Általános
INFORMATIKAI
Tanszék

Operációs rendszerek

Jegyzet

Dr. Vadász Dénes

Miskolc, 2002. szeptember

Tartalom

0. Bevezetés

Az *Operációs rendszerek* c. tantárgy a műszaki informatikai szak informatikai tantárgycsoportjának része. A tárgyat a 3. félévben ajánlott felvenni. Heti 3 óra előadás, 1 óra gyakorlat időtartamban kerül meghirdetésre. A tantárgy szerepel a programozó matematikus szak tantervében, a műszaki menedzser szak informatikai szakirány tantervében is. Felvételéhez szükséges a *Számítógépek, számítógéprendszerek* és a *Szoftverfejlesztés I*. c. tantárgyak tananyagának ismerete.

A tárgy célja: az operációs rendszerek fejlesztése, működése és működtetése során felmerülő alapfogalmak, koncepciók, filozófiák, megfontolások, megoldások megismerése, továbbá az operációs rendszerek belső szerkezetének, működésének megértése. Nem cél, hogy operációs rendszert tudjunk írni, de értsük meg a belsejét, működését. Egy egyszerű felhasználótól ezt nem szokás elvárni, nála megelégszünk azzal, ha tudja használni a számítógéprendszert, az operációs rendszert. Informatikusoktól azonban többet várunk el, nekik a működés alapelveivel tisztába kell lenniük.

Ma a (szoftver)piacon nagyon sok és sokféle operációs rendszert vásárolhatunk. Működésük alapelveiben vannak közös és általános koncepciók, és persze vannak jelentős különbségek, specifikumok is. A tárgyban megpróbáljuk összefoglalni az általános törvényszerűségeket. Mégis a legnagyobb hangsúly a Unix szerű operációs rendszereken lesz, egy-két VAX/VMS Windows NT/2000 koncepciót fogunk érinteni, az MS-DOS-t éppen csak megemlítjük. Az elveket a gyakorlattal viszont konkrét operációs rendszereken vethetik össze: használniuk kell a Miskolci Egyetem különböző számítógéprendszereit. A gyakorlatok feladatai elsősorban valamilyen UNIX rendszerhez kapcsolódnak.

Milyen nagyobb témaköri egységei lesznek a tárgynak?

Az operációs rendszerek megismerése, tárgyalása során használunk bizonyos absztrakciókat. Ilyenek:

- a processz fogalom, ami körülbelül egy dedikált processzoron futó program.
- A virtuális memória fogalom, melyet úgy képzelhetünk el, hogy a processzek számára meglehetősen nagy memória áll rendelkezésre, ami részben az elsődleges, részben a másodlagos tárra van leképezve.
- A fájl-rendszer fogalom, azaz, hogy szimbolikus neveken ismertek az összetartozó adatok, használatos a jegyzék (directory), ösvény fogalom.
- Az eszköz fogalom, ami szimbolikus neveken ismert struktúrált, vagy struktúrálatlan adat forrás, vagy adat nyelő periféria lehet. A fájlok és az eszközök az I/O absztrakciókban össze is foglalhatók.
- A felhasználói kapcsolattartó rendszer, ami parancsértelmező burok, vagy grafikus felület lehet. (Ezt az absztrakciót a Számítógépek, számítógéprendszerek c. tárgyban tanultuk, most nem lesz témánk.)

A tantárgy keretei belül szeretnénk azt is tisztázni, hogyan valósulnak meg ezek az absztrakciók, milyen elgondolások, megfontolások vannak mögöttük. A megvalósításokhoz használunk bizonyos "technikákat", ezekkel bizony foglakoznunk kell. A technikák közül néhány felsorolva:

- Processz kreáció és vezérlés.
- Időkiosztás (scheduling) processzorra, diszkekre.
- *Megszakítások* (interrupt) és lekérdezés (polling).

- Eseménykezelés és szinkronizációk.
- Konkurrencia vezérlés, kölcsönös kizárások, holtpont helyzetek elkerülése.
- Processzek közötti kommunikációs technikák.
- Kliens-szerver koncepció.
- Fájlok elérése és blokk-gazdálkodás.
- Bufferezés.
- Memória menedzselés (lapozási, szegmentálási technikák).
- Védelmi, hitelesítési (authentication), felhatalmazási (authorisation) technikák.

Mindezeket átgondolva a tanagyagunk 5 nagyobb csoportba foglalható össze.

1. A operációs rendszerek fogalma, története, struktúrája, osztályozásaik stb.
2. A processz koncepció és kapcsolódó technikák. Ezt a témakört több héten át fogjuk tárgyalni.
3. Az I/O rendszer, fájlrendszerek megvalósításai, technikái, algoritmusai. Ez is több hetes tananyag.
4. A memória menedzsment.
5. Rendszermenedzseri feladatok, beleértve a védelmet, a nyílt rendszer elvet.

A tárgy ütemtervét a hirdetőtáblán, a helyi WWW szolgáltatón részletesebben is megtalálják.

A módszerek

Az előadásokon főleg a tananyag lényeges részeit fogjuk tárgyalni, van ismeretátadó szerepük is, mégis inkább a megértést szolgálják elsősorban. Az előadásokon szereplő kivetített ábrákhoz, mintaprogramokhoz több módon is hozzájuthatnak. Egyrészt on-line adathordozókon most is megkapják az előadások meglehetősen részletes vázlatait, velük együtt az ábrákat, mintaprogramokat is. Az ábrák és mintaprogramok papír mediumon - önköltséges áron - a tanszék adminisztrációjában előre is megvásárolhatók.

Ajánlhatok könyveket is, ezek azonban többnyire angol nyelvűek és nehezen hozzáférhetők: az egyetemen egy-egy példányuk van egyelőre.

1. Tannenbaum: Modern Operating System, Prentice-Hall, 1992
2. Kóczy, Kondorossi szerk.: Operációs rendszerek mérnöki megközelítésben, Panem, 2000
3. Vahalia: UNIX Internals, Prentice Hall, 1996
4. Bach: The Unix System, Prentice-Hall, 1985
5. Deitel: Operating Systems, Addison-Wesley, 1987
6. Leffler et als.:The Design and Implemetation of the 4.3 BSD UNIX Operating System, Addison-Wesley, 1990
7. Silberschatz, Galvin: Operating Systems Concepts, Addison-Wesley, 1994

A gyakorlatok

A gyakorlatokat a tanszéki laborban tartjuk. A gyakorlatokon "vezetett" ismeretátadás nem lesz, a gyakorlatvezetők konzultációs kérdésekre válaszolnak, kiadják és beveszik, ellenőrzik az évközi feladatokat, minősítik azokat.

Néhány kisebb lélegzetű feladatot fognak kapni.

1. Rövid (max. 1 oldal) dolgozat készítése egy kereskedelmi forgalomban beszerezhető, konkrét operációs rendszerről. Saját maguk választhatnak operációs rendszert. Fel kell tüntetni az ismertetés forrását is. A feladat minősítésébe beleszámít, mit tart fontosnak elmondani egy oldalon az operációs rendszerről (és mit hagy ki), milyen forrásművet választ stb.

2. Processz kreáció, processzek közti üzenetváltás pipe-okon, üzenetsorokon (messages) C nyelvi programok segítségével. Specifikálniuk és programozniuk kell.

3. Kis programok egyéb IPC technikákra, szignálozás, osztott memóriahasználat, szemafor mechanizmus használata stb. Meg lehet ezeket oldani egyenként programozva a technikákat, vagy bizonyos összevonásokkal is.

4. Rendszermenedzseri feladatok gyakorlati bemutatása Linux-on: felhasználók felvétele, fájl rendszer készítése, mount-olása, szétesett fájlrendszer rendbetétele, mentések.

A tárgy teljesítése, számonkérés

Az aláírás feltétele a kis feladatok elfogadható szintű elkészítése, egy évközi zárthelyi dolgozat eredményes megírása. Ezt a gyakorlatvezetők minősítik. A tárgyat szóbeli vizsga zárja. A vizsgakérdéseket a félév végén megkapják. A vizsga jegy meghatározásába a gyakorlatvezetők minősítéseit - módosító faktorként - beszámítom. Felhívom a figyelmet, hogy a tárgy tananyaga szigorlati tananyag is.

Miskolc, 2002. szeptember

Dr. Vadász Dénes

1. Operációs rendszerek fogalma, struktúrája

A legtöbb számítógép felhasználó használja az operációs rendszereket, azok szolgáltatásait, anélkül, hogy pontosan meg tudná fogalmazni, mi is az operációs rendszer. Tudják, hogy egy rendszer szoftver, ami kezeli a gépet, parancsokat tud fogadni, tartoznak hozzá eszközök, állományok, katalógusok, ezekben lehet manipulálni, stb. De ha definiálni kell az operációs rendszert, akkor gondban vannak.

Ennek a bizonytalanságnak az az oka, hogy az operációs rendszerek alapvetően két, egymástól független funkciót (funkciócsoportot) valósítanak meg, két, egymástól független nézőpontból szemlélhetők, két szempontból definiálható a lényegük, és rendszerint nem tisztázzák a definíció során, hogy melyik nézőpontból történik a definíció, vagy az operációs rendszer fogalom meghatározásakor éppen keveredik a két nézőpont.

Tanulmányaink során találkoztunk már hasonló problémával. Példaként említhetem a *burok* (shell*)* illetve az *architektúra* fogalmat. A *shell* szó kimondásakor is két dologra gondolhatunk. A shell egyik értelemben egy *parancsértelmező*, egy *folyamat* (process), ami *készenléti jelet* (prompt) ad a felhasználó *terminájára* (v. terminálemulációs ablakában), jelezve, hogy kész *parancsot* fogadni a *szabványos bemeneti csatornán* (standard input), azt elemzi, és vagy ugyanebben a *folyamatban* vagy újabb *folyamatot* indítva valamint csinál.

A *shell* szó másik értelmében egy *programnyelv,* amiben vannak *vezérlő szerkezetek* (soros végrehajtás, elágazások, ciklusok) és *adatszerkezetek* (shell változók és szöveglánc konstansok). A *shell script* kifejezés használatakor erre a második definícióra gondolunk.

Az összefüggés a két nézőpont definíciója között tiszta: a shell értelmező képes feldolgozni a shell programot - akár interaktív akár kötegelt (batch) módon, de a két értelmezés különbözőségét tisztán kell látnunk.

A másik példán az architektúra fogalom. Első értelmezésünkben az *architektúra* egy digitális számítógép általános specifikációja, beleértve az utasításkészletét (instruction set), társszervezését (memory organization), címzési módokat, a B/K műveleteket, sin struktúrákat és vezérlésüket. Az ebből a szempontból nézett architektúra azonosság biztosítja pl. a számítógépek helyettesíthetőségét (compatibility), csatlakoztathatóságát.

Villamosmérnöki (hardver tervezési) szempontból az *architektúra* egy számítógép fő elemei kapcsolódásának leírása, blokkdiagram v. áramköri rajzok, stb. formájában. Az architektúra itt a konfigurációt írja le.

Térjünk vissza az operációs rendszerekhez. Az operációs rendszer hasonlít egy *kormányhoz*, azaz maga nem teljesít valódi funkciót, de lehetőséget ad az *alkalmazások* hasznos tevékenységéhez. Van erőforrás kiosztó (resoruce allocator) és vezérlő (control program) szerepköre. Néha úgy szemléljük az operációs rendszert, hogy az azon programok gyűjteménye, amit a számítógéprendszer szállító a géphez szállított. Máskor: azon programok, amik mindig futnak a számítógépünkön.

Egy jó módszer az operációs rendszer definiálására, ha megragadjuk kettős természetét: az operációs rendszer egyrészt *virtuális gép*, másrészt *erőforrás menedzser*. (Néha így definiálják: *válaszoló gép*).

1.1. Az OS mint kiterjesztett gép (Extended Machine, Virtual Machine)

A legtöbb számítógép gépi nyelvű programozása - különösen a B/K műveletekre gondoljunk - bonyolult, sok odafigyelést igénylő munka. Jól kell ismerni az *architektúrát* (az első értelemben vett módon!), a hardver részleteket. Gondoljunk végig például egy floppy diszk blokk behozatal forgatókönyvet!

A legtöbb programozó (nem is beszélve az általános felhasználóról) nincs olyan intim kapcsolatba az architektúrával, hogy ezt le tudná programozni!

Az operációs rendszer - mint kiterjesztett gép - magasabb absztrakciós szintet biztosít a felhasználó számára. Az eszközöket és állományokat *szimbolikus neveken* engedi kezelni, ezekben magasabb szintű operációkat biztosít (pl. open, read, write *rendszerhívásokat* (system calls, lásd később részletesebben)), sőt, az operációs rendekhez kötődő *parancsértelmezőkkel* még magasabb szintű parancsokat (pl. copy, move, stb.).

Úgy is mondhatjuk, ebből a szempontból nézve az operációs rendszer *elrejti a részleteket* a felhasználó elől, levesz bizonyos *felelősséget* a felhasználó válláról, akár különböző architektúrákon is biztosítja *helyettesíthetőségét*, *egységességet* biztosít a hasonló de részleteikben nagyban különböző eszközök (pl.: floppy diszkek és hard diszkek) kezelésére. Ez egy felülről-lefelé (top-down) megközelítése a problémának. A *virtuális gépet*, amit az operációs rendszer biztosít, könnyebb programozni, mint az alatta létező hardvert. Persze, hogy ezt hogyan biztosítja, ez egy hosszú történet, az Operációs rendszerek tárgy egyik célja, hogy ezt is megismerjük.

Ha úgy tetszik, ebből a szempontból *kényelmessé teszi* (convenience for the users) az operációs rendszer a hardver használatot.

1.2. Az OS mint erőforrás menedzser (Resource Manager)

Egy másik - valójában alulról-felfelé való (bottom-up) megközelítésben az operációs rendszer azért van, hogy egy összetett rendszer részeit menedzselje.

Egy korszerű számítógép processzorokból, tárakból, óraeszközökből, diszkekből, mágnesszalagos tárolókból terminálokból, nyomtatókból, hálózati eszközökből, stb. tevődik össze. Az operációs rendszer feladata, hogy elossza ezeket az erőforrásokat a gépen futó különböző, az erőforrásokért tulajdonképpen vetélkedő programok számára. (Példa lehet itt is: forgatókönyv arra az esetre, amikor két processz ugyanarra a nyomtatóra akar nyomtatni.)

Milyen erőforrásokat kell menedzselnie az operációs rendszereknek?

- A hardver erőforrásokat (processzorok, elsődleges és másodlagos tárak, eszközök stb.),
- a szoftver erőforrásokat (alkalmazások, adatbázisok stb.) és
- az emberi erőforrást) felhasználók, operátorok, rendszermenedzserek stb.).

A menedzselési feladatkörbe az erőforrás kiosztás mellett természetesen beleértjük az erőforrás védelmet (kölcsönös kizárást kritikus esetekben) a konfliktusok feloldását az erőforrások használatának számbavételét (statisztikák készítését, számlázásokat is). Olyan fogalmak merülnek itt fel, mint a hatékonyság, a teljesítmény, a védelem és biztonság, a megbízhatóság stb.

Ha úgy tetszik, ebből a szempontból az operációs rendszer *hatékonyá teszi* (efficiency) a hardver használatát.

1.3. Az operációs rendszerek története

Miután az operációs rendszer elég szorosan kapcsolódik a hardver-struktúrákhoz, történeti fejlődésüket, generációikat a hardver fejlődési generációkhoz köthetjük. Ez az összekapcsolás meglehetős erőltetett, de ad egy bizonyos strukturáltságot. Persze nem kezdjük a kezdet kezdetén - elegendőnek látszik, ha a századunk közepe táján épített, a Neumann elvnek megfelelő csöves számítógépeket tekintjük az első generációnak.

A történeti áttekintés során a számítógép használók munkaköri specializálódásának alakulására, a munkamegosztásbeli fejlődésre is kitérünk, illetve megemlítjük a programozási módszerek fejlődési fokozatait is. .

Durva felosztás szerint megkülönböztethetünk

- hardvereseket, számítógép-építő villamosmérnököket, (HW kezelése, tesztelése, stb.);
- rendszerprogramozókat, rendszer menedzsereket (OS kezelése, fenntartása);
- operátorokat (OS kezelés, eszköz kezelés, foglakozás a felhasználókkal);
- programozókat, beleértve szervezőket is (alkalmazások készítése, stb.);
- mezei felhasználókat (alkalmazások futtatása, stb.).

1.3.1. Az első generáció (1945-1955): Csövek és dugaszoló táblák (Vacuum Tubes and Plugboards) (Prelingual Stage)

H.Aiken (Harward), Neuman J. (Princeton), J.P. Eckert és W.Manchelty (Univ. o. Pennsylvania) és K.Zuse (Németország) - többek között - a 40-es évek közepén már építettek csöves számítógépeket. Ezek szobákat töltöttek meg, nagy áramfelvételük volt - számolási teljesítményük sokkal kisebb, mint a mai legolcsóbb házi számítógépeké.

Ebben az időben még nem volt munkamegosztás: ugyan azok az emberek tervezték és építették, programozták a gépeket, kezelték, futtatták a programokat, elemezték értékelték az eredményeket.

Az "építő-programozó-felhasználó" - ha már létezett a gép - dugasztáblákat készített elő, "írta" a programot és rögzítette az adatokat, remélve, hogy egyetlen cső sem robbant le, berakta a dugasztáblát a számítógépbe és órákat várt, míg néhány kijelző lámpán megjelentek az eredmények.

Az 50-es évek elején annyit fejlődött a dolog, hogy a dugasztáblák helyett lyukkártyákat is lehetett használni a programozáshoz, adatbevitelhez.

A programozás gépi nyelven történik, operációs rendszer még nincs.

1.3.2. Második generáció (1955-1965) Tranzisztorok és kötegelt rendszerek (Exploiting Machine Power)

A tranzisztorok megjelenése radikális változást hozott. Egyrészt kisebbek lettek a számítógépek és alacsonyabb lett a villanyszámla is, másrészt megjelent a munkamegosztás a számítógépes emberek körében. Megkülönböztethetjük a tervezőket és építőket, az operátorokat, a programozókat és a karbantartókat. Az első és utolsó csoport villamos mérnökökből verbuválódik, akár lehetnek ugyanazon személyek is. Az operátor szerepkör az időszak elején még nem létezik, de közben kialakul: gyakorlott perifériakezelők veszik át az eszközök kezelését a programozótól. Nagyobb intézmények, kormányszervek, egyetemek engedhetik meg, hogy számítógépeket beszerezzenek (nemcsak számítógépgyártóknál vannak már számítógépek)

géptermeket berendezzenek, ahol a karbantartók és operátorok (gépkezelők) dolgoznak. Nem vált szét még a programozói és felhasználói szerepkör.

A programozó lyukkártyára lyukasztja a futtatandó programjait, amit assembly nyelven vagy FORTRAN-ban irt. Lyukkártyára kerülnek az adatok is. Ezeket kártyakötegekbe rakják és az így összeállított munkát (job) átadják az operátornak (Eleinte, amíg nincs operátor szerepkör, operátorként tevékenykedik a programozó-felhasználó). Az operátor előbb beolvastatja (load) a FORTRAN fordító programját (ez is kártyaköteg) és elindítja (használja ehhez a Front Panel Switch-eket). Maga a forrásprogram "adat" a fordítónak, amit lefordít (translate), eredménye lyukkártya (esetleg lyuk-szalag). Ezt betölti (load), és elindítja (execute), beolvasva az adatokat. Az eredmények lyukkártyára lyukasztva, esetleg nyomtatóval táblákba listázva jelenhetnek meg. Megfigyelhetjük a "load-translate-load-execute" tevékenység sorozatot. A munkák egy *konzolról* futtahatók. Ha a futó program "lerobbant", vizsgálhatták a memóriatartalmakat, beállíthatták azokat, továbbfuttathattak, mindezt a konzolról lehetett elvégezni. Végül az eredményeket nyomtatták, lyukszalagra, kártyákra lyukasztották. Ehhez fejlődött a hardver közben: egyre általánosabbak lettek a kártyaolvasók, sornyomtatók, az évtized végére a mágnes-szalag olvasók-írók. Fejlődés volt a szoftverekben is: assamblerek, loaderek, linkerek alakultak ki, többszörösen használható rutinokat másoltak a programokba. Fontosak lettek az I/O rutinok, az újabb és újabb eszközökhöz újabb "device driver" programok készültek. Kialakultak a fordítók is: FORTRAN és COBOL. (Increasing the Expressive Power)

Az időveszteségek csökkentésére kialakul a következő számítógép fajták közötti munkamegosztás: egyszerűbb és olcsóbb gépet használnak arra, hogy a munka kártyakötegeket mágnesszalagra tegyék, előkészítendő az igazi futtatást. Egy másik, viszonylag drágább számítógép végzi az "igazi" számításokat - ez a szalagról - gyorsabban betölti a munkához szükséges programokat, az eredményeket ismét szalagra rögzíti. Az output szalag utána áttehető az olcsóbb - ún. karakterorientált - gépre: az eredményeket az lyukasztja ill. nyomtatja ki. Ez a munkamódszer volt az ún. klasszikus *kötegelt feldolgozás* (batch system), ahol az operációs rendszer tulajdonképpen a "load-translate-load-execute" tevékenység sorozatot automatizálta.

Tipikus *off line* előkészítő, és nyomtató gép volt az IMB 1401, míg a feldolgozó: az IBM 7094 ebben az időben. Megnőtt az operátorok szerepe: ők gyűjtötték össze kötegekben a munkákat, vitték át a szalagot a másik gépre, stb.

Az évtized végére oda fejlődött a dolog, hogy *fordítók* programjait már nem kellett a kötegbe kártyaformában beilleszteni, elegendő volt csak egy *vezérlőkártya* beillesztés: a feldolgozó számítógép az input szalag mellett a fordítót (FORTRAN volt ekkor!) egy másik szalagról betöltötte, a köteg futtatása megtörténhetett. Ezzel egyidőben egyre növekedett a fordítás mellett a könyvtári függvények (library rutines) szerepe: a fordítót tartalmazó szalagon könyvtári rutinokat is elhelyeztek, amiket a programok hívhattak, amiket a futtatható programhoz hozáillesztettek. Még később a fordítót, könyvtárakat tartalmazó szalagokra segédprogramokat (utilities) is helyeztek, és ezeket *vezérlőkártyákkal* lehetett betöltetni, aktivizálni: ez így már egy *működtető rendszer* volt, "egy-job-egy-időben" feldolgozással.

A *működtető rendszerek* - az operációs rendszerek ősei, az egy konzolról használható *memóriarezidens monitorok* voltak. Fő részeik: a Job Control Card értelmező; a job ütemező; és a betöltő (loader).

1.3.3. Harmadik generáció (1965-1980): Integrált áramkörök, multiprogramozás

A 60-as évekre két, meglehetősen különböző számítógépfajta alakult ki. Egyik az ún. szószervezésű, nagy, tudományos számításokra alkalmas gépcsalád volt (mint a 7094), elsősor-

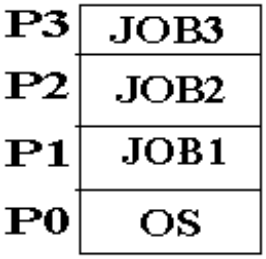
ban a numerikus számításokban jeleskedett. A másik család a karakterorientált gépcsalád, kisebb, és drágább gépek voltak ezek, jól alkalmazhatták adat átalakításra (lyukkártya -> szalag konverzió), rendezésre, nyomtatásra, stb. Előbbieket főleg a tudományos és mérnöki számításokra, utóbbiakat az üzleti életben (bankok, biztosítók, kereskedelmi társaságok) használták elsősorban.

Elő-előfordult, hogy egy cég kezdett dolgozni az olcsóbb, adatfeldolgozó gépen, és igénye támadt a nagyobb, számításigényesebb munkákat is kiszolgáló gépre, mialatt a két vonal meglehetősen különbözött , meg kellett volna venniük mindkét gépfajtát. Mi lett az eredménye ennek a kihívásnak? A két gépfajta "integrálása".

Az IBM válasza a System/360 rendszer volt. Ez tulajdonképpen egy gépsorozat, melyek szoftver kompatibilisek voltak, teljesítményükben (maximális memória, CPU sebesség, I/O eszközellátás) különböztek. Lehetett 360-as rendszert vásárolni adatfeldolgozáshoz, vagy tudományos számításokhoz is. Mellesleg ez volt az egyik első gép, ami IC-ket (igaz, alacsony sűrűséggel) is tartalmazott. És mellesleg azt is megjegyzem, hogy a 360-as leszármazottai a későbbi (egyre fejlettebb technológiákat használó - 370, 4300, 3080 és 3090-es rendszerek. A 360-as gép operációs rendszer az OS/360 nevet viselte. Óriási méretének, komplexitásának oka: nagyon széles igényeket (adatfeldolgozás, tudományos számítások, sok, változatos perifériakezelés, a HW fejlődés mellett a kompatibilitás tartása) kellett kielégíteni.

Itt jelent meg a *multiprogramozás*. Míg a 7094 processzora, ha befejezett egy számítást, várt az eredmény kivitelre, az újabb job betöltésre - ami tudományos számításoknál még elment, de adatfeldolgozásnál a gyakori I/O miatt veszteséges lett volna -, a 360-asnál ezt nem engedhették meg.

A megoldás a memóriát *partíciókra* osztották, és a partíciók mindegyikébe betöltöttek egy-egy munkát. Mikor egyikkel végzett, a CPU veszteség nélkül átkapcsolhatott egy másik feldolgozásra:



1.1. ábra. Memória partíciók

Természetesen, megoldották, hogy a partíciókba betöltött munkák ne zavarják egymást - hardveres védelem volt, hogy át ne címezhessenek.

A másik alapfogalom is megjelent: a *spooling* (Simultaneous Peripheral Operation On Line). Ennek lényege: a munka kötegek kártyaolvasói a rendszer diszkjeire kerültek (nem szalagra), és egy partíció kiürülése esetén gyorsan betöltődhettek a partícióra, kevesebb volt a CPU veszteségidő, nem volt szükség már a szalagokon való munka kötegek gyűjtésére, a szalagok szállítására. Kialakult egy új adatstruktúra: a *job pool*. A job pool munkáiból választhat az operátor, vagy az operációs rendszer egy-egy munkát futásra.

Persze, gondok maradtak: hogy készítsék elő a munkák programjait: azok melyik partícióba fussanak? Fix méretű partícióknál ha nem férnek be, mégis csak van veszteségidő. Ha az operátor nem jól szervez, kiürülhetnek a várakozó sorok.

Válaszként hamarosan kialakult az *időosztásos multiprogramozás* és a *változó méretű partíciókkal* működő memóriagazdálkodás. Az előbbiek: időszeleteket kapnak az egyes partíciókba betöltött munkák, és látszólag párhuzamosan futhatnak (Time Sharing, CPU-Switch, idő kiosztás). Ezzel egy partíciót fenntarthatunk arra, hogy végezze a spooling-ot. Utóbbi nagy segítség: nem kell előre eldönteni a partíciót. Egyre kritikusabbá vált az *erőforrás kiosztás* (resource allocation) és a *védelem* (protection).

Újabb igény merült fel: az *interaktivitás* igénye. Kényelmetlen a batch feldolgozás: várni az eredményekre, kis hiba nem javítható azonnal, stb. Legyen több száz felhasználót egyidejűleg kiszolgáló kényelmes rendszer! A hardver fejlődik: terminál vonal *nyalábolók* (multiplexers) lehetővé teszik, hogy a programozók termináljairól (eleinte írógépszerű eszközök, később katódsugaras terminálok) adhatják meg a JCL utasításokat. Ezekből fejlődtek ki a *parancs-nyelv-értelmezők*. A programozó-felhasználóknak on-line f*ájl-rendszereket* biztosíthatnak, a fájlok csoportokba (cluster, directory) rendezhetők. Biztosítható a fájlokhoz a többszörös hozzáférés. De ne feledjük, ugyanekkor a szokásos kötegelt feldolgozás is megy, az is fontos. Megjelenik a *processz* fogalom, kialakul a *virtuális memória* koncepció.

Fejlődött a hardver: kialakulnak a "kis gépek" (DEC, PDP-1(1961), VAX-780(1978).

Fejlődtek az operációs rendszerek: a MULTICS (ez egy nagy gépre sok interaktív felhasználót szolgált volna) és UNIX (a PDP-7-en!) ekkor alakul ki, ezek már általános célú, multiprogramozású, időosztásos rendszerek voltak.

A munkamegosztás is fejlődött: vannak fejlesztők (elektromérnökök), karbantartók, hardveresek (elektromérnökök), rendszerprogramozók (az operációs rendszerrel foglalkoznak, fejlesztik, illesztik stb.), operátorok (kezelik a rendszert), programozók és felhasználók (az időszak végére). Az ún. nagygépeken kötegelt feldolgozás folyik: munka vezérlő kártyaköteget (Job Control Language kártyák) állít össze és ad át az operátornak a programozó-felhasználó, a forrásprogramokat már diszken tárolják. A kisgépeken terminál előtt ülnek, interaktív parancsnyelvet használnak.

A programozás-történethez megjegyezzük: egy sor *imperatív* és *funkcionális* programnyelv alakul ki (Algol, PL/1, APL, Lisp, Basic, Prolog , C stb.), megjelenik az első objektumorientált nyelv (Smalltalk, 1972), jellemzi az időszakot az ún. *szoftver krízis*. (Reducing the Machine Dependency, Increasing Program Correctness).

1.3.4. Negyedik generáció (1980-1990): Személyi számítógépek, LSI, VLSI (Reducing the Complexity)

A technológia gyorsan fejlődött. Az LSI (Large Scale Integration), később a VLSI (Very Large Scale Integration) lehetővé tette, hogy nagy mennyiségben és viszonylag olcsón állítsanak elő számítógépeket. Ez volt a *személyi számítógépek* (Personal Computer) hajnala. A PC arhitekturája olyan, mint a PDP11 kisszámítógépé, ára viszont csak a töredéke! Nemcsak vállalatok, de magánszemélyek is vásárolják. Munkahelyeken: mindenki számára saját PC biztosítható.

Következmény:

- Visszaesés a védelemben, hiszen mindenki csak a saját rendszeréért felel!
- Óriási az interaktivitás: mindenki parancsnyelveket tanul.
- Felhasználóbarát felhasználói kapcsolattartók kellenek: ne kelljen "guru"-nak lennie a felhasználónak.
- A PC játékokra is jó. Mindenki használja, mindenki "szakértővé" válik.

Gond:

- A gyenge védelem hozza a "vírusokat".
- Óriási kavalkád. Hogy lesz itt egység?
- Tévedések: C64 játékgép professzionális felhasználása.

A személyi számítógépek működtető rendszerei eleinte egyfelhasználós és egytaszkos jellegűek. Céljuk nem a teljesítménynövelés (a CPU és a perifériahasználatra vonatkoztatva), hanem a kényelmes használat (convenience) és a válaszképesség. Eleinte egy *monitor* és a *BASIC értelmező* volt csupán a PC működtető rendszere.

Az évtized végére: megjelenik a *munkaállomás* (Workstation), ami tulajdonképpen erőforrás-gazdag személyi gép (Sun, HP/Apollo, IBM RS6000, DEC munkaállomások, SGI munkaállomások stb.), hálózatra kötve, jó grafikával, növekvő szerepűek a *hálózatok* (networks) és a *párhuzamos rendszerek (paralell systems)*. (Hermes, Linda, paralell C, Java).

Ezek miatt már felmerült a szükség igazi operációs rendszer funkciókra: a megnövelt védelemre (a vírusok, worms-ok ellen, a hálózatba kapcsolás miatt); multitasking-ra (a grafikus felhasználói felülethez, a kényelemhez). (OOP: C++, Java).

Az operációs rendszerek:

- MS-DOS különböző verziói, igen nagy számban értékesítik.
- Macintosh Apple operációs rendszere: az első "ablakozó" felhasználói felülettel.
- Unix származékok (SUN OS, Solaris, HP Unix, AIX, OSF, DEC Unix, Irix stb.), Windows NT a munkaállomásokra.
- OS2 és a Win 95 PC-kre.

Végül:

- Hálózati operációs rendszerek,
- Osztott operációs rendszerek is jelentősek.

És lassan itt vagyunk a mában!

Fontos évszámok, események

1941	Zuse Elektromechanikus kalkulátora, 64 szavas memória, 3 sec a szorzás
1944	MARK I, Elektromechanikus számítógép, Aiken
1945	ENIAC, Electrical Numerical Inegrator and Computer, Mauchly, Eckert
1948	Az első tranzisztor, Bell Lab
1949	EDSAC, Turing, az első szubrutinkönyvtár; UNIVAC I., az első assambly nyelv
1951	EDVAC, Neumann csoportja
1952	1. kereskedelmi fordító (compiler); mikroprogramozás először, Wilkes
1954	US Defense Dept. vásárol számítógépet: UNIVAC I-et (Harvard); FORTRAN, IBM; IBM Assembler; IBM 650, az 1. tömegben gyártott gép
1955	TRIDAC, az 1. Tranzisztort használó gép
1957	Megalakul a DEC; IPL (Information Processing Language)
1958	ALGOL58, ALGOrithmic Language; LISP, LIStProcessing Language
1959	COBOL, Common Business Oriented Language
1960	ALGOL60, Európában népszerű
1962	CTSS, Compatible Time Sharing System
1964	LAN, az 1. helyi hálózat; PL/1 és APL, IBM
1965	Control Data 6600, az 1. sikeres kereskedelmi forgalmú gép BASIC, Beginner's All-purpose Symbolic instruction Code

	MULTICS, MIT: Simula: ARPANet
1966	OS/360
1968	THE, Dijkstra: Burroughs B2500/3500 az 1. kereskedelmi forgalmú gép, ami IC lapkákat használ
1969	Laser printer
1970	Pascal; RC4000 kernel
1971	Intel mikroprocesszor lapka
1972	C nyelv; Smalltalk
1973	A Unix használata általános
1974	Xerox Alto, az 1. munkaállomás
1975	Az 1. PC, Apple, Radio Shack, Commodore PET
1976	MCP, multi-processing rendszer; SCOPE, multi-processing rendszer
1978	VAX
1979	3BSD Unix; Ada
1981	IBM PC
1982	Compaq, az 1. hordozható számítógép; Turbo Pascal, Modula2
1984	Apple Macintosh, grafikus felület; TrueBASIC; SunOS; PostScript
1986	C++
1987	OS/2; X11
1988	NeXT, Unix munkaállomás, objektumorientált grafikus felhasználói felület
1989	Motif szabvány
1990	MS Windows 3.0
1992	Solaris; Modula3
1993	Winows NT

1.4. Direkt futtatás a hardveren - működtető rendszer

Használható a számítógép működtető rendszer nélkül? Ezt a használati módot nevezzük a *hardveren való direkt futtatásnak.*

A válasz: tulajdonképpen igen, de csak a kis bit-szélességű mikrokontrollereknél szokásos ma már. Régebben természetesen ez a futtatási mód is megvolt.

Ekkor persze minden felelősség a programozóé! Teljes részletességben ismernie kell a hardvert, az utasításkészletet stb. És felmerül a további kérdés: hogyan programozható a gép? Hogyan "juttatjuk" be a programot a memóriába? Hogyan indul el a program? (Rövid válaszok erre: külön berendezéseken programozzuk, "beégetjük" a programokat a memóriába, bekapcsolással indulhatnak a beégetett programok.)

Egy általános célú számítógéprendszer persze működtető szoftver nélkül nemigen használható. Legalább egy *monitor* [1]program kell hozzá.

---

[1] A *monitor* kifejezés meglehetősen túlterhelt. Használjuk *működtető rendszer* neveként, néha egy *megjelenítő* neveként, a VAX/VMS egy *segédprogramjának* is ez a neve, és az egyik processzek közötti kommunikációnak,

A monitor működtető program

A *monitor* futtatható szubrutinok gyűjteménye, melyeket rendszerint a ROM-ban tárolnak (nehogy a felhasználó felülírja azokat).

A monitor rutinjai képesek *karaktersorokat* fogadni egy *konzol terminál* billentyűzetéről, ezeket a sorokat egyszerű *parancsként* értelmezni, a parancsokhoz rendelt egyszerű funkciókat végrehajtani, és természetesen képesek a *konzol terminál* megjelenítőjére küldeni karaktersorozatokat.

A monitort a gép gyártója biztosítja. Néha képes kezelni egy-egy mágneses háttértárolón (diszken, korábban dobtárolón) kialakított primitív fájl rendszert. Például gyakori, hogy jegyzék fogalom nélküli, ebből következően hierarchia nélküli - egyszintű - folyamatos blokk elhelyezésű fájlrendszert: ebben a fájloknak van nevük, hosszuk. A nevet, hosszt a fájl kezdő mezőiben tárolják. A fájlokat a monitor szekvenciális végigolvasással betöltheti (load) a memória adott címétől kezdődően, esetleg a fájlt a konzol képernyőjére írhatja (text dokumentum fájlokat), vagy irányíthatja egy nyomtató eszközre.

A monitor parancsai hallatlanul egyszerűek. Rendszerint vannak memória cella teszt és beállító parancsok (*examine mem-cím*, *set érték mem-cím*), fájl betöltő, kiíratő parancsok (*load filenév mem-kezd-cím*, *type filenév* stb.), és természetesen "futtató" parancsok (*go mem-cím*, *run mem-cím* stb.).

Beláthatjuk, hogy már az *examine/set/go* parancshármassal is "programozható" a számítógép.

Egy sor *set*-tel beírunk egy programrészletet (persze tudni kell a gépi instrukciók bináris/oktális/hexa kódjait!), majd a *go kezdő-cím* paranccsal lefuttatjuk a kis programot. Az eredményeket az *examine* paranccsal meg is nézhetjük.

Még jobb az eset, ha van *load/run* parancspár is!

Ekkor - rendszerint tesztprogramot -betölthetünk, utána elindíthatjuk. Szerencsés esetben a betöltött program írja az eredményeit a konzolra, rosszabb esetben az *examine* paranccsal nézegethetjük a teszt-eredményeket (már ha tudjuk, hova teszi azokat a program).

Néha a monitor az eddig említett parancsértelmező/végrehajtó rutinjai mellet tartalmaz néhány I/O rutint is. Ezek a nyomtatókezelést, a konzol megjelenítő és a billentyűzet kezelését, esetleg adott memória tartomány lementését egy fájlba (adott fájlnévvel az egyszintű fájlrendszer elejére/végére) tehetik lehetővé. A primitív I/O rutinok adott speciális címeken vannak, és a "felhasználói" programokból hívhatók egy "ugrás adott címre" instrukcióval. Ha úgy tetszik, ez már majdnem egy "I/O rendszer hívás".

Minden modern operációs rendszer ebből a primitív monitor működtető rendszerből nőtt ki.

Kérdés lehet: vannak-e manapság még monitorok?

A válasz: igen.

A gyártók elsősorban a hardveres mérnököknek szánva a monitort ma is gyárt így számítógépeket. Sokszor a gép bekapcsolásakor nem egy operációs rendszer töltődik, hanem egy moni-

---

processzek szinkronizációját biztosító mechanizmusnak is ez a neve. Ügyeljünk arra, hogy a monitor szó használatánál mindig a megfelelő kategóriára gondoljunk!

tor indul. Esetleg kapcsolókkal (CMOS RAM-ban beállított értékekkel) szabályozzák, OS töltődjön, vagy monitor induljon. A monitort kizárólag tesztelésre használják, a normális üzemmódhoz manapság biztos egy valódi operációs rendszert fognak betölteni.

1.5. Operációs rendszerek osztályozása

Több szempont szerint is osztályozhatjuk az operációs rendszereket. A legfontosabb osztályozási szempontok:

- az operációs rendszer alatti hardver "mérete" szerinti osztályozás szerint:
  - mikroszámítógépek operációs rendszerei;
  - kisszámítógépek (esetleg munkaállomások) operációs rendszerei ;
  - nagygépek (Main Frame Computers, Super Computers) operációs rendszerei.
- A kapcsolattartás típusa szerinti osztályozás szerint:
  - kötegelt feldolgozású operációs rendszerek, vezérlőkártyás kapcsolattartással;
  - interaktív operációs rendszerek.

A következő osztályozási szempontok még fontosabbak, ezeket ezért részletezzük is:

- cél szerinti osztályozás;
- a processz kezelés, a felhasználók száma szerinti, a CPU idő kiosztása szerinti osztályozás;
- a memória kezelés megoldása szerinti osztályozás;
- az I/O koncepciók, a fájl rendszer kialakítása szerinti osztályozás.

1.5.1. Operációs rendszerek osztályai cél szerint

Megkülönböztethetünk *általános célú* és *speciális célú* operációs rendszereket.

Az általános célú operációs rendszerek több célúak: egyidejűleg használjuk azokat programfejlesztésre, alkalmazások futtatására, adatbázisok lekérdezésére, kommunikációra stb.

A speciális célú rendszerek osztálya igen gazdag lehet: vannak folyamatvezérlésre beállított, vannak tranzakció feldolgozásra implementált stb. rendszerek, közös jellemzőjük, hogy egyetlen célt szolgálnak.

1.5.2. Processz kezelés, időkiosztás, felhasználó szám szerinti osztályok

A több, változó feladatszétosztású processzorral rendelkező gépeket az operációs rendszerükkel együtt *multi processing* rendszereknek szokás nevezni.

