Nyílt rendszerek alapszoftverei

Klasszikus változat

Csizmazia Balázs [1]

Tartalomjegyzék

Fejezet 1

Bevezetés

A számítógépen futó programokat két csoportba szokás osztani: a **rendszerprogramok** csoportjára, és a **felhasználói programok** csoportjára. A rendszerprogramok közül a legalapvetőbb az operációs rendszer. Ennek feladata egyrészt az, hogy eltakarja a bonyolult hardver elemek programozását a programozó elől, másrészt pedig ez a szoftver felelős a hardver erőforrásoknak a programok közti elosztásáért, az egyes hardver erőforrások védelméért.

Az operációs rendszerek az utóbbi évtizedekben nagyon nagy fejlődésen mentek keresztül. Az első generációs számítógépekben még nem használtak operációs rendszereket. Megjelenésük a második generációhoz kötődik: a bonyolultabb hardver rendszerekre egyre bonyolultabb operációs rendszereket építettek, majd megjelent a multiprogramozás, a mai operációs rendszerek egy lapvető fontosságú tulajdonsága. A multiprogramozásnak két változata van: a **többtaszkos** (multitasking) illetve a **többfelhasználós** (multi user) rendszer (ez a két forma nem zárja ki egymást). A többfelhasználós rendszerekben egy központi egységen osztozik több felhasználó, de a központi egység nagy sebessége miatt minden felhasználó úgy érzi, hogy egy saját gépe van, amin dolgozik. A többtaszkos rendszer annyit tud, hogy ott egy felhasználó egyszerre több feladatot indíthat el, és az elindított feladatok egyszerre (párhuzamosan) fognak végrehajtódni.

Az operációs rendszerekkel kapcsolatban a jelenlegi kutatások a **hálózati operációs rendszerek** körében történnek. Ezekben a rendszerekben a számítógépek valamilyen dróttal össze vannak kapcsolva, és a felhasználó az operációs rendszer segítségével ezeken a drótokon keresztül adatokat vihet át az egyik gépről a másikra; az egyik gépről (mondjuk Magyarországról) bejelentkezhet egy másik számítógépre (például Kanadába), és Magyarországról úgy dolgozhat, mintha közvetlenül a kanadai számítógép egy képernyőjén dolgozna. Az, hogy az általa begépelt karakterek illetve a végeredmények milyen módon jutnak el tőle Kanadába (és onnan vissza Magyarországra) rejtve marad előle. A kommunikáció történhet akár telefonvonalakon, akár műholdon keresztül - a lényeg az, hogy az információ eljusson az egyik helyről a másikra. Az operációs rendszer feladata ilyenkor az, hogy a megbízhatatlan, rossz minőségű telefonvonalakon egy megbízható kommunikációs csatornát biztosítson a felhasználóknak, amiben az egyik gépről a másikra küldött adatok "nem kallódnak el", és az adatokat a fogadó állomás az elküldés sorrendjében kapja meg.

Eddig már számtalan sok operációs rendszer készült, mindegyik más céllal, más problémakör megoldására. Ma a legelterjedtebb ilyen rendszerek (többek közt): az

OS/360, a CP/CMS, a VAX VMS, a UNIX és az MS-DOS. Már elég idő volt ahhoz, hogy a legfontosabb absztrakciós szintek és szolgáltatástípusok kialakuljanak. Ezek a szolgáltatások a hagyományos operációs rendszerekben két fő témakörbe sorolhatók: folyamatokkal (processzekkel) kapcsolatos, és a fajlokkal kapcsolatos absztrakciós eszközök. A továbbiakban ezekről lesz szó kicsit részletesebben.

1.1 Folyamatok

A folyamat (processz) definíciója UNIX környezetben a következőképpen adható meg: folyamatnak tekinthetünk minden egyes futó programot - az általa lefoglalt memóriával és egyéb erőforrásokkal együtt. (Gyakran használják hasonló értelemben a taszk elnevezést is.) Az operációs rendszer minden egyes futó programról bizonyos információkat tárol egy ún. **processz-táblában**. A folyamatokkal kapcsolatban két alapvető művelet van: új folyamat létrehozása, és egy futó folyamat megállítása (abortálása, lelövése). Ha egy folyamat létrehoz egy új folyamatot, akkor az újonnan létrehozott folyamatot gyermek folyamatnak nevezik, azt a folyamatot, amely a gyermeket létrehozta szülő folyamatnak nevezik. Fontos megoldani az egymással párhuzamosan futó folyamatok egymás közti kommunikációját is.

Minden egyes folyamathoz többek közt hozzá van rendelve egy egyedi ún. folyamat-azonosító (processz-id, pid), és az, hogy ki indította el azt a folyamatot (vagyis az, hogy melyik felhasználó indította el; persze az is tárolva van minden egyes folyamatról, hogy melyik folyamat hozta létre, és még sok más adat). Ilyen jellegű információk nyilvántartása érdekében minden egyes felhasználóhoz hozzá van rendelve egy természetes szám, a felhasználó azonosítója (user-id, uid). A folyamatot elindító felhasználó uid-je be lesz jegyezve a processz-táblába, és később ha kell, akkor onnan ki lehet azt nyerni. A UNIX operációs rendszerben alapértelmezés szerint minden egyes folyamat örökli a szülőjének az uid-jét és a jogait valamint számos más jellemzőjét is. (Ezzel szemben a folyamat-azonosító, a pid például nem örökölhető, mert ekkor az nem lenne egyedi.)

Az egymással párhuzamosan működő folyamatoknak gyakran kell kommunikálniuk valamilyen módon. Az operációs rendszer feladatai közé tartozik a **folyamatok közötti kommunikáció** (Interprocess Communication) megszervezése is.

Sok operációs rendszer a folyamat fogalmat két fő részre osztja: egy **taszkra** és egy vagy több ún. threadre (magyarul: **szál**). A taszk egy "erőforrásgyűjtemény" (fájlok, memóriaterületek és más objektumok) a thread pedig a folyamat "lelke": lényegében egy processzor-állapotból és egy saját stack-ből áll. Egy taszkban egy vagy több thread lehet. Az eredeti (UNIX-szerű) modellben egy folyamat pontosan egy taszkból és egy benne futó threadből áll.

(Szokás megkülönböztetni **preemptív** ill. **nem preemptív** thread-rendszereket is. Az előbbiben minden egyes threadnek van egy-egy időszelete, amíg futhat, majd ha az lejár, akkor egy másik thread kapja meg a CPU-t; az utóbbi modellben a threadnak valamilyen op-rendszer rendszerhívás meghívásával önszántából kell lemondania a CPU-használatról - ez utóbbi forma a program nyomkövetésekor hasznos.)

1.2 Fájlok

Minden számítógépet felszerelnek valamilyen háttértárral, ami adatokat képes tárolni hosszabb időn keresztül (**fájlok** "formájában"). Ezeknek a fájloknak neveket adhatunk. Az, hogy a név hány és milyen karaktert tartalmazhat, nagyon eltérő lehet a különböző operációs rendszerekben. A fájlok kezelését végző operációs rendszer komponenseket gyakran hívják fájlrendszer kezelőnek.

Egy kisebb méretű UNIX rendszerben alaphelyzetben kb. 3000-10000 kisebb-nagyobb fájl van a háttértáron, ezért az ott tárolt fájlokat valahogyan rendszerezni kell. A kialakult legelfogadhatóbb megoldást a **hierarchikus directory-szerkezet** (directory szó jelentése katalógus) jelenti. Ekkor a valamilyen szempont szerint összetartozó fájlok kerülnek egy közös directoryba. A hierarchikusság abban áll, hogy minden egyes directory tartalmazhat ún. aldirectorykat, amik szintén tartalmazhatnak aldirectorykat ...

Az egyetemeken ez például úgy használható ki, hogy a felhasználókat két csoportba osztva (pl. oktatók csoportjába ill. hallgatók csoportjába; persze lehet sok más csoport, ez inkább csak példa értékű) mindkét csoportnak egy-egy külön directoryja lehet, így a hallgatók fájljai védve vannak a kíváncsi oktatók elől (és természetesen fordítva is).

A hierarchikus directory-szerkezetet biztosító operációs rendszerekben az egyes fájlokra úgy hivatkozhatunk, hogy először meg kell adni azt, hogy a fájlt tartalmazó directoryt melyik directorykon keresztül érhetjük el a hierarchikus directory-szerkezet gyökerétől kiindulva, majd meg kell adni a fájlt tartalmazó directory nevét és magának a fájlnak a nevét is. (Ezt nevezik a fájl pathname-jének.) Ha például van egy `user` nevű directory (tegyük fel, hogy ez a directory a directory-szerkezet gyökerében van), aminek van egy `student` nevű aldirectoryja, akkor az abban az aldirectoryban levő `xyz` nevű fájlra a `/user/student/xyz` névvel hivatkozhatunk. (A UNIX operációs rendszerben a pathname egyes tagjait a / jel választja el egymástól, és a fájlnévben a legelső / jel a hierarchia tetején levő ún. gyökér-directoryt jelöli, amely egyetlen más directorynak sem aldirectoryja.) Ha minden egyes fájlra csak ilyen "hosszú módon" (ún. abszolút pathname segítségével) hivatkozhatnánk, akkor nagyon nehéz lenne az élet (és kényelmetlen is!). Ezért alakították ki a **munka-directory** (working directory) fogalmát. Ez azt jelenti, hogy van egy ún. munka-directory, amelyben a fájlokat a gyökértől hozzájuk vezető minden egyes aldirectory nevének felsorolása nélkül érhetjük el. Csak azoknak a directoryknak a nevét kell felsorolni, amely a hierarchiában a munka-directory alatt van. (Az ilyen, a munkadirectoryból kiinduló pathname-eket relatív pathname-nek szokás nevezni.)

Még egy fontos elv van a fájlrendszerekkel kapcsolatban: a **készülékfüggetlenség**. Eszerint az elv szerint a programokat úgy kell megírni, hogy működni tudjanak attól függetlenül, hogy az inputjukat és az outputjukat képező fájlok egy floppy-lemezen vagy egy winchesteren vannak (vagy esetleg az input a billentyűzetről lesz beadva ...).

Egyes operációs rendszerek a fájlokról nem feltételeznek semmiféle **belső struktúrát**: egyszerűen egy **byte-folyamnak** tekintik azokat (ilyen a UNIX). Más rendszerekben a fájlok mondjuk fix vagy változó számú byteot tartalmazó **rekordok sorozata** - ez gyakori volt a lyukkártyás időszakban: minden fájl 80 byteos rekordokból állt. Ma egyre inkább a byte-folyam jellegű fájl kép kerül előtérbe, és a fájlok belső szerkezetét pedig az azt feldolgozó programok "saját belátásuk szerint" alakíthatják ki.

A fájlokhoz minden operációs rendszer nyilvántart bizonyos ún. fájl-attributumokat. Ezek a fájllal együtt a háttértáron lesznek tárolva. Ilyen fájl-attributumok például a

következők (nem minden operációs rendszer ad lehetőséget az itt felsorolt összes fájl-attributum nyilvántartását):

- a fájl mérete byteban VAGY blokkban (operációs rendszertől függ a "VAGY" ..)
- a fájl hozzáféréséhez szükséges jelszó
- a fájl maximális mérete (néhol ez is előre meg van kötve)
- a fájl tulajdonosának azonosítója
- a fájl "system" fájl-e (az operációs rendszerenként változhat, hogy egy fájl "system"-sége milyen lehetőségeket jelent)
- a fájl "archive" fájl-e (ez az egyik operációs rendszer szerű eszközben, az MS-DOS-ban azt jelöli, hogy a fájl ki van-e mentve (BACKUP-olva) vagy sem)
- a fájl "hidden"-e vagy sem
- a fájl léterehozásának dátuma
- a fájl utolsó módosításának dátuma
- utolsó "használat" dátuma
- a fájl jelszavakat tartalmaz, nem nézheti meg senki (esetleg még a rendszergazda sem)
- a fájl egy aldirectory (ilyenkor gyakran az adott aldirectory által tárolt fájlok neveit tartalmazza ...)
- esetleg azt is tárolhatja a rendszer egy fájlról, hogy a fájl a háttértár hibás szektorait (bad blocks) is tartalmazza, ezért nem tanácsos hozzányúlni.

Sok operációs rendszer a fájlokat vagy legalább egy részüket használatuk közben a memóriában tartja. Ezt **cache**-elésnek nevezik, és a memóriának azt a (gyakran dinamikusan változó méretű) részét, amit az operációs rendszer erre felhasznál cache-memóriának nevezik.

Sok fájlrendszer lehetőséget nyújt a fájlok "**memóriába ágyazására**" (memory mapped files). Ez azt jelenti, hogy a folyamatok a memória valamelyik szegmensén (részén) keresztül a fájlba tudnak írni, illetve onnan tudnak olvasni: ha a program a memóriaszegmens 0., 1., 2. ... byteját mondjuk megváltoztatja, akkor vele együtt meg fog változni a háttértárolón tárolt fájl 0., 1., 2. ... byteja is. Ez a megoldás tehát egységessé teszi a fájl- és memóriahozzáférés módját, kapcsolatot teremtve az operációs rendszer fájlrendszere és a memóriakezelő komponense között.

Az operációs rendszer feladata a **fájlrendszer konzisztensségének a biztosítása** is: ez például magába foglalja azt is, hogy az operációs nehogy kétszer ugyanazt a diszk-szektort használja fel egy fájl különböző részeinek a tárolására, mert így adatok vesznének el.

1.3 Memóriakezelés

A memória az egyik legfontosabb (és gyakran a legszűkösebb) erőforrás, amivel egy operációs rendszernek gazdálkodnia kell; főleg a többfelhasználós rendszerekben, ahol gyakran olyan sok és nagy folyamat fut, hogy együtt nem férnek be egyszerre a memóriába. A memóriakezelésről nem lesz szó a későbbi fejezetekben, ezért itt ismertetem a fontosabb fogalmakat.

Amíg a multiprogramozás nem jelent meg, addig az operációs rendszerben nem volt olyan nagy szükség a memóriakezelő részekre. A multiprogramozás megjelenésével azonban szükségessé vált a memóriának a futó folyamatok közötti valamilyen "igazságos" elosztására. A megoldást a **virtuális memóriakezelés** jelentette. Ez úgy működik, hogy az operációs rendszer minden egyes folyamatnak ad a központi memóriából egy akkora részt, amelyben a folyamat még úgy ahogy működik, és a folyamatnak csak azt a részét tartja a központi memóriában, amely éppen működik. A folyamatnak azt a részét, amelyre nincs szükség (mert például már rég nem adódott rá a vezérlés, és feltételezhetjük, hogy rövid időn belül nem is fog végrehajtódni) ki kell rakni a háttértárra (a diszken az ún. **lapozási területre**). Ez a megoldás azért működik, mert a programok legtöbbször egy eljáráson belül ciklusban dolgoznak, nem csinálnak gyakran nagy ugrásokat a program egyik végéről a másikra (ez a **lokalitás elve**).

A központi egység fel van szerelve egy úgynevezett **memóriakezelő egységgel** (MMU), amely figyeli, hogy olyan kódrészre kerül-e a vezérlés, amely nincs benn a központi memóriában (mert például a háttértárra van kirakva). Ha a memóriakezelő egység azt találja, hogy ez az eset áll fenn, akkor az operációs rendszert arra utasítja, hogy rakja ki a háttértárra a folyamatnak azt a részét, amely jelenleg a memóriában van, és azt a részt hozza be a helyére, amelyre ezután szükség lesz.

A virtuális memória kezelése leggyakrabban **lapozással** (paging) történik. Ekkor a virtuális memória (egy folyamat virtuális címtartománya, amit a CPU biztosít) fel lesz osztva egyenlő nagyságú részekre, ún. **lapokra** (pages) - a háttértár és a memória között legalább ennyi byteot fog az operációs rendszer átvinni (vagy ennek többszörösét). A fizikai memória pedig fel lesz osztva ugyanolyan méretű **lapkeretekre** (page frames). Ha mondjuk a virtuális címtartomány 128 KByte, és 64 KByte fizikai memória van a számítógépbe építve, akkor ez 32 lapot, és 16 lapkeretet jelent, ha a lapméret 4 KByte. Ha egy program végrehajt egy olyan (gépi kódú) utasítást, amely a memória valamelyik rekeszére hivatkozik (a hivatkozott memóriarekesz címét nevezik **virtuális címnek**), akkor ezt a címet először a processzor átadja az MMU-nak, ami majd egy fizikai memóriabeli címet állít elő belőle. E feladatának ellátásához az MMU tárol egy ún. **laptáblát** (vagy legalábbis valamilyen módon hozzáfér a laptáblához), amely a lapok és lapkeretek egymáshoz rendelését tartalmazza egy speciális ún. "érvényességi" bittel, ami minden egyes laphoz tárolva van, és a bit értéke azoknál a lapoknál 1, amelyekhez tartozik a fizikai memóriában lapkeret. Az MMU működése során egy kapott virtuális címhez tartozó lapról megvizsgálja, hogy az "érvényességi" bitje 1-e. Ha igen, akkor a megadott laphoz tartozó lapkeret sorszámát visszaadja a CPU-nak (mondjuk ... ez történhet így is), és az a kívánt adatot a megfelelő (fizikai memória-) rekeszből megszerzi (vagyis azt csinál vele, amit a gépi kódú programban a végrehajtott gépikódú utasításban megadtak). Mi történik akkor, ha az "érvényességi" bit 0? Ekkor egy ún. hardware-interrupt (megszakítás) keletkezik, amit laphibának (page faultnak) neveznek. Ekkor kerül végrehajtásra az operációs rendszer memóriakezelő része, ami egy másik

"érvényes" (fizikai memóriabeli) lapnak az 1-es érvényességi bitjét 0-ra állítja, és a hozzá tartozó lapkeretet a diszkre menti (az ún. lapozási területre). A lapkeretet ezután beírja a laptáblába ahhoz a laphoz, amelyhez a laphiba során hozzá akartak férni, betölti a diszkről (lapozási területről) a megfelelő laphoz tartozó lapkeret tartalmát, a laphoz tartozó "érvényességi" bitet 1-re állítja, és az MMU ezután már laphiba nélkül el tudja végezni a címtranszformációt.

Több programnak szüksége lehet esetleg több virtuális címtartományra is. Sok CPU lehetőséget ad **szegmentált memóriakezelés**re, ami annyit jelent, hogy a program több ún. **szegmensben** is tárolhat adatokat, és mindegyik szegmenshez külön-külön laptábla tartozhat (mondjuk ... de ez nem mindig van így). Minden szegmensnek van egy dinamikusan változtatható mérete, ami az adott szegmensben megengedett legmagasabb sorszámú memóriarekeszt adja meg. Ilyen rendszerekben a memória címzésekor meg kell adni egy szegmens-sorszámot és az azon belüli virtuális címet is. Ilyen CPU-kon gyakori az is, hogy az operációs rendszer rövid időre nemcsak egy-egy lapot, hanem egy egész szegmenst visz ki a háttértárra - lényegében azt nevezik **swapping**nek.

A fenti leírás alapján már megérthető a virtuális memóriakezelés lényege, de azt fontos megemlíteni, hogy ez a valódi (működő) operációs rendszereknek az egyik legbonyolultabb része, és nagyon nehéz egyéb szempontoknak is megfelelő, ráadásul **hatékony** memóriakezelőt írni.

1.4 A shell

Az operációs rendszerekhez tartozik a felhasználóval (interaktív) kapcsolatot tartó parancsértelmező, a **shell** (igaz nem olyan szorosan, mint az előbbi pontokban említett részek). A legegyszerűbb shellek csak annyit tudnak, hogy egy tetszőleges programot el tudnak indítani (például azt a programot, amelyiknek a nevét a felhasználó a billentyűzeten beadta). A bonyolultabb shellek pedig akár egy komplett programnyelvet nyújtanak a programozó számára (ilyen shell például a REXX, vagy a UNIX shelljei).

1.5 Védelem

A védelem nagyon fontos, főleg a többfelhasználós operációs rendszerekben. Itt kell védeni a felhasználókat egymás elől, és az operációs rendszer "belső dolgait" a (kíváncsi, rosszindulatú, ügyetlen) felhasználók, és a hibás programok elől. A védelem alapját az képezi, hogy a legtöbb mikroprocesszor kétféle üzemmódban tud működni: **felhasználói** illetve **felügyelői** üzemmódban. A felügyelői üzemmódban minden meg van engedve, a felhasználói üzemmódban néhány dolog nincs megengedve (például az, hogy az egyik felhasználó átírja a másik felhasználó programját, és a felhasználók nem nyúlhatnak az operációs rendszer "beleegyezése nélkül" a különféle hardver perifériákhoz). A felügyelői üzemmód fenn van tartva az operációs rendszernek, a felhasználói üzemmódban pedig az egyes felhasználók programjai futhatnak.

1.6 INPUT/OUTPUT

Az operációs rendszer fő feladatai közé tartozik az is, hogy vezérelje a számítógéphez kapcsolt I/O perifériákat. A perifériákat két csoportba szokták osztani: **blokk-**

elérésűek és **karakter-elérésűek**. Az első csoportba tartoznak azok, amelyeknél a hardver periféria elemi műveletének egy blokk (nagyobb adatterület mondjuk 512 byte vagy annak a többszöröse) beolvasását ill. kiírását lehet tekinteni, és az egyes blokkok "címezhetők".

A karakter-elérésűek csoportjába tartoznak azok, amelyeknél az elemi műveletnek az egy darab karakter kiírása ill. beolvasása tekinthető - itt például "pozícionálásra" eleve nincs lehetőség. Ez a felosztás nem a legjobb, de lényegében megfelelő. Példák blokk-elérésű perifériákra: floppy-disk, winchester, RAM-diszk. Karakter-elérésű perifériák: billentyűzet, RS232-vonal, egér, printer. Az operációs rendszereknek azon részeit, amelyek a hardver perifériák kezeléséért felelősek, eszközmeghajtóknak (device drivereknek) nevezik.

1.7 Operációs rendszerek belső szerkezete

Az operációs rendszer belső szerkezete többféle lehet: például **monolitikus**, **rétegzett** (layered), **virtuális gép**eken alapuló valamint a **kliens-szerver modell**en alapuló.

A monolitikus operációs rendszer (mint például a UNIX) magja egyetlen programból áll. Ebben a programban az eljárások szabadon hívhatják egymást, a köztük levő kommunikáció eljárásparamétereken és globális változókon keresztül zajlik.

A rétegzett szerkezetű operációs rendszer magja több modulból áll, és a modulok között egy export-import hierarchia figyelhető meg: minden modul kizárólag a hierarchiában alatta levő modul interfészét használja.

A virtuális gépeken alapuló operációs rendszerben központi részen helyezkednek el a virtuális gépeket menedzselő (**hypervisor**) rendszerrutinok. Ez a program lehetővé teszi a hardver erőforrásainak (CPU, diszk, perifériák, memória, ...) több operációs rendszer közötti hatékony elosztását. A hypervisor leggyakrabban a számítógép hardverét "többszörözi meg" úgy, hogy a rajta futó operációs rendszerek azt higgyék, hogy övüké az egész gép (pedig "csak" egy virtuális gépen futnak). Ha például egy hardver-megszakítás generálódik, akkor ez a szoftver adja át annak a virtuális gépnek, amelyre ez tartozik (az, hogy kire tartozik egy hardver-megszakítás, többféleképpen eldönthető: például az alapján, hogy a kérdéses I/O eszközt ki használta utoljára). Ilyen hypervisor például az IBM VM/370. Az általa létrehozott és irányított virtuális gépek az IBM /370-es hardver "pontos másolatai", és tudnak futtatni (egymástól függetlenül) AIX, CMS, TSO és más operációs rendszereket.

A kliens-szerver modellen alapuló operációs rendszerek esetében az operációs rendszer központi magja általában egy ún. **mikrokernel**, és maga az operációs rendszer itt több párhuzamosan futó folyamatból áll. Mindegyik folyamat az operációs rendszer valamely jól elkülöníthető részét valósítja meg (például lehet egy vagy akár több fájlszerver folyamat, egy diszkvezérlő, egy printervezérlő folyamat a rendszerben). Ezeknek a folyamatoknak valamilyen kommunikációs eszközt kell biztosítani, és ez a mikrokernel egyik fő feladata (a memóriakezelésen kívül). A legtöbb ma használt ilyen kernel az üzenetátadást (message passing) teszi lehetővé a párhuzamosan működő folyamatok közötti kommunikáció elvégzése érdekében. Az alacsonyszintű üzenetátadás működhet két teljesen különböző host között is ugyanúgy, mint egy hoston belül különböző folyamatok között, ami a kommunikáció formáját egységessé teheti egy gépen belül és két gép között (ezt a transzparenciát nagyon nehéz elérni).

1.8 Osztott rendszerek architekturája

Manapság a számítógépes hálózatok órási ütemben fejlődnek – a nagysebességű hálózatok egyre olcsóbbak lesznek (például a 100 megabit/sec átviteli sebességű Ethernet már sok helyen felváltja az eddig alkalmazott 10 megabit/sec-es Ethernetet, és egyre több helyen alkalmaznak (mikro)processzorok adatvonalainak közvetlen összekapcsolásán alapuló hálózati adatátviteli technológiát), és ezzel párhuzamosan egyre olcsóbban egyre gyorsabb mikroprocesszorokat fejlesztenek ki a hardverfejlesztők laboratóriumaiban. Ennek a fejlődésnek egy nagyon fontos következménye az, hogy mód van nagyteljesítményű mikroprocesszorok nagysebességű hálózaton keresztül történő összekötésére (ezzel osztott rendszerek[1] létrehozására). Ma már kialakítottak több(tíz)ezer mikroprocesszorból álló számítógépes rendszereket, de ezek közül nem viselheti mindegyik az osztott rendszer nevet, mivel az alapszoftvere nem biztosítja a felhasználó felé a kellő mértékű transzparenciát, vagyis a felhasználó akarva-akaratlanul is szembetalálkozik olyan – például folyamatok szinkronizációjával kapcsolatos – problémákkal, amiket az okoz, hogy a rendszer nem egyetlen processzoron van implementálva.

Manapság az osztott rendszerek alkalmazásának számos előnye van, és a jelenleg működő megvalósítások problémái miatt számos hátránya is akad.

Az **osztott rendszerek alkalmazása mellett szól**nak az alábbi tényezők:

- gazdaságosságuk (például a hasonló teljesítményű szuperszámítógépekkel szemben)
- a velük elérhető tényleges feldolgozási kapacitás óriási mértéke (a legtöbb rendszer teljesítménye a bekapcsolt processzorok számának lineáris függvénye)
- az egész rendszer megbízhatóságának, elérhetőségének növekedése

Ahogyan az osztott rendszerek mellett találtunk számos okot arra, hogy alkalmazásukat bárkinek ajánlhassuk, úgy találhatunk több olyan tényezőt is, ami a jelenlegi **osztott-rendszer technológia alkalmazása ellen szól**:

- a megfelelő transzparencia hiánya
- a rendszer komponensei közti kapcsolatot biztosító adatátviteli közeg (vagyis a hálózat ...) megbízhatósága nagyban befolyásolja a rendszer alkalmazhatóságát[2]
- egy osztott rendszer erőforrás-adminisztrációja (itt nem a rendszergazda feladatok ellátására gondolok, hanem például a fájlok és más operációs rendszer erőforrásokra) sokkal erőforrásigényesebb az egyes komponensek összekapcsolatlan állapotbeli adminisztrációjánál

Az osztott rendszerek **hardware komponenseit** képező számítógépeket szokás **multicomputer** illetve **multiprocesszor** halmazra felbontani aszerint, hogy a processzoraik

[1] Definíció: **osztott rendszeren** egy olyan többprocesszoros rendszert érték, amely több (akár közös memóriával nem rendelkező) számítógépen működik, de a rendszer felhasználója mégis úgy látja, mintha az egész rendszer egyetlen nagyteljesítményű számítógép lenne. (Itt a felhasználó elnevezés alatt nemcsak a klasszikus értelemben vett felhasználókat értem (akik a rendszerre készített alkalmazói programokat használják), hanem azokat a programozókat is, akik a rendszerre új programokat fejlesztenek.)

[2] Az Ethernet hálózatokban például nem lehet idő tekintetében felső korlátot adni arra, hogy egy adatcsomagot egy számítógép mikor tud a dróton továbbítani (annak ellenére, hogy mérnökök valószínűleg be tudják bizonyítani, hogy annak a valószínűsége 1 (HF: tényleg mennyi ez a valószínűség?), hogy egy adatcsomagot egy számítógép korlátos időn belül el tud küldeni), ezért az Ethernet általában nem használható valós idejű elosztott rendszerek készítésére.

rendelkeznek-e közös használatú memóriával vagy sem. A multiprocesszor esetén van ilyen közös használatú memória.

A szoftver tekintetében pedig szokás beszélni **lazán kapcsolt** rendszerekről illetve **szorosan kapcsolt** rendszerekről (az előbbiekben csak egy-egy rendszerkomponens van elosztva a hálózaton – ilyen például a hálózati fájlrendszer, az NFS, amelyben csak a fájlrendszer elosztott, és a rendszer többi komponense egymástól teljesen függetlenül működik, míg a szorosan kapcsolt rendszerekben nagyon kevés az egymástól függetlenül működő komponens). A szorosan kapcsolt szoftverek tervezésekor a legnagyobb nehézséget az okozza, hogy a rendszerben senki sem ismer globális információkat a rendszerről, mindig csak annak egy részéről (esetleg csak pár tízezer számítógépről egy millió számítógépet tartalmazó hálózatban) vannak részletes információk.

Megjegyezzük, hogy a kommunikáció, a kommunikáló felek szerkezete nagyon sokféle lehet az osztott rendszerekben. Ilyen területen is kialakultak már "szokásos" megoldások: kliens-szerver párok kapcsolatát gyakran a **távoli eljáráshívási** modellre építik; (kettőnél) több komponensű rendszereket pedig gyakran az **üzenetszórásra** (broadcasting) építik: itt az egyik résztvevő (rendszerkomponens) olyan eszközökkel rendelkezik, amivel egy üzenetet küldhet az összes többi résztvevőnek.

1.8.1 A távoli eljáráshívás

A helyi eljáráshívás során a hívó fél a hívott eljárás számára átadandó argumentumokat általában a programvégrehajtási veremre helyezi, majd átadódik a vezérlés a hívott eljárásnak, amely a veremről leolvashatja az argumentumokat, és elvégzi velük a feladatát. Miután az eljárás befejeződött, a végeredményeket (a hívó eljárásnak visszaadandó értékeket) rárakhatja a veremre, és miután a hívó eljárás visszakapja a vezérlést, kiolvashatja a veremből az egyes visszatérési értékeket, és felhasználhatja azokat továbbműködése során.

A távoli eljáráshívás során a hívó és a hívott eljárások nem feltétlenül ugyanazon a számítógépen vannak: mialatt egy eljárás végre lesz hajtva a távoli gépen, addig a hívó eljárás futása felfüggesztődik (mondjuk a válasz visszaérkezéséig). A paraméterátadás megoldására kialakultak már elfogadható technikák – ezzel majd egy későbbi kiadásban részletesebben fogunk foglalkozni.

1.8.2 Üzenetszórás

Míg a távoli eljáráshívás általában a kétkomponensű kliens-szerver kapcsolatok implementálására használható, addig az üzenetszórás ennél több komponens kapcsolatát (is) képes megteremteni. Természetesen kettőnél több rendszerkomponens kommunikációja megszervezhető az egyes komponensek közötti távoli eljáráshívási műveletekkel, de egy ilyen esetben a kommunikáció lebonyolításához szükséges távoli eljáráshívások száma a kommunikáló felek számánál eggyel kevesebb, azaz azok számának növekedésével egyenes arányban nő. Az üzenetszórás abban különbözik lényegesen a több komponens között elvégzett távoli eljáráshívás között, hogy alkalmazásakor az összes címzetthez egyetlen művelettel lehet az üzenetet elküldeni.

Egy üzenet címzettjeinek a halmazát **csoport**nak nevezzük. Egy csoportot általában folyamatoknak azon halmazából alakítanak ki, amelyek valamilyen tekintetben azonosan viselkednek. Szoftver tekintetben az azonos viselkedés egyik alapja a csoportok közötti

kommunikációjának azon tulajdonsága, hogy a csoportnak küldött üzenetek minden csoport-tagnál megérkeznek, és az üzenetek megérkezési sorrendje minden csoport-tagnál ugyanaz. Most röviden áttekintünk egy – az Amoeba elosztott operációs rendszerben implementált – üzenetszórási protokollt, amely megfelel a fenti követelményeknek (vagyis két tetszőlegesen kiválsztott üzenet egymáshoz viszonyított sorrendje az összes számítógépen megegyezik).

Az algoritmus feltételez egy koordinátor számítógépet, amely a csoportok kommunikációjának koordinálásáért felelős (amennyiben ez a számítógép elromlana, akkor a protokoll új koordinátor választását írja elő – ami például az elosztott rendszerbe kapcsolt számítógépek közül valamilyen módon kiválasztható – akár úgy is, hogy a legmagasabb hardware-azonosítójú hostot választjuk ki e célra).

Az algoritmus a következőképpen működik:

1. Az üzenetszórással elküldendő üzenetet az üzenetszórást kérő folyamat átadja az operációs rendszernek, ami gondoskodik a koordinátorhoz való elküldésről (mondjuk egy hagyományos, távoli eljáráshívásra épülő eszközzel), kiegészítve egy sorozatszámmal (ami az eddig fogadott üzenetek sorozatszámának maximumával kell, hogy megegyezzen).

2. A koordinátor a kapott üzenetet a következő, az eddig használtaknál eggyel nagyobb üzenetsorszámmal kiegészítve a hardware nyújtotta – nem feltétlenül megbízható – üzenetszórási eszközzel mindenki számára elküldi az üzenetet.

3. Egy (broadcast) üzenet fogadásakor az üzenetet fogadó számítógép operációs rendszer összehasonlítja az üzenet sorszámát az eddig kapott üzenetek sorszámának a maximumával. Ha a kapott üzenet sorszáma nem csak eggyel nagyobb az eddigi üzenetek sorszámának maximumánál, akkor az operációs rendszer feltételezheti, hogy valamilyen oknál fogva egy (vagy több) üzenetet nem kapott meg – egyes üzenetek nem jutottak el hozzá (amik persze másokhoz már eljuthattak). Amennyiben alapos a gyanúja annak, hogy egy üzenetet nem kapott meg egy adott számítógép, akkor annak az üzenetnek az újraküldését fogja kérni a koordinátortól.

Megjegyzések:

A koordinátornak el kell tárolnia az összes eddigi olyan üzenetet, amelyek újraküldését valamelyik folyamat még kérheti (emlékezzünk rá, hogy a koordinátornak küldött üzenetek tartalmazzák az addig megkapott, legnagyobb sorszámú üzenet sorszámát).

Ha az üzenet küldője úgy látja, hogy a koordinátor egy bizonyos időn belül nem továbbította (szórta szét ...) az üzenetet, akkor újra küldheti az üzenetszórás iránti kérelmét a koordinátorhoz.

Minden üzenet el van látva egy egyedi azonosítóval is a duplikátumok kiszűrése céljából.

Az is előfordulhat, hogy az üzenet küldője nem az általa küldött üzenetet kapja meg, hanem előbb néhány másikat, és csak aztán a sajátját. Ez akkor lehet, ha többen is egyszerre akartak üzenetszórással kommunikálni, és a koordinátor egy másik üzenetet kapott meg és továbbított előbb.

Ha a koordinátor számítógép összeomlana, akkor új koordinátort kell választani. Az új koordinátornak fel kell készülnie esetlegesen üzenetek újraadására, így az eddig elküldött, de mindenki által nem megkapott üzeneteket az összes potenciális koordinátor-jelöltnek el kell tárolnia.

1.9 Holtpont

Látható, hogy az operációs rendszernek nagyon nehéz feladata az erőforrások igazságos (és hatékony működést eredményező) kiosztása. Ennek megoldása gyakran lehetetlenség, mert nem ismert a folyamatok jövőbeli viselkedése (vagyis nem lehet tudni, hogy egy adott folyamatnak milyen erőforrásokra lesz még szüksége). Vannak olyan "megoszthatatlan" erőforrások (mint például a legtöbb mágnesszalag-egység), amelyet egyszerre csak egy folyamat használhat, és abból adódhatnak a gondok, ha mégis két folyamat próbálja egyszerre használni őket. Tegyük fel, hogy egy multitaskingot biztosító operációs rendszerrel felszerelt gépen egy mágnesszalag-egység van, és egy darab printer. Két folyamat fut, és mindkettő ki akar nyomtatni egy mágnesszalagon tárolt fájlt. Az egyik folyamat megnyitja a mágnesszalag-fájlt (ezzel kizárólagos hozzáférési jogot nyer az egységhez), a másik folyamat (multitaszkos volt a rendszer) ezzel párhuzamosan megnyitja a nyomtatóra irányított fájlt (ezzel kizárólagos hozzáférési jogot nyer a nyomtatóhoz). Ekkor az első folyamat megpróbálhatja megnyitni a printer-fájlt, de mivel az "foglalt", ezért kénytelen várni (folyamatosan), amíg a másik folyamat "el nem engedi" azt. A másik folyamat megpróbálja megnyitni a mágnesszalag-fájlt, de ő is kénytelen várni, amíg a másik folyamat el nem engedi azt. Vagyis **mindkét folyamat olyan esemény bekövetkeztére vár, amelyet a két folyamat közül** *A másik* **folyamat idézhet elő**. Az ilyen helyzeteket nevezik holtpont-helyzetnek, más néven deadlocknak. Léteznek ezt **felismerő** vagy **megelőző** algoritmusok, de ezek az algoritmusok gyakran "kényelmetlenségeket" okoznak a felhasználóknak (pl. meg van kötve, hogy egy felhasználó maximum hány folyamatot indíthat el, mivel a processz-tábla is betelhet, és ez is okozhat holtpontot), ezért több operációs rendszer (mint például. a UNIX) egyszerűen tudomást sem vesz arról, hogy ez bekövetkezhet, és ha bekövetkezne egy ilyen esemény, akkor a felhasználóra bízza az ilyen helyzetbe került folyamatok "kilövését" (ld. majd később).

1.10 Az Intel 80386 mikroprocesszor architektúrája

Az Intel 80386-os mikroprocesszor egy olyan igazi 32-bites mikroprocesszor, amely a programozónak egy szegmentált és lapozásos virtuális memóriát biztosít. Hardver módon támogatja a különféle védelmi rendszerek kialakítását (a programok 4 különböző védelmi kategóriákba sorolhatóak, és az egyes kategóriákban a programok jogai elég finoman szabályozhatók). A 386-os nagyon jó alapot biztosít a UNIX operációs rendszer implementálásához, ezért érdemes a processzor architekturáját közelebbről is megismerni. Erről lesz szó itt röviden.

A processzor memóriakezelésének a lelkét két táblázat köré építik: ezek a táblák a **lokális-** és **globális leíró táblák** (LDT: local descriptor table, GDT: global descriptor table). Ezek a táblák írják le a szegmensek szerkezetét (például a szegmens helyét (báziscímét), méretét (lapokban vagy byteokban mérve) és védelmi jellemzőit). A GDT-t minden program közösen használja, és a fontosabb rendszerszegmensek (mondjuk az operációs rendszer szegmensei) érhetők el ezen keresztül. Minden program rendelkezik egy-egy saját LDT-vel, amelyben a program saját szegmensei vannak leírva - amelyek a programkódot, adatokat és egyéb információkat tartalmazzák.

A processzornak 6 szegmensregisztere van: a CS, DS, SS, ES, FS és GS. Mindegyik szegmensregiszter egy 16-bites ún. szelektor értéket tartalmaz, amely egyértelműen

azonosít egy-egy leírótáblabeli elemet. (Pontosabban a 16 bitből 1 bit mondja meg, hogy a lokális vagy globális deszkriptor tábláról van-e szó; további 13 bit mondja meg, hogy az adott táblán belül melyik sorszámú szegmensről van szó; a maradék két bit pedig védelmi információkat tartalmaz.) A szegmensregiszterek közül a CS a **kódszegmens**t azonosítja (vagyis azt a szegmenst, amely a végrehajtandó kódot tartalmazza), a DS az **adatszegmenst** (vagyis azt a szegmenst, ahonnan a program a futásához szükséges adatokat veheti), az SS pedig a program **verem szegmens**ét azonosítja. A másik három szegmenst (ES, FS, GS) a programozó arra használja, amire akarja - például egyes utasítások elé egy-egy ún. prefix byteot írva elérhető az, hogy a processzor az adott utasítás végrehajtása során a DS szegmens helyett mondjuk az ES-t használja az adatok elérésére.

Ezen kívül egy speciális kontroll-regiszter tartalmazza a laptábla kezdőcímét a memóriában - ez alapján történik az ún. lineáris cím fizikai címmé konvertálása (a 386-os processzor 4 KByte-os lapokat kezel). A **lineáris cím** kiszámítása a következőképpen történik: amikor egy program egy adott szegmens adott bytejára hivatkozik (a byte **offszet**jének nevezik a byte szegmensen belüli helyét), akkor a processzor a szegmensszelektornak megfelelő leírótáblaelemből a báziscímet és a bytehoz tartozó offszetet összeadja, és így kapja meg az ún. lineáris címet. A lapozás természetesen letiltható (vagyis "kikapcsolható"), és ekkor a lineáris cím megegyezik a kérdéses byte fizikai memóriabeli címével.

A processzor védelmi modellje a következő: 4 ún. logikai védelmi **gyűrű** van (ezek közül a 0-ás sorszámú a legtöbb privilegiumot biztosító, míg a 3-as sorszámú a legkevesebb privilegiumot biztosító, ún. legkevésbé privilegizált gyűrű). Minden programról tárolva van, hogy melyik gyűrűben fut (ez az, amit a program nem változtathat meg), és minden egyes szegmeshez is tárolva van a deszkriptortáblában egy-egy ilyen privilegium-szint. A legfontosabb szabály az, hogy a program nem nyúlhat bele a nálánál jobban privilegizált szegmensekbe. Egyéb esetekben ún. (általános) védelmi kizárás keletkezik, amikor az operációs rendszer kapja meg a vezérlést, és a "saját belátása szerint" cselekedhet (mondjuk kilőheti a szabálytalankodó programot). A privilegizáltabb kódszegmensekben tárolt eljárások meghívása is csak ellenőrzött módon történhet - csakis a deszkriptortáblában tárolt ún. call-kapukon keresztül kerülhetünk privilegizáltabb kódszegmensbe. (Egy-egy ilyen call-kapu (call gate) lényegében a privilegizáltabb szegmens "nyilvános" **belépési pont**jait tárolja, és nem lehet csak úgy egy privilegizált szegmens belsejébe "beleugrani".)

Ebben a pontban – folytatva a mikroprocesszor architekturák ismertetését – az IBM által a 90-es években kifejlesztett Power mikroprocesszor architekturát fogom bemutatni. Ez egy csökkentett utasításkészletű (RISC) processzor (az utasításkészlet csökkentése az egy utasítás értelmezésére/dekódolására szükséges idő csökkentése érdekében hasznos – ezzel "gyorsabb" lesz a processzor).

(majd egyszer ...)

1.11 Szabványok

A nyílt rendszerek "nyíltságának" az az alapja, hogy a kommunikációjuk megfelelően (szabványokban) rögzített protokollok alapján történik - ezek a szabványok teszik lehetővé a kommunikációt. A ma kialakult szabványok nagyon sokrétűek és sokfajták: rögzítik azt, hogy az alapszoftvernek mit kell tudnia, az alapszoftver egyes moduljainak milyen interface-ei legyenek (és még sok más dolgot).

A legismertebb alapszofvterrel kapcsolatos szabványok: a **POSIX** (Portable Operating System Inteface for UNIX) és az **XPG3** (X/Open Portability Guide Volume 3),

valamint létezik még az AT&T által kiadott **SVID** (UNIX System V Interface Definitions) szabvány is.

A SVID nyilván az AT&T elképzelései alapján készült (a kialakult 4.3BSD UNIX-szal szemben), a POSIX szabványok inkább az AT&T és a 4.3BSD UNIX "metszete" alapján, míg az XPG3 inkább az AT&T és a 4.3BSD UNIX "uniója" alapján készült el. Természetesen ezek a szabványok sok nem a UNIX-szal kapcsolatos dolgot is tartalmaznak.

Most röviden összefoglaljuk, hogy milyen ismertebb (gyakrabban használt) komponensei vannak a POSIX operációs rendszer interfész szabványoknak:

- **1003.1** : C könyvtári függvények – ide tartoznak a rendszerhívások is, mivel azok is a C könyvtáron keresztül érhetők el.
- **1003.2** : Parancsértelmezők és segédprogramok – vagyis ide tartozik az is, hogy mit kell tudnia egy-egy shellnek.
- **1003.3** : Annak az ellenőrzésére módszerek, hogy egy rendszer illeszkedik-e a POSIX szabványokhoz.
- **1003.4** : Valós-idejű rendszerekre vonatkozó szabványok (itt lehet időbeli korlátokat is biztosítani a szolgáltatásokra).
- **1003.4a** : Többszálú (multithreaded) alkalmazások írására vonatkozó szabványok.
- **1003.5** : Egy szabványos Ada könyvtár interfész a POSIX-hoz.
- **1003.6** : Security – biztonságossági kérdések vannak benne rögzítve.
- **1003.7** : Rendseradminisztrációs lehetőségek.
- **1003.8** : (Hálózati) transzparens fájlelérés biztosítása a POSIX-hoz.
- **1003.9** : Egy szabványos FORTRAN könyvtár interfész a POSIX-hoz.

Fontos szerepe van a szabványosításban (főleg a programnyelvek és a számítógépes hálózatok terén) az ISO-nak (International Standards Organization) és az ANSI-nak (ez a szervezet az ISO tagja).

Érdekes megemlíteni, hogy az Internet világhálózatban használt TCP/IP kommunikációs protokollok a nagy elterjedésük miatt váltak de facto szabvánnyá, és ez mögött nem a fenti "nagy" szabványosító szervezetek állnak.

Léteznek olyan szabványok is, amelyek egy adott processzortípusra lefordított tárgykód hordozhatóságát akarják biztosítani a különféle operációs rendszerek között. Ilyen szabvány például az **iBCS2** (Intel Binary Compatibility Standard, 2. edition). A legtöbb 386-os PC-s UNIX megfelel ennek - ezért ezek a UNIX-ok egymás programjait minden további nélkül képesek futtatni. (Az iBCS2 nem csak ISA vagy EISA buszos 386-osokon (ill. 486-osokon) él! Pont ez a szép benne!)

Természetesen a nyílt rendszerek fogalma nem azonos a UNIX-szal, habár a nyílt rendszerekkel kapcsolatos (és elfogadott) szabványok legtöbbször UNIX-alapúak - a UNIX rendszerből származnak. Az OpenVMS operációs rendszer a DEC példája arra, hogy nem UNIX-os környezetben is biztosítható a "nyílt rendszer" kép.

1.12 Objektum-orientált felületek

Manapság nagyon terjedőben vannak az objektum-orientált eszközök illetve programozási nyelvek, viszont a legtöbb szabvány illetve operációs rendszer szolgáltatás valamilyen procedurális (azaz nem objektum-orientált) interfészen keresztül áll a programozók rendelkezésére. Ezért sokan készítenek a már meglevő procedurális szemléletű programkönyvtárakhoz olyan kiegészítéseket (ún. "köpenyeket"), amelyek objektum-orientált szemléletű hozzáférést biztosítanak az alatta levő nem objektum-orientált felülethez (ezek a kiegészítések általában valamilyen objektum-orientált nyelvi eszközöket (pl. öröklődés, osztályok) biztosító programozási nyelven készülnek, mint például a C++ vagy az Objective-C). Az objektum-orientált eszközök előnyei többek közt az újrafelhasználható eszközök tervezésében ill. készítésében valamint az eszközök – gyakran – könnyebb megérthetőségében, megtanulhatóságában rejlenek.

Eddig már számos köpenykészítési elv kialakult, itt most röviden áttekintjük ezeket.

1.12.1 Egyszerű köpeny

Akkor beszélünk egyszerű köpenyről, ha a köpeny az eredeti procedurális felülethez képest nem nyújt gazdagabb szolgáltatásokat. Ezek a köpenyek gyakran úgy készülnek, hogy egy jól definiált rendszerobjektum köré építik; biztosítják az adott objektum létrehozásához illetve megszűntetéséhez használható konstruktor illetve destruktor műveleteket, valamint objektum-metódusként biztosítják az eredeti nem objektum-orientált felület egyes műveleteit (kiegészítve esetleg konverziós műveletekkel, amik olyankor lehetnek szükségesek, ha az eredeti nem objektum-orientált felülethez hozzáférni szándékozó könyvtárakat ezen objektum-orientált felületre építve akarunk továbbhasználni).

1.12.2 Specializált köpeny

Specializált köpenyekről olyankor szokás beszélni, ha az általunk létrehozott objektum-osztályok metódusai és az eredeti programozói felület eljárásai közt nem lehet egy egyértelmű megfeleltetést létesíteni; ilyenkor általában egy-egy osztályművelet az eredeti procedurális felület több szolgáltatását is igénybe véve igen összetett feladatokat láthat és lát is el. Általában nehezebb ilyen köpenyeket jól megírni, viszont ezeket a köpenyeket gyakran könnyebb felhasználni, mint az egyszerű köpenyeket (ui. az egyszerű köpenyeknél a köpeny használójának még viszonylag jól kell ismernie az eredeti operációs rendszer felületet). Az is igaz, hogy ilyen specializált köpenyeket jobban lehet igazítani az alkalmazás-fejlesztők igényeihez, ami szintén a köpeny használójának a lehetőségeit könnyíti.

1.12.3 Objektum-orientált köpenyek tervezése

Az objektum-orientált eszközök (köpenyek) tervezésekor minden esetben azonosítani kell a rendszerbeli erőforrásokat (objektum osztályokat), valamint a rajtuk végezhető műveleteket. Operációs rendszerek esetén objektumok például a folyamatok, a fájlok, a memória, stb. Ezeken pedig sokféle művelet végezhető (amik természetesen operációs rendszerenként különbözőek lehetnek). Miután megvannak az objektum osztályok,

meg kell határozni a kapcsolatukat – a köztük levő esetleges öröklődési relációkat. Fontos, hogy az objektum-orientált köpenyek hibatűrőek legyenek, az eredeti procedurális felülethez képest, vagyis csökkentsék a hibásan használható operációs rendszer eszközök számát: segítse a programozót a programhibák kiküszöbölésében valamint felderítésében.

1.13 Mi lesz még

E rövid bevezető után a 2. és 3. fejezetben szó lesz a ma legelterjedtebb nyílt rendszernek, a UNIX-nak a működéséről, és a legalacsonyabb szintű programozói interfaceről: a rendszerhívásokról. Az ezután következő fejezetekben a nyílt rendszerek egymás közti kommunikációjáról lesz szó több szempontból is: a 4. fejezet a számítógépes hálózatokat mutatja be, valamint a legalacsonyabb szintű programozói interfacet, amelyet hálózati alkalmazások fejlesztésénél kihasználhatunk: a Berkeley socketokat és az XTI-t (X/Open Transport Interface).

A második fejezet nagyon képlékeny, most (1995-ben) már nem is igazán tetszik nekem, ezért várhatóan változni fog (bár ez a változás majd csak egy-két kiadás elteltével fog realizálódni; addig marad ez a megoldás).

1.14 Kérdések, feladatok

1. Mi az operációs rendszer, és mi a feladata?
2. Mit jelentenek a következő fogalmak?
   - cache-memória
   - gyermek-folyamat
   - taszk
   - processz-tábla
   - fájl
   - fájl-attributum
   - lap
   - lapkeret
   - paging
   - swapping
   - szegmens
   - multicomputer
   - multiprocesszor
3. Mit értünk hierarchikus directory-szerkezeten?
4. Mit értünk készülékfüggetlenségen?
5. Mi a védelem szerepe az operációs rendszerekben? Mit és ki elől kell védeni?
6. Mi a shell?

7. Keressünk példát olyan perifériára, amely nem teljesen illeszthető be sem a blokk-elérésű, sem a karakter-elérésű perifériákról alkotott képbe!

8. Mi a virtuális memóriakezelés?

9. Mi a processzor memóriakezelő egységének a szerepe?

10. *Lehet-e egy (virtuális-) memóriakezelő egység nélküli hardveren virtuális gépeket szimulálni? Mi okozhat itt komoly problémát?

11. *Lehetne-e mondjuk egy Intel 80386-on virtuális 80386-os gépeket szimulálni? Ha igen, akkor mit lehet mondani a szimuláció hatékonyságáról.

12. *A memóriába ágyazott fájlok hossza több operációs rendszerben is csak a processzor memóriakezelő egységének (MMU-nak) a lapméretének többszöröse lehet. Vajon miért? Milyen problémát akarnak így kiküszöbölni?

13. Miért fontos a fájlrendszer konzisztenciájának biztosítása?

14. Mi a holtpont és hogyan alakulhat ki?

15. Mit értünk az osztott rendszer fogalom alatt?

16. Mi a lényeges különbség a szorosan illetve a lazán kapcsolt rendszerek között?

Fejezet 2

A UNIX operációs rendszer

A UNIX operációs rendszer első változatát 1969-ben készítette el Ken Thompson az AT&T-nél egy leselejtezett PDP-7 számítógépen. Miután munkatársai is jó lehetőségeket láttak a programban, az AT&T szoftverfejlesztőinek egy része elkezdett ezzel komolyabban foglalkozni, és egyre újabb, fejletteb változatokat hoztak ki. Mivel a UNIX rendszert C nyelven készítették, nagyon könnyű volt átírni egy új hardverre, ezért nagyon hamar elterjedt az egész világon. Elterjedését segítette az is, hogy az első pár évben a rendszer teljes forráslistája bárki számára (ingyen) hozzáférhető volt. Ennek egy következménye volt az is, hogy a legtöbb helyen az egyetemi oktatásban ezt a rendszert használták. A UNIX hetedik változatának megjelenése után az AT&T látta a UNIX piaci sikerét, és a forráskód már csak a magas jogdíjak megfizetése ellenében volt hozzáférhető.

A UNIX legfőbb gyengesége az lett, hogy nagyon sok (többé-kevésbé eltérő) változata van. (Már kialakult többféle szabvány, például a SVID, amelyben azt specifikálják, hogy mit kell tudnia egy "igazi" UNIX-nak.) Ennek ellenére a UNIX alatt megírt programok sokkal hordozhatóbbak, mint például a DOS alatt megírtak. A UNIX már majdnem minden számítógépre át lett írva (az IBM PC-től kezdve a szuperszámítógépekig).

Az AT&T UNIX változata mellett jelentős a BSD UNIX (Berkeley Software Distribution - ennek a jelenlegi (aktuális) változata a 4.4-es BSD UNIX), és a Microsoft XENIX rendszere is.

A UNIX egy multitaskos és többfelhasználós operációs rendszer, ezért alkalmas arra, hogy az ilyen és ehhez hasonló rendszerekben felmerülő problémákat ezen vizsgáljuk meg.

2.1 Néhány alapvető UNIX-beli fogalom

A következő fogalmak ismeretére lesz szükség:

- **uid**: A felhasználó azonosítója (a rendszernek minden egyes felhasználónak egy egyedi ilyen azonosítója van).
- **gid**: Csoportazonosító. A UNIX rendszerben minden felhasználó be van osztva egy csoportba. A gid annak a csoportnak az azonosítója, amelybe a felhasználó tartozik. (A csoportbeosztás tetszőleges lehet; van olyan rendszer, ahol minden felhasználó egy közös csoportba tartozik, és például az egyetemeken gyakori a teachers illetve students csoport.)
- **pid**: Folyamat-azonosító.

- **pgrp-id**: Folyamat-csoport azonosítója. Ez egyenlő a folyamat-csoport vezetőjének a pid-jével (minden folyamat tagja valamely folyamat csoportnak, minden folyamat megalapíthat egy saját folyamat-csoportot, és lehetőség van például egy folyamat-csoport minden tagjának a "kilövésére" egyetlen művelettel).

- **tgrp-id** : Terminál group-id. Minden folyamathoz ez is tárolva van. Ez egyenlő annak a folyamatnak a pid-jével, amely a folyamathoz tartozó terminál-(képernyő)fájlt legelőször megnyitotta. Ez általában a legelőször elindult login shell.

- **euid**: (effektiv user id) - általában egyenlő az uid-del, a felhasználói azonosítóbval, de bizonyos esetekben (ún. setuid-bites programoknál) más is lehet. Ilyen módon egy adott folyamatnak több jogot lehet adni, mint ami a folyamat elindítójának van.

- **egid**: (effektiv group id) - mint euid, csak a csoport-azonosítóra.

- **szuperfelhasználó**: minden jogokkal rendelkező felhasználó. A felhasználói azonosítója általában 0 szokott lenni minden UNIX-ban, és felhasználói neve (login neve) általában `root`.

2.2 Folyamatok a UNIX rendszerben

A UNIX rendszer egy igazi multitaskos rendszer, minden folyamat létrehozhat egy vagy több gyermek-folyamatot, és a gyermek-folyamatok a szülő-folyamattal párhuzamosan futnak. A gyermek-folyamat örökli a szülőjének a jogait, és egyéb tulajdonságait. A processz-táblában egy folyamathoz (többek között) a következő információk vannak tárolva:

- pid (folyamat azonosító)

- pgrp-id (folyamat-csoport azonosító)

- A szülő-folyamat azonosítója

- A processzor-regiszterek értékei

- A folyamat által elhasznált CPU idő

- A folyamat gyermek-folyamatainak a száma

- A folyamatot elindító felhasználó felhasználó azonosítója és a csoportjának az azonosítója.

- A folyamat effektiv uid-je és gid-je (ezt ld. később)

- A folyamat munka-directoryja (working directory)

- A folyamat megnyitott fájljai

(A folyamat kulcsfogalomnak számít minden operációs rendszerben - nem csak a UNIX-ban. Az, hogy mi tartozik egy folyamathoz nagyon nagy mértékben befolyásolja az egész operációs rendszer lehetőségeit és szerkezetét.)

Az operációs rendszer egy kicsi, de fontos része a **folyamat-ütemező** (scheduler). Mivel a legtöbb UNIX-ot futtató hardveren csak egy processzor van, ezért ezen szimulálni kell a folyamatok párhuzamos futását. Ez pedig a következőképpen történik:

1. Az ütemező a futásra váró összes folyamat közül valamilyen szempont szerint kiválaszt egyet, és átadja annak a vezérlést.
2. (Ha a folyamat valamikor a háttértárra kikerült, valamilyen virtuális memória-kezelő művelet eredményeként, akkor előbb visszahozza onnan.)
3. A kiválasztott folyamat elkezd futni (vagyis megkapja a CPU-használati jogot), egész addig, amíg az ún. időszelete le nem jár. Ez azt jelenti, hogy ha a folyamat időszelete mondjuk 40 millisec., akkor a folyamat (kb.) 40 millisec.-ig fog futni.
4. Ha a folyamat időszelete lejár, akkor ismét az ütemező kezd el működni, az újabb folyamatot választ ki és annak adja át a vezérlést (vagyis vissza az 1. ponthoz).

A fentiekben az egyetlen problémás dolog az, hogy hogyan "jár le a folyamat időszelete" (vagyis egy folyamatnak önmagának "le kell-e mondania" a processzorról, vagy pedig az időszelet lejárta után az operációs rendszer automatikusan el tudja-e venni a folyamattól a processzorhasználati jogot). A multitaskos rendszerekben legtöbbször a második eset áll fenn (a UNIX rendszereknél mindig). Ha az első eset állna fenn, akkor lehetne olyan folyamatot írni, amely sosem mond le a processzor használatáról, és ezzel a teljes operációs rendszer megbénulna, nem tudná ellátni a feladatát (az erőforrások igazságos elosztását).

Az Intel 8088-as mikroprocesszor esetén ez az óra-megszakítások kihasználásával oldható meg. Az IBM PC-ben van egy belső óra, amely 1 másodpercben (kb. 100-szor) egy ún. óra-megszakítást generál. Ennek a megszakításnak van egy sorszáma, tehát tartozik hozzá egyértelműen egy megszakítás-vektor. Az operációs rendszert ilyen gépen úgy írják meg, hogy a megszakítás-vektor az ütemező modul memóriabeli címét tartalmazza, és ilyen módon minden másodpercben kb. 100-szor, amikor a belső óra "üt", akkor az ütemező automatikusan megkapja a vezérlést és új folyamatot választ ki.

