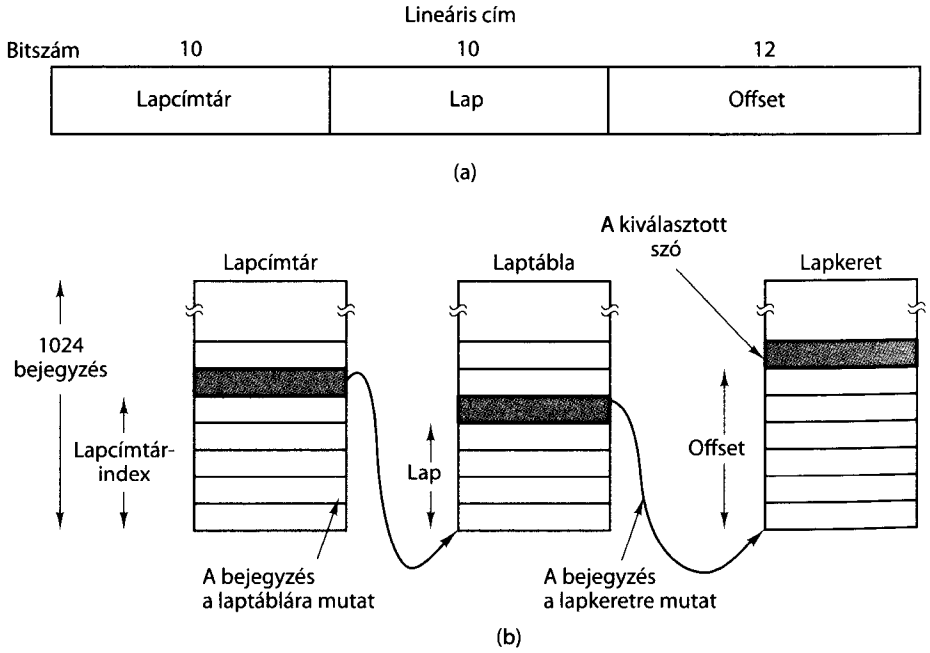
Ha a szegmens a memóriában van, és az offset sem túl nagy, akkor a leíró 32 bites *bázismezőjét* hozzáadjuk az offsethez, így megkapjuk a **lineáris címet** (lásd 4.27. ábra). A *bázismező* a leíróban szétszórt három részből áll, hogy kompatibilis maradjon a 286-ossal, ahol a bázis csak 24 bites volt. A *bázismező (base)* lehetővé teszi, hogy egy szegmens a 32 bites lineáris címtér tetszőleges címén kezdődhessen.

Ha a lapozás tiltott (a globális vezérlőregiszter egy bitjével), akkor a lineáris cím maga a fizikai cím, és változatlanul továbbítódik a memória felé. Így lapozás nélkül a tiszta szegmentálással állunk szemben, ahol a szegmensek kezdőcíme a leírójukban található. A szegmensek átfedhetik egymást, mert túl sok időbe telne a diszjunktságot ellenőrizni.

Ha a lapozás engedélyezett, akkor a lineáris címet virtuális címként értelmezzük, és a laptábla segítségével képezzük le a fizikai címre, ahogy azt a korábbi példákban láttuk. Az egyetlen gond, hogy a 32 bites virtuális címek és a 4 KB-os lapméret miatt egy szegmens egymillió lapot is tartalmazhat, így kétszintű laptáblát kell használni, hogy csökkentsük a kis szegmensek laptáblájának méretét.

Minden futó programhoz tartozik egy **lapcímtár**, amely 1024 darab 32 bites bejegyzést tartalmaz; erre a címtárra egy globális regiszter mutat. Minden bejegyzés egy 1024 32 bites elemet tartalmazó laptáblára mutat. Egy laptáblabejegyzés egy lapkeretre mutat. Ezt a sémát láthatjuk a 4.28. ábrán.

A 4.28.(a) ábrán látható lineáris cím három mezőre van osztva: *dir* (lapcímtárindex), *page* (lap) és *off* (offset). A *dir* mezőt használjuk a laptábla címének kikeresésére a lapcímtárból. A laptáblából a *page* mező értéke alapján választjuk ki a



4.28. ábra. *A lineáris cím leképezése fizikai címre*

lapkeret fizikai címét. Végül az *offset* értékét a lapkeret fizikai címéhez hozzáadva megkapjuk a kért fizikai címet.

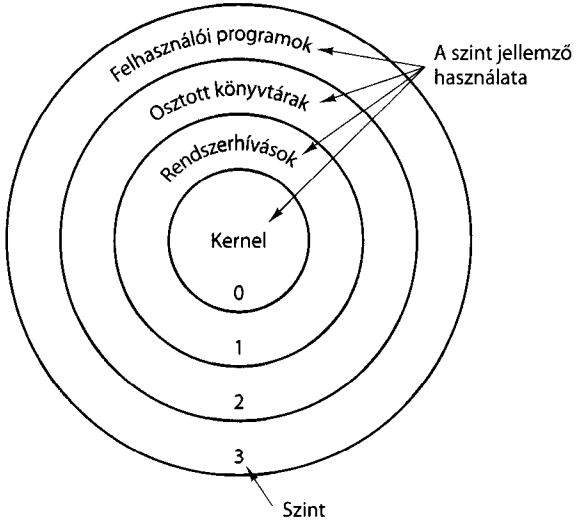
A laptábla egy bejegyzése 32 bites, ebből 20 bit tartalmazza a lapkeret számát. A fennmaradó bitek az elérési, védelmi és egyéb bitek.

Minden laptábla 1024 bejegyzést tartalmaz, amelyek 4 KB-os lapokra mutatnak, így egy laptábla 4 MB memóriát tud kezelni. A 4 MB-nál kisebb szegmensek lapcímtárában csak egyetlen bejegyzés van, amely a szegmens egyetlen laptáblájára mutat. Ilyen módon a kis szegmensek többletköltsége csak két lap, szemben az egyszintű laptábla egymillió lapjával.

Az ismételt memóriahivatkozások megelőzésére a Pentium egy kicsi TLB-t használ a leggyakoribb *dir–page* kombinációk lapkeretre való leképezésére. Ha egy kombináció nincs benne a TLB-ben, akkor a 4.28. ábrán látható módon megkeressük a hozzá tartozó lapkeret számát, és berakjuk a TLB-be. Amíg a TLB-hibák ritkák, a teljesítmény jó.

Ha lapozást használunk, akkor a leíróban a *bázismező* értéke célszerűen mindig nulla. Ugyanis ha nem nulla, akkor egy kis offsetet okoz, ami miatt a lapcímtár egy középső bejegyzését fogjuk használni az első bejegyzés helyett. A *bázismező* létének valódi oka az, hogy lehetővé tegye a tiszta (nem lapozott) szegmentálást, és a kompatibilitást a 286-os rendszerrel, ahol nem volt lapozás (azaz csak tiszta szegmentálás volt).

Ez a modell lehetővé teszi, hogy olyan alkalmazások is fussanak, amelyeknek nincs szükségük szegmensekre, csak egy darab 32 bites, lapozott, virtuális címtérre. Minden szegmensregiszterbe ugyanazt a szelektort kell tölteni, a hozzá tartozó leíróban a *bázis* értéke nulla, a *limit* pedig a maximum. Az offset maga a lineáris cím, mintha csak sima lapozást használnánk. Valójában manapság min-



4.29. ábra. *A Pentium védelmi szintjei*

den Pentium processzoron futó operációs rendszer így működik. Az OS/2 volt az egyetlen, amely kihasználta az Intel MMU architektúrájának minden lehetőségét.

Mindent egybevéve elismerés jár a Pentium tervezőinek. Az ellentétes célok ellenére (tiszta lapozás, szegmentálás és lapozott szegmentálás implementálása, valamint kompatibilitás a 286-ossal) a megoldás hatékony, meglepően egyszerű és világos.

Miután ismertettük a Pentium virtuális memóriájának felépítését, megéri néhány szót szólni a védelemről is, mert ez a téma szorosan kapcsolódik a virtuális memóriához. A Pentium a 4.29. ábrán látható módon négy védelmi szintet támogat, a nulladik szinttől, amely a legerősebb, a harmadik szintig, amely a leggyengébb. Minden egyes futó program példánynak van egy szintszáma; ez a PSW-ben két biten van kódolva. A szegmensekhez szintén tartozik egy-egy szintszám.

Ha a program csak a saját szintjén levő szegmenseket használja, akkor minden simán megy. A magasabb szintű szegmensekről megengedett az adatok elérése. Alacsonyabb szintű szegmenseken levő adatot tilos használni, az ilyen kísérlet megszakítást okoz. Más szinteken (alacsonyabb és magasabb egyaránt) levő eljárások meghívása megengedett, de csak ellenőrzött módon. Egy szintközök közötti hívásnál a CALL utasításnak a cím helyett egy szelektort kell tartalmaznia. Ez a szelektor egy **hívási kapu** nevű leírót jelöl ki, ami a hívott eljárás címét adja meg. Így nem lehet egy másik szinten levő tetszőleges kódszegmens közepébe ugrani, csak a hivatalos belépési pontok használhatók.

A 4.29. ábrán ennek a módszernek egy tipikus felhasználása látható. A nullás szinten fut az operációs rendszer kernele, amely az I/O-t, a memóriát és egyéb kritikus dolgokat kezeli. Az egyes szinten van a rendszerhívás-kezelő. A felhasználói programok meghívhatják az itt levő eljárásokat, hogy rendszerhívásokat hajtsanak végre, de az eljárásoknak csak egy meghatározott és védett halmaza használható. A második szint tartalmazza a könyvtári eljárásokat, esetleg megosztva több futó program között. A felhasználói programok meghívhatják ezeket az eljárásokat, olvashatják az adatokat, de nem módosíthatják azokat. Végül a felhasználói programok a legalacsonyabb védettségű harmadik szinten futnak.

A megszakítások kezelése a hívási kapukhoz hasonló mechanizmust használ. Abszolút címek helyett leírókra hivatkozik; ezek a leírók mutatnak a végrehajtandó eljárásokra. A 4.26. ábrán látható *type* mező tesz különbséget a kód- és adatszegmensek, valamint a különféle kapuk között.

4.7. A MINIX 3 processzuskezelője

A MINIX 3 memóriakezelése rendkívül egyszerű, nem használ lapozást, és ahogy itt tárgyaljuk, cserélgetést sem használ. Amennyiben a MINIX 3-nak kicsi memóriával rendelkező gépen kell futnia, a forráskódban megtalálható cserét bármikor aktiválhatjuk. A gyakorlatban azonban a memóriák ma már olyan nagy méretűek, hogy a cserélgetésre ritkán van szükség.

Ebben a fejezetben egy felhasználói szintű szervert, a **processzuskezelőt (process manager, PM)** fogjuk röviden áttekinteni. A processzuskezelő a processzuskezeléssel kapcsolatos rendszerhívásokat fogadja. Ezek közül több közvetlenül is kapcsolódik a memóriakezeléshez. A fork, exec és brk hívások tartoznak ebbe a kategóriába. A processzuskezelő ezek mellett a különböző szignálokhoz tartozó rendszerhívásokat kezeli, beállítja a processzus tulajdonságait, mint például felhasználó és csoporttagság, illetve a CPU használati idő jelentése. A MINIX 3 processzuskezelőjének feladata a valós idejű óra lekérdezése és beállítása is.

Időnként, amikor a processzuskezelőnek a memóriakezeléssel foglalkozó részére hivatkozunk, „memóriakezelő"-nek fogjuk nevezni. Elképzelhető, hogy egy jövőbeli kiadásban a processzuskezelő és a memóriakezelő teljesen elkülönül, de jelenleg a MINIX 3-ban ezeket a funkciókat egyetlen processzus kezeli.

A PM feladata többek között egy cím szerint rendezett lyuklista karbantartása. Amennyiben például egy fork vagy exec rendszerhívásnál szükség van szabad memóriaterületre, akkor a PM megkeresi a lyuklistában az első megfelelő méretű lyukat. Mivel nincs csere, ha egy processzus bekerül a memóriába, egészen a befejeződéséig ugyanazon a helyen marad. A MINIX 3 soha sem viszi ki lemezre, vagy helyezi át más memóriaterületre a processzusokat, és a processzusnak lefoglalt terület méretét sem változtatja meg.

Ez a memóriakezelési stratégia egy kicsit szokatlan, így némi magyarázatot igényel. Három indokot tudunk felsorolni mellette:

1. A rendszer egyszerűsége, érthetősége így biztosítható.
2. Az eredeti IBM PC CPU-architektúrája miatt (egy Intel 8088).
3. Egyszerű legyen a MINIX 3-at átírni más hardverre.

Először is, oktatási célú mintarendszerként a komplexitást kerülni kellett: egy 250 oldalas forráskódot elég hosszúnak ítéltünk. A lapozás megvalósítása lehetetlen volt, mivel a rendszer arra az eredeti IBM PC-re lett tervezve, amelynek még MMU-ja sem volt. Az akkori számítógépek többsége sem rendelkezett MMU-val, így ez a memóriakezelő stratégia lehetővé tette a szoftver egyszerűbb átírását Macintosh-, Atari-, Amiga-platformokra.

Ma már azonban megkérdőjelezhetjük a megoldás létjogosultságát. Annak ellenére, hogy a rendszer sokat nőtt az évek alatt, az első pont még mindig igaz. De mára más szempontok is fontos szerepet kaphatnak. A modern PC-k 1000-szer több memóriával rendelkeznek, mint az eredeti IBM PC. Annak ellenére, hogy a programok is nagyobbak, a legtöbb rendszer annyi memóriával rendelkezik, hogy ritkán van szükség cserére és lapozásra. Végezetül a MINIX 3 célplatformjai az alacsony szintű eszközökkel bővültek. Manapság operációs rendszerrel rendelkeznek a digitális kamerák, DVD-lejátszók, mobiltelefonok és más eszközök. A MINIX 3, mint operációs rendszer, ésszerű választás ebben a világban, így a lapozás és a csere megvalósítása nem létfontosságú. A háttérben folyik bizonyos munka a lehető legegyszerűbb virtuális memória kezelési megoldásának feltérképezésére. Az ezzel kapcsolatos friss információk megtalálhatók a weboldalon.

Érdemes hangsúlyozni, hogy más operációs rendszerekkel ellentétben, a MINIX 3 memóriakezelése nem része a kernelnek. Felhasználói szintű processzusként fut, és a kernellel a szabványos üzenetváltási megoldással kommunikál. A 2.29. ábrán látható a PM helyzete.

A PM kernelen kívüli implementálása jó példa az **elv** és a **megvalósítás** szétválasztására. A memóriakezelő dönti el, hogy melyik processzus hova kerüljön a memóriában (elv). A memóriatérképet a kernel állítja be (megvalósítás). A szétválasztás miatt könnyű megváltoztatni a memóriakezelő elvet (algoritmust stb.) anélkül, hogy az operációs rendszer alsóbb rétegeit módosítani kellene.

A PM kódjának nagyobb része a MINIX 3-rendszerhívásainak kezelésével (elsősorban a fork és exec) foglalkozik, és csak a kisebb rész foglalkozik a processzus- és lyuklista manipulálásával. A következő alfejezetben megismerjük a memória szerkezetét, azt követően pedig megnézzük madártávlatból, hogyan dolgozza fel a PM a processzuskezelő rendszerhívásokat.

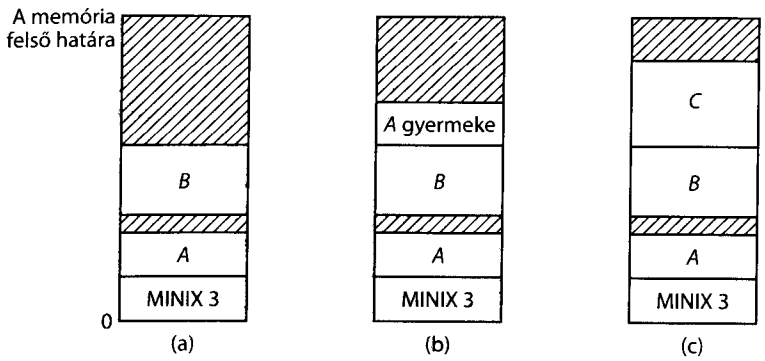
4.7.1. A memória szerkezete

A MINIX 3-programok lefordításánál eldönthetjük, hogy használjanak-e olyan **kombinált kód- és adatrészt (combined I and S space),*** amelynél a program minden memóriaterülete (kód, adat, verem) a memória egy blokkján osztozik és együtt van lefoglalva, illetve felszabadítva, vagy pedig **szeparált kód- és adatrészt (separate I and D space)** használjanak. Az eredeti MINIX-verziónál a kombinált kód- és adatrész volt az alapértelmezett, a MINIX 3-ban azonban a szeparált. Az érthetőség kedvéért először az egyszerűbb közös modellt vitatjuk meg. A szeparált kód- és adatrészt használó processzusok memóriahasználata sokkal hatékonyabb, de ezen megoldások sokkal komplikáltabbá teszik a rendszert. A bonyodalmakkal az egyszerű esetek megvizsgálása után fogunk foglalkozni.

A MINIX 3-ban normál működés esetén két alkalommal kell a memóriát lefoglalni. Egyrészt, amikor egy processzus szétágazik (fork), ekkor memóriát kell foglalni a gyermekprocesszusnak. Másrészt, amikor egy processzus az exec-kel elindít maga helyett egy másik processzust, ekkor fel kell szabadítani a processzusnak kiutalt memóriát, és ezt a lyuklistába be kell illeszteni, valamint le kell foglalni az új processzusnak szükséges részt. Az új processzus nem feltétlenül az éppen felszabadított memóriarészbe kerül, ez attól függ, hol talál erre alkalmas lyukat. Memóriát akkor is fel kell felszabadítani, ha egy processzus kilépéssel befejeződik vagy leállítják egy szignállal. Létezik egy harmadik eset is: egy rendszertaszk igényelhet memóriát saját használatra is, például egy memóriameghajtó memóriát igényel a RAM-lemezhez. Ez csak a rendszer inicializálásakor fordulhat elő.

A 4.30. ábra mutatja a memóriafoglalás két módját fork és exec esetén. Az (a) esetben két processzusunk van a memóriában, az *A* és a *B*. Ha az *A* szétágazik (fork), akkor a (b) ábrán látható állapotba jutunk. A gyermekprocesszus az *A* pon-

* Az *I and D space* utasítás- és adatrészként is használatos *(A szerk.)*



4.30. ábra. *Memóriafoglalás. (a) Eredeti állapot. (b) Az* A *fork-ja után. (c) Miután a gyermekprocesszus egy* exec*-et hajtott végre. A vonalkázott terület a szabad memória. A processzusnak kombinált kód- és adatrésze van*

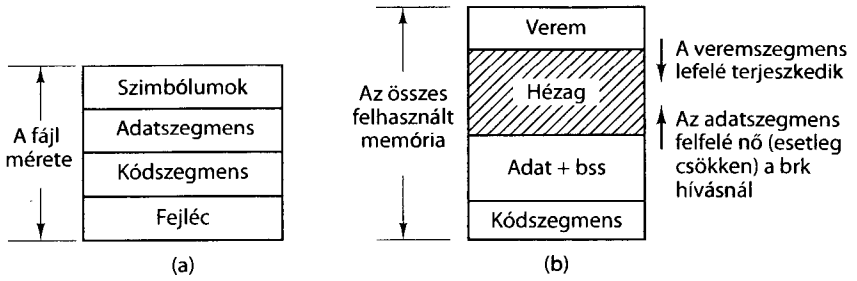
tos másolata. Ha a gyermek elindítja (exec) a *C* fájlt, akkor a memória a (c) állapotba kerül. A gyermekprocesszus képe a *C*-vel helyettesítődik.

Megjegyzendő, hogy a gyermeknek lefoglalt régi memóriaterület még azelőtt felszabadul, hogy a *C*-nek helyet foglalnánk, így a *C* használhatja a gyermek memóriaterületét. Ezen a módon feltételezve azt, hogy nagy lefoglalatlan memóriablokk létezik, a fork és exec párosok sorozatával a processzusok szomszédosak lehetnek, nem lesz lyuk közöttük, ellentétben azzal, ha az új memóriát a régi felszabadítása előtt foglaljuk le. Amennyiben az új memória még a régi felszabadítása előtt lenne lefoglalva, lyukak maradnának.

A megoldás nem triviális. Példaként tegyük fel, hogy elfogyott a memória az exec futtatására. A szabad memória mennyiségét még az előtt kell megállapítani, mielőtt a gyermek memóriáját felszabadítanánk, így a gyermek valamilyen módon jelezhetné a hibát. Ez azt jelenti, hogy a gyermek memóriaterületét lyukként kell kezelni, annak ellenére, hogy az még használatban van.

A fork vagy az exec rendszerhívásnál memóriát kell foglalni az új processzusnak. Az első esetben pontosan ugyanannyit, mint a szülőprocesszus mérete. A második esetben a PM a végrehajtandó fájl fejlécéből határozza meg a processzus méretét. Miután ez a memóriafoglalás lezajlott, a processzus semmilyen módon sem tud magának több memóriát szerezni.

Az eddig elmondottak olyan programokra vonatkoztak, amelyek kombinált kód- és adatrésszel lettek fordítva. Az olyan programok, amelyeknek szeparált kód- és adatrésze van, élvezhetik ennek az előnyeit, egy fejlettebb memóriakezelési módot, az **osztott kód**ot használva. Amikor egy ilyen processzus végrehajt egy fork-ot, akkor elég csak az adat- és veremterület másolatának helyet foglalni, a szülő kódját a gyermekprocesszus is használhatja. Amikor egy processzust exec-kel indítunk el, akkor az operációs rendszer átnézi a processzustáblát, hogy egy másik processzus használja-e már a kért kódot. Ha igen, akkor elég csak az adatnak és a veremnek helyet foglalni, és a memóriában levő kód közösen használható. A közös kód bonyolítja a processzusok befejeződését. Ha egy processzus befejeződik, akkor az adat- és veremterületét mindig fel kell szabadítani. A kódjának lefoglalt területet csak akkor szabadíthatja fel, ha a processzustáblába segítségével meggyőződött arról, hogy azt nem használja más processzus. Elfordulhat az, hogy amennyiben egy processzus kódrészét más processzus is használatba vette, akkor a processzus indításakor több memóriát foglaltunk le, mint amennyit a befejeződésekor felszabadítottunk.



4.31. ábra. *(a) A program, ahogy a lemezen egy állományban tárolódik. (b) Egy egyszerű processzus belső memóriaképe. Mindkét ábrán az alsó lemez- vagy memóriacímek alul, a felső címek felül vannak*

A 4.31. ábrán láthatjuk, hogyan van tárolva a program a lemezen lévő fájlban, és hogyan néz ki a memóriába töltés után MINIX 3-processzusként. A lemezen lévő fájl fejléce tartalmazza a programkép különböző részeinek és az egésznek a méretét. Az inicializálatlan statikus változók tárolására az operációs rendszer megnöveli a kép adatrészének méretét a fejléc *bss* mezőjében megadott értékkel, a hozzáadott részt pedig nullákkal tölti fel. A lefoglalandó memória teljes méretét a *total* mező tartalmazza. Ha például egy program kódrésze 4 KB-os, adatrésze a *bss*-sel együtt 2 KB, verme 1 KB-os, és a program fejlécében összesen 40 KB memória lefoglalása szerepel, akkor 33 KB kihasználatlan memória lesz az adat- és a veremszegmens között. A lemezen levő programfájl tartalmazhatja a szimbólumtáblát is; ez a nyomkövetéshez kell, és nem töltjük be a memóriába.

Ha a programozó tudja, hogy az *a.out* nevű program adat- és veremszegmensének együttes növekedése legfeljebb 10 KB, akkor a fájl fejlécében levő adat megváltoztatására kiadhatja a

```
chmem =10240 a.out
```

parancsot, amely megváltoztatja a fejléc mezőt annak érdekében, hogy az exec hívásakor a PM a kód- és adatszegmens méreténél 10 240 bájttal többet foglal le. A fenti példánál maradva, így minden egyes exec híváskor összesen 16 KB-ot foglalunk le. Ebből a felső 1 KB a verem, 9 KB a hézag, ahova a verem- és az adatszegmens nyúlhat, ha nagyobbra nő.

Egy olyan program, amely szeparált kód- és adatrészt használ (ezt a szerkesztő által a fejlécben beállított bit jelzi), a *total* mező csak az adat- és veremszegmens együttes méretét tartalmazza. Egy olyan program részére, amelynek a kódrésze

4 KB, adatrésze 2 KB, verme 1 KB és teljes mérete 64 KB, 68 KB-ot foglal le a rendszer (4 KB utasítás és 64 KB adat és verem), ebből 61 KB szolgál az adat és a verem futás közbeni növekedésére. Az adatszegmens határát csak a brk rendszerhívással lehet módosítani. A brk ellenőrzi, hogy az adatszegmens új mérete ütközik-e az aktuális veremmutatóval, ha nem, akkor elvégzi a változtatást. Ez teljes egészében a processzusnak eredetileg lefoglalt memóriaterületen belül zajlik, az operációs rendszer nem foglal le újabb memóriát. Ha az adatszegmens beleütközik a verembe, akkor a rendszerhívás meghiúsul.

Megemlítünk egy jelentésbeli különbséget. Ha a „szegmens" szót használjuk, akkor az operációs rendszer által definiált memóriaterületre gondolunk. Az Intel processzoroknak belső **szegmensregiszterei** és **szegmensleíró-táblái** vannak, amelyek a szegmentálás hardvertámogatásáról gondoskodnak. Az Intel hardvertervezőinek szegmensfogalma hasonló, de nem mindenben azonos a MINIX 3-ban használt szegmenssel. Ebben a szövegben a szegmens szó a MINIX 3 adatstruktúrái által definiált memóriaterületként értelmezendő. Amikor a hardverről beszélünk, akkor határozottan a szegmensregiszterekre és a szegmensleírókra hivatkozunk.

Ezt a figyelmeztetést általánosíthatjuk. A hardvertervezők gyakran próbálnak támogatást adni a gépen futó operációs rendszernek, a regiszterek és a processzor egyéb elemeinek elnevezése gyakran visszatükrözi, hogy mire használható az adott dolog. Ezek a jellemzők gyakran hasznosak az operációs rendszerek íróinak, de nem mindig ugyanúgy, ahogy a hardvertervezők előre gondolták. Ez zavarhoz vezethet, mert ugyanannak a szónak kétféle értelme van az operációs rendszer és az alatta levő hardver szempontjából.

4.7.2. Üzenetkezelés

Ahogy a MINIX 3 többi komponense, a processzuskezelő is üzenetvezérelt. A rendszer inicializálása után a PM belép a főciklusába, amelyben üzenetekre várakozik, teljesíti az üzenetben levő kéréseket, és elküldi a válaszokat.

A processzuskezelő által kapott üzenetek két kategóriába tartozhatnak. A magas prioritású kommunikációra a kernel és az olyan rendszerszerverek között, mint amilyen a PM is, a **rendszerértesítő üzenetek**et használják. Ezeket a speciális eseteket a fejezet implementációról szóló részében tárgyaljuk. A PM-hez beérkező üzenetek többsége a felhasználói processzusoktól származik. A 4.32. ábra mutatja a lehetséges üzenettípusokat inputparamétereikkel és a válaszban visszaadott értékeikkel együtt.

A fork, az exit, a wait, a waitpid, a brk és az exec szorosan kötődik a memória lefoglalásához és felszabadításához. A kill, az alarm és a pause rendszerhívások, valamint a sigaction, a sigsuspend, a sigpending, a sigmask és a sigreturn szignálokhoz kapcsolódnak. Ezek szintén hatással vannak a memóriára, mert ha egy processzust megszüntetünk, akkor a hozzá tartozó memória felszabadul. A hét get/set rendszerhívás nem foglalkozik a memóriakezeléssel, viszont szorosan kapcsolódik a processzusok kezeléséhez. Minden más rendszerhívást a fájlrendszer vagy a PM kezel. Azért kerültek ide, mert a fájlrendszer már túl bonyolult volt.

A time, stime és times hívások, valamint a ptrace, amely a nyomkövetésnél hasznos, ugyanilyen okból van itt.

A reboot az operációs rendszer minden részében használatos, de elsődleges dolga egy szignál segítségével ellenőrzött módon leállítani a processzusokat, így a PM jó hely ennek kezelésére. Ugyanez igaz a svrctl-re is, amelynek a legfontosabb feladata a csere engedélyezése vagy tiltása a PM-ben.

Üzenettípus	Bemenő paraméterek	A válasz
fork	(nincs)	Gyermek azonosítója (Gyermek felé: 0)
exit	Kilépési állapot	(Nincs válasz, ha sikeres)
wait	(nincs)	Állapot
waitpid	Processzusazonosító és opciók	Állapot
brk	Új méret	Új méret
exec	A kezdeti verem címe	(Nincs válasz, ha sikeres)
kill	Processzusazonosító és a szignál	Állapot
alarm	Várakozás ideje (másodperc)	Hátralévő idő
pause	(nincs)	(Nincs válasz, ha sikeres)
sigaction	Szignál száma, új akció, régi akció	Állapot
sigsuspend	Szignálmaszk	(Nincs válasz, ha sikeres)
sigpending	(nincs)	Állapot
sigprocmask	Mód, új és régi szignálhalmaz	Állapot
sigreturn	Környezet	Állapot
getuid	(nincs)	Valódi és tényleges UID
getgid	(nincs)	Valódi és tényleges GID
getpid	(nincs)	Processzus és szülő azonosítója
setuid	Új felhasználói azonosító	Állapot
setgid	Új csoportazonosító	Állapot
setsid	Új azonosító	Processzuscsoport azonosítója
getpgrp	Új azonosító	Processzuscsoport azonosítója
time	Mutató a helyre, ahova az aktuális idő megy	Állapot
stime	Mutató aktuális időre	Állapot
times	Mutató egy pufferre, a processzus és gyermekórák számára	Az eltelt idő indulás óta
ptrace	Kérés, processzusazonosító, cím, adat	Állapot
reboot	Mód (leáll, újraindít, pánik)	(Nincs válasz, ha sikeres)
svrctl	Kérés, adat (a függvénytől függ)	Állapot
getsysinfo	Kérés, adat (a függvénytől függ)	Állapot
getprocnr	(nincs)	Processzusszám
memalloc	Méret, mutató a címhez	Állapot
memfree	Méret, cím	Állapot
getpriority	Pid, típus, érték	Prioritás
setpriority	Pid, típus, érték	Prioritás
gettimeofday	(nincs)	Idő, eltelt idő az indítás óta

4.32. ábra. *A processzuskezelővel való kommunikációra szolgáló üzenetek, bemenő paramétereik és válaszértékük*

Felfigyelhetünk arra, hogy a két utolsónak tárgyalt hívás, a reboot és a svrctl, nem található meg az 1.9. ábrán. Ez igaz a 4.32. ábra még nem említett getsysinfo, getprocnr, memalloc, memfree és getsetpriority hívásaira is. Ezek egyike sem része a POSIX szabványnak és nem a hagyományos felhasználói processzusok kiszolgálása a feladatuk. A MINIX 3-rendszer számára lettek létrehozva. Egy monolitikus kernellel rendelkező rendszerben ezek a műveletek a kernelbe fordított függvényhívásokként lennének megvalósítva. A MINIX 3-ban azonban az operációs rendszer részeit képező komponensek a felhasználótérben futnak, és ezért további rendszerhívásokra van szükség. Egyesek nem sokkal tesznek többet annál, minthogy egy egyszerű interfészt nyújtanak a kernelhívások felé. Kernelhívásoknak nevezzük azokat a hívásokat, amelyek kernelszolgáltatásokat vesznek igénybe a rendszertaszkon keresztül.

Mint már az első fejezetben említettük, bár van sbrk könyvtári függvény, nincs sbrk rendszerhívás. A könyvtári függvény kiszámolja a szükséges memóriát a paraméterként megkapott növelő és csökkentő tényezők figyelembevételével, és ezután végrehajt egy brk rendszerhívást a méret beállítására. Hasonlóan nincs külön rendszerhívás a *geteuid* és a *getegid* függvényekhez. A getuid és a *getgid* hívás egyaránt visszaadja az effektív és a valódi azonosítót. Hasonló módon a *getpid* visszaadja a hívó processzus és a szülőjének az azonosítóját is (PID).

Az üzenetkezelésnél használt legfontosabb adatszerkezet a *table.c*-ben deklarált *call_vec* tábla. Ez tartalmazza a különböző üzeneteket feldolgozó eljárások címét. Ha egy üzenet érkezik a PM-hez, akkor a főciklus az üzenet típusát elhelyezi a *call_nr* globális változóba. Ezt az értéket használja a *call_vec* vektor indexeként, hogy megtalálja az üzenetet kezelő eljárást. Ez az eljárás hajtja végre a rendszerhívást. Az eljárás által visszaadott értéket a válaszüzenetben elküldi a hívó processzusnak, hogy tájékoztassa a rendszerhívás sikerességéről vagy sikertelenségéről. Ez a módszer hasonlít az 1.16. ábrán látott, 7. lépésben alkalmazott rendszerhívás-kezelőkre mutató mutatókat tartalmazó táblához, de itt felhasználói szinten és nem kernelszinten fut.

4.7.3. A processzuskezelő adatszerkezetei és algoritmusai

A processzuskezelőnek két fontos adatszerkezete van: a processzusok és a lyukak táblája. Ebben az alfejezetben ezeket nézzük meg.

A 2.4. ábrán láttuk a processzustábla néhány mezőjét, amelyek egyik része a kernelnek, másik része a processzuskezelőnek, illetve a fájlkezelőhöz kellett. A MINIX 3-ban az operációs rendszernek ez a három része saját külön processzustáblát használ, amelyben csak az adott részhez szükséges mezők találhatók meg. Az egyszerűség kedvéért a bejegyzések kevés kivétellel megfelelnek egymásnak, így a PM táblájának *k*-adik eleme ugyanahhoz a processzushoz tartozik, mint a fájlrendszer táblájának *k*-adik eleme. Amikor egy processzus létrejön vagy megszűnik, mind a három elem módosítja a saját tábláját, hogy az tükrözze az új állapotot.

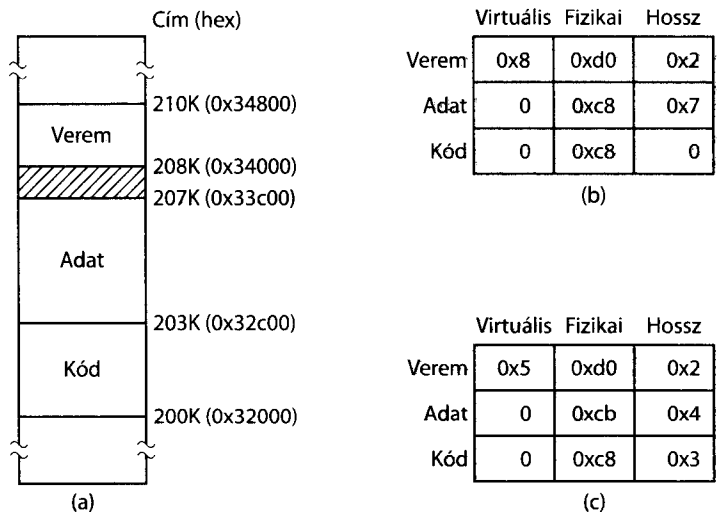
Azok a processzusok képeznek kivételt, amelyek nem láthatók a kernelen kívül, vagy azért, mert a kernelbe lettek fordítva, mint az *ÓRA (CLOCK)* és a

RENDSZER (SYSTEM) taszkok, vagy azért, mert olyanok, mint az *ÜRESJÁRAT (IDLE)* és a *KERNEL*. A kernel processzustáblájában az ezeknek megfelelő bejegyzések negatív számokkal vannak jelölve, és ezek a bejegyzések nem szerepelnek a processzuskezelő vagy fájlrendszer processzustáblájában. Tömören, amit eddig mondtunk a *k*-adik elemről, azok 0 vagy pozitív *k* értékekre igazak.

Processzusok a memóriában

A PM processzustábláját *mproc*-nak hívják, az */src/servers/pm/mproc.h* fájlban található a definíciója. A processzus memóriafoglalását leíró mezőkön kívül néhány egyéb információt is tartalmaz. A legfontosabb mező az *mp_seg* tömb, amelynek három bejegyzése van a kód-, az adat- és a veremszegmens leírására. Mindegyik elem egy struktúra, amely a megfelelő szegmens hosszát, virtuális és fizikai címét tartalmazza, bájt helyett **memóriaszelet**ben **(click)** megadva. A memóriaszelet mérete implementációfüggő, a régi MINIX-verziókban 256 bájt volt, a MINIX 3-ban 1024 bájt. Mindegyik szegmensnek memóriaszelet-határon kell kezdődnie, és mérete csak a memóriaszelet egész számú többszöröse lehet.

A memóriafoglalás módját a 4.33. ábra szemlélteti, ahol processzusunknak 3 KB-os kódja, 4 KB-os adatrésze van, majd egy 1 KB-os hézag után a 2 KB-os verem következik, így a teljes memóriafoglalás 10 KB. A 4.33.(b) ábrán azt az esetet láthatjuk, amikor a processzusnak nincs szeparált kód- és adatrésze, ilyenkor a kódszegmens mindig üres, az adatokat és a kódot az adatszegmens tartalmazza. Amikor a processzus a 0-s virtuális címre hivatkozik, akár ráugrik, akár olvassa



4.33. ábra. *(a) Egy processzus a memóriában. (b) A memória leírása kombinált kód- és adatrész esetén. (c) A memória leírása szeparált kód- és adatrész esetén*

(azaz utasítás vagy adat), a 0x32000-es (decimálisan 200 KB) fizikai címet használja; ez a cím a 0xc8-as memóriaszeletben van.

Jegyezzük meg, hogy a verem kezdetének virtuális címe a processzus számára kezdetben lefoglalt memória nagyságától függ. Ha a *chmem* paranccsal módosítjuk a fájl fejlécét, hogy nagyobb dinamikus területet biztosítsunk (nagyobb a hézag a verem- és az adatszegmens között), akkor a verem magasabb virtuális címen kezdődik. Ha a verem mérete egy memóriaszelettel nő, akkor a veremszegmens bejegyzésének a (0x8, 0xd0, 0x2) hármasról (0x7, 0xcf, 0x3)-re *kell* változnia. Vegyük észre, hogy ebben a példában, ha nem lett volna megnövelve a teljes memória mérete, akkor a verem növelése egy memóriaszelettel megszüntette volna a helyközt.

A 8088-as processzoron nincs veremtúlcsordulási hiba, így a MINIX úgy definiálja a vermet, hogy a 32 bites processzorokon is csak akkor lépjen fel hiba, ha a verem már az adatszegmenst írja felül. Így ez a változtatás csak a következő brk rendszerhívásnál történik meg, ahol az operációs rendszer kiolvassa a veremmutatót (SP) és újraszámolja a szegmensbejegyzéseket. Egy olyan gépen, ahol van veremmegszakítás, a veremszegmens már akkor növelhető, amikor a veremmutató kilép a szegmensből. A MINIX 3 nem így működik a 32 bites Intel processzorokon; ennek okát a következőkben ismertetjük.

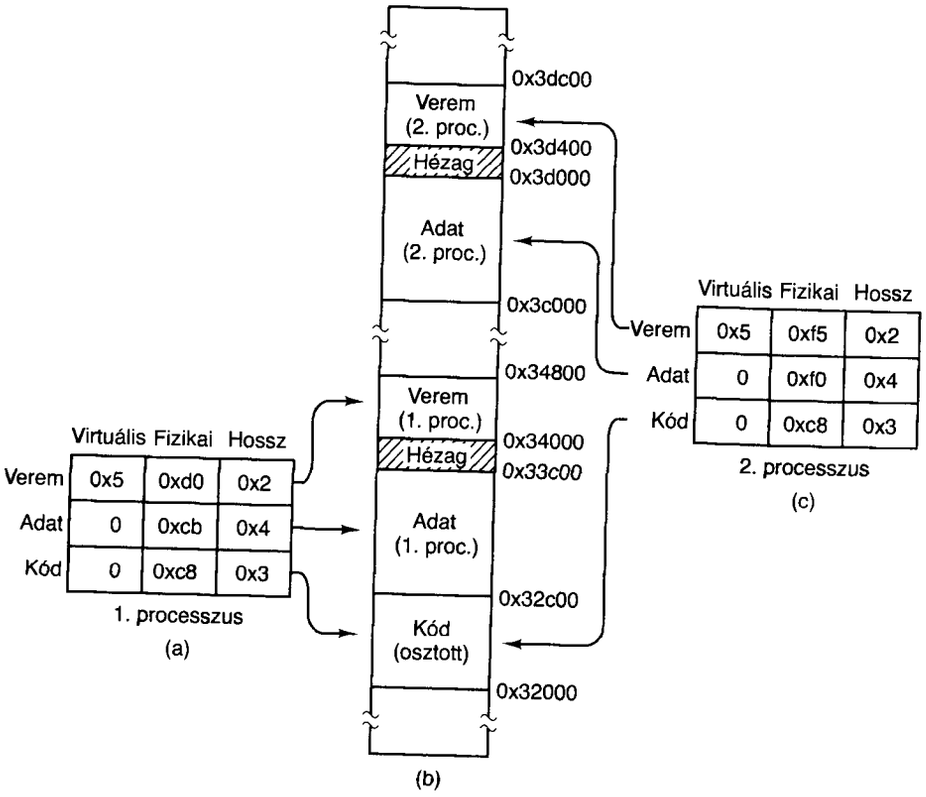
Korábban már említettük, hogy a hardvertervezők nem mindig ugyanazt produkálják, mint amit a programozók szeretnének. A Pentiumon még védett módban sem okoz megszakítást a MINIX 3, ha a verem kinövi a szegmensét. Bár az Intel hardvere védett módban észreveszi a szegmensen kívüli memória elérésének kísérletét (a szegmens a 4.26. ábrán látható leíróval van definiálva), a MINIX 3-ban az adat- és a veremszegmens leírója mindig ugyanaz. A MINIX 3 által definiált adat és verem ugyanazt a szegmens használja, és a közöttük levő hézagot használhatják növekedésre, azonban ezt csak a MINIX 3 felügyelheti. A processzor nem veszi észre a hézaggal kapcsolatos hibákat, mert a hézagot az adat- és a veremterület részeként tartja számon. Természetesen azt észreveszi a hardver, ha a kombinált adat-hézag-veremterületen kívüli memóriát próbáljuk elérni. Ez arra elég, hogy egy processzust megvédjünk a többi processzustól, de arra nem, hogy önmagától is.

Olyan döntést hoztunk, hogy megosztottuk a hardver által definiált szegmenst, így a MINIX 3 dinamikusan méretezheti a hézagot. A másik lehetőség az, hogy az adat és a verem külön-külön szegmensben van. Ez ugyan biztonságosabb, de sokkal memóriaigényesebb lenne. Mindenki számára rendelkezésre áll a forráskód, hogy kipróbálja a különböző változatokat.

A 4.33.(c) ábra mutatja a szegmensbejegyzéseket szeparált kód- és adatrész esetén. Itt az adat- és a kódszegmens mérete egyaránt nagyobb nullánál. A 4.33.(b) és (c) ábrán látható *mp_seg* tömböt elsősorban a virtuális címek fizikai címekre való leképezésére használjuk. Ha adott egy virtuális cím, és az, hogy melyik címtérbe tartozik, akkor egyszerű dolog eldönteni azt, hogy a cím legális (azaz a szegmensen belülre esik), vagy nem, és ha legális, akkor könnyen megadható a hozzá tartozó fizikai cím. Ezt a leképezést az *umap* kerneleljárás végzi el például az I/O-feladathoz vagy felhasználói térből és a felhasználói térbe másoláshoz.

Megosztott programkód

A verem- és az adatterület tartalma a processzus végrehajtása alatt változik, de a kód változatlan marad. Ezért ez utóbbi megosztható több processzus között, amelyek ugyanazt a programkódot hajtják végre, például több felhasználó egyszerre futtatja ugyanazt a parancsértelmezőt. A **megosztott programkód**dal, vagy röviden **osztott kód**dal **(shared text)** hatékonyabb memóriakihasználás érhető el. Ha az exec elindít egy processzust, akkor először megnyitja a programfájlt, és betölti a fejlécét. Ha a processzus szeparált kód- és adatrészt használ, az *mproc* táblában keres az *mp_dev*, az *mp_ino* és az *mp_ctime* mezőben. Ezek a többi processzus által végrehajtott programfájlok eszköz- és i-csomópont számát, valamint az állapotváltozás idejét tartalmazzák. Ha egy másik processzus ugyanazt a programkódot hajtja végre, nem kell a kódot még egy példányban betölteni. Az új processzus



4.34. ábra. *(a) Az előző ábrán látható processzus (szeparált kód- és adatrész) memóriatérképe. (b) A memória elrendezése a második processzus indulása után. A második processzus is ugyanazt a megosztott kódot hajtja végre. (c) A második processzus memóriatérképe*

mp_seg[*T*] (kódszegmens) mezője a már betöltött kódterületre fog mutatni, csak az adat és a verem lesz az újonnan lefoglalt helyen (lásd 4.34. ábra). Ha a processzus kombinált kód- és adatrészt használ, vagy nem fut még példánya, a 4.33. ábra szerint foglalunk memóriát, a kódot és az adatokat is betöltjük a lemezről.

A szegmensadatokon kívül az *mproc* tábla tartalmazza a processzusnak és szülőjének az azonosítóját *(pid),* a valódi és effektív felhasználói és csoportazonosítót *(uid, gid),* információt a szignálokról és a kilépési állapotot, ha a processzus már befejeződött, de a szülője még nem hajtotta végre a wait-et. Az időzítő a sigalarm-hoz, valamint a gyermekprocesszus által használt összesített felhasználói és rendszeridő is megtalálható az *mproc* táblában. A MINIX 3-ban ezeket a feladatokat a processzuskezelő látja el, ezzel szemben a MINIX régebbi verzióiban ezeket a feladatokat a kernel látta el.

A lyukak listája

A processzuskezelő másik nagy táblája a **lyuktábla**, az *src/servers/pm/alloc.c*-ben definiált *hole,* amely kezdőcím szerint rendezve tartalmazza a lyukakat. Az adat- és a veremszegmens közötti hézag nem számít lyuknak, mert az már egy processzushoz tartozik, ezért nincs benne a szabad lyukak listájában. A lyuktábla minden eleme három mezőt tartalmaz: a lyuk kezdőcímét és hosszát memóriaszeletben megadva és a következő listaelem címét. A lista egyszeresen láncolt, így könnyű megtalálni egy adott lyuk után következő lyukat, de az előző lyukért az egész listát elölről végig kell keresni.

Az *alloc.c* teljes forráskódja megtalálható a mellékelt CD-n. A lyuklistát definiáló kód rövid, így megtekinthetjük a 4. 35. ábrán.

A lyukak és szegmensek méretét hatékonysági okokból számoljuk bájtok helyett memóriaszeletben. 16 bites módban 16 bites számokat használunk a címzéshez, így 1024 bájtos memóriaszeletben 64 MB memóriát tudunk címezni, 32 bites módban pedig $2^{32} \times 2^{10} = 2^{42}$ bájtot, azaz 4 TB-ot (4096 gigabájtot).

A lyuklista két fő művelete egy megadott méretű memória lefoglalása és a lefoglalt memória felszabadítása. Memória lefoglalásához a lyuklista elejétől kezdve keresünk, amíg egy megfelelő méretű lyukat nem találunk (first fit). A szegmensnek a lyukból adunk memóriát, és abban a ritka esetben, amikor a lyuk és a szegmens mérete pontosan megegyezik, eltávolítjuk a lyukat a listából. Ez a módszer gyors és egyszerű, de külső és belső töredezettséget is okozhat (az utolsó memóriaszeletben akár 1023 bájt is veszendőbe mehet).

```
PRIVATE struct hole {
   struct hole *h_next;     /* a következő bejegyzésre mutató pointer */
   phys_clicks h_base;      /* hol kezdődik a lyuk? */
   phys_clicks h_len;       /* milyen nagy a lyuk? */
} hole[NR_HOLES];
```

4.35. ábra. *A lyuklista egy lyukstruktúrát* (struct hole) *tartalmazó tömb*

Amikor egy processzus befejeződik, akkor az adat- és a veremszegmensét vissza kell láncolni a szabad listába. Ha a processzus kombinált kód- és adatrészt használ, akkor ezzel az összes számára lefoglalt memória felszabadult, mert az ilyen processzusok nem foglalnak külön memóriát a kódjuknak. Ha a processzus szeparált adat- és kódrésszel rendelkezik, és a processzustábla szerint más processzus nem használja ezt a kódot, akkor ezt is fel kell szabadítani. Mivel ilyenkor a kód- és az adatszegmens nem feltétlenül szomszédos, két memóriaszeletet kell visszaadni. Minden visszaadott memóriaterületet össze kell olvasztani az esetleges szomszédos lyukakkal, hogy ne fordulhassanak elő egymás melletti lyukak. Emiatt a lyukak száma, helye és mérete a rendszerműveletek alatt állandóan változik. Amikor minden felhasználói processzus befejeződött, akkor az egész memória újra a rendelkezésre áll. Ez nem feltétlenül egyetlen lyuk, mert a fizikai memóriát megszakíthatják az operációs rendszer számára használhatatlan területek, például a 640 KB és 1 MB közötti ROM és I/O-terület az IBM-kompatibilis gépeken.

4.7.4. A fork, az exit és a wait rendszerhívás

Amikor egy processzus létrejön vagy megsemmisül, akkor memóriát kell foglalni vagy felszabadítani, és módosítani kell a kernel és a fájlrendszer processzustábláját. Ezeket a tevékenységeket mind a PM irányítja. A processzusok fork-kal történő létrehozásának és végrehajtásának lépéseit a 4.36. ábrán láthatjuk.

Mivel egy fork-ot nehéz és kényelmetlen menet közben leállítani, a PM egy számlálót tart fenn, amely az éppen futó processzusok számát adja meg, így könnyen eldönthető, hogy van-e szabad hely a processzustáblában. Ha a tábla nincs tele, akkor megkísérlünk memóriát foglalni a gyermekprocesszusnak. Ha a processzusnak szeparált kód- és adatrésze van, akkor elég csak az adat- és veremszegmensnek helyet foglalni. Ha ez a lépés sikerült, akkor a fork biztos, hogy jól működik. Ezután feltölti a lefoglalt memóriát, beállítja a processzustábla bejegyzését, választ egy PID-et, és tájékoztatja a rendszer többi részét az új processzus létrejöttéről.

Egy processzus akkor fejeződik be, ha a következő két dolog megtörtént: (1) a processzus kilépett (vagy leállították egy szignállal); és (2) a szülőprocesszus vég-

1. A processzustábla telítettségének ellenőrzése.
2. Memória lefoglalása a gyermekprocesszus adat- és veremszegmensének.
3. A szülő adatainak és vermének átmásolása a gyermek területére.
4. Szabad bejegyzés keresése a processzustáblában és a szülő adatainak bemásolása.
5. A gyermek memóriatérképének beírása a processzustáblába.
6. A gyermek processzusazonosítójának kiválasztása.
7. Informálni a kernelt és a fájlrendszert a gyermekről.
8. Elküldeni a gyermek memóriatérképét a kernelnek.
9. Elküldeni a választ a szülőnek és a gyermeknek.

4.36. ábra. *A* fork *rendszerhívásnál végrehajtandó lépések*

rehajtott egy wait rendszerhívást, hogy megtudja, mi történt. Ha egy processzus már kilépett, vagy leállították, de a szülő még nem hajtotta végre a wait-et, akkor a processzus felfüggesztett, ún. **zombi állapot**ba kerül. Az ilyen processzus nem kerül ütemezésre, és nem kap szignálokat, felszabadítjuk a hozzá tartozó memóriát, de nem távolítjuk el az azonosítóját a processzustáblából. A zombi állapot átmeneti, és ritkán tart hosszú ideig. Amikor a szülő végrehajtja a wait-et, akkor a processzustábla bejegyzése felszabadul, és az operációs rendszer többi része is tudomást szerez a processzus befejeződéséről.

A probléma az, hogy mi van akkor, ha a szülőprocesszus már leállt. Ha nem teszünk semmit, akkor az összes gyermekprocesszus zombi marad. Ezért megváltoztatjuk a processzustáblát, hogy az *init* processzus legyen a szülő. Amikor a rendszer elindul, az *init* elolvassa az */etc/ttytab* fájlból a terminálok listáját, és mindegyikhez elindít (fork) egy bejelentkező processzust. Ezután blokkolódik, és a processzusok leállására vár. Ezen a módon gyorsan megszabadulhatunk az árva zombiktól.

4.7.5. Az exec rendszerhívás

Amikor a terminálon begépelünk egy parancsot, akkor a parancsértelmezőről leválik egy új processzus, amely végrehajtja a kért parancsot. Lehetséges lenne a fork-ot és az exec-et egyetlen rendszerhívássá összevonni, de két külön rendszerhívást használunk, mert így egyszerűbb a be- és kimenet átirányításának megvalósítása. Amikor a parancsértelmező szétválik, és a szabványos bemenet át van irányítva, akkor a gyermekprocesszus lezárja a bemenetet, és megnyitja az újat, majd végrehajtja a parancsot. Emiatt az elindított processzus az átirányított bemenetet örökli. A szabványos kimenetet hasonló módon kezeljük.

Az exec a legbonyolultabb rendszerhívás a MINIX 3-ban. Az aktuális memóriaképet az újjal kell helyettesítenie, beleértve a vermet is. Az új képnek természetesen binárisan kompatibilisnek kell lennie. A futtatandó fájl lehet egy szkript is, amelyet egy értelmező (például perl) fog futtatni. Ez esetben a bináris kép, amelyet a memóriába kell tölteni, a parancsértelmező bináris képe a szkript nevével mint argumentummal. Ebben a szakaszban csak az egyszerű esettel foglalkozunk, később amikor az exec megvalósításával foglalkozunk, majd kitérünk a szkript futtatásához szükséges egyéb feldolgozásra is. Az exec a munkáját a 4.37. ábrán látható lépésekben végzi el.

Mindegyik lépés több kisebb részből áll, néhány ezek közül meghiúsulhat. Például lehet, hogy nincs elég memória az új processzusnak. Ezért a régi processzus memóriaképét nem szabad tönkretenni, amíg az exec sikerességéről meg nem bizonyosodunk, így visszaadhatjuk a vezérlést a hívó processzusnak, ha valami hiba történt. Normális esetben az exec-nek nincs visszatérési értéke, de ha nem sikerül, akkor hibakóddal visszakapja a vezérlést a hívó processzus.

A 4.37. ábra néhány lépése több magyarázatot igényel. Az első kérdés, hogy van-e elég memória. Miután meghatároztuk, hogy mennyi memória kell (ez attól is függ, hogy a kódszegmens megosztható-e más processzusokal), keresünk a lyuk-

1. A jogok ellenőrzése. Végrehajtható-e a fájl?
2. A fejléc beolvasása, amelyből megkapjuk a szegmensek méretét és a teljes méretet.
3. A paraméterek és a környezeti változók átmásolása a hívótól.
4. Memória lefoglalása és a hívó processzus memóriájának felszabadítása.
5. Betölti az új processzus vermét.
6. Betölti az új processzus adatait (és kódját).
7. A setuid, setgid bitek ellenőrzése, karbantartása.
8. A processzustábla bejegyzésének kitöltése.
9. Tudatni a kernellel, hogy a processzus futtatható.

4.37. ábra. *Az* exec *rendszerhívásnál végrehajtandó lépések*

listában egy megfelelő méretű lyukat, még *mielőtt* a régi processzus memóriaterületét felszabadítanánk. Ha először felszabadítjuk a régi területet, és utána a keresés sikertelen, akkor nehéz visszatérni a régi állapotba.

Azonban ez a vizsgálat nagyon szigorú. Olyan esetben is visszautasíthatja az exec végrehajtását, amikor lenne elég memória. Például tegyük fel, hogy az exec-et hívó processzus 20 KB-ot foglal el, és kódját nem osztja meg más processzusokkal, a memóriában egy 30 KB-os lyuk van, és az elindított processzusnak 50 KB-ra van szüksége. Ha a felszabadítás előtt vizsgáljuk a lyuklistát, akkor úgy találjuk, hogy csak 30 KB szabad hely van, és az exec meghiúsul. Ha először felszabadítjuk a régi memóriaterületet, akkor az exec sikeres lehet, attól függően, hogy az új 20 KB-os lyukat összeolvaszthatjuk-e a 30 KB-ossal. A bonyolultabb rendszer-implementációk egy kicsit jobban kezelik az ilyen problémákat.

Ha olyan processzust akarunk indítani, amelynek szeparált kód- és adatrésze van, akkor két lyukat kell keresni a listában, egyet a kód-, egyet az adatszegmensnek. Ennek a két szegmensnek nem kell szomszédosnak lennie.

Újabb probléma adódhat, ha a processzus virtuális címteret használ. A gond az, hogy a memóriát nem bájtokban, hanem 1024 bájtos memóriaszeletben kezeljük. Mindegyik memóriaszelet pontosan egy szegmenshez tartozik, nem lehet például olyan, hogy egy szegmens egyik fele adat, másik fele verem, mivel az egész memóriakezelés memóriaszeletenként történik.

Hogy lássuk, miért okoz problémát ez a megszorítás, tekintsük a 16 bites Intel processzor (8088, 80286) 64 KB-os címterét, ahol 1024 méretű blokkokkal, 64 memóriaszeletünk lehet. Tegyük fel, hogy egy elkülönített adat- és kódszegmenssel rendelkező programnak 40 000 bájt hosszú kódja, 32 770 bájt adata és 32 760 bájtos verme van. Az adatszegmens 33 memóriaszeletet foglal el. Bár az utolsó memóriaszeletnek csak az utolsó 2 bájtja használt, ettől függetlenül az egész memóriaszeletet le kell foglalnunk az adatszegmens részére. Együtt nem férnek el 64 memóriaszeletben, bár bájtban mért összméretük beleférne a virtuális címtérbe. Elméletben ez a probléma minden olyan gépen létezik, amely egy bájtnál nagyobb memóriaszelet-méretet használ, de a gyakorlatban a Pentium szintű processzoroknál ritkán fordul elő, mert azok nagy (4 GB-os) szegmenseket is megengednek. Az a rendszer, amely nem ellenőrzi a ritka, de lehetséges körülményeket, minden előzmény nélkül összeomolhat, amikor ezek a körülmények előfordulnak.

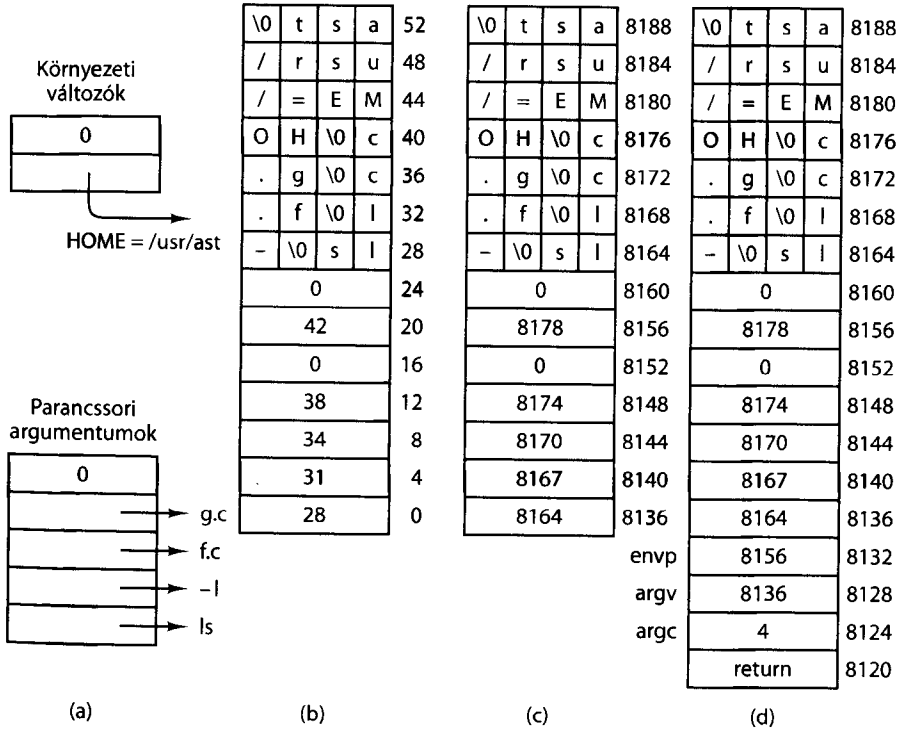
A másik lényeges probléma, hogy miként töltsük fel kezdetben a processzus vermét. Az exec-et hívó könyvtári függvénynek három paramétere van:

```
execve(name, argv, envp);
```

ahol a *name* egy mutató a végrehajtandó fájl nevére, az *argv* egy mutatótömbre mutat, amelynek elemei a program parancssori paramétereire mutatnak, az *envp* hasonlóképpen a környezeti változókat tartalmazza.

Könnyű lenne úgy implementálni az exec-et, hogy egy üzenetben adja át ezt a három mutatót a PM-nek, és az maga olvassa ki a fájlnevet és a két tömböt. Ezután egyenként át kellene vinni az argumentumokat és a környezeti változókat. Ez minden elemre egy, de lehet, hogy több üzenetet jelent a rendszertaszk felé, mert a PM nem tudja előre, hogy mekkorák lesznek ezek az adatok.

E módszer jobb megértéséhez teljesen más stratégiát választunk. Az *execve* könyvtári eljárás saját vermet épít fel, és ennek a címét adja át a PM-nek. A verem felépítése a felhasználói címtérben nagyon hatékonyan megoldható, mert az egyes objektumok a lokális memóriában, és nem egy másik címtérben vannak.



4.38. ábra. *(a) Az* execve*-nek átadott tömbök. (b) Az* execve *által felépített verem. (c) A verem, miután a PM elvégezte a módosításokat. (d) A verem, ahogy a* main *látja a végrehajtás indulásakor*

Hogy jobban megértsük ezt a módszert, tekintsük a következő példát. Amikor a felhasználó begépeli az

```
ls -l f.c g.c
```

parancsot, akkor azt a parancsértelmező feldolgozza, és egy

```
execve("/bin/ls", argv, envp);
```

könyvtári függvényhívást hajt végre. A két mutatótömb tartalma a 4.38.(a) ábrán látható. Az *execve* eljárás a parancsértelmező címterében felépíti a 4.38.(b) ábrán látható kezdeti vermet. A verem változtatás nélkül átmásolódik a PM-hez az exec végrehajtása alatt.

Amikor végül a vermet az elindított processzus területére másoljuk, az nem a nullás virtuális címre, hanem a lefoglalt memória végére kerül, a virtuális címet a fájl fejlécének memóriaméretet megadó mezője határozza meg. Például legyen ez a teljes méret 8192, így a program által elérhető legutolsó bájt a 8191-es virtuális címen van. Ezért a PM-nek a 4.38.(c) ábrán látható módon meg kell változtatnia a veremben levő mutatókat, hogy azok az áthelyezett címekre mutassanak.

Amikor az exec befejeződik, és a program elindul, a verem a 4.38.(c) ábrán látható állapotban van, a veremmutató értéke 8136. Azonban megoldásra vár még egy probléma. A végrehajtandó fájl főprogramja valahogy úgy van deklarálva, hogy

```
main(argc, argv, envp);
```

Ami a C fordítót illeti, a *main* is csak egy függvény, nem tudja, hogy a *main* különleges, olyan kódot fordít belőle, amely a három paramétert a szabványos C-konvenciók szerint kezeli, azaz a legutolsó paraméter van elöl. A három paraméternek (egy egész szám és két mutató) a visszatérési cím előtti három szót kell elfoglalnia a veremben. Természetesen a 4.38.(c) ábra nem így néz ki.

A megoldás az, hogy a program ne a *main* függvénnyel kezdődjön. Ehelyett a kód nullás címére szerkesztett **crtso (C run-time, start-off)** nevű, assembly nyelvű indítóeljárás (-rutin) kapja meg először a vezérlést. Ez először berakja ezt a három szót a verembe, majd egy call utasítással meghívja a *main* függvényt. A *main* indulásának pillanatában a verem állapota a 4.38.(d) ábrán látható. Ezzel a trükkel a *main*-t a szokásos módon hívhatjuk meg (valójában ez nem trükk, hanem a hívás szokványos módja).

Ha a programozó a *main* végén elhanyagolja az *exit* meghívását, akkor a végrehajtás visszatér a *main*-t hívó indítórutinra, amikor a *main* befejeződik. Mivel a *main* egy közönséges függvény, a végére a fordító odarakja a visszatéréshez szükséges utasításokat. Így a *main* visszaadja a vezérlést a hívó eljárásnak, és az meghívja az *exit*-et. A 32 bites *crtso* kódjának legnagyobb része a 4.39. ábrán látható. A megjegyzések világossá teszik az utasítások szerepét. Ami hiányzik, az a környezet inicializálása, amennyiben a programozó nem definiálta, a verembe rakott regiszterek feltöltése, és néhány sor, ami a lebegőpontos segédprocesszor meglétét jelző állapotbitet állítja be. A teljes programkód megtalálható az *src/lib/i386/rts/crtso.s* fájlban.

```
push    ecx     ! A verembe rakja a környezeti változók címét
push    edx     ! A verembe rakja a paraméterek címét (argv)
push    eax     ! A verembe rakja a paraméterek számát (argc)
call    _main   ! main(argc, argv, envp)
push    eax     ! A verembe rakja a kilépési állapotot
call    _exit
hlt             ! Megszakítás, ha az exit meghiúsul
```

4.39. ábra. *A C indítórutin* (crtso) *legfontosabb része*

4.7.6. A brk rendszerhívás

A *brk* és *sbrk* könyvtári eljárásokat használjuk az adatszegmens felső határának állítására. Az előbbi paramétere az abszolút méret (bájtokban mérve), és a brk-t hívja meg. Az utóbbi az aktuális mérettől való eltérést (ez negatív és pozitív érték is lehet) kapja meg, kiszámolja az új méretet, és meghívja a brk-t. Jelenleg nincs sbrk rendszerhívás.

Érdekes kérdés, hogy honnan tudja az *sbrk* az aktuális méretet az új méret kiszámításához. A megoldást a *brksize* nevű változó jelenti, ami mindig az aktuális méretet tárolja. A változó értéke kezdetben a fordítóprogram által beállított szimbólum, amely a kód és adat együttes méretét (egyesített adat- és kódrész esetén), vagy csak az adat méretét (szeparált adat- és kódrész esetén) adja meg. A szimbólum neve fordítóprogram-függő, így egyik definiciós fájlban sem található meg. A *brksize.s* nevű fájlban van, ennek helye rendszerfüggő, de ugyanabban a könyvtárban van, mint a *crtso.s*.

A brk egyszerű feladat a processzuskezelő számára. Csak ellenőrizni kell, hogy belefér-e még minden a címtérbe, módosítani a táblát, és informálni a kernelt.

4.7.7. Szignálkezelés

Az első fejezetben úgy írtuk le a **szignálok**at mint módszert arra, hogy információt küldjünk olyan processzusoknak, amelyek nem feltétlenül várnak inputra. Szignálok halmazát definiáltuk, minden szignálhoz tartozik egy alapértelmezett tevékenység, amely vagy leállítja a processzust, amelyiknek küldték, vagy figyelmen kívül hagyják. Könnyű dolog lenne a szignálkezelőt implementálni, ha csak ez a két lehetőség lenne. Vannak azonban olyan rendszerhívások is, amelyekkel a processzusok megváltoztathatják a szignálokra adott válaszukat. Egy processzus a sigkill kivételével bármilyen szignált figyelmen kívül hagyhat. Sőt a processzus által definiálhatja az adott szignált **elkapó szignálkezelő** eljárást is, amely végrehajtódhat, ha egy szignál érkezik. A sigkill-t nem lehet ilyen módon kezelni, ha egy processzus egy sigkill-t kap, akkor azonnal befejeződik. A programozónak úgy tűnik, hogy az operációs rendszer csak kétszer foglalkozik a szignálokkal: az előkészítő szakaszban, amikor a processzus módosíthatja a szignálokra adott válaszát, és a válasznál, amikor a szignál generálódik, és a hozzá tartozó választevékenység

Felkészülés: a programkód felkészül egy lehetséges jelre
Válasz: a jel megérkezett és a kód reagált rá
Takarítás: a processzus normális állapotának a visszaállítása

4.40. ábra. *A szignálkezelés fázisai*

lezajlik. A szignálra adott válasz a programozó által írt szignálkezelő eljárás lefuttatása is lehet. A 4.40. ábrán látható harmadik fázis is bekövetkezik. Amikor a programozó által írt eljárás befejeződik, akkor egy speciális rendszerhívás visszaadja a vezérlést a processzus normál működésére. A programozónak erről nem kell tudni, úgy írhatja meg a szignálkezelőt, mint egy közönséges eljárást, az operációs rendszer gondoskodik a meghívásáról, és befejeződése után rendbe teszi a vermet.

Az előkészítő fázisban számos rendszerhívás van, amelyet a processzus végrehajthat, hogy megváltoztassa egy szignálra adott válaszát. A legáltalánosabb a sigaction, amellyel megadhatjuk, hogy a processzus figyelmen kívül hagyjon egy szignált, saját szignálkezelő eljárást adhatunk meg, vagy visszaállíthatjuk az alapértelmezett tevékenységet. A sigprocmask rendszerhívással blokkolhatunk egy szignált, ezáltal a szignálok egy sorba kerülnek, és csak akkor fejtik ki hatásukat, ha azt a processzus újra engedélyezi. Ezt a hívást bárhol lehet használni, még a szignálkezelő eljáráson belül is. A MINIX 3-ban a szignálkezelés előkészítő szakaszát teljes egészében a PM bonyolítja, mivel a szükséges adatok a PM processzustáblájában vannak. Minden processzushoz tartozik néhány *sigset_t* típusú, egyébként bittérképváltozó, amelyekben minden szignált egy bittel reprezentálunk. Az egyik ilyen változó azokat a szignálokat adja meg, amelyeket a processzus figyelmen kívül hagy, egy másik azokat, amelyeket elkap, és így tovább. Minden processzushoz tartozik még egy *sigaction* struktúrákból álló tömb, amelynek minden eleme egy szignálhoz tartozik. A struktúra a 4.41. ábrán látható. A tömbelem tartalmazza a szignált kezelő eljárás címét és egy *sigset_t* változót, amely megmondja, hogy milyen szignálokat kell a kezelő végrehajtása alatt blokkolni. Ezt a mezőt akkor használjuk, ha a szignál alapértelmezett tevékenysége helyett egy saját eljárást akarunk végrehajtani.

Itt említjük meg, hogy a rendszerprocesszusok, mint például a processzuskezelő, nem tudnak szignálokat fogadni. A rendszerprocesszusok a *SYS_MESS*-nek nevezett új kezelőtípust használják, amely tudatja a PM-mel, hogy továbbítsa a szignált egy *SYS_SIG* értesítő üzenetben. A *SIG_MESS* üzenetek után nem kell takarítani.

```
struct sigaction {
   __sighandler_t sa_handler;  /* SIG_DFL, SIG_IGN, SIG_MESS vagy egy mutató
                                  a függvényre */
   sigset_t sa_mask;           /* a kezelő alatt blokkolandó szignálok */
   int sa_flags;               /* speciális jelzők */
}
```

4.41. ábra. *A sigaction struktúra*

Ha egy szignál generálódik, akkor a MINIX 3-nak több alrendszere is működésbe lép. Először a PM a fenti adatstruktúrából kitalálja, hogy melyik processzusnak kell kapnia a szignált. Ha ez elkapja a szignált, akkor meg kell hívni a szignálkezelő eljárását. Ehhez a processzus aktuális állapotát el kell menteni, hogy később visszatérhessünk a normális végrehajtáshoz. Ezt az információt a szignált kapó processzus vermében tároljuk, így ellenőrizni is kell, hogy van-e elég hely a veremben. Ezt az ellenőrzést a PM végzi, majd a rendszertaszk rakja a verembe az információt. A rendszertaszk állítja át a processzus programszámlálóját, hogy az a szignálkezelőt hajtsa végre. Amikor a processzus szignálkezelője befejeződik, akkor a sigreturn rendszerhívás hajtódik végre. A rendszerhívás alatt a PM és a kernel is közreműködik a processzus regisztereinek visszaállításában, hogy az folytathassa a normális működését. Ha a processzus nem kapja el a szignált, akkor az operációs rendszer az alapértelmezett tevékenységet hajtja végre, ami lehet, hogy leállítja a processzust, és a fájlrendszert meghíva a lemezen egy **core dump** (a processzus memóriaképe) fájlt készít. Ebben a feladatban részt vesz a kernel, a fájlrendszer, valamint a PM. Végül a PM megismételheti, akár többször is, ezeket a tevékenységeket, lehet, hogy egyetlen szignált egy processzuscsoportnak kell elküldeni.

A MINIX 3 által ismert szignálok a POSIX szabványnak megfelelő */include/signal.h* fájlban vannak definiálva, listájuk a 4.42. ábrán látható. Minden kötelező POSIX-szignál definiált a MINIX 3-ban, de jelenleg nem mind támogatott. Például a POSIX-ban számos szignált találunk a processzusvezérlésre, amivel egy futó processzust a háttérbe vagy vissza helyezhetünk. A MINIX 3 ezt nem támogatja, így azok a processzusok, amelyek ilyen szignálokat generálnak, nem hozhatók át a MINIX 3 alá. A MINIX 3 nem támogatja a munkavezérlést, de az ilyen szignált generáló programok portolhatók rá. Ezeket a szignálokat a rendszer figyelmen kívül hagyja. A munkavezérlést azért nem támogatja a rendszer, mert azt arra találták ki, hogy lehetőséget adjon a felhasználónak egy program elindítására, majd ettől leválva más feladat elvégzésére. A MINIX 3-mal egy program elindítása után a felhasználó az ALT-F2 billentyűk lenyomásával tud egy új virtuális terminálra váltani, hogy amíg a program fut, valami mást tudjon csinálni. A virtuális terminálokat a „szegény ember" ablakozó rendszereként tekinthetjük, amely, amennyiben a helyi konzol előtt ülünk, szükségtelenné teszi a feladatkezelést és annak szignáljait. A MINIX 3 ezenkívül még definiál néhány nem POSIX-szignált, valamint a régi forráskódok miatt néhány szinonimát a POSIX-nevekre.

Szignálokat a hagyományos Unix-rendszerekben a kill rendszerhívás és a kernel generálhat. A MINIX 3-ban egyes felhasználói szintű processzusok olyan dolgokat tesznek, amelyek a tradicionális rendszerekben a kernel feladatkörébe tartoznak. A 4.42. ábrán megtekinthető a MINIX 3 által ismert valamennyi szignál és azok származási helye. Speciális billentyűkombinációkkal kezdeményezhetünk sigint, sigquit sigkill szignálokat. A sigalrm szignált a processzuskezelő kezeli. A sigpipe szignált a fájlrendszer generálja. Bármelyik processzusnak bármilyen szignált küldhetünk a *kill* program segítségével. Egyes kernelszignálok a hardvertől függnek. Például a 8086-os és a 8088-as processzor nem támogatja az illegális műveletkódok felismerését, ez a képesség csak a 286-ostól felfele érhető el, ahol megszakítás történik az illegális kódok végrehajtásakor. Ezt a szolgáltatást a hardver

Szignál	Leírás	Mi generálja?
SIGHUP	Felfüggesztés	KILL rendszerhívás
SIGINT	Megszakítás	TTY
SIGQUIT	Kilépés	TTY
SIGILL	Illegális utasítás	Kernel (*)
SIGTRAP	Nyomkövetés	Kernel (M)
SIGABRT	Abnormális befejeződés	TTY
SIGFPE	Lebegőpontos kivétel	Kernel (*)
SIGKILL	Leállítás (nem kapható el és nem hagyható figyelmen kívül)	KILL rendszerhívás
SIGUSR1	A felhasználó által definiált szignál (1)	Nem támogatott
SIGSEGV	Szegmentálási hiba	Kernel (*)
SIGUSR2	A felhasználó által definiált szignál (2)	Nem támogatott
SIGPIPE	Egy olyan adatcsőbe írás, amelyet senki sem olvas	FS
SIGALRM	Ébresztőóra, idő lejárt	PM
SIGTERM	Processzusbefejező szignál	KILL rendszerhívás
SIGCHLD	A gyermekprocesszus befejeződött vagy leállt	PM
SIGCONT	Folytatás a megállítás után	Nem támogatott
SIGSTOP	Megállító szignál	Nem támogatott
SIGTSTP	Interaktív megállító szignál	Nem támogatott
SIGTTIN	A háttérben futó processzus olvasni akar	Nem támogatott
SIGTTOU	A háttérben futó processzus írni akar	Nem támogatott
SIGKMESS	Kernelüzenet	Kernel
SIGKSIG	Kernelüzenet függőben	Kernel
SIGKSTOP	Kernel leáll	Kernel

4.42. ábra. *A POSIX és a MINIX 3 által definiált szignálok. A (*)-gal jelzett szignálok hardverfüggők. Az (M) betűvel jelzett szignálokat a POSIX nem támogatja, csak a MINIX 3, hogy kompatibilis maradjon a régebbi programokkal. A kernelszignálok MINIX 3-specifikusak, és a kernel generálja őket, hogy értesítse a rendszertaszkokat a rendszereseményekről. Számos elavult nevet és szinonimát nem soroltunk fel*

nyújtja. Az operációs rendszer írójának kell megírnia a kódot, hogy a megszakításban generálja a szignált. A második fejezetben láttuk, hogy a *kernel/exception.c* tartalmazza ezt a kódot, és csak számos különböző feltétel fennállása esetén hajtódik végre. Így egy illegális utasítás végrehajtása esetén a sigill szignál csak 286-os vagy fejlettebb/újabb processzorokon generálódik, ha 8088-ason fut a MINIX 3, akkor soha nem találkozunk ezzel a szignállal.

Az, hogy a hardver bizonyos feltételek mellett megszakításokat vált ki, még nem jelenti azt, hogy az operációs rendszerek írói teljesen ki is használják ezeket a lehetőségeket. Például az Intel 286-ostól kezdve a memóriavédelem sokféle megsértése válthat ki hibát. A *kernel/exception.c*-ben levő kód ezekből mind a sigsegv szignált generálja. Különböző kivételek generálódnak a hardver által definiált veremszegmens és más szegmensek határtúllépésénél, mert ezeket a hibákat különbözőképpen kell kezelni. Azonban a MINIX 3 memóriahasználata miatt a hardver nem tudja az összes előforduló hibát detektálni. A hardver min-

den szegmenshez egy kezdőcímet és egy méretet rendel. Egy hardverszegmensbe összevonva található meg a verem- és az adatszegmens. A hardver által definiált szegmenskezdőcím ugyanaz, mint a MINIX 3-ban használt, de a szegmens mérete nagyobb a MINIX 3-ban definiáltnál. A hardver által definiált adatszegmenst a MINIX 3 csak akkor használhatná el teljes egészében az adatokra, ha a verem üres lenne. Hasonlóan a verem csak akkor foglalhatná el a teljes szegmenst, ha az adatterület mérete nulla lenne. Bár a hardver sokféle memóriavédelmi hibát észrevesz, nem veszi észre a leggyakoribb veremhibát, amikor a verem beleér az adatterületbe, mert a hardver regisztereiben és leíróiban a verem és az adat területe fedi egymást.

Elképzelhető lenne, hogy a kernelhez hozzátegyünk egy olyan kódot, amely, amint a processzus futási időt kap, mindig megvizsgálná az adott processzus regisztereit, és egy sigsegv szignált generálna, ha a processzus túllépte a MINIX 3 által megállapított adat- és veremszegmens határát. Ez nem éri meg, mert nagyon sok pluszutasítás végrehajtását jelentené.

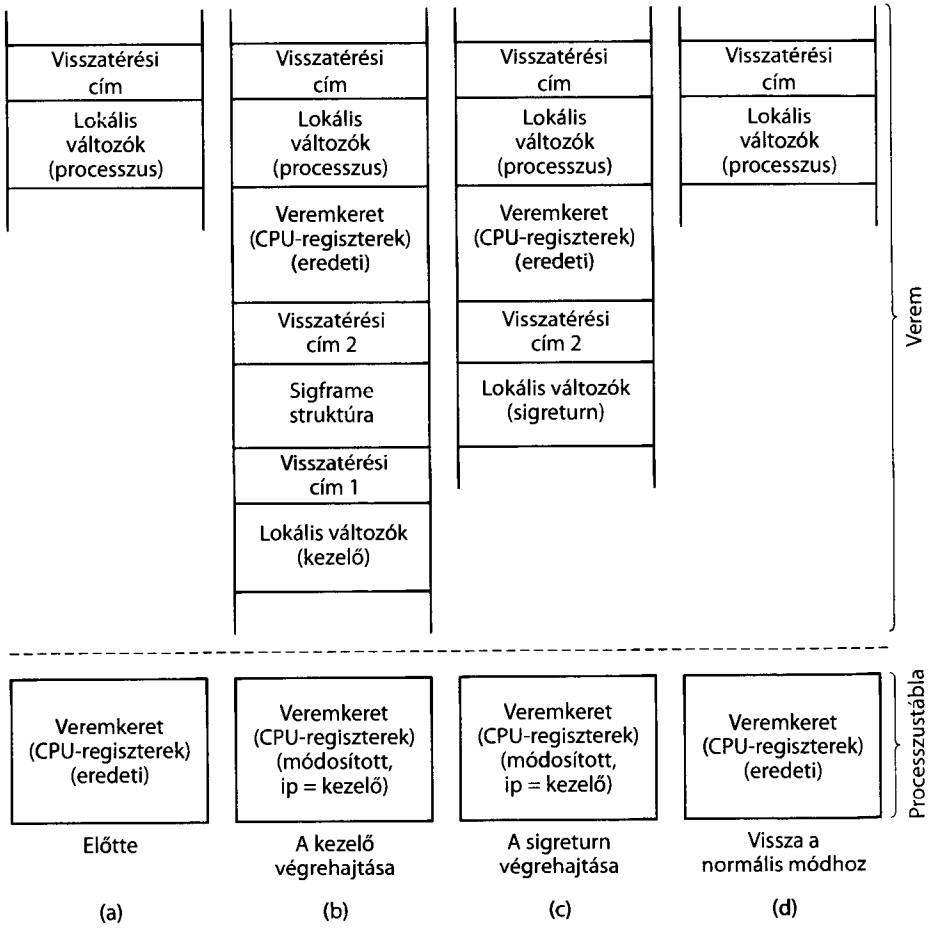
A PM minden szignált ugyanúgy kezel, függetlenül az eredetétől. Ha egy processzus egy szignált küld, akkor a PM ellenőrzi, hogy van-e joga hozzá. Egy processzus akkor küldhet szignált egy másiknak, ha a küldő egy rendszergazdai processzus, vagy a küldő processzus valós vagy effektív felhasználói azonosítója megegyezik a fogadó processzus valós vagy effektív felhasználói azonosítójának bármelyikével. Ezenkívül még számos feltétel van, ami a szignálok továbbítását szabályozza. Például egy zombi processzus nem kaphat szignált. Egy processzus, amely a sigaction meghívásával figyelmen kívül hagy egy szignált, vagy a sigprocmaskkal blokkolja, nem kapja meg az adott szignált. A szignál blokkolása és figyelmen kívül hagyása különböző dolog, mert a blokkolt szignálok várakoznak, és miután a processzus feloldja a blokkolást, megkapja a szignálokat. Végül, ha a szignállal megcélzott processzus vermében nincs elég hely az állapot elmentésére, akkor az operációs rendszer leállítja a processzust.

Ha minden feltétel teljesül, akkor a processzus megkapja a szignált. Ha a processzus nem gondoskodott a szignál elkapásáról, akkor nem kell a processzusnak információt átadni. Ebben az esetben a PM végrehajtja az alapértelmezett tevékenységet a szignálra, ami általában a processzus leállítását jelenti, de lehet, hogy core dump fájlt is létrehoz. Néhány szignálra az alapértelmezett akció a szignál figyelmen kívül hagyása. A 4.42. ábrán látható nem támogatott szignálok a POSIX-ban definiáltak, de mivel a szabványban ez engedélyezett, a MINIX 3 figyelmen kívül hagyja őket.

A szignál elkapása a processzus saját szignálkezelő eljárásának végrehajtását jelenti; ennek a címét a processzustábla *sigaction* struktúrája tárolja. A második fejezetben láttuk, hogyan tároljuk a megszakított processzus vermében a processzus újraindításához szükséges információt. A verem módosításával érjük el, hogy a processzus szignálkezelője megkapja paraméterként a szignál azonosítóját. Szintén a verem módosításából következik, hogy a processzus szignálkezelőjének végrehajtása után a sigreturn rendszerhívás következik. Ezt a rendszerhívást sohasem hívja meg közvetlenül a felhasználó processzusa. Úgy hajtódik végre, hogy a kernel belerakja a verembe a címét, így a szignálkezelő befejeződésekor ezt a címet veszi elő

visszatérési címként. A sigreturn visszaállítja az eredeti veremállapotot, így a megszakított processzus a megszakítás helyétől folytathatja végrehajtását.

Mivel a szignálküldés utolsó műveletét a rendszertaszk hajtja végre, itt érdemes azt összefoglalni, hogy miként adja át a felhasznált adatokat a PM a kernelnek. A szignálkezeléshez az is szükséges, hogy amikor egy processzus szignálkezelője fut, és az időosztás miatt egy másik processzus kapja meg a processzort, akkor a processzor visszakapása után a szignálkezelő úgy fusson tovább, mintha semmi sem történt volna. Azonban a processzustáblában csak egy hely van a processzusregiszterek értékének mentésére, amelyek az eredeti állapot visszaállításához kellenek. Erre a problémára a 4.43. ábrán látható a megoldás. Az (a) ábra mutatja a pro-



4.43. ábra. *A processzus verme (felül) és a bejegyzés a processzustáblában (alul) a szignálkezelés egyes fázisai alatt. (a) A processzus állapota, amikor nincs végrehajtás alatt. (b) Az állapot, amikor a szignálkezelő elindul. (c) A* sigreturn *végrehajtása alatti állapot. (d) Az állapot, miután a* sigreturn *befejeződött*

cesszus megszakítása utáni állapotot. A félbeszakítás időpontjában a regiszterek értéke a kernel processzustáblájába másolódik. Ebben a helyzetben generálódhatnak a szignálok, mert a processzusnak küldött szignálokat egy tőle különböző, másik processzus generálja. Mivel a szignált generáló processzus és a szignál célprocesszus különböző processzusok, így sohasem futhatnak egy időben.

A szignálkezelés előkészítéseként a processzustáblából a regiszterek értéke a processzus saját vermébe másolódik, megőrzésre. Ezután egy *sigframe* struktúrát helyezünk a verembe; ez tartalmazza a szignálkezelő befejeződése után a sigreturn által használt információkat: a könyvári eljárás címét, amely meghívja a sigreturn-t *(ret addr1)* és egy másik visszatérési címet *(ret addr2)*, ahol a processzus végrehajtása megszakadt. Amint láttuk, az utóbbi címet nem használjuk a normális végrehajtásnál.

Bár a programozó közönséges eljárásként írja meg a szignálkezelőt, nem a call utasítással hívjuk meg. A programszámláló értékét módosítjuk úgy a processzustáblában, hogy amikor a processzus *újra végrehajtásra* kerül, akkor a szignálkezelő induljon el. A 4.43.(b) ábra mutatja a szignálkezelő végrehajtása alatti állapotot. Mivel a szignálkezelő egy közönséges eljárás, a befejeződésekor a *ret addr1*-et kivesszük a veremből, és végrehajtjuk a sigreturn-t.

A 4.43.(c) ábra mutatja a sigreturn végrehajtása alatti állapotot. A *sigframe* fennmaradó része most a sigreturn lokális változóit tartalmazza. A sigreturn tevékenységének része a veremmutató kiigazítása, hogy amikor befejeződik, mint egy normális eljárás, akkor a *ret addr2* címet használja visszatérési címként. Azonban a sigreturn nem ilyen módon fejeződik be, hanem a többi rendszerhíváshoz hasonlóan úgy, hogy a kernel ütemezője dönthet: átvált egy másik processzus végrehajtására. Végül is a processzus újra ütemezésre kerül, és ezen a címen folytatódik tovább, mert ez a cím van a kernel processzustáblájában. Ez a cím azért van benne a veremben is, mert így a felhasználó nyomon követheti a programot, és ez a cím bírja rá a nyomkövetőt a verem helyes használatára, amikor a szignálkezelőt futtatja. A verem mindig úgy néz ki, mint a rendes processzusoknál: a lokális változók a visszatérési cím felett vannak.

A sigreturn igazi feladata a szignál fogadása előtti állapot visszaállítása. A legfontosabb, hogy a processzustáblában a regiszterek értékét a verembe mentett másolat alapján visszaállítsa az eredeti értékre. Amikor a sigreturn befejeződik, akkor a 4.43.(d) ábrán látható helyzetbe jutunk, amelyben a processzus újra ütemezésre vár, abban az állapotban, ahogy megszakították.

A legtöbb szignál alapértelmezett tevékenysége a processzus leállítása. A PM-nek kell arra figyelnie, hogy mely szignálokat nem kezel le, blokkol vagy hagy figyelmen kívül a szignált kapó processzus. Ha a szülőprocesszus várakozik, akkor a leállított processzust eltávolítja a processzustáblából. Ha a szülő nem várakozik, akkor a processzus zombivá válik. Bizonyos szignálokra (például a sigquit) a processzuskezelő az aktuális könyvtárban egy core dump fájlt készít a processzusról.

Könnyen megtörténhet, hogy egy olyan processzus kap szignált, amely blokkolt, mert read-re várakozik egy olyan terminálon, amelyen éppen nincs elérhető adat. Ha a processzus nem kapja el a szignált, akkor a szokásos módon leállítható. Ha elkapja, akkor felmerül a kérdés, hogy mit csináljunk a szignál feldolgozása után. Térjen vissza a processzus a várakozáshoz, vagy folytatódjon a következő utasításnál?

A MINIX 3-ban az olvasási rendszerhívás *EINTR* hibakóddal tér vissza, így a processzus látja, hogy az olvasás megszakadt egy szignál miatt. Nem egészen nyilvánvaló annak eldöntése, hogy egy processzus egy rendszerhíváson blokkolt-e. A PM ezt a fájlrendszertől tudja meg.

A POSIX által ajánlott, de nem kötelező működés az, hogy a read visszatér a szignál vételének pillanatáig beolvasott bájtok számával. Az *EINTR* visszaadása lehetővé teszi, hogy riasztást állítsunk be, és elkapjuk a sigalrm szignált. Ez egyszerű módja az időtúllépés megvalósításának, például befejezni a *login*-t és megszakítani a modemvonalat, amikor a felhasználó meghatározott ideig tétlen.

Felhasználói szintű időzítők

A szignálokat leggyakrabban az alvó processzusok programozott felébresztésére használják. Egy hagyományos operációs rendszerben a kernel vagy a kernelszinten futó időzítőmeghajtó kezeli a riasztásokat. A MINIX 3-ban a riasztások kezeléséért a processzuskezelő a felelős. Ezzel a kernel egyszerűbbé válik és a terhelése is csökken. Ha az igaz, hogy minden 1 kódsorra szükségképpen b hiba jut, akkor ebből logikusan következik, hogy egy kisebb kernel kevesebb hibalehetőséget rejt. Amennyiben a hibák száma nem is változott, a hatásuk kevésbé lesz kritikus, mivel nem a kernelszinten, hanem a felhasználói térben fejtik ki hatásukat.

Tudunk-e úgy riasztásokat kezelni, hogy ne függjünk a kerneltől? A MINIX 3-ban a válasz természetesen nemleges. A riasztásokat és a hozzá tartozó időzítőket elsőként a kernelszintű időzítőtaszk kezeli egy láncolt listában vagy várakozó sorban; ez látható a 2.49. ábrán. A várakozási sor elején lévő időzítők lejárati ideje az óra integrált áramkör minden megszakítására összehasonlítódik az aktuális idővel, és amennyiben lejárt, akkor az időzítőtaszk főciklusa aktiválódik. Az időzítőtaszk hatására egy értesítés lesz kiküldve a riasztást kérő processzusnak.

A MINIX 3 innovációja, hogy a kernelszintű időzítők csak rendszerprocesszusokat szolgálnak ki. A processzuskezelő egy másik várakozási sort kezel a riasztást kérő felhasználói processzusnak. Csak a várakozási sor elején található processzusnak kér riasztást az órától a processzuskezelő. Amennyiben nem kerül új kérés a várakozási sor elejéhez, nem szükséges kérést intézni az órához sem. (Mivel az időzítőtaszk nem kommunikál közvetlenül egyetlen más processzussal sem, ezért a riasztáskérelem az időzítőtaszktól a rendszertaszkon keresztül kerül a processzusokhoz.) Amikor a rendszer az óramegszakítás után lejárati riasztást detektál, értesítést küld a processzuskezelőnek. A PM az ezután szükséges összes teendőt ellátja: ellenőrzi a saját időzítő várakozási sorát, jelez a felhasználói processzusoknak, illetve amennyiben van még aktív riasztási kérelem a várakozási sorának elején, akkor igényel egy újabb riasztást.