Az egy processzoros gépek működtető rendszere lehet *single tasking* (egyidőben egy processz lehetséges), vagy *multi tasking* rendszer (kvázi párhuzamosságban több processz fut).

Az egyidejű felhasználók száma szerint beszélhetünk *egyfelhasználós* (*single user*) rendszerekről és *többfelhasználós* (*multi user*) rendszerekről. Az utóbbiak mindenképp *multi tasking* vagy *multi processing* rendszerek kellenek, hogy legyenek, az *egyfelhasználós* rendszer lehet *single tasking* is.

A CPU időkiosztása lehet szekvenciális (egy processz teljes feldolgozása után kapcsol a másikra), kooperatív-event polling rendszerű, megszakítás vezérelt (interrupt driven), vagy beavatkozó-megszakításvezérelt (preemptiv-interrupt driven).

Az *event polling* rendszer már lehet többfelhasználós/többfeladatos rendszer. A processzek között előre beállított, körkörös sorrend van. Az a processz lesz aktív, amelyik *eseményt* (billentyű lenyomás, ablakba belépés stb.) kap, és addig aktív, amíg új esemény aktívvá nem tesz egy másik processzt. A kooperatív rendszerekben egy-egy processz saját elhatározásából is lemondhat a processzorról, átadhatja a vezérlést a soron következő processznek.

A *megszakítás vezérelt* rendszerekben minden I/O megszakítás bekövetkezésekor újraértékelik a processzek prioritási állapotait, és a legmagasabb prioritású kapja a CPU-t. Ha nincs I/O megszakítás, a futó processz nem mond le önszántából a CPU-ról.

A *beavatkozó rendszerű* időkiosztásnál nemcsak az I/O megszakításoknál értékelik újra a prioritási állapotokat, hanem bizonyos óra megszakításoknál is. Elveszik a CPU-t a futó processztől akkor is, ha az továbbra is futásra kész állapotban van, ha találnak nála magasabb prioritásút. Az időkiosztási algoritmus szerint e rendszeren belül megkülönböztethetünk *klasszikus időosztásos* (*time sharing*) és *valós idejű* (*real time*) rendszereket, az utóbbiak az *igéretvezérelt időosztású* rendszerek egy alrendszerét képezik.

Jellemezzünk néhány ismert operációs rendszert!

MS DOS: személyi számítógépekre általános célú, egyfelhasználós, egyfeladatos (a TSR-Terminate and Stay Resident programokkal, illetve a *system* rendszerhívással többfeladatos ugyan, de egyidőben csak egy processz lehet aktív), ebből következően szekvenciális időosztású operációs rendszer.

MS Windows: általános célú, egyfelhasználós, többfeladatos rendszer. Sokan, köztük magam is, nem szívesen nevezik igazi operációs rendszernek. Az MS DOS fölé épülő felhasználói kapcsolattartó igazán, igaz, *kooperatív-event polling*-gal többfeladatossá teszi a DOS-t.

OS/2: általános célú, egyfelhasználós, többfeladatos rendszer. Igazi operációs rendszer: időkiosztása beavatkozó-megszakításvezérelt, sőt, hangolható és használható valós idejű rendszernek.

Windows NT: általános célú (kliens és szerver is), többfelhasználós, többfeladatos rendszer (multi-processing). Preemptive-interrupt driven az időkiosztása. Minden taszk védett.

Win 95: egyprocesszoros, általános célú (Home, Personal), Intel platformú, többfeladatos, beavatkozó-megszakításvezérelt, 32 bites alkalmazásokat is kezelő rendszer.

VAX/VMS: kis- és mikrogépek általános célú (de speciális célra hangolható), többfelhasználós, többfeladatos, sőt, multi processing rendszere. Időkiosztása hangolható beavatkozó időosztásos rendszernek, vagy valós idejű rendszernek is. Tanszékünkön csak egyprocesszoros VAX-ok voltak, és klasszikus időosztásos VMS implementációkat használtunk.

Unix rendszerek: általános célú, többfelhasználós, többfeladatos rendszerek. A korszerű Unix-ok képesek többproceszoros rendszereket működtetni, tehát multi processing jellegűek (nálunk pl. a régi *zeus* 4 porocesszoros, operációs rendszere az Irix multiprocessingben működik, de ugyanaz az Irix a kis Indigókat pusztán multi taskinggal is tudja működtetni). A klasszikus Unix-ok preemptive time sharing időkiosztásúak. Fejlesztik a valós idejű Unix alapú rendszereket (ilyen pl. az OS9, a QNX, az RTLinux). A mikro- és kisgépek, a munkaállomások, sőt, a nagygépek (szerverek) is működtethetők Unix származékokkal. Kapcsolattartóik szerint minden változat (interaktív parancsnyelvi, interaktív GUI, kötegelt parancsfeldolgozásos kapcsolattartás) előfordul.

1.5.3. Memória menedzselés szerinti osztályozás

Alapvetően megkülönböztetünk *valós címzésű* és *virtuális címzésű* rendszereket.

A valós címzésű rendszereken belül a *fix*, vagy *változó partíciókra* osztott memóriakezelés lehetséges (ezeknek további aleseteit lásd majd később).

A virtuális címzésű rendszerek alosztályai a klasszikus *ki/be söprő* (*swapping in/out*) rendszerek, a klasszikus *igény szerinti ki/be lapozó* (*demand paging*) rendszerek és a *ki/be söprő és lapozó* (*swapping and paging*) rendszerek.

MS DOS: valós címzésű, változó partíciókra osztó rendszer.

Windows NT: virtuális címzésű lapozó rendszer.

VAX/VMS: virtuális címzésű, lapozó és söprő rendszer.

Unix: virtuális címzésű rendszer. Az egyszerűbb (és korai) Unix-ok ki/be söprő rendszerek, ma már ki/be söprő és lapozó a memóriakezelés.

1.5.4. Az I/O koncepciók, a fájlrendszer megvalósítása szerinti osztályozás

Az osztályozási szempont nem elég pontos, sokféle megoldást találunk az egyes operációs rendszerek implementációiban. Mondhatjuk, vannak operációs rendszerek, melyek egy sajátos fájlrendszert képesek kezelni (pl. az MS-DOS a FAT-os fájlrendszert), és vannak, melyekhez tartozik ugyan saját fájlrendszer, de más koncepciójú fájlrendszereket is kezelnek (pl. a Linux meglehetősen sok fájlrendszert ismer). Megjegyezzük, hogy a mai operációs rendszerek a fájlrendszerük külső megjelenítési formáiban mind hierarchikus faszerkezetű struktúrát biztosítanak a jegyzékfogalom szülő-gyermek reláció megvalósításával.

Szűkíthetjük is a szempontokat: csakis a fájlrendszer implementációjában igen fontos két megoldandó feladat koncepciója szerint osztályozzuk magukat a fájlrendszereket.

A megoldandó feladatok (amik lehetőséget adnak az osztályozásra):

1. Hogyan rendelik az OS-ek a fájlnévhez a fájl blokkjait

Már nem használt megoldás a *folyamatos allokáció*. Használatos viszont az *indextáblás hozzárendelés*. A Unix-ok jellegzetes módszere az ún. *i-listás* hozzárendelés. Egyes fájlrendszerekben *kiterjedéseket* (extents) - amik folyamatos allokációjú blokkok egysége - rendelnek a fájlnevekhez.

2. Hogyan kezelik az OS-ek a szabad blokkokat

Használatos megoldás a *bit-térképek* vagy *foglaltsági térképek* alkalmazása. A térkép az eszközön lehet egyetlen folyamatos terület, de lehet megosztott, több darabból álló is. Másik, szintén gyakori implementációban *láncolt listán tartják nyilván a szabad blokkokat*.

MS DOS: Jellegzetes a DOS *FAT* (*File Allocation Table*) *táblás* megoldása, ami a fenti két feladatot egyben megoldja. A FAT tábla egyszerre foglaltsági térkép és indextábla.

Windows NT: ismeri a FAT fájlrendszert, van saját NTFS és CDFS fájlrendszere.

VAX/VMS: A fájl blokkok allokációja indextábla segítségével, a foglaltság bit térképpel megoldott.

Unix: A fájl blokkok hozzárendelés az említett *i-listán* található *i-bögök* segítségével történik, a szabad blokkok menedzselése a *superblock*-ból kiinduló *szabad blokkok láncolt listája* segítségével megoldott. Az egyes Unix megvalósításoknak lehet saját fájlrendszere is, és ismerhetnek más fájlrendszereket is (Pl. a Linux saját fájlrendszere az ext2 fájlrendszer, de kezelni tudja a FAT-os, a klasszikus Unix-os stb. fájlrendszereket is.

1.6. OS struktúrák

Már láttuk, mi az operációs rendszer. Az korábban is láttuk, kívülről hogy látszik: biztosít a felhasználónak egy szimbolikus nevekkel kezelhető eszköz- és fájlrendszert, láthatunk benne processzeket, látunk felhasználókat, van egy programozható burok, vagy egy könnyen kezelhető grafikus felhasználói kapcsolattartónk, kapcsolatot és ülést kell létesítenünk, hogy használhassuk.

Az az illúziónk, hogy a fájloknak a fájlrendszerben helyük és méretük van, a processzeknek pedig életük van.

Milyen az operációs rendszer belülről? Milyen szempontok szerint nézhetjük a struktúrákat?

A nézőpontok:

1. Milyen szolgáltatásokat biztosít az OS? Milyen funkcionális részei vannak?
2. Milyen felületeket (interface) ad a felhasználóknak, a programozóknak?
3. Komponenseit hogy állítják elő, köztük milyen interfészek vannak?

ad 1. A szolgáltatások szerinti komponensek (funkcionális struktúra)

- Processz (taszk, fonál) menedzsment komponensek;
  - A kreációt és terminálódást biztosító funkciók,
  - processz kontroll és ütemezési funkciók
    - állapot-nyilvántartás,
    - CPU ütemezés,
    - szinkronizációs mechanizmusok, beleértve e kölcsönös kizárási mechanizmusokat is,
    - processzközti kommunikációs mechanizmusok.
- Memória menedzselő komponensek;
  - Memóriahasználat nyilvántartása,
  - Memória allokálás/felszabadítás,
  - Címleképzés (mapping) segítése,
    - közben ki/besöprés vagy lapozás.
- Másodlagos tároló menedzsmentje,
  - szabad terület menedzselése,
  - blokk allokáció,
  - diszk blokk scheduling;
- I/O menedzsment komponensei,
  - bufferezés,
  - device driver interface elemek,
  - speciális driver-ek;
- Fájlrendszert megvalósító és menedzselő komponensek
  - Jegyzékstruktúra megvalósítás (directory implementáció, fájl attribútumok rögzítése),
  - Blokk-hozzárendelési módszerek,
  - Fájlok/jegyzékek kreálása, törlése, írása, olvasása,
  - Fájlrendszert létrehozó, használatba vevő, helyreállító segédprogramok,

    - mentések és visszaállítások segédprogramjai.
- Védelmi komponensek (néha "beépítve" az egyéb funkciókba, néha megkülönböztethető komponensként szerepel)
- Networking, hálózatkezelő komponensek (néha beleértik az I/O menedzsmentbe)
- Felhasználói kapcsolattartó rendszer (CLIs) (némely esetben nem is része a kernelnek),
    - parancsértelmező,
    - GUI.

ad 2. Interfészek a felhasználóknak, programozóknak

Ez a struktúra a funkcionális struktúrához közeli. Itt is azt nézzük, milyen szolgáltatásokat biztosít a kernel, de oly módon, hogy milyen a felület a szolgáltatás kérelemhez. A szolgáltatási felületeknek három nagy csoportja van:

- Rendszerhívások osztályai, rendszerhívások
    - A programozó felhasználó milyen API hívásokat programozhat. (Unix-oknál lehet SVID, BSD vagy POSIX szabvány)
- Rendszerprogramok, segédprogramok (utilities) osztályai
    - "Szabványosított" az összetettebb szolgáltatásokat biztosító segédprogramok készlete is.
- Kapcsolattartók (CLIs).

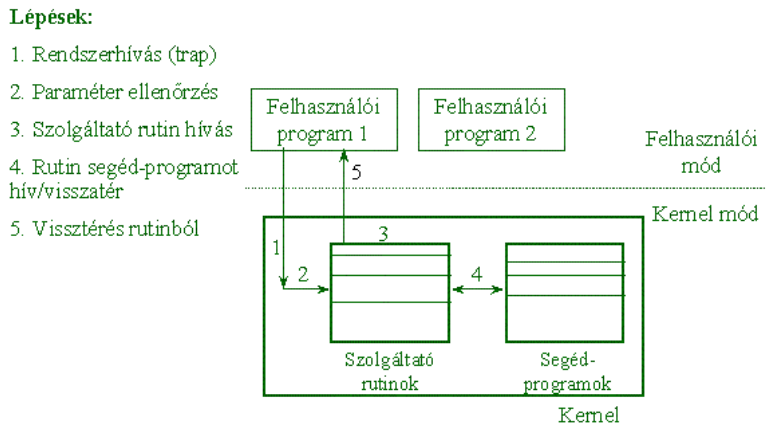
ad 3. Implementációs struktúrák:

- Egyszerű, monolitikus rendszer;
- Réteges rendszer;
- Virtuális gépek;
- Kliens-szerver modell;
- Vegyes szerkezetek.

Nézzünk meg néhány lehetséges struktúrát! Azt is láttuk már, hogy az operációs rendszer mint egy réteg, elválaszt a hardver részletektől. Valóban, az operációs rendszer *rendszermagja* (*kernel*) elválasztó réteg.

Mi most éppen arra vagyunk kíváncsiak, milyen szerkezetű lehet a rendszermag.

1.6.1. Monolitikus rendszer



1.2. ábra. Monolitikus struktúra

Az a struktúra, mikor nincs is struktúra (1.2. ábra).

A monolitikus rendszer magja névvel ellátott *szolgáltató eljárások* (*service rutins*) gyűjteménye, melyek hívhatók

- a felhasználói programból (processzből) úgynevezett *rendszer hívással* (*kernel call, system call*), vagy
- egy másik szolgáltató rutinból (call).

A két hívási mód megkülönböztetésének vannak okai:

- A *kernel call* kívülről egyszerű függvény- vagy eljáráshívásnak tűnik: megadjuk a szolgáltató rutin nevét és aktuális paramétereit. Valójában ez nemcsak egyszerű függvény- vagy eljárás hívás paraméter átadással, hanem egyben *trap*: a processzor *felhasználói módból kernel módra* vált. (A *trap* fogalmat később részletezzük.)
- Monolitikus rendszerben a kernel szolgáltató rutinjai egymást korlátlanul hívhatják. Ekkor már nem szükséges a módváltás, hiszen a kernelen belőli kódot a processzor kernel módban hajtja végre. Paraméterátadás természetesen szükséges lehet.

A szolgáltató rutinok természetesen adnak valamilyen szolgáltatást: kezelik a hardvert.

A szolgáltató rutinokból a visszatérések csak abban különböznek, hogy felhasználói programba való visszatérésnél megtörténik a futási mód visszaváltása felhasználói módra.

A monolitikus rendszer rutinjait assembly, esetleg magas szintű nyelven írják. Ezeket lefordítják (compile) és összeszerkesztik (linkelik) egyetlen betölthető programba, lementik egy fájlba. Ez a fájl a rendszerindításkor betöltődik és a kernel ott áll szolgáltatásra készen.

Ha nagyon akarjuk, a monolitikus rendszerek is strukturálhatók, rétegezhetők. A legfelső réteg lehet egy *diszpécser rutin*, ami a *trap*-et végrehajtja, a hívás paraméterit átveszi, esetleg ellenőrzi, végül átadja a vezérlést és a paramétereket az alatta lévő rétegbe szervezett szolgáltató eljárásnak. Ebbe a rétegbe az összes kernel hívással megszólítható szolgáltató rutint képzeljük: ez a második rétege a kernelnek. A szolgáltató rutin elvégzi a szolgáltatást, szükség esetén egy, harmadik rétegbe szervezett segéd-eljárást is hívhat. A harmadik rétegbe szervezett segéd-eljárások segítik a szolgáltatás biztosítását. Természetesen, a rétegek programjait meg kell írni, le kell fordítani, és össze kell szerkeszteni: itt lehet választani, hogy egy betölthető program fájlba szerkesztjük az összes réteg rutinjait, vagy külön betölthető fájlba az első és második, illetve másik fájlba a harmadik réteg rutinjait.

Ezzel eljutottunk a réteges struktúrájú OS kernelekhez.

1.6.2. Réteges struktúrájú OS-ek

Tipikus példa erre a Dijkstra professzor és hallgatói által készített THE nevű operációs rendszer (1968). A THE egyszerű kötegelt feldolgozási rendszerű operációs rendszer volt. A rendszernek 6 rétege volt:

5	Operátor	Operátor
4	Felhasználói programok	Független processzek
3	I/O menedzsment	Virtuális I/O eszközök
2	Operátor-processz kommunikáció	Virtuális operátor konzolok
1	Memória-dob menedzsment	Virtuális szegmentált memória
0	Processzor allokálás, multiprogramozás, szinkronizáció	Virtuális CPU-k

1.3. ábra. A THE rétegei

Mi is a rétegezés lényeges előnye?

- Egy réteg magasabb szintű operációkat biztosít a felette lévő számára és
- elrejti az alatta lévő részleteket.

Ugyanakkor jól meghatározott interfészek vannak közöttük.

A **0. réteg** kiosztja a CPU-t a processzeknek, kapcsolja a CPU-t köztük. E réteg felett egy-egy processz elől el van rejtve, hogy más processzek is vannak.

Az **1. réteg** feladata: egy-egy processz számára helyet biztosít részben a fő memóriában, részben a dobtáron. Igény esetén lapokat mozgat a dobtár és a fő memória között. E réteg felett egy processz nem kell törődjön, vajon kódja-adata a memóriában van-e, kezelheti teljes címtartományát.

A **2. réteg** feladata: kommunikációt biztosít egy-egy processz és az operátor konzol terminál között. Felette: minden processz úgy képzeli, van saját konzolja.

A **3. réteg** feladata: I/0 kezelés, bufferezés minden processz számára. Felette: egy processz absztrakt I/O eszközöket képzel magának, nem kell törődjön a különbözőségekkel.

A **4. rétegben** találhatók a felhasználói programok. Nem kell aggódniuk a CPU kiosztás, a memóriakezelés, a konzollal való kommunikáció és a I/O eszközök menedzselése miatt. Több program is lehet e rétegben.

Az **5. rétegben** van az egyetlen operátor processz. Látja maga alatt a felhasználói programokat. Indíthatja, lelőheti őket.

A rétegek közötti interfészek explicit-, illetve könyvtár függvény hívások.

A THE rétegezettsége csak egy tervezési cél volt: végül is összelinkelték egyetlen betölthető program fájlba.

A THE oktatási célú volt. Hasonló réteges felépítésű, és már nemcsak oktatási célú rendszer volt a MULTICS (Tannenbaum, AT&T, MIT).

A MULTICS-ban a rétegek koncentrikus köröknek képzelhetők. A rétegek közötti interfész explicit-, vagy könyvtár függvény hívások; mindegyike egy *trap* egyben: szigorú ellenőrzéssel, vajon jogos-e a hívás. A programtechnikailag is réteges a MULTICS: mindegyik gyűrűje önállóan betölthető program.

A rétegezési koncepciót a korszerű operációs rendszerek ma már természetszerűleg használják. Egyik nyilvánvaló megvalósítása a Virtuális Fájlrendszer koncepció, vagy a dinamikusan linkelt futásideji könyvtári rutinok (DLLs) koncepciója. Ennél egy-egy szolgáltatási funkciót biztosító rutin nemcsak egyszerűen önállóan betölthető komponens, hanem dinamikus a betöltés: nem feltétlenül a rendszer startup során töltődik a rutin, hanem csak akkor, amikor szükség van rá, azaz dinamikusan.

1.6.3. Virtuális gépek (Virtual Machines)

Tipikus példa erre az operációs rendszer struktúrára is van, ez az IBM VM/370 rendszere (1970).

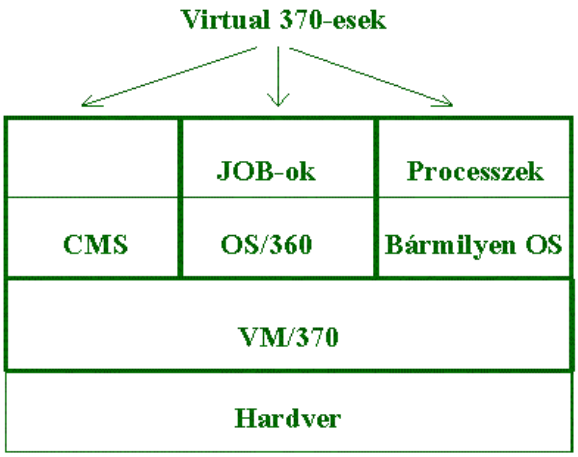
A hatvanas években kibocsájtott OS/360 rendszer kötegelt feldolgozási rendszer volt. A felhasználói igényelték volna az időosztást, de a hivatalosan fejlesztett IBM időosztásos rendszer, a TSS/360 kibocsájtása késett, amikor kész lett, kiderült, hogy túl nagy és lassú. Közben egy fejlesztő csoport, IBM's Scientific Center, Cambridge, MA, egy radikálisan különböző megoldással rukkolt elő, amit az IBM el is fogadott. Ez volt a Virtual 370 koncepció: a VM/370.

Azon az egyszerű elgondoláson alapult, hogy egy időosztásos rendszertől elvárjuk:

1. biztosítsa a multiprogramozást,
2. biztosítson egy kiterjesztett gépet, aminek kényelmesebb a kezelés, mint egy valós gépnek.

A két követelményt a VM/370 rendszer teljesen szétválasztva biztosítja.

A rendszer lelke a Virtual Machine Monitor, ez fut a puszta hardveren, úgy biztosítja a multiprogramozást, hogy több virtuális gépet emulál a felette lévő réteg számára. Az emulált gépek azonban nem egyszerű kiterjesztett gépek, hanem pontos másolatuk valódi gépeknek, beleértve a kernel/user módváltási mechanizmust, az I/O eszközöket, a megszakításrendszert stb., szóval mindent, ami egy valódi gépen van. Minden emulált gép bit szinten azonos az igazi hardverrel, ezért aztán futtatható rajta bármely olyan operációs rendszer, ami az igazi gépen futtaható. Így is volt, a VMM által emulált gépen futtattak OS/360 rendszert, CMS rendszert stb. (1.4. ábra)



1.4. ábra. A VM/370 rendszer

Hogy megértsük: az OS/360 fix, vagy változó memóriapartíciókkal dolgozó, kötegelt rendszerű OS, az IBM 360 gépre fejlesztették JCL (Job Control Language) nyelvvel vezérelhető.

A CMS (Conversational Monitor System) egy egyfelhasználós interaktív rendszer, futhatott 360-as, 370-es gépeken, kezelte ennek memóriáját, háttértárolóit.

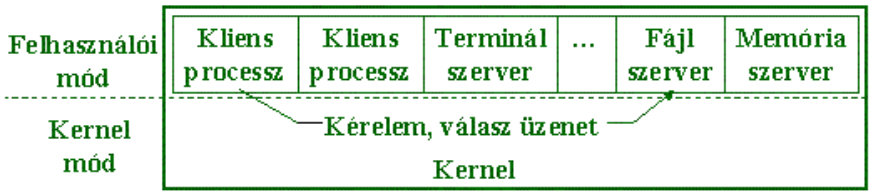
Az eredmény így értékelhetjük: a multiprogramozás megvalósult, de teljesen szeparáltan! A VM/370 nem a szokásos operációs rendszer funkciókat adja, hanem gépeket emulál. Valódi operációs rendszer is kell a VM/370 fölé egy magasabb rétegben.

Virtuális gép szerkezetű a MACH mikrokernele is (lásd később!) A mikrokernel koncepció is egyfajta virtuális gép koncepció: a hardver különbségeket "elrejtő" mikrokernel virtuális gépként szerepel az OS "szokásos" kernele számára.

A virtuális gép koncepció természetszerűleg vezet át kliens-szerver modellhez: mikrokernel, a "virtuális gép" szolgáltatást biztosít a kliensekként szereplő funkcionális elemek számára.

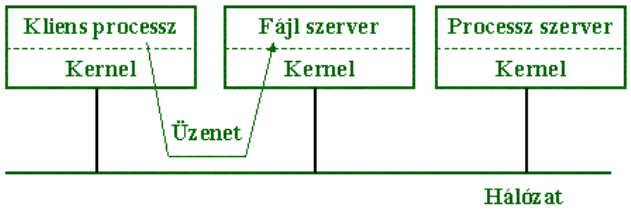
1.6.4. A kliens-szerver modell

Modern operációs rendszerek fejlesztési trendje, hogy minél kisebb legyen a kernel, hogy az operációs rendszer minél több funkcióját tegyük magasabb rétegekbe. Lehetőleg minél több OS funkció kerüljön a felhasználói módú, legfelső rétegbe. A törekvés a kliens-szerver architektúrával közelíthető.



1.5. ábra. A kliens szerver modell

Az elgondolás lényege, hogy a felhasználói processzszek - mint *kliens processzek* - üzenetküldéssel igényeljenek szolgáltatást a *szolgáltató processzektől*. A szolgáltató processzek programjai önállóan linkelhetők, betöltődhetnek a rendszer egész életére, vagy időlegesen (daemon processzek), futhatnak kernel, de akár felhasználói módban is. A szolgáltatók, miután elkészültek a munkájukkal, üzenetküldéssel válaszoljanak. A modell az alábbi ábrán látható.



1.6. ábra. Osztott rendszer

Itt a kernel csak a kommunikációt (és az időosztást) biztosítja.

Tisztán ilyet sohasem valósítottak meg (legalábbis nem tudok róla!), de bizonyos szolgáltatásokat több operációs rendszerben ezzel a struktúrával valósítanak meg, és ebből a gondolatból fejlődött "distributed system" fogalom: hálózati operációs rendszer-szolgáltatások természetes szerkezete ez.

1.7. A VAX/VMS, a Unix és a Windows NT struktúrája

Esettanulmányként nézzük meg három -elterjedt - operációs rendszer struktúráját. Három ábrán három, különböző ábrázolási móddal mutatjuk be a struktúrákat.

1.7.1. A VAX/VMS réteges struktúrája

Jellegzetes a VAX/VMS gyűrűs, réteges felépítése. Tudni kell hozzá, hogy a VAX CPU 4, különböző privilegizáltságú módban futhat, melyből legmagasabb privilegizáltságú, legmagasabb rangú a kernel mód. A futási módok növekvő ranggal az 1.7. ábrán láthatók.

Az áttérés a magasabb rangú futási módra egy-egy *trap* mechanizmussal szabályozottan történhet. Az egyre magasabb rangú futási módokban egyre nagyobb a végrehajtható gépi utasításhalmaz, és egyre szélesebb a címtartomány.

Jel	Futási mód neve	Rang
U	User mode	4
S	Supervisor mode	3
E	Executive mode	2
K	Kernel mode	1

1.7. ábra. A VAX processzor futási módjai

A VAX/VMS MONITOR segédprogram az operációs rendszer teljesítményéről, állapotairól készít statisztikákat. (A MONITOR használatára lásd a VAX/VMS Monitor Utility Reference Manual dokumentumot.) Meglehetősen sok teljesítményosztály mintavé-

telezhető, figyelhető a segédprogrammal, többek között az egyes processzor futási módokban eltöltött idő is. Hívják a MONITOR-t a következő parancssorral:

$ MONITOR MODES

A segédprogram adott gyűjtési és megjelenítési frekvenciával (az alapértelmezések megváltoztathatók a /INTERVAL=seconds. és a /VIEWING_TIME=seconds opciókkal) százalékos megosztásban mutatja meg a processzor által a különböző futási módokban eltöltött időt.. A megjelenített sorok értelmezése:

Interrupt Stack A megszakítási veremben eltöltött idő. Ez valójában kernel módú futás. Főleg a bufferelt I/O kérések kiszolgálását jellemzi.

MP Synchronisation Az az idő, amit a processzorok szinkronizálására fordítanak. Csak többprocesszoros rendszeren van értelme.

Kernel Mode A kernel módban eltöltött idő, leszámítva a megszakítási veremben eltöltött időt, hiszen azt már előbb kimutattuk.

Executive Mode Executive módban eltöltött idő. Jegyezzük meg, hogy executive módban főleg az RMS rutinok futnak, tehát ezt jellemzi ez a sor.

Supervisor Mode A supervisor módban eltöltött idő. A DCL parancsértelmező kódja fut ebben a módban.

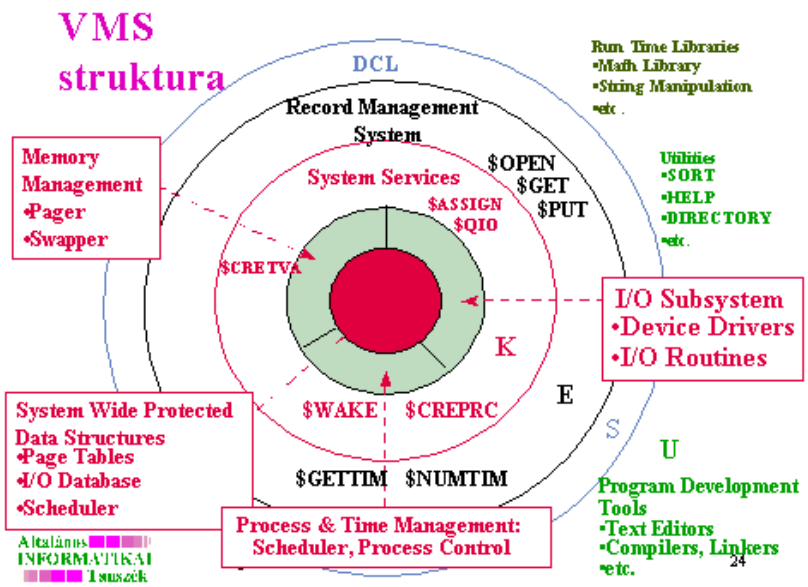
Compatibility Mode A VMS operációs rendszerhez megvásárolható szoftvertermék segítségével VMS alatt futtathatók a DEC cég korábbi operációs rendszerére, az RSX-11-re kifejlesztett programok. A compatiblity mód tulajdonképpen tehát nem processzor futási módra utal, hanem azt mutatja, hogy az ún. Compatiblity Mode Image (RSX-11 futtatható program) instrukcióival mennyi időt tölt el a CPU.

Idle Time A processzor üres ideje.

A VAX/VMS struktúráját az 1.8. ábrán mutatjuk be. Elemezzük az ábrát! Az ábrán fel vannak tüntetve az egyes rétegekbe tartozó komponensek és a réteghez tarozó futási módok. A gyűrűs rétegek komponensei használhatják a belsőbb rétegek szolgáltatásait, futási mód váltó trap instrukciókat tartalmazó hívásokkal.

A U-val jelölt felhasználói módú, legkülsőbb rétegben futnak a programfejlesztő eszközök, a segédprogramok és az alkalmazások (a hozzájuk szerkesztett futásideji könyvtári rutinokkal együtt).

Az S-sel jelölt supervisor módú rétegben a DCL (Digital Command Language Interpreter) kódja fut. Emlékezzünk arra, hogy a VAX/VMS-hez a DCL értelmező szorosan kapcsolódik, s bár használhatnánk más burok-programot is (ami U módban futhat), nem hagyható ki a DCL, nem hagyható el ez a réteg.



1.8. ábra. A VAX/VMS struktúrája

Executive módban futnak az *RMS* (Record Management Services) *rutinjai*. Ezek segítik az alkalmazásokat a fájlok és a fájlok tartalmának kezelésében. Bár kezelik a karakterorientált eszközöket is, az előnyük főleg a tömegtárolókra szervezett fájlok kezelésében érvényesíthető. Változatos diszk fájl szervezési módokat, rekord formátumokat és rekord elérési módokat biztosít: támogatja a soros, relatív és indexelt szervezést, a fix és változó hosszú rekordformátumokat, és lehetővé teszi bármelyik szervezési módnál a soros elérést, a megfelelő szervezéseknél a direkt elérést (kulcs szerinti elérést az indexelt szervezésnél, relatív rekord szám szerinti elérést a relatív szervezésnél. Az RMS interpretálja a rekordstruktúrát, míg a fájlok tömegtárolókon való elhelyezkedéséért az adatmenedzsment egy másik szintjéhez tartozó komponensek felelnek: a szolgáltató processzek (ACP: Ancillary Control Processes), vagy az XQP (Extended QIO Processor) processzor. Az RMS rutinok hívásai mind a futási idejű könyvtárakban található szokásos I/O függvényekkel, eljárásokkal történhet (ekkor a trap-et a könyvtári rutin kiváltja), mind pedig közvetlen RMS rutin hívással (lásd a VMS Record Management Services Manual dokumentumot!).

A K jelű réteg az interfész a kernelhez (System Services), a réteg rutinjai és a rétegen belüli operációs rendszer komponensek kernel módú futást igényelnek. E réteg "szolgáltatása" csak annyi, hogy interakciókat valósít meg a lényegi kernel komponenseivel (kivéve a tulajdonképpen a kernelhez tartozó I/O alrendszernek az eszközfüggetlen szolgáltatásait, mert azok tényleg szolgáltatnak is).

A lényegi kernelben az ábra szerint négy, valójában öt elem található.

Legbelül láthatjuk az **operációs rendszer adattábláit** (Systemwide Protected Data Structures). Ezek természetesen nem szolgáltató rutinok. Ezeket a további komponensek közösen használják, ezek egy bizonyos interfészt már biztosítanak közöttük.

Az I/O alrendszer (I/O Subsystem) tartalmazza az eszközmeghajtó szoftvereket (Device Drivers) és a hozzájuk tartozó adatstruktúrákat. Tulajdonképpen az I/O alrendszerhez tartoznak, de az E rétegben vannak az eszköz-független rutinok (legfontosabb közülük a $QIO rutin).

A memória menedzselő alrendszer (Memory Management) legfontosabb része a *laphiba kezelő* (Pager, vagy Page Fault Handler), ami a VMS virtuális memória-rendszer támogatását valósítja meg. Másik fontos eleme a *ki/be söprő* (Swapper), ami a fizikai memória még jobb kihasználását biztosítja. A használt adatstruktúrák: a Laptáblák (Page Map Tables) és a lapkeret adatbázis (Page Frame Number Database). Az alrendszerhez az interfészt jelentenek az

executive réteg szolgáltató rutinjai, amik lehetővé teszik egy-egy processz virtuális címtartományának növelését, csökkentését, illetve a címtartomány egy részének leképzését egy fájlba.

A folyamatkezelő és időzítő alrendszer (Process and Time Management) egyik eleme (Scheduler) választja ki futásra a "legjobb" processzt, mialatt elveszi a CPU-t az éppen futó processztől. Ugyanez kezeli az időszolgáltatásokat, az időzítési rendszer szolgáltatásait. A processz vezérlő része (Process Control) segíti a processzek kreálását, a processzek szinkronizációját, bizonyos processzek közti kommunikációkat. Ez az alrendszer felelő elsősorban az időzítési adattáblákért.

Az 1.8. ábrán nem látható a kernel további alrendszere, a **vegyes szolgáltatások** (Miscellaneous Services) rutinjainak csoportja. Nem teljes a felsorolás, de ide tartozik a *logikai név kezelés*, a *szöveglánc feldolgozó szolgáltatások*, amiket az alkalmazások igényelhetnek, és ide tartoznak olyan szolgáltatások is, pl. bizonyos *szinkronizációs technikák*, a *rendszer pool manipulálás*, amiket a rendszer egyéb rutinjai kérhetnek. Mind az alkalmazások és segédprogramok, mind a rendszer rutinok használják a minden "objektumra" kiterjedő *lock management* védelmi mechanizmust.

Láttuk tehát a VAX/VMS kernel és környezete struktúráját, és kérdés merülhet fel, hogy is vannak a komponensek programtechnikailag megvalósítva?

Gyakorlatilag háromféle módon.

Egy részük *eljárás/függvény jellegű kód*, ami szinkron *call* hívással, paraméterátadással aktiválható. Rétegek átlépésekor természetesen megvalósított az ellenőrzött *trap* is. A hívás lehet beágyazva futás idejű könyvtári rutinokba, vagy lehet közvetlen *rendszerhívás* (system call), rutinhívás (RMS call). A visszatérések is a szokásos eljárásból/függvényből való visszatérési technikák ez esetben.

Egy másik részük *kivételes eseményt vagy megszakítást kiszolgáló rutin*. A megszakítások aszinkron jellegűek, bizonyos kivételes események szintén aszinkron jellegűek, mások szinkron jellegűek, bár váratlanok, a futó processzek szempontjából. Bármilyen is az esemény jellege, a kiszolgáló rutinba való belépés más, mint a *call*-lal való belépés. A futó processzhez tartozó regiszterállapotok lementése után az esemény elemzéstől függően kerül a végrehajtás fonala a kiszolgáló rutinhoz.

A szolgáltatások harmadik csoportja *önálló processzként megvalósított*. Ezek a *rendszer processzeknek* is nevezett folyamatok privilegizáltak, ami azt jelenti, hogy részben vagy egészben magas rangú módban futnak. Ezek egy része végigéli a rendszer életét, egy részük pedig szükség esetén megszületik, elvégzi feladatát és meghal. Az alkalmazások a rendszer processzek egy részének szolgáltatásait közvetlenül nem kérik: a rendszer processzek az alkalmazások számára transzparensen, a "háttérben", igénybejelentés nélkül is végzik a szolgáltatásokat. Más részüktől az alkalmazások igényelhetnek szolgáltatást, ekkor ez *processzek közti kommunikációs mechanizmusokkal* (Inter Process Communication) történik.

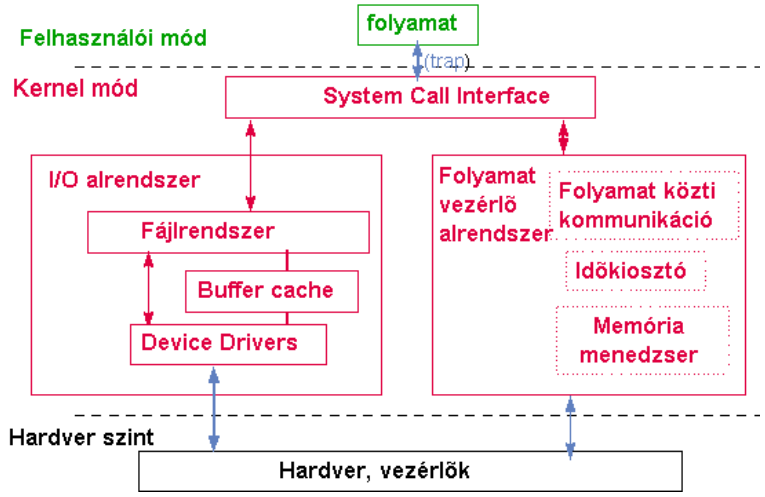
Szeparált processzként megvalósított például a *swapper* (a memória menedzselő alrendszer része), vagy a szintén a memóriamenedzsmenthez tartozó *módosított lapok készletét kiíró*, de eddig nem említett processz. Mindkettő teljes egészében kernel futási módú, igénybejelentést nélkül is szolgáltat.

Szeparált processzekként megvalósítottak az *ACP*-k (Ancillary Control Process), amiket az I/O alrendszernél már említettünk. Ezek az adatmenedzsment egyik szintjét segítő

processzek, szintén privilegizáltak. Mint említettük, a fájlok eszközökön való elhelyezéséért felelősek. Lehetnek diszkeket, mágnesszalagokat kezelő ACP-k (hacsak nincsenek ezekhez speciális processzorok), de vannak hálózati eszközöket, terminálvonalakat kezelő ACP-k is. Az ACP-k sajátos interfésszel rendelkeznek a kernelhez, hiszen szorosan együtt kell dolgozniuk. A kernel a működésüket az RMS rutinokkal is összehangolja (emlékezzünk, az RMS rutinok felelősségére a rekordformátumokért, vagyis a fájlok tartalmáért). Az alkalmazások igénybejelentésére szolgáltatnak.

1.7.2. A Unix kernel funkcionális felépítése

A Unix kernel implementációjához legalább két futási mód kell: a felhasználói mód és a kernel mód. Az ábrán, ami más ábrázolási technikával készült, mint az előző, a futási szinteket (User level, Kernel level) tüntettünk fel, amik értelemszerűen megfelelnek a futási módoknak. Természetesen nincs akadálya annak, hogy több futási móddal rendelkező CPU-ra Unixot valósítsunk meg.



1.9. ábra. A Unix kernel funkcionális felépítése

Nézzük most az 1.9. ábrát! Ez a két futási módú hardverre implementálható Unix kernel legalapvetőbb struktúrája. Láthatunk hasonlóságokat és különbözőségeket a VMS kernelhez viszonyítva: megfigyelhető az itt is önálló *I/O alrendszer*, látható, hogy a *memória menedzsment*, a *scheduler* és a *folyamatközi kommunikáció* szolgáltatás a *folyamatvezérlő alrendszer* (Process Control Subsystem) részeként feltüntetett. Az alrendszerek funkcióit itt nem részletezzük, ezek hasonlóak a VMS funkciókkal, később még lesz róluk szó. Az ábrán néhol látszik a réteges felépítés (lásd az I/O alrendszert). Az viszont nem látszik az ábrán, hogy vannak kernel szintű adatbázisok, adattáblák is.