2.3 A folyamatok közötti kommunikáció (IPC) a UNIX rendszerben

Kezdetben a UNIX csak a **pipe**okat tartalmazta mint IPC-eszközt (ezt ld. később), ami egy nagyon primitív eszköznek bizonyult, mert a kommunikáció csak olyan folyamatok között történhetett, amelyeknek van közös őse. Az IPC más eszközei a UNIX rendszerbe csak a fejlesztésének egy késői szakaszában kerültek be, ezért az AT&T UNIX (ezzel együtt a Microsoft XENIX-e) és a BSD UNIX (ezzel együtt például az Ultirx) rendszerek más eszközöket nyújtanak a programozók számára az IPC megszervezésére. Itt az AT&T UNIX által nyújtott eszközök lesznek bemutatva, mert később az lett szabványosítva (az X/Open szabványba ez került be), igaz a másik tábor, a Berkeley (BSD) UNIX fejlesztőinek csoportja ezt a tényt mindmáig egyszerűen figyelmen kívül

hagyta. (A legtöbb BSD UNIX forgalmazó azért ad valamilyen könyvtárakat, amelyek az AT&T rendszerhívásokat implementálják vagy szimulálják.) Az AT&T UNIX háromféle eszközzel rendelkezik ezen a téren: **szemafor**okkal, **osztott memóriá**val (shared memory) és **üzenetátadás**sal (message passing).

A szemafor egy 0 és 32767 közötti értéket vehet fel, és a programozó ennek az értékét növelheti és csökkentheti úgy, hogy e két művelet **oszthatatlan művelet**nek tekinthető, vagyis az operációs rendszer gondoskodik arról, hogy ne tudják ketten egyszerre ugyanannak a szemafornak az értékét megváltoztatni úgy, hogy a változtatás a rendszerben inkonzisztenciát okozna. (Inkonzisztencia például onnan eredhetne, hogy ha két folyamat egyszerre próbálja egy 2 byteos szemafor értékét megváltoztatni, és eközben az egyiknek kiosztott időszelet lejár miután a két byte egyikét átállította (de a másik byteot még nem), a vezérlést megkapja a másik folyamat, az átállítja a szemafor értékét a maga elképzelései szerint, majd amikor a vezérlést visszakapja az előző folyamat, akkor az átállítja a másik byteot, és ezzel kaotikus állapotok alakulhatnak ki).

Az osztott memória használata lehetővé teszi azt, hogy egy bizonyos memóriaterületet (nyilván a benne tárolt adatokkal együtt) egyszerre több folyamat is lássa.

Az üzenetek lehetővé teszik egy-egy **memóriaterület (buffer) tartalmának átküldését az egyik folyamattól a másiknak**. Két primitív művelet van az üzenetek kezelésére: üzenet elküldése és üzenet fogadása. Az üzenetfogadó primitív lehet **blokkoló** vagy **nem-blokkoló** aszerint, hogy ha a végrehajtásának pillanatában még nem érkezett a folyamat számára üzenet, akkor a program várjon egy üzenet beérkezéséig vagy fusson tovább jelezve azt, hogy nem volt beolvasható üzenet.

2.4 A UNIX fájlrendszere

A UNIX operációs rendszer fájlrendszere hierarchikus, a rendszerben egyetlen gyökér-directory van, és annak tetszőleges mélységben tetszőleges számú aldirectoryja lehet, azoknak az aldirectoryjainak szintén tetszőlegesen sok aldirectoryjuk lehet, stb. A UNIX a fájlt egyszerűen egy byte-folyamnak tekinti. A UNIX rendszerben létezik a munka-directory koncepciója (minden folyamatnak van egy saját munka-directoryja). A UNIX abban lényegesen különbözik az MS-DOS-tól, hogy csak egy gyökér-directoryja van, és a mágneslemezeken levő külön fájlrendszerek (a lemezen kialakított directory-strukturák) beilleszthetők (mountolhatók) a gyökérdirectory- rendszerbe. Erre nézzünk egy példát – a UNIX gyökér-fájlrendszere tartalmaz több directoryt:

fájlnév	szerepe
/etc	egy aldirectory a rendszerfájloknak
/etc/passwd	a jelszófájl a /etc directoryban (ez FILE!)
/etc/limit	egy aldirectory a /etc directoryban
/usr	a felhasználók directoryjait tartalmazó directory
/usr/csb	a "csb" nevű felhasználó fájljait tartalmazó directory
/lib	a UNIX C könyvtárait tartalmazó directory
/tmp	temporális fájlokat tartalmazó directory
/bin	Fontosabb rendszerprogramokat tartalmazó directory.

Most tegyük fel, hogy van egy fájlrendszerünk egy floppy-diszken, amely a következő dolgokat tartalmazza:

fájlnév	szerepe
/alma	valamilyen fájl
/mylib	a fájlrendszeren egy "mylib" nevű directory
/mylib/alma	a mylib directoryn belül az alma nevű fájl
/barack	egy tetszőleges barack nevű fájl

A felhasználó kérheti a UNIX rendszert, hogy illessze be a floppy-diszken levő fájlrendszert a gyökér-fájlrendszer valamelyik üres directoryjába. Most nézzük, hogy mi lesz akkor, ha a fenti gyökér-fájlrendszer `/usr/csb` directoryjába beillesztjük a fent elképzelt floppy-diszken levő fájlrendszert. Ezután a floppyn levő fájlokat a következő neveken érhetjük el:

/usr/csb/alma	(a floppyn a /alma nevű fájl)
/usr/csb/mylib	(a floppyn a /mylib directory)
/usr/csb/mylib/alma	(a floppyn a /mylib/alma fájl)
/usr/csb/barack	(a floppyn a /barack nevű fájl)

Természetesen ha ezekre a fájlokra máshol lenne szükség (pl. a `/tmp` directoryban, akkor oda is be lehetne őket illeszteni - mountolni). Ez az alapja a UNIX ún. **készülékfüggetlenség**ének. A fájlokra való hivatkozáskor nem kell tudni azt, hogy az adott fájl milyen fizikai periférián van. Ehelyett elég a fájlnak a fájlrendszer-hierarchiabeli nevét megadni.

A UNIX-ban minden egyes fáhlhoz tartoznak ún. **védelmi bitek** (ezeket **rwx bitek**nek is nevezik). A fájlokat ezekkel lehet védeni a jogosulatlan hozzáférés ellen. Mint már szó volt róla, minden felhasználónak van egy saját uid-je, gid-je. Lényegében ez a UNIX fájlvédelmének az alapja. Minden egyes fájlhoz tárolva van az őt létrehozó felhasználó uid-je és gid-je, illetve a fent említett rwx- bitek (összesen 9 bit). Az rwx-ből az "r" betű a fájl elolvasási jogát jelenti, a "w" betű a fájl felülírási (módosítási) jogát jelöli, és a végrehajtható programok esetén az "x" bit a végrehajtási jogot jelöli. Az rwx-bitekkel meg lehet adni, hogy a fenti 3 jog közül a fájlt létrehozó felhasználónak milyen jogai vannak a fájlon; a fájlt létrehozó felhasználó csoporttársainak milyen jogai vannak; és végül meg lehet adni, hogy azoknak az "egyéb" felhasználóknak, akik a fenti két halmaz egyikébe sincsenek benn mik legyenek a jogai. (A védelmi bitek sorrendje ugyanez; először a tulajdonos, majd a csoporttársak, végül pedig az egyéb felhasználók jogait kell megadni.) Ha egy fájl védelmi bitjei `r-xr-x--x` alakúak, akkor a tulajdonos a fájlt elolvashatja, végrehajthatja (de nem módosíthatja); a tulajdonos csoporttársainak ugyanez a joguk; az egyéb felhasználóknak pedig csak végrehajtási joguk van. A fájlhoz tartozó védelmi biteket csak a fájl tulajdonosa vagy a root (superuser) változtathatja meg.

Ezen kívül néhány végrehajtható fájlnál érdemes használni a UNIX egy más speciális lehetőségét: a sticky bitet. Ha ez a bit be van állítva egy végrehajtható fájlon, és a fájlban tárolt programot végrehajtják, akkor a program befejeződése után a kódszegmense (ha az nem változott meg) megmarad a winchester virtuális memória területén - így egy következő végrehajtás valószínűleg gyorsabban fog elkezdődni, mivel a kódszegmens újbóli betöltése megspórolható. (Egy speciális védelmi bitről - a setuid bitről később lesz szó.)

A UNIX rendszerben egy másik nagyon jó ötlet az ún. speciális fájlok koncepciója. Ilyen módon az egyes hardware perifériákat a fájlrendszerbeli nevükön érhetjük el. Az

ötlet azon alapszik, hogy például egy 360 KByte-os (mondjuk IBM PC/XT formátumú) floppy-diszket tekinthetünk úgy, mint egy 360 KByte-os fájlt. Ha a lemezen a blokkméret 512 byte, akkor az ilyen fájl első karaktere a diszk 0-adik blokkjában az első byte; a fájl 512-edik karaktere a diszk 0-adik blokkjában az utolsó (512-edik) byte, például az 1025-ödik byte pedig a diszk második blokkjának az első karaktere ... Vannak **blokk-speciális fájl**ok, és karakter-speciális fájlok. A blokk-speciális fájlok abban különböznek a karakter-speciális fájloktól, hogy a blokk-speciális fájloknak néhány "gyakrabban használt" blokkja a cache memóriában marad a gyorsabb elérés érdekében, míg a karakter- speciális fájloknál erre nincs lehetőség.

A UNIX fájlrendszrében a `/dev` directory tartalmazza az ilyen speciális fájlokat. Például a legtöbb rendszerben a `/dev/fd0` néven a 0-ás lemezegységet érhetjük el (PC-ken ez az MS-DOS alatt az `A:` jelű lemezegység).

A UNIX másik érdekes lehetősége a **pipe** (csővonal). Ez a párhuzamos folyamatok közötti kommunikáció (IPC) egyik eszköze. Ezt tényleg úgy tekinthetjük, mint két folyamat közti csővonalat: az egyik folyamat írhat ebbe a csővonalba valamilyen adatokat, a másik folyamat pedig kiolvashatja a beírt adatokat a pipe-ból. (A pipe csak egy half-duplex (egyirányú) kapcsolatot biztosít, vagyis ha mindkét résztvevő folyamat akar a másiknak adatokat küldeni, akkor a kétirányú adatátvitelhez két ilyen pipe-ra van szükség.) A pipe-ra írni és arról olvasni is a UNIX fájlműveleteivel lehet. Sőt lehetőség van arra is, hogy pipeokat névvel lássunk el (a pipeok nevei ekkor fájlnevek lesznek, amin keresztül bármelyik folyamat tud a pipera írni - esetleg arról olvasni, ehhez csak a pipehoz tartozó fájl nevét kell ismernie).

Érdekes még, hogy a hálózati kapcsolatok is lényegében fájldeszkriptorokon keresztül érhetők el, de erről majd később lesz szó.

A UNIX egy többfelhasználós operációs rendszer, így egy fájlt egyszerre több felhasználó is változtathat. Ez gyakran problémák forrása is lehet. Nézzük a következő helyzetet: egy bank számítógépén UNIX rendszer fut, a számítógéphez több képernyő van kapcsolva (minden egyes pénztáros asztalán van egy-egy terminál). Tegyük fel, hogy két pénztárhoz egyszerre megy oda két ügyfél, és ugyanarra a bankszámlára mindketten 500 forintot akarnak rakni. Ekkor mindkét számítógép kiolvassa a számla aktuális egyenlegét (ez legyen mondjuk 1000 forint), kiszámolja, hogy mennyi lesz az 500 forint berakása után az új egyenleg (1500 forint), és visszaírja azt. Mivel ezek az ügyfelek lehet, hogy nem is tudnak arról, hogy a másik is ugyanarra a számlára rakott be 500 forintot, boldogan mennek hazafelé a bank-igazolással, amelyen új egyenlegként 1500 forint van feltüntetve, nem veszik észre, hogy a gép hibázott. Mivel a gyakorlatban az ilyen eseteket nem szabad elhanyagolni (egy nagyobb bank központi számítógépére több ezer terminált rá lehet kapcsolni), és sok olyan számlaszám van, amelyre sokan fizetnek be pénzt (pl. közhasznú alapítványokra), ezért erre kell valami megoldást találni.

A UNIX az ilyen esetekre nyújtja a **fájl- ill. rekord-lefoglalás** (fájl and record locking) lehetőségét. Egy folyamat kérheti az operációs rendszert, hogy egy fájlt vagy annak egy részét "foglaljon le" egészen addig, amíg a folyamat el nem végzi rajta a dolgát (mondjuk inkább úgy, hogy a feladatát ...). A lefoglalás ideje alatt más folyamat nem nyúlhat a fájl lefoglalt részéhez; ha mégis hozzá akarna nyúlni, akkor várnia kell addig, amíg a fájlt lefoglaló folyamat "elengedi" a fájlt.

Másik fontos eltérés a UNIX és az MS-DOS fájlrendszere között az, hogy a UNIX fájlrendszerben nem egy közös nagy MS-DOS-ban használt FAT-szerű táblázatban tárolják azt, hogy a fájl a diszk melyik blokkjain helyezkedik el, hanem minden egyes

fájlhoz külön tárolják (egy-egy a fájlhoz tartozó ún. i-nodeban) azt, hogy a fájl a diszk melyik részén helyezkedik el. Az i-node tárolja azt is, hogy ki a fájl tulajdonosa (mi volt a tulajdonos uid-je és gid-je akkor amikor a fájlt létrehozta), mekkora a fájl hossza (byteban mérve), mikor hozták létre, mikor módosították utoljára, mik a fájl védelmi bitjei. Minden egyes i-nodenak van egy sorszáma, és a directoryk (lényegében speciális fájlként - amit megnyithatunk és ezeket az információkat kiolvashatjuk belőle) a következő információk sorozatát tartalmazzák:

maximum 14 karakteres fájlnév	hozzá tartozó i-node sorszáma

Ez azért hasznos, mert így egy fájlra több directoryból is hivatkozhatunk. Ezzel megoldhatjuk azt, hogy két vagy több felhasználó ugyanazt a fájlt elérje a saját directoryjából. Erre nézzük a következő példát: a `/usr/pista` directoryban van egy ilyen bejegyzés :

test.c	234

a `/usr/mariska` directoryban pedig a következő bejegyzés van (a szerkezet a fent bemutatottak szerint értendő):

proba.c	234

Itt a `test.c` illetve a `proba.c` fájloknak más a neve, de a két fájl (tartalma) egy és ugyanaz. Ha mondjuk a `/usr/mariska/proba.c` fájl le lesz törölve, akkor a fájl (a benne tárolt adatokkal) a lemezről nem lesz letörölve, mert még hivatkoznak rá `/usr/pista/test.c` néven. (Egy ilyen hivatkozást neveznek az i-nodera mutató linknek - nyilván az i-node tartalmaz egy referenciaszámlálót, és a fájl csak akkor lesz ténylegesen törölve, ha e referenciaszámláló értéke nullára csökken.) Eszerint egy fájl a lemezről csak akkor lesz letörölve, ha rá egyetlen link sem vonatkozik.

Itt érdemes megjegyezni azt is, hogy az i-node tartalmazza azokat az információkat, amely alapján a fájl helye a diszken "feltérképezhető" a következők szerint . Az i-node tartalmaz 13 darab blokk-címet (a UNIX csak blokk-elérésű perifériákon tud fájlrendszert létrehozni) - most a példa könnyebb megértése érdekében feltételezzük, hogy a blokkméret 1 KByte. Ebből a 13 blokk-címből 10 darab ún. direkt cím. Azaz azok közül az első tartalmazza annak a (diszk-)blokknak a címét, amelyen a fájl első 1024 (1 Kbyte) darab byteja van. A második tartalmazza annak a (diszk-)blokknak a címét, amelyen a fájl következő 1024 byteja van, stb. . A 11-edik blokk-cím egy ún. **egyszeres indirekt** blokk-cím (single indirect): az a blokk, amelyet ez az egyszeres indirekt blokk-cím megcímez mind direkt címeket tartalmaz. (Ha egy blokk-cím mérete n byte, akkor összesen 1024/n darab direkt cím fér ide.) Vagyis az egyszeres indirekt blokkban tárolt első cím a fájl 11-edik kilobyteját tartalmazó diszk-blokk címét tartalmazza.

A 13-ból a 12-edik egy ún. **kétszeres indirekt** blokk-cím (double indirect address): az a blokk, amelyet ez a kétszeres indirekt blokk-cím megcímez mind egyszeres indirekt címeket tartalmaz - ezek szerkezete olyan, mint azt az előbb leírtam.

A 13-adik pedig egy **háromszoros indirekt** blokk-cím (triple indirect): az ez által megcímzett blokk 1024/n darab kétszeres indirekt blokk- címet tartalmaz. (Ha egy fájl hossza mondjuk 4567 byte, akkor csak az első 5 direkt cím van kitöltve "értékes" információkkal. A többi helyen mindegy mi van.)

2.5 A UNIX shelljei

A UNIX-ban (az MS-DOS-hoz hasonlóan) többfajta shell lehet, és minden egyes felhasználó a rendelkezésre álló shellek közül választhat egyet magának, amit majd használni fog. Jelenleg a legelterjedtebb shellek a következők: Bourne-shell (ez minden UNIX rendszerben megtalálható); C-shell (ezt a shell már kevesebben használják); a Korn-shell pedig a legújabb shell, ami a Bourne-shell kiterjesztésének tekinthető (azzal kompatibilis).

A UNIX shelljei programozhatók (mint például a nagy IBM gépek shellje, a REXX), a shell "programnyelve" a leginkább a C nyelvhez hasonlít. Hatékonyságát mutatja, hogy adatbáziskezelő rendszereket is írtak már benne (ehhez egyéb "szabványos" utilityket is felhasználva, mint például az awk - a shell önmagában nem volt ehhez elég).

2.6 Védelem a UNIX operációs rendszerben

A UNIX rendszer védelmi rendszere elég jó. A rendszerbe bejelentkezni minden felhasználó csak a jelszavával tud. A UNIX megvédi az egyik felhasználó futó programjait egy másik felhasználó illegális beavatkozása elől (vagyis nem tudom a tanár programját "kilőni"). Másrészt pedig lehetőség a fájlok védelmére a jogosulatlan hozzáférések elől. A UNIX védelmi koncepciójához hozzátartozik az is, hogy van egy ún. szuperfelhasználó (legtöbbször ő a "rendszergazda" - a root), aki majdnem minden védelmi korlátot megsérthet, mindenkinek a fájljaihoz hozzáférhet. (Erre szükség is van, különben nem tudna a rendszerben levő fájlokról biztonsági másolatokat készíteni. Ha nem készülnének biztonsági másolatok, akkor egy rendszerhiba esetén lehet, hogy egyes felhasználóknak nagyon sok értékes munkája veszne el.) (Most már érthető, hogy miért próbálják meg például az egyetemeken a hallgatók a szuperfelhasználó jelszavát kitalálni.)

A UNIX ellenőrizhető módon lehetőséget ad arra, hogy egy felhasználó valamely más felhasználó jogaival rendelkező folyamatot elindítson. Ennek eszköze a **setuid bit**. Ez azt jelenti, hogy ha egy végrehajtható program el van látva a setuid bittel, akkor a program végrehajtása alatt nem a saját uid-ünknek megfelelő jogaink vannak, hanem annak a felhasználónak a jogaival rendelkezünk, akié a program (az lesz a folyamat effektiv user-id-je). Ezzel nagyon óvatosan kell bánni! Különösen veszélyesek lehet az, ha egy program ún. setuid root program, vagyis ha valaki ezt a programot végrehajtja, akkor a végrehajtás ideje alatt a szuperfelhasználó jogaival rendelkezik. Ilyen setuid root program például a `passwd`, amellyel bármelyik felhasználó megváltoztathatja a saját jelszavát. A programnak a szuperfelhasználói jogok azért kellenek, hogy a jelszófájlt (`/etc/passwd`) meg tudja változtatni. A setuid bites programokat a következőképpen lehet felismerni: az `ls -l` parancs által kiadott directory-listában a program tulajdonosára vonatkozó `rwx` bitek `x` betűje helyén egy `s` betű áll (pl. `rws--x--x`). Később még lesz szó róla, hogy milyen eszközökkel lehet viszonylag "biztonságos" setuid root programokat írni.

Az IPC eszközökhöz is tárolva van a létrehozó folyamat felhaszálói azonosítója és csoport-azonosítója (uid ill. gid), illetve tárolva vannak hozzá a szokásos rwx-bitek is. Ez teszi biztonságossá a folyamatok közötti kommunikációt.

2.7 A UNIX INPUT/OUTPUT rendszere

A UNIX a hardver berendezésekkel a **device driverek** segítségével kommunikál (ott van "elrejtve" az operációs rendszer gépfüggő, hardver-függő része). A device driverek nem illeszthetők be olyan dinamikusan egy hagyományos UNIX rendszerbe, mint ahogyan az az MS-DOS alatt megy (a CONFIG.SYS fájlon keresztül), és a device driverek nem olyan felépítésűek, mint mondjuk a DOS megfelelőik (egy UNIX device drivernek más szolgáltatásokat kell nyújtania, mint egy DOS device drivernek). Azt érdemes tudni, hogy egyik PC-s UNIX sem használja a BIOS-t, mert a BIOS rutinok úgy vannak megírva, hogy várnak az I/O művelet befejeződéséig, és csak ezután adják vissza a vezérlést az operációs rendszernek, ami egy multiprogramozott operációs rendszerben elfogadhatatlan.

A device drivereket a UNIX kernel többi részével össze kell linkelni (itt most a linkage editorra gondoljunk, ne a fájlrendszer linkjeire). A device driverek bizonyos szolgáltatásokat nyújtanak a kernelnek (pl. minden device drivernek lehet egy olyan rutinja, amelyet a rá vonatkozó open()-nél kell meghívni ...). Ezek a rutinok egy az egész kernelre nézve globális tömbben vannak felsorolva. Egy device driver szolgáltatásainek ezen tömbbeli indexét nevezik a devicehoz tartozó major device numbernek. (Lényegében ezen keresztül lehet a device drivereket egymástól megkülönböztetni.)

Minden device driver kezelhet több (egymástól akár független akár nem független) perifériát is, amiket az ún. minor device numberekkel különböztethet meg egymástól. (Egy speciális fájlhoz tartozó az i-nodeban a 13 blokk- cím helyén van többek közt a minor ill. major device number tárolva.) Amikor egy speciális fájlt megnyitunk a fájlrendszerben tárolt neve alapján, akkor az operációs rendszer az i-node alapján tudja, hogy az egy speciális fájl, megszerzi az i-nodeból a major ill. minor device numbereket, és a device driver open() szolgáltatását végző rutint meghívja (többek közt a minor device numberrel mint paraméterrel).

A device driverek lényegében három csoportba sorolhatóak: a karakteres interface-t nyújtóak, a blokk-interfacet nyújtóak és a STREAMS driverek.

2.7.1 A UNIX architekturájának modernizálása

A UNIX operációs rendszer egy monolitikus rendszer: van ugyan jó néhány logikailag egymástól elkülöníthető komponense, de a jelenlegi strukturában még akadnak problémák az új hardware-re történő átíráskor. A Carnegie Mellon egyetemen az utóbbi években leginkább azzal foglalkoztak, hogy hogyan lehetne a UNIX-ot (ők a kísérletezéseikre a Berkeley UNIX 4.3BSD változatát használták) gépfüggő illetve gépfüggetlen részekre bontani. E munka eredményeként jött létre a Mach mikrokernel, amely a módosított Berkeley UNIX "gépfüggő" részeiből lett kifejlesztve. A Mach-ot is egy operációs rendszerré alakították: a UNIX "gépfüggetlen" részét úgy írták át, hogy az a Mach operációs rendszer szolgáltatásait használja ki ott, ahol valamilyen gépfüggő szolgáltatást kell elvégeznie. Az így elkészült rendszer teljesen kompatibilis az eredeti Berkeley UNIX-szal (így bináris kompatibilitást is el tudtak érni, vagyis a régebbi 4.3BSD-n megírt programokat újra se kell fordítani ahhoz, hogy az új Mach-alapú rendszereken használjuk őket).

A Mach mikrokernel szerkezete

Ez a rész röviden vázolja a Mach 3 kernel felhasználói számára nyújtott felületét.

A Mach rendszert operációs rendszerek alapjának tervezték. Széleskörű lehetőségekkel rendelkezik a memóriakezelés terén, biztosítja a folyamatonként (Mach terminológiában taszkonként) több végrehajtási pont létrehozásának lehetőségét (angol nevén threadek alkalmazását) és jó folyamatok (taszkok) közti kommunikációs eszközöket nyújt. A Mach alapvető jellemzői a következők:

- Rugalmas memóriakezelési technikák biztosítása a futó folyamatoknak.
- Transzparens hozzáférési lehetőség biztosítása a hálózaton keresztül elérhető erőforrásokhoz, ehhez a megfelelő kommunikációs módszerek biztosítása.
- A korábbi szoftver-környezetekkel való kompatibilitás biztosítása (például a Berkeley UNIX rendszerrel).
- Lehetőséget kell teremteni minél magasabbfokú párhuzamosításra mind az operációs rendszerben mind pedig az alkalmazói programokban.

A 2.5-ös és korábbi változataiban a Mach rendszert beágyazták a Berkeley UNIX-ba és így biztosították a "hagyományos" (UNIX-szerű) operációs rendszer szolgáltatások jó részét. A 3.0-ás változattól kezdve a Mach már csak alapvető kernel szolgáltatásokat nyújt, nem biztosítja a kernelbe ágyazva a korábbi változatokban megismert széleskörű operációs rendszer szolgáltatásokat (mint például a fájlkezelési operációs rendszer szolgáltatásokat). Itt az operációs rendszer funkcionalitást nem a kernelbe ágyazva, hanem ún. kernelen kívül futó szerverekkel biztosítják.

Természetesen a taszkok kommunikációján kívül vannak más szolgáltatások is, amiket a kernelnek kell biztosítania – ilyenek például a következők:

- A taszkok és végrehajtási pontjaik (threadek) kezelése.
- A taszkok virtuális memóriájának biztosítása és kezelése.
- Hardware perifériák kezelése (például a processzorok, perifériák és az óra), valamint egy magasabbszintű hardware-interfész biztosítása az operációs rendszer szolgáltatásokat biztosító taszkok felé.

A Mach kernel absztrakciói: Bár a Mach kernel tervezésekor cél volt a kernel által biztosított absztrakciós eszközök számának csökkentése, nem volt cél az ezen eszközök által biztosított lehetőségek szűken tartása. A legfontosabb kernel absztrakciók a következők:

- **taszk** – Erőforrások lefoglalására képes egység (minden taszknak van egy saját memóriacímtartománya, és minden taszkhoz tárolva vannak a taszk port-hozzáférési jogosultságai is).
- **thread** – Programvégrehajtási pont (kicsi az erőforrásigénye).
- **port** – Kommunikációs csatorna, hozzáférés csak megfelelő adatküldési/adatfogadási jogon keresztül lehetséges.
- **üzenet** – Adatobjektumok gyűjteménye.
- **memória objektum** – A memóriakezelés belső egysége (ezeken keresztül vezérelhetik az egyes taszkok a memóriakezelést).

Taszkok és threadek: Mivel minden operációs rendszer nyilvántart bizonyos információkat a folyamatokról, mint például a futtató felhasználó azonosítóját, signalkezelHők[1] állapotát, és ez a folyamat absztrakció operációs rendszerenként más és más, ezért a Mach kernel (a 3.0-ás változattól kezdve) nem biztosítja a más operációs rendszerekben megtalálható folyamat fogalmat, a Mach folyamat fogalma sokkal primitívebb.

Egy taszk összes threadje hozzáfér a taszk összes erőforrásához. Két taszknak alapértelmezés szerint nincsenek közös erőforrásaik (bár lehetnek közös erőforrásaik, ehhez azonban a taszkoknak gondoskodniuk kell arról, hogy a közös hozzáférésű erőforrások mindkettőjükben elérhetők legyenek – ehhez a Mach kernel biztosít nem túl bonyolultan használható mechanizmusokat).

Egy thread kis ráfordítással létrehozható, működéséhez kevés erőforrásra van szüksége (ez azért van így, mert a threadhez tartozó belső állapot minimális – altalában a processzor regiszterkészlete – és az általa elért erőforrások kezelését az őt tartalmazó taszk végzi). Egy multiprocesszoros rendszerben egy taszknak egyidejűleg több threadje is végrehajtódhat.

A Mach memóriakezelése: A Mach kernel biztosítja a nagy, nem feltétlenül összefüggő memória kezeléséhez szükséges mechanizmusokat. Minden taszk rendelkezik egy kernel által manipulált címtérképpel, amely vezérli a virtuális memóriacímek fizikai memóriacímekre történő leképezését. Mint az a virtuális memóriát biztosító rendszerek nagy részére jellemző, a taszkok virtuális címtartományának mindig van egy-egy része, amely nincs benn a fizikai memóriában egy-egy adott időpillanatban, ezért kell valamilyen mechanizmus a fizikai memóriának a taszkok virtuális címtartományának a cache-elésére való használatára (és valószínűleg ez a Mach-ban megjelenő egyik legnagyobb ötlet). Nem úgy, mint más virtuális memóriát kezelő rendszerek, a Mach kernel nem implementálja ennek a cache-elésnek az összes részletét, ehelyett lehetőséget nyújt felhasználó által készített taszkoknak arra, hogy e cache-elés egyes részleteit megvalósítsák.

[1] Kivételes események kezeléséért felelős eljárások.

Egy taszk allokálhat új memóriaterületeket, deallokálhat memóriaterületeket és módosíthatja a rájuk vonatkozó védelmi információkat. Ezenkívül egy taszk meghatározhatja az egyes memóriarészeinek az **öröklési jellemzőit** is. Egy új taszk létrehozásakor mindig ki kell jelölni egy már meglevő taszkot, amelyet az új taszk memóriaszerkezetének kialakításakor az operációs rendszer mintának fog tekinteni. Ilyenkor a meglevő (mintaként használandó) taszkban levő egyes memóriarészek öröklési jellemzői határozzák meg azt, hogy az újonnan létrehozott taszkban a megfelelő memóriarész definiált legyen-e, illetve amennyiben azt a részt az új taszk örökli, akkor a szülőtaszkkal egy közös, megosztott memóriaterületként használja azt, vagy pedig saját példányt kell, hogy kapjon.

A Mach rendszerben a legtöbb memóriamásolási művelet copy-on-write módszerrel van optimalizálva: a másolandó memóriarészek tartalma nem lesz egyből lemásolva, hanem másolás után a haználók a két példányt "írásvédetten" ugyan, de közösen használják tovább. Ha bármelyik használó megpróbálja a közösen használt "írásvédett" memóriarész tartalmát módosítani, akkor a megfelelő rész tényleges lemásolására a módosítás pillanatában kerül sor. Ez a késleltetett memóriamásolás egy fontos hatékonyságot növelő optimalizáció a Mach kernelben.

Bármely memóriarész mögött van egy-egy **memória objektum**. Egy **memória-kezelő** taszk biztosítja a (fizikai) memóriában levő lapok tartalma és az ugyanehhez az információtartalomhoz tartozó, nem a fizikai memóriában tárolt információ (egy absztrakt memória objektum) közti kapcsolatot, konverziót. A Mach kernel tartalmaz egy alapértelmezésként használható memória-kezelőt, amely olyan memória objektumok létrehozását biztosítja, amelyek kezdetben nulla kódú értékekkel vannak feltöltve, és a rendszer lapozási területére lesznek szükség esetén kilapozva.

Taszkok közti kommunikáció: A taszkok közötti kommunikáció a Mach eszköztárának egy nagyon fontos elemét alkotja. A Mach egy kliens/szerver alapú rendszerstruktúrát feltételez, ahol egyes taszkok (kliensekként) más taszkok (szerverek) szolgáltatásait érhetik el a az őket összekötő kommunikációs csatornán keresztül küldött üzenetek segítségével. Mivel a Mach kernel önmagában kevés szolgáltatást nyújt (például a fájlkezelési szolgáltatás sem része), ezért egy "átlagos" Mach taszk sok más taszkkal kell, hogy kommunikáljon annak érdekében, hogy hozzáférjen a számára szükséges szolgáltatásokhoz. A taszkok közötti kommunikáció kommunikációs csatornáját **port**nak nevezik. Egy port egy egyirányú csatorna, amelyen lehet korátos mennyiségű **üzenet**. Egy üzenet tartalmazhat adatokat, memóriarészeket és portok hozzáférési jogait. Egy **port hozzáférési joga** egy név, amellyel az adott taszk a porthoz való hozzáférési jogosultságát azonosítja (a Mach kernellel szemben). Egy taszk csak akkor férhet hozzá egy porthoz, ha a portra vonatkozóan a megfelelő hozzáférési jogokkal rendelkezik. Egy adott portra vonatkozóan csak egyetlen taszknak lehet **olvasási joga**. Ez az egy taszk jogosult a portra küldött üzenetek feldolgozására (beolvasására). Egyidejűleg egy adott portra vonatkozóan több taszknak is lehet adatküldési joga, amivel lehetőségük van az adott portra üzenetek küldésére. Két taszk kommunikációjakor a küldő az elküldendő adatokból felépít egy üzenetet, és egy üzenetküldési műveletet hajt végre egy olyan porton, amelyre adatküldési joggal rendelkezik. Később az a taszk, amelynek erre a portra vonatkozóan olvasási joga van, végrehajt egy üzenetolvasási műveletet. Megjegyezzük, hogy ez az üzenetátvitel egy aszinkron művelet. Az üzenet átmásolása általában a korábban is említett copy-on-write optimalizációs technikával történik.

UNIX implementációja a Machon

A UNIX Mach rendszeren való implementálása mögötti alapötletet már láttuk: szét kell szedni a hagyományos UNIX-ot egy gpefüggetlen és egy gépfüggő részre, és ezután a gépfüggetlen (kisebb) rész átvitele után az operációs rendszer további komponenseinek a lefordítása már sokkal egyszerűbb.

Az utóbbi években már nemcsak a Carnegie Mellon-on dolgoznak ezen a projekten, hanem a világon sokfelé. Egyre több operációs rendszert módosítanak úgy, hogy a gépfüggő részeiket Mach szolgáltatások végrehajtására cserélik, így azoknak a hordozhatósága is megnő. Jelenleg már a Linux operációs rendszernek is készül a Machfeletti változata.

A Mach-alapú operációs rendszerek szerkezetük alapján két csoportba sorolhatók: az **egyszerveres** és a **többszerveres** implementációk csoportjára. Az egyszerveres implementációk (ilyen lesz például a Mach-alapú Linux, és ilyen a már elkészült Mach-alapú 4.3BSD) a UNIX "gépfüggetlen" funkcionalitását egyetlen programban "monolitikusan" implementálja. Ezzel szemben a többszerveres implementációk (ilyen lesz például a GNU Hurd operációs rendszere) több program (ún. szolgáltató program, szerver) segítségével valósítják meg a UNIX funkcionalitást: itt például lehet, hogy külön program foglalkozik a fájlrendszerrel, külön program a memóriakezeléssel, külön program végzi a felhasználók igazolását, a perifériakezelést, stb.

2.8 Kérdések, feladatok

1. Mi alapján különbözteti meg a UNIX az abszolút és a relatív pathname-eket.
2. Ismertesse a UNIX fájlrendszerének a belső szerkezetét!
3. Milyen esetben mi a ésszerűbb az 512 byteos diszk-blokk választás, mint az 1024 byteos blokkok? Miért?
4. Milyen információk kerülhetnek bele a UNIX rendszer processz-táblájába?
5. Hogyan valósítható meg az ütemező az Intel 8088 mikroprocesszoron? s az Intel 80386-on?
6. Mit jelentenek a következő fogalmak?
   - időszelet
   - rwx-bitek
   - szuperfelhasználó
   - speciális fájl
   - fájl-attribútum
   - i-node
   - i-node-ra mutató link
   - setuid bit
   - single indirect
   - major device number

- minor device number

7. Pista (uid=234; gid=17) írt egy levelet, és a következő értékekre állította a levelet tartalmazó fájl védelmi bitjeit: rwxr—. (A - jel azt jelenti, hogy az ott levő jog nincs megadva.) El tudja-e ezt a levelet olvasni Mariska (uid=252; gid=17)? s Zsolt (uid=756; gid=50) el tudja olvasni?

8. Mi a pipe?

9. Sok programozó osztott memória használatával szimulálja az üzenetátadást, mivel az gyorsabb megoldásnak tűnik. Tényleg gyorsabb? Vagy nem feltétlenül?

10. A hierarchikus directoryszerkezetet ábrázolni lehet egy a gyökér-directoryból kiinduló fastrukturával. Mi a helyzet, ha bejönnek a UNIX-ban megismert LINK-ek? Ekkor milyen gráffal lehetne hasonló módon jellemezni a fájlrendszert?

Fejezet 3

Rendszerhívások

Az operációs rendszerek a felhasználói programok elől eltakarják a géphez kapcsolt "nyers" hardver részleteket, és a programozónak így kényelmes programfejlesztési környezetet biztosítának. Sőt a legtöbb komoly rendszerben az operációs rendszer nem is engedi meg, hogy az "átlagos" felhasználók beleturkáljanak a rendszer lelkivilágába (például a UNIX rendszerben sincs mindenkinek megengedve az, hogy a floppy-diszkhez tartozó speciális fájlt megnyissa és használja). Ehelyett az operációs rendszer ún. **szolgáltatás**okat nyújt a programoknak. Ilyen szolgáltatás például egy adott nevű fájl megnyitása, egy adott fájlból karakterek olvasása, illetve fájlba karakterek írása, stb ...

Ezeket a szolgáltatásokat a programok (a szolgáltatások (ki)használói) ún. **rendszerhívás**okon keresztül érhetik el. Ezt úgy kell érteni, hogy minden felhasználói program a processzor felhasználói üzemmódjában fut, számol, ciklust szervez, stb ... Ha valami olyan műveletet akar végrehajtani, amely esetleg más futó programokat megzavarna (például ilyenek a fájlműveletek), akkor meg kell kérnie az operációs rendszert, hogy hajtsa végre neki a kért műveletet. Az operációs rendszer ellenőrizheti, hogy a felhasználónak van-e joga ahhoz, amit kért, és ha nincs, akkor a rendszerhívást visszautasíthatja. Ha pedig a felhasználónak jogosultsága van az adott művelet végrehajtására (például egy winchester formattálására), akkor az operációs rendszer a kért szolgáltatást elvégzi. Ilyen módon tudja az operációs rendszer az első fejezetben említett feladatát, a hardver erőforrások védelmét és elosztását helyesen ellátni.

A következőkben egy működő operációs rendszer (a UNIX) rendszerhívásain keresztül lesz bemutatva az, hogy milyen eszközök állnak a programozó rendelkezésére. A UNIX rendszerhívásainak csak egy nagyon szűk részhalmaza lesz bemutatva - főleg azok, amelyek "közösek" minden UNIX-ban (a 7-es változattól kezdve). A rendszerhívások itt is a már korábban említett szempontok szerint lesznek csoportosítva.

A rendszerhívásokról még annyit érdemes megjegyezni, hogy egy egész típusú érték a visszatérési értékük, és a negatív visszatérési érték legtöbbször a rendszerhívás végrehajtása alatt fellépő hibát jelzi, vagy azt, hogy a rendszerhívás hiba nélkül lefutott. Hiba esetén az `errno` egész típusú globális változó tárolja a hiba okát: minden egyes hibafajtát egy-egy egész szám azonosít, és az `errno` változóban az éppen végrehajtott rendszerhívás sikertelenségét okozó hiba kódja van. Ezt a hibakódot kiírni (a hozzá tartozó szokásos angol nyelvű szövegekkel) a `perror` könyvtári rutinnal lehet (ez nem rendszerhívás!). (Az `errno`-beli hibakódok egy `errno.h` include fájlban vannak "olvasható" formában: ugyanis ott vannak C nyelvű makródefiníciókkal a hibakódoknak megfelelő "standard" konstansok megadva. Ez egyébként is jellemző a UNIX-ra és más

rendszerekre is, hogy bizonyos konstansoknak szimbolikus megfelelőit belerakják a C könyvtárba vagy header-fájlokba. Ez azért jó, mert a programkód ezek használatával olvashatóbb lesz, és ha esetleg a konstanst megváltoztatják (vagyis azt, amit az operációs rendszer vár), akkor minden programban megkeresni és átírni ... az nagyon nagy munka lenne). Ilyen konstansokat (ún. manifest-konstansokat) ebben a leírásban is külön említés nélkül fogok használni. Az egyes rendszerhívásokhoz tartozó referencia kézikönyv lapok (amit a UNIX `man` parancs kiír) tartalmazzák azokat a header-fájlokat, amelyek az ott használható manifest-konstansokat (... makrókat) definiálják.

3.1 Folyamatokat kezelő rendszerhívások

A folyamatokat kezelő rendszerhívások a következők: `fork()`, `exec()`, `exit()`, `wait()`, `getpid()`, `getppid()`, `setpgrp()` és `nice()`. A `fork()` rendszerhívással lehet egy új folyamatot létrehozni. A létrehozott gyermek-folyamat a szülő-folyamat pontos másolata lesz, vagyis a gyermek-folyamat a szülő-folyamat majdnem minden jellemzőjét örökli (a pid-et például nem!). A gyermek- és a szülő-folyamat egymással párhuzamosan fognak futni. A `fork()` rendszerhívás C nyelvben egy `int` típusú értékkel tér vissza, és ez az, ami alapján meg lehet különböztetni, hogy melyik a szülő- ill. melyik a gyermek-folyamat. A gyermek-folyamatban a visszatérési érték: 0, míg a szülő-folyamatban a visszatérési érték egyenlő a gyermek-folyamat pid-jével. Negatív visszatérési érték a rendszerhívás sikertelenségét jelzi (a hiba oka lehet például az, hogy nem volt elég memória a gyermek-folyamat létrehozásához). A `fork()` a következő formában fordul elő a leggyakrabban:

```
valtozo=fork();          /* Megszuljuk a gyermeket ... */
if (valtozo < 0) {
     /* Hibauzenet kiirasa, a sikertelen rendszerhivasrol! */
} else {
     if (valtozo == 0) {
          ...                 /* A gyermek ezt csinalja  ... */
     } else {
          ...                 /* A szulo pedig ezt ... */
     }
}
```

Mi az, amit a gyermek-folyamat fork után a szülőtől örököl?

- A folyamatot futtató felhasználóra vonatkozó információkat (a futtató felhasználó azonosítóját, a futtató felhasználó csoportjának az azonosítóját)
- Effektiv user id-et (ha a program setuid bites, akkor ez eltérhet a programot futtató felhasználó azonosítójától)
- Effektiv csoport azonosítót
- Folyamat-csoport azonosítóját
- Munkadirectory
- Signal-kezelő eljárások

- umask értéket (ld. később)

Mi az, ami a fork után eltér a szülő és a gyermek között?

- Folyamat-azonosító
- Szülő folyamat azonosítója
- A gyermek folyamatnak saját másolata van a szülő folyamat fájldeszkriptorjairól
- Ha a szülő valamikorra egy ALARM signalt kért, azt a gyermek nem fogja megkapni.

A leggyakrabban a gyermek-folyamatnak a szülő feladatától teljesen eltérő dolgot kell csinálnia, például egy másik fájlban tárolt programot kell végrehajtania. Erre való az **exec rendszerhívás**, amely a UNIX kernelnek talán a legbonyolultabb rendszerhívása. Az exec több különböző formában érhető el, itt az `execle()` hívás lesz bemutatva. Amikor egy C nyelvű programot az operációs rendszer shelljéből elindítunk, akkor a shell-parancsként beadott programnév után írt egyéb programparaméterek hogyan lesznek a programból elérhetők. A C program fő-eljárását a következő módon kell deklarálni:

```
main(argc, argv, envp)
int argc;
char **argv, **envp;
```

A program ezután a híváskor megadott paraméterek darabszámát az `argc` paraméteren keresztül tudja megkapni, maguk a paraméterek pedig az `argv` (karakteres tömb) változón keresztül érhetők el. Az `envp` karakter tömb a shell-változókat tartalmazza. Az mindig igaz, hogy `argc > 0`, és az `argv[0]` nem üres, mert a nulladik parancs-paraméter az maga a beadott parancsnév szokott lenni. Példa az `execle()` hívásra:

```
r=execle("/bin/ls","ls","-l","alma.c",(char *)0, envp);
```

Az első paraméter a végrehajtandó bináris program fájlnevét tartalmazza. A következő paraméter magának a parancsnak a nevét tartalmazza (ez nem kell, de így szokás), a harmadik és a negyedik paraméter a végrehajtandó parancs egyéb paramétereit tartalmazza. A `(char *)0` a programnak átadandó paraméter-felsorolás végét jelzi. Az utolsó (`envp`) paraméter az új programnak átadandó shell-változókra mutat.

Az exec() rendszerhívások végrehajtásukkor ellenőrzik, hogy a végrehajtandó fájl rwx-bitjei közül az x-bit be van-e állítva, az új folyamat befér-e még a memóriába. Ha a fenti feltételek nem teljesülnek, akkor az exec rendszerhívás sikertelen lesz (ezért kell figyelni az exec rendszerhívás visszatérési értékét). (Az exec rendszerhívás nem zárja le automatikusan a korábban használt fájlokat! A megnyitott fájlokat az újabb, exec-elt program tovább használhatja.)

Ha egy szülő-folyamat megszül egy gyermek-folyamatot, majd a gyermek elvégzi a feladatát, akkor a gyermeknek meg kell hívnia az `exit()` rendszerhívást. Az exit rendszerhívásnak egyetlen paramétere van, egy 0 és 255 közé eső egész szám, az ún. `exit-státusz`. A szülő folyamat lekérdezheti a gyermek-folyamat exit-státuszát, és ebből például arra következtethet, hogy a gyermek-folyamat milyen eredménnyel tudta

a feladatát ellátni. A szülő-folyamat a gyermek- folyamat befejeződésére a `wait()` rendszerhívás segítségével várakozhat. A wait() rendszerhívás egyetlen paramétere egy egész típusú változóra mutat, és a rendszerhívás végrehajtása után a befejeződött gyermek-folyamat exit-státusza kerül a megadott változó magasabb helyiértékű bytejába. Ha a gyermek-folyamat végrehajtotta az exit() rendszerhívást, a szülő pedig még nem adta ki a wait rendszerhívást, akkor a gyermek-folyamat általában ún. **zombie-processz** lesz, vagyis az általa lefoglalt memória felszabadul, de a processz-táblában még foglalja a helyet. (Ha egy szülő-folyamat nem hajt végre egyetlen wait()-et sem, akkor előbb- utóbb betelhet a processz-tábla, és a rendszer nem lesz képes újabb folyamatokat elindítani.)

A fenti rendszerhívásokat például a következőképpen használhatjuk:

```
main(argc, argv, envp)
  int argc;
  char **argv, **envp;
{
  int eredm;

  /*
     .
     .
     .
  */

  if (fork() == 0) {
       execle("/bin/ls","ls","-l",(char *)0,envp);
  } else {
       wait(&eredm);
  }
}
```

A fenti program elindít egy gyermek-folyamatot, amely végrehajtja a UNIX `ls` parancsát, és a szülő megvárja, amíg a gyermek-folyamat befejeződik. Lényegében a UNIX shelljei is így működnek (ezt később egy részletesebb példán is megnézhetjük).

A `getpid()` és a `getppid()` rendszerhívások közül az előbbi az őt végrehajtó folyamat pid-jét, az utóbbi pedig az őt végrehajtó folyamat szülő-folyamatának a pid-jét adja vissza (mindkét függvény egész típusú értéket ad vissza).

A `setpgrp()` rendszerhívás az őt végrehajtó folyamat folyamat-csoport azonosítóját a folyamat azonosítójára (pid-jére) állítja, és az új folyamat-csoport azonosítót adja vissza. A folyamat-csoport azonosító azért hasznos, mert a `kill()` rendszerhívással signal-t lehet küldeni egy folyamat-csoport minden egyes tagjának (ld. később).

A `nice()` rendszerhívással lehet egy folyamat ütemezési prioritását módosítani. Az alapértelmezés szerinti prioritási érték 20; ezt az értéket növelve csökken a folyamat prioritása. A `nice()` paraméterében megadott értéket az operációs rendszer hozzáadja a folyamat prioritásához. Negativ paramétert csak szuperfelhasználó jogú folyamatoktól fogad el az operációs rendszer.

3.2 A fájlrendszer rendszerhívásai

A fájlrendszer-kezelő rendszerhívások a következők: `access()`, `creat()`, `open()`, `close()`, `read()`, `write()`, `lseek()`, `stat()`, `fstat()`, `dup()`, `pipe()`, `link()`, `unlink()`, `mount()`, `umount()`, `sync()`, `chdir()`, `mkdir()` és az `rmdir()`. Fontos megemlíteni, hogy ezek a rendszerhívások nem azonosak a C szabványos I/O könyvtár `fopen()`, `fclose()`, ...függvényeivel. Ha lehet, akkor ne keverjük egy programon belül a fenti két függvénycsoportot, mert meglepetések érhetnek! (Ha lehet, akkor a szabványos I/O könyvtárat használjuk, mert a programunk hordozhatóbb lesz. Az `open()`, `close()`, ...rendszerhívások általában csak a UNIX rendszerekben vannak meg, míg az `fopen()`, `fclose()` ...rutinok gyakorlatilag minden C nyelvi környezetben elérhetők.)

3.2.1 Alapvető, fájlokkal kapcsolatos rendszerhívások

Az `access()` rendszerhívással lehet lekérdezni az operációs rendszertől azt, hogy egy adott fájlon egy adott műveletet elvégezhetünk-e. Lekérdezhetjük, hogy egy adott nevű fájl létezik-e, olvashatjuk-e, írhatjuk-e illetve végrehajthatjuk-e azt. Az ellenőrzés **nem az effektiv user- ill. group-id alapján** történik.

A `creat()` és az `open()` rendszerhívás foglalkozik a fájlok megnyitásával. A `creat()` létrehoz egy, az első paraméterében megadott nevű fájlt, ha eddig az adott nevű fájl nem létezett; ha pedig az első paraméterében megadott fájl már létezik, akkor annak a hosszát 0 bytera állítja. A `creat()` rendszerhívás visszatérési értéke egy egész típusú érték, egy ún. **fájldeszkriptor**. Ezzel lehet később a fájlműveleteknél erre a fájlra hivatkozni. A `creat()` ezenkívül beállítja a második paraméterében megadott érték szerint az új fájl védelmi bitjeit.

Az `open()` rendszerhívás egy már meglevő fájlt nyit meg írásra vagy olvasásra. A megnyitandó fájl neve az `open()` első paramétere, a második paraméter értéke általában 0, ha a fájlt olvasásra nyitjuk meg; 1, ha a fájlt írni akarjuk (ezeknek megfelelő konstansok: az `O_RDONLY`, `O_WRONLY` illetve `O_RDWR`, amelyek az `<fcntl.h>` nevű header-fájlban vannak deklarálva). Ennek a visszatérési értéke szintén egy fájldeszkriptor.

A `close()` rendszerhívás a paraméterében megadott fájldeszkriptorhoz tartozó fájlt lezárja. Itt érdemes megjegyezni azt, hogy minden egyes folyamat fájldeszkriptorjai 0-tól kezdődően vannak sorszámozva, és ha egy új fájlt megnyitunk, akkor mindig a legkisebb "értékű" fájldeszkriptor lesz a fájlhoz rendelve. (Ez azt jelenti, hogy ha pl. a 0-ás fájldeszkriptort lezárjuk, akkor a következő `open()` rendszerhívás a megnyitott fájlt a 0-ás fájldeszkriptorhoz fogja kötni. Minden program elindulásakor három "szabványosan" megnyitott fájlon dolgozhat: a 0-ás deszkriptorú **szabványos bemeneten**, az 1-es deszkriptorú **szabványos kimeneten** és a 2-es deszkriptorú **szabványos hibacsatornán**. Alaphelyzetben ezek a terminálhoz (billentyűzethez ill. képernyőhöz) vannak rendelve, de a shell-ben ezek átirányíthatóak tetszőleges fájlba.)

A következő lényeges rendszerhívások: a `read()` és a `write()` rendszerhívások. Ezekkel lehet egy már megnyitott fájlon valamilyen műveletet végezni: a fájlból adatokat olvasni, és a fájlba adatokat írni. Mindkét rendszerhívás első paramétere annak a fájlnak a fájldeszkriptorja, amelyen az adott fájlműveletet el akarjuk végezni. A második paraméter tartalmazza annak az adatterületnek a címét, ahonnan az adatokat a fájlba akarjuk írni (ill. read rendszerhívásnál: ahová a fájlból olvasni akarunk). Az utolsó paraméter a fenti adatterület hosszát tartalmazza, vagyis a maximálisan be-

olvasandó (ill. kiírandó) byteok számát. Ezeknek a rendszerhívásoknak egész típusú a visszatérési értéke, és a visszatérési érték megadja a rendszerhívás során ténylegesen kiírt ill. beolvasott byteok számát. (Például ha a `read()` rendszerhívás fájlvégéhez ér, és nincs a fájlban már annyi karakter, amennyit a rendszerhívás harmadik paraméterében megadtunk, akkor csak annyi karaktert fog beolvasni, amennyi a fájlban van, és ennek megfelelően a visszatérési értéke kisebb lesz a harmadik paraméter értékénél).

A fenti rendszerhívások használatára láthatunk példát a következő programrészletben:

```
fggv()
{
  int fd1,fd2;
  char buf[512];
  int rc;

  fd1=open("/usr/csb/filenev",0); /* 0 = olvasasra nyitom meg */
  fd2=creat("/usr/csb/ujfile",0644); /*  rw-r--r-- biteket \'all\'\i t */
  /*
    Az fd1 filebol atmasoljuk az fd2 fileba maximum az elso 512
    darab karaktert.
  */
  rc=read(fd1,buf,512);
  if (rc>0) write(fd2,buf,rc);
  /*
       ...
  */
  close(fd2);
  close(fd1);
}
```

A `lseek()` rendszerhívással lehet egy fájlban "pozícionálni" (ahol a pozícionálás értelmezett) úgy, hogy pl. a következő `read()` onnan kezdi el olvasni a fájlt, ahová pozícionáltunk. A lseek rendszerhívásnak három paramétere van: az első és a harmadik egész, a második pedig (C-ben) long típusú (ezt a programban a konstansok megadásakor ne feledjük el!). Az első paraméter annak a fájlnak a deszkriptorát tartalmazza, amelyben pozícionálni akarunk. A második paraméter tartalmaz egy "pozíciót" (pl. lehet, hogy azt tartalmazza, hogy a fájl hányadik karakterére akarunk pozícionálni). Ha a harmadik paraméter `SEEK_SET`, akkor a fájlban arra a karakterre kell pozícinálni, amelynek a pozícióját a második paraméter tartalmazza. Ha a harmadik paraméter `SEEK_CUR`, akkor a fájlban beállítandó pozíció egyenlő az aktuális pozíciónak és a második paraméternek az összegével. Ha pedig a harmadik paraméter értéke `SEEK_END`, akkor az új fájlpozíció egyenlő lesz a fájl hosszának és a második paraméternek az összegével. A rendszerhívás visszatérési értéke long típusú, és az új aktuális fájlpozíciót tartalmazza. Megjegyezzük, hogy a `SEEK_SET`, `SEEK_CUR`, `SEEK_END` makrók (konstansok) az `<unistd.h>` header-fájlban vannak definiálva.

3.2.2 A fájlrendszer és a memóriakezelő kapcsolata

A UNIX operációs rendszer újabb változatai lehetőséget adnak a fájlok memórián keresztül történő elérésére **a fájl**nak a **memóriába ágyazásával** (ezzel lehetőség nyílik a fájl tartalmának memóriaműveletek segítségével történő módosítására, amely gyakran hatékonyabb a hagyományos `read()` illetve `write()` rendszerhívásoknál.) Erre az `mmap()` UNIX rendszerhívást használhatjuk, melynek prototípusa a következő:

```
caddr_t mmap(caddr_t addr, size_t len, int prot, int flags,
             int fd, off_t offset);
```

Az `mmap()` rendszerhívás első paramétere `caddr_t` típusú (az ilyen típusú változók egy tetszőleges memóriacímet kaphatnak értékül, általában `char *` típussal implementálják). Az első argumentumban azt a memóriacímet kell megadnunk, ahova be akarjuk ágyazni a kérdéses fájlt (vagy annak egy részét). A programok hordozhatósága érdekében itt 0-t adjunk meg, ugyanis ekkor az operációs rendszer maga választ egy szabad memóriaterületet, ahova a fájlt beágyazza. Sikeres végrehajtás esetén az `mmap()` rendszerhívás az `fd` argumentumban kijelölt fájldeszkriptorú fájl `offset` argumentumában adott pozíciótól kezdődő `len` argumentumban bájtokban megadott hosszú részét beágyazza, és a rendszerhívás visszatérési értéke a beágyazott fájl-rész memóriabeli kezdőcíme (a beágyazás kezdőcíme).

A `prot` paramétert a `PROT_READ`, `PROT_WRITE`, `PROT_EXEC` szimbólikus konstansok bitenkénti VAGY művelettel képzett kapcsolatával állíthatjuk elő aszerint, hogy a beágyazott fájl-részen milyen műveleteket akarunk megengedni (olvasni,írni vagy végrehajtani akarjuk a részeit). Ha egy fájlra nincs írási engedélyünk a hozzá tárolt rwx-bitek alapján, akkor a memóriába ágyazással sem módosíthatjuk azt.