Az eddig leírtakból még nem tűnik úgy, hogy a megoldás sok feladatot venne le a kernel válláról. Azonban vannak más megfontolások is. Először is, előfordulhat, hogy több időzítő is lejár egy órajelre. Az, hogy két processzus egy időben kérjen riasztást, valószínűtlennek tűnhet. Azonban annak ellenére, hogy az időzítőtaszk minden integrált áramkör megszakítására megvizsgálja a lejárati időket, mint azt

már láttuk, a megszakítások időnként le vannak tiltva. Egy PC BIOS-hívás elegendő megszakítást okozhat ahhoz, hogy sok munkára legyen szükség az elmaradás behozásához. Ez azt jelenti, hogy az időzítőtaszk által kezelt idő több órajelet is ugorhat, így már könnyen lehetséges, hogy több processzusnak is egyszerre jár le az időzítője. Mivel ezeket a processzuskezelő kezeli, a kernelszintű kódnak nem kell átfutnia a láncolt listáit a takarításhoz és többszörös értesítés generálásához.

Másodszor, a riasztásokat le lehet mondani. A felhasználó processzusa még azelőtt befejeződhet, hogy a kért időzítő lejárt volna. Az is előfordulhat, hogy az időzítő csak azt biztosította, hogy a processzus ne várjon végtelen sok ideig egy adott eseményre. Amint ez az esemény bekövetkezik, az időzítőt lemondják. Azzal, hogy a processzuskezelő foglalkozik a lemondásokkal, sok terhet vesz le a kernel válláról. A kernelszintű láncolt listáknak csak az első elemével kell foglalkozni, vagy amikor az lejárt, vagy amikor a processzuskezelő megváltoztatja azt.

Az időzítők megvalósítása érthetőbb lesz, ha most teszünk egy kis kitérőt a riasztást kezelő függvények bemutatására. Nehéz teljes képet adni egy-egy függvény megismerésével a riasztás kezelésről, mivel nagyon sok függvényt használunk úgy a kernel, mint a processzuskezelő kódjából.

Amikor a PM beállít egy riasztást a felhasználói processzusnak, a *set_alarm* függvényt használja az időzítő beállítására. Az időzítőstruktúra tartalmazza többek között a lejárati időt, a riasztást igénylő processzus azonosítására szolgáló mezőt, valamint egy mutatót, amely a végrehajtandó függvényre mutat. A riasztások kezelésére ez mindig a *cause_sigalarm* függvény. A következő lépésben a rendszertaszkot megkérik egy kernelszintű riasztás beállítására. Amikor az időzítő lejár, a kernel figyelő processzusa, a *cause_alarm* lefut, és elküld egy értesítést a processzuskezelőnek. Ebben a folyamatban több függvény és makró is szerepel, de az értesítést végül a PM *get_work* függvénye kapja meg. A PM főciklusa detektálja az értesítést mint *SYN_ALARM* üzenetet, és ezután meghívja a PM *pm_expire_timers* függvényét. Ezután a PM térben több más függvény is lefut. A *tmrs_exptimers* könyvtári függvény hatására lefut a figyelő *cause_sigalarm* függvénye, amely meghívja a *checksig* függvényt, az pedig a *sig_proc* függvényt. A *sig_proc* eldönti, hogy leállítsa-e a processzust, vagy küldjön egy *SIGALM* üzenetet. Végezetül a szignál küldéséhez szükség van a kernelszintű rendszertaszk segítségére is, mivel manipulálni kell a processzustáblát, és a processzus terében lévő vermet is. Ez látható a 4.43. ábrán.

4.7.8. Egyéb rendszerhívások

A PM néhány további rendszerhívást is lekezel. Valós idejű órákkal foglalkozik a time és az stime. A times visszaadja a processzus naplózott időit. Azért vannak itt kezelve, mert a PM kényelmes hely erre. (Egy másik idővel kapcsolatos hívásra az utime-ra még visszatérünk, amikor az 5. fejezetben a fájlrendszert kezeljük, mivel a módosítási időket az i-csomópont szerkezet tárolja.)

A *getuid* és a *geteuid* könyvtári függvények egyaránt a getuid rendszerhívást használják, amely mindkét értéket visszaadja. Hasonlóan a getgid rendszerhívás visszaadja a valódi és az effektív értéket, amelyet a *getgid* és a *getegid* függvény használ fel. Ugyanígy működik a getpid, a processzus és a szülőprocesszus azonosítójával tér vissza, a setuid és a setgid is egyszerre állítja a valós és az effektív értéket. Két további rendszerhívás van ebben a csoportban, a getpgrp és a setsid. Az előbbi a processzus csoportazonosítóját adja vissza, az utóbbi pedig beállítja azt az aktuális processzusazonosító értékére. Ez a hét művelet a legegyszerűbb MINIX 3-rendszerhívás.

A ptrace és a reboot rendszerhívásokat szintén a PM hajtja végre. Az előbbi a programok nyomkövetését segíti, az utóbbi több szempontból is hatással van a rendszerre. Azért tartozik a PM-hez, mert az első dolga az, hogy az *init* kivételével az összes processzust leállítsa egy szignállal. Ezután, hogy befejezze a munkáját, meghívja a fájlrendszert és a rendszertaszkot.

4.8. A MINIX 3 processzuskezelőjének implementációja

A PM munkájának általános ismeretében most lássuk magát a kódot. A PM teljes egészében C-ben íródott, a forráskód megjegyzésekkel ellátott, így a legtöbb rész feldolgozása nem hosszú vagy bonyolult feladat. Először röviden áttekintjük a definíciós (header) fájlokat, majd a főprogramot, végül az előbbiekben ismertetett rendszerhíváscsoportokhoz tartozó fájlokat.

4.8.1. A definíciós fájlok és az adatszerkezetek

A PM forráskönyvtárában számos ugyanolyan nevű fájl van, mint a kernel és a fájlrendszer forráskönyvtárában. Saját környezetükben ezek a fájlok hasonló funkciót töltenek be. A struktúrák párhuzamos tervezése miatt könnyebben megérthető az egész MINIX 3 szerkezete. A PM még sok egyedi nevű saját definíciós fájllal is rendelkezik. Hasonlóan a rendszer más részeihez, a globális változók a PM-nél is a *table.c* fájlban találhatók. Ebben az alfejezetben a definíciós fájlokat és a *table.c*-t vizsgáljuk meg.

Mint a MINIX 3 többi fő részének, a processzuskezelőnek is van egy fő definíciós fájlja, a *pm.h* (17000. sor). Ezt minden fordításnál felhasználjuk, és ez illeszti be a tárgymodulokhoz szükséges definíciós fájlokat a teljes rendszerre vonatkozó */usr/include* könyvtárból és alkönyvtáraiból. A *kernel/kernel.h*-ban beillesztett legtöbb fájlt itt is használjuk. A PM-hez szükségesek még az *include/fcntl.h* és az *include/unistd.h* fájlban levő definíciók. A PM saját *const.h, type.h , proto.h* és *glo.h* változatát szintén beilleszti a *pm.h*. Hasonló struktúrát láthattunk a kernelben.

A *const.h* (17100. sor) a PM által használt konstansokat definiálja.

A *type.h* jelenleg nem használt, csak azért létezik ilyen vázformában, hogy a PM-ben a fájlok ugyanolyan szerkezetűek legyenek, mint a MINIX 3 többi részé-

ben. A *proto.h* (17300. sor) azokat a függvénydefiníciókat gyűjti egybe, amelyek a PM-ben mindenütt szükségesek.

Egyes függvények (17313. és 17314. sor) áldefiníciójára (dummy) is szükség van, amikor a cserét befordítjuk a MINIX 3-ba. Azzal, hogy a makrókat ide helyeztük, egyszerűvé tettük a csere nélküli fordítást. Ha nem így lenne, akkor ezeknek a függvényhívásoknak az eltávolítását sok más forrásfájlban is el kellene végezni.

A PM globális változóit a *glo.h*-ban (17500. sor) deklaráltuk. Az *EXTERN* használatával ugyanazt a trükköt alkalmazzuk, mint a kernelben, nevezetesen, hogy az *EXTERN* egy makró, ami *extern*-re fejtődik ki, kivéve a *table.c* fájlban. Itt null értékű lesz, így az *EXTERN*-ként deklarált változókhoz tárolóterület is rendelhető.

Az első ilyen változó, az *mp,* egy mutató az *mproc* struktúrára, amely a processzustábla PM-hez tartozó részében található, és a feldolgozás alatt lévő hívást kezdeményező processzushoz tartozó adatokat tartalmazza. A második változó a *procs_in_use* számon tartja, hogy a processzustáblának hány használt bejegyzése van, így könnyű eldönteni, hogy a fork végrehajtható-e.

Az *m_in* üzenetpuffer a kérésüzenetek tárolására szolgál. A *who* az aktuális processzus indexe, az

```
mp = &mproc[who];
```

miatt az *mp*-hez kapcsolódik. Amikor egy üzenet érkezik, kivesszük belőle a rendszerhívás számát, és a *call_nr* változóba helyezzük.

Ha egy processzus abnormálisan fejeződik be, akkor a MINIX 3 egy fájlba írja a processzus memóriaképét. A fájl nevét a *core_name* határozza meg. A *core_sset* egy bittérkép, amely megadja, hogy milyen szignálok okozzanak ilyen befejeződést, az *ign_sset* pedig egy bittérkép a figyelmen kívül hagyható szignálokról. Figyeljünk arra, hogy a *core_name extern*-ként és nem *EXTERN*-ként van deklarálva. A *call_vec* tömb szintén *extern*-ként van deklarálva. Az okokra a *table.c* ismertetése során térünk ki.

Az *mproc.h* fájlban (17600. sor) van a processzustáblának a PM-hez tartozó része. A legtöbb mezőt jól leírja a hozzá tartozó megjegyzés. Néhány mező a szignálkezeléssel áll kapcsolatban, az *mp_ignore,* az *mp_catch,* az *mp_sig2mess,* az *mp_sigmask,* az *mp_sigmask2* és az *mp_sigpending* bittérképekben minden bit a processzusnak elküldhető szignált reprezentál. A *sigset_t* típus 32 bites egész, így a MINIX 3 32-féle szignált támogathat. Jelenleg 22 szignál van definiálva, de mivel a POSIX megengedi, jelenleg nincs mind támogatva. Az első szignálhoz tartozik a legkisebb helyi értékű (jobb oldali) bit. A POSIX szabványos függvényeket definiál, amelyekkel a bittérképpel reprezentált szignálhalmazba be lehet tenni egy szignált, vagy ki lehet venni egyet anélkül, hogy a programozónak tudnia kellene részletekről. A szignálkezelésben fontos az *mp_sigact* tömb. Minden szignáltípushoz tartozik egy *sigaction* típusú elem (ez a struktúra az *include/signal.h* fájlban van definiálva). A *sigaction* struktúra három mezőt tartalmaz:

1. Az *sa_handler* mező adja meg, hogy a szignált alapértelmezés szerint vagy saját eljárással kell kezelni, esetleg figyelmen kívül kell hagyni.

2. Az *sa_mask* mező *sigset_t* típusú, és meghatározza, hogy mely szignálokat kell blokkolni, amíg a szignálkezelő fut.
3. Az *sa_flags* mező a szignálhoz használt állapotbiteket tartalmaz.

Ez a tömb flexibilis szignálkezelést tesz lehetővé.

Az *mp_flags* mező különféle bitek gyűjteménye, mint ahogyan ez jelezve van a fájl végén. A mező típusa előjel nélküli egész, 16 bites a kisebb gépeken, és 32 bites 386-ostól felfelé.

A következő mező a processzustáblában az *mp_procargs*. Amikor clindul egy új processzus, akkor a verem a 4.38. ábrán látható módon épül fel, és ebben a mezőben tároljuk az új processzus *argv* tömbjének mutatóját. Ezt a *ps* parancs használja fel. Például a 4.38. ábrán a 8164 értéket tárolnánk itt, azért, hogy ezt a *ps* ki tudja majd írni a parancssorra,

```
ls -l f.c g.c
```

ha olyankor hajtjuk végre, amikor az *ls* még aktív.

Mint látni fogjuk, az *mp_swapq* mezőt a MINIX 3-ban nem használjuk. Akkor használjuk csak, amikor a csere engedélyezve van. Ilyenkor egy cserét váró processzusokat tartalmazó várokozási sorra mutat. Az *mp_reply* mezőben készül a válaszüzenet. A régebbi MINIX-verziókban egy hasonló mező volt definiálva a *glo.h* fájlban, és lefordítva a *table.c* fordítása során. A MINIX 3-ban az üzenetek összeállításához szükséges terület minden processzusnak külön-külön a rendelkezésére áll. Azzal, hogy minden processzustábla sorhoz egy-egy választerület van biztosítva, a PM akkor is képes a bejövő üzenetek kezelésére, amikor a válaszüzenetet nem lehet az összeállítása után azonnal elküldeni. A PM nem tud egyszerre több kérést kezelni, de amennyiben szükséges, a válaszokat el tudja halasztani, és megpróbálhatja utolérni saját magát azzal, hogy amikor az egyes kéréseket kiszolgálta, nekiállhat kiszolgálni a még függő kéréseket.

A processzustábla utolsó két eleme nagyzolásnak tűnhet. Az *mp_nice* helyet biztosít arra, hogy a processzus udvariasságát jelezze (nice), így a felhasználók lecsökkenthetik a processzusaik prioritását, például azért, hogy teret adjanak egy fontosabb processzusnak. A MINIX 3-ban azonban ezt a mezőt belső használatra is alkalmazzuk, mégpedig a rendszerprocesszusok (szerverek és eszközmeghajtók) szükségleteitől függő különböző prioritásainak beállítására. Az *mp_name* mező segítségével a programozó kényelmesen megtalálhatja hibakeresés közben a megfelelő processzustáblasort a memóriamentésben. Van egy olyan rendszerhívás, amely a processzustáblában keres processzusnév alapján, és visszaadja a processzus ID-jét.

Végezetül megemlítjuk, hogy a processzustábla processzuskezelőhöz tartozó része egy *NR_PROCS* (17655. sor) tömbként van deklarálva. Emlékezzünk rá, hogy a processzustábla kernelhez tartozó része *NR_TASKS + NR_PROCS* tömbként volt deklarálva a *kernel/proc.h* fájlban (5593. sor). Mint már az előzőkben elmondtuk, az olyan felhasználói szintű processzusok, mint például a processzuske-

zelő, nem tudnak a kernelbe fordított processzusokról. Ezek valójában nem első osztályú processzusok.

A következő fájl, a *param.h* (17700. sor) olyan makrókat tartalmaz, amelyekkel a rendszerhívások elérik paramétereiket. Tartalmaz még tizenkettő makrót a válaszüzenet mezőihez, valamint három makrót, amelyeket csak a fájlrendszerhez küldött üzenetekben használunk. Példaként, amennyiben a kifejezés:

```
k = m_in.pid;
```

megjelenik egy fájlban, amelyben a *param.h* be van illesztve, a C előfeldolgozó fordítás előtt a

```
k = m_in.m1_i1;
```

alakra konvertálja (17707. sor).

Mielőtt a végrehajtható kóddal folytatnánk, nézzük meg a *table.c*-t (17800. sor). Fordítás során itt allokálódnak a *glo.h*-ban és az *mproc.h*-ban *EXTERN*-nel deklarált globális változók. A

```
#define _TABLE
```

utasítás miatt az *EXTERN*-ből null karakterlánc lesz. Ez ugyanaz a módszer, mint amit a kernel kódjában láttunk. Mint már az előzőkben említettük, a *core_name* *extern*-ként és nem *EXTERN*-ként van deklarálva a *glo.h*-ban. Most megnézzük, hogy miért. A *core_name* egy inicializáló karakterlánccal van deklarálva. Az inicializálás nem lehetséges egy *extern* definíción belül.

A másik fontos dolog a *table.c*-ben a *call_vec* tömb (17815. sor). Ez egy inicializált tömb, és így nem lehet *EXTERN*-ként deklarálni a *glo.h*-ban. Amikor egy rendszerhívást kérő üzenet érkezik, kivesszük belőle a rendszerhívás számát, és ezt használjuk a *call_vec* indexeként, hogy megtaláljuk a rendszerhívást kezelő eljárást. Azok a számok, amelyekhez nincs rendszerhívás, a *no_sys* eljárást hívják meg, amely csak egy hibakódot ad vissza. Megjegyzendő, hogy a *call_vec* tömböt a *_PROTOTYPE* makróval definiáltuk. Ez egy inicializált függvénytömb, így a *_PROTOTYPE* a legegyszerűbb mód arra, hogy a forráskód a klasszikus (Kernighan–Ritchie) és a szabványos C-vel is kompatibilis legyen.

Egy utolsó megjegyzés a definíciós fájlokról: mivel a MINIX 3-at még mindig aktívan fejlesztik, vannak még kiforratlan részek. Az, hogy egyes forrásfájlok a *pm/*-ben a kernelkönyvtárból is illesztenek be definíciós fájlokat, például az egyik ilyen kiforratlan megoldás. Amennyiben erről nem tudunk, nehéz lehet megtalálni egyes fontos definíciókat. Azokat a definíciókat, amelyeket több fontos MINIX 3-komponens is használ, az *include/* könyvtárban lévő definíciós fájlokban kellene elhelyezni.

4.8.2. A főprogram

A PM-et a kerneltől és a fájlrendszertől függetlenül fordítjuk. Ebből következik, hogy saját főprogramja van, amely akkor indul el, amikor a kernel inicializálta önmagát. A belépési pont a *main.c* 18041. sorában van. Miután az *pm_init* meghívásával inicializálta magát, a PM belép a 18051. sorban levő ciklusba. A ciklusmagban meghívja a *get_work*-ot, amely a bejövő üzenetekre várakozik. Ezután a kérés feldolgozására meghívja a *call_vec*-ből kiválasztott *do_XXX* eljárást, és végül visszaküldi a választ, ha szükséges. Ez már ismerős konstrukció, így működik az I/O is.

Az előző leírás egy kicsit leegyszerűsített. Mint ahogyan a 2. fejezetben említettük, bármelyik processzusnak küldhetünk **értesítő üzenet**et. Ezeket a *call_nr* mező speciális értékeivel jelezzük. A 18055–18062. sorokban két, a PM által fogadható értesítő üzenet típust szűrünk ki, amelyek hatására a megfelelő kódrész végrehajtódik. A *valid_call_nr* üzenethez is tartozik egy szűrő a 18064. sorban, mielőtt még kísérletet tennénk a kérés végrehajtására (18067. sor). Annak ellenére, hogy a helytelen kérés valószínűtlen, érdemes letesztelni, mivel a teszt nagyon olcsó, és enélkül a következmények súlyosak lehetnek.

Egy másik említésre méltó rész a *swap_in* hívás a 18073. sorban. Mint már említettük a *proto.h*-val kapcsolatban, amennyiben a csere tiltva van (a MINIX 3-ban ez az alapértelmezett helyzet), akkor ez egy ál (dummy) függvényhívás, amely nem tesz semmit. Amennyiben viszont a csere engedélyezve van, akkor ez az a hely, ahol le kell ellenőrizni, hogy egy processzus behozható-e.

Végezetül, annak ellenére, hogy a 18070. sorban található megjegyzés azt jelzi, hogy ez az a hely, ahol a választ visszaküldik, ez mégsem egészen így van. A *setreply* hívására létrejön egy válasz az előzőkben említett aktuális processzus processzustábla-bejegyzésében. Ezután a ciklus 18078–18091. sorokban a processzustábla minden bejegyzését megvizsgálják, és minden elküldhető függő választ elküldenek, kihagyva azokat, amelyeket az adott pillanatban nem lehet elküldeni.

A *get_work* (18099. sor) és a *setreply* (18116. sor) eljárás végzi az aktuális üzenet fogadását és a válasz visszaküldését. Mivel a kernelnek nincs saját bejegyzése a processzustáblában, ezért az első függvény kis trükköt alkalmaz, hogy a kerneltől érkező üzenet PM-től származónak látszódjék. A második függvény, mint azt már említettük, nem küldi el valójában a választ, csak beállítja későbbi küldésre.

A processzuskezelő inicializálása

A *main.c* fájlban a leghosszabb eljárás a PM-et inicializáló *pm_init*. Ezt a rendszer indulása után többet már nem használjuk. Annak ellenére, hogy a szervereket és a meghajtókat külön-külön fordítjuk le és külön-külön processzusban futnak, közülük egyeseket indításkor a **betöltési memóriakép (boot image)** részeként betölt a betöltési felügyelőprogram. Nehéz elképzelni egy operációs rendszer működését PM vagy fájlrendszer nélkül, így induláskor ezek a részek valószínűleg mindig betöltődnek. Egyes eszközmeghatjók szintén a betöltési memóriakép részei. Bár a

cél az volt, hogy minél több MINIX 3-meghajtó legyen önállóan betölthető, nem igazán lehetett például a fájlrendszermeghajtó korai betöltését elkerülni.

A *pm_init* legfontosabb feladata az előre betöltött processzusok futásához szükséges PM-táblák inicializálása. Mint már említettük a PM két fontos adatszerkezetet kezel: a **lyuktáblát** (vagy **üresmemória-tábla**) és a processzustábla egy részét. Először a lyuktábláról beszélünk. A memória inicializálása komplikált. Egyszerűbb lesz megérteni, ha először megtekintjük a memória szervezését a PM indítása közben. A MINIX 3 minden információt biztosít ehhez.

Mielőtt a MINIX 3 betöltési memóriaképe betöltődik a memóriába, a betöltési felügyelőprogram meghatározza a rendelkezésre álló memória felépítését. Az indítómenüben az ESC billentyű lenyomásával tekinthetjük meg az indítóparamétereket. Egy sor a képernyőn az üresmemória-blokkokat mutatja az alábbi módon:

```
memory= 800:923e0,100000:3df0000
```

(A MINIX 3 indítása után is meg tudjuk ezt tekinteni a *sysenv* parancs segítségével vagy az F5 billentyű lenyomásával. A pontos számok természetesen mások lehetnek.)

Ez két üresmemória-blokkot ír le. Ezenfelül két foglalt memóriablokk is van. A 0x800 cím alatti memória a BIOS adatait, az elsődleges indítórekordot (master boot record) és az indítóblokkot tartalmazza. A pontos felhasználás valójában nem érdekes, és nem is áll rendelkezésünkre a betöltési felügyelőprogram indításakor. A 0x800 címen kezdődő üres memória az IBM-kompatibilis gépek „alapmemória"-ja (base memory). Ebben a példában a 0x800 (2048) címtől kezdődően 0x923e0 (599008) bájt áll a rendelkezésünkre. E fölött van 640 KB-tól 1 MB-ig található „felső memóriaterület" (upper memory), amely a ROM és dedikált I/O-adapterek RAM részére van elkülönítve, kizárva a hagyományos processzusokat. Végezetül a 0x100000 (1 MB) címtől 0x3df0000 bájt szabad hely van. Ezt a tartományt gyakran „kibővített memória"-nak (extended memory) nevezik. A példából megállapíthatjuk, hogy a gépben 64 MB RAM memória van.

Amennyiben figyelemmel kísértük a számokat, azt vehettük észre, hogy az üres memória mérete 638 KB-tal kevesebb, mint vártuk volna. A MINIX 3 betöltési felügyelőprogram igyekszik a lehető legmagasabb memóriacímre betölteni magát. Esetünkben 52 KB helyet igényel, így valójában 584 KB üres memória van. Itt célszerű megemlítenünk, hogy a memória sokkal összetettebb is lehet, mint ahogyan itt felvázoltuk. Például van egy MINIX-lemezt szimuláló MINIX-függvény, amely még nem lett átírva MINIX 3-ra és MS-DOS-fájlt használ. Ezért a MINIX 3 betöltési felügyelőprogramja előtt be kell töltenünk néhány MS-DOS-komponenst. Amennyiben ezek nem a jelenleg használt memória szomszédságában lennének betöltve, a betöltési felügyelőprogram több mint két üres régiót jelezne.

Amikor a betöltési felügyelőprogram betölti a betöltési memóriaképet a memóriába, kiírja a képernyőre a képhez tartozó komponensek adatait. A 4.44. ábrán látható egy ilyen képernyő egy része. Ebben a példában (tipikus, de lehetséges, hogy nem teljesen azonos azzal, ami az olvasó előtt jelenik meg, mivel ez a MINIX 3 egy kibocsátás előtti verzióján készült), a betöltési felügyelőprogram

cs	ds	kód	adat	bss	verem	
0000800	0005800	19552	3140	30076	0	kernel
0100000	0104c00	19456	2356	48612	1024	pm
0111800	011c400	43216	5912	6224364	2048	fs
070e000	070f400	4352	616	4696	131072	rs

4.44. ábra. *A betöltési felügyelőprogram által kiírt első néhány betöltési memóriakép komponens memóriahasználata*

betöltötte a kernelt a 0x800 címtől kezdődő üres memóriába. A PM, a fájlrendszer, a reinkarnációs szerver és más komponensek, amelyek itt nem jelennek meg, az 1 MB-tól kezdődő üresmemória-blokkba töltődtek be. Ez egy tervezői döntés volt, 588 KB alatt elegendő memória maradna még ezeknek a komponenseknek. Amikor azonban, mint a példánkban is, a MINIX 3-at nagy blokkgyorsítótárral fordítjuk, a fájlrendszer nem fér el a kernel fölötti térben. Egyszerűbb volt, de semmiképpen sem elemi fontosságú, mindent a magasabb memóriarégiókba tölteni. Ezzel semmit sem veszítettünk, futás közben a memóriakezelő a többi memóriarészhez hasonlóan felhasználhatja ezt az 588 KB alatti lyukat.

A PM inicializálása, a hamis riasztások elkerülése végett egy ciklussal kezdődik, amely során letiltja a processzustáblában található minden elem időzítőjét. Ezután a mellőzendő, illetve a szemetet tartalmazó memóriakiírásokat (core dump) okozó szignálokat definiáló globális változók lesznek inicializálva. A következő lépésben az előbb látott memóriahasználatot dolgozzuk fel. A 18182. sorban, mint az előzőkben szintén láttuk, a rendszertaszk megkapja a betöltési felügyelőprogram *memória* karakterláncát. A mi példánkban két bázis:méret pár van, amelyek az üres memória blokkjait mutatják. A *get_mem_chunks* hívás (18184. sor) átkonvertálja az ASCII karakterláncot binárissá, és beírja a bázis- és méretértékeket azt itt definiált elemekként:

```
struct memory {phys_clicks base; phys_clicks size;};
```

a *mem_chunks* tömbbe (18192. sor). A *mem_chunks* tömb még nem a lyuklista, ez csak egy olyan tömb, amelyben a lyuklista inicializálása előtt gyűjtjük az információt.

A kernel lekérdezése és a kernel memóriahasználat clickbe való konvertálása után a *patch_mem_chunks* függvényt hívjuk meg, hogy kivonja a kernel memóriahasználatát a *mem_chunks* tömbből. A memória, amely a MINIX 3 indítása előtt használva volt, most a kernel memóriahasználataként van elkönyvelve. A *mem_chunks* nem teljes, de a betöltési memóriaképben lévő normál processzusok által használt memória is el lesz könyvelve a 18201–18239. sorok közötti processzustábla-bejegyzéseket inicializáló ciklusban.

A betöltési memóriakép részét képező processzusok attribútumait a *kernel/table.c* fájlban (6095–6109. sor) deklarált *image* táblában találhatjuk. A főciklusba lépés előtt a sys_getimage kernelhívás a 18197. sorban egy másolatot biztosít az *image* tábláról a PM-nek. (Pontosabban ez nem igazi kernelhívás, hanem egy az *include/minix/syslib.h* fájlban definiált több tucat makró közül, amelyek

könnyen használható interfészeket biztosítanak a sys_getinfo kernelhívás felé.) A kernelprocesszusok nem láthatók a felhasználói térből, és a processzustábla PM- (valamint az FS-) részeit nem kell a kernelprocesszusokkal kapcsolatos információkkal inicializálni. Valójában nem foglalunk le helyet a kernelprocesszusbejegyzések számára. A kernelprocesszusok negatív azonosítóval rendelkeznek (programtáblaindex), és mellőzve vannak a 18202. sorban lévő tesztben. A kernelprocesszusok számára nem kell meghívniuk a *patch_mem_chunks* függvényt sem, mert a kernelmemória használatában a kernelbe fordított processzusok már bele vannak számítva.

A processzustáblához hozzá kell adnunk a rendszer- és a felhasználói processzusokat, amelyek más-más elbánásban részesülnek (18210–18219. sor). Az *init* az egyetlen felhasználói processzus, amely a rendszerindításkor betöltődik, így egy ellenőrzés történik az *INIT_PROC_NR* azonosítóra (18210. sor). Induláskor ezenkívül csak rendszerprocesszusok töltődnek be. A rendszerprocesszusok speciálisak: nem lehet őket cserélni, rendelkeznek egy dedikált elemmel a kernel *priv* táblájában, valamint speciális, az állapotjelzőikkel jelzett jogosultságaik vannak. A szignálkezeléshez minden processzus megfelelő alapértelmezett beállításokkal inicializálódik (van ugyan némi különbség a rendszerprocesszusok és az *init* között). Először lekérik minden processzus memóriatérképét a *get_mem_map* függvény segítségével, amely többek között a sys_getinfo rendszerhíváson alapul, majd a *patch_mem_chunks* hívás segítségével beállítják a *mem_chunks* tömböt (18225–18230. sor).

Végezetül egy üzenetet küldenek a fájlrendszernek, amely hatására minden processzusnak készül egy bejegyzés (18233–18236. sor) a processzustábla FS-hez tartozó részébe. Mivel a betöltési memóriakép minden processzusa a rendszergazdához tartozik, és azonos alapértelmezett értékekkel rendelkezik, ezért az üzenetnek elegendő a processzus PID-jét tartalmaznia az FS processzustábla elemeinek inicializálásához. Az üzenetek elküldéséhez a send műveletet használjuk, így nem várunk választ. Az üzenet elküldése után a processzus neve megjelenik a konzolon (18237. sor):

```
Building process table: pm fs rs tty memory log driver init
```

Ebben az esetben a képernyő meghajtója beépül az alapértelmezett lemezmeghajtóba. Több lemezmeghajtót is befordíthatunk a betöltési memóriaképbe, a *label=* indítóparaméterrel tudjuk az alapértelmezett meghajtót megadni.

A PM saját speciális processzustábla-bejegyzéssel rendelkezik. A főciklus befejeződése után a PM néhány módosítást hajt végre a saját bejegyzésén, majd elküld egy végső, *NONE* szimbolikus azonosítójú üzenetet a fájlrendszernek. Mivel ezt az üzenetet a sendrec hívás segítségével küldi el, a PM blokkolódik, amíg nem kap választ. Amíg a PM végigmegy az inicializáló kódon, addig a fájlrendszer a *receive* ciklust hajtja végre (24189–24202. sor, a kód a következő fejezetben lesz leírva). A *NONE* processzusazonosítójú üzenet tudatja a fájlrendszerrel, hogy minden rendszerprocesszus inicializálva lett, így ki tud lépni a ciklusból, és elküld (send) egy szinkronizáló üzenetet a PM-nek, hogy az ki tudjon lépni a blokkolt állapotából.

Ezután az FS már szabadon folytathatja saját inicializációját, és a PM inicializációja is majdnem befejeződött. A 18253. sorban meghívódik a *mem_init*, amely inicializálja a futó rendszermemória kezeléséhez szükséges, az üres memória régióit tartalmazó láncolt listát és a hozzá tartozó változókat a *mem_chunks*-ba összegyűjtött információk segítségével. A rendes memóriakezelés a MINIX 3 által használt és a szabad memóriát kiíró üzenet:

```
Physical memory: total 63996 KB, system 12834 KB, free 51162 KB
```

után indul.

A következő függvény a *get_nice_value* (18263. sor). A betöltési memóriakép processzusainak „udvariassági szintjé"-nek megállapítására kerül meghívásra. Az *image* tábla tartalmaz egy várakozásisor-értéket a betöltési memóriakép minden processzusához, amely alapján a processzus az ütemező megfelelő prioritású várakozási sorába kerül. Az értékek 0-tól, amely az olyan magas prioritású processzusokat jellemzi, mint a *CLOCK (órataszk),* 15-ig tartanak, amely az *IDLE*-t jellemzi. A hagyományos Unix-rendszerekben az „udvariasság szintje" pozitív és negatív értékeket is felvehet. Ezért a felhasználói processzusok számára a *get_nice_value* leképezi a kernel prioritás értékeket a 0 köré. Ehhez felhasználja az *include/sys/resource.h*-ban makróként definiált *PRIO_MIN, PRIO_MAX* –20 és +20 értékű konstansokat. Ezek a *kernel/proc.h*-ban definiált *MIN_USER_Q* és a *MAX_USER_Q* között skálázódnak, lehetővé téve a *nice* parancs használatát akkor is, amikor több vagy kevesebb várakozási sor alkalmazása mellett döntöttünk. A felhasználói szintű processzusok fájának gyökere az *init,* amely a 7-es prioritású sorban helyezkedik el, és egy 0 „udvariassági szint"-et kap, amelyet a gyermekei megörökölnek a fork után.

A *main.c* két utolsó függvényét már említettük. A *get_mem_chunks* csak egyszer kerül meghívásra (18280. sor). A betöltési felügyelőprogramtól ASCII karakterláncként visszakapott hexadecimális bázis:méret párokkal jellemzett memória az információt átalakítja memóriaszeletté és a *mem_chunks* tömbbe írja. A *patch_mem_chunks,* amely folytatja az üresmemória-lista építését, többször is meghívódik; egyszer a kernel hívja magának, majd az *init*-hez, ezután meghívja még minden, a *pm_init* főciklusba inicializált rendszerprocesszus is. Futása során kijavítja a nyers betöltési felügyelőprogram információkat. A munkája egyszerűbb, mivel az adatokat memóriaszelet-mértékegységben kapja meg. A *pm_init* megkapja minden egyes processzus kód- és adat-memóriafoglalásának báziscímét és méretét. Minden egyes processzus számára meg lesz növelve az üres blokkok tömbjében a legutolsó elem bázisa az adat- és a kódszegmensek együttes hosszával. Ezek után ennek a blokknak a mérete lesz lecsökkentve ugyanezzel az értékkel, hogy jelezzük a processzushoz tartozó memóriaterület foglaltságát.

4.8.3. A fork, az exit és a wait implementációja

A fork, az exit és a wait rendszerhívás a *forkexit.c* fájlban levő *do_fork*, *do_pm_exit* és *do_waitpid* eljárásokkal vannak megvalósítva. A *do_fork* eljárás (18430. sor) a 4.36. ábrán látható lépéseket követi. Megjegyzendő, hogy a *procs_in_use* (18445. sor) második hívása lefoglalja a rendszergazda számára a processzustábla utolsó néhány helyét. A gyermekprocesszus memóriaigényének kiszámításához az adat- és a veremszegmens közötti területet is be kell számítani, a kódszegmenst viszont nem kell figyelembe venni. Két lehetőség van: vagy megosztható a szülő kódszegmense, vagy a processzusnak egyesített adat- és kódrésze van, és ekkor kódszegmensének hossza nulla. A számítás elvégzése után az *alloc_mem* eljárással foglaljuk le a memóriát. Ha ez sikeres volt, akkor a gyermek és a szülő kezdőcímét memóriaszeletből abszolút bájtra konvertáljuk, és a sys_copy segítségével egy üzenetet küldünk a rendszertaszknak, hogy végezze el a másolást.

Ezután van egy üres hely a processzustáblában. A *proc_in_use*-ra vonatkozó előbbi teszt garantálja, hogy van ilyen. Miután megtaláltuk, kitöltjük: először a szülő bejegyzését másoljuk ide, majd módosítjuk az *mp_parent, mp_flags, mp_child_utime, mp_child_stime, mp_seg, mp_exitstatus* és *mp_sigstatus* mezőket. Ezek közül néhány mező különleges kezelést igényel. Az *mp_status* mezőnek csak néhány bitje örökölt. Az *mp_seg* tömb tartalmazza a kód-, adat- és veremszegmensek címét. Ha a processzus kódszegmense megosztható, akkor kód bejegyzése a szülő kódszegmensére mutat.

A következő lépés a processzusazonosító meghatározása a gyermek részére. A *get_free_pid* azt csinálja, amire a neve utal. A feladata nem olyan egyszerű, mint gondolnánk, ezért erre még visszatérünk.

A sys_fork és a *tell_fs* tudatja a kernellel és a fájlrendszerrel, hogy egy új processzus jött létre, és módosítani kell a tábláikat. (A *sys_* kezdetű könyvtári eljárások egy-egy üzenetet küldenek a kernelen belül futó rendszertaszknak a 2.45. ábrán látható szolgáltatások igénybevételére.) A processzusok létrehozását és megszüntetését mindig a PM kezdeményezi, és a befejezés után a folyamat a kernelben és a fájlrendszerben folytatódik.

A *do_fork* végén a gyermekprocesszusnak szóló üzenetet küldünk ki explicit módon. A gyermek azonosítóját tartalmazó válaszüzenetet a kérésre érkezett válaszként a *main* ciklusban küldjük el a szülőnek.

A PM által kezelt következő rendszerhívás az exit. Ezt a rendszerhívást a *do_pm_exit* eljárás fogadja (18509. sor), de a munka nagy részét átadja a néhány sorral lejjebb levő *pm_exit* eljárásnak. A munkamegosztást az indokolja, hogy a *pm_exit*-et akkor is meghívjuk, ha a processzus befejeződik egy szignál miatt. A feladat ugyanaz, csak a paraméterek mások, ezért kényelmes így szétvágni a dolgokat.

Amennyiben a processzus időzítőt futtat, akkor a *pm_exit* első dolga leállítani azt. A gyermek által használt idő ezután a szülő idejéhez adódik. Ezután értesíti a kernelt és a fájlrendszert, hogy a processzus nem fut többé (18550. és 18551. sor). A sys_exit rendszerhívás üzenetet küld a rendszertaszknak, hogy törölje a processzus által használt bejegyzést a kernel processzustáblájából. Ezután a memóriakezelő felszabadítja a processzusnak lefoglalt memóriát. A *find_share* függvény megkeresi, hogy a processzus megosztja-e másokkal a kódszegmensét, ha nem, akkor

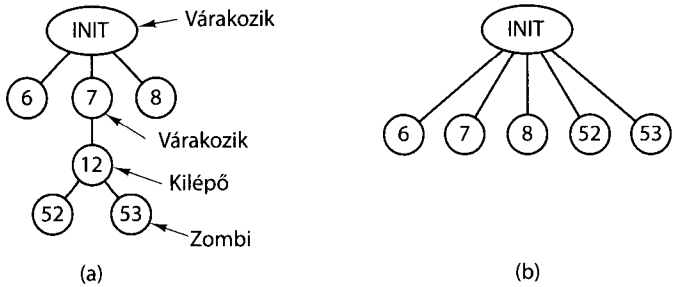
a *free_mem* eljárással felszabadítja a szegmenst. Ezt követi az adat- és a veremszegmens felszabadítása egyetlen művelettel, szintén a *free_mem* segítségével. Ha a processzus szülője várakozik, akkor a *cleanup* eljárás üríti ki a processzustábla bejegyzését. Ha a szülő nem várakozik, akkor a processzus zombi állapotba kerül, amit az *mp_flags ZOMBIE* bitje jelez, és egy *SIGCHILD* szignált küld a szülőnek.

Miután a processzus teljesen megszűnt vagy zombi lett, a *pm_exit* végignézi a processzustáblát (18582–18589. sor), és a processzus összes gyermekének átállítja a szülőbejegyzését az *init* processzusra. Ha az *init* várakozik, és a gyermekprocesszus már leállt, akkor erre a gyermekre is meghívja a *cleanup*-ot. Egy ilyen helyzetet láthatunk a 4.45.(a) ábrán. A 12-es számú processzus állt le, a szülője (7-es) várakozik rá. A 12-esre meghívjuk a *cleanup*-ot, így az 52-es és 53-as processzus az *init* gyermeke lesz; ezt a 4.45.(b) ábra mutatja. Ebben az állapotban az 53-as, amely már leállt, egy várakozást végrehajtó (wait) processzus gyermeke, így meg lehet szüntetni.

Amikor egy szülőprocesszus wait vagy waitpid hívást hajt végre, akkor a program futása a 18598. sorban levő *do_waitpid* eljárásban folytatódik. A két hívás a paraméterekben különbözik, így a végrehajtott tevékenység is más. A lokális változókat a 18613–18615. sorban állítjuk be, így a *do_waitpid* bármelyik hívást le tudja kezelni. A 18623–18642. sorban levő ciklus végignézi a processzustáblát gyermekprocesszusok után kutatva, és ha talál ilyet, akkor megvizsgálja, hogy zombi-e, mert akkor eltávolítható. Ha talált egy zombi gyermekprocesszust (18630. sor), akkor eltávolítja, és a *do_waitpid* visszatér a *SUSPEND* kóddal. Ha egy nyomkövetett gyermekprocesszust talál, akkor a legyártott válaszüzenetet módosítja, hogy jelezze a processzus leállását, majd a *do_waitpid* befejeződik.

Ha a wait-et hívó processzusnak nincs gyermeke, akkor az egy hibakódot kap vissza (18653. sor). Ha van gyermeke, de egyik sem zombi vagy nyomkövetett, akkor megvizsgáljuk, hogy meghívták-e a *do_waitpid*-et egy olyan jelzőbittel, amely jelzi, hogy a szülő nem akar tovább várakozni. Ha igen (ez a normális eset), akkor a 18648. sorban beállítunk egy jelzőbitet, jelezve, hogy várakozik, és a szülőt felfüggesztik, amíg egy gyermeke be nem fejeződik.

Amikor egy processzus befejeződik, és a szülője vár, akkor az események sorrendjétől függetlenül meghívjuk a *cleanup* (18660. sor) eljárást, és ez elvégzi a végső teendőket. A szülőt fel kell ébreszteni a wait vagy waitpid által okozott vára-



4.45. ábra. *(a) Az állapot, amikor a 12-es processzus befejeződik. (b) A befejeződés utáni állapot*

kozásból, vissza kell neki adni a befejeződött gyermekprocesszus azonosítóját és a kilépés-, valamint a szignálkódját. A fájlrendszer már felszabadította a gyermeknek lefoglalt memóriát, a kernel már nem ütemezi, és felszabadította a kernel processzustáblájában a processzus bejegyzését. Ennél a pontnál a gyermekprocesszus örökre eltűnt.

4.8.4. Az exec implementációja

Az exec forráskódja a 4.40. ábrán látható leírást követi; ez a *exec.c* fájlban lévő *do_exec* (18747. sor) eljárásban található. Néhány egyszerű ellenőrzés után a PM beolvassa a végrehajtandó fájl nevét a felhasználói szintből (18773–18776. sor). Emlékezzünk rá, hogy az exec-et megvalósító könyvtári eljárások a régi kernelképen belül hoznak létre egy vermet (lásd a 4.38. ábra). Ez a verem lesz behelyezve a PM memóriaterületére a következő lépésben (18782. sor).

A következő lépés ciklusként van megvalósítva (18789–18801. sor). Az egyszerű bináris futtatandó esetén csak egyszer megy végig a cikluson. Először ezzel az esettel foglalkozunk. A 18791. sorban egy üzenetet küld a fájlrendszernek, hogy váltson át a felhasználó könyvtárára, mert a megadott útvonal és fájlnév a felhasználó és nem a PM könyvtárához képest értelmezendő. Ezután az *allowed* lesz meghíva – és amennyiben engedélyezve van, megnyitja a fájlt. Ha a teszt nem sikerül, akkor az érvényes fájlleíró helyett egy negatív a végrehajtásszámmal tér vissza, és a *do_exit* befejeződik, jelezve a hibát. Ha a fájl létezik és futtatható, akkor a PM beolvassa a fájl fejlécét a *read_header* segítségével, és kiolvassa belőle a szegmensek méretét. Az egyszerű bináris fájlra a *read_header* hatására kilép a ciklusból a 18800. sorban.

A következőkben a futtatható szkript kezelését írjuk le. A MINIX 3, mint a legtöbb Unix-szerű rendszer, támogatja a futtatható szkripteket. A *read_header* ellenőrzi a fájl első két bájtját a mágikus szó (**shebang szekvencia**, #!) után keresve, és amennyiben megtalálta azt, akkor egy speciális kódot ad vissza, jelezve, hogy a fájl egy szkript. Az ilyen módon megjelölt első sor megadja a parancsértelmezőt és az esetleges speciális paramétereket, opciókkal is. Egy szkript például ilyen első sorral is kezdhető:

```
#! /bin/sh
```

így tudattuk, hogy a Bourne-parancsértelmező fogja értelmezni vagy:

```
#! /usr/local/bin/perl -wT
```

jelzi a Perl-értelmezőt és az esetleges hibák jelzését igénylő kapcsolókat. Ez azonban bonyolítja az exec dolgát. Amikor egy szkriptet szeretnénk futtatni, akkor a *do_exec*-nek nem a szkriptet, hanem az értelmezőt kell a memóriába töltenie. Amikor szkriptet kell kezelni, akkor a *patch_stack* lesz meghíva a ciklus alján lévő 18801. sorban.

A *patch_stack* munkája egy példával illusztrálható legjobban. Tegyük fel, hogy egy néhány argumentummal rendelkező Perl-szkriptet hajtunk végre a parancssorban:

```
perl_prog.pl file1 file2
```

Amennyiben a Perl-szkript a mágikus szót tartalmazó sorral kezdődik, akkor a *patch_stack* létrehoz egy olyan vermet a Perl bináris futtatására, mintha ezt írtuk volna be:

```
/usr/local/bin/perl -wT perl_prog.pl file1 file2
```

Ha ez sikeres volt, akkor a sor első részével, a parancsértelmező bináris útvonalával tér vissza. Ezután a ciklus törzsét hajtja végre még egyszer: beolvasva a fájl fejlécét és meghatározva a futtatandó fájlszegmens méreteit. A szkript első sorában megjelölt parancsértelmező nem lehet szkript. Ez az, amiért az *r* változót használjuk. A változót csak egyszer lehet növelni, ezzel egyre korlátoztuk a *patch_stack* hívásainak számát. Amennyiben a második alkalommal is szkriptet jelez a kód, akkor a 18800. sorban lévő ellenőrzés megszakítja a ciklust. A szkript kódja, amelyet szimbolikusan az *ESCRIPT* reprezentál, egy negatív szám (a 18741. sorban van definiálva). Ebben az esetben, a 18803. sorban lévő ellenőrzés hatására a *do_exit* egy hibakóddal fog visszatérni, amely jelzi, hogy a fájl nem futtatható, vagy pedig a parancssor túl hosszú.

Az exec munkájának befejezéséhez maradt még egy kis teendő. A *find_share* a futó processzusok közül keres olyat, amely megoszthatja az új processzussal a kódszegmensét (18809. sor), a *new_mem* pedig lefoglalja az új kép számára szükséges memóriát, valamint felszabadítja a régi memóriát. Az elindított program futásához a memóriában lévő képnek és processzustáblának is készen kell lennie. A 18819–18821. sorokban a futtatandó fájl i-csomópontja, fájlrendszere és módosítási ideje lesz elmentve a processzustáblába. A vermet ezután a 4.38.(c) ábrán látható módon előkészítik, és átmásolják a memóriába lévő új képbe. Ezután az *rw_seg* (18834–18841. sor) átmásolja a kód- (amennyiben nem megosztott kód) és adatszegmeneseket a merevlemezről a memóriába. Amennyiben a *setuid* vagy *setgid* bitek be vannak állítva, akkor értesíteni kell a fájlrendszert, hogy tegye bele az effektív azonosító információkat a processzustábla FS-hez tartozó részében lévő bejegyzésbe (18845–18852. sor). A fájltábla PM részébe egy az új program argumentumaira mutató mutatót mentünk el azért, hogy a *ps* parancs meg tudja ezt mutatni a parancssoron. A továbbiakban szignál bittérképeket állítunk be, illetve értesítjük az FS-t, hogy zárjon le minden olyan fájlleírót, amelyet le kell zárni exec után. Végezetül a parancs neve lesz elmentve azért, hogy a *ps* vagy pedig a hibakeresés közben ki lehessen írni a képernyőre (18856–18877. sor). A kernelt általában az utolsó lépésben értesítik, de ha a nyomkövetés engedélyezve van, akkor egy szignált már el kellett küldenünk (18878–18881. sor).

Bár a vezérlés összes lépése a *do_exec* eljárásban van, a munka részleteit végző eljárások az *exec.c*-ben találhatók. A *read_header* eljárás (18889. sor) nemcsak

a fájl fejlécét és a szegmensek méretét olvassa be, hanem ellenőrzi, hogy a fájl valóban egy MINIX 3 által futtatható fájl-e azon a processzoron, amelyre az operációs rendszer aktuális példányát fordították. A 18944. sorban lévő *A_I80386* konstans értéke az #ifdef...#endif szekvenciával van definiálva a fordításkor. A bináris futtatható programoknak ahhoz, hogy a 32 bites MINIX 3-on és Intel processzoron elfogadhatók legyenek a fejlécükben, tartalmazniuk kell ezt a konstanst. Amennyiben a MINIX 3-at 16 bites módban fordítanánk, akkor ez az érték *A_I8086* lenne. Amennyiben kíváncsiak vagyunk a lehetséges értékekre, támogatott processzorokra, az *include/a.out.h* fájlban találjuk ezek listáját.

A *new_mem* eljárás (18980. sor) ellenőrzi, hogy van-e elég memória az új processzus számára. Az adat- és a veremszegmensnek kell helyet keresni, ha a kódszegmens megosztható egy másik processzussal, vagy egy lyukat a kombinált kód-, adat- és a veremszegmensnek. Egy lehetséges javítás, hogy az utóbbi esetben két lyukat keresünk. A MINIX korai változataiban a kód-, az adat- és veremszegmenseknek foytonosnak kellett lenniük. Ez a MINIX 3-ban már nem szükséges. Ha talált megfelelő méretű lyukat, akkor felszabadítja a régi memóriát, és lefoglalja az újat. Ha nincs elég memória, akkor az exec rendszerhívás meghiúsul. A memóriafoglalás után a *new_mem* beállítja a memóriatérképet az *mp_seg* mezőben, és a sys_newmap kernelhívással értesíti a kernelt.

A *new_mem* fennmaradó része feltölti nullákkal a bss szegmenst, a hézagot és a vermet. (A bss az adatszegmensnek az a része, amely az inicializálatlan globális változókat tartalmazza.) A munkát a rendszertaszk hajtja végre a sys_memset segítségével (19064. sor). Sok fordítóprogram külön kódot generál, hogy a bss szegmenst lenullázza, de a MINIX 3 olyan fordítóprogramokkal is tud dolgozni, amelyek ezt nem teszik meg. Az adat- és a veremszegmens közötti hézagot azért töltjük fel nullákkal, hogy a brk rendszerhívással kiterjesztett adatszegmens új területe nullákat tartalmazzon. Ez nemcsak a programozónak kényelmes, hogy az új változók kezdőértéke nulla, hanem a többfelhasználós rendszerek biztonságát is növeli, mert az új processzus nem éri el az előző processzus által otthagyott adatokat.

A következő eljárás a *patch_ptr* (19074. sor), amely a veremben levő mutatók módosítását végzi el, a 4.38.(b) és (c) ábrán látható módon. A veremben minden mutatóhoz hozzáadja a báziscímet.

A következő két függvény szorosan együttműködik. Az előzőkben már leírtuk a feladatukat. Amikor egy szkriptet futtatunk, a szkript értelmező binárisa az, amelyet le kell futtatni. Az *insert_arg* (19106. sor.) karakterláncokat illeszt be a verem PM-nél található másolatába. Ezt a *patch_stack* vezérli, amely a szkript minden, a mágikus szót tartalmazó sorában lévő karakterláncát megtalálja, majd meghívja az *insert_arg* függvényt. A mutatókat természetesen ki kell javítani. Az *insert_arg* dolga egyszerű ugyan, de sok minden elromolhat, és ezeket tesztelni kell. Itt érdemes megemlíteni, hogy a lehetséges hibák lekezelése a szkriptek esetén különösen fontos. A szkripteket felhasználók írják, akik minden számítástechnikai szakember szerint a problémák legfőbb okai. Komolyra fordítva a szót, a legfontosabb különbség a szkript és a lefordított bináris között az, hogy a lefordított bináris esetén megbízhatunk a fordítóban, hogy sok hibát már a fordításkor kiszűr. A szkriptek nincsenek ilyen módon validálva.

A 4.46. ábrán látható az *f1* fájlon dolgozó *s.sh* szkript esetén az előzőkben leírt feldolgozás. A parancssor a következő is lehet:

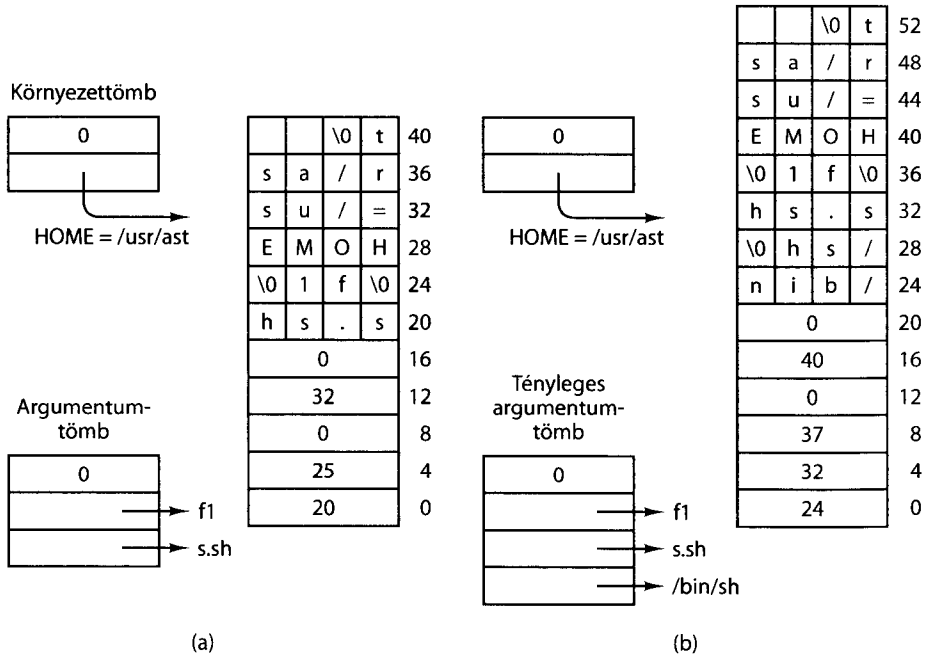
```
s.sh f1
```

a szkript mágikus szót tartalmazó sora, amely jelzi, hogy a Bourne-parancsértelmezőt kell hívni:

```
#! /bin/sh
```

A 4.46. ábra (a) részében láthatjuk a hívó memóriaterületéről átmásolt vermet. A (b) részben láthatjuk a *patch_stack* és az *insert_arg* segítségével végrehajtott transzformációkat. Mindkét diagram kapcsolódik a 4.38.(b) ábrához.

Az *exec.c*-ben definiált következő függvény a *rw_seg* (19208. sor). Az exec futtatása során egyszer vagy kétszer hívódik meg az adat-, valamint esetenként a kódszegmens betöltésére. Ahelyett, hogy blokkról blokkra olvasnánk a fájlt, és a beolvasott blokkot a célterületre másolnánk, a fájlrendszer az egész szegmenst közvetlenül a felhasználó területére olvassa be. Valójában ezt a hívást kissé különlegesen dolgozza fel a fájlrendszer, mert úgy tűnik, mintha a felhasználó processzusa olvasná be a szegmenst. Csak a fájlrendszer olvasási rutinjának első néhány



4.46. ábra. *(a) Az* execve*-nek átadott tömbök és a szkript futása közben létrehozott verem. (b) A* patch_stack *feldolgozása után a verem és a tömbök így néznek ki. A szkript nevét átadják a szkriptértelmező programnak*

sorából derül ki, hogy valami trükk következik. Ezáltal jelentősen gyorsul a szegmensek betöltése.

Az *exec.c* utolsó eljárása a *find_share* (19256. sor). Ez a futó processzusok közül keres a fájlok i-csomópont, eszköz és módosítási idő paramétereinek összehasonlításával egy olyat, ami megoszthatja az új processzussal a kódszegmensét. Ez egy egyszerű keresés az *mproc*-ban a megfelelő mezők alapján. A keresésből természetesen ki kell hagyni az új processzust.

4.8.5. A brk implementációja

Ahogy már láttuk, a MINIX 3 az alap-memóriamodellt használja: amikor létrehozunk egy processzust, akkor az adatai és a verme számára egy összefüggő memóriaterületet biztosítunk. A processzust soha nem helyezzük át a memóriában, és sohasem visszük ki a lemezre, a területének mérete nem változik. Csak annyi történhet, hogy az adatszegmens elvesz a hézag alsó, a verem pedig a felső részéből. Ezek miatt a brk implementációja a *break.c*-ben különösen egyszerű. Csak elvégzi az ellenőrzést, hogy az új méret legális-e, és módosítja a processzustáblát.

A rendszerhívást kezelő eljárás ugyan a *do_brk* (19328. sor), de a munka javát az *adjust* (19361. sor) végzi. Utóbbi ellenőrzi, hogy az adatszegmens összeütközik-e a veremmel. Ha igen, akkor a brk nem hajtható végre, de a processzust nem állítja le azonnal. Azért, hogy a processzus megállapíthassa (jó eséllyel) azt is, ha a verem túl nagyra nőtt, és ez esetben maradjon elég hely a veremnek egy rövid futásra, a vizsgálat előtt egy biztonsági faktort *(SAFETY_BYTES)* adunk az adatszegmens tetejéhez. Ez esetben a hívás után a processzus visszakapja a vezérlést (egy hibaüzenettel), így kiírhatja a megfelelő üzeneteket, és leállhat.

Megjegyzendő, hogy a *SAFETY_BYTES* és a *SAFETY_CLICKS* #define utasításokkal vannak definiálva az eljárás közepén (19393. sor). Ez szokatlan, mert az ilyen definíciók a fájl elején vagy a definíciós fájlokban szoktak lenni. A forráskód megjegyzéséből kiderül az ok: a programozónak nehéz volt meghatározni a biztonsági faktor méretét. Így hívta fel a figyelmet erre, és valószínűleg a kísérletezést is ösztönözni szerette volna.

Az adatszegmens kezdőcíme állandó, így amennyiben az *adjust*-nak meg kell változtatnia az adatszegmens méretét, csak a méretmezőt kell megváltoztatnia. A verem egy rögzített végponttól lefelé terjeszkedik, így ha az *adjust* észreveszi, hogy a veremmutató, amelyet paraméterként megkapott, a verem alsó határa alá ért, akkor módosítja a verem kezdetét és hosszát.

4.8.6. A szignálkezelés implementációja

A szignálokhoz a 4.47. ábrán felsorolt nyolc POSIX-rendszerhívás kapcsolódik. Ezeket a hívásokat, valamint magukat a szignálokat is a *signal.c* fájlban dolgozzuk fel.

A *sigaction* és a *signal* függvények a sigaction rendszerhívást használják; ezekkel a függvényekkel határozza meg egy processzus, hogyan válaszoljon az egyes

Rendszerhívás	A rendszerhívás szerepe
sigaction	Módosítja a szignálra adott választ
sigprocmask	Megváltoztatja a blokkolt szignálok halmazát
kill	Szignált küld egy másik processzusnak
alarm	ALRM szignált küld önmagának megadott idő múlva
pause	Felfüggeszti a processzus futását a szignál érkezéséig
sigsuspend	Megváltoztatja a blokkolt szignálok halmazát, majd PAUSE
sigpending	Megvizsgálja a függőben levő (blokkolt) szignálokat
sigreturn	A szignálkezelő után takarít

4.47. ábra. *A szignálokhoz kapcsolódó rendszerhívások*

szignálokra. A *sigaction* a POSIX szabvány része, legtöbb esetben ez az ajánlott hívás. A *signal* könyvtári függvényt pedig a szabványos C definiálja, és ezért használnunk kell az olyan programokban, amelyeket nem POSIX-rendszerekben is le akarunk fordítani. A *do_sigaction* (19544. sor) kódja egy ellenőrzéssel kezdődik, amely megvizsgálja, hogy létezik-e a megadott kódú szignál, és ellenőrzi azt is, hogy vajon a hívás nem kísérlet-e a sigkill szignálra adott válasz megváltoztatására (19550–19551. sor). (A sigkill szignált nem szabad figyelmen kívül hagyni, elkapni vagy blokkolni. Ez az utolsó lehetőség, amivel a felhasználók vezérelhetik a processzusaikat, és a rendszergazda felügyelheti a felhasználókat.) A *sigaction* egyik paramétere a *sig_osa* mutató, amely egy *sigaction* struktúrára mutat, ebben kapjuk meg a szignálhívás előtti attribútumait, az új attribútumokat pedig a másik paraméterben, a *sig_nsa* mutatóban kapjuk meg.

Első lépésként meghívjuk a rendszertaszkot, hogy másolja át az aktuális attribútumokat a *sig_osa* által mutatott struktúrába. A *sigaction*-t úgy is meg lehet hívni, hogy a *sig_nsa* mutató *NULL* értékű, ilyenkor csak lekérdezzük, de nem változtatjuk meg a szignál attribútumait. Ebben az esetben a *do_sigaction* azonnal visszatér (19560. sor). Ha a *sig_nsa* nem *NULL*, akkor a szignál új tevékenységét megadó struktúrát a PM területére írjuk.

A 19567–19585. sorok aszerint módosítják az *mp_catch*, az *mp_ignore* és az *mp_sigpending* bittérképeket, hogy a szignálhoz tartozó új akció a szignál figyelmen kívül hagyása, az alapértelmezett tevékenység vagy a szignál elkapása. A *sigaction* struktúra *sa_handler* mezőjét használjuk a szignál vagy a *SIG_IGN*, illetve a *SIG_DFL* speciális szignálok (ezeket értenünk kellene, amennyiben az eddig tárgyalt POSIX-jelkezelést megértettük) elkapásakor végrehajtandó eljárásra mutató pointer átadására. Egy speciális, a MINIX 3-ra jellemző *SIG_MESS* is beérkezhet; ezzel a továbbiakban még foglalkozunk.

Annak ellenére, hogy a műveletek egyszerű bitmanipulációk, amelyeket megvalósíthattunk volna egyszerű makrókban is, a *sigaddset* és a *sigdelset* könyvtári függvényeket használjuk a szignálok bittérképének kezelésére. Azért hogy könnyen hordozható programokat írhassunk még olyan rendszerekben is, ahol a szignálok száma több, mint az *integer* típus bitszáma; ezek a függvények szerepelnek a POSIX szabványban. A könyvtári függvények használata könnyebbé teszi a MINIX 3 átvitelét más architektúrákra.

Az előzőkben egy speciális esetet említettünk. A *SIG_MESS* kódot (19576. sor) csak speciális jogosultságú (rendszer-) processzusok használhatják. Normális esetben ezek a processzusok blokkolódnak a kérésüzenetekre várva. Így a hagyományos szignálfogadási eljárás, amely során a PM megkéri a kernelt, hogy egy szignálkeretet helyezzen el a vevő vermében, el lesz halasztva mindaddig, amíg egy üzenet fel nem ébreszti a fogadót. A *SIG_MESS* kód jelzi a PM-nek, hogy a normál üzeneteknél magasabb prioritású értesítő üzenetet kézbesítsen. Mivel az értesítő üzenet argumentuma tartalmazza a függő szignálok halmazát, segítségével több szignált is kézbesíthetünk egy üzenetben.

Végül kitöltjük a PM processzustáblájának szignálokhoz kapcsolódó többi mezőjét is. Minden szignálhoz van egy bittérkép, az *sa_mask,* amely megadja, hogy a szignálkezelő futása alatt milyen szignálokat kell blokkolni. Minden egyes szignálhoz tartozik még egy *sa_handler* nevű mutató; ez a szignálkezelő eljárás címét tartalmazza, vagy egy-egy különleges értéket annak a jelzésére, hogy a szignált figyelmen kívül hagyjuk, az alapértelmezett módon kezeljük vagy üzenetgenerálásra használjuk. A könyvtári eljárás címét, amely a szignálkezelő leállásakor a sigreturn-t meghívja, az *mp_sigreturn* változóban tároljuk. Ez a cím annak az üzenetnek az egyik mezője, amelyet a PM kap.

A POSIX megengedi a processzusoknak, hogy megváltoztassák a saját szignálkezelésüket, még a szignálkezelő eljáráson belül is. Ez arra jó, hogy a szignál feldolgozása közben megváltoztassuk a következő szignálra adott választ, majd később visszaállítsuk az eredeti állapotot. Az alábbi rendszerhívások ezeket a szignálmanipulációkat támogatják. A sigpending-et a *do_sigpending* (19579. sor) eljárás kezeli; ez az *mp_sigpending*-et adja vissza, így a processzusok ellenőrizhetik, hogy mely szignálok várakoznak. A sigprocmask-hoz a *do_sigprocmask* függvény tartozik; ez a blokkolt szignálok halmazát adja vissza, és lehetőséget nyújt, hogy megváltoztassuk egyetlen szignál blokkoltságát, vagy akár az egész halmazt helyettesítsük egy új értékkel. Ha a szignál nem blokkolt, akkor a *check_pending* meghívásával (19635. és 19641. sor) ellenőrizzük, hogy van-e várakozó szignál. A *do_sigsuspend* (19657. sor) kezeli a sigsuspend rendszerhívást. Ez a hívás felfüggeszti a processzust, amíg az egy szignált nem kap. A többi itt tárgyalt eljáráshoz hasonlóan ez is bittérképekkel dolgozik, az *mp_flags* sigsuspended bitjét állítja be, ezzel megelőzi a processzus végrehajtását. Itt újra meghívjuk a *check_pending*-et. Végül a *do_sigreturn* kezeli a sigreturn-t, ezt a felhasználói szignálkezelő befejezésénél használjuk. Visszaállítja azt az állapotot, ami a kezelő indulásánál érvényben volt, és meghívja a *check_pending*-et (19682. sor).

Amikor egy felhasználó processzus, mint a *kill* parancs, meghívja a kill rendszerhívást, a PM *do_kill* (19689. sor) függvénye lesz meghíva. A kill egyetlen meghívásával egy egész processzuscsoportnak lehet szignált küldeni, így a *do_kill* a *check_sig* eljárással alkalmas címzetteket keres a processzustáblában.

Néhány szignál, mint például a sigint, a kerneltől ered. A *ksig_pending* (19699. sor) akkor lesz aktiválva, amikor a kernel elküldi a függő szignálokat tartalmazó üzenetet a PM-nek. Több processzusnak is lehet függő szignálja, így a 19714–19722. sorok közötti ciklus ismételten lekéri a rendszertaszktól a függő szignálokat és átadja őket a *handle_sig* függvénynek, majd amikor már nincs több függő szignál, akkor ezt tudatja a rendszertaszkkal. Az üzenetek egy bittérképpel együtt érkeznek, lehetővé téve a kernelnek több szignál generálását egy üzenettel. A következő függvény a *handle_sig,* amely a bittérképet bitenként dolgozza fel (19750–19763. sor). Egyes kernelszignálok különleges figyelmet igényelnek: a processzus ID-t időnként megváltoztatják annak érdekében, hogy a szignált egy csoport kapja meg (19753–19757. sor). Egyébként minden beállított bit egy *check_sig* hívást eredményez, hasonlóan, mint a *do_kill* esetében.

Riasztások és időzítők

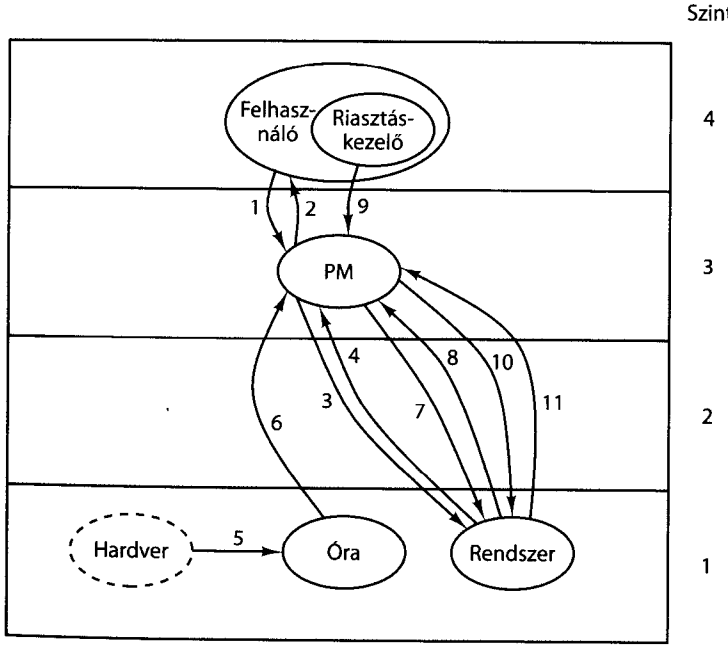
Az alarm rendszerhívást a *do_alarm* eljárás (19769. sor) kezeli. A *set_alarm* függvényt hívja meg, amely egy külön függvény, mivel arra is használják, hogy amikor van még aktív időzítővel rendelkező processzus, akkor kikapcsolja azt az időzítőt. Ezt a *set_alarm* hívás 0 riasztási idővel történő meghívásával valósítják meg. A *set_alarm* a processzuskezelőn belül kezelt időzítőket használja a munkája során. Először megállapítja, hogy van-e már időzítő a célprocesszus számára, ha van, akkor lejárt-e már, ha igen, akkor a rendszerhívás vagy az előző időzítésből maradt idővel tér vissza, vagy ha nem volt előző időzítés, akkor nullával. A kódok közötti megjegyzések rávilágítanak a hosszú időzítőkkel kapcsolatos néhány problémára. Egy igen csúnya kódrész (19918. sor) a hívás argumentumát, amely másodpercben van megadva, elosztja a másodpercenkénti óraimpulzusokat megadó *HZ* nevű konstanssal, így megkapjuk az időt impulzusszámban. Három típuskonverzió szükséges az eredmény *clock_t* adattípusúvá alakításához. A következő sorban pedig a számítás inverzét hajtjuk végre, az impulzusokat megadó *clock_t*-ből *unsigned long* típusúba lesznek az adatok konvertálva. Az eredmény az eredeti argumentum *unsigned long* típuskonverziójával lesz összehasonlítva. Amennyiben nem egyenlők, az azt jelenti, hogy a kért idő valamelyik adattípusnál a korlátokon kívül esett, és ekkor a „soha" érték lesz behelyettesítve. Végezetül vagy a *pm_set_timer,* vagy a *pm_cancel_timer* lesz meghíva, amelyek segítségével időzítőt adunk hozzá vagy veszünk el a processzuskezelő időzítősorából. Az első híváshoz tartozó legfontosabb argumentum a *cause_sigalrm,* amely az időzítő lejártakor meghívandó felügyelőfüggvényt specifikálja.

A *pm_XXX_timer* függvények elrejtik a kernelszinten kezelt időzítőkkel történt kommunikációt. Minden riasztási igény, amely végezetül riasztási kérelemmel végződik, végső soron a kernelszintbeli időzítőbeállítást fogja igényelni. Az egyetlen kivétel ez alól, ha egy időben több igény ugyanabban az időben lejáró időzítőt igényelne. A processzusok azonban lemondhatják vagy megszüntethetik a riasztásaikat. Kernelhívásra csak akkor van igény, ha a processzuskezelő időzítő várakozási sorának első elemét szeretnénk módosítani.

Amikor egy PM számára létrehozott időzítő a kernelszintű időzítő várakozási sorában lejár, a rendszertaszk értesíti erről a tényről a PM-et egy *SYN_ALARM* típusú üzenettel, amelyet a PM főciklusa detektál. Ennek hatására meghívódik a *pm_expire_timers,* amely végül meghívja a *cause_sigalrm* függvényt.

A *cause_sigalrm* (19835. sor) az előbb említett felügyelőfüggvény. Megkapja az értesítendő processzus számát, leellenőriz néhány állapotjelzőt, majd alapállapotba hozza az *ALARM_ON* állapotjelzőt, és kiküldi a *SIGALRM* szignált a *chek_sig* függvény meghívásával.

A *SIGALRM* szignálra az alapértelmezett tevékenység a processzus leállítása, amennyiben nem kapták el. A *SIGALRM* elkapására egy kezelőeljárást kell installálni a sigaction segítségével. A saját eseménykezelővel elkapott *SIGALRM* szignál esetén fellépő események teljes sorozata a 4.48. ábrán látható. Három üzenetsorozat van. Az (1) üzenettel a felhasználó egy a PM-nek küldött üzenettel alarm hívást kezdeményez. A PM ekkor létrehoz egy időzítőt, a processzusnak fenntartott időzítő várakozási sorban, és egy üzenettel nyugtázza ezt (2). Ezután egy darabig semmi sem történik. Amikor a kérésre létrehozott időzítő eléri a várakozási sor elejét, vagy mert az előtte lévő időzítők lejártak, vagy mert lemondták őket, egy új üzenetet küldenek (3) a rendszertaszknak, hogy hozzon létre egy új kernelszintű időzítőt a processzuskezelőnek. Amikor ez sikerült, akkor a rendszertaszk nyugtázza ezt egy üzenettel (4). Egy darabig ismét semmi sem történik. Amint az időzítő eléri a kernelidőzítő sorának elejét, az óra-megszakításkezelő detektálja, hogy az lejárt. A szekvencia többi üzenete már gyorsan követi egymást. Az óra-



4.48. ábra. *Üzenetek a riasztásnál. A legfontosabbak: (1) A felhasználó hívja az* alarm*-ot. (3) A PM megkéri a rendszertaszkot az időzítő beállítására. (6) Az óra szól a PM-nek, hogy lejárt az idő. (7) PM kéri a szignál küldését a felhasználóhoz. (9) A kezelő befejeződik a* sigreturn *meghívásával (részletek lásd a szöveg)*

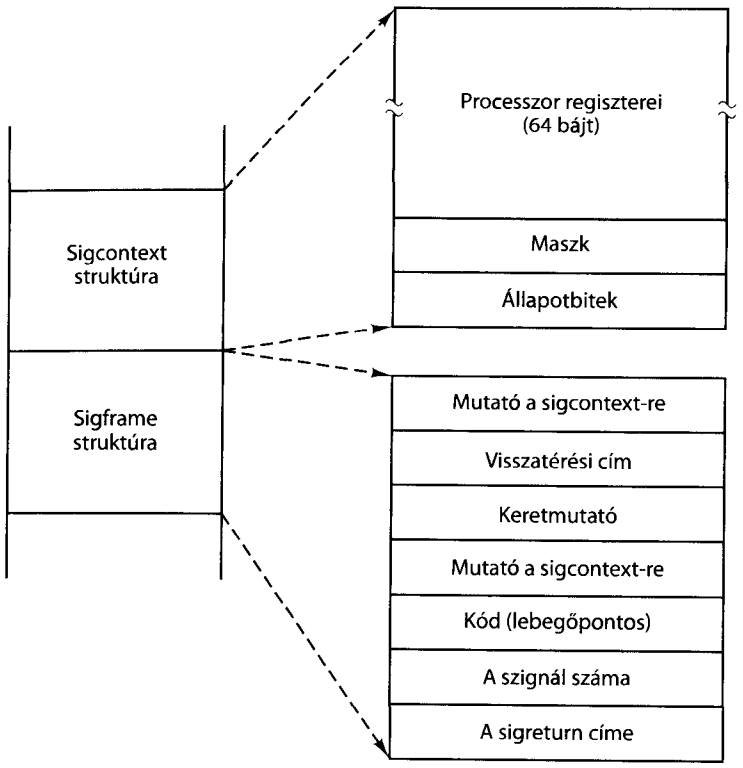
megszakításkezelő a *HARD_INT* üzenetet (5) küldi el az időzítőtaszknak, amely ennek hatására frissíti az időzítőit. A *cause_alarm* időzítő felügyelőfüggvény egy értesítést (6) küld a PM-nek. A PM ekkor frissíti az időzítőit, és a processzustábla hozzá tartozó részéből megállapítja, hogy a *SIGALRM* szignálhoz van-e regisztrált kezelője a kérdéses processzusnak, majd elküld egy üzenetet (7) a rendszertaszknak, hogy végezze el a szignálküldéshez szükséges veremmanipulációkat. Ezt a (8) üzenettel nyugtázza. Ezután a felhasználói processzust ütemezik, hogy végre tudja hajtani a kezelőjét, amelyben meghívja a sigreturn (9) hívást, amely visszatér a PM-hez. Ezután a processzuskezelő elküld egy üzenetet (10) a rendszertaszknak a takarítás inicializálására, amit az nyugtáz (11). Az ábrán nincs jelölve egy másik üzenetpár (3), amely a PM-ből megy a rendszertaszkhoz a rendszer eddigi futási idejének lekérdezésére.

A következő a pause rendszerhívás, amelyet a *do_pause* eljárás (19835. sor) kezel. Annak ellenére, hogy segítségével a riasztásig (vagy más szignálig) felfüggeszthetünk egy processzust, nincs igazából közeli kapcsolata a riasztásokkal és időzítőkkel. Csak egy bitet kell beállítani és a *SUSPEND* kódot kell visszaküldeni, amely visszatartja a PM főciklusát a válaszküldéstől, így blokkoljuk a hívó processzust. Még a kernelt sem kell értesíteni, mert az tudja, hogy a processzus blokkolt.

Szignálokat támogató függvények

Korábban a *signal.c* számos segédfüggvényét említettük. Most nézzük meg ezeket részletesebben. A legfontosabb a *sig_proc* (19864. sor), amely ténylegesen küldi a szignálokat. Először elvégez egy sor vizsgálatot. Szignálküldést megkísérelni megszűnt vagy zombi (19889–19893. sor) processzusoknak olyan komoly probléma, ami rendszerösszeomlást okozhat. Egy éppen nyomon követett processzus leáll, ha szignált kap (19894–19899. sor). Ha a címzett processzus nem veszi figyelembe a szignált, akkor a *sig_proc* a 19902. sorban befejezi a munkáját. Néhány szignálra ez az alapértelmezett tevékenység, például a POSIX szabványú, nem kötelezően támogatandó, a MINIX 3-ban nem támogatott szignálokra. Ha a szignál blokkolt, akkor csak egy bitet állítunk be a processzus *mp_sending* bittérképében. A legfontosabb teszt (19910. sor) segítségével megkülönbözteti azokat a processzusokat, amelyek elkaphatják a szignálokat azoktól, amelyek nem kaphatják el. Azoknak a szignáloknak a kivételével, amelyeket a rendszerszolgáltatásoknak küldendő üzenetté konvertáltak, minden egyéb szempontot kiiktattunk eddig, és ha a processzus nem kapja el a szignált, akkor le kell állítanunk.

Először az elkapható szignálok feldolgozását nézzük át (19911–19950. sor). A kernelnek küldendő üzenet egyes részei a PM processzustáblájának néhány részletét tartalmazzák. Ha a szignállal megcélzott processzust korábban felfüggesztettük a sigsuspend-del, akkor a felfüggesztés idején elmentett szignálmaszk kerül az üzenetbe, egyébként az aktuális szignálmaszk (19914. sor). Az üzenetben elküldött többi adat néhány cím a processzus címteréből: az aktuális veremmutató, a szignálkezelő címe, a *sigreturn* könyvtári eljárás címe, amelyet a szignálkezelő befejeződésekor kell meghívni.



4.49. ábra. *A szignálkezelő végrehajtásának előkészítése során a verembe tett* sigcontext *és* sigframe *adatszerkezetek. A processzor regiszterei a környezetváltáskor felhasznált veremkeret másolatai*

Ezután helyet foglalunk a processzus vermében; a verembe rakott struktúrát a 4.49. ábra mutatja. Mivel a processzustáblában a megfelelő struktúrák megváltoznak, a szignálkezelő végrehajtása alatt a *sigcontext* részt is a verembe tesszük a mostani állapot elmentése érdekében, így ezt később vissza tudjuk állítani. A *sigframe* részben van a visszatérési cím és az adatok a sigreturn-hoz, amelyekkel vissza lehet állítani a processzus állapotát, amikor a szignálkezelő befejeződött. A visszatérési címet és a keret címét jelenleg még nem használja a MINIX 3, azért vannak, hogy működjön a nyomkövető, ha valaki a szignálkezelő eljárást akarja vizsgálni.

A struktúra, amelyet a processzus vermébe rakunk, eléggé nagy. A 19923. és a 19924. sorban levő kód foglalja le a helyet a veremben, ezt követi az *adjust* meghívása, hogy ellenőrizzük, van-e elég hely a veremben. Ha nincs elég hely, akkor a C-ben ritkán használt goto utasítással a *doterminate* címkére ugrunk, és leállítjuk a processzust (19926–19927. sor).

Probléma merülhet fel az *adjust* meghívásánál. A brk implementációjánál már láthattuk, hogy az *adjust* hibakóddal tér vissza, ha a verem *SAFETY_BYTES* bájtnyira megközelíti az adatszegmenst. Azért kell ilyen tűréshatárról gondoskodni,

mert a verem ellenőrzése csak alkalmanként és csak szoftveresen történhet. Ez a tűréshatár jelenleg valószínűleg túlzott, mert pontosan tudjuk, hogy mennyi hely kell a veremben a szignálnak, és mennyi többlethely kell a szignálkezelőnek, amely feltételezhetően egy viszonylag egyszerű függvény. Lehetséges, hogy néhány processzus szükségtelenül fejeződik be, mert az *adjust* sikertelen. Ez biztosan jobb, mintha a processzus időnként rejtélyesen hibázna, de talán valamikor a jövőben ezeket az ellenőrzéseket precízebben is beállíthatjuk.

Ha a veremben van elegendő hely a struktúrának, akkor két további állapotjelzőt vizsgálunk. Az *SA_NODEFER* bit jelzi, hogy a szignál feldolgozása közben blokkolni kell-e a többi ugyanilyen típusú szignált. Az *SA_RESETHAND* jelzi, hogy a szignálkezelőt alapállapotba kell-e állítani a szignál vételekor. (Ez a régi *signal* hívás pontos szimulálására szolgál. Bár ezt a „lehetőséget" gyakran a régi hívás hibájának tekintették, a régi tulajdonságok támogatása az ilyen hibák támogatását is megköveteli.) Ezután a sys_sigsend (19940. sor) kernelhívás segítségével értesítik a kernelt, hogy tegye a sigframe-t a verembe. Végül a szignál várakozását jelző bitet nullázzuk, majd az *unpause* eljárással befejezzük az összes olyan rendszerhívást, amely a processzust felfüggesztette, ha volt egyáltalán ilyen. Amikor a processzus legközelebb végrehajtásra kerül, akkor a szignálkezelő eljárás fog lefutni. Amennyiben valamiért az összes eddigi említett vizsgálat sikertelen, a PM összeomlik (19949. sor).

Az előbbiekben kivételként említett rendszerszolgáltatásokhoz tartozó üzenetekké konvertált szignálok a 19951. sorban vannak vizsgálva, és a hatásukra meghívódik a sys_kill kernelhívás. Ennek hatására a rendszertaszk értesítő üzenetet küld a szignált fogadó processzusnak. Emlékezzünk arra, hogy a többi értesítéssel ellentétben a rendszertaszk által küldött értesítés szokásos tartalmán kívül származásáról is hordoz információt, valamint egy időbélyeget is tartalmaz. Ezek mellett átadja a szignálok bittérképét is, amelyek segítségével a processzus megtudhatja a függőben lévő szignálok adatait. Amennyiben a sys_kill sikertelen, akkor a PM összeomlik. Siker esetén a *sig_proc* tér vissza (19954. sor). Amennyiben a 19951. sor ellenőrzése sikertelen, a futtatás átugrik a *doterminate* címkéhez.

Most nézzük meg a *doterminate*-tel címkézett kódot, amely leállítja a processzust (19957. sor). A címke és a goto a legkönnyebb módja annak, hogy az *adjust* esetleges sikertelenségét kezeljük. Itt olyan szignálokat dolgozunk fel, amelyeket valamilyen okból nem tudtak vagy nem volt szabad fogadni. Lehetséges, hogy a szignál egy mellőzendő szignál volt, ekkor a *sig_proc* visszatér. Egyébként a processzust le kell állítani. A kérdés csak az, hogy a processzus memóriaképére szükség van-e a hibakereséshez? Végezetül a processzus úgy fejeződik be, mintha a *pm_exit* függvényt hívták volna meg (19967. sor).

A *check_sig* eljárással (19973. sor) ellenőrzi a PM, hogy a szignál elküldhető-e. A

```
kill(0, sig);
```

a hívó csoportjába tartozó minden processzusnak (azaz amelyeket ugyanarról a terminálról indítottak) elküldi a megadott szignált. A kerneltől származó szignálok és a reboot rendszerhívás több processzusra is hatással lehet. A fenti okok miatt a *check_sig* végignézi egy ciklussal (19996–20026. sor) a processzustáblát,

hogy megtalálja azokat a processzusokat, amelyeknek el kell küldeni a szignált. A ciklusban számos feltételt vizsgálunk, ha mind teljesül, akkor a *sig_proc* eljárással elküldjük a szignált a processzusnak (20023. sor).

Ahogy már többször láttuk, a *check_pending* eljárást (20036. sor) a forráskódban sok helyről meghívjuk. A *do_sigmask,* a *do_sigreturn* vagy a *do_sigsuspend* eljárás által kezelt processzus *mp_sigpending* bittérképének bitjeit *(do_sigmask, do_sigreturn, do_sigpending)* vizsgálja meg sorban, hogy egy eddig blokkolt szignál blokkolatlan lett-e. Ha talál egy várakozó és blokkolatlan szignált, akkor azt a *sig_proc*-cal elküldi. Mivel minden szignál kezelésénél végrehajtódik a *sig_return,* az összes várakozó, nem blokkolt szignált el tudjuk küldeni.

Az *unpause* eljárás (20065. sor) az olyan processzusoknak küldött szignálokkal foglalkozik, amelyeket pause, wait, read, write vagy sigsuspend hívás miatt felfüggesztettek. A PM processzustáblája alapján vizsgálja a pause-t, a wait-ct és a sigsuspend-et, és ha nem talált ezek miatt felfüggesztett processzust, akkor a fájlrendszerhez fordul, és az a saját *do_unpause* eljárásával megvizsgálja, hogy van-e várakozás írás vagy olvasás miatt. A művelet minden esetben ugyanaz: egy hibakódot küld válaszként a várakozó processzusnak, és beállít egy bitet, amitől a processzus végrehajtása folytatódhat, és feldolgozhatja a szignált.

Az utolsó eljárás a fájlban a *dump_core* (20093. sor), amely a processzus memóriaképét írja fájlba. A fájl fejléce tartalmazza a szegmensek méretét, valamint a processzus állapotát és a processzustáblából a processzusra vonatkozó információk másolatát. A fejléc után a szegmensek memóriaképe következik. Egy hibakereső program értelmezni tudja ezeket az információkat, és segítséget nyújthat a programozónak a hibát okozó utasítás meghatározásában.

A fájlba írás forráskódja egyszerű. Az előző alfejezetben említett probléma itt is fellép, de más formában. Ha meggyőződünk arról, hogy a kiírt veremszegmens megfelelő méretű, a 20120. sorban meghívjuk az *adjust* eljárást, ami a biztonsági határ miatt sikertelen lehet. A *dump_core* nem ellenőrzi az eredményt, mindenképpen kiírja a memóriaképet, de rossz lehet a fájlban a verem állapota.

Az időzítőket támogató függvények

A MINIX 3 processzuskezelője kezeli az olyan felhasználói processzusok által létrehozott riasztási kérelmeket, amelyek számára nem engedélyezett a közvetlen kapcsolat a kernellel vagy a rendszertaszkkal. Ez az interfész elrejti az időzítőtaszknál beállított riasztási időzítés minden részletét. Csak a rendszerprocesszusok állíthatnak be időzítést a kernelnél. Az ehhez szükséges támogatás a *timers.c* (20200. sor) fájlban található.

A processzuskezelő a riasztási kérelmeket listába rendezi, és megkéri a rendszertaszkot, hogy értesítse, amikor egy-egy riasztási idő elérkezett. Amikor riasztás érkezik a kerneltől, a PM továbbadja ezt a célprocesszusnak.

Három időzítőket támogató függvény van. A *pm_set_timers* beállít egy időzítőt, és hozzáadja a PM időzítőlistájához, a *pm_expire_timer* ellenőrzi a lejárt időzítőket, a *pm_cancel_timer* eltávolítja az időzítőt a PM időzítőlistájából. Mindhárom

felhasználja az időzítők könyvtárban *include/timers.h* deklarált függvényeket. A *pm_settimer* a *tmrs_settimer*-t, a *pm_expire_timer* a *tmrs_expire*-t, míg a *pm_cancel_timer* a *tmrs_clrtimers*-t hívja meg. Mindannyian a láncolt listák kezelését végzik, igény szerint: mozgás a láncolt listában, beszúrás, törlés. A rendszertaszkot csak az időzítő várakozási sor első elemének beszúrásakor és törlésekor kell meghívni azért, hogy módosítsa a kernelszintű időzítő várakozási sort. Ilyenkor a *pm_XXX_timer* függvények a sys_setalarm kernelhívást használják a kernelszintű segítség igénybevételére.

4.8.7. A többi rendszerhívás implementációja

A processzuskezelő a *time.c*-ben található három idővel kapcsolatos rendszerhívása: time, stime, times. Ezeket foglaltuk össze a 4.50. ábrán.

A valós idejű órát a kernelen belüli időzítőtaszk kezeli, amely nem vált üzeneteket mással, csak a rendszertaszkkal. Így a valós időt csak a rendszertaszkon keresztül tudjuk beállítani vagy lekérdezni. Ezt a feladatot a *do_time* (20320. sor) és a *do_stime* (20341. sor) látja el. A valós idő az 1970. január elseje óta eltelt, másodpercben mért időt jelenti.

A processzusok naplózási információit a kernel kezeli külön-külön. Minden óraütésre megváltoztatja egyes processzusok könyvelt idejét. A kernel nem tud a szülő-gyermek viszonyról, így a processzuskezelőre hárul a gyermekprocesszusok idejének összegzése. Amikor egy gyermek kilép, akkor az ideje hozzáadódik a szülő könyvelt idejéhez a processzustábla PM által kezelt részében. A *do_times* a sys_times kernelhívással lekéri a szülő könyvelt idejét a rendszertaszktól, majd visszaadja a válaszüzenetben a gyermek által eddig felhasznált rendszer- és felhasználói időt.

A *getset.c* fájl csak egy eljárást tartalmaz, a *do_getset*-et (20415. sor), amely a PM hét fennmaradó, POSIX által előírt rendszerhívását kezeli. Ezeket a 4.51. ábrán soroltuk fel. Olyan egyszerűek, hogy nem kell mindegyikhez külön eljárás. A getuid és a getgid egyaránt visszaadja a valódi és az effektív felhasználói és csoportazonosítót.

Az azonosítók beállítása egy kicsit bonyolultabb, mint kiolvasásuk. Ellenőrizni kell, hogy a hívó processzus jogosult-e az azonosítók beállítására. Ha az ellenőrzés sikeres, akkor a fájlrendszerrel tudatjuk az új azonosítót, mert a fájlok elérési joga is ettől függ. A setsid egy új viszonyt hoz létre; ezt nem tehetik meg az olyan processzusok, amelyek már csoportvezetők. Ezt a 20463. sorban ellenőrizzük. A mun-

Hívás	Feladat
time	Lekéri a valós idejű időt és az ütemezés idejét
stime	Beállítja a valós idejű órát
times	Lekéri a processzusok elkönyvelt idejét

4.50. ábra. *A három idővel foglalkozó rendszerhívás*

Rendszerhívás	Leírás
getuid	Visszaadja a valódi és a tényleges felhasználói azonosítót
getgid	Visszaadja a valódi és a tényleges csoportazonosítót
getpid	Visszaadja a processzus és szülője azonosítóját
setuid	Beállítja a hívó valódi és tényleges felhasználói azonosítóját
setgid	Beállítja a hívó valódi és tényleges csoportazonosítóját
setsid	Létrehoz egy új feladatcsoportot, az azonosítót adja vissza
getpgrp	A processzuscsoport azonosítóját adja vissza

4.51. ábra. *A* servers/pm/getset.c *rendszerhívásai*

kát a fájlrendszer fejezi be azzal, hogy vezérlőterminál nélküli viszonyvezetővé teszi a processzust.