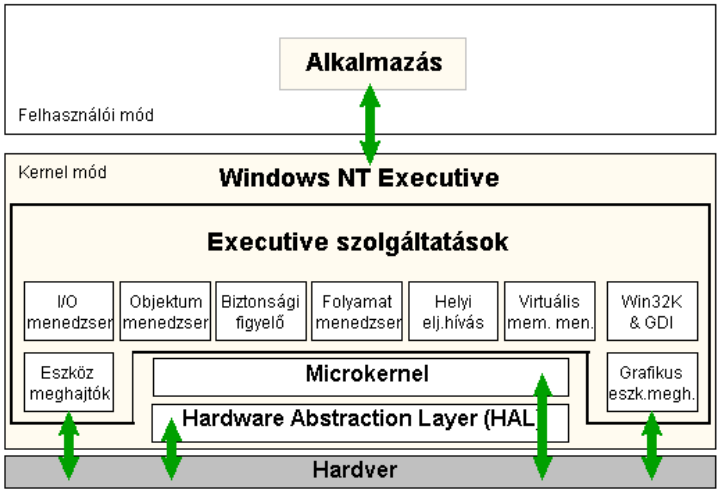
A VMS-hez hasonlóan, a kernel itt is szolgáltatásokat biztosít a felhasználói folyamatok számára.

Gyakorló feladatként elemezzünk egy *read(fp, buffer, size)* rendszerhívási forgatókönyvet.

Bizonyos szolgáltatások itt is *eljárás jellegű rutinok* (call jelleggel hívhatók), itt is vannak *eseménykezelő rutinok* (esemény bekövetkezésre, akár aszinkron jelleggel hívódnak), és itt is vannak önálló folyamatként megvalósított szolgáltatások (pl. a *swapper* és a *pagedaemon*). Bármelyen is az implementáció, a felhasználói folyamatból a szolgáltatás ún. *rendszerhívással* (system call) igényelhető.

1.7.3. A Windows NT 4.0 struktúrája.

Az 1.10. ábrán jól látható az NT moduláris felépítése. Jól észrevehető a mikrokernel architektúra.



1.10. ábra. A Windows NT 4.0 kernel szerkezete

A *HAL* modul tulajdonképpen csatoló a hardver és a mikrokernel között, célja a hardver különbözőségeket (processzorok architektúrája, processzorok száma, elrendezése stb.) elrejteni a magasabb rétegek elől. Bár az operációs rendszer része, szokás szerint a hardvergyártók készítik és szállítják a géppel együtt. A HAL-nak köszönhető, hogy az NT sok platformon (pl. Intel, DEC Alpha stb.) használható.

A *mikrokernel* látja el az alapvető rendszerfunkciókat: a megszakításkezelést, a fonalak (szálak) ütemezését, szinkronizálásokat.

Az Executive szolgáltatások moduljait használhatják az alkalmazások (és az ún. környezeti rendszerek.

Az *Objektum menedzser* egységes szabályrendszer segítségével vezérli az objektumok létrehozását, elnevezését, biztonsági tényezőit.

A *Folyamat menedzser* hozza létre és törli a taszkokat, a szálakat (fonalakat), szorosan együttműködve a memória menedzserrel és a biztonsági rendszerrel.

A *Helyi eljáráshívás alrendszer* (hasonlít az RPC-hez) kezeli az alkalmazások hívásait, melyekkel a környezeti alrendszeren (vagy kiszolgáló alrendszer, nem látszik az ábrán!) át szolgáltatásokat igényel.

Az *I/O alrendszer* biztosítja a fájlrendszereket (FAT, NTFS, CDFS), a cache buffer (átmenti gyorsító tároló a központi memória és a háttértárak között) funkciókat, az eszköz driver-eket.

A *Virtuális memóriamenedzser* értelemszerűen a memória-gazdálkodást segíti.

A *megjelenítő rendszerből* a konzol rendszer nem a kernel része (az felhasználói módban fut). Kernel komponens viszont a *Win32K* ablakmenedzser: kezeli az ablakokat, a képernyőt, eljuttatja az inputokat az alkalmazásokhoz. A *GDI* (Graphics Device Interface) grafikus eszköz csatoló pedig képernyő rajzoló primitívek gyűjteménye. Végül, a megjelenítő alrendszerhez tartoznak a grafikus eszközmeghajtók (driver) is.

1.8. Hogyan juthat a vezérlés a kernelbe?

Tulajdonképpen háromféle módon:

1. A felhasználói processzekből *rendszerhívással* (system call). Valójában ez egy futásideji könyvtári függvény hívása, aminek a paraméterei a felhasználói címtartományban vannak. A hívó folyamatra nézve szinkron. Implicite *trap* (futási módváltás) van benne.

2. *Megszakítás* (IT: interrupt) *generálásával* a hardverből. Aszinkron, és ritkán kapcsolatos az éppen futó processzel, a processznek "nincs is tudatában", hogy mi okozza a problémát.

3. *Kivételes esemény* (Exeption Condition), hiba (error) *előállása* esetén a hardverből. Szintén váratlan, de általában az éppen futó processzel kapcsolatos. Szinkron olyan értelemben, hogy a processznek "tudatában van" a felmerült probléma. Bizonyos irodalom ezt a belépési módot egyszerűen *trap*-nek nevezi. (Más szerzők a három belépési módot közös *trap* névvel illetik, ismét mások a *trap* kifejezést a rendszerhívásbeli futási mód váltásra értik.)

1.8.1. A kernelbe való belépés eseményei

Miután a klasszikus Unix-ok a legegyszerűbb operációs rendszerek, a kernelbe való belépés és kilépés forgatókönyvet Unix-os példán mutatjuk be. Nyilvánvaló, hogy más operációs rendszereknél hasonló forgatókönyvek szerint történik a be/kilépés. Ezek után a kernelbe lépés "története":

- A hardver átkapcsol kernel módba. A memória-elérés kernel privilégiummal történik ezután, a verem mutató átáll a kernel szintű veremre, minden privilegizált instrukció végrehajtása engedélyezett.
- A PC és a PSW (programszámláló és program státus szó regiszterek) a processz kernel szintű veremére töltődnek (push). Ez hardveres letöltés.
- A rendszerhívás/trap kódja (system call száma/signal kódja) is rátöltődik a veremre.
- Egy assembly rutin lementi az általános célú regisztereket is. Ezek után már magas szintű nyelven írt rutinok is hívhatók. Így is történik, C-ben írt rutin hívódik, a belépés fajtájától függően.
- Hívódik
  - syscall() rendszerhíváshoz. Ez egy diszpécser, elosztja a vezérlést.
  - trap() a kivételes esemény belépés esetén, ami szintén eloszt, a kódtól függően.
  - a megfelelő *device driver* IT kiszolgálója, megszakítás belépés esetén.

A diszpécser feladatait tovább részletezhetjük:

- Kiveszi a rendszerhívás paramétereinek számát.
- Ellenőrzi a paramétereket, vajon a felhasználói címtartományban vannak -e, majd bemásolja azokat a kernel címtartományába. Ez azért fontos, hogy a kiszolgálás mellékhatása (side effect) semmiképp ne rontsa el a processz felhasználói területét.
- Felkészül arra, hogy interrupt-tal, trap-pel megszakíthatják.
- Meghívja a megfelelő rutint.

1.8.2. Visszatérés a kernelből

A visszatérés a megfelelő szolgáltatásból függ a belépéstől.

A klasszikus rendszerhívás szolgáltatásból először a diszpécserhez tér vissza a vezérlés, méghozzá azzal a jelzéssel, hogy a szolgáltatás sikeres volt, vagy nem. Mielőtt a diszpécser a hívójának adná az eredményt, megvizsgálódik, kapott -e közben *szignált* a folyamat. Ha igen, a *signal handler* működik, de végül a vezérlés mindenképp visszatér a diszpécserhez. Ekkor az esetleges hibás szolgáltatás hibakódját a globális *errno* változóba írja, majd egy assembly rutin visszaveszi az általános regiszterek tartalmát (pop). A visszatérési értéket hordozó regiszter a szolgáltatási hiba esetén -1 értéket kap, különben 0-t, vagyis a hívó a system call visszatérési értékeként csak jelzést kap, volt-e hiba vagy sem, a hiba jellegére az errno vizsgálatával következtethet). Ezután végrehajtanak egy *return-from-interrupt* instrukciót. Ez visszaállítja a PC és PSW tartalmat, és visszaállítja a felhasználói módot. Ugyanezzel vissza-

áll az SP is, és a processz user szintű vermére mutat. Ezzel folytatódik a processz felhasználói módú futása.

Kérdés merülhet fel, jó-e ez így? Mi van, ha IT kiszolgálás rutinja fut (kernel módban) és erre jön egy magasabb szintű megszakítás, hogy történik ekkor a "belépés" és a "visszatérés"?

A válasz: kernel módú futásban viszonylag kevés az "elővételi jog", a *preemption*. Ilyen kérés esetén

- belépéskor nincs módváltás,
- kilépéskor sincs módváltás.

Nincs tehát verem-mutató váltás sem, ugyanazt a kernel szintű vermet használják. A többi esemény hasonló a fentiekhez.

Kérdés merülhet fel, melyek a leggyakoribb trap-ek? (Most trap alatt mindhárom belépésre gondolunk.)

- Leggyakoribb az óraeszköz megszakítása. Ez állítja a napi időt számontartó mezőket, támogatja a időkiosztás vezérlését, a rendszer időtúllépés (timeout) funkciók végrehajtását.
- Második leggyakoribb trap a szokásos rendszerhívások trap-je.
- Ezután jönnek a további trap-ek.

1.9. A rendszerhívások osztályai

Processz menedzsment

- end, abort
- load, execute
- create, terminate processes
- get,set process atributes
- wait for time/termination
- wait for event, signal event
- allocate free memories

```
fork()  exec?() exit()  sigaction()     [signal()]
kill()  residual()      pause()
```

Fájlokkal, jegyzékekkel kapcsolatos rendszerhívások

- create, delete files
- open, close files
- read, write, reposition
- get,set file attributes
- ugyanezek jegyzékekre is

```
creat() open()  close() read()  write()
lseek() stat()  mkdir() rmdir() link()
ulink() chdir() chmod()
```

Eszköz manipulációk

- request device, release device
- read, write, reposition

- get, set device attributes
- logically attach, detach devices

Informálódó, beállító

- get, set time, date etc.

1.10. Irodalom

1. Tannenbaum: Modern Operating Systems, Prentice Hall, 1992
2. Kóczy, Kondorossi szerk.: Operációs rendszerek mérnöki megközelítésben, Panem, 2000
3. Vahalia: UNIX Internals, Prentice Hall, 1996
4. Bach: The Unix System, Prentice Hall, 1985
5. Silberschatz, Galvin: Operating Systems Concepts, Addison-Wesley, 1994
6. Babócsy, Füzessy: Windows NT 4.0 hálózatok, NeTen, 1998

2. A folyamat (process, task) koncepció

Az eddigi elképzelésünk szerint egy Neumann elvű gép úgy működik, hogy a processzor veszi a soron következő gépi instrukciót és azt végrehajtja. A programszámláló regiszter tartja nyilván, melyik a soron következő gépi instrukció. Egy-egy instrukció végrehajtása után a programszámláló regiszter „automatikusan” felveszi a soron következő instrukció címét, ezzel a processzor „tudja”, hogy melyik a soron következő instrukció.

Tudjuk, hogy a memóriában tároltak az instrukciók: van egy elképzelésünk a tárolt instrukciók sorozatáról, és tudjuk, hogy a CPU által végrehajtott instrukciófolyam is egy instrukciósorozat. A processzor által végrehajtott instrukciósorozat nem mindig egyezik a tárolási sorozattal, hiszen ugró instrukciók is lehetségesek. A klasszikus Neumann gép által végrehajtott instrukciófolyamot eddig struktúrálatlannak láttuk.

Valójában a végrehajtott instrukciófolyamnak van struktúráltsága! A hardver már ad egyfajta struktúrát: a CPU által végrehajtott instrukciófolyam egy része *felhasználói módban* fut, más részei pedig *kernel módban*. Felhasználói és kernel módú instrukciófolyam szakaszok vannak, köztük az a bizonyos *futási mód váltás* (mode switch, trap) és a *visszaállítás* jelenség. A futási mód váltást egy speciális instrukciósorozat végrehajtása valósítja meg ,ami csakis az operációs rendszer magja által ellenőrzötten történhet. A futási mód koncepció egy alapvető védelmi mechanizmus.

A processzor által végrehajtott instrukciófolyamot más módon is struktúrálhatjuk, "szakaszolhatjuk". Egyfajta "logikai" szakaszolás a következő: egy adott program futásához tartozó instrukciófolyam - kezdetétől a végéig - egy szakasz lehet. Egy-egy ilyen szakaszt nevezhetünk munkának (job), folyamatnak (processz). Ez a szakaszolás, ez a struktúrálás nem következik a hardverből! A CPU-nak édesmindegy, hogy melyik program kódját futtatja éppen! Ezt a szakaszolást az operációs rendszer végzi, tartja nyilván, menedzseli. Az operációs rendszernek fontos a processz fogalom.

2.1. A folyamat fogalom

Szinte minden operációs rendszer kulcsfontosságú fogalma a *folyamat* fogalom. A folyamat kifejezést a köznyelv és más szakterületek is használják. A folyamaton általában tevékenységek sorozatának végrehajtását értjük, ahol a tevékenységeknek van kezdete és vége.

Az operációs rendszerek tárgyalása során is használjuk a folyamat (angolul process) kifejezést. Elfogadhatjuk a magyaros kiejtésű *processz* nevet is. Kapcsolatos kifejezések lesznek a munka (job), a feladat (task), vagy magyaros kiejtéssel a taszk, a fonál, vagy szál (thread) kifejezések is.

Most fogadjuk el a következő definíciót a folyamat fogalomra: a *processz* egy végrehajtási példánya egy párhuzamosságot nem tartalmazó végrehajtható programnak. A processz egy futó program-példány.

Láthatóan megkülönböztetjük a *processz* fogalmat a *végrehajtható program* (*executable program/image, loadable program/image*) fogalomtól! A processz más entitás, mint a program!

Egy program lehet szövegszerkesztővel készült *forrás program*. Lefordítva a forrás programot *tárgyprogram*ot kapunk. Tárgyprogramokat futásra kész állapotba hozhatunk összeszerkesztő (linker, task builder) eszközök segítségével, az eredmény a *futásra kész*, *végrehajtha-*

tó, *betölthető program*: eddig a program kifejezéshez mindig kapcsolódhatott a *fájl* fogalom. A programoknak *helyfoglalásuk van*, valamilyen tömegtárolón fájlok formájában tároljuk őket, *hosszméretük* van, *statikus*ak.

Egy processz születik. Születése során, vagy születése után egy végrehajtható program betöltődik a memóriába és fut. A processz vetélkedik az erőforrásokért, "él", időt használ, viselkedése *dinamikus*. Végül a processz megszűnik (exitál, terminálódik).

Pongyola, de nagyon szemléletes fogalmazás a következő: a processzben fut a program. Érezzük, hogy a processzhez hozzátartozik a végrehajtható program, hozzátartoznak annak kód és adat szegmensei, de további információk is tartoznak a processzekhez, melyek a „program állapotban" még nem létezhettek. A processznek van például veremtára (a függvényei, eljárásai paramétereinek cseréjéhez). A folyamathoz hozzátartoznak a CPU regiszterek pillanatnyi értékei is. Hozzátartoznak még bizonyos adminisztrációs információk is: vagyis mindazon információk, melyek szükségesek, hogy a processz futni tudjon.

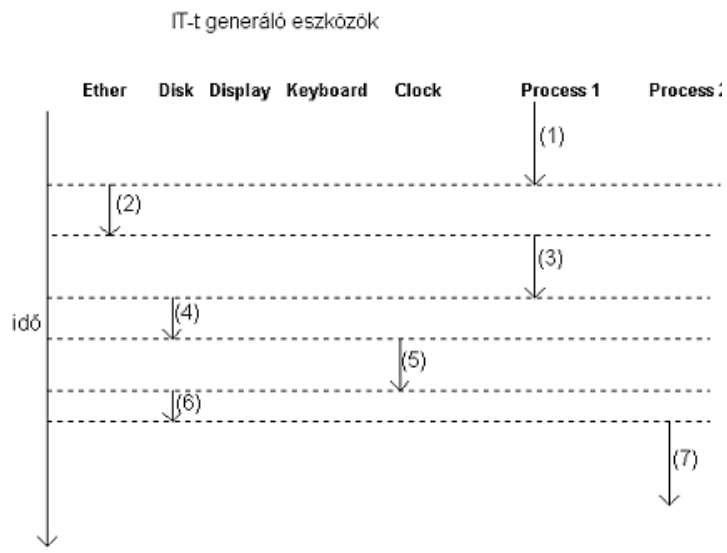
Egy *multiprogramozású rendszerben* (multitasking, multiprocessing) egy időben több folyamat él. Folyamatok születnek, vetélkednek, együttműködnek, kommunikálnak, végül megszűnnek.

A mostani – klasszikus - folyamat fogalom használatánál kikötjük még, hogy a folyamatok szekvenciálisak. Ez azt jelenti nekünk, hogy egy kijelölt processzoron futnak (sequential process: one execution on a dedicated CPU), egy processzhez egy dedikált programszámláló regiszter (PC, Program Counter) tartozik, ami egyetlen helyre tud mutatni a program szövegben. A klasszikus processz egy *szálon* fut, a „benne futó" végrehajtható program nem tartalmaz párhuzamosságokat. Egy konkurrens nyelven írt program futás közben több végrehajtási fonállal rendelkezhet. Párhuzamos programozási környezetben ezért a konkurrens program futásközben neve sokszor a taszk név, a végrehajtási menet neve a fonál, vagy szál (thread), vagy éppen a „könnyűsúlyú" processz.

A klasszikus elképzelésben a független processzekhez független processzorok, ezzel független programszámláló regiszterek tartoznak. Egy-egy processz vezérlési menete közvetlenül nem befolyásolja a többi menetét. Közvetetten persze a processzek az erőforrásokért való vetélkedéssel, a processzek közötti kommunikációs mechanizmusok segítségével befolyásolhatják egymást!

Az egyik legfontosabb erőforrás, amiért a processzek vetélkednek, éppen a processzor lehet! Multiprogramozott rendszerben több processz létezhet egyidőben mint ahány processzor van: éppen az operációs rendszer feladata, hogy biztosítson a processzek számára „saját logikai processzort", ütemezze számukra a valódi processzor idejét, „kapcsolja" számukra a valódi processzort, gondoskodjon az elválasztott logikai processzorok" állapotának megtartásáról.

A klasszikus processz a dedikált logikai processzoron szekvenciálisan fut. A processz szemszögéből tekintett szekvencialitás a valóságban nem az. Nézzünk erre egy egyszerű, egy processzoros rendszerbeli példát (2.1. ábra):



2.1. ábra. A processzek a valóságban nem szekvenciálisan futnak

Az ábrán megszakítás (IT: interrupt) generálásra képes eszközök kezelői (handlers) és két processz vetélkednek a CPU-ért. Kiinduláskor (1) a Process 1 fut (itt a fut azt jelenti, övé a valódi CPU). Az Ether hálózati kártya megszakítja a futását, „elveszi" a Process 1-től a CPU-t (2). A hálózati megszakítás kiszolgálása után a Process 1 visszakapja a CPU-t, a (3) jelzés vonalán halad tovább a végrehajtás fonala, a processz szemszögéből nézve futása szekvenciálisan folytatódik. A Process 1 „elképzelése szerint" az (1)-es szekvenciális folytatása a (3)-as. Idővel újabb megszakítás következik: a Disk eszköztől. Kiszolgálásához a CPU a (4)-es kódrészletet hajtja végre. Az ábrán a (4) jelű diszk megszakítást kiszolgáló rutint a magasabb prioritású IT megszakítja, annak kiszolgálása után a (6)-on folytatódik a diszk megszakítás kiszolgálása, majd a (7) fonalon a Process 2 futása.

Miután a klasszikus processz koncepció szerint minden processznek van dedikált, saját processzora (ezt az illúziót biztosítja számára a processzor ütemezés), e processzor futási módja nevezhető a *processz futási módjának* is. A futási mód koncepció a processzorhoz tartozik valójában, de ezzel az áttétellel processzek futási módjáról is szoktunk beszélni. Nyilvánvaló, hogy a processzek a „saját kódjukat" felhasználói módban hajtják végre, a rendszerhívások kiszolgáló kódok, az események kezelőinek (handlerek) kódjai viszont kernel módban futnak.

2.1.1. Folyamat tartalom (process context) fogalom

Nehéz lefordítani a *process context* kifejezést. Folyamat környezet fordítás nem jó, mert ezt a *process environment* fogalomra használtuk (azaz az *sh* burok környezeti változóinak definícióit tartalmazó szövegsorokból álló információkra). A folyamat tartalom, esetleg folyamat szövegkörnyezet használható kifejezések. Míg nem találunk jobbat, használjuk a magyarosan is kiejthető, írható *processz kontextus* kifejezést. (Megjegyezzük még, hogy egyes szerzők a process environment kifejezés alatt is a processz kontextust értik, azaz nem szűkítik le az environment-et környezeti változók definíciós tábláira.)

A folyamat kontextus definíciója: adatstruktúrákba rendezve minden olyan információ, ami a folyamat futásához szükséges.

Más fogalmazásban: minden olyan információ, ami a rendszer számára szükséges, hogy a CPU-t a folyamatok között kapcsolja, a folyamatok szekvencialitásának illúzióját biztosítva.

Intuitíve érezzük is, melyek ezek az információk:

- a program (image) kódszegmense(i), (szekciói);
- a program adatszekciói;

- a processz veremtára(i) (stack, heap), az argumentumátadáshoz, bizonyos változók futásideji helyfoglalásához stb.;
- a folyamat menedzselési információk (néha nevezik attribútumoknak):
  - a kód aktuális instrukciója, ami éppen fut, vagy következik (PC);
  - azonosítási információk (pid, ppid, pname stb.);
  - tulajdonossági, családtagi információk (uid, gid stb.);
  - állapot információk (memória menedzseléshez, időkiosztáshoz stb.);
  - prioritások, limitek, quóták (memória menedzseléshez, időkiosztáshoz stb.)
  - I/O menedzsment információk;
  - mutatók a szekciókhoz stb.

Ezek az információk azonosítják a processz által használt erőforrásokat és azt az instrukciót, amelynél a processz éppen fut.

A *processz kontextus* többféle módon szemlélhető:

Egyik szemléletmódban beszélhetünk

- *hardver kontextus*ról (a regiszterek pillanatnyi értékeiről, mint menedzselési információkról) és
- *szoftver kontextus*ról (kódszegmensek, adatszegmensek, egyéb menedzselési információk stb.).

Egy másik szemléletmódban ugyanaz a kontextus lehet

- felhasználói szintű kontextus (user level context), ami a felhasználói címtartományban (user address space) van, vagy
- rendszer szintű kontextus (system level context). Ennek lehet
  - statikus része (static part) (főleg a menedzselési információk tartoznak ide), és
  - lebegő (dinamikus) része (dinamic or volatile part), amihez a regiszter állapotok tartoznak (register context). Dinamikusnak, lebegőnek (volatile) nevezzük ezt a részt, mert a regiszterek értékeit időnként maguk a regiszterek hordozzák, időnként azonban le vannak mentve valamilyen verembe. Kérdéses, hogy a lebegő kontextust rendszer szintűnek tekintsük-e, vagy sem! Bizonyos regisztereket felhasználói módban használhatunk, ezekre a felhasználói módban futó instrukciók használhatják, de vannak regiszterek, amiket felhasználói módban nem programozhatunk, illetve vannak, melyeket a CPU ugyan használ, de egyáltalán nem programozhatók.

Érdemes „körüljárni” a címtartományok és a kontextus fogalmakat!

A mai számítógéprendszerek rendszerint virtuális memóriamenedzselésűek. A későbbiekben a virtuális memóriát részletesebben tárgyaljuk, most azonban előlegezzünk meg a koncepcióból annyit, hogy akár a processzek, akár a kernel szolgáltatások által használt memória címek ún. virtuális címek. Egy processz egy instrukciója nem hivatkozik közvetlenül fizikai memória címre. Egy processznek az az „elképzelése”, hogy van egy meglehetősen nagy virtuális címtartománya (vagy akár több címtartomány szakasza) és e címtartomány elemekhez a hardver MMU (Memory Management Unit) és az OS kernel együtt biztosít memória rekeszeket is. Az MMU és az OS közösen, a processz számára „észrevehetetlenül” (transzparensen) „leképzi” a virtuális címeket fizikai címekre, és biztosítja, hogy a címzett fizikai rekeszben ott is legyen az érvényes (valid) információ (instrukció vagy adat). Az MMU bizonyos regiszter(ek)ben tárolja az éppen futó processz (current process) „leképzési táblá(i)nak” kezdő címé(i)t. Amikor az éppen futó processz „elveszti” a processzort és egy másik processz „megkapja” azt (ezt nevezzük processz kontextus kapcsolásnak, Process Context Switch), a

„nyertes” processz lebegő (volatile) kontextusának a regiszterekbe való betöltésével Az MMU regiszterek a nyertes címleképző tábláinak címeit tartalmazza.

Beláthatjuk, hogy egy processz címtartományának egy részét a felhasználói szintű kontextusát tartalmazó cellák címzésére használja (címtartományának egy része felhasználói szintű címtartomány), de egy másik része a kernel kódjait és adatait tartalmazó rekeszeket címez. Ezt a részt nevezik kernel (rendszer) címtartománynak. Beláthatjuk, hogy ez utóbbi tartományhoz tartozó címre csakis kernel módban hivatkozhatunk. Viszont csak egyetlen kernel „fut” a rendszerben, vagyis minden processz a saját kernel címtartományát ugyanarra az egyetlen kernel címtartományra képzi le. A kernel nyilvántart globális adatstruktúrákat és processzenkénti adatstruktúrákat (per process data srtuctures) is. A kernel közvetlenül elérheti az éppen futó processz címtartományához tartozó címeket (miután az MMU regiszterek a processz „leképzési tábláit” címzik), szükség esetén közvetve más processzek címtartományait is kezelheti.

Azt beláthatjuk, hogy egy processz címtartománya kiterjed a felhasználói címtartományra (e mögött vannak a kódszegmensei, az adatai, a vermei stb., azaz a tartalom szerinti kontextusának egy része) és a kernel címtartományra (e mögött vannak adminisztrációs információi legalább, mint a kontextusának részei). Vita merülhet fel viszont azon a kérdésen, hogy vajon a kernel szolgáltatások kódjai és adatai hozzátartoznak-e processz kontextushoz? Ráadásul a rendszerhívással igényelt szolgáltatások és a kivételes események kezelői a kurrens processz javára „dolgoznak”, míg a megszakítás kezelők nagy valószínűséggel nem köthetők egyetlen processzhez sem. Azt is tudomásul kell vennünk, hogy bizonyos kernel feladatokat kliens-szerver stílusban, önálló szolgáltató processzek teljesítenek. Ezeknek saját kontextusuk van, abban futnak. Rendszerint tisztán kernel módú a futásuk. A legtöbb kernel szolgáltatást azonban kernel rutinok biztosítanak (még az önálló processzben megvalósított szolgáltatáshoz is kell processz közti kommunikációs rendszerhívás, kell hozzá kernel rutin). Nos, többféle szemlélet létezhet.

Egy egyszerűbb, a független processz modellből levezethető tárgyalásmódban a kernel rutinok – akár rendszerhívás szolgáltatások, akár esemény kezelők – nem részei egyetlen processz kontextusának sem. E szerint a fogalmazás szerint ezek rendszerint egy-egy *processz kontextusa fölött* futnak (természetesen kernel módban). (A „rendszerint” kitétel azért szükséges, mert elképzelhető, hogy minden processz „blokkolt”, nincs éppen futó processz, aminek a kontextusa fölött futhatna kernel kód; ámbár, a korszerű operációs rendszerek ilyenkor egy „idle”, tevékenységet nem folytató processzt futtatnak.) Tudomásul veszszük, hogy ezzel a tárgyalásmóddal együtt jár az, hogy egyes szolgáltatások a futó processz javára, más szolgáltatások nem a javára „dolgoznak” a processz kontextus felett. A független processz modellben a kernel *dedikált virtuális gépet* biztosít minden processznek. A rendszerhívások ebben a tárgyalásmódban virtuális instrukciók, a kezelők ennek a virtuális gépnek eseménykezelői. A független processz modellben processz állapotokról (lásd később), kontextus kapcsolásról sem beszélünk (ez transzparens a modellben), de a tárgyalásmód megengedi a modell meghaladását: a processz futási módok, a processz kapcsolás fogalom tárgyalását.

Egy másik tárgyalásmód szerint a rendszer-hívás szolgáltatások és a kivételkezelők hozzátartoznak a processz kontextushoz (annak ellenére, hogy ezeket nem a felhasználó programozta), míg a megszakítás kezelők (melyek szolgáltatásai nem mindig köthetők egy bizonyos processzhez) nem. Ez a tárgyalásmód bevezethet egy új fogalmat: a *rendszer kontextus* fogalmat. E szerint a tárgyalás szerint a CPU vagy egy *processz kontextusában* fut, vagy a *rendszer kontextusban*. A processz kontextusban fut a CPU, amikor egy processz kódja haj-

tódik végre, mikor egy rendszerhívás szolgáltató rutin fut, mikor egy kivételes esemény szolgálódik ki. Rendszer kontextusban fut a processzor megszakítás kiszolgálása esetén. Ez a tárgyalásmód ismeri a *processz kontextus váltás* fogalmat és egy processz kontextusról *rendszer kontextusra való váltást* is.

2.1.2. A processz kontextus adatstruktúrái

A *processz címtartománya* (process address space) megragadja a processz kontextust.

A folyamatok kezeléséhez az operációs rendszer magja adatstruktúrákba rendezve tartja nyilván minden processz kontextusát, annak pillanatnyi állapotát. Az adatstruktúrák elemei a *processz tábla* (Process Table), a *processz tábla bejegyzés* (Process Table Entry), a *processz leíró* (Process Descriptor), végül a processz tábla bejegyzésekből, illetve a processz leírókból készült, az *állapotokat nyilvántartó láncolt listák*. Az adatstruktúra elemeknek más elnevezései is lehetnek: leggyakoribb név a processz tábla bejegyzésre pl. a *processz vezérlő blokk* (Process Control Block, PCB) elnevezés.

Az egyik legfontosabb elem, a kiinduló pont, a processz tábla.

A processz tábla (Process Table) az operációs rendszer magja által kezelt, adott méretű táblázat. Implementációja szerint manapság struktúrák készlete (array of structures), melyben egy-egy bejegyzés egy-egy ***processz belépési pont***, vagy *processz vezérlő blokk* (Unixban: *proc structur*e). A Process Table méretét (bejegyzéseinek maximális számát) a rendszergeneráláskor szabhatják meg. A táblának annyi bejegyzése kell legyen, hogy az egyszerre élő processzek mind bejegyezhetők legyenek. (Gondoljuk meg, mi történne, ha a tábla „betelne”?)

A processz tábla a kernel címtartományában van. Új bejegyzése (processz belépési pont) keletkezik egy processz kreálásakor, megszűnik a bejegyzése a hozzátartozó processz megszűntekor. A bejegyzése a hozzátartozó processzről "statikus" adatokat tartalmaz: olyanokat, melyek a processz létezését jelzik, de még nem (feltétlenül) elegendőek a processz futásához. A belépési pont, vagy PCB tartalmazza az azonosítókat (pid, ppid, pname stb.), tartalmaz statikus korlátokat (quótákat) és attribútumokat (statikus számlázási, időkiosztással kapcsolatos információk, erőforrás használati limitek stb.), a kód és adatszegmens hosszakat, mutatókat ezekhez (ami a memóriaallokálást, betöltésüket segíti, később ténylegesen a szegmensekre mutatnak), a vermek hosszát stb. Egy-egy bejegyzés tartalmaz előre/hátra mutató pointereket is, melyek a bejegyzések láncolt listájának kezeléséhez szükségesek.

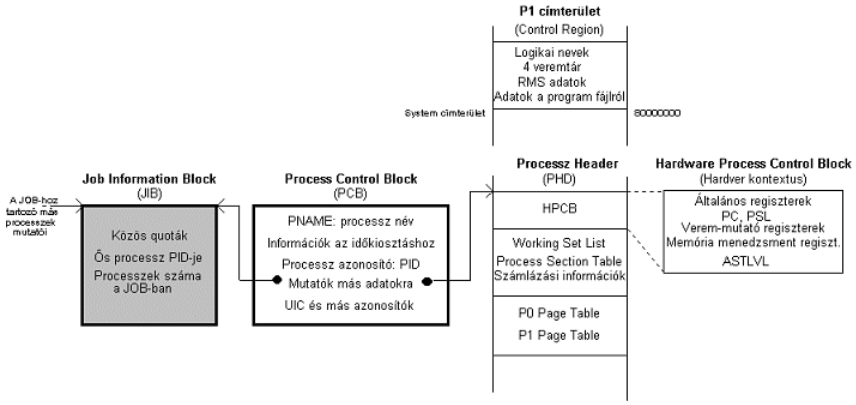
A processz tábla rezidens, azaz nem söpörhető, nem lapozható ki. A processz tábla belépési pont, vagy processz vezérlő blokk tartalma operációs rendszer függő, de szinte minden rendszerben tartalmazza a fenti "statikus" adatokat, némely rendszerben ennél többet is. Unix rendszerekben sajátos neve van: *proc structure* (leírása a *proc.h* header fájlban található).

A **processz leíró** (Process Descriptor) a kernel által kezelt struktúra, a kernel címtartományában. A processz futtathatóvá válásakor keletkezik, terminálódása során szűnik meg. Nem feltétlenül rezidens, azaz kisöpörhető, kilapozható. "Dinamikus" adatokat tartalmaz a hozzátartozó processzről, olyanokat, melyek a processz futásához (is) kellenek. Tartalmazza a vermek mutatóit, a volatile kontextust (vagy mutatót hozzá), a számlázási, ütemezési, állapot információkat, az erőforrás korlátokat, az aktuális eszköz, jegyzék, a kontroll terminál eszköz azonosítót, a nyitott I/O folyamok leíróit stb. Természetesen tartalmaz next/prev pointereket a láncolt listákon való nyilvántartás implementációhoz. A Unix-okban jellegzetes neve van: U-area.

Nézzük a processz kontextust leíró adatszerkezeteket két operációs rendszerben, a VAX/VMS-ben és a UNIX-ban.

2.1.3. VAX/VMS adatstruktúrák, melyek leírják a processz kontextust

Nézzük a 2.2. ábrát!



Az ábra közepén látható PCB-ből (ez a Process Table bejegyzése) indulunk ki.

A **PCB** statikus, permanens (nem lapozható, nem söpörhető ki!).

Tartalma: (nézzük az ábrát)

2.2. ábra. A VAX/VMS processz kontextust leíró adatstruktúrák

- PNAME (processz név mint egyedi azonosító);
- Scheduling information (Információk az időkiosztáshoz);
- Mutatók további adatstruktúrákhoz: némelyik a nyíllal mutat az ábrán is (JIB-re, HPCB-re, PHD-re);
- UIC and other identifiers (felhasználói azonosító, további azonosítók).

A Job Information Block (JIB)

Statikus tábla bejegyzése, az ún. munka (job) fogalomhoz kapcsolódik. Mi a *job*? Közös ősű folyamatok családja. Vannak közös erőforrásaik, limitjeik, korlátaik: ezeken osztoznak, egymás rovására használják ezeket.

A JIB bejegyzés tartalma: (olvassuk az ábrát)

- Pooled Quotas: a közös korlátok.
- Az ős processz PID-je.
- A család számossága.

A Process Header (Processz fejrész)

PHD statikus, nem permanens (lehet kisöpörve, részben kilapozva!) A processz virtuális címtartományába (lásd memory management) esik (ugyanakkor ebben van leírva a processz virtuális címtartománya!) A processz fejléc tartalma:

- Hely a HW (volatile) kontextus kimentésére (HPCB).
- Working Set List: (munkakészlet lista, lásd majd a memóriamenedzselésnél) azon memória lapok leírása, melyek a processzhez tartoznak. Jegyezzük meg: ez a rész nem lapozható ki. Főleg az ún. *page fault* kezelő használja.
- Process Section Table (processz szekcióit leíró tábla). Kilapozódhat.
- Accounting Information (számlázási információk). Kilapozódhat.
- P0 Page Table: Virtual Address Space Description (virtuális címtartomány leírók)

- P1 Page Table. Meghatározza az ábrán is látható Control Region-t, aminek a tartalmát tovább olvashatjuk. Kilapozódhat.

Ezzel tulajdonképpen minden megvan, mert a kontextus további részei ezekkel meghatározottak.

Egy absztrakciós szinttel lejjebb persze tovább részletezhetünk!

Hardware context (HPCB)

Dinamikus. Mikor a processz fut (övé a CPU), akkor a CPU regiszterek pillanatnyi értéke a HW kontextus. A legfontosabb regiszterek:

- Általános regiszterek.
- Program Counter és Program Status Longword (programszámláló és programállapot szó).
- Verem mutató regiszterek (4 db. a 4 futási módhoz: U, S, E, K)
- Memória menedzsment regiszterek.
- Asynchoronous System Trap Level Register.

Control Region (P1 Space)

Csak akkor elérhető, ha a processz fut. A virtuális címtartomány alsó résztartománya. Tartalma az ábrából:

- Logikai nevek a processzhez.
- Maga a 4 veremtár.
- RMS adatok, (nyitott adatfolyamok stb. leírói)
- Adatok a végrehajtható program (image) fájlról. (Nem maga az a kód és adat!)

P0 Space

Az ábrán nem is látható! Tartalma: a program szöveg, adat szekciók.

2.1.4. Unix processz kontextust leíró adatszerkezetek

Kiindulópont itt is a processz tábla egy bejegyzése, amit szokásos Unix terminológiával *proc structure*-nak nevezünk. A processz tábla permanens, nem lapozódhat, söprődhet ki (lásd: 2.3. ábra).

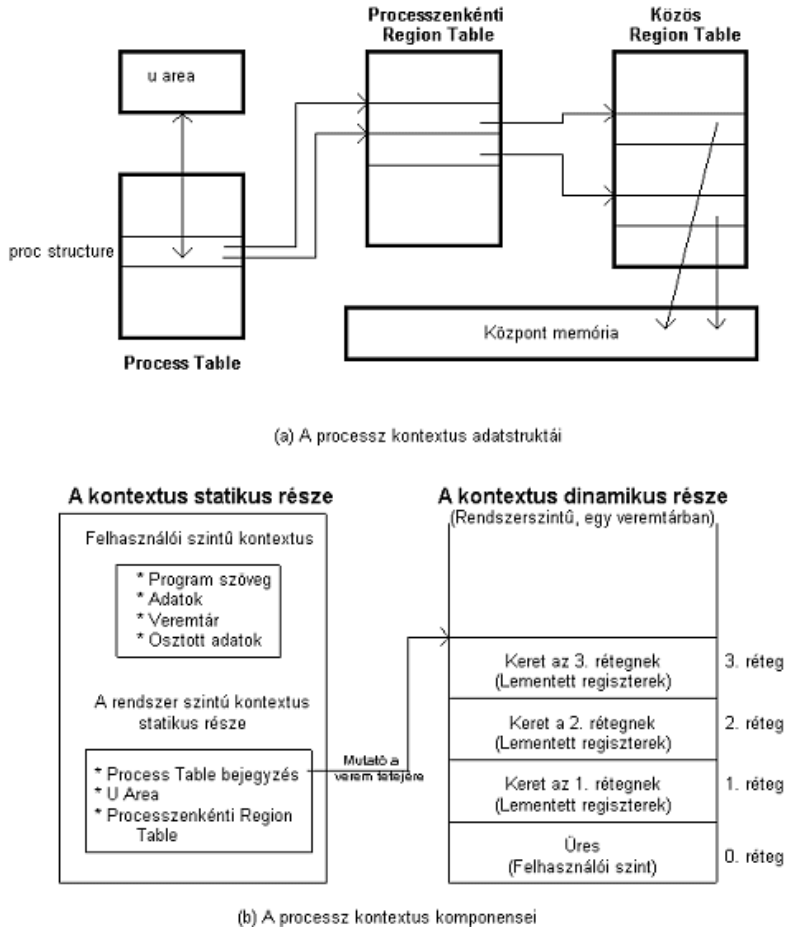
Proc structure (Process Table Entry)

Tartalma (nem teljesen):

- uid (tulajdonság jelzésére, de csak a BSD rendszerekben);
- pid, ppid, gpid (azonosításhoz);
- program méret, memória igény (memória menedzsmenthez);
- állapotjelző a memória menedzsmenthez;
- állapotjelzők az időkiosztáshoz (state fields az SVID rendszerekben) és scheduling paraméterek, mutatók az állapotlistákhoz;
- signal mezők (a kiosztott, de még nem kezelt szignálok értékelésére);
- mutatók (*u-area*-ra, *region table*-kra).

U-area (jellegzetes Unix terminus technikus ez is)

Kisöpörhető, kilapozható, nem permanens. Mérete 1K-4K rendszertől függően. Tartalma durván két részre osztható: a *user structure*-ra, és a processzhez tartozó *kernel verem*re (per-process execution stack for the kernel).