A `flags` argumentum értéke vagy `MAP_PRIVATE` vagy pedig `MAP_SHARED` lehet (általában). Ha itt `MAP_PRIVATE`-et adunk meg, akkor a fájlon végzett módosítások időlegesek, azaz az eredeti fájlon nem jelennek meg, a fájl többi felhasználója nem látja azokat.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <fcntl.h>

main()
{
  int fd;
  char *ra;

  fd=creat("cmmap1t",0666);
  write(fd,"alma",4);
  close(fd);

  fd=open("cmmap1t",O_RDWR);
  ra=mmap(0,4,PROT_WRITE|PROT_READ,MAP_SHARED,fd,0);
  if (ra == (caddr_t)(-1)) perror("mmap sikertelen");
  *(ra+2)='f';
```

```
  munmap(ra,4);
  close(fd);
}
```

3.3 Egyéb, fájlokkal kapcsolatos rendszerhívások

A `stat()` és `fstat()` rendszerhívással fájloknak különféle jellemzőit lehet lekérdezni. Mindkét rendszerhívás első paramétere azt tartalmazza, hogy melyik fájlról akarunk bővebb információt: az `fstat()` első paramétere egy megnyitott fájlra vonatkozó fájldeszkriptor, míg a `stat()` rendszerhívás első paramétere egy string, amely egy abszolút vagy relatív fájlnevet tartalmaz. Mindkét rendszerhívás második paramétere egy `stat` típusú struktúrára mutató pointer. Ebben a strukturában adja vissza az operációs rendszer a fájlról a következő információkat:

```
struct stat {
  short int st_dev;       /* Melyik periferian van a filet tartalmazo
                             directorybejegyzes */
  unsigned short st_ino;  /* Hanyas a file inode-janak a sorszama*/
  unsigned short st_mode; /* Egyeb dolgok, pl. rwx-bitek */
  short int st_nlink;     /* Hany link kapcsolodik a filehoz */
  short int st_uid;       /* A file tulajdonosanak uid-je */
  short int st_gid;       /* A file tulajdonosanak gid-je */
  short int st_rdev;      /* A blokk/karakter-specialis fileoknal ez
                             tartalmazza a hozza tartozo I/O egyseg
                             belso sorszamat */
  long st_size;           /* A file meretet tartalmazza */
  long st_atime;          /* A file legutolso hasznalatanak a
                             datumat tartalmazza */
  long st_mtime;          /* Az utolso modositas datuma */
  long st_ctime;          /* A file letrehozasanak a datuma */
};
```

A következő példa bemutatja a `stat()` használatát:

```
/*
 * Megallapitja az egyetlen parametereben atadott file tipusat.
 */

#include <sys/types.h>
#include <sys/stat.h>

main(argc,argv)
  int argc;
  char **argv;
{
  struct stat tmpstat;
```

```
  if (argc != 2) {
    printf("Usage: %s filename\n",argv[0]);
    exit(-1);
  }
  if (stat(argv[1],&tmpstat) < 0) {
    perror("stat");
    exit(-1);
  }
  printf("%s: ",argv[1]); /* Filenevet kiirom */
  switch (tmpstat.st_mode & S_IFMT) {
    case S_IFDIR: printf("directory");
                  break;
    case S_IFCHR: printf("karakter-specialis file");
                  break;
    case S_IFBLK: printf("blokk-specialis file");
                  break;
    case S_IFREG: printf("kozonseges file");
                  break;
    default:      printf(" ?");
                  break;
  }
  printf("\n");
}
```

A `dup()` rendszerhívás egyetlen paramétere egy megnyitott fájlra mutató fájldeszkriptor. Visszatérési értéke egy másik fájldeszkriptor lesz, amely ugyanarra a fájlra vonatkozik, mint amit a paraméterében megadtak (bármelyik fájldeszkriptoron is olvasunk, a fájlból egy karaktert csak egyszer fogunk beolvasni, mivel a fájlbeli pozíció a két deszkriptornál mindig meg fog egyezni).

A `pipe()` rendszerhívás segítségével a már korábban is megemlített pipe-okat tudjuk létrehozni. Ennek a paramétere egy olyan két-elemű tömb, amelynek mindkét eleme egész típusú, és a rendszerhívás végrehajtása után egy-egy fájldeszkriptort tartalmaz. Amit az 1-es indexű tömbelemben levő fájldeszkriptorú fileba írunk, azt a 0-ás indexű tömbelemben levő fájldeszkriptorú fájlból olvashatjuk ki. Ennek a használatára példa a következő programrészlet:

```
#include <stdio.h>

main()
{
  int pfd[2]; /* A pipe filedeszkriptorjai ebbe a tombbe kerulnek*/
  int szam;

  /*
    .
    .
  */
  pipe(pfd);
```

```
    if (fork() == 0) {
         close(pfd[0]);
         close(1);
         dup(pfd[1]);  /* standard output:=pipe file */
         close(pfd[1]);
         printf("%d",23); /* Elkuldjuk a masik folyamatnak */
    } else {
         close(pfd[1]);
         close(0);
         dup(pfd[0]);  /* Uj standard input: masik folyamattol */
         close(pfd[0]);
         scanf("%d",&szam); /* Fogadjuk a masik folyamat outputjat */
    }
    /*
      .
      .
    */
}
```

A `link()` rendszerhívással lehet egy fájlra vonatkozó linket létrehozni, az `unlink()` rendszerhívással pedig egy link megszüntethető (vagyis a felhasználó ezt úgy látja, hogy a fájl a directoryjából törlődik). A link-nek két string típusú paramétere van, az első annak a létező fájlnak neve, amelyre a létrehozandó link mutasson; a második pedig az újonnan létrehozandó fájl nevét tartalmazza. Ha egy más felhasználó fájljait (egyenként ...) meglinkeljük megunknak, akkor a fájlhoz hozzáférhetünk, de a védelmi bitjeit nem változtathatjuk meg! Az `unlink()` egyetlen paramétere a törlendő fájl neve.

A `mount()` rendszerhívással lehet egy fájlrendszert "beilleszteni" valamelyik directory alá. Legyen például egy 286-os AT-n az első lemezegységnek a UNIX alatti neve: `/dev/fd0`. Ha a benne levő lemez egy érvényes directory-struktúrát (fájlrendszert) tartalmaz, és azt valamelyik winchesteren levő directory alá be akarjuk rakni, akkor egy ilyesmi rendszerhívást kell kiadni:

```
mount("/dev/fd0","/usr/csb/aldir",0);
```

A mount rendszerhívás által beillesztett fájlrendszert az `umount()` rendszerhívással lehet a rendszerből "kiilleszteni". Ennek alkalmazására tekintsük a következő programrészletet:

```
umount("/dev/fd0");
```

(A `mount()` és az `umount()` rendszerhívásokat csak a rendszergazda adhatja ki!)

A `sync()` rendszerhívással lehet a cache-memóriában tárolt "aktualizált" lemezblokkokat a lemezre fizikailag kiiratni (a rendszerhívásnak nincsenek paraméterei).

A `chdir()` rendszerhívással lehet az aktuális directoryt (munkadirectoryt) beállítani. Egyetlen paramétere az új munkadirectory nevét tartalmazza.

Az `mkdir()` ill. `rmdir()` rendszerhívásokkal lehet egy új directoryt létrehozni, ill. egy meglevő directoryt törölni. Mindkét rendszerhívás egyetlen paramétere a létrehozandó (ill. törlendő) directorynak a neve.

3.4 Fájlok konkurrens elérése

Ebben a részben arról lesz szó, hogy a UNIX milyen szolgáltatásokat nyújt párhuzamos folyamatok fájlhozzáférésének a szinkronizálására. A UNIX a fájlok konkurrens kezelésekor kialakult problémás helyzeteket a korábban már ismertetett rekord-lefoglalás lehetőségét biztosítja. A rekord-lefoglalás az `fcntl()` rendszerhívás segítségével implementálható a következőkben leírtak szerint (az `fcntl()` használatához szükség van az `<fcntl.h>` fájl include-olására is).

A UNIX rekord-lefoglalása kétféle lehet: **írási** vagy **olvasási** célú. A kettő közötti lényeges különbség az, hogy egy rekordterületet (azaz a fájl egy bizonyos intervallumát) írás céljából egyszerre legfeljebb csak egy folyamat foglalhatja le, és csak akkor, ha azt a részt senki más nem foglalta még le sem írási sem pedig olvasási szándékkal); továbbá meg kell említeni, hogy olvasási célú rekordterület lefoglalást egyidejűleg több folyamat is kérheti akár ugyanarra a rekord-területre is. Fontos tudni, hogy a párhuzamos folyamatok kölcsönös kizárása az `fcntl()` rendszerhívásnál valósul meg, és nem a fájlműveleteknél: vagyis ha valamelyik folyamat nem tartja be a rekord-lefoglalási szabályokat, akkor az operációs rendszer megengedi neki, hogy kedve szerinti módosításokat illetve olvasásokat végezzen a fájlon, de az eredmény korrektségét ilyenkor nem lehet garantálni.

Az `fcntl()` rendszerhívás a következőképpen használható:

```
struct flock lock_data;
int fd, cmd, rc;

...

rc=fcntl(fd,cmd,&lock_data);
```

A rendszerhívás hatására az `fd` fájldeszkriptorral azonosított fájl `lock_data` argumentumban megadott része a `cmd` argumentumban megadott módon lefoglalásra kerül. A rendszerhívás írási célú rekord-lefoglalás esetén sikertelen lesz, ha a fájlra nincs írási engedélyünk.

A rekordlefoglalás módja (a `cmd` argumentum értéke alapján) háromféle lehet:

- `F_GETLK` : ennek hatására a `lock_data` argumentumban kijelölt részre vonatkozó rekord-lefoglalási információkat kapphatjuk vissza (az operációs rendszer ekkor kitölti az aktuálisan érvényes rekord-lefoglalási információi alapján a `lock_data` struktúra megfelelő mezőit – ld. később).

- `F_SETLK` : ennek hatására az operációs rendszer módosítja a fájlra vonatkozóan tárolt rekord-lefoglalási információit, vagyis ezzel lehet egy-egy rekordterületet lefoglalni vagy egy korábban kért lefoglalási kérelmet hatástalalnítani. A `lock_data` struktúra `ltype` komponensének értéke alapján a következő rekordlefoglalást végezhetjük el:

  - `F_RDLCK` : ha ezt az értéket állítjuk be, akkor olvasási céllal foglalhatjuk le a többi strukturakomponensben specifikált fájl-rekordot.
  - `F_WRLCK` : ha ezt az értéket állítjuk be, akkor írási céllal foglalhatjuk le a többi strukturakomponensben specifikált fájl-rekordot.

  - `F_UNLCK` : ha ezt az értéket állítjuk be, akkor a többi strukturakomponensben specifikált fájl-rekordra vonatkozóan a korábbiakban kiadott rekord-lefoglalási kérelmet vonhatjuk vissza, érvényteleníthetjük.

  Megjegyezzük, hogy ha a rekordlefoglalási igény a rendszerhívás végrehajtásának pillanatában nem kielégíthető, akkor a rendszerhívás sikertelen visszatérési értékkel tér vissza a rekordlefoglalási igény kielégítése nélkül.

- `F_SETLKW` : ez ugyanazt teszi, mint az `F_SETLK`, de ez a folyamatot megvárakoztatja olyan esetben, amikor a rekord-lefoglalási igénye nem kielégíthető. A folyamat egészen addig vár, amíg a rekordlefoglalási igényei ki nem elégíthetők, majd a rendszerhívás visszatérésekor a megfelelő rekordlefoglalási igényeket az operációs rendszer érvényesítette.

Most áttekintjük a `lock_data` argumentum egyes komponenseinek a szerepét (vagyis azt, hogy az `flock` struktúra egyes komponensei mit hogyan írhatnak le):

```
struct flock {
       short l_type;
       short l_whence;
       off_t l_start;
       off_t l_len;
       pid_t l_pid;
};
```

- `l_type` : értéke a korábbiakban írtak alapján `F_RDLCK` vagy `F_WRLCK` vagy pedig `F_UNLCK` lehet.

- `l_whence` : értéke `SEEK_SET` vagy `SEEK_CUR` vagy `SEEK_END` lehet aszerint, hogy az `l_start` argumentum kezdőpozíciójaként a fájl elejét vagy az aktuális fájlpozíciót vagy pedig a fájl végét kell-e tekinteni.

- `l_start` : értéke a lefoglalandó rekordterület kezdőpozícióját adja meg. Ez egy byte-ban mért offszet az `l_whence`-hez mind kezdőpozícióhoz képest.

- `l_len` : itt kell megadni a lefoglalandó rekordterület hosszát byteokban mérve. Ha itt 0 értéket adunk meg, akkor a fájl végéig az egész fájl tartalmát foglaljuk le, akár a később hozzáfűzött adatokkal együtt is.

- `l_pid` : az `F_GETLK` itt adja vissza a többi komponensben specifikált rekordterületet lefoglaltan tartó folyamat azonosítóját.

A rekordlefoglalás használatára lássuk a következő példát:

```
#include <fcntl.h>
#include <stdio.h>

main(int argc, char **argv)
{
  struct flock lock_data;
```

```
  int fd, cmd, rc, rf;

  fd=open("probafile",O_RDWR);
  rf=fork();
  lock_data.l_whence=SEEK_SET;
  lock_data.l_start=0;
  lock_data.l_len=10;
  if (rf==0) {
    lock_data.l_type=F_RDLCK;
    rc=fcntl(fd,F_SETLK,&lock_data); /* gyermek */
    fprintf(stderr,"Gyermek  tovabbfutott!\n");
    sleep(5);
    lock_data.l_type=F_UNLCK;
    rc=fcntl(fd,F_SETLK,&lock_data); /* gyermek */
  } else {
    sleep(1);
    fprintf(stderr,"Szulo: elkezd futni a lock elott ...\n");
    lock_data.l_type=F_WRLCK;
    rc=(-1);
    while (rc==(-1)) {
      rc=fcntl(fd,F_SETLK,&lock_data); /* szulo */
      if (rc==(-1)) fprintf(stderr,"Szulo:  SETLK sikertelen ... ujraprobalom\n");
      sleep(1);
    }
    fprintf(stderr,"Szulo lockolt es tovabbfutott!\n");
  }
  close(fd);
}
```

A program elindulása előtt hozzunk létre egy `probafile` nevű fájlt, amire van írási jogunk (én ezt úgy tettem, hogy a gépem jelszófájlját átmásoltam erre a névre abba a directoryba, ahol a programot futtatni akartam).

A program elindulása után kettéágazik: szül egy gyermekfolyamatot. A gyermek lefoglalja olvasási céllal a `probafile` fájl első 10 karakterét, és vár 5 másodpercig, mialatt a lefoglalást fenntartja, majd 5 másodperc után a lefoglalást megszűnteti, majd lezárja a fájlt. A szülő vár 1 másodpercet, majd ciklusban másodpercenként egyszer megpróbálja lefoglalni a fájl első 10 karakterét írási céllal.

A program futtatásakor a következőket írja ki a szabványos hibacsatornára:

```
Gyermek  tovabbfutott!
Szulo: elkezd futni a lock elott ...
Szulo:  SETLK sikertelen ... ujraprobalom
Szulo:  SETLK sikertelen ... ujraprobalom
Szulo:  SETLK sikertelen ... ujraprobalom
Szulo:  SETLK sikertelen ... ujraprobalom
Szulo lockolt es tovabbfutott!
```

Ez érthető, mert a szülő folyamat 1 másodpercet vár, majd utána próbálja meg

lefoglalni 1 másodpercenként a fájlt, és ez csak azután sikerül, miután a gyermeke a fájlt már nem foglalja olvasási céllal.

3.5 Kivételes események kezelésének rendszerhívásai

Ebbe a csoportba tartoznak a következő rendszerhívások: `signal()`, `kill()` és az `alarm()`, valamint a POSIX 1003.1-konform rendszerekben néhány más rendszerhívás. Ebben a részben először bemutatjuk a kivételes események kezelésére bevezetett **signal** absztrakciót, majd bemutatjuk a hagyományos UNIX eszközöket e kivételes események kezelésére, végül megmutatjuk a POSIX 1003.1 szabvány ide vonatkozó módosításait, amelyek a hagyományos UNIX eszközök tervezése során elkövetett hibák elkerülése érdekében készültek.

3.5.1 A signalok feladata

A signalok lényegében a megszakításoknak a magasabb szintű megfelelői (nevezhetjük őket szoftver-megszakításoknak is). Egy signalt egy program vagy az operációs rendszertől, vagy egy másik programtól (folyamattól ...) kaphat (így ez is tekinthető párhuzamos programok kommunikációs eszközének). Többfajta signal is van (pl. a BREAK billentyű lenyomásakor egy ún. SIGINT nevű signal generálódik a programnak).

3.5.2 Hagyományos signalkezelési technikák

A programok szabadon rendelkezhetnek arról, hogy egy adott fajtájú signallal mit akarnak csinálni. Erre való a signal rendszerhívás. Ennek első paramétere a signal-fajtát adja meg, második pedig megmondja, hogy mit kell az olyan fajta signalokkal csinálni (lehetséges értékei: `SIG_IGN` = a signalt nem kell figyelembe venni; `SIG_DFL` = a rendszerben alapértelmezésnek számító dolgot kell csinálni; ezenkívül megadhatjuk a program egy eljárásának a címét is, amely az adott típusú signalok esetén lesznek meghívva). Példa:

```
signal(SIGINT, SIG_IGN); /* Nem fogadom ezutan a SIGINT-et */
```

A UNIX rendszerben a fontosabb signalok a következők:

- SIGHUP : Megszakadt a felhasználói terminál és a gép közötti kapcsolat
- SIGINT : "DEL" billentyűt megnyomták
- SIGQUIT : Ctrl- billentyűt megnyomták
- SIGILL : A processzor "illegal instruction"-t talált
- SIGIO : Egy aszinkron I/O esemény bekövetkezéséről kapunk ezzel értesítést.
- SIGFPE : A lebegőpontos számításokat végző egység valami kivételes esetet jelez.
- SIGKILL : A folyamatot meg akarják állítani (ezt a signalt nem lehet ignorálni vagy elkapni!)

- SIGSEGV : A program megszegte a szegmentálási szabályokat
- SIGPIPE : Olyan pipe-ra akarunk írni, amit senki sem olvas
- SIGPWR : Áramkimaradás van (a rendszer ilyenkor általában szünetmentes tápegységről megy, és hamarosan leállása várható).
- SIGALRM : Korábban `alarm()` rendszerhívással kért signal megérkezése
- SIGCLD : A folyamat egy gyermek-folyamata befejeződott.
- SIGURG : A folyamat sürgősnek minősített adatokat kap (ld. később a hálózatokról szóló részt).
- SIGUSR1 : A signal jelentését a felhasználó (programozó) szabadon definiálhatja.
- SIGUSR2 : A signal jelentését a felhasználó (programozó) szabadon definiálhatja.
- SIGSYS : A folyamat egy ismeretlen rendszerhívást hajtott végre.
- (Ezeken kívül más signal-ok is vannak, de ez most nem érdekes.)

A `kill()` rendszerhívással lehet egy signalt küldeni egy folyamatnak. A rendszerhívás első paramétere az elküldendő signal típusa, a második araméter pedig annak a folyamatnak az azonosítója, akinek a signalt szántuk. (Ha a pid negatív, akkor a signal minden olyan folyamatnak el lesz küldve, amelynek a folyamat-csoport azonosítója egyenlő pid argumentumának abszolutértékével.) Van egy speciális típusú signal, a SIGALRM. Ezt lehet az `alarm()` rendszerhívással generálni. Az `alarm()` egyetlen paramétere egy egész szám. Abban az egész számban kell megadni az operációs rendszernek, hogy hány másodperc múlva küldjön a folyamatunknak egy SIGALRM signalt.

3.5.3 POSIX signal-szemantika

A POSIX signal-feldolgozást irányító rutinjait úgy tervezték, hogy azok alapvetően signal-halmazokkal foglalkozzanak: ehhez bevezették a `sigset_t` adattípust, mely adattípushoz definiálták a manipulációjára használható műveleteket is. Ezek a műveletek a következők[1] :

```
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set,int sig);
int sigdelset(sigset_t *set,int sig);
int sigismember(sigset_t *set,int sig);
```

A `sigemptyset()` függvény az argumentumában megadott signal-halmazt üresre állítja, azaz olyan állapotúra, hogy egyetlen signal sem kerül bele. A `sigfillset()` függvény az argumentumában megadott signal-halmazt olyan értékűre állítja be, hogy a rendszerben az összes definiált signal-típus benne legyen. A `sigaddset()` függvénnyel

[1]Ezekre a műveletekre azért van szükség, mert a signalok implementációfüggő – esetleg túlságosan nagy – száma miatt nem lehet minden architekturán egyformán implementálni – mondjuk egészekkel, és az ilyen adattípusok kezelése esetleg nem lenne hordozható, ezért e leírás keretében ezt kerülni fogom.

lehet az első argumentumában megadott signal-halmazba a második argumentumban megadott signal-típust betenni; a `sigdelset()` függvénnyel pedig az első argumentumában levő signal-halmazból a második argumentumában megadott signalt kivenni. A `sigismember()` függvény visszatérési értéke 1 (igaz), ha az első argumentumában megadott signal-halmaz tartalmazza a második argumentumában megadott signalt.

A POSIX szabvány definiálja a **maszkolt signal**ok fogalmát: ez egy folyamatonként (processzenként) nyilvántartott attribútum, mely tartalmazza az összes olyan signalt, amelyeket a folyamat **blokkolt**, vagyis azokat a signalokat, amelyeknek a kiváltását a folyamat megtiltotta (az operációs rendszernek). Természetesen egy folyamatnak küldhetnek olyan signalokat, amiket az éppen blokkolt, de azoknak a kiváltásával az operációs rendszervárni fog a blokkolás befejezéséig. Az ilyen signalokat, amiket egy folyamat blokkol, de küldött már neki valaki: **várakozó** signaloknak is nevezik.

Az egyes signalok kezelésének mikéntjét a `sigaction()` rendszerhívással jelölhetjük ki. Ennek paraméterezése a következő:

```
int sigaction(int sig, const struct sigaction *newact,
              struct sigaction *oldact);
```

A `sigaction()` rendszerhívás első argumentuma azt határozza meg, hogy melyik signal kezelését akarjuk megváltoztatni vagy lekérdezni[2] A második argumentumban lehet kijelölni az első argumentumban megadott signal-típus új (mostantól érvényes) kezelésmódját, illetve ott `NULL` pointert megadva a rendszerhívás nem módosítja a signal éppen érvényes kezelésmódját. A harmadik argumentumban (ha ott nem `NULL` értéket adunk át) az operációs rendszer visszaadja a signal eddigi kezelésének módjára vonatkozó információkat.

A signal-kezelés módját leíró `struct sigaction` struktúra felépítése a következő:

```
struct sigaction {
        void (*sa_handler)();
        sigset_t sa_mask;
        int sa_flags;
};
```

Az `sa_handler` komponense értéke vagy a `SIG_IGN` vagy `SIG_DFL` szimbolikus konstansok valamelyike (lásd részletesebben a 3.5.2 pontot), vagy pedig a signal-kezelő függvény címe lehet. Ha egy signal-kezelő függvény van az `sa_handler` komponensben, akkor az `sa_mask` komponens a signal-kezelő függvény futása alatt a még maszkolandó signalok halmazát kell, hogy tartalmazza (vagyis a signal-kezelő futásakor a maszkolt signalok halmaza ki fog bővülni az `sa_mask` halmazban megadott signalokkal, a signal-kezelő lefutásának végéig). Az éppen kiváltott signal automatikusan belekerül az `sa_mask` halmazba, így azt nem kell belerakni abba. Az `sa_flags` a signal-kezelés folyamatát befolyásolja, és a POSIX az egyetlen `SA_NOCLDSTOP` lehetséges flaget definiálja a következőképpen: ha ez be van állítva, akkor az operációs rendszer nem fog `SIGCHLD`[3] signalt generálni olyankor, amikot a folyamat valamely gyermek-folyamatának felfüggesztődik vagy tovább folytatódik a futása.

Lehetőség van az éppen maszkolt signalok halmazának lekérdezésére és módosítására is: erre használható a `sigprocmask()` rendszerhívás. Ennek prototípusa a következő:

[2] A `sigaction()` rendszerhívás használható a signal éppen érvényes kezelésmódjának lekérdezésére is.

[3] A SIGCHLD a Berkeley UNIX SIGCLD signaljának a megfelelője az AT&T System V UNIX rendszerében.

```
int sigprocmask(int hogyan, const sigset_t *newset, sigset_t *oldset);
```

Az első argumentum értéke háromféle lehet: vagy `SIG_BLOCK` vagy `SIG_UNBLOCK` vagy pedig `SIG_SETMASK`. `SIG_BLOCK` érték esetén a második (`newset`) argumentumban levő signalok hozzáadódnak a maszkolt signalok halmazához. `SIG_UNBLOCK` esetén a második argumentumban levő signalok ki lesznek véve a maszkolt signalok halmazából, míg a `SIG_SETMASK` esetén a maszkolt signalok halmazát `newset`-re állítja az operációs rendszer, az addigi tartalmától függetlenül. Ha a `newset` argumentum értéke `NULL`, akkor a maszkolt signalok halmaza nem módosul. Ha az `oldset` argumentum értéke nem `NULL`, akkor az általa kijelölt helyre az operációs rendszer eltárolja az éppen maszkolt signalok halmazát.

A POSIX definálja még a `sigpending()` illetve a `sigsuspend()` rendszerhívásokat is: ez előbbivel egy folyamat lekérdezheti a számára küldött, de várakozó signalok halmazát; míg a `sigsuspend()` függvénnyel a maszkolt signalok halmaza módosítható, majd az operációs rendszer a folyamat futását felfüggeszti addig, amíg legközelebb egy signalt nem kap.

3.6 Még egy kicsit a folyamatokról

A későbbiekben bemutatott példaprogramok (pl. majd a TCP alapú konkurrens szerver) működésének megértéséhez fontos tisztában lenni azzal, hogy mi történik miután egy folyamat meghal (végrehajtja az `exit()` rendszerhívást vagy kill signalt küld önmagának, stb ...), hogy szerezhet erről tudomást a szülő folyamat.

Egy szülő-folyamat egy gyermek-folyamat befejeződésére a `wait()` rendszerhívással várakozhat. Egyetlen paramétere egy int típusú objektumra mutató pointer, ahova (a magasabb helyiértékű byteba) a meghalt gyermek exit-státusza kerül (ami 0 szokott lenni, ha nem volt hiba; egyébként nem 0, de ez inkább egy konvenció, aminek a betartása nem kötelező) - ha a pointer nem NULL-pointer. Visszatérési értéke egyenlő a befejeződott gyermek-folyamat azonosítójával (ha van ilyen). Ha a folyamatnak nincs egyetlen (futó vagy befejeződött) gyermeke, akkor a visszatérési értéke -1.

A következő esetek lehetségesek miután egy gyermek-folyamat végrehajtotta az `exit()` rendszerhívást, ami után nincs visszaút:

- A szülő folyamat már végrehajtott egy `wait()` rendszerhívást, és ekkor a rendszerhívás a fent említett módon visszatér.

- A szülő folyamat (még) nem hajtott végre `wait()`-et. Ekkor a gyermek-folyamatból zombie-processz lesz (erről korábban már volt szó). Ez azt jelenti, hogy minden általa lefoglalt erőforrás fel lesz szabadítva, kivéve a processz-táblabeli bejegyzés, amiben az exit-statusz van tárolva (ennek az az oka, hogy nem lehet tudni azt, hogy a szülő végrehajt-e majd valamikor egy wait rendszerhívást ...)

- Ha a folyamat azonosítója és folyamat-csoport azonosítója megegyezik, és a folyamat "login"-shell folyamat volt (azaz a terminal group azonosító 2.1 is egyenlő a folyamat azonosítójával), akkor akkor a folyamat-csoport minden tagja megkap egy SIGHUP signalt. (Ezért halhatnak meg a háttérben futó folyamataink, ha kijelentkezünk - és ezt a signalt ignoráltatja a nohup UNIX parancs a folyamattal.)

- A UNIX másik lehetősége, hogy miután egy folyamat egy `exit()` rendszerhívást végrehajtott, azután egy SIGCLD (Signal the Death of a Child) signalt küld a szülőnek. Ez azért jó, mert a signal-handlerben le lehet kérdezni a meghalt gyermek-folyamat exit-státuszát egy wait hívással, így elkerülhető a zombie-folyamatok véletlen elszaporodása. (Csak az AT&T System V UNIX ad arra lehetőséget, hogy a programnak ne is kelljen törődni a zombie-processzekkel. Ehhez végre kell hajtani a

  ```
  signal(SIGCLD, SIG_IGN);
  ```

  rendszerhívást. Ez azt eredményezi, hogy ezután ha egy gyermek meghal, a UNIX mag nem generál a szülőnek SIGCLD signalt, a gyermek a zombie állapot kihagyásával örökre feledésbe merül.)

Még egy fontos dolog van, amin a UNIX készítőinek el kellett gondolkodniuk: mi legyen akkor, ha a szülő-folyamat hal meg előbb?

Ekkor a gyermekhez tárolt szülő-folyamat azonosító mező érvénytelen folyamat-azonosítót tartalmaz! Ezt úgy oldották meg, hogy bevezettek egy speciális (1-es azonosítójú) folyamatot, az `init` folyamatot (ez indítja el a terminálokon a login: folyamatokat). Az "árva" gyermek-folyamatoknak ez lesz szülő-folyamatukként tárolva. Ez majd időnként figyeli, és megtisztítja a "halott árva zombie" folyamatok által lefoglalt processztábala-bejegyzésektől a processz-táblát. (Az init folyamat sosem hajt végre `exit()` rendszerhívást.)

3.7 INPUT/OUTPUT eszközöket vezérlő rendszerhívás

A UNIX rendszerben egyetlen I/O-vezérlő rendszerhívás van: az `ioctl()`, de ez a különböző perifériákon (karakteres speciális fájlokon) más-más dolgokat végezhet (a paramétereit az adott perifériához tartozó device driver kapja meg és úgy értelmezi, ahogyan akarja). A PC-ken például ez a rendszerhívás kapcsolhatja a képernyőt grafikus megjelenítési módba.

Egyetlen használatát érdemes megnézni ennek a rendszerhívásnak: a terminál-módok beállítását.

A UNIX rendszerekben a terminál device driverje három különféle módon működhet: **raw**, **cbreak** illetve **cooked** módban. A raw üzemmódban a terminálon (billentyűzeten, RS-232 vonalon, ...) beadott karakterek egyenként át lesznek adva a felhasználói programnak, ekkor minden billentyű "egyenrangú", a speciális billentyűknek (pl. backspace, CTRL-S, CTRL-Q stb.) ekkor nincs semmi hatásuk a programra. A cbreak módban a karakterek szintén egyenként lesznek a felhasználói programnak átadva, de a speciális billentyűket ekkor az operációs rendszer "elkapja", feldolgozza, és ezeket nem adja tovább a felhasználói programnak. A speciális billentyűket és hatásukak a következő táblázat foglalja össze (a lista nem teljes):

- CTRL-S : A képernyőreírást fel kell függeszteni
- CTRL-Q : A képernyőreírást folytatni kell (mivel CTRL-S-sel valamikor meg lett állítva).

- CTRL-D : Fájlvége generálása
- DEL : SIGINT signal küldése a futó programjainknak (ezt a program letilthatja).
- CTRL-\ : SIGQUIT signal küldése a futó programnak, hogy álljon le, és a memória tartalmát az operációs rendszer egy `core` nevű fájlba kimenti az aktuális directoryba. (Azaz egy core-dumpot készít róla az operációs rendszer, amit később egy debuggerrel elemezni lehet ...).

A cooked módban az operációs rendszer sorvége- (vagy fájlvége-)jellel lezárt egész sorokat olvas be a billentyűzetről, és a beolvasott sort egyben adja á a felhasználói programnak. A felhasználó a backspace billentyűt használhatja a beadott karakterek törlésére, és a fent említett speciális funkciójú billentyűk is hatásosak.

A terminál-paraméterek egy `termio` struktúra típusú változóba lekérdezhetők az operációs rendszertől[4], és ha akarjuk, akkor ezt megváltoztathatjuk, majd visszaadhatjuk az operációs rendszernek, hogy állítsa be a belső strukturáit ennek megfelelően. A struktúra szerkezete a következő (itt is a teljesség igénye nélkül):

```
#define NCC ....

struct termio {
     short c_iflag;
     short c_oflag;
     short c_cflag;
     short c_lflag;
     char c_line;
     short c_cc[NCC];
};
```

Az egyes mezőkben a fent említett módokat, RS-232 terminálnál a baudrate értékét, a karaktertörlő billentyű és más speciális billentyűk kódját lehet megadni, és még sok-sok más dolgot (pl. az operációs rendszer végezzen-e a beadott karaktereken `CR-->LF` konverziót vagy sem, . . . – ez hasznos lehet például terminál-emulátor programok készítésénél, de erről most nem írok részletesebben).

A terminálparamétereket a

```
ioctl(tfd, TCGETA, &tpars);
```

rendszerhívással lehet lekérdezni egy `tpars` (struct termio típusú) változóba (tfd a terminálhoz kapcsolt fájl, lehet például 0 is, ha a szabványos bemenet a terminálhoz van kötve). A paramétereket átállítani így lehet:

```
ioctl(tfd, TCPUTA, &tpars);
```

RAW módba kapcsoláshoz a következő paramétereket kell átállítani:

[4] A POSIX szabványban kicsit módosult a struktúra szerkezete, és átnevezték `termios` névre, de lényeges változások nem történtek – kényelmi függvények bevezetésén és egy-két új terminálparaméter bevezetésén kívül.

```
/* ioctl ... */
oldlflag=lflag; /* Elozo modot elmentem */
lflag=(lflag & ~(ECHO | ISIG | ICANON); /* Karaktervisszairast
          (ECHO-t) is kikapcsolom */
```

Vissza az "előző" módba:

```
lflag=oldlflag;
/* ioctl ... */
```

(Megjegyzés: ezek a műveletek egyes UNIX rendszerekben lehet, hogy máshogy mennek, esetleg más mezőnevek vannak a termio struktúrában, ... ennek érdemes utánanézni mondjuk a "man termio" paranccsal. Ezeknek a beállítására egyébként létezik egy **curses** nevű standard könyvtár is benne a `raw()`, `cooked()`, `cbreak()` függvényekkel, amit ha kell használhatunk.)

A UNIX kernelnek azt a részét nevezik **line discipline** modulnak, amely a teriminálinterfaceen bejövő karakterek feldolgozását végzi a fenti szempontok szerint.

3.8 Egyéb rendszerhívások

Ebbe a csoportba azok a rendszerhívások kerültek, amelyeket érdemes megismerni, de a fent említett csoportok közül egyikbe sem tartoznak olyan szorosan. Az itt ismertetendő rendszerhívások a következők: `chmod()`, `chown()`, `chgrp()`, `getuid()`, `getgid()`, `geteuid()`, `getegid()`, `setuid()`, `setgid()` és a `time()`.

A `chmod()` rendszerhívással lehet beállítani (ill. átállítani) egy fájl védelmi bitjeit. Az első paramétere a fájl neve, a második paraméter pedig a védelmi bitek új értéke. A `chown()` rendszerhívással lehet egy fájl tulajdonosának az azonosítóját megváltoztatni. A rendszerhívás első paramétere a fájl neve, második paraméter pedig az új tulajdonos felhasználói azonosítója. A harmadik paraméter az új tulajdonos csoport-azonosítója. (Ezt a rendszerhívást csak a fájl "jelenlegi" tulajdonosa vagy a root (a szuperfelhasználó) hajthatja végre.)

A `getuid()`, `getgid()` rendszerhívás a programot futtató felhasználó azonosítóját illetve csoport-azonosítóját adja vissza eredményül. A `geteuid()` ill. `getegid()` rendszerhívás hasonlóan működik, kivéve ha egy setuid bittel ellátott programban hajtják végre. Ekkor annak az uid-jét ill. gid-jét adja vissza eredményül, akié a futtatható program volt (pontosabban: az effektiv felhasználói azonosítót és csoport-azonosítót adja vissza).

A `setuid()` és `setgid()` rendszerhívásokkal a folyamathoz tárolt felhasználói azonosítót és csoport-azonosítót lehet átállítani. A szuperfelhasználó (illetve az a folyamat, amely setuid root módon lett elindítva) bármilyen felhasználói- és csoport-azonosítóra beállíthatja a folyamathoz tárolt felhasználói- és csoport-azonosító értékeket, de ez visszafele nem megy. Egy "mezei" felhasználói folyamat nem tudja a tárolt azonosítókat megváltoztatni.

A `time()` rendszerhívásnak egy paramétere van (de annak értéke NULL-pointer is lehet). A rendszerhívás eredményül visszaadja az 1970 január 1-e óta eltelt másodpercek számát. Ha a paraméter nem NULL- pointer, akkor ezt az értéket beírja a paramétere által mutatott memóriacímtől.

3.9 Egy összetettebb példa: a shell

Ebben a részben egy minimális képességű shell (parancsértelmező) funkcióit ellátó program forráslistája van. A programon keresztül a UNIX alapvetőbb rendszerhívásait ismerhetjük meg (ezért kerültem a szabványos I/O könyvtár használatát, a `printf()`-et és egyebeket).

A shell főprogramja – a `main()` függvénynél ciklusban kiír egy prompti (`$`) karaktert – ezzel jelezve a felhasználónak, hogy parancsra vár – majd beolvas egy sort a szabványos bemenetről (ami ugye leggyakrabban a billentyűzetről olvasást jelenti interaktív shelleknél).

Ezután a beadott parancsot a `darabol()` nevű eljárással szétszedi részeire, az `argstrs` nevű paraméterében megadott tömböt (a benne levő mutatókat) a beadott parancs egyes komponenseire állítja: az első (azaz a nulla indexű) elemet a végrehajtandó parancs nevére állítja, a következő elemet a végrehajtandó parancs első argumentumára állítja, stb. Az egyes parancsneveket/argumentumokat terminálja a `\0` karakterrel.

A szétdarabolás után a főprogramban szül egy gyermek-folyamatot, ami először ellenőrzi, hogy az utolsó argumentumban a szabványos kimenetet akarták-e átirányítani, és ha úgy találja, hogy azt akarták (vagyis egy `>` karakterrel kezdődik), akkor lezárja az addig érvényes szabványos kimenetet, és megnyit egy új fájlt (ami a szabványos kimenet helyén, az 1-es fájldeszkriptorral jön létre, mivel az a legelső szabad fájldeszkriptor – ne felejtsük el, hogy a 0-ás fájldeszkriptoron a szabványos bemenetet nem bántottuk, nem zártuk le, ezért a program ilyen szempontból helyesen működik). A gyermek-folyamat ezután végrehajtja az `execvp()` rendszerhívással a kívánt programot.

A szülő-folyamat várakozik, amíg a gyermeke befejeződik, majd új parancsot kér. Megjegyezzük, hogy a folyamatok háttérbeli elindíthatósága azt jelenti, hogy ezt a várakozást (a `wait()` rendszerhívást) ki kellene hagyni, és a gyermek-folyamat halálát jelző signalt kellene kezelnie a shellünknek.

```
/* minsh.c
 *
 * Egy minimalis shell
 * A szabvanyos bemenet (standard input) csatornarol olvas programneveket es
 *   program-argumentumokat, es vegrehajtja a megadott programokat a megadott
 *   argumentumokkal.
 * A szabvanyos kimenet atiranyithato - elegge primitiven:   >fajlnev
 *   metakarakterekkel ... mint az igazi shellekben, DE itt ez csak es
 *   kizarolag az utolso argumentumban fordulhat elo, vagyis a parancssor
 *   vegen.
 */

#include <fcntl.h>

#define CBUFSIZE 1024 /* Parancsbuffer merete */
#define NR_ARGSTRS 32 /* Maximalis argumentumszam */
#ifndef NULL
#define NULL ((void *)0)
#endif
```

```
void darabol(pbuf,argstrs,argn)
  char *pbuf;
  char **argstrs;
  int *argn;
{
  int poziciosszamlalo=0,pbufhossz;
  int local_argn;

  local_argn=0;
  pbufhossz=strlen(pbuf);
  while (isspace(pbuf[poziciosszamlalo]) && poziciosszamlalo<pbufhossz) {
    pbuf[poziciosszamlalo]='\0';
    poziciosszamlalo++;
  }
  while (poziciosszamlalo < pbufhossz) {
    argstrs[local_argn]= &pbuf[poziciosszamlalo];
    local_argn++;
    while (!(isspace(pbuf[poziciosszamlalo])) && poziciosszamlalo<pbufhossz)
      poziciosszamlalo++;
    while (isspace(pbuf[poziciosszamlalo]) && poziciosszamlalo<pbufhossz) {
      pbuf[poziciosszamlalo]='\0';
      poziciosszamlalo++;
    };
  };
  argstrs[local_argn]=NULL;
  local_argn++;
  *argn=local_argn;
}

void main(argc,argv,envp)
  int argc;
  char **argv, **envp;
{
  int gy_status; /* A befejezodott gyermekfolyamat allapotarol ad informaciot */
  char parancsbuf[CBUFSIZE];
  char *argstrings[NR_ARGSTRS];
  int argnum,fn,nchars;
  char *stdoutfnev;

  do {
    write(1,"$ ",2);
    if ((nchars=read(0,parancsbuf,CBUFSIZE)) > 0) { /* -1 hiba, 0 fajlvege */
      darabol(parancsbuf,argstrings,&argnum);
      if (argnum > 1) { /* Ha megadtak valami (vegrehajtando-) fajlnevet */
        if (fork() == 0) {
          fn=1;
          if ((argnum > 2) && (argstrings[argnum-2][0]=='>')) {
```

```
            stdoutfnev=&(argstrings[argnum-2][1]); /* Ronda, de vilagos ... */
            close(1); /* lezarja az ezelotti szabvanyos kimenet fajlt */
            fn=creat(stdoutfnev,0744);
            argstrings[argnum-2]=NULL; argnum=argnum-1;
          }
          if (fn == 1)
            execvp(argstrings[0],argstrings);
          else
            perror("Standard kimenet atiranyitasa sikertelen ");
          perror("execvp nem sikerult ");
          exit(-1); /* Hiba - execvp sikertelen */
        } else {
          wait(&gy_status); /* szulo var */
        }
      }
    }
  } while (nchars != 0);
  if (nchars == (-1)) perror(" hiba a standard input olvasasakor ");
}
```

3.10 Daemon folyamatok

A UNIX operációs rendszerben vannak olyan feladatok, amiket a háttérben (gyakran "észrevétlenül") futó folyamatok végeznek el. Ilyen például a rendszeres feladatok futtatását végző `cron` program. Ezeket a programokat **daemon** folyamatoknak is nevezik. Mivel ezeket a folyamatokat nem a terminálról indítják, általában a rendszerindításkor automatikusan indulnak el, ezért nem illik a képernyőre írniuk (esetleg nagyon nagy rendszerhibák esetén ezt azért megtehetik), és nem illik olyan erőforásokat fenntartaniuk, amikre nincs szükségük. Ahhoz, hogy egy daemon folyamat a feladatának ellátásához szükséges erőforrásokon kívül minél kevesebb más erőforrást foglaljon le, kialakítottak olyan konvenciókat, amiket a daemon folyamatokat írva követni érdemes. Ebben a pontban ezekről a konvenciókról lesz szó, majd bemutatunk egy C kódrészletet, amely e konvenciók alapján lett megírva, és felhasználhatjuk saját daemon folyamataink készítésekor.

- Egy daemon folyamatnek először is végre kell hajtania egy `fork()` rendszerhívást, majd a szülő részének egy `exit()`-et. Ezzel egyrészt a gyermek – a leendő daemon – nem lesz processz-csoport vezetője, másrészt pedig mivel a szülő végrehajt egy `exit()` rendszerhívást, ezért a shell úgy tekintheti, hogy a daemont elindító parancs befejeződött (vagyis eredményül a daemon fut, de a shell promptot is visszakapjuk).

- Végre kell hajtani a `setsid()` rendszerhívást. Ezzel a folyamat egy új processz-csoportot képez, aminek ő lesz a vezetője, és a vezérő terminált sem köti le a továbbiakban (vagyis a kijelentkezésünk után újabb bejelentkezés is történhet rajta).

- A munka-directoryt állítsuk a gyökérre (/) vagy a /tmp-re. Ezzel a daemonunk nem akadályozza annak a fájlrendszernek az esetleges kiilesztését `umount()` rendszerhívással, amely az elindításakor aktuális munkadirectoryt tartalmazta.

- Állítsuk a folyamat `umask` értékét nullára.
- Zárjuk le a felesleges fájlokat.

A fenti lépéseket a következő eljárás megteszi:

```
int daemon_init(void)
{
  int pid;

  close(0);
  close(1);
  close(2);
  pid=fork();
  if (pid < 0) return (-1);
  if (pid != 0) exit(0); /* a szulo folyamat befejezi a futasat */
  setsid();
  chdir("/");
  umask(0);
  return(0);
}
```

3.11 POSIX-threadek

Igaz ugyan, hogy a "hagyományos" UNIX rendszerek (egy-két kivétellel) nem biztosítanak lehetőséget több szálon futó (multi-threaded) alkalmazások készítésére rendszerszolgáltatás szinten, de manapság egyre többféle olyan C könyvtár kapható (mind a kommersz, mind pedig a szabad szoftver szférában), amely lehetővé teszi a UNIX folyamatok továbbosztását, többszálú futtatását. Ebben a pontban erről fogok részletesebben írni. Mivel a legtöbb ilyen kiegészítő könyvtár a POSIX thread szabványát követi (ez a POSIX 1003.4a), ezért én is ezt a szabványt fogom bemutatni. Megjegyzem, hogy a POSIX thread-szabványának csak egy részét fogom ismertetni; nem célom a szabvány teljes körű bemutatása.

3.11.1 POSIX threadek létrehozása és megszűntetése

Egy új POSIX threadet (a továbbiakban P-thread) a `pthread_create()` függvénnyel hozhatunk létre. Ennek prototípusa a következő:

```
int pthread_create  (pthread_t *thread,
                     pthread_attr_t *attr,
                     pthread_func_t func,
                     any_t arg);
```

A `pthread_create()` függvény létrehoz egy új threadet az `attr` argumentumában megadott attributumokkal[5] . A thread futása a `func` argumentumban megadott

[5]Thread attributum például a thread számára rendelkezésre álló végrehajtási verem mérete.

függvénynek, az `arg` argumentumban átadott értékkel mint függvényargumentummal történő végrehajtásából áll (vagyis ez úgy viselkedik, mintha végrehajtanánk párhuzamosan a `func(attr);` C nyelvű függvényt ...). Az első argumentumban azt a helyet kell megadni, ahova a thread-könyvtár az elindított thread azonosítóját teheti. Ha az elindítandó thread attributumaival kapcsolatban nincsenek különösebb igényeink, akkor megadhatjuk az `attr` argumentumként a `pthread_attr_default` értéket.

Miután a thread befejezte a feladatát, végre kell hajtania a `pthread_exit()` függvényt. Ennek a függvénynek egyetlen argumentuma van, a befejeződött thread kilépési kódja, amit a thread elindítója a thread befejezése után megkap. Ennek a függvénynek a prototípusa a következő:

```
void pthread_exit(any_t status);
```

P-threadek várhatnak más P-threadek befejeződésére a `pthread_join()` függvénnyel. Ennek prototípusa a következő:

```
int pthread_join(pthread_t thread, any_t *exit_status);
```

A `pthread_join()` függvényt végrehajtó thread megvárja az első argumentumában megadott thread befejeződését, majd a második argumentumában visszaadja a befejeződött thread kilépési kódját (amit a `pthread_exit()` végrehajtásakor a befejeződő thread megadott).

Mivel a `pthread_join()` függvényt a már befejeződött P-threadekre vonatkozóan is végrehajthatjuk (és ez működik is!), ezért a P-thread könyvtár kénytelen az összes elindított threadről információkat nyilvántartani (például a thread kilépési kódját). Ahhoz, hogy a nyilvántartott információk ne foglalják le a memóriát, a programozónak lehetősége van a `pthread_detach()` függvénnyel azt tanácsolnia a P-thread könyvtárnak, hogy a függvény egyetlen argumentumában specifikált threadre vonatkozóan nem fogja többé végrehajtani a `pthread_join()` függvényt, ezért a megadott threadre vonatkozóan nyilvántartott információkat a továbbiakban nem kell tárolni. A prototípusa a következő:

```
int pthread_detach(pthread_t *thread);
```

3.11.2 POSIX threadek identitása

A `pthread_self()` függvény használható a thread identitásának meghatározására. E függvény visszatérési értékeként megadja az őt végrehajtó thread thread-azonosítóját. A függvény prototípusa a következő:

```
pthread_t pthread_self(void);
```

A `pthread_t` típusú adatelemek egyenlőségvizsgálatára használható a `pthread_equal()` függvény. Két argumentuma egy-egy P-threadet azonosítson, visszatérési értéke pedig csak akkor igaz, ha az argumentumaiban megadott thread-azonosítók ugyanazt a threadet azonosítják. Ennak a függvénynek a prototípusa a következő:

```
int pthread_equal(pthread_t thread1, pthread_t thread2);
```

3.11.3 POSIX threadek szinkronizációja

A POSIX szabvány a threadek szinkronizációjára két alapvető eszközt biztosít: a **mutex**eket és az **őrfeltétel-változókat**.

Mielőtt e két szinkronizációs eszközt megismernénk, megjegyezzük, hogy mind a mutexeket, mind pedig az őrfeltétel-változókat használatuk előtt inicializálni kell, amire a következő függvényeket használhatjuk:

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       pthread_mutexattr_t *mattr);
int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t *cattr);
```

Mindkét függvény második argumentuma az adott szinkronizációs eszköz jellemzőit kell, hogy rögzítse (erről részletesebben a 3.11.4 pontban fogok írni).

Egy mutex általában a **szabad** és a **foglalt** állapotok valamelyikében van, és a mutexeket két művelettel lehet manipulálni: a `LOCK` és az `UNLOCK`. A `LOCK` művelet a szabad mutexet foglaltra állítja, a foglalt mutexnél pedig addig várakozik a program, amíg a mutex szabaddá nem válik, és azután állítja foglaltra a mutexet. Az `UNLOCK` művelet a mutexet szabadra állítja. Megjegyezzük, hogy mind a `LOCK` mind pedig az `UNLOCK` műveletnél leírtak atomi, oszthatatlan módon történnek, ezért egy szabad mutexet két egymástól független `LOCK` semmiképpen nem állíthat foglalttá: ilyenkor először csak az egyik `LOCK` lesz sikeres, majd miután a sikeres `LOCK`-ot végrehajtó kiadja az `UNLOCK`-ot, azután fogja a másik (`LOCK`-ra várakozó) a mutex állapotát foglalttá módosítani. (A mutex lényegében egy bináris szemafor funkcióját valósítja meg; a `LOCK` és az `UNLOCK` pedig a Dijkstra-féle P/V szemaforműveleteket implementálják.)
A `LOCK` művelet a `pthread_mutex_lock()` függvénnyel van implementálva, míg az `UNLOCK` a `pthread_mutex_unlock()` függvénnyel implementálható. Ezeknek a prototípusa a következő:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

A `pthread_mutex_trylock()` függvény lefoglalt (`LOCK`-olt állapotban levő) mutex megadása esetén nem várakozik, mint a `pthread_mutex_lock()`, hanem `-1` visszatérési értékkel azonnal visszatér, és az `errno` változóban az `EBUSY` hibakód-konstanst helyezi el. Ha szabad a mutex, akkor pedig végrehajt rajta egy `LOCK` lefoglalási műveletet, és úgy tér vissza.

Az őrfeltétel-változókat hosszabb várakozások esetén érdemes használni. A mutexeket csak rövid ideig tartó, threadek közti kölcsönös kizárás megszervezésére használjuk, és egy mutexet foglaló threadet soha ne várakoztassuk meg hosszabb ideig (a gyakorlatban általában a mutexekkel egy-egy (konstans) értékadás oszthatatlanságát szokták biztosítani úgy, hogy az értékadás elé beraknak egy `LOCK`, mögé pedig egy `UNLOCK` műveletet egy, az értékadás bal oldalán álló változót "védő" mutexszel). Az őrfeltétel-változókat általában a hosszabb idejű várakoztatásokra használják, amikor egészen addig kell várni, amíg egy erőforrás fel nem szabadul. Általában minden őrfeltétel-változóhoz tartozik egy mutex objektum is, aminek a használatát az őrfeltétel-változón értelmezett két

alapművelet bemutatásakor ismertetek. Az őrfeltétel-változókon két alapvető művelet van definiálva: a várakozás és a "szabad" jelzés küldése az őrfeltétel-változón várakozó folyamat(ok) részére.

Egy őrfeltétel-változón való várakozás úgy van definiálva (és megvalósítva), hogy `UNLOCK`-olja a hozzá tartozó mutexet, és várakozik, amíg az őrfeltétel-változón egy másik thread nem jelez "szabad" jelzést (és e két művelet "atomi", oszthatatlan módon hajtódik végre). Természetesen a várakozó művelet hívása előtt a mutexet `LOCK` művelettel át kell állítani. Az őrfeltétel-változónál várakozási művelet prototípusa a következő:

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
```

A művelet első argumentum annak az őrfeltétel-változónak a neve, amelyre vonatkozóan később a "szabad" jelzést várjuk; a második argumentum pedig a – hívás előtt – `LOCK`-olt mutexet adja meg. Ennek a függvénynek a használatakor előfordulhat – ha kis valószínűséggel is –, hogy a visszatérésekor az őrfeltétel-változó nincs "szabad" állapotban, vagyis tovább kell várni (ez akkor fordulhat elő, ha egy őrfeltétel-változónál egyidejűleg több thread várakozott, és a "szabad" jelzés után először ütemezett thread lefoglalja az erőforrást, a többi threadnek pedig újra várakoznia kell). A függvény visszatérésekor az garantálva van, hogy a mutex `LOCK`-olt állapotban van a visszatérés után, így az előbb említett – "hamis szabad jelzés" – problémára megoldást jelenthet e függvénynek a ciklusban való végrehajtása addig, amíg ténylegesen megszereztük az erőforrást.
A várakozásra vonatkozóan időkorlátot is lehet adni a `pthread_cond_timedwait()` függvénnyel (ilyenkor nincs garantálva, hogy az erőforrás a megadott időn belül felszabadul, csak az, hogy ez a függvény legkésőbb az adott idő múlva visszatér).

Egy őrfeltétel-változóra "szabad" jelzés küldése a `pthread_cond_signal()` illetve a `pthread_cond_broadcast()` függvénnyel lehetséges. Ezek közül az első csak egy threadet enged továbbfutni, a második pedig az őrfeltétel-változón várakozó összes threadet továbbengedi[6]. E két függvény prototípusa a következő:

```
int pthread_cond_signal(pthread_cond_t *cond);

int pthread_cond_broadcast(pthread_cond_t *cond);
```

Mindkét függvény egyetlen argumentuma az az őrfeltétel-változó, amelynél várakozó folyamatok futását tovább akarjuk engedni. Az őrfeltétel-változók használatára tekintsük a következő példát – az első programrészlet egy erőforrást lefoglalni szándékozó threadből származik, míg a második egy, az erőforrást eddig birtokló, de a használati jogáról lemondani szándékozó thread által lesz végrehajtva.

```
pthread_mutex_lock(&mutex);
...
while (!szabad)
```

[6] Vagyis ez okozhatja a fent említett "hamis szabad jelzés" problémát – ugyanakkor gondoljuk meg, hogy egyes problémák megoldása során erre szükség lehet, mivel például egy adatbázis olvasását egyszerre több folyamat is végezheti, írását pedig legfeljebb egy, ezért ha egy, az adatbázist egyedül használó író folyamat munkájának befejezése után az összes olvasót egyszerre akarjuk továbbindítani (hiszen köztük nem kell kölcsönös kizárást biztosítani), akkor az összes várakozó folyamatot tovább kell engedni, és az egy másik megoldandó probléma lesz, hogy író folyamatokat ne engedjünk tovább, csak az olvasókat (ha van olvasó).

```
  pthread_cond_wait(&cond, &mutex);
szabad=FALSE;
pthread_mutex_unlock(&mutex);
```

Tekintsük az erőforrás használati jogról lemondani szándékozó threadből származó részletet:

```
pthread_mutex_lock(&mutex);
szabad=TRUE;
pthread_mutex_unlock(&mutex);
pthread_cond_signal(&cond);
```

Mind a mutexeknek mind pedig az őrfeltétel-változóknak megvannak a megszüntető műveletei (destruktor műveletek). Ezeknek a prototípusa a következő:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_cond_destroy(pthread_cond_t *cond);
```

3.11.4 Mutexek illetve őrfeltétel-változók attributumai

Mind a mutexek, mind pedig az őrfeltétel-változók egyes jellemzőit létrehozásukkor rögzíteni kell. Erre való a létrehozó `pthread_mutex_init()` illetve a `pthread_cond_init()` függvények második argumentumában megadható (pontosabban megadandó) attributum objektum – vannak mutex attributum objektumok illetve őrfeltétel-változó attributum objektumok is (ezek nem azonosak!).

Egy mutex attributum változót (típusa `pthread_mutexattr_t` a C nyelven) a `pthread_mutexattr_init()` függvénnyel inicializálhatunk, amelynek egyetlen argumentuma az inicializálandó mutex attributum objektum (memóriabeli címe). A függvény prototípusa a következő:

```
int pthread_mutexattr_init(pthread_mutexattr_t *mattr);
```

Egy mutex attributum változót megszüntetni a `pthread_mutexattr_destroy()` művelettel lehet. Ennek prototípusa a következő:

```
int pthread_mutexattr_destroy(pthread_mutexattr_t *mattr);
```

Az őrfeltétel-változó attributum objektumok létrehozását illetve megszüntetését a fentiekhez hasonló funkciójú függvényekkel végezhetjük el, melyeknek prototípusa a következő:

```
int pthread_condattr_init(pthread_condattr_t *cattr);
int pthread_condattr_destroy(pthread_condattr_t *cattr);
```

3.11.5 Könyvtárak thread-biztossága

Az operációs rendszer fejlesztői környezetéhez tartozó könyvtárak gyakran használnak globális változókat úgy, hogy a módosításuk (illetve kiolvasásuk) nincs mutexekkel védve (a változó írói valamint olvasói közt nincs megvalósítva a megfelelő kölcsönös kizárás). Ez sok problémát okozhat, aminek az a következménye, hogy ezek a könyvtárak módosítás nélkül nem használhatóak több threadet alkalmazó programokban. Ezen lehet segíteni ún. thread-köpenyekkel, amik egy egyszerű szinkronizációs mechanizmust támogatnak: egyetlen közös globális mutexszel védik a könyvtár függvényeit úgy, hogy egyszerre legfeljebb egy függvény lehet aktív. Ezt a szinkronizációs mechanizmust úgy meg lehet valósítani, hogy minden függvény törzse elé el kell helyezni egy, a globális mutexre vonatkozó `LOCK` műveletet, valamint a törzs végére egy ugyanarra a mutexre vonatkozó `UNLOCK` műveletet.

Gondoskodni kell még a mutex inicializálásáról, amit egy, a program elején meghívandó segédfüggvénnyel oldhatunk meg.

Vagyis az eredeti könyvtár egy függvénye a következőképpen módosul (ha "tisztességes" programot akarunk írni, akkor gondoskodni kell ennek a mutexnek a deallokálásáról is – ehhez meg kell írni ugyanitt egy eljárást, amit például a főprogramunk végén meghívhatunk):

```
...
static int kell_inic=TRUE;
static pthread_mutex_t mutex;
static pthread_mutexattr_t mattr;
...
fv_nevxxx(...)
{
  pthread_mutex_lock(&mutex);
  fv_nevxxx_EREDETI_TORZSE; /* Ide jon a fuggveny eredeti torzse */
  pthread_mutex_unlock(&mutex);
}
...
fv_init()
{
  pthread_mutexattr_init(&mattr);
  pthread_mutex_init(&mutex);
}
```

A thread-ekkel kapcsolatban problémát okozhat az, ha egy rendszerhívás blokkol (például egy `read()` művelet addig blokkol, amíg nincs meg a szükséges mennyiségű input adat. Ezt a megfelelő fájldeszkriptor nem-blokkolóra állításával lehet kiküszöbölni (persze ezzel a programunkat kicsit bonyolultabb lesz megírni, de lehetővé válik az I/O párhuzamossá tétele, ami miatt a thread-eket gyakran használják).

Egy `fd` fájldeszkriptorra vonatkozó műveleteket egy folyamaton belül a következő függvényhívással tehetjük nem-blokkolóvá:

```
fcntl(fd, F_SETFL, fcntl(fd, F_GETFL, 0) | O_NONBLOCK);
```

Vagyis ezután az `fd` fájldeszkriptorra vonatkozó I/O műveletek csak annyi adatot írnak olvasnak be/írnak ki, amennyinek "hely van", vagyis amennyit várakozás nélkül ki tudnak írni, majd visszatérnek (általában visszaadva azt, hogy mennyi adatot sikerült kiírni). Egy fájldeszkriptorra vonatkozó műveletek blokkolóra visszaállítása a következő függvényhívással végezhető el:

```
fcntl(fd, F_SETFL, fcntl(fd, F_GETFL, 0) & ~O_NONBLOCK);
```

3.11.6 Folyamatok kommunikációja a UNIX-ban

A UNIX operációs rendszernek több elterjedt változata is van: egyik változata az AT&T ill. Novell UNIX változata (a System V Release 4), a másik lényeges változata a Berkeley UNIX (például a 4.3BSD). Vannak más UNIX-szerű rendszerek is, mint például a Linux, amelyről itt nem írok részletesebben, mert – legalábbis számunkra – nincs lényeges eltérés az "igazi" UNIX rendszerek és a Linux között.

A UNIX rendszerek nem tekinthetők osztott rendszereknek, bár a UNIX hálózati lehetőségei nagyon sokrétűek. A transzparencia (például a kommunikáció transzparenciájának) hiánya talán a legfőbb oka annak, hogy a UNIX rendszert általában nem tekintik osztott operációs rendszernek. A UNIX sokféle kommunikációs eszközzel rendelkezik:

- Az SVR4 UNIX tartalmazza a **szemaforokat**, **üzenetcsatornákat**, **osztott memóriát** mint kommunikációs eszközöket, amelyek a Columbus UNIX-ban lettek kifejlesztve (a UNIX egy kutatási változatában), és a System V UNIX-nak szabványos része; a BSD UNIX-nak ezek az eszközök sokáig nem voltak részei, vagyis az ezeket az eszközöket alkalmazó programokat nehezebb volt átvinni a nem System V származék UNIX rendszerekre. Ezeknek az eszközöknek fontos jellemzője, hogy bárki hozzáférhet, aki a hozzájuk tartozó ún. **globális egyedi kulcsot (unique key)** ismerik. Létezik olyan rendszerszolgáltatás, amely létrehoz egy új kommunikációs eszközt (bármelyiket a három közül) egy megadott egyedi kulccsal – ha az adott egyedi kulcshoz még nem hozták létre a kommunikációs eszközt, akkor a rendszer azt létrehozza, egyébként pedig a rendszer hibaüzenetet ad. Továbbá van olyan művelet, amely egy már létrehozott kommunikációs eszközhöz (egyedi kulcsa alapján) hozzáférési lehetőséget ad a folyamatnak. Természetesen vannak olyan műveletek, amelyekkel a kommunikációs eszközön a kommunikációt végzhetjük.