Az eddig említett rendszerhívásokkal ellentétben, a *misc.c*-ben definiált rendszerhívások a POSIX szabványok szerint nem szükségesek. Ezekre a felhasználói szintű eszközmeghajtóknak és a MINIX 3 szervereinek a kernellel történő kommunikációjához van szükség; erre a monolitikus operációs rendszerekben nincs feltétlenül szükség. A 4.52. ábra felsorolja ezeket a rendszerhívásokat és feladatukat.

Az első kettőt a PM teljes egészében kezeli. A *do_allocmem* kiolvassa a kérést a beérkezett üzenetből, átalakítja memóriaszeletté, és meghívja az *alloc_mem* függvényt. Ezt használja például a memóriameghajtó, amikor memóriát foglal a RAM-lemeznek. A *do_freemem* hasonló, csak a *free_mem* függvényt hívja meg.

A következő hívásoknak már segítségül kell hívniuk a rendszer más részeit is. Úgy gondolhatunk rájuk, mint interfészekre a rendszertaszk felé. A *do_getsysinfo* az üzenetben átadott kéréstől függően több dolgot is tehet. Meghívhatja a rendszertaszkot, hogy lekérje a kernelről szóló *kinfo* struktúrában (az *include/minix/type.h*-ban van definiálva) tárolt információkat. Arra is használhatjuk, hogy megadja a processzustábla PM-hez tartozó részének címét, vagy lemásolja az egész processzustáblát egy másik processzusnak. A legutolsó művelet a *sys_datacopy* (20582. sor) hívással zárul. A *do_getprocnr* egyedül is képes a PID segítségével megtalálni a megfelelő sort a processzustábla hozzá tartozó részében, vagy segítségül hívja a rendszertaszkot, amennyiben név alapján kell keresnie.

A következő két hívás ugyan nem szükséges a POSIX szerint, de ettől függetlenül valamilyen formában jó eséllyel a legtöbb Unix-rendszerben megtalálhatók. A

Rendszerhívás	Leírás
do_allocmem	Lefoglal egy memóriarészt
do_freemem	Felszabadít egy memóriaterületet
do_getsysinfo	Lekéri a PM-mel kapcsolatos információkat a kerneltől
do_getprocnr	Visszaadja a processzustáblában a sorszámot PID vagy név alapján
do_reboot	Minden processzust leállít, értesíti az FS-t és a kernelt
do_getsetpriority	Beállítja vagy lekérdezi a rendszerprioritást
do_svrctl	Egy processzusból szervert csinál

4.52. ábra. *Speciális MINIX 3-rendszerhívások a* servers/pm/misc.c*-ben*

do_reboot elküldi a *KILL* szignált minden processzusnak, és szól a fájrendszernek, hogy készüljön fel az újraindításra. Csak miután a fájlrendszernek sikerült szinkronizálnia, azután lesz értesítve a kernel a *sys_abort* hívással (20667. sor). Mivel újraindítás lehet egy rendszerösszeomlás eredménye, de a rendszergazda is utasíthatja a rendszert leállásra, a kernelnek tudnia kell, hogy melyikről van szó. A *do_getsetpriority* támogatja a híres „udvarias" Unix-lehetőséget, amely segítségével egy felhasználó csökkentheti egy processzus prioritását, hogy jó szomszédja legyen a többi processzusnak (valószínűleg azok is sajátjai). A MINIX 3-ban ennél többről van szó: a rendszerkomponensek közötti relatív prioritás finomhangolására használják. Egy hálózati vagy lemezmeghajtó, amely gyors adatfolyamokat kezel, nagyobb prioritást kaphat, mint a lassú adatfolyamokat kezelő meghajtók, például a billentyűzet. Egy ciklusba ragadt, nagy prioritású processzus prioritását, amely ezáltal más processzusokat akadályoz, viszont érdemes ideiglenesen csökkenteni. A prioritás változtatása a második fejezetben leírtak szerint, a processzus alacsony vagy magas prioritású várakozási sorban történő ütemezésével valósul meg. Amikor ezt a kernelben lévő ütemező kezdeményezi, akkor nincs szükség a PM bevonására, ezzel szemben a felhasználói processzusoknak rendszerhívásokat kell használniuk. A PM szintjén csak az üzenetből kell kiolvasni az értéket, illetve új üzenetet kell generálni az új értékkel. A *sys_nice* kernelhívás elküldi az új értéket a rendszertaszknak.

A *misc.c* utolsó függvénye a *do_svrctl*. Jelenleg a csere engedélyezése vagy tiltása a feladata. A többi függvény, amelyeket ez a hívás valaha kiszolgált, valószínűleg a reinkarnációs szerverben lesz megvalósítva.

Az utolsónak megvizsgált rendszerhívás a *trace.c*-ben kezelt ptrace. A fájl megtalálható a mellékelt CD-ROM-on és a MINIX 3 weboldalán is. A ptrace-t a hibakereső programok használják. Ennek a rendszerhívásnak a 4.53. ábrán látható tizenegy parancsot adhatjuk át paraméterként. A PM-ben a *do_trace* ezek közül négyet dolgoz fel: *T_OK, T_RESUME, T_EXIT, T_STEP*. A nyomkövetés engedélyezését és befejezését befolyásoló kéréseket itt kezelik. A többi parancsot a *sys_trace* könyvtári eljárással átadják a rendszertaszknak, amely a processzustáb-

Parancs	Leírás
T_STOP	Megállítja a processzust
T_OK	Engedélyezi, hogy a szülő nyomon követhesse a processzust
T_GETINS	Értéket ad vissza a kódrészből
T_GETDATA	Értéket ad vissza az adatrészből
T_GETUSER	Értéket ad vissza a processzustáblából
T_SETINS	Beállít egy értéket a kódrészben
T_SETDATA	Beállít egy értéket az adatrészben
T_SETUSER	Beállít egy értéket a processzustáblában
T_RESUME	Folytatja a végrehajtást
T_EXIT	Kilépés
T_STEP	Beállítja a nyomkövetési bitet

4.53. ábra. A servers/pm/trace.c *nyomkövető parancsai*

la kernelhez tartozó részéhez is hozzáférhet. A nyomkövetéshez két segédeljárás található. A *find_proc* megkeresi a processzustáblában a vizsgált processzust. A *stop_proc* pedig, amikor erre jelzést kap, leállítja a nyomkövetett processzust.

4.8.8. A memóriakezelés segédeljárásai

A fejezetet két fájl rövid ismertetésével zárjuk, amelyek a processzuskezelő számára biztosítanak segédfüggvényeket. Ezek az *alloc.c* és a *utility.c*. Mivel a fájlok belső részleteiről itt nem beszélünk, de a CD mellékleten és a MINIX 3 weboldalán természetesen megtalálhatók.

Az *alloc.c* fájl az, ahol a rendszer nyilvántartja, hogy melyek a szabad, illetve a foglalt memóriarészek. Három belépési pontja van:

1. *alloc_mem* – lefoglal egy megadott méretű memóriablokkot;
2. *free_mem* – visszaad egy memóriablokkot, ami többé már nem kell;
3. *mem_init* – a PM indulásakor inicializálja a szabadlistát.

Ahogy azt már korábban is említettük, az *alloc_mem* a kezdőcím szerint rendezett lyuklistában first fit algoritmussal keres. Ha egy olyan lyukat talál, amely nagyobb a megadott méretnél, akkor szétvágja, és a maradékot meghagyja a szabadlistában, csak a megadott mennyiségű memóriát foglalja le. Ha az egész lyuk kell, akkor a *del_slot* eljárással kiveszi a listából.

A *free_mem* ellenőrzi, hogy a felszabadított terület összeolvasztható-e a szomszédos lyukakkal. Ha igen, akkor a *merge* eljárás segítségével összefogja a lyukakat és módosítja a listát.

A *mem_init* felépíti a kezdeti szabadlistát, amely az egész elérhető memóriát tartalmazza.

Az utoljára megvizsgált fájl a *utility.c*, amely a PM különböző részein használt különféle eljárásokat tartalmazza.

A *get_free_pid* üres PID-et keres a gyermekprocesszusok számára. Igyekszik elkerülni a számításba vehető problémákat. A maximális PID-érték 30000. Használhattunk volna *PID_t*-be férő maximális PID-értéket is, de a régi, kisebb típusokat használó programokkal kapcsolatos problémák elkerülése végett választottuk a 30000-es értéket. Miután egy hosszú életű processzusnak például a 20-as PID-értéket adtuk, 30000 új processzust tudunk indítani és leállítani a PID-számláló egyszerű növelésével, a maximális érték elérése után 0-val kell újrakezdeni, egész 20-ig lehet számlálni. Egy még használt PID újra kiosztása tragédiát okozna (tegyük fel, hogy valaki a 20-as processzuson szeretne szignált küldeni). Egy változó tartalmazza az utoljára kiosztott PID-értéket, és amennyiben meghalad egy maximális értéket, egy új indítás következik PID 2-vel (mivel az *init* PID-je mindig 1), az egész processzustáblát végigkeresi, hogy megbizonyosodjon arról, hogy a kiadandó PID nem használt. Amennyiben használt, az eljárást addig ismételgeti, amíg nem talál szabad PID-et.

Az *allowed* eljárás ellenőrzi, hogy a megadott elérés engedélyezett-e a fájlra. Például a *do_exec*-nek tudnia kell, hogy a fájl végrehajtható-e.

A *no_sys* eljárást sohasem szabad meghívni, azt az esetet kezeli, amikor a felhasználó egy illegális rendszerhívás-azonosítóval hívja meg a PM-et.

A *panic* eljárást akkor hívja meg a PM, ha egy olyan hiba történik, amelyből nem tud kilábalni. Jelenti a hibát a kernelnek, amely erre leállítja a MINIX 3-at.

A *utility.c* következő eljárása a *tell_fs*, amely összeállít egy üzenetet, és elküldi a fájlrendszernek, ha azt értesíteni kell a PM által feldolgozott eseményekről.

A *find_param* függvényt a figyelőprogramtól átvett paraméterek értelmezésére használják. Jelenleg arra szolgál, hogy a MINIX 3 betöltése előtti memóriahasználatot kiderítse, de más információk kiderítésére is lehetne használni.

A fájlban található következő két függvény a *sys_getproc* könyvtári függvényhez biztosít interfészt. Ez a függvény meghívja a rendszertaszkot a kernel processzustáblájából származó információk kinyerésére. A *sys_getproc* más szempontból egy az *include/minix/syslib.h* fájlban definiált makró, amely paramétereket ad át a sys_getinfo kernelhívásnak. A *get_mem_map* a processzus memóriatérképét kéri le. A *get_stack_ptr* lekéri a veremtáblát. Az itt felsorolt függvények mindegyikének szüksége van a processzusszámra, amely a sorszámot jelenti processzustáblában, nem a PID-et. A *utility.c* utolsó függvénye a *proc_from_pid*, amely megadja ezt a támogatást, a PID alapján visszaadja a processzussorszámot.

4.9. Összefoglalás

Ebben a fejezetben általánosan és a MINIX 3 esetében is megvizsgáltuk a memóriakezelést. Láthattuk, hogy a legegyszerűbb rendszerek nem használnak lapozást vagy cserét. Ha egy program betöltődik a memóriába, akkor a befejeződéséig ott is marad. A beágyazott rendszerek leggyakrabban így működnek, jó esélyel ROM-ban lévő kóddal. Néhány operációs rendszer egyszerre csak egy programot enged futni a memóriában, míg mások a multiprogramozást is támogatják.

A következő továbblépés a csere. Ha egy rendszer ezt használja, akkor több processzust tud futtatni, mint amennyi a memóriában elfér. Azokat a processzusokat, amelyeknek nem jut hely, kitesszük a lemezre. A szabad területeket a memóriában és a lemezen bittérképpel vagy szabadlistával tartjuk nyilván.

A fejlettebb számítógépekben gyakran valamilyen virtuális memóriát használnak. A legegyszerűbb esetben mindegyik processzus címtere lapoknak nevezett, egyenlő méretű blokkokra oszlik, amelyeket a memória bármelyik lapkeretébe elhelyezhetünk. Több lapcserélési algoritmust ajánlottunk, a két legismertebb a második lehetőség (SC) és az öregítő. A lapozásos rendszer jó működéséhez nem elég az algoritmus megválasztása, figyelni kell a munkahalmaz meghatározására, a memóriafoglalási elvekre és a lapméretre is.

A szegmentálás segíti a változó méretű adatszerkezetek kezelését, és egyszerűsíti a szerkesztést, valamint a kód és az adatok megosztását a programok között. Ezenkívül a különböző szegmenseknek eltérő védelmi szintjük lehet. Néha a szeg-

mentálást és a lapozást kombinálják, hogy kétdimenziós virtuális memóriát hozzanak létre. Az Intel Pentium a lapozást és a szegmentálást is támogatja.

A MINIX 3 memóriakezelése nagyon egyszerű: akkor foglalunk memóriát, amikor egy processzus fork vagy exec rendszerhívást hajt végre. A lefoglalt memória mérete a processzus futása során nem változik. Az Intel processzorokon kétféle memóriamodellt használ a MINIX 3. A kisebb processzusoknál a kód és az adatok ugyanabban a szegmensben vannak. A nagyobb processzusok szeparált kód- (vagy utasítás-) és adatrészt használnak, az ilyen processzusok egymás között megoszthatják kódszegmensüket, így a fork-nál csak az adatoknak és a veremnek kell helyet foglalni. Ugyanez a helyzet az exec-nél is, ha az elindított processzus már fut a gépen.

Az üres memória karbantartása lyuklistával és first fit algoritmussal történik. Annak ellenére, hogy ezzel is a processzuskezelő foglalkozik, mégis a legtöbb dolga a processzusok kezelésével kapcsolatos rendszerhívásokkal van. Számos rendszerhívás támogat POSIX-szignálokat, és mivel a legtöbb szignál alapértelmezett tevékenysége a processzus befejezése, ezt is a PM-nek kell megtennie. Sok rendszerhívás nem kapcsolódik közvetlenül a memóriakezeléshez, de a fájlrendszerhez még kevésbé, így mivel a PM kisebb, az a legkényelmesebb, ha ezeket is idesoroljuk.

Feladatok

1. Egy számítógépnek négy processzus tárolására elegendő memóriája van. Ezek a processzusok az idő felében I/O-ra várakoznak. Mekkora része marad kihasználatlan a processzoridőnek?
2. Tekintsünk egy cserélő rendszert, amely kezdőcím szerint rendezve a következő lyukakat tartalmazza: 10 KB, 4 KB, 20 KB, 18 KB, 7 KB, 9 KB, 12 KB és 15 KB. Melyik lyukat választja ki a következő kérésekre a first fit? Adjuk meg a választ best fit, next fit és worst fit esetén is.
   (a) 12 KB
   (b) 10 KB
   (c) 9 KB
3. Egy számítógép 1 GB RAM-mal rendelkezik; ez 64 KB-os egységekben foglalható. Mekkora az üres memóriát nyomon követő bittérkép mérete KB-ban?
4. Most vegyük figyelembe az előző kérdésnél a lyuklistát is. Mennyi memóriára lenne szükség a legjobb és a legrosszabb esetben? Tegyük fel, hogy az operációs rendszer az alsó 512 KB memóriát használja.
5. Mi a különbség a fizikai és a virtuális címek között?
6. A 4.8. ábrán látható laptábla alapján adjuk meg az alábbi virtuális címek fizikai párját.
   (a) 20
   (b) 4100
   (c) 8300

7. A 4.9. ábrán a virtuális cím lapmezője 4 bites, a fizikai cím lapmezője 3 bites. Általában megengedhető-e, hogy a virtuális lapbitek száma kisebb, egyenlő vagy nagyobb legyen, mint a fizikai lapbitek száma? Magyarázzuk meg válaszunkat.
8. Az Intel 8086-os processzor nem támogatja a virtuális memóriát, ennek ellenére néhány cég kiadott az eredeti 8086-osra épülő rendszereket, amelyek tudták a lapozást. Hogyan működhettek ezek? (Tanács: gondoljunk az MMU logikai helyére.)
9. Ha egy utasítás végrehajtása 1 ns-ig tart, egy laphiba további *n*sec-ot ad hozzá, akkor adjunk meg egy formulát az utasítások tényleges végrehajtási idejének kiszámítására, ha minden *k*-adik utasításnál következik be laphiba.
10. Egy gépnek 32 bites címtere és 8 KB-os lapjai vannak. A laptábla teljesen hardveres, bejegyzésenként egy 32 bites szóval. Amikor egy processzus elindul, akkor a laptáblája a memóriából a regiszterekbe másolódik; egy szó másolása 100 ns-ig tart. Ha egy processzus egy tized másodpercig fut (beleértve a laptábla betöltésének idejét), akkor a processzor idejének hány százalékát fordítjuk a laptábla betöltésére?
11. Egy gép 32 bites címeket és kétszintű laptáblát használ. A virtuális címeket három részre osztjuk: egy 9 bites mező a felső szintű laptáblához, egy 11 bites mező a másodlagos laptáblához és az eltolás. Mekkora a lapméret, és hány lapra oszlik a címtér?
12. Az alábbiakban egy 512 bájtos lapokat használó gép rövid assembly nyelvű programja látható. A program az 1020-as címen kezdődik, a veremmutató 8192 (a verem a nullás cím felé terjeszkedik). Adjuk meg a processzus laphivatkozási sorozatát. Minden utasítás 4 bájtos (egy szó), és a sorozat egyaránt tartalmazza az utasítás- és az adathivatkozásokat.
    Töltse a 6144-es szót a nullás regiszterbe.
    Tegye a nullás regisztert a verembe.
    Hívja meg az 5120-as címen levő eljárást. (A visszatérési cím a verembe kerül.)
    Vonja ki 16-ot a veremmutatóból.
    Hasonlítsa össze az aktuális paramétert a 4 konstanssal.
    Ugorjon az 5152-es címre, ha egyenlők.
13. Tegyük fel, hogy egy 32 bites virtuális cím az *a*, *b*, *c*, *d* mezőkre oszlik. Az első három mezőt a háromszintű laptáblához használjuk, az utolsó mező az eltolás. Függ-e a lapok száma mind a négy mező méretétől? Ha nem, akkor melyektől függ?
14. Egy számítógép, amelynek processzusai 1024 lapos címteret használnak, a memóriában tartja a laptábláját. A laptáblából egy szót 500 ns alatt lehet kiolvasni. E többletköltség csökkentésére a gép TLB-t alkalmaz, amely 32 darab virtuális lap-fizikai lapkeret párt tartalmaz, és a lekérdezése 100 ns-ig tart. Mekkora találati arány szükséges ahhoz, hogy az átlagos többletterhelést 200 ns-ra csökkentsük?
15. A VAX-gépek TLB-je nem tartalmaz *R* bitet. Miért? Ez a hiány csak a korra jellemző (1980), vagy volt mögötte más ok is?

16. Egy gép 48 bites virtuális és 32 bites fizikai címeket használ, a lapméret 8 KB. Hány bejegyzés szükséges a laptáblában?
17. Egy 64 bites virtuális címterű RISC CPU 8 GB RAM-ot használ invertált laptáblával és 8 KB lapméretekkel. Mi a TLB minimális mérete?
18. Egy számítógépnek négy lapkerete van. A lapok betöltésének és utolsó elérésének idejét, valamint az *R* és *M* bitek értékét az alábbi táblázatból olvashatjuk ki (az idők órajelciklusban adottak).

Lap	Betöltés ideje	Utolsó hivatkozás	*R*	*M*
0	126	279	0	0
1	230	260	1	0
2	120	272	1	1
3	160	280	1	1

(a) Melyik lapot cseréli ki az NRU algoritmus?
(b) Melyik lapot cseréli ki a FIFO algoritmus?
(c) Melyik lapot cseréli ki az LRU algoritmus?
(d) Melyik lapot cseréli ki a második lehetőség algoritmus?

19. Ha FIFO lapcserélési algoritmust használunk négy lapkerettel és nyolc lappal, akkor hány laphiba történik a 0172327103 hivatkozási sorozat feldolgozásánál, ha a lapkeretek kezdetben üresek? Mi a helyzet az LRU esetén?
20. Egy kis számítógépnek nyolc lapkerete van, mindegyik egy lapot tartalmaz. A lapkeretek az alábbi virtuális lapokat tartalmazzák: *A*, *C*, *G*, *H*, *B*, *L*, *N*, *D* és *F* ebben a sorrendben. A betöltési idejük: 18, 23, 5, 7, 32, 19, 3 és 8. A referenciabitjeik: 1, 0, 1, 1, 0, 1, 1 és 0; a módosított bitjeik: 1, 1, 1, 0, 0, 0, 1 és 1. Milyen sorrendben vizsgálja a lapokat a második lehetőség algoritmus, és melyiket választja?
21. Van-e olyan eset, amikor az „óra" és a második lehetőség algoritmus más lapot választ ki lapcserére? Ha igen, akkor mikor?
22. Tegyük fel, hogy egy számítógép PFF lapcserélési algoritmust használ, és van elegendő memória az összes processzus memóriában tartására laphiba nélkül. Mi történik?
23. Egy kisszámítógépnek négy lapkerete van. Az első óraütemben az *R* bitek értéke 0111 (a nullás lap 0, a többi lap 1). A következő pillanatokban ezek az értékek rendre 1011, 1010, 1101, 0010, 1010, 1100 és 0001. Ha az öregítő algoritmust használjuk egy 8 bites számlálóval, akkor adjuk meg a négy laphoz tartozó számláló értékét az utolsó órajel után.
24. Mennyi ideig tart egy 64 KB-os program betöltése a lemezről, ha a lemez átlagos elérési ideje 30 ms, forgásideje 20 ms, és egy sávon 32 KB adat fér el? A lapok véletlenszerűen szétszórtak a lemezen.
(a) 2 KB-os lapméretnél?
(b) 4 KB-os lapméretnél?
(c) 64 KB-os lapméretnél?
25. Az előző feladat eredményeit látva miért olyan kicsik a lapok? Nevezze meg a 64 KB-os lapméret két hátrányát a 4 KB-os lapmérethez viszonyítva.

26. Az egyik első időosztásos gépnek, a PDP–1-nek 4 KB memóriája volt 18 bites szavakból. Egyszerre csak egy processzus lehetett a memóriában. Amikor az ütemező úgy döntött, hogy egy másik processzust kezd el futtatni, akkor a memóriában levő processzust kiírta egy mágnesdobra. A dob palástján 4 KB adat fért el 18 bites szavakból. A dob írása vagy olvasása tetszőleges szónál kezdődhetett. Vajon miért ilyenre szerkesztették a dobot?
27. Egy beágyazott számítógépben minden processzusnak 4 KB-os lapokra osztott 64 KB-os címtere van. Egy processzus kódszegmensének mérete 32 768 bájt, az adatok mérete 16 386 bájt, a veremé 15 870 bájt. Belefér-e ez a processzus címterébe? Ha a lapméret 512 bájt lenne, akkor beleférne? Emlékezzünk arra, hogy egy lap nem tartalmazhat két különböző szegmenst.
28. Megfigyelték, hogy két laphiba között végrehajtott utasítások száma egyenesen arányos a processzusnak lefoglalt lapkeretek számával. Ha a lefoglalt memóriát megkétszerezzük, akkor a két laphiba közti átlagos idő is duplázódik. Tegyük fel, hogy egy utasítás végrehajtása 1 μs-ig tart, laphiba esetén pedig 2001 μs-ig (azaz 2 μs egy laphiba kezelése). Ha egy processzus 60 másodpercig fut, miközben 15 000 laphibát okoz, akkor kétszer annyi lapkeret birtokában mennyi ideig futna?
29. A Frugal Computer Company operációs rendszerének tervezői úgy gondolták, hogy csökkentik az új operációs rendszerükhöz szükséges háttértár mennyiségét. A főtervező azt javasolta, hogy a programkódot ne írják ki a lemezre a cserénél, hanem mindig közvetlenül a programfájlból olvassák be. Van ezzel a megközelítéssel valami probléma?
30. Mi a különbség a külső és a belső töredezettség között? Melyik fordul elő a lapozásos rendszerekben? Melyik fordul elő a tiszta szegmentálást alkalmazó rendszerekben?
31. Ha egyszerre használunk szegmentálást és lapozást, mint például a Pentiumban, akkor először a szegmensleírót, majd a laptáblát kell megnézni. Működhet-e a TLB hasonló módon kétszintű kereséssel?
32. Miért szükséges a MINIX 3 memóriakezeléséhez a *chmem* segédprogram?
33. A 4.44. ábrán látható a MINIX 3 első négy komponensének memóriahasználata. Mi lesz a cs értéke a következő rs után beöltött komponens esetén?
34. Az IBM-kompatibilis számítógépek 640 KB és 1 MB között tárolják a ROM- és I/O-memóriarészeket; a felhasználói processzusok nem használhatják ezt. Miután a MINIX 3 felügyelőprogramja áthelyezi magát 640 KB alá, a használható hely tovább zsugorodik. A 4.44. ábrán mennyi memória áll a processzus rendelkezésére a felügyelőprogram és a foglalt rész között, amennyiben az betöltési felügyelőprogram 52 256 bájtot foglalt le?
35. Az előző számít-e, ha a betöltési felügyelőprogram pontosan annyi memóriát használ, amennyit leírtunk, vagy ezt az egységet memóriaszeletekre kerekítjük?
36. A 4.7.5. alfejezetben láttuk, hogy az exec rendszerhívás a megfelelő lyuk keresését az előző processzus memóriájának felszabadítása előtt végzi. Ez a módszer nem az optimális megoldást adja. Írjuk át az algoritmust, hogy jobb legyen.
37. A 4.8.4. alfejezetben megmutattuk, hogy a lyukakat a kód- és adatszegmens számára jobb külön megkeresni. Implementáljuk ezt az ötletet.

38. Tervezzük újra az *adjust* eljárást, hogy elkerüljük a szignált kapó processzus felesleges megszüntetését a túl szigorú veremteszt miatt.

39. MINIX 3-ban egy processzus aktuális memóriafoglalását az alábbi paranccsal tudjuk lekérni:

```
chmem +0 a.out
```

ennek azonban az a mellékhatása, hogy átírja a fájlt, megváltoztatva az idő adatait. Módosítsuk úgy a *chmem* parancsot egy új *showmem* parancsba, hogy csak egyszerűen kiírja a szükséges adatokra.

5. Fájlrendszerek

Minden számítógépes alkalmazás esetén szükség van információ tárolására és visszakeresésére. Minden program futása közben az információ egy részét tudja tárolni a programhoz tartozó memóriaterületen. Azonban ennek a tárolásnak a virtuális tár mérete határt szab. Néhány alkalmazás esetén ez a kapacitás elegendő, más esetekben – mint például repülőjáratok helyfoglalási rendszere, banki alkalmazás vagy vállalati nyilvántartás esetén – már túl kicsi.

A másik probléma a belső tárolás esetén, hogy a processzus befejeződésekor az információ elvész. Sok alkalmazás esetén (például adatbázisoknál) az információt meg kell őrizni hetekig, hónapokig vagy akár örökre. Elfogadhatatlan, hogy a processzus befejeződésekor elvesszen. Továbbá nem veszhet el akkor sem, ha a számítógép összeomlik.

A harmadik megoldandó probléma az, hogy gyakran biztosítani kell, hogy több processzus egy időben hozzáférhessen ugyanazon információhoz. Ha egy telefontudakozó adatait a processzushoz tartozó memóriában tárolnánk, akkor csak ez a processzus férhetne hozzá. Ez a probléma úgy oldható meg, hogy az információ tárolását minden processzustól függetlenné tesszük.

Hosszú távú információtárolás esetén tehát három alapvető követelményt kell kielégíteni:

1. Biztosítani kell igen nagyméretű információ tárolását.
2. Az információ életben maradjon az azt használó processzus befejeződésekor.
3. Több processzus egy időben hozzáférhessen az információhoz.

Az említett problémák szokásos megoldása az, hogy az információt mágneslemezen vagy más külső tárolón tároljuk. Az ilyen tárolás egységét nevezzük **fájl**nak. Processzusok a fájlokat olvashatják és szükség esetén új adatot írhatnak. Az információ fájlban való tárolásának **tartós**nak kell lennie, vagyis nem befolyásolhatja processzusok indítása és befejeződése. Fájl csak akkor tűnhet el, ha a tulajdonosa azt félreérthetetlenül kitörli.

A fájlokat az operációs rendszer kezeli. Az operációs rendszerek tervezésének fontos témái közé tartozik a fájlok szerkezete, elnevezése, hozzáférése, használa-

ta, védelme és megvalósítása. Egészében véve, az operációs rendszer azon része, amely a fájlokkal foglalkozik, a **fájlrendszer**, és ez a fejezet témája.

A felhasználó nézőpontjából a fájlrendszerrel kapcsolatban a legfontosabb az, hogy miként jelenik meg számára, azaz mi a fájlok tartalma, hogyan lehet fájlokat elnevezni, védeni, milyen műveletek végezhetők fájlokkal, és így tovább. Azok a részletek, amelyek arra vonatkoznak, hogy vajon láncolt listán vagy bittérképen történik-e a szabad tárhelyek nyilvántartása, vagy hogy hány szektor van egy logikai blokkban, kevéssé érdekes a felhasználó számára, ugyanakkor nagyon fontosak a fájlrendszer tervezői számára. Ezért a fejezetet több részre osztottuk. Az első két rész a fájlok és a könyvtárak felhasználói felületével foglalkozik. Ezután vizsgáljuk részletesen a fájlrendszerek megvalósítását, majd a biztonsági és védelmi mechanizmusokkal foglalkozunk, végül a MINIX 3 fájlrendszerének a leírását adjuk meg.

5.1. Fájlok

Ebben az alfejezetben a felhasználó nézőpontjából vizsgáljuk a fájlokat, tehát azt, hogy miként használhatók, milyen tulajdonságaik vannak.

5.1.1. Fájlnevek

A fájl egy absztrakciós mechanizmus, amely lehetővé teszi az információ lemezen történő tárolását és későbbi visszaolvasását. Ezt olyan módon kell biztosítania, hogy a felhasználót megkímélje azoktól a részletektől, hogy hol és hogyan tárolják az információt, és hogy a mágneslemez-tároló ténylegesen hogyan is működik.

Valószínűleg a legfontosabb jellemzője minden absztrakciós mechanizmusnak az a mód, ahogyan az objektumokat elnevezi és kezeli. Ezért a fájlrendszerek vizsgálatát a fájlok elnevezésének tanulmányozásával kezdjük. Amikor egy processzus létrehoz egy fájlt, nevet ad neki. Ha a processzus befejeződik, a létrehozott fájl továbbra is létezik, és minden további processzus az adott névvel férhet hozzá.

A fájlok elnevezésére vonatkozó pontos szabályok eltérők az egyes operációs rendszerekben, de mindegyik lehetővé teszi 1–8 karakterből álló azonosító használatát érvényes fájlnévként. Tehát az *andrea*, *bruce* és *cathy* lehetséges fájlnevek. Gyakran számjegyek és speciális karakterek is megengedettek, így a *2*, *fontos!* és a *2.14. ábra* szintén érvényes fájlnevek. Több fájlrendszer esetén a név hossza 255 karakter is lehet.

Néhány fájlrendszer különbséget tesz kis- és nagybetű között, míg mások esetén a kis- és nagybetűk egyenértékűek. A Unix (és minden változata) az első, az MS-DOS a második kategóriába tartozik. Unix-rendszerben a következő fájlnevek mind különbözők: *maria*, *Maria*, *MARIA*, MS-DOS-rendszerben pedig ugyanazt a fájlt azonosítják.

A Windows ezen szélsőséges esetek közé esik. A Windows 95 és Windows 98 fájlrendszere az MS-DOS fájlrendszerén alapszik, ezért annak sok tulajdonságát, köztük a fájlnevek képzését is örökölte. Minden új változat javításokat adott a rendszerhez, de az általunk vizsgált tulajdonságok többnyire azonosak az MS-DOS és a „klasszikus" Windows-változatokkal. A Windows NT, Windows 2000 és Windows XP is támogatja az MS-DOS-fájlrendszert, azonban ezeknek a rendszereknek saját fájlrendszerük is van (**NTFS**), amelynek eltérő tulajdonságai vannak (mint például Unicode fájlnevek). Ez a fájlrendszer a későbbi verziókban további változásokon ment keresztül. Ebben a fejezetben a korábbi rendszerekre Windows 98 néven hivatkozunk. Jelezni fogjuk, ha valamely tulajdonság nem alkalmazható az MS-DOS vagy a Windows 95 esetén. Hasonlóan, ha újabb rendszerre hivatkozunk, akkor az vagy az NTFS, vagy a Windows XP fájlrendszere, és az olyan esetekben, amikor egy tulajdonság eltér a Windows NT vagy a Windows 2000 rendszerekben, akkor azt jelezzük. Ha csak annyit mondunk, hogy Windows, akkor az összes, Windows 95 utáni Windows-fájlrendszert értjük alatta.

Sok operációs rendszer támogatja a két részből álló fájlnevek használatát, ahol a két részt a pont karakter választja el, mint *prog.c*. A pont után álló részt **kiterjesztés**nek nevezzük, és általában a fájl tartalmára utal. Az MS-DOS-rendszerben például a fájlnév első része legfeljebb 8 karakterből állhat, a kiterjesztés pedig legfeljebb 3 karaktert tartalmazhat. Unixban a kiterjesztés, ha van, akármilyen hosszú lehet, és több kiterjesztés is megengedett, például *prog.c.bz2*, ahol a *.bz2* kiterjesztés arra utal, hogy a *prog.c* fájlt bzip2 algoritmussal tömörítették. Az 5.1. ábra néhány gyakran használt kiterjesztést és azok jelentését tartalmazza.

Néhány rendszerben (például a Unix-ban) a kiterjesztés csak konvenció, és használatának nincs következménye. A *file.txt* nevű fájl bizonyára valamilyen szöveges adatot tartalmaz, de ez a név csak a tulajdonost emlékezteti és nem mond semmit a számítógép (operációs rendszer) számára. Ugyanakkor a C fordítóprogram megkövetelheti, hogy a fordítandó programot tartalmazó fájl nevének *.c* kiterjesztése legyen, máskülönben visszautasítja a fordítást.

Kiterjesztés	Jelentés
file.bak	Biztonsági másolat
file.c	C nyelvű forráskód
file.gif	Compuserve Graphical Interchange Format (GIF) kép
file.html	WWW HyperText Markup Language dokumentum
file.iso	ISO CD képfájl (CD-re íráshoz)
file.jpg	JPEG szabvány szerint kódolt képállomány
file.mp3	MPEG 3-as audioformátum szerint kódolt zene
file.mpg	MPEG szabvány szerint kódolt videoállomány
file.o	Tárgykód (lefordított, de nem összeszerkesztett)
file.pdf	PDF (Protable Document Format) dokumentum
file.ps	PostScript állomány
file.tex	TEX kiadványszerkesztőre írt dokumentum
file.txt	Közönséges szöveges állomány
file.zip	Tömörített állomány

5.1. ábra. *Tipikus fájlkiterjesztések*

Az ilyen jellegű konvenciók különösen hasznosak lehetnek akkor, ha ugyanaz a program különböző fájlokat képes feldolgozni. A C fordítónak például, adhatunk egy listát, amelynek elemeit lefordítja és összeszerkeszti, ahol a lista elemei lehetnek C forrásnyelvű állományok (például *foo.c*) vagy assembly állományok (például *bar.s*), vagy tárgykódok (például *other.o*) is. A kiterjesztés ilyenkor alapvető, mert ez mondja meg a fordítónak, hogy mely fájlok a C, melyek az assembly, illetve a tárgykódfájlok.

Ezzel szemben a Windows nagyon is érzékeny a kiterjesztésre, és jelentést rendel hozzájuk. A felhasználó (vagy processzus) regisztrálhatja az operációs rendszerben a kiterjesztéseket, és megadhatja, hogy melyikhez milyen program tartozik. Amikor a felhasználó duplán kattint egy fájl nevére, akkor a kiterjesztéshez regisztrált program elindul, paraméterként a fájl nevét véve. Például egy *file.doc* nevű fájlra duplán kattintva a Microsoft Word program indul, és betölti szerkesztésre a *file.doc* fájlt.

Vannak, akik furcsának találhatják, hogy a Microsoft alapértelmezés szerint néhány gyakori kiterjesztést láthatatlanná tesz, mert ezek olyan fontosak. Szerencsére a Windows legtöbb ilyen, „alapértelmezés szerint rossz" beállítását a felkészült felhasználó, aki tudja, hol keresse őket, módosíthatja.

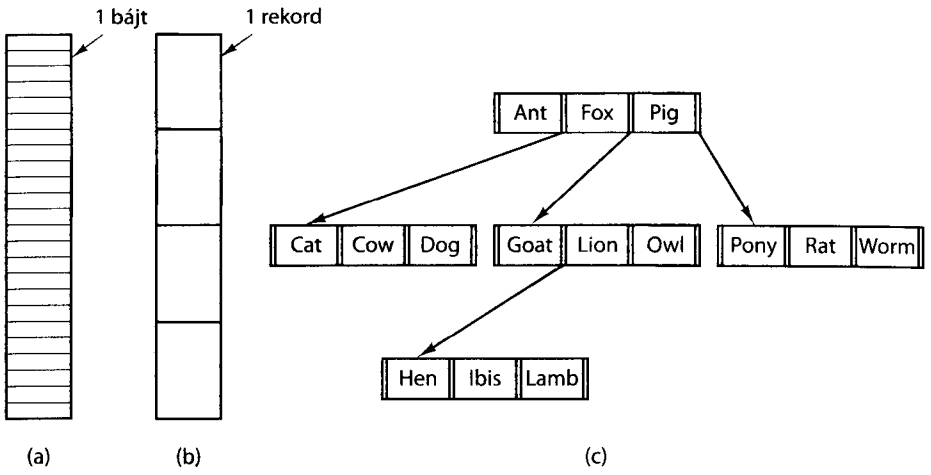
5.1.2. Fájlszerkezet

A fájlok többféle módon strukturálhatók. A három általános lehetőséget az 5.2. ábra szemlélteti. Az 5.2.(a) ábrán látható fájlt strukturálatlan bájtsorozat alkotja. Valójában az operációs rendszer nem tudja és nem is törődik azzal, hogy mi van a fájlban. Amit lát, az csupán egy bájtsorozat. Minden jelentést felhasználói program rendel a fájlhoz. A Unix-és a WINDOWS 98 is ezt a megközelítést használja.

Akkor a legnagyobb a flexibilitás, ha az operációs rendszer bájtsorozatnak tekinti a fájlt. A felhasználói programok bármit tárolhatnak a fájlban, és bármely alkalmas módon elnevezhetik azt. Az operációs rendszer nem ad segítséget, de nem is korlátoz. Ez utóbbi nagyon fontos lehet, ha a felhasználó nem megszokott dolgokat akar művelni.

Az első módosítás az egyszerű fájlszerkezethez képest az 5.2.(b) ábrán látható. Ebben a modellben a fájl rögzített hosszúságú rekordok sorozata, és minden rekordnak meghatározott belső szerkezete van. Ekkor az az elv érvényesül, hogy minden olvasási művelet egy teljes rekordot ad, és minden írási művelet egy teljes rekordot ír felül vagy hozzáad a fájlhoz. Évekkel ezelőtt, amikor 80 oszlopos lyukkártyát használtak, sok operációs rendszer fájlrendszere 80 karakter hosszú rekordok alkotta fájlokra épült. Ezek a rendszerek szintén támogatták a 132 karakteres rekordokat, ami a sornyomtatóknak felelt meg (akkortájt a nyomtatók nagy, 132 oszlopos láncos sornyomtatók voltak). A programok 80 karakter egységben olvastak be adatokat, és 132 karakter egységben írtak ki, jóllehet az utolsó 52 lehetett szóköz is. Manapság nincs olyan általános célú rendszer, amely ezt a rendszert használná.

A harmadik típusú fájlszerkezetet az 5.2.(c) ábra mutatja. Ebben a szervezésben a fájl rekordokból felépülő fa, a rekordok hossza nem feltétlenül azonos, és



5.2. ábra. *Három lehetséges fájlszerkezet. (a) Bájtsorozat. (b) Rekordsorozat. (c) Fa*

minden rekord egy rögzített pozícióban **kulcsmező**t tartalmaz. A fa rekordjai oly módon rendezettek, hogy gyorsan megkereshető legyen egy adott kulcsú rekord.

Az alapvető művelet itt nem a „következő" rekord elérése, bár ez is lehetséges, hanem adott kulcsú elem elérése. Az 5.2.(c) ábrán látható *zoo* fájl esetén például kérhetjük a rendszertől azt a rekordot, amelynek kulcsa *pony,* nem törődve azzal, hol van a fájlban ez a rekord. Továbbá új rekordokkal bővíthetjük a fájlt. Ekkor az operációs rendszer, és nem a felhasználó dönti el, hogy hova kerüljön az új rekord. Ez a fájltípus nyilvánvalóan teljesen különböző a Unix és a Windows 98 által használt bájtsorozat-felfogástól, de az adatfeldolgozásra használt a nagyszámítógépek még ma is használják.

5.1.3. Fájltípusok

Sok operációs rendszer különböző típusú fájlok kezelését támogatja. A Unix és a Windows például közönséges (reguláris) fájlokat és könyvtárakat is megenged. A Windows XP **metaadat**fájlokat is alkalmaz; ezekkel később foglalkozunk. A Unix karakterspecifikus és blokkspecifikus fájlokat is kezel. **Közönséges** fájlok azok, amelyek a felhasználók által kezelt információkat tárolják. Az 5.2. ábrán minden fájl közönséges fájl. A **könyvtárak** rendszerfájlok, amelyek a fájlrendszer szerkezetének megvalósítását teszik lehetővé. A könyvtárakat később fogjuk tárgyalni. A **karakterspecifikus** fájlok bemenet/kimenet megvalósításához kapcsolódnak, és az olyan soros I/O-eszközöket modellezik, mint a terminálok, nyomtatók és hálózatok. A **blokkspecifikus** fájlok a mágneslemezegységeket modellezik. Ebben a fejezetben elsősorban közönséges fájlokkal foglalkozunk.

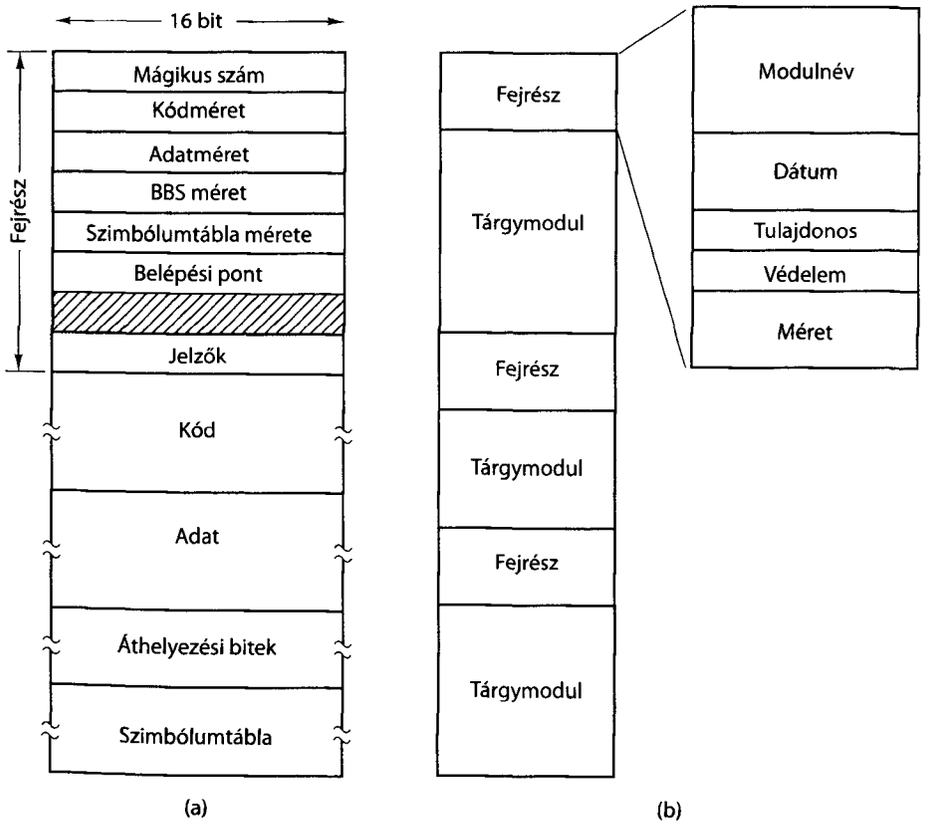
A közönséges fájlok vagy ASCII, vagy bináris fájlok. Az ASCII fájlok szövegsorokat tartalmaznak. Néhány rendszerben minden sort a kocsi vissza jel zár. Más

rendszerekben a sorvégejelet használják elválasztójelként. Néha mindkét jelet kell használni (például a Windowsban). Természetesen nem kell, hogy minden sor azonos hosszúságú legyen.

Az ASCII fájlok nagy előnye, hogy tartalmuk megjeleníthető úgy, ahogy van, kinyomtatható és szerkeszthető közönséges szövegszerkesztővel. Továbbá, ha sok program használ ASCII fájlt bemenetként és kimenetként, akkor az egyik program kimenetét a másik bemenetéhez lehet kapcsolni, például a parancsértelmezőben adatcső használatával. (A processzusok összecsövezése nem egyszerű dolog, de az információnak standard konvenció szerinti interpretálása, mint az ASCII, sokat segít.)

Más fájlok binárisak, ami csak azt jelenti, hogy nem szöveges (ASCII) fájlok. Az ilyen fájlok nyomtatásának eredménye érthetetlen, látszólag tele van hulladékkal. A bináris fájloknak általában van valamilyen belső struktúrájuk, amelyet a rájuk alkalmazott programok ismernek.

Például az 5.3.(a) ábrán egy korábbi Unix-rendszerbeli egyszerű, végrehajtható bináris fájlt látunk. Bár technikailag a fájl nem más, mint bájtsorozat, az operációs rendszer csak akkor tudja végrehajtatni, ha a fájl megfelelő formájú. Ennek öt része



5.3. ábra. *(a) Végrehajtható fájl. (b) Archív fájl*

van: fej, szöveg, adat, áthelyezési bitek, szimbólumtábla. A fejrész az ún. **mágikus számmal** kezdődik, amely a végrehajtható fájlokat azonosítja (megvédve attól, hogy nem szándékoltan végrehajtsunk egy nem végrehajtható fájlt). Ezt követi a különböző részek hossza, a belépési pont és a jelzőbitek. A fejrész után következik a program szöveg- és adatrésze. Ezek a részek betöltődnek a memóriába, és áthelyezési bitek alapján áthelyeződnek. A szimbólumtábla hibakeresésre használatos.

A második példánk bináris fájlra egy archív állomány, szintén Unix-rendszerben. Ez könyvtári eljárások (modulok) lefordított, de össze nem szerkesztett gyűjteményéből áll. Minden modulállomány a fejrésszel kezdődik, amely tartalmazza a modul nevét, a létrehozás dátumát, a tulajdonost, a védelmi kódot és a méretet. Ugyanúgy, mint a végrehajtható állományokban, a modulfej itt is tele van bináris számokkal. Nyomtatásuk zagyvaságot eredményezne.

Minden operációs rendszernek fel kell ismernie egy fájltípust, a rendszerben végrehajtható fájl típusát, de néhány rendszer ezenkívül más típust is felismer. A régi TOPS–20-rendszer (a DECsystem 20 operációs rendszere) odáig elment, hogy megvizsgálta minden végrehajtandó fájl létrehozási idejét. Ezután megkereste a forrásfájlt, és összehasonlította annak utolsó módosítási idejével. Ha a forrás fiatalabb volt, mint a bináris, akkor automatikusan lefordította a programot. Unix-fogalmakkal ez azt jelentette, hogy a *make* be volt építve a parancsértelmezőbe. Ebben a rendszerben a kiterjesztés kötelező volt, így a rendszer meg tudta állapítani, hogy a bináris melyik forrásprogramból keletkezett.

A szigorúan típusolt fájlok esetén problémát okozhat, ha a felhasználó olyat tesz, amire a rendszer tervezői nem készültek fel. Tekintsünk egy olyan rendszert, amelyben a kimenet fájltípusa *dat* (adatfájl). Ha egy felhasználó olyan formázó programot ír, amely *.c* fájlokat (C forrásprogram) konvertál (például standard bekezdéseket produkál), a művelet eredményét kimenetfájlba írja, akkor ennek típusa *dat* lesz. Ha a felhasználó a C fordítónak adja bemenetként, akkor a rendszer visszautasítja, mert kiterjesztése nem megfelelő. Megkísérelve a *p.dat* állomány átmásolását a *p.c* névre, ismét visszautasít (mert védi a felhasználót a hibáktól).

Amíg az ilyen felhasználóbarát rendszer segítheti a kezdőt, hátráltatja a tapasztalt felhasználót, mert sok erőfeszítést igényel tőle, hogy kijátssza a rendszert.

5.1.4. Fájlelérés

A korai operációs rendszerek csak egyféle fájlelérést biztosítottak: a **szekvenciális elérést**. Ezekben a rendszerekben minden processzus a megadott sorrendben olvashatta a bájtokat vagy rekordokat az elsőtől kezdve, de nem tudott átlépni egységeket és a sorrendtől eltérően olvasni. A szekvenciális fájlok elejére lehet állni, így akárhányszor újraolvashatók. A szekvenciális fájlok akkor alkalmazhatók jól, ha a tároló médium mágnesszalag, és nem mágneslemez.

Amikor a mágneslemez-tárolók használatba jöttek, lehetővé vált a fájlok bájtjainak vagy rekordjainak nem sorrendben történő olvasása is, vagy a kulcs, nem pedig pozíció szerinti olvasás. Az olyan fájlt, amelynek bájtjai vagy rekordjai tet-

szőleges sorrendben olvashatók, **közvetlen elérésű** fájlnak hívjuk. Sok alkalmazás igényel ilyen fájlokat.

A közvetlen elérésű fájlok alapvetők sok alkalmazásnál, mint például az adatbázisoknál. Ha egy utas jelentkezik, hogy helyet akar foglalni egy adott járatra, akkor a helyfoglaló programnak el kell érnie a járat adatait tartalmazó rekordot anélkül, hogy előbb végigolvasná több ezer járat rekordjait.

Két módszer használatos annak meghatározására, hol kezdődjön az olvasás. Az első esetén minden read olvasó műveletben meg kell adni az olvasandó rekord pozícióját. A másik módszer egy speciális műveletet, a seek-et használja a kezdő pozíció beállítására, ezt követően a fájl az aktuális pozíciótól kezdődően szekvenciálisan olvasható.

Néhány nagygépes rendszerben a fájl létrehozásakor meg kell adni, hogy a fájl szekvenciális vagy közvetlen elérésű legyen. Ez lehetővé teszi, hogy a rendszer különböző tárolási módszert használjon a két esetben. A modern operációs rendszerek nem tesznek ilyen különbséget, minden fájl automatikusan közvetlen elérésű.

5.1.5. Fájlattribútumok

Minden fájlnak van neve és adattartalma. Ezenfelül minden operációs rendszer más információkat is rendel a fájlokhoz, mint például a létrehozás dátuma és a fájl mérete. Ezeket a többletadatokat nevezzük a fájl **attribútumai**nak, de vannak akik **metaadat**nak hívják. Az attribútumok listája erősen változik rendszerről rendszerre. Az 5.4. ábra táblázatában felsoroltunk néhány lehetséges attribútumot. Nincs olyan rendszer, amelyben mindegyik szerepelne, de valamennyi megtalálható a rendszerek valamelyikében.

Az első négy attribútum azt szabályozza, hogy ki és hogyan érheti el a fájlt. Minden séma lehetséges, néhányat később tanulmányozunk. Néhány rendszerben jelszó szükséges a fájl eléréséhez, amelynek a fájl attribútumának kell lennie.

A jelzőbitek vagy rövid mezők, amelyek specifikus tulajdonságokat szabályoznak vagy engedélyeznek. A rejtett fájlok például nem jelennek meg az összes fájl listázásakor. Az archív jelző azt jelenti, hogy a fájlt archiválni kell. Az archiválóprogram törli a bitet, az operációs rendszer pedig 1-re állítja, valahányszor a fájl módosul. Az ideiglenességjelző annak megjelölésére szolgál, hogy a fájlt létrehozó processzus befejeződésekor a fájlt törölni kell.

A rekord hossza, a kulcs pozíciója és a kulcs mérete csak azoknál a fájloknál használatos, amelyeknél lehetséges kulcs szerinti keresés. Ezek az információk a kereséshez szükségesek.

A különböző időattribútumok annak nyomon követésére szolgálnak, hogy a fájlt mikor létesítették, mikor volt az utolsó hozzáférés, illetve az utolsó módosítás. Ezek az információk sokféle célra használhatók. Például az a forrásfájl, amely módosult a hozzá tartozó tárgykód létrehozása után, újrafordítást igényel. Ehhez ezek a mezők szolgáltatják a szükséges információt.

Az aktuális méret megmondja, hogy a fájl jelenleg mekkora. Néhány régebbi nagygépes operációs rendszer megköveteli, hogy a fájl létrehozásakor megadjuk,

Mező	Értelmezés
Védelem	Ki érheti el a fájlt és milyen módon
Jelszó	Jelszó, amelyet az eléréshez meg kell adni
Létrehozó	A fájl létrehozójának azonosítója
Tulajdonos	Az aktuális tulajdonos azonosítója
Csak olvasható jelző	0, ha írás és olvasás megengedett, 1, ha csak olvasható
Rejtettségi jelző	0 a normál eset, 1, ha listázásban nem megjelenítendő
Rendszerjelző	0 normál fájl, 1 rendszerfájl esetén
Archív jelző	0, ha archiválva volt, 1, ha archiválásra kijelölt
ASCII/bináris jelző	0, ha ASCII, 1, ha bináris a fájl
Közvetlen elérés jelző	0, ha csak szekvenciális, 1, ha közvetlen elérésű a fájl
Ideiglenességjelző	0, ha normál fájl, 1, ha törölni kell a processzus befejeződésekor
Zároltságjelző	0, ha nem zárolt, 1, ha zárolt a fájl
Rekord hossza	A bájtok száma egy rekordban
Kulcs pozíciója	A kulcs pozíciója a rekordban
Kulcs hossza	A kulcsmező hossza bájtokban
Létesítési idő	A fájl létrehozásának dátuma és időpontja
Utolsó hozzáférés ideje	Az utolsó hozzáférés dátuma és időpontja
Utolsó módosítás ideje	Az utolsó módosítás dátuma és időpontja
Aktuális méret	A bájtok száma a fájlban
Maximális méret	A lehetséges maximális fájlméret bájtban

5.4. ábra. *Néhány lehetséges fájlattribútum*

hogy a fájl maximálisan mekkora lehet, mert ennek megfelelően foglal előre tárhelyet számára. A modern operációs rendszerek azonban már elég okosak ahhoz, hogy enélkül is megoldják a feladatot.

5.1.6. Fájlműveletek

A fájlok azért vannak, hogy információt tároljunk bennük, amit később visszakereshetünk. A különböző rendszerek különböző műveleteket biztosítanak tárolásra és visszakeresésre. Alább összefoglaljuk a fájlokkal kapcsolatos leggyakoribb rendszerhívásokat.

1. **LÉTESÍTÉS** (create). Adatot nem tartalmazó, üres fájl létrehozása. A rendszerhívás célja, hogy jelezze a fájl keletkezését, és beállítsa a fájl bizonyos attribútumait.
2. **TÖRLÉS** (delete). Ha a fájlra nincs szükség a továbbiakban, törölni kell, hogy felszabaduljon az általa elfoglalt lemezterület. Minden rendszer biztosítja ezt a műveletet.
3. **MEGNYITÁS** (open). A fájlt használni kívánó processzusnak a használat megkezdése előtt meg kell nyitnia a fájlt. Az open műveletnek az a célja,

hogy a rendszer beolvassa a fájl attribútumait a memóriába és néhány, a fájlhoz tartozó lemezcímet a további fájlműveletek gyorsítása végett.

4. **LEZÁRÁS** (close). Ha a fájlt feldolgozó műveletek befejeződtek, akkor az attribútumaira és a lemezcímekre a továbbiakban nincs szükség, tehát a fájl lezárható, és ezzel belső memóriaterület szabadítható fel. Sok rendszer azzal ösztönzi a fájlok lezárását, hogy korlátozza a processzus által egy időben nyitva tartható fájlok számát. A lemezre írás blokkokban történik, és a fájl lezárása kikényszeríti az utolsó blokk kiírását akkor is, ha az még nincs tele.
5. **OLVASÁS** (read). A fájlból adatokat olvashatunk be a memóriába. A beolvasott bájtok általában az aktuális pozíciótól kezdődően olvasódnak be. A read művelet kezdeményezőjének meg kell adnia a beolvasandó bájtok számát, és hogy az adatok a memóriában hova kerüljenek.
6. **ÍRÁS** (write). A fájlba írt adatok általában szintén az aktuális pozíciótól íródnak ki. Ha az aktuális pozíció a fájl végén van, akkor a kiírt bájtokkal növekszik a fájl mérete. Ha az aktuális pozíció belső, akkor felülírás történik, és a felülírt adatok mindörökre elvesznek.
7. **HOZZÁTOLDÁS** (append). Ez a művelet az írás korlátozott változata; írni csak a fájl végére lehet. Azok a rendszerek, amelyek csak kevés számú fájlműveletet tartalmaznak, általában nem rendelkeznek append művelettel. Sok rendszerben egy művelet többféle módon is megvalósítható, és ezekben a rendszerekben általában van append is.
8. **POZICIONÁLÁS** (seek). Közvetlen elérésű fájlok esetén szükség van az írás és olvasás helyének megadására. Az egyik általános megoldás erre a seek rendszerhívás, amely a fájl aktuális pozícióját megadott címre állítja át. A seek művelet végrehajtása után az olvasás, illetve az írás a beállított pozíciótól történik.
9. **ATTRIBÚTUMLEKÉRDEZÉS** (get attributum). A processzusoknak feladatuk elvégzéséhez gyakran szükségük van arra, hogy megtudják a fájl attribútumait. Például a Unix *make* parancsa arra való, hogy nagy szoftverfejlesztések forrásait kezelje. A *make* parancs végrehajtásakor először megvizsgálja az összes forrás- és tárgykódfájl módosítási idejét, és megállapítja, hogy minimálisan hány fordítás szükséges ahhoz, hogy minden aktualizált legyen. Ennek elvégzéséhez le kell kérdeznie a fájlok attribútumait, nevezetesen a módosítási időt.
10. **ATTRIBÚTUMBEÁLLÍTÁS** (set attributum). Bizonyos attribútumokat a felhasználó is beállíthat és módosíthat a fájl létrehozása után. Ez a rendszerhívás ezt a célt szolgálja. Nyilvánvaló példa erre a védelmi mód. A legtöbb jelzőattribútum is ebbe a kategóriába tartozik.
11. **ÁTNEVEZÉS** (rename). Gyakran előfordul, hogy a felhasználó meg akarja változtatni egy már létező fájl nevét. Ez a rendszerhívás ezt a célt szolgálja. Az átnevezés persze megoldható oly módon is, hogy először a fájlt átmásoljuk az új névre, majd a régit töröljük.
12. **ZÁROLÁS** (lock). A fájl vagy annak egy részének zárolása megakadályozza, hogy egyidejűleg több processzus is hozzáférjen. Például egy repülőgépes helyfoglaló rendszerben az adatbázis zárolása lehetetlenné teszi, hogy ugyanazt azt ülőhelyet több utasnak is kiadják.

5.2. Könyvtárak

A fájlok nyilvántartására az operációs rendszerek rendszerint **könyvtárakat** vagy **mappákat** használnak, amelyek sok rendszerben maguk is fájlok. Ebben az alfejezetben a könyvtárakat tárgyaljuk, szervezésüket, tulajdonságaikat és a rajtuk végezhető műveleteket.

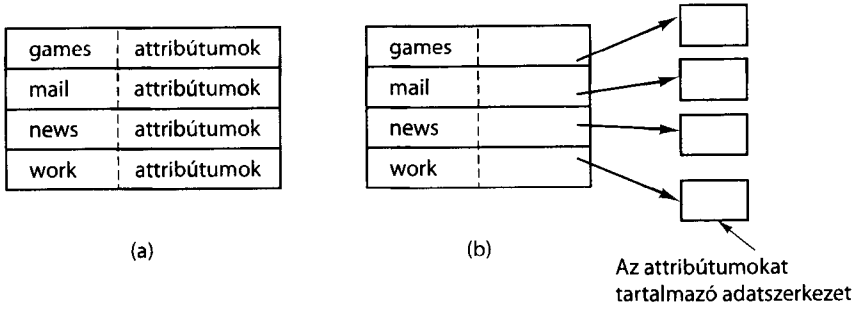
5.2.1. Egyszerű könyvtárszerkezet

Minden könyvtár tipikusan egy bejegyzést tartalmaz minden fájlhoz. Egy lehetőséget mutat az 5.5.(a) ábra, ahol minden bejegyzés tartalmazza a fájl nevét, attribútumait és azon lemezterület címét, ahol a fájlt tárolják. Egy másik lehetőség látható az 5.5.(b) ábrán. Itt minden könyvtári bejegyzés a fájl nevét és azon adatszerkezet címét tartalmazza, ahol a fájl attribútumai és lemezcímei tárolódnak. Mindkét lehetőség általánosan használt.

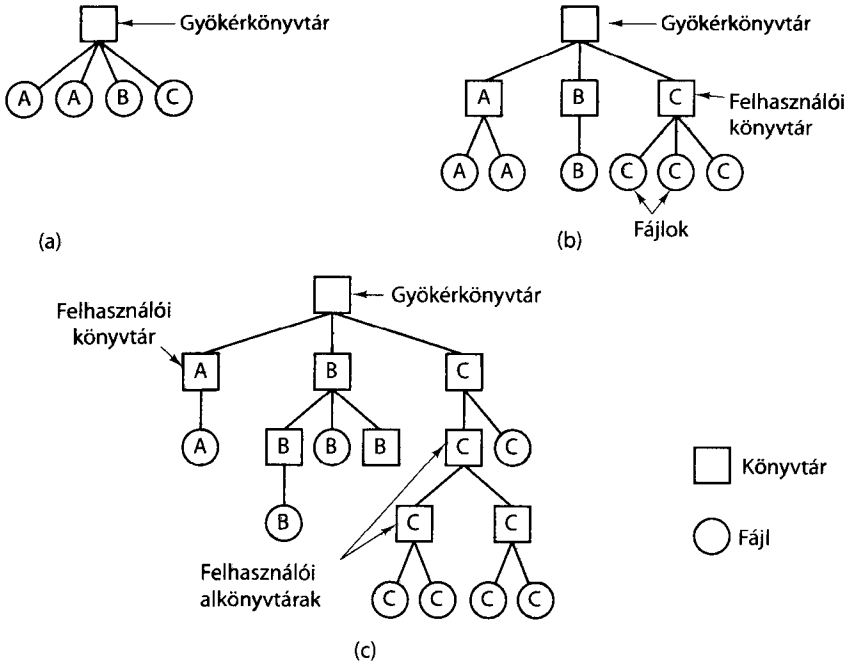
A fájl megnyitásakor a rendszer addig keres a könyvtárban, amíg a megnyitandó fájl nevét meg nem találja. Ezután kiolvassa a fájl attribútumait és a lemezcímeket vagy közvetlenül a könyvtári bejegyzésből, vagy az általa mutatott címről, és eltárolja ezeket az adatokat a memóriában. Így minden további fájlművelet a memóriában találja a szükséges információkat.

A könyvtárak száma rendszerről rendszerre változik. A legegyszerűbben tervezett rendszerben csak egyetlen könyvtár van, amely az összes felhasználó minden fájlját tartalmazza, mint azt az 5.6.(a) ábra mutatja. A korai személyi számítógépeken az egykönyvtáras rendszer általános volt, részben azért, mert csak egy felhasználó volt.

Az egykönyvtáras rendszereknél több felhasználó esetén az a probléma keletkezik, hogy különböző felhasználók ugyanazt a fájlnevet alkalmazhatják saját fájljaik elnevezésére. Például ha az *A* felhasználó egy *mailbox* nevű fájlt létesít, majd később a *B* felhasználó is létesít egy *mailbox* nevű fájlt, akkor a *B* fájlja felülírja *A* fájlját. Következésképpen ezt a módszert nem használják többfelhasználós



5.5. ábra. *(a) Attribútumok a könyvtári bejegyzésben. (b) Attribútumok más helyen*



5.6. ábra. *Három fájlrendszerterv. (a) Egyetlen közös könyvtár. (b) Felhasználónként egy könyvtár. (c) Tetszőleges számú fa felhasználónként. A betűk könyvtárakat vagy fájltulajdonosokat jelölnek*

rendszerekben, de használható kis beágyazott rendszerekben, mint például a PDA-k (personal digital assistant) vagy mobiltelefonok.

A konfliktus, amely abból ered, hogy különböző felhasználó ugyanazt a fájlnevet alkalmazza, úgy oldható fel, hogy minden felhasználó külön saját könyvtárat kap. Ezáltal nem keletkezik ütközés, ha két felhasználó ugyanazt a fájlnevet alkalmazza, és nem gond, hogy ugyanaz a név több könyvtárban is előfordul. Ez a tervezés az 5.6.(b) ábrán látható szerkezetet eredményezi. Ez a módszer használható lenne például többfelhasználós rendszerben, vagy olyan személyi számítógépek alkotta egyszerű hálózatban, amelyek közös hálózati fájlszervert használnak.

Ez a módszer feltételezi, hogy amikor egy felhasználó meg akar nyitni egy fájlt, akkor az operációs rendszer tudja, hogy ki a felhasználó, és ennek megfelelő könyvtárban keressen. Tehát az egyszintű könyvtárrendszerrel ellentétben itt szükség van valamilyen bejelentkeztető eljárásra, amellyel a felhasználó megadja a nevét és jelszavát.

E módszer legegyszerűbb megvalósítása esetén minden felhasználó kizárólag a saját könyvtárában lévő fájlokat érheti el.

5.2.2. Hierarchikus könyvtárszerkezet

A kétszintű hierarchia megszünteti a felhasználók közötti fájlnévkonfliktust. Egy másik probléma az, hogy a sok fájlt használó felhasználók kisebb csoportokba akarják csoportosítani a fájljaikat. Például egy professzor külön csoportba akarja rendezni az előadásjegyzeteit az éppen készülő könyvének fejezeteitől. Általános hierarchiára van tehát szükség (vagyis könyvtárak alkotta fára). Ezzel a megközelítéssel minden felhasználó annyi könyvtárat létesíthet, amennyire szüksége van, így a fájlok természetes módon csoportosíthatók. Ezt a megközelítést mutatja az 5.6.(c) ábra. Itt az *A*, *B* és *C* könyvtárakat tartalmazza a gyökérkönyvtár, mindegyik különböző felhasználóhoz tartozik, kettő közülük alkönyvtárakat hozott létre munkáinak tárolására.

Tetszőleges számú alkönyvtár létrehozásának lehetősége nagy hatású eszközt nyújt az állományok szervezésére. Ennek köszönhetően majdnem minden modern PC vagy fájlszerver fájlrendszere ilyen módon szerveződik.

Azonban, mint már korábban is jeleztük, a történelem gyakran ismétli önmagát új technológiák alkalmazásával. A digitális fényképezőgépeknek valahol tárolniuk kell a képeket, szokásosan flash memóriakártyán. A legelső digitális fényképezőgépek egyetlen könyvtárban tárolták a képeket *DSC0001.JPG, DSC0002.JPG* stb. neveken. Nem sok idő telt el, és a fényképezőgépek gyártói olyan fájlrendszereket készítettek, mint amit az 5.6.(b) ábra mutat. A fényképezőgép-tulajdonosok nem értették, hogyan lehet többszörös könyvtárakat használni, vagy ha meg is értették, valószínűleg nem látnák hasznát. Végső soron mindez (beágyazott) szoftver, és semmibe sem kerül a gyártónak. Lehetnek a jövőben digitális fényképezőgépek teljes hierarchiájú fájlrendszerrel, többszörös felhasználónévvel, 255 karakteres fájlnevekkel?

5.2.3. Útvonal megadása

Ha a fájlrendszer szervezése könyvtárak fáiból áll, akkor szükség van olyan módszerre, amely lehetővé teszi fájlnevek megadását. Két különböző módszer használatos. Az első módszer esetén a fájl nevét **abszolút (teljes) útvonal**lal adjuk meg, amely a gyökérkönyvtártól indulva a fájlig vezet. Például a */usr/ast/mailbox* útvonal azt jelenti, hogy a gyökérkönyvtár tartalmazza a *usr* könyvtárat, amelynek alkönyvtára az *ast* könyvtár, és ebben található a *mailbox* fájl. Az abszolút útvonal mindig a gyökérkönyvtárral kezdődik, és egyértelműen azonosítja a fájlt. Unix esetén az útvonal elemeit elválasztó jel a /. Windows esetén pedig a \. Tehát az útvonalat a két rendszerben az alábbiak szerint kell írni:

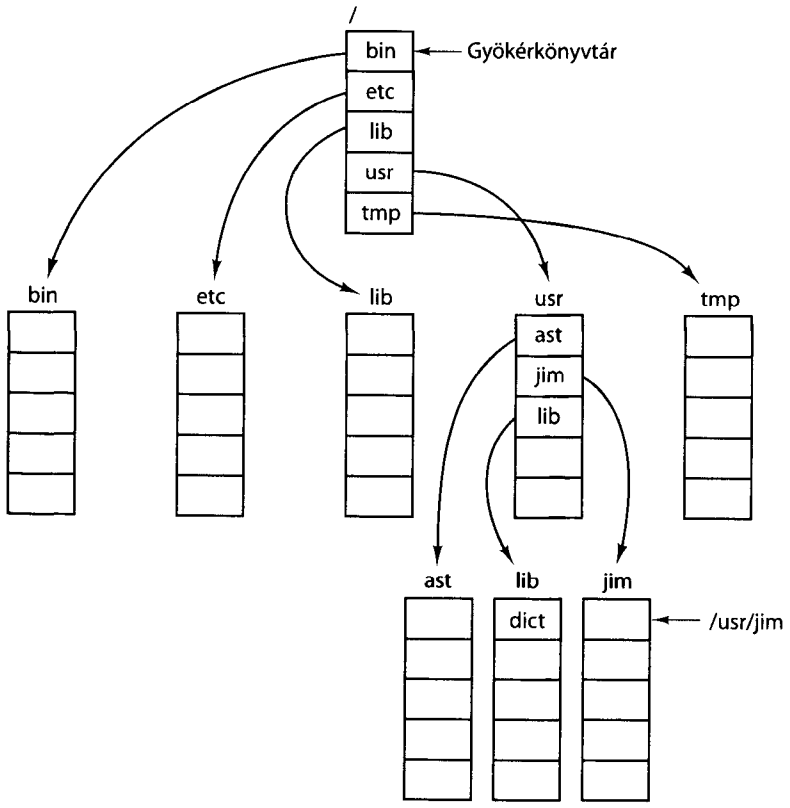
Windows	\usr\ast\mailbox
Unix	/usr/ast/mailbox

Mindegy, milyen jel az elválasztó, az útvonal akkor és csak akkor abszolút, ha az elválasztójellel kezdődik.

A másik módszer a fájl megnevezésére a **relatív útvonal**. Ez a módszer a **munkakönyvtár** (vagy más néven **aktuális könyvtár**) fogalmára épül. A felhasználó kijelölhet egy könyvtárat, amely az aktuális munkakönyvtár lesz, és ezután minden olyan útvonal, amely nem a gyökérkönyvtárral kezdődik, az aktuális könyvtárhoz relatív módon értelmeződik. Például ha az aktuális könyvtár a */usr/ast,* akkor az a fájl, amelynek abszolút útvonalmegadása */usr/ast/mailbox,* egyszerűen megadható a *mailbox* relatív útvonallal. Más szóval, az alábbi két Unix-parancs hatása megegyezik, ha az aktuális könyvtár a */usr/ast:*

```
cp /usr/ast/mailbox /usr/ast/mailbox.bak
cp mailbox mailbox.bak
```

A relatív forma gyakran kényelmesebb, de ugyanazt jelenti, mint az abszolút forma.



5.7. ábra. *Egy Unix könyvtári fa*

Ha egy program úgy akar elérni egy megadott fájlt, hogy az elérés ne függjön attól, mi az aktuális könyvtár, akkor mindig abszolút útvonalmegadást kell használnia. Például a helyesírás-ellenőrző programnak szüksége van a */usr/lib/dictionary* fájlra az ellenőrzés elvégzéséhez. Ekkor a program csak abszolút fájlnévvel hivatkozhat rá, mert nem tudni, hogy akkor mi lesz az aktuális könyvtár, amikor végrehajtják. Az abszolút útvonalmegadás mindig működik, függetlenül attól, hogy éppen mi az aktuális könyvtár.

Természetesen ha a helyesírás-ellenőrző program sok fájlt használ a */usr/lib* könyvtárból, akkor eljárhat úgy is, hogy előbb az aktuális könyvtárat a megfelelő rendszerhívással */usr/lib*-re állítja, majd csak a *dictionary* relatív útvonalmegadást használja. Az aktuális könyvtár explicit megváltoztatása, ismerve a könyvtár abszolút nevét a könyvtárfában, lehetővé teszi relatív nevek használatát.

A legtöbb rendszerben minden processzusnak saját munkakönyvtára van, tehát amikor a processzus befejeződik, akkor nem befolyásolja más processzusok működését, és nem marad vissza változás a fájlrendszerben. Ily módon mindig teljesen biztonságos megváltoztatni az aktuális könyvtárat, ha az kényelmessé teszi a munkát. Másrészről, ha egy *könyvtári eljárás* megváltoztatja az aktuális könyvtárat, és a befejeződése előtt nem állítja vissza az eredeti állapotot, akkor a program további működése bizonytalanná válik, mert az a feltételezése, hogy hol van, érvénytelenné válik. Éppen ezért a könyvtári eljárások ritkán változtatják meg az aktuális könyvtárat, vagy ha mégis megteszik, akkor gondoskodnak arról, hogy befejeződés előtt visszaállítsák az eredeti állapotot. A legtöbb operációs rendszerben, amelynek könyvtárszerkezete hierarchikus, minden könyvtárban van két speciális könyvtári bejegyzés, ezek a . és .., kiejtésük általában pont és pontpont. A pont az aktuális könyvtárra való hivatkozás, a pontpont pedig az aktuális könyvtár ősére hivatkozik. Ezek használatára tekintsük az 5.7. ábrán látható fát. Tegyük fel, hogy egy processzus aktuális könyvtára a */usr/ast*. Ekkor használhatja a .. hivatkozást a fában felfelé haladáshoz. Például a

```
cp ../lib/dictionary .
```

parancs átmásolhatja a */usr/lib/dictionary* fájlt a saját könyvtárába. Az első argumentum arra utasít, hogy menjünk eggyel feljebb a fában, majd lefelé a *lib* könyvtárba, és onnan vegyük a *dictionary* fájlt.

A második argumentum csupán az aktuális könyvtárat nevezi meg. Ha a *cp* parancs második argumentuma egy könyvtár (itt a pont), akkor oda bemásolja az összes fájlt, amelyet az első argumentumban megadtunk. Nyilvánvalóan természetesebb lenne a másolást elvégezni a

```
cp /usr/lib/dictionary .
```

paranccsal. Itt a pont használata megkímélt a *dictionary* szó kétszeri leírásától. Mindazonáltal a

```
cp /usr/lib/dictionary dictionary
```

és a

```
cp /usr/lib/dictionary /usr/ast/dictionary
```

parancs egyaránt helyesen működik.

5.2.4. Könyvtári műveletek

A könyvtárakat kezelő megengedett műveleteket tekintve nagyobb az eltérés ez egyes rendszerekben, mint az a fájlműveletek esetén tapasztalható. A lehetséges műveletek és azok hatásainak szemléltetésére tekintsük az alábbi mintát (a Unix-rendszerből).

1. **LÉTESÍTÉS** (create). Új könyvtárat létesít. A létrehozott könyvtár üres, kivéve a pont és pontpont könyvtárakat, amelyeket a rendszer (vagy néhány esetben az *mkdir* parancs) automatikusan létesít.
2. **TÖRLÉS** (delete). A megnevezett könyvtárat törli. Csak üres könyvtár törölhető. Egy könyvtár akkor és csak akkor üres, ha csak a pont és pontpont alkönyvtárakat tartalmazza, és ezek nem törölhetők.
3. **MEGNYITÁS** (opendir). A művelet végrehajtása után a könyvtár olvasható lesz. Például egy könyvtár összes fájljának listázását végző program először megnyitja a könyvtárat, majd kiolvassa a benne található összes fájl nevét. Mielőtt olvasni kezdenénk egy könyvtárat, előbb meg kell nyitnunk, hasonlóan a fájlok olvasásához.
4. **LEZÁRÁS** (closedir). Miután a könyvtárat olvastuk, le kell zárni, hogy felszabadítsuk a megnyitáskor lefoglalt memóriát.
5. **OLVASÁS** (readdir). Ez a rendszerhívás a megnyitott könyvtár egy bejegyzését adja eredményül. Korábban lehetséges volt könyvtár olvasása a read fájlművelettel, ennek azonban az volt a hátránya, hogy a programozónak ismernie kellett a könyvtárak belső szerkezetét. Ezzel ellentétben a readdir művelet mindig egy könyvtári bejegyzést ad szabványos formában, függetlenül a könyvtár tényleges szerkezetétől.
6. **ÁTNEVEZÉS** (rename). A könyvtárak több tekintetben hasonlítanak a fájlokra, így ugyanúgy átnevezhetők, mint a fájlok.
7. **KAPCSOLÁS** (link). A kapcsolás olyan technika, amely lehetővé teszi, hogy ugyanazok a fájlok több könyvtárban is előforduljanak. A művelethez meg kell adni egy létező fájlnevet és egy útvonalat. A rendszerhívás hatása az lesz, hogy kapcsolás jön létre a létező fájl és a megadott útvonal között. Ily módon ugyanazok a fájlok több könyvtárban is megjelenhetnek. Az ilyen kapcsolást, amely növeli az i-csomópont számlálóját (ez számon tartja, hogy hány könyvtári bejegyzés tartalmazza a fájlt), néha **merev kapcsolás**nak is nevezik.
8. **LEKAPCSOLÁS** (unlink). A művelet egy könyvtári bejegyzést töröl. Ha a törlendő csak egy könyvtárban fordul elő (a normális eset), akkor törlődik a fájlrendszerből. Ha több könyvtárban is előfordul, akkor csak a megnevezett útvonal törlődik, a többi megmarad. Unixban a fájltörlés (korábban tárgyaltuk) valójában lekapcsolást jelent.

A felsorolt műveletek a legfontosabbak, de léteznek egyebek is, például a védelmi információ kezelését végző rendszerhívások.

5.3. Fájlrendszerek megvalósítása

Most már itt az idő, hogy áttérjünk a felhasználói szempontról a megvalósítás szempontjainak tárgyalására. A felhasználókat az érdekli, hogyan lehet fájlokat elnevezni, milyen műveleteket végezhetünk velük, hogyan néz ki a könyvtári fa és hasonló felületi kérdések. Az implementálókat az érdekli, hogyan lehet fájlokat és könyvtárakat tárolni, lemezterületeket kezelni, hogyan lehet elérni azt, hogy az egész hatékonyan és megbízhatóan működjön. A következőkben ezeket a területeket vizsgáljuk, hogy lássuk az előnyöket és a hátrányokat.

5.3.1. Fájlrendszerszerkezet

A fájlrendszereket általában mágneslemeztárolókon tárolják. A lemeztárolók szerkezetét a 2. fejezetben áttekintettük, amikor a MINIX 3 betöltését vizsgáltuk. Röviden felidézzük az ott elhangzottakat. A legtöbb lemez partíciókra van osztva, és minden partíció független fájlrendszert tartalmaz. A lemez 0. szektora, az **MBR (Master Boot Record – elsődleges indítórekord)** a számítógép elindítására használatos. Az MRB végén található a partíciós tábla. Ez a táblázat tartalmazza minden partíció kezdetének és végének a címét. Egy partíciót ki lehet jelölni aktív partícióként. A számítógép indításakor a BIOS betölti és végrehajtja az MBR-ben lévő kódot. Az MBR program először megkeresi az aktív partíciót, majd beolvassa és végrehajtja az aktív partíció első blokkját, amelyet **indítóblokk**nak nevezünk. Az indítóblokk programja tölti be az adott partícióban lévő operációs rendszert. Az egységesség kedvéért minden partíció tartalmaz indítóblokkot, akkor is, ha az adott partíció nem tartalmaz indítható operációs rendszert. Mivel a későbbiek során a partíció tartalmazhat indítható operációs rendszert, ezért az indítóblokk fenntartása mindenképpen jó ötlet.

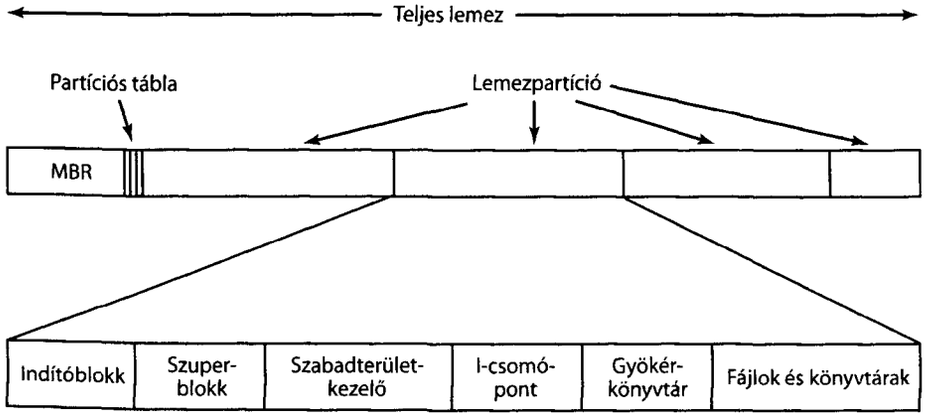
A fent leírtaknak teljesülni kell az operációs rendszertől függetlenül minden olyan hardverplatformra, amelyen a BIOS képes több operációs rendszer betöltésére. A terminológia függhet az alkalmazott operációs rendszertől. Például az elsődleges indítórekord elnevezése lehet **IPL (Initial Program Loader – kezdő programbetöltő)**, **Volume Boot Code – kötetbetöltő kód** vagy egyszerűen **masterboot (elsődleges betöltő)**. Néhány operációs rendszer nem követeli meg, hogy aktív partíciót jelöljünk ki, hanem egy menüt kínál fel, amelyből kiválaszthatjuk, hogy melyik partícióról akarunk indítani, és a várakozási idő letelte után az alap-

értelmezett választás érvényesül. Az MBR-nek a BIOS által történt betöltése után a folytatás eltérő lehet. Például az operációs rendszert betöltő program a partíció egynél több blokkját is elfoglalhatja. A BIOS-nak csak az a feladata, hogy az első blokkot betöltse, de ez a blokk aztán további blokkokat tölthet be, az operációs rendszer megvalósításának megfelelően. A rendszer készítője adhat egy szokványos MBR-t, de ennek működnie kell standard partíciós táblával, ha több operációs rendszert is használunk.

PC-kompatibilis rendszerekben csak négy **elsődleges partíció** lehet, mert csak négyelemű tömb számára van hely az elsődleges indítórekord és az első 512 bájtos szektor vége között. Néhány operációs rendszer lehetővé teszi, hogy a partíciós táblában megjelöljük a partíciót, mint **kiterjesztett partíciót**, amikor is egy mutatót tartalmaz a **logikai partíciók** láncolt listájára. Ez lehetővé teszi, hogy akárhány további partíciót hozzunk létre. A BIOS nem tudja az operációs rendszert indítani logikai partícióról, ezért elsődleges partícióról kell indítani, ami aztán kezelni tudja a logikai partíciókat.

A kiterjesztett partíció egy alternatíváját használja a MINIX 3, amely lehetővé teszi, hogy egy partíció **alpartíciós táblát** tartalmazzon. Ennek az az előnye, hogy ugyanaz a kód, amely az elsődleges partíciókat kezeli, kezelheti az alpartíciókat is, mert ezeknek ugyanaz a szerkezete. Az alpartíciók potenciális felhasználási módja, hogy különböző partíció lehet a gyökér, lapozó, a binárisok és a felhasználói fájlok számára. Így ha probléma lép fel egy partícióval, az nem terjed át más partíciókra, és az operációs rendszer új változata egyszerűen telepíthető lesz néhány, de nem az összes alpartíció átírásával.

Nem minden lemezes tárolót lehet partíciókra osztani. Hajlékonylemez-tárolók általában az első szektorban tartalmazzák az indítóblokkot. A BIOS beolvassa a lemezről az első szektort, megvizsgálja a mágikus számot, hogy ellenőrizze, végrehajtható kódot tartalmaz-e. Ezzel elkerüli egy formázatlan vagy tönkrement lemez első szektorának végrehajtását. Az elsődleges indítóblokknak és az indítóblokknak ugyanaz a mágikus száma, tehát a kód lehet akármelyik. Továbbá, amit



5.8. ábra. *Lehetséges fájlrendszerszerkezet*

eddig mondtunk, az nemcsak elektromágneses lemeztárolókra igaz. Olyan eszközök, mint a digitális fényképezőgépek, PDA-k, amelyek tartós tárolót alkalmaznak (például flash), általában rendelkeznek olyan résszel, amely szimulálja a mágneslemezt.

A lemezpartíciók, attól eltekintve, hogy indítóblokkal kezdődnek a fájlrendszertől függően, nagyon eltérő szerkezetűek lehetnek. A Unix típusú fájlrendszerek az 5.8. ábrán látható elemeket tartalmazhatják. Az első a **szuperblokk**. Ez tartalmazza a fájlrendszer valamennyi fontos paraméterét, és beolvasódik a memóriába a gép indításakor vagy a fájlrendszer első hozzáférésekor.

A következő a fájlrendszer szabad blokkjairól adhat információt. Ezt követheti az i-csomópontok leírása; ez olyan adatszerkezetek tömbje, amely minden fájlhoz egy bejegyzést tartalmaz, megadva a fájl minden szükséges adatát, és hogy hol találhatók a fájl blokkjai. Ezt követheti a gyökérkönyvtár, amely a fájlrendszer fájának gyökere. A partíció további része tartalmazza a könyvtárakat és fájlokat.

5.3.2. Fájlok megvalósítása

A legfontosabb kérdés a fájlok tárolásának megvalósításánál valószínűleg annak nyilvántartása, hogy mely lemezblokkok mely fájlokhoz tartoznak. A különböző operációs rendszerek eltérő módszereket használnak. Ebben a részben néhány ilyen módszert vizsgálunk.

Folytonos helyfoglalás

A legegyszerűbb helyfoglalási séma szerint minden fájl tárolására összefüggő blokkok alkotta lemezterületet használnak. Tehát egy 50 K méretű fájl 50 egymást követő lemezblokkban foglal helyet, ha a blokkméret 1 K. Ennek a sémának két lényeges előnye van. Először is egyszerű megvalósítani, mivel annak nyilvántartása, hogy a fájl hol van tárolva a lemezen, csupán két számot igényel, nevezetesen az első blokk címét és a blokkok számát. Az első blokk címének ismeretében bármely blokk címe egyszerű összeadással kiszámítható.

Másodszor, az olvasás hatékonysága kiváló, mivel a lemezről egyetlen művelettel beolvasható a teljes fájl. Csak egy pozicionálást kell végrehajtani (az első blokk címére). Ezt követően nincs szükség további pozicionálásra, nincs forgási késleltetés, tehát az adatátvitel a lemez teljes átviteli sebességével történhet. Így a folytonos helyfoglalást egyszerű megvalósítani és nagyon hatékony.

Sajnos a folytonos helyfoglalásnak van egy súlyos hátránya is: idővel a lemez töredezetté válik, fájlokat és lyukakat tartalmaz. Kezdetben a töredezettség nem okoz problémát, mivel az új fájlokat a lemez végére, a korábbiak után lehet kiírni. Végül azonban a lemez megtelik, és szükségessé válik a tömörítés, ami megengedhetetlenül költséges, vagy a lyukakat kell újrahasznosítani. Az újrahasznosítás a lyukakat tartalmazó kétirányú lista kezelését igényli. Azonban amikor egy új fájlt létesítünk, tudnunk kell a végső méretét, hogy megfelelő lyukat válasszunk a tárolására.

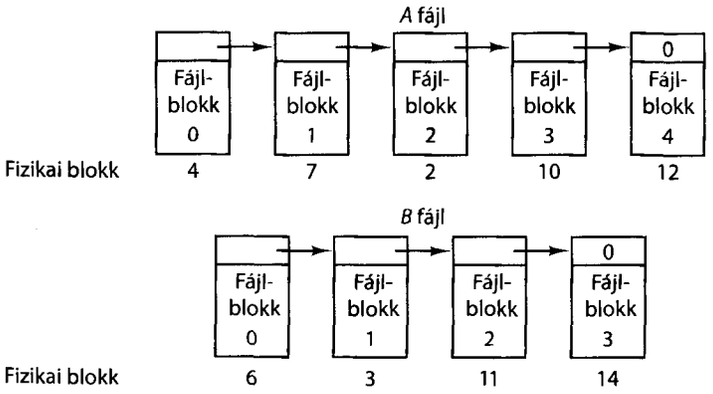
Amint azt már említettük az 1. fejezetben, a történelem ismételheti önmagát az informatikában, amint a technológia új generációja jelentkezik. A folytonos helyfoglalást ténylegesen használták évekkel ezelőtt lemezes fájlrendszerek megvalósítására egyszerűsége és hatékonysága miatt (a felhasználóbarátság akkor nem számított). Aztán elvetették a módszert, mert fájl létrehozásakor tudni kellett a méretét, ami kellemetlen volt. A CD-ROM, DVD és más, csak írható optikai eszköz kifejlesztésének hatására hirtelen ismét jó ötletnek bizonyult a folytonos helyfoglalás. Az ilyen tárolók esetén a folytonos helyfoglalás megfelelő, és valóban széles körben használják is. Ezeknél minden fájl mérete előre ismert, és nem változik a CD-ROM-fájlrendszer újbóli felhasználásáig. Tehát fontos, hogy tanulmányozzuk a régi rendszereket és elképzeléseket, amelyek fogalmilag tiszták és egyszerűek, mert meglepő módon alkalmazhatóvá válhatnak jövőbeli rendszerekben.

Láncolt listás helyfoglalás

A második módszer fájlok tárolására az, amikor a lefoglalt lemezterületet lemezblokkok láncolt listájával adjuk meg, mint az az 5.9. ábrán látható. Minden blokkban az első szó a következő blokkra mutató pointert tartalmazza. A blokk fennmaradó része adatot tárol.

Ellentétben a folytonos helyfoglalással, itt a lemez minden blokkja felhasználható. Nem veszítünk töredezettség miatt (kivéve az utolsó blokk belső töredezettségét). Továbbá minden könyvtári bejegyzésnél elegendő csak az első blokk címét tárolni. A többi a pointereket követve az elsőtől elérhető.

Másrészről, jóllehet a szekvenciális olvasás magától értetődő, a közvetlen elérés rendkívül lassú lesz. Az *n*-edik blokk eléréséhez az operációs rendszernek be kell olvasnia az azt megelőző *n* – 1-ediket. Világos, hogy ilyen sok olvasás bántóan lassú lesz. Az is igaz, hogy az adatblokk mérete már nem lesz kettőhatvány, mert a mutató tárolása elvesz néhány bájtot. Bár nem végzetesen, de az ilyen méret csökkenti a hatékonyságot, mert a programok többsége kettőhatvány méretű adat-



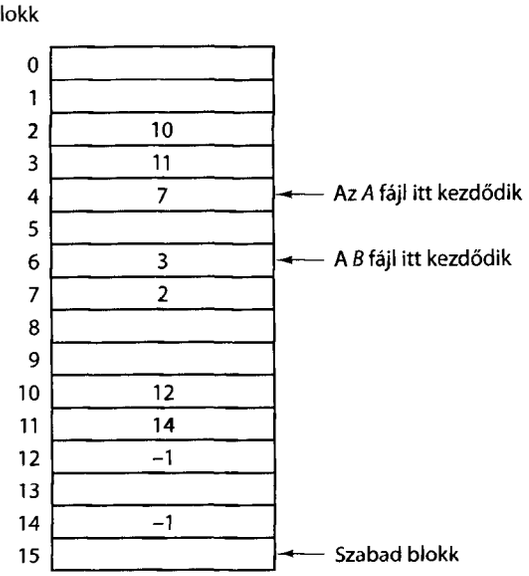
5.9. ábra. *Fájl tárolása blokkok láncolt listájában*

blokkot ír, illetve olvas. Mivel minden blokkban néhány bájtot elfoglal a következő blokkra mutató pointer, ezért egy teljes blokkméretnyi olvasás megköveteli két blokkbeli adat összevonását, ami extraköltséget eredményez a másolás miatt.