2.3. ábra. A Unix processz kontextus adatstruktúrái.

A ***user structure*** tartalmazza azokat az információkat, melyek a rendszer számára akkor fontosak, amikor a processz *nincs kisöpörve* (a ki/besöprés fogalmakat lásd a memóriamenedzselésnél). A user structure szerkezetét megtalálhatjuk a *user.h* header fájlban.

Tartalma:

- visszamutató a Process Table Entry-re;
- a valós (real) és effective felhasználói (uid) és csoport (guid) azonosítók (lásd később!);
- állapotjelzők (a BSD rendszerekben);
- időmezők (CPU felhasználás idejének kimutatására, külön *felhasználói mód*ra, *kernel mód*ra)
- szignálokra való reakciókat leíró tömb;
- nyitott I/O folyamok leírói, jellemzői;
- default eszköz és katalógus, bejelentkezési terminál;
- a processzhez tartozó rendszer bufferek címei (lásd majd az I/O kezelésnél);
- pillanatnyi rendszerhívás paraméterek, visszatérési érték, hibakód;
- erőforrás limitek, quóták, privilégiumok.

Region tables (régiókat leíró táblák)

Ezek bejegyzései memória régiókat írnak le, melyekben megtalálhatók a processz kódja adatai. Közvetve vehető mi, hol. Van processzenkénti *region table* leírás, és rendszer *region table*, - így valósulhat meg az osztott memória (shared memory).

Processz kontextus a tartalom szerint

A 2.3. ábra alsó, (b) jelű részét figyeljük. Az ábra feliratai érthetők (legalábbis a bal oldal, a kontextus statikus része). Érdekes a jobb oldal (Dynamic Portion of Context). A rendszer szintű dinamikus kontextus ez a rész: a kernel verme (stack). Ide mentődik a HW kontextus

(a regiszterek tartalma, volatile context), rétegekben. (A *process context switch* tárgyalásánál résztelezzük!)

2.2. A folyamatok vezérlése (Process Control)

A folyamatok

- születnek,
- élnek, van aktivitásuk, vannak kapcsolatok közöttük:
  - versenyeznek erőforrásokért (pl. CPU, RAM, diszkek stb.),
  - konfliktusmentesen megosztoznak erőforrásokon (pl. kernel kódon),
  - kommunikálnak, együttműködnek,
  - szinkronizáció lehet köztük (minden kapcsolatuk igényli a szinkronizációt),
- végül exitálnak.

Egyszerű rendszerekben a rendszerbetöltés (startup) folyamán minden processz "kézből" készül és végig éli a rendszer életét. A szokásos operációs rendszerekben azonban általában: processzt csak processz kreálhat (processz kérhet kreációs szolgáltatást a kerneltől). A kernel a kreációs kérésre elkészíti és tölti a processz tábla belépési pontot, memóriát biztosít a kód és adatszegmenseknek, elkészíti és kiölti a processz leírót, ekkor "elkészíti" a volatile kontextust (itt kap értéket a PC!), végül futtatható állapotba helyezi a processzt. Processz kreáló rendszerhívások lehetnek a create(), run(), load(), exec(), system(), illetve a fork(), vfork() rendszerhívások. Operációs rendszer függő, hogy milyen kreáló rendszerhívásokat implementálnak: tanulmányozni kell a dokumentumokat! (Unix-ban pl. a fork() az általános kreátor!)

Ha elfogadjuk, hogy processzt csak processz kreálhat, akkor beláthatjuk

- szülő-gyermek reláció alakul ki (vö. jegyzék-aljegyzék reláció a fájloknál);
- processz hierarchia alakulhat ki (v.ö. fájlrendszer).

(Így alakul ki a VAX/VMS *Job* fogalom, a Unix *process group* fogalom.)

Természetesen kell egy *ősszülő* processz.(v.ö. *root* jegyzék). Most nem részletezzük, de van ilyen!

A gyermek processz készítése során a *processzek élete* szempontjából két eset lehetséges:

- A szülő folytatja futását a gyermekével párhuzamosan;
- A szülő várakozik, amíg gyermeke(i) terminálódik(nak).

Két lehetőség van a szülő-gyermek processzek viszonylatában a processz kontextusok, processz címtartományok szempontjából is:

- A gyermek másolata (duplikátuma) a szülőnek, ugyanaz a program fut benne;
  - de a címtartományaik különböznek (elsősorban az adatszekcióik érdekesek: azok különböznek), vagy
  - osztoznak a címtartományuk legtöbb részén (közösek az adatszekcióik, csak a vermeik különböznek).
- A gyermek új, más programot futtat.

Processzek menedzselése

Processzek felfüggeszthetik futásukat (blokkolódhatnak), "elaludhatnak", akár meghatározott ideig, akár előre meghatározatlan időre, meghatározatlan vagy meghatározott esemény bekövetkeztéig. Tanulmányozni kell az ide tartozó rendszerhívásokat, melyek szintén OS függő-

ek! Ilyen hívások jöhetnek szóba: pause(), suspend(), sleep(), delay(), wait(). Blokkolt processz "felébredhet", ha bekövetkezik a várt esemény: jelzést kap (erről). Tanulmányozni kell ezért a szignálozó rendszerhívásokat is: alarm(), resume(), send-signal(), kill() stb.

A processzek menedzselési információi, az attribútumai jellegzetesen a kernel címtartományában vannak. Szükség lehet azok "megszerzésére" (kernel címtartományból a felhasználói címtartományba másolására), vagy "beállítására" (beírás a kernel táblákba). A get-attribute(), set-attribute() rendszerhívás-család segítségével kérhetjük ezeket a szolgáltatásokat a kerneltől. Persze, megint csak operációs rendszertől függ, hogy e két családban milyen konkrét hívások a megvalósítottak. Néhány Unix-os hívást felsorolunk: getpid(), getppid(), getuid(), setuid(), nice(), getitimer(), setitimer(), schedctl(), getrlimit(), setrlimit() stb.

A processzek terminálódása, megszűnése

Befejezve futását egy processz az exit(), abort() rendszerhívással kérheti az operációs rendszert, hogy törölje őt a *létező processzek készletéből (process pool*-ból).

Ekkor adhat vissza a szülőjének visszatérési értéket, szignálozhat a szülőjének. A terminálódás során a processzhez rendelt erőforrásokat (memória, nyitott fájlok, I/O bufferek stb.) az operációs rendszer visszaveszi. Ez az ún. szokásos, normális terminálódás.

Lehet futó processzt "megölni", megfelelő *szignál* kikézbesítésével is (lásd később). Leggyakoribb kikényszerített terminálás az az eset, mikor a szülő terminálja gyermekeit, mert

- a gyermek valamilyen hozzárendelt erőforrás-korlátot túllép;
- már nincs szükség arra a feladatra, amit a gyermek végez;
- a szülő terminálódna, de az operációs rendszer nem engedi, hogy a gyermek túlélje a szülőjét (cascading termination).

Processzek együttműködése

- Az operációs rendszer által futtatott, együtt élő processzek lehetnek
- *független* (independent) processzek: nincs hatásuk egymásra (valójában rejtett hatásuk persze van: erőforrásokért ők is vetélkedhetnek);
- *kooperáló* processzek: tervezett hatásuk van egymásra (pl. felhasználják egymás eredményeit, szinkronizálják egymást stb.).

Mi a kooperáció kiváltó oka?

- A vezérelt információ-osztás, pl. közös fájlokon való osztozás.
- A sebességnövelés: valamely feladat részekre osztva, párhuzamos processzekben megoldva gyorsabb (a teljesítménynövekedés feltétele itt: legyen több CPU, vagy több I/O csatorna).
- A modularitás. Bonyolult rendszerek kezelési módja a strukturális dekompozíció: az áttekinthetőség, kezelhetőség így jobb.
- A kényelem: a felhasználó gyakran igényli a párhuzamos tevékenységet (editál, fordít, nyomtat párhuzamosan).

A kooperáció processzek közötti kommunikációs mechanizmusokat, szinkronizációs mechanizmusokat és kölcsönös kizárási mechanizmusokat igényel.

2.2.1. UNIX subprocess system calls

Az érintett rendszerhívások (tanulmányozzák az on line man-nel!):

`fork(), exec?(), system(), exit()`

Érdekes lehet még a

```
nice(),  getpid(),  getppid(),  wait(),  waitpid(),  wait3()
```

és fájlkezelő rendszerhívások, amik hasznosak lesznek példáinkban. Ezeket is tanulmányozzák!

```
open(),  creat(),  read(),  write()
```

Tanulmányozható még a POSIX vfork() hívás is.

Jegyezzük meg: a Unix-ban a processz kreálás alapvetően a fork()-kal, vagy a fork()/exec() "villával" történik.

2.2.2. Processzek készítése Unix-ban: a fork ( )

Alapvető módszer új processz készítésére. A fork a koncepciója szerint a processz instrukciófolyamát két, konkurrens instrukciófolyammá osztja. A Unix-ban (POSIX) ezt gyermek processz kreálással valósítja meg.

Prototípus deklarációja:

```
pid_t fork( );
```

Hívásának szintaxisa:

```
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
pid_t  pid;
...
pid = fork( );
```

Szemantika:

Készít egy új gyermek folyamatot (ha sikerül!), melynek kontextusa a *pid*-et és CPU időfelhasználást kivéve (nézd a manual-ban a pontos különbségeket) ugyanaz, mint a készítőé. A legfontosabb talán az, hogy a szülőben és a gyermekben ugyanaz a programszöveg, ugyanott fut.

A forgatókönyve:

1. Ellenőrzi, készíthető-e a gyermek processz. Meghatározza a gyermek *pid*-jét. Előkészíti a virtuális memória kisöprési/kilapozási területét.
2. Bejegyzést készít a processz táblába a gyermek számára. Ehhez bizonyos adatokat a szülőtől örököl (uid, gid, signal mask, ppid stb.), mások nullával, vagy az új processzre jellemző töltődnek.
3. Elkészíti a virtuális memória címleképző tábláit.
4. Elkészíti a gyermek processz leíróját. Ennek adatait részben a szülőjéből másolja, részben a gyermek címleképzéshez igazítva állítja be.
5. A nyitott adatfolyamok hivatkozásait inkrementálja.
6. Inicializálja a gyermek hardver kontextust. Ez a szülő regisztereinek „másolásával” történik, majd a gyermeket „futtathatóvá” teszi.
7. Mindkét processzben visszatér, de a szülőben
   - a gyermek pid-jével (hiba esetén negatív értékkel);
   - gyermekben 0 (zéró) értékkel.

Példa:

```
main( )   {
int ppid, pid, gpid;

  ppid=getpid( );          // lekérdem a pid-et
                           //...
  if((pid=fork())==0) {    // itt a gyermek fut
         gpid=getpid( );   // Gyermekben a gyermek pid-je
                           //...
         exit(0);          // gyermek megszüntetése
  }
                           // itt a szülő fut
                           //...
        exit(1);           // szülöt megszüntet
}
```

2.2.3. A processz kontextus (statikus rész) változtatása

Új programot (*image*-et, kódot és adatot) töltök a kontextusba.

Erre szolgál az exec?( ) rendszerhívás család. Több változata van, ezek elsősorban az argumentumokban különböznek. Mi itt csak az execl() hívást részletezzük.

Prototípus deklarációja:

```
int execl(char *path, char *arg0, char *arg1, ...,
                      char *argn, 0);
```

Hivatkozás:

```
#include <fcntl.h>
#include  <unistd.h>
int status;
…
status=execl("public/child", "child", "elso", 0);
```

Szemantika:

A hívó processz felhasználói címtartományára (user address space) rátölti a "path"-tal jelölt végrehajtható programot (kódját, adatait), vermeket állít neki, és belépési címtől kezdően futásra kész állapotba teszi (futtatja). Átadja az aktuális argumentumokat neki (változó argumentumlistájú, az utolsó argumentum utáni nullával jelződik a lista vége).

Az exec?( ) függvények csak az argumentumaikban különböznek. Keressék őket az on-line man-ban.

2.2.4. Esettanulmányok

A következő apró programokat tanulmányozzuk:

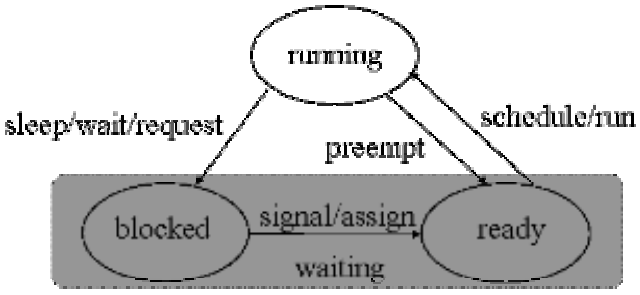
Első az ex_sys.c. A programban a *system( )* rendszerhívás az érdekes. Ez érvényes parancsot futtat, megvárja, míg fut. Implicit fork( )/exec ( ) villa van benne!

Nézzék az ex_7_4.c és az ex_7_41.c programokat is. Ezek másoló programok, saját magukat *fork*-olják. A két processz ugyanazt a fájlt ugyanabba célba másolja. Időzítési problémák miatt - ha szerencsénk van - lehet tévesztés a pufferezés miatt.

Végül a parent.c és a child.c programokat tanulmányozzák. A parent.c elindít egy gyermek processzt, ami különbözik a szülőtől. A szülő megvárja a gyermek lefutását. A gyermek szöveget ír a szabványos kimenetre.

2.3. Processz állapotok

Láttuk, minden processz önálló entitás a saját programszámlálójával és kontextusával. Lehet közöttük együttműködési kapcsolat, a legegyszerűbb példa: egyik processz készít valamilyen kimenetet, ami a másik processz input-ja. A két processz futásának relatív sebességétől függően előfordulhat, hogy a második processznek várnia kell, amíg az első az output-ját elkészíti. A második *blokkolt*, amíg az inputja elkészül. Kérdés merülhet fel, hogyan "billen" ki ebből az állapotból a blokkolt processz. Másrészt az is előfordulhat, hogy egy processz ugyan nem vár semmire, tehát futhatna, de az operációs rendszer egy másik processznek adja át a CPU-t: ekkor is "vár" a processzünk, most a CPU-ra, ezt az állapotát feltétlenül meg akarjuk különböztetni az input-ra való várakozástól. Azt mondhatjuk, hogy a processzek - életük során - különböző *állapotokban* (state) lehetnek, az állapotok között különböző *állapotátmenetek* lehetségesek. A legegyszerűbb és legáltalánosabb *állapot és állapotátmenet* diagram a 2.4.ábrán látható, ahol az ellipszisekkel jelöltek az *állapotok*:



2.4. ábra. Processz állapotok

- *running* - futó állapot, a processzé a CPU;
- *blocked* - blokkolt, alvó (sleeping) állapot, mikor a processz egy esemény bekövetkezésére vár;
- *ready* - futásra kész (computable) állapot, mikor a processz futhatna, ha megkapná a CPU-t.

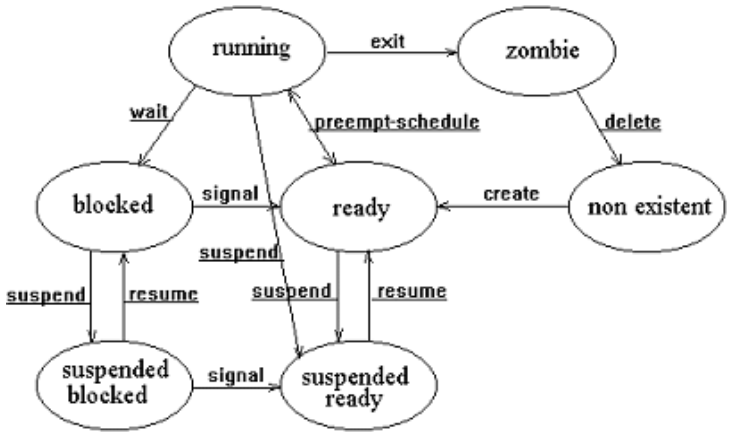
Az ábrán nyilakkal jelöltük az *állapotátmeneteket:*

- *wait/sleep/request* - várakozz (blokkolódj) eseményen állapotátmenet;
- *signal/respond* - jelzés az esemény bekövetkezésére;
- *preempt* - a CPU elvétele a processztől;
- *schedule* - A CPU kiosztása a processznek.

Kérdés merülhet fel, mi váltja ki az állapotátmeneteket a folyamat szemszögéből nézve? A válasz: egyedül a *wait/request/sleep* állapotátmenetet kezdeményezi maga a processz (pl. egy diszk blokk behozatalának kérelmével), az összes többi átmenetet a processz szemszögéből nézve külső entitás váltja ki (a *signal*-t pl. a diszkvezérlő megszakítása, a *preempt-schedule* átmeneteket a kernel időkiosztó alrendszere stb.).

Itt már megérthető a *folyamat környezet kapcsolás* (Process Context Switch) fogalom: ez egy-egy átmenet két processz számára, mikor is az egyiktől elvevődik (preempt/wait/sleep/request), a másiknak kiosztódik (schedule) a CPU.

Mint említettük, a fenti ábra a legegyszerűbb állapotdiagramot mutatja. Nem is látható rajta például, hogy keletkezik egy folyamat, hogy szűnik meg, és nem látható az sem, hogy az egyes konkrét operációs rendszerek e háromnál több állapotot és köztük több állapotátmenetet biztosítanak. Gondot jelenthet az is, hogy az egyes operációs rendszerekben az állapotok és állapotátmenetek elnevezése különböző lehet (pl. a VAX/VMS a futó (running) állapotot CUR (current) állapotnak nevezi, a blokkolt állapotot - a blokkolás fajtájától függően - különböző neveken tartja számon stb.). Még mindig egy általános - egyetlen konkrét rendszerhez sem kötött - állapot diagram az alábbi (2.5. ábra):



2.5. ábra. Processz állapotok

A *zombie* állapot: exitálódott processz állapota, amíg a szülő processz tudomásul nem veszi az exitálódást.

Non-existent állapot: a processz még nem létezik.

Suspended állapotok: felfüggesztett állapotok. A processzek ebben az állapotban nem képesek az erőforrásokért vetélkedni, sem a memóriáért, sem a CPU-ért nem jelenthetnek be igényt. Az ilyen állapotú processzek esélyesek a "kisöprésre". Egy processzt csakis egy másik processz "függeszthet fel", természetesen a védelmi eszközöket figyelembe véve. Általában a rendszermenedzser (az ő processze) függeszthet fel processzeket, ha azokat ki akarja zárni az erőforrásokért való vetélkedésből.

A processzek állapotáról a *burok* segítségével érdeklődhetünk. Unix rendszerekben a

```
> ps -l
```

parancs, (az állapotkódokat és további módosítókat vedd a man ps-ből) tájékoztat processz állapotokról.

VAX/VMS-ben a

$ SHOW SYTEM vagy

$ SHOW PROCESSES vagy a MONITOR segédprogram segítségével informálódhatunk a processzek állapotáról.

Miután a VAX/VMS folyamat állapotok kódjai elég sajátságosak, idemásoltam az állapotkódokat, rövid magyarázatokkal (2.1. táblázat).

2.1. táblázat

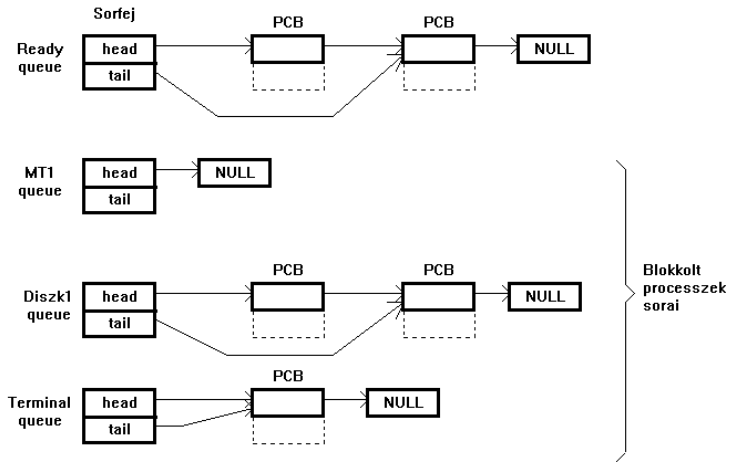
STATE	Magyarázat
CEF	Common event flag wait. Közös esemény flagre várakozás
COLPG	Collied page wait. Page fault történt, mialatt egy másik processz számára lapbeolvasás folyik
COM	Computable (Ready). Rezidens processz futásra kész

COMO	Computable, outswapped. Futásrakész, de ki van söpörve
CUR	Current. Éppen futó processz
FPG	Free-page wait. A processz a fizikai memória szabad lapjára vár
HIB	Hibernate wait. A processz hibernálva van
HIBO	Hibernate wait, outswapped. Hibernált, kisöpört
LEF	Local event flag wait. Saját esemény flag(ek kombinációinak) beállítására vár, rendszerint I/O-val kapcsolatban
LEFO	Local event flag wait, outswapped. Mint előző, plussz kisöpört
MUTEX	Mutual exclusion semaphore wait. Kölcsönös kizárási szemaforra vár, biztosítandó, hogy egyszerre csak egy processz férjen ugyanahhoz a kódhoz
MWAIT	Miscellaneous resource wait. Dinamikus erőforrásra vár. Az erőforrás kódja RWxxx (lásd alább)
PFW	Page fault wait. Page-fault következett be, a processz várja, hogy a kívánt lap a fizikai memóriába kerüljön
SUSP	Suspended. Felfüggesztett processz
SUSPO	Suspend wait, outswapped. Felfüggesztett, kisöpört

A Sun Solaris operációs rendszerben a processz állapot kódok (S oszlop) a következők:

- O a futó processz,
- S alvó (*sleeping,* blocked), erőforrásra váró,
- R *runable*, futásra kész processz
- Z *zombie*, terminált processz,amire a szülő nem vár, de még nem vette tudomásul, hogy terminálódott
- T *stopped*, leállított processz

Ugyanekkor az F (flags) oszlopot is érdemes nézni és értelmezni, ha a processzek állapotát meg akarjuk határozni.



2.6. ábra. A ready sor és különböző I/O várakozó sorok

2.3.1. A processz állapotok nyilvántartása

A processz kontextusban, legtöbbször a PCB-ben rögzítettek az állapotok. De ha minden döntéshez innen kellene kikeresni - végigolvasva a processz tábla bejegyzéseit - a megfelelő állapotú processzeket, nagy lenne a veszteségidő. Ezért az operációs rendszerek többnyire *láncolt lista adatszerkezeteken*, *sorokon* (queue) is nyilvántartják a különböző állapotú processzeket. Meglehetősen sok sor alakítható ki (lásd 2.6. ábra).

2.3.2. A fonalak (threads)

A *fonál* (thread, Lightweight Process: LWP) a CPU használat alapegysége. Egyedileg tartozik hozzá a programszámláló regiszter (PC), a regiszterkészlet és veremtár címtartomány; osztozik más fonalakkal a kód- és adatszekciókon, egyéb erőforrásokon (nyitott adatfolyamok, szignálok stb.), azaz a *taszk címtartományon*.

Új kifejezés ekkor a *taszk* (feladat), a Heavyweight Process.

A klasszikus processz tulajdonképpen taszk, egyetlen fonállal, egyetlen végrehajtási menettel.

Egy taszk nem csinál semmit, ha nincs benne egyetlen fonál sem.

Miután a fonálnak legtöbb erőforrása (osztottan) megvan a taszkjában, a fonalak közötti CPU kapcsolás, maga a fonál-kreáció "olcsóbb", mint a klasszikus taszk (processz) kontextus kapcsolás (Process Context Switch), mint a klasszikus taszk (processz) kreáció. A fonál kapcsolás például csak regiszterkészlet kapcsolás.

Némely - konkurens programozási - rendszer megengedi a felhasználói szintű (user level) fonalakat, felhasználói szintű könyvtár-hívásokkal megvalósítva, azaz nem rendszerhívásokkal megvalósítva. Nincs ekkor kernelbe való belépés (trap) a fonál kapcsolásnál. Felhasználói szintű fonalakkal megvalósított *szerver* taszkban egy-egy fonál tartozhat pl. egy-egy *kliens* kérelméhez, és egy fonál blokkolása, egy másik fonálra való CPU kapcsolás hatékony kiszolgálást biztosíthat több kérelemnek.

Hátránya is van persze a felhasználói szintű fonalaknak. Ha egy fonál request/wait rendszerhívást ad ki, az egész taszk blokkolt lesz, amíg a hívás ki nem szolgálódik.

Láthatjuk, a fonál sokban hasonlít a processzhez. Lehetnek állapotai (futó, blokkolt, futásra kész), kreálhat gyermek fonalakat. Egy-egy fonál szekvenciálisan fut, de a fonalak mégsem annyira függetlenek egymástól, mint a processzek. Minden fonál elérheti a taszkja címtartományát, például a társtér fonalak vermét is! A taszkon belül nincs védelem! De nincs is rá feltétlenül szükség, ha volna, megoldható a gond a taszk, processz koncepcióval: fonál kreáció helyett taszk, processz kreációval.

Próbáljuk meg tisztán megragadni a *taszk*, *processz*, *fonál* fogalmakat! A továbbiakban ezekből a processz fogalmat fogjuk használni, elsősorban az egyfonalas klasszikus proceszt értve a processzen. A *job* fogalmat is fogjuk említeni: elsősorban a kötegelt feldolgozású rendszereken futó programok számára, néha a több processzből álló alkalmazásoknál a processzek összefoglaló neveként.

Legalapvetőbb fogalmaink a taszk, a processz és a fonál fogalmak. További egységeket is láthatunk programozói szemmel: rutinokat (eljárásokat, függvényeket), utasításokat, végül instrukciókat. A rutinok közül egy aktív: amelyikben a vezérlés fut. Láthatjuk a hívási rendet (trace), ez is egy bizonyos szülő-gyermek reláció eredménye. Az a képzetünk, hogy a hívó rutin "le van fagyasztva", amíg a hívott vissza nem adja a vezérlést, vissza nem tér. Nos, ez a "lefagyasztás" valóban megvan, bizonyos szempontból hasonlít a processz, vagy a fonál dinamikus kontextus lementéséhez, de ezt a nyelvi fejlesztő rendszer biztosítja, nem az operációs rendszer.

2.4. Az időkiosztás (Process Scheduling)

2.4.1. Alapok

Erőforrásokért vetélkedő processzeknél alapvető feladata az erőforrások igénybevételének ütemezése. El kell dönteni, melyik processz kapja meg az erőforrást.

Ilyenkor meg kell különböztetnünk e tématerületen belül három feladatot.

- Erőforrás *hosszútávú hozzárendelés* (allocation),
- az erőforrás *ütemezés* (scheduling),
- *erőforrás kiosztása* (dispatching), az ütemezés után az a technika, amivel az erőforrást hozzárendelik a processzhez.

Valójában bármilyen erőforrás ütemezéséről és kiosztásáról beszélhetünk, az ütemezési algoritmusok, a módszerek és eljárások hasonlók lehetnek. Beszélhetünk így CPU ütemezésről, diszk ütemezésről stb. Kiemelkedő fontosságú ezek közül a CPU ütemezés és kiosztás.

A hosszútávú hozzárendelés során történik döntés, hogy egy processz melyik CPU-n fusson egyáltalán. Tulajdonképpen csak többprocesszoros rendszeren értelmezhető ez a mozzanat. Egyprocesszoros rendszerekben nincs ez a feladat, a továbbiakban nem is foglalkozunk vele. A processzhez így hozzárendelt CPU-t maga a processz "pszeudo párhuzamosságban" használja más processzekkel. Időnként ún. "döntési helyzetek" vannak, melyben el kell dönteni, hogy a következőkben melyik processz legyen a "nyertes", a CPU használó. Ez tulajdonképpen az ütemezés (scheduling). Az ütemezés valójában optimálás: a "legjobb" processz lesz a nyertes processz. Nyilvánvaló, hogy csakis a futásra kész állapotú processzek vesznek rész ebben az optimálásban.

A kernel egyik fő feladata a futásra kész processzek közül egy számára a CPU kiosztása. El kell döntenie, melyik futásra kész állapotú processz kapja meg a CPU-t. *Scheduler*-nek, *ütemezőnek* hívják kernelnek azt a részét, amelyik ezzel a döntéssel foglalkozik. Jól látjuk itt is a két különböző feladatot:

- az ütemezést, döntést arról, melyik processz kapja meg a CPU-t (scheduler algoritmusok, technikák);
- a dispatching-et, a CPU kiosztást, ami maga a CPU átkapcsolása egyik processzről a másikra (Process Context Switch mechanizmus megvalósítást).

Be kell látnunk, hogy az időkiosztás (és algoritmusa) független magától az átkapcsolási algoritmustól.

A régi, kötegelt rendszerekben az ütemezés és időkiosztás egyszerű volt, legtöbbször a *first come, first served* (a munkák felmerülési sorrendjében szolgáljuk ki őket) ütemezéssel a *run-to-completion* (fuss, míg befejeződik) módszer volt az általános, de az alább tárgyalt algoritmusok közül egyesek már ott is alkalmazhatók voltak.

Mit várunk el a CPU ütemező (scheduling) algoritmusoktól?

A kielégítendő kritériumok:

1. Pártatlanság: minden folyamat (processz, taszk, job) korrekt módon (nem feltétlenül egyenrangúan) kapjon CPU-t.
2. Hatékonyság: a CPU lehetőleg a legnagyobb százalékban legyen kihasználva.
3. Válaszidő: az interaktív felhasználóknak a válaszidőt minimalizálják, (ne vesszék el türelmüket, hiszen a "vakon" nyomogatott gombok tovább lassíthatnak).

4. Fordulási idő (turnaround time): a kötegelt munkák (job) fordulási idejét minimalizálni kell.
5. Teljesítmény: az időegységre eső job-feldolgozás, interaktív munka maximalizálása. (Lássuk be, ez különbözik a fent említett hatékonyságtól.)

Láthatók bizonyos ellentmondások a kritériumok között. A válaszidő minimalizálása eredményezheti a fordulási idő növekedését! Vagy: a számításigényes munkák korlátozás nélküli előnyben részesítése javítja a hatékonyságot, de nem biztosítja a korrektséget, és az összevont teljesítmény is csorbulhat. Komplikációt jelent, hogy a processzek, taszkok egyediek és nehezen jósolható viselkedésük. Mégis, van lehetőség elfogadható ütemező algoritmusokat találni, hiszen a processzek gyakran blokkoltak, várnak valamire, ez lehetőséget biztosít a többi futására. Technikai alapot nyújt, hogy a korszerű rendszerekben mindig van óraeszköz, ami periódikusan megszakítást generál, és ezzel lehetőséget biztosít, hogy

- az időt szeletekre (time slice, quantum) bontsuk,
- az erőforrás (pl. CPU) felhasználás idejét (processzenként) mérjük,
- bizonyos időnként az állapotokat kiértékeljük, és processzek közti kapcsolást valósítsunk meg.

Az ütemező (scheduler) *döntési stratégiája* - mely futásra kész processz kapja a CPU-t - alapvetően a következők egyike lehet:

- Nem beavatkozó *stratégia* (non-preemptive). Ez továbbá lehet:
  - *run-to-completion jellegű*: a processz, ha megkapta a CPU-t, addig használja, míg a (rész)feladatát el nem végzi,
  - *együttműködő* (cooperative) jellegű: a processz, ha megkapta a CPU-t, saját döntése szerint lemondhat róla.
- *Szelektív beavatkozó* (selective preemptive) stratégia: bizonyos processzek futásába nem lehet beavatkozni (rendszerint a rendszer processzeknél), más processzektől elveszik a CPU-t, még ha nem is mondana le róla.
- *Beavatkozó* (preemptive) stratégia: bár a folyamatok nem mondanának le a CPU használatáról, beavatkozva elveszik tőlük bizonyos körülmények között. Azokat az operációs rendszereket tartjuk valódi időosztásos rendszereknek, melyeknél létezik a beavatkozás (preemption) lehetőség. Az MS-DOS feletti MS-Windows nem biztosítja ezt, ott együttműködő (cooperative) időkiosztás van csak.

Ütemezési döntési helyzetek a következő esetekben léphetnek fel:

1. Amikor egy processz futó állapotból blokkolt állapotba megy (wait/sleep/request állapotátmenet), pl. I/O kérés vagy gyermek processzre való várakozás miatt stb.
2. Amikor egy processz futó állapotból futásra kész állapotba megy (preemption állapotátmenet), pl. egy megszakítás bekövetkezés miatt.
3. Amikor egy processz blokkolt állapotból futásra kész állapotba megy (signal/respond állapotátmenet), pl. egy I/O befejeződése.
4. Amikor egy processz terminálódik.

Az 1. és 4. esetekben az érintett processz szempontjából nincs is ütemezési döntési helyzet: másik processzt kell kiválasztani a futásra kész processzek sorából (ha az a sor nem üres). Van "helyzet" viszont a 2. és 3. esetben.

Ha ütemezési döntések csakis az 1. és 4. helyzetekben lépnek fel, akkor mondhatjuk, az ütemezés nem beavatkozó. Ha a 2. és 3. helyzetekben is lehet döntés, az ütemezés beavatkozó.

Az ütemezési algoritmusok vizsgálatához szükségünk van a processzek életének bizonyos szempontú jellemzésére. Megfigyelések szerint a processzek élete során vannak ún. *CPU-*

lázas (CPU burst) és *I/O-lázas* (I/O burst) életszakaszok. Egy processz a CPU burst időszakában a CPU-t kívánja használni, az I/O burst szakaszában elsősorban az I/O csatornákat használná, ilyenkor a CPU szempontjából blokkolt. A processzek futása CPU-lázas szakaszszal kezdődik, azt követheti I/O igényes futási rész, majd újabb "számolásigényes" életszakasz. Az egyes processzek jellemezhetők a "lázas" szakaszaik számával, hosszával.

A nagyon "számolásigényes" (CPU bound) processzeknek rendszerint kevés, de nagyon hosszú CPU burst periódusból állnak.

Az I/O igényes (I/O bound) processzeknél rendszerint rövidek a CPU-lázas szakaszok, ezek főleg az I/O csatornákat használják.

A következőkben vizsgáljunk időkiosztási algoritmusokat.

2.4.2. Igénybejelentési sorrend szerinti kiszolgálás (Fist Come - First Served)

Nagyon egyszerű, könnyen megvalósítható algoritmus ez. A processz készletben (process pool, job pool) létező processzek a beérkezésük sorrendjében kapnak kiszolgálást: ha egy előbb érkezett processz futásra késszé válik (pl. CPU-lázas szakaszában van), akkor ő kapja meg a CPU-t. Egyszerű, de nagy hátránya, hogy kialakulhat a *convoy effect*, hosszú ideig várakozhatnak processzek, amíg egy CPU igényes processz a CPU lázas szakaszaival végez (lassú teherautó mögött összegyűlnek a különben gyorsabb haladásra képes személyautók, de képtelenek előzni).

2.4.3. A legkisebb igényű először (Shortest Job First) algoritmus

Más néven: Shortest Job Next algoritmus. Régi, nagyon egyszerű időkiosztási algoritmus. Nemcsak történelmi okokból említjük, hanem azért is, mert ma is használják ezt az algoritmust pl. a *printer spooling sorok* kiszolgálására: itt persze könnyebb a dolog, a legrövidebb nyomtatandó fájl nyomtatása valószínűleg a legrövidebb időigényű, azt pedig könnyű megállapítani. De a régebbi kötegelt feldolgozásoknál a CPU kiosztására is alkalmazták, miután bizonyítható, hogy az *átlagos fordulási idő* (avarage turnaround time) így a legkisebb.

Tételezzük fel, hogy van a, b, c, ...x időigényű munka. Ha ebben a sorrendben futnak, az egyes munkák fordulási ideje:

a, a + b, a +b +c, ...

Ezek átlaga:

$$T_{atl} = \frac{(a + (a + b) + (a + b + c) + ... + (a + b + c + ... + x))}{n}$$

azaz

$$T_{atl} = \frac{(n \cdot a + (n-1) \cdot b + (n-2) \cdot c + ...x)}{n}$$

vagyis, ha a, b, c, ... x rendezett, a $T_{átl}$ minimumot ad.

A CPU kiosztás vezérlésénél egyetlen gond, hogyan lehet megmondani előre, hogy az egyes munkák mennyi időt fognak igényelni. Erre a válasz.

- A régi kötegelt rendszerekben tapasztalati adatokból jól lehetett ezt becsülni. A Job Control Language nyelven adott vezérlő kártyákon fel is kellett tüntetni a várható, becsült fordulási idő értéket. (Aki persze előnyt akart kicsikarni, az "csalt", de a nagy "téveszté-seket" lehetett szankcionálni.)
- Az idősorozatokkal jellemezhető munkák várható idejének becslésére jó szokott lenni az öregedés (aging) becslési algoritmus. (Sok helyütt használják az aging-et, nézzük ezért meg egy kis példán a lényegét.)

Az *aging* algoritmus

Tételezzük fel, hogy a, b, c, ...munkák (CPU lázas szakaszok) ismétlődve felmerülnek. Ismételt felmerülési idejük nem feltétlenül egyenlő. Valamely munka (szakasz) várható ideje a korábbi időiből becsülhető, a korábbi idők súlyozott összegéből vett átlaggal. A súlyozással azt befolyásoljuk, hogy milyen arányban vegyük figyelembe az egyre régebbi értékeket. Az öregedés lényege: a régebbi idők egyre kisebb súllyal befolyásoljanak, egyre jobban felejtse el a rendszer a régebbi értékeket, "korosodjon".

Legyen pl. egy munka idősorozata

$T = (T_0, T_1, T_2, \ldots T_i, \ldots)$ tényleges idők (tényleges CPU-lázas szakaszok idői)

A következő rekurzív formula jó lehet a becslés számítására (B_i: az i-edik becsült idő):

$$B_0 = T_0$$
$$B_{i+1} = a \cdot T_i + (1-a) \cdot B_i$$

ahol $0 <= a <= 1$.

Az a értékkel azt befolyásoljuk, mennyire "felejtsen" a becslésnél az algoritmus. A formula exponenciális átlagot biztosít.

Legyen pl. az $a = 1/2$, ekkor $B_{i+1} = \frac{(T_i + B_i)}{2}$

és a becsült sorozat

$$B = \left(T_0, \left(\frac{T_0}{2} + \frac{T_1}{2} \right), \left(\frac{T_0}{4} + \frac{T_1}{4} + \frac{T_2}{2} \right), \left(\frac{T_0}{8} + \frac{T_1}{8} + \frac{T_2}{4} + \frac{T_3}{2} \right), \ldots \right)$$

(Ráadásul a 2-vel való osztás bit-eltolással (shift) nagyon gyorsan végezhető!).

2.4.4. Prioritásos algoritmusok

Ha úgy tetszik, a FC-FS illetve a SJF algoritmus egy-egy speciális esete volt a prioritásos algoritmusoknak. Az elsőnél a "korábban érkezés", a másodiknál a "rövidebb becsült időszakaszom következik" jelenti a magasabb prioritást.

A *prioritás* a processzek fontossága. Foglalkoznunk kell a prioritásokkal. Léteznie kell egy *prioritási függvénynek*, ami a processzek fontosságát jelzik. A prioritás - a fontosság - sokmindentől függhet:

- a processz memóriaigényétől,
- a processz eddigi CPU használati idejétől,

- a processz (várható) összes CPU használati idejétől,
- a processznek a rendszerben eltöltött idejétől (ez biztos nagyobb, mint az eddigi CPU használati ideje),
- külsőleg adott prioritási értéktől,
- a processz időszerűségétől (timeliness), ami azt veszi figyelembe, hogy a processz élete során a fontossága változhat,
- a rendszer terhelésétől (system load) stb.

A FC-FS algoritmusban a prioritás a processzek érkezési sorrendje volt. A SJF algoritmus prioritása a processzek várható futásideje. Másik, szintén egyszerű prioritás függvény lehet a külső prioritásérték, a processz életében ez statikus, nem változik: eredménye lehet az "éhhalál" (starvation), egy magas statikus prioritású processz megakadályozhatja, hogy más processzek CPU-t kapjanak. Ezért elvárjuk, hogy a prioritásértékek korrekt módon változzanak a processzek élete során.

A további algoritmusok mind összefüggenek valahogy a prioritásokkal, valamilyen dinamikus prioritásfüggvény kiszámítása segíti az ütemezést.

2.4.5. Igéretvezérelt időkiosztás (Policy Driven Scheduling)

Interaktív rendszereknél jól használható algoritmus ez. Alapja, hogy mérhető a processzek

- rendszerben eltöltött eddigi ideje: az *élet-idő*,
- eddig felhasznált CPU ideje: a *cpu-idő*.

A filozófia: reális ígéret, hogy *n* számú processz esetén egy processz a CPU 1/n-ed részét kapja. Ehhez kiszámítandó az ún. *jogosultsági-idő*:

jogosultsági-idő = élet-idő/n

Amelyik processznél a *cpu-idő*/*jogosultsági-idő* arány a legkisebb, annak magasabb a prioritása, az kapja meg a CPU-t.

Az ígéret vezérelt időosztás speciális esete a **valós idejű** (Real-Time) időkiosztás.

Itt az egyes processzek kapnak egy *garanciát*, melyen belül biztosan megkapják a CPU-t, ha igényt jelentenek be rá. A *garantált idő* kiszámítása előzetesen megtörténik, és persze, a *garanciával rendelkező* processzek száma korlátozott.