- A Berkeley UNIX kommunikációs eszköze a **socket** rendszer. A socket lényegében egy kommunikációs végpont absztrakciója, a kommunikáció pedig ezeknek az összekapcsolásán keresztül folyhat. Minden socket része valamely **kommunikációs domain**-nek: általában annak, hogy programok kommunikálhassanak, szükséges feltétele, hogy a kommunikációt létesítő (összekapcsolt) socketok azonos domainbe tartozzanak. Minden sockethoz tartozik egy cím (ez a cím lehet egy Internetbeli IP cím vagy egy DECnet cím – vagy akár egy fájlnév – attól függően, hogy a socket Internet, DECnet, vagy UNIX domainben lett létrehozva). Továbbá minden sockethoz tartozik egy karakter-sorozat, amely a kommunikáció során elküldött, de még nem feldolgozott (beolvasott) adatokat tárolja. A socket rendszerben lehetőség van socketok létrehozására, sockethoz cím hozzárendelésére, két azonos domainben levő socket összekapcsolására, és lehetőség van összekapcsolt socketok közt adatátvitelre.

A 4.4BSD UNIX kernelje jeleneg tartalmazza a TCP/IP, az X.25, a Xerox Network System és az OSI TP4 és az OSI CLNP protokollcsaládok implementációját, és az ezekhez való hozzáférést a socket interfészen keresztül teszi lehetővé.

A Berkeley socket rendszer az AT&T TLI-jéhez képest (ld. ezt lejjebb) egyszerűbb, kissé hiányos programozói interfész a hálózati szolgáltatásokhoz, ugyanis tervezésekor még nem voltak kikristályosodva a transzport szolgáltatóktól szélesebb körben elvárt szolgáltatások és egyes szolgáltatások absztrakciója sem volt igazán sikeres (ezzel kapcsolatban gyakran említik a sürgős adattal kapcsolatban biztosított szolgáltatások különbözőségeit is, amit azonban a transzport-protokollok sürgős adat fogalmának definíciójának különbözőségei okozzák). A Berkeley socket rendszer néhány hiányosságára a TLI-ről szóló pont végén fogok utalni.

- Az AT&T UNIX System V Release 3.0 részeként lett ismert a TLI (Transport Layer Interface) mint egy interfész réteg az OSI referenciamodell transzport rétegének szolgáltatásaihoz (a szerkezete, felépítése sokmindenben az ISO Transzport Szolgáltatásának Definícióján alapszik). Terminológiájában a TLI a kommunikációs végpontot (ami általában egy processz) **transzport végpontnak** nevezi, az operációs rendszer által a felhasználói programoknak nyújtott hálózati transzport-protokoll szolgáltatások implementációját pedig **transzport szolgáltatónak** nevezi. Míg a Berkeley socket rendszer a BSD UNIX-ban rendszerhívásokként érhető el, addig a TLI egy könyvtár, amely kapcsolatot teremt a transzport szolgáltató és a transzportvégpont között (a transzport szolgáltató általában STREAMS device driverek és a rájuk épülő STREAMS modulok formájában van a kernelben implementálva), tehát a **TLI nem egy transzport-protokoll implementáció**. Az AT&T UNIX System V Release 4.0-ig (SVR4) a UNIX AT&T változata nem tartalmazott transzport szolgáltatót, majd a SVR4-ben megjelent a DARPA Internet TCP/IP protokollcsaládjának egy implementációja transzport szolgáltatóként.

  A Berkeley socket interfész az ISO Transzport Szolgáltatásának Definíciójánál korábban született, így az ISO Transzport Szolgáltatásainak egyes részei a Berkeley socket rendszeren keresztül nem elérhetők:

  - A Berkeley socket rendszer nem teszi lehetővé az ISO Transzport Protokoll Definícióban szereplő kapcsolatfelvétel elutasítást: egy kapcsolat automatikusan felépül mielőtt a felhasználói program bármit is megtudhatna a kommunikációs partnerről, és csak ezután van lehetőség a kapcsolat lebontására, amiről azonban a kommunikációs partner tudomást szerezhet (de egyes szolgáltatásoknál ezt illene "titkosan" kezelni).
  - A Berkeley socket rendszerben nincs lehetőség alkalmazásoknak transzport-protokolltól független elkészítésére: általában nincs lehetőség a használandó transzport-protokoll futásidőbeni (hordozható módon történő) kiválasztására, vagy legalábbis a rendelkezésre álló lehetőségek sokkal rugalmatlanabbak a TLI-nyújtotta lehetőségeknél.
  - A Berkeley socket rendszer nem teszi lehetővé kapcsolatfelvételkori illetve kapcsolatlezáráskori adatok átvitelét, míg a TLI erre lehetőséget nyújt.

  A TLI ezen kívül biztosítja a *Network Selection Facility* név alatt összefoglalt lehetőségeivel azt, hogy a programokat transzport-protokoll független módon

lehessen elkészíteni, lefordítani, és a használandó transzport-protokollt csak a program futtatásakor kell specifikálni bizonyos UNIX környezetváltozókban. A programok a transzport szolgáltató egyes paramétereit a futásidőben lekérdezhetik és ezzel lehetőségük van[7] alkalmazkodni a helyi transzport szolgáltatóhoz (azaz a transzport-protokoll egyes jellemzőit, például azt, hogy biztosítja-e a sürgős adat továbbítását vagy sem, biztosít-e lehetőséget kapcsolatfelvételkori adatcserére vagy sem, ...)

A UNIX a fenti alapvető kommunikációs eszközöket nyújtja az alkalmazásoknak. Erre számos alkalmazást építettek, mint például a folyamatok távoli elindítását lehetővé tevő `rsh` alkalmazást (ez az ún. távoli shell szolgáltatás), a távoli terminál szolgáltatást nyújtó `rlogin` illetve `telnet` alkalmazást, a hálózati fájlrendszert: az **NFS**-t. Lehetőség van távoli eljáráshívás végzésére is (például a Sun Microsystems által ingyen bárkinek a rendelkezésére bocsátott **RPC** és **XDR** szoftver-csomagokkal), amelynek a lényege az, hogy a hálózati interfészhez (például a socket rendszerhez) a karakter-sorozat átvitelre épülő kapcsolati modell helyett egy eljárás-hívási kapcsolati modellt biztosít a programozónak: azaz a programozó a hálózatban elosztott[8] alkalmazás-komponensek közti kommunikációt nem a komponensek közt létrehozott adatcsatornaként látja, hanem a távoli eljáráshívási szoftver a szokásos eljáráshívások formájában teszi lehetővé az alkalmazás-komponensek kommunikációjának megszervezését, és ezzel leveszi a programozó válláról az alacsonyszintű kommunikációs adatcsatorna kezelésének gondjait (általában az RPC-szoftverek az eljárás-fejek specifikációja alapján legenerálják az alacsonyszintű kommunikációs adatcsatorna kezelését végző eljárásokat).

A UNIX operációs rendszerben számos más eszközt megtalálhatunk együttműködő folyamatok kommunikációjának megoldására: minden UNIX rendszerben megtalálható kommunikációs eszköz a **pipe**. Ez lényegében egy egyirányú adatcsatorna, mely kizárólag közös őstől származó folyamatok közt jöhet létre (egy pipe végén nem kell feltétlenül különböző folyamatoknak lenni – egy pipenak akár mindkét végét is kezelheti egy folyamat (ezzel például megoldható adatok átmentése egy `exec()` rendszerhívást követően az újonnan elindított alkalmazás részére). Megjegyezzük, hogy a UNIX mai változataiban (pl. System V Release 4, 4.4BSD) a pipe már kétirányú (full-duplex) kapcsolatot biztosíthat a folyamatok közt, de még továbbra is lényeges korlátozás az alkalmazhatóságával kapcsolatban az, hogy a kommunikálni szándékozó folyamatoknak egy közös őstől kell származniuk. Erre a problémára a **névvel ellátott pipe-ok** nyújthatnak megoldást: ekkor ugyanis a hagyományos pipe-októl eltérően a kommunikációs csatorna végeihez a kommunikálni kívánó felek nem egy, a processzre nézve lokális, csak leszármazottak felé örökölhető fájldeszkriptoron keresztül férhetnek hozzá, hanem a kommunikációs csatorna végpontjaihoz hozzá lehet rendelni egy fájlrendszerbeli nevet, amit az összes folyamat elérhet függetlenül attól, hogy van-e közös ősuk vagy nincs (itt a fájlrendszerben használatos rwx-bitek segítségével korlátozható a kommunikáló partnerek köre). Fontos megemlíteni, hogy mind a pipe mind pedig a névvel ellátott pipe a UNIX fájlrendszer szokásos műveleteivel kezelhető (többek közt `read()` illetve `write()` rendszerhívások segítségével).

[7] Ez alatt azt kell érteni, hogy akkor van lehetőség a helyi transzport szolgáltatóhoz való alkalmazkodásra, ha a programot olyanra írták, hogy alkalmazkodni tudjon a helyi transzport szolgáltatóhoz.

[8] Az elosztott fogalmat itt nem a korábban már bemutatott **osztott** rendszer osztottságának értelmében használom, hanem egyszerűen az alkalmazás szétdarabolását értem ezalatt.

3.12 Kérdések

1. Próbáljuk ki, hogy mit ír ki a következő program! (Miért érdekes ez?)

```
#include <stdio.h>

main()
{
     int fd1,fd2;
     char c;

     fd1=creat("/tmp/tfil1",0770);
     write(fd1,"alma",4);
     close(fd1);
     fd1=open("/tmp/tfil1",0);
     fd2=dup(fd1);
     read(fd1,&c,1);
     printf("%c",c);
     read(fd2,&c,1);
     printf("%c",c);
     close(fd2);
     close(fd1);
}
```

2. Az `fstat` rendszerhívás nem fölösleges? Minden esetben lehet a `stat` rendszerhívással "helyettesíteni"?

3. Mit csinál a következő program? (Tegyük fel, hogy a C fordítónk nem végez optimalizációt.)

```
#include <signal.h>

int i;

elj1()
{
     i=23;
}

main()
{
     printf("START!\n");
     signal(SIGINT,elj1);
     i=0;
     while (i==0) {
          kill(getpid(),SIGINT);
     }
     printf("STOP!\n");
```

```
}
```

4. Bővítsük a shellünket további UNIX shell metakarakterekkel! (Például a `>>`, `&`, `<`, stb...)

5. Bővítsük a shellünket shell-változók kezelésének lehetőségével.

6. Bővítsük a shellünket egyszerűbb vezérlési szerkezetekkel.

7. Bővítsük a shellünket a pipe lehetőséggel. Gondoljuk meg, hogy a pipe-ba kapcsolt folyamatokat milyen sorrendben kell/érdemes elindítani ahhoz, hogy a szokásos UNIX shellekben megismert módon működjenek.

8. Jellemezzük és hasonlítsuk össze egymással a threadek szinkronizációjára használatos eszközöket!

9. Készítsünk el egy thread-biztos B-fa könyvtárat: egy olyan B-fa adatszerkezetet kezelő könyvtárat, amelynek az eljárásai egyszerre több threadből hívhatóak, és próbáljunk meg minél finomabb szinkronizációs mechanizmust beépíteni, azaz próbáljunk meg minél magasabb fokú párhuzamosságot megengedni.

10. A thread-ekről szóló részben ismertetett thread-köpeny mechanizmus illetve implementáció milyen esetekben nem megfelelő (milyen esetben okozhat holtpontot)?

Fejezet 4

Hálózatok

A hálózat általában több egymáshoz kapcsolt számítógépből áll, amelyek között lehetőség van információcserére és erőforrásmegosztásra. A hálózatba kapcsolt gépeket **host**oknak nevezik, és minden ilyen hostnak van egy egyedi azonosítója (és ezzel együtt egy egyedi címe). Szokás megkülönböztetni a **helyi hálózat**okat (LAN- okat), és a **nagytávolságú hálózat**okat (WAN-okat). A LAN-ok átviteli sebessége több megabit/sec., míg a WAN-oké gyakran csak 9600 bit/sec. Az **internet** (vagy internetwork) több ilyen egymással összekapcsolt LAN-ból illetve WAN-ból áll (és ez nemcsak az **INTERNET**re vonatkozik).

A számítógépek közötti kommunikáció szigorú szabályok (**protokollok**) szerint zajlik, és mivel ezek a protokollok nagyon összetettek, ezért ún. logikai szintekre osztották őket azért, hogy könnyebben lehessen őket kezelni és implementálni. Egy lehetséges szintfelosztás a következő: **fizikai kapcsolat szint**je, **adatkapcsolati szint**, **hálózati szint** és a **transzport szint**.

A fizikai szint specifikálja a fizikai adatátviteli közeg paramétereit: például azt, hogy mekkora lehet az összekapcsolt gépek közt kifeszített drót maximális hossza, mekkora feszültség jelenti a logikai 1-et illetve a logikai 0-át, stb.

Az adatkapcsolati szint specifikálja például az ellenőrzőösszeg kiszámítási módját, az átviendő bitfolyam feldarabolásának (az ún. keretképzés) szabályait és még sok más dolgot. Ez és a fizikai szint egyértelműen meghatározzák az egymással fizikailag (például dróttal) összekapcsolt számítógépek között lezajló kommunikáció szabályait.

A hálózati szint feladata annak a megoldása, hogy a kommunikáció ne csak az egymással közvetlen kapcsolatban levő gépek között folyhasson, hanem két tetszőleges hálózatba kapcsolt host között is, ezért a hálózati réteg (illetve a hálózati protokoll) feladata a csomagoknak a forráshosttól a célhostig történő elirányítása, az útvonalkijelölés (**routing**).

A transzport szint feladata az, hogy adatokat fogadjon a felette levő szoftverrétegtől, kisebb darabokra vágja szét azokat és továbbítsa a hálózati réteg felé. Majd a célállomáson az ottani transzport szint feladata lesz az egyes csomagoknak az eredeti sorrendbe rendezése. A legtöbb hoston egyszerre több folyamat is futhat, ezért szükség lehet több transzport-kapcsolatnak a hálózati rétegre történő a **multiplexelés**ére (illetve demultiplexelésére). Ennek a megoldása is a transzport szintre tartozik.

Egyes szabványok ezeken kívül más szinteket is definiálnak, az OSI például a viszonyszintet, amely a több fél között kialakult kapcsolat szinkronizálására szolgál; az adatábrázolási szintet, amely például a hálózaton keresztülhaladó adatok belső

ábrázolásának, titkosításának és tömörítésének módjával foglalkozik. Ez azért is fontos, mert az ún. **host byte order** gyakran eltér a **network byte order**től. (A network byte order – a hálózati bytesorrendre vonatkozó szabvány – a Motorola proceszorok belső ábrázolási formájával egyezik meg, és nem az Intelével! A reprezentációs problémák azoknál a gépeknél is előjönnek, ahol egy byte nem 8, hanem esetleg 9 vagy 10 bitet tartalmaz (ilyen is van ...). Ezért a legtöbb hálózati terminológiában a bitnél "szélesebb" adategységként nem a byteot, hanem az **oktett**et használják, ami mindenképpen 8 bitből áll, és az egyes hostok feladata a "helyi" byte<-->oktett transzformáció megoldása.)

Fontos, hogy ezek a protokollok azt nem specifikálják, hogy a programok ezeket a szolgáltatásokat hogyan, milyen **programozói interfész**en keresztül vehetik igénybe. Ez általában nagyon operációs rendszertől függő dolog: a UNIX rendszerekben ilyen interfészek a **Berkeley socket interface** és a **TLI** (Transport Layer Interface, szabványosított változatának neve XTI). A BSD UNIX nagy elterjedtsége miatt talán a socket interfész az elterjedtebb. Az AT&T UNIX alatt is létezik socket interfészt biztosító rendszerkönyvtár.

A kommunikáció általában úgy megy, hogy a felhasználói program átadja az átviendő adatokat a transzport-szintnek, az feldarabolja az adatokat kisebb csomagokra, kiegészíti egy transzport-fejléccel, és ezt átadja a hálózati szintnek. A hálózati szint további szükséges fejlécekkel egészíti ki azt, és továbbadja az alatta levő szintnek, s.í.t. Az adatok rendeltetési helyén pedig ugyanez történik, csak fordított irányban.

4.1 A hálózati kapcsolat modellje

A hálózati kapcsolat kétféle lehet: **összaköttetés-alapú** vagy **összeköttetés-mentes**. sszeköttetés-alapú kapcsolat esetén egy "virtuális csatorna" keletkezik a kommunikáló felek között, amely véd az adatismétléstől illetve adatvesztéstől. Összeköttetés-mentes kapcsolat esetén a kommunikáció ún. **datagram**ok közvetítésével zajlik, senki sem garantálja azt, hogy egy elküldött datagram megérkezik a céljába, sem pedig azt, hogy ha az elküldött datagramok megérkeznek, akkor csak egy példányban illetve csak az elküldés sorrendjében érkezhetnek meg.

A kommunikáció általában nem szimmetrikus: az egyik, a kezdeményező fél (a **kliens**) a másik féltől (**szervertől**) valamilyen szolgáltatást kér. A hálózati kommunikáció leggyakrabban erre az ún. **kliens-szerver modell**re épül.

Egy szerver szerkezete általában kétféle lehet: **iterativ** vagy **párhuzamos** (ezt nevezik konkurrensnek is). Az iterativ szerver úgy működik, hogy ciklusban fogadja, és kielégíti a hozzákapcsolódott kliensek igényeit. A konkurrens szerver minden egyes kliensével való kapcsolattartáshoz szül egy gyermek-folyamatot, ami a kliens által igényelt szolgáltatásokat elvégzi - majd általában leáll.

4.2 A TCP/IP protokollcsalád

A mai egyik legnagyobb hálózat a DARPA Internet, amely a 70-es évek végén ill. a 80-as évek elején készült el. A DARPA Internetbe kapcsolt gépek egy ún. **TCP/IP protokollcsalád** segítségével kommunikálnak egymással.

4.2.1 A fizikai és az adatkapcsolati szint

Ebben a protokollcsaládban a fizikai és az adatkapcsolati szintet egy ún. **Ethernet hálózati csatlakozó** biztosítja. A hostok között végigmegy egy kábel, és ez az Ethernet egy ún. **CSMA/CD** (Carrier Sense Multipla Access with Collision Detect) technikával biztosítja azt, hogy egyszerre csak egyvalaki használja ezt a kábelt, illetve ha ezt nem sikerül elérnie, akkor az egymással "ütköző" adókat véletlen ideig várakoztatja, majd megpróbálhatnak újraadni. Minden egyes Ethernet csatlakozó-nak van egy 48-bites egyedi címe, és minden egyes Ethernet csatlakozó csak azokat az csomagokat veszi le a kábelről, amelynek ő a címzettje (vagy az üzenet egy ún. **broadcast üzenet** volt, amit mindenkinek meg kell kapnia). Egy host akár több Ethernet csatlakozóval is rendelkezhet, amelyek más-más LAN-okon vannak (**multihomed host**). Ekkor ez a host képes lesz routing-feladatokat is ellátni: például az egyik hálózatról üzeneteket átküldeni a másikra, ha szükséges (az ilyen hostokat **gateway** gépeknek is nevezik).

Megjegyzés: A TCP/IP protokollok felsőbb szintjei nagymértékben kihasználják azt, hogy alattuk egy "üzenetszórásos" (azaz broadcast-re is képes tudó) hálózati csatlakozó van, de megoldottnak tekinthető a TCP/IP protokolloknak soros (mondjuk egy szabványos RS-232) vonalon keresztüli használata (az IBM PC-ken a TCP/IP megy ArcNET hálózati interface felett is).

További megjegyzés: röviden ismertetem néhány hálózati csatlakozó eszköz adatátviteli sebességét (mivel nincs mindegyikről részletes információm, ezért a részletesebb ismertetéstől inkább eltekintek). Ethernet kábelként (az épületet behálózó gerincvezetékként) napjainkban kétféle technológiát használnak: egyrészt a vékony Ethernet kábelt, másrészt a vastag Ethernet kábelt. Mind a vékony mind pedig a vastag Ethernet kábeleken megvalósítható egy 10 Mbit/sec sebességű adatátvitel (a ma (1996-ban ...) legelterjedtebb hálózati kábelek úgy tudom, hogy még inkább a vékony Ethernet kábelek). A vastag Ethernet kábeleket alkalmazva megfelelő (nagy sorozatban is elérhető) Ethernet kártyával 100 Mbit/sec sebességű adatátvitelt lehet megvalósítani. Helyi hálózatok összekapcsolására egy kézenfekvő fizikai rétegbeli átviteli közeg a telefonvonal (modemeken keresztül). Ezeken ma a digitális telefonközpontokon jól használhatóak a 14400 bit/sec sebességű átvitel megvalósítására képes modemek, esetleg a 28800 bit/sec sebességű modemek (míg a gyenge nem digitális telefonközpontokon – úgy tudom –, hogy a 2800-9600 bit/sec átviteli sebesség az igazán elterjedt). Magyarországon úgy tudom nem használják, de Amerikában egyre inkább terjednek az ún. T1-es telefonvonalak; a transzkontinentális T1 vonalakon jóval nagyobb adatátviteli sebességek érhetők el: kb. 1.5 Mbit/sec.

4.2.2 A hálózati szint (IP)

A TCP/IP hálózati szintű protokollja az **Internet Protocol** (ez az IP), ez végzi az csomagoknak a forráshosttól célhostig irányítását. Minden egyes hostnak (pontosabban minden egyes Ethernet vagy más hálózati csatlakozónak) van egy ún. **Internet címe** (IP-címe), ami lényegében egy 4-byteos szám, és a 4 byteot pontokkal elválasztva decimálisan szokás megadni. Ez a cím teljesen független az Ethernet-címektől, és úgy lett kialakítva, hogy az útvonalkijelölő algoritmust a lehető legegyszerűbben lehessen megoldani. Az IP-cím két részből áll: egy **net-id**ből és egy **host-id**ből. A net-id azonosítja az Internetben az egész LAN-t, míg a host-id a LAN-on belül a hostot azonosítja. Aszerint, hogy a 4

byteból mekkora rész a host-id és mekkora a net-id szokás megkülönböztetni A, B és C osztályú IP- címeket a következőképpen:

```
          0 2 4 6  8                              -adik bit
          ........|........|........|........
Class:A   0 net-id|           host-id
Class:B   10     net-id    |     host-id
Class:C   11         net-id         | host-id
```

Mivel a csupa 0 és a csupa 1 host-id broadcast műveletek számára van fenntartva, ezért egy C osztályú helyi hálózatban maximum 254 host lehet bekötve. (A hostokat nemcsak a fenti 4-byteos címekkel lehet azonosítani, hanem szoktak nekik más - könnyebben megjegyezhető - neveket is adni (például kutya vagy macska vagy tehén vagy valami más emberibbnek nevezett nevet). Ekkor a hálózatban kell lennie egy (vagy több) úgynevezett **nameserver** gépnek, amely az "emberibb" hostnevet 4-byteos Internet címmé alakítja (mivel az operációs rendszer belül csak ezt a formát használja). Ezt a hostnév `-->` IP-cím transzformációt bármelyik programunkban elvégeztethetjük az operációs rendszerrel - erre a későbbiekben még visszatérünk. (A fent említett neveket domain neveknek is szokták nevezni.)

Megjegyezzük, hogy a hálózatba kapcsolt komponensek nevekkel való ellátása az Internetben szépen meg van oldva host szinten (vagyis az egyes hostokat, azok hálózati interfészeit el tudjuk látni egy-egy névvel), de a meglevő névsémába nem igazán illeszthetők be más hálózati erőforrások, és nincs lehetőség absztrakt hálózati objektumok létrehozására (ilyen objektum lehetnek például a felhasználói csoportok, vagy akár felmerülhet az egyes gépeken futó folyamatok címezhetőségének a kérdése is). Ezeknek a problémáknak – úgy tűnik – egy jobb megoldását kínálja a CCITT X.500 szabványa: itt abból indulnak ki, hogy minden objektumnak vannak bizonyos **attributumai**, és az objektum azonosításakor az attributumait kell megadni (például ilyen tekintetben az `ullman.inf.elte.hu` számítógép `csb` és `c` nevű felhasználói által alkotott `admin` nevű csoportjára az `/ORSZÁG=hu/INTÉZMÉNY=elte/SZERVEGYSÉG=informatika/SZERVER=ullman/CSOPORT=admin` módon hivatkozhatnánk, magára a számítógépre pedig az `/ORSZÁG=hu/INTÉZMÉNY=elte/SZERVEGYSÉG=informatika/SZERVER=ullman` módon hivatkozhatnánk. Ennek a módszernek az a szépsége, hogy mindenfajta objektumot be tudunk illeszteni ebbe a jelölésrendszerbe (pontosabban: leírásrendszerbe).

Az IP-routing kissé leegyszerűsítve a következőképpen megy: adott egy forrás-IP-cím és egy cél-IP-cím. Ha a két cím net-id része megegyezik, akkor a két állomás egy hálózaton van (ha nincsenek ún. **subnet**ek ... mert ha vannak, akkor a host-idből még néhány bit a net-idhez lesz kapcsolva, és úgy lesz az egyenlőség vizsgálva), és ekkor az ún. **ARP** protokollal a küldő állomás meghatározza a célállomás IP- címe alapján annak az Ethernet-kártya címét, és oda küldi a csomagot. Ha a két net-id különbözik, akkor a forráshostból a hálózatba kapcsolt ún. **default gateway**nek lesz elküldve a csomag, aki majd biztosan továbbítani tudja a rendeltetési címre. Az ARP-s routingot **direkt**, míg az utóbbi default gatewayeset **indirekt routing**nak nevezik. (Az INTERNET hálózat központjában van egy úgynevezett Core Gateway System, amely nagyobb

kapacitású gateway gépekből áll, amelyekben a routingot más gatewayek közötti protokollokkal és algoritmusokkal kiegészítve is végzhetik.) Az IP-routing egy speciális lehetősége az ún. **source routing**, az IP-csomag küldője által történő útvonalkijelölés: ilyenkor az IP-csomagot kiegészítik olyan opciókkal, amiben fel van sorolva az, hogy az IP-csomagot milyen routereken keresztül milyen sorrendben merre kell a cél-címéhez továbbítani (vagyis ekkor az útvonalkijelölés nem a default gateway ismeretei alapján történik).

Az IP-routingot kissé megnehezítik a **mobil számítógépek** elterjedése. Ennek a problémának a megoldására fejlesztették ki az **MIP** (Mobile IP Extension) protokollt. Eszerint minden egyes mobil számítógépnek van egy állandó IP-címe, ez alapján egy "anya-hálózata" (ez az állandó IP-címének a net-id része alapján meghatározható); de egy mobil számítógép nem csak az "anya-hálózaton" keresztül kapcsolódhat az Internetre, hanem bármelyik ún. IAP-n (Internet Access Point, Internet Hozzáférési Pont) keresztül (egy mobil számítógép térbeli mozgásával – "vándorlásával" – természetesen változhat az az IAP, amelyen keresztül belép az Internetbe). A MIP protokoll lényegében arra épül, hogy az "anya-hálózat" routerei ismerik a mobil gépek aktuális IAP-jeit, így a mobil gépeknek küldött IP-csomagokat tudják továbbítani nekik az IAP-n keresztül az IP protokoll source-routing lehetőségének segítségével. Természetesen a mobil gépekről küldött csomagok továbbíthatók az IAP-felé mint default router felé, és onnan a "hagyományos" IP-routing segítségével eljuttathatóak a rendeltetési helyükre.

Az **ARP protokoll** a következő: kíváncsiak vagyunk egy adott IP- című gép Ethernet címére. Ehhez úgy kell egy ARP-csomagot kiküldeni, hogy azt miden egyes rendszer megkapja (**broadcast** segítségével). Minden egyes host, amely egy ARP csomagot beolvas a hálózatról beolvas, ellenőrzi, hogy nem az ő IP-címe van-e benne, arra kíváncsi-e az ARP csomag küldője. Ha az van benne, akkor kiküld egy válasz-csomagot a kérdezőnek, amely tartalmazza az Ethernet-címét - ez megoldható, mivel nyilván a saját Ethernet címét minden egyes host le tudja valahogyan kérdezni.

Soros vonalon az Internet Protokoll egy speciális változatát, az **SLIP**-t (Serial Line Internet Protocol) használják. Az ilyen jellegű hardver eszközök nem támogatják az ARP-t (nem is lenne sok értelme, mert egy drótnak csak két vége van), ezért az operációs rendszerbe az SLIP-kapcsolatok végpontjairól az információkat "kézzel" kell bekonfigurálni (vagyis azt, hogy melyik soros vonalon milyen IP-című host érhető el).

Nyilvános hálózatokban gyakran használt hálózati protokoll az X25.

4.2.3 A transzport szint

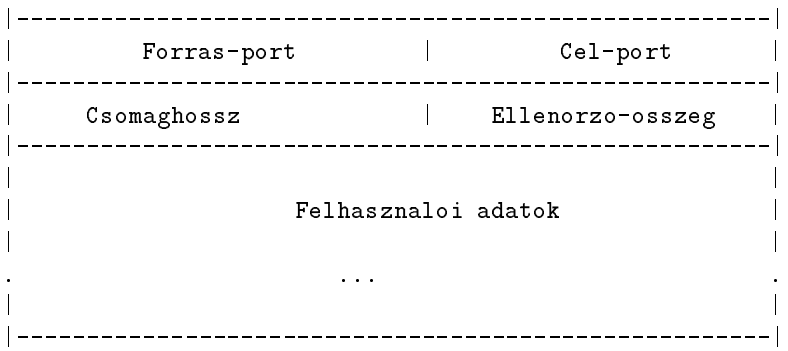
A TCP/IP protokollcsalád két transzport-szintű protokollja a **TCP** (Transmission Control Protocol) és az **UDP** (User Datagram Protocol). A TCP öszeköttetés-alapú, míg az UDP nem az.

Az UDP protokoll

Az UDP által nyújtott szolgáltatások lényegében megegyeznek az IP által biztosított szolgáltatásokkal. Egy lényeges bővítés az IP-hez képest az, hogy az UDP több kommunikációs **port**ot (TSAP-ot, Transport Service Access Point-ot) biztosít, ahonnan ill. ahova csomagokat lehet küldeni, és ha egy program UDP-n keresztül akar kommunikálni

másokkal, akkor az UDP csomagok elküldésekor meg kell adni a cél-host IP-címe mellett annak az ottani UDP-portnak a sorszámát, ahova a csomagot küldeni akarja (az UDP fejléc tartalmazza annak a helyi UDP-portnak a sorszámát is, ahonnan a csomagot küldték, így a célállomás azt kiolvasva tudja, hogy honnan küldték a csomagot, és ez alapján tudhatja, hogy hova küldjön egy esetleges választ). Ezzel – mármint az UDP portokkal – lehetőség van egy hoston egyszerre több UDP-kapcsolat létrehozására is, és nincsenek olyan megkötések, hogy egy folyamat egyszerre csak egy darab UDP kapcsolatot létesíthet (mert az UDP-portok azonosítói és a folyamatok azonosítói egymástól teljesen függetlenek).

Az UDP fejrész a következő mezőkből áll:



Az ellenőrző összeg számítása a következőképpen történik: 2-byteos szavanként kell összeadni az UDP fejrészt, az IP pszeudo-fejrészt, és az elküldendő adatokat, majd az összeg 1-es komplemensét kell képezni. Az említett IP pszeudo-fejrész itt a forrás ill. cél-hoszt IP címéből, valamint a forrás/cél UDP-port sorszámból áll.

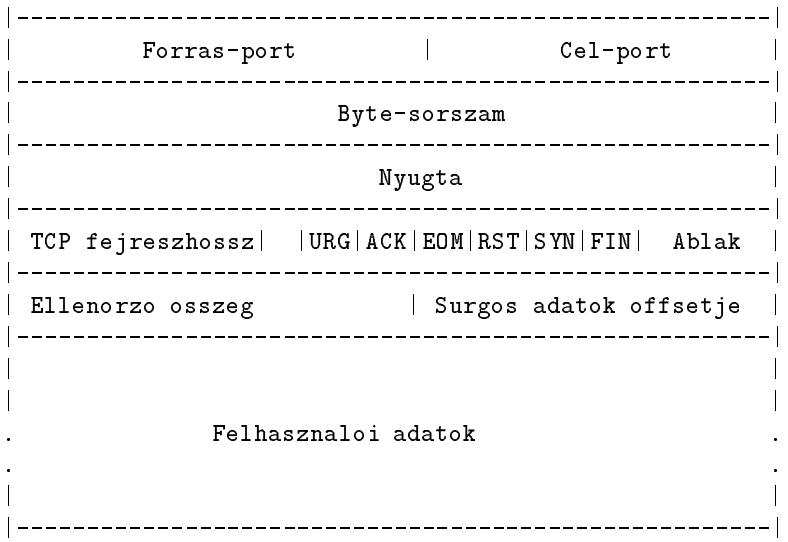
A TCP protokoll

A TCP a protokollcsaládnak talán a leggyakrabban használt kommunikációs transzportprotokollja. (FONTOS: **A TCP egy protokoll, és nem a kommunikációt megvalósító program**!) Ha TCP protokollal küldünk egy byte-folyamot (tetszőleges adatokat), akkor a TCP azt maximum 64 byteos darabokra bontja, és ezeket a darabokat egyenként átadja az IP-nek (természetesen a TCP fejléccel ellátva), hogy küldje el a rendeltetési helyére. Az IP nem garantálja azt, hogy a csomagok a célállomásnál meg is érkeznek, ezért a TCP feladata az, hogy adott esetben (pl. egy bizonyos idő lejártával) az egyes csomagokat újra elküldje, mivel lehet, hogy az előző példány elveszett valahol. A célállomáson a megérkezett csomagok sorrendje nem biztos, hogy az elküldés sorrendjével megegyezik, ezért a TCP feladata ennek a rendezése is (ha szükséges). A TCP természetesen a csomagduplázás ellen is védelmet nyújt.

A TCP protokoll a megbízhatóságot az ún. **PAR** (Positive Acknowledgement with Retransmission) technikával biztosítja. Ez azt jelenti, hogy a célállomás TCP-t megvalósító szoftvere nyugtázza a csomag kézbesítését, miután a hálózati szinttől (az IP-től) megkapta.

Egy hoston egyszerre több TCP kapcsolat is élhet, és itt is, mint az UDP-nél az egyes kapcsolatok külön-külön **TCP-port**on (TSAP-on) vannak. A TCP-kapcsolatok **full-duplex**ek, vagyis kétirányúak, és az elküldött adatokat a TCP strukturálatlan byte-folyamnak tekinti. A TCP-vel ezen kívül lehetőség van **sürgős adat**ok továbbítására is. A protokoll előírja, hogy ha sürgős adatot küldtünk, akkor az adat fogadóját a célhoston erről értesíteni kell, és meg kell adni a lehetőséget a sürgős adat soron kívüli feldolgozására is. Az értesítés módja (mivel oprendszer-függő) nincs a protokoll által specifikálva.

A TCP fejrész a következő mezőkből áll:



(A rajzon a TCP fejrész vízszintes mérete 32 bit.)

A TCP portok 0-tól 1023-ig ún. **foglalt portok**. Ezeknek a kiosztási jogát a DARPA fenntartja magának. Például a távoli bejelentkezés (TELNET) protokoll szervere mindig a 23-as TCP porton vár arra, hogy valaki rákapcsolódjon és bejelentkezzen rajta az adott hostra. Az általában jellemző, hogy a fontosabb, szélesebb körben használt protokollok egy "mindenki által ismert" (**well-known**) sorszámú port-on várnak kapcsolatokra.

A TCP fejrész bitjei a következők:

- URG: Ennek a bitnek 1 az értéke, ha a sürgős adatok offsetje fejrészmező ki van töltve (és érvényes ...). Ez az offset azt mutatja majd meg, hogy az aktuálisan átvitt byte után hányadik byte után következik a sürgős adat.
- SYN: Öszeköttetés-felépítéskor fontos
- ACK: "Nyugta" mező érvényes adattal van-e kitöltve
- FIN: A küldőnek nincs több elküldendő adata (ezzel zárul le az egyik irányban a kapcsolat).

- RST: Inkonzisztens állapotba került összeköttetés megszüntetésére vonatkozó kérelem.

- EOM: End Of Message flag (nem kell a TCP kapcsolat használójától adatokra várni (amíg a 64 byteos szokásos TCP üzenethossz összejönne), mert rövid időn belül lehet, hogy nem lesz új elküldendő adat). Ez például a TELNET protokollnál hasznos. Milyen ciki lenne, ha a terminálon beadott karaktereket a terminálunk csak 64 byte beadása után küldené el a távoli hosztra, ahova bejelentkeztünk.

Az ellenőrző összeg számítása a következőképpen megy: 2-byteos szavanként kell összeadni a TCP fejrészt, az IP pszeudo-fejrészt, és az elküldendő adatokat, majd az összeg 1-es komplemensét kell képezni. A byte sorszám megmondja, hogy az egész átküldendő adatfolyamból most épp hányadik byteot küldjük át (a TCP üzenet első bytejának az adatfolyamon belüli sorszáma). A nyugta mező megmondja, hogy a kommunikációs partner az elküldendő adatfolyamunknak hányadik byteját várja (és ez nem lehet és nem is lesz monoton csökkenő!). A TCP ellenőrző összegnél említett IP pszeudo-fejrész a forrás ill. cél-hoszt IP címéből, valamint egyéb protokoll-információkból és az aktuális TCP üzenet hosszából áll. (A byte sorszám kezdőértéke nem minden kapcsolat esetén nulla - a kezdeti értéket a kapcsolat létrehozásakor a két partner egyezteti.)

4.3 TCP/IP konfiguráció

Minden egyes számítógép operációs rendszere biztosít valamilyen lehetőséget a TCP/IP hálózati réteg alapvető paramétereinek (pl. a host IP címe, neve, ...) a beállítására. A legtöbb rendszerben a Berkeley UNIX-ból származó segédprogramok használhatóak ilyen célra, de sok helyen készítettek olyan interaktiv segédprogramokat is, amelyek interaktivan megkérdezik a felhasználót/rendszergazdát a szükséges információról, majd végrehajtják a Berkeley UNIX-ból származó segédrutinokat a megfelelő felparaméterezéssel.

Megjegyezzük, hogy az alább látható példák egyes számítógépeken másként működhetnek; lehet, hogy más számítógépen más formában adják ki az eredményüket. Ennek oka az egyes operációs rendszerek segédprogramjainak a minimális különbözősége lehet. Ha ilyen jellegű problémákkal találkozunk, akkor nézzük meg a megfelelő parancsok referencia kézikönyvbeli leírásukat a parancs pontos működéséről: általában a kiírt angol szövegek megfogalmazása más- és más lehet, de a lényeges információk (pl. a gép IP-címe, Ethernet címe) minden esetben kiírásra kerül.

4.3.1 Hálózati csatlakozók

Egy számítógép – mint már említettük – egyszerre több hálózati csatlakozóval is el lehet látva. Ezeknek a csatlakozóknak a neveit a `netstat` parancs `-i` argumentummal történő végrehajtásával tudhatjuk meg. Erre tekintsük a következő példát:

```
Name    MTU  RX-OK RX-ERR RX-DRP RX-OVR  TX-OK TX-ERR TX-DRP TX-OVR Flags
lo     2000      0      0      0      0    886      2      0      0 BLRU
eth0   1500      0      0      0      0      0      1      0      0 BRU
```

A hálózati csatlakozók neveit az első oszlopból tudhatjuk meg; a többi oszlop a következő információkkal szolgál:

- MTU : A csatlakozón elküldhető adat maximális mérete (a csatlakozó miatt használandó Ethernet vagy más fejlécek nélkül).
- RX-OK : A csatlakozón fogadott hibátlan csomagok száma.
- RX-ERR : A csatlakozón fogadott hibás csomagok száma.
- RX-DRP : A csatlakozón érkezett, de valamilyen oknál fogva nem feldolgozott csomagok száma.
- RX-OVR : A csatlakozón érkezett, de a TCP/IP szoftver lassúsága miatt nem fogadott szoftver.
- TX-OK : A csatlakozón elküldött hibátlan csomagok száma.
- TX-ERR : A csatlakozón elküldött hibás csomagok száma.
- TX-DRP : A csatlakozóra küldött , de valamilyen oknál fogva eldobott csomagok száma.
- TX-OVR : A csatlakozón valamilyen oknál fogva nem elküldöptt csoamgok száma.
- Flags : A csatlakozó állapotát, jellemzői írja le.

4.3.2 IP cím beállítása

Egy hálózati csatlakozó IP-címének a beállítását az `ifconfig` UNIX paranccsal végezhetjük. Ha a számítógépünk két hálózati csatlakozóval van felszerelve: az egyik a loopback (neve: `lo`) csatlakozó, amely a rá küldött csomagokat egyszerűen visszajuttatja a helyi számítógépre (ezzel a helyi hostot címzi meg), a másik pedig egy Ethernet hálózati kártya (neve: `eth0`), akkor azok IP-címeit a következő parancsokkal állíthatjuk be:

```
$ /etc/ifconfig lo 127.0.0.1
$ /etc/ifconfig eth0 157.181.52.1
```

A példában a loopback csatlakozóhoz a `127.0.0.1`, az Ethernet csatlakozóhoz pedig a `157.181.52.1` címet rendeltük.

A számítógépünk egyes hálózati csatlakozóihoz rendelt IP-címeket szintén az `ifconfig` paranccsal kérdezhetjük le. Erre tekintsük a következő példát:

```
$ ifconfig
lo        Link encap Local Loopback
          inet addr 127.0.0.1  Bcast 127.255.255.255  Mask 255.0.0.0
          UP BROADCAST LOOPBACK RUNNING  MTU 2000  Metric 1

eth0      Link encap 10Mbps Ethernet  HWaddr 00:43:23:76:34:47
          inet addr 157.181.53.43  Bcast 157.181.33.255  Mask 255.255.255.0
          UP BROADCAST RUNNING  MTU 1500  Metric 1
```

Itt látható a számítógép mindkét hálózati csatlakozójának a rövid jellemzése: IP-címe, a részhálózaton broadcast célokra használható IP-cím, valamint a network mask konstans értéke, amely a hálózati interfész IP-címéből kijelöli azt, hogy mely bitek tartoznak a korábban már említett net-id-hez (hálózat azonosító). A net-id-et az IP-cím megfelelő bytejainak a mask megfelelő bytejaival bitenkénti VAGY kapcsolatba hozással kaphatjuk meg. Az `eth0` csatlakozónk net-id-je tehát a következő: `157.181.53.0`.

4.3.3 ARP és RARP protokollok

A csomagok Internetbeli útvonalkijelöléséhez használt ARP protokoll implementációjához az `arp` parancs segítségével férhetünk hozzá. Az `arp -a` parancs kiírja a rendszer ARP protokoll táblázatának a tartalát (ez a táblázat szolgál az ARP-vel szerzett információk időleges cache-elésére). Használatára tekintsük a következő példát:

```
$ arp -a
Address                  HW type         HW address
157.181.52.1             10Mbps Ethernet 00:80:73:41:65:56
157.181.52.2             10Mbps Ethernet 00:80:73:41:65:57
```

A fentiekből leolvasható, hogy annak a gépnek, amelyen az `arp -a` parancsot végrehajtották, két másik számítógéppel van – állandó jellegű – kapcsolata, leolvasható ezek IP címe, hálózati csatlakozójuk típusa, Ethernet kártya címei.

Az `arp` paranccsal megtudhatjuk akár egy-egy host Ethernet címét is az IP-címe alapján a host nevét mint argumentumot megadva. Az `ullman.inf.elte.hu` számítógépen kiadva a következő parancsot, megkaphatjuk a `tomx.inf.elte.hu` számítógép Ethernet címét:

```
$ arp tomx.inf.elte.hu
tomx.inf.elte.hu (157.181.52.2) is on 10Mbps Ethernet 00:80:73:41:65:57
```

Ebből megtudhattuk, hogy a `tomx.inf.elte.hu` számítógép Ethernet kártyájának a címe `00:80:73:41:65:57`.

A fordított irányú (Ethernet cím `-->` IP cím) címtranszformációt az RARP protokollal végezhetjük (ezt például X-terminálok használják IP címük meghatározására, ha csak az Ethernet címüket ismerik: az Ethernet címüket a hálózaton egy RARP csomagban broadcast-elik (mindenkinek elküldik), és minden egyes hoston az ott futó `rarpd` démon kikeresi az Ethernet cím alapján a host IP-címét, és visszaküldi az információt az RARP-t kezdeményező X-terminálnak).

Az `rarpd` program a `/etc/ethers` fájlt használja az IP-címek meghatározására, melynek formátuma a következő: soronként tartalmaz egy Ethernet címet és egy IP-címet (hostnevet is tartalmazhat, amely esetben a nameserver segítségével lesz az adott hostnév IP-címmé transzformálva). Tekintsük a következő példát a `/etc/ethers` fájlra:

```
00:80:73:41:34:34 ncd92.inf.elte.hu
00:81:7b:1e:65:23 ncd93.inf.elte.hu
00:80:73:41:8a:44 157.181.52.99
```

4.3.4 Routing táblák

4.4 Kérdések, feladatok

1. Mi a hálózatok fizikai, adatkapcsolati, hálózati és transzport szintjének a feladata?
2. Mit értünk összeköttetés-alapú kapcsolaton?
3. Milyen szerkezetű szervereket ismer?
4. Mi a különbség a TCP és az UDP között?
5. Mi az ARP protokoll szerepe? Hogyan működik?
6. Hogyan történik az IP-routing?
7. Mi az oktett?

Fejezet 5

A Berkeley socketok

A socketok a UNIX-ban a hálózati (és helyi ...) kommunikáció végpontjai, ezeken keresztül lehet a helyi transzport- és más hálózati rétegekhez hozzáférni. Lényegében postaládaként viselkednek: hozzájuk rendelhetünk bizonyos címeket, és fogadhatjuk róluk az oda érkezett adatokat, illetve adatokat küldhetünk el onnan másoknak. A socketokat eredetileg a BSD UNIX-ban készítették el, és az őket manipuláló eljárások ott rendszerhívásokként érhetőek el, de később az AT&T UNIX-hoz is készítettek egy socket-emulációs könyvtárat, amely az AT&T kernel más lehetőségeire épül.

A socket könyvtárak lehetővé teszi hálózati szoftverek készítését úgy, hogy a program készítése során nem a program alatt levő protokoll lehetőségeire kell a figyelmet összpontosítani. A socketok létrehozásakor meg kell adni azt, hogy a socket milyen kommunikációs domainbe tartozik (például TCP/IP Internet vagy X25). Később ha majd a sockethoz "címet" rendelünk, akkor a címet az adott domain-nek megfelelő formátumban kell az operációs rendszerrel közölni. Ezen kívül meg kell adni a kommunikáció módját is (pl. összeköttetés alapú byte-folyam jellegű vagy datagram jellegű ...). Ez alapján a rendszer kiválasztja az adott domain-en belül azt a kommunikációs protokollt, amely az adott módú kommunikáció megvalósítására képes. Ha több protokollt is ismer a rendszer, amely a (domain,mód) párnak megfelel, akkor explicit módon meg lehet adni azt, hogy azok közül melyiket akarom használni (ha ez eltér a rendszerben ismert default értéktől). Például ha az Internet domain-en belül byte-folyam jellegű kommunikációt akarok végezni, akkor a rendszer automatikusan a TCP protokollt választja ki, mert az Internet domain-en belül ez az alapértelmezés szerinti (bytefolyam-jellegű kommunikáció esetén). Ha az Internet domain-en belül datagram jellegű kommunikációt akarok, akkor a rendszer az UDP-t választja ki, mert az Internet domain-en belül mindig az az alapértelmezés szerinti protokoll datagram jellegű kommunikáció esetén (arről még lesz szó részletesebben, hogy mikor mi az alapértelmezés szerinti protokoll ...). Ezután a program visszakap egy egész típusú ún. socket-leírót, amivel a socketra hivatkozhat. Amennyire ez lehetséges, a socketok úgy viselkednek, mint a fájlok, tehát ha van értelme, akkor meg vannak rájuk engedve a szokásos read, write, close, ... fájlműveletek.

5.1 Egy összeköttetés-alapú kliens-szerver kapcsolat menete

Az jellemző, hogy egy kliens illetve szerver összeköttetés-alapú kommunikáció létrehozásakor milyen rendszerhívásokat használ. Ezt foglalja össze a következő két táblázat:

Egy kliens a következő rendszerhívásokkal kommunikálhat a szerverrel:

Socket létrehozása	socket()
Socket címének kijelölése	bind()
Kapcsolatfelvétel a szerverrel	connect()
Adatátvitel	write(), read()
	send(), recv()
Adatátvitel befejezése (EOF)	shutdown()
Socket lezárása	close()

Egy szerver a következő rendszerhívásokkal kommunikálhat a kliensekkel:

Socket létrehozása	socket()
Socket címének kijelölése	bind()
Kliensekre várás állapotba kerülés	listen()
Kliens kapcsolatkérelmének elfogadása	accept()
Adatátvitel	write(), read()
	send(), recv()
Adatátvitel befejezése (EOF)	shutdown()
Socket lezárása	close()

5.2 Egy nem összeköttetés-alapú kliens-szerver kapcsolat menete

Az jellemző, hogy egy kliens illetve szerver nem összeköttetésalapú (...összeköttetésmentes) kommunikáció létrehozásakor milyen rendszerhívásokat használ. Ezt foglalja össze a következő táblázat:

Egy kliens és szerver közti kommunikáció a következő rendszerhívásokkal folyhat le:

Socket létrehozása	socket()
Socket címének kijelölése	bind()
Adatátvitel	sendto(), recvfrom()
Adatátvitel befejezése (EOF)	shutdown()
Socket lezárása	close()

5.3 Socketok címzése az Internet domainben

Az Internet domain socketjaihoz tartozó címek formátumát a következő struktúra szerint kell megadni az operációs rendszernek (deklarálva a `<netinet/in.h>` header fileban van):

```
struct in_addr {
     unsigned long s_addr;  /* 4 byteos IP cim */
};

struct sockaddr_in {
     short  sin_family; /* =AF_INET */
     unsigned short sin_port; /* 16 bites port-sorszam */
     struct in_addr sin_addr; /* IP cim */
     char sin_zero[8]; /* Kihasznalatlan */
};
```

A `sin_family` mező tartalmazza a cím típusát (AF_INET = Address Family in the Internet Domain). A `sin_port` tartalmazza a TCP vagy az UDP port sorszámot (aszerint, hogy a socket milyen típusú: TCP vagy UDP). A `sin_addr` mező tartalmazza a megcímzett objektum IP címét. Ez utóbbi két mezőt az ún. hálózati ábrázolási formában kell megadni (ld. később a `htons`, ill. `htonl` függvényeket).

5.4 Konverzió a hálózati- és host byte-ábrázolásmód között

A hálózatba kapcsolt hostok között van olyan, amelyik a kétbyteos egészeket úgy tárolja a memóriában (ez a host byte order, vagyis a memóriabeli tárolás formátuma), hogy az alacsonyabb helyiértékű byteot a kisebb sorszámú memóriacímen, a magasabb helyiértékű byteot pedig a nagyobbik sorszámú memóriacímen. Vannak olyan hostok is, amelyeknél ez pont fordítva van, és ezért definiáltak egy ún. network byte ordert, amely a Motorola belső tárolási formátumának felel meg, és a hálózati portokat, és címeket ilyen network byte orderben kell megadni a rendszernek. Ehhez azonban a host byte orderről konvertálni kell az adatokat a network byte orderre.

Négy konverziós rutin van, amelyet használhatunk:

- `htons` : host byte order `-->` network byte order (16 bites adat)
- `ntohs` : network byte order `-->` host byte order (16 bites adat)
- `htonl` : host byte order `-->` network byte order (32 bites adat)
- `ntohl` : network byte order `-->` host byte order (32 bites adat)

(Ha numerikus (egész) adatokat viszünk át két host között, akkor is hasznos (és szükséges) ez a konverzió, ha hordozható programot akarunk készíteni.)

5.5 Kommunikációs végpont (socket) létrehozása

Egy socketot a `socket()` rendszerhívással lehet létrehozni. Ennek a rendszerhívásnak három paramétere van: a címformátum (domain neve), a socket típusa és a használandó kommunikációs protokoll. A socket rendszerhívás formátuma a következő:

```
sd=socket(address_family,socket_type,socket_protocol);
```

A következő táblázatban néhány domain-nevet lehet látni (a UNIX C header-fileokban ezek mint numerikus értékű makrók vannak deklarálva):

Address family	Leírás
AF_INET	Internet domain
AF_DECNET	DecNET domain
AF_NS	Xerox Network System domain
AF_UNIX	Lokális hoszton belüli IPC

A socket típusán a socket "minőségét" kell megadni. Ennek értékei a következők lehetnek:

Socket type	Leírás
SOCK_DGRAM	Datagrammok továbbítását támogató socket. összeköttetésmentes kapcsolat kialakítására alkalmas nem garantál megbízhatóságot.
SOCK_STREAM	Megbízható, összeköttetés-alapú, byte-folyam jellegű adatátvitelt támogat. Az elküldött csomagok "egybefolynak".
SOCK_SEQPACKET	Megbízható, összeköttetés-alapú, byte-folyam jellegű adatátvitelt biztosít, de megtartja a csomaghatárokat.
SOCK_RAW	Alacsonyszintű kommunikációs protokollok elérését teszi lehetővé (ld. később).

A sockethoz a fenti típusok alapján ki lesz választva egy alapértelmezés szerinti hálózati protokoll, de ha az nem felel meg az igényeknek, akkor a harmadik paraméterben ezt felülbírálhatjuk. Ha megfelel, akkor ott 0-t adjunk meg.

Az alapértelmezés szerinti hálózati protokollok (socket type és domain alapján) a következők:

	AF_INET	AF_NS
SOCK_STREAM	TCP	SPP
SOCK_DGRAM	UDP	IDP
SOCK_SEQPACKET	–	SPP
SOCK_RAW	IP	IDP

A rendszerhívás visszatérési értéke hiba esetén -1, egyébként pedig egy socket-descriptor, amivel később a socketra hivatkozhatunk.

Megjegyezzük, hogy az IP protokoll eléréséről e fejezet végén még részletesebben is lesz szó, de addig a leggyakrabban használt transzportprotokollokat: a TCP-t és az UDP-t fogjuk áttekinteni.

5.6 Socket címének kijelölése

Egy sockethoz a létrehozása után még nincs semmiféle cím hozzárendelve (egyes domainekben, ilyen például az Internet, hozzárendelődik egy "még nem foglalt" cím, de az nem mindig "a megfelelő cím", ezért gyakran szükséges annak a felülbírálata - például azért, hogy a szerver "well-known" portjának a címét beállítsuk a szerver socket-ján).

Az általában igaz, hogy az operációs rendszer "nem oszt ki" alapértelmezésként beálllított címként 5000-nél nagyobb port-számot, ezért a saját szervereinknek (amelyeknek egy szabad "well known" portot kell kiosztani) 5000-nél nagyobb sorszámú portot nyugodtan kijelölhetünk.

A `bind()` rendszerhívással lehet egy sockethoz egy címet rendelni. Ennek formája a következő:

```
bind(sd, name, namelength);
```

Itt sd a socket-leíró, a name paraméter tartalmazza a pointert a sockethoz rendelendő címre (Internet domainben pointer egy sockaddr_in strukturára), és mivel a címek hossza domainenként más és más, ezért a namelength paraméter tartalmazza a name paraméterben levő cím hosszát byteban mérve.

(Fontos, hogy az Internet domainben egy szerver (mondjuk TCP) sockethoz a hozzábind-olt IP-cím (sin_addr) azt adja meg, hogy a socket melyik hálózati csatlakozón hajlandó elfogadni rákapcsolódási kérelmet. Ott a hálózati csatlakozónak az IP címét network byte orderben kell megadni, és van egy speciális konstans, az INADDR_ANY, amely azt jelenti ha sockethoz bind-oljuk, hogy a socket bármelyik hálózati csatlakozón(!) elfogad rákapcsolódási kérelmet. "Használat előtt" természetesen ezt is network byte orderűre kell konvertálni.)

5.7 Kapcsolat létrehozása

Erre az összeköttetés-alapú klienseknél és szervereknél van szükség. A kapcsolatfelvétel a szerver részéről úgy történik, hogy a szerver a socketján végrehajtja a `listen()` rendszerhívást, és ezzel bejelenti az operációs rendszernek, hogy a socketja kész a kliensek rákapcsolódási kérelmeinek fogadására. Az első paraméter itt is a socket-descriptor, míg a második paraméter egy egész szám, és azt mondja meg, hogy ha egyszerre több kliens is rá akarna kapcsolódni a szerver socketra, akkor hány rákapcsolódási kérelmet rakjon bele egy várakozási sorba (a többit "eldobja", visszautasítja). Ennek értéke általában (alapértelmezés szerint) 5. Ezután a szerver az `accept()` rendszerhívással vehet le a fenti várakozási sorból (ciklusban ...) egy-egy kérelmet és kapcsolódhat a klienshez. (Ha nincs ilyen kérelem, akkor a rendszerhívás vár, amíg lesz egy.) Az accept rendszerhívás első paramétere egy socket-descriptor, második ill. harmadik paramétere pedig egy socket-cím struktúra (cím szerint átadva ...) ill. egy cím-hosszat tartalmazó egész számra mutató pointer, ahova a távoli kommunikációs partner címének a hosszát adja vissza a rendszer. A második paraméterben megadott címre visszatéréskor a rákapcsolt kliens címét írja vissza. A rendszerhívás visszatérési értéke egy új socket-descriptor, amivel a kapcsolatra hivatkozni lehet.

A kapcsolat létrehozása a klienseknél a `connect()` rendszerhívással zajlik. Ennek első paramétere a socket-descriptor, amin keresztül rá akarunk valahova kapcsolódni.

Második paraméter egy socket-cím struktúra, ami a szerver címét tartalmazza. Harmadik paramétere pedig az előbbi struktúra hosszát adja meg. Visszatérési értéke -1, ha a rendszerhívás sikertelen volt.

A rendszerhívások alakja tehát a következő:

```
listen(sd, nrofreq);

newsd=accept(sd, name, namelength);

connect(sd, name, namelength);
```

Ezután megkezdődhet az adatátvitel.

5.8 Adatátvitel összeköttetés-alapú kapcsolatok esetén

Az adatátvitel többféleképpen is mehet: mehet a `read()` ill. `write()` vagy a `send()` és `recv()` rendszerhívásokkal.

A `write()` rendszerhívás paraméterezése a következő:

```
write(sd, buff, size);
```

Ez a `buff` bufferből `size` darab byteot elküld az `sd` socket-descriptorhoz tartozó (hálózati vagy más) kapcsolatra.

A `read()` rendszerhívás paraméterezése a következő:

```
read(sd, buff, size);
```

Ez a `buff` bufferbe `size` darab byteot beolvas az `sd` socket-descriptorhoz tartozó (hálózati vagy más) kapcsolatról.

Mindkettő rendszerhívás visszatérési értéke hiba esetén -1, egyébként pedig az átvitt byteok mennyisége (vigyázni kell! lehet, hogy `size`-nél kisebb!). Ha a távoli gép a hálózati kapcsolatot lezárta, akkor a read rendszerhívás visszatérési értéke 0.

A `send()` rendszerhívás paraméterezése a következő:

```
send(sd, buff, size, flags);
```

Ez a `buff` bufferből `size` darab byteot elküld az `sd` socket-descriptorhoz tartozó (hálózati vagy más) kapcsolatra.

A `recv()` rendszerhívás paraméterezése a következő:

```
recv(sd, buff, size, flags);
```

Ez a `buff` bufferbe `size` byteot beolvas az `sd` socket-descriptorhoz tartozó (hálózati vagy más) kapcsolatról.

A fenti két rendszerhívásnál ha a `flags` paraméter 0, akkor ugyanúgy viselkednek, mint a read illetve write rendszerhívások. Ezen kívül más értékekeit is felvehet, mint például az MSG_OOB-t, ami azt jelenti, hogy a protokoll által definiált sürgős adatként kell az elküldött byteokat kezelni. Másik speciális `flag` az MSG_PEEK, amely a recv rendszerhívásnál adható át, és az eredménye az, hogy az adatokat bemásolja a rendszer a megadott bufferbe, de az eredeti helyükön is meghagyja. Mindegyik rendszerhívás az átvitt (beolvasott ill. kiírt) adatbyteok számát adja vissza.