Láncolt listás helyfoglalás memóriabeli táblázattal

A láncolt listás helyfoglalás mindkét hátránya kiküszöbölhető, ha a mutatót kivesszük minden blokkból, és egy táblázatban (index) összegyűjtve a memóriában tároljuk. Az 5.10. ábra azt mutatja, hogyan néz ki ekkor a táblázat az 5.9. ábrán adott fájlokra. Mindkét ábrán két fájl látható. Az *A* fájl a 4, 7, 2, 10 és 12 blokkokat használja ebben a sorrendben, míg a *B* fájl a 6, 3, 11 és 14 blokkokat használja, szintén ebben a sorrendben. Az 5.10. ábrán látható indextáblát használva, a 4. blokkal indulva és a láncolást követve sorrendben a többi blokk is elérhető. Hasonlóan a 6. blokkal indulva a *B* fájl blokkjai elérhetők. Mindkét lánc a –1 speciális mutató értékkel végződik, amely nem lehet valódi blokk címe. Az ilyen táblázat neve **FAT (File Allocation Table – fájlhelyfoglalási táblázat)**.

Ezt a szervezést használva a teljes blokkméret rendelkezésre áll adat tárolására. Továbbá a közvetlen elérés is sokkal egyszerűbb. Bár most is a láncolást kell követni ahhoz, hogy egy adott fájlpozíció lemezcímét megkapjuk, de az egész lánc a memóriában van, így nem kell lemezhez fordulni. Mint az előző módszernél, itt is elegendő minden könyvtári bejegyzésnél csak az első blokk címét tárolni, a többi ebből elérhető, függetlenül a fájl nagyságától. Az elsődleges hátránya ennek a módszernek az, hogy a teljes táblázatnak a memóriában kell lennie a működéshez.



5.10. ábra. *Láncolt listás helyfoglalás fájlhelyfoglalási táblázat használatával a memóriában*

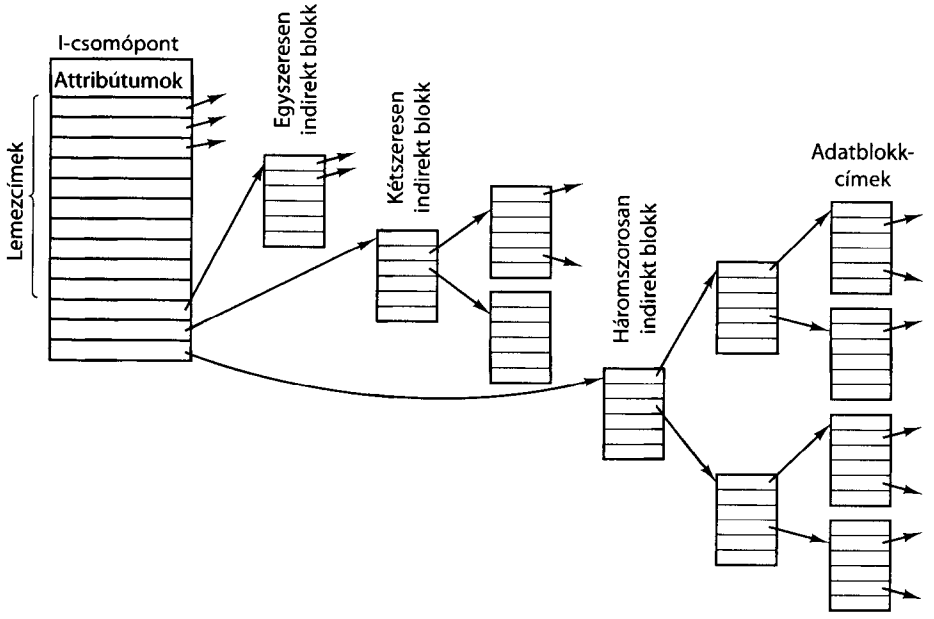
Egy 20 GB kapacitású lemez 1 KB méretű blokkokkal 20 millió elemű táblázatot igényel. Minden elem legalább 3 bájt méretű, de gyorsabb elérés miatt 4 bájt is lehet. Tehát a táblázat 60 MB vagy 80 MB helyet igényel a főtárban, attól függően, hogy tárra vagy sebességre optimalizálunk. Elképzelhető a táblázatnak lapozható memóriába helyezése, de ekkor számolni kell a lapozási forgalom növekedésével. Az MS-DOS és a Windows 98 kizárólag FAT fájlrendszert használ, amit a későbbi Windows-verziók is támogatnak.

I-csomópont

Az utolsó helyfoglalási módszer, amelyet tárgyalunk, az ún. **i-csomópont (index-csomó** vagy **i-csomó)** módszere. Ennél minden fájlhoz tartozik egy kis táblázat, az i-csomópont, amelyben a fájl attribútumait és a fájlhoz tartozó blokkok lemezcímét tároljuk. Ezt mutatja az 5.11. ábra.

Az i-csomópont ismeretében a fájl minden blokkja elérhető. A nagy előnye ennek a módszernek a memóriatáblázatot használó láncolt listással szemben, hogy csak az i-csomópontnak kell a memóriában lennie, ha a fájl meg van nyitva. Ha minden i-csomópont n bájtos, és legfeljebb k fájl lehet egyidejűleg nyitva, akkor csak kn bájt szükséges az i-csomópontok memóriában tárolásához – csak ennyi memóriát kell előre lefoglalni.

Ennek a tömbnek a mérete messze kisebb, mint az előző módszernél szükséges táblázat mérete. Az ok egyszerű. A láncolt listás táblázat mérete arányos a lemez



5.11. ábra. *I-csomópont három indirektblokk-szinttel*

kapacitásával. Ha a lemez n blokkból áll, akkor a táblázatnak n eleme van. A táblázat mérete lineárisan nő a lemez méretével. Ezzel szemben az i-csomópontos sémánál a szükséges memória mérete lineárisan arányos az egyidejűleg nyitva lehető fájlok számával. Nem számít, hogy a lemez 1 GB, 10 GB vagy 100 GB kapacitású.

Az a probléma az i-csomóponttal, hogy mindegyik rögzített számú lemezcímet tartalmaz. Mit lehet tenni, ha a fájl mérete túlhaladja ezt? Az egyik megoldás az, hogy az utolsó cím nem adatblokk címe lesz, hanem egy **indirekt blokk** címe, amely további lemezblokkcímeket tartalmaz. Ez az ötlet tovább folytatható, **kétszeresen indirekt, háromszorosan indirekt** blokkok alkalmazásáig. Ezt mutatja az 5.11. ábra.

5.3.3. Könyvtárak megvalósítása

Tudjuk, hogy mielőtt egy fájlból olvasnánk, azt meg kell nyitni. Amikor egy fájl megnyitásra kerül, az operációs rendszer a felhasználó által megadott fájlnév alapján megkeresi a hozzá tartozó könyvtári bejegyzés helyét. Először természetesen a gyökérkönyvtárat kell megtalálni. A gyökérkönyvtár lehet a partíció kezdetéhez képest rögzített helyen. Alternatív megoldásként a helye meghatározható más információkból, például a klasszikus Unix-fájlrendszerben a szuperblokk tartalmaz információt az adatszerkezet méretéről, ami megelőzi az adatterületet. A szuperblokkban megtalálható az i-csomópontok helye. Az első i-csomópont mutat a gyökérkönyvtárra, amely a Unix-fájlrendszer létesítésekor keletkezett. Windows XP esetén az indítószektor (amely lényegesen nagyobb, mint egy közönséges szektor) tartalmazza az **MFT**-t **(Master File Table – mesterfájltáblázat)**, amely alapján megtalálhatjuk a fájlrendszer többi részének helyét. Miután megtaláltuk a gyökérkönyvtárat, a könyvtárfa bejárásával megtalálhatjuk a keresett bejegyzést. A könyvtári bejegyzés tartalmazza azokat az információkat, amelyek szükségesek a fájl számára foglalt lemezblokkok megkereséséhez. Rendszertől függően ez vagy a fájl által elfoglalt teljes lemezterület címe (folytonos helyfoglalás esetén), vagy az első blokk címe (mindkét láncolt listás módszernél), vagy az i-csomópont száma. Mindegyik esetben a könyvtári rendszer fő feladata, hogy a szöveges fájlnévhez hozzárendelje azt az információt, amely az adatok eléréséhez kell.

Ehhez szorosan kapcsolódik az attribútumok tárolásának kérdése. Minden fájlrendszer kezel olyan attribútumokat, mint a fájl tulajdonosa, létrehozási dátum, amit valahol tárolni kell. Az egyik nyilvánvaló lehetőség szerint magában a könyvtári bejegyzésben. A legegyszerűbb formában a könyvtár fix méretű bejegyzést tartalmaz minden egyes fájlhoz, amely tartalmazza a fájl nevét (fix hosszban), az attribútumokat és egy vagy több (valami felső korlátig) lemezcímet, amelyek a blokkok helyét adják meg. Ez látható az 5.5.(a) ábrán.

Azok a rendszerek, amelyek i-csomópontot alkalmaznak, tárolhatják az attribútumokat magában az i-csomópontban, nem pedig a könyvtári bejegyzésben. Ezt mutatja az 5.5.(b) ábra. Ekkor egy könyvtári bejegyzés rövidebb lehet, csak a fájlnevet és az i-csomópont számát tartalmazza.

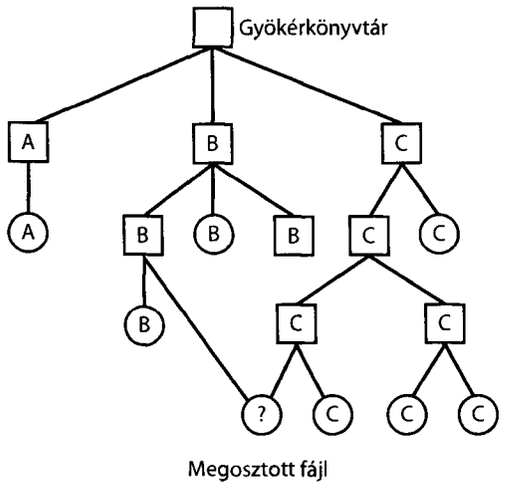
Megosztott fájlok

Az 1. fejezetben röviden megemlítettük a fájlok **kapcsolását (link)**, amely lehetővé teszi, hogy több, együtt dolgozó felhasználó megosszon fájlokat. Az 5.12. ábra mutatja ismét az 5.6.(c) ábrán látható fájlrendszert, ahol most a *C* felhasználó egy fájlja a *B* felhasználó könyvtárában is megtalálható.

Unixban, ahol a fájlattribútumokat i-csomópont tárolja, a megosztás könnyen megvalósítható; akárhány könyvtári bejegyzés mutathat ugyanarra az i-csomópontra. Az i-csomópontban van egy mező, amelynek értéke eggyel növekszik minden olyan alkalommal, amikor új kapcsolat létesül rá, illetve eggyel csökken, ha kapcsolatot törlünk. Csak akkor törlődik ténylegesen maga az i-csomópont és az adat, amikor ez a számláló nulla lesz.

Az ilyenfajta kapcsolást **merev kapcsolás**nak **(hard link)** nevezik. Nem mindig lehetséges azonban a fájlokat merev kapcsolással megosztani. A legfontosabb korlátozást az jelenti, hogy a könyvtárak és i-csomópontok egy adott fájlrendszer (partíció) adatszerkezetéhez tartoznak, tehát egy fájlrendszer könyvtára nem mutathat egy másik fájlrendszer i-csomópontjára. Továbbá minden fájlnak csak egy tulajdonosa és jogosultsága lehet. Ha a megosztott fájl tulajdonosa törli a fájlhoz tartozó könyvtári bejegyzését, a másik felhasználó fennakad, mert a jogosultságok miatt nem törölheti a saját könyvtárából.

Fájlok megosztásának alternatív megoldása az, amikor egy új típusú fájlt létesítünk, amely egy másik fájl elérési útvonalát tartalmazza. Ez a megoldás működik különböző fájlrendszerek között is. Valójában ha van olyan lehetőség, amellyel meg lehet adni hálózati címet is, akkor ezzel a módszerrel másik számítógépen lévő fájlra is lehet hivatkozni. Ezt a megoldást **szimbolikus kapcsolás**nak **(symbolic link)** nevezik a Unix-rendszerekben, **rövidút**nak **(shortcut)** a Windowsban és **álnév (alias)** az Apple Mac OS-rendszerben. Szimbolikus kapcsolás használható



5.12. ábra. *Megosztott fájlt tartalmazó fájlrendszer*

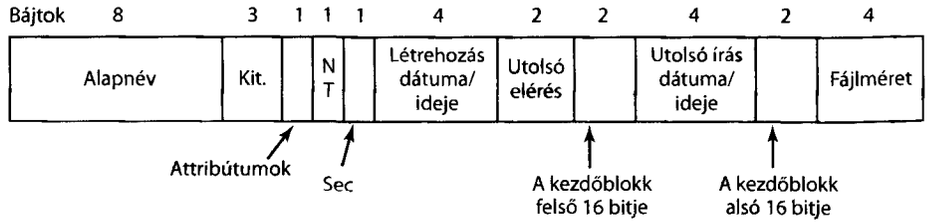
olyan rendszerekben is, ahol az attribútumokat a könyvtári bejegyzés tartalmazza. Kis gondolkodás meggyőzhet arról, hogy bonyolult lenne szinkronizálni többszörös könyvtári bejegyzések attribútumait. A fájl minden módosítása az erre a fájlra hivatkozó összes bejegyzés módosítását vonná maga után. De a szimbolikus kapcsolást megvalósító extra könyvtári bejegyzés nem tartalmaz a hivatkozott fájlra vonatkozó attribútumot. A szimbolikus kapcsolás hátránya, hogy ha töröljük a fájlt, vagy csak átnevezzük, akkor a hivatkozás érvényét veszti.

Windows 98-könyvtárak

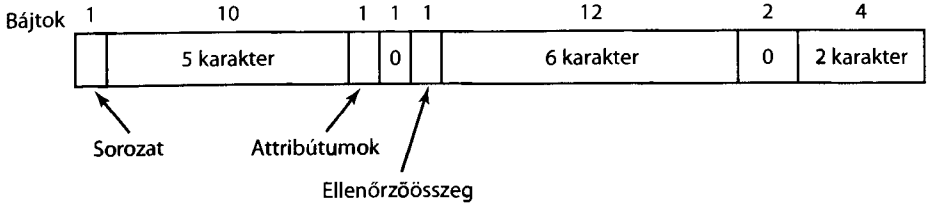
A Windows 95 eredeti fájlrendszere azonos volt az MS-DOS rendszerével, de második kiadása már támogatta a hosszabb fájlneveket és nagyobb fájlokat. Erre a rendszerre a Windows 98 elnevezéssel hivatkozunk, jóllehet megtalálható bizonyos Windows 95 rendszerben is. A Windows 98-ban kétféle könyvtári bejegyzés létezik. Az elsőt, amelyet az 5.13. ábra mutat, alapbejegyzésnek nevezzük.

Az alapbejegyzés tartalmazza mindazokat az információkat, amelyeket a Windows korábbi változatai, és még néhány egyebet. Az *NT* mezővel kezdődő 10 bájt a régebbi Windows 95-ben nem volt használatos, szerencsére (vagy a későbbi változatok ismeretében inkább szándékosan). A legfontosabb javítás az, hogy a kezdő blokk címe 16 helyett 32 biten adható meg. Ez 2^{16} blokkról 2^{32} blokkra növelte a fájlrendszer maximális méretét.

Ez a séma csak az MS-DOS (és CP/M) régi típusú, 8 + 3 karakteres fájlneveit támogatja. Hogyan kezeli a hosszabb fájlneveket? A hosszabb fájlnevek kezelése, fenntartva a régi rendszerrel való kompatibilitást, további könyvtári bejegyzésekkel oldható meg. Az 5.14. ábra egy lehetséges megoldást mutat olyan könyvtári bejegyzésre, amely 13 karaktert tartalmaz hosszú fájlnevek számára. Hosszú nevek számára a rendszer automatikusan generál rövidített formát, amit az 5.13. ábrán mutatott könyvtári bejegyzés *Alapnév* és a *Kit.* mezőin helyez el. Hosszú nevekhez annyi könyvtári bejegyzést használ, amennyi szükséges, az alapbejegyzés előtt helyezi el fordított sorrendben. Minden hosszú bejegyzés *Attribútum* mezője a 0x0F értéket tartalmazza, ami nem érvényes érték a régebbi (MS-DOS és Windows 95) rendszerekben, tehát ha egy régebbi rendszer olvassa (például egy hajlékonylemezen), akkor figyelmen kívül hagyja. A *Sorozat* mező egy bitje mondja meg, hogy melyik az utolsó bejegyzés.



5.13. ábra. *A Windows 98 alap könyvtári bejegyzése*



5.14. ábra. *Hosszú fájlnév (részlet) a Windows 98-ban*

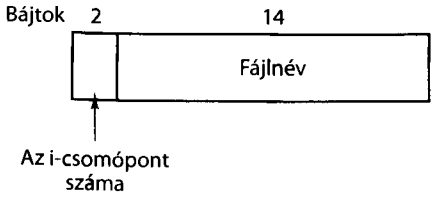
Ez elég bonyolultnak látszik, és valóban az: fenntartani a visszafelé kompatibilitást, hogy a régebbi rendszer működhessen, miközben további új funkciókat kell biztosítani az új számára. A tiszta megoldás hívei nem mennének bele ilyen bonyodalmakba, de ők valószínűleg nem is gazdagodnak meg új operációs rendszerek eladásából.

Unix-könyvtárak

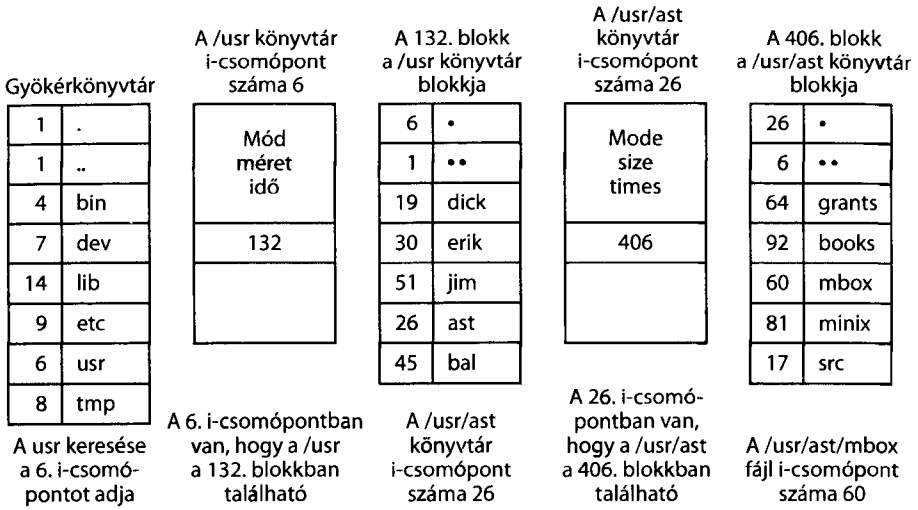
A tradicionális Unix-könyvtárszerkezet különösen egyszerű, mint az 5.15. ábrán látható. Minden bejegyzés csak a fájlnevet és a hozzá tartozó i-csomópont sorszámát tartalmazza. Minden információt, mint a típus, méret, tulajdonos és a lemezblokkok címei, az i-csomópont tartalmaz. Néhány Unix-rendszerben más az elrendezés, de minden esetben végső soron a bejegyzés csak a nevet, mint szöveget és az i-csomópont sorszámát tartalmazza.

A fájl megnyitásakor a megadott név alapján a rendszernek meg kell keresnie a fájlhoz tartozó lemezblokkokat. Nézzük meg, hogyan keres a rendszer a */usr/ast/mailbox* útvonal esetén. Példánkban a Unixot vettük alapul, de alapvetően hasonló algoritmust használ minden hierarchikus könyvtári rendszer. Először a rendszer a gyökérkönyvtárat keresi meg. Az i-csomópontok egyszerű tömböt alkotnak, amelynek helyét a szuperblokkban találjuk. A tömb első eleme a gyökérkönyvtár i-csomópontja.

A fájlrendszer megkeresi az útvonal első, *usr* eleméhez tartozó bejegyzést a gyökérkönyvtárban, hogy megtudja annak i-csomópont számát. Az i-csomópont elérése a lemezen egyszerű, mert minden i-csomópont lemezcíme kiszámítható a sorszámából, ugyanis az i-csomópontokat a lemezen rögzített helyen folytonosan tárolják.



5.15. ábra. *Unix V7 könyvtári bejegyzés*



5.16. ábra. *A /usr/ast/mbox keresésének lépései*

Az i-csomópontból megállapítja a */usr* könyvtár helyét a lemezen, ahol megkeresi az útvonal második elemét, az *ast*. Az *ast*-hez tartozó könyvtári bejegyzésben megtalálja a */usr/ast* könyvtár i-csomópontjának számát. Ezt az i-csomópontot kiolvasva megtudja magának a könyvtárnak a helyét a lemezen, amelyben megkeresi az *mbox*-hoz tartozó bejegyzést. Ennek i-csomópontját ezután beolvassa a memóriába, és ott is tartja mindaddig, amíg a fájlt le nem zárják. A keresési eljárást illusztrálja az 5.16. ábra.

A relatív útvonal keresése teljesen hasonló, csak a keresés nem a gyökérkönyvtártól indul, hanem az aktuálistól. Tudjuk, hogy minden könyvtár tartalmazza a . és .. bejegyzéseket, amelyek a létesítéskor keletkeznek. A . bejegyzés tartalmazza az aktuális könyvtár i-csomópontjának számát, a .. bejegyzés pedig az őskönyvtár i-csomópontjának számát. Tehát a *../dick/prog.c* bejegyzés keresése az aktuális könyvtárban a .. keresésével kezdődik, az őskönyvtár i-csomópontjának ismeretében keresi a *dick* könyvtárat és aztán abban a *prog.c* fájlt. Nincs szükség speciális mechanizmusra az ilyen esetek kezeléséhez. Ami a könyvtári rendszert illeti, a nevek itt is egyszerű ASCII szövegek, mint más esetekben.

NTFS-könyvtárak

A Microsoft alapértelmezett fájlrendszere az **NTFS (New Technology File System – új technológiájú fájlrendszer)**. Nincs lehetőségünk az NTFS részletes leírására, csak röviden áttekintjük a rendszert érintő problémákat és megoldási módjait.

Az egyik probléma a hosszú fájlnevek és útvonalak. Az NTFS megenged hosszú fájlneveket (255 karakterig) és útvonalakat (32 767 karakterig). Mivel a régebbi Windows-rendszerek nem tudják olvasni az NTFS-t, így a visszafelé kompatibili-

tás miatt nincs szükség bonyolult könyvtári szerkezetre, és a fájlneveket tartalmazó mezők változó hosszúságúak. Egy másodlagos 8 + 3 karakteres név biztosítja, hogy a régebbi rendszerek olvasni tudják a hálózatban megosztott NTFS-fájlokat.

Az NTFS a Unicode használatával biztosítja a többszörös karakterkészlet alkalmazását fájlnevek képzésére. A Unicode 16 biten tárolja a karaktereket, ami elegendően nagyméretű jelkészletet ad különböző nyelvek számára (például japán). Azonban a többszörös nyelv megengedése az eltérő karakterkészlet mellett további problémát okoz. Még a latin alapú nyelvek esetén is gond van. Például spanyolban rendezés esetén néhány karakterkombináció egyetlen betűnek számít. A ch vagy ll betűkkel kezdődő szavakat megelőzik a cz, illetve lz betűkkel kezdődők. A kisbetű-nagybetű leképezés még bonyolultabb. Ha az alapértelmezés szerint a fájlnevekben meg is különböztetjük a kis- és nagybetűket, szükség lehet ezt figyelmen kívül hagyó keresésre. Latin alapú nyelvek esetén ez nyilvánvaló, legalábbis a felhasználók anyanyelvét tekintve. Általánosan, ha csak egyetlen nyelvet használunk, akkor a felhasználók valószínűleg értik a szabályt. Azonban a Unicode lehetővé teszi nyelvek vegyes használatát, görög, orosz és japán nyelvű fájlnevek egyaránt előfordulhatnak egy nemzetközi szervezet által használt könyvtárban. Az NTFS azt a megoldást használja, hogy minden fájl tartalmaz egy attribútumot a fájlnév kisbetű-nagybetű konverziójára.

Sok problémát az NTFS több attribútum bevezetésével old meg. Unixban minden fájl egy bájtsorozat. NTFS-ben a fájl attribútumok gyűjteménye, ahol minden attribútum egy bájtsorozat. Az NTFS alap adatszerkezete az **MFT (Master File Table – mesterfájltáblázat)**, amely 16 attribútumot tartalmazhat, mindegyik 1 KB maximális méretű lehet. Ha ez nem elég, akkor egy MFT attribútumban megadhatunk egy mutatót, amely olyan fájlra mutat, amely az attribútum folytatását tartalmazza. Ezt **nem rezidens attribútum**nak hívják. Az MFT maga is egy fájl, amely a fájlrendszerben minden fájl és könyvtár számára tartalmaz egy bejegyzést. Mivel az MFT nagyon nagyra is nőhet, NTFS-fájlrendszer létesítésekor a partíció kb. 12,5%-a lefoglalódik az MFT növelése számára. Ezért töredezettség nélkül növekedhet, legalábbis amíg el nem fogy a lefoglalt terület. Ha elfogy, akkor egy újabb, nagyobb terület foglalódik le számára. Így amikor az MFT töredezetté válik, akkor kevés számú nagyméretű töredékből áll.

Mi a helyzet az NTFS-ben az adatokkal? Az adat csupán egyfajta attribútum. Valójában egy NTFS-fájl egynél több adatsort is tartalmazhat. Ez a tulajdonság eredetileg azt szolgálta, hogy Windows-szerverek Apple Macintosh-klienseknek is nyújthassanak fájlmegosztást. Az eredeti Macintosh operációs rendszerben (a Mac OS 9-től) a fájlok két adatsort tartalmaztak: erőforrássort és adatsort. A többszörös adatsornak más felhasználása is van, például nagyméretű grafikus állomány esetén egy kisméretű, szimbolizáló kép rendelhető hozzá. Minden adatsor maximálisan 2^{64} bájt méretű lehet. A másik véglet az, hogy az NTFS kisméretű, néhány százbájtos fájlokat magában az attribútumfejben tud tárolni. Ezt **közvetlen fájl**nak nevezik (Mullender és Tanenbaum, 1984).

Csak érintettünk néhány olyan kérdést, amellyel az NTFS-t megelőző egyszerűbb rendszerek nem foglalkoztak. Az NTFS kifinomult védelmi, titkosítási és adattömörítő képességgel is rendelkezik. Mindezen tulajdonságok leírása és a megvalósítás tárgyalása nem fér bele e könyv kereteibe. Alaposabb tárgyalását lásd (Tanenbaum, 2001), vagy elérhető a világhálón.

5.3.4. Lemezterület-kezelés

A fájlokat rendszerint mágneslemezen tárolják, ezért a rendszer tervezői számára a fő szempont a lemezterület kezelése. *n* bájtból álló fájl tárolására két általános stratégia használatos: *n* bájt hosszú, folytonosan lefoglalt lemezterület, illetve nem feltétlenül összefüggő blokkokból álló lemezterület. Az összefüggés hasonló, mint a szegmentált, illetve lapozott memóriát kezelő rendszerek esetén.

Az összefüggő helyfoglalás esetén, amint azt már láttuk, az a nyilvánvaló probléma, hogy a fájl növekedésekor valószínűleg a lemezen más helyre kell átmásolni a fájlt. Ugyanez a probléma a szegmensekkel, csakhogy a memóriamező átmásolása viszonylag gyors művelet a lemezen való másoláshoz képest. Éppen ezért majdnem minden fájlrendszer azonos méretű darabokra tördelve tárolja a fájlokat, mely darabok nem feltétlenül szomszédosak a lemezen.

Blokkméret

Ha eldöntöttük, hogy azonos méretű blokkokban tároljuk a fájlokat, azonnal felvetődik a kérdés, hogy mekkora legyen a blokk mérete. Tekintve a lemezek szervezését, nyilvánvalóan helyfoglalási egység lehetne a szektor, a sáv és a cilinder. Lapozást alkalmazó rendszerekben a lapméret a fő vitatéma. Azonban egy nagyméretű helyfoglalási egység, mint a cilinder választása, azt jelentené, hogy minden, akár 1 bájt hosszú fájl számára egy teljes cilindert lefoglalnánk. Másrészről, kicsi blokkméret esetén minden fájl sok blokkból állna. Kis blokkból álló fájl olvasása lassú, mert rendszerint minden blokk olvasása egy pozicionálást és egy körbefordulást igényel.

Példaként tekintsünk egy olyan lemezt, amelyben egy sáv kapacitása 131 072 bájt, a körülfordulási idő 8,33 ms, az átlagos pozicionálási idő pedig 10 ms. Ekkor *k* bájt méretű blokk beolvasásának ideje ezred másodpercben

$$10 + 4{,}165 + (k/131072) \times 8{,}33$$

ami a pozicionálási idő, a rotációs késleltetés és az átviteli idő összegeként adódik. Az 5.17. ábrán látható folytonos görbe az adatátviteli arányt ábrázolja a blokkméret függvényében.

A tárolási hatékonyság kiszámításához meg kell becsülni az átlagos fájlméretet. Korábbi vizsgálatok azt mutatták, hogy Unixban az átlagos fájlméret 1 KB (Mullender és Tanenbaum, 1984). Egy 2005-ben végzett felmérés szerint a könyv egyik szerzőjének munkahelyén, ahol 1000 felhasználó több mint 1 millió Unix-fájlt használt, a medián méret 2475 bájt, ami azt jelenti, hogy a fájlok fele kisebb, fele

pedig nagyobb, mint 2475. Mellékesen, a medián jobb mérték, mint az átlag, mert nagyon kevés fájl is jelentősen módosíthatja az átlagot, de a mediánt nem. Néhány 100 MB-os hardverkézikönyv vagy bemutatóvideó nagymértékben torzíthatja az átlagot, de a mediánt nem.

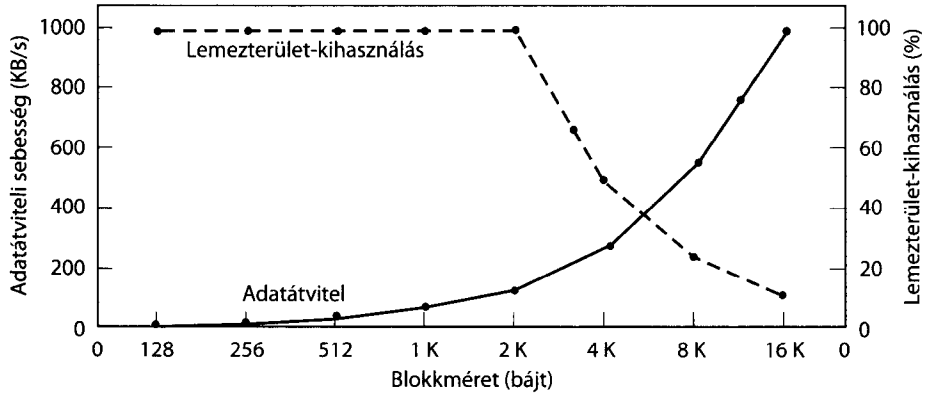
Vogels (Vogels, 1999) vizsgálatai a Cornell Egyetemen azt mutatják, hogy a Windows NT fájlhasználata jelentősen különbözik a Unix-fájlhasználattól. Megfigyelte, hogy az NT fájlhasználata bonyolultabb a Unixnál. A következőket írja:

> *Amikor néhány karaktert begépelünk a* jegyzettömb *szövegszerkesztőben, aztán elmentjük egy fájlba, akkor ez 26 rendszerhívást eredményez, amelyből 3 hibás megnyitás, 1 fájl felülírás és 4 további megnyitás és lezárás művelet lesz.*

Ennek ellenére azt az eredményt kapta, hogy a (használattal súlyozott) medián fájlméret csak olvasásra 1 KB, csak írásra 2,3 KB és írás-olvasásra 4,2 KB. Azt a tényt figyelembe véve, hogy a Cornell széles körű tudományos számításokat is végez, továbbá a mérési módszer eltérő, a 2 KB körüli medián konzisztens eredménynek tekinthető.

Ha azzal az egyszerűsítő feltétellel élünk, hogy minden fájl mérete 2 KB, az 5.17. ábrán látható szaggatott görbét kapjuk a lemez tárkihasználási hatékonyságára.

A két görbe a következőképpen értelmezhető. A blokkelérési idő gyakorlatilag a pozicionálási és forgási idő függvénye, így tekintettel arra, hogy 14 ms idő kell egy blokk eléréséhez, minél több adatot mozgatunk, annál jobb. Tehát az adatátviteli arány növekszik a blokkméret növelésével (amíg az átvitel olyan nagy nem lesz, amikor már az átviteli idő dominál). Kis blokkoknál, amelyek mérete kettőhatvány, és 2 KB-os fájl esetén nincs veszteség. 2 KB-os fájl és 4 KB vagy nagyobb blokk esetén azonban lemezterületet vesztünk. A valóságban kevés fájl mérete lesz pontosan többszöröse a blokkméretnek, tehát mindig lesz veszteség az utolsó blokkban.

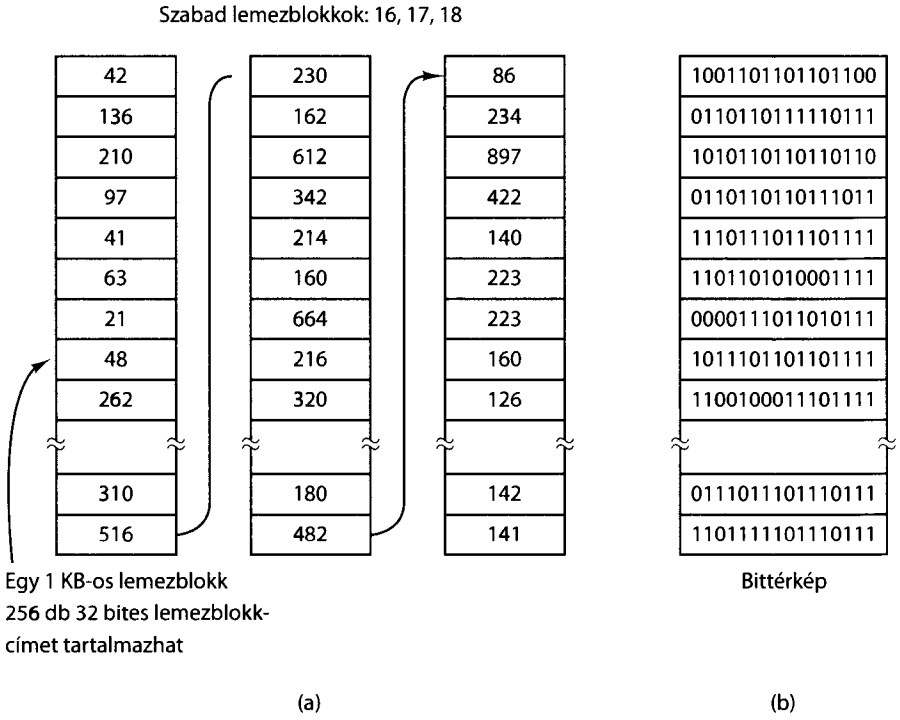


5.17. ábra. *A folytonos görbe (bal oldali skála) a lemez adatátviteli sebessége. A szaggatott görbe (jobb oldali skála) a lemezterület-kihasználási hatékonyság. Minden fájl mérete 2 KB*

A görbék azonban azt mutatják, hogy a tárkihasználás és az átviteli hatékonyság szükségszerűen ellentétesek. Kisméretű blokk esetén rossz az átviteli hatékonyság, de jó a tárkihasználás. Kompromisszumot kell találni a blokkméret megállapításánál. A 4 KB jó választás lehet, azonban néhány operációs rendszer már régen rögzítette a blokkméretet, amikor a fájlméretek még mások voltak. Unix esetén az 1 KB az általánosan alkalmazott. MS-DOS esetén bármely 512 és 32 KB közötti kettőhatvány lehet, de a lemez kapacitásának függvénye, és ennek nincs köze a fenti érveléshez (minden partícióban a blokkok maximális száma 2^{16}, ami nagy blokkméretet követel nagy lemezeknél).

A szabad blokkok nyilvántartása

Miután megválasztottuk a blokkméretet, a következő megoldandó dolog, hogy a szabad blokkokat hogyan tartsuk nyilván. Két módszert használnak széles körben, amit az 5.18. ábra szemléltet. Az első módszer szerint a szabad blokkokat blokkok láncolt listájában tárolják, minden blokk annyi szabad blokk címét tartalmazza, amennyi csak elfér benne. 1 KB blokkmérettel számolva, és feltéve, hogy 32 bit elég egy blokkcím tárolására, egy blokkban 255 blokkcím lesz. (Megjegyzendő, hogy egy elemet a láncolás megvalósítására használunk.) Így egy 256 GB kapacitású lemez



5.18. ábra. *(a) Szabad blokkok láncolt listában. (b) Bittérkép*

szabad listája legfeljebb 1 052 689 blokkból áll, ami elég az összes 2^{28} lemezblokk számára. Gyakran szabad blokkokat használnak a szabad lista tárolására.

A másik technika bittérképet használ a szabad blokkok nyilvántartására. Ekkor egy n blokkot tartalmazó lemez szabad blokkjainak kezelése n bitet igényel. A szabad blokkokat 1, a lefoglaltakat 0 jelöli (vagy fordítva). Így egy 256 GB kapacitású lemezen 2^{28} 1 KB-os blokk van, így bittérképe 2^{28} bitet igényel, ami 32 768 blokk. Nem meglepő, hogy a bittérkép tárigénye kisebb, mivel blokkonként csak 1 bitet igényel, ellentétben a láncolt lista 32 bites igényével. Csak akkor lesz kisebb a tárigény láncolt lista esetén, ha a lemez majdnem tele van (tehát kevés szabad blokk van). Másrészről, ha sok szabad blokk van, akkor egyet kölcsönvehetünk a listaelem számára, tehát nem vesztünk lemezterületet.

A szabad listás módszer esetén csak egy listaelem blokkját kell a memóriában tartani. Fájl létesítésekor ebben a blokkban lévő blokkcímeket használhatjuk az új fájl számára. Ha ez kifogy, akkor beolvassuk a memóriába a lista következő elemét. Hasonlóan fájl törlésekor a fájl blokkjait hozzáadjuk a memóriában lévő listaelemhez, ha ez megtelik, akkor a listaelemet kiírjuk a lemezre.

5.3.5. Fájlrendszerek megbízhatósága

A fájlrendszer meghibásodása gyakran nagyobb bajt jelent, mint magának a számítógépnek a meghibásodása. Ha a számítógép meghibásodik, például tűz, villámlás miatt, vagy egy csésze kávé borul a billentyűzetre, az bosszantó, de általában minimális ráfordítással kicserélhető a meghibásodott alkatrész. Az olcsó személyi számítógépek helyett akár új gép is vásárolható néhány óra alatt, csak el kell menni egy kereskedőhöz (kivéve az egyetemeken, ahol a megrendeléshez három bizottság, öt aláírás és 90 nap kell).

Ha a számítógép fájlrendszere végérvényesen elromlik, akár hardver-, akár szoftverokok miatt, vagy ha patkányok szétrágják a szalagos mentéseket, az információ helyreállítása bonyolult, időigényes és sokszor lehetetlen. Azon személyek számára, akiknek programjai, dokumentumai, vásárlói adatai, adózási tételei, adatbázisai, piaci tervei vagy más adatai mindörökre elvesznek, a következmények katasztrofálisak lehetnek. Jóllehet a fájlrendszer nem adhat védelmet az eszközök fizikai meghibásodásával szemben, de segítheti az információ védelmét. Ebben a részben a fájlrendszer néhány védelmi kérdésével foglalkozunk.

A hajlékonylemezek általában kifogástalanok, amikor kikerülnek a gyárból, de használat közben keletkezhetnek hibás blokkok. Érdemes megjegyezni, hogy ez manapság inkább igaz, mint akkor volt, amikor széles körben használatosak voltak. Hálózatok és nagy kapacitású hordozható eszközök, mint az írható CD-k, elterjedésével a hajlékonylemezeket ritkábban használjuk. A hűtőventilátorok szele, a szennyezett levegő keresztülmegy az eszközmeghajtón, így a ritkán használatos meghajtó olyan szennyezett lehet, hogy tönkreteszi a hajlékonylemezt. A gyakran használt meghajtó kisebb eséllyel károsítja a lemezt.

A merevlemezek kezdettől fogva tartalmazhatnak hibás blokkokat; ennek az az oka, hogy túlságosan költséges lenne a minden hibától mentes gyártás. Amint

azt a 3. fejezetben láttuk, a merevlemez hibás blokkjait általában maga a vezérlő kezeli, helyettesíti ezeket erre a célra fenntartott tartalékkal. Ezeken a lemezeken minden sáv legalább egy szektorral nagyobb, mint kellene, így legalább egy hibás hely kihagyható két szektor közötti hézagként. Továbbá cilinderenként van néhány tartalék szektor, így a vezérlő automatikusan áthelyezheti a hibás blokkot, ha úgy találja, hogy a szokásosnál többször kell ismételten olvasni vagy írni. Tehát a felhasználónak általában nem kell foglalkoznia a hibás blokkok kezelésével. Mindazonáltal, ha egy modern IDE- vagy SCSI-lemeznél hiba lép fel, akkor az végzetes, mert kifogynak a tartalék szektorok. Az SCSI-lemez jelzi a felfedezett hibát (recovered error), amikor áthelyez egy blokkot. Ha ezek az üzenetek gyakorivá válnak, akkor a felhasználó tudhatja, hogy itt az idő új lemezt vásárolni.

A régebbi lemezeknél van egy egyszerű szoftvermegoldás a hibás blokkok kezelésére. A felhasználónak vagy a fájlrendszernek gondosan össze kell gyűjtenie az összes hibás blokkot egy fájlba. Ezáltal ezek az adatok törlődnek a szabad listából, így azok a jövőben nem használhatók adatok tárolására. Amíg ezt a hibásblokk-fájlt nem olvassuk és írjuk, nem lesz gond. Vigyázni kell azonban, amikor mentést készítünk, hogy ezt a fájlt ne olvassuk, és ne próbáljuk menteni.

Mentések

A legtöbb ember nem gondol arra, hogy megéri időt és fáradságot szentelni a fájlok mentésére, mindaddig, amíg hirtelen tönkre nem megy a lemeze. A vállalatok azonban (általában) tisztában vannak adataik értékével, ezért rendszerint legalább naponta készítenek mentést, szokásosan mágnesszalagra. A modern szalagok néhány tíz, vagy akár néhány száz gigabájt kapacitásúak, és fillérekbe kerül egy gigabájt. Azonban a mentés nem olyan triviális, mint ahogy elsőre gondolnánk, ezért foglalkozunk néhány vonatkozásával.

A szalagra mentés általában az alábbi két problémát hivatott megoldani:

1. Helyreállítás katasztrófa esetén.
2. Helyreállítás hibázás esetén.

Az első esetben lemezösszeomlás, tűzeset, árvíz vagy más természeti csapás után kell újraindítani a számítógépet. A gyakorlatban ilyen eset ritkán fordul elő, ezért van, hogy sokan nem törődnek mentéssel. Az ilyen emberek hasonló okokból nem biztosítják házukat tűzesetre.

A második eset akkor következik be, amikor a felhasználó véletlenül töröl fájlokat, amikre később szüksége lehet. Az ilyen esetek olyan gyakoriak, hogy a Windows törlés esetén speciális könyvtárba, a **Lomtár**ba helyezi, és nem törli ténylegesen a fájlt, ahonnan később kihalászható és helyreállítható lesz. A mentés továbbviszi ezt az elvet, és lehetővé teszi törölt fájlok visszaállítását napokkal vagy hetekkel később.

Mentés készítése hosszú időt és nagy helyet vesz igénybe, ezért fontos, hogy hatékonyan és kényelmesen végezzük. Ezek az okok indokolják a következő tényezők figyelembevételét. Először, a teljes fájlrendszert mentsük, vagy csak egy

részét? Sok telepítésnél a végrehajtható (bináris) fájlok elhelyezése korlátozott a fájlrendszerfában. Ezeket nem szükséges menteni, ha mind helyreállítható a gyártó által adott CD-ROM-ról. Hasonlóan, a legtöbb rendszerben van egy könyvtár az ideiglenes fájloknak. Ezeket sem szükséges menteni. Unixban a speciális fájlokat (I/O-eszközök) a */dev/* könyvtár tartalmazza. Ezeket nemcsak nem szükséges menteni, de veszélyes is, mert fennakadhat a mentés, amikor ezeket olvasni akarja. Röviden: általában nem a teljes fájlrendszert mentjük, hanem csak meghatározott könyvtárakat és azok tartalmát.

Másodszor, nem érdemes menteni azokat a fájlokat, amelyek az utolsó mentés óta nem változtak. Ez az ötlet vezet az **inkrementális mentés** módszeréhez. A legegyszerűbb formájában az inkrementális mentés azt jelenti, hogy periodikusan, hetente vagy havonta teljes mentést végzünk, és naponta csak azokat a fájlokat mentjük, amelyek megváltoztak az utolsó teljes mentés óta. Még jobb, ha csak azokat mentjük, amelyek az utolsó mentésük óta megváltoztak. Ez a módszer minimalizálja a mentési időt, de a helyreállítás bonyolultabb, mert először az utolsó teljes mentést kell visszaállítani, aztán az inkrementális mentéseket fordított időrendi sorrendben, a legrégebbit először. A helyreállítás megkönnyítése végett gyakran használnak kifinomultabb inkrementális mentési sémákat.

Harmadszor, mivel óriási adattömeget kell általában menteni, ezért mielőtt szalagra írnánk, érdemes tömöríteni. Azonban sok tömörítő algoritmus esetén egyetlen hibás szalaghely hibát eredményez, aminek az lehet a hatása, hogy egy teljes fájl, vagy akár az egész szalag olvashatatlan lesz. Tehát alaposan meg kell fontolni, hogy alkalmazzunk-e tömörítést.

Negyedszer, bonyolult menteni egy aktív fájlrendszert. Ha a mentés közben fájlok keletkeznek, törlődnek vagy módosulnak, akkor a mentés inkonzisztens lehet. A mentés azonban több óráig is eltarthat, ezért nem elfogadható, hogy a mentés idejére inaktívvá tegyük a rendszert. Ezért kifejlesztettek olyan algoritmusokat, amelyek gyors pillanatfelvételt készítenek a fájlrendszer állapotáról, a kritikus adatszerkezetek másolatát elkészítve, ami után blokkok másolásával és nem helyben módosításával lehet a fájlokat és könyvtárakat módosítani (Hutchinson, 1999). Így a fájlrendszert gyakorlatilag befagyasztják a pillanatfelvétel állapotában, amely ezután kényelmesen menthető lesz.

Végezetül, mentés készítése egy szervezet számára sok, nem technikai probléma megoldását igényli. A világ legjobb biztonsági rendszere is haszontalan, ha a rendszergazda őrizetlenül hagyja a mentési szalagokat, amikor kimegy a szobájából a nyomtatott anyagokért. Egy kémnek elegendő besurranni és zsebre rakni a kisméretű kazettát és vidáman távozni. Viszlát biztonság! Hasonlóan a napi mentés mit sem ér, ha tűz esetén elégnek a mentések is. Ezért a mentéseket máshol kell tárolni, ami újabb biztonsági kockázatot jelent. Ezeknek és más praktikus szervezési problémáknak az alaposabb tanulmányozása céljából javasoljuk Nemeth és társai könyvét (Nemeth et al., 2001). A következőkben csak a mentések technikai részével foglalkozunk.

Kétféle stratégia szerint lehet lemeztárolót szalagra menteni: fizikai vagy logikai mentés. **Fizikai mentés** esetén a 0. blokktól kezdve sorban minden blokkot kiírunk a szalagra, egészen az utolsóig. Egy ilyen program annyira egyszerű, hogy

100% valószínűséggel megírható hibamentesen, ami valószínűleg nem mondható el más hasznos programról.

Érdemes azonban néhány megjegyzést fűzni a fizikai mentéshez. Például nem érdemes menteni a nem használt blokkokat. Ha a mentést végző program hozzáférhet a szabad blokkokat tároló adatszerkezethez, akkor a mentés kihagyhatja azokat. Ha azonban nem mentjük a szabad blokkokat, akkor minden blokkhoz ki kell írni a sorszámát is, hiszen most már nem teljesül, hogy a *k*-adik blokk a *k*-adik lesz a mentési szalagon.

A második gond a hibás blokkok mentése. Ha minden hibás blokkot áthelyez az eszközvezérlő, mint azt az 5.4.4. alfejezetben leírtuk, akkor a fizikai mentés helyesen működik. Ellenkező esetben, ha a hibás blokkok láthatók az operációs rendszer számára, és kezeli ezeket valamilyen térképpel, akkor alapvető, hogy a mentést végző program is hozzáférjen ehhez az információhoz, hogy elkerülhesse a hibás blokkok olvasását, ami lemezolvasási hibák sokaságát eredményezné.

A fizikai mentés fő előnye, hogy egyszerű és nagyon gyors (alapvetően a lemez átviteli sebességével futhat). A fő hátránya, hogy nem képes kihagyni kijelölt könyvtárakat, nem tud inkrementálisan menteni, és nem lehet egyedi fájlokat helyreállítani. Ezen okok miatt a legtöbb rendszer logikai mentést alkalmaz.

A **logikai mentés** egy vagy több kijelölt könyvtárban lévő minden olyan fájlt és könyvtárat ment rekurzívan, amely egy megadott időpont óta változott (vagyis az utolsó inkrementális mentés vagy teljes mentés óta). Logikai mentés esetén tehát a szalag gondosan kiválasztott fájlokat és könyvtárakat tartalmaz, amelyek alapján szükség esetén egyszerűen helyre lehet állítani egyedi fájlokat és könyvtárakat.

Ahhoz, hogy egyedi fájl helyesen helyreállítható legyen, a fájlútvonalat megadó minden információt menteni kell. Tehát a logikai mentés első lépése a könyvtári fa szerkezetének elemzése. Nyilvánvalóan minden megváltozott fájlt és könyvtárat menteni akarunk. De a helyes helyreállításhoz minden olyan, akár nem módosult könyvtárat is menteni kell, amely egy módosult fájlhoz vagy könyvtárhoz vezető úton található. Ez azt jelenti, hogy nemcsak az adatot kell menteni (fájlnevet és i-csomópontra mutató pointert), hanem a könyvtár minden attribútumát is, hogy az eredeti jogosultságokat helyre tudjuk állítani. Először a könyvtárakat és azok attribútumait kell szalagra írni, majd a megváltozott fájlokat (az attribútumaikkal együtt). Ez lehetővé teszi, hogy a könyvtárakat és fájlokat egy másik számítógép fájlrendszerében helyreállítsuk. Ezzel a módszerrel a mentési és helyreállító programmal teljes fájlrendszereket át tudunk vinni egyik számítógépről a másikra.

A másik ok, hogy módosult fájl útjába eső nem módosult könyvtárakat is lementsünk, az, hogy ezáltal inkrementálisan helyre tudjunk állítani egyedi fájlokat (egy esetleges véletlen törlés után). Tegyük fel, hogy vasárnap este végzünk teljes mentést és hétfőn este inkrementálisat. Kedden törlődik a */usr/jhs/proj/nr3* könyvtár és minden alkönyvtára, valamint az ezekben lévő fájlok. Szerdán reggel egy felhasználó kéri a */usr/jhs/proj/nr3/plans/summary* fájl helyreállítását. Azonban a *summary* fájlt nem lehet helyreállítani, mert nincs könyvtár, ahová tehetnénk. Előbb az *nr3/* és a *plans/* könyvtárakat kell helyreállítani. Ezek tulajdonosait, módjait és dátumait csak akkor tudjuk megállapítani, ha ezeket is elmentettük a szalagra, annak ellenére, hogy nem változtak az utolsó teljes mentés óta.

Fájlrendszer mentési szalagról történő helyreállítása egyszerű. Azzal kell kezdeni, hogy létrehozunk egy üres fájlrendszert a lemezen. Ezután a legutolsó teljes mentést állítjuk helyre. Mivel a szalagon előbb vannak a könyvtárak, ezért a helyreállítás során előbb létrejönnek a szükséges könyvtárak, létrehozva a fájlrendszer vázszerkezetét. Majd maguk a fájlok állítódnak helyre. Ezt az eljárást kell ismételni az első inkrementális mentéssel, majd a másodikkal, és így tovább.

A logikai mentés egyszerű ugyan, de szükség van néhány trükkös fogásra is. Az egyik a következő. Mivel a szabad blokkok listája nem fájlban tárolódik, így nem mentődik, és ezért az összes helyreállítás után a semmiből kell felépíteni. Ez mindig megtehető, mert a szabad blokkok halmaza komplementere a fájlok által elfoglalt blokkok halmazának.

Ami a kapcsolásokat illeti, ha egy fájl két vagy több könyvtárhoz van kapcsolva, akkor fontos, hogy a fájlt csak egyszer állítsuk helyre, és minden rámutató könyvtári bejegyzés is helyre legyen állítva.

A következő probléma, hogy a Unix-fájlokban lehet lyuk. Legálisan megtehetjük, hogy megnyitunk egy fájlt, írunk bele néhány bájtot, aztán tetszőleges pozícióra pozicionálás után kiírunk még néhány bájtot. A közbülső blokkok nem részei a fájlnak, ezért nem lehet menteni, illetve helyreállítani ezeket. Memóriaképek (core dump) gyakran tartalmaznak nagy lyukakat az adatszegmens és a verem között. Ha nem helyesen kezeljük az ilyen fájlokat, akkor előfordulhat, hogy a helyreállítás 0-val tölti ki a lyuk területét, tehát akkora méretű lesz, mint a virtuális memória (azaz 2^{32} vagy 2^{64} bájt).

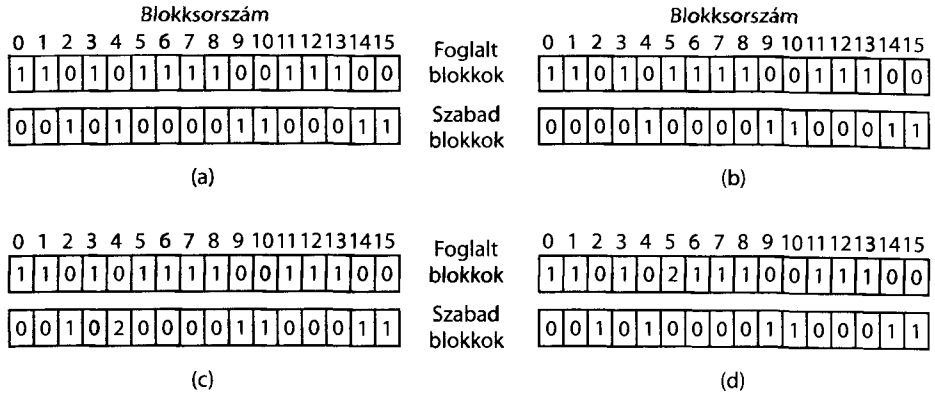
Végül a speciális fájlokat, adatcsöveket és hasonlókat sose mentsük, függetlenül attól, hogy milyen könyvtárban vannak (helyük nem kötött a */dev* könyvtárhoz). Fájlrendszerek mentéséről bővebben lásd (Chervenak, 1998; Zwicky, 1991).

Fájlrendszerek konzisztenciája

Egy másik terület, ahol a megbízhatóság jelentkezik, az a fájlrendszerek konzisztenciája. Sok rendszer úgy dolgozik, hogy blokkot olvas, feldolgozza, később pedig kiírja. Ha a rendszer olyan állapotban omlik össze, amikor nem minden módosult blokk került kiírásra, a fájlrendszer inkonzisztens állapotba kerül. Ez a probléma különösen kritikus, ha olyan blokk maradt kiíratlan, amely i-csomópontot, könyvtári bejegyzéseket vagy szabadlista-elemeket tartalmazott.

Az inkonzisztencia problémájának kezelésére a legtöbb számítógép rendelkezik olyan segédprogrammal, amely ellenőrzi a fájlrendszer konzisztenciáját. Ilyen a Unixban az *fsck*, a Windowsban a *chkdsk* (vagy *scandisk* a korábbi változatokban). Ez a program futtatható minden rendszerbetöltéskor, különösen összeomlás után. Az alábbiakban leírjuk, hogyan működik az *fsck*. A *chkdsk* működése ettől eltérő, mert más fájlrendszerekre alkalmazható, de az általános elv, amely a rendszer redundanciáját használja ki, itt is érvényes. Ezek a fájlrendszer-ellenőrzők az egyes fájlrendszereket (lemezpartíciókat) egymástól függetlenül ellenőrzik.

Kétféle konzisztencia-ellenőrzés lehet: blokk és fájl. A blokk-konzisztencia ellenőrzésére a program két táblázatot épít fel, mindegyik egy számlálót tartalmaz



5.19. ábra. *Fájlrendszerállapotok. (a) Konzisztens. (b) Hiányzó blokk. (c) Duplikált szabad blokk. (d) Duplikált adatblokk*

blokkonként, amelyeknek kezdeti értéke 0. Az első számláló azt mutatja, hány fájlban fordul elő a blokk, a másik számláló pedig azt, hányszor fordul elő a blokk a szabad listában (vagy a szabad blokkok bittérképén).

A program ezután végigolvassa az összes i-csomópontot. Az i-csomópont alapján sorra tudja venni az összes blokkot, amely a fájlhoz tartozik. Minden blokk esetén növeli eggyel az első táblázatban az adott blokk számlálóját. A program ezután a szabad listát vagy a bittérképet vizsgálja, hogy megtalálja az összes blokkot, amely szabadnak van jelölve. Minden előforduláskor a második táblázat megfelelő elemét növeli eggyel.

Ha a fájlrendszer konzisztens, akkor minden blokk számlálója vagy az első, vagy a második táblázatban 1. Ezt illusztrálja az 5.19.(a) ábra. Azonban összeomlás következtében a táblázatok az 5.19.(b) ábrán látható eltérést mutathatják, ahol is a 2. blokk mindkét számlálója 0. Ezt mint **hiányzó blokk**ot jelzi a program. Ugyan a hiányzó blokkok nem veszélyesek, de tárvesztést okoznak és csökkentik a lemez kapacitását. A hiányzó blokkok kijavítása nyilvánvaló: egyszerűen hozzá kell venni a szabad blokkok listájához.

Egy másik lehetséges esetet mutat az 5.19.(c) ábra. Itt azt látjuk, hogy a 4. blokk kétszer fordul elő a szabad listában. (Duplikáció csak szabad listás rendszerben fordulhat elő, bittérképnél nem.) A megoldás ebben az esetben is egyszerű, újra fel kell építeni a szabad listát.

A legrosszabb, ami előfordulhat, hogy egy blokk két vagy több fájlban is előfordul, mint az 5. blokk az 5.19.(d) ábrán. Ha valamelyik fájlból, de csak az egyikből törölnénk a blokkot, akkor ez azt eredményezné, hogy egyszerre lenne szabad és foglalt. Ha mindkettőből törölnénk, akkor pedig kétszer szerepelne a szabad listában.

Az elfogadható megoldás az, hogy egy új blokkot foglalunk, átmásoljuk bele az 5. blokk tartalmát, és beillesztjük az egyik fájlba. Ezzel a fájlok információtartalma nem változik, habár majdnem biztosan torzul, de legalább a fájlrendszert

konzisztenssé tettük. A hibát jelezni kell, hogy a felhasználó vizsgálja meg a károsodást.

A blokk-konzisztencia mellett az ellenőrző program a könyvtári rendszert is ellenőrzi. Ekkor is számlálókat használ, de most fájlonként és nem blokkonként. A gyökérkönyvtártól indulva rekurzívan járja be a fát, és megvizsgál minden könyvtárat. Minden könyvtár minden fájlja esetén növeli a hozzá tartozó számlálót. Emlékeztetünk, hogy merev kapcsolás miatt egy fájl esetleg több könyvtárban is előfordulhat. Szimbolikus kapcsolás nem számít, ezért ebben az esetben nem kell növelni a hivatkozásszámláló értékét sem.

Ezt elvégezve egy olyan i-csomópont szerint indexelt listát kapunk, amely megadja, hogy a fájl hány könyvtárnak eleme. Ezt összeveti magában az i-csomópontban tárolt kapcsolatszámlálóval. Konzisztens fájlrendszerben a két érték minden i-csomópontra megegyezik. Azonban kétféle hiba is előfordulhat: az i-csomópontban tárolt kapcsolatszámláló több, avagy kevesebb lehet.

Ha az i-csomópontban tárolt kapcsolatszámláló nagyobb, mint a könyvtári bejegyzések száma, akkor a kapcsolatszámláló még így is nagyobb lenne nullánál, ha az összes fájlt törölnénk, vagyis az i-csomópont nem törlődne. Ez a hiba nem veszélyes, de tárhelyveszteséget okoz, mert lesz olyan fájl, amely egyetlen könyvtárban sem szerepel. A hiba úgy javítható, hogy az i-csomópont számlálóját a helyes értékre állítjuk.

A másik hiba potenciálisan katasztrofális. Ha két könyvtári bejegyzés ugyanahhoz a fájlhoz van kapcsolva, de az i-csomópont azt mondja, hogy csak egy van, akkor akármelyik bejegyzést törölve az i-csomópont számlálója nullává válna. Ekkor a rendszer megjelöli mint használaton kívülit, és felszabadítja az általa foglalt összes blokkot. Az eredmény az lesz, hogy az egyik könyvtári bejegyzés olyan i-csomópontot tartalmaz, amely használaton kívülivé vált, és a blokkjait hamarosan más fájlok fogják használni. Ismét az a javítási megoldás, hogy a számlálóértéket az i-csomópontban a könyvtári bejegyzések tényleges számára állítjuk be.

Ez a két művelet, tehát a blokk- és a könyvtári ellenőrzés hatékonysági okok miatt gyakran egybe van integrálva (vagyis az i-csomópontokat csak egyszer kell végigjárni). Más, heurisztikus ellenőrzések is lehetségesek. Például a könyvtáraknak meghatározott formájuk van, i-csomópont számmal és ASCII névvel. Ha egy i-csomópont száma nagyobb, mint az i-csomópontok darabszáma a lemezen, akkor a könyvtár biztosan sérült.

Továbbá minden i-csomópontnak van jogosultsági módja, ami lehet ugyan legális, de mégis furcsa, mint például a 0007, ami azt jelenti, hogy a felhasználó és csoportja nem érheti el, de mindenki más olvashatja, írhatja és végrehajthatja. Hasznos lehet legalább jelezni azokat a fájlokat, amelyeknél a kívülállóknak több joga van, mint a tulajdonosnak. Az olyan könyvtár, amelynek sok – mondjuk 1000-nél több – bejegyzése van, szintén gyanús. A felhasználói könyvtárban találhatók olyan fájlok, amelyek tulajdonosa szuperfelhasználó, és SETUID bitje be van állítva, potenciális biztonsági problémát jelenthetnek, hiszen egy ilyen fájlt végrehajtva egy normál felhasználó is szuperfelhasználói jogokat szerezhet. Kis erőfeszítéssel igen terjedelmes lista gyűjthető össze a legális, de gyanús dolgokról, amelyeket célszerű jelenteni.

Az előző bekezdések azzal foglalkoztak, hogyan lehet védeni a felhasználót a rendszer összeomlása ellenében. Néhány fájlrendszer azzal is törődik, hogy védje a felhasználót önmaga ellen. Ha a felhasználó az

```
rm *.o
```

parancsot szándékozott begépelni, hogy törölje a fordító által készített tárgykódfájlokat, de helyette az

```
rm * .o
```

sikeredett (szóköz van a * után), akkor a parancs kitörli az összes fájlt az aktuális könyvtárból, és aztán szól, hogy nem találja a *.o* nevű fájlt. Néhány rendszer törlés esetén csak azt teszi, hogy egy bitet beállít a könyvtárban vagy az i-csomópontban, jelezve, hogy a fájl töröltté vált. Addig azonban nem szabadítja fel a lefoglalt blokkokat, amíg azokra ténylegesen nincs szükség. Így ha a felhasználó időben észreveszi az elkövetett hibát, akkor egy speciális programmal visszanyerheti a törölt fájlokat. Windowsban a törölt fájlok egy speciális, ún. újrahasznosító *(recycle bin)* könyvtárba kerülnek, ahonnan később szükség esetén visszanyerhetők. Nyilvánvalóan nem szabadul fel lemezhely ebből a könyvtárból, amíg a törlés ténylegesen meg nem történik.

Az ilyen mechanizmusok azonban nem biztonságosak. Egy biztonságos rendszer valójában felülírná a törölt blokkot 0-val vagy véletlen bitekkel, így egy másik felhasználó már nem tudná helyreállítani a törölt fájl adatait. Sok felhasználó nem törődik azzal, hogy az adat meddig marad meg. Gyakran előfordul, hogy titkos vagy érzékeny adat visszaállítható törölt lemezről (Garfinkel és Shelat, 2003).

5.3.6. Fájlrendszer hatékonysága

A lemez elérése sokkal lassúbb, mint a memória elérése. Egy memóriabeli szó kiolvasása tíz nanomásodpercig tarthat. Egy blokk beolvasása 10 MB/s átviteli sebességű mágneslemezről 32 bites szavanként negyvenszer lassabb, de ehhez még hozzájön 5–10 ezred másodperc, ami a sávra pozicionálás ideje, és a várakozás, amíg a kívánt szektor az olvasófej alá fordul. Ha csak egy szóra van szükség, akkor a memóriaelérés nagyságrendben milliószor gyorsabb a lemez elérésénél. Az elérési idő ilyen mértékű különbsége miatt sok fájlrendszert a hatékonyságot növelő optimalizálással terveznek. Ebben az alfejezetben három ilyen módszert vizsgálunk meg.

Gyorsítótár

A leggyakrabban alkalmazott technika a lemezhez fordulások csökkentésére a **blokkgyorsítótár**, vagy más néven **puffergyorsítótár**. (Angolul *cache,* ami a francia *cacher* szóból származik, és jelentése rejteni.) Ebben a szövegösszefüggésben a

gyorsítótár olyan lemezblokkok kollekciója, amelyek logikailag a lemezhez tartoznak, de a hatékonyság növelése érdekében a memóriában tartjuk őket.

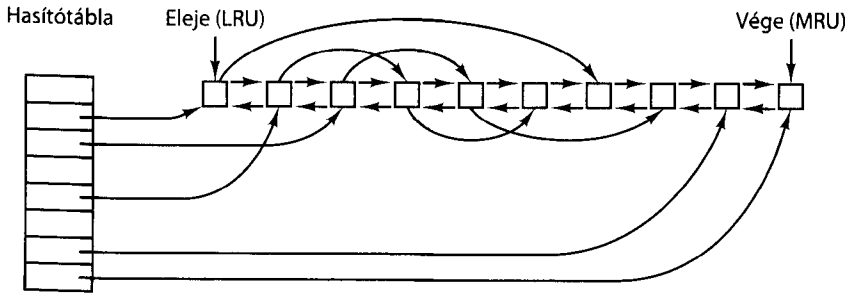
A gyorsítótár kezelésére különböző algoritmusok vannak, de az általános az, hogy olvasási igény esetén előbb ellenőrzik, hogy a kért blokk a gyorsítótárban van-e. Ha igen, akkor lemezhez fordulás nélkül kielégíthető az igény. Ha a kért blokk nincs a gyorsítótárban, akkor először beolvasódik a lemezről a gyorsítótárba, majd a kért helyre másolódik. Az erre a blokkra vonatkozó minden további kérés kielégíthető a gyorsítótárból.

A gyorsítótár működését szemlélteti az 5.20. ábra. Mivel sok (gyakran több ezer) blokk van a gyorsítótárban, ezért szükséges olyan módszer, amely gyorsan meg tudja határozni, hogy egy adott blokk bent van-e. Szokásosan hasítótáblás (hash table) módszert alkalmaznak. Az ütközésfeloldást láncolással végzik, azaz az azonos hasítófüggvény-értékű blokkok egy láncolt listában szerepelnek.

Ha a gyorsítótárba történő beolvasáskor az már tele van, akkor előbb ki kell írni a lemezre valamelyik bent lévő blokkot, hogy legyen hely az új blokk számára. Ez a szituáció nagyon hasonlít a lapozásra, és a 4. fejezetben megismert szokásos lapozási algoritmusok mindegyike – mint például a FIFO, a második lehetőség és az LRU – alkalmazható itt is. Kellemes különbség a lapozás és a gyorsítótár-kezelés között az, hogy a gyorsítótárbeli hivatkozások viszonylag ritkák, így az összes blokkot pontos LRU-sorrendben tarthatjuk a láncolt listában.

Az 5.20. ábrán láthatjuk, hogy az ütközésfeloldáshoz használt láncon kívül az összes blokk egy kétirányú láncban van a használatuk szerinti sorrendben, a legutoljára használt az első, az utoljára használt pedig az utolsó a láncban. Ha egy blokkra hivatkozás történik, akkor ki lehet venni a láncból és a sor végére lehet tenni. Ilyen módon pontos LRU-sorrendben tarthatók a blokkok.

Sajnos van egy csapda. Olyan helyzettel van dolgunk, hogy lehetséges pontos LRU, mégis kiderül, hogy az LRU nem kívánatos. A problémát a korábbi szakaszban tárgyalt rendszerösszeomlás és fájlrendszer-konzisztencia okozza. Az a helyzet, amikor egy kritikus blokk – például i-csomópontot tartalmazó blokk – a gyorsítótárban van és módosult, de nem lett kiírva a lemezre, miközben a rendszer összeomlott, a fájlrendszer inkonzisztens állapotát eredményezi. Ha az i-csomópont blokkját az LRU-lánc végére rakjuk, akkor könnyen előfordulhat, hogy sokára ér a lánc elejére, és így nem kerül kiírásra.



5.20. ábra. *Gyorsítótár-adatszerkezet*

Továbbá, bizonyos blokkokra – mint az i-csomópont blokkok – rövid időtartam alatt ritkán hivatkoznak kétszer egymás után. Ezek a megfontolások egy módosított LRU-sémához vezetnek, amely két tényezőt vesz figyelembe:

1. Valószínű-e, hogy a blokkra hamarosan ismét szükség lesz?
2. Fontos-e a blokk a fájlrendszer konzisztenciája szempontjából?

A blokkok mindkét kérdés szerint kategóriákba csoportosíthatók, úgymint i-csomópont blokk, indirekt blokk, könyvtári blokk, teli adatblokk, csonka adatblokk. Azok a blokkok, amelyekre hamarosan nem lesz ismét szükség, az LRU-lánc elejére és nem a végére mennek, így a puffereik gyorsan újra felhasználásra kerülnek. Azok a blokkok, amelyekre hamarosan ismét szükség lehet, mint például a csonka adatblokkok, hisz ezekbe írni fogunk, a lánc végére kerülnek és sokáig ott is maradnak.

A második kérdés független az elsőtől. Ha egy blokk fontos a fájlrendszer konzisztenciája szempontjából (alapvetően minden blokk ilyen, kivéve az adatblokkokat), és módosult, akkor azonnal kiírandó a lemezre – függetlenül attól, hogy az LRU-lista melyik végén volt. Ha a kritikus blokkokat gyakran kiírjuk, akkor nagyban csökkentjük annak valószínűségét, hogy összeomláskor a fájlrendszer tönkremegy. Bár a felhasználó nem örül, ha rendszerösszeomlás miatt elvész egy fájlja, de valószínűleg még boldogtalanabb lenne, ha a teljes fájlrendszert elvesztenék.

A fájlrendszer integritásának megtartása fontos, de emellett nem kívánatos az adatblokkokat sem sokáig a gyorsítótárban tartani kiírás nélkül. Tekintsük azt a helyzetet, amikor valaki személyi számítógépén könyvet ír szövegszerkesztővel. Még ha periodikusan mentést is végez az író, akkor is előfordulhat, hogy a teljes dokumentum a gyorsítótárban marad anélkül, hogy a lemezre valaha is kiírásra kerülne. A rendszer összeomlása esetén a fájlrendszer nem sérülne, de az egész napi munka elveszne.

Ez a helyzet nem fordulhat elő nagyon gyakran, hacsak nem vagyunk peches felhasználók. A rendszerek két megközelítést használnak a probléma megoldására. A Unix biztosít egy sync nevű rendszerhívást, amely kikényszeríti az összes módosult blokk azonnali kiírását a lemezre. Az operációs rendszer indításakor elindul egy program, a neve általában *update*, amely a háttérben ül, és 30 másodpercenként sync hívásokat kezdeményez, egyébként alszik. Ennek az a hatása, hogy összeomlás miatt legfeljebb 30 másodpercnyi munka veszhet el.

A Windows azt csinálja, hogy minden olyan blokkot azonnal kiír a lemezre, amelybe írás történt. Az olyan gyorsítótárat, amely minden módosult blokkot azonnal kiír, **írásáteresztő gyorsítótár**nak nevezik. Ezek sokkal több lemezes I/O-műveletet igényelnek, mint a nem írásáteresztő gyorsítótárak. A különbség a Unix és az MS-DOS módszere között jól látható, amikor egy program 1 KB méretű blokkot teleír, karakterenként végezve az írást. A Unix-rendszer összegyűjti a karaktereket a gyorsítótárban, és 30 másodpercenként, vagy akkor, amikor a blokk kikerül a gyorsítótárból, kiírja a lemezre. A Windows minden egyes karakter írásakor lemezhez fordul. Természetesen a legtöbb program használ belső puffert, így

rendszerint nem karakterenként, hanem soronként vagy még nagyobb egységekben végez write rendszerhívást.

A gyorsítótár-stratégiák különbségének az a következménye, hogy ha Unix esetén egy (hajlékony-) lemezt csak úgy kiveszünk, mielőtt sync-et végrehajtottunk volna, majdnem mindig adatvesztés lesz, és gyakran a fájlrendszer is megsérül. Windows esetén nincs ilyen probléma. Azért választottak a két rendszerben különböző stratégiát, mert a Unixot olyan környezetben fejlesztették ki, ahol a lemezek nem voltak cserélhetők, míg a Windows tipikusan hajlékonylemez-világban született. Ahogy a merevlemezek általánossá váltak, a Unix megközelítése lett a norma a jobb hatékonyság miatt, és ma már a Windows is ezt használja.

Blokk előreolvasása

A másik technika a fájlrendszer hatékonyságának növelésére az, hogy megpróbáljuk a blokkokat a gyorsítótárba tölteni, még mielőtt kellenének, ezzel növelve a találati arányt. Kiváltképpen, mivel sok fájlt szekvenciálisan olvasunk. Amikor a fájlrendszertől egy fájl k-adik blokkját kérjük, akkor az szolgáltatja ezt a blokkot, de szolgai módon ellenőrzi, hogy a $k + 1$-edik blokk benn van-e a gyorsítótárban. Ha nincs, akkor beütemezi a beolvasását abban a reményben, hogy ha szükség lesz rá, akkor a gyorsítótárban legyen, vagy legalábbis úton.

Természetesen ez az előreolvasási stratégia csak szekvenciálisan olvasott fájlok esetén hasznos. Direkt elérésű fájlok esetén az előreolvasás nem segít. Valójában kárt okoz azzal, hogy leköti a lemez átviteli kapacitását azzal, hogy haszontalan blokkokat olvas be és esetleg hasznosakat töröl a gyorsítótárból (és azzal is leköti az átvitelt, hogy a módosult blokkokat kiírja, hogy mindehhez így csináljon helyet). Az előreolvasás hasznosságának eldöntéséhez a fájlrendszer nyilvántarthatja minden nyitva lévő fájlhoz annak elérési sémáját. Például egy bit jelezheti, hogy a fájl szekvenciális, avagy direkt elérésű módban van-e. Kezdetben a kétségek miatt szekvenciális módra állítódik be. Azonban mihelyt egy pozicionálás végrehajtódik, ez a bit törlődik. Ha később ismét szekvenciális olvasás történik, akkor a bit ismét beállítódik. Ilyen módon a fájlrendszer megbecsülheti, hogy érdemes-e az előreolvasást alkalmazni. Nem katasztrófa, ha időnként rossznak bizonyul a becsült stratégia, hisz ez csak egy kis veszteséget okoz az átviteli sávszélességben.

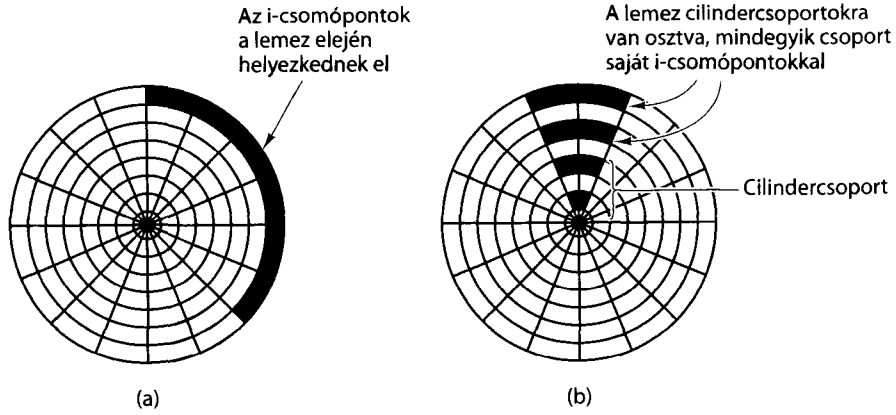
A lemezfej mozgásának csökkentése

A fájlrendszerek hatékonysága nemcsak a gyorsítótár és előreolvasás alkalmazásával növelhető. Egy másik fontos technika arra szolgál, hogy csökkentsük a lemezegység fejének mozgását azáltal, hogy azokat a blokkokat, amelyeket valószínűleg egymás után igényelnek, egymáshoz közel helyezzük el, lehetőleg azonos cilinderen. Amikor egy kimenetfájlt írunk, akkor a fájlrendszernek az igény szerinti sorrendben egymás után kell a blokkokat lefoglalnia. Ha a szabad blokkokat bittérképen tartjuk nyilván, amely teljesen a memóriában van, akkor eléggé könnyű az egymáshoz lehető legközelebbi blokkok lefoglalása. Szabad lista esetén, amelynek csak egy része van a memóriában, sokkal nehezebb egymáshoz közeli blokkokat lefoglalni.

Azonban még szabad lista esetén is lehetséges bizonyos blokkcsoportosítás. A trükk az, hogy a lemeztárat nem blokkegységben foglaljuk le, hanem egymást követő blokkokban. Ha egy szektor 512 bájtos, akkor a rendszer 1 KB méretű blokkot (2 szektor) használhat, de a helyfoglalási egység 2 blokk (4 szektor) lehet. Ez nem egyenértékű azzal, amikor a blokkméret 2 KB, mivel a gyorsítótár most is 1 KB-os blokkokkal dogozik, és az átvitel is 1 KB méretű blokkban történik. De szekvenciálisan olvasva egy fájlt, feltéve, hogy a rendszer egyébként tétlen, a pozicionálások száma a felére csökken, ami tekintélyes hatékonyságnövekedés. Egy változat erre a témára az, amikor figyelembe vesszük a rotációs pozicionálást. Blokkok lefoglalásakor a rendszer a fájlban egymást követő blokkokat azonos cilinderen igyekszik elhelyezni.

Egy másik szűk keresztmetszetet jelent az i-csomópontot alkalmazó rendszereknél az, hogy még kisméretű fájl olvasása is két lemezhez fordulást igényel: egyet az i-csomópont, egyet pedig az adatblokk eléréséhez. Egy i-csomópont szokásos elhelyezését mutatja az 5.21.(a) ábra. Itt minden i-csomópont a lemez elején található, így az i-csomópont és a blokkjai közötti távolság körülbelül a cilinderek számának felével egyenlő, ami hosszú pozicionálást eredményez.

Könnyen növelhetjük a hatékonyságot, ha az i-csomópontokat nem a lemez elejére tesszük, hanem a közepére, így felére csökkenthetjük az átlagos pozicionálási időt. Egy másik ötlet szerint, amelyet az 5.21.(b) ábra mutat, a lemezegységen a cilindereket csoportokba osztják, minden csoportnak saját i-csomóponti blokkjai és szabad listája van (McKusick et al., 1984). Új fájl létrehozásakor bármely i-csomópont választható, de a rendszer arra törekszik, hogy a blokkokat ugyanabban a csoportban foglalja le, ahol az i-csomópont van. Ha ez nem lehetséges, akkor egy közeli csoportot választ.



5.21. ábra. *(a) Az i-csomópontok a lemez elején vannak. (b) A lemezcilinder csoportokba van szervezve, csoportonként saját i-csomópontokkal és blokkokkal*

5.3.7. Naplózott fájlrendszer

A technológia fejlődése nyomást gyakorol a jelenlegi fájlrendszerekre. Különösen az, hogy a processzorok egyre gyorsabbak lesznek, a lemezek egyre nagyobbak és olcsóbbak (de nem sokkal gyorsabbak), és a memóriák mérete exponenciálisan növekszik. Az a paraméter, amely nem javul ugrásszerűen, a lemezek pozicionálási ideje. Ezen tényezők kombinációja azt eredményezi, hogy sok rendszerben szűk keresztmetszet alakul ki a teljesítményt tekintve. A Berkeley Egyetemen kutatást végeztek ezen problémák enyhítése érdekében, megtervezve egy teljesen új fájlrendszert, az **LFS-t (Log-structured File System – naplózott fájlrendszer)**. Ebben a részben röviden bemutatjuk, hogyan működik az LFS. Teljesebb leírását lásd (Rosenblum és Ousterhout, 1991).

Az LFS tervezését vezérlő elv az volt, hogy a processzorok egyre gyorsabbak, a RAM mérete egyre nagyobb és a lemezgyorsítótár gyorsan növekszik. Ennek következtében lehetővé vált, hogy az összes olvasási igény nagyon jelentős részét a gyorsítótárból, lemezhez fordulás nélkül lehet kielégíteni. Ebből az észrevételből következik, hogy a jövőben a lemezhez fordulások nagy része írási művelet lesz. Tehát nem eredményez teljesítményjavulást az előreolvasási mechanizmus, amit néhány rendszerben alkalmaznak, amikor is blokkokat beolvasnak a gyorsítótárba, még mielőtt azokra igény lenne.

A helyzet még rosszabb, ugyanis a legtöbb fájlrendszerben az írások nagy része kis szeletekben történik. A kicsi írások különösen nem hatékonyak, mivel 50 ezred másodperc átvitelt tipikusan 10 ms-os pozicionálás és 4 ms-os forgási késedelem előz meg. Ezekkel a paraméterekkel a lemez hatékonysága 1%-ra esik vissza.

Hogy lássuk, hogy a kicsi írások honnan származnak, tekintsük egy új fájl létesítését Unix-rendszerben. A művelet végrehajtásához lemezre kell írni a könyvtár i-csomópontját, a könyvtári blokkot, a fájl i-csomópontját és a fájlt magát. Ezek az írások ugyan elhalaszthatóak, de ez komoly konzisztenciaproblémákat eredményezne, ha a tényleges kiírás előtt összeomlana a rendszer. Éppen ezért az i-csomópontok kiírását általában azonnal elvégzik.

Ezen érvek miatt az LFS tervezői elhatározták, hogy a Unix-fájlrendszer új megvalósítását készítik el oly módon, hogy elérjék a lemez sávszélességének teljes kihasználását, még akkor is, ha a terhelés nagyrészt véletlenszerű, kicsi írásokból áll. Az alapötlet az, hogy az egész lemezt egy naplóvá szervezik. Periodikusan és speciális igény esetén a memóriapufferben található összes függőben lévő írást összegyűjtik egy szegmensbe, és azt egyben kiírják a napló végére. Ekkor egy szegmens vegyesen tartalmazhat i-csomópontot, könyvtári blokkot és adatblokkot is. Minden szegmens elején található egy összefoglaló rész, amely megadja, hogy mi van a szegmensben. Ha a szegmens átlagos mérete kb. 1 MB lesz, akkor a lemez majdnem teljes sávszélessége kihasználható.

Ebben a tervben továbbra is vannak i-csomópontok, és ezek szerkezete megegyezik a Unixban használttal, de most az i-csomópontok a naplóban szétszórtan vannak, nem pedig a lemezen rögzített helyen. Azonban az i-csomópont elérése után a blokkok elérése a szokásos módon megy. Természetesen az i-csomópont elérése most sokkal nehezebb, mivel címe nem számítható ki az i-csomópont sorszámából, mint a Unix esetén. A megoldás az, hogy az i-csomópontok sorszámával indexelt táblázatot alkalmaznak. Az *i* indexű elem tartalmazza az *i* sorszámú i-csomópont lemezcímét. A táblázatot lemezen tárolják, de ez is gyorsítótáron keresztül működik, így a legtöbbet használt részei majdnem mindig a memóriában vannak a hatékonyság növelése érdekében.

Összefoglalva az eddig mondottakat, kezdetben minden kiírandó a memóriapufferben van, és periodikusan minden pufferezett kiírandót kiírunk a lemezre egy szegmensben a napló végére. Fájl megnyitásakor most az i-csomópontok címtáblázatában kell megkeresni az i-csomópont címét, majd azt elérve megtaláljuk a blokkok címeit. Minden blokk maga is valamelyik szegmensben van a naplóban.

Ha a lemez végtelen kapacitású lenne, akkor készen is lennénk. Azonban a valódi lemezek végesek, így végül a napló elfoglalja a teljes lemezt, és innentől nem tudnánk több új szegmenst kiírni a naplóba. Szerencsére sok létező szegmensben lehetnek olyan blokkok, amelyek a továbbiakban nem kellenek, például amikor felülírunk egy fájlt, akkor annak i-csomópontja új blokkokra mutat, de a régiek még foglalják a helyet korábban kiírt szegmensekben.

Mindkét probléma megoldására az LFS **takarító**fonalat üzemeltet, amely tömöríti a naplót annak cirkuláris bejárásával. A munkát a napló első szegmensével kezdi, megvizsgálja az összefoglalót, hogy milyen i-csomópontok és fájlok vannak benne. Ezután ellenőrzi az i-csomópontok címtáblázatában, hogy az i-csomópontok aktuálisak-e, az adatblokkok használatban vannak-e. Átlépi azokat, amelyek nem aktuálisak. Azokat az i-csomópontokat és adatblokkokat, amelyek aktuálisak, beolvassa a memóriába, hogy aztán kiíródjanak a következő szegmensben. Az így feldolgozott szegmenst megjelöli, hogy szabaddá vált, így a napló újra felhasználhatja. Ily módon a takarító végigjárja a naplót, eltávolítva a régi szegmenseket a végéről, és memóriába tölt minden élő adatot, amely kiírásra kerül a következő szegmensben. Következésképpen a lemez egy nagy cirkuláris puffer lesz. A rendszer írást végző fonala az elejéhez illeszt új szegmenseket, takarítófonala pedig a végéről távolítja el a régieket.

A könyvelés nem triviális, ugyanis amikor a takarító egy fájlblokkot visszaír egy új szegmensbe, akkor meg kell keresni a fájl i-csomópontját (valahol ott van a naplóban), azt aktualizálni kell, majd kiírni a következő szegmensben. Az i-csomópontok címtáblázatát is módosítani kell, hogy az új címet tartalmazza. Mindazonáltal az adminisztráció elvégezhető, és a teljesítményeredmények azt mutatják, hogy ez a bonyolultság megéri. Az idézett cikk szerint a mérések azt mutatják, hogy az LFS egy nagyságrenddel felülmúlja a Unixot kicsi írások esetén, miközben teljesítménye ugyanolyan jó vagy jobb olvasásnál és nagyméretű írásnál.

5.4. Biztonság

A fájlrendszerek gyakran tartalmaznak olyan információt, amelyek nagy értéket képviselnek a felhasználói számára. Ezért minden fájlrendszerrel szemben fő követelmény ezen információk megvédése a jogosulatlan felhasználástól. A következő

alfejezetben a biztonság és a védelem különböző kérdéseivel foglalkozunk. A kérdések egyaránt érintik az időosztásos rendszereket és a személyi számítógépek hálózatait, amelyek lokális hálózaton keresztül megosztott kiszolgálóhoz kapcsolódnak.

5.4.1. Biztonsági környezet

A „biztonság" és „védelem" kifejezéseket gyakran felcserélhetően használják. Célszerű azonban a két fogalom között különbséget tenni. Az egyik az az általános problémakör, amely annak biztosítását jelenti, hogy jogosulatlan személy ne olvashassa és módosíthassa a fájlokat, ami egyrészt technikai, szervezési, jogi és politikai problémákat jelent; másrészt olyan specifikus operációsrendszer-mechanizmusokat, amelyek a biztonságot szolgálják. A félreértés elkerülése végett mi a **biztonság** kifejezést használjuk az általános problémára és a **védelmi mechanizmus** kifejezést a specifikus operációsrendszer-mechanizmus számára, amely a számítógépes információ védelmét szolgálja. Azonban a határ közöttük nincs jól definiálva. Először a biztonság kérdésével foglalkozunk, hogy lássuk a probléma természetét, majd a védelmi mechanizmust és a biztonság elérését lehetővé tevő modelleket tanulmányozzuk.

A biztonságnak több oldala van. A három legfontosabb a fenyegetés (veszélyek), a behatolás és a véletlen adatvesztés.

Fenyegetés

Biztonsági szempontból a számítógépes rendszereknek három általános célja van, az ezekhez tartozó veszélyekkel, amit az 5.22. ábra mutat. Az első a **bizalmas adatkezelés**, amely azt jelenti, hogy titkos adatok titkosak maradnak. Még pontosabban, ha bizonyos adatok tulajdonosa úgy dönt, hogy ezek az adatok csak meghatározott emberek számára legyenek elérhetők, mások számára pedig nem, akkor a rendszernek biztosítania kell, hogy az adatok nem jogosult személyekhez ne jussanak el. A nyilvánvaló minimum az, hogy a tulajdonos meg tudja adni, hogy ki mit láthat, és a rendszer érvényesíteni tudja ezeket a specifikációkat.

A második cél az **adatintegritás**, ami azt jelenti, hogy jogosulatlan felhasználó ne legyen képes adatmódosításra a tulajdonos beleegyezése nélkül. Az adatmódosítás itt nemcsak változtatást jelent, hanem törlést és hamis adatok hozzáadását is. Ha egy rendszer nem tudja garantálni, hogy a benne elhelyezett adatok változatlanok maradnak mindaddig, amíg a tulajdonos nem akar változtatni, akkor az mint

Célok	Veszélyek
Bizalmas adatkezelés	Expozíció
Adatintegritás	Adathamisítás
Rendelkezésre állás	Szolgáltatásmegtagadás

5.22. ábra. *Biztonsági célok és veszélyek*

információs rendszer nem ér sokat. Az integritás általában sokkal fontosabb, mint a biztonság.

A harmadik cél a **rendelkezésre állás**, ami azt jelenti, hogy senki se zavarhassa meg a rendszert úgy, hogy használhatatlan legyen. Az ilyen **szolgáltatásmegtagadás**-támadások gyakorisága növekszik. Például egy internetszolgáltatást nyújtó számítógép lebénítható, ha kérések özönét küldjük rá, mivel csupán a kérések vizsgálata és elutasítása felemészti a CPU idejét. Ha mondjuk 100 μs kell egy weboldalkérés beolvasásához, akkor bárki, aki képes 10000 kérést generálni másodpercenként, blokkolhatja a gépet. Elfogadható modellek és technológiák vannak a biztonság és az integritás biztosítására, de a szolgáltatásmegtagadás-támadások kivédése sokkal nehezebb.

A biztonság egy másik aspektusa a **magánélet**: megvédeni a személyeket attól, hogy a róluk készült adatokkal visszaéljenek. Ez azonnal jogi és morális problémákat vet fel. Vezethet-e valamely állami szervezet dossziét mindenkiről abból a célból, hogy elfogja a csalókat, akár társadalombiztosítási, akár adócsalásról legyen szó? Megengedhető-e, hogy a rendőrség a szervezett bűnözés elleni harc érdekében bárkinek bármilyen adatát átnézze? Milyen jogaik lehetnek a munkaadóknak és a biztosítóknak? Mi történik, ha ezek a jogok ütköznek a személyiségi jogokkal? Ezek rendkívül fontos problémák, de tárgyalásuk meghaladja e könyv kereteit.