2.4.6. Roud-Robin scheduling

Egyszerű, korrekt, széleskörben használható, könnyen megvalósítható, elég régi algoritmus ez. Nem szokták magyarra fordítani az elnevezést: elég furcsa lenne a *kerge-rigó* név!

Alapja az óra eszköz: segítségével megvalósított időszeletekre való osztás, az *időszelet* (time slice, quantum) fogalom.

Módszere: ha valamely processz megkapta a CPU-t, addig futhat, amíg a hozzá rendelt időszelet (*quantum*) tart. (Ha közben blokkolódik, akkor persze eddig sem.) Ha letelik az ideje, a *scheduler* elveszi tőle a CPU-t (preempt), és átadja egy másik processznek: ez a *processz kapcsolás* (Context Switch). A *scheduler* a futásra kész processzeket egy *listán* tartja nyilván. CPU kapcsolás esetén a CPU-ról *lekapcsolt* (preempted process) a *lista* végére kerül, míg a *lista* elején álló processz megkapja a CPU-t (scheduled process). A listán a processzek "körben járnak". A lista elején álló processznek legmagasabb a prioritása, ha úgy tetszik.

Megfontolásra érdemes a round robin időkiosztásnál a az időszelet nagyság és a CPU kapcsolási (Context Switch) időigény aránya. A kapcsolási idő egy adott érték, azon aligha változtathatunk, de meggondolhatjuk, mekkorára válasszuk az quantum értéket.

Tételezzük fel, hogy a kapcsolás időigénye 5 msec, ugyanakkor a quantum értéke 20 msec. Ezekkel az értékekkel 20% a "veszteségidő". Ha quantum értéket 500 msec-re állítjuk, a veszteség már csak 1 %. De gondoljuk csak meg, az 500 msec 1/2 sec, mekkorák lesznek a válaszidők, ha mondjuk 10 interaktív processz áll a sorban? Jól meg kell tehát fontolni a quantum értékét!

2.4.7. Többszintes prioritás-sorokon alapuló scheduling (Multilevel Feedback Queue Scheduling)

A klasszikus round robin azt feltételezi, minden processz egyforma fontosságú. Ez persze nem egészen így van! Vannak fontosabb személyek, akiknek a processzei előnyt kell élvezzenek, valahogy figyelembe kell venni a *külső prioritásokat* is. Persze, a nagy külső prioritású processzek nem akadályozhatják meg, hogy a kevésbé fontos processzek egyáltalán ne kapjanak CPU-t, a scheduler által biztosított, processzekhez rendelt *belső prioritásoknak* dinamikusan változniuk kell. A belső prioritásokat a scheduler egy prioritási függvény szerint dinamikusan állítja elő, amely függvénynek csak egyik paraméter a külső prioritás. Azzal, hogy a processzek belső prioritása dinamikusan állítandó, biztosítható a korrektség, és egyéb célok is elérhetők (pl. I/O igényes processzek előnyben részesíthetők, hiszen ezek gyakran várakoznak eseményekre, míg a számításigényes proceszek hajlamosak lennének kisajátítani a CPU-t). A dinamikus prioritásszámítással figyelembe vehető a proceszek memóriaigénye is, az processzek *eddig felhasznált CPU ideje*, az *élet idejük*, az *időszerűségük* stb. is.

Az elgondolás ezek után a következő: ha a futó processz időszelete lejárt, történjen meg a dinamikus prioritás értékek meghatározása, és a maximális dinamikus prioritással rendelkező processz kapja a CPU-t. Egyenlő prioritások esetén képezzünk listákat, és a lista elején álló processz legyen a kiválasztott (Round Robin, de a preempted processz nem biztos, hogy erre a listára kerül).

2.4.8. A VAX/VMS scheduling algoritmusa

Elegáns, több megfontolást is figyelembe vevő megoldása van ennek a rendszernek.

A VAX/VMS-ben 32 prioritási szint van, amit két, egyenként 16 szintből álló csoportra bontanak. A 31-től 16-ig tartozó szintek - minél nagyobb a szint száma, annál nagyobb a prioritás - foglaltak a valós idejű processzek számára. A többi - reguláris - processz a 0 - 15 prioritási szinteken osztozik.

A valós idejű processzekhez rendelt prioritás érték a processz élete során nem változik: statikus az érték. a reguláris processzek prioritási értéke viszont dinamikusan alakul.

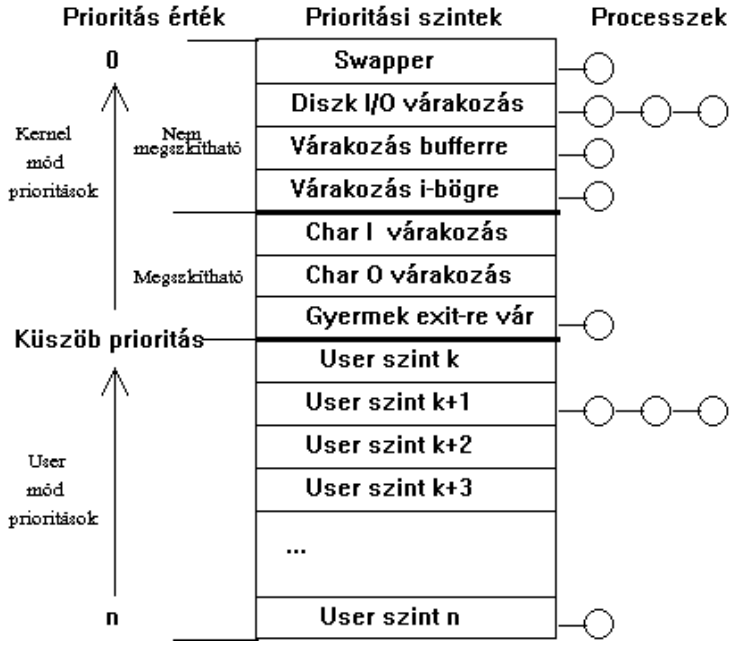
Minden processz a keletkezésekor kap egy *alap* (base) *prioritási értéket*, ez egyben a processz minimális prioritási szintje. (Az alap prioritás szintet a rendszergazda állítja be, a SYUAF fájlban tárolva, ahol a felhasználó számlaszám információi tárolódnak.) A dinamikus prioritásérték kiszámítása a következők szerint történik.

Minden rendszer eseményhez hozzárendelt egy prioritás növekmény, aminek a nagysága az esemény jellegétől függ. (Pl. terminál olvasás esemény nagyobb növekménnyel rendelkezik, mint egy terminál olvasás esemény, ami viszont nagyobb, mint egy diszk I/O eseményé.) Amikor a processz "felébred" valamelyik ilyen esemény bekövetkezésére, a növekmény hoz-

záadódik a pillanatnyi prioritási értékéhez, és a processz besorolódik a prioritási szinthez tartozó listára. Természetesen, maximum 15 prioritást kaphat így a processz. A lekapcsolt processz (preempted process) pillanatnyi prioritása, miután elhasználta a rendelkezésére álló CPU időszeletet, csökken 1 értékkel, és besorolódik az így meghatározott listára. A csökkentésnek is van korlátja, ez az alap prioritás-érték, ez alá a pillanatnyi prioritás nem mehet. Ezzel a processz prioritása az alap prioritás és a 15-ös szint között fluktuál. A diszpécser mindig a legnagyobb prioritási szinthez tartozó lista elejéről választ a CPU kapcsoláshoz.

2.4.9. A Unix időkiosztása

A Unix rendszerek szokásos időkiosztási algoritmusa *round robin with multilevel feedback* típusú: a kernel a lejáró időszeletű processzt lekapcsolja (preempt) és elhelyezi valamelyik prioritási szinthez tartozó sorra. A Unix prioritási szintjei a 2.7. ábrán láthatók.



2.7. ábra. Prioritási szintek a Unix-ban

Ami ebből rögtön látható, a processzeknek külön *user-mode prioritási*, külön *kernel-mode prioritási* értéke lehet, ezzel a prioritások tartománya is két résztartományból áll. A két tartomány több prioritási értéket foglal össze, mindegyik értékhez egy sor tartozik, amire a hozzátartozó processzek fel vannak fűzve. A *user-mode prioritási* listákon lévő processzektől el lehet venni a CPU-t (ezek lehetnek "preempted" processzek, amikor kernel-mode/user-mode visszaváltás történik.) A két prioritási osztály között van a *küszöb* (treshold) *prioritás*. A kernel szintű prioritások tovább osztályozhatók: nem megszakítható és megszakítható prioritás osztályokra. A prioritási értéket a Process Table belépési pont megfelelő mezejében tartják nyilván.

A kernel a következőképpen számítja ki a prioritásértéket:

- A *kernel prioritás* egy fix érték, attól függ, mi az oka, nem függ a futási karakterisztikától (pl. attól, hogy CPU igényes, vagy I/O igényes-e a processz). Pl. annak, hogy diszk I/O-ra váró, alvó processz prioritása magasabb, mint a bufferre váróé, az az oka, hogy az I/O-ra várónak már van buffere, ha feléled, nagyobb az esély, hogy feldolgozva a buffert, elereszti, ezzel esélyt ad a buffert igénylő processznek is. Hasonlóan, a többi kernel prioritásérték rendnek megvan a magyarázata.
- A kernel a *user mód prioritást* (p-usrpri) a kernel módból felhasználói módba váltáskor számítja ki, de az óra megszakítás kezelő (clock handler) bizonyos időközönként (VAX 4.3BSD rendszernél pl. 40 msec-onként, SVID rendszereknél nagyobb, de max. 1 sec-os intervallumokban) "igazítja" a prioritásértékeket.
  - a futó processz p-cpu mezejét az órakezelő minden óramegszakításkor a CPU használattal arányosan növeli. Ezzel számontartják, hogy a quantumban mennyi CPU-t használt a processz.

- a fenti p-cpu értéket quantumonként "öregítik" (aging): p-cpu = p-cpu/const1 (rendszerint const1 = 2, azaz a CPU használattal arányos érték mindig feleződik)
- ezután a prioritás így számítódik ki (a nagyobb érték alacsonyabb prioritást jelent!): p-usrpri = PUSER + p-cpu/ const2 + const3 * p-nice, ahol
  - const2 = const3 = 2, rendszerint;
  - PUSER a processz bázis prioritása: külső prioritásérték;
  - p-cpu az öregített CPU használat;
  - p-nice a felhasználó által kontrollált nice érték (-20 és + 20 intervallumban adhatja a felhasználó a nice() rendszerhívással. A negatív érték növeli, a pozitív csökkenti a processz esélyét, alapértelmezés a 0).

A p-usrpri egy adott maximális érték fölé nem mehet.

Hogy megértsük, nézzük ezt egy példán.

Az adott rendszeren, ha egy processz megkapja a CPU-t és végig használja azt a időszeletében, a p-cpu növekmény legyen 60.

A bázis prioritás (PUSER) szintén legyen 60. Három processz prioritását követjük nyomon, közülük az *A* és a *B* processz éppen p-cpu = 0 "öregített" prioritásértékkel indul, a *C* processz pedig p-cpu = 7 értékkel. Az *A* processz legyen előbb a 60-as prioritáshoz tartozó listán. A nice érték mindhárom processznél legyen 0.

Ekkor a prioritások így változnak (2.8. ábra):

Quantum	process A		process B		process C	
	p-uspri	p-cpu	p-uspri	p-cpu	p-uspri	p-cpu
1.	**60**	**0** **1** **2** **...** **60**	60	0	63	7
2.	75	30	**60**	**0** **1** **2** **...** **60**	61	3
3.	67	15	75	30	**60**	**1** **2** **3** **...** **61**
4.	**63**	**7** **8** **9** **...** **67**	67	15	75	30
5.	76	33	**63**	**7** **8** **9** **...** **67**	67	15

2.8. ábra. Példa a prioritások változására

Befolyásolhatjuk-e az időkiosztást?

Kismértékben. Tanulmányozzuk a következő rendszerhívásokat: getpriority() setpriority(), schedctl(), getitimer(), setitimer(), npri()! A rendszermenedzser állíthatja, hangolhatja a quantum értéket.

A rendszermenedzser beállíthatja a bázis prioritási értéket. (Ez mind a VAX/VMS-ben, mind a Unix-okban hatásos.)

A felhasználó kismértékben befolyásolhatja processzei időkiosztását a nice() hívással. Ezzel saját processzei között bizonyos kontrollt valósíthat meg, de pozitív argumentumú hívással a processzének esélyeit más felhasználók processzeihez képest is rontja. A felhasználó javíthatja helyzetét, ha az alkalmazásait több processzben írja meg. Egyes rendszerekben a felhasználó választhat ütemezési stratégiát (korlátok között, rendszerint csak a szuperuser).

Gyakorlatok:

Nézzék át az idővel kapcsolatos rendszerhívásokat, adatszerkezeteket (stime(), time(), times(), alarm(), pause(), nice() stb). Készítsenek kis programokat ezek használatára.

2.4.10. CPU kapcsolás mechanizmusok

A processzek közötti *CPU kapcsolás* (Process Context Switch) tulajdonképpen az az operáció, mely során az éppen futó processztől elvesszük a CPU-t és egy kiválasztott futásra kész processznek adjuk oda. Feltételezzük, hogy az ütemezés megtörtént, ki van jelölve a futásra kész processzek közül a „nyertes" processz.

Szűkebben megfogalmazva az operációban az éppen futó processz hardver kontextusát (dinamikus/volatile kontextust) *lementjük valahová*, a kiválasztottnak kontextusát pedig *felveszszük*.

Legfontosabb a PC (Program Counter) regiszter lementés-felvétel! Szorosan véve ez a kapcsolás.

Az ide tartozó processz állapotátmenetek:

- A lementés
  - *wait/reques/sleept* (blokkolódás) állapotátmenetnél; vagy
  - *preemption* (beavatkozás, elvétel) állapotátmenetnél történhet.
- A felvétel *schedule* (futásra készből futó állapotra váltás) állapotátmenetet okoz.

A CPU kapcsolás - elég gyakori - hardver függő operáció. Egyes architektúrákon léteznek *speciális gépi instrukciók*, amik a kapcsolást megvalósítják: lementik, ill. felveszik a hardver kontextust. Ezek szigorúan kernel módban végrehajtható instrukciók természetesen.

Egyes processzoroknak több regiszterkészlete van, azok váltásával megtörténhet a kapcsolás, nem is szükséges lementés-felvétel. Igaz, hogy regiszterkészlet váltással nem a processzek közötti kapcsolást szokták megvalósítani – hiszen valószínűleg több processz van, mint ahány regiszterkészlet – hanem megszakítás kiszolgáláshoz a processz kontextusról a rendszer kontextusra való váltást (lásd 2.1.1. fejezet)! Ha óramegszakítás kiszolgálásról van szó (ez történik a rendszer kontextusban), akkor előfordulhat, hogy a megszakítás kezelő „úgy dönt", szükséges a processz kontextus váltás: ekkor a processz volatile kontextust (a megfelelő regiszterkészletből) mégis lementik, a nyertes processz kontextust felveszik (az előbb lementett készletbe), majd a megszakítás kiszolgálásból való visszatéréskor visszaváltják a regiszterkészletet (máris futhat a nyertes processz).

Egyes architektúrákon nincsenek speciális instrukciók erre a célra, nincs több regiszterkészlet sem. Ilyenkor a "szokásos" gépi instrukciókkal kell megoldani a kapcsolást.

Le kell menteni a hardver kontextust. Kérdés merül fel: hova mentsünk le (honnan emeljünk fel)? Lehetne

- a Process Table sorába. Hátrány: nagy lesz a tábla. A lementés/felvétel hardveresen nem biztos, hogy támogatott.
- Veremtárba. Előny: hardveres push/pop is van, ezt kihasználhatjuk. Hátrány: kimerülhet a veremtár.

Az utóbbi az általános! Legföljebb az a kérdés, hogy melyik veremtárba mentsünk! Az biztos, hogy nem a processzenkénti felhasználói szintű verembe, de az már rendszerfüggő, hogy processzenkénti kernel verembe mentenek-e, vagy egy, minden processz számára közös kernel verembe! Az utóbbi esetben lehetséges a veremkimerülés, az előbbiben nem valószínű!

Nézzünk egy lehetséges megoldást, ami a processzenkénti kernel verembe ment, onnan vesz fel, így valósítva meg a processz kapcsolást.

Egy lehetséges Context Switch mechanizmus

Emlékezzünk a processz kontextust leíró általános adatszerkezetekre! Láthatjuk ezeket a 2.3. ábra b. részén.

A processz kontextus dinamikus része - ez megegyezés szerint a kernel kontextushoz tartozik - a futó processznél a regiszterekben, blokkolt és futásra kész processzeknél egy veremtár kereteiben (frames) van tárolva.

Amikor egy processz felhasználói módban fut, a Dynamic Portion of Context a 0-ás rétegen áll. Új réteg keletkezik (és ebbe lementődnek a regisztertartalmak), ha

- rendszerhívással vagy kivételes esemény kiszolgálására (trap) belépünk a kernelbe;
- ha megszakítás (interrupt) következett be, és ezzel beléptünk a kernelbe;
- ha Context Switch következett be a processz számára (preemption állapotátmenet: elveszik tőle a CPU-t).

Megszűnik egy réteg, ha

- system service rutinból visszatérünk felhasználói szintre;
- megszakítás kiszolgálásból visszatérünk;
- Context Switch következett be (egy processznek odaadják a CPU-t).

Ebből kiszámítható, mekkora legyen az érintett verem:

- egy réteg a rendszerhívásnak, kivételnek;
- egy réteg a Context Switch-nek;
- a megszakítás szintek száma szerinti réteg: a megszakításoknak.

A Context Switch lépései

1. Döntsd el, lehetséges-e a Context Switch.
2. Mentsd le a kurrens processz dinamikus kontextusát a *save_context()* algoritmussal.
3. Találd meg a "legjobb" processzt, ami megkapja a CPU-t (scheduling algoritmusok végeredménye).
4. Vedd vissza ennek dinamikus kontextusát (a verméből a legfelső réteget) a *resume_context()* algoritmussal.

Pszeudó kódok a Context Switch-hez

Tételezzük fel, hogy az adott rendszeren egy függvény visszatérési értéke az R0 regiszterben történik, a PC regiszter pedig a programszámláló regiszter, PSW a státus szó.

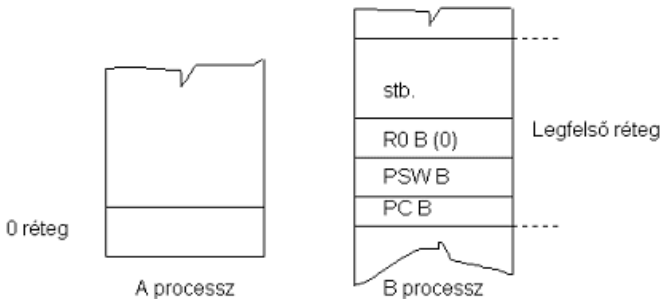
Ekkor a kódok:

```
int save_context ()  {
   R0 <- 0
   SWIT {
   PUSH PC; PUSH PSW;  // Ez általában hardveres push
      Push általános regiszterek;
      R0  <- (-1);
   JUMP a lementett PC utáni utasításra (Return-re);
   }
   Return;
}

int resume_context( pid_t pid)
{
   A pid-del azonosított processz verméről vedd vissza
   a regisztereket, utoljára (hardveres pop-pal) a PSW, PC-t.
   // ide a vezérlés sohasem juthat el! (Miért? Mert a
   // visszavett PC a save_context Return-jére mutat!)
}
```

A Context Switch kódja ezután:

```
if (save_context())   {
   Találd meg a másik processzt, ennek pidje: new_pid;
   resume_context(new_pid);
   // Ide a vezérlés nem juthat el!
}
   // Innen futhat a visszavett kontextusú processz
   .
   .
```

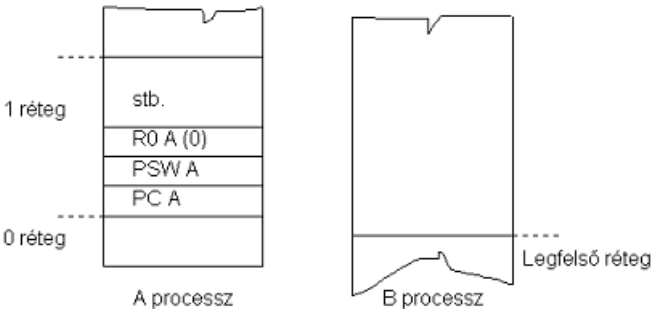
Példa: Képzeljük el, hogy van A és B processzünk, a B a futásra kész legjobb (volatile kontextusa a vermében lementve van), az A fut (volatile kontextusa a regiszterekben van). Ekkor a veremtáraik állapota (2.9. ábra):



2.9. ábra. Két processz dinamikus kontextusát tartalmazó veremtára

Az A processz kontextusán végrehajtott if(save_context()) egy új réteget készít az A vermében. Ebben a rétegben az R0 még 0 értékű, de a save_context () visszatérési értéke már -1! Az pedig az if utasításban igaz értéket jelent, azaz bemegy a vezérlés az if testébe, kiválasztja a másik processzt (esetünkben legyen ez a B), és végrehajt egy resume_context (B)-t. Ez a resume_context visszaveszi a B legfelső rétegét. Ebben az R0-t is, ami itt 0! Vé-

gül a visszavett PC a B kontextusa fölötti save_context Return-jére mutat, azaz a vezérlés ismét egy if(save_context()) kódban fut, de ott a visszatérési érték (ami az R0-ban jön), 0. Ez az if-nek hamis értéket jelent: a vezérlés az if teste utáni utasításra ugrik. Az eredmény a 2.10. ábrán látható.



2.10. ábra. Veremtárak a context switch után

Vegyük észre a következőket!

A save_context() rutinból – akármely processz kontextusán hajtódott végre - kétszer tér vissza a vezérlés!

Először a rutin a processz kontextusán történő direkt meghívásakor tér vissza - 1 visszatérési értékkel, amikor is ténylegesen lementődik a hardver kontextus. Többek között ekkor a PC regiszter érték is lementődik (és a PC lementett értéke éppen a processz kontextusán lévő Return-re mutat).

Másodszor is van visszatérés a processz kontextusán futó save_context()-ből! Egy másik processz kontextusán futó resume_context() ugyanis felveszi a hardver kontextust, a PC-t is, ami belemutat a save_context()-be! Igaz, ekkor a visszatérési érték már 0 lesz, az if-ben szétválasztódik a két visszatérés.

A resume_context()-ből viszont sohasincs visszatérés! Meghívott resume_context() mindig „átpasszolja" a vezérlést egy másik processz save_context()-jének belsejébe.

Gyakorló feladatok

Tanulmányozzák a sigsetjmp() és siglongjmp() rendszerhívás családot! Ezek éppen úgy viselkednek, mint a save_context() és resume_context() párok, az első lementi a regisztertartalmakat, a második felveszi azokat (igaz, a felhasználói címtartományban deklarált struktúrába, struktúrából), és az elsőben megfigyelhetik a kétszeres visszatérést. Tanulmányozzák a kiadott mintaprogramot. Készítsenek hasonló kis programokat!

3. Hiba- és eseménykezelés

3.1. Alapfogalmak: események, kivételek, megszakítások

Az esemény legáltalánosabban: folyamatok (taszkok, processzek, fonalak, rutinok, utasítások, instrukciók) futását befolyásoló, váratlan időpontban bekövetkező történés. Az esemény bekövetkezése esetén reagálni kell rá: **le kell kezelni**. A lekezelés megváltoztatja az instrukció-folyam normális menetét.

Az eseményt a hardver vagy szoftver generálja és detektálja.

Az esemény változás valamilyen *entitás* állapotán. Egy információs állapot előállása: egy állapottér állapotvektora. Fogalmazhatunk úgy, hogy egy esemény bekövetkezése azt jelenti, hogy előáll a neki megfelelő állapot, vagy fordítva, ha előállt az állapot, akkor bekövetkezett az esemény. Ezért az állapotot sokszor *feltétel állapotnak* (condition, error condition, exeption condition), vagy röviden *feltételnek* szokták nevezni.

Ha a feltétel előáll (bekövetkezett az esemény), akkor az valahogyan *jelződik*. Más fogalmazással: *jelzés keletkezik*. Ismét más fogalmazással: valamilyen entitás *jelzést küld* valamilyen entitásnak. A jelződés alapja a lekezelhetőségnek, ami kapja a jelzést, az lekezelheti az eseményt.

A lekezelés megváltoztatja a futás menetét:

- a normális futás menetét abba kell hagyni. (Kérdés: a gépi instrukciót? Az utasítást? A rutint? A fonalat (thread)? A processzt? A taszkot?)
- Reagálni, kezelni a helyzetet: kezelő instrukciófolyammal, rutinnal, processzel.
- Dönteni kell arról, hogy lekezelés után visszaadhatjuk-e a futást az abbahagyott entitásra (instrukcióra, vagy utána; utasításra vagy utána; rutinba, vagy annak elejére, vagy utána; stb.), vagy az operációs rendszernek adjuk a vezérlést.

Az eseményt és lekezelését eddig általánosan tárgyaltuk. Kíséreljük meg a pontosbítást és az osztályozást!

Az események osztályai

Egy processz szemszögéből lehet

- belső esemény: amit a processz állít elő;
- külső esemény: valami, a processzen kívüli entitás állítja elő (pl. perifériavezérlő interrupt).

Az előállító entitás szerint lehet

- hardver által generált és detektált (megszakítások, hibák, anomáliák);
- szoftver által generált és detektált (szoftver által generált megszakítások, szoftver által generált és emulált kivételes feltétel állapotok, a szűkebb értelemben vett események).

A megszakítások.

A legalacsonyabb szintű események a megszakítások (interrupts).

A processz szemszögéből nézve külső esemény. (Ha a processz saját magának küld szoftver megszakítást, akkor tekinthető belső eseménynek is.) Előállítója rendszerint a hardver (eszközvezérlők), de lehet szoftver is. Jelzése a CPU-nak szól. Lekezelői az operációs rendszer kernel IT kezelő (IT hadler) rutinjai. Kezelésének módja: az aktuális instrukció befejeződik, a

kontextus dinamikus része lementődik (rendszerint részben hardveresen!), a lekezelő rutin fut, visszatérve a dinamikus kontextus felvevődik és a soron következő instrukció futhat. Az IT prioritási szinteknek megfelelően IT kezelést megszakíthat megszakítás. A bekövetkezett, de még le nem kezelt megszakítások hardveres sorokon várnak lekezelésre.

Jól installált operációs rendszer kernel esetén minden IT lekezelhető.

A kivételek (exeption), hibaesemények

A nevükből következően valamilyen abnormális helyzetet jelentenek. Néha azt mondjuk, olyan események, amelyek a normális futás során nem jelentkeznek, kivételesek, vagy valamilyen hibára utalnak, gondot jelent jelentkezésük.

Lehetnek alacsony szintűek (pl. túlcsordulás bekövetkezése a CPU-ban), de lehetnek magasabb szintűek is (pl. laphiba (page fault), ami - bár nevében ott a hiba szó - egész normális jelenség; vagy pl. fájlnév tévesztés miatti nyitási hiba stb.).

Alacsony vagy magas szint? Ez azt jelenti, hogy a kivétel jelzése instrukciónak, rutinnak, fonálnak stb. szól? Vagy a kezelés szintjét jelenti?

Mindkettőt!

A klasszikus kivétel bekövetkezése esetén az éppen futó entitás (instrukció, utasítás, rutin, fonál, processz) nem fejezhető be! Fel kell függeszteni, le kell kezelni a kivételt, és - ha egyáltalán lehetséges - a felfüggesztett entitást elölről kezdve újra kell futtatni (pl. az instrukciót laphiba esetén); vagy folytatni kell (pl. rutint onnan, ahol felfüggesztésre került).

Az operációs rendszerek - felhasználva persze az IT kezelés amúgy is meglévő adottságait - képesek alapértelmezés szerinti módon (default handler-ekkel) kezelni a kivételeket. Többnyire biztosítanak a programozó számára is programozási eszközöket, melyekkel a kivételkezelés részbeni felelőssége, a kivételkezelés lehetősége (legalább részben) biztosított. Itt a magasabb szint értelmet nyer tehát, veszíti jelentését viszont a "hiba" jelző: hiszen nem feltétlenül jelentenek - akár az operációs rendszer, akár a mi általunk lekezelt - kivételek hibákat, legfeljebb csak gondot.

A szűkebb értelemben vett esemény

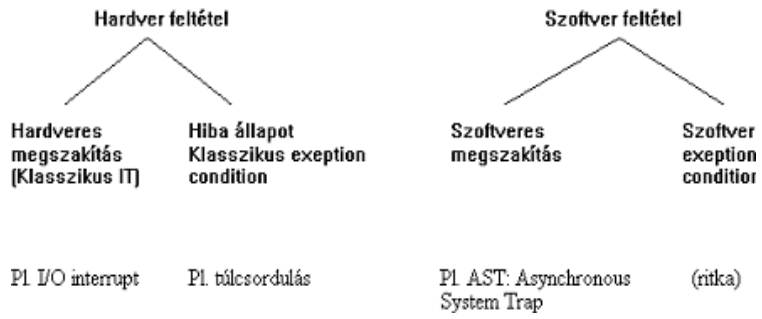
Ezek bekövetkezését a programozó előre látja, vár a bekövetkezésükre, bár a bekövetkezés idejét nem tudja: váratlanok és aszinkron jellegűek. Tipikus példa erre az X11 eseményvezérelt programozása (event driven programing): pl. az egér egy gombjának lenyomása, egy ablak feltárulása esemény az alkalmazás számára, neki jelződik, ő reagál rá.

Jellegzetesen magas szintűek az események, feltétlenül kellenek hozzá magas szintű fejlesztő eszközök, a programozó hatáskörébe tartozik maszkolásuk, lekezelésük stb. Jelzéseik legtöbbször egy eseménysorba kerülnek, lekezelésük e sor figyelésével történhet.

Miután a szűkebb értelemben vett események kezelése magas szintű feladat, a továbbiakban nem foglakozunk velük külön. Megemlítjük, hogy kezelésükben igénybe vehetők a kernel megszakítás-kezelési, vagy kivételkezelési technikái is.

A további fogalmak már csak a megszakításokra és a kivételekre vonatkoznak.

A feltétel (condition) fogalom: az az információs állapot, ami előáll, mikor egy esemény bekövetkezik. Lehet *hardver feltétel (hardware condition)*, vagy *szoftver feltételről (software condition)* beszélni.



3.1. ábra. A feltételek osztályai

Ha az információs állapot hibára, anomáliára utal szoktuk *hiba feltételnek (condition)*, vagy *kivételnek (exeption condition)* is nevezni.

A megszakítások és a hibaállapotok hasonlósága, különbségei

Mindkettő átadja a vezérlést egy kiszolgáló rutinra (handler). A kiszolgálások megszakítás, vagy hibaállapot specifikusak, bár nagyon sokban hasonlóak.

- A megszakítás (IT) aszinkron a kurrens instrukciófolyam végrehajtásában. Kiszolgálása két instrukció között (vagy egy adott instrukció végrehajtásában egy jól definiált ponton történik.
- Az exeption condition - bár váratlan - szinkron jelleggel fordul elő mint direkt hatás egy instrukció végrehajtása során. Kiszolgálása az instrukció végrehajtása közben történik, azaz a kérdéses instrukció végrehajtása folytatódik, vagy éppen megismétlődik. Tipikus példa a laphiba kivétel (page fault exeption condition): egy instrukció a címleképzés során generál laphibát. Ez azt jelenti, hogy az adott című laphoz nincs lapkeret rendelve a fizikai tárban. Lekezelése behozza a kérdéses lapot a fizikai tár egy lapkeretébe, ezután az instrukció sikeresen megismételhető (vagy annak címleképzési része sikeresen folytatható).
- A megszakítás rendszerint nem kapcsolódik a futó processzhez. Kiszolgálása ugyan történhet a kurrens processz kontextusa fölött, de nem valószínű, hogy a futó processz javára.
- Az exeption condition általában a kurrens, futó processz része: kiszolgálása a futó processz instrukciófolyamának kiterjesztése, kapcsolódik a kurrens processzhez, a futó processz javára történik.
- A megszakítások kiszolgálásának rendje összefügg a megszakítási prioritási szintekkel (Interrupt Priority Level). Kiszolgálásuk letiltható (lemaszkolható) az IPL (Interrupt Priority Level) szint állításával, de egy megszakítás kiszolgálását magasabb prioritású megszakítás blokkolhatja. Befutott, de nem kiszolgálható megszakítások sorbaállva várnak kiszolgálásra (pending interrupts). Rendszerint hardveres a sorképzés.
- A kivételes események kiszolgálása nem blokkolódhat. A kivételes események kiszolgálási rendje rendszerint független az IPL szintektől, kivételes eseményt másik kivételes esemény "nem szakíthat meg".
- Némely rendszernél a megszakítások kiszolgálására közös rendszer vermet használnak (system wide stack), mivel a megszakítások "system wide" jellegűek. A kivételes események kiszolgálására minden rendszernél processzenkénti kernel verem használatos.

A szignál fogalom

A szignál kifejezés meglehetősen túlterhelt, több értelemben is használjuk. Próbáljuk meg tisztázni az értelmezéseket.

Jelzés (signal): esemény bekövetkezésekor - vagyis a feltétel állapot előállásakor - jelzés (signal) keletkezik, szoktuk mondani. Jelzések keletkeznek normál futás közben események hatására. Jelzések értesítik a CPU-t, vagy egy-egy processzt az eseményekről. Úgy is fogalmazhatunk: a CPU, vagy egy processz jelzést kap, kézbesítenek neki egy jelzést. Az események jelződnek. Ebben az értelemben az esemény és a jelzése lehet még szinkron vagy aszinkron is.

Ezzel a terminológiával egyszerűsíthetjük a tárgyalást:

- van egy jelzés készlet;
- jelzés keletkezhet hardver vagy szoftver eredetből;
- a keletkezett jelzés kikézbesítődik egy processznek. vagy a CPU-nak. A kikézbesített jelzések sorokba rendeződhetnek, várva, hogy lekezeljék őket.
- A kikézbesített jelzéseket a lekezelő rutin (handler) lekezeli.

A jelzéseknek van egy készlete. Minden jelzéshez hozzárendelt egy kezelő (handler). Jelzést kézbesít ki a rendszer, amikor detektál egy hardver eseményt (pl. exeption condition-t, amikor detektál egy szoftver feltételt (pl. stop kérelmet terminálról), illetve processzek küldhetnek szignált más processzeknek (Unixban a *kill ()*, *killpg(), sigsend()* rendszerhívásokkal, VMS-ben az AST mechanizmuson keresztül.)

Ez a szignál fogalom tágabb értelmezése, azt hangsúlyozza, hogy az események jelződnek.

Szignálozás, aszinkron események kezelése processzekben

A szignál fogalom ebben a szűkebb értelmezésben az előző szignál fogalomból levezethető: aszinkron eseményeket jelzünk processzek számára. *Jelzések keletkezhetnek* (signal generation), keletkezett *jelzések kézbesíthetők* (delivering of signals) ki processzek számára. A processzek előzőleg *rendelkezhetnek a szignálokról* (signal dispositio), hogy mi legyen a rendszerakció a kikézbesített szignálokra. A szignálok keletkezésének és kézbesítésének megkülönböztetése magával hozza a *felfüggesztett szignálok* (pending signals) fogalmat. A processzek a *szignálokat blokkolhatják* is (blocked or masked signals), ami valójában a kikézbesítés megakadályozása. Miután a blokkolás nem a keletkezés, hanem a kézbesítés akadályozása, a blokkolás hatással van a felfüggesztéssel, továbbá kapcsolat van a *ignorációs kezelési rendelkezéssel* (ignored signals, blocked pending signals).

A szignálkezeléshez az OS kernelek biztosítanak programozói felhasználói felületeket (APIs). A szignálozó rendszerhívásokkal alkalmazásainkban rendelkezhetünk egyes szignálok kezeléséről, blokkolhatunk és megszüntethetjük a blokkolását egyes szignáloknak, generálhatunk magának a processznek vagy más processznek szignálokat. Mindezek hasznosak lehetnek a processzek közötti szinkronizációban, az eseményvezérelt programozásban. Mi ezekből Unix API-kat fogunk nézni. Említhető ezekből a BSD szignálkezelő API, az SVID szignálkezelő API és a POSIX API. A legtöbb Unix mindhárom felületet ismeri. Ez annyiban gondot jelent, hogy az egyes rendszerhívás családok keverten korlátozottan használhatók: elsősorban a szignálok blokkolásának, a diszpozíciónak eltérő adatstruktúrákban való nyilvántartása miatt. Külön gondot jelent annak "felfedése", mely szignálokat tudja és milyen korlátok között, "kezelni" a programozó a szignálozó rendszerhívásokkal.

Mielőtt "szignálozni" kezdenénk, tisztázzuk az ide kapcsolódó fogalmakat!

Jelzés keletkezés (szignál generálás, szignál postázás)

A jelzés (ebben az értelemben) egy processz számára aszinkron esemény bekövetkezésekor keletkezik A generálódás oka lehet éppen egy hardver esemény/hiba, egy kivételes esemény - pl. aritmetikai hiba, kontroll terminál vonal megszakadás (hangup), kontroll terminálon "megszakítás" (ctrl/break) előidézés stb. - a processzben. Ekkor egyszerűen azt mondjuk, *jelzés keletkezett*. De az is lehet, hogy egy más processz küld jelzést az "áldozat" processzünknek. Ekkor a jelzés generálódás helyett mondhatjuk: *jelzést postáztak* a processznek (sending signal, signal to post). Egy processz küldhet szignált explicit rendszerhívással (kill, sigsend), de rejtettebb módon is: pl. gyermek processz exitálása során mindig küld egy megszűntem (SIGCLD vagy SIGCHLD sorszámú) jelzést a szülőjének. A küldő processz számára a küldés ugyan szinkron, de a célzott processz számára nyilvánvalóan aszinkron ez a fajta jelzés generáció.

Rendelkezés a jelzésről (signal dispositio)

Az eseményeket kezelni kell, léteznek tehát szignálkezelők is. A rendelkezés a szignálokról, vagy szignál diszpozíció annak specifikálása, hogy *milyen rendszerakció menjen végbe*, ha szignált kap egy processz. Arról rendelkezünk, hogy mi legyen a szignálkezelő, hogyan történjen a kezelés.

Nyilvánvaló, hogy diszponálni "előre kell": mielőtt a szignált a processz megkapja, előtte kell megmondani a kezelés módját. A diszpozíció megváltoztat(hat)ja a kezelést, ez a változtatás addig él, amíg újra nem diszponálunk. Ugyanazon a szignál két diszpozíciója között lehet, hogy nem is kézbesítik ki ezt a szignált.

Nyilvánvaló az is, hogy minden szignálnak kell legyen alapértelmezés szerinti kezelője (default handler), ami kezel, ha másként nem rendelkeztünk előre. A default handler-ek általában "gorombák": nem adják vissza a vezérlést, terminálják az "áldozat" processzt. Természetes, hogy az *alapértelmezési kezelés explicite be is állítható* (diszponálható, visszaállítható).

Rendelkezéssel (diszponálással) beállíthatjuk, hogy egy szignálnak *saját kezelője* legyen: egy saját magunk által írt függvény legyen a kezelő. Nem minden szignálra írhatjuk ez elő, de elég sokra. Külön megfontolásokat igényel a saját kezelő írása: befolyásol a szignálok blokkolása és az ebből fakadó szignál-felfüggesztés, befolyásol az a tény, hogy mi történjen egy rendszerhívás kiszolgálással, ha éppen azt "függeszti fel" egy saját kezelő, annak eldöntése, hogy a kezelőnek saját verme legyen vagy sem, megfontolandó, hogy a saját kezelő lefutása után visszaálljon-e a default kezelés vagy sem stb.

Rendelkezéssel (diszpozíció) beállíthatjuk azt is, hogy *hagyjuk figyelmen kívül* (ignoratio) a szignált.

A Unixokban a "szignálozásban kezelhető" szignálok diszpozíciója processzek keletkezésekor (forkoláskor) a szülő diszpozíciókból inicializálódik. Öröklődnek diszpozíciók. A függő szignálok (lásd később) elvesznek. Az exec?() rendszerhívás a szignál dispozíciókat az alapértelmezési kezelőre (SIG_DFL) állítja. A SIGKILL és SIGSTOP szignálok diszpozíciója nem változtatható (nem ignorálható, saját kezelés sem írható elő).

Jelzés kézbesítése (delivering of signal), függő szignálok (pending signals)

Akkor mondjuk a szignált *kikézbesítettnek* (delivered signal), *amikor a diszponált akció beindult.* A szignál keletkezésekor az még csak generált szignál (generated or posted signal).

Amikor az akció indul, akkor már kézbesített a szignál. A keletkezés és kézbesítés között idő telik el, ezt az időt szokásosan nem is detektálhatja a kérdéses "áldozat" processz. Ez idő alatt a *szignált függőben lévőnek* (pending signal) mondjuk. Általában minden keletkezett szignál függő egy rövid ideig. Az ún. blokkolt szignálok hosszabb ideig is függőek lehetnek.