5.9 Adatátvitel nem öszeköttetés-alapú kapcsolatok esetén

Erre a következő két rendszerhívást használhatjuk:

```
sendto(sd, buff, size, flags, to, addrlen);
/*
 * struct sockaddr *to;
 * int addrlen;
 */

recvfrom(sd, buff, size, flags, from, addrlen);
/*
 * struct sockaddr *from;
 * int *addrlen;
 */
```

Ezeknél a rendszerhívásoknál az első négy paraméter nem szorul magyarázatra. Az ötödik paraméter a sendto rendszerhívás esetén az adat rendeltetési helyének a címe, míg a recvfrom rendszerhívás esetén ott adja vissza az operációs rendszer, hogy honnan érkezett az adat. Az `addrlen` paraméter a `from` ill. a `to` paraméterben megadott cím méretét tartalmazza!

5.10 Kapcsolat (socket) lezárása

Ha egy socketot nem akarjuk tovább használni (nincs szükség az onan jövő adatokra), akkor le kell zárni. Erre a szokásos close rendszerhívást használhatjuk. Alakja:

```
close(sd);
```

Öszeköttetés-alapú socketoknál lehetőség van arra is, hogy csak az egyikirányú adatáramot zárjuk le (ekkor a socket nem szűnik meg!). Ez a shutdown rendszerhívással megy.

Ennek alakja:

```
shutdown(sd, mode);
```

Itt `sd` a socket-descriptor, mode pedig 0, ha nem akarunk több adatot beolvasni a socketról, 1 pedig akkor, ha nem akarunk több adatot írni a socketra.

5.11 Több socket párhuzamos figyelése (select)

A select rendszerhívással lehetőség van párhuzamosan több socket állapotának a figyelésére is (hogy érkezett-e rá adat vagy sem stb.). A rendszerhívás alakja:

```
nrofsdsfound=select(nsds, readsds, writesds, exceptsds, tmout);
```

Az `nsds` paraméter megmondja, hogy a 0-tól nsds-1 -ig terjedő socket- (ill. fájl-)descriptorokat kell figyelnie a programnak. A `readsds` egy pointer, és azok a bitek vannak beállítva az általa mutatott helyen, amelyeket a 0..nsds-1 descriptor-intervallumból olyan szempontból kell figyelni, hogy adat érkezett rá valahonnan. A `writesds` ehhez hasonló, de a rendszer ennél azt figyeli, hogy kész van-e a socket adat küldésére. Az `exceptsds` hasonlóan adja meg, hogy mely descriptorokat kell figyelni "kivételes események" szempontjából (pl. sürgős adat érkezése). A `tmout` paraméter egy pointer, és ha NULL, akkor a rendszer addig vár, amíg valamely kívánt esemény bekövetkezik; egyébként pedig a pointer által mutatott címen levő `tmout` strukturában megadott másodperc ill. microsec. ideig vár valamely eseményre, és ha semmi sem történik addig, akkor TIMEOUT hibával tér vissza. Megjegyzeeük, hogy az, hogy egy socket kész az adatküldésre csak annyit jelent, hogy a rá vonatkozó write/send rendszerhívások végrehajtásukkor nem blokkolnak, és nem jelenti – például egy esetleges hálózati hiba bekövetkezte után – azt, hogy a hálózati kapcsolat már helyesen működik, és az adat a kommunikációs kapcsolat egyik végéről a másikra átküldhető.

A C könyvtár szabványos makrokat bocsát a felhasználó rendelkezésére, amelyekkel a `readsds/writesds/exceptsds` mezőket a socket-descriptorok alapján könnyű kitölteni (ezeket a makrokat használjuk, mert lehet, hogy később a belső reprezentáció meg fog változni).

Ezzel a rendszerhívással lehet microsec. pontosságú órát a programba építeni.

5.12 A kommunikációs partner címének megszerzése

A folyamatok a szülőjüktől gyakran örökölnek megnyitott socketokat. Ha egy folyamatnak szüksége van annak meghatározására, hogy ki van a socket-kapcsolat másik végén (öszeköttetés-alapú kapcsolatok esetén), akkor azt megteheti a getpeername rendszerhívással (ez egyes rendszerekben nem rendszerhívás, hanem könyvtári függvény, amit valamilyen más rendszerhívással valósítanak meg, de ez most számunkra nem érdekes). Ennek alakja a következő:

```
getpeername(sd, name, namelength);
```

A `name` paraméter egy socket-cím struktúra, itt kapom vissza a partner címét. A struktúra harmadik eleme egy pointer egy egész típusú értékre, amely a cím hosszát tartalmazza (visszatéréskor, és ez természetesen a visszaadott cím hosszára vonatkozik).

A sockethoz (explicit módon vagy defaultként) rendelt helyi címet a getsockname rendszerhívással lehet lekérdezni. Paraméterezése ugyanaz, mint a getpeername függvényé.

5.13 Hálózatokkal kapcsolatos könyvtári segédfüggvények

Ebben a részben olyan hasznos könyvtári rutinok lesznek ismertetve, amelyre bonyolultabb, felhasználóbarátabb programok írásánál szükség lehet.

5.13.1 Hostnévről IP-címre transzformáció

Sok programban szükség van a fent említett transzformációra. Ez a `gethostbyname()` könyvtári rutinnal megy. Paramétere a kérdéses host neve, és visszaad egy pointert, ami a következő strukturára mutat (egy gethostbyname számára statikus adatterületen!) :

```
struct hostent {
     char *h_name;   /* A host hivatalos neve */
     char **aliases; /* Alias-nevek tombje */
     int h_addrtype; /* Pl. AF_INET ... */
     int h_length;   /* A host cimenek a hossza */
     char *h_addr_list; /* A host cime(i), NULL-pointerrel
                           jelezve a lista veget. */
};
```

A rutin használatára majd láthatunk példákat a későbbiekben.

5.13.2 Hálózati szolgáltatások adatbázisa

A szabványos szerverek (mint például az FTP vagy a TELNET) a hálózatban minden hoston a megfelelő "jól ismert" porton várakoznak a kliensek rákapcsolódására. A "jól ismert" port sorszámát általában nem égetik bele a szerverbe, hanem egy külső adatbázisban tárolják (a `/etc/services` fileban), a programok onnan kérdezhetik le. Az adatbázis lekérdezését egy `getservbyname()` könyvtári függvénnyel lehet elvégezni. Ez egy servent strukturára mutató pointerrel tér vissza. A struktura szerkezete a következő:

```
struct servent {
     char *s_name; /* A szolgaltatas hivatalos neve */
     char **s_aliases; /* A szolgaltatas egyeb hasznalt nevei */
     int s_port; /* A port sorszama, ahol a szervernek a
                    a kliensekre kell varnia. Network byte
                    orderben */
     char *s_proto; /* A hasznalando protokoll */
};
```

Ha például a TCP alapú FTP protokollt akarjuk használni, akkor a következőképp kell a fent említett rutint használni:

```
struct servent *sp;
struct sockaddr_in serv_addr;

...

/* serv_addr struktura kitoltese */
sp=getservbyname("ftp","tcp");
serv_addr.sin_port=sp->s_port;
/* ... */
```

5.14 A socketokkal kapcsolatos további rendszerhívások

Ebben a részben olyan socketokkal kapcsolatos dolgok lesznek ismertetve, amelyek nélkül meg lehet ugyan élni, de "igényesebb" programokat nem lehet ezek nélkül megírni.

5.14.1 TCP sürgős adat továbbítása

Mint már említve volt, a TCP lehetőséget ad sürgős adatok (TCP urgent data) küldésére is. A socket interface "generikus" ilyen irányú megtervezése (vagyis az Out of Band data kezelése) nagyon nehéz volt, mivel az egyes protokollok más-más dolgokat engednek meg Out of Band data-nak. A TCP protokoll egyszerre akár több, és akármilyen hosszú üzenetet tud továbbítani, míg mondjuk az XNS hálózat egy időben legfeljebb 1 bytenyi sürgős adatot (Out of Band data-t) tud kezelni (ha csak ezt az XNS lehetőséget használjuk ki a programunkban, akkor könnyen átvihető lesz TCP-re, míg fordítva ez nem szokott menni). Még az is bonyolítja a helyzetet, hogy a TCP már azelőtt jelezni képes a kommunikációs partnernek a sürgős adat létezését, még mielőtt maguknak az adatoknak az átvitele megtörténne.

A sürgős (OOB) adatok elküldése és fogadása annyiból áll, hogy a `send()` ill. `recv()` rendszerhívásoknál megadjuk az MSG_OOB konstans flaget. Az OOB adatokat a többi "normál" adattól teljesen függetlenül lehet beolvasni (mintha egy külön kommunikációs csatornán kapnánk őket), és az

```
ioctl(sock, SIOCATMARK, &answer);
```

`ioctl()` rendszerhívással kérdezhetjük meg az operációs rendszertől, hogy a "normál" adatfolyamnak ezen a pontján küldték-e az OOB adatot (eszerint a feltétel szerint lesz az `answer==1` állítás igaz vagy hamis, azaz az `answer` változó értéke 1 lesz, ha az adatfolyamnak ezen a pontján küldték az OOB adatot). Lásd erre a következő példát!

```
sigurg_signal_handler()
{
  int atmark, buf[1024];

  while(1) {
    if (ioctl(sock, SIOCATMARK, &atmark)<0) {
      perror("ioctl");
      exit(2);
      }
    if (atmark) break;
    read(sock, buf,1024);
  }
  if (recv(sock, &atmark, 1, MSG_OOB) <0) {
    perror("recv");
    exit(1);
    }
  }
}
```

A fenti programrész a SIGURG signal (a SIGURG signal azt jelzi, hogy sürgős adatot küldött a kommunikációs partner, de nem biztos, hogy azt már meg is kaptuk ...) kezelését végzi. Beolvassa (és "eldobja") a normál adatokat egész addig, amíg a sürgős adat elküldési helyéig el nem jut, és ott beolvas 1 bytenyi sürgős adatot. (A SIGURG signalról ld. a következő pontot.)

5.14.2 A socketokhoz kapcsolódó SIGIO és SIGURG signalok

Az eddigiekben csak a socketok "szinkron" módú kezeléséről volt szó (vagyis amikor a rendszerhívások csak azután fejeződnek be, miután az operációs rendszer a műveletet teljesen elvégezte). Lehetőség van bizonyos eseményeknek az "aszinkron" módú kezelésére is. Ezzel kapcsolatosak az operációs rendszer SIGIO és SIGURG nevű signaljai.

A rendszert "meg lehet kérni" a korábban már bemutatott `signal()` rendszerhívással, hogy olyan esetekben küldjön egy SIGIO signalt valamelyik folyamatnak, amikor valamelyik socketon (beolvasandó) adat érkezett. Hasonló módon kérhető a "sürgős adat érkezését" jelző signal is. Ehhez először a signal() rendszerhívással a kívánt signal-handler eljárást ki kell jelölni, majd meg kell adni annak a folyamatnak az azonosítóját (`fcntl(socketdescriptor, F_SETOWN, procid)`) rendszerhívással, amelynek a signalt küldeni kell; és végül "engedélyezni" kell a signal-küldést egy (`fcntl(socketdescriptor, F_SETFL, FASYNC)`) rendszerhívással (vagyis ezzel azt mondjuk meg az operációs rendszernek, hogy a signal-handler "kész" a signalok fogadására).

Erre példa a következő programrészlet (itt mindkét signal-handlert kijelöltük, de ez nem "kötelező" - a saját programunkba csak azt kell belerakni, amelyre szükségünk van).

```
#include <fcntl.h>
#include <signal.h>
#include <errno.h>
int iosignalhandler();
int urgsignalhandler();

int sd;

..

sd = socket( ... );

..

signal(SIGIO,iosignalhandler);
signal(SIGURG,urgsignalhandler);

if (fcntl(sd,F_SETOWN,getpid())<0) { /* Mi kapjuk a signalt */
     perror("hiba az 1. fcntl-nel");
     exit(1);
}
```

```
/* Ezutan  kapunk majd signalokat - ha valami erkezik */

if (fcntl(sd,F_SETFL,FASYNC)<0) {
     perror("hiba a 2. fcntl-nel");
     exit(1);
}
```

5.14.3 UDP broadcast lehetőség

Lehetőség van arra is, hogy a (helyi) Ethernet hálózatra kapcsolt (vagyis a kábelre fizikailag rákötött) hostok mindegyikének "egy művelettel" (egyszerre) küldjünk valamilyen üzenetet (magyarán broadcast üzenetet küldjünk).

Ezt a következő socket-opció beállításával lehet elérni:

```
if ((setsockopt(s,SOL_SOCKET,SO_BROADCAST,&on,sizeof(on))<0) {
     /*
      * hiba a setsockopt-nal ...
      */
      perror("hiba a setsockpt-nal");
      ....
}
```

Ezután a sockethoz kell rendelni egy címet (AF_INET címformátummal, INADDR_ANY címmel, és annak az UDP portnak a sorszámát kell hozzárendelni, amelyre a broadcast üzenetet a többi gépen küldeni akarjuk.)

5.14.4 Socket aszinkron üzemmódra állítása

Az `accept()`, `connect()` és `write()` rendszerhívásoknak esetenként várniuk kell, amíg valamekkora memória felszabadul. Ez azt jelenti, hogy ha egy folyamat a fenti rendszerhívások közül végrehajt egyet, és épp nincs elég szabad memória (egy speciálisan erre a célra kijelölt kernel-területen), akkor a folyamat nem fut tovább, amíg a szükséges mennyiségű memória fel nem szabadul. Ezt meg lehet kerülni a következő példában bemutatott rendszerhívással:

```
#include <fcntl.h>

..

sd=socket( ... );

..

if (fcntl(sd,F_SETFL,FNDELAY) < 0) {
     perror("hiba az fcntl-nel");
     exit(1);
}
```

Ezután a fent említett rendszerhívások közül az `accept()` és a `connect()` negatív (vagyis hibát jelző) visszatérési értékkel tér vissza, és az `errno` változóba EWOULDBLOCK hibakód kerül, ha a rendszerhívás végrehajtásához a rendszernek várnia kellene. Ekkor a rendszerhívás nem hajtódik végre. Az adatküldő (pl. `write()` és `send()`) rendszerhívások (amelyek visszatérési értékként az elküldött adatbyteok számát adják vissza) ha nincs elég memória, akkor (esetleg) kevesebb byteot küldenek el, és ezt adják vissza.

5.15 Példák a socket rendszer használatára

A következő pontok egy-egy példán keresztül mutatják be a leggyakoribb kliens és szerver feladatokat UNIX környezetben. Természetesen a valós ÉLETBEN más kliens és szerver strukturák is léteznek, de itt nincs lehetőség mindent bemutatni.

5.15.1 Példa egy egyszerű iteratív összeköttetés-alapú szerverre

A következő program lefoglal egy TCP portot, kiírja annak a címét a képernyőre, és végtelen ciklusban vár a TCP-porton egy kapcsolatra, és kiszolgálja a rákapcsolódott klienst (a szolgálat annyi, hogy a kliens által elküldött byteokat a szabványos kimenetre kiírja).

```
/*
 * Pelda arra, hogy hogyan mukodik egy (iterativ) szerver.
 * A program vegtelen ciklusban figyel egy adott TCP portot, beolvassa
 * es a kepernyore irja az onnan jovo byteokat, majd uj kapcsolatra
 * var.
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>    /* sockaddr_in struktura */
#include <netdb.h>         /* /etc/hosts tabla */
#include <stdio.h>
#define TRUE 1

main()
{
        int sock, length;
        struct sockaddr_in server;
        int msgsock;
        char buf[1024];
        int retval;
        int i;

        sock = socket(AF_INET, SOCK_STREAM, 0);
        if (sock < 0) {
```

```
                perror("hiba a socket-nel");
                exit(1);
        }
        server.sin_family = AF_INET;
        server.sin_addr.s_addr = htonl(INADDR_ANY);
        server.sin_port = htons(0);
        if (bind(sock, &server, sizeof(server))) {
                perror("hiba a bind-nel");
                exit(1);
        }
        length = sizeof(server);
        if (getsockname(sock, &server, &length)) {
                perror("hiba a getsockname-nel");
                exit(1);
        }
        fprintf(stderr,"TCP port:%d\n", ntohs(server.sin_port));

        listen(sock, 5);
        while(1) {
                msgsock = accept(sock, 0, 0);
                if (msgsock == -1)
                        perror("hiba az accept-nel");
                else do {
                        bzero(buf, sizeof(buf));
                        if ((retval = read(msgsock, buf, 1024)) < 0)
                                perror("hiba a read-nel");
                        i = 0;
                        if (retval == 0)
                                fprintf(stderr,"Kapcsolat lezarva\n");
                        else
                                fprintf(stderr,"String:%s\n", buf);
                } while (retval != 0);
                close(msgsock);
        };
        close(sock); /* Sosem sullyedunk idaig */
}
```

5.15.2 Példa egy összeköttetés-alapú kliensre

A következő program a megadott host megadott TCP portjával felépít egy kapcsolatot, és adatokat küld át oda.

```
/*
 * Hasznalata: programnev hostnev TCPport-number
 *
 * A program letrehoz a megadott hoston, a megadott TCP porttal
 * egy kapcsolatot. Rair egy uzenetet es megall.
```

```
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define DATA "Isten hozta ornagyur."

main(argc, argv)
        int argc;
        char *argv[];
{
        int sock;
        struct sockaddr_in server;
        struct hostent *hp, *gethostbyname();
        char buf[1024];

        sock = socket(AF_INET, SOCK_STREAM, 0);
        if (sock < 0) {
                perror("hiba a socket-nel");
                exit(1);
        }
        server.sin_family = AF_INET;
        hp = gethostbyname(argv[1]);
        if (hp == NULL) {
                fprintf(stderr, "%s: ismeretlen host\n", argv[1]);
                exit(2);
        }
        bcopy(hp->h_addr, &server.sin_addr, hp->h_length);
        server.sin_port = htons(atoi(argv[2]));

        if (connect(sock, &server, sizeof(server)) < 0) {
                perror("hiba a connect-nel");
                exit(1);
        }
        if (write(sock, DATA, sizeof(DATA)) < 0)
                perror("hiba a write-nal");
        close(sock);
}
```

5.15.3 Példa egy select-et használó összeköttetés-alapú szerverre

A következő program lefoglal egy TCP portot, kiírja annak a címét a képernyőre, és végtelen ciklusban vár a TCP-porton egy kapcsolatra, és kiszolgálja a rákapcsolódott klienst (a szolgálat annyi, hogy a kliens által elküldött byteokat a szabványos kimenetre kiírja). Ha egy megadott időn belül nem érkezik egy klienstől sem rákapcsolódási igény, akkor megfelelő hibaüzenetet ír ki, és előről kezdi az egészet.

```
/*
 * Ez egy egyszeru szerver, amely var egy kliens rakapcsoladasara, de
 * egy timeout erteket is definial, es ha azon az idon belul nem
 * kapcsolodik ra egy kliens sem, akkor azt eszreveszi es fut tovabb.
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define TRUE 1

main()
{
        int sock, length;
        struct sockaddr_in server; /* Internet domain-beli cim */
        int msgsock; /* Erre a socket-descriptorra accept-alja
                        a kapcsolatot */
        char buf[1024];
        int retval;
        fd_set ready;
        struct timeval to; /* timeout erteke */

        sock = socket(AF_INET, SOCK_STREAM, 0);
        if (sock < 0) {
                perror("hiba a socket-nel");
                exit(1);
        }
        server.sin_family = AF_INET;
        server.sin_addr.s_addr = htonl(INADDR_ANY);
        server.sin_port = htons(0);
        if (bind(sock, &server, sizeof(server))) {
                perror("hiba a bind-nal");
                exit(1);
        }
        length = sizeof(server);
        if (getsockname(sock, &server, &length)) {
```

```
                perror("hiba a getsockname-nel");
                exit(1);
        }
        fprintf(stderr,"TCP port:%d\n", ntohs(server.sin_port));

        /* Kliensekre var */
        listen(sock, 5);
        while(1) {
                FD_ZERO(&ready); /* \"ures halmazt hoz l\'etre */
                FD_SET(sock, &ready); /*sock socketot figyelni kell */
                to.tv_sec = 5; /* Timeout   */
                to.tv_usec=0; /* Microsec. */
                if (select(sock + 1, &ready, NULL, NULL, &to) < 0) {
                        perror("hiba a select-nel");
                        continue;
                }
                if (FD_ISSET(sock, &ready)) {  /* Be van allitva? */
                        msgsock=accept(sock,(struct sockaddr *)0,
                                       (int *)0);
                        if (msgsock == -1)
                                perror("hiba az accept-nel");
                        else do {
                                bzero(buf, sizeof(buf));
                                if ((retval=read(msgsock,buf,1024))<0)
                                        perror("hiba a read-nel");
                                else if (retval == 0)
                                        fprintf(stderr,"End...\n");
                                else
                                        fprintf(stderr,"%s\n", buf);
                        } while (retval > 0);
                        close(msgsock);
                } else
                        fprintf(stderr,"Nincs kapcsolat ... \n");
        };
        close(sock); /* Idaig nem sullyedhetunk */
}
```

5.15.4 Példa egy konkurrens összeköttetés-alapú szerverre

A következő program lefoglalja az 5678-as TCP portot, kiírja annak a sorszámát a képernyőre (vagyis az 5678-at), és végtelen ciklusban vár azon a TCP-porton egy kapcsolatra. Ha egy kliens rá akar kapcsolódni, akkor elfogadja a rákapcsolódási kérelmet, majd szül egy gyermek-folyamatot, amely a szokásos (korábban is látott) "szolgáltatást" elvégzi.

A SIGCLD signalt a program ignorálja, így a meghalt gyermek-folyamatok nem "hal-

mozódnak fel" zombi-proceszek formájában.

```
/*
 * Pelda arra, hogy hogyan mukodik egy (konkurrens) szerver.
 * A program vegtelen ciklusban figyel egy adott TCP portot, beolvassa
 * es a kepernyore irja az onnan jovo byteokat, majd uj kapcsolatra
 * var.
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <signal.h>
#define TRUE 1

main()
{
        int sock, length;
        struct sockaddr_in server;
        int msgsock;
        char buf[1024];
        int retval;
        int i;

        signal(SIGCLD,SIG_IGN);
        sock = socket(AF_INET, SOCK_STREAM, 0);
        if (sock < 0) {
                perror("hiba a socket-nel");
                exit(1);
        }
        server.sin_family = AF_INET;
        server.sin_addr.s_addr = htonl(INADDR_ANY);
        server.sin_port = htons(5678);
        if (bind(sock, &server, sizeof(server))) {
                perror("hiba a bind-nel");
                exit(1);
        }
        length = sizeof(server);
        if (getsockname(sock, &server, &length)) {
                perror("hiba a getsockname-nel");
                exit(1);
        }
        fprintf(stderr,"TCP port:%d\n", ntohs(server.sin_port));

        listen(sock, 5);
        do {
```

```
                msgsock = accept(sock, 0, 0);
                if (msgsock == -1)
                        perror("hiba az accept-nel");
                else
                {
                  if (fork() == 0)
                  { /* Gyermek processz */
                    close(sock);
                    do {
                        bzero(buf, sizeof(buf));
                        if ((retval = read(msgsock, buf, 1024)) < 0)
                                perror("hiba a read-nel");
                        i = 0;
                        if (retval == 0)
                                fprintf(stderr,"Kapcsolat lezarva\n");
                        else
                                fprintf(stderr,"String:%s\n", buf);
                    } while (retval != 0);
                    close(msgsock);
                    exit(0);
                  }
                  else
                  {
                    close(msgsock);
                  }
                }
        } while (TRUE);
        close(sock); /* Sosem sullyedunk idaig */
}
```

5.15.5 Példa egy összeköttetés-mentes (datagram) szerverre

A következő program lefoglal egy UDP portot, kiírja annak a címét a képernyőre, és vár az UDP-porton egy datagramra, és ha kap egyet, akkor kiírja a tartalmát.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>

/*
 * A program letrehoz egy UDP socketot, nevvel latja el, es egy ra
 * erkezo csomagot fogad es kiir.
 */

main()
```

```
{
        int sock, len;
        struct sockaddr_in server, from;
        char buf[1024];
        int addrlen;

        sock = socket(AF_INET, SOCK_DGRAM, 0);
        if (sock < 0) {
                perror("hiba a socket-nel");
                exit(1);
        }
        server.sin_family = AF_INET;
        server.sin_addr.s_addr = htonl(INADDR_ANY);
        server.sin_port = htons(0);
        if (bind(sock, &server, sizeof(server))) {
                perror("hiba a bind-nal");
                exit(1);
        }

        len = sizeof(server);
        if (getsockname(sock, &server, &len)) {
                perror("hiba a getsockname-nel");
                exit(1);
        }
        fprintf(stderr,"UDP port:%d\n", ntohs(server.sin_port));

        if (recvfrom(sock, buf, 1024, 0, &from, &addrlen) < 0) {
                perror("hiba a recvfrom-nal");
                exit(2); }
        fprintf(stderr,"Fogadott szoveg: >>%s<<\n", buf);
        close(sock);
}
```

5.15.6 Példa egy összeköttetés-mentes (datagram) kliensre

A következő program a megadott host megadott UDP portjára adatokat küld.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define DATA "Isten hozta ornagyur ..."

/*
```

```
 * Hivas: udpkliens hostname port-nr.
 *
 * Ez a program a parametereben megadott hostra, es a megadott
 * UDP-portra kuld egy datagramot.
 */

main(argc, argv)
        int argc;
        char *argv[];
{
        int sock;
        struct sockaddr_in server;
        struct hostent *hp, *gethostbyname();

        sock = socket(AF_INET, SOCK_DGRAM, 0);
        if (sock < 0) {
                perror("hiba a socket-nel");
                exit(1);
        }
        hp = gethostbyname(argv[1]);
        if (hp == NULL) {
                fprintf(stderr, "%s: ismeretlen host\n", argv[1]);
                exit(2);
        }
        bcopy(hp->h_addr, &server.sin_addr, hp->h_length);
        server.sin_family = AF_INET;
        server.sin_port = htons(atoi(argv[2]));
        if (sendto(sock,DATA,sizeof(DATA),0,&server,sizeof(server))<0)
                perror("hiba a sendto-nal");
        close(sock);
}
```

5.16 A hálózati réteg (IP protokoll) elérése

Fejezet 6

Security

Ebben a fejezetben az operációs rendszerekkel kapcsolatban felmerülő biztonságossági kérdésekről lesz szó (leginkább a UNIX rendszerben felmerülő gondokról). Ez a nyílt rendszerekkel kapcsolatban az egyik legfontosabb kérdéskör, mert a fájlrendszerben (és a rendszer többi részében) tárolt információk a tulajdonosuknak komoly értéket jelenthetnek. A biztonságosságra főleg a hálózatok jelenléte miatt kell ügyelni (mivel mondjuk egy hibásan, átgondolatlanul megírt FTP-szerver akár mindenkinek mindent megengedhet - mivel ez egyes implementációkban az FTP szerver gyakran setuid root program azaz végrehajtásakor szuperfelhasználói jogokkal rendelkezik). Tehát a hangsúly a rendszerhez való hozzáférés szabályozásán valamint a kommunikáció biztonságossá tételén van (nem szabad megengedni, hogy a rendszerben tárolt adatokhoz illetéktelenek hozzáférjenek) - az nyilvánvaló, hogy ez a kérdés túlmutat az adattitkosítás kérdésén - és az élet más területeivel is szoros kapcsolatban van: például a rendőrség hozzáférhet az emberekről tárolt adatokhoz vagy sem; vagy mi számítson banktitoknak?

Bizonyos szempontból ide tartozik az adatvesztés elkerülése is (pl. mert leég a ház, vagy véletlenül rossz floppyt formattálunk, ...), de ezekkel a kérdéssel itt nem akarok foglalkozni (nem is igen tudnék mit írni, azon kívül, hogy rendszeresen csináljunk minden adathordozón tárolt információról biztonsági másolatot).

6.1 Tervezési elvek

A következő pontok néhány hasznos szempontot mutatnak be, amelyet biztonsági rendszerek tervezésekor érdemes követni:

- A rendszerterv és a használt biztonsági protokoll legyen nyilvános. Könnyelműség azt feltételezni, hogy a rendszerbe behatolni akarók azt nem fogják ismerni.
- Alapértelmezés mindig legyen az, hogy valaki valamihez nem férhet hozzá.
- A securityvel kapcsolatos kérdéseket a rendszer tervezésének korai fázisában tisztázni kell, és a security csomagot a rendszer magjába integrálni kell - annak nincs sok értelme, hogy egyetlen modult megcsinálunk úgy, hogy az nagyon erős biztonsági előírásoknak feleljen meg, a többi modul pedig nem tartalmaz ilyen célú részeket.

- Az alkalmazott biztonsági intézkedések pszichológiailag elfogadhatóak legyenek. A rendszert valószínűleg emberek fogják használni, ezért a rendszert "barátságossá" kell tenni.

- Ha lehet kerüljük az egész rendszer felett "teljes hatalommal bíró" rendszergazda (superuser, supervisor ...) koncepciót. A rendszert bontsuk moduljaira (valamilyen szempontok szerint - mondjuk egy-egy modul egy-egy fontosabb erőforrás kezelését végezze), és az egyes moduloknak legyenek (külön-külön) felügyelői.

6.2 A felhasználó azonosítása

Komoly problémát jelent a felhasználó azonosítása a rendszerrel szemben (mindig felmerülhet a kérdés, hogy a felhasználó tényleg az, akinek vallja magát?). Erre is létezik már több (pszichológiailag is elfogadható) megoldás - a leggyakoribbak a következők:

- A felhasználónak van valamilyen jelszava, amit csak ő (és mondjuk az operációs rendszer) tud, és a rendszer a bejelentkezéskor ezt a **jelszó**t megkérdezi tőle. Ha a felhasználó a jelszót beadja, és az egyezik a rendszer által ismert jelszóval, akkor nagy a valószínűsége annak, hogy az a személy, aki bejelentkezett tényleg az, akinek kiadja magát. Ilyen rendszerekben problémát az okoz, hogy hol tárolja a rendszer a jelszavakat. Ezt vagy úgy oldják meg, hogy a jelszófájl egy speciális fájl-attributummal van ellátva, és senki sem olvashatja el a tartalmát (de ekkor biztonsági másolatot sem lehet róla készíteni!). Egy másik módszer a jelszófájl tárolására az, amit a UNIX operációs rendszernél alkalmaznak: a jelszófájl mindenki által olvasható (általában `/etc/passwd` a neve). Viszont ebben a fájlban a jelszó nem nyílt szöveg, hanem titkosított formában van tárolva (a 2. oszlopban van a jelszó). Amikor a rendszerbe bejelentkezik egy felhasználó, és beadja a jelszavát, akkor a rendszer elvégzi rajta a "szokásos" (de természetesen egyirányú) titkosítási folyamatot, és az eredményt összehasonlítja a jelszófájlban tárolt titkosított jelszóval. Ha a kettő megegyezik, akkor a felhasználó jó jelszót adott be, és be lehet engedni a rendszerbe.

- Gyakran használnak ún. **egy alkalomra szóló jelszavak**at. Ekkor a felhasználó kap egy füzetet, amelyben mondjuk a következő 1000 bejelentkezéshez szükséges jelszavak vannak. Ennek a módszernek az a gyenge pontja, ha a felhasználó elveszti ezt a füzetet.

- Más módszer a kérdés-feleletek módszere. Ekkor a felhasználó (mondjuk amikor először leült a géphez) beadott az operációs rendszernek nagyon sok kérdést és rájuk a helyes válaszokat. Egy következő bejelentkezéskor az operációs rendszer ezek közül a beadott kérdések közül választ találomra mondjuk ötöt, és megkérdi rájuk a választ. Ha a válaszok helyesek, akkor a felhasználót beengedi a rendszerbe.

- Az azonosítás történhet akár fizikai és biológiai tulajdonságok felhasználásával is (pl. a retina erezete az emberekre jellemző, és hosszú időn keresztül lényegében állandó - ezt kihasználva lehet egy olyan berendezést csinálni, amelybe a bejelentkezéskor bele kell nézni, és elvégzi az azonosítást).

6.3 A 4.3BSD UNIX r-programjai

A 4.3BSD UNIX-ban vannak több olyan program, amely lehetővé teszi, hogy fájlokat másoljunk egyik hostról a másikra (rcp - remote copy) parancs, egy másik hoston shell scripteket futtassunk (rsh - remote shell) vagy bejelentkezzünk valamely más hostra és azon interaktívan dolgozzunk (rlogin - remote login).

Ezek a programok lehetőséget nyújtanak annak megadására, hogy ezek a szolgáltatások mely hostokról érhetőek el - ezzel bizonyos fokú illetékességvizsgálatra adnak lehetőséget.

Érdemes megnézni, hogy hogyan történik mindez!

A 4.3BSD UNIX-os hostokon van egy `/etc/hosts.equiv` nevű file, amely tartalmazza azoknak a hostoknak a neveit, amelyeken azonosak a kiosztott useridek (felhasználói nevek) a helyi hoston levő kiosztással. Ekkor ha egy ilyen "ekvivalens" hostról mondjuk a kutya nevű felhasználó egy fájlt akar másolni egy olyan helyi directoryba, amelyre a helyi kutya nevű felhasználónak írási joga van, akkor ez a művelet meg van engedve. Van egy másik fontos fájl is: a `.rhosts` nevű (ez a felhasználó HOME-directoryjában található). Ebben is lehetnek hostnevek felsorolva (ahol a tulajdonos userid-je (login neve!) megegyezik a helyi login-névvel) vagy lehetnek benne `(hostnév,login-név)` párok is. Ez utóbbi esetben ha a távoli hostról a megadott login-nevű felhasználó be akar jelentkezni annak a felhasználónak a login-nevével, akinek a HOME directoryjában a `.rhosts` file ezt a (hostnév,login-név) párt tartalmazza, akkor az azonosítás a jelszó bekérése nélkül sikeresnek tekinthető.

Ha saját szervereket írunk, ahol ezt az r-programokban alkalmazott illetékességvizsgálatot a `ruserok()` könyvtári függvénnyel végezhetjük el. Ennek alakja:

```
ret=ruserok(rhost, superuser, remote_username, local_username);
     char *rhost;
     int superuser;
     char *remote_username;
     char *local_username;
```

Eredmény: ret=0, ha az illetékességvizsgálat sikeres volt, azaz a felhasználó beléphet jelszóbeadás nélkül, egyébként `ret` értéke -1 lesz. (Az `rhost` paraméter annak a hostnak a nevét tartalmazza, amelyről be akarnak jelentkezni (`gethostbyaddr(getpeername())`) módon kaphatjuk ezt meg. A `superuser` változó akkor 1, ha a bejelentkező ezen a gépen superuser jogokat akar-e kapni. A maradék két paraméter a távoli és a helyi hostokon a felhasználói nevek.)

6.4 A Kerberos illetékesség-vizsgáló protokoll

A Kerberos elkészítésének célja az volt, hogy a hálózatba rakott "nem megbízható" számítógépekkel se lehessen a rendszerben illetéktelenül dolgozni, a szerver és a kliens közt áramló adatokat titkosítani lehessen, és a **kliensek ne tagadhassák le kilétüket a szerverek előtt**. A Kerberos rendszer a következőképpen működik:
Van egy belső adatbázis, amely (**azonosító**, **kulcs**) párokat tárol, ahol az azonosítók között előfordul az egész rendszerhez hozzáférő összes felhsználó azonosítója, és minden egyes szerverhez is tartozik egy-egy azonosító.

- Amikor a felhasználó bejelentkezik valamelyik számítógépbe, és már beírta a nevét a `login:` üzenet után, a login program (aki a bejelentkezéskor fellépő feladatokat végzi el) egy üzenetet küld a Kerberos illetékesség-vizsgáló szervernek. Az üzenet tartalmaz két szöveget: az egyik szöveg a felhasználó azonosítója, a másik szöveg pedig az ún. **jegykiadó szerver** (ticket-granting szerver, TGS) nevét tartalmazza (ennek a szerepéről később lesz szó).

- Az illetékesség-vizsgáló szerver a fent említett adatbázisból kikeresi a kapott üzenetben lévő két azonosítóhoz tartozó kulcsokat (a felhasználói azonosítóhoz tárolt kulcs megegyezik a felhasználó titkosított jelszavával). Ezután a szerver egy válaszüzenetet küld vissza, ami tartalmaz egy titkos ún. TGS-kulcsot, és egy ún. **jegyet**. Ez a TGS-jegy tartalmazza a felhasználói azonosítót, a jegykiadó szerver (TGS) nevét, annak a számítógépnek az Internet címét, amelyen a felhasználó dolgozik, és a fent említett TGS-kulcsot. A TGS-jegy az üzenetben titkosítva van a felhasználó által nem ismert kulccsal (a jegykiadó szerver azonosítójához tartozó kulccsal), és a teljes üzenet titkosítva van a felhasználó titkosított jelszavával, hogy más ne férjen hozzá a tartalmához.

- A felhasználó jelszavának beadása után az azonosító szervertől visszakapott üzenetet a login program a felhasználó titkosított jelszavával dekódolja, és a TGS-kulcsot valamint a (titkosított) TGS-jegyet eltárolja.

- Ezután ha a felhasználónak valamilyen hálózati szolgáltatásra van szüksége, akkor a jegykiadó szervertől (TGS-től) kérnie kell egy "jegyet" a kívánt szerverhez, és a szervernél ezzel a jeggyel azonosíthatja magát.

- Ha például a nyomtató szervert akarjuk használni, akkor ahhoz egy jegy kéréséhez a jegykiadó szerverhez egy olyan üzenetet kell küldeni, amely a korábban eltárolt TGS-jegyet, a nyomtató szerver nevét és egy, a korábban megkapott TGS-kulccsal titkosított adatmezőt (ellenőrző mezőt). Ez az ellenőrző mező tartalmazza a felhasználó bejelentkezési nevét, a gépjének az Internet címét, és a pillanatnyi időt.

- A jegykiadó szerver a kapott üzenet ellenőrzése után válaszüzenetet küld, amiben van egy új kulcs, és egy új (titkosított) jegy. Ez a titkosított jegy a szolgáltatást kérő kilétét igazolja, és a nyomtató szerver kulcsával van titkosítva (ami a Kerberos adatbázisában van, és **csak a nyomtató szerver és a Kerberos férhet hozzá** - ui. a nyomtató szerverbe például "be van huzalozva".). A válaszüzenet a korábbi TGS-kulccsal van titkosítva.

- A felhasználó programja az üzenetet dekódolja, és a nyomtató szervernek a fenti "új jegyet" kell elküldenie (másik két adatmező mellett), ezzel igazolhatja kilétét. A másik két adatmezők közül az egyik a nyomtató szerver nevét, a másik pedig (a korábban említett ellenőrző mezőhöz hasonlóan) a felhasználó bejelentkezési nevét,

a gépjének az Internet címét, és a pillanatnyi időt tartalmazza (ez a mező az előző pontban említett új kulccsal van titkosítva, ezért ezt a nyomtató szerver dekódolni tudja).

A két helyen is említett ellenőrző mező tartalmazza a pillanatnyi időt is. Ez azért lényeges, mert ezek az "igazolványok" viszonylag rövid élettartalmúak, nehogy valami illetéktelen "hálózatlehallgató" újra fel tudja használni. Viszont ekkor meg kell oldani azt, hogy a hálózatban levő gépek órájai nagyjából együtt legyenek.

6.5 Tanácsok setuid root programok írásához

Ha lehet ne írjunk ilyen programokat, mivel ezek futtatáskor korlátozás nélkül garázdálkodhatnak a rendszerben (ez talán a legkomolyabban legmegfontolandóbb tanács). Ha arra van szükség, hogy bizonyos fájlokhoz (itt nem rendszerfájlokra gondolok, hanem mondjuk egy programmal szállított licensz-fájl vagy valami hasonló) csak a program férjen hozzá, akkor elég mondjuk egy új felhasználót létrehozni, és a programhoz tartozó "kényes" fájlokat ezen új felhasználó tulajdonaként kell kezelni, és amely programnak ehhez hozzá kell férnie, azt el kell látni egy setuid bittel (de ettől még nem setuid root programot kapunk!).

A setuid root programok írásakor törekedjünk arra, hogy minél rövidebb legyen a program azon szakasza, ahol a program kihasználhatja azt, hogy "mindent szabad neki".

Például azokat a szervereket szokták setuid root bittel ellátni, amelyek a felhasználótól megkövetelik, hogy beadja a jelszavát és a login nevét (userid-jét). Ha jó a beadott jelszó, akkor a szerver végrehajtja a `setuid()` és `setgid()` UNIX rendszerhívásokat annak a felhasználónak a felhasználói- illetve csoport-azonosítójával, aki bejelentkezett, és a szerver ezután már el is veszti a superuser jogokat - ezután már csak azt szabad neki is, amit a bejelentkezett felhasználó interaktívan is megtehetne a rendszerben.

Ha egy szervert úgy akarunk megírni, hogy azt bárki használhassa anélkül, hogy ténylegesen bejelentkezett volna a rendszerbe (mint például az anonymous FTP szolgáltatás), akkor hozzunk létre egy új felhasználót (az anonymous FTP szolgáltatás esetén ennek a felhasználónak mindig `ftp` a neve), amelyre a szerver olyankor `setuid()`-el (és `setgid()`-el), ha a kliens nem tudott neki más ("jobb") felhasználói azonosítót és jelszót beadni.

A setuid root programok minden fájlhoz hozzáférnek - függetlenül attól, hogy ki indította el őket. A UNIX operációs rendszer lehetőséget nyújt a korábban bemutatott `access()` rendszerhívással, hogy egy fájlra vonatkozóan eldöntsük, hogy aki a setuid root programot elindította, az a saját jogán olvashatja-e, felülírhatja-e és végrehajthatja-e azt a fájlt. (Ez az út azért járható, mert az `access()` a valódi felhasználói azonosító alapján vizsgálja a jogokat, nem pedig az effektiv felhasználói azonosító alapján; a setuid bit csak az effektiv felhasználói azonosítót állítja – a valódi felhasználói azonosító is tárolva van a processz-táblában, amit a `getuid()` rendszerhívással lekérdezhetünk vagy a `setuid()` rendszerhívással megváltoztathatunk).

A setuid programok mindig zárják le a megnyitott fájljaikat, mert a gyermek-folyamatként végrehajtott folyamatok a megnyitott fájlokat öröklik, és így sok értékes információhoz juthatnak (mindegy, hogy miért - ennek lehet az oka programhiba is vagy lehet akár egy ravasz rendszerprogramozó is).

A programok által használt temporális fájlokat úgy hozzuk létre, hogy azokra a tulajdonosuknak minden (ésszerű) jogokat meg kell adni, míg másoknak ne legyen semmi joguk a fájlra vonatkozóan (vagyis a 0700 oktális szám gyakran megfelel a `creat()`-nél).

Mindig ellenőrizzük a rendszerhívások visszatérési értékét, és ha szükséges, az `errno` változó értékét is. Ezzel kapcsolatban ismert a következő security-lyuk a UNIX egy korábbi verziójával kapcsolatban: bárki rendszergazda jogokhoz juthatott egy kis ügyességgel, ugyanis a `su` parancsban volt egy óriási hiba. Az `su` rendszerprogram olyan esetben, amikor nem tudja megnyitni a jelszófájlt (neve `/etc/passwd`), jelszóbekérés nélkül rendszergazda jogokkal illeti meg a végrehajtóját. A UNIX készítői úgy gondolták, hogy a jelszófájl hiánya olyan komoly probléma – véletlenül nem is fordulhat elő –, hogy ez a működés elfogadható. Viszont a `su` rendszerprogramban a jelszófájlt megnyitó `open()` rendszerhívás sikertelenségének nem vizsgálták meg az okát vagyis azt, hogy valóban nem létezik-e a jelszófájl, vagy pedig más oka van a fájlmegnyitás sikertelenségének. Ezt a legtöbben úgy játszották ki, hogy megnyitottak annyi fájlt, amennyit csak lehetett[1] , és végrehajtották a `su` parancsot. Ekkor az elindított `su` parancs örökölte az őt elindító folyamat megnyitott fájldeszkriptorait, de a jelszófájlt már nem tudta megnyitni, mert nem volt szabad fájldeszkriptora: már megnyitott annyi fájlt, amennyi a folyamat számára maximálisan engedélyezve volt. Így tehát az `open()` rendszerhívás sikertelen volt, annak ellenére, hogy volt egy olvasható jelszófájl, mégis úgy tekintette a rendszer, hogy "nagy baj van", és rendszergazda jogokkal ajándékozta meg az óvatlan végrehajtót ...

[1] Vagy kicsit kevesebbet .... persze vigyáztak arra, hogy a `su` utility be tudja kérni a jelszót a `/dev/tty` fájlról, és ehhez azt meg kellett tudnia nyitni. Itt megjegyezzük, hogy a UNIX rendszerben statikus korlát az egy folyamat által egyidejűleg megnyitható fájlok száma.

Fejezet 7

A rendszermag szerkezete

Ebben a fejezetben áttekintjük az előző fejezetekben bemutatott rendszerszolgáltatások implementálását: azt, hogy milyen módon vannak implementálva a UNIX-szerű rendszerekben. Nyilván több implementáció is létezik (System V, BSD, Linux, Solaris, ...) és mindegyik bemutatására nincs lehetőség (a forráskódban lehetnek nagyobb különbségek is), itt inkább a fontosabb adatszerkezetek és a rendszererőforrások kapcsolatát és kezelésmódját fogom bemutatni. (Az előforduló példák leginkább a 4.3BSD rendszer korai változataiból származnak; a memóriakezelésnél bemutatott régió rendszer pedig a System V Release 3.0-ból származik[1].)

Először áttekintjük a folyamatok kezelésének a fontosabb részleteit, majd a hátterében levő memóriakezelési mechanizmusokat valamint a fájlkezelés fontosabb kérdéseit fogjuk tisztázni.

Meg kell említeni, hogy a legtöbb adatszerkezet a régebbi rendszermag implementációkban statikus méretű volt (pl. egy fix méretű vektorban voltak tárolva), amit ma már megpróbálnak kiküszöbölni, így a rendszer maximális mértékben képes lesz kielégíteni a változó erőforrásigényeket. Azt, hogy egy tábla fix méretű-e vagy sem, itt nem fogom leírni (nem is írhatom), mivel ez egyrészt rendszermag implementációnként változik másrészt pedig ugyanannak a rendszernek a különféle (például más-más gyártóktól származó) változatai egyes részletekben eltéréseket mutathatnak egymástól.

Nyilvánvalóan a következő pontokban leírt információk nem általános érvényűek (vagyis egy-egy rendszer implementálója dönthet úgy, hogy valamit máshogyan implementál), viszont az eddig elkészült monolitikus operációs rendszer implementációkban az itt közölt elveket követik a leggyakrabban.

A UNIX-szerű rendszerek forráskódját eddig általában nagyrészt C nyelven készítették el a hardverfüggő részek kivételével. A hardverfüggő részeket pedig általában alacsonyszintű assembly nyelven írják (sorok számát tekintve ez gyakran az egész rendszer méretének az 5 százaléka alatt van). A hardverfüggő részek leginkább az eszközmeghajtókat kezelő, valamint a memóriakezelő részekben találhatóak, míg ennél kevesebb a hardverfüggő részek mennyisége a folyamatok kezelését végző komponensekben, valamint a fájlrendszerkezelőben.

[1] A bemutatott eszközök kiválasztásánál elsősorban a szemléletesség volt a célom, nem volt cél, hogy minden eszközt ugyanabból az operációs rendszerből válasszak.

7.1 A folyamatok kezelése

Először áttekintjük a folyamatok kezeléséhez szükséges adatszerkezeteket, majd megnézzük azt, hogy milyen módon lehet az egyes folyamatkezelő rendszerhívásokat implementálni.

7.1.1 A folyamatkezelés adatszerkezetei

Korábban már említettem, hogy egy operációs rendszer megtervezésekor döntő szerepe van annak, hogy mi minden kerül egy folyamatba – azaz milyen erőforrások gyűjteménye a folyamat. Ez alapvetően meghatározza a rendszerben futó programok környezetét. A legtöbb rendszermag implementáció tartalmaz egy `folyamatleíró` adatszerkezetet, amelyben összegyűjtik egy-egy folyamat állapotának az összes jellemzőjét. Ez a folyamatleíró adatszerkezet logikailag két fő részre bontható: az egyik rész azokat az információkat tartalmazza, amelyeknek minden időpillanatban a memóriában kell lenniük (ezt nevezik a System V, BSD rendszerekben `proc` strukturának), míg a másik rész (a `user` (felhasználói) struktura) azokat az információkat tartalmazza a folyamatról, amikre csak a rezidens állapotában van szükség (vagyis olyankor, amikor a folyamat a fizikai memóriában tartózkodik, azaz a memóriakezelő nem rakta ki a háttértárra – a fizikai memória kis mérete miatt). A rendszer tartalmaz egy ún. processz-táblát, amely (gyakran fix méretű) folyamatonként tartalmaz egy hivatkozást a megfelelő folyamat `user` illetve `proc` strukturájára, valamint tartalmazza egy egész konstanst, ami a folyamat exit-státuszát és állapotát jellemzi, és a következő állapotok valamelyikét jelölheti ki a folyamat aktuális állapotaként:

- `SSLEEP` : A folyamat egy esemény bekövetkeztére vár (ld. később).
- `SRUN` : A folyamat futásra kész (lehet, hogy pillanatnyilag nem ő, de az ütemező átadhatja neki a vezérlést).
- `SZOMB` : A folyamat befejeződött, de a visszatérési értékét (exit-státuszát) még nem kérdezte meg a szülője, ezért ez a processz-tábla bejegyzés még nem lett megszüntetve.
- `SSTOP` : A folyamat futása meg lett állítva (például egy signallal).
- `SIDL` : Egy segédállapot, amelyet a rendszermag létrehozás alatt álló, még nem futtatható folyamatok számára tart fenn.

A `proc` struktúra a következő információkat tartalmazza:

- Ütemezési jellemzők (például a folyamat prioritását), a folyamat processzorterhelésének a mértékét, ...
- Folyamatazonosítót (a szülőfolyamat azonosítójával együtt).
- A folyamatot futtató felhasználó azonosítóját.
- Memóriakezelési információk:
  - Egy hivatkozás a program kódjának a szerkezetét leíró táblázatra (ezt nevezik a BSD-ben `text` struktúrának – erről nemsokára részletesebben is írok).

  - A folyamat laptáblájának a címét is tartalmazza.
  - A folyamat `user` strukturájának a memóriabeli vagy (kilapozott állapotban) háttértárolón levő helyét tartalmazza.

- Egy esemény megnevezését is tartalmazhatja, amelynek a bekövetkezéséig a folyamat várakozik (vagyis nem fut). A rendszermag egy-egy esemény bekövetkeztekor (például amikor egy blokkot beolvasott a diszkről) megnézi azt, hogy mely folyamatok várakoznak az adott eseményre, és továbbindítja őket. Egy folyamat egyszerre legfeljebb csak egy esemény bekövetkezésére várakozhat.

- Tartalmazza azt, hogy milyen signalokat küldtek a folyamatnak (ez lehet akár egy 32-bites egész szám, egy-egy bitje egy-egy signalnak felelhet meg). Továbbá tartalmaz arról is információkat, hogy az egyes signalokat a folyamat milyen módon kezeli (nem veszi figyelembe őket, vagy van a folyamatban egy signal-kezelő eljárás, vagy pedig az alapértelmezés szerinti signalkezelési módot választotta a folyamat). Amennyiben egy folyamat valamilyen eseményre várakozás közben signalokat kap, akkor bejegyzi ide, hogy milyen signalt kapott, és a kapott signalt csak azután fogja feldolgozni, miután az várt esemény bekövetkezett, és a folyamat továbbfut.

- Tartalmazza, hogy a folyamat mikorra kért "ébresztőt" (amennyiben önként várakozó állapotba ment valamennyi idő elteltéig, vagy egy ALARM signalt kért az operációs rendszertől).

A fenti adatok szerepe alapján látható, hogy állandóan a memóriában kell őket tartani (ezek nélkül nem lehetne "igazságosan" kiválasztani azt, hogy egy kontextuscsere után melyik folyamat kapja meg a processzort).

A folyamatok `proc` struktúrája egyrészt fel van fűzve egy kétirányú listába, amely listán végigmenve az összes folyamat adataihoz hozzáférhetünk. Korábban már említettem, hogy egy folyamat tetszőleges számú utódot (gyermek-folyamatot) hozhat létre, így kialakulhat a folyamatoknak egy fa szerkezetű strukturája, amely egy általános fa (vagyis nem mondhatunk egy felső korlátot arra vonatkozóan, hogy a fában egy folyamatot reprezentáló pontnak legfeljebb hány gyermeke lehet). Ezt a fa szerkezetű hierarchiát is tárolni kell, amit a legtöbb rendszerben nem közvetlenül általános faként tárolnak, hanem az általános fa adatszerkezetet bináris fára visszavezetve tárolják (a bináris fára való visszavezetéses tárolás úgy történik, hogy a bináris fa minden egyes csomópontjában van egy hivatkozás az általános fabeli megfelelő pont első gyermekére, valamint van egy hivatkozás az általános fabeli testvéreire – ez a módszer az adatszerkezeteket tárgyaló könyvekben magtalálható alapvető módszer; számunkra ez már kevésbé lényeges).

Egy folyamat `user` strukturája a következő információkat tartalmazza:

- Tartalmazza azt, hogy a folyamat "felhasználói" üzemmódban fut-e, vagy pedig valamilyen rendszerhívást végrehajtva a rendszermag valamelyik részében "fut".

- A rendszerhívások paramétereit és végrehajtásuk állapotát leíró adatszerkezeteket.

- A folyamathoz tartozó programvégrehajtási pont jellemzőit (processzorregiszterek értékét, a végrehajtási verem tetejére mutató veremmutató).

- A folyamathoz tartozó második (kernel módban használt) veremmutató értékét (egy folyamathoz leginkább biztonságossági okok miatt két verem tartozik: az egyik vermet a program az eljárások paramétereinek, lokális változóinak és a visszatérési címek tárolására használja – erre szoktak egyszerűen végrehajtási verem néven hivatkozni); a másik veremet a rendszermag olyankor használja, amikor a folyamat kérésére valamilyen szolgáltatást elvégez (ezt nevezik a folyamat rendszermagbeli vermének (angolul kernel stack), és a `user` struktura nemcsak a veremmutatót, hanem magát ezt a második (rendszermagbeli) vermet is tartalmazza. A két veremre elsősorban biztonsági okok miatt van szükség: egyrészt azzal, hogy a (fix méretű – néhány KB méretű) rendszermagbeli verem számára szükséges (elég kicsi) hely állandóan le van foglalva, nem fordulhat elő az a kellemetlen helyzet, hogy a rendszermag működése során "megbénul" egy esetleges memóriabetelés miatt (ciki lenne, ha a rendszermag egy eljárást sem tudna meghívni, mivel mondjuk betelt a memória); továbbá mivel a folyamat "felhasználói" módban nem fér hozzá a rendszermagbeli vermének a tartalmához, ezért nem tud hozzáférni a vermen levő olyan információkhoz, amit a kernel "rajta hagyott", és nem törölt onnan (ezekhez az információkhoz a kernelen kívül másnak általában nincs semmi köze).
- Itt van tárolva a folyamat fájldeszkriptoraihoz tarozó fájl-táblabeli hivatkozás (vagyis az, hogy melyik fájldeszkriptorhoz a globális fájltábla melyik eleme tartozik – emlékezzünk rá, hogy erre azért is szükség van, mert egy folyamat "osztódása" után a szülő és a gyermek folyamat ugyanazokhoz a fájlokhoz férhet hozzá, a fájl-pozíciókat pedig megosztva használják). Itt vannak tárolva fájldeszkriptorhoz kötött információk is (például az, hogy a fájlt a folyamat "osztódása" után automatikusan le kell zárni).
- A folyamat erőforráshasználatának jellemző adatai (azaz miből mennyit használt (CPU időt)).
- A folyamatot futtató felhasználó csoport-azonosítója, valamint az effektiv felhasználói és csoportazonosítók is itt lesznek tárolva.
- A folyamat munka-directoryja is itt van tárolva.
- A folyamat gyökér-directoryja is itt van tárolva (ne felejtsük el, hogy ez is változhat – gondoljunk csak a `chroot()` rendszerhívásra).

Egy folyamat a memóriában több ún. szegmensből áll (egyes implementációkban a szegmens helyett a régió elnevezést is használják). A szegmenseknek három "osztályát" szokás megkülönböztetni (ez az osztályozás a szegmensek szerepe alapján történik): vannak végrehajtható utasításokat tartalmazó ún. kódszegmensek, adatokat tartalmazó adatszegmensek, valamint a végrehajtási vermet tartalmazó veremszegmensek. Egy folyamat általában mindhárom szegmenstípussal rendelkezik; a szegmenseket a virtuális memóriában úgy szokás elhelyezni, hogy a virtuális címtartomány valamelyik "végén" (leggyakrabban az alján) helyezkedik el a fix méretű kódszegmens (ennek a mérete adott, miután a programot lefordítottuk), a fennmaradó helyen pedig a verem és az adatszegmens helyezkedik el – általában ezek nem rögzített méretűek, kezdetben a virtuális címtartomány két "szélén" helyezkednek el, és egymás irányába híznak. A szegmensek általában egy szegmenstáblába vannak rendszerezve.

Már említettem, hogy minden folyamathoz nyilván van tartva egy hivatkozás egy ún. `text` strukturára. Ez a `text` struktúra pointereket tartalmaz a folyamat végrehajtható kódját tartalmazó kódszegmensekre. Minden `text` struktúra tartalmaz egy referenciaszámlálót (ez egy egész szám), amelyben tárolja, hogy hány folyamat hivatkozik rá a `proc` strukturájában (vagyis az általa leírt végrehajtható kódot hány program használja)), valamint tartalmazza a rá hivatkozó folyamatok `proc` strukturájának a listáját.

7.1.2 A folyamatkezelés rendszerhívásai

Már láttuk a folyamatok szerkezetének a kezelésével kapcsolatos fontosabb adatszerkezeteket; most áttekintjük azt, hogy a folyamatokat kezelő rendszerhívások (például a `fork()` és az `exec()`) (illetve a `wait()` és az `exit()`) hogyan lesznek megvalósítva, mit kell csinálniuk ezeken az adatszerkezeteken a feladatuk elvégzéséért.

A `fork()` rendszerhívás létrehoz egy új `proc` struktúra objektumot az új gyermekfolyamat részére és megfelelően kitölti azt (egyedi folyamatazonosítót kap, bejegyzi a futtató felhasználó azonosítóját, ...). Ezután a szülő folyamat `user` strukturáját lemásolja, és a másolatot az új folyamat megkapja. Mivel mind a szülő, mind pedig a gyermek folyamat ugyanazt a kódot hajtja végre (mindkettőnek a `proc` strukturája ugyanarra a `text` strukturára mutat), ezért a szülő `text` strukturájának a referenciaszámlálóját növelni kell eggyel a konzisztencia érdekében, és az újonnan létrehozott folyamat `proc` strukturáját be kell csatolni a folyamatok korábban már említett kétirányú listájába illetve a folyamatok fa-strukturájú hierarchiájába. Ezután az adat- illetve veremszegmensek számára lesz hely lefoglalva (pontosan akkora, amekkora a szülőben volt a megfelelő szegmens mérete), és a szülő adat- illetve veremszegmensének a tartalma át lesz másolva a gyermekfolyamat megfelelő szegmenseibe. Megjegyezzük, hogy ezt a szegmensmásolást gyakran optimalizálják olymódon, hogy a gyermek és a szülő folyamatok ugyanazt a memóriát használják egészen addig, amiíg valamelyikük nem módosítja, és a tényleges másolásra csak a módosításkor (általában csak a módosított részt tartalmazó lapon) kerül sor. Ez az optimalizálás a `fork()` műveletet nagyon "olcsóvá" teheti a legtöbb alkalmazásakor, mivel a `fork()` szolgáltatást használó programok nagyon nagy százalékában a `fork()` rendszerhívást egy `exec()` rendszerhívás követi a gyermekfolyamatban, aminek az eredményeként a folyamat virtuális címtartományának a kód-, adat- illetve veremszegmensét ki lehet dobni ... Természetesen a processz-táblában is létrejön egy új bejegyzés a megfelelő `proc` illetve `user` strukturákkal, a folyamat pedig futásra kész `SRUN` állapotban lesz bejegyezve.