Behatolás

A legtöbb ember nagyon rendes és jogkövető, miért aggódunk a biztonság miatt? Mert van néhány ember, aki nem olyan rendes és bajt akar okozni (esetlegesen saját hasznára). A biztonsági irodalomban az olyan embert, aki olyan helyeken jár, ahol semmi keresnivalója nincs, **behatolónak**, néha **ellenségnek** nevezik. A behatolók kétféleképpen dolgoznak. A passzív behatoló csak el akar olvasni olyan fájlokat, amelyekre nincs jogosultsága. Az aktív behatolók sokkal kártékonyabbak, ők jogosulatlan változtatásokat akarnak végezni. Egy behatolás ellen védett rendszer tervezésénél figyelembe kell venni, hogy milyen behatolás ellen akarunk védekezni. A következő kategóriákat lehet megkülönböztetni:

1. Képzetlen felhasználó alkalmi kíváncsiskodása. Sok embernek van otthon számítógépe, amely fájlkiszolgálóhoz kapcsolódik, és emberi tulajdonság, hogy néhányuk elolvassa mások elektronikus leveleit, más fájljait, ha nincsenek ebben korlátozva. A legtöbb Unix-rendszer például alapértelmezés szerint mindenki által olvasható jogosultságot állít be az újonnan létesített fájlokra.
2. Szaglászás bennfentes által. Hallgatók, rendszerprogramozók, operátorok és más technikai személyek gyakran tekintik személyes kihívásnak, hogy helyi számítógépek biztonsági rendszerét feltörjék. Ők sokszor magasan képzettek, és hajlandók sok időt fordítani efféle tevékenységre.
3. Határozott pénzszerzési szándék. Előfordul, hogy banki alkalmazásban lévő programozó lopási céllal akar betörni a banki rendszerbe. A séma változatos,

célja lehet megváltoztatni a szoftvert úgy, hogy csonkítás helyett kerekítést végezzen a kamaton, így megtartsa a váltópénz törtrészét, vagy megcsapoljon évek óta nem használt számlákat, vagy zsaroljon („Fizess, vagy tönkreteszem a bank adatbázisát!").

4. Kereskedelmi vagy katonai kémkedés. A kémkedés versenytárs vagy idegen ország komoly és jól megalapozott lopási szándéka, amely arra irányul, hogy programot, gyártási titkot, szabadalmat, technológiát, áramkörtervet, piaci stratégiát vagy más hasonló dolgot eltulajdonítson. A kémkedés gyakran alkalmaz lehallgatást, vagy akár olyan antennát, amelyet a számítógép felé irányítanak, hogy felfogják a gép által kibocsátott elektromágneses hullámokat.

Világosan kell látni, hogy nagy különbség van a katonai titkok ellopása és aközött, amikor hallgatók mókás üzeneteket illesztenek a számítógép rendszerébe. A kifejtendő erőfeszítés nagysága nyilvánvalóan függ attól, hogy ki az ellenség, aki ellenében védekezni kell.

Rosszindulatú programok

A biztonsági ártalom egy másik kategóriájába tartoznak a kártékony programok, vagy más néven **malware**. Bizonyos értelemben a rosszindulatú program készítője is behatoló, aki gyakran igen felkészült. Az a különbség a hagyományos behatoló és a malware között, hogy az előbbi egy személy, aki maga akar betörni a rendszerbe, hogy kárt okozzon, addig az utóbbi egy program, amelyet ilyen ember írt, és aztán elterjesztett a világban. Néhány malware-t csak azért írtak, hogy kárt okozzon, másokat sokkal speciálisabb céllal készítenek. Ez egyre nagyobb probléma és igen kiterjedt irodalma van (Aycock és Barker, 2005; Cerf, 2005; Ledin, 2005; McHugh és Deek, 2005; Treese, 2004; és Weiss, 2005).

A malware legismertebb változata a vírusvírus, alapvetően egy programkódrészlet, amely más programhoz kötődve reprodukálni tudja magát, a biológiai vírus analógiájára. A vírus a reprodukciója mellett mást is csinálhat. Például üzenetet vagy képet jeleníthet meg a képernyőn, lejátszhat zenét, vagy más veszélytelen dolgot is csinálhat. Sajnos módosíthat, törölhet vagy ellophat fájlokat (elektronikus levélben elküldve azt valahová).

A vírus futása alatt használhatatlanná is teheti a számítógépet. Ezt **DOS- (Denial Of Service – szolgáltatásmegtagadás)** támadásnak nevezik. A szokásos cél az erőforrások vad elfogyasztása, mint a CPU vagy a lemez kitöltése szeméttel. A vírusok (és más malware-ek, amelyeket majd tárgyalunk) **DDOS- (Distributed Denial Of Service – osztott szolgáltatásmegtagadás)** támadásra is felhasználhatók. Ekkor a vírus közvetlenül a megfertőzés után nem csinál semmit. Egy meghatározott időpontban azonban a vírus sok ezer példánya szerte a világban mind egyszerre elkezdi ugyanazt a weblapot vagy más hálózati szolgáltatást kérni, ezzel túlterheli a szolgáltató számítógépét és a hálózatot.

A malware-t gyakran haszonszerzés céljából készítik. A legtöbb (ha nem az összes) levélszemetet (spam) olyan hálózatba kapcsolt számítógépek terjesztik,

amelyeket vírus vagy más malware fertőzött meg. Az ilyen rosszindulatú program által megfertőzött számítógép „szolgává" válik, az állapotáról jelentéseket küld valahol az interneten található gazdájának. A gazdagép ezután a szolga által begyűjtött és elküldött címlista alapján terjeszti a levélszemetet. A haszonszerzést célzó malware egy másik fajtája a **kulcsgyűjtő**. A kulcsgyűjtő a megfertőzött számítógépen minden billentyűleütést tárol. Nem túl bonyolult olyan programot írni, amely kiszűri az olyan információt, mint a felhasználónév és jelszó páros vagy hitelkártya-számlaszám. Ezeket aztán elküldi a gazdának, aki felhasználhatja, vagy eladhatja bűnözőknek.

A vírushoz hasonló a **féreg**. Amíg a vírus más programokhoz kapcsolódik, és csak akkor aktív, ha a gazdaprogramját végrehajtjuk, addig a féreg önálló program. A féreg úgy terjed, hogy a hálózaton továbbküldi saját másolatait más számítógépekre. A Windows-rendszereknek általában van egy indítókönyvtáruk minden felhasználó számára; az ebben a könyvtárban lévő valamennyi program automatikusan elindítódik a bejelentkezéskor. A féregnek csak azt kell tennie, hogy berakja magát a távoli számítógép indítókönyvtárába. Más módszer (amit még nehezebb felfedezni) is van arra, hogy a távoli gép végrehajtsa a fájlrendszerébe másolt programot. A féreg hatása ugyanaz lehet, mint a vírusé. Valójában nem egyszerű megkülönböztetni a vírust a féregtől, néhány malware mindkét módszert használja a terjedésre.

A malware egy másik csoportjába tartozik a **trójai faló**. Ez olyan program, amely kétségtelenül hasznos tevékenységet végez, lehet játék vagy állítólagos segédprogram „javított" változata. Azonban amikor a trójai falovat végrehajtják, akkor más tevékenységet is végez, indíthat férget vagy vírust, vagy más rosszindulatú dolgot tehet. A trójai faló hatása körmönfont és rejtett. Ellentétben a féreggel és a vírussal, a trójai falovat a felhasználó saját elhatározásából tölti le, és amikorra kiderül, hogy mi is valójában, törlik a letöltési helyről.

Egy másik biztonsági probléma volt annak idején a munkatársak megbízhatatlanságából származó **logikai bomba**. Ez az eszköz egy kis kódrészlet, amelyet a vállalat egy programozó alkalmazottja készít, és titokban beilleszt a készített operációs rendszerbe. Amíg a programozó naponta „eteti" a kívánt jelszóval, addig nem történik semmi. Ha azonban a programozót előzetes figyelmeztetés nélkül elbocsátják, vagy nem kapja meg a beígért juttatást, a következő napon, amikor a rendszer nem kapja meg a jelszót, működésbe lép a bomba.

A működés hatása lehet lemeztörlés, fájlok véletlenszerű törlése, kulcsfontosságú programokon végrehajtott nehezen kideríthető változtatás vagy fontos fájlok titkosítása. Az utóbbi esetben a vállalat választhat, vagy hívja a rendőrséget (ami több hónappal későbbi ítéletet eredményez, vagy még azt sem), vagy enged a zsarolásnak, és csillagászati összegért újra alkalmazza a volt programozót mint konzultánst, hogy a hibát kijavítsa (és reménykedhet, hogy közben nem ültet el újabb logikai bombát).

A malware egy másik változata a **kémprogram**. Kémprogramot általában webhely meglátogatásával kaphatunk. A legegyszerűbb formája a **süti**. A süti egy kisméretű fájl, amelyet a webszolgáltató és a webböngésző kicserél. Ennek legális célja van. A süti olyan információt tartalmaz, amely lehetővé teszi a használója

azonosítását. Ez olyan, mint amikor kapunk egy jegyet, ha leadjuk kerékpárunkat javításra. Amikor visszatérünk a műhelybe, a jegy egyik felét összeillesztik a kerékpáréval (és a számlával). A webkapcsolat nem tartós, például amikor érdeklődést mutatunk egy könyv megvásárlására egy internetes könyvesboltban, akkor a bolt szolgáltatója megkéri a webböngészőt, hogy fogadjon el egy sütit. Amikor befejezzük a böngészést és egy másik könyvet is megjelöltünk vásárlásra, akkor olyan oldalra kattintunk, ahol a vásárlást befejezzük. Ekkor a webkiszolgáló visszakéri a böngészőtől az eltárolt sütit. Az ebben tárolt információkból össze tudja állítani a megvásárolandó könyvek listáját.

Normálisan az ilyen célra használt sütik élettartama hamar lejár. A süti nagyon hasznos technika, az e-kereskedelem erre épül. Azonban van olyan weboldal, amely nem ilyen jóindulatúan használja. Például hirdetések gyakran reklámoznak olyan vállalkozást, amely nem az információ szolgáltatója. A hirdetők ezért a privilégiumért fizetnek a weboldal tulajdonosának. Ha egy kerékpáralkatrészt hirdető oldal sütit helyez el, majd később egy ruházati cikkeket árusító helyre látogatunk, akkor ugyanaz a hirdető cég ezen az oldalon is adhat reklámot, és begyűjthet olyan sütit, amelyet máshonnan kaptunk. Hirtelen olyan reklámot láthatunk, amely olyan ruhákat reklámoz, amelyeket speciálisan kerékpározók használnak. A hirdetők sok információt begyűjthetnek ilyen módon az érdeklődési körünkről, ezért nem kell sok mindent közölni magunkról.

Ami rosszabb, weboldalak sokféleképpen letölthetnek végrehajtható kódot számítógépünkre. A legtöbb böngésző elfogad olyan **bővítményeket**, amelyekkel a böngésző képességei kiterjeszthetők, például új típusú fájl megjelenítése a képernyőn. A felhasználók gyakran elfogadják a bővítményt anélkül, hogy tudnák, mit csinál. Vagy például a felhasználó szívesen elfogad olyan bővítményt, amely táncoló cica kurzort ad. Elég egy hiba a böngészőben, hogy távolról nemkívánatos programot telepítsenek a gépünkre, esetleg olyan oldalra csábítva, amely kihasználhatja sebezhetőségünket. Minden esetben, amikor idegen programot fogadunk el, önként vagy sem, fennáll a veszély, hogy ártalmas számunkra.

Véletlen adatvesztés

A kártékony behatolók fenyegetése mellett értékes adatokat véletlenül is elveszíthetünk. A véletlen adatvesztés néhány általános esete:

1. Természeti csapás: árvíz, földrengés, háború, zavargás, vagy ha rágcsálók megrágják a szalagokat, lemezeket.
2. Hardver- vagy szoftverhibák: hibás processzorműködés, olvashatatlan lemez vagy szalag, telekommunikációs hiba, programhiba.
3. Emberi tévedés: hibás adatbevitel, nem megfelelő szalag vagy lemez behelyezése, hibás program indítása, vagy valami más tévedés.

A legtöbb ezek közül kezelhető megfelelő mentéssel, lehetőleg az eredeti adatoktól távol tárolva. Az adatok védelme a véletlen adatvesztés ellen egyszerűbbnek tűnhet, mint a leleményes behatolók elleni védekezés, de a gyakorlatban az előbbi több kárt okoz, mint az utóbbi.

5.4.2. Általános biztonság elleni támadások

A biztonsági hézagok kiderítése nem egyszerű dolog. Egy rendszer biztonságának ellenőrzésére szokásos módszer egy **tigriscsapat**ként vagy **behatoló csapat**ként ismert szakértői csapat megbízása azzal, hogy kíséreljen meg betörni a rendszerbe. Hebbard és társai (Hebbard et al., 1980) is ezt próbálták végzett hallgatók segítségével. Évek során ezek a behatoló csapatok számos olyan területet fedeztek fel, ahol a rendszerek valószínűleg gyengék. Alábbiakban felsorolunk néhány nagyon gyakori betörési kísérletet, amelyek sokszor sikeresek. Operációs rendszer tervezésekor legyünk óvatosak, hogy rendszerünk ellenálljon az ilyen támadásoknak.

1. Igényeljünk memórialapot, lemezterületet, szalagterületet, és csak olvassuk el tartalmukat. Sok rendszer nem törli a tartalmat, mielőtt a felhasználó számára lefoglalja, így az tele lehet hasznos információval, amit az előző felhasználó írt a területre.
2. Próbálkozzunk illegális rendszerhívással, vagy legális hívással, illegális paraméterekkel, vagy legális hívással és legális, de értelmetlen paraméterrel. Sok rendszert össze lehet így zavarni.
3. Bejelentkezés közben nyomjunk DEL, RUBOUT vagy BREAK billentyűt. Néhány rendszerben a jelszót ellenőrző program megszakad, és a bejelentkezést sikeresnek tekinti.
4. Próbáljuk meg módosítani a felhasználói területén lévő komplex operációsrendszer-adatszerkezeteket. Bizonyos (különösen nagygépes) rendszerekben egy fájl megnyitásakor a program nagy adatszerkezeteket épít fel, amelyben ott van a fájl neve és más paraméterek, és ezt átadja a rendszernek. A fájl írása és olvasása során a rendszer néha módosítja ezeket az adatszerkezetcket. Bizonyos mezők megváltoztatása tönkreteheti a rendszer biztonságát.
5. Próbáljuk meg becsapni a felhasználókat azzal, hogy programunk a „login:" szöveget írja a képernyőre, bejelentkezési képernyőt szimulálva. Sok felhasználó megpróbál bejelentkezni, begépeli az azonosítóját és jelszavát, amit aztán a program eltárol gonosz mestere számára.
6. Keressünk kézikönyvekben „Ne tedd *X*-et!" formájú szöveget. Próbáljuk ki az *X* tevékenység lehetséges variációit.
7. Győzzünk meg egy rendszerprogramozót, hogy módosítsa a rendszert úgy, hogy az ugorja át a biztonsági ellenőrzést a mi azonosítónkkal bejelentkező felhasználó esetén. Ez a támadás **csapóajtó** néven ismert.
8. Ha semmi nem megy, akkor próbálkozzunk a számítóközpont titkárnőjének megvesztegetésével. A titkárnő valószínűleg könnyen hozzáférhet csodálatos információkhoz és rendszerint alulfizetett. Ne becsüljük le az emberi tényezők okozta problémákat.

Ezek és más támadási kísérletek megtalálhatók Linde könyvében (Linde, 1975). Bőséges irodalma van a biztonságnak és a biztonság ellenőrzésének, különösen a világhálón. A Windows-rendszerekre vonatkozó friss munkát lásd (Johanson és Riley, 2005).

5.4.3. Tervezési elvek a biztonság érdekében

Saltzer és Schroeder (Saltzer és Schroeder, 1975) kimutatott több általános elvet, amelyek útmutatóként használhatók biztonságos rendszerek tervezéséhez. Az alábbiakban ötleteik rövid felsorolását adjuk (a MULTICS-rendszer alapján).

Először is a rendszer terve legyen nyilvános. Annak feltételezése, hogy a behatoló nem ismeri a rendszer működését, csak a tervezők becsapására jó.

Másodszor, az legyen az alapértelmezés, hogy nincs hozzáférés. A legitim hozzáférés visszautasításából származó hibák sokkal gyorsabban jelezhetők, mint azok, amelyekben jogosulatlan hozzáférés megengedett.

Harmadszor, ellenőrizni kell az aktuális jogosultságokat. A rendszer ne csinálja azt, hogy ellenőrzi a jogosultságokat, megállapítja, hogy a hozzáférés megengedett, aztán ezt az információt eldugja valahova. Sok rendszer csak akkor ellenőrzi a jogosultságokat, amikor a fájlt megnyitja, később már nem. Ez azt jelenti, hogy a felhasználó, aki megnyit egy fájlt, majd hetekig megnyitva tartja, folyamatosan rendelkezik a hozzáféréssel, akkor is, ha a fájl tulajdonosa közben már régen megváltoztatta a jogosultságokat.

Negyedszer, a processzusoknak a lehető legkisebb jogosultságuk legyen. Egy szövegszerkesztő, amelynek csak az általa szerkesztett fájlhoz van hozzáférése (amit az indításakor meg kell adni), akkor sem tud nagy kárt okozni, ha tartalmaz trójai falovat. Ez az elv kifinomult védelmi mechanizmust követel. Ebben a fejezetben később tárgyalni fogunk ilyen mechanizmusokat.

Ötödször, a védelmi mechanizmus egyszerű, egységes legyen, és az operációs rendszer legalsó rétege valósítsa meg. Egy létező, nem megbízható rendszert biztonságossá tenni szinte lehetetlen. A biztonság, ugyanúgy, mint a helyesség, nem hozzáadható tulajdonság.

Hatodszor, a választott séma pszichológiailag elfogadható legyen. Ha a felhasználó úgy érzi, hogy fájljainak védelme túl sok munkát igényel, akkor inkább nem törődik vele. Ennek ellenére hangosan panaszkodik, ha valami nincs rendben. Ekkor az „Ez az Ön hibája" válasz általában nem elfogadható.

5.4.4. Felhasználó azonosítása

Sok védelmi séma azon a feltételezésen alapszik, hogy a rendszer ismeri minden felhasználója kilétét. Azt a feladatot, amely belépéskor a felhasználó kilétének azonosítását jelenti, **felhasználóazonosítás**nak nevezzük. A legtöbb azonosítási módszer olyan dolog azonosságának a megállapításán alapszik, amit a felhasználó tud, vagy amivel rendelkezik, vagy amivel azonos.

Jelszavak

A legszélesebb körben használatos azonosítás jelszó megadását követeli a felhasználótól. A jelszavak védelmét egyszerű megérteni és megvalósítani. A Unixban a következőképpen működik: a bejelentkeztető (login) program bekéri a felhasználó azonosítóját és jelszavát. A jelszót azonnal titkosítja. A login program ezután a jelszófájlban, amely felhasználónként egy ASCII szövegsort tartalmaz, megkeresi a felhasználó azonosítóját tartalmazó sort, és összehasonlítja az ott titkosítva tárolt jelszót a begépelt és titkosított jelszóval. Ha egyezést talál, akkor a belépést engedélyezi, egyébként visszautasítja.

A jelszavas azonosítást könnyű becsapni. Gyakran olvasható, hogy középiskolások vagy akár általános iskolások megbízható otthoni gépüket használva behatoltak szigorúan titkos rendszerbe, amely valamely óriás cég vagy állami intézmény tulajdona. A behatolás látszólag valamennyi esetben a felhasználó azonosítójának és jelszavának megsejtése miatt történt.

Habár több friss kutatás is folyt ezen a területen (többek között Klein, 1990), a jelszavas biztonságról a klasszikus mű Morris és Thompson Unixra vonatkozó munkája (Morris és Thompson, 1979) maradt. Összegyűjtötték a gyakori jelszavakat: utónevek, családnevek, utcanevek, városnevek, közepes méretű szótár szavai (visszafelé is), autórendszámok és rövid véletlen szövegek. Ezután ismert jelszó-titkosítási algoritmussal titkosították ezeket a szavakat, és elkezdték ellenőrizni, hogy melyek fordulnak elő a listájukban. Azt tapasztalták, hogy több mint 86 százalékuk előfordul.

Ha minden jelszó 7 karakterből állna, akkor a 95 nyomtatható karaktert felhasználva a keresési tér mérete 95^7, ami körülbelül 7×10^{13}. Ha másodpercenként 1000 jelszót tudnánk titkosítani, akkor 2000 év kellene a teljes lista felépítéséhez. Továbbá a lista 20 millió mágnesszalagot töltene meg. Ha legalább azt követelnénk meg, hogy minden jelszó tartalmazzon legalább egy kisbetűt, egy nagybetűt, egy speciális karaktert és legalább 7 vagy 8 hosszú legyen, már az is jelentős javulást eredményezne a szabadon választható jelszavakhoz képest.

Jóllehet nem várható el a felhasználótól, hogy alkalmas jelszót válasszon, mégis Morris és Thompson bemutatott egy technikát, amelynek alkalmazása majdnem lehetetlenné tenné az általuk próbált támadásokat (előre elkészített titkosított jelszavak listája). Az volt az ötletük, hogy minden jelszóhoz adjunk hozzá egy n bites véletlen számot. A jelszó változásával maga a véletlen szám is változna. A véletlen számot is titkosítottan kell tárolni a jelszófájlban. Pontosabban titkosításkor a véletlen számot hozzá kell adni a jelszóhoz, és az így kapott szót kell titkosítani és tárolni a jelszófájlban.

Vizsgáljuk meg, hogy milyen következményekkel jár ez a behatolóra nézve. A behatoló a gyakori jelszavakat titkosított formában összegyűjti egy listába rendezetten, hogy a keresést gyorsítsa, és a listát egy *f* fájlban tárolja. Ha a behatoló úgy sejti, hogy *Marilyn* jó jelszó lehet, akkor most már nem elég csak a *Marilyn* szót titkosítva felvennie a listába, hanem fel kell vennie a *Marilyn0000, Marilyn0001, Marilyn0002,* és így tovább, szavakat is. Ami azt jelenti, hogy mind a 2^n elemet el kell helyeznie az *f* fájlban. Ez a technika 2^n-szeresére növeli a fájl méretét. A Unix

ezt a módszert használja, ahol $n = 12$. Ezt a technikát a jelszófájl **sózás**ának is hívják. Néhány Unix magát a jelszófájlt olvashatatlanná teszi, és biztosít egy programot, amellyel kérésre keresni lehet benne, de késleltetéssel, hogy a támadót lassítsa.

Habár ez a módszer védelmet nyújt az olyan behatolókkal szemben, akik előre gyártott titkosított jelszavakkal operálnak, de keveset tesz az olyan felhasználók védelmében, akiknek az azonosítója *David* és a jelszava is *David*. Az egyik lehetséges mód arra, hogy rávegyük a felhasználót jobb jelszó választására, hogy a számítógép adjon tanácsokat. Vannak számítógépek, amelyek jelszóként használható, véletlenszerű, könnyen kiejthető, de értelmetlen szavakat generálnak, úgymint fotally, garbungy vagy bipitty (célszerű persze nagybetűt és speciális karaktert is bevenni).

Más gépek megkövetelik, hogy a felhasználó rendszeresen változtassa jelszavát, ezzel is csökkentve a jelszó ellopása okozta kárt. Ennek legszélsőségesebb esete az **egyszer használatos jelszó**. Ekkor a felhasználó kap egy jelszavakat tartalmazó listát. Minden bejelentkezéskor a listából a következő jelszót kell használnia. Így ha a behatoló felfedez egy jelszót, azzal nem megy semmire, mert a következő alkalommal másikat kell alkalmaznia. Javasolni kell persze a felhasználónak, hogy ne veszítse el ezt a listát.

Az majdnem kimaradt, hogy mialatt a jelszót gépeljük, a begépelt karakterek nem látszódhatnak a képernyőn, megvédve bennünket a terminál közelében leskelődő szemek elől. Kevésbé nyilvánvaló, hogy jelszót sohasem tárolhatunk – még titkosítottan sem – a számítógépen. Továbbá még a számítóközpont munkatársai sem tárolhatják jelszavak titkosított példányait. Bárhol is tárolunk jelszavakat titkosított formában, az bajok forrása lehet.

A jelszavas azonosítás elképzelésének egyik változata az, amikor minden új felhasználótól bekérünk egy hosszú kérdés-felelet listát, amelyet titkosítottan tárol a számítógép. A kérdéseket úgy kell megválasztani, hogy a felhasználónak ne kelljen leírnia. Tipikus kérdések a következők lehetnek:

1. Ki Margit nővére?
2. Milyen utcában volt az általános iskolája?
3. Mit tanított Weöres tanár úr?

Bejelentkezéskor a rendszer véletlenszerűen választ egy kérdést a listából, és ellenőrzi a választ.

Egy másik változat az ún. **kihívás-válasz**. Ekkor a felhasználó választ egy algoritmust, amikor megkapja az azonosítóját, például x^2. Bejelentkezéskor a rendszer megjelenít egy argumentumot, például 7, amire a felhasználónak a 49 értéket kell begépelnie. Az algoritmus különböző lehet reggel és este, a hét különböző napjain, különböző terminálokról történő bejelentkezéskor, és így tovább.

Fizikai azonosítás

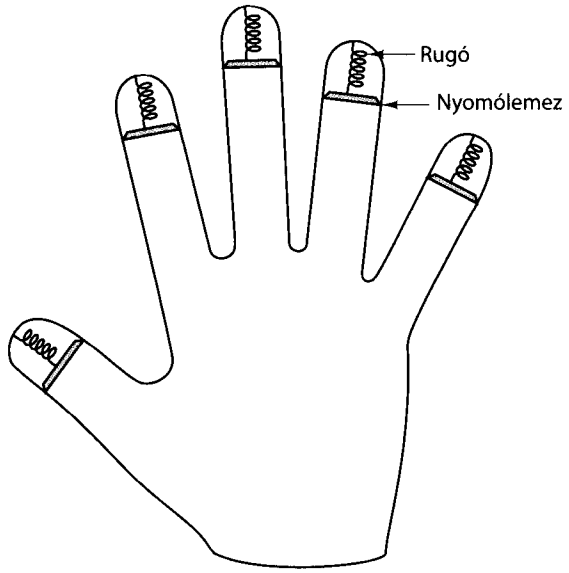
Teljesen különböző azonosítási megközelítés az, amikor azt ellenőrzik, hogy a felhasználónál van-e egy adott tárgy, például egy műanyag kártya, amelyen mágneses csík van. A kártyát beillesztik a terminálba, amely ellenőrzi, hogy kié a kártya. Ez a módszer kombinálható jelszóval, így a felhasználó csak akkor tud bejelentkezni, ha (1) megvan a kártyája, ha (2) tudja a jelszót. A banki pénzkiadó automaták gyakran így működnek.

Ismét más a nehezen hamisítható biometriai tulajdonságok mérésén alapuló azonosítási rendszer. Például a terminálba épített ujjlenyomat- vagy hanglenyomat-felismerő azonosíthatja a felhasználót. (Az eljárás gyorsítható, ha a felhasználónak meg kell adnia az azonosítóját is, így nem kell az adatbázisban keresni, csak összehasonlítani.) A közvetlen vizuális felismerés még nem megoldott, de eljöhet annak is az ideje.

További technika az aláírás-elemzés. A felhasználó a terminálhoz kötött speciális tollal aláírja a nevét, amelyet a gép összevet a gépben tárolt változattal. Még jobb módszert kapunk, ha nem az aláírásokat hasonlítjuk, hanem a gép az írás közben végzett tollmozgásokat hasonlítja. Egy jó utánzó le tudja másolni az aláírást, de nem ismeri a tollvonások sorrendjét.

Az ujjhossz szerinti azonosítás meglepően praktikus. Ekkor minden terminálhoz tartozik egy olyan eszköz, mint amit az 5.23. ábra mutat. A felhasználó bedugja a kezét ebbe az eszközbe, az leméri az ujjak hosszát, és összeveti az adatbázisával.

Sorolhatnánk továbbá a bonyolult példákat, de a következő két lehetőség érdekes szempontokat világít meg. Macskák és más állatok úgy jelölik ki területiu-



5.23. ábra. *Ujjhosszat mérő berendezés*

mukat, hogy a határát körbevizelik. A macskák kétségtelenül felismerik egymást a vizeletükről. Képzeljük el, hogy valaki előáll egy kis eszközzel, amely vizeletet tud elemezni. Ez holtbiztos azonosítást tenne lehetővé. Minden terminált fel kellene szerelni egy ilyen eszközzel és egy diszkrét felirattal: „Bejelentkezéshez legyen szíves ide mintát adni." Ez teljesen feltörhetetlen rendszer lenne, de a felhasználókkal valószínűleg nehezen lehetne elfogadtatni.

Hasonló mondható el arról a rendszerről is, amely egy rajzszegből és egy kis spektrográfból állna. A felhasználónak bejelentkezéskor a hüvelykujját bele kellene nyomnia a rajzszegbe, hogy vérmintát vegyenek, amit a spektrográf elemezne. Itt az a probléma, hogy minden azonosítási módszernek a felhasználói közösség számára pszichológiailag elfogadhatónak kell lennie. Az ujjhossz mérése valószínűleg nem ütközne problémába, de még az a kevésbé tolakodó megoldás, mint az ujjlenyomat számítógépes tárolása is, sok ember számára elfogadhatatlan lehet.

Ellenintézkedések

A számítógépeket üzembe helyezők, akik komolyan veszik a biztonságot – néhányuk csak miután már előfordult behatolás és jelentős károkat okozott –, gyakran tesznek lépéseket a jogosulatlan belépés megnehezítése érdekében. Például minden felhasználó csak meghatározott terminálról, a hét meghatározott napjain és meghatározott időben jelentkezhet be.

Telefonvonalon keresztül való bejelentkezés az alábbiak szerint működhet. Bárki bejelentkezhet telefonvonalon, de a bejelentkezés után a rendszer azonnal bontja a vonalat, és az előre megegyezett számon visszahívja a felhasználót. Ez az intézkedés azt jelenti, hogy a behatoló nem próbálkozhat akármilyen telefonszámról, csak a felhasználó (otthoni) számáról tudna betörni. Minden esetben, akár van visszahívás, akár nincs, a rendszer legalább 10 másodpercet tölt a jelszó ellenőrzésével, és ezt az időtartamot minden sikertelen próbálkozás után emelnie kell, hogy csökkentse a behatoló próbálkozási lehetőségeit. Három sikertelen próbálkozás után a vonalat meg kell szakítani, és értesíteni kell a biztonsági szakembert.

Minden bejelentkezést rögzíteni kell. Amikor egy felhasználó bejelentkezik, a rendszernek közölni kell azt, hogy mikor és honnan jelentkezett be legutóbb, így az esetleges behatolást észreveheti.

A következő megteendő lépés csapda felállítása a behatoló elfogására. Az egyszerű séma a következő: speciális felhasználói azonosító egyszerű jelszóval (mint a guest azonosító, guest jelszóval). Amikor bárki bejelentkezik ezen a néven, a rendszer biztonsági specialistáját azonnal értesíteni kell. Az operációs rendszerben a könnyen megtalálható, a behatolók akció közbeni elfogására tervezett hibák is lehetnek csapdák. Alapmunka Stoll cikke (Stoll, 1989), amely leírja egy egyetem számítógépébe betörő, katonai titkokat kereső kém elfogását.

5.5. Védelmi mechanizmusok

Az előző alfejezetekben több potenciális biztonsági problémát tanulmányoztunk, ezek egy része technikai, mások nem. A következő részekben azokra az operációs rendszerben használt részletes technikai módszerekre koncentrálunk, amelyekkel fájlok és más erőforrások védhetők meg. Minden ilyen technika világosan különbséget tesz az eljárásmód (mely adatok kik ellen védelmezendők) és a mechanizmus között (hogyan valósítja meg a rendszer a védelmet). Az eljárásmód és a mechanizmus elválasztásáról írt Sandhu (Sandhu, 1993). Nálunk a hangsúly a mechanizmuson van, és nem az eljárásmódon.

Néhány rendszerben a védelmet egy program, az ún. **referenciamonitor** végzi. Valahányszor hozzáférést kezdeményeznek egy potenciálisan védelmezett erőforráshoz, a rendszer megkéri a referenciamonitort, hogy ellenőrizze a jogosultságot. A referenciamonitor táblázatokban keresve hozza meg a döntést. A továbbiakban leírjuk azt a környezetet, amelyben a referenciamonitor működik.

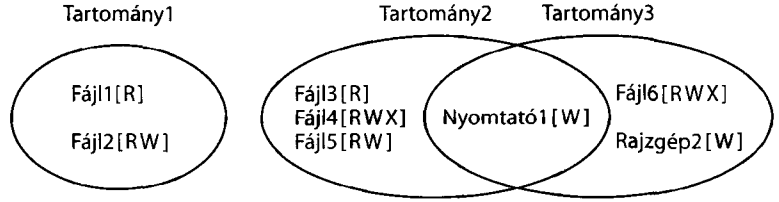
5.5.1. Védelmi tartományok

Minden számítógépes rendszer sok olyan objektumot tartalmaz, amelyek védelmet igényelnek. Ezek lehetnek hardver- (többek között processzorok, memóriaszegmensek, lemezmeghajtók, nyomtatók), illetve szoftverobjektumok (többek között processzusok, fájlok, adatbázisok, szemaforok).

Minden objektumnak egyedi neve van a rendszerben, amellyel hivatkozni lehet, és mindegyikhez tartozik egy véges sok műveletből álló halmaz, amelyeket a processzusok az objektumon végrehajthatnak. Például a read és a write művelet fájlokon működik, az up és down pedig szemaforokra vonatkozik.

Nyilvánvaló, hogy szükség van olyan módszerre, amellyel megtiltható, hogy egy processzus hozzáférjen olyan objektumokhoz, amelyek elérésére nincs jogosultsága. Továbbá az ilyen mechanizmusnak biztosítania kell, hogy igény esetén a processzusok által végezhető műveleteket a legális műveletek egy részhalmazára korlátozzuk. Például egy A processzus jogosult olvasni az F fájlt, de nem jogosult írni.

A különböző védelmi mechanizmusok tárgyalásához hasznos bevezetni a tartomány (domain) fogalmát. A **tartomány** (objektum, jogok) párosok halmaza. Min-



5.24. ábra. *Három védelmi tartomány*

den pár tartalmaz egy objektumot és egy rajta végezhető műveleteket tartalmazó halmazt. A **jogok** ebben az összefüggésben művelet-végrehajtási jogosultságot jelentenek.

Az 5.24. ábra három tartományt mutat, feltüntetve az objektumokat és a jogokat: [Read, Write, eXecute] (Olvasás, Írás, Végrehajtás). Vegyük észre, hogy a *Printer1 (Nyomtató1)* objektum egyszerre két tartománynak is eleme. Bár ez a példa nem mutatja, de lehetséges, hogy ugyanaz az objektum több különböző tartomány eleme legyen, mindegyikben *különböző* jogokkal.

Minden processzus minden egyes időpillanatban meghatározott védelmi tartományban fut. Más szóval, meghatározott objektumokat érhet el, és minden objektumra meghatározott jogokkal rendelkezik. A processzusok futás közben átválthatnak egyik tartományról egy másikra. A tartományváltoztatás szabályai erősen rendszerfüggők.

A védelmi tartomány fogalmának konkrétabb kifejtése végett tekintsük a Unix-rendszert. Minden Unix-processzus tartományát egyértelműen meghatározza a hozzá tartozó uid (felhasználóazonosító) és gid (csoportazonosító). Minden adott (uid, gid) párra egyértelműen meghatározható mindazon objektumok halmaza (fájlok, beleértve az I/O-eszközöket is, amelyeket speciális fájlok reprezentálnak), amelyeket a processzus elérhet, és az is, hogy az objektum elérhető-e írásra, olvasásra és végrehajtásra. Bármely két processzus ugyanazon (uid, gid) értékkel pontosan ugyanazon objektumokat érheti el. Különböző (uid, gid) párral rendelkező processzusok által elérhető fájlok halmaza különböző lesz, bár a legtöbb esetben jelentős átfedés van.

Továbbá minden Unix-processzus két részből áll: a felhasználói és a kernelrészből. Amikor egy processzus rendszerhívást végez, akkor a felhasználói részről átvált a kernelrészre. A kernelrész más objektumokhoz férhet hozzá, mint a felhasználói rész. A kernelrész hozzáférhet például a fizikai memória minden lapjához, a teljes lemezterülethez és minden más védett erőforráshoz. Tehát a rendszerhívások tartományváltást eredményeznek.

Amikor egy processzus exec műveletet hajt végre egy fájlon, aminek SETUID vagy SETGID bitje be van állítva, új tényleges uid vagy gid értékre tesz szert.

Tartomány	Tárgy: Fájl1	Fájl2	Fájl3	Fájl4	Fájl5	Fájl6	Nyomtató1	Rajzgép2
1	Olvasás	Olvasás Írás						
2			Olvasás	Olvasás Írás, Végrehajtás	Olvasás Írás		Írás	
3						Olvasás Írás, Végrehajtás	Írás	Írás

5.25. ábra. *Védelmi mátrix*

Tartomány	Tárgy: Fájl1	Fájl2	Fájl3	Fájl4	Fájl5	Fájl6	Nyomtató1	Rajzgép2	Tartomány1	Tartomány2	Tartomány3
1	Olvasás	Olvasás Írás								Enter	
2			Olvasás	Olvasás Írás, Végrehajtás	Olvasás Írás		Írás				
3						Olvasás Írás, Végrehajtás	Írás	Írás			

5.26. ábra. *Védelmi mátrix, ahol a tartományok is objektumok*

Olyan program futtatása, amely SETUID vagy SETGID beállítású, szintén tartományváltást jelent, mert jogosultságai megváltoznak.

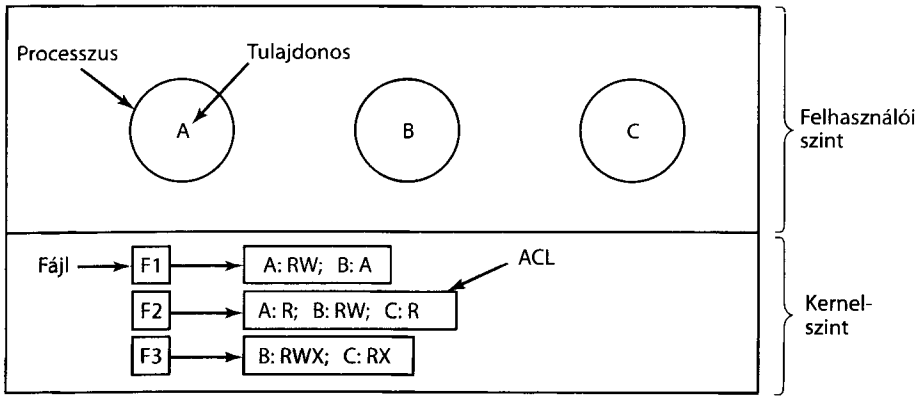
Fontos kérdés, hogy miként tudja a rendszer nyilvántartani, hogy mely objektumok milyen tartományban vannak. Fogalmi szinten elképzelhetünk egy hatalmas mátrixot, amelynek sorai a tartományok és oszlopai az objektumok. A mátrix adott eleme tartalmazza azokat a jogokat, amelyekkel a tartományban az objektum rendelkezik. Az 5.24. ábrán adott tartományok mátrixos ábrázolását mutatja az 5.25. ábra. Ha adott egy ilyen mátrix és az aktuális tartomány sorszáma, akkor a rendszer el tudja dönteni, hogy egy adott objektum elérése megengedett-e.

A tartományváltás szintén beilleszthető a mátrixmodellbe, mert vegyük észre, hogy a tartomány is tekinthető objektumnak, amelyen az enter (belépés) művelet értelmezett. Az 5.26. ábra csak abban különbözik az 5.25. ábrával adott mátrixtól, hogy itt feltüntettük a három tartományobjektumot is. Az 1. tartománybeli processzusok átválthatnak a 2. tartományba, de ha már ott vannak, nem mehetnek vissza. Ez a szituáció modellezi a Unix SETUID-programok végrehajtását is. Példánkban más tartományváltoztatás nem megengedett.

5.5.2. Hozzáférést vezérlő listák

A gyakorlatban alig fordul elő, hogy az 5.26. ábrán bemutatott védelmi mátrixot ténylegesen tárolnák, mivel az nagy és ritka. A legtöbb tartománynak az objektumok többségéhez egyáltalán nincs elérési joga, tehát nagyon nagy, majdnem üres mátrix tárolása tárterület-pazarlást jelentene. Két praktikus módszer kínálkozik: tároljuk a mátrix sorait vagy oszlopait úgy, hogy csak a nem üres elemet tároljuk ténylegesen. Ebben a szakaszban az oszlop szerinti tárolást vizsgáljuk, a következőben pedig a sor szerintit.

Az első technika abból áll, hogy minden objektumhoz hozzárendelünk egy (rendezett) listát; ez azokat a tartományokat tartalmazza, amelyek az adott objektumot elérhetik (az elérési móddal együtt). Az ilyen listát **hozzáférést vezérlő listá-**



5.27. ábra. *Hozzáférést vezérlő lista használata*

nak hívják, rövidítve **HVL**, és az 5.27. ábra szemlélteti. Ezen három tartományt látunk, *A*, *B* és *C*, és három fájlt, *F1*, *F2* és *F3*. Az egyszerűség kedvéért feltételezzük, hogy a három tartomány egy-egy felhasználóhoz tartozik; ezek *A*, *B* és *C*. A biztonsági irodalomban a felhasználókat gyakran **alanyok**nak vagy **megbízó**nak nevezik, hogy megkülönböztessék azoktól a dolgoktól, amelyeknek tulajdonosai; vagyis az **objektumok**tól, mint például a fájlok.

Minden fájlhoz tartozik egy HVL. Az *F1* fájl listájában két elem van (pontosvesszővel elválasztva). Az első elem azt mondja, hogy minden processzus, amelynek tulajdonosa az *A* felhasználó, olvashatja és írhatja a fájlt. A második szerint minden processzus, amelynek tulajdonosa a *B* felhasználó, olvashatja a fájlt. Ezen felhasználók minden más hozzáférése és minden más felhasználó akármilyen hozzáférése tiltott. Megjegyzendő, hogy a jogosultságokat a felhasználó adja meg, nem a processzus. A védelmi rendszer szerint minden *A* tulajdonában lévő processzus olvashatja és írhatja az *F1* fájlt. Nem számít, hány van, egy vagy akár 1000. A felhasználó, és nem a processzus azonosítója számít.

Az *F2* fájl listájában három elem van: *A*, *B* és *C* mindegyike olvashatja, és a *B* ezenfelül írhatja is. Más hozzáférés nem megengedett. Az *F3* fájl nyilvánvalóan végrehajtható program, *B* és *C* mindegyike olvashatja és végrehajthatja, *B* pedig írhatja is.

Ez a példa a HVL által megvalósítható védelmi mechanizmus legalapvetőbb formáját mutatja. A gyakorlatban sokszor jóval kifinomultabb rendszereket alkalmaznak. Kezdésként mi csak három jogosultságot vettünk: olvasás, írás és végrehajtás. Lehetnek más pótlólagos jogok is. Ezek egy része generikus, vagyis minden objektumra érvényes, más részük objektumspecifikus. A generikusra példa a törlés- és a másolásjog. Ezek minden objektumra alkalmazhatók, típusuktól függetlenül. Objektumspecifikus jog például üzenet hozzáadása a levelesládához, könyvtári objektum tartalmának ábécé szerinti rendezése.

Mindeddig a vizsgált HVL-elemek csak felhasználókhoz tartoztak. Sok rendszer alkalmazza a felhasználói **csoport** fogalmát. A csoportoknak van neve és sze-

Fájl	Hozzáférést vezérlő lista
Password	tana, sysadm: RW
Pigeon_data	bill, pigfan: RW; tana, pigfan: RW; ...

5.28. ábra. *Két hozzáférést vezérlő lista*

repelhetnek HVL-ben. A csoportok szemantikájának két változata lehet. Egyes rendszerekben minden processzusnak van felhasználói azonosítója (UID) és csoportazonosítója (GID). Ilyen rendszerekben egy HVL-elem a következőképpen nézhet ki:

```
UID1, GID1: jog1; UID2, GID2: jog2; ...
```

Ilyen feltételek mellett egy objektumra vonatkozó hozzáférési kérés esetén ellenőrzik a hívó UID-jét és GID-jét. Ha szerepelnek a HVL-listában, akkor az ott felsorolt jogokat megkapja. Ha a (UID, GID) pár nem szerepel a listában, akkor a hozzáférést megtagadják.

A csoportok ilyen módon való használata ténylegesen a **szerep** fogalmának a bevezetését eredményezi. Tekintsünk egy rendszert, amelyben Tana rendszer-adminisztrátor, és így szerepel a *sysadm* csoportban. Továbbá képzeljük el, hogy a vállalat klubokat működtet alkalmazottai számára, és Tana tagja a galambászok klubjának. A klubtagok a *pigfan* csoportba tartoznak, és hozzáférhetnek a galambok adatait tartalmazó adatbázishoz. A HVL egy része az 5.28. ábrán látható képet mutathat.

Ha Tana hozzáférést kezdeményez valamelyik fájlhoz, akkor a kimenetel attól függ, hogy melyik csoportba van éppen bejelentkezve. Bejelentkezéskor a rendszer megkérdezheti, hogy melyik csoportba kíván belépni, vagy két különböző felhasználói azonosító/jelszó lehet a megkülönböztetésre. A sémának az a célja, hogy Tana ne férhessen hozzá a password fájlhoz, ha galambászsapka van rajta. Ezt csak akkor teheti meg, ha adminisztrátorként jelentkezett be.

Bizonyos esetekben a felhasználó hozzáférhet adott fájlhoz, függetlenül attól, hogy melyik csoportba jelentkezett be. Ezek az esetek a **helyettesítőjel** bevezetésével kezelhetők, amelynek jelentése: mindenki. Például az alábbi elem a password fájl esetén azt jelenti, hogy Tana hozzáférhet, függetlenül attól, hogy melyik csoportba van bejelentkezve.

```
tana, *: RW
```

Egy másik lehetőség az, hogy a felhasználó megkapja a jogot, ha van olyan csoport, amelyre a jog biztosítva van, és a felhasználó tagja ennek a csoportnak. Ebben az esetben a több csoportba is tartozó felhasználónak nem kell megadnia, hogy melyik csoportba jelentkezik be. Mindegyik minden időpontban érvényes. Ennek a hátránya, hogy kisebb zártságot biztosít: Tana szerkesztheti a password fájlt a galambásztalálkozó során.

A csoportok és a helyettesítőjel együtt biztosítják fájlok hozzáférésének szelektív blokkolását. Például a

```
virgil, *: (none); *, *: RW
```

Virgil kivételével mindenkinek olvasási és írási jogot ad a fájlra. Ez azért működik, mert a bejegyzést balról jobbra szekvenciálisan értelmezik, és az első illeszkedő tagot alkalmazzák, a továbbiakat nem is vizsgálják. Virgil esetén az első tag illeszkedik, és a talált jog, most a (none) alkalmazódik. A keresés itt befejeződik. Azt, hogy a továbbiak szerint mindenki jogot kapna, nem is vizsgáljuk.

Egy másik módszer az lehet, hogy a HVL elemei nem (UID, GID) párokat tartalmaznak, hanem vagy UID, vagy GID elemeket. Például a *pigeon_data* fájlhoz az alábbi elem tartozna:

```
debbie: RW; phil: RW; pigfan: RW
```

ami azt jelenti, hogy Debbie, Phil és a *pigfan* csoport minden tagja olvashatja és írhatja a fájlt.

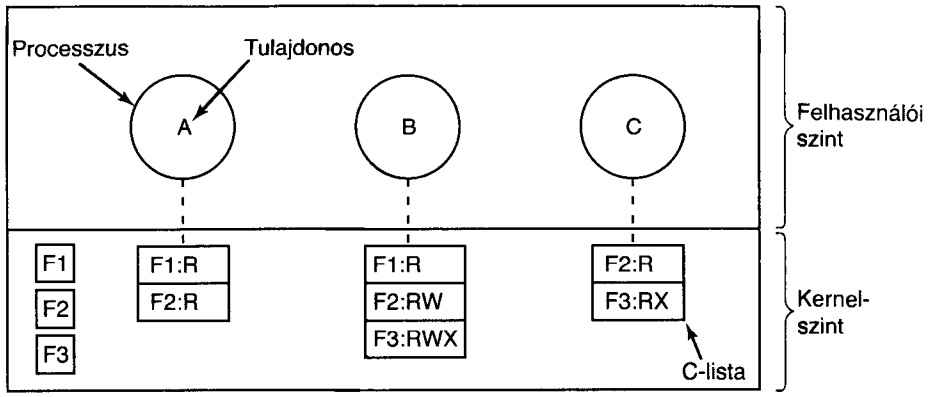
Előfordul, hogy egy felhasználó vagy csoport rendelkezik bizonyos jogokkal egy fájlhoz, de később a fájl tulajdonosa vissza akarja vonni a jogot. Hozzáférést vezérlő listák esetén viszonylag egyszerű a jog visszavonása. Nem kell mást tenni, mint szövegszerkesztővel átírni a megfelelő HVL-elemet. Azonban ha a fájl megnyitott állapotban volt a módosításkor, akkor a megváltozott jogok csak a következő megnyitáskor érvényesülnek. Minden megnyitott fájl esetén azok a jogok érvényesek, amelyekkel a megnyitáskor rendelkezett, még akkor is, ha azután a felhasználó egyáltalán nem jogosult a fájl elérésére.

5.5.3. Képességi listák

Az 5.26. ábrán látható védelmi mátrix felszeletelésének másik módja a soronkénti szeletelés. Ennél a módszernél minden processzushoz tartozik egy lista, amely azokat az objektumokat és a megfelelő hozzáférési jogokat tartalmazza, amelyeket a processzus elérhet, más szóval az adott processzus tartományát. Ezt **képességi listának** vagy **C-listának** nevezik, és az egyes elemeket a listán **képességek**nek hívják (Dennis és Van Horn, 1966; Fabry, 1974). Három processzus képességi listája látható az 5.29. ábrán.

Minden képesség a tulajdonosnak valamilyen jogot biztosít valamely objektumra. Az 5.29. ábra szerint például az *A* processzus tulajdonosa olvasási joggal rendelkezik az *F1* és *F2* fájlokra. Egy képesség általában egy fájl (vagy még általánosabban egy objektum) azonosítóját és a hozzá rendelt jogok bittérképét tartalmazza. Unix-rendszerekben a fájlazonosító valószínűleg az i-csomópont sorszáma lenne. A képességi listák maguk is objektumok, ezért más képességi listák hivatkozhatnak rájuk, ezzel lehetővé válik résztartományok megosztása.

Teljesen nyilvánvaló, hogy a képességi listákat meg kell védeni a felhasználóktól. A védelemre három módszer ismert. Az első módszer speciális hardverarchitektúrát, ún. **címkézetarchitektúrát** igényel, ahol minden memóriabeli szónak van egy extra bitje (címke), amely megmondja, hogy az képességet tartalmaz-e. A címke bit nem használatos az aritmetikai, összehasonlító és egyéb közönséges műveletekben, és nem módosítható, csak kernel módban futó program (vagyis az operációs rendszer) által. Ténylegesen építettek címkézett architektúrájú gépeket, amelyek igen jól működnek (Feustal, 1972). Az IBM AS/400 gép az egyik népszerű példa erre.



5.29. ábra. *Képességek használata esetén minden processzusnak van képességi listája*

A második módszer az, amikor a C-listát az operációs rendszer területén tárolják, és a processzusok csak a pozíciójukkal hivatkozhatnak a képességre. Egy processzus mondhatja, hogy „kérem 1 KB beolvasását a 2. képesség által mutatott fájlból". Ez a címzési mód hasonló a Unix által használt fájlleíróhoz. A Hydra- (Wulf et al., 1974) rendszer így működött.

A harmadik módszer szerint a C-listát a felhasználó területén tárolják, de titkosítva, így a felhasználó nem ronthatja el. Ez a módszer különösen alkalmas elosztott rendszerekben, és a következőképpen működik. Amikor a kliensprocesszus kérést küld a távoli kiszolgálónak, például egy fájlszervernek, hogy hozzon létre számára egy objektumot, a kiszolgáló létrehozza az objektumot, és generál egy nagy véletlen számot, az ellenőrző mezőt. A kiszolgáló fájl táblázatában lefoglalódik egy cella az objektum számára, ahol tárolja az ellenőrző mező értékét, a lemezblokkok címét és még egyebeket. Unix-terminológiát használva, az ellenőrző mező értékét a fájl i-csomópontjában tárolja a kiszolgáló. Ezt nem küldi vissza a felhasználónak, és sohasem adja ki a hálózatra. Ezután a kiszolgáló a felhasználó számára az 5.30. ábrán látható formájú képességet hoz létre. A felhasználóhoz visszaküldött képesség tartalmazza a kiszolgáló azonosítóját, az objektum azonosítóját (a kiszolgáló táblázatbeli indexét, alapvetően az i-csomópont sorszámát) és a jogosultságok bittérképét. Frissen létrehozott objektum minden jogosultsági bitje 1-re van állítva. Az utolsó mező tartalma egy olyan *f* egyértelmű függvény értéke az objektum, jogok és az ellenőrző mező argumentumokra alkalmazva, amely

Szerver	Objektum	Jogok	f(Objektum, Jogok, Ellenőrzés)

5.30. ábra. *Titkosítással védett képesség*

függvény kriptográfiailag biztos. Korábban tárgyaltuk az ilyen tulajdonságú függvényeket.

Amikor a felhasználó objektumelérést kezdeményez, a kérés részeként elküldi a képességet a kiszolgálónak. A kiszolgáló kiolvassa ebből az objektum táblabeli indexét, így találja meg az objektumot. Ezután kiszámítja az *f(objektum, jogok, ellenőrző)* értéket, az első két argumentumot a kérésben találja, a harmadikat pedig a saját táblázatában. Ha ez az érték megegyezik a képességben lévő negyedik paraméterrel, akkor az igényt engedélyezi, egyébként elutasítja. Ha egy felhasználó megpróbálja felhasználni más felhasználó objektumát, akkor nem tudja helyesen előállítani a negyedik mező értékét, mert nem tudja az ellenőrző értéket, így a kérését elutasítják.

A felhasználó kérhet a kiszolgálótól gyengébb képességet is, például csak olvasási jogot. A kiszolgáló először ellenőrzi, hogy a képesség érvényes-e. Aztán kiszámítja az *f(objektum, új_jogok, ellenőrző)* értéket, és tárolja a negyedik mezőben. Vegyük észre, hogy az ellenőrző érték ugyanaz, mert más fontos képességek ettől függnek.

Ezt az új képességet küldi vissza a kiszolgáló a processzusnak. A felhasználó odaadhatja ezt egy barátjának, egyszerűen elküldi neki levélben. Ha a barát a jogosultság-bitvektorban átállít 1-re olyan bitet, amely 0 volt, akkor a kiszolgáló ezt ki tudja deríteni, mert az *f* függvény értéke nem fog egyezni. Mivel a barát nem ismeri az ellenőrző értéket, ezért nem tud előállítani olyan képességet, amely megfelelne a megváltoztatott biteknek. Ezt a módszert az Amoeba-rendszerben fejlesztették ki, és intenzíven használták (Tanenbaum, 1990).

A specifikus objektumfüggő jogok mellett, mint az olvasás, végrehajtás a (kernel által és kriptográfiailag is védett) joga, vannak **generikus jogok** is, amelyek minden objektumra alkalmazhatók. Példák generikus jogokra:

1. Képesség másolása: új képesség létrehozása az adott objektumhoz.
2. Objektum másolása: objektum másolatának létrehozása új képességgel.
3. Képesség törlése: képesség törlése a C-listából, de az objektum nem változik.
4. Objektum törlése: objektum végleges eltávolítása a képességgel együtt.

Érdemes megjegyezni a képességi rendszerekkel kapcsolatban, hogy hozzáférési jog visszavonása meglehetősen bonyolult kernel módban. A rendszer számára nagyon nehéz megkeresni egy objektum képességeit a C-listákban, mert azok a lemezen szétszórtan helyezkednek el. Az egyik lehetséges megközelítés az, hogy a képességek nem az objektumra mutatnak, hanem egy közvetett objektumra. Mivel az a mutató, amely a közvetett objektumról a tényleges objektumra mutat, megszüntethető, ezáltal a rendszer érvényteleníthetí a képességet. (Amikor később egy közvetett objektum képességét kapja a rendszer, a felhasználó felfedezheti, hogy a közvetett objektum most null objektumra mutat.)

Az Amoeba séma esetén egyszerű a visszavonás. Csak annyit kell tenni, hogy megváltoztatjuk az objektumban az ellenőrző mező értékét. Egy csapásra minden képességet érvényteleníthetünk. Azonban egyik séma sem teszi lehetővé a szelektív visszavonást, például csak John jogosultságait, de senki másét nem. Ez a probléma valamennyi képességi rendszerben fennáll.

Egy másik probléma annak biztosítása, hogy egy képesség jogos tulajdonosa ne tudja 1000 példányban továbbadni azt barátainak. Kernel által kezelt képességek esetén, mint a Hydra-rendszerben ez megoldható, de a megoldás nem működik osztott rendszerekben, mint az Amoeba.

Másrészt a képességekkel nagyon elegánsan megoldható mobil kódok szeparálása. Külső program indításakor olyan képességi listát kap, amely csak azokat a képességeket tartalmazza, amelyeket a számítógép tulajdonosa adni akar, mint például írási jog a képernyőre, olvasási és írási jog csak a program számára létrehozott könyvtárra. Amikor egy mobil kód processzusa elindul csak ezekkel a korlátozott képességekkel, akkor nem tud más rendszererőforrásokat igénybe venni, és így ténylegesen korlátozva van a könyvtárára, anélkül hogy a kódot módosítani kellene, vagy interpreterrel kellene végrehajtani. Program futtatása a lehető legkevesebb jogosultsággal – ez a **legkevesebb privilégium elv**eként ismert és hatékony irányelv biztonságos rendszerek megvalósításakor.

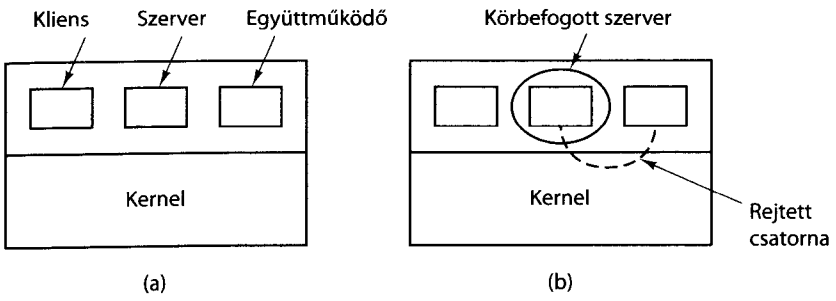
Röviden összefoglalva, a HVL és a képességek módszer bizonyos értelemben egymás ellentétjei. A képességek nagyon hatékonyak, mert ha egy processzus azt mondja, hogy „nyisd meg a 3. képesség által mutatott fájlt", akkor semmilyen ellenőrzést nem kell végezni. HVL esetén (lehet, hogy hosszú) keresést kell végrehajtani a listában. Ha csoportokat nem alkalmaznak, akkor egy mindenki által olvasható fájlhoz fel kell venni a listába az összes felhasználót. A képességek lehetővé teszik egy processzus elszeparálását, de a HVL nem. Másrészről a HVL lehetővé teszi jogok szelektív visszavonását, de a képességek nem. Végül ha egy objektumot törlünk, de a képességet nem, vagy ha a képességet töröljük, de az objektumot nem, akkor probléma keletkezik. A HVL esetén nincs ilyen probléma.

5.5.4. Rejtett csatornák

Még hozzáférést vezérlő lista és képességi lista alkalmazása esetén is lehetnek rések a biztonsági rendszerben. Ebben a részben azt vizsgáljuk, hogyan szivároghat ki információ a rendszerből még akkor is, ha szigorúan bebizonyították, hogy matematikailag lehetetlen a szivárgás. Az ötlet Lampsontól (Lampson, 1973) származik.

Lampson eredeti modelljét egyetlen időosztásos rendszerre fogalmazta meg, de az ötlet adaptálható lokális hálózatokra és más, többfelhasználós rendszerekre is. A legegyszerűbb formájában három processzus fut ugyanazon a védett gépen.

Az első processzus, a kliens munkára fogja a második processzust, a szervert. A kliens és a szerver nem bízik teljesen egymásban. Például a szerver feladata az,



5.31. ábra. *(a) A kliens, szerver és az együttműködő processzusok. (b) A körbefogott szerver rejtett csatornákon keresztül mégis tud adatokat juttatni az együttműködőnek*

hogy segítse a klienst adóbevallás kitöltésében. A kliens aggódik, hogy a szerver titokban eltárolja pénzügyi adatait, titkos listát készít arról, hogy ki menyit keresett, amit később elad. A szerver pedig attól tart, hogy a kliens ellopja az adóbevallást kezelő programot.

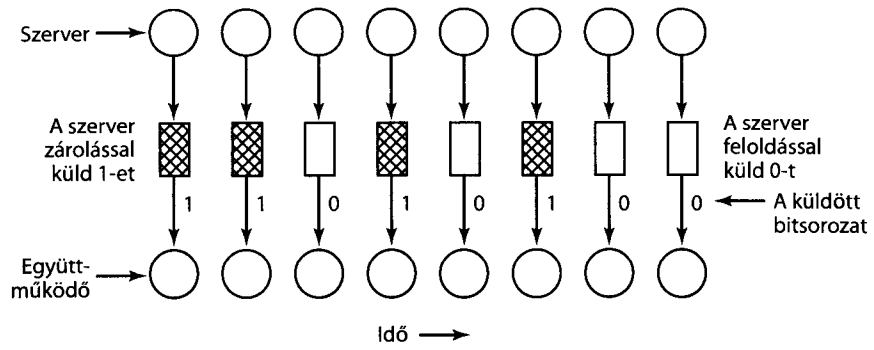
A harmadik processzus az együttműködő, amelyik konspirál a szerverrel, hogy az tényleg ellopja a kliens bizalmas adatait. A szerver és az együttműködő tulajdonosa tipikusan ugyanaz a felhasználó. Ezt a három processzust mutatja az 5.31. ábra. Ennek a gyakorlatnak az a célja, hogy olyan rendszert tervezzünk, amely lehetetlenné teszi, hogy a szerver jogosulatlanul átadja az együttműködőnek azokat az adatokat, amelyeket legitim módon kapott a klienstől. Lampson ezt **bezárás (confinement) problémá**nak nevezi.

A rendszer tervezése szempontjából az a cél, hogy körbevegyük vagy bezárjuk a szervert úgy, hogy ne tudjon átadni információt az együttműködőnek. A védelmi mátrix sémát használva könnyen garantálni tudjuk, hogy a szerver ne tudjon kommunikálni az együttműködővel úgy, hogy az ír olyan fájlba, amelyet az együttműködő elolvashat. Valószínűleg azt is tudjuk biztosítani, hogy a szerver és az együttműködő a rendszertaszkok közötti kommunikációs mechanizmusát felhasználva se tudjon kommunikálni.

Sajnos létrehozhatók körmönfontabb kommunikációs csatornák is. Például a szerver bitsorozattal a következő módon tud információt cserélni: 1-es bit küldése végett olyan keményen dolgozik, ahogy csak tud egy adott időtartamig; 0-s bit küldése végett pedig ugyanennyi ideig alszik.

Az együttműködő megpróbálhatja úgy kideríteni, hogy mit küld a szerver, hogy körültekintően figyeli a válaszidőket. Általában kisebb lesz a válaszidő, ha a szerver 0-t küld, és nagyobb (keményen dolgozik), ha 1-et küld. Ez a kommunikációs lehetőség **rejtett csatorna**ként ismert, és az 5.31.(b) ábra illusztrálja.

Természetesen a rejtett csatorna zajos, sok idegen információt is tartalmazhat, de megbízhatóan is lehet kommunikálni zajos csatornákon hibajavító kódot használva (úgymint Hamming-kód vagy még kifinomultabb módszerek). Hibajavító kód használata csökkenti a rejtett csatorna amúgy is szűk sávszélességét, de még így is tekintélyes mennyiségű információ áramoltatható ki. Teljesen nyilvánvaló,



5.32. ábra. *Rejtett csatorna megvalósítása fájlzárolással*

hogy nincs olyan védelmi modell, amely megvédhetne az ilyen problémáktól, ha a rendszer a védelmi tartomány elvén alapszik.

A CPU-terhelés modulálása nem az egyetlen lehetőség rejtett csatorna megvalósítására. A lapozási arány szintén modulálható (gyakori lapozás 1-et jelent, ritkán lapozás 0-t). Ténylegesen a rendszer minden időzített teljesítmény csökkentési módszere rejtett csatorna lehet. Ha az operációs rendszer biztosít valamilyen mechanizmust fájlok zárolására, akkor a szerver valamely fájlt zárolva 1-et tud küldeni, a zárolást felszabadítva pedig 0-t. Néhány rendszerben megengedett, hogy egy processzus olyan fájl zároltsági állapotát is le tudja kérdezni, amelyhez egyébként nincs hozzáférése. Ezt a rejtett csatornát szemlélteti az 5.32. ábra, ahol egy fájlt zárolnak és elengednek valamely rögzített időtartamra, amelyet ismer a szerver és az együttműködő is. Ebben a példában az 11010100 titkos bitsorozatot továbbítja a szerver.

Adott *S* fájl zárolása és elengedése nem különösebben zajos csatorna, de nagyon pontos időzítést igényel, hacsak nem nagyon alacsony a bitarány. A megbízhatóság és a hatékonyság még tovább növelhető nyugtázó protokoll használatával. Ez a protokoll két további, *F1* és *F2* fájlt használ, amelyeket a szerver, illetve a közreműködő zárol, hogy szinkronizálja a két processzust. Miután a szerver zárolja az *S* fájlt, az *F1* fájl zároltságát ellentétesre változtatja, ezzel jelzi, hogy egy bitet küldött. Közvetlenül azután, hogy az együttműködő elolvasta a bitet, *F2* fájl zároltságát ellentétesre változtatja, ezzel jelzi a szervernek, hogy készen áll újabb bit fogadására, és várakozik addig, amíg az *F1* zároltsága ellentétesre változik, tehát újabb bitet olvashat. Mivel ebben a sémában időzítés nem szerepel, ezért a protokoll teljesen megbízható még leterhelt rendszerben is, és gyorsaságát a processzusok prioritása határozza meg. A sávszélesség növelésére használhatunk akár két fájlt, avagy bájtszéles csatornát nyolc jelzőfájllal, *S0*, ..., *S7*.

Dedikált eszköz (szalagegység, plotter stb.) lefoglalása és felszabadítása szintén használható üzenetküldésre. A szerver lefoglalja az eszközt 1-es bit küldése végett, és felszabadítja, ha 0-s bitet akar küldeni. Unixban a szerver létrehozhat egy fájlt 1-es bit jelzése végett, és törölheti 0-s bit jelzése érdekében; az együttműködő az access rendszerhívást használhatja a fájl létezésének kiderítésére. Ez a módszer

akkor is működik, ha az együttműködőnek nincs hozzáférési joga a fájlhoz. Sajnos sok más rejtett csatorna is lehetséges.

Lampson egy másik módszert is említ a szerverprocesszus tulajdonosa számára történő információ kisziárogtatására. Tegyük fel, hogy a szerverprocesszus fel van jogosítva arra, hogy számlázás végett megmondja a kliensnek, mennyi munkát végzett számára. Ha a számla tényleges értéke mondjuk 100 dollár, és a kliens bevétele, amelyet ki akar szivárogtatni, 53 ezer dollár, akkor a szerverszámla értékként 100,53 dollár adatot közölhet a tulajdonosnak.

Az összes rejtett csatorna kiderítése rendkívül nehéz feladat. A gyakorlatban nagyon keveset lehet tenni. Nem vonzó az olyan javaslat, amely véletlenszerű lapozási hibát előállító processzust működtetne, vagy más módon csökkentené a rendszer teljesítményét, hogy csökkentse a rejtett csatorna sávszélességét.

5.6. A MINIX 3 fájlrendszere

Mint bármely más fájlrendszernek, a MINIX 3 fájlrendszerének is el kell látnia mindazokat a feladatokat, amelyeket az előbbiekben tanulmányoztunk. Az állományok számára helyet kell lefoglalnia, illetve felszabadítania, nyomon kell követnie a lemez blokkjait és a szabad területeket, biztosítania kell néhány módozatot az állományok jogosulatlan felhasználás elleni védelmére, és így tovább. E fejezet további részében közelebbről is megvizsgáljuk a MINIX 3-at, hogy lássuk, hogyan is éri el ezeket a célokat.

Jelen fejezet első részében, az általánosság kedvéért, folyamatosan a Unixra utaltunk a MINIX 3 helyett, bár a kettő külső csatolófelülete gyakorlatilag megegyezik. Mostantól a MINIX 3 belső felépítésére fektetjük a hangsúlyt. A Unix belső felépítésével kapcsolatos információ a következő helyeken található meg: (Thompson, 1978; Bach, 1987; Lions, 1996; és Vahalia, 1996).

A MINIX 3 fájlrendszere egy hatalmas C program, amely a felhasználói területen fut (lásd 2.29. ábra). A felhasználói processzusok a fájlok írásakor és olvasásakor üzeneteket küldenek a fájlrendszer számára. Ezáltal közlik, hogy mit is akarnak elvégeztetni. A fájlrendszer elvégzi a feladatot, majd visszajelez. A fájlrendszer valójában egy hálózati állománykiszolgáló, amely történetesen ugyanazon a gépen fut, mint a hívó.

Ennek a konstrukciónak van néhány fontos következménye. Egyrészt a fájlrendszert a MINIX 3 egyéb részeitől majdnem teljesen függetlenül lehet módosítani, tesztelni és kísérletezni vele. Másrészt az egész állományrendszer egyszerűen telepíthető minden olyan számítógépre, amely rendelkezik C fordítóval, ott lefordítható, és ezután úgy használható, mint egy különálló Unix-szerű távoli állománykiszolgáló. Mindössze az üzenetek küldését és fogadását illetően kell változtatásokat eszközölnünk, mivel ez rendszerről rendszerre változik.

A következő részekben áttekintést adunk a fájlrendszer-felépítés számos kulcsterületéről. Megvizsgáljuk az üzeneteket, a fájlrendszer szerkezetét, a bittérképeket, az i-csomópontokat, a blokkgyorsítótárakat, a könyvtárakat és elérési utakat, a fájlleírókat, a fájlokhoz való hozzáférés letiltását, valamint a speciális fájlokat (és adatcsöveket). Ezek megismerése után egy egyszerű példán keresztül bemutatjuk, hogyan lehet a darabokat összeilleszteni; nyomon követjük, mi is történik valójában, amikor egy felhasználói processzus a read rendszerhívást hajtja végre.

5.6.1. Üzenetek

A fájlrendszer 39 végrehajtást kérő üzenettípust fogad el. Kettő kivételével ezek mindegyike MINIX 3-rendszerhívás. A két kivétel olyan üzenetet jelent, amelyeket a MINIX 3 más részei állítanak elő. A rendszerhívás-üzenetek közül 31 felhasználói processzus eredményeként keletkezik. Hat rendszerhívás-üzenet olyan rendszerhívásokra szolgál, amelyeket először a memóriakezelő vizsgál; ez ezután meghívja a fájlrendszert, hogy elvégeztesse azzal a munka egy részét. Két másik üzenetet szintén a fájlrendszer dolgoz fel. Az üzeneteket az 5.33. ábrán mutatjuk be.

A fájlrendszer szerkezete alapvetően megegyezik a memóriakezelő, illetve az összes beviteli-kiviteli (I/O) feladat szerkezetével. Tartalmaz egy főciklust, amely arra vár, hogy valamilyen üzenet érkezzen. Az üzenet megérkezése után megállapítja annak típusát, amelyet a későbbiekben abban a táblában való keresésre használ, amely az összes típust kezelő fájlrendszer eljárásainak mutatóit tartalmazza. Mindezek után meghívja a megfelelő eljárást, amely feladatának elvégzése után egy állapotértéket küld vissza. Ezt követően a fájlrendszer választ küld a hívónak, visszamegy a ciklus elejére, és várja a következő üzenetet.

Felhasználói üzenetek	Inputparaméterek	Válaszérték
access	Fájlnév, hozzáférés módja	Állapot
chdir	Új munkakönyvtár neve	Állapot
chmod	Fájlnév, új jogosultsági mód	Állapot
chown	Fájlnév, új tulajdonos, csoport	Állapot
chroot	Új gyökérkönyvtár neve	Állapot
close	Bezárandó fájl fájlleírója	Állapot
create	Létrehozandó fájl neve, védelmi kódja	Fájlleíró
dup	Fájlleíró (DUP2 esetén 2 fájlleíró)	Új fájlleíró
fcntl	Fájlleíró, funkciókód, egyéb operandusz	Kéréstől függ
fstat	Fájlnév, átmeneti adattár	Állapot
ioctl	Fájlleíró, funkciókód, egyéb operandusz	Állapot
link	Fájlnév, amelyhez csatolunk; fájlnév, amelyet csatolunk	Állapot
lseek	Fájlleíró, eltolás, honnan	Új fájlpozíció
mkdir	Fájlnév, jogosultsági mód	Állapot
mknod	Könyvtár vagy speciális fájl neve, jogosultsági mód, cím	Állapot

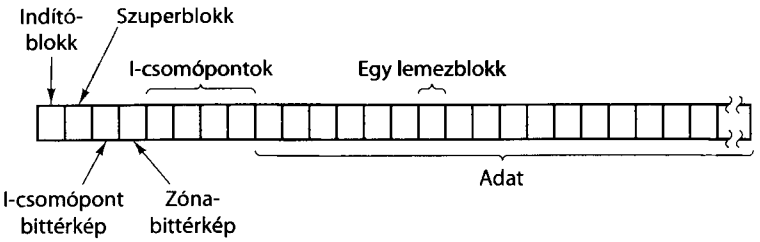
5.33. ábra. *A fájlrendszer üzenetei. A fájlnév paraméter mindig a nevet kijelölő mutatót jelenti. Az állapot jelentése mindig* OK *vagy* HIBÁS

Felhasználói üzenetek	Inputparaméterek	Válaszérték
mount	Speciális fájl, felcsatolási pont, csak olvasható állapot jelzője	Állapot
open	Megnyitandó fájl neve, írható/olvasható állapot jelzője	Fájlleíró
pipe	Két fájlleírót kijelölő mutató (módosított)	Állapot
read	Fájlleíró, átmeneti adattár, bájtok száma	Beolvasott bájtok sz.
rename	Fájlnév, fájlnév	Állapot
rmdir	Fájlnév	Állapot
stat	Fájlnév, állapot ideiglenes tára	Állapot
stime	Pontos idő mutatója	Állapot
sync	(semmi)	Mindig OK
time	Annak mutatója, ahová a pontos idő kerül	Állapot
times	Processzus és gyermekei idejének átmeneti tárának mutatója	Állapot
umask	Módsablon komplemense	Mindig OK
umount	Lecsatolandó speciális fájl neve	Állapot
unlink	Lekapcsolandó fájl neve	Állapot
utime	Fájlnév, állományidők	Mindig OK
write	Fájlleíró, átmeneti adattár, bájtok száma	Kiírt bájtok száma
Memóriakezelő üzenetek	**Inputparaméterek**	**Válaszérték**
exec	Processzusazonosító (pid)	Állapot
exit	Processzusazonosító (pid)	Állapot
fork	Elől processzusazonosítója, utód processzusazonosítója	Állapot
setgid	Processzusazonosító, valódi és effektív csoportazonosító (gid)	Állapot
setsid	Processzusazonosító	Állapot
setuid	Processzusazonosító, valódi és effektív felhasználóazonosító (uid)	Állapot
Egyéb üzenetek	**Inputparaméterek**	**Válaszérték**
revive	Újraélesztendő processzus	(Nincs válasz)
unpause	Ellenőrizendő processzus	(Lásd a szövegben)

5.33. ábra. *Folytatás*

5.6.2. A fájlrendszer felépítése

A MINIX 3 fájlrendszere önálló logikai egység i-csomópontokkal, könyvtárakkal és adatblokkokkal. Bármely blokkeszközön, például hajlékonylemezen vagy merevlemez partícióján tárolható. A fájlrendszer felépítése minden esetben ugyanaz. Az 5.34. ábra egy hajlékonylemez, vagy kisméretű merevlemez partíciójának szerkezetét mutatja 64 i-csomóponttal és 1 KB méretű blokkokkal, így legfeljebb



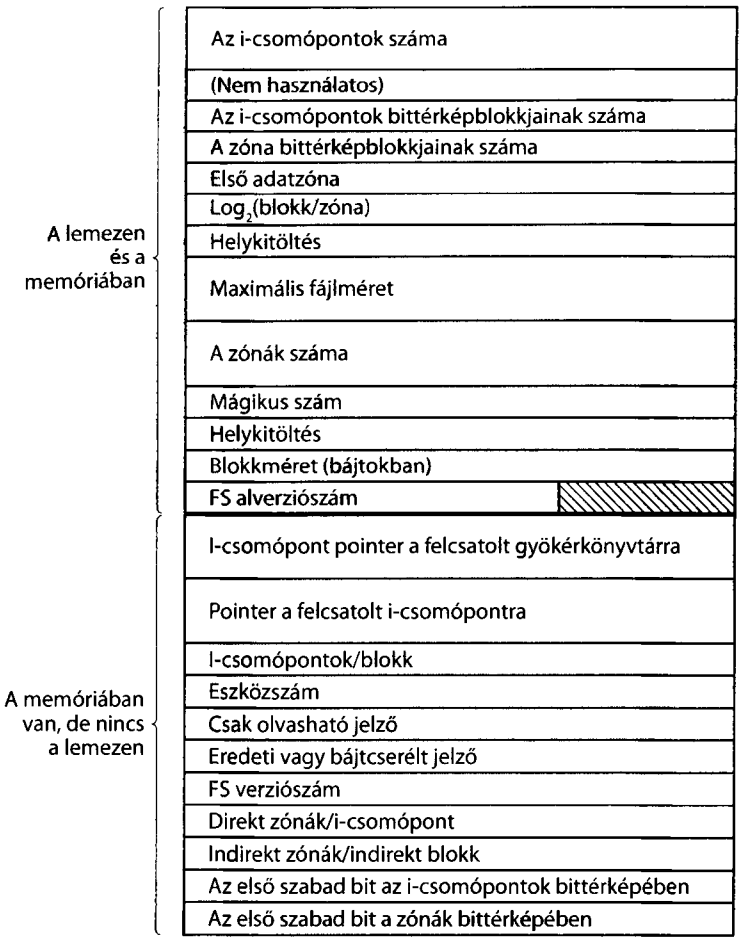
5.34. ábra. *Hajlékonylemez vagy kisméretű merevlemez partíciójának elrendezése 64 i-csomóponttal és 1 KB-os blokkmérettel (tehát egymást követő két 512 bájtos szektor alkot egy blokkot)*

8192 1 KB-os blokkot tartalmazhat, tehát a fájlrendszer legfeljebb 8 MB-os lehet. Még hajlékonylemez esetén is a 64 i-csomópont olyan kevés, hogy az ábrán mutatott négy helyett jóval több blokk kell. Praktikusabb nyolc blokkot lefoglalni az i-csomópontoknak, de ekkor az ábra nem lenne ilyen szép. A modern merevlemezeknél az i-csomópontok és a zónabittérkép mérete természetesen jóval nagyobb, mint egy blokk. Az 5.34. ábrán látható komponensek relatív mérete fájlrendszerenként eltérő lehet, és függ a méreteiktől, a fájlok megengedett maximális számától, és így tovább. De mindegyik komponens előfordul a megadott sorrendben.

Minden állományrendszer egy ún. **indítóblokk**al kezdődik, amely egy végrehajtható kódot tartalmaz. Mérete mindig 1024 bájt (két szektor), annak ellenére, hogy a MINIX 3 máshol nagyobb blokkméretet használ. A számítógép bekapcsolása után a hardver az indítóegységről a memóriába olvassa az indítóblokkot, odaugrik, és elkezdi végrehajtani annak programkódját. Az indítóblokk programkódja magának az operációs rendszernek a betöltését kezdi el. A rendszer elindulása után az indítóblokkra többé nincs szükség. Nem minden lemezmeghajtó használható indítóegységként, azonban az egységesség fenntartása érdekében minden blokkeszköz rendelkezik az indítóblokk programkódja számára fenntartott blokkal. Ez a stratégia a legrosszabb esetben egy blokk elpazarlásához vezet. Azt megelőzendő, hogy a hardver egy nem indítható egységet próbáljon meg elindítani, egy ún. **mágikus szám** kerül az indítóblokk meghatározott helyére. Ez akkor (és csakis akkor) történik meg, amikor a végrehajtó kódot az eszközre írjuk. Eszközről történő indítás során a hardver (valójában a BIOS programkód) egyáltalán nem kísérli meg a betöltést olyan egységről, amelyen nincs jelen a mágikus szám. Ez biztosítja, hogy értelmetlen adat még véletlenül se működhessen indítóprogramként.

A **szuperblokk** tartalmazza a fájlrendszer felépítésére vonatkozó információt. Az indítóblokkhoz hasonlóan, a szuperblokk is mindig 1024 bájtos, függetlenül attól a fájlrendszer blokkméretétől. Ezt szemlélteti az 5.35. ábra.

A szuperblokk fő feladata a fájlrendszer különböző részei méretének számontartása. A blokkméret és az i-csomópontok számának ismeretében könnyű kiszámítani az i-csomópont bittérképének méretét és az i-csomópontok blokkjainak számát. 1 KB méretű blokkot véve például alapul, a bittérkép minden blokkja 1024 bájtot (8192 bitet) tartalmaz, így legfeljebb 8192 i-csomópont állapotát ké-



5.35. ábra. *A MINIX 3 szuperblokkja*

pes nyilvántartani. (Valójában az első blokk csak 8191 i-csomópontot képes kezelni, mivel nincs nulladik i-csomópont, annak ellenére, hogy ez is kap egy bitet a bittérképen.) 10 000 i-csomóponthoz a bittérkép 2 blokkjára van szükség. Mivel az i-csomópontok egyenként 64 bájtot foglalnak el, egy 1 KB méretű blokk legfeljebb 16 i-csomópontot tartalmazhat. Ahhoz, hogy 64 felhasználható i-csomópont mindegyikét számon tarthassuk, 4 darab 1 KB-os lemezblokk szükséges.

A zónák és a blokkok közötti különbséget a továbbiakban még részletesen tárgyaljuk. Jelen pillanatban elegendő ezekről annyit elmondanunk, hogy a lemeztárolás számára 1, 2, 4, 8 vagy általánosan 2^n blokkos egységek (zónák) foglalhatók le. A zónabittérkép a szabad tárolókapacitást blokkok helyett zónákban tartja nyilván. A zónák és a blokkok mérete a MINIX 3 által használatos szabványos hajlékonylemezek mindegyikénél ugyanaz (1 KB), vagyis ezeken az eszközökön a

zóna és a blokk első közelítésben megegyezik. Amíg ezen fejezet egy későbbi részében vissza nem térünk a tárfoglalás részleteire, a „zóna" „blokk"-kal való helyettesítése nyugodtan megtehető.

Megemlítjük, hogy a blokkok zónánkénti száma nincs a szuperblokkban tárolva, mivel erre az információra soha nincsen szükségünk. Mindössze a blokk/zóna hányados 2-es alapú logaritmusát kell ismernünk, amelyet a zóna → blokk, illetve blokk → zóna áttéréseknél a biteltolás mérőszámaként használunk. Ha például egy zóna 8 blokkból áll, akkor – mivel $\log_2 8 = 3$ – a 128. blokkot tartalmazó 16. zóna megtalálásához a 128-at kell 3 bittel jobbra eltolnunk.

A zónabittérkép csupán az adatzónákat foglalja magában (tehát a bittérképekhez és i-csomópontokhoz felhasznált blokkok nincsenek benne). Az első adatzóna az 1-es számú zónához van hozzárendelve. Hasonlóan az i-csomópont bittérképhez a zónabittérkép 0. bitje sem használatos, ami azt eredményezi, hogy a zónabittérkép első blokkja csupán 8191 zónát tud tárolni, míg a rá következő blokkok mindegyike 8192-t. Ha megvizsgáljuk a bittérképeket egy frissen formázott lemezen, azt vesszük észre, hogy mind az i-csomópont bittérképen, mind a zónabittérképen két bit is 1-re van állítva. Az egyik a nem létező 0. i-csomópontra, illetve zónára, a másik a fájlrendszer lemezen történő létrehozásakor megjelenő gyökérkönyvtárra utal.

A szuperblokkban tárolt információ redundáns, mert a rendszernek egyszer ilyen, másszor olyan formában van rá szüksége. Mivel a szuperblokk számára 1 KB hely van fenntartva, érdemes ezen információt az összes olyan formában kiszámolni és elraktározni, amire később szükség lehet; ez ugyanis sokkal kifizetődőbb, mint a végrehajtás folyamán újra és újra kiszámolni azt. A lemez első adatzónájának zónaazonosító száma kiszámolható lenne például a blokk vagy a zóna méretéből, illetve az i-csomópontok vagy zónák számából is, de sokkal gyorsabb, ha ezt a számot a szuperblokkban tároljuk. A szuperblokk fennmaradó részét úgyis elpazaroljuk, tehát egy további szavának felhasználása semmibe nem kerül.

A MINIX 3 betöltésekor a gyökéreszköz szuperblokkja beíródik a memória egy táblájába. Ehhez hasonlóan további fájlrendszerek szuperblokkjai is a memóriába kerülnek azok felcsatolásakor. A szuperblokktábla lemezen jelen nem lévő mezőkből áll. Ezekben találhatók azok a jelzők, amelyek lehetővé teszik, hogy egy adott eszköz pusztán olvasható legyen vagy a szabványtól eltérő bájtsorrend-megállapodást kövessen, illetve azok a mezők, amelyek oly módon gyorsítják a hozzáférést, hogy kijelölik a bittérképek azon pontjait, amelyektől lefelé az összes bit használtként van feltüntetve. A szuperblokktábla tartalmaz továbbá egy olyan mezőt is, amely annak az eszköznek az azonosítóját tárolja, ahonnan szuperblokk érkezett.

Mielőtt egy lemez MINIX 3-fájlrendszerként ténylegesen használható lenne, létre kell rajta hozni az 5.34. ábrának megfelelő felépítést. A fájlrendszer létrehozására az *mkfs* program szolgál. Ezt a programot vagy egy paranccsal lehet működésbe hozni, például az

```
mkfs /dev/fd1 1440
```

parancs az 1-es meghajtóban lévő hajlékonylemezen létrehoz egy 1440 blokkból álló üres fájlrendszert, vagy megadható egy olyan, előre elkészített prototípusfájl, amely azokat a könyvtárakat és fájlokat sorolja fel, amelyeknek az új fájlrendszerben feltétlenül jelen kell lenniük. Az említett parancs beír továbbá a szuperblokkba egy mágikus számot is, amellyel az új fájlrendszert érvényes MINIX 3fájlrendszerként tünteti fel. A MINIX 3-fájlrendszere folyamatosan fejlődött, néhány jellemzője (például az i-csomópontok mérete) a korábbi változatokban a jelenlegitől eltérő volt. A mágikus szám azonosítja a fájlrendszert létrehozó *mkfs* program verzióját, így a különböző változatok közötti eltérések áthidalhatók. Nem MINIX 3 formátumú fájlrendszer, mint például egy MS-DOS-lemez, felcsatolását a mount rendszerhívás, amely sok más dolog mellett a mágikus számot is ellenőrzi a szuperblokkban, megtagadja.

5.6.3. A bittérképek

A MINIX 3 két bittérkép használatával tartja számon, hogy mely i-csomópontok és zónák szabadok. Egy állomány eltávolításakor rendkívül egyszerű meghatározni, hogy a bittérkép melyik blokkja tartalmazza a felszabadítandó i-csomópont bitjét, majd a szokásos, gyorsítótár használatán alapuló eljárást követve azt megtalálni. A megfelelő blokk kikeresése után a felszabadítandó i-csomópont bitjét 0-ra állítjuk. A zónákat hasonló módszerrel távolítjuk el a zónabittérképből.

Az állomány létrehozása előtt a fájlrendszernek az első szabad i-csomópont megkeresése végett a bittérképek blokkjait egyesével végig kell néznie. Ennek megtörténte után a fájlrendszer ezt az i-csomópontot az új állomány számára lefoglalja. Valójában a szuperblokk memóriában lévő másolata egy olyan mezőt is tartalmaz, amely az első szabad i-csomópontra mutat, így amíg ezt az i-csomópontot fel nem használtuk, nincs szükség keresésre. Felhasználása után azonban e mező értékét fel kell frissítenünk, hogy az a következő új, üres i-csomópontra (amelyről gyakran kiderül, hogy a rá következő vagy egy, az eredetihez közel lévő) mutasson. Ehhez hasonlóan, egy i-csomópont felszabadításakor ellenőrizzük, vajon a szabad i-csomópont nem azelőtt az i-csomópont előtt áll-e, amelyet a mutató jelen pillanatban tartalmaz, és szükség esetén felfrissítjük a mutató értékét. Ha a lemezen minden i-csomópont foglalt, a kereső eljárás 0 értékkel tér vissza. Ez az oka annak, hogy a 0. i-csomópont nem használatos (így ugyanis jelezni tudja, ha a keresés eredménytelen volt). (Amikor az *mkfs* parancs létrehoz egy új fájlrendszert, kinullázza a 0. i-csomópontot, és a bittérkép legelső bitjét 1-re állítja, amivel eléri, hogy a fájlrendszer soha nem fogja megpróbálni ennek az i-csomópontnak a lefoglalását.) Mindaz, amit az előbbiekben az i-csomópont bittérképről elmondtunk, a zónabittérképre is érvényes; ha szabad helyre van szükségünk, logikailag az első üres zónát keressük. A bittérképen való folyamatos keresést kiküszöbölendő megtartunk azonban egy olyan mutatót is, amely az első szabad zóna helyét tartalmazza.

Ezen ismeretek birtokában most már el tudjuk magyarázni a zónák és blokkok közötti különbséget is. A zónák mögött az az elv húzódik meg, hogy az ugyanazon

állományhoz tartozó lemezblokkok ugyanazon a cilinderen helyezkedjenek el azért, hogy az állomány folyamatos olvasásakor a rendszer teljesítménye javulhasson. Ez több blokk egyidejű lefoglalásának lehetővé tételével érhető el. Ha például a blokkméret 1 KB és a zónaméret 4 KB – a zónabittérkép a zónákat tartja nyilván, és nem a blokkokat – egy 20 MB-os lemez 5 K darab 4 KB méretű zónából áll, amelyhez a hozzá tartozó zónabittérképben 5 K bit társul.

A fájlrendszerek többsége blokkokkal dolgozik. A lemezátvitelek mindig blokkokkal történnek, és az ideiglenes gyorsítótár is az egyedi blokkokkal dolgozik. A rendszernek csak néhány része, amelyek a tényleges lemezcímeket tartják számon (például zónabittérkép és i-csomópontok), ismeri a zónákat.

A MINIX 3-fájlrendszer fejlesztése során néhány, tervezéssel kapcsolatos, döntést is meg kellett hozni. Amikor 1985-ben a MINIX-et kigondolták, a lemezkapacitások még kicsik voltak, és úgy gondolták, hogy sok felhasználónak csak hajlékonylemezei lesznek. Ezért döntöttek úgy, hogy a V1 fájlrendszerben a lemezcímeket 16 bitesre korlátozzák – elsődlegesen azért, hogy nagy részüket közvetett blokkokban tárolhassák. 16 bites zónaazonosítóval és 1 KB méretű zónákkal legfeljebb 64 KB zóna címezhető meg, ami a lemez tárolókapacitását 64 MB-ban korlátozza. Abban az időben ez óriási tárolási kapacitást jelentett, és úgy tervezték, hogy a lemezek növekedésével, a blokkok méretének megváltoztatása nélkül, egyszerűen 2 KB vagy 4 KB méretű zónákra térnek majd át. A 16 bit hosszúságú zónaazonosítók egyszerűen tették lehetővé az i-csomópontok méretének 32 bájton való tartását is.

Mivel a MINIX folyamatosan fejlődött, és egyre általánosabbá váltak a nagyobb lemezek, nyilvánvalóan változtatásra volt szükség. Mivel sok állomány mérete 1 KB-nál kisebb, a blokkméret növelése a lemez sávszélességének pazarlásához, majdnem üres blokkok olvasásához és írásához, illetve ezeknek az ideiglenes gyorsítótárban való tárolásán keresztül az értékes központi memória pazarlásához vezetne. Megnövelhették volna a zónaméretet, ami nagyobb lemezterület elpazarlását eredményezi. A kisebb lemezek későbbi hatékony kezelhetősége szintén kívánatos volt. Egy másik ésszerű megoldás az lett volna, ha a nagy és kis eszközökön eltérőnek választják a zónaméretet.