Jelzések blokkolása, a blokkolás megszűntetése

A szignálok blokkolhatók is. A blokkolással megakadályozhatjuk, hogy kikézbesítsék azokat a processznek. A blokkolás megszüntetés azt jelenti, hogy engedélyezzük a kikézbesítést. A blokkolás persze nem a keletkezést, hanem a kézbesítést akadályozza meg. Nyilvánvaló, hogy a blokkolást és a blokkolás megszüntetését is "előre" meg kell mondani a kérdéses processzben (bár van "automatikus" blokkolás és megszüntetése is: saját kezelőre diszponált szignál kezelése során, vagyis míg a kezelő fut, többnyire blokkokolt és normális return-jére megszűnik blokkolása). Egyes rendszerekben a blokkolás és megszüntetése kifejezések helyett a szignálok maszkolása-maszkolás megszüntetése kifejezéseket használják.

Tegyünk különbséget az ignorált és a blokkolt szignál között!

Egy blokkolt és ignorált szignál keletkezésekor "elszáll", megszűnik. Egy blokkolt és nem ignorált szignál keletkezésekor függő szignál lesz, addig, míg megszüntetik a blokkolást (és ekkor kézbesül, megszűnik függősége), vagy amíg mégis diszponálják neki az ignorációt, és ekkor a blokkolás megszűnése nélkül "elszáll", megszűnik (megszűnik függősége is ezzel).

A Unix-ok szignálkezelése

A Unix-okban a szignálokat általában akkor kézbesítik ki, mikor a processz kernel módból visszatér felhasználói módba. Ez minden rendszerhívásból való visszatéréskor, illetve minden IT kezelésből való visszatéréskor megvalósul. Miután az óra IT gyakorisága "megszerezhető" információ, a szignálok felfüggesztési idejére számítható felső korlát. (Vigyázz persze a blokkolt szignálokra!)

Szignál maszk a Unix-okban

Minden processznek van saját szignál maszkja, ami rögzíti, hogy pillanatnyilag mely szignálok blokkoltak (megakadályozott kézbesítésük). Mint említettük, a maszkot a szülő processztől öröklik (pontosabban abból inicializálódik a maszk).

Általában egy diszponált handler futása alatt az adott szignál automatikusan blokkolt, nem kell explicit hívás a blokkolásra, megszüntetésre). Ugyanazon szignál keletkezése esetén tehát az új szignál függő lesz, majd az első kezelő lefutása után kikézbesítődik ez is. Ügyeljünk viszont arra, hogy ha a handlert longjmp családhoz tartozó hívással hagyjuk el, akkor nincs "automatikus" blokkolás megszüntetés: a blokkolást explicit hívással meg kell szüntetni.

3.2. Unix szignálozással kapcsolatos API (Application Program Interface)

A C nyelvi programfejlesztési környezetben a szignálozással kapcsolatos makro definíciók, prototype deklarációk a *signal.h* beleértett fájlban tekinthetők meg. Tanulmányozzuk ezt a fájlt, ami rendszerint a /usr/include/signal.h vagy /usr/include/sys/signal.h elérési útvonalon található.

Megtaláljuk ebben a lehetséges (kezelhető) szignálkészletet definiáló makrókat. Az Irix-ben például pillanatnyilag a következő szignálok definiáltak (és az alapértelmezési diszpozíciójuk a következő):

```
Name  Value Default Event
_____________________________________________________
SIGHUP   1  Exit   Hangup (terminál vonal hangup) [see termio(7)]
SIGINT   2  Exit   Interrupt (Ctrl/break a terminálon) [see
termio(7)]
SIGQUIT  3  Core   Quit [see termio(7)]
SIGILL   4  Core   Illegal Instruction
SIGTRAP  5  Core   Trace/Breakpoint Trap
SIGABRT  6  Core   Abort
SIGEMT   7  Core   Emulation Trap
SIGFPE   8  Core   Arithmetic Exception
SIGKILL  9  Exit   Killed
SIGBUS   10 Core   Bus Error
SIGSEGV  11 Core   Segmentation Fault
SIGSYS   12 Core   Bad System Call
SIGPIPE  13 Exit   Broken Pipe
SIGALRM  14 Exit   Alarm Clock (A real-time óra szignáloz)
SIGTERM  15 Exit   Terminated
SIGUSR1  16 Exit   User Signal 1 (Ezt szabadon használhatja a user)
SIGUSR2  17 Exit   User Signal 2 (Ezt szabadon használhatja a user)
SIGCHLD  18 Ignore Child Status Changed (pl. exitál a gyermek)
SIGCLD   18 Ignore Child Status Changed
SIGPWR   19 Ignore Power Fail/Restart
SIGWINCH 20 Ignore Window Size Change
SIGURG   21 Ignore Urgent Socket Condition
SIGPOLL  22 Exit   Pollable Event [see streamio(7)]
SIGIO    22 Exit   input/output possible signal
SIGSTOP  23 Stop   Stopped (signal)
SIGTSTP  24 Stop   Stopped (user) [see termio(7)]
SIGCONT  25 Ignore Continued
SIGTTIN  26 Stop   Stopped (tty input) [see termio(7)]
SIGTTOU  27 Stop   Stopped (tty output) [see termio(7)]
SIGVTALRM 28 Exit  Virtual Timer Expired
SIGPROF  29 Exit   Profiling Timer Expired
SIGXCPU  30 Core   CPU time limit exceeded [see getrlimit(2)]
SIGXFSZ  31 Core   File size limit exceeded [see getrlimit(2)]
SIGRTMIN 49 Exit   POSIX 1003.1b SIGRTMIN
SIGRTMAX 64 Exit   POSIX 1003.1b SIGRTMAX
```

A signal.h fájlban vannak definiált makrók a szignálkezelés akcióinak definiálására:

- SIG_DFL (void (*) ()) 0 default kezelés,
- SIG_IGN (void (*) ()) 1 ignorlás kérése.

Vannak típus és struktúra definíciók, melyekre szükség lehet; megvannak azon rendszerhívások prototype-jai, melyek a szignálozáshoz kellenek:

A szignál diszpozíció: hogyan állítsuk be egy szignál kezelőjét

Már a legősibb Unix-okban is használható volt erre a *signal()* rendszerhívás. A signal() hívás egyszerű és szinte "korlátlanul" használhatjuk diszponálásra. Az SVID-ben a *sigset()* rendszerhívás szintén diszpozícióra való, és hasonlóan egyszerű a használata. A két rendszerhívási függvény prototípusa

```
void (*signal (int sig, void (*func)()))();
void (*sigset (int sig, void (*disp)()))();
```

Ahol *sig* a szignál száma (használhatjuk a makrókat), a *func* illetve a *disp* a kezelő függvény címe. Használhatjuk itt a SIG_DFL makrót, ami default kezelést, vagy a SIG_IGN makrót, ami ignorálást kér a *sig* szignálra. Illegális szignálra, vagy beállíthatatlan akcióra a két rendszerhívás SIG_ERR értékkel tér vissza (és az errno változó figyelhető). Normálisan visszaadja a korábbi kezelési címet.

Jegyezzük meg: ignorált szignál kezelése így is marad, amíg újra nem állítjuk. SVID rendszerben a signal() hívással diszponált saját függvénnyel lekezelt szignál esetén mielőtt a vezérlés belép a saját kezelő függvényre, visszaállítódik a SIG_DFL: Ez azt jelenti, mielőtt a kezelőből kilépünk, szükség esetén signal() hívással újra állítsuk be a saját kezelőt! A signal() hívás törli a még le nem kezelt *sig* sorszámú függő (pending) szignálokat! Az on-line manuel-ből nézzük meg, mely szignálok kezelése vehető át, melyek ignorálhatók!

Kimondottan BSD Unix-okban a diszpozícióra használhatjuk a *sigvec()* rendszerhívást. Ennek prototípusa (és a hozzátartozó sigvec struktúra definíciója):

```
int sigvec(int sig, struct sigvec *vec, struct sigvec *ovec);
struct sigvec {
          int  (*sv_handler)(int, int);
          int  sv_mask;
          int  sv_flags;
```

Használatához deklarálnunk kell saját *vec* struktúraváltozót, ebben be kell állítanunk a *vec.sv_handler* tagot (SIG_DFL, SIG_IGN vagy saját kezelő címére). Nullázni kell a *vec.sv_flags* tagot (a *vec.sv_mask*-ot is, ha nem akarunk blokkolni), majd hívható a sigvec() függvény a *sig* szignál diszpozíciójára. . Hibátlan futás megcsinálja a diszpozíciót és az *ovec* változóban visszaadja a korábbi beállításokat (ne feledjünk tehát ekkor ovec-et sem deklarálni). A vec, illetve ovec lehet NULL pointer is: nem akarunk változtatni, csak lekérdezni (vec=NULL), változtatunk, de nem érdekel a régi beállítás (ovec=NULL), csak a kezelhetőséget akarjuk ellenőrizni (mindkettő NULL, és kezelhetetlen sig esetén a sigvec() hibázik).

Végül nézzük a POSIX szabványnak megfelelő diszpozíciót! Ehhez a sigaction() rendszerhívás szükséges. A rendszerhívás prototípusa és a hozzátartozó struktúra:

```
int sigaction(int sig, struct sigaction *act, struct sigaction
*oact);
struct sigaction {
     int sa_flags;
     void (*sa_handler)();
     sigset_t sa_mask;
    void (*sa_sigaction)(int, siginfo_t *, void *);
}
```

A "módszer" hasonlít az előzőkhöz: deklarálnunk kell saját *act* és *oact* struktúraváltozókat. Az *act* megfelelő tagjait be kell állítani. Mindenképp beállítandó az *sa_handler* tag (SIG_DFL, SIG_IGN vagy saját kezelő címe), és az *sa_flags* tag (nullára, de nézd még a man-ban a lehetőségeket). (Az *sa_mask* tagot csak blokkolás esetén állítjuk, de erre vannak más rendszerhívások, pl. sigemptyset(). sigaddset(), sigprocmask() stb.). Ezek után hívható a sigaction(). Normális lefutás esetén megtörténik a diszpozíció, és az oact-ban visszajön a korábbi beállítás. Természetesen, itt is használhatók a az act és oact paraméternek a null pointerek.

Szignál generálás, postázás; a kill() és a sigsend()

Szignálokat "hardveresen" is generálhatunk, vagy hardveresen is generálódhatnak, de szükségünk lehet szignálok küldésére is. Egy processz kontrol terminálján kiadott ctrl/break generálja neki a SIGINT szignált, a terminál vonal megszakadása a SIGHUP szignált, gyermekének megszűnése (vagy "stoppolás") a SIGCLG szignált. A SIGALRM előidézhető az alarm() rendszerhívással.

A szignál "postázás" legegyszerűbben a *kill* burok paranccsal, vagy a *kill()* rendszerhívással valósítható meg. A kill parancs használatát tanuljuk meg (pl. a manual-ból). A kill() rendszerhívás prototípusa (a szükséges a types.h include fájl is):

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

Feladata (bár a neve "öld meg"): a *pid* azonosítójú processznek (vagy processz csoportnak) küldj *sig* sorszámú szignált. Természetesen a SIGKILL *sig* is küldhető: az valóban megöli a címzettet, mert az nem kezelhető le csak default módon. Alapvetően tehát a *kill* a processzek közti kommunikáció, a szinkronizáció eszköze, hiszen a küldött szignálok egy részét a címzett lekezelheti.

Jegyezzük meg: a processzek valós vagy effektív tulajdonosainak meg kell egyezniük (kivéve: superuser bármelyik processznek küldhet szignált). Ha a *pid==-1*, akkor a *minden processz megkapja a szignált, melyeknek valós tulajdonosa egyezik a küldő effektív tulajdonosával.*

Visszatérési értéke: Sikeres végrehajtása esetén 0 értékkel tér vissza. Különben a visszatérése -1, és az *errno* változó beállítódik.

A *sigsend()* rendszerhívás prototípusa:

```
int sigsend(idtype_t idtype, id_t id, int sig);
```

Az *idtype* és *id* paraméterektől függően kiválasztott processz(ek)nek küldi a *sig* szignált.

Ha az

idtype == P_PID; az id azonosítójú processznek;

idtype == P_PGID; az id csoportjába tartozó valamennyi processznek;

idtype == P_PUID; az effektív uid-ű processznek;

idtype == P_GID; az effektív gid-ű processzeknek;

idtype == P_ALL; az id-től függetlenül valamennyi processzek (nézd még a manual-t!).

A pause (), sleep(), usleep() és alarm () rendszerhívások

Ha szignálozással akarunk processzeket szinkronizálni, hasznosak ezek a rendszerhívások.

```
int pause (void);
```

Feladata: blokkolja a hívó processzt, amíg az nem kap valamilyen szignált. Miután megkapja a szignált, a pause -1-gyel, errno=EINTR-rel tér vissza. Ha egy szignált saját kezelőre (lehet

az pl. egy ne_csinalj_semmit()) diszponáltunk, a pause()-val való blokkolásból egy neki küldött megfelelő szignál kibillenti, továbbengedi ...

A sleep() rendszerhívás az argumentumában adott másodpercig, az usleep() argumentumában adott másodpercnél kisebb egységig (mivel rendszerfüggő az időkezelés, nézd meg a manualban a sleep() családot) blokkol. Tulajdonképpen nem a szignálozáshoz tartoznak a sleep-ek, de hasznosak a szinkronizációs ütemezésekben, azért említjük meg őket.

Végül megemlítjük az alarm() rendszerhívást is.

```
#include <unistd.h>
unsigned alarm (unsigned sec);
```

Feladata: a hívó processz számára *sec* másodperc múlva SIGALRM sorszámú szignált generál. Ha a *sec* 0, a megelőzőleg kért alarm kívánság törlődik. Természetesen a befutó SIGALRM lekezeléséről gondoskodni kell.

A sigprocmask(), sigsuspend(), sigemptyset(), sigaddset(), sigismember() stb. rendszerhívások

A szignál kézbesítés a POSIX-ban hasonlít az IT kézbesítéshez oly módon is, hogy bizonyos szignálok blokkolhatók (hasonlóan az IT letiltáshoz). Létezik egy globális szignál maszk, ez definiálja, mely szignálok blokkoltak pillanatnyilag. Ez a maszk beállítható és lekérdezhető, a sigprocmask rendszerhívással. A maszk paramétere a *sigprocmask*, *sigsuspend* és *sigaction* rendszerhívásoknak. Tanulmányozzuk ezeket! Tanulmányozzuk a *sigsetop* rendszerhívás családot is (tagjai: *sigemptyset, sigaddset, sigdelset, sigfillset, sigismemberset*).

Különböztessük meg a szignál készletet és a szignál maszkot! Jó példa található a stevens/apue/signals/critical.c fájlban.

Az SVID-ban is lehet "blokkolni-elereszetni" szignálokat (de sohase keverjük a BSD vagy POSIX blokkolásokkal!). Itt használható a *sighold(int sig)* hívás blokkolásra, *sigrelse(int sig)* az eleresztésre. Használható egy speciális diszpozíciós hívás is: a *sigignore(int sig)* is. Általános szabály, hogy egy szignál "automatikusan" blokkolt, mialatt éppen a kezelője fut, és megszűnik blokkolása, miután abból visszatérés van.

3.3. Esettanulmányok, példák, feladatok

Lásd a /home/gi93/os/signals/*.c fájlok között a példákat.

Példa a signal(), alarm()és a pause() rendszerhívásokra alarm.c

Az. alarm_ado.c és alarmra_var.c párban haszánlható programok alarm_ado.c, alarmra_var.c

Gyakorló feladatok

- Írjon programot, melyben a SIGINT szignált lekezeli. A SIGINT bekövetkezésekor a terminálról (stdin) kérjen be sh parancsot, azt hajtsa végre, majd adja vissza a vezérlést. Gondoskodjon a programja terminálódásáról is.
- Szignál kezelője SIGALRM-re írja ki az időt. Példaprogramjában 10 másodpercenként keletkezzen SIGALRM.
- Processz családon belül szinkronizáljon szignál kezeléssel. Szülő processz küldjön pl. 10 másodpercenként jelzést (pl. SIGALRM, SIGINT, SIGUSR1, SIGUSR2) gyermekének. A gyermek a szignált futásának megváltoztatására használja. Specifikáljon erre feladatokat, programokon ezt mutassa is be!

- Vizsgálja, derítse fel a SIGFPE (floating point exeption) állapotot! Hogyan állhat elő? Mi a default kezelése? Átvehető-e a kezelése?
- Tanulmányozza az Irix manualban a wait, waitpid, wait3 rendszerhívásokat! Talál ott a SIGCLD szignálok kezelésére érdekes példákat (SVID, BSD és POSIX kezelésre: több gyermek processz exitje során el ne vesszenek a SIGCLD szignálok)
- Vizsgálja és derítse fel a szignál kezelés és a védelmi rendszer összefüggéseit. Mi a processz csoport? Mik a valós és effektív felhasználói azonosítók? Hogy lehet processz csoportnak jelzést küldeni?
- Írjon programot felhasználva a sigsetjmp és a siglongjmp rendszerhívásokat is. A programnak legyen A jelű és B jelű ciklusa. SIGALRM szignálok hatására kezdje el felváltva az A, illetve a B jelű ciklusát végrehajtani! Vagyis, ha éppen az A cikluson dolgozik, SIGALRM hatására hagyja azt abba, kezdje el a B-t. A B ciklust hajtsa végre mindaddig, amíg újabb SIGALRM érkezik s.í.t. (SIGALRM keltésére használja a kill parancsot!) A megoldás algoritmusa:

```
var: verema,veremb,futott;
main () {
       signal(SIGALRM,kezel);
       while(1)        {
         while(sigsetjmp(verema,1)  == 0)      {
               futott=a;
               fut A;
         }
         while(sigsetjmp(veremb,1) == 0        {
               futott=b;
               fut B;

       }
}
kezel()        {
       signal(SIGALRM, kezel);
       if(futott == a) siglongjmp(verema,1);
       else                              siglongjmp(veremb,1);
}
```

És még egy összefoglalás, a POSIX szignálkezelő rendszerhívások a következők:

```
sigaction (2)        - POSIX signal handling functions.
sigaddset (3)         - POSIX signal set operations.
sigdelset (3)         - POSIX signal set operations.
sigemptyset (3)       - POSIX signal set operations.
sigfillset (3)        - POSIX signal set operations.
sigismember (3)       - POSIX signal set operations.
sigpending (2)        - Return set of signals pending for
process.
sigprocmask (2)       - Alter and return previous state of the
set of blocked signals.
sigsetops (3)         - POSIX signal set operations.
sigsuspend (2)        - Atomically release blocked signals and
wait for interrupt.
```

4. VAX/VMS esettanulmány

Ez a fejezet szándékosan maradt el.

5. Folyamatok közötti információcsere (Inter Process Communication)

5.1. Alapfogalmak, elvek

Több folyamatból összetevődő alkalmazás esetén alapvető mechanizmus az IPC. Multiprogramozású rendszerekben abszolút természetes eszköz. Kooperáló processzek kommunikálnak, szinkronizálják futásukat. Több kommunikációs mechanizmus létezik, ezeket akár együttesen is alkalmazhatjuk.

Az absztrakt probléma kiindulási megfogalmazása a következő: processz legyen képes *üzenetet küldeni* processznek, processz legyen képes *üzenetet fogadni* más processztől.

A kiindulási megoldás: legyen *send(message)* és *receive(message)* rendszerhívás. A processzek között épüljön fel *kommunikációs kapcsolat* (link).

Kérdések jelentkezhetnek azonnal:

- Hogy lehet kommunikációs kapcsolatot létesíteni?
- Az összeköttetés (link) több mint két processz között is lehetséges?
- Hány kommunikációs kapcsolat lehet egy processz-pár között?
- Mi a link kapacitása? Változó vagy fix méretűek lehetnek az üzenetek?
- A kapcsolat egy, vagy kétirányú? És ez az irányitottság hogy van, ha több mint két processz van a kapcsolaton?

Az elvi megoldások, ezzel a típusok a következők:

1. Direkt vagy undirekt kommunikáció. Ki a közvetitő entitás tulajdonosa (kihez kötődik az entitás)?
2. Szimmetrikus vagy asszimmetrikus kommunikáció.
3. automatikus vagy explicit bufferelés.
4. Adatküldés (send by copy) vagy adathivatkozás küldés (send by reference).
5. Fix vagy változó méretű üzenetek.

5.1.1. Direkt kommunikáció. Alapeset.

Az ilyen típusú IPC-ben a kommunikálni kívánó processzek explicite ismerik egymás azonosítóit (pl. neveit). A szükséges rendszerhívások ekkor:

```
send(receiver-name, message)
receive(sender-name, message)
```

vagy ehhez hasonlók.

A kommunikációs kapcsolat jellemzői:

A link automatikusan megvalósul minden kommunikációs pár között.A processzeknek csak egymás azonosítóit kell ismerni a kapcsolat létesítéshez.

Mindíg két procesz között valósul meg a kapcsolat, és rendszerint csakis egy link létezik köztük.

Általában egyirányú, de lehet kétirányú is a kapcsolat (a kapcsolat szimmetrikus).

Asszimmetrikus direkt kommunikáció is lehetséges. ekkor a következőkhöz hasonló rendszerhívások kellenek:

```
send(receiver-name, message)
receive(id,message)
```

Itt a *receive* hívás fogad üzenetet bármely processztől, a küldő azonosítója az *id*-be kerül.

A kapcsolat jellemzője: több küldő és egy fogadó processz lehet.

5.1.2. Indirekt kommunkáció

Az üzeneteket *mailbox*, vagy *port* mechanizmusokon keresztül közvetetik. A mailbox (postaláda) absztrakt objektum, amibe a processz üzeneteket tehet, processz abból üzeneteket vehet ki. A mailbox-oknak van azonosítójuk. Ha két processz osztozik közös postaládán (mailbox-on, port-on) akkor azon át válthatnak üzeneteket.

A szükséges rendszerhívások:

```
create(mailbox)
destroy(mailbox)
share(mailbox)
send(mailbox, message)
receive(mailbox, message)
```

vagy ehhez hasonlók. Jellemzők:

- Link valósul meg processzek között, ha van közös postaládájuk
- A link több mint kér processz között is lehetséges. Változatok:
  - csak egy fogadó kapja meg a betett üzenetet, az, aki a versenyben előbb éri el;
  - akár több fogadó is megkaphat egy üzenetet;
  - címezhető, melyik fogadónak szól az üzenet.
- Több link is lehet processzek között.
- A link lehet egyirányú, vagy kétirányú is.

Változatok a postaláda kötődésére (binding):

1. A mailbox egy processzhez kötődik, úgy is mondhatjuk, a processz tulajdona. Ilyenkor rendszerint a tulajdonos processz a fogadó processz, az adja ki a create, destroy hívásokatt, a receive hívásokat. A feladó processzek a postaláda használói. A share hívással válnak használóvá, send hívásokkal tesznek üzeneteket a ládába. A legfontosabb jellemző: ha a tulajdonos exitál, a postaláda megszűnik. Erről a használóknak értesülniük kell.
2. A mailbox az operációs rendszer tulajdonában van, az operációs rendszerhez kötődik. Ilyenkor a create hívást kiadó processz (kreátor processz) nem válik tulajdonossá (exitálásával nem szünik meg a postaláda), de destroy+receive+send hozzáférési jogokat (privilégiumokat) kaphat. Jó volna, ha volna pass-priv(mailbox,proc-name) rendszerhívás is, esetleg demand-priv(mailbox) hívás: ekkor a kreátor átadhatja privilégiumait, ekkor a kreátor exitálása után megmaradó mailbox-ra igényt jelenthetne be egy processz. Alapértelmezés szerint ugyanis a kreátor processz beállíthatja a privilégiumait. Ha ezt a jogot átadhatja, akár nem kizárólagossággal, akkor az ilyen postaládán keresztül szimmetrikus kétirányú kommunikáció lehetséges több mint két processz között is! Bármelyik tehet be üzenetet, bármelyik kiolvashat üzenetet (a küldő akár visszaolvashatja az előzőleg betett üzenetét). Miután történhetnek programozási hibák, szükség lehet garbage-collection(mailbox) rendszerhívásra is, megszüntetni a használatlan postaláda objektumokat.

5.1.3. A bufferezés kérdései

A kommunikációs kapcsolat, a link kapacitását vizsgáljuk itt. A szemléletmódunk a következő: a linkhez üzenetek sora (queue) kapcsolódik. Alapvetően három mód lehet:

- Zéró kapacitású sor: a sor 0 hosszú lehet. Nem várakozhat a sorban üzenet, a küldő csak akkor küldhet, ha a fogadó egyuttal veszi is az üzenetet. "Randevú" szinkronizációnak nevezzük ezt az esetet.
- Korlátozott kapacitású a sor, maximum n üzenetet tárolhat átmenetileg a linkhez tartozó sor. Ha teli van a sor, a küldőnek várakoznia kell. Ha üres, a fogadónak kell várakoznia.
- Végtelen kapacitású sor: a küldőnek sohasem kell várakoznia.

Vegyük észre, az utóbbi két esetben a küldő nem tudja, vajon üzenete megérkezett-e. Szüksége lehet nyugtázás válaszüzenetre. Az ilyen kommunikációt nevezzük asszinkron kommunikációnak.

5.1.4. Speciális esetek

Az eddigi kategóriákba tisztán nem besorolható esetek is lehetségesek.

- A küldő elküldi az üzenetet, erre nincs korlátozás. Ha azonban fogadó nem veszi az üzenet, miközben a küldö újabb üzenetet küld, az előző üzenet elvész. E séma előnye: mivel nincs korlátozás a küldésre, hosszú üzeneteket nem kell megismételni. Hátrány viszont, hogy a biztos üzenetmegérkezéshez szinkronizáció szükséges.
- A az üzenetküdés halasztódik, amíg az előzőre válasz nem érkezik. Pl. a Thoth operációs rendszerben ezt a sémát valósítottk meg: a P processz üzenete küldve blokkolódik, amíg egy rövid nyugtázás választ nem kap.

5.2. IPC mechanizmusok felsorolása, összevetése

Először felsorolunk néhány mechanizmust, rövid ismertetéssel, majd összevetjük őket.

5.2.1. A klasszikus message rendszer

A Unix megvalósítással külön fejezetben foglakozunk.

A VAX/VMS ismeri a névnélküli (időleges) és a permanens *mailbox* koncepciót. Ezekre példákat a *mailado.c* és a *mailvevo.c* programokban láthatunk.

5.2.2. Csővezeték

Alapvető, korai Unix koncepció ez. Első megvalósításai fájlokon keresztül történt, a kommunikáló processzek szekvenciális végrehajtásával. A mai csővezeték megvalósítás jellemzői:

- Az implementáció lehet fájlokon keresztül, memórián át stb., a felhasználó számára transzparens módon megvalósítva.
  - FIFO jellegű, azaz
  - mindkét irányba mehetnek üzenetek;
  - amit beír egy processz, azt ki is olvashatja, kiolvasva az üzenet kikerül a csőből.
- A név nélküli cső csak származás szerint egy processz családba tartozó processzek között lehetséges. A nyitott cső nyitott adatfolyamnak (stream) számít, a gyermek processzek öröklik a leíróját (descriptor). A felhasználói felület a steram I/O függvényei.

A névnélküli cső (pipe) koncepciót mind a Unix, mind a VMS processzek használhatják.

A Unix-ban megvalósított a nevezett cső (named pipe) is (általában a fájl koncepciót is használva), erre példaprogramokat látunk az *ado.c* és a *vevo.c* programokban.

A csővezetékeken keresztüli kommunikáció szimmetrikus, aszinkron, adatküldéses, változó üzenetméretű jellegű. A névnélküli pipe-ok direkt, a nevezett pipe-ok indirekt, operációs rendszer tulajdon jellegűek.

5.2.3. Fájlokon keresztüli kommunikáció

Szekvenciális processzek közötti, régi, természetes módszer. Temészetesem párhuzamos processzek között is lehet fájlokon, adatbázisokon keresztüli kommunikáció.

Jellmezői: indirekt, operációs rendszerhez kötődő (operációs rendszer tulajdonú), szimmetrikus, aszinkron, adatküldéses, változó üzenetméretú kommunikáció.

Az eddig tárgyalt kommunikációs módszerek kézenfekvőek voltak. Természetesen vannak további módszerek is, melyek talán nem tűnnek olyan természetesnek.

5.2.4. Osztott memória (Shared Memory)

- Az operációs rendszerek természetszerűleg használják az osztott memóra koncepciót: gyakori, hogy a processz kontextusok kód része osztott memóriában van, nem ismétlődik processzenként.
- Az alkalmazások is használhatják! Mind a kód, mind az adat megosztás megvalósítható! Unix megvalósítással részletesebben is foglakozunk majd.

Jelemzés: indirekt, szimetrikus, adatküldéses, fix hosszú üzenetes, zéró bufferelt (randevút mégsem kívánó, mert többnyire operációs rendszerhez kötődő!)

5.2.5. A VAX/VMS logikai név rendszere

Miután a logikai nevek kifejtése a processzeken belül történik, alkalmasak kommunikációra. Itt nem a fájlokon keresztüli kommunikációra gondolunk, hanem arra, hogy lehetséges informálni egy processzt valamilyen tény fennállásáról, vagy éppen a fennálás hiányáról, azzal, hogy létezik-e, vagy sem egy logikai név.

Indirekt, operációs rendszer tulajdonú, asszimmetrikus, fix üzenethosszos, randevú nélküli zéró bufferelt.

5.2.6. Kommunkiáció a burok környezete (shell environment) segítségével

Bár a burok változók (szimbólumok) a parancsértelmező szintjén kifejtődnek (ezzel nem jutnak be a processzbe), az öröklődő környezet a processzek feldolgozhatják (emlékezzünk arra, hogy a fő program a neki adott *argumentumok* mellett az *environment*-et is lekérdezhetik. Lásd: *getenv* rendszerhívás). Ez is egy lehetséges módszer tehát.

Direkt, aszimmetrikus, zéró bufferelt, adatküldéses, változó üzenethosszos.

5.2.7. Esemény jelzők, szemaforok

Alapvetően a szinkronizáció és a kölcsönös kizárás mechanizmusai ezek, de a felhasználó is használhatja ezeket kommunikációs célokra. A VAX/VMS esemény jelzőiről már volt szó, a szemafor mechanizmust később részletesebben tárgyaljuk.

Indirekt és operációs rendszerhez kötődő, vagy direkt, szimmetrikus vagy aszimmetrikus, zéró bufferelt, fix üzenet-hosszú.

5.2.8. Szignálozás, szignálkezelés

Bár a szignál rendszer is a feltételek (condition exeptions) kezelésére, a szinkronizációra implementált rendszer, alkalmas processzek közötti kommunikációra. A lekezelhető szignálok kiküldése, azok lekezelése informáló hatású lehet. Továbbiakban nem részletezzük, de azt hiszem, érthető a dolog.

Direkt, egyirányú, végtelen kapacitású soros.

Két további kommunikációs mechanizmust csak megemlítünk: ezek a számítógépes hálózatok tárgyalásakor részletezhetők.

5.2.9. A BSD socket koncepció

A Networking által "kikényszerített" koncepció, de egyetlen host-on (a Unix domain) is alkalmas IPC mechanizmus.

5.2.10. A Remote Procedure Call mechanizmusok

A Networking és a Distributed Processing következménye, ma nagyon divatos koncepció.

5.2.11. Az IPC mechanizmusok összevetése

Az összehasonlítás során két szempontot vizsgálunk, esszerint minősítjük az egyes mechanizmusokat. Az egyik szempont a gyorsaság (speed), ami lehet lassú, közepes vagy gyors (slow, moderate, fast), a másik, az átvihető információmennyiség (amount of information), ami kicsi, közepes vagy nagy (small, medium, large) lehet (vegyük észre ezek fuzzy jellegét).

5.1. táblázat

Ssz.	Mechanizmus	Speed	Amount of inf.
1.	Messages	Fast	Medium
2.	Cső	Moderate/Slow	Medium (Large)
3.	Fájlok	Slow	Large
4.	Osztott memória	Fastest	Medium
5.	Logikai nevek	Fast	Small
6.	Environments	Fast	Small
7.	Flags, semaphores	Fastest	Very Small
8.	Signals	Fast	Small
9.	Sockets	Moderate*	Medium
10.	RPC	Moderate*	Medium

* A hálózattól függő

5.3. A Unix üzenetsor (message queue) és osztott memória (shared memory) rendszere

5.3.1. IPC - messages

Az SVID Unixokban a processzek tudnak ugyanazon a gazdagépen (hoston)

- készíteni üzenetsorokat (message-queue) (msgget() rendszerhívással),
- küldeni üzeneteket a sorokba (msgsnd() rendszerhívással),
- kiolvasni üzeneteket a sorokból (msgrcv() rendszerhívással),
- vezérelni a sorokat (msgctl()rendszerhívással).

A *message-queue* mailbox jellegű, az OS által ismert, az OS-hez kötődő objektum. Az

```
$ ipcs
```

paranccsal jellemzői lekérdezhetők. (Az ipcs parancs a nemcsak az üzenetsorokról, hanem az osztott memória- és a szemafor objektumokról is ad jelentést.

Kérdések merülhetnek fel bennünk:

1. Hogyan készíthetünk üzenet sort? Ezt csak a készítő processz ismeri, vagy más processzek is? Hogyan azonosítják a processzek a sorokat, ha már vannak? Meddig él egy üzenet sor? Megszüntethető? A védelmi problémákat hogy kezelik?
2. Hogyan írhatunk üzeneteket már ismert, azonosított sorba? Csak a készítő írhat bele? Ha nemcsak az, akkor más processz hogyan azonosíthatja? Írásengedélyek, hozzáférések hogyan állnak? Milyen üzenettípusok lehetnek?
3. Hogyan informálódhatunk üzenet sorokról? Létezik már? Hány üzenet van benne? Kezelhetjük?
4. Egy processzben üzeneteket akarok olvasni. Ehhez azonosítanám. Mi legyen, ha nincs benne üzenet? Várjak, exitáljak?

Általános tudnivalók:

- msgget() üzenetsor készítés, azonosítás (msgid az azonosító),
- msgctl() létező msgid-jű sort lekérdez, átállít, megszüntet, kontrollál,
- msgsnd() azonosított sorba adott típusú, adott méretű üzenetet tesz,
- msgrcv() azonosított sorból adott típusú üzenetet vesz. Amit kivett, azt más már nem érheti el!

Tanulmányozzák a man segítségével a fent rendszerhívásokat! Szükséges beleértett állományok (tanulmányozzák ezeket is! :

- #include <sys/types.h>
- #include <sys/ipc.h>
- #include <sys/msg.h>

Üzenetsor készítés, azonosítás

- Minden üzenetsornak van azonosítója és kulcsa (ID|msgid, key). Új üzenetsor készítése során kell választania kulcsot, meglévő üzenetsor azonosítására (asszociálás) használni kell a kulcsát.
- Az üzenetsoroknak van tulajdonosa, csoporttulajdonosa (uid/gid tartozik hozzá): annak a processznek az uid/gid-je, amelyik készítette. A kontrolláló msgctl() rendszerhívással a tulajdonosságok (a védelmi korlátozásokat figyelembe véve) átállíthatók. (Figyelem! Az üzenetsor az operációs rendszerhez kötődik, de az itteni *tulajdonosság* a Unix-ok szokásos f*ájl-tulajdonossági* kategóriával egyezik! A hozzáférések is a fájlhozzáférési kategóriáknak megfelelő.).
- Vannak írás/olvasás engedélyezések is a tulajdonossági kategóriákra, és vannak
- korlátai (max méret, max üzenetszám, üzenet max méret stb), továbbá
- paraméterei.

Mielőtt továbbmegyünk, nézzünk egy egyszerű esetet. Tételezzük fel, hogy készítek egy üzenetsort, és teszek bele két üzenetet az alábbi programocskával: msgcreate.c

Ezután a $ ipcs valami ilyet ad:

```
indvd 12% ipcs
IPC status from /dev/kmem as of Wed Sep 7 18:11:04 1994
T     ID     KEY        MODE       OWNER   GROUP
Message Queues:
q     50 0x0009fbf1 --rw-rw-rw-   vadasz   staff
Shared Memory:
m      0 0x000009a4 --rw-rw-rw-     root     sys
Semaphores:
indvd 13%
```

Az ipcs eredménytáblázatán a T oszlop a processzközti kommunikációs mechanizmus típusát mutatja: a q az üzenetsorokat jelzi (fenti példában csak egy üzenetsor van), az m az osztott memória objektumokat (ebből is csak egy van itt). A T oszlopban lehetne még s is: ez szemafort jelezne. Az ID oszlop a megfelelő objektum (üzenetsor, osztott memória, szemafor) azonosítóját, a KEY oszlop a kulcsát, a MODE oszlop a fájlrendszerbeli szokásoknak megfelelő hozzáféréseket, az OWNER ill. GROUP oszlopok a tulajdonosságokat mutatja.

Ha lefuttatjuk a msgcreate.c-ből készült processzt, megnézhetjük a párját: msgrcv.c Futtassuk ezt is. Ha most újból kérek $ ipcs -t, azt kell kapjam, mint amit fönt is kaptam. Az egyszerű esettanulmányhoz tartozik még egy kis program: msgctl.c. Ez megszünteti az üzenetsort, "takarít".

Ha tudom, hogy létezik 50-es azonosítójú, és 0x009fbf1-es kulcsú üzenetsor, akkor azt más progamból is megszüntethetem beírva az alábbi rendszerhívásokat:

```
msgid = msgget(0x0009fbf1,NULL);
msgctl(50,IPC_RMID, NULL);
```

Vagy az alábbi két rendszerhívással:

```
msgid = msgget(0x0009fbf1,NULL);
msgctl(msgid,IPC_RMID, NULL);
```

Takarítani lehet még az ipcrm burokparanccsal is. Miután ennek paraméterezése rendszerfüggő, tanulmányozzák a parancsot az adott rendszer on-line manuel-jében!