Az `exec()` rendszerhívás nem hoz létre új `proc` illetve `user` strukturát, hanem az újonnan elindítandó folyamat kódját tartalmazó kódszegmenssel egy új `text` strukturát hoz létre (ha a megfelelő kódszegmenst tartalmazó `text` struktura már létezik, akkor annak a referenciaszámlálóját növeli eggyel), a régi kódot tartalmazó `text` struktúra referenciaszámlálóját pedig csökkenti eggyel, és amennyiben a számláló nullára csökken (vagyis más már nem használja azt a kódot), akkor az operációs rendszer eldobja azt. A végrehajtható programból be lesznek töltve az inicializált statikus illetve globális változókat tartalmazó adatszegmens (ha van ilyen). A fájldeszkriptor tábla olyankor módosítva lesz, ha valamelyik fájldeszkriptorhoz megadták, hogy `exec()` rendszerhívás végrehajtásakor le kell zárni. A signalok kezelését leíró táblázatok majdnem minden változtatás nélkül megmaradnak, csak olyankor kell módosítani, ha valamelyik signal-

hoz egy signal-kezelő eljárás címe volt eltárolva, ugyanis az újonnan betöltött folyamat címtartományában ki tudja mi van azon a helyen ....

Miutát egy folyamat befejeződik és végrehajtotta az `exit()` rendszerhívást, a kernel felszabadítja az általa lefoglalt adat és verem memóriaszegmenseket, csökkenti a kódszegmenst leíró `text` struktura referenciaszámlálóját, és ha ez nullára csökken, akkor felszabadítja a kódszegmensek által elfoglalt memóriaterületeket. Ezután lezárja a befejeződött folyamat még megnyitott fájldeszkriptorjait, és deallokálhatja a `user` strukturáját is. A folyamat exit-státusza (vagyis az az érték, amit az `exit()` rendszerhívás argumentumaként megadtak, ki lesz másolva a `proc` strukturából (ui. kicsivel korábban oda lett berakva) a processz-táblába, majd a szülőfolyamatnak küldve lesz egy `SIGCHLD` signal, jelezve neki, hogy meghalt egy gyermeke, ezután deallokálva lesz a folyamat `proc` strukturája. A gyermek ezután – már erőforrásait felszabadítva – bekerül a zombie folyamatok (processz-táblabeli állapota `SZOMB` lesz) közé egészen addig, amíg a szülője le nem kérdezi az exit-státuszát, majd utána a folyamat processz-tábla bejegyzése is törölve lesz.

A `wait()` rendszerhívás végrehajtása után a szülő folyamat megnézi a gyermekeinek a listáját, hogy valamelyik befejeződött-e (azaz állapota `SZOMB`-e), és ha igen, akkor visszaadja az exit-státuszát, és megszünteti a számára lefoglalt processz-tábla bejegyzést.

7.1.3 Ütemezési kérdések

A rendszermag feladatai közé tartozik az ütemezés megszervezése: a processzor megosztása a futásra kész folyamatok között – lehetőleg igazságosan (az igazságosság nem feltétlenül az egyenlő elosztást jelenti, hanem a folyamatok közt a fontosságuk szerinti elosztást). A kontextuscsere (vagyis a váltás az egyik folyamatról a másikra) vagy akkor lesz, amikor az aktuálisan futó folyamat időszelete lejár (vagyis mondjuk 50 ms-on át futott), vagy amikor a folyamat valamilyen I/O művelet elvégzését kéri a rendszermagtól, és a folyamat kénytelen leállni és várakozni addig, amíg a rendszermag a megadott műveletet elvégzi. Fontos megemlíteni, hogy amikor egy folyamat a rendszermag valamilyen szolgáltatását igénybe akarja venni, és a vezérlés átadódik a rendszermagba, azután a rendszermagban futó folyamat nem lesz megszakítva (vagyis nem történik kontextuscsere) egészen addig, amíg a folyamat vagy el nem hagyja a rendszermagot, vagy pedig önmaga hajlandó lemondani a CPU-használati jogáról (általában azért, mert valamilyen eseményre kell várnia). Erre azért van szükség, mert a rendszermag szolgáltatásait egyszerre több folyamat is igényelheti, és valamilyen módon meg kellett valósítani a konkurrens folyamatok kölcsönös kizárását. Az elég drasztikus megoldásnak tünhet, hogy a rendszermag használata közben a folyamatok nem szakíthatók meg, viszont a rendszermag általában úgy van elkészítve, hogy a folyamatok viszonylag hamar kiléphetnek belőle vagy elég hamar kénytelenek önszántukból a CPU-használati jogot (pl. egy I/O eszköznek munkát adva).

Amikor egy folyamat a rendszermagban tartózkodik, és kezdeményezi egy diszkszektor tartalmának a beolvasását, akkor várnia kell addig, amíg a megadott szektor be lesz olvasva a központi memóriába, és a rendszermag a rendelkezésére bocsátja a megadott blokkot. Ezt a várakozást a folyamat eltöltheti úgy is, hogy folyamatosan végrehajt olyan utasításokat, amelyekkel ellenőrizheti, hogy a rendelkezésére áll-e a megadott diszk-szektor, és ha nem, akkor tovább vár, viszont ez a megoldás CPU-idő pazarláshoz vezetne, hiszen azalatt, amíg egy folyamat várakozik, addig egy másik

folyamatot el lehet indítani (ez az időosztásos multiprogramozás hatékonyságának talán legfőbb alapja). Egy folyamat a rendszermagban a `sleep()` függvényhívással mondhat le a processzorhasználati jogáról (ennek a függvénynek az argumentumaként meg kell adni egy esemény-azonosítót – annak az eseménynek az azonosítóját, aminek a bekövetkezéséig a folyamat várakozni akar). Amikor egy esemény bekövetkezik (például beolvastak egy olyan diszk-szektort, amire valamelyik folyamat várakozik), akkor aki az eseményt előidézte (diszk-blokk beolvasáskor például az az eszközmeghajtó, amelyik a kérdéses blokkot beolvasta) végrehajt egy `wakeup()` függvényhívást, megadva az argumentumaként a bekövetkezett esemény azonosítóját, és ezután az összes olyan folyamat, amelyik a megadott eseményre várakozott újra futásra készen várja, hogy megkapja a következő időszeletét. Fontos látni, hogy egy eseményre egyidejűleg több folyamat is várakozhat, és ha egy esemény bekövetkezik, akkor az összes rá váró folyamat tovább fog futni (ez azt is jelenti, hogy ha például memóriahiány esetén két folyamatnak is szüksége van mondjuk egy-egy szabad memórialapra, akkor mindketten várakozhatnak arra az eseményre, hogy felszabadul egy korábban használatban levő memórialap, és miután felszabadult egy memórialap, akkor mindkét várakozó folyamat el lesz indítva, viszont miután az egyik folyamat – mondjuk az, amelyik előbb fut – megszerezte magának a felszabadult memórialapot, ismét előáll az a helyzet, hogy nincs szabad memórialap, és a másik – addig várakozó – folyamat ezt ellenőrizvén újból várakozásra kényszerül). Felmerülhet a kérdés, hogy hogyan nevezhetik az eseményeket, amiknek a bekövetkezésére várni lehet: tetszőleges egész szám megadható eseménynévként, de a kialakult konvenciók alapján egy, az eseménnyel kapcsolatos fontosabb, az eseményt egyértelműen azonosító rendszermagbeli adatszerkezet memóriabeli címét szokás esemény-azonosítóként megadni: ha például egy adott sorszámú diszk-szektor beolvasására várunk, akkor megadhatjuk annak a memóriarekesznek a címét eseményazonosítóként, ahová a megadott diszk-blokk beolvasását a kerneltől kértük (a megfelelő diszk-blokk buffer fejlécének a címét – ld. erről a későbbieket). A rendszermag sleep/wakeup mechanizmusának van egy hátránya is: a sleep művelet nem egyetlen atomi gépi utasítás végrehajtásából áll, ezért előfordulhat az, hogy az esemény bekövetkezését jelző wakeup-ot az eseményt előidéző rendszermagkomponens a sleep művelet befejeződése előtt meghívja, és mivel akkor még az van nyilvántartva, hogy senki nem vár az adott eseményre, ezért a wakeup nem jelzi az épp várakozni kezdő folyamatnak, hogy nem kell tovább várakoznia. Ezt a problémát úgy oldották meg, hogy az ilyen kritikus helyzetekben az ellenőrzést végző kódot és a sleep függvényhívást "mesterségesen" atomi utasítássá teszik azzal, hogy a végrehajtásuk idejére letiltják a hardver megszakításokat (ezzel senki sem szakíthatja félbe a működését).

7.2 A memóriakezelő implementációja

A memóriakezelés feladata a rendszermag memóriaszükségleteinek a kielégítése. Ez nyilván magába foglalja a a folyamatokat alkotó szegmensek ill. régiók számára való helylefoglalást, valamint az operációs rendszer belső feladatainak ellátásához szükséges memórialefoglalást. Láthattuk, hogy a kernel verem mérete fix, ráadásul nem is túl nagy, ezért azon nem lehet nagyméretű lokális változókat allokálni – ezért a kernel rákényszerülhet arra, hogy más forrásokból jusson a feladatának az ellátásához szükséges memóriához.

A memóriakezelő két fő komponensből áll: az egyik komponens feladata az

előbbiekben már megemlített régió típusú objektumok implementálása a processzor memóriakezelő hardverén (ez proceszortípustól függő kód; a különböző gyártmányú, de azonos processzorú gépek között ez egy hordozható kódnak tekinthető – ennek kell biztosítani a rendszermag lapozásos és szegmentált memóriakezelésének az alapjait). A másik komponens feladata a régió objektumok segítségével a rendszermag memóriaigényeinek a kielégítése.

Mint már említettem, egy folyamat virtuális memóriája különféle ún. régiókból lesz összerakva. Létrehozásakor egy folyamat virtuális címtartománya négy régiót tartalmaz:

- Egy végrehajtható kódot tartalmazó régiót
- Egy, az inicializált (kezdőértékekkel ellátott) globális változókat tartalmazó régiót
- Egy, az inicializálatlan (kezdőértékekkel nem ellátott) változókat tartalmazó régiót (ez a régió nem fix méretű; a mérete a folyamat igényeinek megfelelően bővíthető vagy csökkenthető). A program futása során allokált dinamikus memória (C programokban például a `malloc()` függvényhívással történő memóriaallokálással lefoglalt memória) ebben a régióban lesz lefoglalva.
- A végrehajtási vermet tartalmazó régiót.

Egy régió mérete tehát időben változhat, és méretükkel kapcsolatban egyetlen korlát az az, hogy a régióknak be kell férniük a (virtuális) memóriába.

Egy folyamathoz idővel újabb régiókat is lehet allokálni: például osztott könyvtárak (shared library) alkalmazásakor egy osztott könyvtár betöltése egy újabb programkódot, valamint egy újabb inicializált adatokat tartalmazó régió létrehozását jelenti az osztott könyvtárat alkalmazó folyamat virtuális címtartományában.

Megjegyezzük, hogy az itt bemutatott régió modell nagyon hasonlít az AT&T UNIX System V Release 3.0 operációs rendszerében alkalmazott megoldásra (ott nevezik a memória alotóelemeit régióknak) – igaz néhány helyen az egyszerűbb tárgyalás érdekében kis módosításokat végeztem.

7.2.1 A régióműveletek

Ebben a pontban röviden összefoglalom a régió objektumokon végezhető műveleteket:

- Egy új (üres) régió lefoglalása.
- Egy régió objektum megszüntetése.
- Egy régió tartalmának beágyazása egy folyamat virtuális címtartományába.
- Egy régió lecsatolása egy folyamat virtuális címtartományából.
- Egy régióról másolat készítése.
- Egy régió méretének a módosítása.
- A fájlrendszerből egy program betöltése egy régióba.

- Egy régió rögzítése a fizikai memóriába I/O műveletek idejére, illetve a rögzítés megszüntetése (erre azért lehet szükség, mert az intelligens diszk-kontrollerek a rájuk bízott feladatot (pl. egy diszk-blokk beolvasása) a "háttérben" is el tudják végezni, és meg kell akadályozni azt, hogy egy olyan lapot kilapozzanak, amire egy ilyen intelligens diszk-kontrollertől adatokat várunk (ui. a legtöbb diszk-kontroller ezt nem tudja "követni", és amikor beolvassa az adatokat, akkor már nem oda kell beolvasni, amit korábban megadtak).

7.2.2 A régió rendszer adatszerkezetei

A processz-tábla (valamint a `text` struktúra) tartalmazza egy régiólista címét. Egy ilyen régiólista egy eleme egy-egy folyamat virtuális memóriáját alkotó memóriaterületeket (régiókat) leíró régióleíró strukturára mutat. Egy adott régió leíró strukturájára több folyamat régiólistájáról is mutathat egy-egy pointer – ha az adott régió több folyamat virtuális címtartományának is része. A folyamatonként nyilvántartott régióleíró lista tartalmazza azt is, hogy egy adott régió a folyamat címtartományában hol helyezkedik el, így egy adott régió nem kell, hogy az összes folyamat virtuális címtartományában ugyanott helyezkedjen el.

Egy régióleíró objektum tartalmaz egy hivatkozást a régió tartalmát leíró memórialapok sorozatát leíró "laptáblára". Egy "laptábla" két fontos komponens-adatszerkezete a következő: a hardver memóriakezelő egységének a laptáblája (a hardver memóriakezelő egysége által előírt formában), valamint tartalmazza azoknak a mágneslemez-szektorok címének a sorozatát, ahova az adott régió tartalmát ki lehet lapozni memóriaszűke esetén (ezzel a virtuális memória méretét a virtuális memóriakezelés számára fenntartott lemezterület méretére korlátozza – de ez már implementációs részlet, amit máshogyan megoldva más korlátozásokhoz juthatunk, illetve kellően flexibilis megoldást találva kikerülhetjük a korlátozásokat).

Egy régió el van látva egy típus-azonosítóval, amely típus-azonosító a régió használati módjáról adhat további információkat. Egy régiótípus-azonosító egy konstans, melynek az értéke a következő konstansok közül kerülhet ki:

- `SHARED_TEXT` : Programkódot tartalmazó, nem módosítható tartalmú, több folyamat által is elérhető régió.
- `SHARED_MEMORY` : Adatokat tartalmazó, több folyamat által is elérhető régió.
- `PRIVATE` : Egy folyamat saját (más folyamatokkal nem megosztott) adatait tartalmazó régió.

Egy új folyamat létrehozásakor a virtuális memóriáját alkotó régiókat leíró lista le lesz másolva. Ekkor a `PRIVATE` típusú régióknak egy új példánya lesz létrehozva (átmásolva az eredeti régió tartalmát), a több folyamat által megosztottan használt régiókra új hivatkozások lesznek létrehozva. Egy folyamat megszüntetésekor azok a régiók meg lesznek szüntetve, amelyekre csak az az egy folyamat hivatkozik (vagyis a megszüntetenő folyamat hivatkozik rá, és csak egy folyamat hivatkozik rá).

7.2.3 A lapozás implementálása

A fizikai memóriát a rendszermag implementációk általában három fő részre bontják: az egyik rész a magát a rendszermagot a kód illetve adatszegmenseivel együtt tartalmazó rész; a másik rész a fizikai memória lapkereteiről nyilvántartott információkat tartalmazza (ez a memóriatérkép – eredeti angol nevén core map); a harmadik részt pedig maguk a fizikai memóriában levő lapkeretek alkotják. A rendszermag eddig elkészült implementációinak közös jellemzője, hogy a rendszermagnak és a core mapnak teljes egészében állandóan a fizikai memóriában kell lennie, azaz nem lapozható ki.

A lapozás implementációjának a központi adatszerkezete az előbb már említett core map. A core map minden egyes fizikai memóriában levő lapkerettel kapcsolatban nyilvántartja, hogy az adott lapkereten levő lap a virtuális memóriát tartalmazó diszken hol helyezkedik el (akkor, ha nincs a memóriában), azaz hova kell majd kilapozni; a core map tárolja azt is, hogy az adott lapkereten levő lap melyik régiónak a része; azt is tárolja, hogy a lap a r'egión belül hányadik lap. Ezenkívül a core map az éppen nem használt (szabad) lapkereteket egy listába fűzi, valamint minden egyes lapkerethez tárolja azt is, hogy az a lapkeret éppen szabad-e, valamint azt, hogy az adott lapkeret tartalmát nem szabad kilapozni a háttértárra (mivel például egy I/O művelet a háttérben dolgozik rajta). Megjegyezzük, hogy a core map lapkeretenként egy fix méretű adatszerkezet: a Berkeley UNIX-ban egy core map bejegyzés mérete 16 bájt – ennyi plusz információt kell tárolni a lapozás megszervezéséhez minden egyes 4096-bájtos lapkerethez (vagyis ezzel a memóriának kevesebb, mint az egy százaléka "veszik el").

A lapozást a rendszermag és egy felhasználói üzemmódban futó folyamat (a lapozó daemon) együttesen implementálják. A rendszermag tárolja a szabad lapkeretek számát, és ha a szabad lapkeretek száma egy küszöbérték alatt van (pl. az összes lapkeretek számának a harminc százalékát nem éri el), akkor a lapozó démon a core mapben tárolt információk alapján felszabadít újabb lapkereteket. A lapcsere algoritmusként az óra (ez lényegében FIFO) algoritmushoz hasonló kétmutatós óra algoritmust alkalmazzák. Az óra algoritmus onnan kapta a nevét, hogy modellezhető egy óra mutatójával, amely "végighalad" a core map-bejegyzéseken, és az összes nem szabad, és kilapozás ellen nem védett lapot kilapozhatónak nyilvánítja, ha pedig egy lapot az előző menetben kilapozhatónak nyilvánított, és az előző menet óta nem volt rá hivatkozás, akkor ki is lapozza. A kétmutatós óra algoritmusban az órának két "mutatója" van: az egyik mutató a másikat valamekkora (pl. a memória 25 százalékának megfelelő) rést kihagyva követi, és az elől lévő mutató által mutatott, nem szabad és kilapozás ellen nem védett lapot kilapozhatónak nyilvánítja a rendszer, majd amikor az óra második mutatója egy kilapozhatónak nyilvánított, és azóta nem használt lakeretet talál, akkor a megfelelő lapkeret tartalmát kilapozza a háttértárra. Vagyis az egy- illetve kétmutatójú óra algoritmus lényegében ugyanaz, csak egy lap kilapozhatóvá nyilvánítása és a kilapozása között eltelt idő két mutatós esetben kisebbé tehető.

7.2.4 A programbetöltés

Amikor egy folyamat végrehajt egy `exec()` rendszerhívást, akkor ellenőrzi azt, hogy a megadott végrehajtandó fájl valóban végrehajtató-e (azaz a hozzá tartozó rwx-bitek közt megvan-e a megfelelő x bit, valamint a kernel ellenőrzi a végrehajtható fájlokra vonatkozó néhány formai követelmény teljesülését). Ha végrehajtható, akkor betölti az egyes

logikai részeit a memóriába (létrehozva a szükséges régiókat a memóriában - lapozásos memóriekezelővel ellátott operációs rendszerek esetében gyakori a futó programoknak az "igény szerinti" belapozása, azaz a háttértárról csak azok a lapok lesznek belapozva, amelyekre rákerül a vezérlés, és csak akkor lesznek belapozva, amikor szükség lesz rájuk), és ha minden problémamentesen sikerüt, akkor a folyamat korábbi címtartományát eldobja (az azt leíró régiók közül megszünteti azokat, amelyekre más folyamatok nem hivatkoznak), és az újonnan beolvasott régiókat jelöli ki a folyamat végrehajtható kódját, adatait illetve végrehajtási vermét tartalmazó régióként, és elkezdi az új kódszegmens végrehajtását az "elejétől". Nyilván ha az új régiók betöltése sikertelen (például mert elfogyott a memória), akkor a betöltött új régiókat el kell dobni, és az `exec()` rendszerhívás hibakóddal kell, hogy visszatérjen. Ezért az nem jó megoldás, hogy a korábbi kód, adat illetve verem szegmenseket mindjárt a rendszerhívás megkezdésekor kidobjuk, mert lehet, hogy nem tudjuk az új régiókat a diszkről beolvasni.

A végrehajtható fájlokra vonatkozó egyetlen formai követelmény – az hagyományos ún. `a.out`[2] fájlformátum esetén – az, hogy a fájl elején kell lennie egy fejlécnek, ami térképként tartalmazza az egyes szegmensek helyét a végrehajtható programot tartalmazó fájlon belül. E fejléc szerkezete a következő C nyelvű struktúrával jellemezhető:

```
struct a_out_fejlec {
  unsigned long a_magic;
  unsigned long a_text;
  unsigned long a_data;
  unsigned long a_bss;
  unsigned long a_syms;
  unsigned long a_entry;
  unsigned long a_trsize;
  unsigned long a_drsize;
}
```

7.3 Az eszközmeghajtók implementációja

A rendszermag a különféle hardver perifériákkal foglalkozó részeket egy – a gépfüggősége miatt – jól elkülöníthető modulban tartalmazza: az eszközmeghatjtók moduljában. A legtöbb rendszermag implementáció az eszközöket (illetve az eszközök vezérléséért felelős rendszermagrészeket, az eszközmeghajtókat) két csoportba osztja: a karakteres elérésű és a blokkonkénti elérésű eszközök csoportjára.

Ha a rendszerhez egy új eszközt akarunk illeszteni, akkor meg kell írni az eszköz vezérléséért felelős eszközmeghajtót (ez néhány jól definiált, az eszközzel kapcsolatot tartó eljárás megírását jelenti), és a megírt eljárásokat egy, az eszközmeghajtó eljárásokat tartalmazó, globális vektorba be kell tenni (a karakteres eszközök esetén e vektor neve `cdevsw`, a blokkos elérésű eszközök esetén pedig `bdevsw`). Egy eszköz major device numberje az e vektorokon belüli indexével egyenlő.

[2]Van több ismert végrehajtható-fájl formátum (mint például az a.out, x.out, COFF, ELF), amelyeknek hsonló a szerepe, de néhány extra lehetőség meglétében különböznek egymástól. Ebben a leírásban az ún. `a.out` fájlformátumot, a legrégebb óta használt végrehajtható-fájl formátumot fogjuk megismerni. Ez megfelel a szokásos rendszermag-implementációk igényeinek, és könnyű is megérteni.

A blokk-elérésű eszközöket meghajtó eszközmeghajtó a következő eljárásokat mint belépési pontokat tartalmazza:

- `init()` : A rendszer indításakor lesz végrehajtva, feladata az eszköz pontos azonosítása.

- `open()` : Ez az eljárás az eszközmeghajtóhoz tartozó speciális fájlra vonatkozóan kiadott `open()` rendszerhívás hatására lesz végrehajtva. Általában ellenőriznie kell, hogy a hozzá tartozó hardver periféria a rendszer indításakor helyesen fel lett-e ismerve, illetve működik-e.

- `close()` : Ez az eljárás akkor lesz végrehajtva, ha az eszközt használó összes folyamat az eszközre vonatkozóan korábban megnyitott speciális fájlt lezárta. Ilyenkor az eszközmeghajtót és az eszközt egy "alapállapotba" kell hozni. Például ezt használják mágnesszalagok visszatekerésére, miután már senki sem használja (igaz, ritka a blokk elérésű mágnesszalagegység).

- `strategy()` : Egy eszközre való írási vagy az eszközről történő olvasási művelet kezdeményezését lehet vele kérni. Argumentumaként meg kell adni egy diszk-blokk buffer fejlécének a címét (ld. részletesebben a buffer cache leírásakor), és a hívó folyamat a buffer fejlécének a címét mint eseményazonosítót használhatja, ha várakozni akar az I/O művelet befejeződésére (amit az eszközmeghajtó többi része (pl. interrupt-vezérelten) megold).

- `ioctl()` : Eszközfüggő műveleteket lehet ebbe berakni. Az eszközhöz tartozó speciális fájlra vonatkozóan kiadott `ioctl()` híváskor lesz végrehajtva. Egy diszk esetében itt implementálhatók például a diszk meta-információinak (méret, partíciós tábla) a lekérdezésének és beállításának a műveletei.

A karakteres elérésű eszközöket meghajtó eszközmeghajtó a következő eljárásokat mint belépési pontokat tartalmazza:

- `init()` : A rendszer indításakor lesz végrehajtva, feladata az eszköz pontos azonosítása.

- `open()` : Ez az eljárás az eszközmeghajtóhoz tartozó speciális fájlra vonatkozóan kiadott `open()` rendszerhívás hatására lesz végrehajtva. Általában ellenőriznie kell, hogy a hozzá tartozó hardver periféria a rendszer indításakor helyesen fel lett-e ismerve, illetve működik-e.

- `close()` : Ez az eljárás akkor lesz végrehajtva, ha az eszközt használó összes folyamat az eszközre vonatkozóan korábban megnyitott speciális fájlt lezárta. Ilyenkor az eszközmeghajtót és az eszközt egy "alapállapotba" kell hozni. Például ezt használják mágnesszalagok visszatekerésére, miután már senki sem használja.

- `read()` : Karaktereket olvas be az adott eszközről.

- `write()` : Karaktereket ír az adott eszközre.

- `ioctl()` : Eszközfüggő műveleteket lehet ebbe berakni. Az eszközhöz tartozó speciális fájlra vonatkozóan kiadott `ioctl()` híváskor lesz végrehajtva. Egy mágnesszalag esetében itt implementálhatók például a szalagvége karakter felírása a szalagra, valamint a szalag meta-információinak (méret, partíciós tábla) a lekérdezésének és beállításának a műveletei.

- `select()` : Ellenőrzi, hogy van-e beolvasni való adat az eszközről; illetve ellenőrzi, hogy van-e hely az eszközön, ahova új adatokat lehet felírni.

- `stop()` : Ezt meghívva lehet szüneteltetni egy eszközre a karakterek kiiratását (például egy terminálon a ctrl-S karakterek megnyomásakor szünetelni fog a kiírás).

A rendszermag forráskódjának a lefordításakor gyakran több eszközmeghajtót befordítanak, mint amennyire ténylegesen szükséges lenne. A rendszermagba befordított eszközmeghajtókat úgy nevezik, hogy statikusan bekonfigurált eszközmeghajtók. A rendszer beindításakor az összes statikusan bekonfigurált eszközmeghajtó `init()` végre lesz hajtva, és ekkor ellenőrzi a kernel, hogy az eszköz hozzá van-e kapcsolva a géphez, lekérdezi a harver paramétereit, stb. Ezt a folyamatot nevezik autokonfigurációnak.

7.4 A buffer cache szerepe és implementációja

A blokk elérésű eszközök kezelését végző eszközmeghajtók és a gépfüggetlen "magasszintű" rendszermag között helyezkedik el egy szoftverréteg, a buffer cache. A buffer cache ún. buffer objektumokat kezel: egy buffer egy diszk-blokk (illetve más blokk elérésű eszközök "blokkjainak") tartalmát tartalmazza, és a legfontosabb célja az, hogy az adott blokkot használó illetve módosító összes folyamat ugyanazon a "példányon" dolgozzanak (vagyis ha az egyik folyamat megváltoztatja egy blokk buffer tartalmát, akkor az összes többi párhuzamosan futó és ugyanazt a blokkot használó folyamat azonnal a módosított diszk-blokk tartalmat lássa). Vagyis az összes folyamat egy adott diszk-blokk tartalmát a rendszerben csak és kizárólag a rendszer memóriájában egyetlen példányban tárolt blokk-bufferen keresztül érheti el. E bufferelésnek egy következménye az is, hogy a gyakran használt diszk-blokkok a memóriában lesznek tárolva, ezzel meggyorsítva a diszk-blokkhoz való hozzáférést (és mivel a memóriában tárolt diszk-blokk tartalmat hatékonysági okok miatt csak késleltetve írja fel a diszkre, ezért a rendszer érzékennyé válik az áramkimaradásokra – ez önmagában egy hátrány, de a bufferelés több előnye miatt általában nem tekintik nagy problémának).

A buffer cache kulcsfontosságú objektumai a buffer fejlécek, és a hozzájuk tartozó adatterületek (ez utóbbi adatterületekre példa egy-egy diszk-blokk tartalma). A buffer fejlécek egy-egy bufferrel kapcsolatban a következő információkat tartalmazzák:

- Annak az eszköznek az azonosítóját (major/minor device sorszám), amelyről a bufferhez tartozó adatterületen tárolt memóriaterület tartalmát betöltöttük.

- A bufferben tárolt adatterület címét (pl. szektor-sorszámát) azon az eszközön belül, amelyről a buffer tartalmát beolvastuk.

- Egy mutató egy adatterületre, ahol az eszközről beolvasott blokk/szektor tartalma van, illetve tárolva van, az adatterület mérete, és az, hogy hogy az adott adatterületen mennyi az "értékes" adat.

- A buffer állapotát leíró jelzőket (ld. később).

A buffer fejlécek több különböző listába is fel vannak fűzve különféle célokkal. A következőkben felsorolom azt, hogy milyen listák vannak, amelyre a buffer fejlécek fel lehetnek fűzve:

- **Foglalt bufferek** : ezen a listán azok a bufferek vannak, amelyeknek a tartalmának a fizikai memóriában **kell** lennie (ilyen például egy fájlrendszer szuperblokkja).

- **Cache bufferek** : ezen a listán azok a bufferek vannak, amelyeknek a tartalmát éppen senki sem használja (azért kerültek be, mivel tartalmukat valaki kicsit korábban használta); ha nem lesz más szabad buffer, akkor ezek "újrafelhasználhatók" lesznek helyszűke esetén.

- **Elévült bufferek sora** : ezen a listán azok a bufferek lesznek, amelyekre a későbbiekben nagy valószínűséggel nem fognak hivatkozni. Ha szükség lenne új bufferek beolvasására, és nincs más szabad buffer-fejléc, akkor innen el lehet venni bármelyik buffert.

- **Üres buffer fejlécek sora** : csak buffer fejlécek vannak, amelyek nem tárolnak adatot (nincsenek használva).

- **Egy eszközmeghajtónak átadott bufferek sora** : minden blokk elérésű eszköznek van egy sora, amelyen azok a bufferek (illetve fejléceik) vannak, amelyeken az adott eszközmeghajtónak valamit el kell végeznie (pl. a buffer tartalmát vissza kell írnia a diszkre, ...).

A következőkben leírom a buffer cache rendszer interfészét alkotó eljárásokat:

- Egy eszköz azonosítóját és annak egy blokkjának a címét megadva meghívhatjuk a `getblk()` vagy a `bread()` műveleteket, amely visszaad egy buffer fejlécet az adott eszköz adott blokkjához. A visszaadott buffer a `getblk()` hívása után nem feltétlenül tartalmazza a megadott eszköz-blokk tartalmát (csak akkor tartalmazza, ha a `B_DONE` jelző a buffer állapotjelzői között be van állítva). Ha a `bread()` műveletet hívjuk meg, akkor a megadott eszközről a megadott blokk tartalma be lesz olvasva, és a bufferhez kapcsolt adatterületre lesz írva. Mindkét eljárás meghívása után a buffer "használat alatt lévőnek" lesz nyilvánítva: addig, amíg ez a jelző ki nem lesz kapcsolva, más nem végezhet rajta semmit. Megjegyezzük, hogy a `getblk()` műveletet olyankor szokták meghívni, ha a diszk-blokk illetve a buffer tartalmát teljesen át akarják írni, így nincs szükség az előző tartalmára.

- A `breada()` művelet hasonlít a `bread()`-hez, de van még egy argumentuma, amelyben megadhatjuk, hogy még hány blokknyit olvasson előre a megadott egységen (de a további blokkok aszinkron lesznek beolvasva, vagyis az eljárást végrehajtó folyamat futása csak az explicite megadott blokk beolvasásáig lesz felfüggesztve).

- A `brelse()` műveletnek meg kell adni egy buffer fejlécének a címét, és ezután a buffer rákerül a szabad buffereket tartalmazó listákra.

- A `bwrite()` művelet a buffert annak az eszközmeghajtónak a buffer-sorára rakja, amelyről a buffer tartalmát beolvastuk, és a buffert visszairatja az eszközre oda, ahonnan beolvastuk. Vár, amíg a buffer nem lett visszaírva. Ezt akkor szokás meghívni, ha pontosan tudni akarjuk, hogy sikerült-e a visszaírás, és kíváncsiak vagyunk a visszaírás esetleges sikertelenségének az okára.

- A `bawrite()` művelet a `bwrite()`-hoz hasonlóan működik, de nem vár a buffer visszaírásáig. Ez biztosítja a maximális párhuzamosságot, mivel a visszaírás a folyamat további futásával párhuzamosan történhet.

- A `bdwrite()` művelet nem kezdeményez I/O műveletet a buffer tartalmának a visszaírására, de a buffert rárakja a szabad bufferek listájára, és "módosított"-nak jelzi. Ez azt jelenti, hogy mielőtt valaki újrafelhasználná (leszedné a szabad bufferet tároló listáról), a buffer tartalmát vissza kell írni oda, ahonnan beolvastuk. Ezt az eljárást akkor használják, ha nem biztosak abban, hogy a buffert már vissza kell írni (mivel például az utolsó rá vonatkozó `write()` rendszerhívás nem töltötte fel teljesen, így várható, hogy módosítani fogják).

Már említettem, hogy minden buffer fejléce tartalmaz egy jelzőt, amely a buffer állapotát írja le. E jelző értékét a következő biteknek megfelelő állapotjellemzők logikai "VAGY" kapcsolatával jellemezhetjük:

- `B_READ` : ez azt jelzi, hogy a buffert az eszközmeghajtónak átadtuk, hogy olvassa be a tartalmát a diszkről.

- `B_WRITE` : ez azt jelzi, hogy a buffert az eszközmeghajtónak átadtuk, hogy írja ki a tartalmát a diszkre. Megjegyezzük, hogy a `B_WRITE` konstans értéke nulla, vagyis az eszközmeghajtó ha nem találja a `B_READ` jelzőt beállítva, akkor `B_WRITE`-ot feltételez.

- `B_DONE` : ez a jelző ki lesz kapcsolva mielőtt a buffert átadjuk az eszközmeghajtónak valamilyen művelet elvégzésére, majd miután az eszközmeghajtó elvégezte a kijelölt műveletet, beállítja ezt a bitet.

- `B_ERROR` : a `B_DONE` jelzővel közösen jelezhetik, hogy a bufferen kért I/O műveletet az eszközmeghajtó befejezte, de a műveletet nem lehetett sikeresen végrehajtani.

- `B_BUSY` : azt jelzi, hogy a buffert valaki használja. Amíg ez a jelző be van állítva, addig más nem használhatja a buffert. Ha `getblk()` vagy `bread()` művelettel akarunk beolvasni egy használatban levő buffert, akkor ezek addig várnak, amíg ezt a jelzőbitet ki nem kapcsolja a buffer aktuális használója (a `sleep()`/`wakeup()` mechanizmussal vár), majd beállítja ezt a bitet, és visszaadja a buffert a kérőnek.

- `B_ASYNC` : ez a jelző akkor lesz beállítva, ha a buffert a `bawrite()` művelettel írták vissza. Ha ez a jelző be van állítva, akkor a bufferre vonatkozó I/O művelet befejezése után a bufferre vonatkozóan automatikusan végre kell hajtani a `brelse()` műveletet.

- `B_DELWRIT` : ez a jelző akkor lesz beállíva, ha a buffert a `bdwrite()` művelettel írták vissza. Ha a `getblk()` egy adott buffer keresésekor a buffert megtalálta, és ez a bit be van állítva, akkor a buffer tartalmát átadja az eszközmeghajtónak visszaírásra mielőtt visszatérne a hívóhoz.

7.5 A fájlrendszer implementációja

A UNIX-szerű operációs rendszerek rendszermagjai (így a BSD, a System V és a Linux is) a fájlrendszereket csak blokk-elérésű eszközökön, a buffer cache-nek a fájlrendszerkezelő és az eszközmeghajtó közé helyezve tudják létrehozni és kezelni – például egy floppyn, winchesteren, CD-ROM egységen .... A továbbiakban a fájlrendszert tartalmazó eszközt diszknek nevezzük, de ez alatt bármilyen blokk-elérésű eszközt is érthetünk. Ebben a pontban először áttekintjük a fájlrendszer implementációjával kapcsolatos diszken tárolt adatszerkezeteket, majd áttekintjük az ezeket módosító mechanizmusokat.

A fájlrendszer szerkezete és implementációja sokféle lehet; az eredeti UNIX Version 7 fájlrendszer még viszonylag sok korlátozást tartalmaz (például max. 14 karakter hosszú fájlnevek, nincsenek szimbolikus linkek). Lényegében ugyanezt a fájlrendszert tartalmazza a UNIX System V Release 3, és egy ugyanilyen szemantikájú fájlrendszert implementáltak a MINIX operációs rendszerben, amit átvettek a Linux operációs rendszerben is, és mint a Linux első (és egyben legprimitívebb) fájlrendszerét nagyon sokáig használták (kezdetben nem lehetett Linuxot futtatni MINIX operációs rendszer licensz nélkül, mivel a Linuxnak akkoriban nem volt fájlrendszer-létrehozó segédprogramja). Érdemes megjegyezni, hogy a MINIX fájlrendszere a diszken tárolt formátumában lényegesen különbözött a UNIX Version 7 fájlrendszerétől (míg a szabad blokkok illetve i-node-ok a Version 7 UNIX-ben lista adatszerkezetben voltak nyilvántartva, addig a MINIX bittérképeket vezetett be ezeknek az információknak a tárolására, ami lényegesen gyorsabbnak bizonyult a korábbi megoldásoknál). A 4.2BSD illetve 4.3BSD operációs rendszerek fájlrendszerének felépítése lényeges újításokat tartalmazott, leginkább a jobb hatékonyság elérése, valamint a korábbi fájlrendszer implementációk zavaró korlátozásainak a kiküszöbölése érdekében. Ez utóbbi kategóriába eső változások közül talán a szimbolikus linkek megjelenése, valamint a maximum 255 karakter hosszú fájlnevek megjelenése a legfontosabb. A jobb hatékonyságot pedig többek közt a diszk író-olvasó fejének a mozgásának a minimalizálásával, valamint a diszk hardver adottságainak (pl. interleaving) a kihasználásával érték el. A 4.2BSD ill. 4.3BSD új, ún FFS (Fast File System – magyarul: gyors (elérésű) fájlrendszer) fájlrendszerének alapján tervezték meg a Linux operációs rendszer **ext2** fájlrendszerét a hasonló hatékonyság és szolgáltatások biztosítása érdekében.

A modern rendszermag implementációk többféle fájlrendszertípust is tudnak egyszerre kezelni – általában egy ún. virtuális fájlrendszer rétegnek a fájlrendszer kezelő és a buffer cache közé történő beillesztésével: amikor egy fájlt használni/módosítani akarnak, akkor a virtuális fájlrendszer réteg megvizsgálja, hogy az adott fájl milyen felépítésű fájlrendszeren van (ekkor egy fájl szerkezetét leíró adatstrukturában (amit majd később még látni fogunk) a fájlt tartalmazó fájlrendszer típusát is tárolni kell), és a megfelelő fájlrendszer fájlkezelő műveletét implementáló eljárást hívja meg a feladat elvégésére. A továbbiakban egy igen elterjedt fájlrendszert fogunk megismerni: a 4.2BSD FFS fájlrendszerét – ez talán a legbonyolultabb és a leghatékonyabb az eddig elkészült fájlrendszer implementációk közül, de szerkezetükben és funkciójukban az összes többi eddig elkészült fájlrendszer hasonlít rá, így a többi fájlrendszerről is egy átfogó és pontos képet alkothatunk a FFS megismerése után.

7.5.1 A diszken tárolt adatszerkezetek

Egy fájlrendszer egy mágneslemezegységen vagy annak egy particióján helyezkedhet el – e két eset között a különbség csak annyi, hogy a teljes diszk kezelés szempontjából tekinthető egy olyan partíciónak, amely a teljes diszket lefedi, míg egy partíció tekinthető egy olyan "kisebb diszknek", amelyen a szektorok számozása nem nullánál kezdődik, és a mérete is kisebb, mint a teljes diszk mérete. A továbbiakban nem teszünk különbséget e két eset között az előbbi meggondolások miatt, és a fájlrendszert tartalmazó diszket vagy diszk-partíciót ezután logikai diszknek fogom nevezni.

A logikai diszk első szektora az ún. boot-szektor, ami tartalmazza az ún. elsődleges operációs rendszer betöltő programot. Az elsődleges operációs rendszer betöltő programnak a mérete nagyon kicsi, ezért általában csak annyit tud tenni, hogy betölti az utána levő szektorokról (előre lefixált elhelyezkedésű) másodlagos operációs rendszer betöltő programot. A másodlagos operációs rendszer betöltő program pedig már ismeri a logikai diszken tárolt fájlrendszer formátumát, és onnan tölti be a gyakran több megabyte méretű operációs rendszert (amelynek a fájlrendszerben egy jól meghatározott helyen kell lennie (a System V rendszereknél gyakran a `/unix` a neve, a BSD rendszereknél pedig `/vmunix` a neve; a Linux rendszerben pedig a leggyakoribb a `/vmlinux` vagy `/vmlinuz` fájlnév), de ez a "jól meghatározott" hely követelmény már csak egy fájlnevet rögzít, ahol az operációs rendszer magja megtalálható, nem pedig fix diszk-szektor sorszámokat feltételez).

A logikai diszknek az elsődleges illetve másodlagos operációs rendszer betöltő programját tartalmazó szektorokat követő része ún. cilinder-csoportokra van felosztva (a Linux operációs rendszer ext2 fájlrendszerénél hasonló a helyzet, de ott ezeket az egységeket blokk-csoportoknak nevezik): a fizikai diszk egymást követő valahány ún. cilindere (koncentrikus kör alakú hengerei) alkot egy cilinder-csoportot, és ezt az absztrakciót olyan meggondolásból hozták létre, és úgy alakították ki, hogy egy cilinder-csoporton belül a fizikai lemez olvasófejének minimális mozgást kelljen végeznie. A gyakorlatban egy cilinder-csoport mérete 8 megabájt körül van (de ennél lehet több is vagy akár kevesebb is). Minden cilinder-csoport szerkezete hasonló: van benne egy szuperblokk, egy cilindercsoport-blokk, egy i-node tábla valamint adatokat (fájlokat és directorykat) tartalmazó szektorok.

A szuperblokk tartalmazza a fájlrendszer méretére és szerkezetére vonatkozó információkat:

- A fájlrendszer mérete blokkokban
- Egy blokk mérete bájtokban
- A blokknál kisebb területfoglalási egység méretét (fragment[3]-méretet)
- A fájlrendszerben maximálisan allokálható i-node-ok száma, valamint a még szabad i-node-ok száma (ezekről az objektumokról korábban már írtam, de még szó lesz róluk részletesebben).

[3]Mivel a fájlrendszer blokk-mérete gyakran 4 vagy 8K, és a fájlok nagy része ezeknél jóval kisebb (egyes jelentése szerint egy "tipikus" rendszerben az átlagos felhasználói fáhl mérete kb. 1.5K), ezért ha egy 1.5K méretű fájlnak lefoglalunk egy 8K méretű blokkot, akkor a blokk kihasználatlan része (ebben a példában ez 6.5K) elveszne, ezért a FFS fájlrendszer tervezői bevezették a blokknál kisebb allokációs egységet, az ún. fragmentet. Megjegyezzük, hogy egy fájlnak csak az utolsó blokkja helyett lehet töredékblokkokat, fragmenteket allokálni.

- A még nem foglalt (szabad) fájlrendszerblokkok listájának első néhány elemét (az új BSD FFS fájlrendszerben ez átkerült a cilindercsoportonként kezelt cilindercsoport-blokkba, amit hamarosan részletesebben megismerünk).
- Az i-node-ok számát egy cilinder-csoporton belül.

Láthattuk, hogy minden cilinder-csoport tartalmaz az elején egy szuperblokkot: mivel a szuperblokk olyan fontosságú, hogy nélküle az egész fájlrendszerrel semmit sem lehetne kezdeni (ui. nem lehetne pontosan tudni olyan adatokat, mint például a cilinder-csoportok mérete), ezért az összes cilinder-csoport a szuperblokk egy-egy másolatát tartalmazza, és az összes szuperblokk-példányban ugyanazok az adatok találhatóak meg (vagyis ha az egyik szuperblokk megsérülne, akkor egy másik szuperblokkot használva a fájlrendszer tartalma még megmenthető).

Egy cilinder-csoportban a szuperblokkot a cilindercsoport-blokk követi, amelynek a feladata a cilinder-csoportra lokális jellemzők leírása. Ez a következő lényeges információkat tartalmazza:

- A cilinder-csoporton belül az adatblokkok száma.
- A cilinder-csoporton belül az i-node-ok száma.
- A cilinder-csoportban a szabad blokkok száma és elhelyezkedése.
- A cilinder-csoportban a szabad fragmentek száma és elhelyezkedése.
- A cilinder-csoportban a szabad inode-ok száma és elhelyezkedése.

Minden cilinder-csoport tartalmazza a cilinder-csoportot leíró blokk után a cilinder-csoport i-node tábláját. Már említettem, hogy az i-node a fájlrendszer objektumainak (fájlok, directoryk) az attributumait, a fájlrendszerbeli elhelyezkedését és egyéb jellemzőit tartalmazó objektum. Egy i-node tehát a következő fájl-attributumokat és egyéb jellemzőket tárol egy fájlról:

- A fájlra mutató linkek száma (a szimbolikus linkek nélkül!).
- A fájl rwx-bitjei.
- A fájl típusa (reguláris fájl, directory, blokk-speciális, karakter-speciális, szimbolikus link, vagy más)
- A tulajdonos felhasználói és csoportazonosítója.
- A fájl hossza.
- A fájl utolsó módosításának a dátuma.
- A fájl utolsó elérésének a dátuma.
- A fájl i-node-jának az utolsó módosításának a dátuma.
- A fájl elhelyezkedését leíró "térkép" (ennek a térképnek a szerkezetéről már írtam a 2.4. pontban).

A cilinder-csoport i-node tábla utáni blokkjai a fájlrendszerben tárolt fájlok blokkjait, valamint a fájlok elhelyezkedésének a feltérképezését leíró indirekt blokkok (ld. 2.4. pontban) tartalmát tartalmazza.

Korábban már azt is láthattuk, hogy a directoryk (könyvtárak) is fájlok, de a fájltípusukként nem a reguláris fájl van nyilvántartva, hanem a directory fájl, és azt is láttuk, hogy a tartalma is rögzített formátumú. A korábban már bemutatott (System V ill. MINIX) directory formátuma csak a legfeljebb 14-karakteres fájlnevek használatát tette lehetővé, és egy directory-bejegyzés (a directoryben levő egy-egy fájlról nyilvántartott adat) 16 bájt, fix hosszúságú volt. A 4.3BSD FFS fájlrendszerében egy fájl neve maximum 255 karakter hosszú lehet, és a directoryk szerkezete úgy módosult, hogy egy directorybejegyzés (ami egy fájlnevet tartalmaz, és egy directory pedig ilyen directory-bejegyzések sorozatából áll) változó hosszúságú lehet, és a directoryn belül kialakítottak egy teljes szabad-lista kezelést, amely listára az egyes directory-bejegyzések egymás után fel vannak fűzve, és ha egy új fájlt hozunk létre egy directoryban, akkor a rendszermag először megnézi, hogy valamelyik directory-bejegyzésben van-e annyi hely, hogy azt két bejegyzéssé osztva elfér-e az új fájl neve és i-node-sorszáma a szükséges lista-adminisztrációs adatokkal vagy sem. Ha van elég hely, akkor az új fájl felvitele így megoldható; ha nincs elég hely, akkor a directoryt tartalmazó speciális fájl részére újabb blokk lesz lefoglalva, és ott egy új listaelemként újabb directory-bejegyzés lesz allokálva.

Korábban már láttuk a linkek tárolásmódját és nyilvántartását a directory-bejegyzésekben. A BSD FFS fájlrendszerben bevezetett szimbolikus linkek tárolása attól lényegesen eltér: a szimbolikus linkek egy speciális szimbolikus link típusú fájlban vannak tárolva; a fájl tartalma ilyenkor maga a szimbolikus link által mutatott fájlrendszerbeli objektumnak a neve (vagyis ott nem i-node sorszám van tárolva). Ezzel lehetővé vált az, hogy egy szimbolikus link egy másik fájlrendszerben (pl. másik diszken tárolt) fájlra is mutathasson, nincsenek olyan korlátozások, mint a korábban bemutatott (ún. hard link típusú) linkeknél, hogy csak egy fájlrendszeren belül lehet linkeket létrehozni, de fájlrendszerek között "átmutató" linkeket nem lehet létrehozni. Természetesen ennek megvan az a hátránya, hogy előfordulhatnak olyan szimbolikus linkek, amelyek a "semmibe" mutatnak, mert például letörölték azt a fájlt, ahová a szimbolikus link mutat. A szimbolikus linkek konzisztenciájának az elfogadható időn belüli ellenőrzésének nincs kialakult módszere, nem is foglalkoznak vele.

A diszk-blokkok illetve az i-node tábla elemeit alkotó i-node-ok foglaltságát általában egy bittérképpel kezelik: az adatterület elején lefoglalnak annyi blokkot, amely legalább annyi bitet tartalmaz, ahány diszk-blokk (illetve i-node) van az adott cilider-csoporton belül. Ezen az adatterületen minden egyes objektumról (diszk-blokknak vagy i-nodenak) tárolva van az, hogy szabad-e vagy pedig értékes információt tartalmaz (azaz foglalt). Egy 0-ás bit jelzi azt, ha szabad; egy 1-es bit jelzi a foglaltságot. A UNIX System V Release 3.2 változatában illetve korábban még nem vették át a BSD UNIX FFS fájlrendszerét és a szabad blokkokat egy listában tárolták, ami lassabb megoldásnak bizonyult (a Linux ext2 fájlrendszerének az elődje, az ext fájlrendszerben is ezt a szabadlista kezelési módszert választották (annak ellenére, hogy a MINIX fájlrendszerben, amiből kiindultak, nem ezt használták), de miután látták a hatékonyság romlását, visszatértek a bittérképes módszerre).

7.5.2 Az adatszerkezeteken operáló műveletek

Az előző pontban áttekintettük a fájlrendszer szerkezetét a diszken. Ott láthattuk a fontosabb adatszerkezeteket, amiknek a kezelése és konzisztenciájának biztosítása a rendszermag fájlrendszer-kezelőjének a feladata. Ebben a pontban áttekintjük azt, hogy a rendszermag hogyan kezeli a fájlrendszereket, illetve milyen segédműveletek megvalósítását tartalmazza.

A szuperblokk kezelését általában a buffer cache szintjén, a buffer cache eljárásaival végzik: a fájlrendszereket a fájl-hierarchiába illesztő `mount()` rendszerhíváskor lesz a szuperblokk beolvasva, és egészen a megfelelő `umount()` rendszerhívásig bennmarad a memóriába. A szuperblokkokat a beillesztett fájlrendszerekre vonatkozó egyéb információkkal együtt (például azzal az információval együtt, hogy melyik directorynál van beillesztve a fájlrendszer tartalma) egy ún. mount táblában tárolják.

A használt (azaz valamelyik rendszermag-komponens által hivatkozott) i-node-ok tárolására is van egy tábla, az ún. i-node tábla. Az i-node tábla egyrészt tárolja a logikai diszken levő fájlrendszerről behozott i-node tartalmát (statikus jellemzőket), valamint az i-node dinamikus jellemzőit: azokat a jellemzőket, amiket csak egy "használatban levő" i-node esetén kell nyilvántartani. Az i-node dinamikus jellemzőit a következő adatok alkotják:

- Melyik eszközről lett beolvasva.
- Mi az adott i-node sorszáma az eszközön tárolt i-node-ok között.
- Az i-node-ra van-e egy másik fájlrendszer illesztve (ld. mount tábla).
- Ha az i-node-ra egy másik fájlrendszer rá van illesztve, akkor tárolni kell annak a fájlrendszernek a mount-tábla beli azonosítóját (általában az i-node sorszámát).
- Hány megnyitott fájl hivatkozik erre az i-node-ra.

Nyilvánvaló, hogy kell egy művelet egy i-node-nak az i-node táblába töltésére, és egy módosított i-node tartalmának a logikai diszken levő fájlrendszerre visszaírására. Ezeket általában úgy oldják meg, hogy a megadott logikai diszkről beolvassák azt a blokkot, amely az i-node-ot tartalmazza, módosítják a bufferben a tartalmát, majd a buffert szinkron módon visszaírják a diszkre (vagyis várnak, amíg vissza nem lesz írva). Amíg a buffer tartalmát manipulálják, addig a buffer használatára kizárólagos jogot formál és kap a rendszer megfelelő része.

Fontos látni, hogy ha egy i-node már benn van az i-node táblában, akkor nem szabad mégegyszer betölteni, hanem a rá vonatkozó hivatkozások számát kell eggyel növelni, és az i-node táblabeli előző példányát kell másoknak is használni (ui. a POSIX és más szabványok a fájlrendszer szemantikáját úgy definiálják, hogy ha egy folyamat valamilyen változtatást végez egy fájlon (ezzel a fájlt leíró i-node-on), akkor a változtatás után mindenki ugyanazt a módosított fájlt látja, és ez a követelmény így – az i-node újraolvasása nélkül – könnyen biztosítható; persze erre léteznek más (bonyolultabb) megoldások, de tudtommal senki sem használta őket).

Szükség van egy új (addig szabad) i-node-ot allokáló műveletre, illetve egy i-node deallokáló műveletre. Ezeknek a műveleteknek az eredményeképpen nemcsak az i-node táblában történik változás, hanem a logikai diszken (az azt tartalmazó adathordozón)

is; a megfelelő cilinder-csoport blokkokat aktualizálni kell (ez nyilván a megfelelő blokk beolvasásával illetve visszaírásával jár, de a jelenlegi implementációk ezeknek nem szoktak egy külön táblát létrehozni és kezelni). A MINIX fájlrendszerben mivel a bittérképek a szuperblokkal együtt voltak tárolva, ezért az i-node allokáló és deallokáló műveletek a szuperblokk objektum műveletei voltak.

A cilinder-csoport i-node tábla utáni részét, az adatblokkok területének a kezelését végző eljárásokat is meg kell írni: kell egy adatblokk-lefoglaló és egy adatblokk felszabadító eljárás. Ezeknek az eljárásoknak a megfelelő cilinder-csoport blokkot (az adatblokk-foglaltsági táblával) módosítaniuk kell az elvégzett művelet alapján.

7.5.3 Allokációs stratégiák

Az előbbiekben áttekintettük az FFS fájlrendszer főbb adatszerkezeteit. Ezen ismeretek birtokában már mindent meg tudunk csinálni egy fájlrendszerrel, de ahhoz, hogy ezeket a MINIX (vagy System V Release 3.2) rendszermagjának fájlrendszeréhez képest eléggé elbonyolított adatszerkezeteket optimálisan ki lehessen használni, kialakítottak egy-egy blokk- illetve i-node-lefoglalási stratégiákat. Ezeket ismertetem ebben a pontban.

Az optimalizálások általában a diszk olvasófejének a mozgás-pályáját próbálják meg minimálissá tenni bizonyos tipikus művelet-sorozatok alkalmával.

Például egy adott directory tartalmát listázó `ls` parancs a directoryban levő összes fájlt és directoryt (azok i-nodeját) elér annak érdekében, hogy azokról információkat gyűjtsön be (pl. egy fájl hosszát ki tudja írni, ...). Ezért az egy directoryban levő fájlok i-nodejaik a szülődirectoryval együtt ugyanabban a cilinder-csoportban vannak. Mivel nem lehet az összes fájl és directory ugyanabban a cilinder-csoportban, az is egy konvenció, hogy a directoryk i-nodejaikat az szülő directory i-nodeját tartalmazó cilinder-csoportba nem rakják, hanem az mindig egy másik cilinder-csoportba kerül (a megfelelő cilinder-csoportot úgy választják ki, hogy megnézik, hogy melyik cilinder-csoportban van a legtöbb szabad (nem használt) i-node, és ebben a cilinder-csoportban hoznak létre egy új directoryt). Természetesen ha valamelyik cilinder-csoportban nincs szabad (nem használt) i-node, akkor egy másik cilinder-csoportot kell választani.

Az adatblokkok elhelyezésére is vannak szabályok, méghozzá itt inkább arra törekednek, hogy egy fájl adatblokkjai abbad a cilinder-csoportban helyezkedjenek el, amelyben a fájl i-node-ja van. Mivel nagy fájlok esetén egy fájl összes adatblokkja nem kerülhet ugyanabba a cilinder-csoportba, ezért az is egy szabály, hogy egy fájl minden egyes fix nagyságú részét (pl. megabájtját) megpróbálják más-más cilinder-csoportokba elhelyezni, de ha nem sikerül, akkor persze feladják az optimalizációs törekvéseket, és keresnek megfelelő nagyságú szabad helyet, és ott helyezik el az adatokat.

7.5.4 Fájlnévről - i-nodera transzformáció

A fájlkezelő rendszer egyik gyakori feladata a fájl abszolut vagy relativ elérési neve alapján a fájlhoz tartozó i-node megkeresése. Ennek az algoritmusnak tehát az inputja egy fájlnév (abszolut vagy relativ), outputja pedig egy i-node hivatkozás illetve egy logikai érték, ami azt jelzi, hogy sikeres volt-e a transzformáció.

A transzformáció első lépéseként meg lesz határozva egy kezdő directory: abszolut fájlnevek esetén (ha a fájl elérési nevének első karaktere egy "/" jel) a gyökér-directory;

relativ fájlnevek esetén (vagyis ha a fájl elérési nevének az első karaktere nem egy "/" jel) pedig a folyamat munkadirectoryja.

A rendszermag ellenőrzi, hogy ez valóban egy directory-e, és valóban van-e rá keresési engedélyünk. Ha nem directory vagy nincs jogunk benne keresésre, akkor egy hibakóddal visszatér.

Ezután le lesz választva a fájl elérési nevének a következő komponense (a következő "/" karakterig tart), és a rendszermag megkeresi az ilyen nevű bejegyzést a kezdő directoryban, és hibakóddal tér vissza, ha nem találja; ha pedig megtalálja, akkor a directoryban a hozzá tartozó i-node sorszáma alapján be lesz töltve az i-node, és ezt az új i-node-ot mint kezdő directoryt felhasználva ugyanezt ismétli, amíg sikeres a transzformáció ("elfogyott a fájl elérési neve") vagy sikertelen a transzformáció. A sikertelenséget több tényező is okozhatja: vagy nincs olyan directory, amire a fájl elérési nevében hivatkoztak (az is lehet, hogy van olyan nevű directory bejegyzés, de az nem directoryra, hanem egy fájl típusú i-node-ra mutat), vagy nincs a directoryra keresési jog. Természetesen az is hiba, ha az inputként megadott fájl elérési névben még vannak komponensek, de a directory-hierarchiában már nem tudunk "lejjebb" menni (mert a továbbhaladáshoz szükséges komponens hiányzik).

Ha menet közben olyan i-node-hoz értünk, amelyre egy másik fájlrendszer van beillesztve, akkor a beillesztett fájlrendszer mount táblában levő szuperblokkja alapján be lesz töltve a beillesztett fájlrendszer gyökér-directoryja, és a keresés ott folytatódik. Visszafelé ugyanez megvan: ha a ".." (szülő directory) bejegyzésre egy gyökér-directory esetében hivatkoznak, akkor megkeresik azt, hogy az a bizonyos hivatkozott gyökér directory melyik i-node-ra van ráillesztve, és ennél az i-node-nál folytatódik a fájlnévről i-node-ra transzformáció.

A hard linkek kezelése semmi problémát nem okoz az algoritmusunknak; a szimbolikus linkek feldolgozása pedig csak annyi nehézséget okoz, hogy vigyázni kell, nehogy végtelen ciklusba keveredjen a transzformációs algoritmus valamilyen véletlenül vagy szándékosan rosszul létrehozott szimbolikus link miatt. Ezt úgy oldják meg, hogy korlátozzák az egy fájlnév i-node-ra transzformálása során maximálisan előfordulható szimbolikus linkek számát.