Végül is a lemezmutatók méretének 32 bitesre való növelése mellett döntöttek. Ez a MINIX V2 fájlrendszer számára lehetővé teszi az 1 KB méretű blokkokból és zónákból felépített, akár 4 terabájtos, 4 KB méretű blokkok és zónák (ami ma az alapértelmezett érték) esetén 16 TB kapacitású eszközök kezelését is. Azonban más szempontok korlátozhatják a méretet (például 32 bites pointerek esetén 4 GB a maximális kapacitás). A lemezpointerek méretének növelésével növelődik az i-csomópont mérete. Ez nem szükségképpen rossz, mert azt eredményezi, hogy a MINIX V2 (és ma már a V3) i-csomópontja kompatibilis lesz a standard Unix i-csomópontjával, helyet adva három értéknek, így több indirekt és kétszeresen indirekt zóna lehet, valamint hely a későbbi kiterjesztésre háromszoros indirekt zónákra. A zónák azonban egy további váratlan problémát is eredményeztek. Ezt legjobban egy egyszerű példán keresztül – ismét csak 4 KB nagyságú zónákat és 1 KB méretű blokkokat alapul véve – mutathatjuk be. Tételezzük fel, hogy állományunk 1 KB hosszúságú, ami azt jelenti, hogy egy zóna van számára

lefoglalva. Az 1 KB és 4 KB közötti blokkok (a korábbi használat eredményeképpen) értelmetlen adatokat tartalmaznak, de semmilyen hibát nem okoznak, mert az állomány hossza az i-csomópontban egyértelműen 1 KB-nak van definiálva. Az értelmetlen adatot tartalmazó blokkok valójában nem kerülnek beolvasásra a blokkgyorsítótárba, mivel az olvasás blokkonként és nem zónánként történik. Az állomány vége utáni olvasások mindig egy 0 értékű számlálóval és zéró adattal térnek vissza.

Valaki ezután megkeresi a 32 768-as helyet, és beír oda egy bájtot. Az állomány mérete ezzel 32 769-re változik. Az 1 KB-os adatolvasási kísérletekkel kiegészített keresések a későbbiekben lehetővé teszik a blokk korábbi tartalmának olvasását, ami alapvető biztonsági hiányosság.

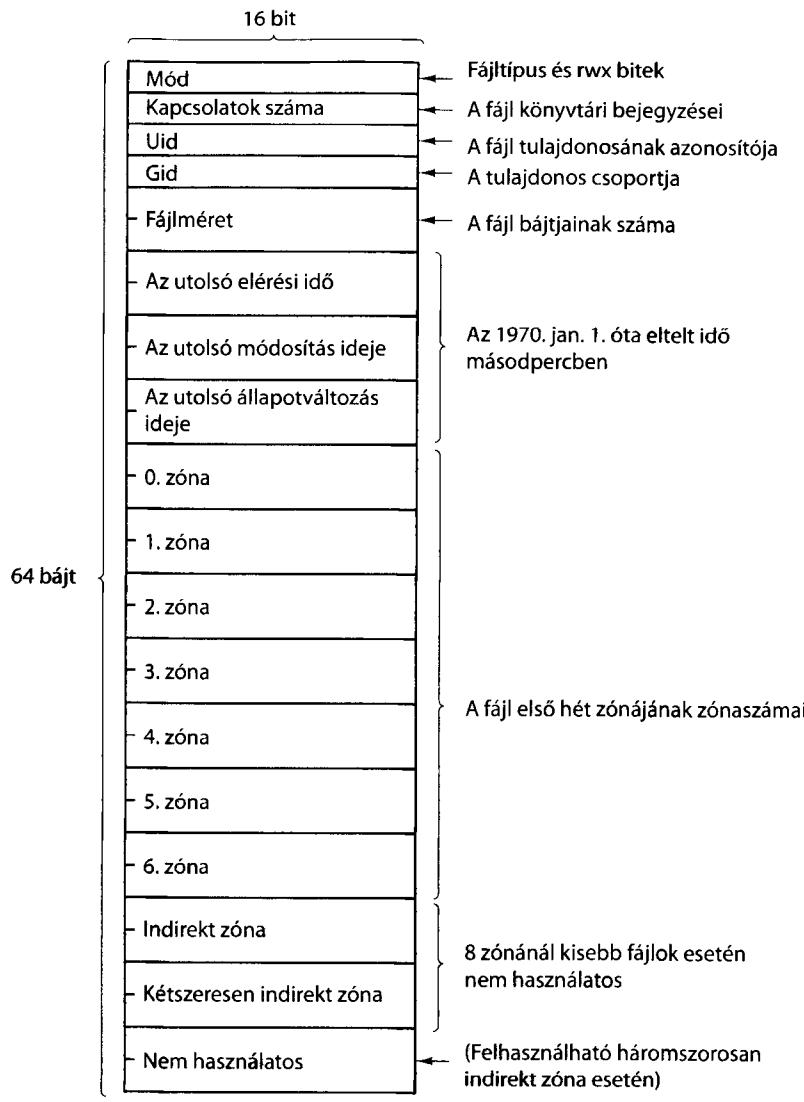
A probléma megoldása az, hogy az állomány vége utáni írás esetén ellenőrizzük, nem következik-e be az előbb bemutatott helyzet, illetve ténylegesen kinullázzuk a korábban utolsóként nyilvántartott zóna összes, eddig még le nem foglalt blokkját. Annak ellenére, hogy az itt vázolt helyzet csak ritkán fordul elő, azt a programkódnak tudnia kell kezelni, ami a rendszert egy kicsit bonyolultabbá teszi.

5.6.4. Az i-csomópontok

A MINIX 3 i-csomópontjának felépítését az 5.36. ábra mutatja. Ez majdnem teljesen megegyezik a Unix i-csomópontjának felépítésével. A lemez zónamutatói 32 bites mutatók, és mindössze 9 darab van belőlük: 7 közvetlen és 2 közvetett. A MINIX 3 i-csomópontok 64 bájtot foglalnak el, akárcsak a szabvány Unix i-csomópontok és egy 10. mutatónak (ez a triplán közvetett mutató) is van hely, bár a fájlrendszer szabványváltozata ennek használatát nem támogatja. A MINIX 3 i-csomópontjának elérése, módosítási ideje és az i-csomópont tartalmának utolsó változtatásához tartozó idők szabványosak, akárcsak a Unixban. Ezek közül az utolsó szinte minden állományművelet során, az állomány olvasását kivéve, felfrissítésre kerül.

Egy állomány megnyitásakor az állományhoz tartozó i-csomópont kikeresése után a memóriában lévő *inode* i-csomópont táblába töltődik és az állomány lezárásáig ott marad. Az *inode* i-csomópont táblának van néhány, lemezen meg nem található további mezője is, úgymint az i-csomópont eszköze és azonosítója, amelyek ismeretében a fájlrendszer pontosan tudja, hová kell írnia, ha az i-csomópont tartalmában változás következik be az alatt az idő alatt, míg a memóriában van. Az *inode* i-csomópont táblában minden i-csomóponthoz tartozik egy számláló is. Ha ugyanazt az állományt többször nyitjuk meg, i-csomópontjának csak egyetlen másolata tárolódik a memóriában, de a számláló értéke minden megnyitásnál eggyel nő és minden bezárásnál eggyel csökken. Az i-csomópont csak akkor kerül ki a táblából, amikor a számláló értéke nullára csökken. Ha a memóriában töltött idő alatt az i-csomópont tartalma megváltozott, kikerüléskor a lemezen újraíródik.

Az állomány i-csomópontjának fő feladata az, hogy megadja az állomány adatblokkjainak helyét. Az első 7 zóna helye magában az i-csomópontban van. Ez azt jelenti, hogy a szabványos, tehát 1 KB méretű zónákkal, illetve blokkokkal rendelkező kiosztás esetén a 7 KB-ig terjedő állományok nem igényelnek közvetett blokkokat. 7 KB-os méret felett azonban, az 5.10. ábrán vázoltnak megfelelően – annyi különbséggel, hogy csak a szimplán és a duplán közvetett blokk használatos – közvetett zónákra is szükség van. 1 KB blokk-, illetve zónaméret és 32 bites zónaazonosító mellett a szimplán közvetett blokknak 256 eleme van, ami negyed megabájt tárolókapacitást jelent. A duplán közvetett blokk 256 szimplán közvetett blokkot tud kijelölni. Ezáltal legfeljebb 64 MB-nyi terület válik elérhetővé. 4 KB-os blok-



5.36. ábra. *A MINIX 3 i-csomópontja*

kokkal a kétszeresen indirekt címzés 1024 × 1024 érhető el, ami több mint egymillió 4 KB-os blokkot jelent, így a maximális fájlméret 4 GB lehet. A gyakorlatban a 32 bites fájlpozíció miatt a maximális fájlméret $2^{32} - 1$ bájt. Ezekből a számokból az következik, hogy ha 4 KB-os lemezblokkot alkalmaz a MINIX 3, akkor nincs szükség háromszorosan indirekt blokkokra; a maximális fájlméretet a pointer mérete korlátozza le, nem pedig a blokkok nyilvántartásának képessége.

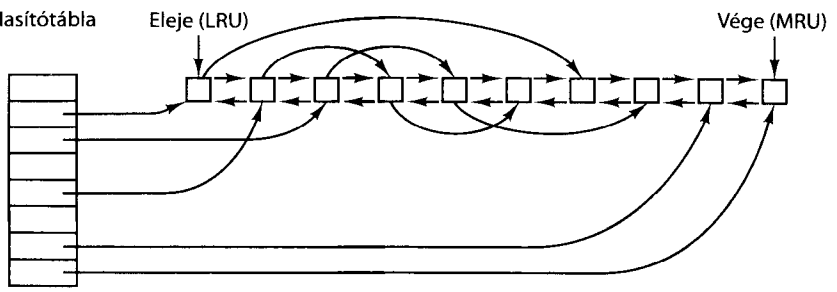
Az i-csomópont tartalmazza még az üzemmódra vonatkozó információt, amely az állomány típusát (egyszerű állomány, könyvtár, speciális blokk- vagy karakterállomány, illetve adatcső) adja meg, ezenkívül közli az állomány védettségét és a SETUID, illetve a SETGID bitek értékét. Az i-csomópont *link* mezője tartja számon, hány könyvtárelem mutat az illető i-csomópontra. Ennek ismeretében a fájlrendszer pontosan tudja, mikor szabadítsa fel az állomány helyét. Ezt a mezőt azonban nem szabad összekevernünk azzal a számlálóval (amely pusztán a memóriában lévő *inode* i-csomópont táblán van jelen, de a lemezen nem), amely azt mutatja, hogy egy adott állomány jelen pillanatban hányszor áll megnyitás alatt, például különböző processzusok által.

Utolsó megjegyzésként megemlítjük, hogy az 5.36. ábrán látható szerkezet speciális célok elérésére módosulhat. Példa erre a MINIX 3 esetén az olyan i-csomópont, amely karakterspeciális eszköz i-csomópontja. Ezek nem igényelnek zónapointert, mert nem hivatkoznak lemezen tárolt adatra. A fő- és mellékeszközszámot az 5.36. ábrán látható 0-adik zóna tartalmazza. Egy másik példa lehet, bár a MINIX 3 nem alkalmazza, hogy kisméretű közvetlen fájl adatát tároljuk magában az i-csomópontban.

5.6.5. A blokkgyorsítótár

A MINIX 3, fájlrendszere jobb működése érdekében, blokkgyorsítótárat használ. A gyorsítótár átmeneti adattárak soraként kerül megvalósításra, ahol az adattárak mindegyike mutatókat, számlálókat és jelzőket tartalmazó fejlécből, illetve egy lemezblokknyi helyet tartalmazó törzsből áll. A pillanatnyilag használaton kívüli, átmeneti adattárak egy kétszeresen láncolt listában vannak a legutóbb használatostól (MRU) a legrégebben használatosig (LRU) egymás után fűzve, ahogy azt az 5.37. ábra mutatja.

Annak gyors eldöntéséhez, vajon egy blokk a gyorsítótárban van-e, vagy sem, az ún. **hasítótábla** használatos. Mindazok az átmeneti adattárak, amelyek olyan blokkot tartalmaznak, amelynek hasítókódja k, a hasítótábla k-adik eleme által kijelölt, egyszeresen láncolt listában vannak egymás után felfűzve. A hasítófüggvény mindössze a blokkszám n alacsony helyi értékű bitjét olvassa ki, így a különböző eszközökről származó blokkok azonos hasítóláncban jelennek meg. Bármely átmeneti adattár ezen láncok valamelyikéhez tartozik. A MINIX 3 betöltése után a fájlrendszer felépülésekor természetesen az összes átmeneti adattár üres, tehát a hasítótábla 0. eleme által kijelölt egyszerű lánchoz tartozik. Ebben a pillanatban a hasítótábla összes többi eleme egy null mutatót tartalmaz. A rendszer elindulása után azonban a 0. láncból átmeneti adattárak kerülnek eltávolításra, és újabb láncok épülnek fel.



5.37. ábra. *A blokkgyorsítótár által használt láncolt listák*

Amikor a fájlrendszernek szüksége van egy blokkra, meghívja a *get_block* nevű eljárást, amely kiszámítja az illető blokk tördelési kódját, és megkeresi a megfelelő láncot. A *get_block* eljárást az eszközszámmal és a blokkazonosítóval hívjuk. A keresés során mindkét azonosítót összehasonlítjuk az átmeneti adattárakból felépített lánc megfelelő mezőivel. Ha megtaláltuk a blokkot tároló átmeneti adattárat, akkor a fejlécében lévő számláló értéke eggyel nő – jelezvén a blokk használatát –, és visszakapjuk az őt kijelölő mutató értékét. Ha a blokkot nem találjuk meg a tördelési listában, akkor felhasználhatjuk az LRU-lista legelső átmeneti adattárát; ez még biztosan használaton kívül van, és az általa tartalmazott blokk az átmeneti adattár felszabadítása céljából kitörölhető.

Miután egy blokkot törlésre jelöltünk ki, átmeneti adattára fejlécében egy másik jelző is ellenőrzésre kerül, hogy megtudjuk, változott-e a blokk tartalma annak beolvasása óta. Ha igen, a változást lemezre írjuk. A szükséges blokkot ezen a ponton a lemezmeghajtónak küldött üzenettel olvastatjuk be. A fájlrendszer működése a blokk megérkezéséig szünetel, majd ezután folytatódik, és a hívó megkapja az illető blokk mutatóját.

Amikor azon eljárás, amelynek a blokkra szüksége volt, befejezte tevékenységét, a blokk felszabadítása céljából meghívja a *put_block* nevű eljárást. Egy blokkot rendszerint azonnal felhasználunk, majd kitörlünk. Mivel azonban az illető blokk törlés előtti ismételt kérésére is van lehetőség, a *put_block* eljárás eggyel csökkenti a használati számláló értékét, és csak akkor teszi vissza az átmeneti adattárat az LRU-listába, ha a számláló értéke nullára változott. Amíg a számláló értéke nullától különbözik, a blokk a tárban marad.

A *put_block* paramétereinek egyike azt határozza meg, hogy milyen blokktípust (például i-csomópontok, könyvtár, adat) kell felszabadítani. A típustól függően két kulcsfontosságú döntést kell meghozni:

1. Vajon a blokkot az LRU-lista elejére vagy végére kell-e tenni?
2. Vajon azonnal lemezre kell-e a blokkot írni (ha tartalma megváltozott), vagy sem?

Az LRU elvnek megfelelően majdnem minden blokk a lista végére kerül. Kivételt képeznek a RAM-lemez blokkjai; mivel már a memóriában vannak, ezért nem érdemes ezeket gyorsítótárba rakni.

Egy megváltozott blokk csak a következő két feltétel valamelyikének teljesülése esetén íródik lemezre:

1. Eléri az LRU-lánc elejét, és onnan el kell azt távolítani.
2. Egy sync rendszerhívás kerül végrehajtásra.

A sync rendszerhívás az LRU-láncot nem vizsgálja végig, csak indexével hivatkozik a gyorsítótárban lévő átmeneti adattárak tömbjében. Ha egy átmeneti adattár még nem törlődött, de tartalma megváltozott, a sync rendszerhívás megtalálja azt, és gondoskodik arról, hogy a lemezen lévő másolat felfrissítődjék.

Az ilyen eljárásmód gondolkodásra késztet. A MINIX egy korábbi változatánál a szuperblokk minden új fájlrendszer felcsatolásakor megváltozott, és azonnal a lemezre íródott, hogy egy esetleges rendszerösszeomlás esetén csökkenjen a fájlrendszer sérülésének veszélye. A szuperblokk csak akkor módosul, ha a RAM-lemez méretét változtatni kell az induláskor, mert a RAM-lemez mérete nagyobb, mint a RAM-eszköz. Azonban a szuperblokkot nem a szokásos módon írják és olvassák, mert a mérete mindig 1024 bájt, mint az indítóblokké, függetlenül a gyorsítótárral kezelt blokkok méretétől. Egy másik elhagyandó tapasztalat, hogy a régebbi MINIX-rendszerekben az */include/minix/conf.h* konfigurációs fájlban definiálható volt egy ROBUST makró, amely ha definiálták, akkor azt okozta, hogy az i-csomópontok, könyvtárak, indirekt és bittérképblokkok azonnal kiíródtak a lemezre. Ezzel a fájlrendszer megbízhatóbb volt, de az ára a lassúbb működés volt. Kiderült, hogy a módszer nem hatékony. Ha épp akkor következik be áramszünet, amikor a blokkok még nem voltak lemezre írva, komoly gondot okozhat annak eldöntése, hogy az elveszett információ i-csomópont vagy adatblokk volt-e.

Itt jegyezzük meg, hogy egy blokk fejlécének azt a jelzőjét, amely a blokk tartalmának megváltozását mutatja, a fájlrendszer azon eljárása állítja be, amely az illető blokkot meghívta és használta. A *get_block* és *put_block* eljárások pusztán a láncolt listákon végeznek műveleteket, és nincs tudomásuk arról, hogy a fájlrendszer melyik eljárásának melyik blokkra van szüksége, és mi célból.

5.6.6. Könyvtárak és elérési utak

A fájlrendszer egy következő fontos alrendszere a könyvtárak és elérési utak kezelése. Sok rendszerhívásnak, mint például az open eljárásnak is, az egyik paramétere fájlnév. Amire igazából szükség van, az az illető fájl i-csomópontja. A fájlrendszer feladata tehát az, hogy a könyvtárfában kikeresse a fájlt, és megtalálja az ehhez tartozó i-csomópontot.

A MINIX korábbi verzióiban a könyvtár egy olyan fájl, amely 16 bájtot tartalmaz, 2 bájtot az i-csomópont és 16 bájtot a fájlnév számára. Ez azt eredményezte, hogy a partíció legfeljebb 64 KB számú fájlt tartalmazhat, és a fájlnevek legfeljebb

14 karakteresek lehetnek, csakúgy, mint a Unix V7 esetén. A lemezek kapacitásával együtt a fájlnevek hossza is nőtt. A MINIX 3 V3 fájlrendszere 64 bájtos könyvtári bejegyzéseket alkalmaz, 4 bájt az i-csomópont sorszámának, 60 bájt pedig a fájlnév számára. Több mint 4 milliárd fájl lehet egy partícióban, ami gyakorlatilag végtelen lehetőség, és azt a programozót, aki 60-nál hosszabb fájlnevet akar írni, vissza kell küldeni az iskolába.

Megjegyzendő, hogy az útvonalhossz nem korlátozott 60 karakterre, például a

/usr/ast/course_material_for_this_year/operating_systems/examination-1.ps

érvényes, mert minden tagja 60-nál kevesebb karakterből áll. A fix, jelen esetben 64 bájt hosszú fájlnév példa a kompromisszumra, az egyszerűséget, gyorsaságot és a tárigényt tekintve. Más operációs rendszerek szokásosan halomba szervezve tárolják a könyvtárakat, a halomban lévő névre mutató fix méretű fejléccel a könyvtár végén. A MINIX 3 módszere nagyon egyszerű, és gyakorlatilag nem kell módosítani a V2 kódját. Nagyon gyors is, mind a névkeresést, mind az új beírást tekintve, mert nem kell a halom kezelésével foglalkoznia. Ennek lemezterület-vesztés az ára, mert a legtöbb fájlnév rövidebb 60 karakternél.

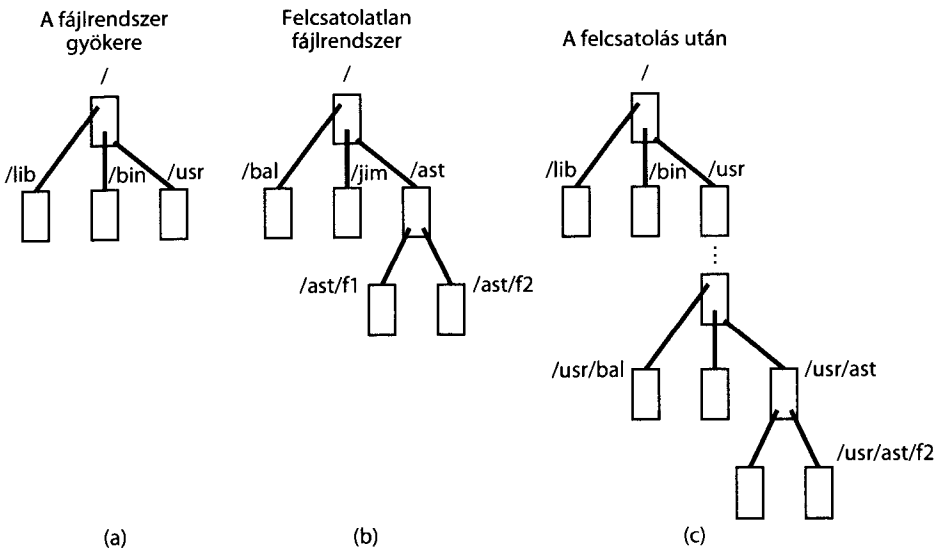
Szilárd meggyőződésünk, hogy a lemezterület-megtakarításra optimalizálni (és némi RAM-megtakarításra, mert a könyvtárak időnként a memóriában vannak) rossz választás. A kód egyszerűsége és helyessége az elsődleges, a gyorsasága másodlagos. Modern lemezek esetén, amelyek kapacitása meghaladja a 100 GB-ot, némi lemezterület megtakarítása bonyolultabb és lassabb kód árán általában nem jó ötlet. Sajnos, sok programozó olyan környezetben nő fel, ahol kisméretű a lemez és még kisebb a RAM, és kezdettől fogva olyan oktatásban részesül, hogy feloldja a kódbonyolultság, a gyorsaság és a tárigény közötti ellentmondást, a tárigény-minimalizálást előnyben részesítve. Ez az implicit feltételezés valóban felülvizsgálatra szorul a jelenlegi tények fényében.

Például a */usr/ast/mbox* elérési út kikeresésekor a rendszer először a gyökérkönyvtárban a *usr*-t keresi meg, majd a */usr/*-ben az *ast*-t, és végül a */usr/ast/*-ben az *mbox*-ot. A tényleges kikeresés során az elérési útnak egyszerre csak egy összetevőjét keressük, amint azt az 5.16. ábra mutatja.

Az egyetlen bonyodalom: mi történik, ha véletlenül egy felcsatolt fájlrendszerbe ütközünk? A MINIX 3 és sok más Unix-alapú rendszer szokásos beállítása olyan, hogy létezik a rendszer indításához és alapvető fenntartásához szükséges állományokat tartalmazó kicsiny gyökérfájlrendszer, míg az állományok többsége – beleértve a felhasználói könyvtárakat is – a */usr/*-hez felcsatolt külön eszközön van. Hogyan is történik valójában egy ilyen felcsatolás? Amikor a felhasználó begépeli a

```
mount /dev/c0d1p2 /usr
```

utasítást, az 1. merevlemez 2. partícióján található fájlrendszer a gyökérfájlrendszer */usr/* könyvtárához kapcsolódik hozzá. A fájlrendszereket összekapcsolásuk előtt és után az 5.38. ábra mutatja.



5.38. ábra. *(a) A gyökérfájlrendszer. (b) Fájlrendszer felcsatolás előtt. (c) A (b) fájlrendszer* /usr/*-hez történő felcsatolása utáni állapot*

A felcsatolási művelet kulcsa a */usr/*-hez tartozó i-csomópont memóriában lévő másolatában a sikeres felcsatolás esetén egy jelzőbit 1-re állítása. Ez a jelző mutatja, hogy az i-csomóponthoz ezután egy felcsatolt fájlrendszer tartozik. A mount hívás az újonnan felcsatolt fájlrendszer szuperblokkját a *super_block* nevű szuperblokktáblába másolja, illetve beállít abban két mutatót. Mindezeken túl, a felcsatolt fájlrendszer gyökér i-csomópontját az *inode* táblába tölti.

Az 5.35. ábráról leolvashatjuk, hogy a memóriában lévő szuperblokkok két olyan mezőt is tartalmaznak, amelyek az összekapcsolt fájlrendszerekkel kapcsolatosak. Ezek közül az első, a *felcsatolt fájlrendszer i-csomópontja,* az újonnan felcsatolt fájlrendszer gyökér i-csomópontjára mutat. A második, a *felcsatolási i-csomópont* úgy kerül beállításra, hogy arra a helyre mutasson, ahová a fájlrendszert felcsatoltuk, jelen esetben a */usr/* könyvtár i-csomópontjára. Ez a két mutató szolgál a bekapcsolt fájlrendszer gyökérhez kapcsolására, illetve tölti be a bekapcsolt fájlrendszer és a gyökér közötti „ragasztó" szerepét [amint azt az 5.38.(c). ábra pontozott vonala jelöli]. Ez a kapcsolat teszi lehetővé egy felcsatolt fájlrendszer működését.

Amikor például a */usr/ast/f2* elérési utat keressük, a fájlrendszer a */usr/*-hez tartozó i-csomópontban észrevesz egy jelzőt, amelynek állapotából tudja, hogy a */usr/*-hez felcsatolt fájlrendszer gyökerének i-csomópontjában kell tovább keresnie. A kérdés: hogyan fogja ezt a gyökérhez tartozó i-csomópontot megtalálni?

A válasz triviális. A rendszer mindaddig vizsgálja a memóriában lévő szuperblokkokat, amíg meg nem találja azt, amelynek felcsatolási i-csomópont mezője */usr/*-re mutat. Ez éppen a */usr/*-hez felcsatolt fájlrendszer szuperblokkja lesz. A szuperblokk ismeretében a felcsatolt fájlrendszer gyökere i-csomópontjának megtalálásához egyszerűen a másik mutatót kell követnünk. Ezt kihasználva a fájlrendszer tovább folytatja a keresést, ami jelen esetben a merevlemez 2. partíciójának gyökérkönyvtárában az */ast/* könyvtár keresését jelenti.

5.6.7. Az állományleírók

Egy fájl megnyitása után a felhasználói processzus egy állományleírót kap vissza, amelyet ezután a read, illetve write hívások használnak. A most következő részben ezen állományleírók fájlrendszeren belüli kezelésével ismerkedünk meg.

A fájlrendszer, az operációs rendszer központi részéhez és a memóriakezelőhöz hasonlóan, a processzustábla egy részét a saját címterületén tartja. Mezői közül háromnak rendkívül fontos szerepe van. Az első kettő a gyökérkönyvtár, illetve a pillanatnyi munkakönyvtár i-csomópontját kijelölő mutató. Egy elérési út megkeresése – ahogy az például az 5.16. ábrán látható – attól függően, hogy az elérési út abszolút vagy relatív-e, mindig a kettő közül valamelyik vizsgálatával kezdődik. Ezen mutatókat a chroot és a chdir rendszerhívások folyamatosan frissítik, hogy azok mindig az új gyökér-, illetve munkakönyvtárra mutassanak.

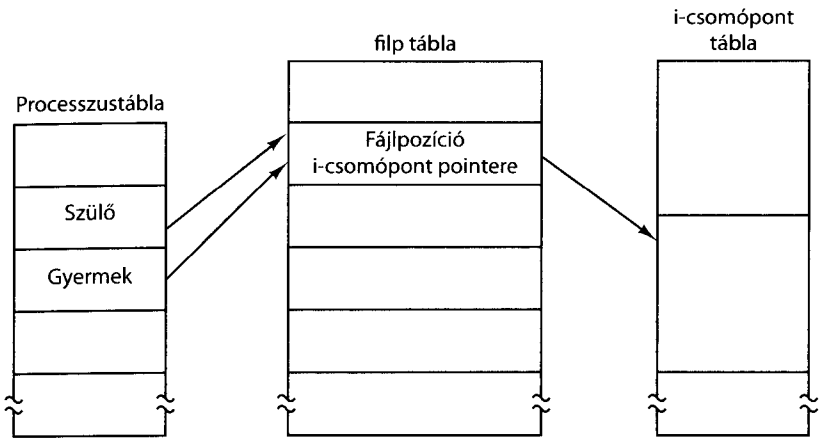
A processzustábla harmadik érdekes mezője az állományleíró azonosítója által kijelölt tömb. Ez arra való, hogy egy állományleíró átadásakor megtalálhassuk a megfelelő állományt. Első pillantásra úgy tűnhet, elégséges annyi, hogy a tömb *k*-adik eleme a *k* állományleíróhoz tartozó állomány i-csomópontjára mutasson, hiszen az állomány megnyitásakor annak i-csomópontja a memóriába kerül, és az állomány bezárásáig ott is marad. Vagyis az állomány biztosan elérhető.

Sajnos ez az előbbi elképzelés nem helytálló, mert a MINIX 3-ban (éppúgy, mint a Unix esetén) az állományok rendkívül ravasz módokon lehetnek megosztva. A problémát az jelenti, hogy minden állomány rendelkezik egy olyan 32 bites számmal, amely a következő beolvasandó vagy kiírandó bájtot jelöli ki. Ezt az ún. **pillanatnyi fájlpozíciót** változtatja az lseek rendszerhívás. A probléma egyszerűen megfogalmazható: hol tároljuk ezt a mutatót?

Az első lehetőség, hogy betesszük az i-csomópontba. Ha két vagy több processzus ugyanazt az állományt nyittatja meg, sajnálatos módon mindegyiknek a saját állománymutatóira van szüksége, mivel azt aligha engedhetjük meg, hogy az egyik processzus lseek hívása befolyásolja a másik processzus legközelebbi beolvasását. Tehát a pillanatnyi állományvég nem kerülhet az i-csomópontba.

Mi lenne, ha azt a processzustáblába tennénk? Miért nem létezik egy második, az állományleíró tömbbel szinkronban lévő, az állományok pillanatnyi végét megadó tömb? Ez a megközelítés szintén nem működik, azonban sokkal árnyaltabb okok miatt. A probléma alapvetően a fork rendszerhívás szemantikájával kapcsolatos. Egy processzus elágazásakor mind a szülő-, mind a gyermekprocesszusnak ugyanazt az állománymutatót kell a pillanatnyilag megnyitás alatt álló állományokhoz birtokolnia.

Hogy a problémát jobban megérthessük, tekintsünk egy olyan parancsértelmezőt, amelynek kimenetét egy állományba irányítottuk át. Amikor a parancsér-



5.39. ábra. *A fájlpozíció-információ átadása a szülő- és gyermekprocesszus között*

telmező az első programot elágaztatja, a szabványos kimenet fájlpozíciója 0. Ezt örökli a gyermekprocesszus, amely mondjuk kiír 1 KB információt. A gyermekprocesszus befejeződésekor a közös fájlpozíciónak 1024-nek kell lennie.

Ezután a parancsértelmező beolvassa a következő részt, és elindít egy másik gyermekprocesszust. Lényeges, hogy ez a második gyermekprocesszus az 1 KB fájlpozíciót örökölje a parancsértelmezőtől, vagyis azon a helyen kezdje meg az írást, ahol az első program azt befejezte. Ha a parancsértelmező a gyermekprocesszusnak nem adná át a fájlpozíció-információt, a második program egyszerűen felülírná az első program kimenetét ahelyett, hogy ahhoz a sajátját hozzáírná.

Mindez azt mutatja, hogy a pillanatnyi fájlpozíció nem kerülhet a processzustáblába sem, mert annak a processzusok között ténylegesen közösnek kell lennie. A MINIX 3-ban ennek biztosítása érdekében a következő megoldás használatos: létrehozunk egy olyan új, közös táblát – ez az ún. ***filp*** tábla –, amely az összes fájlpozíciót tartalmazza. Használatát az 5.39. ábrán szemléltetjük. Miután a fájlpozíciók ezáltal a processzusok között valóban megosztásra kerülnek, a fork rendszerhívás szemantikája helyesen kivitelezhető, és a parancsértelmező is helyesen fog működni.

Bár az egyetlen olyan információ, amelyet a *filp* táblának tartalmaznia kell, a közös állományvég, az i-csomópont mutatójának itt történő feltüntetése megszokott dolog. Ily módon ugyanis a processzustábla állományleíró tömbje mindössze egy *filp* elemet kijelölő mutatót tartalmaz. A *filp* elem tartalmazza továbbá az állomány elérhetőségét (engedélyezési bitek), azokat a jelzőket, amelyek az állomány esetleges speciális megnyitási módját tükrözik, és egy olyan számlálót, amely az adott állományt használó különböző processzusok számát tartja nyilván. Ezzel lehetővé válik, hogy a fájlrendszer pontosan tudja, mikor is fejeződik be az utolsó olyan processzus, amely a *filp* tábla ezen elemét használta. Ennek megtörténte után pedig visszaigényelheti az adott *filp* tábla rekeszt.

5.6.8. Fájlzárolás

A fájlrendszerkezelésnek van még egy olyan területe, amely szintén egy speciális tábla jelenlétét kívánja meg. Ez a fájlokhoz való hozzáférés letiltása vagy zárolása. A MINIX 3 támogatja a POSIX szerinti processzusközi kommunikáció során használatos ún. **felügyeleti fájlzárolás** technikáját. Ez teszi lehetővé egy adott állomány tetszőleges részének, vagy akár több részének is hozzáférhetetlenként való feltüntetését. Maga az operációs rendszer nem kényszeríti ki az állományhoz való hozzáférés megszüntetését. Az egyes processzusoktól elvárjuk azonban, hogy megfelelően viselkedjenek, és mielőtt bármi olyasmit tennének, ami egy másik processzussal ütközne, vizsgálják meg, vajon számukra hozzáférhető-e az adott állomány.

A zárolások számára kialakított különálló tábla létrehozásának okai az előző részben tárgyalt *filp* tábla létrehozásának okaihoz hasonlók. Egy egyszerű processzus egyidejűleg több zárolást is aktiválhat, illetve egy állomány különböző részeit egynél több processzus is zárolhatja (persze ezek a zárolások nem keresztezhetik egymást). Ennek következtében sem a processzustábla, sem a *filp* tábla nem alkalmas a zárolások számontartására. Az i-csomópont erre szintén nem megfelelő, mivel egy állomány több zárolással is rendelkezhet.

A MINIX 3 a zárolások nyilvántartására tehát egy újabb táblát, az ún. *file_lock* táblát használja. A tábla minden egyes rekeszében van hely a zárolás típusának – ez jelzi, hogy a zárolás az adott állomány írására vagy olvasására vonatkozik-e –, a zárolást életbe léptető processzus azonosítójának, a zárolt állomány i-csomópontját kijelölő mutatónak és a zárolt terület első, illetve utolsó bájtjának az állomány kezdetétől számított relatív helye számára.

5.6.9. Adatcsövek és speciális fájlok

Az adatcsövek és a speciális fájlok egy lényeges ponton különböznek a közönséges fájloktól. Amikor egy processzus lemezen lévő fájlba próbál írni vagy onnan olvasni, a művelet majdnem biztosan nem tart tovább néhány tized másodpercnél. A legrosszabb esetben két vagy három lemezelérésre lesz szükség, többre nem. Amikor azonban az olvasás egy adatcsőből történik, teljesen más helyzettel kerülünk szembe: ha az adatcső üres, az olvasónak mindaddig várakoznia kell, amíg egy másik processzus valamilyen adatot nem tesz az adatcsőbe. Ez pedig akár néhány óráig is eltarthat. Ehhez hasonlóan, terminálról történő beolvasáskor is mindaddig várakoznia kell a processzusnak, amíg valaki be nem gépel valamit.

A fájlrendszer azon alapszabálya tehát, amely szerint egy adott kérés kiszolgálása annak befejezéséig tart, nem működik. Ezeket a kéréseket fel kell függesztenie, majd egy későbbi időpontban újra kell őket éleszteni. Amikor egy processzus adatcsőből (vagy adatcsőbe) próbál meg olvasni (vagy írni), a fájlrendszer rögtön ellenőrizheti az adatcső állapotát, hogy felmérje, befejezhető-e az adott művelet, vagy sem. Ha igen, a fájlrendszer befejezi azt, ha nem, feljegyzi a rendszerhívás

paramétereit a processzustáblában. Ily módon a megfelelő időpontban ismét elindíthatja majd a processzust.

Jegyezzük meg, hogy a fájlrendszernek a hívó felfüggesztéséhez semmit sem kell tennie. Mindössze a válasz küldését kell visszatartania, ezzel blokkolva a hívót, amely erre a visszajelzésre vár. Egy processzus felfüggesztése után a fájlrendszer tehát visszakerül a központi ciklusba, ahol a következő rendszerhívásig várakozik. Mihelyt egy másik processzus az adatcső állapotában olyan változást idéz elő, amelynek következtében a felfüggesztett processzus befejezhetővé válik, a fájlrendszer beállítja a megfelelő jelzőt, és miután legközelebb visszakerül a központi ciklusba, kiolvassa a processzustáblából a felfüggesztett processzus paramétereit, és végrehajtja a hívást.

A terminálok és egyéb különleges karakterállományok esetén ettől egy kicsit eltérő a helyzet. Bármely speciális állományhoz tartozó i-csomópont két azonosítót tartalmaz: ezek a fő, illetve a másodlagos eszközazonosító számok. A fő eszközazonosító szám az illető eszköz típusát határozza meg (például RAM-lemez, hajlékonylemez, merevlemez, terminál). Ezt a számot a fájlrendszertábla, amely tulajdonképpen az eszközt rendeli hozzá a megfelelő feladathoz (például I/O-meghajtó), sorszámként használja. Valójában a fő eszközazonosító szám határozza meg, hogy melyik I/O-meghajtót kell meghívni. A másodlagos eszközazonosító szám a meghajtónak paraméterként kerül átadásra, és azt jelöli ki, hogy tulajdonképpen melyik eszközt is fogjuk használni, például a kettes számú terminált vagy az egyes lemezmeghajtót.

Néhány esetben, leginkább a termináleszközök esetében, a másodlagos eszközazonosító szám az adott feladat által kezelt eszközök csoportjáról is hordoz némi információt. Az elsődleges MINIX 3-konzol, */dev/console,* például egy (4, 0) (fő, illetve másodlagos eszközazonosító számú) eszköz. A virtuális konzolokat a meghajtóprogram ugyanazon része kezeli. Ezek a */dev/ttyc1* (4, 1), */dev/ttyc2* (4, 2) stb. eszközök. A soros egyirányú terminálok – például */dev/tty00* és */dev/tty01* – kezeléséhez egy másik alacsony szintű program szükséges. Ezen eszközökhöz a (4, 16), illetve (4, 17) eszközazonosító számokat rendelték. Ezekhez hasonlóan, a hálózati terminálok is ál-terminálmeghajtókat használnak, vagyis ezekhez egy további alacsony szintű program szükségeltetik. A MINIX 3 ezekhez az eszközökhöz, *ttyp0, ttyp1* stb. olyan eszközazonosító számokat rendel, mint például (4, 128), illetve (4, 129). Ezen áleszközök mindegyikéhez tartozik még egy további *ptyp0, ptyp1* stb. eszköz is. Az ezekhez tartozó fő és másodlagos eszközazonosító számpárok a (4, 192), (4, 193) stb. számpárok. Ezeket a számokat azért választották az itt leírt módon, hogy a meghajtó processzus számára megkönnyítsék a különböző eszközcsoportokhoz rendelt alacsony szintű függvények kiválasztását. Arra azonban nincs mód, hogy egy MINIX 3-rendszer 192 darab vagy ennél több terminállal legyen felszerelve.

Amikor egy processzus speciális állományból olvas, a fájlrendszer először kiolvassa az állomány i-csomópontjából a fő és másodlagos eszközazonosító számot, majd a fő eszközazonosító számot a processzustábla indexeként használva hozzárendeli az eszközhöz a megfelelő processzust. Miután a processzus számát megtalálta, olyan üzenetet küld számára, amely paraméterként a másodlagos eszközazonosító számot, a végrehajtandó műveletet, a hívó processzus azonosítóját és átmeneti tárterületének címét, illetve a továbbítandó bájtok számát tartalmazza. Ennek formátuma a 3.15. ábrán láthatóval egyezik meg, azzal az eltéréssel, hogy a *POSITION* változót ebben az esetben nem használjuk.

Amennyiben a meghajtó azonnal folytatni tudja munkáját (például már begépeltük a terminálon a bemenetet), saját belső átmeneti táraiból az adatokat a felhasználóhoz másolja és üzenetet küld a fájlrendszernek, amelyben a munka elvégzéséről tesz jelentést. Ekkor a fájlrendszer egy válaszüzenetben értesíti a felhasználót, és ezzel befejeződik a hívás. Jegyezzük meg, hogy a meghajtó nem másol adatokat a fájlrendszerbe. A blokkeszközökről származó adatok keresztülmennek a blokkgyorsítótáron, de a különleges karakterállományból származók nem.

Ha azonban a meghajtó nem képes azonnal folytatni a munkáját, az üzenetben szereplő paramétereket feljegyzi belső tábláiba, és azonnal egy olyan értelmű üzenetet küld a fájlrendszernek, amely szerint a hívás nem fejezhető be. Ezen a ponton a fájlrendszer ugyanolyan állapotba kerül, mint akkor, amikor azt észleli, hogy valamilyen processzus az üres adatcsőből próbál meg adatot olvasni. Feljegyzi az adott processzus felfüggesztésének tényét, és várja a következő üzenetet.

Miután a meghajtó elegendő adatra tett szert a hívás befejezéséhez, ezeket a még mindig blokkolás alatt álló felhasználó átmeneti adattárába továbbítja, majd erről jelentést tesz a fájlrendszernek. Ekkor a fájlrendszernek mindössze annyit kell tennie, hogy egy üzenet elküldésével megszüntesse a felhasználó blokkolását, és megküldje annak a továbbított bájtok számát.

5.6.10. Egy példa: a read rendszerhívás

Amint azt rövidesen látni fogjuk, a fájlrendszer forráskódjának nagy része a rendszerhívások megvalósítására fordítódik. Éppen ezért helyénvalónak érezzük, hogy mostani áttekintésünket a legfontosabb rendszerhívás, a read tömör jellemzésével zárjuk.

Amikor egy felhasználói program egy közönséges fájlból való beolvasás céljából végrehajtja az

```
n = read(fd, buffer, nbytes);
```

utasítást, akkor a *read* könyvtári eljárás kerül meghívásra három átadott paraméterrel. Az eljárás először létrehoz egy olyan üzenetet, amely a read rendszerhívás kódjával mint az üzenet típusával együtt tartalmazza ezt a három paramétert is. Ezután az üzenetet elküldi a fájlrendszernek és blokkol, várván a visszajelzést. Az üzenet megérkezése után a fájlrendszer az üzenet típusát táblái számára mutatóként használva meghívja a beolvasást kezelő eljárást.

Ezen eljárás az üzenetből kiolvassa az állományleírót, amelyet a későbbiekben a beolvasandó állományhoz tartozó *filp* tábla elemnek és i-csomópontoknak a megkeresésére használ fel (lásd 5.39. ábra). Ezután a kért adat olyan méretű részekre osztódik, amelyek mindegyike elfér egy blokkban. Ha például a fájlpozíció jelen-

leg 600 és 1 KB információt kértünk, a kért adat – 1 KB méretű blokkokat feltételezve – 2 darabra bomlik, 600-tól 1023-ig, illetve 1024-től 1623-ig.

A következő lépésben annak ellenőrzése történik meg egyenként, hogy a darabok nincsenek-e véletlenül a gyorsítótárban. Ha a darabok nincsenek ott, a fájlrendszer megkeresi azt a legrégebben használt átmeneti adattárat, amely jelenleg sincs használatban, és lefoglalja, miközben egy üzenetben arra kéri a lemez eszközkezelőt, hogy az illető tár tartalmát, ha az korábban megváltozott, írja lemezre. Az eljárás ezek után a beolvasandó blokk megszerzésére utasítja az eszközkezelőt.

Miután a blokk bekerült a gyorsítótárba, a fájlrendszer a rendszertaszkhoz küld egy üzenetet, amelyben az adatnak a felhasználói ideiglenes tárterület megfelelő helyére történő bemásolását kéri (például a 600 és az 1023 között lévő bájtok az átmeneti tár elejétől, míg az 1024 és az 1623 közötti bájtok ez után, az ideiglenes tár 424-es helyétől kezdődően íródnak be). A másolat elkészülte után a fájlrendszer egy válaszüzenetben értesíti a felhasználót az átmásolt bájtok számáról.

Ezen üzenet felhasználóhoz történő megérkezése után a *read* könyvtári függvény kiolvassa belőle a válaszkódot, és azt függvényértékként visszaadja a hívónak.

Történik azonban egy olyan lépés is, amely valójában nem része a read rendszerhívásnak. Miután a fájlrendszer az olvasást befejezte és a válaszüzenetet is elküldte – feltéve, hogy egyéb feltételek teljesülése mellett a következő beolvasás is blokkeszközről esedékes –, elindítja a következő blokk beolvasását. Mivel az egymás utáni sorozatos állományolvasások eléggé gyakoriak, ésszerűnek tűnhet az a feltevés, miszerint a soron következő beolvasási kérelem éppen az állomány következő blokkjára irányul. Ez azt is valószínűsíti továbbá, hogy a későbbiekben a kívánt blokk szükség esetén már megtalálható lesz a gyorsítótárban.

5.7. A MINIX 3 fájlrendszerének megvalósítása

A MINIX 3 fájlrendszere meglehetősen nagy (több mint 100 oldalnyi C program), de azért eléggé átlátható. Rendszerhívások végrehajtására irányuló kérések érkeznek, hajtódnak végre és válaszok kerülnek elküldésre. A következő részekben a MINIX 3-at állományonként vizsgáljuk meg, kiemelve a legérdekesebb alkotóelemeket. A forráskód maga is rendkívül sok, az olvasót segítő megjegyzést tartalmaz.

Amikor a MINIX 3 egyéb részeinek forráskódját mutattuk be, akkor általában az eljárások központi ciklusát vizsgáltuk meg először, és csak ezután kerültek sorra a különböző üzenettípusokat kezelő rutinok. A fájlrendszer esetén másként járunk el: először a fontos alrendszereket tárgyaljuk (mint például a gyorsítótár vagy az i-csomópontok kezelése), és csak ezután térünk rá a központi ciklusra, illetve az állományokon műveleteket végző rendszerhívásokra. Ezt követően a könyvtárakon műveleteket végző rendszerhívásokat, majd az ezen kategóriák egyikébe sem besorolható rendszerhívásokat fogjuk megtárgyalni. Végül azt tárgyaljuk, hogyan kezeli a rendszer az eszközspecifikus fájlokat.

5.7.1. Definíciós állományok és globális adatszerkezetek

A fájlrendszerben használatos adatszerkezetek és táblák éppúgy a definíciós állományokban vannak definiálva, mint az operációs rendszer központi részénél (kernel) és a memóriakezelőnél. Az adatszerkezetek közül néhány az *include/* könyvtáraiban lévő ún. rendszer definíciós állományokban található. Az *include/sys/stat.h* definiálja például azt a formátumot, amelyben a rendszerhívások más programoknak az i-csomópontra vonatkozó információt átadják, míg a könyvtárbejegyzés szerkezete az *include/sys/dir.h* definíciós állományban kerül megadásra. A POSIX-nak az említett két állomány mindegyikére szüksége van. A fájlrendszert rendkívül sok, az *include/minix/config.h* globális konfigurációs állományban található definíció befolyásolja, ilyen például az *NR_BUFS,* illetve az *NR_BUF_HASH,* amelyek a blokkgyorsítótár méretét szabályozzák.

A fájlrendszer definíciós állományai

A fájlrendszer saját definíciós állományai a fájlrendszer *src/fs/* elnevezésű forráskönyvtárában találhatók. Közülük a MINIX 3 egyéb részeinek tanulmányozása során már sokat megismertünk. A fájlrendszer központi *fs.h* definíciós állománya (20900. sor) nagyon hasonló az *src/kernel/kernel.h,* illetve az *src/pm/pm.h* definíciós állományhoz, és olyan egyéb definíciós állományokat tartalmaz, amelyek a fájlrendszer többi állományának C forráskódja számára életbe vágóan fontosak. Éppúgy, mint a MINIX 3 egyéb részeinél, a fájlrendszer központi definíciós állománya tartalmazza a fájlrendszer saját *const.h, type.h, proto.h* és *glo.h* állományait. Vizsgáljuk meg most ezeket egy kicsit közelebbről.

A *const.h* (21000. sor) definiál néhány olyan állandót (táblaméretek, illetve -jelzők), amelyeket a fájlrendszer minden része használ. A MINIX 3-nak azonban már történelme van. Egy korábbi változata teljesen más fájlrendszert használt, így azon felhasználók számára, akik ezen régebbi változat által írott állományokat szeretnék használni, mind a régi *V1*, mind a jelenlegi *V2* típusú fájlrendszer támogat. A régebbi rendszerek támogatása nemcsak ezért fontos, hogy más MINIX-rendszer fájljait elérhessük, hanem azért is, mert ez lehetővé teszi fájlok cseréjét.

Más operációs rendszerek, például a Linux, most is támogat régebbi MINIX-fájlrendszert. (Talán ironikus, hogy a Linux még támogatja a régebbi MINIX-fájlrendszert, de a MINIX 3 már nem.) Van néhány segédprogram MS-DOS és Windows számára a régebbi MINIX-fájlok elérésére. A fájlrendszer szuperblokkja tartalmaz egy ún. **mágikus** (vagy **bűvös**) **szám**ot, amelynek segítségével az operációs rendszer felismeri a fájlrendszer típusát; ezeket a számokat a *SUPER_MAGIC,* illetve a *SUPER_V2* és *SUPER_V3* állandók határozzák meg. Van továbbá ezeknek egy *_REV* végződésű változatuk azokra a V1 és V2 verziókra, amelyek a mágikus számot fordított sorrendben tárolják. Ezeket régebbi MINIX-rendszerek eltérő bájtsorrendű (kis-endián helyett nagy-endián) rendszerekre történő portolására lehet használni, így ha egy hordozható lemezt egy más bájtsorrendű gépen írtak,

azt felismerheti. A MINIX 3.1.0 verziója definiálja a *SUPER_V3_REV* mágikus számot, bár ez nem szükséges, de valószínűleg ez a definíció a jövőben beépül.

A *type.h* definíciós állomány (21100. sor) definiálja a régi V1 és az új V2 i-csomópont szerkezetét olyan formában, ahogyan azok a lemezen megtalálhatók. Az i-csomópont szerkezete nem változott a V3 verzióban, tehát a V2 i-csomópont használatos a V3 fájlrendszerben is. A V2 i-csomópont kétszer nagyobb, mint a régi V1 i-csomópont, amelyet eredetileg a merevlemezzel nem rendelkező, 360 KB-os hajlékonylemezt használó rendszerek számára, a tömörség kívánalmát szem előtt tartva szerkesztettek meg. Az új változatban a Unix-rendszereknél használatos három időmező számára biztosítottak helyet. A V1 i-csomópontban mindössze egyetlen időmező kapott helyet, azonban a stat és az fstat rendszerhívások ezt a tényt „elhomályosítva" egy olyan *stat* szerkezetet adnak vissza, amely mindhárom mezőt magában foglalja. Mindkét fájlrendszer támogatásának kialakításánál felmerül azonban egy apró nehézség, amelyet a 21116. sor megjegyzése tesz nyilvánvalóvá. A régebbi MINIX-programok a *gid_t* típust 8 bitesnek várják, így a *d2_gid*-et *u16_t* típusúként kell definiálni.

A *proto.h* definíciós állomány (21200. sor) biztosítja a régi K&R és az újabb ANSI szabvány C fordítók számára egyaránt elérhető formában a függvényprototípusokat. Ez egy hosszú, mindazonáltal eléggé érdektelen állomány. Egy dolgot érdemes azonban megjegyeznünk: mivel nagyon sok, a fájlrendszer által kezelt hívás van, és mivel a fájlrendszer olyan szervezésű, amilyen, a különböző *do_xxx* eljárások nagyszámú állományban vannak szétszórva. A *proto.h* az állományok szerint szervezett, így kényelmes lehetőséget biztosít egy olyan állomány forráskódjának a megtalálására, amely egy jól meghatározott rendszerhívást kezel.

Végezetül a *glo.h* definíciós állomány (21400. sor) definiálja a globális változókat. A beérkező, illetve a válaszüzenetek átmeneti tárolóhelye szintén itt található. Ezek a változók a fájlrendszer tetszőleges része által elérhetők, amit az *EXTERN* makróutasítás már megismert trükkös használata tesz lehetővé. A tárolási hely lefoglalása, éppúgy, mint a MINIX 3 más részeinél, a *table.c* forráskód lefordításakor történik meg.

A processzustábla fájlrendszerbeli részét az *fproc.h* definíciós állomány (21500. sor) tartalmazza. Az *fproc* tömböt az *EXTERN* makróutasítás segítségével hozzuk létre. Ez hordozza az üzemmód maszkot, a pillanatnyi gyökér- és munkakönyvtár i-csomópontját kijelölő mutatókat, az állományleíró tömböt, a felhasználói és csoportazonosítót, valamint minden egyes processzus terminálazonosítóját. A processzus azonosítója és csoportazonosítója szintén itt található meg. A processzus azonosítója szerepel a processzuskezelőben található processzustáblában is.

Néhány további mező arra használatos, hogy a működésük alatt felfüggeszthető rendszerhívások (például üres adatcsőből történő olvasások) paramétereit tároljuk bennük. Az *fp_suspended* és az *fp_revived* mezők valójában csak egy-egy bitet igényelnének, azonban majdnem minden fordító jobb végrehajtható kódot tud készíteni, ha a mező egy teljes karaktert és nem csak egy bitet foglal el. A POSIX szabvány által hívott *FD_CLOEXEC* bitjei számára szintén rendelkezésre áll egy mező. Ezek azt hivatottak jelezni, hogy az adott állományt egy exec rendszerhívás végrehajtása után azonnal be kell-e zárni.

Ezzel eljutottunk azon állományokhoz, melyek a fájlrendszer által fenntartott további táblákat definiálják. Az első, amit *buf.h* néven ismerünk (21600. sor), határozza meg a blokkgyorsítótárat. Az itt megjelenő szerkezeteket az *EXTERN* makróutasítással hozzuk létre. A *buf* tömb tartalmazza az átmeneti adattárakat, amelyek mindegyike egy *b* adatrészből és egy mutatókkal, jelzőkkel, illetve számlálókkal teli fejlécből áll. Az adatrészt öt típus *union* típusaként hozzuk létre (21618–21632. sor), mert előfordulhat, hogy a későbbiekben kényelmesebb lesz rá időnként karaktertömbként, időnként könyvtárként stb. hivatkozni.

A hármas számú átmeneti adattár adatrészének karaktertömb formájában történő hivatkozása során a *buf*[3].*b.b__data* formátum használata szükséges, mert a *buf*[3].*b* hivatkozik magára a *union*-ra, amiből a *b__data* mezőt választjuk ki. Azzal együtt, hogy a fenti szintaktika helyes, rendkívül nehézkes is. Éppen ezért a 21649. sorban létrehozunk egy *b_data* elnevezésű makróutasítást, amely lehetővé teszi, hogy a későbbiekben a *buf*[3].*b_data* formátumot használhassuk. Jegyezzük meg, hogy a *b__data* (ez utal a *union* egy mezőjére) két aláhúzást tartalmaz, míg a *b_data* (a makróutasítás) csak egyet. Ez azért van így, hogy a kettőt megkülönböztethessük egymástól. Az adott blokk más formátumban való elérésére szolgáló további makróutasítások a 21650. és a 21655. sorok között kerülnek definiálásra.

Az átmeneti adattár hasítótábláját, *buf_hash,* a 21657. sorban hozzuk létre. Ennek minden egyes eleme egy átmeneti adattárakból felépülő láncra mutat. A *WRITE_IMMED* jelzi, ha a blokk tartalmának megváltozása után nyomban lemezre kell írni, és a *ONE_SHOT* bit jelzi, hogy a blokkra valószínűleg nem lesz hamarosan szükség. Jelenleg egyik sem használatos, de lehetőséget ad a módosításra, ha valakinek briliáns ötlete támad a hatékonyság és megbízhatóság növelésére alkalmas gyorsítótár-kezelésre.

Végezetül az utolsó sorban a *HASH_MASK* kerül definiálásra az *include/minix/config.h* definíciós állományban beállított *NR_BUF_HASH* változó értékének megfelelően. Azt, hogy egy blokk átmeneti adattárának keresésekor a *buf_hash* melyik eleménél kezdjük a keresést, a *HASH_MASK* blokkazonosítóval vett ÉS művelet eredménye határozza meg.

A *file.h* állomány (21700. sor) tartalmazza a közbenső (*EXTERN* által definiált) *filp* táblát, amelyet a pillanatnyi fájlpozíció és i-csomópont mutatójának tárolására használunk (lásd 5.39. ábra). Ez meghatározza egyben azt is, hogy az illető állomány olvasható-e, írható-e, vagy mindkét művelet elvégezhető-e rajta, illetve azt, hogy hány állományleíró mutat pillanatnyilag erre az elemre.

A fájlzárolási tábla, *file_lock* (*EXTERN* által definiált), a *lock.h* definíciós állományban (21800. sor) található meg. A tömb méretét a *const.h* definíciós állományban 8-ra beállított *NR_LOCKS* változó határozza meg. Ha egy MINIX 3-rendszeren többfelhasználós adatbázist szeretnénk létrehozni, ezt a számot kell megnövelnünk.

Az *inode.h* definíciós állományban (21900. sor) definiáljuk (az *EXTERN* használatával) az *inode* elnevezésű i-csomópont táblát. Ez tartalmazza a pillanatnyilag használatban lévő i-csomópontokat. Amint azt már az előzőkben megtárgyaltuk, egy állomány megnyitásakor annak i-csomópontja a memóriába kerül, és az állomány bezárásáig ott is marad. Az *inode* struktúra definíciója olyan információk-

ról gondoskodik, amelyek a memóriában megvannak ugyan, de a lemezen lévő i-csomópontba nem íródnak be. Megemlítjük, hogy ennek mindössze egy változata létezik, és semmi sem változatfüggő benne. Amikor a lemezről beolvassuk az i-csomópontot, a V1, illetve V2/V3 fájlrendszerek közti különbségek kompenzálódnak. A fájlrendszer többi részének egyáltalán nem szükséges a lemezen lévő fájlrendszer formátumát ismernie, legalábbis addig nem, amíg el nem jön a megváltozott információ lemezre írásának ideje.

Ezen a ponton a mezők többségének elnevezése önmagáért beszél. Az *i_seek* megérdemel azonban néhány megjegyzést. Már korábban is említettük: amikor a fájlrendszer észleli, hogy egy állomány folyamatos olvasás alatt áll, egy optimalizálás eredményeként még azelőtt megpróbál a blokkgyorsítótárba blokkokat beolvasni, hogy erre irányuló kérés történt volna. Közvetlen elérésű állományoknál nincs előreolvasás; amikor egy lseek rendszerhívás történik az *i_seek* mező egyesre állítódik, hogy az előreolvasást megakadályozza.

A *param.h* nevű állomány (22000. sor) hasonló a processzuskezelőben lévő ugyanilyen elnevezésű állományhoz. Paramétereket tartalmazó üzenetmezők számára definiál neveket, így például a programkód hivatkozhat az *m_in.m1_p1* helyett az *m_in.buffer* névvel – ami az *.inm* üzenet számára fenntartott átmeneti adattár egy mezőjét választja ki.

A *super.h* (22100. sor) definíciós állományban találjuk a *super_block* nevű szuperblokktábla definícióját. A rendszer elindulása után ide töltődik be a gyökéreszköz szuperblokkja. További fájlrendszerek felcsatolásakor ezek szuperblokkjai szintén ide íródnak be. Éppúgy, mint a többi tábla esetén, a *super_block* szintén az *EXTERN* makróutasítás segítségével definiálódik.

A fájlrendszer tárolóhely-foglalása

Az ebben a részben utolsóként tárgyalandó állomány nem definíciós állomány. Mindazonáltal, ahogy azt a processzuskezelő vizsgálatakor már megtettük, most is helyénvalónak tűnik a definíciós állományok áttekintése után azonnal megvizsgálni a *table.c* állományt, hiszen ennek lefordításához az összes definíciós állományra szükségünk van. A legtöbb, már említett adatszerkezet – a blokkgyorsító, a *filp* tábla, és így tovább –, éppúgy, mint a fájlrendszer globális változói és a processzustábla fájlrendszerbeli része, az *EXTERN* makróutasításon keresztül definiálódik. Amint azt a MINIX 3 más részeinél már láttuk, a tárolóhely lefoglalása valójában a *table.c* fordításakor történik meg. Ez az állomány azonban tartalmaz még két, kezdeti értékekkel feltöltött tömböt is. A *call_vector* az a központi ciklusban használt, mutatókból álló tömb, amely meghatározza, hogy melyik eljárás melyik rendszerhívási számot kezeli. Hasonló táblával korábban a processzuskezelés során találkoztunk.

5.7.2. A táblák kezelése

A legfontosabb táblák mindegyikéhez – a blokkokhoz, az i-csomópontokhoz, a szuperblokkokhoz stb. tartozó táblák mindegyikéhez – tartozik egy olyan állomány, amely a táblát kezelő eljárásokat fogja össze. A fájlrendszer többi része erőteljesen kihasználja ezeket az eljárásokat, amelyek a legfontosabb csatolófelületet alkotják a táblák és a fájlrendszer között. Éppen ezért célszerű a fájlrendszer forráskódjának tanulmányozását ezekkel az állományokkal kezdeni.

A blokkok kezelése

A blokkgyorsítótárat a *cache.c* állományban lévő eljárások kezelik. Ez az állomány az 5.40. ábrán feltüntetett kilenc eljárást foglalja magában. Ezek közül az első, a *get_block* eljárás (22426. sor) az, amelynek használatával a fájlrendszer megkapja az adatblokkokat. Amikor a fájlrendszer valamely eljárásának szüksége van egy felhasználói adatblokkra, könyvtárblokkra, szuperblokkra vagy bármely más blokkra, a *get_block* eljárást hívja meg, átadva annak az eszköz- és a blokkazonosító számot.

A *get_block* eljárás, hívása után, először megvizsgálja a blokkgyorsítótárat, hogy a kívánt blokk nincs-e ott véletlenül. Ha ott van, akkor visszaküldi a blokk mutatóját, ha nem, be kell a blokkot olvasnia. A gyorsítótárban a blokkok az *NR_BUF_HASH* által összefűzött láncok formájában vannak egymáshoz kapcsolva. Az *NR_BUF_HASH* az *NR_BUF* változóval együtt, amely a blokkgyorsítótár méretét határozza meg, állítható változó. Mindkettő értékét az *include/minix/config.h* definíciós állományban tudjuk szabályozni. Ennek a résznek a végén néhány mondat erejéig majd visszatérünk még a blokkgyorsítótár és a hasítótábla méretének optimális beállítására. A *HASH_MASK* értéke *NR_BUF_HASH* – 1. 256 hasítólánc esetén a maszk értéke 255, így az összes különböző láncban szereplő blokkok azonosítója ugyanarra a 8 bitből álló karakterfüzérre végződik, azaz 00000000, 00000001, ... vagy 11111111.

Az első lépés rendszerint az, hogy a blokk hasítóláncát keressük meg. Egyetlen speciális esetben azonban, amikor egy kevés adatot tartalmazó állományból épp egy üres helyet olvasunk be, az előbb említett keresés nem történik meg. Ez az oka

Eljárás neve	Tevékenység
get_block	Egy blokk írásra vagy olvasásra való elővétele
put_block	A get_block által korábban elővett blokk visszaküldése
alloc_zone	Új zóna foglalása (az állomány meghosszabbítása céljából)
free_zone	Egy zóna felszabadítása (az állomány törlésekor)
rw_block	Egy blokk lemez és gyorsítótár közötti átvitele
invalidate	Valamely eszközhöz tartozó, a gyorsítótárban lévő összes blokk törlése
flushall	Egy eszköz megváltozott tartalmú blokkjainak kiírása
rw_scattered	Szétszórt adatok eszközről eszközre történő olvasása/írása
rm_lru	Egy blokk törlése az LRU-listából

5.40. ábra. *A blokk-kezelés során használatos eljárások*

a 22454. sorban szereplő vizsgálatnak. Máskülönben a forráskód következő két sora, a blokk azonosítóján egy *HASH_MASK*-kal vett ÉS műveletet végrehajtva, a *bp* mutató értékét azon lánc kezdőpontjára állítaná, amelyben a keresett blokk megtalálható lenne, ha az a gyorsítótárban helyezkedne el. A következő sorban kezdődő ciklus ezt a láncot vizsgálja meg, hogy megnézze, valóban benne van-e a blokk. Ha megtalálja, és az éppen nincs használatban, eltávolítja az LRU-listából. Ha a blokk már használatban van, akkor az egyébként sem lesz az LRU-listában. A 22463. sorban kapja vissza a hívó a megtalált blokkot kijelölő mutatót.

Ha a blokk nincs a hasítóláncban, akkor nincs a gyorsítótárban sem, így az LRU-lista legrégebben használt blokkját vesszük elő. A kiválasztott átmeneti adattárat eltávolítjuk a hozzá tartozó hasítóláncból, hiszen az egy új blokkazonosítót fog kapni, ami egyúttal azt is jelenti, hogy egy másik hasítólánchoz fog tartozni. Amennyiben tartalma a legutóbbi változás óta nem íródott volna lemezre, az most a 22495. sorban megtörténik. A *flushall* eljárás hívásával az összes többi megváltozott tartalmú blokk is kiíródik ugyanarra az eszközre. A pillanatnyilag használt blokkok azonban ilyenkor nem íródnak ki, mivel azok nem tartoznak az LRU-listához. Mindazonáltal, nehezen fogunk blokkot használatban lévőnek találni, hiszen normális esetben egy adott blokk használat után a *put_block* eljárás hatására rögtön felszabadul.

Mihelyt rendelkezésünkre áll az átmeneti adattár, annak minden mezőjét (beleértve a *b_dev* nevűt is) az új paramétereknek megfelelően felfrissítjük (22499–22504. sor). Ezután a kívánt blokk beolvasható a lemezről. Létezik azonban két olyan eset, amikor a blokkot nem szükségszerű beolvasnunk a lemezről. A *get_block* eljárást az *only_search* paraméterrel hívjuk. Ez jelölheti, hogy előolvasásról van-e szó. Az előolvasás során találunk egy megfelelő átmeneti adattárat, amelynek régi tartalmát szükség esetén lemezre írjuk, majd egy új blokkazonosítót rendelünk hozzá, de *b_dev* mezőjét a *NO_DEV* változóra állítjuk, ezzel jelezve, hogy egyelőre még nincs a blokkban érvényes adat. Ennek használatával az *rw_scattered* eljárás tárgyalásánál ismerkedünk majd meg. Az *only_search* annak jelzésére is felhasználható, hogy a fájlrendszernek egy blokkra csupán felülírás céljából van szüksége. Ilyenkor időpazarlás lenne először a régi tartalmat beolvasni. Ezen két eset bármelyikében felfrissítésre kerülnek ugyan a paraméterek, de a tényleges lemezről olvasás elmarad (22507– 22513. sor). Az új blokk beolvasása után a *get_block* a blokk mutatóját visszaadja a hívójának.

Tételezzük fel, hogy a fájlrendszernek átmenetileg egy könyvtárblokkra van szüksége, mert abban egy állománynevet kell kikeresnie. A könyvtárblokk megszerzése céljából ilyenkor a *get_block* eljárást hívja meg. Miután kikereste az állomány nevét, meghívja a *put_block* eljárást (22520. sor), hogy a blokk visszakerüljön a gyorsítótárba, és az átmeneti adattár szabaddá váljék arra az esetre, ha a későbbiekben egy másik blokknak lenne rá szüksége.

A */put_block/* felelős azért, hogy az újonnan visszaadott blokk az LRU-listába kerüljön, vagy ritkább esetekben lemezre íródjék. A 22544. sorban dől el, hogy a blokk az LRU-lista elejére vagy végére kerül-e. A RAM-lemezeken lévő blokkok mindig a sor elejére kerülnek. A blokkgyorsítótár nem sokat tud segíteni a RAM-lemezeknek, mivel ezek már eleve a memóriába tárolódnak, és I/O-műveletek nélkül is elérhetők. A */ONE_SHOT/* bit alapján dől el, hogy az adott blokkot megjelölték-e mint

olyat, amit a közeljövőben valószínűleg nem használnak; amennyiben megjelölték, akkor a sor elejére kerül, hogy gyorsan újrahasználhassák. Ezt a megoldást azonban sohasem vagy csak nagyon ritkán használják. A RAM-lemezen kívüli csaknem valamennyi blokk, amelyekre a közeljövőben szükség lehet, a sor végére kerül.

A blokk LRU-listában történt áthelyezése után egy újabb ellenőrzés történik, hogy megtudjuk, vajon azonnal lemezre kell-e a blokkot írni. Az előző ellenőrzéshez hasonlóan, a *WRITE_IMMED* tesztelése is maradványa egy elhagyott gyakorlatnak, jelenleg nincs megjelölve blokk azonnali kiírásra.

Az állomány méretének növekedésekor az új adatok részére időről időre új zónákat kell lefoglalni. Ezt a feladatot az *alloc_zone* eljárás (22580. sor) végzi el azáltal, hogy egy szabad zónát talál a zónabittérképen. Ha az állomány első zónájáról van szó, nem kell a zónabittérképet végigkutatnia; elegendő a szuperblokk *s_zsearch* mezőjét megnéznie, amely mindig az adott eszköz első szabad zónájára mutat. Amennyiben nem az első zónáról van szó, az *alloc_zone* az állomány zónáinak együtt tartása érdekében megkísérel egy olyan új zónát találni, amely közel van az aktuális állomány utolsó létező zónájához. Ez úgy történik, hogy a zónabittérképen a keresés ezen utolsó zónától kiindulva kezdődik (22603. sor). A bittérkép bitjének azonosítója és a zónaazonosító közötti leképezést a 22615. sor valósítja meg oly módon, hogy az első adatzónához az első bitet rendeli hozzá.

Az állomány törlésekor zónáinak vissza kell kerülniük a bittérképre. Ezért a *free_zone* eljárás (22621. sor) felelős, amely mindössze annyit tesz, hogy meghívja a *free_bit* eljárást paraméterként átadva annak a zónabittérképet és a bitazonosítót. A *free_bit* eljárás használatos a szabad i-csomópontok megadására is; természetesen ekkor az i-csomópont bittérképet kell számára első paraméterként átadni.

A gyorsítótár kezelése blokkok írását és olvasását kívánja meg. Az *rw_block* (22641. sor) szolgál a lemez és a gyorsítótár közti egyszerű csatolófelületként. Egyszerre egy blokkot olvas be vagy ír ki. Ehhez hasonlóan létezik az *rw_inode*, amely egy i-csomópontot olvas be vagy ír ki.

Az állományban található következő eljárás az *invalidate* eljárás (22680. sor). Ezt egy fájlrendszer lecsatolásakor hívjuk meg például azért, hogy az éppen lecsatolt fájlrendszerhez tartozó összes, gyorsítótárban lévő blokkot eltávolítsuk onnan. Ha ez nem történne meg, az eszköz újbóli használatakor (egy másik hajlékonylemezzel) a fájlrendszer az új blokkok helyett a régieket találná meg.

Korábban említettük, hogy a *flushall* (22694. sor) eljárást, amely a legtöbb adat kiírásáért felelős, minden alkalommal meghívja a *get_block*, amikor egy megváltozott blokkot eltávolít az LRU-listából. A sync rendszerhívás szintén hívja ezt az eljárást egy adott eszközhöz tartozó összes, megváltozott információt tartalmazó átmeneti adattár lemezre írásához. A sync periodikusan aktivizálódik a frissítő démon által, és minden felcsatolt eszközre egyszer hívja a *flushall* eljárást. A *flushall* eljárás a gyorsítótárat lineáris tömbként kezeli, tehát minden módosult puffert megtalál, még azokat is, amelyek jelenleg használatban vannak és nincsenek benne az LRU-listában. A gyorsítótár minden pufferét megvizsgálja, és azokat, amelyek a kiürítendő eszközhöz tartoznak, a *dirty* pointertömbbe teszi. Ez a tömb globális, hogy ne a veremben legyen tárolva. Azután átadja ezt a tömböt az *rw_scattered* eljárásnak.

A MINIX 3-ban a lemezre írások funkcióját kivették az eszközvezérlőből, és az *rw_scattered* eljárás (22711. sor) teljes fennhatósága alá helyezték. Ez az eljárás megkap egy eszközazonosítót, egy olyan mutatót, amely az átmeneti adattárakat kijelölő mutatók tömbjére mutat, a tömb méretét jelentő számot és egy olyan jelzőt, amely azt mutatja, hogy írásról vagy olvasásról van-e szó. Első dolga a megszerzett tömb elemeinek a blokkazonosító számok szerinti csoportosítása, amivel azt éri el, hogy a tényleges írás vagy olvasás hatékony módon lesz majd végrehajtható. Ezután folytonos blokkok vektorát képezi, amelyet átad az eszközvezérlőnek a *dev_io* hívásával. Az eszközvezérlőnek nem kell további ütemezést végeznie. A modern lemezegységek esetén valószínű, hogy azok elektronikája tovább optimalizálja a kérések sorrendjét, de ezt nem látja a MINIX 3. Az *rw_scattered* eljárást csak a *flushall* hívja a *WRITING* jelzővel az előbb említett módon. Ezen a ponton a blokkazonosító számok eredete is egyszerűen megérthető. Ezek olyan átmeneti adattárak, amelyek korábban beolvasott blokkokból származó, azonban mostanára már megváltoztatott adatot tartalmaznak. Az egyetlen, az *rw_scattered* eljárást olvasásra felkérő hívás a *read.c* állományban található *rahead* eljárástól származik. Itt mindössze annyit kell tudnunk, hogy az *rw_scattered* hívása előtt a *get_block* eljárást hívjuk sorozatosan előolvasó módban, ily módon lefoglalva az átmeneti adattárak egy csoportját. Ezek az átmeneti tárak érvényes eszközszámok nélküli blokkazonosítókat tartalmaznak. Ez azonban semmilyen problémát nem jelent, mivel az *rw_scattered* hívásakor egy eszközt kijelölő paramétert is kap operandusai között.

Lényeges különbség van abban, ahogy egy eszköz meghajtója az *rw_scattered* eljárástól érkező olvasási kérelemre (az ugyanonnan származó írási kérelemhez viszonyítva) reagál. A néhány blokk kiírására irányuló kérelmet **teljes mértékben** ki kell szolgálni, míg a néhány blokk beolvasására irányuló kérelmet a különböző meghajtók eltérően kezelhetik annak függvényében, hogy az adott meghajtó szempontjából mi számít a leghatékonyabbnak. A *rahead* eljárás gyakran hívja olyan blokkokért is az *rw_scattered* eljárást, amelyekre valójában nincs is szüksége; a legjobb stratégia tehát: annyi blokkot átadni, amennyit különösebb erőfeszítés nélkül csak lehet, de nem bocsátkozni az egész eszközre kiterjedő őrült keresésbe, ami jelentős keresési idővel társulhat. A hajlékonylemez meghajtója például egy sáv határánál befejezheti a keresést, és sok más meghajtó is csupán az egymást követő blokkokat fogja beolvasni. A beolvasás befejezése után az *rw_scattered* eljárás az átmeneti adattárak eszközazonosító mezőjének kitöltésével jelöli meg az éppen beolvasott blokkokat.

Az 5.40. ábra utolsó függvénye az *rm_lru* elnevezésű (22809. sor). Ez az eljárás az LRU-listából távolít el egy blokkot. Ezt csupán a szintén ebben az állományban található, *get_block* eljárás használja, ezért *PUBLIC* helyett *PRIVATE* típusúnak definiáljuk. Ezzel egyúttal azt is sikerül elérnünk, hogy ezen eljárás az állományon kívüli eljárások számára rejtve maradjon.

Mielőtt a blokkgyorsítótárak tárgyalását befejeznénk, ejtsünk néhány szót ezek finomhangolásáról is. Az *NR_BUF_HASH* változónak 2 valamelyik hatványának kell lennie. Amennyiben értéke nagyobb, mint az *NR_BUFS* változó, a hasítóláncok átlagos hossza egynél kisebb lesz. Ha nagyszámú átmeneti adattár, illetve sok

hasítólánc számára van elegendő hely a memóriában, az *NR_BUF_HASH* változó értékét általában úgy választjuk meg, hogy az a 2 következő olyan hatványa legyen, amely az *NR_BUFS* változónál nagyobb. A szövegben 128 blokknak és 128 hasítóláncnak megfelelő beállítások fordulnak elő. Az optimális méret attól függ, hogyan használjuk a rendszert, hiszen ez határozza meg, mennyi információt is kell az átmeneti adattárakban tartanunk. A könyv CD-ROM mellékletében adott teljes MINIX 3-forráskód 1280 átmeneti adattár és 2048 hasítólánc értékre van beállítva. A tapasztalatok azt mutatják, hogy az átmeneti adattárak számának nagyobbra állításával sem javul lényegesen a MINIX 3 újrafordítása során a rendszer teljesítménye, tehát ennyi elegendő a fordítás minden menetében a bináris állományok tárolásához. Más jellegű munkáknál kevesebb átmeneti adattár használata is kielégítő lehet, illetve többet használva belőlük a rendszer teljesítménye akár javulhat is.

A CD-ROM-on található standard MINIX 3-rendszer pufferei több mint 5 MB RAM-ot foglalnak el. Egy másodlagos bináris is van, az *image_small,* amelyet úgy fordítottak, hogy csak 128 puffert használjon, és így egy kicsit több mint 0,5 MB kell a puffereknek. Ez a rendszer telepíthető csupán 8 MB RAM-mal rendelkező számítógépre. A standard verzió 16 MB-ot igényel. Egy kis ügyeskedéssel akár 4 MB vagy kisebb memóriával rendelkező gépre is telepíthető.

Az i-csomópontok kezelése

A blokkgyorsítótár nem az egyetlen olyan tábla, amelynek támogató eljárásokra van szüksége. Az i-csomópont tábla ugyanilyen. Eljárásainak, amelyeket az 5.41. ábrán soroltunk fel, nagy része működését tekintve a blokk-kezelés eljárásaihoz hasonló.

A *get_inode* eljárás (22933. sor) a *get_block* eljáráshoz hasonló. Ha a fájlrendszer valamely részének egy i-csomópontra van szüksége, annak megszerzése céljából a *get_inode* eljárást hívja meg. Ez először végignézi az *inode* táblát, hogy abban jelen van-e már a keresett i-csomópont. Ha igen, megnöveli az i-csomópont használati számlálójának értékét, és visszaküldi az i-csomópontot kijelölő mutatót. Magát a keresést a forráskód 22945–22955. sorok közötti része tartalmazza. Ha az i-csomópont nincs a memóriában, akkor az *rw_inode* eljárás meghívásával beolvasásra kerül.

Ha az adott i-csomópontot használó eljárás befejezte az i-csomóponton kijelölt műveletek elvégzését, a *put_inode* eljárás (22976. sor) meghívásával, amely egyben az *i_count* használati számláló értékét is csökkenti, visszaküldi az illető i-csomópontot. Amennyiben a számláló értéke zérus, ami azt jelenti, hogy a hozzá tartozó állomány már nincs használatban, az i-csomópont a táblából is eltávolítható. Ha időközben tartalma megváltozott, lemezre íródik.

Ha az *i_link* mező értéke zérus, vagyis semmilyen könyvtárelem nem mutat az állományra, az állomány összes zónája felszabadítható. Érdemes megjegyezni, hogy a használati számláló értékének és a kapcsolatok számának nullára csökkenése eltérő okokkal és következményekkel járó, két különböző esemény. Ha az

Eljárás neve	Tevékenység
get_inode	Egy i-csomópont memóriába töltése
put_inode	A továbbiakban már nem használandó i-csomópont visszaküldése
alloc_inode	Új i-csomópont foglalása (új állomány részére)
wipe_inode	Egy i-csomópont néhány mezője tartalmának törlése
free_inode	I-csomópont felszabadítása (állomány törlésekor)
update_times	I-csomópont időmezői tartalmának frissítése
rw_inode	I-csomópont memória és lemez közötti átvitele
old_icopy	I-csomópont tartalmának V1 (lemezen használt) formátumúvá történő átalakítása
new_icopy	V1 formátumban beolvasott i-csomópont tartalmának átalakítása
dup_inode	Annak jelzése, hogy az adott i-csomópontot még más processzus is használja

5.41. ábra. *Az i-csomópont kezelése során használatos eljárások*

i-csomópont egy adatcsövet jelent, minden zónát fel kell szabadítani még akkor is, ha a kapcsolatok száma esetleg nem nulla. Ez a helyzet például akkor következik be, amikor egy adatcsőből olvasó processzus felszabadítja az adatcsövet. Egyetlen processzus számára ugyanis nincs értelme adatcsövet fenntartani.

Új állomány létrehozásakor az *alloc_inode* eljárással (23003. sor) új i-csomópontot kell lefoglalni. A MINIX 3 az eszközök megnyitását csak olvasható állapotban engedélyezi, ezért az eszköz írhatóságának biztosítása érdekében ellenőrizzük a szuperblokkot. Új állomány létrehozásakor bármely i-csomópont megfelel, eltérően a zónáktól, ahol szem előtt tartjuk az ugyanahhoz az állományhoz tartozó zónák egymáshoz közel történő elhelyezését. Az i-csomópont bittérkép átvizsgálásához szükséges idő csökkentése céljából a szuperblokk azon mezője által nyújtott segítséget is kihasználjuk, amelyben az első, jelenleg még felhasználatlan i-csomópont helye van feljegyezve.

Az i-csomópont kiválasztása után a *get_inode* eljárás meghívásával a memóriában lévő táblába kerül, majd mezői, részben a programkód által (23038– 23044. sor) részben a *wipe_inode* eljáráson keresztül (23060. sor) feltöltődnek. A munka ezen speciális megosztására azért került sor, mert a *wipe_inode* eljárásra a fájlrendszer más részében az i-csomópont néhány mezőjének (de nem mindnek!) kinullázásakor van szükség.

Az állomány törlése után a hozzá tartozó i-csomópontot a *free_inode* eljárás (23079. sor) szabadítja fel. Ekkor csupán annyi történik, hogy az i-csomópont bittérkép megfelelő bitjének értéke nullára változik, és a szuperblokkban tárolt, az első használaton kívüli i-csomópontot kijelölő mező felfrissítődik.

Az *update_times* elnevezésű függvény (23099. sor) arra használatos, hogy a rendszerórától megkaphassuk a pontos időt, illetve megváltoztathassuk a frissítésre szoruló időmezők tartalmát. A stat és az fstat rendszerhívások szintén használják az *update_times* függvényt, így ezt *PUBLIC* típusúnak definiáljuk.

Az *rw_inode* eljárás (23125. sor) az *rw_block* eljáráshoz hasonlít. Egy i-csomópont lemezről memóriába töltésére szolgál, amely a következő lépéseken keresztül kerül kivitelezésre.

1. Kiszámítja, hogy a kívánt i-csomópontot melyik blokk tartalmazza.
2. A *get_block* eljárást felhasználva beolvassa ezt a blokkot.
3. Kiolvassa abból az i-csomópontot, amelyet ezután az *inode* táblába másol.
4. A *put_block* eljárást felhasználva visszaküldi a blokkot.

Az *rw_inode* eljárás működése valójában az itt vázoltnál egy kicsit bonyolultabb, így annak néhány kiegészítő függvényre is szüksége van. Először is, mivel a pontos idő lekérdezése kernelhívást igényel, abban az esetben, ha az i-csomópont időmezőinek megváltoztatására van szükség, az i-csomópont memóriában töltött ideje alatt csak beállítjuk az i-csomópont *i_update* mezőjének bitjeit. Amennyiben ezen mező értéke az i-csomópont lemezre írásának pillanatában nullától különböző, meghívjuk az *update_times* függvényt.

Másodszor, a MINIX történeti fejlődése is gondot jelent: a lemezen lévő i-csomópontok régi, *V1* fájlrendszernek megfelelő szerkezete az új, *V2* fájlrendszertől teljesen eltérő. Az átalakításokat két függvény, az *old_icopy* (23168. sor) és a *new_icopy* (23214. sor) végzi el. Az első a memóriában lévő i-csomópont információt alakítja át a *V1* típusú fájlrendszer által megkívánt formátumúra. A második egy ugyanilyen átalakítást végez a *V2 és V3* típusú fájlrendszer i-csomópontjaira. Mivel ezen függvények mindegyike csak ebben az állományban kerül hívásra, mindkettőjüket *PRIVATE* típusúként definiáljuk. Mind a két függvény oda és vissza is elvégzi az átalakítást (lemezről memóriába és memóriából lemezre).

A MINIX régebbi változatait olyan rendszereken hívták életre, amelyek az Intel processzoroktól eltérő bájtsorrendet használnak, és várhatóan a jövőben is alkalmazni fogják a MINIX 3-at ilyen gépeken. A saját lemezén minden rendszer az eredeti bájtsorrendet használja; a szuperblokk *sp->native* mezője azonosítja be a használt bájtsorrendet. Szükség esetén mind az *old_icopy,* mind a *new_icopy* függvények a *conv2,* illetve *conv4* függvényeket hívják meg a bájtsorrend felcseréléséhez. Nyilvánvaló, hogy mindaz, amit eddig mondtunk a MINIX 3-ra nem vonatkozik, mert az nem támogatja a V1 fájlrendszert alkalmazó lemezek használatát. Továbbá, a könyv írásáig senki sem akarta a MINIX 3-at eltérő bájtsorrendű gépre alkalmazni. De ezek a lehetőségek megmaradtak azokra a napokra, amikor a MINIX 3-at még sokoldalúbbá akarnák tenni.

A *dup_inode* eljárás (23257. sor) az i-csomópont használati számlálójának értékét növeli meg, és egy, már megnyitott állomány újbóli megnyitásakor kerül meghívásra. A második megnyitáskor már nincs szükség az állomány i-csomópontjának lemezről történő ismételt beolvasására.

A szuperblokk-kezelés

A szuperblokkokat és a bittérképeket kezelő eljárásokat a *super.c* állomány tartalmazza, amelyben mindössze öt eljárás (lásd 5.42. ábra) található.

Ha egy i-csomópontra vagy zónára van szükségünk, a korábban látottaknak megfelelően az *alloc_inode* vagy az *alloc_zone* eljárások valamelyikét hívjuk meg.

Eljárás neve	Tevékenység
alloc_bit	Egy bit lefoglalása a zóna- vagy i-csomópont bittérképben
free_bit	Egy bit felszabadítása a zóna- vagy i-csomópont bittérképben
get_super	Egy eszköz szuperblokktáblában való keresése
get_block_size	A használandó blokkméret lekérdezése
mounted	Annak meghatározása, hogy egy adott i-csomópont egy felcsatolt (vagy a gyökér-) fájlrendszerben van-e
read_super	Egy szuperblokk beolvasása

5.42. ábra. *A szuperblokk- és a bittérképkezelés során használatos eljárások*

A megfelelő bittérkép átvizsgálására ezek mindegyike az *alloc_bit* (23324. sor) eljárást használja. A vizsgálat a következő három, egymásba ágyazott ciklusból áll:

1. A külső ciklus az adott bittérkép blokkjain fut végig.
2. A középső ciklus egy blokk szavain fut végig.
3. A legbelső ciklus egy szó bitjein fut végig.

A középső ciklus azt vizsgálja meg, hogy az aktuális szó megegyezik-e egy olyan szóval, amely csupa egyest tartalmaz. Ha igen, az illető szóban sem szabad i-csomópont, sem szabad zóna nincsen, így a következő szó kerül vizsgálatra. Ha találunk egy olyan szót, amely ettől eltérő értékű, vagyis legalább egy darab 0 bit szerepel benne, működésbe lép a legbelső ciklus, hogy megtalálja a szabad (vagyis 0 értékű) bitet. Ha az összes blokk átvizsgálása után sem találunk ilyen szót, tehát sem szabad i-csomópontok, sem szabad zónák nincsenek, a *NO_BIT* (0) értéket kapjuk vissza. Az efféle keresések nagyon sok processzoridőt igényelnek, de az *alloc_bit* eljárásnak *kezdetben* átadott, az első használaton kívüli i-csomópontra vagy zónára mutató szuperblokkmezők éppen abban segítenek, hogy ezek a keresések csak rövid ideig tartsanak.

Egy bit felszabadítása sokkal egyszerűbb, mint annak lefoglalása, mert semmiféle keresésre nincs szükség. A *free_bit* eljárás (23400. sor) meghatározza, hogy a bittérkép melyik blokkja tartalmazza a felszabadítandó bitet, majd nullára állítja azt. Ehhez először meghívja a *get_block* eljárást, elvégzi a memóriában a kívánt bit nullázását, majd meghívja a *put_block* eljárást.

A *get_super* elnevezésű következő eljárás (23445. sor) a szuperblokktáblát vizsgálja meg és abban egy adott eszközt keres. Egy fájlrendszer felcsatolásakor ellenőrizni kell például, hogy a fájlrendszer nincs-e már felcsatolva. Ez az ellenőrzés úgy is elvégezhető, hogy a *get_super* eljárást az adott állományrendszerhez tartozó eszköz kikeresésére kérjük fel. Ha nem találja meg a keresett eszközt, a fájlrendszer még nincs felcsatolt állapotban.

A MINIX 3 esetén a fájlrendszer-kiszolgáló képes eltérő méretű blokkokkal is dolgozni, de egy partícióban csak egyféle használható. A *get_blokk_size* függvénynyel (23467. sor) lehet lekérdezni a blokkméretet. Az adott eszköz szuperblokktáblájában keres, és ha az eszköz fel van csatolva, akkor a blokkméretet adja vissza. Ha az eszköz nincs felcsatolva, akkor a *MIN_BLOKK_SIZE* értéket adja.

A következő függvény, a *mounted* (23489. sor) csupán egy blokkeszköz bezárásakor kerül meghívásra. Egy eszköz utolsó bezárásakor a gyorsítótárban hozzá tartozó adatokat rendszerint kidobáljuk onnan. Ez azonban egyáltalán nem kívánatos, ha az adott eszköz véletlenül egy felcsatolt fájlrendszerhez tartozik. A *mounted* függvényt az eszköz i-csomópontját kijelölő mutatóval hívjuk meg. Ha az eszköz a gyökéreszköz, vagy egy felcsatolt fájlrendszerhez tartozik, a függvény *TRUE* értékkel tér vissza.

Végezetül itt találjuk még a *read_super* eljárást (23509. sor) is. Ez részben az *rw_block*, illetve *rw_inode* eljárásokhoz hasonló, azonban csak olvasás esetén kerül meghívásra. A szuperblokk egyáltalán nem olvasódik be a blokkgyorsítótárba, hanem közvetlenül az eszközről olvasódik be az első 1024 bájt. A rendszer normális üzemeltetése során a szuperblokk lemezre írása nem szükséges. A *read_super* ellenőrzi azon fájlrendszer változatát, amelyről épp az előbb olvasott be, és szükség esetén elvégzi az átalakításokat. Így a szuperblokk memóriában lévő másolata még akkor is a szabványos szerkezettel fog rendelkezni, ha a lemezről történő beolvasás során esetleg még teljesen más szerkezettel vagy bájtsorrenddel rendelkezett.

A fájlleírók kezelése

A MINIX 3 a fájlleírók és a *filp* tábla kezeléséhez speciális eljárásokkal rendelkezik (lásd 5.39. ábr), amelyek a *filedes.c* állományban találhatók. Egy állomány létrehozásakor vagy megnyitásakor szükségünk van egy szabad állományleíróra és a *filp* táblában egy szabad rekeszre, amelyek megkeresését a *get_fd* eljárás (23716. sor) végzi el. Ezeket azonban általában nem jelöljük meg azonnal, hiszen ahhoz, hogy biztosak lehessünk a create és az open rendszerhívások sikeres végrehajtásában, még jó néhány egyéb ellenőrzést is el kell végeznünk.

A *get_filp* eljárást (23761. sor) használjuk annak eldöntésére, hogy az állományleíró elérhető-e. Ha igen, a *get_filp* eljárás ennek *filp* mutatóját adja vissza.

Az utolsó eljárás a *find_filp* eljárás (23774. sor). Annak eldöntéséhez szükséges, vajon egy processzus nem egy megszakadt adatcsőbe ír-e (ami azt jelenti, hogy az adatcső egy másik processzus számára olvasás céljából nincs megnyitva). A *filp* táblában elvégezve egy teljes körű keresést, beazonosítja a lehetséges olvasókat. Ha egyet sem talál, az adatcső megszakad és az írás meghiúsul.