A renszerhívások prototípusai

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
   int msgget(key_t key, int flag);
   int msgsnd(int msqid, const void *msgp, size_t msgsz, int
msgflg);
   int msgrcv(int msqid, void *msgp, size_t msgsz, long
msgtyp, int msgflg);
   int msgctl(int msqid, int cmd, .../* struct msqid_ds *buf
*/);
```

A kreáló/asszociáló hívás:

```
id=msgget(key, flag);
```

A rendszerhívás célja: kreáljon üzenetsort, vagy azonosítson (asszociáljon rá) meglévő üzenetsort, hogy azt később használhassok. Az üzenesorhasználat első rendszerhívása tehát ez

kell legyen. Az msgget() hívás a key kulccsal azonosított üzenetsor azonosítójával tér vissza. Ezt az azonosítót használhatjuk majd a későbbi üzenetsorkezelő rendszerhívásokban.

Ha új üzenetsort akarunk kreálni, a key kulcsnak válasszunk egy olyan értéket, amilyen kulcsú üzenetsor nem létezik a rendszeren. Legyen tehát egyedi. Ha asszociálni akarunk, akkor pedig éppen a kívánt kulcsot állítsuk be!.

A flag-be beírhatjuk a "hozzáféréseket". A flag=0664 pl. azt jelenti, hogy mind a tulajdonos, mind a csoporttulajdonos prorocesszel írhatják és olvashatják az üzenetsort, a többiek csak olvashatják. Asszociáláskor nem adhatunk meg "nagyobb" jogokat, mint amit a készítéskor előírtunk. A flag-be bitenkénti vaggyal beírhatjuk az IPC_CREATE makrót is: flag=0666 | IPC_CREATE; Ekkor, ha nem volt még meg ez az üzenetsor, elkészül, ha megvolt, asszociálunk rá.

Az msgget() rendszerhívás hiba esetén negatív értékkel tér vissza (legtöbb rendszeren -1-gyel) és az errno változót vizgálhatjuk, mi is volt a hiba oka. Ha a flag-be nem írtuk be az IPC_CREATE-ot, akkor csak asszociálunk, a visszatérési érték vizsgálatával így kideríthetjük, létezik-e adott kulcsú üzenetsor.

Az üzenetsorba író rendszerhívás

```
int msgsnd(int msqid, const void *msgp, size_t msgsz, int
msgflg);
```

Itt az msqid az msgget()-tel kapott azonosító. Az msgp pointer az üzenetet tartalmazó msgbuf struktúrára kell mutasson. Deklaráljunk tehát (és foglaljunk memóriát) struct msgbuf változót (pointert), amibe az üzenet elküldése előtt azt "összeállíthatjuk". Ennek a struktúrának tagjait a sys/msg.h fájlból kivehetjük, van egy long mtype és egy char mtext[] tagja. Az mtype pozitív egész lehet, az üzenet típusát jelzi, és az üzenetet vevő processz használhatja majd az üzenetek "válogatására". Be kell állítanunk a típust. Az mtext tömbbe írhatjuk az üzenet-testet (legyen elegendő hely itt!) az msgsz által meghatározott hosszon (írjuk be az üzenetet, utána állítsuk be az msgsz változót a hosszának megfelelően. Végül az msgflg-be beírhatjuk az IPC_NOWAIT makrót, vagy 0-t. IPC_NOWAIT-tel nem blokkolódik a hívó, ha az üzenetsor "megtelt" (akár mert meghaladtuk a lehetséges üzenetszámot, akár mert meghaladtuk az üzenetsor hosszot), hanem rögtön visszatér a hívás (persze, nem küldi az aktuális üzenetet).

Az üzenetvétel

```
int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,
int msgflg);
```

Az msqid-vel azonosított sorról veszi az első üzenetet és elhelyezi az msgp pointer által mutatott felhasználó által deklarált struktúrába. Az üzenet ezzel "kikerül" az üzenetsorból. a sor "átrendeződik". Az msgp pointer az üzenetet fogadó msgbuf struktúrára kell mutasson. Deklaráljunk neki struktúraváltozót (pointert és helyet). Az msgsz csonkolja a vett üzenetet, állítsuk tehát elegendő hosszra. Az msgtyp szerepe a következő: ha 0, akkor az első üzenetet veszi. Ha pozitív érték, az első ilyen típussal elküldött üzenete veszi. Ha negatív, az abszolut értékével egyező vagy kisebb típusúakból a legkisebb típusú első üzenetet veszi. Az msgflg specifikálja, mi történjen, ha kívánt típusú üzenet nincs a sorban. Ha IPC_NOWAIT-et állítunk, a hívás azonnal visszatér (negativ értékkel persze), különben a hívó blokkolódik, míg a megfelelő üzenet meg nem érkezik a sorba (vagy mig a sor meg nem szünik). (Lehetséges processzek szinkronizálása: várakozás míg üzenet nem érkezik). Normális visszatérés esetén a visszaadott érték az üzenettestből átvett bájtok száma (tehát nem negatív érték).

Az üzenetsor kontroll

```
int msgctl(int msqid, int cmd, .../* struct msqid_ds *buf */);
```

Változatos kontrollálási lehetőséget biztosító hívás az msqid üzenetsorra. A cmd különböző "parancsokat" tartalmazhat, az elmaradható, vagy NULL pointerrel helyettesithető buf pedig a felhasználói címtartományban lévő (felhasználó által deklarált változó) struktúra pointere. Ez a struktúra a következőképpen deklarált (vö. sys/msg.h, illetve a struct ipc_perm deklráció a sys/ipc.h-ban található):

```
struct msqid_ds {
struct ipc_perm msg_perm;       /* operation permission struct
*/
struct msg      *msg_first;     /* ptr to first message on q
*/
struct msg      *msg_last;      /* ptr to last message on q */
ulong_t         msg_cbytes;     /* current # bytes on q */
ulong_t         msg_qnum;       /* # of messages on q */
ulong_t         msg_qbytes;     /* max # of bytes on q */
pid_t           msg_lspid;      /* pid of last msgsnd */
pid_t           msg_lrpid;      /* pid of last msgrcv */
time_t          msg_stime;      /* last msgsnd time */
long            msg_pad1;    /* reserved for time_t expansion
*/
time_t          msg_rtime;      /* last msgrcv time */
long            msg_pad2;       /* time_t expansion */
time_t          msg_ctime;      /* last change time */
long            msg_pad3;       /* last change time */
long            msg_pad4[4];    /* reserve area */
};
```

Tulajdonképpen arról van szó, hogy egy üzenetsor létrehozása esetén a rendszer a kernel címtartományában felépíti ugyan a sor attribútumait tartalmazó struct msqid_ds struktúrát, de mivel az kernel terület, a felhasználó közvetlenül nem láthatja: nem olvashatja, nem írhatja. Az IPC_STAT paranccsal viszont a kernel struktúra elemeit átmásolhatjuk a felhasználó által már olvasható buf -fal mutatott struktúrába. Az IPC_SET paranccsal pedig a felhasználói bufferből bizonyos elemeket átmásolhatunk a rendszer struktúrába: átállíthatjuk a tulajdonost, csoporttulajdonost, a hozzáféréseket, a szuperuser még a sor teljes hosszát is (nézd a man-ban!).

A legfontosabb cmd parancs azonban az IPC_RMID. Ekkor nem szükséges a buf paraméter (vagy legyen az NULL): megszünteti az üzenetsort.

A POSIX üzenetsorkezelés

Ediddigiekben az SVID üzenetsorkezeléssel kapcsolatos rendszerhívásokkal ismerkedtünk meg. Egyes rendszerek a POSIX üzenetsorkezelő rendszerhívásokat is ismerik. Tanulmányozzuk ezeket is! Látni fogjuk, hogy tulajdonképpen a szokásos fájlkezelő rendszerhívások (write(), read()) mellet üzenetsor nyitó-záró rendszerhívásokat kell megismernünk. Az érintett hívások a következők: mq_open(), mq_close() és mq_unlink().

Feladatok:

Tanulmányozzák a szükséges rendszerhívásokat, header állományokat!

Egy kiválasztott gépen készítsenek üzenetsort, amit a tanulócsoport olvashat, a tulajdonos írhat. Kulcsnak válasszanak egy kellemes hexadecimális számot, amit közöljenek néhány társukkal.

A társaik számára tegyenek be üzeneteket a sorba, mindenkinek más-más üzenettípus-számot adva. A tipus számát is közöljék a társakkal. Ez lehet pl. a névsorbeli sorszám!

A társak futtassanak az adott gépen üzenet-kiolvasó programot. Önkorlátozást kérünk, mindenki csak a saját típusát vegye ki!

A sor kreátora néha nézze meg, kiolvasták-e már az üzeneteket. Ha igen, megszünteteheti a sort.

Készítsünk csak magunknak üzenetsort, tegyünk bele néhány üzenetet, és egy másik process fordított sorrenben olvassa ki az üzeneteket! (Játszani kell a típusokkal! Nézd a manuel-t!) Végül töröljük s sort, hogy ne maradjon szemét magunk után.

Találjanak ki további feladatokat, amikkel az üzentesorok használatát gyakorolják! Az üzenetsorok, ha logout-tal kilépünk, megmaradnak, a bennük lévő üzenetekkel együtt. Mit gondolnak, ha a rendszer újra boot-olják, elvesznek-e a sorok?

5.3.2. IPC - shared memory (osztott memória)

A processzek ugyanazon gazdagépen osztott memória szegmenseket készíthetnek/azonosíthatnak (shmget() rendszerhívás), az osztott memória szegmenseket kontrollálhatják (attributumait lekérdezhetik, megszüntethetik, shmctl() rendszerhívás), illetve a processz virtuális címtartományára leképezhetik az osztott szegmenst (shmat() rendszerhívás), megszüntethetik a leképzést (shmdt() hívás).

A prototípusok

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
   int shmget(key_t key, size_t size, int shmflg);
   void *shmat(int shmid, void *shmaddr, int shmflg);
   int shmdt (void *shmaddr);
   int shmctl (int shmid, int cmd, /* struct shmid_ds *buf
*/);
```

A kreáció, asszociáció

Az shmget() rendszerhívás kreál/asszociál adott key kulcsú osztott memória szegmenst, aminek a bájtokban mért hossza size. Az shmflg szerepe nagyon hasonlít az msgget()-beli flag szerephez: rendelkezik a hozzáférésekről és a kreálás/asszociálás szétválastást oldhatjuk meg vele. Egy nem negatív shmid azonosítót ad vissza siker esetén (amit a további hívásokban használhatunk). Sikertelenség esetén negatív értékkel (többnyire -1-gyel) tér vissya (és az errno vizsgálható).

A leképzésés megszüntetése

A leképzésben (shmat() hívás) a szegmenst "rákapcsoljuk" a processz címtartományára. Egy általunk választott típusú pointerváltozó a hívás visszatérési értékét megkapja, és ezután az adott típusú adatszerkezet a pointerváltozó segítségével használható, a típusra vonatkozó korlátozásokkal. Ha karakter pointernek adjuk a visszatérési értéket, akkor az oszott szegmenst

karakerek tömbjének láthatjuk, ha integer pointernek, akkor integer tömbnek, ha egy struktúra mutatónak: a truktúrát képezzül az osztott szegmensre ...

Az shmat hívásban az első argumentum nyilvánvalóan érthető (az shmget visszatérési értéke). A második argumentum (shmaddr) azt szabja meg, milyen címtartományra képezzük a szegmenst. Ha shmaddr==NULL, akkor a rendszer által választott első lehetséges címtartományra történik a leképzés (magyarul, a rendszerre bízzuk a címeket, és a címtartományunk bővülni fog). Ha az nem NULL, akkor az shmaddr által adott cím és az shmflg-ben SHM_RMD-t beállítás vagy éppen nem beállítás együtt szabja meg a címtartományt (nézd a man-ban pontosan hogy is van).

Az shmdt() megszünteti a leképzést. Kiadása után az osztott szegmens nem "látható" tovább. ügyeljünk arra, hogy argumentumának nem a szegmens azonosítót, hanem a választott pointert (amin az attach után "látjuk" a szegmenst) kell írni. Siker esetén 0-val, hinóba esetén -1-gyel tér vissza.

A kontroll

Szerepe hasonlít az msgctl() szerepéhez, elsősorban az IPC_RMID paranccsal a szegmens megszüntetésére fogjuk használni. Nem részeletezzük, de itt is lekérdezhetők az attribútumok (kernel területről átmásolás a felhsználói területen lévő struct shmid_ds típusú buf-ba) illetve néhány attribútum (uid, gid, mode) "változtatható" (nézd a man-ban).

Gyakorló feladatok

Az osztott memória szegmensek az OS szempontjából IPC mechanizmusok. Azonosításuk a *message* ill. *semaphore* mechanizmusokéhoz hasonló, az *ipcs* shell paranccsal ezek is lekerdezhetőek stb., ezért a mintaprogramocskákban ezeket a programrészeket nem is magyarázzuk.

Nézzék a man segítségével a rendszerhívásokat, tanulmányozzák az *include* fájlokat.

- #include <sys/ipc.h>
- #include <sys/shm.h>

Három példaprogramot ismertetunk az osztott memória használatra:

- shmcreate.c egy választott kulccsal kreál/azonosít osztott memória szegmenst. Azonosítója: shmid, amit kiiratunk.
- shmctl.c az shmcreate.c-vel készitett osztott memória szegmens státusanak lekérdezésére, a szegmens megszüntetésére alkalmazható program. IPC_STAT parancs a státus lekérdezést, IPC_RMID parancs a megszüntetést kéri. A státusbol csak a szegmens méretét és annak a processznek azonosítóját irja ki, amelyik utóljára operált a szegmensen.
- shmop.c shmid-del azonosít osztott memória szegmenst. Ezután a segm nevű pointerváltozót használva a processz virtuális címtartomanyába kapcsolja (attach) a szegmest (shmat() rendszerhívás). Olvassa, irja ezt a címtartományt, végül lekapcsolja (detach) a shmdt() rendszerhívással).

6. Kölcsönös kizárás, kritikus szakasz (Mutual Exclusion, Critical Section)

6.1. Alapfogalmak, elvek, kívánalmak

A processzek közötti kapcsolat lehet együttműködés jellegű (kooperatio), ez processzközti kommunikációs kapcsolatokat (IPCs) kíván. Ha a kapcsolat konfliktusmentes erőforrásmegosztás, akkor az erőforrásra olvashatóság jellegű hozzáférést (reentrans kód, olvasható fájl stb.) kell biztosítani. A vetélkedés erőforrás kizárólagos használatára *kölcsönös kizárást*, a szinkronizálás *ütemezést* kíván. Most az utóbbiakkal foglakozunk.

Arról van szó, hogy közös erőforrásokért vetélkedő, együttműködő processzeknek lehetnek kódrészei, melyek futása alatt kizárólagosságot kell biztosítani, vagy amikre ütemezést kell megvalósítani.

A **kölcsönös kizárás** (Mutual Exclusion) fogalma: a közös erőforrásokért vetélkedő processzek közül egy és csakis egy kapja meg a jogot az erőforrás használatra, ennek a hozzárendelésnek módszerei, eszközei, technikái.

A **kritikus szakasz** (Critical Section) a folyamaton belüli kódrész, melyen belül a kölcsönös kizárást meg kell valósítani, vagy amire az ütemezést meg kell valósítani.

Belépési szakasz (entry section) a folyamaton belül az a kódrész, ahol kéri az engedélyt a kritikus szakaszba való belépésre, míg a **kilépési szakasz** (leave section) az a kódrész, ahol elhagyja a processz a kritikus szakaszt. A folyamatoknak természetesen lehetnek **nem kritikus szakaszaik** is.

A **holtpont** (deadlock) az az állapot, amely akkor következhet be, amikor két (vagy több) folyamat egyidejűleg verseng erőforrásokért, és egymást kölcsönösen blokkolják. Tegyük fel, hogy P folyamat kizárólagos használatra kéri az X és Y erőforrásokat, és azokat ebbena sorrendben kívánja használni. Ugyanakkor Q folyamat kéri az Y és X erőforrásokat, ebben a sorrendben. Ha P folyamat megszerezte az X erőforrást, Q folyamat pedig az Y-t, akkor egyi sem tud továbblépni. hiszen mindkettőnek éppen arra volna szüksége, amit a másik foglal: ez a holtpont helyzet.

Kívánalmak a probléma megoldásához

1. **Biztonsági** (safety) kívánalom: Valósuljon meg a kölcsönös kizárás: ha egy processz kritikus szakaszában fut, más processz ne léphessen be kritikus szakaszába. (Egyidőben csakis egy kritikus szakasz futhat.). Természetesen, ezalatt más processzek a belépési szakaszukat végrehajthatják (éppen az a fontos, hogy azon ne jussanak túl).
2. **Előrehaladási** (progress) kívánalom: általában nem kritikus szakaszban és nem belépési szakaszban futó processz ne befolyásolja mások belépését. Ha egyetlen folyamat sincs kritikus szakaszában és vannak processzek a belépési szakaszukban, akkor csakis ezek vegyenek részt abban a döntésben, hogy melyik fog végül belépni. Ráadásul ez a döntés nem halasztható végtelenségig.
3. **Korlátozott várakozási** (bounded waiting) kívánalom: ha egy processz bejelentette igényét a belépésre, de még nem léphet be, korlátozzuk ésszerűen azt, hogy egy másik processz hányszor léphet be. Egy processz se várakozzon a végtelenségig belépésre azért, mert egy másik újból bejelentve az igényét megint megelőzi.

4. **Hardver és platform** kívánalom: ne legyenek előfeltételeink a hardverre (sem a CPU-k típusára, számára, sem a sebességükre), a processzek számára, relatív sebességükre, az operációs rendszer ütemezésére stb.

Az absztrakt probléma felfogható mint egy **termelő-fogyasztó** (producer-consumer) probléma:

- Vannak termelő (producer) folyamatok, melyek terméket (item, message etc.) állítanak elő és behelyezik egy raktárba.
- Vannak fogyasztó folyamatok, melyet a termékeket kiveszik a raktárból és felhasználják.
- Van egy korlátozott termék-raktár (item pool, message buffer). A korlátozás vonatkozhat a raktár méretére (teli raktárba termelő nem rakodhat, üres raktárból fogyasztó nem vehet ki terméket: ez szinkronizációs probléma, ütemezést kíván). A korlátozás vonatkozhat a raktár használatára (pl. egyidőben több termelő nem használhatja a raktárt: egyetlen "berakó gép" van, vagy egyidőben csak egy folymat, akár termelő, akár fogyasztó használhatja a raktárt: egyetelen "be-kirakó gép" van stb.). A korlátozás a raktárhoz való hozzáférésre kölcsönös kizárási probléma.

A termelő-fogyasztó problémának ismertek a változatai, a klaszikus probléma, az író-olvasó probléma stb. Ezek különböző változatait fogjuk példaként bemutatni az egyes megoldásokban.

```
processz() {
  while (true) {
     nem-kritikus-szakasz1();
     belepes-kritikus-szakaszba();
       kritikus-szakasz();
     kilepes-kritikus-szakaszbol();
     nem-kritikus-szakasz2();
  }
}
```

6.1. ábra

```
process() {  // utemezett processz
  while (true) {
    nem-kritikus-szakasz1();
    belepes-kritikus-szakaszba();
      kritikus-szakasz();
    nem-kritikus-szakasz2()
  }
}
-----------------------------------
process() {  //  utemezo
   ...
   if ( van-utemezesre-varo)
      utemezz();
   ....
}
```

6.2. ábra.

Az alap sémák

Összefoglahatjuk most már a kölcsönös kizárás és a szinkronizáció alapvető programsémáit.

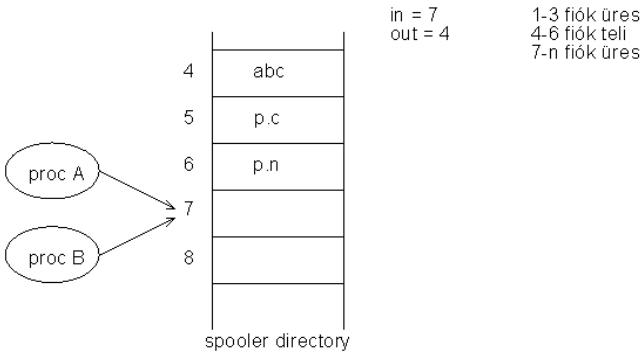
A kölcsönös kizárást megvalósító procesz(ek) struktúrája a 6.1, a szinkronizáció/ütemezés processzpárjainak az alap struktúrái a 6.2. ábrán láthatók.

6.2. Egy kevésbé absztrakt probléma: nyomtatás kimeneti munkaterületről

Multiprogramozási környezetben szokásos megoldás szerint a processzek, melyek nyomtatni akarnak, a nyomtatandó állományt (vagy annak nevét) egy kimeneti munkaterületre (spooler directory) helyezik. Egy szolgáltató processz - a printer daemon - ebből veszi a nyomtatandó anyagokat (vagy nevüket) és kezeli a nyomtató eszközt.

Tételezzük fel, hogy van {A, B, C, ...} *processz készletünk*, melyek nyomtatni akarnak. Van egy P *printer daemon* processzünk is. Van egy *spooler-directory*-nk is, *n* számú "fiókkal" , a nyomtatandó fájlok nevei kerülhetnek egy-egy fiókba (legyen n elég nagy, ettől a korláttól most eltekintünk.) Szükségünk van még két globális váltózóra: egyik az *out*, ami a következő nyomtatandó fájl nevét tartalmazó fiókra mutat a spooler directory-ban, másik az *in*, ami a következő szaban fiókra mutat.

Tételezzük fel a következő helyzetet, mikor is proc A és proc B egyidőben "nyomtatni szeretne" (6.1. ábra):



6.3. ábra. Két processz egyidőben nyomtatna dolgait.

Tegyük fel, hogy proc A "elkapja" az *in*-t, azt megjegyzi, de mielőtt beírná a fájl nevét és növelné az *in*-t, elveszik tőle a CPU-t, odaadva azt proc B-nek.

Proc B veszi az in-t, ami pillantnyilag 7. Beírja a 7. sorszámú fiókba a saját nyomtatandó fájlja nevét, növeli az *in*-t. Ezek után csinálja a saját dolgait.

Valamikor proc A visszakapja a vezérlést, folytatja, ahol abbahagyta: beírja a saját nyomtatandó fájlja nevét a 7. fiókba (felülírva ezzel a proc B nyomtatandóját!), majd növeli az in-t: ez már így 9 lesz. Láthatjuk, a 8. fiókba nem is került fájlnév! Ezzel a proc A is végzett a nyomtatásával, folytathatja saját dolgait.

Beláthatjuk: a proc B outputja sohasem jelenik meg így, ugyanakkor a 8. fiókban határozatlan kérés van a daemonhoz, hogy nyomtasson valamit. A kritikus szakaszok védelme nélkül gondok jelentekeztek, megsértettük a biztonsági kívánalmat.

6.3. Megoldások

6.3.1. Megszakítás letiltás (Disabling Interruts)

A megoldás lényege: a belépési szakaszban letiltunk minden megszakítást, a kilépési szakaszban engedélyezük azokat.

A megoldás hátránya: csak egyetlen CPU esetére jó (4. számú kívánalom megsértve) és a kritikus szakaszban bekövetkező hiba esetén előáll a *holtpont* (dead-lock) helyzet.

Ez a megoldás nagyon veszélyes. Néha a kernel kódban használják, de csak nagyon rövid és nagyon jól letesztelt kritikus szakaszokra.

6.3.2. Váltogatás

Ehhez a megoldáshoz szükséges egy *osztott turn* változó. Ez a változó nyomonköveti, azt mutatja, ki következik. Két processzen bemutatjuk a váltogatás lényegét. Íme a két processz pszeudókódja (6.4.ábra):

```
shared turn=i;

processz-i() {
 while (true) {
   nem-kritikus-szakasz1();
    while(turn!=i) nop();
     kritikus-szakasz();
    turn=j;
   nem-kritikus-szakasz2();
 }
}

processz-j() {
 while (true) {
   nem-kritikus-szakasz1();
    while(turn!=j) nop();
     kritikus-szakasz();
    turn=i;
   nem-kritikus-szakasz2();
 }
}
```

6.4.ábra.

Vegyük észre a kritikus szakaszba való belépés while ciklusát! Amíg teljesül a while feltétele, a vezérlés menete tevékeny várakozásban (busy waiting) a ciklusban marad! A nop() azt jelzi, ne csináljon semmit, azaz a ciklus magja egy no-operation instrukció.

A megoldás megsérti a 2. számú követelményt: a szigorú alternálás miatt egy processz egymás után kétszer csak akkor léphet a kritikus szekciójába, ha közben a másik is átfutott rajta. Azaz akkor is vár a belépésre, mikor a másik nincs a kritikus szakaszban: nem követi, hogy egy processz érdekelt-e vagy sem. A példában persze, sérül a 4. követelmény (csak két procsszre érvényes), bár a váltogatás kiegészíthető több processzre is.

6.3.3. Az érdekeltség nyomonkövetése

A megoldásban az *osztott erd* tömb jelzi, ki érdekelt a kritikus szakasz (a kizárólagos erőforrás) elérésében. Alább láthatók (6.5.ábra) a pszeudókódok (ismét 2 processzre, ami a 4. követelmény sérülését eredményezi, de a több processzre való kiterjesztés itt is megoldható lenne.

```
shared erd[N];

processz-i() {
 while (true) {
   nem-kritikus-szakasz1();
    erd[i]=true;
    while(erd[j]) nop();
      kritikus-szakasz();
    erd[i]=false;
   nem-kritikus-szakasz2();
 }
}

processz-j() {
 while (true) {
   nem-kritikus-szakasz1();
    erd[j]=true;
    while(erd[i]) nop();
      kritikus-szakasz();
    erd[j]=false;
   nem-kritikus-szakasz2();
 }
}
```

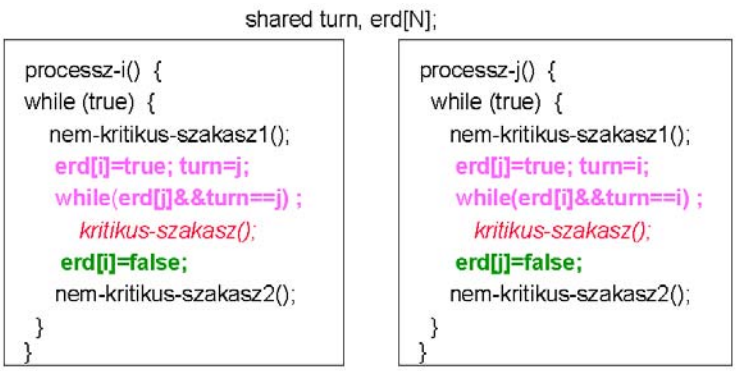
6.5.ábra.

Sajnos, a megoldás nem megoldás: a kölcsönös kizárás ugyan megvalósul, de könnyen kialakulhat holtpont (2. követelmény). Ha ugyanis mindkét processz körülbelül egyidőben bejelenti érdekeltségét, mindkettő tevékeny várakozásban marad a while ciklusában. Pusztán az érdekeltség figyelembe vétele semmiképp sem nem elegendő!

6.3.4. Egymás váltogatás az érdekeltség figyelembevételével

A "ki következik" (turn változó) és a "lock változó" (lásd később) koncepciók kombinációjaként Dekker (holland matematikus) ajánlott először jó megoldást: ez az irodalomban Dekker algoritmusaként ismert.

Peterson még egyszerűbb módszert ajánlott 1981-ben, ami elavulttá tette Dekker algoritmusát. Nézzük Peterson megoldását, ami lényegében egymás váltogatása az érdekeltség figyelembevételével. Az érdekeltség figyelembevétele javítja a puszta váltogatás hibáját. Most már csak a 4. követelmény sérül (csak 2 processzre érvényes a bemutatott példa, de belátható a kiterjeszthetőség sok processzre, és számít a megoldás a processzek korrekt ütemezésére). Nézzük a kódokat (6.6.ábra).



6.6.ábra.

Képzeljük el azt a helyzetet, amikor egyik processz sincs kritikus szakaszában és az i-edik be akar lépni. Beállítja saját érdekeltségét, és a másik következőségét. Utána ráfut a tevékeny várakozás while ciklusára (itt a NOP-ot üres utasítás képviseli helyszűke miatt), és mivel a másik nem érdekelt, továbbjutva beléphet a kritikus szakaszába. Ha most a j-edik szeretne belépni, a while-jában meg kell várnia, míg az első a kritikus szakaszából kilépve megszünteti az érdekeltségét.

Ha mindkét process kb. egyidőben lépne be, akkor mindkettő bejelnti érdekeltségét, majd mindkettő bejegyzi a turn-be a másik számát. Amelyiknek ez utoljára sikerült, az vesztett! A másiknak ugyanis a while-ja "sikertelen" lesz, azaz tovább léphet. A turn-be utoljára író viszont a while tevékeny várakozó ciklusában marad. Nem jelent gondot az sem, ha az egyik processznek csak az érdekeltség beírás sikerül, utána pedig elveszik tőle a CPU-t. Ekkor ugyan a másik "blokkolódik" a tevékeny várakozása ciklusában, de előbb utóbb visszakapja a CPU-t az egyik (4. Követelmény szerint erre biztosan nem szabadna számítanunk), és beállítva a turn-öt továbbengedi a másikat.

6.3.5. A Bakery algoritmus: a sorszámosztás

Az algoritmus a nevét egy vásárlókkal zsúfolt pékségben a sorszámosztásos kiszolgálás rendjétől kapta (hogy miért pont pékség? Ki tudja?) Később látni fogjuk a sorszámosztó-eseményszámláló mechanizmust, annak az alapgondolatát valósítja meg a bakery algoritmus, meglehetősen egyszerű eszközökkel. (Kedves olvasó! Vajon észreveszi-e bakery algoritmus és a sorszámosztó-eseményszámláló különbségét?).

Az elgondolás szerint belépve a boltba minden vásárló kap egy sorszámot, és a legkisebb sorszámmal rendelkezőt szolgálják ki először.

```
shared erd[N]={false, ...};          // sorszamhuzasban erdekelt
 shared s[N]={0, ...};               // a sorszam
processz-i () {
 while (true) {
   nem-kritikus-szakasz1();
   erd[i]=true;   s[i]=max(s[0],s[1],...,s[n-1])+1;    erd[i]=false;
   for(j=0;j<N;j++) {
        while(erd[j])  nop();      // az i-edik nem erdekelt, de ...
        while(s[j]!=0&&(s[j] < s[i] || (s[j]==s[i] && j < i)))  nop();
   }
    kritikus-szakasz();
   s[i]=0;
   nem-kritikus-szakasz2();
 }
}
```

6.7.ábra.

Az alábbi pszeudókód (6.7.ábra) az i-edik processzt mutatja. Vegyük észre az osztott adatstruktúrákat (erd, s), azt, hogy az érdekeltség itt nem a kritikus szakaszra, hanem a sorszámhúzásra vonatkozik, valamit ezek inicializálását!

A kritikus szakasza előtt a processz bejelenti érdekeltségét a sorszámhúzásban, majd sorszámot húz. Sajnos, miután az s[i] értékadás nincs "védve", nem garantált, hogy két processz ne kaphassa ugyanazt a sorszámot. Ha ez előfordulna, akkor az algoritmus sze-

rint a "kisebb nevű" (kisebb pid-ű) lesz a kiszolgált (lásd a második while-ban a j < i relációt). Miután a processz nevek, a pid-ek egyediek és köztük a kisebb mint rendezési reláció egyértelmű, az algoritmusunk determinisztikus lesz. A sorszámhúzás után a processz törli az érdekeltségét a sorszámhúzásra. Ezután a for ciklusban minden processzt vizsgál! Az első while-ban saját maga miatt nem várakozik, de találhat más processzeket, akár magánál "kisebbet" is, akik éppen sorszámot húznak, ekkor várakozik.Ha az ütemezés korrekt (4. követelmény?), előbb-utóbb túljut ezen. A második busy waiting while ciklus "megfogja", ha van nála kisebb sorszámú processz (persze a 0 sorszám nem számít, márpedig mind a kiszolgáltak, mind a boltba be sem lépők ilyenek), vagy ha az azonos sorszámú nála kisebb pid-del rendelkezik. A kritikus szakaszból való kilépés egyszerűen a sorszám nullázásával jelezhető.

6.3.6. Zárolásváltozó használata

Adott egy *osztott lock* változó, kezdeti értéke false, ami tesztelhető és beállítható (false és true értékeket vehet fel). Ha egy processz be akar lépni kritikus szakaszába, teszteli a lock változót. Ha az false, beálítja true-ra, és belép a kritikus szakaszba. Ha az beállított, akkor ciklusban tesztelve a változót (busy waiting), megvárja, míg valaki lenullázza. Fontos követelmény, hogy a tesztelés és beállítás között a vezérlést ne vehessék el a processztől, különben megsértjük az 1. számú követelményt (mint a spooler példában is).

A TSL instrukció

Sok processzornál, különösen azokon, melyeket eleve többprocesszoros rendeszerkhez terveztek, létezik egy atomi *test-and-set-lock* (TSL) instrukció, ami a következőképpen dolgozik. Behoz egy memóriacímen lévő értéket egy regiszterbe, és egy true értéket tesz a memória rekeszbe, majd a behozott értékkel visszatér. A regiszterbe hozás, a letárolás müveletére és visszatérésre garantált, hogy "oszthatatlan", egyetlen más processzor sem érheti el a memória rekeszt, amíg az instrukció végre nem hajtódik. Általában a TSL-t végrehajtó CPU blokkolja a buszt, amíg a TSL-t végrehajtja. A 6.8.ábrán megadjuk a kritikus szakasz védelmét biztosító kódrészletet és a TSL instrukció "forgatókönyvét" is. Figyeljük a kritikus szakasz végét: a zárolásváltozó szokásos értékadással (MOVE instrukcióval) "tisztítható".

```
shared lock=false;

processz-i() {
  while (true) {
    nem-kritikus-szakasz1();
     while(tsl(&lock)) nop();
     kritikus-szakasz();
     lock=false;
    nem-kritikus-szakasz2();
  }
}
```

```
int tsl(*lock) {
    register int temp;
    temp=*lock;
    *lock=true;
    return temp;
} // ez megszakithatatlan
  // instrukcio kell legyen
```

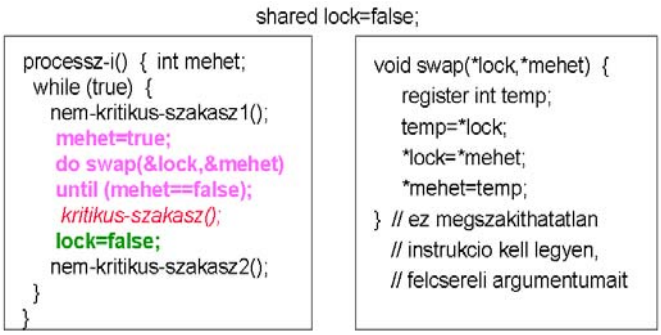
6.8.ábra.

Ha a zárolásváltozó értéke true, valamelyik processz a kritikus szakaszában van. Egy másik processz tevékeny várakozás while TSL-je ugyan rendre átírja a true értéket, de nem tud továbblépni, egészen addig, míg a kritikus szakaszból ki nem lép az érintett processz

A hátrány: megsértettük a 4. követelményt, kikötésünk van a hardverre vonatkozólag (nincs minden CPU-nak TSL instrukciója). És még egy probléma: nem feltétlenül teljesül a 3. követelmény: igaz, extrém ütemezésnél, de előfordulhat, hogy egy processz a kilépési szakaszát (lock=false;) végrehajtva, újból nagyon gyorsan eljutva a belépési szakaszába többször megelőz egy másikat.

A SWAP instrukció

Némely processzornak van oszthatatlan "cserélő" (swap) instrukciója. A zárolásváltozó használat ezzel is megoldható (lásd a 6.9.ábrán a pszeudókódot, benne a swap forgatókönyvét is).



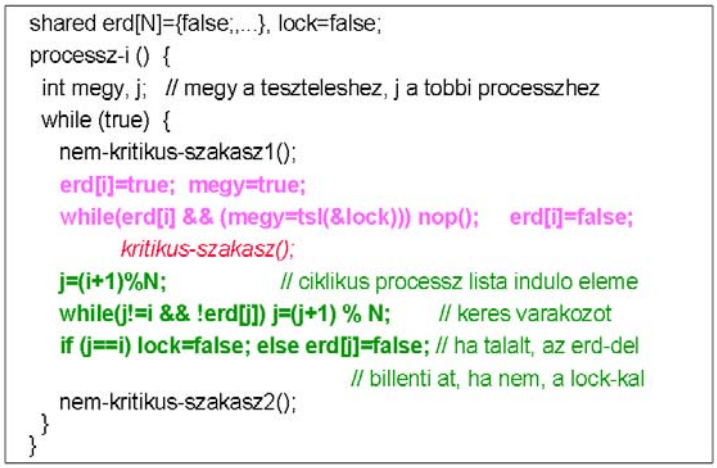
6.9.ábra.

Látható, hogy itt a beállítás (swap) ugyan elválik a teszteléstől (until), közben elvehetik a CPU-t a processztől, de ez nem jelent gondot. Figyeljünk fel arra, hogy a mehet minden processznek saját változója, a swap-ben kap igazán értéket (a tesztelés előtt), ugyanakkor a közös lock is kap értéket. A kritikus szakasz végén a lock a szokásos értékadással kapja a false-t, ezzel esetleg továbbenged más processzt a do until ciklusából. Itt is sérül a 4. követelmény (nincs minden CPU-nak atomi swap-je), és sérülhet a korlátozott várakozási követelmény is.

Korlátozott várakozási követelményt is kielégítő zárolásváltozóhasználat

A fent bemutatott zárolásváltozót használó algoritmusok javíthatók, hogy a 3. követelményt teljesítsék. A TSL instrukciós változatot egészítjük ki, ezen mutatjuk be az elgondolást. Az elgondolás lényege az, hogy nemcsak az operációs rendszer ütemezője ütemez, hanem maguk a belépési és kilépési szakaszok is, azaz a kritikus szakaszokért vetélkedő processzek is.

Az osztott lock zárolásváltozón kívül használjuk az erd tömböt is, false értékekkel inicializálva. Az erd elemei itt azt jelzik, hogy egy-egy processz a belépési szakaszában van. Tekintsük meg az algoritmust (6.10 ábra).



6.10.ábra.

A processz csakis akkor léphet be kritikus szakaszába, ha erd[i]==false, vagy ha megy==false. Belépési szakaszában az erd[i]-t true-ra állítja, az első igénybejelentésekor tehát valószínűleg a megy dönt. A megy-et pedig csak a tsl állíthatja false-ra. A megy garantálja a kölcsönös kizárást. Az előrehaladás is biztosított, hiszen a kilépési szakaszában egy másik procesz vagy a zárolásváltozót, vagy a mi processzünk erd elemét tisztítja. Bármelyiket is billenti, a mi processzünk továbblép a kritikus szakasza felé, és már nem blokkolja saját magát. A korlátozott várakozás pedig a következőképp biztosított: a kilépési szakaszában mindenprocessz ciklikus rendben (i+1, i+2, ..., n-1, 0, 1, ...,i-1) keres belépésre várakozó más processzt (amelyikeknek az erd eleme true), és az elsőt ebben a rendben kiválasztva az erd elemének billentésével "továbbengedi" ezt és csakis ezt a belépési szakaszából. Ha nem talál várakozót, akkor zárolásváltozó false-ra állításával engedélyezi (akár saját magának az újbóli) belépést. A ciklikus végignézés persze

véges, ha az itteni while-ban j==i, akkor már nem érdemes nézni, van-e belépésre várakozó processz.

6.3.7. A tevékeny várakozás (busy waiting) hátrányai

A megszakítás letiltást kivéve minden eddigi megoldásnál a belépési szakaszban egy-egy processz ciklusban várakozott, hogy végre beléphessen. Sokszor csak egy NOP-ot hajt végre, majd újra tesztel, mindenesetre használja a CPU-t, hátráltatva más processzeket.

Felléphet a *priority inversion* probléma is: képzeljük el, hogy van két processzünk, processz H magas prioritással, processz L alacsonnyal. Az ütemezési szabályok szerint a magasabb prioritású processz mindíg megkapja a CPU-t, ha igényli. Egy bizonyos pillanatban legyen L a kritikus szakaszban, ezalatt H váljon futásra kész állapotúvá, a *busy waiting* ciklusa előtt kevéssel. Megkapva H a CPU-t, tesztel és vár ciklusban, és mivel magasabb a prioritása, nem engedi szóhoz jutni L-t, hogy az kijusson a kritikus szakaszából, felszabadítva H-t a várakozó ciklusból. Nyilvánvaló holtponthelyzet alakult ki. Megoldás persze a dinamikus prioritás állítás, de pl valós idejű processzeknél az nem valósítható meg. Más megoldás is kell!

A CPU idő vesztegetése helyett alkalmazhatunk *sleep* és *wakeup* rendszerhívás párokat! A *busy waiting* helyett a várakozó processz *sleep* hívással blokkolt állapotba megy, ott is marad, míg egy másik processz a wakup hívással fel nem ébreszti. Ekkor persze újra kell ellenőriznie a kritikus szakaszba való belépés lehetőségét, de addig is nem vesztegeti a CPU időt. Ilyen megoldásnál persze ügyelni kell arra, hogy valaki kiadja a wakup-ot, és a wakup szignál ne vesszen el!

Ismerkedjünk meg még egy fogalommal! Szakirodalomban használhatják a *spinlock* kifejezést. Ez valójában busy waiting-et (spin) használó szemafor (a szemafort lásd alább!) Előnye az, hogy nem kell contecxt switch, mikor egy processz várakozik egy lock-on. Ha rövidek a várakozások, többprocesszoros rendszeren előnyös lehet.

6.3.8. A szemaforok

1965 körül Dijkstra javasolta a szemafor (semaphore) mechanizmust a kölcsönös kizárások megoldására [E.W. Dijkstra: Cooperating Sequential Processes in F. Genuys ed. Programming Languages, academic Press, NY, 1968, pp43-112.]

A **klasszikus szemafor** egy pozitív egészt tartalmazó változó és egy hozzá tartozó várakozási sor (melyen processzek blokkolódhatnak).

```
DOWN(semaphore S) {
   if (S >= 1)  S:=S - 1;
   else { a hivo helyezze magat az S varakozo sorara}
}

UP(semaphore S)  {
   S:=S + 1;
   if (S varakozo sora nem ures) {egy processzt vegy le rola}
}
```

6.11. ábra

A szemaforon - kivéve az inicializációját - két operáció hajtható végre. Az operációk *atomiak*. Ez két dolgot jelent. Egyik: garantált, hogyha egy operáció elindult, más processz nem férhet a szemaforhoz, míg az operáció be nem fejeződött, vagy a hívója blokkolódott.

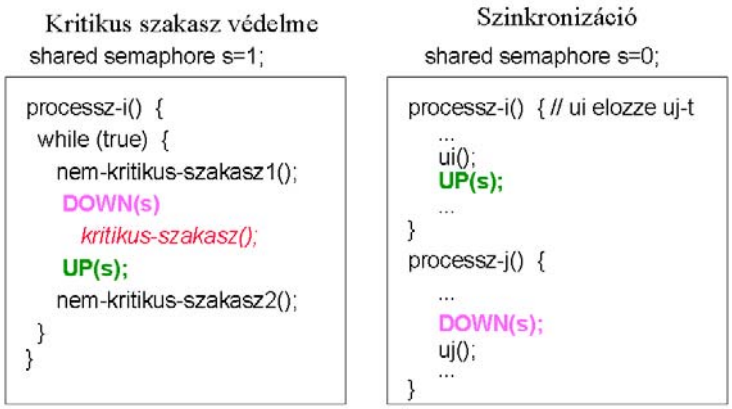
A másik: a szemaforra várakozó, blokkolódott processz "felébredve" végre kell tudja hajtani azt az operációt, amelyikre blokkolódott.

A két operáció:

DOWN operáció (P: passeren), ezen blokkolódhat a hívója,

UP operáció (V: vrijgeven [vrejhéfen]), ez szignáloz, hogy felébredjen egy blokkolódott processz (6.11.ábra)

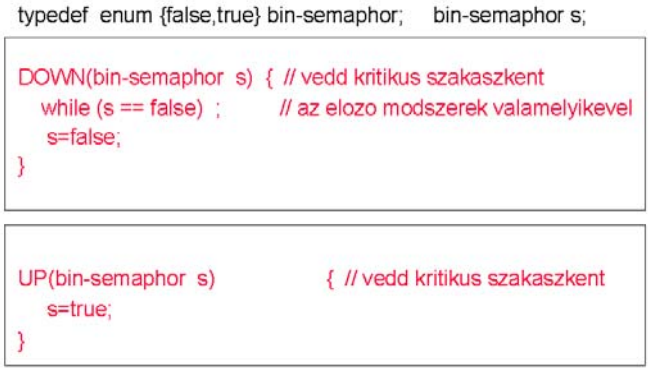
Megjegyzés: ha S == 0, akkor biztos, hogy egy processz a kritikus szakaszban van.