7.5.5 A rendszerhívás interfész

A fájlrendszerkezelő kód egy jól szeparálható részét alkotják a rendszerhívásokat feldolgozó eljárások. Ezek az eljárások lesznek meghívva olyankor, amikor egy alkalmazás egy rendszerhívást hajt végre, és ezek hívják meg az alacsonyszintű i-node és más műveleteket a rendszerhívás feladatának elvégzéséhez.

Azok a rendszerhívások, amelyek fájlokat megnyitnak illetve létrehoznak (általában ezek az `open()` illetve a `creat()` rendszerhívások), általában egy fájlnevet várnak az argumentumaikban, és egy – korábban már említett – fájldeszkriptorral térnek vissza. Azok a rendszerhívások, amelyek egy már megnyitott fájlon végeznek valamilyen műveletet, a megnyitott fájl fájldeszkriptorát várják általában első argumentumukként. Láttuk már, hogy ezek a fájldeszkriptorok egész (C nyelven `int`) típusúak, és ha például kiirattuk az értéküket, akkor láthattuk, hogy általában 0-tól kezdődően lesznek számozva, és van egy maximális értékük is (legfeljebb ennyi fájlt hozhat létre egy folyamat). Már említettem, hogy a folyamat `user` strukturája tartalmazza azt a táblázatot, amely a fájldeszkriptorokhoz a globális fájl tábla egy elemét rendeli. Ebben a globális táblában az

összes megnyitott fájlról nyilvántartanak bizonyos információkat (például azt, hogy hány folyamat hány fájldeszkriptora hivatkozik arra a táblaelemre), de itt van nyilvántartva az is, hogy mi az aktuális fájl-pozíció a fájlon belül (és ha végrehajtunk egy `read()`, `write()` vagy egy `lseek()` rendszerhívást, akkor ez az ebben a táblában tárolt fájl-pozíció alapján tudja, hogy hol tartunk a fájl feldolgozásában illetve ezt a táblázatot megfelelően aktualizálni is fogja). Megjegyezzük, hogy ha kétszer egymás után `open()` rendszerhívással megnyitunk egy fájlt, akkor a globális fájl táblában két táblaelem fog létrejönni; ha viszont `dup()` rendszerhívással egy fájlhoz létrehozunk egy új fájldeszkriptort, akkor az eredeti fájldeszkriptorhoz tartozó globális fájltábla-elemhez fog létrejönni egy új hivatkozás az új fájldeszkriptoron keresztül.

Megjegyezzük, hogy a socket rendszerben egy fájldeszkriptorhoz nem csak egy globális fájltáblabeli elem tartozhat, hanem egy – hasonló célú – globális socket-táblabeli elem is; ezzel ebben a kiadásban nem foglalkozunk.

7.6 A kommunikációs alrendszer implementációja

Ebben a fejezetben a rendszermag két alapvető folyamatok közötti kommunikációs eszközét, a `signal()`-okat, valamint a pipe-okat fogjuk áttekinteni.

Egy signal generálásakor[4] a `proc` strukturába bejegyzésre kerül az, hogy milyen signalt küldtek a folyamatnak. Ha egy folyamat aközben kap signalt, miközben egy rendszerhívást hajt végre, akkor a signal-kezelő csak azután lesz végrehajtva, miután a rendszermagbeli futás befejeződik (mielőtt a folyamat végrehajtaná a visszatérést a felhasználói programba, ellenőrzi, hogy küldtek-e neki egy signalt); egyéb esetben a signal-kezelő azonnal végrehajtható.

A `pipe()` rendszerhívás egy csővonalat hoz létre, a hozzá tartozó fájldeszkriptor-táblabeli bejegyzésekkel együtt. A pipe objektumok általában olyan fájlokként vannak implementálva, amely fájlra vonatkozóan nincs hivatkozás a fájlrendszerben egy directory-bejegyzésből sem. Természetesen tartozik hozzá egy i-node (amit általában a gyökér-directoryt is tartalmazó gyökér-fájlrendszerben allokálnak), amelynek a mérete gyakran korlátozva van (tipikus méretkorlátozás az, hogy a pipe-ok tartalmát képező adatok tárolására csak direkt blokkokat használnak, nem használnak egyszeres, kétszere illetve háromszoros indirekt blokkokat). A pipe-ba írt adatokat a létrehozott i-node-hoz tartozó adatterületen tehát el lehet tárolni; vagyis amikor adatokat írunk egy pipe-ba, akkor a rendszermag ellenőrzi, hogy a felírandó adatok befférnek-e a pipe-ba, és ha igen, akkor beírja (ezzel a pipe méretét megfelelően módosítja); ha pedig nincs elég hely, akkor vagy beír valmennyi adatot a pipe-ba és a `write()` rendszerhívás visszaadja, hogy mennyit írtak a pipe-ba, vagy pedig visszatér azzal, hogy blokkolna a pipe-ba írás. A globális fájltáblában két elem van allokálva: az egyik az írási oldalhoz, a másik pedig az olvasási oldalhoz. Amennyiben az í ró folyamatok teleírják a pipe-ot, akkor az újabb íróknak várniuk kell, amíg az olvasók nem olvassák ki a pipe-ban levő összes adatot, majd a pipe olvasása illetve írása újrakezdődhet – úgy, hogy mind az írási, mind pedig az olvasási fájlpozíció nullára lesz állítva.

[4] Egy signalt vagy a `kill()` rendszerhívással egy másik folyamat, vagy a rendszermag, vagy pedig a felhasználó a ctrl-C billentyűk lenyomásával generálhat.

7.7 Kérdések

- Vajon miért nem lehet hard linkkel fájlrendszerek közötti linket létrehozni?
- Vajon a pipe-okhoz tartozó i-node-okat miért a gyökér-könyvtárat is tartalmazó gyökér-fájlrendszerben szokás létrehozni?

Fejezet 8

A UNIX SYSTEM V STREAMS programozása

Ebben a részben a kommunikációs modulok implementálásának az AT&T által kialakított eszközét: a STREAMS-et mutatjuk be. **Megjegyzés: az ebben a fejezetben leírt információk érdekesek, viszont egyrészt a többi fejezethez kapcsolás, másrészt a leírás részletessége (esetleg még a pontossága is) kívánnivalókat hagy maga után. Ezt eredetileg egy külön leírásként készítettem (az előző fejezetek elkészítése előtt kb. 2 évvel), most csak a teljesség kedvéért raktam bele ebbe a leírásba.**

8.1 Bevezetés

A STREAMS rendszert Dennis Ritchie készítette az ATT Bell Laboratóriumában azzal a céllal, hogy egy moduláris I/O rendszer kiépítésére használható eszközzel bővítse a UNIX rendszert. A STREAMS első kommersz változata a UNIX System V Release 3 operációs rendszerrel lett a fejlesztők és a felhasználók rendelkezésére bocsátva. Ma a STREAMS rendszert használják sok helyen a UNIX terminálok vezérlésére, hálózati protokollok implementálására és ez az alapja számos más felhasználói software rendszernek. Itt említjük meg, hogy a STREAMS segítségével tudták implementálni az ATT UNIX System V Release 4.0 rendszerben a 4.3BSD UNIX rendszerben használt socketeket (a fenti két operációs rendszerben a sockets rendszer `teljesen kompatibilis` egymással, így a Berkeley UNIX sockets rendszerhívásait használó programokat könnyebben átvihetjük az ATT UNIX rendszerekre). A STREAMS mechanizmust csak az ATT UNIX tartalmazza, a 4.3BSD UNIX-ban ez nincs implementálva. Ott is van lehetőség hasonló struktúrák felépítésére, de ott ez kissé nehézkesebben megy.

8.1.1 Alapfogalmak

A *STREAMS* egy UNIX kernelbe beépített mechanizmus, mely lehetővé tesz egy kétirányú kapcsolat kiépítését a felhasználói programok és a karakteres, adatkommunikációs *STREAMS device driverek* között. Eredetileg a UNIX terminálok vezérlésére alakították ki, de később alkalmazhatónak bizonyult hálózati protokollok implementálására is. (A Release 4 UNIX már STREAMS terminál-drivereket használ.) A STREAMS device driverek adatokat közvetíthetnek a felhasználói programok és a hardware berendezések között, de vannak speciális driverek is, például a *multiplexer driverek*, amelyek legtöbbször nem állnak a hardware perifériákkal közvetlen kapcsolatban.

Az adatáramlás a driver és a felhasználói program között egyszerre két irányban folyhat: *lefelé* a felhasználótól a driverig (ezt nevezik *write oldalnak*) és *felfelé* a drivertől a felhasználó irányába (ezt nevezik `read oldalnak` is). Mindkét irányú adatáramot (`streamet`) egymás mögé rakott sorok (*queuek*) segítségével implementálják. Egy stream a STREAMS device drivertől indul és a *stream-fejben* végződik. A stream-fej tartja a kapcsolatot a felhasználó és a stream alsóbb részei között. Minden egyes STREAMS driverre vonatkozó `open` rendszerhívás egy új stream-fejet hoz létre (amennyiben a driver ezt támogatja).
A driver és a felhasználói program közé beilleszthetünk ún. *STREAMS modulokat*, amik se nem driverek, se nem stream-fejek (vagyis egy STREAMS modult nem szokás például `open` rendszerhívással megnyitni). Feladatuk a rajtuk keresztülmenő adatok valamilyen módosítása - például protokollinformációkkal kiegészíthetik azokat, vagy az egyes modulok a nekik szóló információkat kivehetik a sorból. Ha a STREAMS drivert megnyitjuk, akkor létrejön a stream-fej. A modulokat ezután csak közvetlenül a stream-fej alá rakhatjuk az `ioctl` rendszerhívással (a megfelelő paraméterek beállításával).
A STREAMS driverek teljesen a kernel területén helyezkednek el, ezért ezek írásakor különösen kell vigyázni, nehogy valami hibát csináljunk!

8.1.2 A STREAMS előnyei

A STREAMS segítségével megírt programok előnyei: egyszerűbb szerkezetűek (a feladat szintenkénti megoldását is támogatja), könnyen alkalmazkodnak bármilyen konfigurációhoz, és hordozhatóak. Felhasználható a hagyományos UNIX karakteres device driverek helyett, és a folyamatok közötti kommunikáció megoldására is.
Egy streamet dinamikusan konfigurálhatunk a futásidőben, ezzel szemben egy hagyományos UNIX device driver a futásidőben már kevésbé (vagy egyáltalán nem) megváltoztatható. (Lehetőség van arra, hogy egy hagyományos UNIX karakteres device drivert például `ioctl` hívásokkal módosítsunk, de ez sokkal áttekinthetetlenebb lenne, mint az azonos feladat STREAMS megoldása.)
Az egyes modulok kicserélhetőek, így ugyanazt a softwaret alkalmazhatjuk többféle konfigurációban is. A STREAMS jó eszközöket nyújt például a hálózati softwareek hardwarefüggő és hardwarefüggetlen részének elkülönítéséhez - a felsőbb szinteket már teljesen hardwarefüggetlenül kódolhatjuk, használhatjuk.

8.1.3 A STREAMS rendszer vezérlése

A STREAMS rendszert a rá vonatkozó `ioctl` hívásokkal vezérelhetjük. Ennek prototípusa a következő :

```
int ioctl(int fd,int command,arg);
```

Itt `fd` egy nyitott STREAMS driverre vonatkozó filedeszkriptor. A `command` paraméter tartalmazza a végrehajtandó művelet kódját, és ettől függ az `arg` paraméter értéke. A rendszerhívás során fellépett hibákat (ha az *üzenet*et nem tudta átadni a stream-fej mögött levő modulnak) a szokásos módon jelzi. (Lásd erről részletesebben a STREAMIO leírást az egyes hibaüzenetekről!) Az `ioctl` hívás után az `errno` változó értéke `EINVAL` lesz, és a hívás nem hajtódik végre, ha az `fd` által specifikált stream már hozzá van kapcsolva egy multiplexer driverhez (ld. később), vagy a `command` paraméter

tartalma nem egy jó STREAMS parancs-érték.

A következőkben a leggyakrabban használt STREAMS `ioctl` parancsok lesznek ismertetve :

- `I_PUSH` : Befűz egy modult közvetlenül az `fd` által megadott stream stream-feje alá. Az `arg` paraméter a befűzendő modul nevére mutató karakter-pointer. Ezután meghívódik a befűzött modul open (megnyitó) rutinja. Hiba esetén az `errno` változó lehetséges értékei :
  - EINVAL : A megadott modulnév nem ismert a kernelen belül.
  - EFAULT : Az `arg` a program címtartományán kívülre mutat.
  - ENXIO : Open rutin hibát jelez, vagy `hangup`-ot kapott az `fd`-vel megadott stream.
- `I_POP` : Leszedi a megadott stream tetején levő modult. A hívásban `arg` értéke 0 kell legyen. Hiba esetén az `errno` változó lehetséges értékei :
  - EINVAL : Nincs már modul ennek a streamnek a stream-feje alatt.
  - ENXIO : `Hangup`-ot kapott az `fd`-vel megadott stream.
- `I_STR` : Egy STREAMS `M_IOCTL` (ha szükséges, akkor utána még egy `M_DATA`) üzenetet generál az alapján, amire az `arg` paraméter mutat, és elküldi a lefelé menő streamen. A felhasználó így küldhet `ioctl` hívásokat a moduloknak és a drivereknek. A rendszer vár addig, amíg az üzenetet feldolgozó modul visszajelzést ad arról, hogy sikeres volt-e az ioctl hívás. Ha egy megadott időn (default=15 sec.) belül nem érkezik visszajelzés, akkor az ioctl hívás timeout hibával megszakad. Az `arg` paraméter egy `strioctl` strukturára mutat. Ez tartalmazza a következő mezőket :

```
int ic_cmd;    /* Milyen ugyben kuldjuk ezt ? */
int ic_timout; /* Mennyi ido mulva lesz timeout ? */
int ic_len;    /* A lekuldendo adatterulet hossza */
char *ic_dp;   /* Pointer az elkuldendo adatteruletre */
```

  Az egyes mezők jelentése a következő :

  - `ic_cmd` : A driver (vagy modul) ez alapján tudja meg, hogy mit kell csinálnia.
  - `ic_timout` : Megadja, hogy maximum mennyi ideig kell várakozni a modul (vagy driver) válaszára, vagyis mennyi idő múlva következzen be a timeout. Ennek értékei a következők lehetnek :
    - `-1` : végtelen sokáig kell várni.
    - `0` : a rendszerben defaultnak számító ideig kell várni.
    - `>0` : a paraméterben megadott ideig kell várakozni a válaszra.
  - `ic_len` : Az ioctl hívás előtt megadja, hogy milyen hosszú a streamen leküldendő ioctl-hez kapcsolódó adat hossza. Az ioctl hívás után a driver (ill. modul) által felküldött válasz hosszát tartalmazza (byteokban mérve).

– `ic_dp` : Arra az adatterületre mutat, ahol a streamen leküldendő információ van. A hívás befejeződésekor ide fogja a rendszer beírni az üzenetet feldolgozó modul által visszaküldött választ, így e terület nagysága legalább akkora legyen, mint a visszaküldhető leghosszabb válasz nagysága.

Ha az ioctl hívás sikertelen (ioctl visszatérése értéke : -1), akkor a hiba okát az `errno` változó tartalmazza. Ennek lehetséges értékei a következők :

– ENOSR : A stream-fej nem tud lefoglalni memóriaterületet az ioctl üzenetnek, mert nincs elég memória.
– EFAULT : Valamelyik pointer (`arg` vagy `ic_dp`) által meghatározott memóriaterület a process memóriatartományán kívül esik.
– EINVAL : Vagy az `ic_len` által megadott hossz nem esik az adott rendszeren megengedett tartományba, vagy a `ic_timout` értéke -1-nél kisebb.
– ENXIO : `Hangup`-ot kapott az `fd`-vel megadott stream.
– ETIME : A stream-fej a `ic_timout` paraméterben megadott időn belül nem kapott választ, a hívás timeout miatt befejeződik.

8.1.4 A STREAMS üzenettípusai

Üzeneteknek nevezzük a modulok láncán fel-le menő információkat, hibaüzeneteket, stb. Egy üzenet (message) egy vagy több *üzenetblokkból* áll. A STREAMS rendszerben egy üzenetblokk és az adatblokkok felépítését a következő strukturák tartalmazzák :

```
struct msgb {
     struct msgb *b_next;   /* queue-n kovetkezo message */
     struct msgb *b_prev;   /* elozo message a queuen */
     struct msgb *b_cont;   /* tovabbi messageblokkok */
     unsigned char *b_rptr; /* elso hasznos adatbyte*/
     unsigned char *b_wptr; /* utolso hasznos byte utani adatbyte */
     struct datab *b_datap; /* Adatblokkra pointer */
};

typedef struct msgb mblk_t;

struct datab {
     struct datab *db_freep; /* Belso hasznalatra */
     unsigned char *db_base; /* A buffer elso bytejara mutat */
     unsigned char *db_lim;  /* A buffer utolso utani byteja */
     unsigned char db_ref;   /* Hany uzenet mutat erre az adatra */
     unsigned char db_type;  /* Uzenettipus */
     unsigned char db_class; /* Belso hasznalatra */
};

typedef struct datab dblk_t;
```

Megjegyzés: az egyes üzenetek nem biztos, hogy a teljes adatblokkot lefoglalják. Azt, hogy egy üzenet értékes része az adatblokkon belül hol kezdődik az üzenetblokknak a `b_rptr` mezőjéből tudható meg. Az üzenetblokk `b_wptr` mezője pedig az adatblokk utolsó értékes byteja utáni bytera mutat.

A STREAMS megengedi az üzenetek *osztályozását*. A különféle üzenettípusokat felhasználva világosabb szerkezetű programokat írhatunk. Ekkor a STREAMS szolgáltatást végző rutin leggyakrabban csak egy üzenettípusok szerinti elágazást tartalmaz, és egy-egy ág egy-egy üzenettípus feldolgozásáért felelős. A STREAMS nagyon sokféle üzenettípust ismer. Egy `mblk_t *bp;` módon deklarált üzenet típusát a `bp->b_datap->db_type` kifejezés adja meg.

- `M_CTL` : A STREAMS modulok egymás közti illetve driver és modulok közti protokollinformációk tartoznak ebbe az üzenettípusba. A felhasználói programok nem küldhetnek lefelé ilyen típusú üzeneteket, mivel ezeket a stream-fej kiszűri.

- `M_DATA` : Adatokat tartalmazó üzenetek. A felhasználó ezeket a `putmsg` és `write` rendszerhívásokkal írhatja bele egy streambe, és a `getmsg` és `read` rendszerhívásokkal olvashatja ki a streamből.

- `M_DELAY` : Ezzel az üzenettel kérheti valamelyik STREAMS modul a drivertől az output késleltetett kiadását (például azért, mert nagyon lassú az output periféria). Az üzenet formátuma nincs pontosabban meghatározva, a programozó döntheti el, hogy hogyan akarja felhasználni. A felhasználói programok nem küldhetnek lefelé ilyen üzeneteket, mivel ezeket a stream-fej kiszűri.

- `M_IOCTL` : A stream-fej a felhasználó `ioctl` hívásait ilyen üzenet formájában továbbítja az alatta lévő modulokhoz. Az ioctl-ben ekkor az `I_STR` parancsot kell megadni. Az a modul, aki egy ilyen üzenetet feldolgozott, köteles ezt a stream-fej irányába nyugtázni, mert a stream-fej addig nem továbbít üzeneteket, amíg nem biztos benne, hogy az ioctl hívás hibátlanul lement.

- `M_PROTO` : Protokoll-információkat a hozzá tartozó adatokkal együtt tartalmazó üzenetek. A felhasználó ezeket `putmsg` rendszerhívással írhatja a streambe, és `getmsg` rendszerhívással olvashatja ki a streamből. Ezek az üzenetek *nem kezelhetők read/write hívással*!.

- `M_PCPROTO` : Szintén protokoll-információkat tartalmaz, de *magas a prioritása*. A felhasználó ezeket `putmsg` rendszerhívással írhatja a streambe, és `getmsg` rendszerhívással olvashatja ki a streamből. Ezek az üzenetek *nem kezelhetők read/write hívással*!. A normál (alacsony prioritású) protokolladatoktól ezeket a létrehozásukkor egy flaggel különböztethetjük meg a `putmsg` hívásban. A hívás szintaxisa:

```
putmsg(fd,ctlptr,dataptr,flags)
int fd;
struct strbuf *ctlptr;
struct strbuf *dataptr;
int flags;
```

  Ha a `flags` paraméter `RS_HIPRI`-re van állítva, akkor az üzenet magas prioritású üzenet lesz. (Egyébként a `flags` paraméter értéke : 0.) Lásd erről részletesebben a `UNIX Programmer's Reference Manualt`.

- `M_IOCACK` és `M_IOCNAK` : Ezek az üzenetfajták az `ioctl` hívások nyugtázására valók. Az `M_IOCNAK` üzenet egy hibakódot visz magával a stream-fej irányába (csak hibakódot lehet így a stream-fej irányába küldeni, válaszadatokat nem). A `M_IOCACK` válaszadatokat is küldhet felfelé (ez a pozitív visszajelzés).

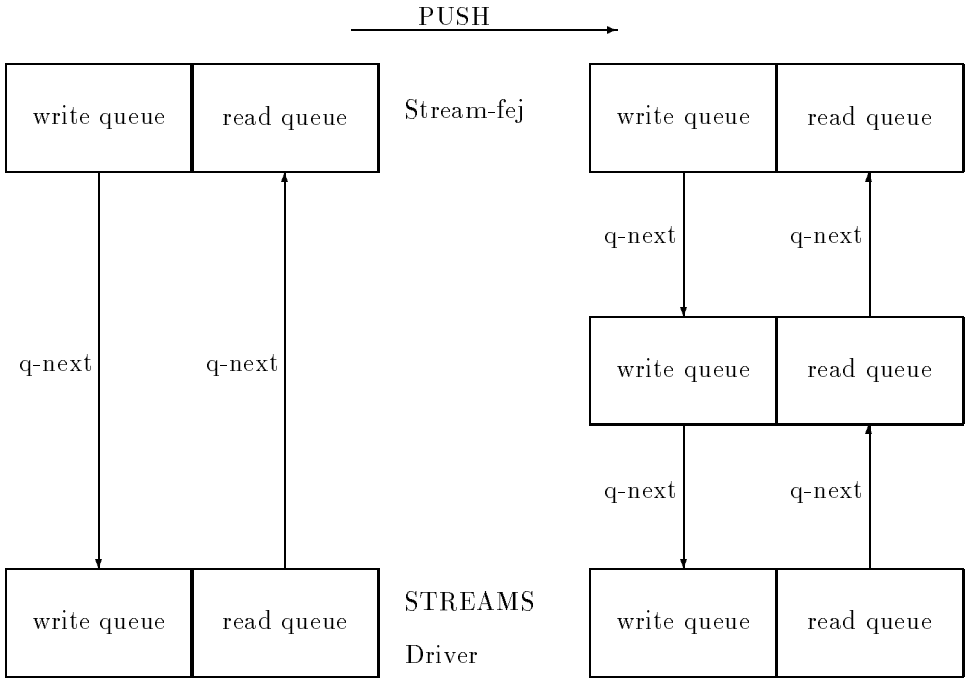
- `M_ERROR` : Ezt az üzenetet a modulok vagy a driverek küldhetik felfelé a stream-fej irányába, ha a lefele haladó queuen valami hiba van. Ha ez az üzenet eléri a stream-fejet, akkor az ezután kiadott rendszerhívások a `close` és a `poll` kivételével hibával fejeződnek be, és a rendszer az üzenet első bytejában megadott hibakódot adja vissza az `errno` változóban hibakódként. A `poll` rendszerhívás a `POLLERR` hibával tér vissza. Végül egy `M_FLUSH` üzenet lesz a lefelé menő streamen elküldve.

- `M_SIG` és `M_PCSIG` : Signalokat lehet a stream-fejen keresztül bizonyos (például signalokra várakozó) folyamatoknak küldeni. Az `M_PCSIG` a magas prioritású.

- `M_FLUSH` : Egy streamen lévő modulok mindegyikét arra utasítja, hogy a queue-jaikat ürítsék ki. Minden modulnak és drivernek kezelnie `kell` ezt az üzenetet. (Erre megoldást jelent például a kernel `flushq()` rutinjának a meghívása.) Ez egy magas prioritású üzenet.

8.1.5 Egy STREAMS-et használó program

A következő program az előbbieknek a használatát mutatja be - inkább a programozó szemszögéből. A program először megnyitja a /dev/bcndm0 STREAMS drivert, ráhelyezi a birk nevű STREAMS modult, majd ezen I/O műveleteket végez. (A modul ill. driver implementációja később lesz ismertetve.) Ez azt jelenti, hogy a megnyitott fileba írt illetve az onnan beolvasott adatok (*üzenetek*) keresztülmennek a birk nevű modulon is, ami a feladatának megfelelő dolgokat elvégezheti rajtuk. Látható, hogy a kernelben levő modulokat nevükkel lehet azonosítani (a modulnevek bekerülnek egy `fmodsw` kernel táblázatba). A program végül leveszi a driverről a birk nevű modult, és lezárja a megnyitott filet.
A következő ábra bemutat egy streamet az `I_PUSH` előtt és után.




```
#include <stropts.h>     /* Konstansok miatt kell */
#include <fcntl.h>

main()                   /* STREAMS pelda */
{
  char buf[512];
  int fd,count;

  /* Driver megnyitasa : */
  if ((fd=open("/dev/bcndm0",O_RDWR)) == -1 )
    { perror("/dev/bcndm0"); exit(1); }
  /* Modul rahelyezese : */
  if (ioctl(fd,I_PUSH,"birk") == -1)
    { perror("Push birk"); exit(2); }

  while ((count = read(0,buf,512) ) > 0) /* STANDARD inputrol olvas */
    if (write(fd,buf,count) != count)
    { perror("/dev/bcndm0"); exit(3); }

  /* Modul leszedese : */
  if (ioctl(fd,I_POP,0) == -1)
    { perror("Pop birk"); exit(2); }
  close(fd);
}
```

8.1.6 Az ide tartozó rendszerhívások

Ebben a pontban fel vannak sorolva azok a UNIX rendszerhívások, amelyek felhasználói szinten kapcsolódnak a STREAMS-hez. Ezek leírásának a tanulmányozása hasznos lehet.

- `open(2)` : STREAMS device driverre vonatkozó file megnyitása
- `close(2)` : STREAMS file lezárása
- `read(2)` : Adatok olvasása egy fileból
- `write(2)` : Adatok írása egy fileba
- `ioctl(2)` : `ioctl` hívások. A STREAMS rendszer speciális ioctl hívásairól lásd a `streamio(7)`-et is.
- `getmsg(2)` : Üzenet fogadása egy streamből
- `putmsg(2)` : Üzenet írása egy streamre
- `poll(2)` : STREAMS fileon input/output multiplexelés

8.2 A STREAMS driverek felépítése

Ebben a fejezetben ismertetve lesz az, hogy hogyan és mit kell tudnia egy STREAMS device drivernek. Majd bemutatunk egy egyszerű STREAMS device drivert, és két szintén egyszerű szerkezetű STREAMS modult. A driver egy tetszőleges nagyságú csak írható memóriát biztosít a programozónak. Vagyis a hozzá érkező üzeneteket mind eldobja, mint a `/dev/null` UNIX device driver. A modulok közül az egyik egy olyan modul lesz, amely nem változtatja meg a rajta átmenő adatokat - olyan, mintha ott se lenne. A másik modulnak kicsit több a haszna, a nyomkövetésnél jól lehet használni. A rajta átmenő üzenetek hexa dumpját írja ki.

8.2.1 Mire kell vigyázni egy driver készítésekor

A driverek logikailag a kernel legalsó szintjén vannak, ezért azok írásakor különösen vigyázzunk a következőkre :

- A driverek teljesen a kernelben dolgoznak, ezért a kis hibák is a UNIX rendszer leállását vonhatják maguk után.
- A driverek nem használhatnak rendszerhívásokat, és más ezekre épülő könyvtári rutinokat.
- A driverekben nem lehet a standard lebegőpontos aritmetikát használni.

8.2.2 STREAMS szolgáltatások

A STREAMS driverekben a driver megnyitását, az adatok továbbítását és a driver lezárását más-más eljárás végzi. Ezeket az eljárásokat a kernel a neki átadott táblázatok alapján tudja elérni, és amikor végrehajtja őket, akkor paraméterként átad az eljárásoknak információkat, ami megkönnyíti a munkájukat. A leggyakrabban használt rutinok és feladatuk leírása a következő pontok témája.
Általában külön eljárásokat írhatunk a lefelé és a felfelé haladó streamhez, de írhatunk a két stream által közösen használt rutint is. Ha valamelyik szolgáltatásra nincs szükség, akkor azt nem kell kódolni (ld. később).

Driver/modul megnyitása (open)

```
Szintaxis :

int xxopen(queue_ptr,dev,flag,sflag)
queue_t *queue_ptr;  /* A read-queuera a pointer */
dev_t dev;
int flag,sflag;
```

Ez a rutin STREAMS drivereknél a driver megnyitásánál, a STREAMS moduloknál a modulra vonatkozó `I_PUSH` hívásnál lesz végrehajtva. A hagyományos UNIX device driverekhez hasonlóan ennek az eljárásnak inicializálási feladatai vannak. A `queue_ptr` paraméter az új streamre egy pointer. A `dev` paraméter a device numbert tartalmazza, a `flag` paraméter megegyezik a hagyományos UNIX device drivereknél használt `flag`-gel jelölt paraméterrel (`open` flagek, ld. `oflags` az open rendszerhívás paraméterei között). A `dev` paraméter mind a major mind a minor device numbert tartalmazza. Ha szükségünk van ezek közül valamelyikre, akkor az a `major()` illetve `minor()` kernel segédrutinnal (legtöbb helyen ez egy makro) kinyerhető a `dev` paraméterből. Az `sflag` a stream megnyitásának speciális jellemzőit tartalmazza. Értéke lehet :

- 0 : normál open-nél lesz meghívva, vagyis amikor egy STREAMS driverre utaló spciális filet nyitunk meg. Ekkor a `dev`-ben megadott minor device number a filerendszerből kinyert szám lesz.
- MODOPEN : Modulkénti megnyitáskor lesz meghívva.
- CLONEOPEN : Clone-driver típusú megnyitást jelzi. Ekkor nem lesz átadva a `dev` paraméterben érvényes minor device number, hanem a drivernek magának kell egy belső táblázatából egy minor device numbert a hívó programnak adni.

Az, hogy egy driver hogyan lesz megnyitva, vagyis az `sflag=0` lesz vagy `sflag=CLONEOPEN` lesz, az a `/dev` directoryban levő speciális filera utaló bejegyzés tartalmától függ. A clone-drivernek a major device numberja 63 legyen (SINIX rendszerben a `clone` driver major device numberja pont 63), a minor device numberje pedig legyen egyenlő az igazi device major device numberjével. Ettől függetlenül csinálhatunk még több directory-bejegyzést is a filerendszerbe az igazi driver saját major és minor device numberjeivel. Ezekkel az egyes minor device-okat direkt el tudjuk érni (ha például csak a kontroller 2. portja jó nekünk, akkor azt). Erre láthatunk példát a következő

táblázatban.

Drivernév	Major device nr	Minor device nr	Megjegyzés
clone	63	0	Clone-driver
x0	40	0	Kontroller 0. portja (direkt)
x1	40	1	Kontroller 1. portja (direkt)
...	..	..	..
xcln	63	40	Kontroller a Clone-driveren keresztül

Ha az `xcln` filet megnyitjuk, aminek a major device numberje egyenlő a clone driver major device numberjével, akkor a clone driver lesz először végrehajtva, és (mint egy diszpécser) keres egy üres minor devicet, és ezt adja tovább a driver open rutinnak. A megnyitó rutin egy nulla visszatérési értékkel jelzi, ha nem volt hiba a futása közben, vagy `OPENFAIL` értékkel térjen vissza, ha a megnyitás eredménytelen volt. Az open és a close rutinok végrehajtásakor a kernel lokkolja a device i-nodeját, így egyszerre csak egy open vagy close rutin lehet aktív major/minor deviceonként. Megjegyzés: Ha a felhasználói programokból megnyitjuk egy STREAMS device driver valamelyik minor device-ját, majd később ismét kiadunk egy open rednszerhívást ugyanarra a minor devicera, akkor a második open rendszerhívásnál az open rutin nem lesz újra meghívva.

Driver/modul lezárása (close)

```
Szintaxis :

xxclose(queue_ptr)
queue_t *queue_ptr;  /* Pointer a read queuera */
```

A rutin a STREAMS driverre vonatkozó close rendszerhívásnál és az `I_POP` ioctl hívásnál lesz végrehajtva. Ezután a felhasználói program már nem tudja elérni a device drivert (vagy modult). Sikeres végrehajtás esetén a visszatérési érték: `0`, sikertelen végrehajtás esetén pedig az `errno` változóba rakandó érték legyen. Megjegyzés: a close rutin csak akkor lesz meghívva, amikor az adott minor devicera vonatkozó legutolsó filet is lezárták.

Üzenet fogadása (put rutin)

```
Szintaxis :

int xxput(qp,mp)
queue_t *qp;
mblk_t *mp;
```

Ez az eljárás akkor lesz végrehajtva, amikor a drivernek (vagy modulnak) adatot kell fogadnia a `qp` paraméterében megadott streamről. A `mp` paraméter a beérkezett üzenetre pointer. Az üzenetet vagy `el kell dobnia`, vagy `módosítva tovább kell adnia`. Ha továbbadja, akkor vagy egy saját sorába (`putq()` függvénnyel), vagy a streamen levő következő modulnak (`putnext()` makróval), vagy pedig az ellenkező irányú sorba (`qreply()` függvénnyel) adhatja tovább. Ha egy saját sorába adja tovább, akkor az üzenet még átmegy egy service rutinon is (ld. később). Az üzenettovábbítás általában az üzenetre mutató pointer továbbadását jelenti.

A service rutin

```
Szintaxis :

int xxsrv(qp)
queue_t *qp;
```

A service rutin csak a saját sorára mutató pointert kapja meg paraméterként. A `getq` függvénnyel tudja a sorából a következő üzenetet megszerezni, majd ezután ha tudja, akkor továbbadhatja a streamen következő modulnak. Ez alapján egy service rutin vázlatos felépítése a következő lehet :

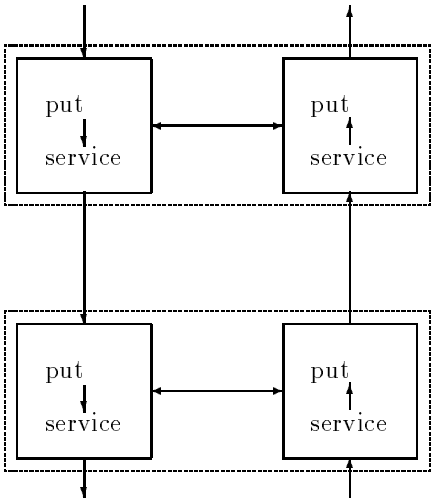
```
int xxsrv(qp)
  queue_t *qp;
{
  /* Lokalis deklaraciok - reentrans kod erdekeben */
  mblk_t *mp;

  while (mp=getq(qp) != NULL)
  {
    if (canput(qp->q_next) || (a message magas prioritasu))
    {
       dolgozd_fel_az_uzenetet();
       if (tovabb_kell_adni_az_uzenetet())
       {
         putnext(qp,mp);
       }
    } else {
       putbq(qp,mp);
       return;
    }
  }
}
```

Az üzenettípusok értékei úgy vannak kiosztva, hogy azok az üzenetek alacsony prioritásúak, amelyeknek adatblokkjában a db_type mező értéke QPCTL-nél kisebb (a QPCTL makro a saját operációs rendszerünk valamelyik headerfile-jában van definiálva).

Service rutin vs. put rutin

A service rutin opcionális, csak akkor szükséges, ha a put rutin az üzenetet nem tudja a beérkezésének " pillanatában " rögtön feldolgozni (mert például a hardware periféria még nem ért el egy bizonyos belső állapotot) vagy a stream már túl van terhelve és nem fér bele az üzenet. Minden modulhoz és driverhez meg lehet adni egy *low water mark* és egy *high water mark* értéket. Ez azért jó, mert el lehet vele érni egy nagyjából egyenletes adatáramlást (nem alakul ki *adatdugó*). Ha az üzenetek által lefoglalt byte-ok száma meghaladja a modulhoz megadott high water mark értéket, akkor egy `QFULL` flag be lesz állítva. Ennek a flagnek az állapotát az előtte levő modul a `canput` kernelfüggvénnyel lekérdezheti, és ha a flag be van állítva, akkor normál (alacsony) prioritású üzeneteket

már nem adhat tovább. (Ezeket a saját sorába visszarakhatja a `putbq` STREAMS kernel segédfüggvénnyel.) A service rutinra csak akkor jut a vezérlés, ha az addig üres sorba egy új üzenet került, vagy az alatta következő modulban elérte az üzenetek által lefoglalt byte-ok száma a low water markot. Ezen kívül máskor is megkaphatja a vezérlést, de erről saját magának kell gondoskodnia (a `qenable` és az `enableok` STREAMS kernel



segédrutinok segítségével).

Driver ioctl rutin

A hagyományos karakteres UNIX device driverekben egy külön belépési pont az `ioctl` hívásokat feldolgozó eljárás. Ezzel szemben a STREAMS device driverek esetében az `ioctl` hívásokat többnyire a stream-fej dolgozza fel, azokból bizonyos típusú üzeneteket generál, ami le lesz küldve a STREAMS drivernek (vagy modulnak). Az `I_STR` hívás például arra utasítja a stream-fejet, hogy egy `M_IOCTL` típusú üzenetet küldjön le a streamen. Ekkor az `ioctl` hívások feldolgozása az `M_IOCTL` típusú üzenetek driveren belüli feldolgozását jelenti.

8.2.3 Kritikus szakaszok védelme

Gyakran szükség van arra, hogy amíg egy bonyolultabb adatstrukturát egy STREAMS driveren vagy modulon belül megváltoztatunk, addig letiltsuk a megszakításokat. Például azért, mert ha a driver az adatstruktúra módosítását nem fejezte be, és a működését megszakítja egy másik driver, amely az inkonzisztens állapotban levő adatstrukturát tudja csak módosítani, akkor az egész rendszer épsége veszélyeztetve van. A kritikus szakasz elején az interruptokat az `splstr()` hívással lehet letiltani, a visszatérési értéket a kritikus szakasz végéig kell eltárolni. A kritikus szakasz végén vissza kell állítani a processzor interrupt prioritási szintjét az `splx()` kernel hívás segítségével a kritikus szakaszba belépés előtti szintre. Ez vázlatosan a következőképp oldható meg:

```
rutin()
{
  int regispl;

  ...
  regispl=splstr();
  ...
     /* Kritikus szakasz */
  ...
  splx(regispl);
  ...
}
```

8.2.4 Fontosabb adatszerkezetek

A következőkben ismertetett adatstrukturák általában minden STREAMS driverben megvannak. Azért fontosak, mert a kernel ezek alapján találja meg a driver különféle szolgáltatásait végző eljárásait.

A modul információs struktura

Ez a struktura az egyes stream sorok jellemzőit tartalmazza. A felépítése a következő (ISC UNIX 3.2 alapján) :

```
struct module_info {
  ushort mi_idnum;
  char *mi_idname;
  short mi_minpsz;
  short mi_maxpsz;
  ushort mi_hiwat;
  ushort mi_lowat;
};
```

Az `mi_idnum` mező tartalmazza a modul azonosítószámát. A `mi_idname` mező tartalmazhatja a modul nevére (egy stringre) mutató pointert (a kernel ez alapján a név alapján meg tudja találni). A `mi_minpsz` és a `mi_maxpsz` mezők a modul által elfogadott üzenetek minimális illetve maximális méretét szabják meg. Ezek a mezők mindig csak a stream-fejhez legközelebb levő STREAMS modulnál érdekesek, mert a STREAM-fej úgy fogja a lefelé küldött adatokat üzenetekre szétvágni, hogy az egyes üzenetek mérete a megadott minimális és maximális üzenetméret közé essen. A legfelső modul alatti modulokban megadott maximális és minimális üzenetméretet a STREAMS rendszer nem használja semmire, a programozó maga dönti el, hogy valamire akarja-e azt használni. A maradék két mező a low ill. high water mark értékeket tartalmazzák. Ezeknek az értékeknek a kiválasztásakor figyelembe kell venni azt, hogy a stream-fej a 64 bytenál rövidebb üzeneteknek is 64 byteot foglal le, nehogy a STREAMS rendszer rendelkezésére álló memória túlságosan feldarabolódjon.

A qinit struktura

Ebben a strukturában kell tárolni azt, hogy az egyes soroknak a szolgáltatásait melyik eljárásokban kódoltuk. A struktura felépítése a következő :

```
struct qinit {
  int   (*qi_putp)();         /* put eljarasra mutat */
  int   (*qi_srvp)();         /* service eljarasra mutat */
  int   (*qi_qopen)();        /* open ill I_PUSH eseten meghivva */
  int   (*qi_qclose)();       /* lezarasnal meghivva */
  int   (*qi_admin)();        /* 3bnet admin funkcioja */
  struct module_info *qi_minfo;  /* module_info struktura */
  struct module_stat *qi_mstat;  /* statisztikai struktura */
};
```

Ezt a strukturát a driveren belül **nem szabad megváltoztatni** (mivel ez másokra is kihatással lenne). Az `open` és `close` rutinokra mutató pointerek a read `qinit` strukturában legyenek, mivel a rendszer mindenképpen azokat hajtja végre. (Érdemes a write oldalra a könnyebb olvashatóság érdekében ugyanezeket az eljárásokat bejegyezni.)

A streamtab struktura

Ez a struktura tartalmazza a driverek read ill. write queue-jainak a qinit strukturáira mutató pointert. Felépítése :

```
struct streamtab {
  struct qinit *st_rdinit;     /* Felso read queue */
  struct qinit *st_wrinit;     /* Felso write queue */
  struct qinit *st_muxrinit;   /* Also read queue */
  struct qinit *st_muxwinit;   /* Also write queue */
};
```

Az alsó soroknak csak a multiplexer drivereknél van szerepük.

8.2.5 További hasznos tanácsok

A következő szabályokat érdemes betartani a rendszer konzisztensségének megtartása érdekében.

STREAMS device driverek készítésénél

- Azokat az üzeneteket, amikkel a driver nem tud mit kezdeni, el kell dobni (`freemsg()` kernel hívás segítségével).
- A drivernek fel *kell* dolgoznia minden M_IOCTL üzenetet. Ha ez nem történik meg, akkor a STREAM-fej (esetleg végtelen sokáig) leblokkol, mert hiába vár az `ioctl()` hívás nyugtázására.
- Ha egy driver nem tud mit kezdeni egy M_IOCTL üzenettel, akkor azt egy M_IOCNAK üzenettel nyugtázza. Ez az eset például akkor fordulhat elő, ha elírás vagy egyéb felhasználói programbeli hibák miatt a drivernek rossz (esetleg más drivereknek szánt) M_IOCTL üzenetek lesznek átadva.

STREAMS modulok készítésénél

- Azokat az üzeneteket, amik nem a modulnak szólnak, változtatás nélkül tovább kell adni.

- A modulnak szóló M_IOCTL üzeneteket feldolgozásuk után a modulnak nyugtáznia kell vagy egy M_IOCACK vagy pedig egy M_IOCNAK típusú üzenettel. (Azokat az M_IOCTL üzeneteket, amelyek nem a modulnak szólnak, változtatás nélkül tovább kell adni.)

Megjegyzések a példadriverhez

A driverben csak a write queue-hoz tartozó put eljárás van implementálva - a többi szolgáltatás helyén a `nulldev` eljárás címe van. Ez az eljárás a kernelben egy üres eljárás. C nyelven kb. a következő lehet :

```
nulldev()
{
}
```

A kernelt disassemblálva a következő assembly sorok tartoznak ehhez az eljáráshoz (INTERACTIVE 3.2 UNIX alapján) :

```
nulldev()
   pushl %ebp
   movl %esp,%ebp
   leave
   ret
```

A nulldev eljárás mellett létezik egy `nodev` nevű is, amit hasonló esetekben használhatunk. A kettő között a különbség az, hogy a nulldev- vel kódolt szolgáltatást kiváltó rendszerhívás a sikeres végrehajtásának megfelelő visszatérési értéket ad vissza, míg a nodev-vel jelölt szolgáltatásokat igénybe vevő rendszerhívások hibakóddal térnek vissza.

8.2.6 A driver hibaüzenetei

A device driverekben (így a STREAMS driverekben is) a hibaüzeneteket a `cmn_err()` kernel rutin segítségével lehet kiírni - ennek részletesebb leírását lásd az általánosan használható kernel rutinokról szóló részben. Helyrehozhatatlan hibák esetén ez a rutin viszi a UNIX rendszert a `panic` állapotba, ami végső soron a processzor shutdown állapotba rakását jelenti. A `cmn_err()` kernel hívás szintaxisa a következő:

```
#include "sys/cmn_err.h"

  int cmn_err(severity, format, arguments)
  char *format;
  int severity, arguments;
```

8.2.7 A driver listája

```
#include <sys/types.h>
#include <sys/stream.h>
#include <sys/cmn_err.h>
#define NULL   ((char *)(0))

extern nulldev();
static int bcndput(), bcndopen(), bcndclose();

static struct module_info modinfo =
/* A stream queue jellemzoit irja le */
  { 0,         /* Modulazonosito szam - itt nem fontos */
    (char *)0, /* Modul neve - itt nem fontos */
    0,         /* Minimalis message-meret */
    INFPSZ,    /* Maximalis message-meret - INFPSZ=vegtelen */
    0, 0       /* High; low water mark; 0=nincs ellenorzes */
  };

static struct qinit bcndwinit =
  /* "write" queue - lefele halado adatok */
  { bcndput,  /* A message feldolgozasat vegzo eljaras */
    nulldev,   /* Service procedure - itt nincs */
    bcndopen,  /* open-nel ill. PUSH-nal meghivott rutin */
    bcndclose, /* utolso close-nal vagy pop-nal vegrehajtva */
    nulldev,   /* admin bejegyzes (csak a 3bnet-hez kell) */
    &modinfo,  /* Modulinformacios struktura */
    NULL       /* Statisztikakat tartalmazo tablazat */
  };

static struct qinit bcndrinit =
  /* "read" queue - felfele halado adatok */
  { nulldev,   /* Errol a driverrol nem fognak olvasni */
    nulldev,   /* Service proc. nincs */
    bcndopen,  /* open-nel, PUSH-nal meghivott rutin */
    bcndclose, /* utolso close-nal vagy pop-nal meghivott r. */
    nulldev,   /* admin bejegyzes */
    &modinfo,  /* Modulinformacios struktura */
    NULL       /* Statisztikakat tartalmazo tablazat cime */
  };

struct streamtab bcndinfo =
  /* A kernel ez alapjan tud tajekozodni a driverben */
  { &bcndrinit, /* Read queue-rol informaciok */
    &bcndwinit, /* Write queue-rol informaciok */
    NULL,       /* Multiplex. read queue */
    NULL        /* Multiplex write queue */
```

```
  };

static int bcndput(q,mp)
  /* "write" rutin */
  queue_t *q;   /* Pointer a WRITE-queue-ra */
  mblk_t *mp;   /* Az adatblokkra pointer */
{
  cmn_err(CE_CONT, "bcndput... " ); /* Nem lesz ujsor kiirva ... */
  freemsg(mp); /* Visszaadjuk a rendszernek a stream-fej altal
                  az uzenetnek lefoglalt memoriateruletet */
}

static int bcndopen(q, dev, flag, sflag)
  queue_t *q;
  dev_t   dev;
  int     flag,sflag;
{
  cmn_err(CE_NOTE, "bcndopen" );
  return 0;
}

static int bcndclose(q)
  queue_t *q;
{
  cmn_err(CE_NOTE, "bcndclose" );
  return 0;
}
```

8.2.8 A driver kernelbe linkelése

A UNIX kernel bizonyos fokig moduláris szerkezetű. Van egy (nagy) központi része (ennek neve: `os.o`, mérete az Siemens i486 Release 4 UNIX esetén kb. 1.5 MByte), és vannak egyéb kiegészítések, ilyenek például a többféle (0.5K, 1K, 2K) filesystemek, az ATT `xt` driver. Nem minden UNIX konfiguráció alatt van szükség az összes ilyen kiegészítő driverre (az `xt` driverre pl. csak akkor, ha olyan speciális terminálokkal rendelkezünk, ami ezt kihasználhatja). Azt, hogy az aktuális UNIX software-konfiguráció ezek közül mit tartalmaz az ún. master fileokban van leírva. Két ilyen master file van: az egyik az `mdevice`, a másik az `sdevice`. Mindkettő file a `/etc/conf/cf.d` directoryban található.
Ezeknek a fileoknak a pontos szerkezetéről a `man` paranccsal nyerhetünk pontosabb információkat. Itt csak a legalapvetőbb tudnivalók lesznek ismertetve, ami a STREAMS drivereknél fontos. Az `mdevice` file tartalmazza az összes létező driver leírását, az `sdevice` file pedig az aktuális konfigurációt írja le.

Az mdevice file

Az `mdevice` fileból egy részlet (a fejléc nem tartozik a filehoz!):

```
Device Funkciok Karakte- Prefix Block     Char.     Min. Max.    DMA
 name  listaja  risztika       major nr. major nr. unit unit  channel

tape    ocrwi       ioc   tape_    0         41      0    1       -1
ldterm  -           Si    ldtr     0         0       1    32      -1
ptem    -           Si    ptem     0         0       1    32      -1
timod   -           Si    tim      0         0       1    4       -1
tirdwr  -           Si    trw      0         0       1    8       -1
udp     I          iSco   udp      0         29      1    256     -1
```

- `Devicename` : Ez a device (vagy a modul) belső neve. Maximum 8 karakter hosszú lehet.
- `Funkciolista` : Ez a mező egy vagy több karaktert tartalmazhat, amely a meglévő `driver` szolgáltatásokat írja le ha egyik szolgáltatás sem létezik, akkor egy mínusz jelet kell ide rakni). A driver szolgáltatások többek közt a következők (erről részletesebben lásd a megfelelő UNIX referencia kézikönyveket) :
  - `o` : open rutin
  - `c` : close rutin
  - `r` : read rutin
  - `w` : write rutin
  - `i` : ioctl rutin
  - `s` : startup rutin
  - `I` : init rutin
- `Karakterisztika` : Ez a device egyes jellemzőit tartalmazza. Itt csak a fontosabbak lesznek megemlítve :
  - `i` : A device driver installálható.
  - `c` : A device egy karakteres elérésű egység.
  - `b` : A device blokk elérésű.
  - `o` : Ehhez a devicehoz csak egy `sdevice` beli sor tartozhat.
  - `S` : Ez STREAMS jellegű driver ill. modul.
  - `r` : Erre a devicera szükség van még a legminimálisabb kernel konfigurációban is.

  A STREAMS device drivereknél itt az `Sc` betüknek elő kell fordulniuk. Azoknál a STREAMS moduloknál, amelyek egyben nem device driverek : `c` betü ne forduljon elő!
- `Prefix` : Ez a mező tartalmazza azt a szöveget, ami alapján a kernel megtalálja a belépési pontokat tartalmazó táblázatot. (STREAMS drivernél: ha a `streamtab` strukturának a neve (mint a példában) : `bcndinfo`, akkor ez a mező `bcnd` szöveget tartalmaz.

- `Block major nr.` : A legtöbb esetben ennek a mezőnek a tartalma legyen 0, mert a device numberek az installáláskor az `idinstall` végrahajtásakor lesznek kiosztva.
- `Char major nr.` : Ld. a `Block major nr.` mezőt.
- `Min. unit` : Ez tartalmazza azt a minimális számot, amit az `sdevice` tartalmazhat az `unit` mezőben.
- `Max. unit` : Ez tartalmazza azt a maximális számot, amit az `sdevice` maximum tartalmazhat az `unit` mezőben.
- `DMA channel` : Itt lesz megadva, hogy a device melyik DMA csatornát használja. Ha a device nem használ DMA csatornát, akkor ide `-1` kerül.

Az sdevice file

Az `sdevice` fileból egy részlet (a fejléc nem tartozik a filehoz!):

```
Device Con   Unit Ipl    Type   Vector   SIOA  EIOA   SCMA    ECMA
 name  figu-
       re

tape    N     1    0       0      0        0     0       0       0
ldterm  Y     16   0       0      0        0     0       0       0
ptem    Y     16   0       0      0        0     0       0       0
timod   Y     1    0       0      0        0     0       0       0
tirdwr  Y     1    0       0      0        0     0       0       0
udp     Y     256  0       0      0        0     0       0       0
```

Ez a file tartalmazza azt, hogy az `mdevice` fileban specifikált deviceok közül az aktuális konfigurációba mely driverek kerültek bele, és melyek nem. Ez a file az `/etc/conf/sdevice.d` directoryban levő komponensekből lesz összeállítva. (A fenti directoryban levő fileokat vagy a rendszerrel szállították, vagy később lett installálva az `idinstall` paranccsal.) A file bejegyzései a következők :

- `Device name` : A driver belső nevét tartalmazza. Valamelyik `mdevice` bejegyzés nevével meg kell egyeznie.
- `Configure` : Ez a mező `'Y'`-t tartalmazzon akkor, ha installálni kell a kernelbe, egyébként `'N'`-et.
- `Unit` : Ez az érték a device driver által vezérelhető aldevice-ok számát tartalmazza. (Maximális és minimális értéke az `mdevice` fileban van feltüntetve.)
- `Ipl` : Ez a mező azt határozza meg, hogy a driver interrupt rutinja mely rendszer ipl (ipl=interrupt priority level) szinten fusson. Értéke 0 és 8 közt lehet. Ha a drivernek nincs interrupt rutinja, akkor ebbe a mezőbe 0 kerüljön.
- `Type` : Ez a mező tartalmazza az interruptkiosztás módját. Értékei itt nem lesznek megnevezve. Ha a driver nem tartalmaz interrupt rutint, akkor ide 0 kerüljön.

- `Vector` : Ez a mező a devicehoz rendelt interrupt vektor (sor-)számát tartalmazza. Ha a devicehoz nem tartozik interrupt vektor, akkor e mező értéke 0 legyen.
- `SIOA` : A Start I/O Addresst tartalmazza a mező. Ha ilyen dolog nem tartozik a driverhez, akkor értéke 0 legyen.
- `EIOA` : Az End I/O Addresst tartalmazza a mező. Ha ilyen dolog nem tartozik a driverhez, akkor értéke 0 legyen.
- `SCMA` : A Start Controller Memory Addresst tartalmazza a mező. Ha ilyen dolog nem tartozik a driverhez, akkor értéke 0 legyen.
- `ECMA` : Az End Controller Memory Addresst tartalmazza a mező. Ha ilyen dolog nem tartozik a driverhez, akkor értéke 0 legyen.

8.2.9 Driver installálás ISC UNIX alatt

Ha egy device drivert elkészítünk, akkor azt egy speciális (Driver Software Packagenek nevezett) formában terjeszthetjük például mágneslemezen. Az INTERACTIVE UNIX az ilyen formátumú device drivereket automatikusan tudja installálni, ez nagy könnyebbség a felhasználónak. (Erről a formátumról az INTERACTIVE UNIX leírásában olvashatunk.) De a driver fejlesztésénél elég hosszadalmas munka lenne minden egyes tesztverziónál egy DSP formátumú lemezt létrehozni, ezért a következőkben bemutatjuk a driver installálásának manuális változatát. Ehhez általában a `root` néven kell bejelentkezni.

Objectkód létrehozása

Ha a driverünk forráskódja egyetlen C nyelvű programból áll (ennek neve legyen most `Driver.c`, akkor azt a szokásos módon a

```
cc -c Driver.c
```

UNIX parancs segítségével fordíthatjuk le. (Bármi lehet a driver neve, csak az a fontos, hogy a keletkezett objectkód át legyen majd nevezve a `Driver.o` névre.) Ha viszont a driver forráskódja több C nyelvű programból áll, akkor azokat egyenként a fenti módon kell lefordítani, és az `ld -r` paranccsal az így keletkezett objecteket lehet egy modullá összeszerkeszteni, aminek a neve legyen `Driver.o`.

Masterfileok változtatása

Adni kell a drivernek egy `nevet`. (Például a master fileokban ez a név fogja azonosítani a drivert. A példadriverünknél legyen ez a név:`bcnd`, ezt a nevet használjuk a továbbiakban is.)
Kreálni kell egy system file bejegyzést (ez kerül az `sdevice` fileba) a következőképp: az `/etc/conf/sdevice.d` directoryban hozzunk létre szövegszerkesztővel egy `bcnd` nevű filet. Ebbe írjuk a driverhez tartozó `sdevice` bejegyzést. Legyen ez például a következő :

```
bcnd      Y     1    0       0      0        0    0       0      0
```

Az `mdevice` filet szintén ki kell egészíteni. Ez úgy megy, hogy az aktuális directoryban létre kell hozni egy `Master` nevű filet, ami az új mdevice bejegyzést tartalmazza. Legyen ez a következő :

```
bcnd      -          Sic   bcnd      0         0       1    4     -1
```

A következő UNIX shell parancs megváltoztatja az `mdevice` filet (úgy, hogy a Master filet az aktuális directoryból törli). (A leírásban van az, hogy a Master nevű filet a rendszer törli az aktuális directoryból, amikor a példadrivereket beraktam a rendszerbe, akkor nekem nem törölte le ezt a filet.)

```
/etc/conf/bin/idinstall -a -m -k bcnd
```

Ezután az `/etc/conf/cf.d/mdevice` fileban nézzük meg a blokk/karakteres major device numbert, később még szükség lesz rá.

Speciális file bejegyzése

Kreálj egy `bcnd` nevű filet a `/etc/conf/node.d` directoryban, és egészítsd ki azt a Node formátumnak megfelelően. (Vagyis 4 mező legyen egy rekordban, és az egyes mezők jelentése a következő :
1.: Driver neve (itt : `bcnd`)
2.: A device speciális filejának a neve
3.: `'b'` vagy `'c'` betű (blokk/character device drivernek)
4.: Minor device number)
A driverünknél ez a következőképp néz ki :

```
bcnd        bcndm0      c          0
```

További fileok kreálása

Ha a drivert leteszteltük, és már hibátlanul működik, akkor a `/etc/conf/init.d`, a `/etc/conf/rc.d` és a `/etc/conf/sd.d` directorykat is a szükséges scriptekkel kiegészíthetjük.

A kernel újralinkelése

Kreálni kell egy `/etc/conf/pack.d/bcnd` nevű directoryt, és be kell vinni oda a `Driver.o` és a `Space.c` nevű fileokat, és csinálni kell egy másolatot a régebbi UNIX kernelről a következő paranccsal :

```
cp /unix /unix.bak
```

Majd végre kell hajtani a `/etc/conf/bin/idbuild` shell scriptet, ami újralinkeli a kernelt. Ha nem volt hiba, akkor shutdown után a UNIX rendszert újra betöltve tesztelhető a driver. (A device speciális fileok csak a következő UNIX reboot után lesznek megkreálva.)