Fájlzárolás

A POSIX rekordzároló függvényeit az 5.43. ábrán láthatjuk. Az állomány egy részét az fcntl rendszerhívás végrehajtásakor, kiválasztva az *F_SETLK* vagy az *F_SETLKW* kérések közül a megfelelőt, zárolhatjuk az írás és olvasás, vagy csak az írás elől. Azt, hogy az adott állomány egy része zárolt állapotban van-e, az *F_GETLK* kérés használatával lehet eldönteni.

A *lock.c* állományban mindössze két függvény található. Az fcntl rendszerhívás a *lock_op* függvényt (23820. sor) az 5.43. ábrán bemutatott műveletek valamelyi-

Kérés kódja	Jelentés
F_SETLK	Az állomány egy adott részének írás és olvasás elől való zárolása
F_SETLKW	Az állomány egy adott részének írás elől való zárolása
F_GETLK	Annak meghatározása, hogy az állomány egy adott része zárolás alatt áll-e

5.43. ábra. *A POSIX rekordzároló műveletei. Az egyes műveleteket az* fcntl *rendszerhívás használatával aktiválhatjuk*

kének a kódjával hívja meg. Ez a kiválasztott terület elérhetőségét illetően végez el néhány hibaellenőrzést: ha egy új zárolást hozunk létre, az nem ütközhet egy már meglévő zárolással, illetve egy létező zárolásból, annak eltávolításakor nem jöhet létre két új zárolás. Bármilyen zárolás feloldásakor az ugyanebben az állományban található másik függvény, a *lock_revive* (23964. sor) kerül meghívásra. A *lock_revive* függvény működésbe hozza az eddigi zárolások miatt felfüggesztett processzusok mindegyikét. Ez a módszer azonban egy kompromisszumon alapszik; annak pontos meghatározása, hogy mely processzusok vártak egy adott zárolás megszüntetésére, további programkódot igényelne. Azokat a processzusokat, amelyek újraéledésük után még mindig zárolt állományra várakoznak, újra felfüggesztjük. Ez a stratégia azon alapul, hogy a zárolást csak ritka esetekben használjuk. Amennyiben a MINIX 3 alatt egy sokfelhasználós központi adatbázis kiépítése lenne a feladatunk, a fenti stratégiát más formában lenne kívánatos megvalósítanunk.

A *lock_revive* függvényt hívjuk meg abban az esetben is, amikor egy zárolás alatt lévő állományt bezárunk. Ez olyankor történhet meg például, amikor egy zárolt állományon még műveleteket végző processzust szüntetünk meg.

5.7.3. A főprogram

A fájlrendszer központi ciklusa a *main.c* állományban található meg a 24040. sorral kezdődően. Szerkezetét tekintve nagyon hasonló a processzuskezelőnél és az I/O-eszközkezelőknél megismert központi ciklusokhoz. A *get_work* hívása a következő kérést tartalmazó üzenet megérkezését várja (hacsak egy olyan processzszus folytathatóvá nem válik, amely korábban egy adatcsőben vagy egy terminálon került felfüggesztésre). Ezenkívül a *who* globális változóhoz hozzárendeli a hívó processzustáblája megfelelő rekeszének azonosítóját, illetve a *call_nr* globális változóhoz a végrehajtandó rendszerhívás azonosítóját.

A központi ciklusba való visszatérés után három jelző beállítása történik meg: az *fp* kijelöli a hívó processzustáblájának megfelelő rekeszét, a *super_user* megadja, hogy a hívó a szuperfelhasználó-e, vagy sem. A bejelentésüzeneteknek van a legnagyobb prioritása, a *SYS_SIG* üzenetet ellenőrzi először, hogy megtudja, a rendszer leállítás állapotában van-e. A második legnagyobb prioritású a *SYS_ALARM,* ami azt jelenti, hogy a fájlrendszer által beállított időtartam lejárt. A *NOTIFY_MESSAGE* azt jelenti, hogy egy eszközvezérlő készen áll és *dev_status* állapotba került. Ezután következik a fő mutatvány – meghívjuk a rendszerhívást

elvégző eljárást. Ezt a *call_vecs* elnevezésű eljárásokat kijelölő mutatók tömbjéből a *call_nr* értékét mutatóként használva választjuk ki.

A vezérlés központi ciklusba történő visszakerülése után, ha a *dont_reply* jelzőt korábban megfelelően állítottuk be, nem lehetséges válasz küldése (például egy processzust azért függesztettünk fel, mert üres adatcsőből próbált meg olvasni). Egyébként a válaszküldésen van a sor a *reply* (24087. sor) hívásával. A központi ciklus utolsó utasítását oly módon szerkesztették meg, hogy az érzékeli egy állomány szekvenciális beolvasását, és a rendszer teljesítményének javítása érdekében a gyorsítótárba olvassa annak következő blokkját, még mielőtt erre külön utasítást kapna.

Két másik függvény közvetlenül kapcsolódik a fájlrendszer központi ciklusához. A *get_work* eljárás (24099. sor) azt ellenőrzi, hogy egy korábban felfüggesztett eljárás nem került-e időközben folytatható állapotba. Ha igen, ez az eljárás elsőbbséget élvez az új üzenetekkel szemben. A fájlrendszer csak akkor hívja új üzenetért az operációs rendszer központi részét (24124. sor), ha már nincs több elvégzendő belső feladat. Néhány sorral továbblépve találjuk a *reply* (24159. sor) hívást, amely azután hívódik, hogy a rendszerhívás befejeződött, akár sikeresen, akár nem. Ez üzenetet küld vissza a hívónak. A processzust leállíthatja egy szignál, ezért az operációs rendszer központi része által visszaküldött állapotkódot figyelmen kívül hagyjuk. Ebben az esetben egyébként sincs mit tenni.

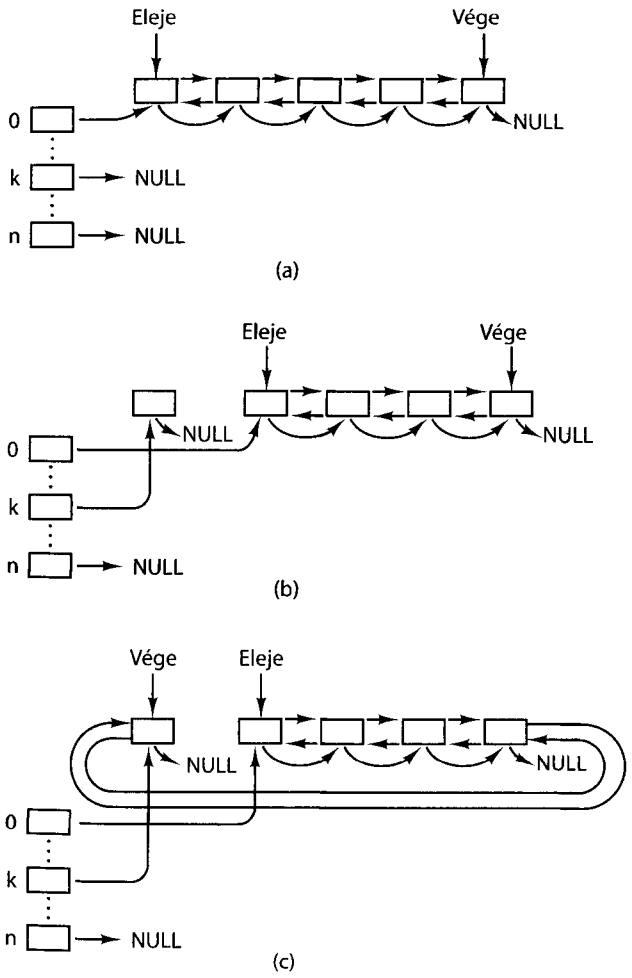
A fájlrendszer kezdeti beállítása

A *main.c* állomány fennmaradó része olyan függvényekből áll, amelyek a rendszer elindításakor használatosak. A legfontosabb szerepet az *fs_init* játssza, amely még a fájlrendszer központi ciklusának megkezdése előtt hívódik meg. A 2. fejezetben a processzusok ütemezésének vizsgálatakor láttuk a 2.43. ábrán a MINIX 3 indításakor a porcesszusok kezdeti ütemezését. A fájlrendszer alacsonyabb prioritással lett beütemezve a sorban, mint a processzuskezelő, tehát biztosak lehetünk, hogy indításkor a processzuskezelőnek esélye van arra, hogy előbb fusson, mint a fájlrendszer. A 4. fejezetben vizsgáltuk a processzuskezelő inicializálását. A PM felépíti a processzustáblát, hozzáadva magát és a betöltési memóriakép más processzszusait, aztán mindegyikről üzenetet küld a fájlrendszernek, így az FS inicializálni tudja a megfelelő elemeket a fájlrendszer FS részében. Most már folytathatjuk a tevékenység második részével.

Amikor a fájlrendszer elindul, azonnal belép saját ciklusába az *fs_init*-ben a 24189–24202 sorokban. Az első utasítás a ciklusban a *receive* hívása, amely lekérdezi a PM-nek a 18235. sorban meghívott *pm_init* inicializáló függvény által küldött üzenetét. Minden üzenet tartalmazza a processzusszámot és PID-et. Az elsőt indexként használja a fájlrendszer a processzustáblázathoz, a másodikat pedig elmenti az *fp_pid* mezőben. Ezután a szuperfelhasználó valós és effektív uid-jét, gidjét és a csupa 1-est tartalmazó *umask*-ot állítja be. Amikor a NONE szimbolikus érték érkezik az üzenet processzusszám mezőjében, akkor a ciklus befejeződik, és üzenetet küld vissza a processzuskezelőnek, jelezve, hogy minden rendben.

Ezután a fájlrendszer saját inicializálása befejeződik. Először fontos konstansok értékének érvényességét ellenőrzi. Ezután több függvényhívás következik, amelyek inicializálják a blokkgyorsítótárat, az eszközáblázatot, betöltik a RAM-lemezt, ha kell, és betöltik a gyökéreszköz szuperblokkját. Ettől a pillanattól a gyökéreszköz elérhető és egy ciklus indul a processzustábla FS részének ellenőrzésére, és a betöltési memóriaképbeli valamennyi processzus felismeri a gyökérkönyvtárat, és azt használja munkakönyvtárként (24228–24235. sor).

Az *fs_init,* miután befejeződött a processzuskezelővel a kommunikációja, először a *buf_pool* elnevezésű függvényt hívja meg (24132. sor). Ez a blokkgyorsítótár által használt láncolt listákat hozza létre. Az 5.37. ábra a blokkgyorsítótár alapállapotát mutatja, amikor mind az LRU-listában, mind a hasítóláncban lévő blokkok össze vannak kapcsolva. A megértéshez segítségül szolgálhat, ha megvizsgáljuk, hogyan is alakul ki az 5.37. ábrán bemutatott állapot. A gyorsítótár *buf_pool* függvény általi kezdeti beállítása után az összes átmeneti adattár azonnal az LRU-listába kerül, és amint azt az 5.44.(a) ábrán vázoltuk, mindegyikőjük a 0. hasítólánchoz kapcsolódik hozzá. Ha szükségünk van egy átmeneti adattárra, illetve amíg az használatban van, az 5.44.(b) ábrán bemutatott helyzethez jutunk. Itt azt láthatjuk, hogy az egyik blokk az LRU-listából eltávolításra került, és most egy másik hasítóláncban található.



5.44. ábra. *A blokkgyorsítótár inicializálása. (a) Mielőtt még bármely átmeneti adattár használatban lenne. (b) Egy blokk kérése után. (c) A blokk felszabadítása után*

A blokkokat rendszerint azonnal felszabadítjuk, amelyek ezután nyomban visszakerülnek az LRU-listába. Az 5.44.(c) ábra mutatja be a blokk LRU-listába való visszakerülése utáni helyzetet. Annak ellenére, hogy ezt a blokkot már nem használjuk, szükség esetén a már kiolvasott információt belőle újra kiolvashatjuk, így a blokkot a hasítóláncban megtartjuk. A rendszer huzamosabb működése során majdnem minden blokk felhasználásra kerül és a különböző hasítóláncok között lesz véletlenszerűen szétszórva. Az LRU-lista ekkor az 5.37. ábrának megfelelően fog kinézni.

A *buf_pool* után a *build_dump* függvény hajtódik végre, amit majd később vizsgálunk az eszközkezelő más függvényeivel együtt. Ezután a *load_ram* hívódik meg, amely a következőkben tárgyalandó *igetenv* (24241. sor) függvényt használja. Ez a függvény az indítóparaméter nevét kulcsként használva lekérdezi a kerneltől a numerikus eszközazonosítót. Ha egy működő MINIX 3-rendszerben kiadjuk a *sysenv* parancsot, akkor azt látjuk, hogy kimenetként a *sysenv* az eszközöket numerikusan jeleníti meg, például

```
rootdev=912
```

A fájlrendszer számokkal azonosítja az eszközöket. A szám értékét a 256 × *major* + *minor* képlet adja, ahol *major* a főeszközszám, *minor* pedig a mellékeszközszám. A példában a *major* értéke 3, a *minor* értéke 144, ami a */dev/c0d1p0s0* eszköznek felel meg, amely szokásos telepítési helye a MINIX 3-nak két lemezmeghajtós gépeken.

A *load_ram* függvényhívás (24260. sor) memóriahelyet foglal a RAM-lemez számára, és betölti ide a gyökérkönyvtárat, ha az indítóparaméter ezt előírja. Ez az *igetenv* függvényt használja a *rootdev, ramimagedev* és *ramsize* paraméterek lekérdezésére, amelyek az indítási környezetben lettek beállítva (24278–24280. sor). Ha az indítóparaméterek között szerepel a

```
rootdev=ram
```

akkor a *ramimagedev* nevű eszközről a RAM-lemezbe másolja a gyökérfájlrendszert, blokkról blokkra, figyelmen kívül hagyva a fájlrendszer adatszerkezetét. Ha a *ramsize* indítóparaméter nagyobb, mint a betöltési eszköz fájlrendszerének mérete, akkor is a kért terület foglalódik le, és a RAM-lemez fájlrendszer méretét megnöveli a paraméter szerinti méretre (24404–24420. sor). Ez az egyetlen alkalom, amikor a fájlrendszer valaha is szuperblokkot ír, de csakúgy, mint a szuperblokk olvasásakor, most sem kerül gyorsítótárba, hanem a *dev_io* közvetlenül az eszközre írja.

Két dolgot érdemes itt megemlíteni. Az első a 24291-től 24307. sorokban lévő programrészlet, amely a CD-ROM-ról történő indítást kezeli. Erre használatos a *cdprobe* függvény, amelyet nem tárgyaltunk. Az érdeklődő olvasó megnézheti az *fs/cdprobe.c* programot a mellékelt CD-n, vagy a weben. A második az, hogy függetlenül a MINIX 3 blokkméretétől, az indítóblokk mindig 1 KB méretű, és a szuperblokk az eszköz második 1 KB-os blokkjától töltődik be. Minden más megoldás bonyolult lenne, mert a blokkméret csak a szuperblokk betöltése után válik ismertté.

Ha a *ramsize* értéke nullától különböző, a *load_ram* függvény az üres RAM-lemez számára helyet foglal. Ebben az esetben a RAM-lemez mindaddig nem használható fájlrendszerként, amíg azt az *mkfs* paranccsal erre elő nem készítettük, hiszen a fájlrendszer egyetlen struktúrája sem másolódott rá. Ugyanakkor egy ilyen RAM-lemez, ha a fájlrendszer fordításakor arra lehetőséget biztosítottunk, felhasználható másodlagos gyorsítótárként.

A *main.c* állomány utolsó függvénye a *load_super* függvény (24426. sor). Ez állítja be a szuperblokktábla kezdeti értékét és olvassa be a gyökéreszköz szuperblokkját.

5.7.4. Egyedi fájlokon végzett műveletek

Ebben a szakaszban azokat a rendszerhívásokat tekintjük át, amelyek (a könyvtárakon végzett műveletekkel szemben) az egyedi fájlokon – egyidejűleg csak egy példányon – hajtanak végre valamilyen műveletet. Azzal kezdjük, hogy megvizsgáljuk, hogyan történik egy fájl létrehozása, megnyitása, valamint bezárása. Ezután részletesen megvizsgáljuk azt a technikát, amellyel egy fájl írása és olvasása történik, majd áttekintjük, miben különböznek az adatcsövek, valamint az ezeken végzett műveletek a közönséges fájlokon végzett műveletektől.

Fájlok létrehozása, megnyitása és lezárása

Az *open.c* állomány hat rendszerhívás forráskódját tartalmazza. Ezek sorrendben a következők: create, open, mknod, mkdir, close és lseek. A create és az open rendszerhívásokat együtt, majd ezután a többieket külön-külön vizsgáljuk meg.

A Unix régebbi változataiban a create és az open hívások különböző célok megvalósítására szolgáltak. Egy nem létező állomány megnyitása hibát eredményezett; egy új állományt minden esetben először azzal a create hívással kellett létrehozni, amely egyben a már létező állományok hosszának nullára csökkentését is végezte. A POSIX-rendszerben nincs szükség a két elkülönülő hívásra. A POSIX-beli open rendszerhívás egyaránt lehetővé teszi új állományok létrehozását, valamint már létező állományok hosszának nullára csökkentését, így a create hívás mindössze az open hívás egy lehetséges alműveletét szimbolizálja, és csupán a régebbi programokkal való kompatibilitás fenntartása végett van rá szükség. A create, valamint az open hívásokat kezelő eljárások a *do_creat* (24537. sor) és a *do_open* (24550. sor)

eljárások. (Akárcsak a processzuskezelőnél, a fájlrendszerben is használatos az a megállapodás, hogy az xxx rendszerhívást a *do_XXX* névre hallgató eljárás valósítja meg.) Egy állomány megnyitása vagy létrehozása a következő három lépés végrehajtását jelenti.

1. Az i-csomópont megtalálása (annak lefoglalása, illetve amennyiben új állományról van szó, kezdeti értékekkel való feltöltése).
2. A megfelelő könyvtárbejegyzés megtalálása vagy létrehozása.
3. Az állományhoz tartozó állományleíró beállítása és visszaküldése.

A create és az open hívások mindegyike két dolgot tesz meg: megszerzi az állomány nevét, és meghívja azt a *common_open* elnevezésű eljárást, amely a két hívás végrehajtása során felmerülő közös feladatok elvégzéséért felelős.

A *common_open* eljárás (24573. sor) működését azzal kezdi, hogy ellenőrzi, vajon a szabad állományleíró és a *filp* tábla rekeszei rendelkezésre állnak-e. Ha a hívó eljárás egy új állomány létrehozását kérte (az *O_CREAT* bit korábban történő egyesre állításával), akkor a 24594. sorban található *new_node* eljárás kerül meghívásra. A könyvtárbejegyzés létezése esetén a *new_node* eljárás a már létező i-csomópontot kijelölő mutatót adja vissza. Ellenkező esetben létrehozza mind az új könyvtárbejegyzést, mind annak i-csomópontját. Ha az i-csomópont valamilyen okból nem hozható létre, a *new_node* eljárás beállítja az *err_code* globális változót. Egy hibakód azonban nem minden esetben jelent egyben hibát is. Ha a *new_node* egy már létező állományt talál, a hibakód ezen állomány létezésére utal, tehát ebben az esetben ez elfogadható hiba (24597. sor). Ha az *O_CREAT* bitet nem állítottuk egyesre, akkor egy másik módszerrel, a *path.c* állományban lévő *eat_path* függvénnyel (ezt a későbbiekben még részletesen tárgyaljuk) keresünk i-csomópontot. Ennél a pontnál a következő fontos dolgot kell megértenünk: ha nem találunk, vagy nem sikerül létrehoznunk egy i-csomópontot, a *common_open* eljárás végrehajtása, még mielőtt elérné a 24606. sort, egy hibaüzenettel megszakad. Ellenkező esetben az állományleíró kijelölésével és a *filp* tábla egy rekeszének igénylésével folytatódik a végrehajtás. Ezt követően, ha éppen új állományt hoztunk létre, a 24612. és 24680. sorok közötti rész egyszerűen kimarad a végrehajtásból.

Ha az állomány nem új, a fájlrendszernek meg kell vizsgálnia, miféle állományról van szó, milyen az állomány védettségi állapota stb., hogy eldönthesse, megnyitható-e az illető állomány. A 24614. sorban szereplő *forbidden* hívása először egy általános vizsgálatot végez el az *rwx* biteken. Ha az állomány egy teljesen közönséges állomány és a *common_open* eljárást az *O_TRUNC* bit 1 értéke mellett hívtuk meg, az állomány hossza nullára állítódik majd (24620. sor) újból a *forbidden* eljárás kerül hívásra, jelen esetben azért, hogy megbizonyosodjunk az állomány írhatóságáról. Ha az engedélyek ezt lehetővé teszik, a *wipe_inode* és *rw_inode* meghívásával az i-csomópont új kezdeti értékeket kap és lemezre íródik. Az egyéb típusú állományok esetében (könyvtárak, speciális állományok, valamint nevesített adatcsövek) elvégezzük rajtuk a megfelelő vizsgálatokat. Egy eszköz esetén például a 24640. sorban (a *dmap* struktúrát felhasználva) az adott eszköz megnyitása céljából meghívjuk a megfelelő rutint, míg egy névvel ellátott adatcső

esetén a *pipe_open* eljárás hívására kerül sor (24646. sor), és az adatcsövekre érvényes, különböző vizsgálatokat hajtunk rajta végre.

A *common_open* eljárás forráskódjának nagy része, akárcsak a fájlrendszer más eljárásai esetén, nem más, mint a hibákat és illegális kombinációkat kiszűrő program. Ez nem valami hangzatos dolog, azonban egy hibáktól mentes, megbízható fájlrendszer működéséhez elengedhetetlen. Ha valami nincs rendben, a már korábban lefoglalt állományleíró és a *filp* tábla rekesze, az i-csomópontotal együtt, szabaddá válik (24683–24689. sor). Ebben az esetben a *common_open* negatív értékkel tér vissza, ami hibára utal. Ha minden rendben van, az állományleíró, amely egy pozitív szám, kerül visszaadásra.

Ezzel elérkeztünk ahhoz a ponthoz, hogy a *new_node* eljárás (24697. sor) működését egy kicsit részletesebben is megvizsgálhassuk. Ez foglalja le az i-csomópontot és juttatja el a fájlrendszerbe a create és az open rendszerhívások számára szükséges elérési utat. A később tárgyalásra kerülő mknod és mkdir rendszerhívások szintén használják ezt az eljárást. A 24711. sor utasítása a legutolsó könyvtárnévvel bezárólag kiértékeli az elérési utat (vagyis összetevőnként megvizsgálja azt); a három sorral később megjelenő *advance* eljárás pedig azt vizsgálja meg, hogy az elérési út utolsó összetevője megnyitható-e. Például az

```
fd = creat("/usr/ast/foobar", 0755);
```

utasítás végrehajtásakor a *last_dir* megpróbálja a táblákba tölteni a */usr/ast* könyvtárhoz tartozó i-csomópontot, illetve visszaadni az ezt kijelölő mutatót. Ha az állomány nem létezik, rövidesen szükségünk lesz erre az i-csomópontra ahhoz, hogy a *foobar* állományt a könyvtár tartalmához adhassuk. Az összes egyéb, állományokat hozzáadó vagy törlő rendszerhívás szintén a *last_dir* eljárást használja az elérési útban szereplő utolsó könyvtár megnyitásához, mielőtt ott bármit is csinálna.

Ha a *new_node* eljárás észleli, hogy nem létezik az adott állomány, a 24717. sorban meghívja az *alloc_inode* eljárást, és egy mutató visszaküldésével egy új i-csomópontot foglal le és tölt be. Ha nincs több szabad i-csomópont, a *new_node* végrehajtása meghiúsul, és a *NIL_NODE* változóval tér vissza.

Amennyiben volt még lefoglalható i-csomópont, a végrehajtás a 24727. sornál az i-csomópont néhány mezőjének kitöltésével, lemezre való kiírásával és az állomány nevének a célkönyvtárban történő elhelyezésével (24732. sor) folytatódik. Látjuk, hogy a fájlrendszernek folyamatos hibavizsgálatot kell végeznie, és egy hiba felbukkanása esetén mindazokat az erőforrásokat – például i-csomópontok vagy blokkok –, amelyek kapcsolatban állnak a hibával, szabaddá kell tennie. Ha a MINIX 3-at ilyenkor, amikor például kifogyott az i-csomópontból, hagynánk összezavarodni ahelyett, hogy visszaállítaná az adott hívás előtti állapotot, és a hívónak visszaküldene egy hibakódot, a fájlrendszer lényegesen egyszerűbb lenne.

Amint azt már korábban említettük, az adatcsövek speciális kezelést igényelnek. Ha egy adatcső számára nem áll legalább egy olvasó/író pár rendelkezésre, a *pipe_open* eljárás (24758. sor) felfüggeszti a hívót. Ellenkező esetben viszont azt a *release* eljárást hívja meg, amely a processzustáblában az adatcsőhöz kapcsolódó,

felfüggesztett állapotban lévő processzusokat próbálja beazonosítani. Ha sikerrel jár, az összes processzus végrehajtása folytatódik.

Az mknod rendszerhívást a *do_mknod* eljárás (24785. sor) kezeli, amely a *do_creat* eljárásra hasonlít azzal a különbséggel, hogy ez először létrehozza az i-csomópontot, majd egy könyvtárbejegyzést is hozzárendel. A munka nagy részét valójában a 24797. sorban meghívott *new_node* végzi el. Ha az i-csomópont már létezik, hibaüzenetet kapunk vissza. Ez ugyanaz a hibaüzenet, amely a *new_node* eljárás *common_open* által történő hívásakor elfogadható hibát eredményezett; most azonban a hibaüzenet visszajut a hívóhoz, amely ennek megfelelően fog cselekedni. Itt nem szükséges az a lépésről lépésre történő vizsgálat, amelyet a *common_open* eljárásnál láttunk.

Az mkdir rendszerhívás kezeléséért a *do_mkdir* eljárás (24805. sor) a felelős. Éppúgy, mint az előzőkben tárgyalt rendszerhívásoknál, a *new_node* eljárás itt is lényeges szerepet játszik. Az állományoktól eltérően a könyvtárak mindig rendelkeznek kapcsolatokkal, és sohasem teljesen üresek, hiszen bármely könyvtár a létrehozásától kezdve két elemet mindig tartalmaz: a . és a .. elemeket, amelyek magára a könyvtárra, illetve az eggyel feljebb lévő könyvtárra utalnak. Azt, hogy egy állomány hány darab kapcsolattal rendelkezhet, a *LINK_MAX* változó szabja meg (ez az *include/limits.h* definíciós állományban mint *SHRT_MAX*, 32767 értékre van beállítva a standard Intel 32 bites rendszereken futó MINIX 3 számára). Mivel egy adott könyvtárban az eggyel feljebb elhelyezkedő könyvtárra történő hivatkozás egy odamutató kapcsolat, a *do_mkdir* első dolga annak ellenőrzése, hogy lehetséges-e az eggyel feljebb elhelyezkedő könyvtárban további kapcsolatok létrehozása (24819. és 24820. sor). Az ellenőrzés hibaüzenet nélküli végrehajtása után meghívja a *new_node* eljárást, amelynek sikeres befejezése esetén megjelennek a . és a .. könyvtárbejegyzések (24841. és 24842. sor). Ezen műveletek egyike sem bonyolult, ennek ellenére előfordulhatnak zavarok (például betelt a lemez). A dolgok teljes összezavarodását elkerülendő fel kell tehát arra is készülnünk, hogy vissza tudjuk állítani a processzus megkezdése előtti állapotot, ha időközben az adott processzus befejezhetetlenné válna.

Egy állomány bezárása egyszerűbb, mint megnyitása. Ezt a *do_close* eljárás (24865. sor) végzi el. Az adatcsövek és speciális állományok némi odafigyelést igényelnek, azonban a közönséges állományoknál az említett eljárásnak szinte csak annyi teendője van, hogy csökkentse a *filp* számláló értékét, ellenőrizze, hogy ezen érték nem nulla-e, illetve, ha ez bekövetkezne, a *put_inode* eljárás használatával az i-csomópontot visszaküldje, utolsó lépésként pedig eltávolítsa a zárolásokat, és újraélessze mindazokat a processzusokat, amelyek eddig felfüggesztett állapotukban az adott állományon lévő zárolás feloldására várakoztak.

Jegyezzük meg: egy i-csomópont visszaküldése azt jelenti, hogy számlálója az *inode* i-csomópont táblában nullára csökken, vagyis végső soron törlődik a táblából. Ennek a műveletnek azonban semmi köze az i-csomópont felszabadításához (ami azt jelenti, hogy a bittérképen úgy állítjuk be a megfelelő bitet, hogy az ezután az i-csomópont újrahasználhatóságát mutassa). Az i-csomópont csak akkor szabadul fel, ha a hozzá tartozó állományt az összes lehetséges könyvtárból eltávolítottuk.

Az *open.c* állomány utolsó eljárása a *do_lseek* eljárás (24939. sor), amelyet egy pozicionálás megtörténte után az új fájlpozíció értékének beállításához hívunk meg. A fájlpozíció beállítására irányuló határozott kérés nem egyeztethető össze a folyamatos beolvasással, ezért a 24968. sorban az előreolvasás nem engedélyezett.

Fájl olvasása

Egy állomány megnyitása után írható vagy olvasható állapotba kerül. Az írás és az olvasás során egyaránt sok függvényt használunk. Ezek a *read.c* állományban találhatók. Először ezeket tárgyaljuk, majd a *write.c* elnevezésű következő állományra lépünk tovább, amely mindössze a speciálisan csak íráskor használt függvények forráskódját tartalmazza. Az írás és olvasás művelete számos ponton különbözik egymástól, ahhoz azonban elég sok hasonlóság van közöttük, hogy a *do_read* eljárásnak (25030. sor) egy jelző *READING*-re való beállítása után mindössze a közös *read_write* eljárást kelljen meghívnia. A következő részben a *do_write* hasonlóan egyszerű működését látjuk majd.

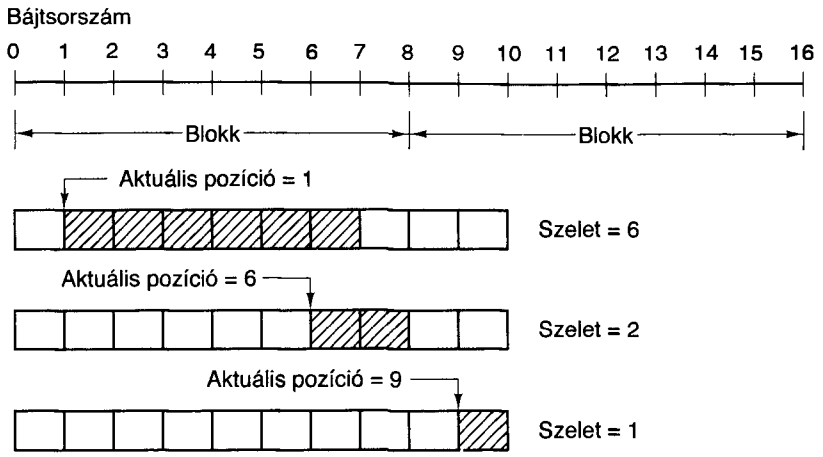
A *read_write* eljárás a 25038. sornál kezdődik. A 25063. és 25066. sorok között egy olyan speciális forráskódrész található, amelyre a processzuskezelőnek van szüksége ahhoz, hogy a fájlrendszerrel teljes szegmenseket tudjon a felhasználói területre töltetni. A rendes hívások feldolgozása a 25068. sorban kezdődik. Először néhány érvényesség-ellenőrzés történik (például véletlenül nem olyan állományokból próbálunk-e olvasni, melyek csak írásra vannak megnyitva), majd néhány változó kap kezdeti értéket. Mivel a karakterspeciális fájlokból történő olvasás nem a blokkgyorsítótáron keresztül történik, ezeket a 25122. sorban kiszűrjük.

A 25132. és 25145. sorok közti ellenőrzések csak írás során kerülnek végrehajtásra, és csak azokra az állományokra van hatásuk, amelyek mérete nagyobbá válhat, mint amennyi az írás alatt álló eszközre ráfér, vagy az olyan írásokra, amelyek az állomány vége *utáni* hozzáírás során az állományban egy lyuk megjelenését eredményezik. Amint azt a MINIX 3 áttekintése során már megismertük, a zónánkénti több blokk jelenléte olyan problémákat hozhat felszínre, amelyek speciális kezelést igényelnek. Az adatcsövek szintén speciálisak, így ezek is ellenőrzésre szorulnak.

Az olvasási művelet szíve, legalábbis közönséges állományok esetében, a 25157. sorban kezdődő ciklus. Ez a ciklus tördeli olyan darabokra a kért adatot, hogy a darabok mindegyikének mérete éppen egy szimpla lemezblokk méretével egyezzék meg. Egy vizsgált darab a pillanatnyi helyen kezdődik, és addig terjed, amíg a következő feltételek valamelyike nem teljesül.

1. Beolvastuk az összes bájtot.
2. Elértük a blokk határát.
3. Az állomány végébe ütköztünk.

Ezen feltételek olyanok, hogy valamelyik kielégítéséhez bármely darabnak mindössze egy blokkból kell állnia. Az 5.45. ábrán három példát mutatunk arra, hogyan



5.45. ábra. *Három példa annak szemléltetésére, hogyan is történik egy 10 bájtos állomány esetén az első darab méretének meghatározása. A blokkméret 8 bájt, a kért bájtok száma 6. A kívánt darabot a vonalkázott terület jelenti*

történik egy darab hosszának megállapítása 6, 2, illetve 1 bájt hosszúságú darabok esetén. A tényleges számolás a 25159. és a 25169. sorok között valósul meg.

Az adott darab beolvasását valójában az *rw_chunk* eljárás végzi. A vezérlés visszaszerzése után különböző számlálók és mutatók értékét növeli meg, majd elkezdődik a következő iteráció. Ha a ciklus a végére érve befejeződik, az állomány vége és más változók értékei (például adatcsőmutatók) felfrissíthetők.

Végezetül az előreolvasás során az az i-csomópont, amelyből, illetve az a pozíció, ahonnan kezdve az olvasás történik, globális változókban raktározódik, így a felhasználónak küldött válaszüzenet után a fájlrendszer elkezdhet a következő blokk beolvasásán dolgozni. A fájlrendszer sok esetben a következő lemezblokkra várva felfüggesztésre kerül. Ezen idő alatt a felhasználói processzus elkezdheti az imént beolvasott adat feldolgozását. A feldolgozás és az I/O-műveletek közötti időbeli átfedés lényegesen javíthat a rendszer teljesítményén.

Az *rw_chunk* eljárás (25251. sor) megkapja az i-csomópontot és a fájlpozíciót, ezeket valódi fizikai blokkazonosítókká alakítja, majd kéri a blokk (illetve annak egy része) felhasználói területre történő átvitelét. A relatív fájlpozíció fizikai lemezcímmé történő átváltását az a *read_map* függvény végzi el, amely tud az i-csomópontokról és a közvetett blokkokról. A 25280. sorban található *b* és a 25281. sorban lévő *dev* változók tartalmazzák közönséges állományok esetén a fizikai blokk- és eszközazonosítót. A 25303. sorban a *get_block* hívásával utasítjuk a gyorsítótár kezelőjét az adott blokk megkeresésére és szükség esetén beolvasására. A 25295. sorban a *rahead* gyorsítótár-kezelő hívása biztosítja a blokk beolvasását a gyorsítótárba.

Miután birtokunkban van a blokkot kijelölő mutató, a 25317. sorban a *sys_vircopy* kernelhívás gondoskodik a blokk kívánt részének a felhasználói területre történő

átviteléről. Ezután a *put_block* eljárás felszabadítja a blokkot, amely a későbbiekben – amikor arra sor kerül – a gyorsítótárból törölhetővé válik. (Miután a *get_block* megkapta a blokkot, az mindaddig nincs az LRU-listában, és nem is kerül oda vissza, amíg a blokk fejlécében lévő számláló a blokk használatát jelzi, így a blokk ez idő alatt mentesül a gyorsítótárból való kikerülés alól; a *put_block* eljárás csökkenti ezen számláló értékét, és amikor az eléri a nullát, a blokk visszakerül az LRU-listába.) A forráskód a 25327. sorban jelzi, vajon az írás művelet betöltötte-e a blokkot. Mindazonáltal a *put_block* számára az *n* változóban átadott érték egyáltalán nem befolyásolja, hogy a listában tulajdonképpen hová is kerül vissza a blokk; ilyenkor minden blokk az LRU-lista végére kerül.

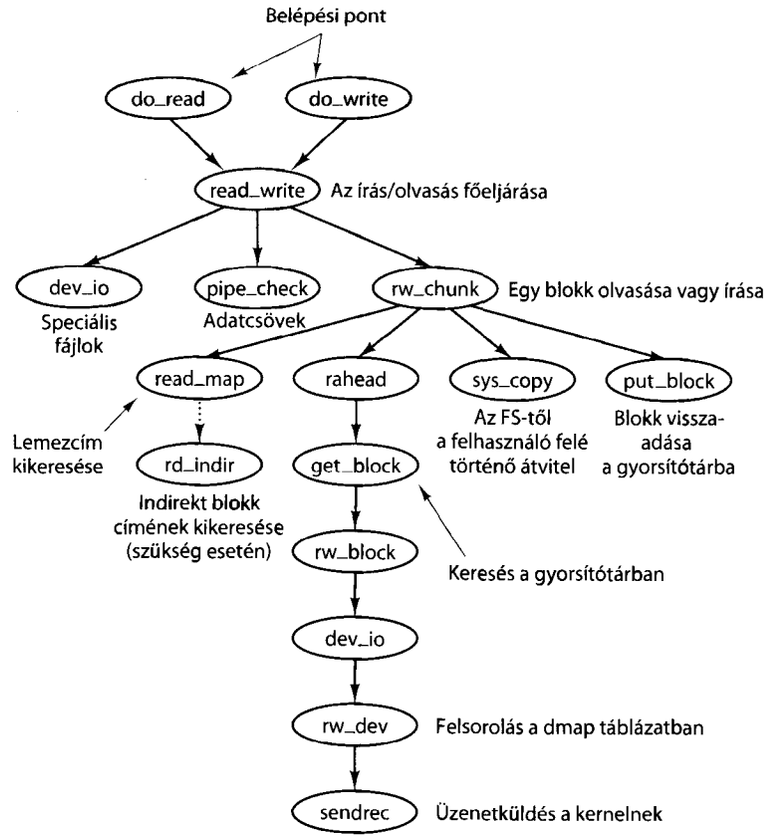
A logikai fájlpozíciót fizikai blokkazonosítóvá a *read_map* függvény (25337. sor) alakítja át az i-csomópont figyelembevételével. Azon blokkok esetén, amelyek elég közel vannak az állomány kezdetéhez, azaz az első hét zóna valamelyikébe esnek (természetesen az i-csomópontok által meghatározott zónákról van szó), elegendő egy egyszerű számolás a kívánt zóna, majd blokk meghatározásához. Messzebb elhelyezkedő blokkok esetén szükségünk lehet egy vagy több közvetett blokk beolvasására is.

Egy közvetett blokk beolvasására az *rd_indir* eljárást (25400. sor) használjuk. A függvényhez adott megjegyzés kissé idejétmúlt. A 68000 típusú processzort támogató kód eltávolításra került csakúgy, mint a MINIX V1 fájlrendszer kódja. Azonban érdemes megjegyezni, hogy ha valaki hozzá akar venni más fájlrendszereket vagy platformokat támogató részt, ahol a lemezes adatábrázolás vagy a bájtsorrend eltérő, beilleszthető ebbe a fájlba. Ha zavaros konverzióra lenne szükség, akkor azt itt elvégezve elérhetnénk, hogy a fájlrendszer további része azonos formátumban látná az adatokat.

A *read_ahead* eljárás (25432. sor) átalakítja a logikai címeket fizikai blokkazonosítókká, annak ellenőrzése céljából, vajon a blokk a gyorsítótárban van-e (ha nem lenne ott, beolvassa) meghívja a *get_block* eljárást, majd azonnal visszaküldi a blokkot. Tehát semmilyen műveletet nem végez az adott blokkon. Mindössze annak esélyét szeretné növelni, hogy a blokk a közelben legyen, amennyiben a következőkben esetleg szükség lenne rá.

Jegyezzük meg, hogy a *read_ahead* eljárás mindössze a *main* eljárásban lévő központi ciklusból hívható, a read rendszerhívás feldolgozásának részeként egyáltalán nem. Azt is fontos megértenünk, hogy a *read_ahead* hívása csak azután történik meg, miután egy üzenet visszaküldésre került, így a felhasználó abban az esetben is folytathatja futását, ha a fájlrendszernek az előreolvasás során egy lemezblokkra még várnia kell.

Magát a *read_ahead* eljárást úgy tervezték, hogy az mindig csak a következő blokk beolvasását kérje. A munka tényleges elvégzéséhez a *read.c* állomány utolsó függvényét, a *rahead* függvényt hívja meg. A *rahead* függvény (25451. sor) a „ha egy kicsit több jó, akkor a sokkal több, még jobb" elven működik. Mivel a lemezek és más tárolóeszközök esetén sok esetben viszonylag hosszú ideig tart a kívánt első blokk megtalálása, azonban ezt követően számos, egymás mellett elhelyezkedő blokk viszonylag gyorsan beolvasható, egy kevés pluszráfordítással lényegesen több blokk beolvasása lehetséges. Egy előzetes beolvasási kérelem érkezik a *get_*



5.46. ábra. *Egy állomány beolvasásakor használt főbb eljárások*

block eljáráshoz, amely a gyorsítótárat egyidejűleg több blokk érkezésére készíti fel. Ezután a blokkok egy listájával az *rw_scattered* eljárást hívjuk meg. Az ezután történő dolgokat már korábban megtárgyaltuk. Ebből most csak annyit idézünk fel, hogy az eszközmeghajtók *rw_scattered* általi hívásakor mindegyik meghajtónak csak annyi kérésre szabad válaszolnia, amennyit hatékonyan tud kezelni. Ez így elég bonyolultan hangzik, azonban éppen ezek a bonyolítások teszik lehetővé a hatalmas adatállományokat lemezről beolvasó alkalmazások lényeges felgyorsulását.

Az 5.46. ábra mutatja be egy állomány olvasása során használatos főbb eljárások egymáshoz való viszonyát, pontosabban azt, hogy melyik is hívja melyiket.

Fájl írása

A fájlokba történő írás forráskódja a *write.c* állományban található. Egy fájl írása hasonló az olvasásához, csak most a *do_write* eljárás (25625. sor) a *WRITING* jelző beállítása után hívja meg a *read_write* eljárást. Az írás és olvasás közti leg-

főbb különbség az, hogy az írás új lemezblokkok lefoglalását is igényelheti. A *read_map* eljárással megegyező *write_map* (25635. sor) az i-csomópontban, illetve a hozzá tartozó közvetett blokkokban nem a fizikai blokkazonosítókat keresi ki, hanem ezek helyére ír be újakat (ha pontosak akarunk lenni, akkor zóna- és nem blokkazonosítókról kell beszélnünk).

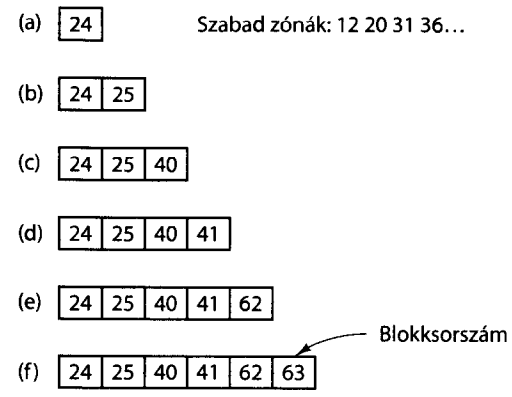
A *write_map* eljárás forráskódja hosszú és eléggé részletes, hiszen sok lehetőséggel kell foglalkoznia. Ha a beszúrandó zóna az állomány elejéhez közel van, akkor ez az információ egyszerűen bekerül az i-csomópontba.

A legrosszabb helyzet akkor fordul elő, amikor az állomány hossza túllépi a szimplán közvetett blokk által kezelhető méretet, azaz szükség lesz a duplán közvetett blokkra is. Ekkor helyet kell foglalni a szimplán közvetett blokk számára, majd címét a duplán közvetett blokkban kell elhelyezni. Erre, akárcsak olvasáskor, most is egy külön eljárás, a *wr_indir* elnevezésű szolgál. Amennyiben a duplán közvetett blokk igénylése helyesen történt ugyan, de a lemez telítettségi állapota miatt a szimplán közvetett blokk számára már nem tudunk helyet foglalni, a duplán közvetett blokkot vissza kell küldenünk, nehogy megrongáljuk a bittérképet.

Ha ezen a ponton pánikba esnénk, és egyszerűen bedobnánk a törülközőt, a forráskód sokkal egyszerűbb lehetne. A felhasználó szempontjából azonban sokkal praktikusabb, ha a lemezterület teljes betöltöttségekor a write rendszerhívás egy hibaüzenettel tér vissza ahelyett, hogy egy megrongálódott fájlrendszer következményeként összeomlana a rendszer.

A *wr_inder* eljárás (25726. sor) a szükséges adatátalakítások elvégzésére a *conv4* rutint hívja meg, majd beteszi egy közvetlen blokkba az új zónaazonosítót. Jegyezzük meg, hogy ezen függvény neve, sok más íráshoz és olvasáshoz használt függvényhez hasonlóan, szintén nem igaz a szó szoros értelmében. A lemezre történő tényleges írást a blokkgyorsítótár működését fenntartó függvények végzik el.

A *write.c* állomány következő eljárása a *clear_zone* eljárás (25747. sor), amely a hirtelen az állomány közepébe kerülő blokkok kitörlésének problémáját hivatott megoldani. Ez akkor következhet be, amikor egy állományvég utáni pozicionálást egy adat kiírása követi. Ez a helyzet szerencsére csak ritkán fordul elő.



5.47. ábra. *(a)–(f) 1 KB-os blokkok egymás utáni lefoglalása 2 KB hosszúságú zónák esetén*

Valahányszor új blokkra van szükség, az *rw_chunk* meghívja a *new_block* eljárást (25787. sor). Az 5.47. ábrán egy folyamatosan növekedő állomány hat egymás utáni állapota látható. Ebben a példában a zóna 2 KB, a blokk pedig 1 KB hosszú.

A *new_block* legelső hívásakor a 12-es zónát (24-es és 25-ös blokkot) foglalja le. A következő alkalommal a 25-ös blokkot használja fel, amelyet már korábban lefoglalt ugyan, de eddig még nem használt el. A harmadik híváskor a 20-as zónát (40-es és 41-es blokk) foglalja le, és így tovább. A *zero_block* függvény (25839. sor) egy blokkot ürít ki korábbi tartalmának kitörlésével. Működésének előbbi leírása tényleges forráskódjánál lényegesen hosszabb.

Adatcsövek

Az adatcsövek sok szempontból hasonlók a közönséges fájlokhoz. A most következő részben a különbségek bemutatására fektetjük a hangsúlyt. A tárgyalandó forráskód a *pipe.c* állományban található.

Először is az adatcsöveket eltérő módon, a create rendszerhívás helyett a pipe rendszerhívás használatával hozzuk létre. Ezt a hívást a *do_pipe* eljárás (25933. sor) kezeli. A *do_pipe* valójában az adatcső számára lefoglal egy i-csomópontot, és két ehhez tartozó állományleírót ad vissza. Az adatcsövek a felhasználó helyett a rendszerhez tartoznak és egy külön erre kijelölt eszközön (amelyet az *include/minix/config.h* definíciós állományban állíthatunk be) találhatók. Mivel az adatcsőben lévő adatokat nem kell megőrizni, ez az eszköz jól megfér a RAM-lemezen is.

Az adatcső írása és olvasása szintén eltér egy kicsit az állományok írásától és olvasásától, mivel egy adatcső kapacitása véges. Egy, már tele lévő adatcsőbe való írás kísérlete az író processzus felfüggesztését eredményezi. Hasonlóan üres adatcsőből történő olvasó az olvasási processzus felfüggesztéséhez vezet. Az adatcső valójában két mutatóval rendelkezik: az aktuális pozíció (ezt használják az olvasó processzusok), a másik pedig a méretet (ezt használják az író processzusok).

A *pipe_check* eljárás (25986. sor) hajtja végre mindazokat az ellenőrzéseket, amelyek azt hivatottak eldönteni, hogy egy adott művelet elvégezhető-e az adatcsövön. Ezen ellenőrzések mellett, amelyek a hívó felfüggesztéséhez vezethetnek, a *pipe_check* a *release* eljárást is meghívja, hogy megvizsgálja, vajon egy korábban adathiány vagy éppen túl sok adat jelenléte következtében felfüggesztett processzus jelenleg tovább folytatható-e. Az újraindítások alvó olvasó processzusok esetében a forráskód 26017., míg alvó író processzusok számára a 26052. sorában történnek. A megszakadt adatcsőbe (nincsenek olvasók) történő írás szintén ezen a ponton válik nyilvánvalóvá.

Egy processzus felfüggesztése a *suspend* eljárással (26073. sor) történik. Ez az eljárás a processzustáblába menti a hívás paramétereit, és a fájlrendszer válaszüzenetét megakadályozandó, a *dont_reply* jelzőt *TRUE*-ra állítja.

A *release* eljárást (26099. sor) használjuk annak ellenőrzésére, hogy egy adatcsővel kapcsolatos korábban felfüggesztett processzus folytatható-e. Ha az eljárás talál ilyen processzust, meghívja a *revive* függvényt, amely a megfelelő jelzőt úgy állítja be, hogy azt a központi ciklus a későbbiekben észrevegye. Ez a függvény

nem rendszerhívás, mivel azonban ez is az üzenettovábbítási technikát használja, szintén bekerült az 5.33.(c) ábrába.

A *pipe.c* állomány utolsó eljárása a *do_unpause* elnevezésű (26189. sor). Amikor a processzuskezelő egy processzus számára próbál meg jelzéseket küldeni, először ki kell találnia, hogy az adott processzus nincs-e egy adatcső vagy egy speciális állomány miatt felfüggesztve (ha ez történne, azt először az *EINTR* hibaüzenettel életre kell keltenie). Mivel a processzuskezelő semmit nem tud az adatcsövekről és a speciális állományokról, a fájlrendszertől üzenetben érdeklődik felőlük. Ezt az üzenetet dolgozza fel a *do_unpause,* amely az adott processzust újraindítja, ha az fel lenne függesztve. A *revive* függvényhez hasonlóan a *do_unpause* sem rendszerhívás annak ellenére, hogy a rendszerhívásokkal sok hasonlóságot mutat.

A *pipe.c* programban az utolsó két függvény a *select_request_pipe* (26246. sor) és a *select_match_pipe* (26278. sor) a select rendszerhívást támogatja, amelyet itt nem tárgyalunk.

5.7.5. Könyvtárak és elérési utak

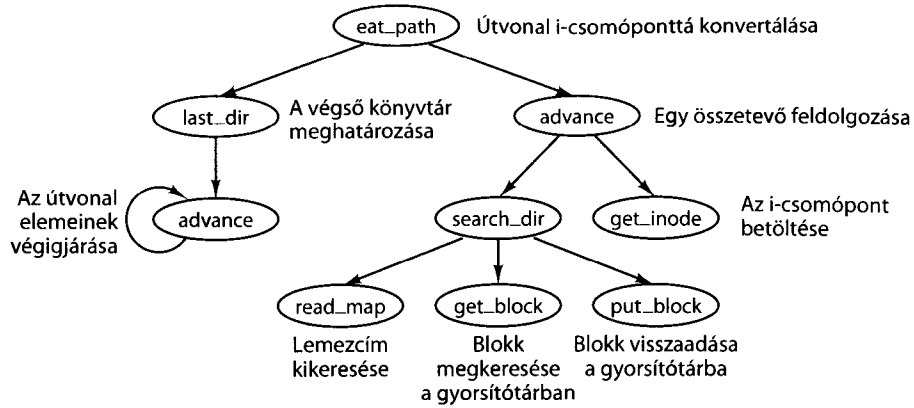
Az előbb megismertük, hogyan történik az állományok írása és olvasása. Következő feladatunk az, hogy megismerkedjünk az elérési utak és könyvtárak kezelésével.

Az elérési út i-csomóponttá történő átváltása

Sok rendszerhívás (például open, unlink és mount) tartalmaz paraméterként elérési utakat (például állománynevek) is. Ezen hívások legtöbbjének, még mielőtt magát a hívást kezdenék el végrehajtani, meg kell szerezniük a megnevezett állomány i-csomópontját. Az elérési út i-csomóponttá való átalakítását részletesebben is megvizsgáljuk. Nagy általánosságban ezt az 5.16. ábrához kapcsolódóan egyszer már megtettük.

Az elérési utak elemzése a *path.c* állományban történik. Ennek legelső eljárása, az *eat_path* (26327. sor) kapja meg az elérési utat kijelölő mutatót, elemzi, kikeresi a memóriába töltendő i-csomópontot, majd ennek mutatójával tér vissza. Az utolsó könyvtár i-csomópontjának megszerzéséhez a *last_dir* eljárást hívja meg, az elérési út utolsó összetevőjének megszerzéséhez pedig az *advance* eljárást mozgósítja. Sikertelen keresés esetén – ennek oka lehet például az, hogy az elérési útban szereplő könyvtárak valamelyike nem létezik, vagy létezik, de keresés nem engedélyezett benne – az i-csomópontot kijelölő mutató helyett a *NIL_INODE* kerül visszaküldésre.

Az elérési utak lehetnek abszolútak vagy relatívak, és egymástól a / jellel elválasztva sok összetevőből állhatnak. Ezekkel a kérdésekkel a *last_dir* eljárás foglalkozik. Annak eldöntésére, hogy abszolút vagy relatív elérési útról van-e szó, először az elérési út legelső karakterét vizsgálja meg (26371. sor). Abszolút elérési út esetén a *rip* mutatót úgy állítja be, hogy az a gyökér i-csomópontjára mutasson, míg relatív elérési út esetén az aktuális munkakönyvtár i-csomópontját jelölje ki.



5.48. ábra. *Az elérési utak kikeresése során használt néhány eljárás*

Ezen a ponton a *last_dir* ismeri tehát az elérési utat és azon könyvtár i-csomópontjának mutatóját, amelyben az elérési út első összetevőjét kell keresnie. A 26382. sorban belép egy ciklusba, és összetevőről összetevőre kielemzi az elérési utat. Amikor ennek a végére ér, visszaküldi az utolsó könyvtárat kijelölő mutatót.

A *get_name* (26413. sor) olyan segédeljárás, amely a karakterláncokból kiszedi az egyes összetevőket. Ennél sokkal érdekesebb az *advance* eljárás (26454. sor), amely egy könyvtárat kijelölő mutatót és egy karakterláncot paraméterként kapva, az illető könyvtárban az adott karakterláncot keresi. Ha megtalálja, visszaküldi a hozzá tartozó i-csomópont mutatóját. Az összekapcsolt fájlrendszerek közötti átvitel részletei szintén itt kerülnek definiálásra.

Annak ellenére, hogy az *advance* eljárás felügyeli a karakterlánc kikeresését, a karakterlánc tényleges könyvtárbejegyzéssel való összehasonlítása a fájlrendszerben kivételes helyet elfoglaló *search_dir* eljárásban (26535. sor) történik; ez az egyetlen olyan hely ugyanis, ahol ténylegesen a könyvtárállományok kerülnek vizsgálatra. Az eljárás két egymásba ágyazott ciklust tartalmaz, az egyik a könyvtár blokkjain, a másik egy adott blokk elemein fut végig. A könyvtárba történő új nevek bejegyzéséhez vagy onnan való kitörléséhez szintén a *search_dir* eljárást használjuk. Az elérési utak kikeresése során használatos főbb eljárások egymás közötti kapcsolatait az 5.48. ábra szemlélteti.

Fájlrendszerek felcsatolása

Van két olyan rendszerhívás, amely a fájlrendszert mint egészet befolyásolja. Ezek a mount és az umount hívások. Ezek teszik lehetővé a különböző másodlagos eszközökön lévő, egymástól teljesen független fájlrendszerek oly módon történő egymáshoz „ragasztását", hogy azok egy egységes, illesztési problémáktól mentes adatszerkezetet alkossanak. A felcsatolás, amint azt az 5.38. ábrán láttuk, valójában azáltal érhető el, hogy beolvassuk a felcsatolásra kijelölt fájlrendszer gyöke-

rének i-csomópontját, valamint szuperblokkját, és a szuperblokkban beállítunk két mutatót. Ezek egyike azt az i-csomópontot jelöli ki, ahová a felcsatolás történt, a másik pedig a felcsatolt fájlrendszer gyökerének i-csomópontját mutatja. Ezek a mutatók tartják össze a fájlrendszereket.

A mutatók beállítása a *do_mount* eljárás közvetítésével a *mount.c* állomány 26819. és 26820. sorában történik meg. A mutatók beállítását megelőző kétoldalnyi forráskód szinte csak a fájlrendszer felcsatolásakor lehetséges hibaüzenetek feldolgozásával foglalkozik. Ezek közül csak néhányat sorolunk most fel.

1. A kijelölt speciális állomány nem blokkeszköz.
2. A speciális állomány blokkeszköz ugyan, de már felcsatolás alatt áll.
3. A felcsatolásra kijelölt fájlrendszer bűvös száma hibás.
4. A felcsatolásra kijelölt fájlrendszer érvénytelen (nem léteznek i-csomópontok).
5. Az az állomány, amelyre a felcsatolást kértük, nem létezik, vagy egy speciális állomány.
6. Nincs hely a felcsatolt fájlrendszer bittérképe számára.
7. Nincs hely a felcsatolt fájlrendszer szuperblokkja számára.
8. Nincs hely a felcsatolt fájlrendszer gyökeréhez tartozó i-csomópont számára.

Talán nem a leghelyénvalóbb már megint ugyanazt hangsúlyoznunk, de egy gyakorlati operációs rendszerhez hozzátartozik az is, hogy forráskódjának lényegi hányada olyan apró tevékenységek ellátását szolgálja, amelyek ugyan nem jelentenek szellemi kihívást, azonban a rendszer megbízható működése szempontjából döntők. Ha a felhasználó, mondjuk, havonta egyszer hibás hajlékonylemezt próbál felcsatolni, aminek következtében rendszere összeomlik, és egy megrongálódott fájlrendszert hagy hátra, a felhasználó nem önmagát, hanem a tervezőt fogja hibáztatni, és a rendszert minősíti ezután megbízhatatlannak.

Thomas Edison a híres feltaláló egyszer egy rendkívül találó megjegyzést tett. Úgy nyilatkozott, hogy „zseninek" lenni 1 százalék ötletet és 99 százalék izzadságos munkát jelent. Egy jó és egy silány rendszer között nem az előbbi briliánsan kivitelezett ütemező algoritmusai tesznek különbséget, hanem az összes apró részlet pontos kivitelezése.

A fájlrendszer lecsatolása sokkal egyszerűbb, mint annak hozzákapcsolása – sokkal kevesebb dolgot lehet elrontani. A *do_umount* eljárás (26828. sor) hívásával kezdődik a munka, amely két részre bomlik. A *do_umount* maga ellenőrzi, hogy a szuperfelhasználó hívta-e meg, a nevet eszközszámmá konvertálja, és aztán hívja az *umount*-ot (26846. sor), amely befejezi a műveletet. Az egyetlen lényeges pont annak biztosítása, hogy semmilyen processzus ne használjon olyan állományokat, illetve munkakönyvtárakat, amelyek a lecsatolandó fájlrendszerhez tartoznak. Ennek ellenőrzése rendkívül egyszerű: át kell vizsgálni a teljes i-csomópont táblát, hogy lássuk, vannak-e a memóriában olyan i-csomópontok, amelyek a lecsatolandó fájlrendszerhez tartoznak (leszámítva a gyökeréhez tartozó i-csomópontot). Ha vannak ilyenek, az umount rendszerhívás végrehajtása sikertelen lesz.

A *mount.c* állomány utolsó eljárása a *name_to_dev* eljárás (26839. sor), amely egy speciális állomány elérési útját alapul véve megszerzi annak i-csomópontját, majd kiolvassa abból az állományhoz rendelt fő és másodlagos eszközazonosító számot. Ezek az azonosítók ugyanis magában az i-csomópontban vannak tárolva, rendszerint ott, ahová az első zóna kerülne. Ez a rekesz a speciális állományoknál azért használható fel erre a célra, mert a speciális állományok nem rendelkeznek zónákkal.

Fájlok kapcsolása és szétkapcsolása

Vizsgálatunk következő állomása a *link.c* állomány, amely az állományok összekapcsolását vagy szétkapcsolását végzi. A *do_link* eljárás (27034. sor) abban nagyon hasonlít a *do_mount* eljáráshoz, hogy forráskódja majdnem teljes egészében a lehetséges hibák vizsgálatára fordítódik. Néhány a

```
link(file_name, link_name);
```

hívás során előforduló hibát az alábbiakban sorolunk fel:

1. A *file_name* nem létezik vagy nem érhető el.
2. A *file_name* már elérte lehetséges kapcsolatainak maximális számát.
3. A *file_name* egy könyvtárat jelent (ehhez csak a rendszergazda kapcsolhat hozzá valamit).
4. A *link_name* már létezik.
5. A *file_name* és a *link_name* különböző eszközökön található.

Ha nem ütközünk semmilyen hibába, akkor a fenti utasítás hatására a *link_name* karakterláncnak megfelelő néven egy új, a *file_name* állományhoz tartozó i-csomópont azonosítójával rendelkező könyvtárbejegyzés jön létre. A forráskódban a *name1* felel meg a *file_name* változónak, míg a *name2* jelenti a *link_name* változót. A tényleges könyvtárbejegyzést a 27086. sorban a *do_link* által meghívott *search_dir* hozza létre.

Az állományok és könyvtárak eltávolítása ezek szétkapcsolásával történik. Az unlink és az rmdir rendszerhívások munkáját egyaránt a *do_unlink* eljárás (27104. sor) végzi el. Ebben az esetben is rengeteg ellenőrzést kell elvégeznünk; annak ellenőrzését, hogy egy állomány létezik-e, illetve egy könyvtár nem hozzákapcsolási pont-e, a *do_unlink*-ben található közös programkód végzi el, majd annak megfelelően, hogy melyik rendszerhívásról van szó, meghívja a *remove_dir* vagy az *unlink_file* eljárások egyikét. Rövidesen ezeket is megtárgyaljuk.

A *link.c* állományban támogatott másik rendszerhívás a rename. Akik Unixot használnak, ismerik a parancsértelmező *mv* parancsát, amely tulajdonképpen ezt a rendszerhívást használja; a név a hívás egy másik tevékenységét tükrözi. A rename nem csupán arra szolgál, hogy egy állomány nevét a könyvtáron belül megváltoztassuk, de használatával az adott állományt könnyen az egyik könyvtárból a másik-

ba mozgathatjuk. Ráadásul ez egyetlen oszthatatlan elemi műveletben történik, így bizonyos versenyhelyzeteket is el tudunk kerülni. A munkát a *do_rename* eljárás (27162. sor) végzi el. A parancs végrehajtása előtt jó néhány feltétel teljesülését ellenőrizni kell. Ezek közül néhányat az alábbiakban sorolunk fel:

1. Az eredeti állománynak léteznie kell (27177. sor).
2. A régi elérési út nem lehet olyan könyvtár, amely a könyvtárfában az új elérési út fölött helyezkedik el (27195–27212. sor).
3. Sem a „.", sem a „.." nem használható régi vagy új névként (27217. és 27218. sorok).
4. Az elérési utakban szereplő utolsó könyvtárak mindegyikének ugyanazon az eszközön kell lennie (27221. sor).
5. Az elérési utakban szereplő utolsó könyvtárak mindegyikének írhatónak és kereshetőnek kell lennie, egy olyan eszközön, amely szintén írható (27224. és 27225. sorok).
6. Sem a régi, sem az új név nem eshet egybe olyan könyvtárral, amelyhez egy fájlrendszer épp fel van csatolva.

Ezeken túl van még néhány egyéb feltétel is, amelyeket csak az új név létezésekor kell megvizsgálni; ezek közül a legfontosabb az, hogy az adott névvel már rendelkező állomány törölhető legyen.

A *do_rename* eljárás forráskódjában találkozhatunk néhány olyan, a tervezéssel kapcsolatos döntéssel is, amelyek célja bizonyos problémák előfordulásának csökkentése. Egy állomány teljesen betöltött lemezen való, már létező névre történő átnevezése még akkor is sikertelen lenne – amennyiben a műveletet nem a régi állomány törlésével kezdenénk (éppen ez történik a 27260. és 27266. sorok között) –, ha az állomány a végén egyáltalán nem foglalna el több helyet, mint kiinduláskor. Ugyanezt az elvet használjuk fel a 27280. sorban, vagyis az ugyanabban a könyvtárban történő új név létrehozása előtt töröljük a régit, annak elkerülése érdekében, hogy az adott könyvtár esetleg egy többletblokkot igényeljen. Ha azonban az új és a régi állomány más könyvtárba kerülnek, az előbb említett elv lényegtelenné válik, és a 27285. sorral még a régi állomány törlése előtt létrehozzuk az új állományt (a másik könyvtárban). Mindez azért történik így, mert a rendszer épsége szempontjából egy olyan összeomlás, amely két, ugyanarra az i-csomópontra mutató állománynevet hagy maga után, sokkal ártalmatlanabb, mint egy olyan, amelynek eredményeként olyan i-csomópont jön létre, amelyhez semmilyen könyvtárbejegyzés nem tartozik. Annak valószínűsége, hogy épp egy átnevezési művelet közepén futunk ki a szabad tárterületből, rendkívül alacsony – a rendszerösszeomlás valószínűsége még ennél is alacsonyabb –, azonban az ilyen eseteknél a legrosszabb eshetőségre való felkészülés semmibe nem kerül.

A *link.c* állományban fennmaradó függvények a korábban már megtárgyaltakat támogatják. Közülük az elsőt, *truncate* (27316. sor), a fájlrendszer más részei is meghívják. Ez egy adott i-csomópont zónáin fut végig, és a közvetett blokkokkal együtt felszabadítja azokat. A *remove_dir* függvény (27375. sor) számos, a könyvtár törölhetőségét ellenőrző vizsgálatot végez el, majd ezután meghívja az *unlink_file* eljárást (27415. sor). Ha semmilyen hiba nem jelentkezik, a könyvtár törlődik és i-csomópontjának kapcsolatok számát figyelő számlálója eggyel csökken.

5.7.6. További rendszerhívások

A rendszerhívások utolsó csoportja rendkívül tarka, olyan elemek tartoznak ide, mint például az állapot, a könyvtárak, a védelem, az idő, illetve ehhez hasonló dolgok.

A könyvtárak és fájlok állapotának megváltoztatása

A *stadir.c* állomány négy rendszerhívás forráskódját tartalmazza; ezek: chdir, fchdir, chroot, stat és fstat. A *last_dir* eljárás vizsgálata során láttuk, hogyan történik egy elérési út kikeresése – kezdve az elérési út legelső karakterének vizsgálatával, hogy eldönthessük, vajon az / jel-e vagy sem. Ennek eredményétől függően ugyanis egy mutató vagy a gyökérkönyvtárat, vagy az éppen aktuális munkakönyvtárat fogja tartalmazni.

Az egyik munkakönyvtárról (vagy gyökérkönyvtárról) egy másikra történő átváltás csupán a könyvtárak mutatóinak a hívó processzustáblájában történő megcserélését jelenti, amit a *do_chdir* (27542. sor) és a *do_chroot* (27580. sor) eljárások végeznek. Először mindkettő a szükséges ellenőrzéseket hajtja végre, majd meghívja a *change* függvényt (27594. sor), amely további ellenőrzéseket végez, majd hívja a *change_into* eljárást (27611. sor), amely megnyitja az új könyvtárat, és a régit ezzel helyettesíti.

A *do_fchdir* (27529. sor) eljárás támogatja az fchdir-t, amely egy alternatív módszert ad a chdir művelet elvégzésére, amikor is az argumentum egy fájlleíró, és nem elérési út. Először ellenőrzi, hogy a fájlleíró érvényes-e, ha igen, akkor hívja a *change_into* eljárást, amely elvégzi a munkát.

A *do_chdir* eljárásban felhasználói chdir hívások esetén a forráskód 27552. és 27570. sorai közötti rész nem kerül végrehajtásra. Ez a rész speciálisan a processzuskezelő hívásai számára van fenntartva, amikor az exec hívások kezelése érdekében a felhasználói könyvtárra váltunk át. Ha ugyanis a felhasználó megpróbálja mondjuk a saját könyvtárában található *a.out* állományt futtatni, a processzuskezelőnek egyszerűbb erre a könyvtárra átváltania, mint azt megkeresni, hogy hol is van az illető állomány.

A stat és az fstat rendszerhívás lényegében megegyezik egymással. Különbség csak a fájlok megadásában van: míg az első egy megnyitott állomány elérési útját adja eredményül, a második annak állományleíróját. A felsőszintű *do_stat* (27638. sor) és *do_fstat* (27658. sor) eljárások a munka elvégzéséhez a *stat_inode* függvényt hívják meg. A *stat_inode* függvény hívása előtt a *do_stat* eljárás megnyitja az állományt az i-csomópontjának megszerzése céljából. Ily módon mind a *do_stat*, mind a *do_fstat* egy i-csomópontot kijelölő mutatót ad át a *stat_inode* függvénynek.

A *stat_inode* függvény (27673. sor) összes tevékenysége abból áll, hogy az i-csomópontból információt olvas ki és azt egy átmeneti adattárba másolja. A 27713. és 27714. sor *sys_copy* hívásával ezt az átmeneti adattárat a felhasználói területre ténylegesen át kell másolni, mivel ennek mérete túlságosan nagy ahhoz, hogy egy üzenetbe beférhessen.

Végül lássuk a *do_fstatfs* (27721. sor) függvényt. Az fstatfs nem POSIX-függvény, bár a POSIX definiál egy hasonló fstatvfs függvényt, amely azonban sokkal nagyobb adatszerkezetet ad vissza. A MINIX 3 fstatfs csak az információ egy részét adja, a fájlrendszer blokkméretét. A függvény prototípusa:

```
_PROTOTYPE(int fstatfs, (int fd, struct statfs *st));
```

A *statfs* struktúra egyszerű, elfér egy sorban:

```
struct statfs { off_t f_bsize; /* fájlrendszer blokkméret */ };
```

Ezek a definíciók az *include/sys/statfs.h* definíciós állományban vannak.

Adatvédelem

A MINIX 3 védelmi technikája az *rwx* biteket használja. Minden állomány esetén három ilyen bitcsoport létezik: a tulajdonos, annak csoportja és az ezeken kívüli egyéb felhasználók számára. A biteket a chmod rendszerhívással állíthatjuk, amelyet a *protect.c* állományban lévő *do_chmod* eljárás (27824. sor) kezel. A védelmi mód megváltoztatása, egy sor érvényességi vizsgálat elvégzése után, a 27850. sorban történik.

A chown rendszerhívás annyiban hasonlít a chmod híváshoz, hogy mindketten az állományokhoz tartozó i-csomópontok belső mezőjének tartalmát írják felül. A kivitelezés is hasonló, bár a *do_chown* eljárással (27862. sor) csak a rendszergazda változtathatja meg egy állomány tulajdonosát. A közönséges felhasználók ezzel a hívással csak saját állományaik csoportazonosítóját állíthatják be.

Az umask rendszerhívás teszi lehetővé, hogy a felhasználó létrehozzon egy sablont (ezt a processzustáblában tárolja), amely a későbbi create hívások védelmi bitjeit módosítja. A hívás teljes kivitelezése mindössze egyetlen utasítást igényelne (27907. sor), ha a hívásnak a régi sablonértéket nem kellene eredményül visszaküldenie. Ez a többletfeladat azonban a szükséges forráskódsorok számának megháromszorozódásához vezet (27906–27908. sor).

Az access rendszerhívás teszi lehetővé egy processzus számára, hogy megvizsgálhassa, vajon elérhet-e egy állományt a megadott módon (például olvasás számára) vagy sem. Ez a *do_access* eljáráson keresztül (27914. sor) valósul meg, amely az állomány i-csomópontjának megszerzése után meghívja a *forbidden* elnevezésű belső eljárást (27938. sor). Ez vizsgálja meg, vajon le van-e tiltva az elérés. A *forbidden* eljárás az *uid* és a *gid* vizsgálatán túl az i-csomópontban található információt is megvizsgálja. Annak függvényében, hogy ott mit talál, kiválasztja a három

rwx bitcsoport egyikét, és ellenőrzi, hogy megengedett vagy tiltott-e az ennek megfelelő elérés.

A *read_only* eljárás (27999. sor) egy aprócska belső eljárás, amely azt határozza meg, hogy azon állományrendszer, amelyben a neki átadott i-csomópont található, csak olvasásra vagy írásra és olvasásra van-e felcsatolva. Ez a csak olvasásra felcsatolt állományrendszerbe történő írás megelőzése érdekében szükséges.

5.7.7. Az I/O-eszközcsatoló

Amint már többször is megemlítettük, a MINIX 3-at úgy tervezték, hogy robusztus operációs rendszer legyen azáltal, hogy minden eszközvezérlő felhasználói térben futó processzus legyen, amely közvetlenül nem fér hozzá a kernel adatszerkezetéhez vagy kódjához. Az elsődleges előnye ennek a megközelítésnek, hogy egy hibás eszköz nem okozhatja a rendszer összeomlását, de van más következménye is. Az egyik az, hogy az eszközvezérlőket nem kell közvetlenül az indítás után elindítani, indíthatók a betöltés befejeződése után bármikor. Ebből az is következik, hogy az eszközvezérlő akármikor leállítható, újraindítható vagy helyettesíthető másik vezérlővel, a rendszer futása közben. Ez a flexibilitás természetesen bizonyos feltételekhez kötött, például nem indítható ugyanarra az eszközre több vezérlő. Azonban ha a lemezmeghajtó összeomlik, akkor újraindítható RAM-lemez másolatról.

A MINIX 3 eszközvezérlői a fájlrendszerben érhetők el. Felhasználói I/O-kérésre válaszul a fájlrendszer üzenetet küld a felhasználói térben futó eszközvezérlőnek. A *dmap* táblázatban minden lehetséges eszköz főszámához van egy elem. Ez biztosítja az eszköz fősorszáma és az eszközvezérlő közötti megfeleltetést. A következőkben vizsgálandó két fájl a *dmap* táblázat kezelésével kapcsolatos. A táblázatot magát a *dmap.c* deklarálja. Ez a fájl tartalmazza a táblázat inicializálását és egy új rendszerhívás, a *devctl* célja az eszközvezérlők indításának, leállításának és újraindításának támogatása. Ezután a *device.c* fájlt nézzük, amely eszközök szokásos műveleteit tartalmazza; ezek az open, close, read, write és ioctl.

Eszköz megnyitásakor, lezárásakor, olvasásakor vagy írásakor a *dmap* szolgáltatja a műveletet elvégző eljárás nevét. Ezen eljárások mindegyike a fájlrendszer névterében van. Néhány eljárás nem csinál semmit, de néhány hívja az eszközvezérlőt a kívánt I/O-művelet elvégzésére. Szintén a táblázat szolgáltatja minden eszköz főszámához a processzusszámot.

Valahányszor egy új eszközt adunk a MINIX 3-hoz, ebben a táblázatban egy új sor keletkezik, amely tartalmazza azt a műveletet, amelyet megnyitáskor, lezáráskor, olvasáskor vagy íráskor végezni kell, ha kell. Egy egyszerű példa erre, ha egy mágnesszalageszközt hozzáadunk a MINIX 3-hoz, és a hozzá tartozó speciális fájlt megnyitjuk, akkor a táblázatbeli eljárásnak meg kell tudnia mondani, hogy az eszköz használatban van-e már.

A *dmap.c* a *DT* makródefinícióval kezdődik (28115–28117. sor), amely a *dmap* táblázat inicializálását végzi. Ez a makro egyszerűvé teszi új eszköz hozzáadását, amikor átkonfiguráljuk a MINIX 3-rendszert. A *dmap* táblázat elemeit az *include/*

minix/dmap.h definiálja, minden elem tartalmaz egy függvényre mutató pointert, amely az olvasás és írás műveletet végzi, olyat, amely a megnyitást és lezárást végzi, processzusszámot (processzustáblázatbeli indexet, és nem PID-et), továbbá jelzőket. A tényleges táblázat egy ilyen elemeket tartalmazó tömb, amelynek a deklarációja a 28132. sorban van. Ez a táblázat a fájlrendszerben globálisan hozzáférhető. A táblázat méretét *NR_DEVICE* tartalmazza és értéke 32, ami majdnem kétszer akkora, mint ahány eszközt a jelenlegi MINIX 3-változat támogat. Szerencsére a C nyelv szerint minden nem inicializált változó értéke 0 lesz, ami biztosítja, hogy a nem használatos elemekben ne jelenjen meg hamis információ.

A *dmap* deklarációját követi az *init_dmap PRIVATE* deklarációja. Ezt *DT* makrók tömbje definiálja minden lehetséges főeszközre. Minden egyes makró kifejtése a tömb egy elemét inicializálja fordítási időben. Néhány ilyen makró áttekintése segít megérteni a működésüket. Az *init_dmap[1]* a memóriavezérlő definiálását adja, amely 1 főszámú eszköz. A makró a következőképpen néz ki:

```
DT(1, gen_opcl, gen_io, MEM_PROC_NR, 0)
```

A memóriavezérlő mindig jelen van, és a rendszer indításakor betöltődik. Az 1 első paraméter azt jelenti, hogy ez az eszköz kötelező. A *gen_opcl* argumentum adja meg azt a függvényt, amelyet megnyitás és lezárás esetén kell hívni, a *gen_io* azt, amelyet olvasás és írás esetén kell hívni, a *MEM_PROC_NR* a memóriavezérlő számára fenntartott processzustáblázat elemét jelöli ki, és az utolsó 0 argumentum azt mondja, hogy nem kell jelzőt beállítani. Nézzük a következőt, ami az *init_dmap[2]*. Ez a hajlékonylemez-meghajtót definiálja, és a következőképpen néz ki:

```
DT(0, no_dev, 0, 0, DMAP_MUTABLE)
```

Az első „0" argumentum azt jelenti, hogy az eszköznek nem kell a betöltési memóriaképben szerepelnie. A második argumentum azt mondja, hogy az eszköz megnyitásakor az alapértelmezett *no_dev* függvényt kell hívni. Ez a függvény az *ENODEV* „nincs ilyen eszköz" hibaüzenettel tér vissza. A következő két 0 szintén alapértelmezett érték, mivel az eszköz nem nyitható meg, ezért nem lehet I/O-műveletet végezni, és a 0 processzustábla-index azt jelenti, hogy nincs processzus hozzárendelve. A *DMAP_MUTABLE* argumentum azt jelenti, hogy az elem megváltoztatható. (Megjegyzendő, hogy ennek a jelzőnek a hiánya a memóriavezérlő esetén azt jelenti, hogy az inicializálás után nem változtatható.) A MINIX 3 konfigurálható úgy, hogy a betöltő tartalmazzon hajlékonylemez-meghajtót, de úgy is, hogy ne. Ha a hajlékonylemez-meghajtót tartalmazza a betöltési memóriakép és a *label=FLOPPY* indítóparaméterrel specifikáltuk, hogy ez az alapértelmezett lemezes eszköz, akkor a fájlrendszer indításakor a hozzá tartozó elem meg fog változni. Ha a hajlékonylemez-meghajtót nem tartalmazza a betöltési memóriakép, vagy tartalmazza, de nem az alapértelmezett lemezes eszköz, akkor a hozzá tartozó elem nem fog megváltozni az FS indításakor. Azonban még így is később felélesztheto a hajlékonylemez-meghajtó. Ezt tipikusan az */etc/rc* végezheti az *init* futásakor.

A *do_devctl* (28157. sor) az első függvényhívás, amely devctl szolgáltatást hív. A jellenlegi változat nagyon egyszerű, két kérést ismer; ezek a *DEV_MAP* és a *DEV_UMAP,* és az utóbbi *ENOSYS* hibaüzenettel tér vissza, ami azt jelenti, hogy „a függvény nincs implementálva". Ez nyilvánvalóan stophézag. *DEV_MAP* esetén a *map_drive* függvény hívása következik.

Hasznos lehet annak elmagyarázása, hogyan használatos a devctl, és mi a jövőbeli felhasználásának a terve. A **reinkarnációs szerver (RS)** szerverprocesszust használja a MINIX 3 az operációs rendszer futás közbeni felhasználói szerverek és vezérlők indításának a támogatására. A reinkarnációs szerver csatolófelületéül szolgál a *service* segédprogram, és az */etc/rc*-ben láthatunk példát a használatára. Például

```
service up /sbin/floppy -dev /dev/fd0
```

Ez a parancs azt eredményezi, hogy a reinkarnációs szerver devctl hívást alkalmaz a */sbin/floppy* bináris fájl elindítására, amely a */dev/fd0* eszközspecifikus fájl számára eszközvezérlő lesz. Ehhez az RS exec-kel hajtja végre a binárist, de beállít egy jelzőt, amely megtiltja a futását mindaddig, amíg rendszerprocesszussá nem transzformálta. Amikor a processzus már a memóriában van, és ismert a processzustábla-beli indexe, a megadott eszköz főeszközszámát meghatározza. Ezt az információt tartalmazza a devctl *DEV_MAP* műveletet kezdeményező fájlszervernek visszaküldött üzenet. Az I/O-csatolók inicializálása szempontjából ez a legfontosabb része a reinkarnációs szerver feladatának. A teljesség kedvéért megemlítjük, hogy az eszközvezérlő inicializálásának befejezéséhez az RS végrehajt egy sys_privctl hívást, hogy a rendszerprocesszus inicializálja az eszköz *priv* processzustáblájának elemét, és engedélyezze a futást. Emlékeztetünk a 2. fejezetre, hogy a dedikált *priv* táblaelem teszi lehetővé, hogy az egyébként közönséges felhasználói processzus rendszertaszkká váljon.

A reinkarnációs szerver új és alapvető a MINIX 3 jelenlegi kiadásában. A tervek szerint a későbbi MINIX 3-kiadások még hatékonyabb reinkarnációs szervert tartalmaznak, amely már eszközöket nemcsak indítani, de leállítani és újraindítani is képes lesz. Továbbá képes lesz az eszközök figyelésére és hibás működés esetén automatikusan újra tud indítani. A legfrissebb információk megtalálhatók a *www.minix3.org* oldalon, valamint a *comp.os.minix* hírcsoportnál.

Folytatva a *dmap.c* programot, a *map_driver* függvény a 27178. sornál kezdődik. A működése egyszerű. Ha a *dmap* táblaelem *DMAP_MUTABLE* jelzője be van állítva, akkor megfelelő értékeket ír minden elembe. A megnyitás és lezárás kezelésére három különböző változata van; az egyiket a *style* paraméter választja ki, amelyet az RS által a fájlrendszernek küldött üzenet tartalmaz (28204–28206. sor). Megjegyezzük, hogy a *dmap_flags* nem módosul, ha eredetileg *DMAP_MUTABLE* volt, a devctl hívás után is az marad.

A harmadik függvény a *dmap.c* programban a *build_map*. Ezt az *fs_init* hívja a fájlrendszer indításakor, még mielőtt a főciklusába belépne. Először végigmegy az *init_dmap* tábázat minden elemén, és minden olyan elemre, amelynek nem *no_dev* a *dmap_opcl* mezője, a kifejtett makrót bemásolja a globális *dmap* táblá-

zatba. Ezzel inicializált lesz a táblaelem. Egyébként nem inicializált eszköz esetén alapértelmezett érték kerül a *dmap* táblaelembe. A *build_map* további része még érdekesebb. Több lemezeszközt is tartalmazó betöltési memóriakép is készíthető. Alapként az *src/tools/* könyvtárbeli *Makefile* az *at_wini, bios_wini* és *floppy* vezérlőket tartalmazza. Ezek mindegyike kap egy címkét, és a *label=item* indítóparaméter mondja meg, hogy aktuálisan melyik eszköz töltődik be és aktivizálódik mint alapértelmezett lemezvezérlő. A 28248. és 28250. sorban az *env_get_param* hívás olyan könyvtári rutinokat hív, amelyek végső soron a *sys_getinfo* kernelhívást alkalmazzák a *label* és *controller* indítóparaméterek elérésére. Végül a 28267. sorban meghívásra kerül a *build_map,* amely módosítja *dmap* táblában a betöltési eszközhöz tartozó elemet. A kulcskérdés itt az, hogy a processzusszámot *DRVR_PROC_NR* értékre állítja be, amely a processzustáblában a 6. elem.

A továbbiakban a *device.c* programot vizsgáljuk, amely az eszközök futás közbeni I/O elvégzéséhez szükséges eljárásokat tartalmazza.

Az első a *dev_open* (28334. sor). Ezt a fájlrendszer más részei hívják, leggyakrabban a *common_open* a *mai.c*-ben, amikor open műveletet kell végrehajtani eszközspecifikus fájlra, de meghívódik *load_ram* és *do_mount* eljárásokból is. A tevékenysége tipikus több itt tárgyalt eljárás esetén. Meghatározza az eszköz főszámát, ellenőrzi annak érvényességét, és értékével meghatározza a *dmap* táblából a meghívandó függvényre mutató pointert, és meg is hívja a 28349. sorban.

```
r = (*dp->dmap_opcl)(DEV_open, dev, proc, flags)
```

Lemezeszköz esetén a meghívott függvény *gen_opcl,* terminál esetén *tty_opcl.* Ha a hívás *SUSPEND* értéket ad vissza, akkor valami súlyos hiba történt, open hívás nem lehet sikertelen ilyen módon.

A következő hívás, a *dev_close* (28357. sor) egyszerűbb. Nem várható, hogy érvénytelen eszközre hívják meg, és nem jelent veszélyt a sikertelen lezárás, így a kód rövidebb, mint az azt magyarázó szöveg: csupán egy sor. Azzal végződik, hogy meghívja ugyanazt a **_opcl* eljárást, amelyet megnyitás esetén is hívott.

Amikor a fájlrendszer értesítést kap az eszközvezérlőtől, meghívja a *dev_status* (28366. sor) függvényt. Az értesítés üzenet azt jelenti, hogy valamilyen esemény történt, és ez a függvény felelős azért, hogy kiderítse, mi volt az esemény, és elindítsa a megfelelő tevékenységet. A bejelentés származási helye egy processzus, tehát először kikeresi a *dmap* táblázatban, hogy melyik eszköztől származik (28371–28373. sor). Előfordulhat, hogy a bejelentés hibát tartalmaz, ezért nem hiba, ha nem talál a táblázatban neki megfelelő elemet és a *dev_status* így tér vissza. Ha megtalálta, akkor a belép a 28373–28389. sorokban leírt ciklusba. A ciklusmag minden egyes végrehajtásakor egy üzenetet küld a vezérlőprocesszusnak, amelyben annak állapotát kérdezi le. Háromféle válasz várható. A *DEV_REVIVE* üzenetek kaphatjuk, ha az I/O-kérést küldő eredeti processzus előzőleg felfüggesztett állapotban volt. Ekkor a *revive* (*pipe.c;* 26146. sor) eljárás hívódik. A *DEV_IO_READY* üzenetet kaphatjuk, ha select hívást hajtott végre az eszköz. Végül *DEV_NO_STATUS* üzenetet kaphatunk, és ténylegesen ilyet várunk, de valószínűleg csak azután, hogy a másik két típus valamelyikét már megkaptuk. Ezért

a *get_more* változót használjuk arra, hogy a *DEV_NO_STATUS* üzenet eléréséig ismételje a ciklusmagot.

Amikor tényleges I/O-műveletet kell végezni, akkor a *dev_oi* (28406. sor) hívódik meg a *read_write* (25124. sor) eljárásból karakterspeciális fájl kezelésére, illetve az *rw_block* (22661. sor) eljárásból blokkspeciális fájl kezelésére. Ez standard üzenetet készít (lásd 3.17. ábra) és ezt elküldi a megfelelő eszközvezérlőnek vagy a *gen_io,* vagy a *ctty_io* meghívásával, a *dmap* táblázat *dp->dmap_driver* elemének függvényében. Amíg a *dev_io* várja a választ az eszköztől, a fájlrendszer várakozik. Nincs belső multiprogramozás. Általában ez a várakozás rövid ideig tart (kb. 50 ms). De lehetséges, hogy adat nem lesz elérhető, különösen valószínű ez terminál esetén. Ebben az esetben a válasz lehet *SUSPEND,* amely időlegesen felfüggeszti a hívó alkalmazást, de a fájlrendszer folytatja működését.

A *gen_opcl* (28455. sor) eljárást lemezes eszközökre hívja a rendszer, amelyek hajlékonylemezek, merevlemezek vagy memóriaalapú eszközök. Üzenetet állít össze, majd ugyanúgy, mint az olvasásnál és írásnál, a *dmap* táblázatból meghatározza, hogy *gen_io,* avagy *ctty_io* alkalmazandó, és ennek megfelelő üzenetet küld az eszközt vezérlő processzusnak.

Termináleszköz megnyitására a *tty_opcl* (28482. sor) függvényt hívja. Ez hívja a *gen_opcl* eljárást, miután szükség szerint módosította a jelzőket, és ha a hívást a *tty*-t a terminált hívó processzus vezérlő terminájává tette, akkor bejegyzi ezt a processzustábla *fp_tty* elemében.

A */dev/tty* egy fikció, nem tartozik egyetlen eszközhöz sem. Ez egy mágikus megnevezés, amelyet az interaktív felhasználó használhat saját terminájának megnevezésére, függetlenül attól, hogy fizikailag melyik terminált használja. A */dev/tty* megnyitására és lezárására a *ctty_ipcl* (28518. sor) eljárást hívja. Ez megvizsgálja, hogy az aktuális processzus *fp_tty* processzustábla-elemét módosította-e korábbi *ctty_opcl* hívás, kijelölve a vezérlő terminált.

A setsid rendszerhívás megköveteli a fájlrendszertől bizonyos beállítások elvégzését, amelyet a *do_setsid* (28534. sor) végez el. Ez módosítja az aktuális processzus processzustábla-elemét, bejegyezve, hogy a processzus szekcióvezető, és nincs vezérlő processzusa.

Az ioctl rendszerhívást elsődlegesen a *device.c* kezeli. Azért tárgyaljuk itt, mert szorosan kapcsolódik az eszközvezérlő csatolófelületéhez. ioctl hívásakor *do_ioctl* függvény hívódik (28554. sor), amely felépít egy üzenetet, és elküldi azt a megfelelő eszközvezérlőnek.

POSIX-kompatibilis programok számára két függvényt deklarál az *include/termios.h* termináleszközök vezérléséhez. A C programkönyvtár ioctl hívásokra fordítja le ezeket a függvényeket. Nem termináleszközök esetén több művelet megvalósítására használatos az ioctl; ezek többségét a 3. fejezetben tárgyaltuk.

A következő függvény, a *gen_io* (28575. sor) az igazi munkavégző ebben a programban. Ha az elvégzendő művelet akár megnyitás, lezárás, olvasás, írás, akár ioctl, ez a függvény hívódik meg a művelet befejezésére. Mivel a */dev/tty* nem fizikai eszköz, ezért amikor egy rá hivatkozó üzenetet kell küldeni, akkor a *ctty_io* hívás (28652. sor) megadja az eszköz helyes fő- és mellékszámát, és behelyettesíti azokat az üzenetbe az elküldés előtt. A ténylegesen használt fizikai eszköz *dmap*

táblázatbeli elemét használja a hívás megvalósítására. A MINIX 3 jelenlegi konfigurációjában a *gen_io* hívás adja az eredményt.

A *no_dev* (28677. sor) hívódik meg a nem létező eszközökre, például hálózati eszközre olyan gépen, amelyiken nincs hálózati támogatás. A visszatérési érték az *ENODEV* státusz. Ez megvéd az összeomlástól nem létező eszközre történt hivatkozás esetén.

A *device.c* utolsó függvénye a *clone_opcl* (28691. sor). Néhány eszköz megnyitásakor speciális feldolgozást igényel. Az ilyen eszközöket „klónozzák", ami azt jelenti, hogy sikeres megnyitás után helyettesítik egy új eszközzel, amelynek új egyedi mellékazonosítója van. A MINIX 3 nem használja ezt a képességet, azonban használatos hálózati eszközök engedélyezésénél. Az olyan eszköz, amely ezt igényli, a *dmap* táblázat *dmap_opcl* mezőjében a *clone_opcl* értékkel jelzi ezt. Ez a reinkarnációs szerverből *STYLE_CLONE* megadásával történő hívással érhető el. Amikor *clone_opcl* megnyit egy eszközt, akkor ugyanúgy kezdődik, mint a *gen_opcl* esetén, de egy új mellékeszközszámot adhat vissza a visszatérő üzenet *REP_STATUS* mezőjében. Ha ez történik, akkor egy új ideiglenes fájlt létesít, ha lehet új i-csomópontot létesíteni. Látható könyvtári elem nem születik. Ez nem is szükséges, mert a fájl már meg van nyitva.

Idő

Minden fájl tartalmaz három 32 bites számot az idő kezelésére. Kettő ezek közül az utolsó hozzáférést és az utolsó módosítást tárolja. A harmadik az i-csomópont állapotának utolsó módosítását rögzíti. Ez az idő a fájl majdnem minden hozzáférésekor változik, kivéve az olvasást (read) és a végrehajtást (exec). Ezeket az időket az i-csomópont tartalmazza. A hozzáférési és módosítási időket a fájl tulajdonosa vagy a szuperfelhasználó a utime rendszerhívással beállíthatja. A *time.c* program *do_utime* eljárása (28818. sor) hajtja végre a rendszerhívást, amely behozza az i-csomópontot és tárolja benne az időt. A 28848. sorban az idő módosítási igényét jelző bitet beállítja, így a rendszer nem végez költséges és redundáns *clock_time* hívást.

Amint azt az előző fejezetben láttuk, a valós időt úgy lehet meghatározni, hogy a rendszer indítása óta eltelt időt (amelyet az időzítőtaszk kezel) hozzá kell adni az indításkori valós időhöz. Az stime rendszerhívás adja meg a valós időt. A legtöbb munkát a processzuskezelő végzi, de a fájlrendszer is tárolja az indításkori időt a *boottime* globális változóban. A processzuskezelő minden stime híváskor üzenetet küld a fájlrendszernek. A fájlrendszer *do_stime* (28859. sor) eljárása aktualizálja *boottime* értékét az üzenetből.

5.7.8. Egyéb rendszerhívások

Ebben a részben áttekintünk néhány fájlt, amelyek pótlólagos rendszerhívásokat támogatnak. A következő alfejezetben tárgyalunk olyan függvényeket, amelyek a fájlrendszer általános segédprogramjai.

A *misc.c* állományban néhány olyan rendszerhíváshoz kapcsolódó eljárás kapott helyet, amely máshová egyáltalán nem illeszthető be.

A *do_getsysinfo* egy csatoló felület a sys_datacopy kernelhíváshoz. Ez az információs szerver (IS) nyomkövetésének támogatásához kell. Ez biztosítja az IS számára, hogy a fájlrendszer adatszerkezeteiről másolatot tudjon készíteni, amelyet megmutathat a felhasználónak.

A dup rendszerhívás az állományleírót duplikálja. Más szavakkal, egy olyan új állományleírót hoz létre, amely ugyanarra az állományra mutat, mint az eredeti. A hívásnak létezik egy dup2 elnevezésű változata is. A hívás mindkét változatát a *do_dup* eljárás kezeli. Ez az eljárás a régebbi bináris programok támogatása céljából került be a MINIX 3-ba. Jelenleg egyik hívás sem használatos már. A MINIX 3 C könyvtár jelenlegi változata minden olyan esetben, amikor egy C forráskódban ezen hívások valamelyikébe ütközik, az fcntl rendszerhívást hozza működésbe.

Egy megnyitott állományon elvégzendő műveletek végrehajtásának kérésére a legkedveltebb mód az fcntl rendszerhívás, amelyet a *do_fcntl* eljárás kezel. A műveletek az 5.49. ábrán bemutatott POSIX-ban definiált jelzők segítségével kerülnek kijelölésre. A hívást egy állományleíróval, egy kérési kóddal és az adott kérés által esetlegesen szükséges további operandusok átadásával kezdeményezzük. Például a régi

```
dup2(fd, fd2);
```

hívásnak megfelelő új hívás az

```
fcntl(fd, F_DUPFD, fd2);
```

formában áll elő. A kérések többsége egy jelző beolvasása vagy módosítása; éppen ezért a forráskód csupán néhány sorból áll. Az *F_SETFD* kérés például azon bit beállítását végzi el, amely egy állomány azonnali bezárását jelöli ki, ha a tulajdonos processzus egy exec hívást hajtana végre. Az *F_GETFD* kérés annak megállapítására szolgál, hogy egy állományt be kell-e azonnal zárni, miután egy exec hívás került végrehajtásra. Az *F_SETFL* és az *F_GETFL* kérések azon jelzők beállítását teszik lehetővé, amelyek egy adott állomány nem zárolt állapotát vagy a hozzáírási művelet adott állományon történő elvégezhetőségét mutatják.

Kérés kódja	Jelentés
F_DUPFD	Állományleíró másolatának létrehozása
F_GETFD	A „exec esetén bezárandó" jelző beolvasása
F_SETFD	A „exec esetén bezárandó" jelző beállítása
F_GETFL	Az állományállapot-jelzők beolvasása
F_SETFL	Az állományállapot-jelzők beállítása
F_GETLK	Egy állomány zárolási állapotának beolvasása
F_SETLK	Írás/olvasás zárolás létrehozása egy állományon
F_SETLKW	Írászárolás létrehozása egy állományon

5.49. ábra. *Az* fcntl *rendszerhívás számára szükséges POSIX-paraméterek*

A *do_fcntl* kezeli továbbá az állományzárolásokat is. Az *F_GETLK, F_SETLK* vagy *F_SETLWK* kérésekkel történő hívás átfordítódik a korábban már alaposan megtárgyalt *lock_op* eljárás meghívására.

A következő rendszerhívás a sync hívás, amely lemezre írja a memóriába töltés óta megváltozott összes blokkot és i-csomópontot. A hívást a *do_sync* eljárás dolgozza fel. Ez egyszerűen végignézi az összes táblát, hogy van-e bennük megváltozott elem. Először az i-csomópontokat kell feldolgozni, mivel az *rw_inode* a blokkgyorsítótárba irányítja kimenetét. Miután minden megváltozott i-csomópont a blokkgyorsítótárba került, az összes megváltozott blokk lemezre íródik.

A fork, exec, exit és set rendszerhívások valójában a memóriakezelő hívásai, azonban eredményüket a fájlrendszernek is el kell küldeni. Egy processzus elágazásakor lényeges, hogy erről az operációs rendszer központi része, a processzuskezelő és a fájlrendszer egyaránt értesüljenek. Ezek a „rendszerhívások" nem a felhasználói processzustól érkeznek, hanem a processzuskezelőtől. A *do_fork, do_exit* és *do_set* eljárások a processzustábla állományrendszerbeli részébe jegyzik fel a lényeges információkat. A *do_exec* kikeresi és (a *do_close* eljáráson keresztül) bezárja az „exec esetén bezárandó"-ként megjelölt valamennyi állományt.

Ezen állomány utolsó függvénye valójában nem rendszerhívás, de úgy kezeljük, mintha az lenne. Ez a *do_revive* függvény. Akkor hívjuk meg, ha egy taszk korábban nem tudta a fájlrendszer által kért munkát befejezni – például egy felhasználói processzus számára nem tudott bemeneti értékeket adni –, azonban időközben teljesítette azt. Ilyenkor a fájlrendszer újraéleszti a processzust és egy válasz üzenetet küld számára.

A *select.h* és a *select.c* a select rendszerhívásnak nyújt támogatást. A select akkor kell, amikor egy önálló processzusnak többszörös I/O-folyamot kel kezelnie, például kommunikációs vagy hálózati program. Részletesebb tárgyalása meghaladja a könyv kereteit.

5.7.9. Fájlrendszer-segédeljárások

A fájlrendszerben van néhány olyan általános célú eljárás, amelyet eltérő helyzetekben is használunk. Ezeket a *utility.c* állományban gyűjtöttük össze.

A *clock_time* a rendszertaszknak küld üzeneteket a pontos valós idő lekérdezése céljából. A *fetch_name* eljárásra azért van szükség, mert nagyon sok rendszerhívás rendelkezik paraméterként fájlnévvel. Ha a fájlnév rövid, a felhasználó által a fájlrendszernek küldött üzenetben ez szerepel. Ha azonban a fájl neve hosszú, akkor a felhasználói területen lévő nevet kijelölő mutató kerül be az üzenetbe. A *fetch_name* mindkét esetet ellenőrzi, és ilyen vagy olyan módon megszerzi a fájl nevét.

Két függvény a hibák egy általános osztályát kezeli. A *no_sys* hibakezelő akkor kerül meghívásra, ha a fájlrendszer olyan rendszerhívást kap, amely nem a sajátjai közül való. A *panic* elnevezésű pedig, ha valami katasztrofális dolog történt, kiír egy üzenetet, és az operációs rendszer központi részét a törülköző bedobására kéri.

Az utolsó két függvény, a *conv2* és a *conv4* az eltérő bájtsorrendű processzorok problémájának megoldását segítik. Ezek a rutinok olyan adatszerkezetek, mint

például egy i-csomópont vagy bittérkép, amelyek lemezről történő beolvasásakor vagy oda irányuló kiírásakor kerülnek meghívásra. A lemezt létrehozó rendszer bájtsorrendje a szuperblokkban van feljegyezve. A fájlrendszer többi részének szükségtelen bármit is tudnia a lemezen alkalmazott bájtsorrendről.

Végül tekintsük azt a két fájlt, amelyek specializált segédszolgáltatásokat nyújtanak a fájlkezelőnek. A fájlrendszer kérheti a rendszertaszkot riasztás beállítására, de ha több időpontra is be akar állítani riasztást, akkor ezeket egy saját láncolt listában kezelheti, csakúgy, mint azt az előző fejezetben láttuk a processzuskezelőnél. A *timers.c* tartalmazza ezeket a segédprogramokat. Végül a MINIX 3 egy egyedi CD-ROM-kezelést valósít meg, amely egy MINIX 3-lemezt rejt több partícióval a CD-ROM-on, és lehetővé teszi élő MINIX 3 indítását CD-ROM-ról. A MINIX 3-fájlok nem láthatók olyan operációs rendszerben, amely csak standard CD-ROM-fájlformátumokat ismernek. A *cdprobre.c* program betöltéskor megkeresi a CD-ROM-eszközt és azokat a fájlokat, amelyek a MINIX 3 betöltéséhez kellenek.

5.7.10. Egyéb MINIX 3-komponensek

Az előző fejezetben tárgyalt processzuskezelő és az ebben a fejezetben tárgyalt fájlrendszer felhasználói szerverek; ezek olyan szolgáltatásokat nyújtanak, amelyek hagyományos tervezésű rendszerek esetén egyetlen monolitikus kernelbe integrálhatók. Azonban ezek nem az egyedüli szerverek a MINIX 3-ban. Vannak más felhasználói szerverek is, amelyek rendszerjogosultsággal futnak és az operációs rendszer részének tekinthetők. Ebben a könyvben nincs hely a részletes tárgyalásukra, de legalább meg szeretnénk említeni őket.

Az egyikkel már foglalkoztunk ebben a fejezetben. Ez a reinkarnációs szerver, az RS, amely közönséges processzust indít, amelyet átvált rendszertaszkká. A MINIX 3 jelenlegi változatában arra szolgál, hogy elindítson egy olyan eszközvezérlőt, amely nem része a betöltési memóriaképnek. A későbbi változatban képes lesz vezérlő leállítására és újraindítására, és ténylegesen figyelni a vezérlőket, automatikusan leállítani és újraindítani hibás működés esetén. A reinkarnációs szerver forráskódja az *src/server/rs* könyvtárban található.

Az IS információs szervert is megemlítettük futólag. Ez nyomkövetési információkat készít, amit a PC stílusú billentyűzeten egy funkcióbillentyű lenyomásával lehet kezdeményezni. Az információs szerver forráskódja az *src/server/is* könyvtárban található.

Az információs szerver és a reinkarnációs szerver relatíve rövid programok. Van egy harmadik, nem kötelező szerver is, az INET hálózati szerver. Ez meglehetősen nagy, a mérete hozzávetőlegesen akkora, mint a MINIX 3 betöltési memóriaképéé. Ezt a reinkarnációs szerver indítja, teljesen úgy, mint az eszközvezérlőket. Az *inet* szerver forráskódja az *src/server/inet* könyvtárban található.

Végül megemlítünk egy rendszerkomponenst, amelyet eszközvezérlőnek és nem szervernek tekinthetünk. Ez a napló eszköz. Mivel az operációs rendszer nagyon sok komponense önálló processzusként fut, kívánatos adni egy egységes

eszközt a diagnosztikai, figyelmeztető- és hibaüzenetek kezelésére. A MINIX 3 eszközvezérlőt biztosít a */dev/klog* áleszközre, amely üzeneteket fogad, és kiírja azokat a fájlba. A forráskód az *src/server/log* könyvtárban található.

5.8. Összefoglalás

Kívülről nézve a fájlrendszer állományok és könyvtárak, valamint az ezeken végzett műveletek összességéből áll. Az állományok írhatók és olvashatók, a könyvtárak létrehozhatók és megszüntethetők, az állományok pedig az egyik könyvtárból a másikba mozgathatók. A legtöbb modern fájlrendszer támogatja a könyvtári rendszer hierarchikus felépítését, amelyben a könyvtáraknak végtelen sok alkönyvtáruk lehet.

Belülről nézve a fájlrendszer eléggé másként fest. A fájlrendszer tervezőinek olyan dolgokkal kellett törődniük, mint a tárolóhely lefoglalása, illetve annak nyomon követése, hogy melyik blokk melyik állományhoz tartozik. Azt is láttuk, hogyan lehet a különböző rendszereknek egymástól eltérő könyvtárszerkezete. A fájlrendszer megbízhatósága és teljesítménye szintén fontos kérdések.

A biztonság és az adatvédelem mind a felhasználók, mind a tervezők szempontjából életbe vágóan fontos. Megtárgyaltunk néhány régebbi rendszerben jelen lévő biztonsági hiányosságot és olyan általános problémát, amelyek sok rendszernél felbukkannak. Megnéztük, hogy működik a jogosultságok kezelése jelszóval, hozzáférést vezérlő listával és más módszerekkel, valamint ezek használata nélkül. Mindezeken túl, az adatvédelem kapcsán megtárgyaltuk annak egy mátrixmodelljét is.

Végezetül részletesen megvizsgáltuk a MINIX 3 fájlrendszerét, amelynek mérete ugyan óriási, ő maga azonban nem túlzottan bonyolult. Ez fogadja a felhasználói processzusok munkavégzésre irányuló kéréseit, végzi el az eljárási mutatók táblájában a megfelelő kijelöléseket, majd hívja meg azt az eljárást, amely a kért rendszerhívást végzi. Részegységekből történő felépítésének, valamint az operációs rendszer központi részén kívül való elhelyezkedésének köszönhetően a MINIX 3-ról leválasztható és kisebb módosítások elvégzése után akár egy önálló hálózatifájl-kiszolgálóként is alkalmazható.

Belülről megvizsgálva, a MINIX 3 ideiglenesen egy ún. blokkgyorsítótárban helyezi el az adatokat, és ha egy fájlt szekvenciálisan kezelünk, akkor kísérletet tesz az adatok előreolvasására. Amennyiben ennek a gyorsítótárnak elég nagy a mérete, az adott programcsoportot egymás után többször is megkérő műveleteknél, mint például fordítás során, a programkód nagy része már a memóriában lesz megtalálható.

Feladatok

1. Az NTFS Unicode fájlneveket használ. A Unicode karakterek 16 bitesek. Mi az előnye a Unicode fájlneveknek az ASCII fájlnevekkel szemben?
2. Néhány fájl elején mágikus szám található. Mi ennek a szerepe?
3. Az 5.4. ábrán néhány fájlattribútum látható. A táblázatban nem szerepel a paritás. Hasznos attribútum lenne a paritás? Ha igen, mire lehetne használni?
4. Adjon meg az */etc/passwd* fájl számára 5 különböző elérési utat. (Tanács: használja fel a . és a .. könyvtárbejegyzéseket.)
5. A soros elérésű állományokkal rendelkező rendszerek mindig rendelkeznek egy olyan művelettel, amely az állományok elejére mutat vissza. Szükséges-e ezen művelet a véletlen hozzáférésű állományokat támogató rendszerek esetében?
6. Néhány operációs rendszer a rename rendszerhívást használja egy állomány átnevezésére. Van-e valamilyen különbség, ha ezt a hívást használjuk egy állomány átnevezésekor azzal szemben, ha az állományt egy új néven másik állományba másoljuk át, majd töröljük a régit?
7. Tekintsük az 5.7. ábrán bemutatott könyvtárszerkezetet. Ha a pillanatnyi munkakönyvtár */usr/jim,* mi annak az állománynak az abszolút elérési útja, amelynek relatív elérési útja *../ast/x*?
8. Tekintsük a következő javaslatot. A fájlrendszer egyetlen gyökérkönyvtára helyett legyen minden felhasználóhoz egyedi gyökérkönyvtár. Ezzel flexibilisebbé válna a fájlrendszer? Indokolja a válaszát.
9. Unix-fájlrendszerben a chroot paranccsal a gyökérkönyvtár beállítható tetszőleges könyvtárra. Van ennek biztonsági következménye? Indokolja válaszát.
10. Unix-rendszerben beolvasható a könyvtári bejegyzés. Miért szükséges külön parancs a könyvtári bejegyzés olvasására, amikor az is csak fájl? A felhasználók maguk nem olvashatják a könyvtárakat?
11. A standard PC-k egyidejűleg legfeljebb négy operációs rendszert tartalmazhatnak. Van olyan módszer, amivel ez növelhető? Milyen következményei lennének a javaslatának?
12. Az állományok folyamatos foglalása, mint azt a szövegben említettük, a lemez felaprózódásához vezet. Vajon ez egy belső vagy külső felaprózódás? Vonjunk párhuzamot ez és egy – az előző fejezetben megtárgyalt – dolog között.
13. Az 5.10. ábra az MS-DOS eredeti FAT-fájlrendszerének szerkezetét mutatja. Eredetileg a fájlrendszer csak 4096 blokkot tartalmazott, így 4096 elemű táblázat (12 bit) elég volt. Ha ezt a sémát közvetlenül kiterjesztenénk úgy, hogy 2^{32} blokkot tartalmazzon, akkor mekkora helyet foglalna el a FAT?
14. Képzeljünk el egy olyan operációs rendszert, amely csak egyetlen könyvtár létezését támogatja, viszont ebben a könyvtárban tetszőleges számú és akármilyen hosszú névvel rendelkező állomány előfordulhat. Lehetséges-e valamilyen módon egy közelítőleg hierarchikus állományrendszert szimulálni? Ha igen, milyen módon?
15. A szabad lemezterületet egy üres listával vagy bittérképpel lehet számon tartani. A lemezcímek tárolása *D* számú bitet igényel. Tételezzük fel, hogy van egy

B blokkal rendelkező lemezünk, amelyen F üres blokk van. Határozzuk meg, milyen feltételek mellett foglal el az üres lista kevesebb helyet, mint a bittérkép. Adjuk meg a lemez tárterületének százalékában, hogy a $D = 16$ választás mellett mekkora lemezterületnek kell ehhez üresen maradnia.

16. Tételezzük fel, hogy minden Unix-állomány első részét ugyanabban a lemezblokkban tároljuk, mint a hozzá tartozó i-csomópontot. Milyen előnyünk származna ebből?

17. Egy állományrendszer teljesítménye nagyban függ a gyorsítótárban való megtalálási arány (a gyorsítótárban megtalált blokkok aránya) értékétől. Ha a gyorsítótárból történő kérés végrehajtásához 1 ms, a lemezről történő kérés végrehajtásához viszont 40 ms valamint, illetve a gyorsítótárban való megtalálási arány h, határozzunk meg egy kérés végrehajtásához szükséges átlagos időt. Ábrázoljuk ezen összefüggést h függvényében, ha h a (0, 1) intervallumban változik!

18. Mi a különbség a merev és a szimbolikus kapcsolás között? Adjuk meg mindegyik előnyét.

19. Adjunk meg három csapdát, amelyre figyelni kell fájlrendszer mentésekor.

20. Egy hajlékonylemezen 400 cilinder van, ezek mindegyike 8 sávot tartalmaz 512 bájtos blokkokkal. Egy pozicionálás cilinderenként 1 ms-ig tart. Ha egy állomány blokkjait nem próbáljuk meg egymáshoz közel elhelyezni, két logikailag egymás után következő blokk (vagyis az állományban egymás után található) átlagos pozicionálási időt igényel, ami 5 ms. Ha azonban az operációs rendszer megpróbálja az összetartozó blokkokat csoportosítva elhelyezni, akkor a blokkok között az átlagos távolság 2 cilindernyire csökkenthető, és így a pozicionálási idő 100 ms-ra csökken. Mennyi ideig tart egy 100 blokkból álló állomány beolvasása a két esetben, ha a forgási késleltetés 10 ms, az átvitelhez pedig 20 ms szükségeltetik blokkonként?

21. Van-e számottevő haszna a tárterület időnkénti tömörítésének? Indokolja válaszát.

22. Mi a különbség egy vírus és egy féreg között? Hogyan szaporodnak?

23. Diplomája megszerzése után egy olyan óriási egyetem számítóközpontjának vezetői posztját pályázza meg, amely éppen most szabadult meg ósdi operációs rendszerétől és vezette be a Unixot. Pályázata sikerrel jár. Munkakezdés után negyed órával helyettese ordítva rohan be az Ön irodájába: „Néhány hallgató rájött a jelszók titkosítására használt algoritmusunkra, és elküldte a hirdetőtáblára." Mit tenne Ön ebben a helyzetben?

24. Két számítástudományi szakos hallgató, Carolyn és Elinor, az i-csomópontokról beszélgetnek. Carolyn azt a nézetet vallja, hogy mivel a memóriák napjainkban már nagyok és olcsók, egy állomány megnyitásakor sokkal egyszerűbb az i-csomópont táblában az állomány i-csomópontjáról egy újabb másolatot elhelyezni, mint a táblát végignézni, hogy az i-csomópont nincs-e már ott. Elinor nem ért egyet ezzel a véleménnyel. Kinek van igaza?

25. Az n bit hosszúságú véletlen számokon alapuló Morris–Thompson-féle védelmi eljárást arra tervezték, hogy azzal megnehezítsék egy külső behatoló számára, hogy egyszerű karakterfüzérek előre történő titkosításával nagy mennyiségű

jelszót fejthessen meg. Védelmet nyújt-e ezen eljárás egy olyan hallgató ellen, aki a saját gépén a rendszer-adminisztrátor jelszavát próbálja meg kitalálni?

26. Egy számítástudományi tanszék helyi hálózata nagyszámú Unix-gépből áll. A felhasználók bármely gépen kiadhatják a

```
machine4 who
```

formájú parancsot, és végrehajthatják a *machine4* elnevezésű gépen anélkül, hogy oda fizikailag belépnének. Ezt úgy oldották meg, hogy a felhasználó operációs rendszerének központi része elküldi a távoli gépnek a parancsot és a felhasználó azonosítóját. Biztonságos-e ez az eljárás, ha az operációs rendszerek központi része mind megbízható (azaz védelmi hardverrel felszerelt nagy, időosztásos miniszámítógépek rendszere)? Hogyan változik a helyzet, ha a gépek közül néhány védelmi hardverrel fel nem szerelt, hallgatói személyi számítógép?

27. Egy állomány törlésekor annak blokkjai általában a szabad listába kerülnek, de nem törlődnek azonnal. Gondolja, jó ötlet volna az operációs rendszerrel az egyes blokkokat még azelőtt töröltetni, hogy azok felszabadításra kerülnének? Válaszában térjen ki mind a biztonsági, mind a teljesítménnyel kapcsolatos összetevőkre, és magyarázza meg ezek hatásait.

28. Az általunk megtárgyalt három védelmi technika a képességeket, az elérést szabályozó listák és a Unix *rwx* bitjei. Állapítsuk meg egyenként, hogy az alább felsorolt védelmi problémáknál melyik technika használható a három közül.

(a) Ken mindenki számára, kivéve munkatársait, olvashatóvá szeretné tenni állományait.

(b) Mitch és Steve szeretne néhány titkos állományt megosztani.

(c) Linda állományai egy részét nyilvánossá szeretné tenni.

A Unix esetén a csoportokat olyan osztályokként képzeljük el, mint szak, hallgatók, titkárnők és így tovább.

29. Meg tud-e bármit is támadni a trójai faló típusú vírus egy olyan rendszerben, amelynek védelme képességekre épül?

30. A *filp* tábla méretét jelenleg az *fs/const.h* definíciós állományban található *NR_FILPS* változó határozza meg. Mivel több felhasználót szeretne hálózati rendszerében elhelyezni, szeretné az *include/minix/config.h* definíciós állományban található *NR_PROCS* változó értékét megnövelni. Hogyan definiálható *NR_FILPS* az *NR_PROCS* változó függvényében?

31. Tételezzük fel, hogy egy technikai áttörésnek köszönhetően a „nem felejtő" RAM – ez áramkimaradásnál is megbízhatóan őrzi meg a benne lévő adatokat – a hagyományos RAM-hoz képest mindenfajta árbeli vagy teljesítménybeli hátrány nélkül jelenik meg a piacon. Mely pontokon befolyásolná ez az új fejlesztés a fájlrendszertervezést?

32. A szimbolikus kapcsolások olyan állományok, amelyek közvetett módon mutatnak más állományokra vagy könyvtárakra. A közönséges kapcsolatoktól eltérően, például a MINIX-ben pillanatnyilag alkalmazott közönséges kapcsolatoktól eltérően, a szimbolikus kapcsolatok saját, adatblokkot kijelölő i-csomóponttal rendelkeznek. Az adatblokk tartalmazza azon állomány elérési út-

ját, amelyhez a kapcsolatot hozzárendeltük, és az i-csomópont lehetővé teszi, hogy a kapcsolatnak eltérő tulajdonosa vagy engedélyei legyenek, mint annak az állománynak vannak, amihez azt hozzákapcsoltuk. A szimbolikus kapcsolat és az állomány vagy könyvtár, amelyre az mutat, akár különböző eszközökön is lehetnek. A szimbolikus kapcsolatok nem alkotják részét az 1990-es POSIX szabványnak, de a jövőben minden bizonnyal bekerülnek majd a POSIX-ba. Hozzunk létre szimbolikus kapcsolatokat a MINIX 3 számára.

33. A MINIX 3-ban jelenleg a 32 bites fájlpointer szabja meg a maximális fájlméretet. A jövőben 64 bites pointereket használva 2^{32}–1-nél nagyobb fájlok is lehetnek, ami azt eredményezi, hogy háromszorosan indirekt blokkok is kellenek. Módosítsuk az FS-t úgy, hogy háromszorosan indirekt blokkokat is kezeljen.

34. Mutassuk meg, hogy rendszerünk a ROBUST jelző beállításával az összeomlásokkal szemben többé-kevésbé megbízhatóbbá válik. Azt, hogy a MINIX 3 aktuális változata hogyan reagál az ilyen irányú változtatásokra, még nem tanulmányozták alaposabban; az bármelyik irányba elmozdulhat. Vizsgáljuk meg alaposan, mi történik akkor, amikor egy megváltozott tartalmú blokkot eltávolítanak a gyorsítótárból. Vegyük figyelembe, hogy egy megváltozott tartalmú adatblokk általában egy megváltozott i-csomópont, illetve bittérkép megjelenésével jár együtt.

35. Dolgozzunk ki egy olyan eljárást, amellyel lehetővé válik egy „idegen" állományrendszer támogatása, például egy MS-DOS-állományrendszernek a MINIX 3-állományrendszer valamely könyvtárához történő hozzákapcsolása.

36. Írjunk két olyan programot, C-ben vagy parancsértelmező nyelven, amelyek MINIX 3-ban rejtett csatornán üzenetet küldenek és fogadnak. Tanács: a jogosultságbit akkor is lekérdezhető, ha a fájl egyébként nem elérhető, továbbá a *sleep* paranccsal vagy rendszerhívással adott ideig várakoztatni lehet. Mérjük az átviteli arányt tétlen rendszer esetén. Aztán mesterségesen nagymértékben terheljük le a rendszert sok háttérprocesszus indításával, és ismét mérjük az átviteli arányt.

37. Implementáljuk MINIX 3-ban a közvetlen fájlt, tehát amikor az adatot magában az i-csomópontban tároljuk, amivel lemezhez fordulást takarítunk meg.

6. További irodalom

Az előző öt fejezetben jó néhány témát körbejártunk. Jelen fejezetet segítségül szánjuk mindazok számára, akik az operációs rendszerekről megszerzett tudásukat tovább szeretnék fejleszteni. A 6.1. alfejezet tartalmazza a javasolt irodalmat, míg a 6.2. alfejezetben soroltuk fel betűrendben a könyvünk megírásához felhasznált összes könyvet, valamint tudományos cikket.

A felsorolt hivatkozásokon túl az operációs rendszerekről további cikkek találhatók a kétévente megrendezett *ACM Symposium on Operating Systems Principles* (ACM), valamint az évente megrendezett *International Conference on Distributed Computing Systems* (IEEE) elnevezésű konferenciák kiadványaiban. Hasonlóan jól használható a USENIX *Symposium on Operating Systems Design and Implementation* c. kiadványa is. Az *ACM Transactions on Computer Systems* és az *Operating Systems Review* c. folyóiratokban szintén találhatunk az operációs rendszerekkel kapcsolatos munkákat.

6.1. Ajánlott irodalom

Következzenek a javasolt irodalmak fejezetenkénti bontásban.

6.1.1. Bevezetés és általános témájú munkák

Bovet és Cesati: *Understanding the Linux Kernel.* 3. kiadás
Ez talán a legjobb választás azoknak, akik a Linux-kernel belső működését szeretnék megérteni.

Brinch Hansen: *Classic Operating Systems*
Az operációs rendszerek már elég régóta jelen vannak ahhoz, hogy néhány közülük klasszikusnak számítson: rendszerek, amelyek hatására a világ máshogyan tekint a számítógépekre. A könyv 24 cikk gyűjteménye az operációs rendszerek jövőjét meghatározó rendszerekről, olyan kategóriákba gyűjtve, mint nyitott üzemű,

kötegelt, multiprogramozásos, időosztásos, PC és osztott operációs rendszerek. Akit érdekel az operációs rendszerek története, olvassa el ezt a könyvet.

Brooks: *The Mythical Man-Month: Essays on Software Engineering*
Szellemes, szórakoztató, ugyanakkor rendkívül sok információt tartalmazó könyv arról, hogyan *ne* írjunk operációs rendszert, egy olyan szerzőtől, aki már végigjárta ezt az utat. A könyv tele van hasznos tanácsokkal.

Corbató: *On Building Systems That Will Fail*
A szerző, az időosztási technika atyja, a Turing-díj átvételekor megtartott előadásában ugyanazokat a problémákat boncolgatja, amelyeket Brooks is tárgyal a *Mythical Man-Month* című könyvében. Arra a végkövetkeztetésre jut, hogy minden bonyolult rendszer végső soron megbukik, tehát ahhoz, hogy egyáltalán esélyünk legyen a sikerre, a tervezés során alapvetően el kell kerülnünk a bonyolultságot, és az egyszerűség fenntartására kell törekednünk.

Deitel: *Operating Systems.* 3. kiadás
Az operációs rendszerekről szóló általános tankönyv. A hagyományos témák mellett a Linux és a Windows XP operációs rendszerekkel kapcsolatos esettanulmányokat is tartalmaz.

Dijkstra: *My Recollections of Operating System Design*
Az operációs rendszertervezés egyik úttörőjének visszaemlékezései, kezdve azokkal az időkkel, amikor még az „operációs rendszer" kifejezés nem is létezett.

IEEE, *Information Technology – Portable Operating System Interface (POSIX), első rész: System Application Program Interface (API) [C nyelven]*
Ez a kiindulási alap. Jelentős része valójában egészen könnyen megérthető, különösen a „Rationale and Notes" címre hallgató B. függelék, amely megvilágítja, hogy miért is úgy történnek a dolgok, ahogyan. A szabványdokumentumra való hivatkozásnak van még egy előnye: ebben, definíció szerint, nincsenek hibák. Ha ugyanis egy makróutasítás nevében szereplő elgépelés a szerkesztési folyamaton túljut, akkor a továbbiakban az már nem hibaként, hanem a hivatalos változatként él tovább.

Lampson: *Hints for Computer System Design*
Butler Lampson, aki a világon egyike az újszerű operációs rendszerek vezető tervezőinek, az évek során megszerzett tapasztalataira alapozva jó néhány tippet, javaslatot, valamint útmutatást gyűjtött össze ebben a cikkében, és rendezte szórakoztató és rendkívül sok információt nyújtó módon egymás mellé. Akárcsak Brook könyvét, ezt is minden leendő operációsrendszer-tervezőnek célszerű elolvasnia.

Lewine: *POSIX Programmer's Guide*
Ez a könyv a szabványdokumentumokhoz viszonyítva sokkal olvashatóbb formában tárgyalja a POSIX szabványt. Ezen túlmenően útmutatást nyújt a régebbi

programok POSIX szabványra való átalakításához és az új programok POSIX-környezetben történő fejlesztéséhez. Számos forráskódpéldát és néhány teljes programot is tartalmaz. Részletesen bemutatja az összes POSIX által használt könyvtári függvényt és definíciós állományt is.

McKusick és Neville-Neil: *The Design and Implementation of the FreeBSD Operating System*
Ezt kell elolvasnunk, ha a modern Unix-verziók, jelen esetben a FreeBSD belső működésének alapos magyarázatára vagyunk kíváncsiak. Tartalmazza a processzszusok, az I/O, a memóriakezelés, hálózatkezelés és szinte minden más leírását.

Milojicic: *Operating Systems: Now and in the Future*
Tegyük fel, hogy feltehetünk egy sor kérdést a világ 6 vezető operációsrendszer-szakértőjének a területről és a fejlődés irányáról. Ugyanazokat a válaszokat kapjuk? *Súgunk*: nem. Nézzük meg, mit mondanak.

Ray és Ray: *Visual Quickstart Guide: Unix.* 2. kiadás
A könyvünkben szereplő példák megértését elősegíti, ha nem ismeretlen számunkra a Unix használata. Ez a sok Unix-könyv egyike, amely kezdőknek mutatja be a Unix használatát. Bár megvalósítása különbözik, a felhasználó felé a MINIX Unixnak látszik, így ez vagy egy ehhez hasonló könyv a MINIX használatában is segítséget nyújt.

Russinovich és Solomon: *Microsoft Windows Internals.* 4. kiadás
Tűnődött már valaha azon, milyen a Windows belső működése? Ne tűnődjünk tovább. Ez a könyv elmond mindent, amit feltételezhetően tudni akar a processzusokról, memóriakezelésről, I/O-ról, hálózatkezelésről, biztonságról és rengeteg másról.

Silberschatz és Galvin: *Operating System Concepts.* 7. kiadás
Egy újabb operációs rendszerekkel foglalkozó tankönyv, amely felöleli a processzszusokat, a tárkezelést, az állományokat és az osztott rendszereket. Két konkrét esetet is bemutat: a Linuxot és a Windows XP-t.

Stallings: *Operating Systems.* 5. kiadás
Még egy tankönyv az operációs rendszerekről. Tartalmazza az összes szokásos témakört, foglalkozik egy keveset az osztott rendszerekkel és egy függelék erejéig a sorban állás elméletével is.

Stevens és Rago: *Advanced Programming in the Unix Environment.* 2. kiadás
Ez a könyv bemutatja, hogyan kell olyan C programokat írni, amelyek a Unix rendszerhívási csatolófelületét és a szabvány C könyvtárakat használják. A példák a FreeBSD 5.2.1, a Linux 2.4.22, a Solaris 9, a Darwin 7.4.0 és a Mac OS X 10.3-alapú FreeBSD/Mach változataira vonatkoznak. Részletesen leírja ezen rendszer POSIX-hoz való viszonyát.

6.1.2. Processzusok

Andrews és Schneider: *Concepts and Notations for Concurrent Programming*
Ez egy bevezetés (illetve áttekintés) a processzusok, valamint a processzusok közötti kommunikáció rejtelmeibe, beleértve a tevékeny várakozást, a szemaforokat, monitorokat, az üzenetküldést és egyéb megoldásokat. A cikk ezen elvek különböző programozási nyelveken történő megvalósítását is bemutatja.

Ben-Ari: *Principles of Concurrent and Distributed Programming*
A könyv három részből áll. Az első rész többek között a kölcsönös kizárásról, a szemaforokról, a monitorokról és az étkező filozófusok problémájáról szól. A második rész az osztott számításokat tárgyalja, valamint azokat a nyelveket, amelyek hasznosak az osztott számításokhoz. A harmadik rész az együttműködés alapelveiről és megvalósításáról szól.

Bic és Shaw: *Operating System Principles*
Ez az operációs rendszerekről szóló tankönyv négy fejezetben tárgyalja a processzusokat, nemcsak a szokásos alapelveket, hanem elég sok dolgot a megvalósításról is.

Miló et al.: *Process Migration*
Ahogy a PC-klaszterek fokozatosan kiszorítják a szuperszámítógépeket, a processzusok egyik gépről másikra mozgatása (például terheléselosztáshoz) egyre fontosabbá válik. Ebben az áttekintő műben a szerzők a migráció működésének mikéntjét tárgyalják, annak előnyeivel és csapdáival együtt.

Silberschatz és Galvin: *Operating System Concepts.* 7. kiadás
A könyv 3. és 7. fejezete közötti rész a processzusok, valamint a processzusok közötti kommunikáció részleteivel foglalkozik, beleértve az ütemezés, a kritikus szekciók, a szemaforok, a monitorok és a processzusok közötti kommunikáció klasszikus problémáinak kérdéseit.

6.1.3. Bevitel/kivitel

Chen et al.: *RAID: High Performance Reliable Secondary Storage*
A ma elérhető legjobb rendszereknél a gyors I/O elérése céljából a többszörös lemezmeghajtók párhuzamos használata szokásos. A szerzők tárgyalják magát az ötletet, továbbá megvizsgálják a különböző szervezéseket a teljesítmény, az ár és a megbízhatóság szempontjából.

Coffman et al.: *System Deadlocks*
Rövid bevezető a holtpontok tárgykörébe, abba, hogy mi is okozza őket, és hogyan kerülhetők el és hogyan észlelhetők.

Corbet et al.: *Linux Device Drivers*. 3. kiadás
Ha igazán tudni akarjuk, hogyan működik az I/O, próbáljunk meghajtóprogramot írni. Ez a könyv bemutatja, hogy Linux alatt ezt hogyan tehetjük meg.
Geist és Daniel: *A Continuum of Disk Scheduling Algorithms*
Egy általánosított lemezíró/olvasó fejet mozgató kar ütemezésére közöl algoritmust a cikk. Az ezzel kapcsolatos kiterjedt szimulációs és kísérleti eredmények szintén megtalálhatók benne.

Holt: *Some Deadlock Properties of Computer Systems*
A holtpontok tárgyalása. Holt cikkében egy olyan irányított gráf modellt vezet be, amelynek alkalmazásával kielemezhető néhány holtponthoz vezető helyzet is.

IEEE *Computer* magazin, 1994. március
A *Computer* magazinnak ez a száma nyolc olyan, a fejlettebb I/O-t tárgyaló cikket közöl, amelyek lefedik a szimuláció, a nagy teljesítményű tárolás, a gyorsítótárak, a párhuzamos számítógépek I/O-műveletei és a multimédia-alkalmazások területeit.

Levine: *Defining Deadlocks*
Ebben a rövid cikkben Levine érdekes kérdéseket vet fel a holtpontok szokványos definícióival és példáival kapcsolatosan.

Swift et al.: *Recovering Device Drivers*
A meghajtóprogramok az operációs rendszer többi részéhez képest egy nagyságrenddel nagyobb hiba-előfordulással rendelkeznek. Tehetünk-e valamit a megbízhatóság javítása érdekében? A cikk bemutatja, hogyan használhatók az árnyékmeghajtók e cél elérése érdekében.

Tsegaye és Foss: *A Comparison of the Linux and Windows Device Driver Architecture*
A Linux és a Windows meglehetősen különböző meghajtóprogram-architektúrával rendelkezik. A cikk mindkettőt tárgyalja, bemutatva, miben hasonlítanak és miben térnek el.

Wilkes et al.: *The HP AutoRAID Hierarchical Storage System*
A nagy teljesítményű merevlemez-rendszereknél egy olyan fontos, új fejlesztés a RAID („olcsó lemezek redundáns tömbje"), amelyben kicsi lemezek egész sora dolgozik együtt, hogy biztosítsa a rendszer óriási sávszélességét. Ebben a cikkben a szerzők a HP laborjaiban megépített rendszerük néhány részletét teszik közzé.

6.1.4. Memóriakezelés

Bic és Shaw: *Operating System Principles*
A könyv három fejezetet szentel a memóriakezelés, a fizikai memória, a virtuális memória és az osztott memória tárgyalására.

Denning: *Virtual Memory*
A virtuális memória különböző tulajdonságait vizsgáló, klasszikusnak számító cikk. Denning egyike volt ezen terület úttörőinek; az ő nevéhez fűződik a munkahalmaz elvének megalkotása.

Denning: *Working Sets Past and Present*
Számos memóriakezelő és lapozó algoritmus remek áttekintése. A cikk átfogó irodalomjegyzéket is tartalmaz.

Denning: *The Locality Principle*
Egy újabb keletű visszatekintés a lokalitás alapelvének történetére és a memórialapozáson túli alkalmazhatóságainak tárgyalása.

Halpern: *VIM: Taming Software with Hardware*
Ebben a provokatív cikkben Halpern azt bizonygatja, hogy óriási összegeket költenek szoftverek gyártására, hibakeresésére és fenntartására, amelyek memóriaoptimalizálással járnak együtt nemcsak az operációs rendszerek, de fordítók és más szoftverek szintjén is. Továbbá, hogy makroökonómiai szempontból jobb lenne ezt a pénzt több memória vásárlására költeni az egyszerű, egyértelmű és megbízható szoftverek elérése érdekében.

Knuth: *The Art of Computer Programming*. 1. kötet.
Ebben a munkában az „először megfelelő", a „legjobban illeszkedő" és egyéb, memóriakezeléssel kapcsolatos algoritmusokat tárgyalja, valamint hasonlítja össze a szerző.

Silberschatz et al.: *Operating System Concepts*. 7. kiadás
A könyv 8. és 9. fejezete foglalkozik a memóriakezeléssel, beleértve a memória és a háttértároló közötti adatmozgatást, a lapozást és a szegmentálást. Az írók a lapozó algoritmusok több változatát is bemutatják.

6.1.5. Fájlrendszerek

Denning: *The United States vs. Craig Neidorf*
Miután egy fiatal számítógépes kalóz rájött, hogyan működik a telefonhálózat, és erről adatokat is nyilvánosságra hozott, számítógépes csalással vádolták meg. A cikk ezzel az üggyel foglalkozik, amely olyan alapvető kérdéseket hozott a felszínre, mint például a szólásszabadság kérdése. A cikk megjelenése után eltérő véleményeket, valamint Denning részéről egy kemény hangú választ váltott ki.

Ghemawat et al.: *The Google File System*
Tegyük fel, hogy elhatározzuk, az egész internetet otthon szeretnénk tárolni, hogy igazán gyorsan találjunk meg dolgokat. Hogyan csinálnánk? Az első lépés mondjuk 200 000 PC vásárlása lehetne, akármilyen PC-k megteszik. A második lépés pedig e cikk elolvasása lenne, hogy csinálja ezt a Google.

Hafner és Markoff: *Cyberpunk: Outlaws and Hackers on the Computer Frontier*
A világ különböző számítógépes rendszereibe behatoló fiatal, számítógépes kalózokról szóló három történet a *New York Times* számítógépekkel foglalkozó tudósítójának, aki az interneten terjedő, illegális behatoló programokról is írt, valamint társszerzőjének tollából.

Harbron: *File Systems: Structures and Algorithms*
A fájlrendszerek tervezéséről, alkalmazásáról és teljesítményéről szóló könyv. A fájlrendszerek szerkezetét és a kapcsolódó algoritmusokat egyaránt tárgyalja.

McKusick et al.: *A Fast File System for Unix*
A Unix-fájlrendszert teljesen átdolgozták a 4.2 BSD változatban. A cikk összefoglalja az új fájlrendszer tervezését és felépítését, különös hangsúllyal a fájlrendszer teljesítményére.

Satyanarayanan: *The Evolution of Coda*
A mobil számítástechnika terjedésével a fix rendszerek és a mobil eszközök közötti integráció és szinkronizáció megoldása egyre sürgetőbb. A Coda volt az egyik úttörő ezen a területen. Ennek fejlődését és működését írja le a könyv.

Silberschatz et al.: *Operating System Concepts*. 7. kiadás
A könyv 10. és 11. fejezete foglalkozik a fájlrendszerekkel, beleértve többek között a fájlokkal végzett műveleteket, az elérési módozatokat, a kompatibilitási szemantikát, a könyvtárakat, a védelmi eljárásokat és ezek megvalósítását.

Stallings: *Operating Systems*. 5. kiadás
A könyv 16. fejezete részletesen megtárgyalja a biztonsággal kapcsolatos témákat, különös tekintettel a számítógépes kalózokra, a vírusokra és egyéb veszélyekre.

Uppuluri et al.: *Preventing Race Condition Attacks on File Systems*
Vannak helyzetek, amikor egy processzus azt feltételezi, hogy két művelet végrehajtása oszthatatlan, nincs közbeeső művelet-végrehajtás. Ha egy másik processzus mégis behatol ezek közé, és végrehajt egy másik műveletet, a biztonság sérülhet. Ez a cikk a problémát és annak javasolt megoldását tárgyalja.

Yang et al.: *Using Model Checking to Find Serious File System Errors*
A fájlrendszer hibái adatvesztéshez vezethetnek, így felfedezésük rendkívül fontos. A cikk egy formális technikát ír le, amely segít e hibák felderítésében, még mielőtt bármi kárt okoznának. Egy tényleges fájlrendszerkódon lefuttatott modellellenőrzés eredményét is bemutatják.

6.2. Betűrendes irodalomjegyzék

ANDERSON, T.E., BERSHAD, B.N., LAZOWSKA, E.D. és LEVY, H.M.: Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism, *ACM Trans. on Computer Systems,* vol. 10, pp. 53-79, 1992. febr.

ANDREWS, G.R. és SCHNEIDER, F.B.: Concepts and Notations for Concurrent Programming, *Computing Surveys,* vol. 15, pp. 3-43, 1983. márc.

AYCOCK, J. és BARKER, K.: Viruses 101, *Proc. Tech. Symp. on Comp. Sci. Education,* ACM, pp. 152-156, 2005.

BACH, M.J.: *The Design of the Unix Operating System,* Upper Saddle River, NJ: Prentice Hall, 1987.

BALA, K., KAASHOEK, M.E és WEIHL, W.: Software Prefetching and Caching for Translation Lookaside Buffers, *Proc. First Symp. on Oper. Syst. Design and Implementation,* USENIX, pp. 243-254, 1994.

BASILI, V.R. és PERRICONE, B.T.: Software errors and Complexity: An Empirical Investigation, *Commun. of the ACM,* vol. 27, pp. 43-52, 1984. jan.

BAYS, C.: A Comparison of Next-Fit, First-Fit, and Best-Fit, *Commun. of the ACM,* vol. 20, pp. 191-192, 1977. márc.

BEN-ARI, M: *Principles of Concurrent and Distributed Programming,* Upper Saddle River, NJ: Prentice HaJl, 1990.

HIC, L.F. és SHAW, A.C.: *Operating System Principles,* Upper Saddle River, NJ: Prentice Hall, 2003.

BOEHM, H.-J.: Threads Cannot be Implemented as a Library, *Proc. 2004 ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.,* ACM, pp. 261-268, 2005.

BOVET, D.P. és CESATI, M.: *Understanding the Linux Kernel,* 2. kiad., Sebastopol, CA, O'Reilly, 2002.

BRINCH HANSEN, P.: *Operating System Principles* Upper Saddle River, NJ: Prentice Hall, 1973.

BRINCH HANSEN, P.: *Classic Operating Systems,* New York: Springer-Verlag, 2001.

BROOKS, F. P., Jr.: *The Mythical Man-Month: Essays on Software Engineering,* Anniversary Ed., Boston: Addison-Wesley, 1995.

CERF, V.G.: Spam, Spim, and Spit, *Commun. of the ACM,* vol. 48, pp. 39-43, 2005. ápr.

CHEN, H, WAGNER, D. és DEAN, D.: Setuid Demystified, *Proc. 11th USENIX Security Symposium,* pp. 171-190, 2002.

CHEN, P.M., LEE, E.K., GIBSON, G.A., KATZ, R.H. és PATTERSON, D.A.: RAID: High Performance Reliable Secondary Storage, *Computing Surveys,* vol. 26, pp. 145-185, 1994. jún.

CHERITON, D.R: An Experiment Using Registers for Fast Message-Based Interprocess Communication, *Operating Systems Review,* vol. 18, pp. 12-20, 1984. okt.

CHERVENAK, A., VELLANSKI, V. és KURMAS, Z.: Protecting File Systems: A Survey of Backup Techniques, *Proc. 15th Symp. on Mass Storage Systems,* IEEE, 1998

CHOU, A., YANG, J.-F., CHELF, B. és HALLEM, S.: An Empirical Study of Operating System Errors, *Proc. 18th Symp. on Oper. Syst. Prin.,* ACM, pp. 73-88, 2001.

COFFMAN, E.G., ELPHICK, M.J. és SHOSHANI, A.: System Deadlocks, *Computing Surveys,* vol. 3, pp. 67-78, 1971. jún.

CORBATÓ, F.J.: On Building Systems That Will Fail, *Commun. of the ACM,* vol. 34, pp. 72-81, 1991. szept.

CORBATÓ, E.J., MERWIN-DAGGETT, M. és DALEY, RC: An Experimental TimeSharing System, *Proc. AFIPS Fall Joint Computer Conf,* AFIPS, pp. 335-344, 1962.

CORBATÓ, F.J., SALTZER, J.H. és CLINGEN, C.T.: MULTICS-The First Seven Years, *Proc. AFlPS Spring Joint Computer Cant,* AFIPS, pp. 571-583, 1972.

CORBATÓ, F.J. és VYSSOTSKY, V.A.: Introduction and Overview of the MULTICS System, *Proc. AFIPS Fall Joint Computer Conf,* AFIPS, pp. 185-196, 1965.

CORBET, J., RUBINI, A. és KROAH-HARTMAN, G.: *Linux Device Drivers,* 3rd Ed. Sebastopol, CA: O'Reilly, 2005.

COURTOIS, P.J., HEYMANS, F. és PARNAS, D.L.: Concunent Control with Readers and Writers, *Commun. of the ACM,* vol. 10, pp. 667-668, 1971. okt.

DALEY, RC. és DENNIS, J.B.: Virtual Memory, Processes, and Sharing in MULTICS, *Commun. of the ACM,* vol. 11, pp. 306-312, 1968. máj.

DEITEL, H.M., DEITEL, P. J. és CHOFFNES, D. R : *Operating Systems,* 3rd Ed., Upper Saddle River, NJ: Prentice-Hall, 2004.

DENNING, D.: The United states vs. Craig Neidorf, *Commun. of the ACM,* vol. 34, pp. 22-43, 1991. márc.

DENNING, P.J.: The Working Set Model for Program Behavior, *Commun. of the ACM,* vol. 11, pp. 323-333, 1968a.

DENNING, P.J.: Thrashing: Its Causes and Prevention, *Proc. AFIPS National Computer Conf,* AFIPS, pp. 915-922, 1968b.

DENNING, P.J.: Virtual Memory, *Computing Surveys,* vol. 2, pp. 153-189, Sept. 1970. DENNING, P.J.: Working Sets Past and Present, *IEEE Trans. on Software Engineering,* vol. SE-6, pp. 64-84, 1980. jan.

DENNING, P.J.: The Locality Principle, *Commun. of the ACM,* vol. 48, pp. 19-24, 2005. júl.

DENNIS, J.B. és VAN HORN, E.C.: Programming Semantics for Multiprogrammed Computations, *Commun. of the ACM,* vol. 9, pp. 143-155, 1966. márc.

DIBONA, C., OCKMAN, S. és STONE, M. eds.: *Open Sources: Voices from the Open Source Revolution,* Sebastopol, CA: O'Reilly, 1999.

DIJKSTRA, E.W.: Co-operating Sequential Processes, in *Programming Languages,* Genuys, F. (Ed.), London: Academic Press, 1965.

DIJKSTRA, E.W.: The Structure of THE Multiprogramming System, *Commun. of the ACM,* vol. 11, pp. 341-346, 1968. máj.

DIJKSTRA, E.W.: My Recollections of Operating System Design, *Operating Systems Review,* vol. 39, pp. 4-40, 2005. ápr.

DODGE, c., IRVINE, c. és NGUYEN, T.: A Study of Initialization in Linux and OpenBSD, *Operating Systems Review,* vol. 39, pp. 79-93, 2005. ápr.

ENGLER, D., CHEN, D.Y. és CHOU, A.: Bugs as Inconsistent Behavior: A General Approach to Inferring Errors in Systems Code, *Proc. 18th Symp. on Oper. Syst. Prin.,* ACM, pp. 57-72, 2001.

ENGLER, D.R., KAASHOEK, M.F. és O'TOOLE, J. Jr.: Exokernel: An Operating System Architecture for Application-Level Resource Management, *Proc. 15th Symp. on Oper. Syst. Prin.,* ACM, pp. 251-266, 1995.

FABRY, R.S.: Capability-Based Addressing, *Commun. of the ACM,* vol. 17, pp. 403-412, 1974. júl.

FEELEY, M.J., MORGAN, W.E., PIGIDN, EH., KARLIN, A.R., LEVY, H.M. és THEK· KATH, C.A.: Implementing Global Memory Management in a Workstation Cluster, *Proc. 15th Symp. on Oper. Syst. Prin.,* ACM, pp. 201-212, 1995.

FEUSTAL, E.A.: The Rice Research Computer-A Tagged Architecture, *Proc. AFIPS Conf 1972.*

FOTHERINGHAM, J.: Dynamic Storage Allocation in the Atlas Including an Automatic Use of a Backing Store, *Commun. of the ACM,* vol. 4, pp. 435-436, 1961. okt.

GARFINKEL, S.L. és SHELAT, A.: Remembrance of Data Passed: A Study of Disk Sanitization Practices, *IEEE Security & Privacy,* vol. 1, pp. 17-27, 2003. jan.–febr.

GEIST, R. és DANIEL, S.: A Continuum of Disk Scheduling Algorithms, *ACM Trans. on Computer Systems,* vol. 5, pp. 77-92, 1987. febr.

GHEMAWAT, S., GOBI OFF, H. és LEUNG, S.-T.: The Google File System, *Proc. 19th Symp. on Oper. Syst. Prill.,* ACM, pp. 29-43, 2003.

GRAHAM, R.: Use of High-Level Languages for System Programming, Project MAC Report TM-13, M.I.T., 1970. szept.

HAFNER, K. és MARKOFF, J.: *Cyberpunk: Outlaws and Hackers on the Computer Frontier,* New York: Simon and Schuster, 1991.

HALPERN, M.: VIM: Taming Software with Hardware, *IEEE Computer,* vol. 36, pp. 21-25, 2003. okt.

HARBRON, T.R.: *File Systems: Structures and Algorithms,* Upper Saddle River, NJ: Prentice Hall, 1988.

HARRIS, S., HARPER, A., EAGLE, C., NESS, J. és LESTER, M.: *Gray Hat Hacking: The Ethical Hacker's Handbook,* New York: McGraw-Hill Osborne Media, 2004.

HAUSER, C., JACOBI, C., THEIMER, M., WELCH, B. és WEISER, M.: Using Threads in Interactive Systems: A Case Study, *Proc. 14th Symp. on Oper. Syst. Prin.,* ACM, pp. 94-105, 1993.

HEBBARD, B. et al.: A Penetration Analysis of the Michigan Terminal System, *Operating Systems Review,* vol. 14, pp. 7-20, 1980. jan.

HERBORTH, C.: *Unix Advanced: Visual Quickpro Guide,* Berkeley, CA: Peachpit Press, 2005.

HERDER, J.N.: Towards a True Microkernel Operating System, M.S. Thesis, Vrije Universiteit, Amsterdam, 2005. febr.

HOARE, C.A.R.: Monitors, An Operating System Structuring Concept, *Commun. of the ACM,* vol. 17, pp. 549-557, Oct. 1974; Erratum in *Commun. o.fthe ACM,* vol. 18, p. 95, 1975. febr.

HOLT, R.C: Some Deadlock Properties of Computer Systems, *Computing Surveys,* vol. 4, pp. 179-196, 1972. szept.

HUCK, J. és HAYS, J.: Architectural Support for Translation Table Management in Large Address Space Machines, *Proc. 20th Annual Int'l Symp. on Computer Arch.,* ACM, pp. 39-50, 1993.

HUTCHINSON, N.C., MANLEY, S., FEDERWISCH, M., HARRIS, G., HITZ, D, KLEIMAN, S és O'MALLEY, S.: Logical vs. Physical File System Backup, *Proc. Third USENIX Symp. on Oper. Syst. Design and Implementation,* USENIX, pp. 239-249, 1999.

IEEE: *Information technology-Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language],* New York: IEEE, 1990.

JACOB, B. és MUDGE, T.: Virtual Memory: Issues of Implementation, *IEEE Computer,* vol. 31, pp. 33-43, 1998. jún.

JOHANSSON, J. és RILEY, S: *Protect Your Windows Network: From Perimeter to Data,* Boston: Addison-Wesley, 2005.

KERNIGHAN, B.W. és RITCHIE, D.M.: *The C Programming Language,* 2. kiad., Upper Saddle River, NJ: Prentice Hall, 1988.

KLEIN, D.V.: Foiling the Cracker: A Survey of, and Improvements to, Password Security, *Proc. Unix Security Workshop JI,* USENIX, 1990. aug.

KLEINROCK, L.: *Queueing Systems, Vol.* 1, New York: John Wiley, 1975.

KNUTH, D.E.: *The Art of Computer Programming, Volume 1: Fundamental Algorithms,* 3rd Ed., Boston: Addison-Wesley, 1997.

LAMPSON, B.W.: A Scheduling Philosophy for Multiprogramming Systems, *Commun. of the ACM,* vol. 11, pp. 347-360, 1968. máj.

LAMPSON, B.W.: A Note on the Confinement Problem, *Commun. of the ACM,* vol. 10, pp. 613-615, 1973. okt.

LAMPSON, B.W.: Hints for Computer System Design, *IEEE Software,* vol. 1, pp. 11-28, 1984. jan.

LEDIN, G., Jr.: Not Teaching Viruses and Worms is Harmful, *Commun. of the ACM,* vol. 48, p. 144, 2005. jan.

LESCHKE, T.: Achieving Speed and Flexibility by Separating Management from Protection: Embracing the Exokernel Operating System, *Operating Systems Review,* vol. 38, pp. 5-19, 2004. okt.

LEVINE, G.N.: Defining Deadlocks, *Operating Systems Review* vol. 37, pp. 54-64, 2003a. jan.

LEVINE, G.N.: Defining Deadlock with Fungible Resources, *Operating Systems Review,* vol. 37, pp. 5-11, 2003b. júl.

LEVINE, G.N.: The Classification of Deadlock Prevention and Avoidance is Erroneous, *Operating Systems Review,* vol. 39, 47-50, 2005. ápr.

LEWINE, D.: *POSIX Programmer's Guide,* Sebastopol, CA: O'Reilly & Associates, 1991.

LI, K. és HUDAK, P.: Memory Coherence in Shared Virtual Memory Systems, *ACM Trans. on Computer Systems,* vol. 7, pp. 321-359, 1989. nov.

LINDE, R.R.: Operating System Penetration, *Proc. AFIPS National Computer Conf,* AFIPS, pp. 361-368, 1975.

LIONS, J.: *Lions' Commentary on Unix 6th Edition, with Source Code,* San Jose, CA: Peer-to-Peer Communications, 1996.

MARSH, B.D., SCOTT, M.L., LEBLANC, T.J. és MARKATOS, E.P.: First-Class UserLevel Threads, *Proc. 13th Symp. on Oper Syst. Prin.,* ACM, pp. 110-121, 1991.

McHUGH, J.A.M. és DEEK, E.P.: An Incentive System for Reducing Malware Attacks, *Commun. of the ACM,* vol. 48, pp. 94-99, 2005. jún.

McKUSICK, M.K., JOY, W.N., LEFFLER, S.J. és FABRY, R.S.: A Fast File System for Unix, *ACM Trans. on Computer Systems,* vol. 2, pp. 181-197, 1984. aug.

McKUSICK, M.K. és NEVILLE-NEIL, G.V.: *The Design and Implementation of the FreeBSD Operating System,* Addison-Wesley: Boston, 2005.

MILO, D., DOUGLIS, F., PAINDA VEINE, Y, WHEELER, R. és ZHOU, S.: Process Migration, *ACM Computing Surveys,* vol. 32, pp. 241-299, 2000. júl.–szept.

MILOJICIC, D.: Operating Systems: Now and in the Future, *IEEE Concurrency,* vol. 7, pp. 12-21, 1999. jan.–márc.

MOODY, G.: *Rebel Code* Cambridge, MA: Perseus, 2001.

MORRIS, R. és THOMPSON, K.: Password Security: A Case History, *Commun. of the ACM,* vol. 22, pp. 594-597, 1979. nov.

MULLENDER, S.J. és TANENBAUM, A.S.: Immediate Files, *Software-Practice and Experience,* vol. 14, pp. 365-368, 1984. ápr.

NAUGHTON, J.: A *Brief History of the Future,* Woodstock, NY: Overlook Books, 2000.

NEMETH, E., SNYDER, G., SEEBASS, S. és HEIN, T. R.: *Unix System Administation,* 3rd Ed., Upper Saddle River, NJ, Prentice Hall, 2000.

ORGANICK, E.I.: *The Multics System,* Cambridge, MA: M.I.T. Press, 1972.

OSTRAND, T.J., WEYUKER, E.J. és BELL, R.M.: Where the Bugs Are, *Proc. 2004 ACM Symp. on Softw. Testing and Analysis,* ACM, 86-96, 2004.

PETERSON, G.L.: Myths about the Mutual Exclusion Problem, *Information Processing Letters,* vol. 12, pp. 115-116, 1981. jún.

PRECHELT, L.: An Empirical Comparison of Seven Programming Languages, *IEEE Computer,* vol. 33, pp. 23-29, 2000. okt.

RAY, D.S. és RAY, E.J.: *Visual Quickstart Guide: Unix,* 2. kiad., Berkeley, CA: Peachpit Press, 2003.

ROSENBLUM, M. és OUSTERHOUT, J.K.: The Design and Implementation of a LogStructured File System, *Proc. 13th Symp. on Oper. Syst. Prin.,* ACM, pp. 1-15, 1991.

RUSSINOVICH, M.E. és SOLOMON, D.A.: *Microsoft Windows Internals,* 4. kiad., Redmond, W A: Microsoft Press, 2005.

SALTZER, J.H.: Protection and Control of Information Sharing in MULTICS, *Commun. of the ACM,* vol. 17, pp. 388-402, 1974. júl.

SALTZER, J.H. és SCHROEDER, M.D.: The Protection of Information in Computer Systems, *Proc. IEEE,* vol. 63, pp. 1278-1308, 1975. szept.

SALUS, P.H.: A *Quarter Century of Unix,* Boston: Addison-Wesley, 1994.

SANDHU, R.S.: Lattice-Based Access Control Models, *Computer,* vol. 26, pp. 9-19, 1993. nov.

SATYANARAYANAN, M.: The Evolution of Coda, *ACM Trans. on Computer Systems,* vol. 20, pp. 85-124, 2002. máj.

SEA WRIGHT, L.H. és MACKINNON, R.A.: VM/370-A Study of Multiplicity and Usefulness, *IBM Systems Journal,* vol. 18, pp. 4-17, 1979.

SILBERSCHATZ, A., GALVIN, P.B. és GAGNE, G.: *Operating System Concepts,* 7. kiad., New York: John Wiley, 2004.

STALLINGS, W.: *Operating Systems,* 5. kiad., Upper Saddle River, NJ: Prentice Hall, 2005.

STEVENS, W.R és RAGO, S. A.: *Advanced Programming in the Unix Environment,* 2. kiad., Boston: Addison-Wesley, 2005.

STOLL, C.: *The Cuckoo's Egg: Tracking a Spy through the Maze of Computer Espionage,* New York: Doubleday, 1989.

SWIFT, M.M., ANNAMALAI, M., BERSHAD, B.N. és LEVY, H.M.: Recovering Device Drivers, *Proc. Sixth Symp. on Oper Syst. Design and Implementation,* USENIX, pp. 1-16, 2004.

TAI, K.C. és CARVER, R.H.: VP: A New Operation for Semaphores, *Operating Systems Review,* vol. 30, pp. 5-11, 1996. júl.

TALLURI, M. és IDLL, M.D.: Surpassing the TLB Performance of Superpages with Less Operating System Support, *Proc. Sixth Int'l Conf on Architectural Support for Progr. Lang. and Operating Systems,* ACM, pp. 171-182, 1994.

TALLURI, M., HILL, M.D. és KHALIDI, Y.A.: A New Page Table for 64-bit Address Spaces, *Proc. 15th Symp. on Oper. Syst. Prin.,* ACM, pp. 184-200, 1995.

TANENBAUM, A.S.: *Modern Operating Systems,* 2. kiad., Upper Saddle River: NJ, Prentice Hall, 2001.

TANENBAUM, A.S., VAN RENESSE, R., STAVEREN, H. VAN, SHARP, G.J., MULLENDER, S.J., JANSEN, J. és ROSSUM, G. VAN: Experiences with the Amoeba Distributed Operating System, *Commun. of the ACM,* vol. 33, pp. 46-63, 1990. dec.

TANENBAUM, A.S. és VAN STEEN, M.R.: *Distributed Systems: Principles and Paradigms,* Upper Saddle River, NJ, Prentice Hall, 2002.

TEORY, T.J.: Properties of Disk Scheduling Policies in Multiprogrammed Computer Systems, *Proc. AFIPS Fall Joint Computer Conf,* AFIPS, pp. 1-11, 1972

THOMPSON, K.: Unix Implementation, *Bell System Technical Journal,* vol. 57, pp. 1931-1946, 1978. júl.–aug.

TREESE, W.: The State of Security on the Internet, *NetWorker,* vol. 8, pp. 13-15, 2004. szept.

TSEGAYE, M. és FOSS, R.: A Comparison of the Linux and Windows Device Driver Architectures, *Operating Systems Review,* vol. 38, pp. 8-33, 2004. ápr.

UHLIG, R, NAGLE, D., STANLEY, T, MUDGE, T., SECREST, S. és BROWN, R: Design Tradeoffs for Software-Managed TLBs, *ACM Trans. on Computer Systems,* vol. 12, pp. 175-205, 1994. aug.

UPPULURI, P., JOSHI, U. és RAY, A.: Preventing Race Condition Attacks on File Systems, *Proc. 2005 ACM Symp. on Applied Computing,* ACM, pp. 346-353, 2005.

VAHALIA, U.: *Unix Internals-The New Frontiers,* 2. kiad., Upper Saddle River, NJ: Prentice Hall, 1996.

VOGELS, W.: File System Usage in Windows NT 4.0, *Proc. ACM Symp. on Operating System Principles,* ACM, pp. 93-109, 1999.

WALDSPURGER, C.A. és WEIHL, W.E.: Lottery Scheduling: Flexible Proportional-Share Resource Management, *Proc. First Symp. on Oper. Syst. Design and Implementation,* USENIX, pp. 1-11, 1994.

WEISS, A.: Spyware Be Gone, *NetWorker,* vol. 9, pp. 18-25, 2005. márc.

WILKES, J., GOLDING, R., STAELIN, C. és SULLIVAN, T.: The HP AutoRAID Hierarchical Storage System, *ACM Trans. on Computer Systems,* vol. 14, pp. 108-136, 1996. febr.

WULF, W.A., COHEN, E.S., CORWIN, W.M., JONES, A.K., LEVIN, R., PIERSON, C. és POLLACK, F.J.: HYDRA: The Kernel of a Multiprocessor Operating System, *Commun. of the ACM,* vol. 17, pp. 337-345, 1974. jún.

YANG, J., TWOHEY, P., ENGLER, D. és MUSUVATHI, M.: Using Model Checking to Find Serious File System Errors, *Proc. Sixth Symp. on Oper. Syst. Design and Implementation,* USENIX, 2004.

ZEKAVSKAS, M.J., SAWDON, W.A. és BERSHAD, B.N.: Software Write Detection for a Distributed Shared Memory, *Proc. First Symp. on Oper. Syst. Design and Implementation,* USENIX, pp. 87-100, 1994.

ZWICKY, E.D.: Torture-Testing Backup and Archive Programs: Things You Ought to Know but Probably Would Rather Not, *Prof Fifth Cont on Large Installation Systems Admin.,* USENIX, pp. 181-190, 1991.

Függelék

F.1. A MINIX 3 telepítése

A függelék elmagyarázza a MINIX 3 telepítésének mikéntjét. A teljes MINIX 3-telepítéshez Pentium processzoros gépre van szükség, amely legalább 16 MB RAM-mal, 1 GB szabad lemezterülettel, IDE CD-ROM-mal és IDE merevlemezzel rendelkezünk. A minimális telepítéshez (amely nem tartalmazza a parancsok forráskódját) 8 MB RAM és 50 MB szabad lemezterület szükséges. Soros ATA (SATA), USB és SCSI lemezek pillanatnyilag nem támogatottak. Az USB CD-ROM-ok használatának leírását a *www.minix3.org* honlapon találjuk.

F.1.1. Előkészület

Ha rendelkezésünkre áll a telepítő CD-ROM (a könyvből), az első két lépést átugorhatjuk, de tanácsos ellenőrizni a *www.minix3.org* oldalon, hogy van-e elérhető újabb változat. Ha natív mód helyett szimulátoron kívánjuk a MINIX 3-at futtatni, akkor az F.5 részt olvassuk el először. Ha nem áll rendelkezésünkre IDE CD-ROM, akkor vagy szerezzük be a speciálisan USB CD-ROM-okhoz készült betöltési memóriatérképet (boot image), vagy használjunk szimulátort.

1. Töltsük le a MINIX 3 CD-ROM-képet
A MINIX 3 CD-ROM-képet a MINIX 3 honlapjáról, a *www.minix3.org* oldalról tölthetjük le.

2. Készítsük el az indítható MINIX 3 CD-ROM-ot
Csomagoljuk ki a letöltött fájlt. Egy *.iso* kiterjesztésű CD-ROM-képet és ezt a leírást fogjuk kapni. Az *.iso* fájl bitről bitre egy CD-ROM-képnek felel meg. Írjuk ki CD-ROM-ra, hogy indítható CD-ROM-ot kapjunk.

Ha az *Easy CD Creator 5* programot használjuk, akkor válasszuk ki a „Record CD from CD image" menüpontot a File menüből és a megjelenő dialógusablak-

ban a fájltípust változtassuk *.cif*-ről *.iso*-ra. Válasszuk ki a képfájlt és kattintsunk az „Open"-re. Ezután kattintsunk a „Start Recording" gombra.

Ha a *Nero Express 5* programot használjuk, akkor válasszuk a „Disc Image or Saved Project" lehetőséget, és változtassuk a típust „Image Files"-ra, válasszuk ki a képfájlt, és kattintsunk az „Open"-re. Válasszuk ki a CD-írót, és kattintsunk a „Next"-re.

Ha Windows XP-rendszert használjuk, és nem áll rendelkezésünkre CD-író program, akkor az *alexfeinman.brinkster.net/isorecorder.html* oldalon találunk egy ingyenesen használható szoftvert, amellyel ki tudjuk írni a CD-képet.

3. Határozzuk meg, hogy milyen Ethernet-lapkával rendelkezünk

A MINIX 3 számos Ethernet-lapkát támogat LAN, ADSL és kábelszolgáltatón keresztüli hálózatelléréshez. A támogatottak közé tartozik például az Intel Pro/100, a RealTek 8029 és 8139, az AMD LANCE és több 3Com-lapka. A telepítés közben a telepítő megkérdezi, hogy rendelkezünk-e, és ha igen, milyen lapkával? Derítsük ki ezt már most a gépünkhöz tartozó dokumentációból. Rendelkezésünkre áll egy másik megoldás is, ha Windowst használunk. Indítsuk el az Eszközkezelőt (Device Manager):

Windows 2000: Start > Beállítások > Vezérlőpult > Rendszer > Hardver > Eszközkezelő
Windows XP: Start > Vezérlőpult > Rendszer > Hardver > Eszközkezelő

Ezek közül a „Rendszer" kiválasztásához dupla kattintásra, a többihez egyszeres kattintásra van szükség. A „Hálózati kártyák" melletti + jel kibontásával megkapjuk a kártya típusát. Írjuk ezt le. Ha nincs, vagy nem támogatott a lapkánk, akkor is futtathatjuk a MINIX 3-at, de Ethernet nélkül.

4. Particionáljuk a merevlemezt

Ha kívánjuk, a MINIX 3-at indíthatjuk a CD-ROM-ról is, de ha bármi hasznosat szeretnénk vele csinálni, akkor hozzunk létre egy külön partíciót a merevlemezen.

A particionálás előtt elővigyázatosságból mindenképpen **mentsük az adatainkat külső adathordozóra, CD-ROM-ra vagy DVD-re**. Fájljaink értékesek, védjük őket.

Hacsak nem vagyunk nagy tapasztalattal rendelkező szakértők a lemezparticionálás terén, olvassuk el a *www.minix3.org/doc/partitions.html* oldalon található lemezparticionálási oktatóanyagot. Ha már tudjuk, hogyan kell a partíciókat kezelni, hozzunk létre egy legalább 50 MB méretű, folytonos szabad lemezterületet, vagy ha az összes parancs forráskódjára is szükségünk van, akkor egy 1 GB méretűt. Ha nem tudjuk, hogyan kell a partíciókat kezelni, de rendelkezünk particionáló programmal, amilyen például a *Partition Magic* is, használjuk azt szabad lemezterület létrehozására. Győződjünk meg arról is, hogy legalább 1 elsődleges partíció (vagy elsődleges indítórekord – Master Boot Record – hely) szabad. A MINIX 3-telepítőszkript ezután végigvezet minket a MINIX-partíció elkészítésén, amely vagy az elsődleges, vagy a másodlagos IDE-lemezen lehet.

Ha Windows 95, 98, ME vagy 2000 rendszert futtatunk, és a lemez egyetlen FAT partícióból áll, a CD-ROM-on (vagy a *zeleps.com* helyen) található *presz134.exe* program segítségével csökkenthetjük annak méretét, hogy helyet alakítsunk ki a MINIX számára. Bármilyen más esetben olvassuk el a világhálón elérhető, fentebb említett oktatóanyagot.

Amennyiben 128 GB-nál nagyobb méretű merevlemezzel rendelkezünk, a MINIX 3-partíciónak teljes egészében az első 128 GB-on belül kell lennie (a lemez blokkjainak címzési módja miatt).

FIGYELEM: Ha hibát követünk el a lemez particionálásakor, akár el is vesztjük a lemezen lévő adatainkat, ezért mindenképpen mentsük azokat CD-ROM-ra vagy DVD-re a particionálás megkezdése előtt. A lemezparticionálás nagy figyelmet igényel, tehát óvatosan hajtsuk végre.

F.1.2. Rendszerindítás

Ezen a ponton már rendelkeznünk kell felszabadított lemezterülettel. Ha még nem tettük meg, tegyük meg most, hacsak nem akarunk egy már létező partíciót MINIX 3-partícióvá alakíttatni.

1. Rendszerindítás CD-ROM-ról

Helyezzük be a CD lemezt a meghajtóba, és indítsuk arról a számítógépet. Ha 16 MB vagy annál több RAM-mal rendelkezünk, akkor válasszuk a „Regular", ha csak 8 MB-nyival, akkor a „Small" lehetőséget. Ha a számítógép a merevlemezről indul a CD-ROM helyett, akkor indítsuk újra a gépet, lépjünk be a BIOS beállító programba, változtassuk meg a betöltési eszközök sorrendjét úgy, hogy a CD-ROM a merevlemez előtt szerepeljen a listában.

2. Bejelentkezés adminisztrátorként (root)

Amikor a *login* prompt megjelenik, jelentkezzünk be *root*-ként. Sikeres bejelentkezés után megjelenik a parancsértelmező promptja (#). Ezen a ponton a MINIX 3 teljes mértékben működőképes. Az

```
ls /usr/bin | more
```

begépelésével megnézhetjük, milyen szoftverek állnak rendelkezésünkre. A szóköz lenyomásával görgethetjük a listát. Ha kíváncsiak vagyunk, hogy például a *foo* nevű program mit csinál, gépeljük be a

```
man foo
```

parancsot. A felhasználói kézikönyv a *www.minix3.org/manpages* honlapon is elérhető.

3. Indítsuk el a telepítőszkriptet
A MINIX 3 merevlemezre telepítéséhez gépeljük be a következőt:

```
setup
```

Ez után a parancs után (és minden más parancs után is) nyomjuk meg az ENTER (RETURN) billentyűt. Amikor a telepítőszkript a képernyő alján egy kettőspontot jelenít meg, nyomjuk meg az ENTER billentyűt a folytatáshoz. Ha a képernyő hirtelen teljesen üres lesz, akkor a CTRL-F3 lenyomásával válasszuk a szoftveres görgetést (erre általában csak nagyon régi számítógépek esetében van szükség). Itt jegyezzük meg, hogy a CTRL billentyű azt jelenti, hogy a CTRL lenyomva tartása mellett kell az adott billentyűt lenyomni.

F.1.3. Telepítés a merevlemezre

Az itt következő lépések megfelelnek a képernyőn megjelenőknek.

1. Válasszuk ki a billentyűzet típusát
Amikor a nemzeti billentyűzet kiválasztását kéri a szkript, akkor tegyük meg. Ez és minden más lépés esetében is van egy alapértelmezett választási lehetőség, amely szögletes zárójelek között jelenik meg. Ha az megfelelő, akkor elegendő az ENTER billentyűt megnyomni. A legtöbb esetben az alapértelmezett lehetőség megfelelő a kezdők számára. Az us-swap billentyűzet esetében a CAPS LOCK és a CTRL billentyűk fel lesznek cserélve, ahogyan az a Unix-rendszerek esetében szokásos.

2. Válasszuk ki az Ethernet-apka típusát
Kiválaszthatjuk, hogy az elérhető Ethernet-meghajtóprogramok közül melyik kerüljön telepítésre (esetleg egyikük sem). Válasszunk egyet a lehetőségek közül.

3. Minimális vagy teljes telepítés?
Ha szűkös a rendelkezésre álló lemezterület, az „M" lenyomásával választhatjuk a minimális telepítést, amely az összes bináris állományt magában foglalja, de a forráskódok közül csak a rendszerhez tartozók kerülnek telepítésre. 50 MB elegendő ehhez. Ha legalább 1 GB rendelkezésre áll, akkor válasszuk a teljes telepítést az „F" lenyomásával.

4. Hozzunk létre vagy válasszunk partíciót a MINIX 3 számára
Először arra kell válaszolnunk, hogy értünk-e a MINIX 3-lemezparticionáláshoz? Ha igen, akkor a *part* program segítségével teljes hozzáférést kapunk az elsődleges indítórekord (Master Boot Record) szerkesztéséhez (és esetlegesen annak teljes elrontásához). Ha nem vagyunk szakértők a témában, akkor válasszuk az alapértelmezett lehetőséget, amely egy automatikus lépésről lépésre útmutató a lemezpartíció MINIX 3 számára formázásához.

4.1. Válasszuk ki a lemezt, amelyre a MINIX 3 kerül
Egy IDE vezérlő maximum 4 lemezt tud vezérelni. A *setup* szkript megvizsgálja mindegyiket. Az esetlegesen megjelenő hibaüzeneteket hagyjuk figyelmen kívül. Amikor a meghajtók listázásra kerülnek, válasszunk közülük egyet, és erősítsük meg a választását. Amennyiben két merevlemezzel rendelkezünk, és úgy döntünk, hogy a MINIX 3-at a másodikra telepítjük, de problémába ütközünk a rendszer indításakor, akkor a megoldáshoz olvassuk el a *www.minix3.org/doc/using2disks.html* oldal tartalmát.

4.2. Válasszunk ki egy lemezterületet
Most válasszuk ki, hogy a MINIX 3 milyen lemezterületre kerüljön. Három választási lehetőség van:

(1) Egy szabad terület választása.
(2) Egy létező partíció felülírása.
(3) Egy partíció törlése és a vele szomszédos szabad terület összefűzése.

Az (1) és (2) választása esetén gépeljük be a terület sorszámát. A (3) választásához gépeljük be a

```
delete
```

sort, majd az ezután megjelenő kérdésre adjuk meg a terület sorszámát. Ez a terület felülírásra kerül, és az előző tartalma örökre elvész.

4.3. Erősítsük meg az előző választásainkat
Elértünk arra a pontra, ahonnan már nincs visszaút. A szkript felteszi a kérdést, hogy biztosan folytatni akarjuk-e? **Amennyiben igen, a kiválasztott területen található adat örökre elvész.** Ha biztosak vagyunk a folytatásban, akkor gépeljük be, hogy

```
yes
```

és üssük le az ENTER billentyűt. A CTRL-C lenyomásával kiléphetünk a telepítő szkriptből anélkül, hogy a partíciós tábla megváltozna.

5. Újratelepítési lehetőség
Amennyiben már létező MINIX 3-partíciót választottunk, választhatunk a teljes telepítés (Full install), amely mindent töröl a partícióról, és az újratelepítés (Reinstall) között, amely utóbbi nem befolyásolja a már létező */home* partíció tartalmát. Ez azt jelenti, hogy személyes fájljainkat nyugodtan elhelyezhetjük a */home* katalógusban, és telepíthetünk frissebb MINIX 3-verziókat, amikor azok megjelennek, személyes fájljaink elvesztése nélkül.

6. Adjuk meg a /home méretét
A kiválasztott partíció három részre lesz felosztva: root, */usr* és */home*. Ez utóbbi használható a személyes fájlok tárolására. Adjuk meg, hogy a partíció mekkora része legyen fenntartva a fájljaink számára. Válaszunkat meg is kell erősítenünk.

7. Adjuk meg a blokkméretet
Az 1 KB, 2, 4 és 8 KB méretű blokkok támogatottak, de 4 KB méretnél nagyobb választása esetén a forráskódban egy konstans értéket meg kell változtatni, és a rendszert újra kell fordítani. Ha a memória mérete 16 MB vagy annál nagyobb, akkor válasszuk az alapértelmezettet (4 KB), egyébként használjuk az 1 KB méretet.

8. Várjuk meg a hibás blokkok detektálását
A telepítőszkript most átvizsgálja a partíciókat, hibás blokkokat keresve. Ez sokáig tarthat, nagyméretű partíciók esetében akár 10 percnél is tovább. Legyünk türelmesek. Ha teljesen biztosak vagyunk abban, hogy nincsenek hibás blokkok, akkor az egyes vizsgálatokat a CTRL-C lenyomásával megszüntethetjük.

9. Várjuk meg a fájlok másolását
Amikor az átvizsgálás befejeződik, a fájlok automatikusan átmásolódnak CD-ROM-ról a merevlemezre. Minden fájl neve megjelenítésre kerül másolásakor. A másolás befejeztével a MINIX 3 telepítése kész. Állítsuk le a rendszert a

```
shutdown
```

begépelésével. Mindig így állítsuk le a rendszert az esetleges adatvesztés elkerülése érdekében, a MINIX 3 ugyanis bizonyos fájlokat RAM-lemezen tárol, amelyek csak a rendszer leállításakor kerülnek ténylegesen visszaírásra a merevlemezre.

F.1.4. Tesztelés

Ez a rész bemutatja, hogyan tesztelhetjük a telepítést, módosítás után hogyan fordíthatjuk újra a rendszert, valamint hogyan indíthatjuk újra később. A kezdéshez indítsuk el az új MINIX 3-rendszert. Például ha a 0. vezérlő 0. lemezének 3. partícióját használtuk, gépeljük be, hogy

```
boot c0d0p3
```

és jelentkezzünk be adminisztrátorként (root). Nagyon ritkán előfordulhat, hogy a BIOS által megjelenített meghajtósorszám (amelyet a betöltési felügyelőprogram is használ) nem egyezik meg a MINIX 3 által használttal. Ekkor először próbáljuk meg a telepítőszkript által megjelenített meghajtósorszám használatát. Itt a jó alkalom az adminisztrátori jelszó megadására is. Ha segítségre van szükségünk, olvassuk el a *man passwd* leírást.

1. Fordítsuk le a tesztprogramcsomagot
A MINIX 3 teszteléséhez a parancssorba (#) gépeljük be a következőket:

```
cd /usr/src/test
make
```

és várjunk, amíg mind a 40 fordítás elkészül. A CTRL-D lenyomásával lépjünk ki.

2. A tesztprogramcsomag futtatása
A rendszer teszteléséhez lépjünk be bin felhasználóként (ez szükséges), és a tesztelő programok futtatásához gépeljük be a következőket:

```
cd /usr/src/test
./run
```

Mindnek rendben le kell futnia, bár ez akár 20 percig is eltarthat egy gyors gépen, és akár egy óránál is tovább lassú gép esetében. *Megjegyzés:* a setuid bit megfelelő működésének tesztelése érdekében szükséges, hogy a teszt programcsomag fordítása adminisztrátorként (root), a futtatása bin felhasználóként történjen.

3. A teljes operációs rendszer újrafordítása
Ha minden teszt rendben lefutott, most újrafordíthatjuk a rendszert. Ez nem feltétlenül szükséges, mivel a rendszer lefordított állapotában került telepítésre. Azonban ha később módosítani kívánunk a rendszeren, szükségünk van arra, hogy újra tudjuk fordítani. Emellett a rendszer újrafordítása jó teszt arra, hogy egyáltalán működik-e? Gépeljük be a

```
cd /usr/src/tools
make
```

parancsokat, hogy a számos rendelkezésre álló opció megjelenjen. Most készítsünk egy új indítható rendszerképet a

```
su
make clean
time make image
```

parancsok begépelésével. Ezzel újrafordítjuk az operációs rendszert, beleértve a kernel módú és a felhasználói módú részeket is. Ugye nem tartott sokáig? Ha van hajlékonylemez-meghajtónk, készíthetünk indítható hajlékonylemezt későbbi használatra a formázott lemez behelyezése utáni

```
make fdboot
```

parancs begépelésével. Amikor az elérési útvonalat kell kiegészítenünk, gépeljük be az

```
fd0
```

nevet. Ez a megközelítés pillanatnyilag nem működik USB hajlékonylemezekkel, mivel egyelőre nincs MINIX 3 USB hajlékonylemez-meghajtó program. A merevlemezre telepített betöltési memóriakép frissítése a

```
make hdboot
```

paranccsal történik.

4. A rendszer leállítása és újraindítása
Az új rendszer indításához először állítsuk le a jelenlegit:

```
shutdown
```

Ez a parancs elment bizonyos fájlokat, majd visszakerülünk a MINIX 3 betöltési felügyelőprogramjához. A betöltési felügyelőprogram lehetőségeinek összefoglalóját a

```
help
```

begépelésével kaphatjuk meg. Bővebb információért látogassuk meg a *www.minix3.org/manpages/man8/boot.8.html* oldalt. Most már eltávolíthatjuk a CD-ROM-ot vagy a hajlékonylemezt, és kikapcsolhatjuk a számítógépet.

5. Újraindítás máskor
Ha rendelkezésünkre áll hajlékonylemez-meghajtó egység, akkor a legegyszerűbben az új indítólemez behelyezésével és a számítógép bekapcsolásával indíthatjuk el a MINIX 3-at. Ez pár másodpercet vesz csak igénybe. Másik megoldásként indíthatjuk a MINIX 3 CD-ROM-ról is, jelentkezzünk be bin felhasználóként és gépeljük be a

```
shutdown
```

parancsot, hogy a MINIX 3 betöltési felügyelőprogramjához jussunk. Ekkor gépeljük be a

```
boot c0d0p0
```

parancsot, hogy a rendszer a 0. vezérlő 0. meghajtójának 0. partíciójáról induljon. Természetesen ha a MINIX 3-at a 0. meghajtó 1. partíciójára telepítettük, akkor a


```
boot c0d0p1
```

parancsot kell használnunk, és így tovább.

A harmadik rendszerindítási lehetőség a MINIX 3-partíció aktívvá tétele és a MINIX 3 betöltési felügyelőprogramjának használata a MINIX 3 vagy bármely más operációs rendszer indításához. A részletekért látogassunk el a *www.minix3.org/manpages/man8/boot.8.html* oldalra.

Végül a negyedik lehetőség egy több rendszer indítására képes betöltőprogram telepítése, amilyen például a LILO vagy a GRUB *(www.gnu.org/software/grub)*. Ezek használatával tetszőleges operációs rendszer könnyen indítható. A több rendszer indítására képes betöltőprogramok tárgyalása túlmutat útmutatónk keretein, de a témáról információ található a *www.minix3.org/doc* oldalon.

F.1.5. Szimulátor használata

A MINIX 3 futtatásának egy teljesen más megközelítése az, ha nem natív módban a csupasz hardveren futtatjuk, hanem egy másik operációs rendszer felett. Különböző virtuális gépek, szimulátorok és emulátorok állnak rendelkezésre ilyen a célra. Íme néhány a legnépszerűbbek közül:

- VMware *(www.vmware.com)*;
- Bochs *(www.bochs.org)*;
- QEMU *(www.qemu.org)*.

Olvassuk el a hozzájuk tartozó dokumentációt. Egy program futtatása szimulátoron hasonló ahhoz, ahogyan az a tényleges gépen is történik, ezért térjünk vissza az 1. részhez, szerezzük be a legújabb CD-ROM-ot, és folytassuk onnan az olvasást.

F.2. A MINIX 3 forráskódja – CD melléklet

Rendszerkövetelmények

A CD mellékleten rendelkezésre bocsátott szoftver telepítésének minimális rendszerkövetelményei a következők.

Hardver

A MINIX 3 operációs rendszer futtatásához szükséges hardver:

- Pentium vagy azzal kompatibilis processzorral rendelkező PC
- 16 MB vagy több RAM

- 200 MB szabad lemezterület
- IDE CD-ROM-meghajtó
- IDE merevlemez

NEM TÁMOGATOTT: Soros ATA (SATA), USB és SCSI lemezek. Más lehetséges konfigurációk esetén látogassunk el a *www.minix3.org* oldalra.

Szoftver

A MINIX 3 egy operációs rendszer. Ha meg kívánjuk tartani a jelenlegi operációs rendszerünket és adatainkat (ami ajánlott), továbbá fel szeretnénk készíteni számítógépünket többféle rendszer indítására, particionálnunk kell a merevlemezt. Az alábbi lehetőségeket állnak rendelkezésünkre:

- Partition Magic *(www.powerquest.com/partitionmagic)*

vagy

- The Partition Resizer *(www.zeleps.com)*

vagy

- kövessük a *www.minix3.org/partitions.html* oldalon leírtakat.

Telepítés

A telepítés végrehajtható internetkapcsolat nélkül is, de néhány dokumentáció csak a *www.minix3.org* oldalon érhető el. Teljes telepítési leírás található a CD-n Adobe Acrobat PDF formátumban.

Terméktámogatás

A CD-n található MINIX 3-szoftverrel kapcsolatos további technikai információkért látogassunk el a *www.minix3.org* címen elérhető hivatalos MINIX-oldalra.

F.3. Fájlmutató

20900 servers/fs/fs.h
21400 servers/fs/glo.h
22900 servers/fs/inode.c
21900 servers/fs/inode.h
27000 servers/fs/link.c
23800 servers/fs/lock.c
21800 servers/fs/lock.h
24000 servers/fs/main.c
26700 servers/fs/mount.c
24500 servers/fs/open.c
22000 servers/fs/param.h
26300 servers/fs/path.c
25900 servers/fs/pipe.c
27800 servers/fs/protect.c
21200 servers/fs/proto.h
25000 servers/fs/read.c
27500 servers/fs/stadir.c
23300 servers/fs/super.c
22100 servers/fs/super.h
22200 servers/fs/table.c
28800 servers/fs/time.c
21100 servers/fs/type.h
25600 servers/fs/write.c

Processzuskezelő (process manager)
19300 servers/pm/break.c
17100 servers/pm/const.h
18700 servers/pm/exec.c
18400 servers/pm/forkexit.c
20400 servers/pm/getset.c
17500 servers/pm/glo.h
18000 servers/pm/main.c
20500 servers/pm/misc.c
17600 servers/pm/mproc.h
17700 servers/pm/param.h
17000 servers/pm/pm.h
17300 servers/pm/proto.h
19500 servers/pm/signal.c
17800 servers/pm/table.c
20300 servers/pm/time.c
20200 servers/pm/timers.c
17200 servers/pm/type.h

Tárgymutató

M