8.2.10 Még egy példa: a birka modul

A következő lista egy "birka" modul listáját tartalmazza. A modul nem bánt semmit, ami rajta átmegy (innen ered a neve is). A modul az üzenet továbbításához a putnext() kernel makrot használja.

```
#include <sys/types.h>
#include <sys/stream.h>
#include <sys/cmn_err.h>
#define NULL   ((char *)(0))

extern nulldev();
static int birkwrite(), birkopen(), birkclose();

static struct module_info modinfo =
/* A stream queue jellemzoit irja le */
  { 0xa0,      /* Modulazonosito szam */
    "birk",    /* Modul neve */
    0,         /* Minimalis message-meret */
    INFPSZ,    /* Maximalis message-meret - INFPSZ=vegtelen */
    0, 0       /* High; low water mark; 0=nincs ellenorzes */
  };

static struct qinit birkwinit =
  /* "write" queue */
  { birkwrite, /* A message feldolgozasat vegzo eljaras */
    nulldev,   /* Service procedure - itt nincs */
    birkopen,  /* open-nel ill. PUSH-nal meghivott rutin - comment */
    birkclose, /* utolso close-nal vagy pop-nal vegrehajtva - comment*/
    nulldev,   /* admin bejegyzes (csak a 3bnet-hez kell) */
    &modinfo,  /* Modulinformacios struktura */
    NULL       /* Statisztikakat tartalmazo tablazat */
  };

static struct qinit birkrinit =
  /* "read" queue */
  { birkwrite, /* Message feldolgozasa */
    nulldev,   /* Service proc. nincs */
    birkopen,  /* open-nel, PUSH-nal meghivott rutin */
    birkclose, /* utolso close-nal vagy pop-nal meghivott r. */
    nulldev,   /* admin bejegyzes */
    &modinfo,  /* Modulinformacios struktura */
    NULL       /* Statisztikakat tartalmazo tablazat cime */
  };

struct streamtab birkinfo =
  /* A kernel ez alapjan tud tajekozodni a driverben */
```

```
  { &birkrinit, /* Read queue-rol informaciok */
    &birkwinit, /* Write queue-rol informaciok */
    NULL,       /* Multiplex. read queue */
    NULL        /* Multiplex write queue */
  };

static int birkwrite(q,mp)     /* "put" rutin */
  queue_t *q;   /* Pointer a WRITE-queue-ra */
  mblk_t *mp;   /* Az adatra pointer */
{
  putnext(q, mp); /* A megkapott uzenet tovabbitasa */
  /*
     Ha a fenti sor helyett csak egy
  freemsg(mp);
     sor lenne, akkor ez a korabbi driver modul megfeleloje lenne
  */
}

static int birkopen(q, dev, flag, sflag)
  queue_t *q;
  dev_t   dev;
  int     flag,sflag;
{
  /*
     Mivel minden queuen egyforman nem csinal semmit a modul, ezert
     nem kell semmi informaciot elrakni
  */
  return 0; /* Minden O.K. */
}

static int birkclose(q)
  queue_t *q;
{
  return 0; /* Minden O.K. */
}
```

8.2.11 Egy egyszerű debug modul

A következő lista egy `"debug"` modul listáját tartalmazza. A modul nem változtatja meg a rajta átmenő adatokat, csak egy hexa dumpot ír ki róluk a konzolra. Ha valahol arra vagyunk kíváncsiak, hogy mi megy keresztül egy streamen, akkor a kívánt helyre be kell illeszteni ezt a modult, és futás közben elemezni kell az eredményeket.

```
#include <sys/types.h>
#include <sys/stream.h>
#include <sys/cmn_err.h>
#define NULL   ((char *)(0))
```

```
#define P_FELFELE 1
#define P_LEFELE  2
#define L_EGYSOR  16

extern nulldev();
static int dbugwrite(), dbugopen(), dbugclose();
static dumpmsg();

static struct module_info modinfo =
/* A stream queue jellemzoit irja le */
  { 0xa1,      /* Modulazonosito szam */
    "dbug",    /* Modul neve */
    0,         /* Minimalis message-meret */
    INFPSZ,    /* Maximalis message-meret - INFPSZ=vegtelen */
    0, 0       /* High; low water mark; 0=nincs ellenorzes */
  };

static struct qinit dbugwinit =
  /* "write" queue - lefele haladnak rajta az adatok */
  { dbuglput,  /* Lefele meno adatok feldolgozasa */
    nulldev,   /* Service procedure */
    dbugopen,  /* open-nel ill. PUSH-nal meghivott rutin - comment */
    dbugclose, /* utolso close-nal vagy pop-nal vegrehajtva - comment */
    nulldev,   /* admin bejegyzes (csak a 3bnet-hez kell) */
    &modinfo,  /* Modulinformacios struktura */
    NULL       /* Statisztikakat tartalmazo tablazat */
  };

static struct qinit dbugrinit =
  /* "read" queue - felfele haladnak rajta az adatok */
  { dbugfput,  /* Felfele meno adatok feldolgozasa */
    nulldev,   /* Service proc. nincs */
    dbugopen,  /* open-nel, PUSH-nal meghivott rutin */
    dbugclose, /* utolso close-nal vagy pop-nal meghivott r. */
    nulldev,   /* admin bejegyzes */
    &modinfo,  /* Modulinformacios struktura */
    NULL       /* Statisztikakat tartalmazo tablazat cime */
  };

struct streamtab dbuginfo =
  /* A kernel ez alapjan tud tajekozodni a driverben */
  { &dbugrinit, /* Read queue-rol informaciok */
    &dbugwinit, /* Write queue-rol informaciok */
    NULL,       /* Multiplexer read queue  */
    NULL        /* Multiplexer write queue */
  };
```

```
static int dbugfput(q,mp)     /* "put" rutin - felfele */
  queue_t *q;   /* Pointer a WRITE-queue-ra */
  mblk_t *mp;   /* Az adatra pointer */
{
  dumpmsg(P_FELFELE,mp); /* Hexa dump */
  putnext(q, mp); /* A megkapott uzenet tovabbitasa */
}

static int dbuglput(q,mp)     /* "put" rutin - lefele */
  queue_t *q;   /* Pointer a WRITE-queue-ra */
  mblk_t *mp;   /* Az adatra pointer */
{
  dumpmsg(P_LEFELE,mp); /* Hexa dump */
  putnext(q, mp); /* A megkapott uzenet tovabbitasa */
}

static int dbugopen(q, dev, flag, sflag)
  queue_t *q;
  dev_t   dev;
  int     flag,sflag;
{
  return 0; /* Minden O.K. */
}

static int dbugclose(q)
  queue_t *q;
{
  return 0; /* Minden O.K. */
}

static dumpmsg(qdirection,mp)
  int qdirection; /* Melyik iranyba megy az uzenet? */
  mblk_t *mp;   /* Az adatra pointer */
{ mblk_t *aktmess; /* Az aktualis uzenet */
  register unsigned char *mitki; /* A kiirando karakterre pointer */
  int kiirtak; /* Egy sorba mennyi karaktert irt ki */

  aktmess=mp; /* Feltehetjuk, hogy nem NULLpointer */
  if (qdirection == P_LEFELE)  cmn_err(CE_NOTE,"LEFELE:  ");
  if (qdirection == P_FELFELE) cmn_err(CE_NOTE,"FELFELE: ");
  kiirtak=0; /* Eddig a sorba egy karaktert sem raktunk ki */
  while (aktmess != NULL) {
    mitki=aktmess->b_rptr;  /* Az adatterulet kezdete */
    while (mitki < aktmess->b_wptr) {  /* Adatterulet vege */
      cmn_err(CE_CONT," 0x%b", *mitki);
      kiirtak=kiirtak+1;
```

```
      mitki=mitki+1;
      if (kiirtak == L_EGYSOR) {  /* Betelt egy sor */
        kiirtak=0;
        cmn_err(CE_CONT," \n");
      }
    }
    aktmess=aktmess->b_cont; /* Folytatast is ... */
  }
}
```

8.2.12 Flush kezelése a driverben

Az M_FLUSH üzenetet minden olyan STREAMS modulnak és drivernek kezelnie kell, amely service rutint használ. Az ilyen típusú üzenetek indulhatnak a stream-fejtől, valamelyik modultól vagy a drivertől. Az üzenethez tartozó adatblokk első byteja a következő értékeket tartalmazhatja :

- FLUSHR : A read queuet kell üríteni.
- FLUSHW : A write queuet kell üríteni.
- FLUSHRW : Mind a read, mind a write queuet üríteni kell.

A driverekben az M_FLUSH üzenetek továbbítására a következő szabályok vonatkoznak: ha egy M_FLUSH üzenet ér a driverhez, és csak a FLUSHW flag van beállítva, akkor a driver eldobhatja az üzenetet. Ha pedig az üzenetben be van állítva az, hogy a read queuet üríteni kell, akkor a drivernek törölnie kell azt a részt, amely arra utal, hogy a read queuet üríteni kell, és így kell visszaküldeni az üzenetet a read queuera. A stream-fejnél minden pontosan az ellenkezőképpen történik: ha a read queuen fölfelé olyan M_FLUSH üzenet érkezik, melyben csak a FLUSHR flag van beállítva, akkor a stream-fej eldobja az üzenetet. Ha pedig az üzenet arra utal, hogy a write queuet üríteni kell, akkor az erre utaló flag törölve lesz, és a stream-fej az üzenetet visszaküldi a write queuen.

8.3 Egy STREAMS loopback driver

Az eddig bemutatott driverek nagyon egyszerűek voltak. Ezek segítségével meg lehetett érteni, hogy hogyan illeszkednek a kernelbe a driverek, de a debug modul kivételével kevés haszna van ezeknek a moduloknak. Ebben a fejezetben bemutatunk egy olyan STREAMS drivert, amelynek a feladata az, hogy a write queuen felülről érkező adatokat visszaküldje a read queuen. A következőkben az egyes fejezetek a driver egy-egy részét mutatják be, és az eddig még nem ismertetett, de lényeges dolgok meg lesznek magyarázva, és egyben példát láthatunk arra, hogy hogyan kell az M_FLUSH üzeneteket a drivereknek kezelni. A driver belső neve `lpbk` lesz.

8.3.1 Driver interface strukturák

```
#include <sys/types.h>
#include <sys/stream.h>
#include <sys/cmn_err.h>
```

```
#include <sys/errno.h>
#include <sys/sysmacros.h>

#define NULL   ((char *)(0))

extern nulldev();
static int lpbkwrite(), lpbkopen(), lpbkclose(), lpbkserv();

static struct module_info modinfo =
/* A stream queue jellemzoit irja le */
  { 0,         /* Modulazonosito szam */
    (char *)0, /* Modul neve */
    0,         /* Minimalis message-meret */
    INFPSZ,    /* Maximalis message-meret */
    1024, 512  /* High; low water mark; 0=nincs ellenorzes */
  };

static struct qinit lpbkwinit =
  /* "write" queue - lefele halado uzenetek queueja */
  { putq,      /* A message megy a service rutinnak */
    lpbkserv,  /* Service procedure */
    lpbkopen,  /* open-nel meghivott rutin */
    lpbkclose, /* utolso close-nal lesz vegrehajtva */
    nulldev,   /* admin bejegyzes (csak a 3bnet-hez kell) */
    &modinfo,  /* Modulinformacios struktura */
    NULL       /* Statisztikakat tartalmazo tablazat */
  };

static struct qinit lpbkrinit =
  /* "read" queue */
  { nulldev,   /* Message feldolgozasa */
    lpbkserv,  /* Service procedure */
    lpbkopen,  /* open-nel meghivott rutin */
    lpbkclose, /* utolso close-nal meghivott rutin */
    nulldev,   /* admin bejegyzes */
    &modinfo,  /* Modulinformacios struktura */
    NULL       /* Statisztikakat tartalmazo tablazat cime */
  };

struct streamtab lpbkinfo =
  /* A kernel ez alapjan tud tajekozodni a driverben */
  { &lpbkrinit, /* Read queue-rol informaciok */
    &lpbkwinit, /* Write queue-rol informaciok */
    NULL,       /* Multiplexer read queue */
    NULL        /* Multiplexer write queue */
  };
```

Itt érdemes megjegyezni azt, hogy ha a driver valamelyik queueja tartalmaz service

rutint, és a `put` rutin azon a queuen nem csinálna semmit az üzenettel, csak továbbadná, akkor az ahhoz a sorhoz tartozó `qinit` strukturában `put` rutinként jegyezzük be a `putq()` kernel segédrutint. Itt kihasználtuk azt, hogy a `putq()` egy függvény, és nem makro, ezért a memóriabeli címére hivatkozhatunk.

8.3.2 További deklarációk

Az `lpbk` struktura az egyes minor deviceok aktuális állapotát tartalmazza. Ennek értékei a következő lehet :

- LPBKOPEN : A minor device már meg van nyitva.

```
struct lpbk {
   unsigned lpbk_state;    /* minor device aktualis allapota  */
   queue_t  *lpbk_rdq;     /* a minor devicehoz tartozo queue */
};

/* A minor deviceok allapotat leiro konstansok */

#define LPBKOPEN 01

#define NLPBK    16     /* Maximalisan kezelheto minor deviceok szama */

struct lpbk lpbkmdev[NLPBK]; /* Itt vannak az minor deviceok adatai */

/* A minor deviceok szama van itt elrakva */

int lpbkcnt = NLPBK;

/* Loopback driver ioctl kodok */

#define I_NOARG       65
#define I_ERRNAK      66
#define I_ERROR       67
#define I_SETHANG     68
```

8.3.3 Loopback driver start rutinja

A driver adatstrukturáit a UNIX rendszer betöltésénél inicializálni kell. Erre való a device driverek `start` rutinja. A `start` rutin nem a STREAMS driverek specialitása - minden drivernek lehet ilyen rutinja. A driverünknél ez a következőképpen néz ki:

```
lpbkstart()  /* Nem lehet statikus! */
{
  int i;

  for (i=0;i<NLPBK;i++) {
      lpbkmdev[i].lpbk_state=0;
      lpbkmdev[i].lpbk_rdq  =NULL;
```

```
    }
}
```

Azt a konfigurációs fileokba is be kell jegyezni, hogy a driverhez egy `start` rutin is tartotik. A device driverekhez tartozhat még inicializáló úgynevezett `init` rutin is. A start és az init rutin közt a különbség az, hogy az `init` rutin a kernel memória menedzserének (KMA - kernel memory allocator) elindulása előtt lesz végrehajtva, így `csak` a külső hardware berendezés inicializálására használható. Ezzel szemben a `start` rutin már a KMA elindulása után lesz végrehajtva.

8.3.4 Loopback driver open rutin

```
lpbkopen(q, dev, flag, sflag)
  queue_t *q;
  dev_t dev;
  int flag,sflag;
{
   struct lpbk *mdevp; /* Aktualis minor device strukturara pointer */

   if ((sflag != CLONEOPEN) && (sflag != 0))
      return(OPENFAIL);   /* Csak CLONEOPEN megengedett */
   /* Keresni kell egy nem hasznalt minor device numbert */
   if (sflag == CLONEOPEN) {
     for (dev = 0; dev < lpbkcnt; dev++)
       if (!(lpbkmdev[dev].lpbk_state & LPBKOPEN)) /* Ha nem nyitott */
          break;   /* ... akkor ez kell nekunk */
     }
   if (sflag == 0) dev=minor(dev);

   if ((dev < 0) || (dev >= lpbkcnt)) /* Van meg szabad minor device? */
      return(OPENFAIL); /* NINCS ... */
   mdevp = &lpbkmdev[dev];

   if (!(mdevp->lpbk_state & LPBKOPEN)) {
      mdevp->lpbk_rdq = q;   /* Ez egy tetsz. felhasznaloi struktura */
      q->q_ptr = (caddr_t)mdevp;
      WR(q)->q_ptr = (caddr_t)mdevp;
      return(dev);
   }
   else
      if (q != mdevp->lpbk_rdq)
         return(OPENFAIL);        /* Valaki mar megnyitotta es */
                                  /* meg dolgozik ezen a streamen */
}
```

8.3.5 Loopback driver close rutin

```
lpbkclose(q)
```

```
  queue_t *q;
{
   /* Allitsuk be, hogy a queue mar nincs hasznalva */

   ((struct lpbk *)(q->q_ptr))->lpbk_state &= \verb! !LPBKOPEN;
   ((struct lpbk *)(q->q_ptr))->lpbk_rdq = NULL;
   flushq(WR(q), 1);
}
```

8.3.6 Loopback driver service rutin

A loopback driver service rutinja visszaküldi az M_DATA, M_PROTO, M_PCPROTO üzeneteket, az M_IOCTL üzenetekben megkapott "parancsokat" végrehajtja, kezeli az M_FLUSH üzeneteket, és az egyéb üzeneteket eldobja. Az M_IOCTL üzenetek feldolgozását egy külön eljárás végzi.

```
lpbksrv(q)
queue_t *q;
{
   mblk_t *bp;

   /* A q pointer mindenkeppen a write queuera mutasson */
   q = ((q)->q_flag&QREADR ? WR(q) : q);

   while ((bp = getq(q)) != NULL) {
      if ((bp->b_datap->db_type) < QPCTL && !canput(RD(q)->q_next) ) {
         putbq(q, bp); /* Nincs hely az alacsony priotitasu uzenetnek */
         return;
      }
      switch(bp->b_datap->db_type) {
         case M_IOCTL:  /* ioctl uzenetek */
            lpbkioctl(q, bp);
         break;
         case M_DATA:
         case M_PROTO:
         case M_PCPROTO:
            qreply(q, bp); /* Vissza a masik queuen */
         break;

         case M_FLUSH:
            if (*bp->b_rptr & FLUSHW) {
               flushq(q, FLUSHALL);
               *bp->b_rptr &= \verb! !FLUSHW;
            }
            if (*bp->b_rptr & FLUSHR)
               qreply(q,bp);
            else freemsg(bp);
         break;
```

```
         default:
            freemsg(bp);
         break;
      }
   }
}

lpbkioctl(q, bp)
  queue_t *q;
  mblk_t *bp;
{
   struct iocblk *iocbp;

   iocbp = (struct iocblk *)bp->b_rptr;
   switch(iocbp->ioc_cmd) {
      case I_NOARG:
            /*
             * Minden OK, csak kuldjunk vissza egy visszajelzo uzenetet
             */
            bp->b_datap->db_type = M_IOCACK;
            qreply(q, bp);
            return;
      case I_ERROR:
            /*
             * Hibakodot is adjunk vissza
             */
            iocbp->ioc_error = EPERM;
            bp->b_datap->db_type = M_IOCACK;
            qreply(q, bp);
            return;
      case I_ERRNAK:
            /*
             * Negativ visszajelzest kell generalni ...
             */
            iocbp->ioc_error = EPERM;
            bp->b_datap->db_type = M_IOCNAK;
            qreply(q, bp);
            return;
      case I_SETHANG:
            /*
             * Kuldjunk fel egy visszajelzest, majd egy olyan uzenetet,
             * amely M_HANGUP tipusu - be fog "fagyni" a stream.
             */
            bp->b_datap->db_type = M_IOCACK;
            qreply(q, bp);
            putctl(RD(q)->q_next, M_HANGUP);
```

```
            return;
      default:
            /*
             * Negativ visszajelzes - ismeretlen ioctl hivas miatt
             */
            bp->b_datap->db_type = M_IOCNAK;
            qreply(q, bp);
            return;
    }
}
```

8.3.7 Egy loopback drivert használó program

A következő program bemutatja a loopback driver használatát, és az egyes ioctl hívások eredményét is lehet vele tesztelni.

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <sys/stropts.h>

/* Loopback driverben is definialt konstansok */

#define I_NOARG       65
#define I_ERRNAK      66
#define I_ERROR       67
#define I_SETHANG     68

#define IOBS          1024

struct strioctl ioc; /* Ide kerulnek az ioctl hivas opcioi */

main()
{
   int fd, work;
   char buf[IOBS];

   if ((fd=open("/dev/lpbk", O_RDWR, 0777)) == -1) {
      perror("nem tudtam megnyitni a device drivert\n");
      exit(1);
   }
   printf("Milyen szoveget kuldjek el a loopback drivernek?");

   if ((fgets(buf,IOBS,stdin)) == NULL) {
      perror("fgets failed");
      exit(1);
   }
   /* devicera iras es onnan olvasas */
```

```
   lp_wrt(fd,buf);
   lp_read(fd);

   /* Device ioctl teszt */
   ioc.ic_cmd = I_ERRNAK;
   ioc.ic_timout = 0;
   ioc.ic_len = 0;
   ioc.ic_dp = NULL;
   if ((work = ioctl(fd,I_STR,&ioc)) < 0) {
      perror("Ioctl rendszerhivas sikertelen");
   }
   printf("Visszateresi ertek ioctl rendszerhivasbol: %d\n",work);
   lp_wrt(fd,buf);
   lp_read(fd);

   close(fd);
}

lp_wrt(fd, s)
  int fd;
  char *mit;
{
   if (write(fd,mit,IOBS) < 0) {
      perror("Hiba a wite rendszerhivasnal");
      exit(1);
   }
}

lp_read(fd)
int fd;
{
   char retbuf[IOBS];

   if (read(fd,retbuf,IOBS) < 0) {
      perror("Hiba a read rendszerhivasnal");
      exit(1);
   }
   printf("A beolvasott szoveg:%s\n",retbuf);
}
```

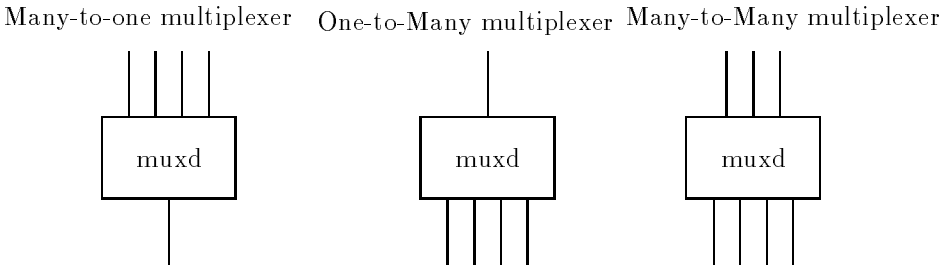
Megjegyzés: ne becsüljük le a loopback driver lehetőségeit! Ezzel lehetőség nyílik például arra, hogy egy `exec` rendszerhívás végrehajtása előtt a rá elküldött adatokat az `exec` rendszerhívással végrehajtott új program megkapja. (Ez azon múlik, hogy az `exec` nem zár le minden filedeszkriptort automatikusan, és a STREAMS rendszer teljes egészében a UNIX kernelen belül helyezkedik el.) A STREAMS-nek ezt a tulajdonságát használták ki akkor, amikor az ATT UNIX System V rendszeren a Berkeley sockets-eket implementálták.
(STREAMS alapú device driverek fejlesztésénél ez a driver és a korábban bemutatott `"debug"` modul nagyon nagy segítség!)

8.4 Multiplexer driverek

Az eddig leírtakon kívül gyakran szükség van arra, hogy az azonos helyről jövő adatokat ne mindig ugyanazon az úton továbbítsuk, hanem az adat jellegétől függően (például a benne tárolt protokoll-információk alapján) más-más irányba továbbítsuk. Ez gyakran szükséges például hálózati protokollok készítésénél. Erre a STREAMS természetesen lehetőséget ad, és ezt a lehetőséget nevezzük *multiplexelésnek*. A multiplexelést a STREAMS rendszerre épített *multiplexer device driverekkel* oldhatjuk meg. (Ezeket a felépítésük hasonlósága miatt nevezik device drivereknek, de külső perifériákkal leggyakrabban nem állnak közvetlen kapcsolatban.) Bár a STREAMS egy teljes eszköztárat biztosít multiplexer driverek készítésére, mégis gyakran felhasználói szinten implementálják a multiplexelést, mivel így a megoldás gyakran egyszerűbb és világosabb.

8.4.1 A multiplexerek elemei

A következő ábrán láthatjuk azokat az elemeket (az eredeti angol nyelvű nevükkel), amelyekből a multiplexelt streameket lehet összerakni.



A *many-to-one multiplexelés* a klónolással implementálható. Ez hasznos lehet például egy terminálonkénti többablakos megjelenítésnél, ahol minden egyes felső streamhez külön-külön ablak van rendelve. A *one-to-many multiplexelés* esetén egy felső streamről kerülnek az adatok az alsó streamek valamelyikére. A *many-to-many multiplexelés* esetén a felső streamek valamelyikén érkező adatok az alsó streamek közül valamelyiken kerülnek tovább (és fordítva).

8.4.2 Egy multiplexer összerakása

A következőkben látni lehet, hogy egy hálózatoknál is használható multiplexer konfigurációt hogyan rakhatunk össze. A hálózatoknál gyakori probléma az, hogy egy csomópontban levő gépben két (különböző) hálózati kártya van (ez a gép köt össze két alhálózatot). Például legyen az egyik egy X.25, a másik pedig egy Ethernet kártya. Ekkor az internet protokoll feladata az adatok megfelelő hálózati kártyára küldése, attól függően, hogy az egyes adatcsomagok rendeltetési helye melyik alhálózatban van. Itt a következőket csinálhatjuk : meg kell nyitni az Ethernet és az X.25 kártya driverét (ezek természetesen STREAMS device driverek; nevük legyen rendre `/dev/ethernet` ill. `/dev/x25`), és meg kell nyitni az internet protokoll multiplexer drivert is (ennek neve legyen a példában `/dev/ip`). Ezután az IP multiplexert össze kell kapcsolni (`I_LINK` ioctl hívással) a fenti két másik driverrel. Ennek a C nyelvű kódja a következő :

```
#include <stropts.h>     /* Konstansok miatt kell */
#inlcude <fcntl.h>

main()             /* STREAMS multiplexer config. */
{
  int fdether,fdx25,fdip;
  int muxx25,muxether;

  /* Driverek megnyitasa : */
  if ((fdether=open("/dev/ethernet",O_RDWR)) == -1 )
    { perror("/dev/ethernet"); exit(1); }
  if ((fdx25=open("/dev/x25",O_RDWR)) == -1 )
    { perror("/dev/x25"); exit(1); }
  if ((fdip=open("/dev/ip",O_RDWR)) == -1 )
    { perror("/dev/ip"); exit(1); }
  /* Multiplexer kiepitese : */
  if (muxx25=ioctl(fdip,I_LINK,fdx25) < 0)
    { perror("I_LINK ioctl sikertelen"); exit(2); }
  if (muxether=ioctl(fdip,I_LINK,fdether) < 0)
    { perror("I_LINK ioctl sikertelen"); exit(2); }
  /* Ezutan az also driverek filedesc.-jai lezarhatok */
  close(fdether);
  close(fdx25);

  pause();  /* Mindig fusson, nehogy a rendszer
               lezarja a fileokat */
  /* Ez nem kell most :
    if (ioctl(fdip,I_UNLINK,muxether) < 0)
      { perror("I_UNLINK ioctl sikertelen"); exit(2); }
    if (ioctl(fdip,I_UNLINK,muxx25) < 0)
      { perror("I_UNLINK ioctl sikertelen"); exit(2); }
  */
}
```

Az összelinkelés után az alsó streamek filedeszkriptorjaikra vonatkozó (ez esetben az fdx25 és az fdether) bármilyen fileművelet (a `close` kivételével) hibával fog befejeződni, vagyis a deszkriptorok használhatatlanok lesznek, ezért érdemes ezeket lezárni. Ennek az is a következménye, hogy ha STREAMS modulokat akarunk például az fdx25 filedeszkriptorhoz tartozó streamre rakni, akkor azt még az összelinkelés előtt kell elintézni. Ha ezután az IP drivert egy másik programból megnyitjuk és adatcsomagokat írunk a rá vonatkozó filedeszkriptorra, akkor az IP driver ezeket az adatokat szétosztja a két alsó driverre. Természetesen az IP drivert úgy kell megírni, hogy felismerje az adatcsomagokból azok rendeltetési helyét.

8.4.3 Multiplexer ioctl-ek

A következőkben a multiplexerek kiépítésekor használt STREAMS `ioctl` parancsokat, és azok paraméterezését ismertetjük (ld. ehhez korábbi `A STREAMS rendszer vezérlése`

című részt) :

- `I_LINK` : Összekapcsol két streamet, ahol `fd` paraméter értéke a multiplexer driver streamjének a filedeszkriptorját tartalmazza, az `arg` paraméter pedig a multiplexer driverhez kapcsolandó stream filedeszkriptorját tartalmazza. Sikeres végrehajtás esetén a hívás egy `multiplexer ID` értékkel tér vissza, amit a multiplexer konfiguráció leépítésekor kell megadni, sikertelen végrehajtás esetén `-1`-et ad vissza. Hiba esetén az `errno` változó lehetséges értékei :
  - EINVAL : Az *fd* stream nem multiplexelhető, vagy valami egyéb okból nem végrehajtható az összekötés (ld. erről részletesebben a `streamio(7)` leírást).
  - EAGAIN : A végrehajtáskor épp nem volt elegendő memória.
  - ENXIO : `Hangup`-ot kapott az `fd`-vel megadott stream.
  - ETIME : Timeout. Hiába várt a rendszer a multiplexer driver visszajelzésére. Visszajelzés nem érkezett a multiplexertől, pedig kellett volna.

  Megjegyzés: az összelinkelés úgy történik, hogy a multiplexer driver streamtab strukturájában megadott lower write illetve read queue információk beíródnak a driver alá linkelt stream stream-fej strukturájába. Ha alulról egy üzenet elérkezik a driver alá belinkelt stream-fejhez, akkor az üzenetet megkapja a lower read queue put rutinja, ami majd továbbadhatja azt valamelyik felső queuera.

- `I_UNLINK` : Szétkapcsol egy előzőleg `I_LINK` hívással összekapcsolt multiplexer konfigurációt. Itt az `fd` paraméter a multiplexerhez kapcsolt stream (multiplexer oldaláról nézve) filedeszkriptorját, az `arg` paraméter pedig az előző pontban említett multiplexer ID-et tartalmazza. (Ha a multiplexelést létrehozó program futása befejeződik, akkor minden általa létrehozott multiplexer kapcsolat automatikusan megszűnik.) Sikertelen végrehajtás esetén `-1`-et ad vissza. Hiba esetén az `errno` változó lehetséges értékei :
  - EINVAL : Rossz paramétereket adtunk meg `fd`-ben vagy `arg`-ban (ld. erről részletesebben a `streamio(7)` leírást).
  - ENOSR : A végrehajtáskor épp nem volt elegendő memória (a STREAMSnek fenntartott memóriaterületen).
  - ENXIO : `Hangup`-ot kapott az `fd`-vel megadott stream.
  - ETIME : Timeout. Hiába várt a rendszer a multiplexer driver visszajelzésére. Visszajelzés nem érkezett a multiplexertől, pedig kellett volna.

8.4.4 Input/Output események figyelése

A STREAMS rendszer és a UNIX kernel lehetőséget ad több stream egyidejű figyelésére is (ez lényegében más operációs rendszerek polling lehetőségeinek felel meg). Fontos ehhez egyrészt a `poll` rendszerhívás, másrészt az `I_SETSIG` STREAMS `ioctl` hívás.

Szinkron I/O polling

A `poll` rendszerhívás használatakor a programban meg kell adni a figyelendő streamek filedeszkriptorjait, és azt, hogy milyen eseményre kell várni, és mennyi idő múlva következzen be a timeout. A rendszerhívás kiadása után a program futása leáll addig,

amíg a timeoutként megadott idő letelik vagy valamelyik kijelölt esemény bekövetkezik. Szintaxisa :

```
#include <stropts.h>
#include <poll.h>

int poll(fds,nfds,timeout)
    struct pollfd fds[];
    unsigned long nfds;
    int timeout;
```

Az egyes paraméterek jelentése a következő :

- fds : Strukturatömb, amelynek minden egyes eleme megadja egy figyelendő stream filedeszkriptorját, a figyelendő eseményeket, és ebbe a tömbbe kerül a visszatérés pillanatában észlelt esemény. A `pollfd` struktura felépítése a következő :

  ```
  int fd;         /* Filedeszkriptor */
  short events;   /* Figyelendo esemenyek */
  short revents; /* Eszlelt esemenyek */
  ```

  Az egyes mezők jelentése a következő :

  - fd : Egy megnyitott (figyelendő) stream filedeszkriptorja.
  - events : A figyelendő eseményeket tartalmazó bitmező.
  - revents : A visszajelzett eseményeket tartalmazó bitmező.

  A figyelésre kijelölhető illetve a visszajelzett események a következők közül kerülhetnek ki :

  - POLLIN : Egy üzenet (nem `M_PCPROTO`) került a stream-fej read queuejának az elejére. (A visszaadott események között ez és a `POLLPRI` egyszerre nem fordulhat elő.)
  - POLLPRI : Egy magas prioritású üzenet (`M_PCPROTO`) került a stream-fej read queuejának az elejére.
  - POLLOUT : A lefelé menő streamra lehet írni (a rajta levő adatok mennyisége még nem érte el a high water markot).
  - POLLHUP : *Nem figyelésre kijelölhető esemény, csak a visszajelzett események közt fordulhat elő.* Azt jelzi, hogy hangupot kapott a megfelelő stream.(A visszaadott események között ez és a `POLLOUT` egyszerre nem fordulhat elő.)
  - POLLNVAL : *Nem figyelésre kijelölhető esemény, csak a visszajelzett események közt fordulhat elő.* Azt jelzi, hogy a hozzá megadott filedeszkriptor nem egy megnyitott streamre vonatkozik.
  - POLLERR : *Nem figyelésre kijelölhető esemény, csak a visszajelzett események közt fordulhat elő.* Azt jelzi, hogy egy hibaüzenet (ld. `M_ERROR` üzenettípus) érte el a stream-fejet.

- nfds : Az ellenőrzendő streamek száma. (Az `fds` tömb mérete.)

- timeout : A msec.-ban megadott timeout-idő. (Ha ez `0` , akkor a rendszerhívás nem várakozik, csak megnézi, hogy van-e valami a stream-fejeknél, ha pedig `-1`, akkor nem lesz timeout, a rendszerhívás tetszőlegesen sokáig fog várni.)

A visszatérési érték jelentése a következő :

- `0` : Timeout.
- `x>0` : A visszatérési érték : `x` egy pozitiv szám, és megadja, hogy hány streamen történt valami. (Vagyis hány streamnek a `revents` mezője tartalmaz nullától különböző értéket.)
- `-1` : Hiba történt. A hiba okáról az `errno` változóban találunk információt. Ezek lehetnek a következők :
  - EAGAIN : Valamelyik kernelen belüli művelet sikertelen volt, de a hívást érdemes mégegyszer megpróbálni.
  - EFAULT : A paraméterek a futó program címtartományán kívülre mutatnak.
  - EINTR : A rendszerhívás végrehajtása alatt egy signal érkezett.
  - EINVAL : Hibás az `nfds` paraméter értéke. (Vagy nullánál kisebb, vagy túllépi a rendszerben megengedett limitet.)

A következő program bemutatja a `poll` rendszerhívás használatát. A program megnyit két streamet, az egyiket az Ethernet hálózati driverhez,a másikat pedig az X.25 kontrollerhez, majd mindkét streamen adatokra vár. Ha az egyik streamen adatot kap, akkor az ott bejövő adatokat a másik streamen elküldi lefelé. A programban az `open` rendszerhívásnál az `O_NDELAY` flag azt jelzi, hogy ha a write rendszerhívás nem tudja a szükséges mennyiségű adatot a drivernek elküldeni, akkor a write ne blokkoljon. A példaprogramban ha a write rendszerhívás nem tud minden adatot visszaírni, akkor a felhasználó hibakezelő rutinja `usrherr()` lesz meghívva.

```
#include <fcntl.h>
#include <poll.h>

main()
{
  struct pollfd pollfds[2];
  char buf[1024];
  int count, i;

  if ((pollfds[0].fd = open("/dev/ethernet", O_RDWR | O_NDELAY)) < 0) {
     perror("open /dev/ethernet sikertelen");
     exit(1);
  }
  if ((pollfds[1].fd = open("/dev/x25", O_RDWR | O_NDELAY)) < 0) {
     perror("open /dev/x25 sikertelen");
     exit(2);
  }
  pollfds[0].events = POLLIN;
```

```
  pollfds[1].events = POLLIN;
  while (1) {
    if (poll(pollfds, 2, -1) < 0) {
       perror("poll rendszerhivas sikertelen");
       exit(3);
    }
    for (i=0;i<2;i++) {
        switch (pollfds[i].revents) {
        case 0 :         /* Semmi sem volt */
          break;
        case POLLIN :    /* Adat erkezett, kuldjuk vissza */
          while ((count = read(pollfds[i].fd, buf, 1024)) > 0)
          if (write(pollfds[1-i].fd,buf,count) != count)
               usrherr();
          break;
        default:
          perror("Erre az esemenyre nem szamitottam!");
          exit(4);
          break;
        } /* switch */
    } /* for */
  } /* while */
} /* main */

usrherr()
{
   ...
}
```

Aszinkron I/O polling - ioctl

A `poll` rendszerhívás mialatt egy eseményre vár, addig a rendszerhívást kiadó program futása gyakorlatilag felfüggesztődik. Néha ez nem megengedhető. Ekkor használhatjuk az `I_SETSIG` STREAMS ioctl rendszerhívást. Ez a hívás arra utasítja a kernelt, hogy ha adat érkezik a megadott filedeszkriptorú stream stream-fejéhez, akkor generáljon egy `SIGPOLL` signalt.
A STREAMS ioctl harmadik paramétere (korábban ezt `arg` jelölte) tartalmazza azokat az eseményeket, amelyek bekövetkezésekor signalt kell generálni. Ez a bitmező a következő értékek összerakásából állhat (az összerakás itt a bitenkénti VAGY műveletet jelenti) :

- `S_INPUT` : Egy alacsony prioritású üzenet (nem `M_PCPROTO`) érkezett az addig üres read queuera.
- `S_HIPRI` : Egy magas prioritású üzenet (`M_PCPROTO`) érkezett a stream read queuejára.
- `S_OUTPUT` : A stream-fejtől lefelé induló write queue már nincs tele - lehet rá adatokat írni.

- `S_MSG` : Olyan STREAMS üzenet érkezett a stream-fejhez, amelynek az a feladata, hogy `SIGPOLL` signált küldjön a programnak.

Ha az `arg` paraméter értéke: `0`, akkor a program ezután nem kap a kerneltől `SIGPOLL` signalokat. Hiba esetén az `errno` változó lehetséges értékei :

- EINVAL : Az `arg` paraméter értéke hibás. Ez a helyzet akkor is, ha az értéke nulla, de a nulla érték nem változtatna a jelenlegi állapoton.
- EAGAIN : Pillanatnyilag nincs elég STREAMS erőforrás (memória) a hívás végrehajtásához, de a hívást érdemes megismételni, hátha sikerülni fog.

8.5 A kernel segédrutinjai

A következőkben ismertetett kernelrutinok a device driverek fejlesztésénél jól felhasználhatók. Vannak egyéb kernel rutinok is (az itt felsoroltakon kívül), de azoknak egy részét az ATT a későbbi UNIX változatoknál nem biztos, hogy továbbra is támogatni fogja.

8.5.1 STREAMS-specifikus hívások

Az ebben a fejezetben ismertetett kernel rutinokat kizárólag STREAMS device driverekben használjuk. Részben azért, mert csak a STREAMS queuekon tudunk velük dolgozni, részben pedig azért, hogy STREAMS driverjeink és moduljaink hordozhatóak legyenek.

allocb - lefoglal egy üzenetblokkot

```
struct msgb *
    allocb(size)
    register int size;
```

Ez a függvény lefoglal egy üzenetblokkot a STREAMS adatterületen, amelynek a mérete legalább `size` byte. Az üzenet típusa `M_DATA` lesz (ezt a programban később meg lehet változtatni). Ha nincs elég memória, akkor az eredmény egy `NULL` pointer.

bufcall - hívj meg egy eljárást, ha lesz szabad memória

```
int bufcall(size, func, arg)
    int size;
    void (*func)();
    long arg;
```

Ezzel az eljárással lehet kérni a kerneltől például egy sikertelen `allocb()` hívás után, hogy ha felszabadul egy legalább `size` méretű memóriaterület (STREAMS üzenetek számára), akkor hívja meg a `func` paraméterben megadott függvényt `arg` paraméterrel (ez a `(*func)(arg);` függvényhívást jelenti majd). Ha a kernel a kérésünket tudomásul vette, akkor a függvény visszatérési értéke 1 lesz, ha ezt a kérésünket visszautasította, akkor 0 lesz a függvény visszatérési értéke.
`FONTOS:` Ha a megadott eljárást a kernel később majd meghívja, akkor sem lesz

garantálva az, hogy az `allocb()` hívással tudunk egy megadott méretű területet lefoglalni, ugyanis a kernelben másoknak is szükségük lehet memóriára, és lehet például az, hogy az épp felszabadult memóriát egy nagyobb prioritású driver lefoglalja előlünk.
Megjegyzés: nagyon gondoljuk meg, hogy mikor használjuk ezt a kernel rutint, mert ezt a hívási igényt a Release 3.2 UNIX rendszerekben még nem tudjuk visszamondani. A veszély abban van, hogy előállhat az a helyzet, hogy egy STREAMS deviceot már lezártunk, esetleg a queueja már más devicenak le lett foglalva, és csak ezután lesz valamikor meghívva a megadott függvény ...
Ezt a hiányosságot az ATT is észrevette, ezért a Release 4.0 UNIX rendszerben már van az `unbufcall()` kernel rutin, amellyel egy korábban bejegyzett és a kernel által elfogadott `bufcall()` hívás hatástalanítható.

canput - van-e elég hely a streamben

```
int canput(q)
    register queue_t *q;
```

Ez a függvény ellenőrzi, hogy van-e még hely a `q` message queueban. Ha a `q` queuenak nincs service rutinja, akkor a `canput()` függvény megkeresi az adott streamen soron következő legelső olyan modult, amelynek van service rutinja. (Ha nem talál ilyen modult, akkor a keresés a stream végén fejeződik be.) Végül ha még fér valami a `q` queueba, akkor a függvény visszatérési értéke: 1 - ekkor lehet újabb üzenetet rakni a queuera, ha a queue betelt, akkor pedig 0 a visszatérési érték, és ekkor a hívó blokkolva lesz. (A blokkolás azt jelenti, hogy nem kerül rá a futásra kész service folyamatok listájára.)

enableok - egy eddig inaktiv queue aktivizálása

```
void enableok(q)
     queue_t *q;
```

A feladata egy korábban kiadott `noenable()` függvény hatástalanítása, ami arra utasította a task schedulert, hogy a `q` queuet iktassa ki. (Ez nem jelenti azt, hogy a `qenable()` kernel függvényhez hasonlóan a sor service rutinja egyből végre is lesz hajtva!)

flushq - flush művelet egy queuen

```
void flushq(q,flag)
     register queue_t *q;
     int flag;
```

Ez a függvény töröl minden messaget a megadott `q` queueból, és felszabadítja az üzenetek által lefoglalt területet a `freemsg()` függvénnyel. A `flag` paraméter értéke, és ezek hatása (ld. `<sys/stream.h>` include filet) :

- FLUSHDATA : csak az `M_DATA`, `M_PROTO`, `M_PCPROTO` és az `M_DELAY` üzeneteket dobja el, a többit meghagyja.
- FLUSHALL : Minden üzenetet kidob a megadott queueból.

freeb - egy darab üzenetblokk felszámolása

```
void freeb(bp)
     register struct msgb *bp;
```

A `freeb` függvény felszabadítja a `bp` pointer által mutatott üzenetblokk által lefoglalt memóriaterületet, a hozzá tartozó adatblokkal együtt. (Ha a hozzá tartozó az adatblokk `db_ref` részének értéke 1-nél nagyobb, akkor csak ez a számláló fog csökkenni, a lefoglalt adatterület nem lesz felszabadítva.)

freemsg - egy teljes üzenet felszámolása

```
void freemsg(bp)
     register mblk_t *bp;
```

Végigmegy a megadott message minden egyes üzenetblokkján, és felszabadítja (ha lehet) a message által lefoglalt teljes memóriaterületet. Ha ez nem lehet (mert az adatblokk `db_ref` részének értéke 1-nél nagyobb), akkor ez a számláló eggyel csökkenni fog. (A `freemsg()` függvény végigmegy a message minden egyes blokkján, és minden egyes blokkot felszabadít a `freeb()` segítségével.)

getmid - modul id. lekérdezése

```
ushort getmid(name)
       char *name;
```

A függvény eredménye a megadott nevű modul modul azonosító száma. (Ha nincs ilyen modul, akkor az eredmény: `0`).

getq - következő message leszedése a queueról

```
mblk_t *getq(q)
       register queue_t *q;
```

Leszedi a következő üzenetet (ha van) a `q` message queueról. A függvény egy pointert ad vissza, ami a leszedett message címét tartalmazza. Ha nincs már message a queuen, akkor a függvény visszatérési értéke `NULL` pointer lesz, és a rendszer beállít a queue adatstrukturájában egy `QWANTR` flaget, ami azt jelzi, hogy a queueról olvasni akarnak. Ha ilyen állapotban egy üzenet érkezik a queuera, akkor a kernel a `qenable()` függvény meghívásával automatikusan újraindítja a service rutint.
Ha a `getq()` rutin leszedett egy üzenetet a `q` message queueról, és ezután a queuen levő üzenetek összhossza kisebb, mint a queuehoz megadott `low water mark` érték, és a queuera már próbáltak korábban új üzenetet rakni, de ez sikertelen volt, akkor a `q` queue mögötti sor service rutinja a `qenable()` függvény meghívásával újra el lesz indítva.

noenable - queue inaktivizálása

```
void noenable(q)
     queue_t *q;
```

A megadott `q` queuet inaktiv állapotba hozza - a STREAMS service scheduler nem adja ennek a sornak a service rutinjára többet a vezérlést. (Inverz művelete: az `enableok()` függvény.)

OTHERQ - megadja a queue párját

```
#define OTHERQ(q)    ((q)->q_flag&QREADR? (q)+1 : (q)-1)
```

A makro eredménye a `q` queue párjára mutató pointer. (Ha a `q` a read queue, akkor az eredmény a hozzá tartozó write queuera mutató pointer; ha `q` egy write queuera pointer, akkor az eredmény a read queuera pointer.)

putctl - kontroll-üzenet továbbítása

```
int putctl(q, type)
    queue_t *q;
    int type;
```

A `putctl()` függvény lefoglal egy üzenet számára az `allocb()` kernelhívás segítségével memóriát, az üzenet típusát beállítja a `type` paraméterben megadottra, majd meghívja a `q` paraméter által mutatott queue `put` rutinját. A függvény visszatérési értéke 1, ha minden rendben ment. Nulla akkor lehet a visszatérési érték, ha a rendszer nem tudott lefoglalni az üzenetblokk számára memóriát, vagy a `type` paraméter értéke az `M_DATA`, `M_PROTO`, `M_PCPROTO`, vagy `M_DELAY` értékek egyike.

putbq - üzenet visszarakása a sorba

```
int *putbq(q, bp)
    register queue_t *q;
    register mblk_t *bp;
```

A `putbq()` függvény a `bp` által mutatott üzenetet visszarakja a `q` által mutatott queue elejére. (A queuban legelőbbre kerülnek a magas prioritású üzenetek, a normál prioritású üzenetek pedig a magas prioritású üzenetek mögé, de a korábban már ott levő alacsony prioritású üzenetek elé kerülnek.) *A service rutin soha ne rakjon vissza magas prioritású üzenetet a saját queuejába, mert ez végtelen ciklust eredményezne a kernelen belül!* Sikeres végrehajtás esetén a függvény visszatérési értéke: 1, sikertelen végrehajtás esetén a visszatérési érték 0 lesz.

putnext - egy üzenetet a következő queuera rak

```
#define putnext(q,mp) ((*(q)->q_next->q_qinfo->qi_putp)((q)->q_next,(mp)))
```

A `putnext()` egy makro, amely a `q` után közvetlenül következő queue `put()` rutinját hívja meg, és átadja neki az `mp` által mutatott üzenetet.

putq - egy megadott üzenetet valamelyik queuera rakja

```
int *putq(q, bp)
    register queue_t *q;
    register mblk_t *bp;
```

A `putq()` függvény a `q` paraméterben meghatározott queura átadja az `mp` által mutatott üzenetet.

qenable - queue aktivizálása

```
void qenable(q)
     register queue_t *q;
```

Ez a függvény a `q` paraméter által meghatározott sort a STREAMS schedulernek arra a listájára rakja, amely a futásra kész sorok adatait tartalmazza. Ez azt jelenti, hogy az adott queue service rutinja rövid időn belül újból futni fog.

qreply - üzenet visszaküldése ellentétes irányban

```
void *qreply(q, bp)
     register queue_t *q;
     mblk_t *bp;
```

Ez a függvény meghatározza a `q` queue párját (a lefele menő streamnek a párja a felfelé menő, ill. fordítva), és azon a `putnext()` makro működéséhez hasonlóan visszaküldi a `bp` pointer által meghatározott üzenetet. Ezt a függvényt szokták használni az `M_IOCTL` üzenetekre küldendő válasz visszaküldésére.

RD - megadja a read queuet

```
#define RD(q)    ((q)-1)
```

A makro paramétere (`q`) egy write queuera mutató pointer, eredménye pedig a `q` queue párjára (vagyis a read queuera) mutató pointer. Ez a művelet azért ilyen egyszerű, mert a kernel a queuekat nem egyenként foglalja le, hanem párosával. Az alacsonyabb memóriacímen van a read queue, a magasabbon pedig a write queue.

splstr - átállítja az interrupt prioritási szintet

```
#define splstr()   spltty()
```

Az `splstr()` makro az `spltty()` kernelhívás segítségével beállítja a processzor interrupt prioritási szintjét a STREAMS driverek és modulok interrupt prioritási szintjére, a kritikus szakaszok védelme érdekében. Ez azt jelenti, hogy az éppen futó driver működését más driverek vagy modulok nem tudják megszakítani. Ez a rutin visszatérési értékként az addigi processzor interrupt prioritási szintet adja. A kritikus szakasz végén az `splx()` kernelhívás segítségével vissza `kell` állítani a processzor interrupt prioritási szintjét az előző értékre. Ezt a rutint csak `igen indokolt esetben` használjuk! Megjegyzés: az Intel 80386-os UNIX rendszerek esetén ez általában IPL=7 szintnek felel meg, és ekkor sem a soros

vonalakról jövő megszakítások, sem az óramegszakítások nem lesznek megengedve. Mivel a rendszeróra is le lesz tiltva, ezért csak olyan rutinokat védjünk így, aminek a végrehajtása nem igényel több időt, mint két órainterrupt közti idő.

unlinkb - egy üzenet első blokkját törli

```
mblk_t *
     unlinkb(bp)
     register mblk_t *bp;
```

Az `unlinkb()` függvény elválasztja a `bp` paraméter által mutatott üzenet első üzenetblokkját a mögötte lévő blokkoktól, és egy pointert ad vissza az üzenet megmaradó részére. Az eredménye NULL pointer lesz, ha nem maradt több üzenetblokk az üzenetben. (Az első üzenetblokk nem lesz automatikusan felszabadítva, így ez a programozó feladata marad.)

WR - megadja a write queuet

```
#define WR(q)     ((q)+1)
```

A `WR` makro paramétere (`q`) egy read queuera mutató pointer, eredménye pedig a `q` queue párjára (vagyis a hozzá tartozó write queuera) mutató pointer.

8.5.2 Általánosan használható kernel rutinok

A következőkben ismertetett kernel hívások mind a hagyományos, mind pedig a STREAMS device driverek készítésénél jól használhatóak. Ezek használata gyakran szükséges, de ronthatják a program hordozhatóságát. (Például a Release 4.0 UNIX módosított major/minor device number kezelése miatt ha áttérünk erre a UNIX rendszerre, akkor módosítani kell a drivereknek azt a részét, amely a `minor()` vagy `major()` rutinokat használják. Ez természetesen nem nagy munka - baj csak akkor van, ha ezt elfelejtjük.)

cmn_err - driver hibaüzenetek kiírása konzolra

```
#include "sys/cmn_err.h"

int cmn_err(severity, format, arguments)
    char *format;
    int severity, arguments;
```

Figyelmeztetés: egy hibásan megadott severity érték a rendszert azonnal panic állapotba viszi.
Az egyes paraméterek jelentése a következő :

- severity : Négy különböző értéke lehet a hiba súlyosságától függően :
  - `CE_CONT` : Ez kb. egy printf() hívással egyenértékű - nem ír az üzenet elé semmit.

  - `CE_NOTE` : Kiírja a paraméterekben megadott szöveget, egy `NOTICE:` üzenetet követően.
  - `CE_WARN` : Kiírja a paraméterekben megadott szöveget, egy `WARNING:` üzenetet követően.
  - `CE_PANIC` : Kiírja a paraméterekben megadott szöveget, egy `PANIC:` üzenetet követően, és a rendszert panic állapotba viszi. (Ekkor a UNIX azonnal leáll, és egy memória dump kerül a swap egységre.)

- format : Egy printf()-hez hasonló formátumot lehet itt megadni. A stringben megadható adatformátumok :
  - `%b` : Két hexadecimális számjegy (egy byte)
  - `%c` : Karakteres
  - `%d` : Előjeles decimális szám
  - `%o` : Előjel nélküli oktális szám
  - `%s` : String (karakter-pointer)
  - `%x` : Hexadecimális (vezető nullákkal együtt írja ki)

  Mezőhossz megadása nem megengedett (például a %9d nem adható meg formátumként). Ilyen módon nem csak a konzolra írhatunk ki hibaüzenetet. Azt, hogy a kiírt üzenet hova kerüljön,a `format` paraméterben megadott üzenet első karaktere határozza meg. Ha ott az első karakter egy felkiáltójel (vagyis: !), akkor az üzenet nem kerül ki a konzolra. Egy belső bufferben lesz tárolva, amit a `crash` rendszerprogram segítségével elemezhetünk. Ha az első karakter egy kalap (vagyis: ^), akkor az üzenet ki lesz írva a konzolra, de nem kerül bele a fent említett bufferbe. Ha ezektől eltérő karakterrel kezdődik az üzenet, akkor a rendszer mind a konzolra, mind a belső bufferébe kiírja azt. Ha a `severity` érték nem `CE_CONT`, akkor az üzenet kiírása után a rendszer még egy újsor karaktert is kiír, míg `CE_CONT` severity érték megadása esetén ilyen újsor karakter nem lesz automatikusan kiírva.
  Megjegyzés: A kiírandó adatok formátumának megadása a különböző UNIX rendszerekben eltérő lehet, ezért ha az egyik UNIX rendszerre készült STREAMS driverünket átvisszük egy másfajta UNIX rendszerre, akkor nézzünk ennek utána a UNIX leírásban, nehogy valami ilyen jellegű hibát kövessünk el.

- arguments : Opcionális argumentek, amit ezzel a rutinnal ki akarunk iratni. A format paraméterrel ezek az argumentumok összhangban legyenek.

major - megadja a major device numbert

```
#include "sys/sysmacros.h"

int major(dev)
    dev_t dev;
```

Ez a kernel hívás arra szolgál, hogy az `open()` rutinnak átadott major és minor device numbert tartalmazó paraméterből kinyerje a major device numbert. Megvalósítása egyes UNIX rendszrekben makroval történik. A `dev` paraméterben leggyakrabban az `open()` rutinnak átadott azonos nevű paraméter lesz megadva. Megjegyzés:Az ISC 3.2 UNIX rendszer alatt a `dev_t` típus short integer típust jelöl.

minor - megadja a minor device numbert

```
#include "sys/sysmacros.h"

int minor(dev)
    dev_t dev;
```

A `minor()` kernel hívás arra szolgál, hogy az `open()` rutinnak átadott major és minor device numbert tartalmazó paraméterből kinyerje a minor device numbert. Megvalósítása egyes UNIX rendszrekben makroval történik.

8.6 Kérdések

1. Megváltozhat-e a bemutatott STREAMS loopback drivernél az üzenetek visszaadási sorrendje a bemenő sorrendhez képest? Ha megváltozhat, akkor jelenthet-e ez problémát mondjuk az Internet Protokoll (IP) implementálásakor?

2. Módosítsuk a korábban bemutatott debug modult úgy, hogy ne csak az üzenetek tartalmát írja ki, hanem minden egyes üzenet elején írja ki annak típusát is!

3. A STREAMS loopback driverünk felfelé haladó (read) queuejának van egy service rutinja, és ahhoz a queuehoz nincs definiálva `put` rutin. Mikor lesz meghívva a service rutin a read queue oldaláról? (Tanács: Gondoljon arra, hogy mit csinál a `getq()` kernel rutin!)

4. Mit gondol, hogy egy `I_LINK` után miért lesz a felhasználói program számára az a stream használhatatlan, amit a multiplexer alá linkeltünk? (Tanács: Gondolja meg, hogyan működik az I_LINK ioctl hívás!)

5. Bővítsük ki a STREAMS loopback driverünket, hogy modulként is lehessen használni!

8.7 Ajánlott irodalom

- 1. M. Ben-Ari: Principles of Concurrent Programming
  Englewood Cliffs, Prentice-Hall International, 1982

  A könyv kiváló stílusban alapos bevezetést nyújt a párhuzamos programozásba. Ismerteti a gyakrabban használt szinkronizációs lehetőségeket (szemaforok, üzenetek átadása, randevúk, ...), és ismertebb párhuzamos algoritmusokat is bemutat.

- 2. Stephen R. Bourne: The UNIX System Reading, Addison-Wesley, 1982

  A könyv egy általános bevezetést nyújt a UNIX rendszer programozásához, ismerteti a UNIX környezetet, amit a felhasználó a gép elé leülve lát.

- 3. Brian W. Kernighan, Rob Pike: The UNIX Programming Environment Englewood Cliffs, Prentice-Hall, 1984 (Magyarul is megjelent)

  A könyvben a UNIX rendszerek "közös része" (Version 7 UNIX) van részletesen (felhasználói és programozói szempontok szerint) ismertetve. A könyvet azért is érdemes elolvasni, mert néhány angol kifejezést nagyon lehetetlen módon fordítottak benne magyarra, és ez többször is megmosolyogtatja az olvasót (azt persze én (Cs.B.) is elismerem, hogy néhány kifejezésnek nagyon nehéz magyar fordítását találni, lehet, hogy néhol nekem sem sikerült megtalálnom az "igazit").

- 4. Brian W. Kernighan, Dennis Ritchie: The C Programming Language Englewood Cliffs, Prentice-Hall, 1978

  Ez a könyv ismerteti a legjobban a C nyelvet (nemcsak a UNIX környezetben használható). A könyv megjelent magyarul, és hasznos mindenki számára.

- 5. A. S. Tanenbaum: Operating Systems Design and Implementation Englewod Cliffs, Prentice-Hall, 1987

  A könyv bemutatja az operációs rendszerek felépítését, a következő pontokra bontva: rendszerhívások (Version 7 UNIX alapján), processzek, input/output, memóriakezelés és a fájlrendszerek. Kiegészítésként benne van egy UNIX-szerű operációs rendszernek a teljes forráskódja kommentekkel és a tervezésénél meghozott döntésekkel és azok indoklásával együtt. (MINIX a rendszer neve).

- 6. A. S. Tanenbaum: Számítógép-hálózatok Novotrade Kiadó Kft. - Prentice-Hall közös kiadvány, 1992

  A könyv az OSI hálózati referenciamodellen keresztül ad elég absztrakt modellt a számítógépes hálózatokról (mind a 7 OSI szintet nagyon részletesen ismertetve). A magyar fordts itt egészen jól olvasható. A könyv nem tárgyalja nagyon részletesen az operációs rendszer és a hálózati szintek közötti interfészt (mint pl. a socket rendszer), ezért csak ez alapján nagyon nehéz lenne "az első" hálózati alkalamzást elkészíteni.

- 7. AT&T Bell Labs.: UNIX System V Release 4.0 Programmer's Guide AT&T Bell Labs.: UNIX System V Release 4.0 Programmer's Reference Manual (Prentice-Hall kiadványok)

A UNIX programozóknak nyújtott lehetőségeit ismertetik. Az előbbi "szakácskönyvi" szinten, míg az utóbbi a rendszerhívások részletes specifikációját tartalmazza. (Ez utóbbi nem igazán olvasmányos.)

- 8. Maurice Bach: The Design of the UNIX Operating System Englewood Cliffs, Prentice-Hall, 1987

  Kifejtett témák (a System V UNIX alapján): kernel, buffer cache, fájlrendszerek, folyamatok, memóriakezelés és az I/O, és az IPC.

- 9. Myril Clement Shaw, Susan Soltis Shaw: UNIX Internals (A systems operation handbook) TAB BOOKS, 1987 ISBN 0-8306-2951-3

  Ez a könyv is a UNIX operációs rendszer belső felépítését mutatja be különféle szempontokból: ismerteti a fájlrendszert, i-node-okat, a UNIX folyamatainak a fogalmát, az I/O-t és az eszközmeghajtókat. Kiegészítésként tartalmaz néhány fejezetet, melyben összefoglalja a UNIX rendszerhívásait, a fontosabb UNIX parancsokat valamint tartalmaz egy rövid összehasonlítást, melyben az AT&T és a Berkeley UNIX-ot hasonlítja össze.

- 10. S. Leffler, M. McKuisk, M. Karels, J. Quaterman: The Design and Implementation of the 4.3BSD UNIX Operating System Addison-Wesley, 1989

  A könyv a 4.3BSD UNIX alapján bemutatja a UNIX rendszer felépítését, szerkezetét. A könyv a témát nagyon részletesen kifejti. A szerzői maguk is részt vettek a 4.3BSD rendszer fejlesztésében.