6.12.ábra.

A szemaforral a kritikus szakasz védelme, illetve a szinkronizáció a következő lehet (6.12.ábra):

Dijkstra professzor a szemafor megvalósításáról (implementációjáról) nem rendelkezett. Nem mondta ki, hogyan kell megvalósítani a várakozási sort, mit jelent a blokkolódás stb. Így az várakozás lehet tevékeny várakozás is, a szignálozás az ebből való kibillentés (spinlock szemafor), de a várakozás lehet az operációs rendszer által biztosított valódi blokkolódás is (sleep rendszerszolgáltatással), a szignálozás a kernel által biztosított wakeup hívással (blokkoló szemafor).



6.13.ábra.

6.14.ábra

A Dijkstra féle szemafor úgynevezett számlálós (counting) szemafor, de imlementálhatunk bináris szemafort is. Nézzük ezeknek a változatoknak a lehetséges implementációit!

Bináris spinlock szemafor megvalósítás

Ez a szemafor false és true értékeket vehet csak fel (azaz nem pozitív egészeket), és tevékeny várakozással oldja meg a "blokkoklódást". Íme a lehetséges implementációja (6.13.ábra), ahol is a két operáció atomiságát az előzőekben tárgyalt és minden követelményt kielégítő megoldások valamelyikével biztosítjuk (ez persze csak jelezve van megvalósítás kódjában):

Tevékeny várakozású számlálós szemafor

Implementációját ennek is bemutatjuk a 6.14.ábrán:

```
typedef struct { int value;          list-of-procs l;}  semaphore
semaphore s;
DOWN(semaphor  s)     {  // vedd kritikus szakaszkent
        if(--s.value < 0) { add a proc.t az s.l-hez;
                            blokkolodj, de folebredve folytasd!
        }
}

UP(semaphor  s)          {  // vedd kritikus szakaszkent
     if(++s.value<1) { vegy le proc.-ot az s.l-rol; wake-up;}
}

int ncount(semaphor  s) {  // vedd kritikus szakaszkent
     if(s.value<0) return abs(s.value); // a varakozok szamat
}                                       //  adja vissza
```

6.15.ábra

```
const N;
var mutex:  semaphore := 1;    // a be-kirakógéphez hozzáférés
var empty:  semaphore := N;    // üres rekeszek a raktárban
var full:       semaphore := 0;    // teli rekeszek jelzése
var buffer: array[0..N-1] of items;        // a raktár


producer i
var m: item;                          // a termék
loop
        m := produce_item();   // terméket gyárt
        DOWN(empty);           // üres rekeszre vár, vagy csökkenti
                DOWN(mutex);// belép a raktárba
                put_item(m);    // terméket egy rekeszbe
                UP(mutex);      // raktárt elhagyja
        UP(full);                  // teli jelzést növeli
        ...
endloop
```

6.16.ábra.

Természetes korlátozási követelmény: betelt raktár esetén a termelők blokkolódjanak, üres raktár esetén a fogyasztók blokkolódjanak.

A pszeudókódok a 6.16. és 6.17.ábrákon láthatók:

Blokkolós számláló szemafor

Ennek is megadjuk egy implementációját, méghozzá a Dijkstra féle két operációt kiegészítve egy harmadikkal, az ncount operációval (6.15.ábra). Ez nagyon hasznos operáció a szemaforhoz rendelt várakozási soron blokkolt processzek pillanatnyi számát adja vissza.

Termelő/fogyasztó probléma megoldása szemaforokkal

Tételezzük fel a következőket:

- Több termelő folyamat lehet, számukat nem korlátozzuk.
- Szintén nem korlátozott a fogyasztók száma.
- Korlátozott a raktár mérete: N számú hely van benne.
- Korlátozott a raktárhoz való hozzáférés: csak egy "ki-berakó gép" van.

```
consumer j
var m: item;
loop
        ...
        DOWN(full);       // teli rekeszre vár, vagy csökkenti
                DOWN(mutex);          // belép a raktárba
                m := get_item();      // kiveszi m-et
                UP(mutex);            // kilép a raktárból
        UP(empty);                    // növeli az üres jelzést
        consume_item(m);              // fogyaszt
        ...
endloop
```

6.17.ábra

6.3.9. Sorszámosztó és eseményszámláló [Reed , 1979]

Alapfogalmak

Az alapgondolat szerint egy sorszámosztó automata segíti egy szolgáltatás igénybevételének sorrendjét: a fogyasztó "tép" egy sorszámot és vár a szolgáltatásra, amíg a sor rákerül (a bakery algoritmusban megismertük ezt a módszert). A szolgáltató egy veremben tartja a kiszolgált fogyasztó sorszámát: annak tetején van az utóljára (vagy az éppen)kiszolgált fogyasztó sorszáma.

A szükséges objektumok és a rajtuk végezhető operációk:

S: sequencer nem csökkenthető integer változó, 0-ra inicializálva.

E: eventcounter sorszámok veremtára.

Operáció S-re:

ticket(S); visszatér a következő sorszámmal.

Operációk E-re:

read(E); visszaadja az E pillanatnyi értékét.

advance(E); növeli az E pillanatnyi értékét.

await(E,v: integer); várakoztat, amíg E eléri v-t.

Az utóbbi két operációt részletezzük:

```
advance(E)
begin E := E + 1;
    Kelts fel a várakozó sorból azt a processzt, aminek
    sorszáma a most igazított E-t elérte;
end

await(E, v: int)
begin if E < v then
        Helyezd a hívó processzt az E-hez tartozó várakozási
        sorba (és hívd a schedulert);
      endif
end
```

A használatuk (legáltalánosabban)

```
shared var E: eventcounter := 1;
S: sequencer := 0;
process i
begin
...
    await(E, ticket(S));
       critical_section();
    advance(E);
...
end
```

A termelő-fogyasztó probléma megoldása ezzel a mechanizmussal

A problémában feltesszük, hogy van egy berakó- és egy kirakógép, azaz korlátozzuk, hogy egy időben csak egy termelő, illetve egy időben csak egy fogyasztó használhatja a raktárt, de egy termelővel párhozamosan dolgozhat egy fogyasztó (persze, egy másik cellában lévő termékkel.) Természetesen termelő csak akkor használhatja a raktárt, ha van üres cella, fogyasztó pedig akkor, ha van töltött cella.

Figyeljük meg az advance(in) és az advance(out) kettős szerepeit! Az advance(in) jelzi, hogy töltődött egy cella: továbbengedhet egy fogyasztót az ő await(in, u+1) várakozásából, és továbbengedhet egy másik termelőt az ő await(in, t) várakozásából. Hasonló az advance(out) kettős szerepe is.

És most lássuk az algoritmusokat:

```
shared const N; // a raktár mérete
shared var
Pticket: sequencer := 0; // termelő sorszámosztó
Cticket: sequencer := 0; // fogyasztó sorszámosztó
in: eventconter := 0;
out: eventcounter :=0;
buffer: array[0..N-1] of items; // a raktár


producer i
var t: integer;
m: item;
begin
...
  loop
      m := produce_item();
      t := ticket(Pticket); // Egy termelő egy időben,
      await(in,t);          // ez biztosítva
      await(out, t-N+1);    // Vár üres cellára
      put_item(m, buffer[t mod N]); // m-et egy cellába tesz
      advance(in);          // raktárfoglatság nő,
                            // és engedély másik termelőnek
  endloop
...
end


consumer j
var u: integer;
m: item;
begin
...
  loop
      u := ticket(Cticket); // egy fogyasztó egy időben,
      await(out,u);         // ez biztosítva
      await(in, u + 1);     // vár termékre
      m := buffer[u mod N]; // kivesz terméket
      advance(out);         // jelzi egy cella kiürülését,
                            // engedélyez más fogyasztót
      consume_item(m);      // felhasználja a terméket
...
  endloop
....
end
```

Nézzék meg a *producer_consumer.eventcounter* fájlt! Ebben C szintaktikával adottak az algoritmusok. Mik a különbségek?

6.3.10. A monitor mechanizmus [Hoare, 1974, Hansen, 1975]

A *monitor* magasabb szintű szinkronizációs mechanizmus: eljárások, változók, adatstruktúrák speciális formájú gyüjteménye. A proceszek hívhatják a monitor eljárásait, de nem férnek a

monitor belső adatstruktúráihoz (information hiding), továbbá biztosított az is, hogy egy időben csak egy processz használhat egy monitort. A fordító biztosítja a monitorba való belépésre a kölcsönös kizárást (szokásosan egy bináris szemaforral), és így a programozónak ezzel már nem kell foglakoznia. Ha egy procesz hív egy monitorban lévő eljárást, az első néhány instrukció ellenőrzi, vajon más processz pillanatnyilag aktív-e a monitorban. Ha igen, a hívó blokkolódik, míg a másik elhagyja a monitort.

A termelő-fogyasztó problémához például az alábbi monitor-vázlat jó lehet:

```
monitor example
var i: integer;
c: condition;

    procedure producer (x);
    ...
    ...
    end;

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

end monitor;
```

A megoldáshoz kell a *conditon* típusú változó, amin két operáció hajtható végre: a

wait(c: condition), és a

signal(c: condition)

A wait(c) blokkolja a hívóját, amíg ugyanezen a c-n valaki signal(c) hívással jelzést nem ad ki.

Ezek után a következő oldalon láthatjuk a termelő-fogyasztó probléma teljesebb monitoros megoldását

```
monitor ProducerConsumer
const N;
var full,empty: condition;
count: integer;

procedure enter;
begin
    if count = N then wait(full);
    enter_item;
    count := count + 1;
    if count = 1 then signal(empty);
end;

procedure remove;
begin
    if count = 0 then wait(empty);
    remove_item;
```

```
    count := count - 1;
    if count = N - 1 then signal(full);
end;

count := 0;

end monitor;


procedure producer;
begin
    while true do
    begin
         produce_item;
         ProducerConsumer.enter;
    end;
end;


procedure consumer;
begin
    while true do
    begin
        ProducerConsumer,remove;
        consume_item;
    end;
end;
```

6.3.11. További klasszikus problémák

Az ebédelő filozófusok problémája

1965-ben Dijkstra bemutatott és megoldott egy szinkronizációs problémát, az azóta klasszikussá vált *Ebédelő filozófusok* problémáját. Ez a holtpont kialakulásának és az ekerülésének klasszikus példája.

A probléma a következő: öt filozófus ül egy aztal körül, mindegyiknek van egy soha ki nem ürülő tányér spagettije és mindegyik tányér mellett - valójában a tányérok között - van egy villa. Spagettit enni viszont csak két villával lehet. A filozófusok felváltva esznek vagy gondolkodnak.

Amikor egy filozófus megéhezik, megpróbálja megszerezni a tányérja melletti bal és jobb villát, bármilyen sorrenben, de egymás után. Ha sikerül két villát szereznie, akkor eszik egy darabig, majd leteszi a villákat és folytatja a gondolkodást.

A kulcskérdés: tudnánk-e írni olyan programot - minden filozófus számára -, ami megoldja a problémát és sohasem okoz holtpontot, egik sem hal éhen, mindegyik tud gondolkodni is (mert azt is szeretnek a filozófusok, nemcsak enni!

Nézzuk először a kézenfekvőnek tűnő megoldást! (dining.phil.rossz) Ebben a take-fork() vár, amíg a specifikált villa elérhető, aztán megragadja azt. Sajnos, ez a kézenfekvő megoldás

rossz. Tételezzük fel, hogy mind az öt filozófus egyszerre felveszi a baloldali villáját, ezután egyik sem vehet jobboldali villát: bekövetkezik a holtpont helyzet.

Ha úgy módosítunk a programon, hogy miután felvette a filozófus a bal villát, ellenőrzi, vajon elérhető-e a jobb villa, s ha nem leteszi a bal villát, még mindíg nem jó a program! Miért? Mert szimultán villafelvétel esetén - ami nem zárható ki - minden filozófus felvesz, majd letesz bal villákat. Éhen fognak halni! Az sem jó, ha a jobboldali villafelvételeket késleltetjük pl. véletlenszerű, vagy akár egy algoritmus szerinti idővel, mert nem szabad időzítési feltételekkel megkötötten megoldani a feladatot!

Jó megoldást közlünk a dining.phil.jo programocskában (szerző: Tannenbaum). Ebben a *mutex* szemafor védi a villákkal való manipulációkat: egyszerre csak egy filozófus vehet-tehet villákat! Másrészt számontartjuk a filozófusok állapotát: ÉHES, ESZIK, GONDOLKODIK állapotokat. Az algoritmus 5 filozófus esetén megengedi, hogy egyszerre két filozófus egyen!

Tetszik?

```
// Ebedlo filozofusok problema.       ROSSZ MEGOLDAS!

#include "prototypes.h"
#define N 5                    // number of philosophers
// i: which philosopher (0 to N-1)

void philosopher(int i) {
  while (TRUE) {
    think();                 // philosopher is thinking
    take_fork(i);            // take left fork
    take_fork((i+1) % N); // take right fork;
                             // % is modulo operator
    eat();                   // yum-yum,spaghetti
    put_fork(i);             // put left fork back on the table
    put_fork((i+1) % N);  // put right fork back on the table
  }
}


// Ebedlo filozofusok jo megoldasa.

#include "prototypes.h";
#define N               5 // number of philosophers
#define LEFT     (i-1)%N // number of i's left neighbor
#define RIGHT    (i+1)%N // number of i's right neighbor
#define THINKING        0 // philosopher is thinking
#define HUNGRY          1 // philosopher is trying to get forks
#define EATING          2 // philosopher is eating

typedef int semaphore; // semaphores are a special kind of int
int state[N);        // array to keep track of everyone's state
semaphore mutex = 1;  // mutual exclusion for critical regions
semaphore s[N];       // one semaphore per philosopher
```

```
void philosopher(int i) { // i: which philosopher (0to N-1)
    while (TRUE) {        // repeat forever
        think();          // philosopher is thinking
        take_forks(i);    // acquire two forks or block
        eat();            // yum-yum,spaghetti
        put_forks(i);     // put both forks back on table
    }
}

void take_forks(int i) {  // i: which philosopher (0 to N-1)
    down(&mutex);         // enter critical region
    state[i] = HUNGRY; // record fact that philo i is hungry
    test(i);           // try to acquire 2 forks
    up(&mutex);        // exit critical region
    down(&s[i]);       // block if forks were not acquired
}

void put_forks(int i) { // i: which philosopher (0 to N-1)
    down(&mutex);      // enter critical region
    state[i] = THINKING; // philosopher has finished eating
    test(LEFT);        // see if left neighbor can now eat
    test(RIGHT);       // see if right neighbor can now eat
    up(&mutex);        // exit critical region
}

void test(int i) {     // i: which philosopher (0 to N-1)
    if (state[i]==HUNGRY &&
        state[LEFT]!=EATING && state[RIGHT]!=EATING) {
            state[i] = EATING;
            up(&s(i));
    }
}
```

Az alvó borbély probléma

Borbélyüzlet, egy vagy több borbély és ugyanannyi borbélyszék van benne. Ezen kívül van N szék a várakozó vendégeknek.

Ha nincs vendég, a borbély alszik a borbélyszékében. Ha jön egy vendég, felkelt egy borbélyt, az hajat vág. Ha vendég jön, de minden borbélyszék (ezzel borbély) foglalt, akkor

- ha van üres várakozószék, akkor leül és várakozik. (Nem feltétlenül érkezési sorban szolgálják majd ki.)
- Ha a várakozószékek is foglaltak, akkor azonnal elhagyja a borbélyüzletet.

Lássuk Tannembaum megoldását!

```
//   Alvo borbely

#include "prototypes.h"
```

```
#define CHAIRS 5            // # chairs for waiting customers

typedef int semaphore;      // use your imagination

semaphore customers = 0; // # of customers waiting for service
semaphore barbers = 0;   // # of barbers waiting for customers
semaphore mutex = 1;     // for mutual exclusion
int waiting = 0;       // customers are waiting (not being cut)

void Barber(void) {
  while (TRUE)   {
    down(customers);// go to sleep if # of customers is 0
    down(mutex);          // acquire access to 'waiting'
    waiting = waiting - 1;// decrement # of waiting customers
    up(barbers);       // one barber is now ready to cut hair
    up(mutex);            // release 'waiting'
    cut_hair();      // cut hair (outside critical region)
  }
}

void Customer(void) {
    down(mutex);              // enter critical region
    if (waiting < CHAIRS) {  // if there are free chairs
        waiting = waiting + 1; // incr # of waiting customers
        up(customers);     // wake up barber if necessary
        up(mutex);         // release access to 'waiting'
        down(barbers);     // sleep if # of free barbers is 0
        get_haircut();     // be seated and be serviced
    } else {
        up(mutex);         // shop is full; do not wait
      }
}
```

6.4. A Unix szemafor mechanizmusa

A Unix szemafor mechanizmusa blokkoló jellegű, számlálós implementáció. De további jellegzetességek is vannak a Unix szemafor implementációban. Foglaljuk ezeket össze röviden, még akkor is, ha most ezeket nem valószínű, hogy értjük:

- Szemafor készlet lehet egyetlen Unix szemaforban.
- Operáció készlet hajtható végre egyetelen szemafor operációban (de azért az operáció atomi).
- A elemi operációk lehetnek blokkolók, de lehet nem blokkolók is.
- Az elemi oerációk nencsak 1 értékkel csökkenthetnek, növelhetnek (non-unitary-k).
- Lehetséges 0 ellenőrzés is.
- Blokkolódás esetén természetes, hogy a szemafor elemei az operáció előtti értéküket visszaveszik (automatikus undo az elemi operációkra).
- Eredeti szemafor-elem érték visszaállítás (undo) specifikálható a processz terminálódásához is.

Foglaljuk össze röviden a Unix szemaforokkal kapcsolatos rendszerhívásokat is:

- semget() rendszerhívással lehet szemafort (benne elemi szemaforok készletét) készíteni, vagy meglévő szemaforra asszociálni.
- semop() rendszrhívás a szemafor operáció. Ezzel operáció készlet hajtható végre akár vegyesen down és up, vagy 0 ellenőrzés, akár vegyesen a szemafor készlet elemein.
- semctl() rendszerhívás szolgál a szemafor kontrollálására: inicializálásra, megszüntetésre, pillanatnyi értékek, belső adatok lekérdezésére.
- Bizonyos hasonlóságokat fedezhetünk fel az üzenetsor (message queue) és az osztott memóriahasználat (shared memory) redndszerhívásaival.

Mielőtt megpróbáljuk részletezni a Unix szemaforhasználatot, tanácsoljuk a nyájas olvasónak, hogy tanulmányozza a példaprogramokat, a man lapjait, ezekből sokat tanulhat.

6.4.1. A Unix szemafor elemi szemaforok készlete

Egy egyedi (single) szemafor egy pozitív egész lehetne (0 és 32767 tartományban). A korszerű Unix-ok nem egyedi szemaforokat kezelnek, hanem szemafor készleteket (semaphor set). Egy szemafor készlet kötött számú elemi szemafort tartalmaz. Példáinkban az nsem változóval adtuk meg a készletekben az elemi szemaforok számát.

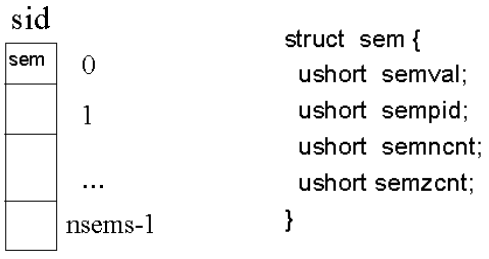
Semaphor set készíthető (kreálható) a semget() rendszerhívással. Ugyancsak a semget() rendszerhívás alkalmas létező szemafor készlet azonosítására, hacsak explicite nem tudjuk a szemafor készlet azonosítóját (példáinkban sid, semid, vagy empty az azonosító).

Az a processz, amelyik semget() hívással készített szemafort, az lesz a szemafor tulajdonosa. Ő határozza meg a key kulcsot, ami a szemafor külső azonosítására szolgál, ő határozza meg a készletben a szemaforok számát, (az nsem értékkel), és ő állít be használati engedélyeket (rw) a szemafor készletre (a flags-szel). Más processzek a semget() hívással asszociálhatnak a szemaforra. Ismerniük kell a key kulcsot, az nsems értéket, és a flags-szel vizsgáltathatják, vajon létezett-e már a szemafor.

A semget() hívás szintaxisa:

```
sid=semget(key,nsems,flags);
```

A semget nsems számú elemi szemaforból álló szemafort kreál key kulccsal, beállítva a hozzáféréseket is, vagy asszociál a key kulcsú szemaforra. Visszaadott értéke, a sid ezek után a processzben használható belső szemaforazonosító, vagy pedig a kreáció/asszociáció sikertelenségét jelző érték.



6.18 .ábra.

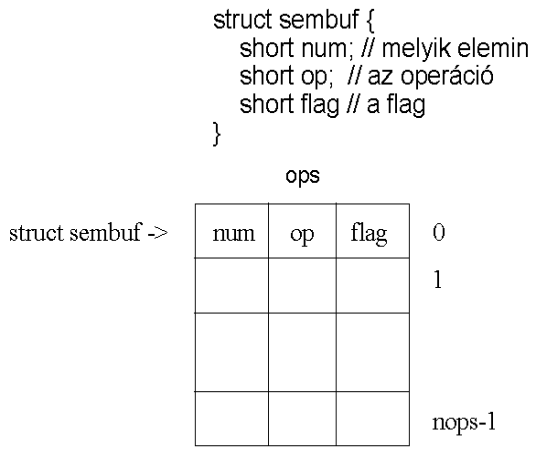
Az elemi szemaforok rendre 0, 1, ..., nsems-1 index-szel azonosítottak. A sid szemafor, és egy elemi szemafor adatstruktúráját is láthatjuk a 6.18. ábrán.

6.4.2. A Unix szemafor operációja elemi operációk készlete elemi szemaforok készletén

Az engedélyekkel rendelkező processzek operációkat hajthatnak végre a szemaforokon (semop()). A rendszerhívás szintaxisa:

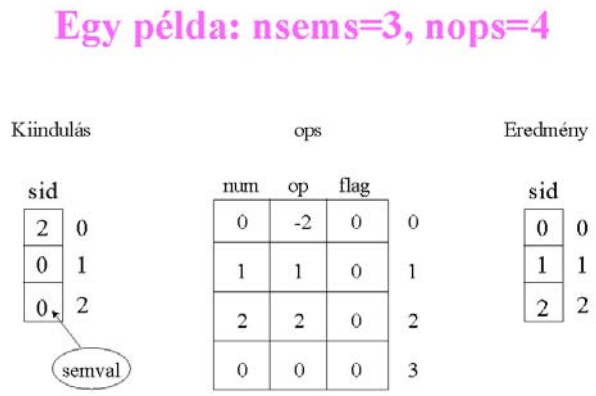
```
semop(sid,ops,nops);
```

A semop a sid szemaforon nsops számú operációt hajt végre (atomi módon). Az ops-ban definiáltak az operációk, az is, hogy melyik elemi szemaforon hajtódjanak végre, és még az ops-ban flag-ek is befolyásolhatják az operációkat.



6.19.ábra

Egy elemi operáció és az operációkészlet adatstruktúráját látjuk a következő (6.19.) ábrán. Az elemi operációk indexe is 0 és nops-1 között lehet. A struct sembuf első tagja a num: ez fogja indexelni, hogy melyik elem szemaforon kell az elemi operációt végrehajtani. Második tagj az op: ez maga az operáció. Harmadik tagja a flag, ami az elemi operációt módosíthatja. A flag beállítható pl. IPC_NOWAIT makróval: ez azt jelenti, hogy ha blokkolódnia kellene az elemi operáción a semop-nak, akkor se blokkolódjon (ilyenkor persze, az elemi operáció nem hajtható végre, de erről a semop visszatérési értékével meggyőződhetünk, kezelhetjük az esetet.) A flag beállítható SEM_UNDO makróval: terminálódáskor álljon vissza minden ilyen operáció eredménye).



6.20.ábra.

Nézzünk ezután egy példát! A 3 elemi szemaforból álló készleten 4 elemi műveletet hajtunk végre. A 6.20.ábrán balra láthatók a sid szemafor elemi szemaforainak kindulási értékei, középen az ops, jobbra a sid végeredménye.

Alapvetően háromféle elemi operáció lehetséges:

- inkrementálás (up), (V), ha az op > 0,
- dekrementálás (down), (P), ha az op < 0,
- 0 ellenőrzés, ha az op = 0.

Az up operáció során a megfelelő elemi szemafor értéke az op értékével növekszik.

Ha az op < 0, akkor szükség esetén blokklódik a hívó, amíg a megfelelő semval >= |op| lesz, és a semval értékéhez hozzáadódik az op (azaz levonódik az abszolút értéke).

Ha az op = 0, blokkolódik a hívó, míg a kijelölt semval 0 nem lesz.

Látjuk, az *up* operációban pozitív argumentummal, a *down* operációban negatív argumentummal hívjuk a semop() *system call*-t. Ezek az operációk atomiak.

A *down* operáció nemcsak egyszerű dekrementáció, hanem egyben tesztelés, vajon a szemafor-elem pillanatnyi értéke nagyobb-e a dekremnetációs értéknél. (Az is tesztelhető, hogy egy szemafor érték éppen 0-e, vagy sem!) Ez a tesztelés sikeres, vagy nem.

A *down* operáció lehet un. "blocking" vagy "non blocking" jellegű, az IPC_NOWAIT *flag* beállítástól függően. Ha nem sikerül a dekrementáció ilyen esetben, abortál a semop, de továbbfuthat a hívó.

Non-blocking jellegű *down* esetén (IPC_NOWAIT beállítva) sikeres teszt után normális a dekrementáció és normális a visszatérés a semop()-ból, sikertelen teszt után a semop() visszatérési értéke viszont -1 és nem dekrementál. A non-blocking down haszna akkor jelentkezhet, ha szemaforral védett erőforráshasználat során ahelyett, hogy várakoznánk a foglalt erőforrásra, valamilyen més hasznos feladatot akarunk megoldani. Maga a tény, hogy az erőforrás nem áll rendelkezésre, kiderül, a processz azonban futhat tovább más vezérlési szálon, mint az erőforráshasználat kritikus szakasza.

Blocking jellegű *down* esetén sikeres teszt után normális dekrementáció és vissztérés következik, sikertelenség esetén viszont a hívó processz blokkolódik a szemaforon, amíg a szemafor érték meg nem engedi az operációt, vagyis amíg egy másik processz nem inkrementálja a szemafort oly mértékben, hogy a dekrementáció megtörténhessen. A processz tehát akkor kerül le a szemafor várakozó soráról, ha más processzek inkrementlják a szemafort, vagy megszüntetik azt.

Egyetlen semop() hívásban a szemafor készlet elemeire blokkolt/nem-blokkolt operációk nsops számú akciója (operáció készlet) hajtódhat végre. Az operációk készlete akármelyik szemafor-elemre vagy akár az összes elemre kérhető! Mindazonáltal nem hajtódik végre operáció, amíg az összes végre nem tud hajtódni! Ez azt jelenti, hogy blokkolt szemafor operációknál ha az összes operáció-elem nem hajtódhat végre sikeresen (vagyis ha processz blokkolódik), az összes megelőző elemi operáció eredménye visszaáll, hogy ne legyen változás, amíg minden végre nem hajtódhat!

Pl: ha 10 szemaforos készleten 6 operációt hajtanánk végre egyetlen semop() hívással (mondjuk mind a hatot minden szemafor-elemre), és a 6 operáció-elemből 3 sikerül, de a negyedik nem, továbbá ez éppen blokkoló, akkor a a hívó processz blokkolódik, de a megelőző 3 operáció eredményei visszaállítódnak, amíg a maradék operációk is sikerülhetnek. Ez egy *automatikus undo* akció.

Tovább is bonyolíthatjuk! Bármely operáció, ami megelőzi vagy követi a blokkoló operációt, beleértve magát a blokkoló operációt is, specifikálható SEM_UNDO *flag*-gel. Ez a *flag* azt kéri, hogy ha a hívó processz terminálódik, a sikeres semop()-ok erdeményei álljanak vissza. (Az ok nyilvánvaló: ha szemafor segítségével lefoglalunk valmilyen erőforrást és mielőtt felszabadítjuk a processz terminálódik, ne maradjon foglalt az erőforrás! Általában kivételes különleges események miatti terminálódásra igen hasznos lehetőség ez.)

A Unix-ok biztosítják a szemaforokhoz az un. *undo* (csináld vissza) mechanizmust. Létezik egy *undo* struktúra, melyekben feljegyződnek a *blocking* operációk, ha a SEM_UNDO beállított (és nincs IPC_NOWAIT).

6.4.3. A semctl rendszerhívás

A szemafort azonosító processzek - az engedélyektől is függően - semctl() rendszerhívással átállíthatják az engedélyeket, beállíthatják és lekérdezhetik a szemafor pillanatnyi értékeit, megkérdezhetik, melyik processz operált utoljára a szemaforon, mely processzek várakoznak a szemafor várakozó sorában stb.

A szintaxis:

`semctl(sid, snum,cmd,arg);`

Ez a hívás a sid szemafor snum indexű elemi szemaforán cmd parancsot hajt végre. Ha a parancsnak operandusa van, akkor az az arg. A lehetséges parancsokból néhány: IPC_RMID makró adja a sid szemafor megszüntetését. A SETVAL makró kifejtve a beállításhoz jó, kell neki az arn. Pl. ezzel inicializálhatunk elemi szemafort. A GETVAL pedig éppen lekérdez pillanatnyi értékeket. A semctl használatát a példaprogramokból, a manual lapból elsajátíthatjuk.

A példaprogramocskák:

- `semset.c` Kreál/azonosít szemafor készletet, benne egyetlen szemafort. Figyeljük az azonosítás technikáját! Bekéri stdin-ről a kezdő értéket, és ezt beállítja. (Megjegyzem, a többi programocska ugyanezen szemafor készletet azonosítja, hasonló semget() hívásokkal.)
- `semval.c` Lekérdi és kiírja a pillanatnyi szemafor értéket.
- `semkill.c` Megszünteti a példácskák szemafor készletét.
- `semup.c` Ez is futtatható program, a sembuf.sem_op=1 értékkel inkrementálja a szemafort.
- semdown.c Ez is futtatható, a sembuf.sem_op=-2, tehát 2-vel dekrementál. Próbáljuk ki SEM_UNDO-val is, és anélkül is! Processzként felfüggesztődik, ha a szemafor pillanatnyi értéke nem engedi a dekrementációt, de SEM_UNDO beállítással - miután sikeresen továbbjutott, akár várakozás után, akár anélkül - exitnél "visszacsinálja" a dekrementációt. Ha valóban dekrementálni akarunk, vegyük ki a SEM_UNDO-t.
- `downup.c` Három függvény, hozzá kell linkelni a producer és a consumer programokhoz! Figyelem! A down-ját írjuk át SEM_UNDO beállítás nélkülire is, így is próbáljuk ki!
- `downup_uj.c` Öt függvény, más technikával, mint a downup.c.
- `consumer.c` Futtatható. Szemafor azonosító az empty változó (bár a szemafor maga ugyanaz, mint az előző programocskákban). Végrehajt egy up-ot, és kiirja a pillanatnyi értéket. Tulajdonképpen a producer párjának szántam.
- `producer.c` Main program ez is, hívja a down és a val függvényeket. N-szer akar down-olni, és persze, blokkolódik, ha nem elég nagy az "üres rekeszek" száma: az empty szemafor! Továbbfut, ha egy *consumer* "kivesz" egy terméket: növeli 1-gyel az `empty` szemafor értéket.

Javasolt gyakorlatok a fenti programocskákkal és az ipcs, ill. ps parancsokkal:

1. *semset*-tel készíts szemafort. Adj neki pl. 5 értéket. *ipcs*-szel nézd meg, *semval*-lal kérdezd le. *semup*-pal növeld az értéket, *semval*-lal mégegyszer győződj meg a növelt értékről. *semdown*-nal csökkentsd az értékét. Sikerült? Hogy van *semdown*-ban a SEM_UNDO beállítva? *ps*-sel nézheted, felfüggesztődött-e a *semdown*.
   Próbálj ki ezekkel további forgatókönyveket! A végén *semkill*-lel megszüntetheted a szemafort!
2. Compiláld, linkeld össze a *producer*-t és a *consumer*-t, az downup.c két változatával is! (Egyikben a *down*-ra SEM_UNDO legyen!)
   Utána *semset*-tel készíts szemafort, mondjuk 4 kezdő értékkel. (Ne felejtsd, a *producer-consumer* ugyanazt a szemafort használja, mint az előző programocskák használtak!)
   Háttérben indíts el a *producer*-t. Mit csinál ez? *ps*-sel nezd, blokkolt-e. *semval*-lal nézheted a pillanatnyi értéket!

Most indítd a *consumer*-t! Mi lesz ekkor a *producer*-rel? ps-sel is nézd! *semval* is kipróbálható!
Mi lenne, ha *semset*-tel egy nagyobb, pl. 8 értéket adunk a szemafornak? Blokkolódik a *producer*? Mi a szemafor érték, ha downup.c-ben SEM_UNDO beállított?

3. Próbálj további játékokat! Végén a *semkill*-lel itt is törölhetsz.

7. Memóriamenedzselés

A memória igen fontos erőforrás. Bár a gépek egyre nagyobb központi memóriával rendelkeznek, a memóriával mindenképp gazdálkodni kell. mert az alkalmazások is és az operációs rendszer magja is egyre nagyobb memóriát igényelnek. (Érvényes itt Parkinson törvénye: "Programs expand to fill the memory aviable to hold them.") A memóriagazdálkodás feladatköre a védelem biztosítása is, egy processz ne érhesse el más processzek memóriáját, a kernel memóriát csak ellenőrzötten használhassák, ugyanakkor osztott memóriahasználatot tudjunk biztosítani.

A memóriamenedzselés fontos operációs rendszerbeli feladat, de igen szoros az együttműködése a hardverrel. Éppen ezért nagyon függ a HW architektúrától.

A *memória* (tár) címekkel rendelkező rekeszek (bájtok, szavak) készlete. A gépi instrukciókat a memóriából veszi a CPU a programszámláló regiszter (PC) pillanatnyi értéke szerint, másrészt az instrukciók címrésze hivatkozhat a memória rekeszre (az instrukció operandusa a memóriában van). Lássuk be: az instrukciókban a memória cellák címei szerepelnek, és az instrukció "nem tudja", hogyan generálódik az a cím ami végül is a buszra kerül, hogy a memória cella tartalma a processzorba jusson (vagy fordítva).

A memória celláknak van címe, a címek készlete a *címtartomány* (address space). Egy cím csakis egy címtartomány elemét veheti fel. Megkülönböztethetünk fizikai címtartományokat (physical address space) és logikai címtartományokat.

A *fizikai (vagy valós) címtartományok* elemei olyan címek, amik kiadhatók a buszra, a fizikai memóriacellák címei alkotják a készletet. A fizikai tárnak is van címtartománya. Ez nem folytonos feltétlenül, olyan értelemben is, hogy lehetnek olyan címek, amik mögött nincs memória-cella! Ráadásul itt még átfedések is lehetnek: ugyanazon címhez több memória rekesz is tartozhat (tipikus példa az IBM PC, lásd később), ilyenkor persze meghatározott, hogy melyik cellát szólítja meg a buszra kiadott cím (ez további címkötődési probléma).

Az instrukciókban szereplő címek készlete a *logikai címtartomány*. A virtuális memóriamenedzselő rendszerekben ez a *virtuális címtartomány*. Meg kell jegyeznünk, hogy a logikai címtartomány nem feltétlenül folytonos, egydimenziós tartomány. Lehet több tartomány is (régió fogalom az SVID-ban, szegmensek a Linuxban stb.). Itt is lehetséges, hogy egy címhez (egy címtartomány szakaszhoz) nem is tartozik memória.

A címkötődés (Address Binding)

Programjaink több lépcsőn mennek át fejlesztésük során, ebből következően több címkötődési eset lehetséges. A lépcsők: a fordítás (compilation), a taszképítés (link), a betöltés (load) és a végrehajtás (run), bármelyikben történhet a címkötődés.

- **Kötődés a fordítás során**: abszolút cím generálódik már a tárgymodulokban. Ha itt nincs kötődés, akkor a tárgymodulokban relatív címek generálódnak.
- **Kötődés a taszképítés során**: abszolút címek generálódnak a végrehajtható modulokban (executable files), különben azokban relatív címek vannak. Ha relatív címek vannak, akkor a program áthelyezhető (relokálható). Egy fajta relatív cím a virtuális cím (lásd később).
- **Kötődés a betöltés során**: a végrehajtható programfájlban relokálható címek a betöltés során "átíródnak", és abszolút címekké válnak. Ha nincs kötődés a betöltés során, akkor a processz kódszegmensének címei még mindig relatív címek (pl. virtuális címek).

- **Kötődés a futás során, dinamikus a kötődés**. A processz kódjában az instrukciók ún. *logikai címeket* tartalmaznak. A logikai cím lehet relatív cím, virtuális cím. A CPU tehát logikai címet kap feldolgozásra, ezeket átképzi fizikai címekké és ez adható ki a buszra.

Mint említettük, az operációs rendszerek magjának fontos része a memória menedzselő alrendszer, amelyik szorosan együttműködik a hardverrel: az *MMU*-val (Memory Management Unit). A memóriamenedzselő alrendszernek két nagyobb, jól megkülönböztethető feladatosztálya van. Az egyik

- a *memória allokálás* feladatköre. Ez lehet *statikus* (egy processz keletkezésekor címtartományt és memóriát kell biztosítani), vagy *dinamikus* (a processz futása során további memóriát igényel, ami természetesen címtartomány bővítéssel járhat). A memória allokálás.- a mai rendszerekben – kisöprési-kilapozási területek biztosításával és leképzési táblák létrehozásával (bővítésével) járnak.
- A másik feladatkör a *címleképzés* (a címkötődés) segítése. Ez a feladatkör különösen a dinamikus címkötődéses rendszereknél – a mai virtuális címzésű rendszereknél feltétlenül – fontos. A címleképzés a logikai (virtuális) cím "átalakítása" valós címmé, közben ki- és besöprési, ki- és belapozási alfeladatok merülhetnek fel.

Különböző memóriamenedzselő sémák lehetségesek. Ezek együtt fejlődtek a hardver és szoftver fejlődéssel.

7.1. Memóriamendzsment osztályok

Több szempont szerint is osztályozhatunk. Egyik osztályozás szerint vannak olyam memória menedzselő rendszerek, melyek

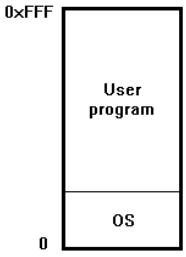
- mozgatják a processzek kontextusát (vagy annak egy részét) a fő memória és a másodlagos memória (háttértárak: diszkek, régebben dobok) között:
  - ki-belapozó rendszerek;
  - ki-besöprő rendszerek;
- nem mozgatják a kontextusokat.

További osztályozás szerint lehetnek:

1. valós címzésű rendszerek, ezen belül
   - monoprogramozású,
   - multiprogramozású rendszerek, ezen belül
     - fix partíciós,
     - változó partíciós rendszerek;
2. virtuális címzésű rendszerek, ezen belül
   - lapozós (paging) rendszerek,
   - szegmens-leképző, ki-besöprő rendszerek,
   - szegmentáló, ki-besöprő és ugyanakkor lapozó rendszerek.

Nézzünk ezekre tipikus példákat, közben magyarázva a lényeget is.

7.1.1. Valós címzésű monoprogramozás



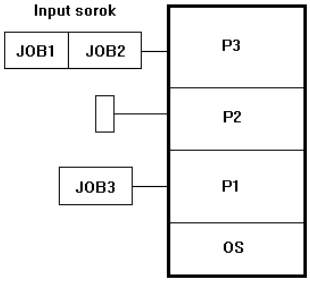
7.1. ábra. Valós címzésű monoprogramozás

A legegyszerűbb séma ez, mely szerint egy időben egy processz (vagy taszk, job) van a memóriában és az elfoglalhatja az operációs rendszer mellett a teljes memóriát. A program fejlesztése során a fordító-taszképítő (compiler-linker) valós címeket generál, a cím és memóriakötődés történhet bármelyik fázisban.

Hasonlít erre a rendszerre a nálunk valamikor elterjedt MS DOS operációs rendszer memóriamenedzselése. Bár az MS-DOS-ban ugyan létezhet több processz egyidőben a memóriában, de mindig csak egy aktív belőlük. Megemlítjük azt is, hogy a .COM fájloknál a kötődés a betöltés során, az .EXE fájloknál futás során történik.

7.1.2. Valós címzésű multiprogramozás

A fordító-taszképítő itt is valós címeket állít elő, a futtatható programokba valós címek vannak tehát. A taszképítést és a memóriamenedzselést is befolyásolja a két lehetséges alosztály: fix, vagy változó partíciós lehet ez a rendszer. Tipikus példa volt erre a rendszerre az IBM System 360-as rendszere. Ez közös input sorokkal dolgozott, de könnyen elképzelhetjük a szeparált bemeneti sorokkal dolgozó rendszereket is.



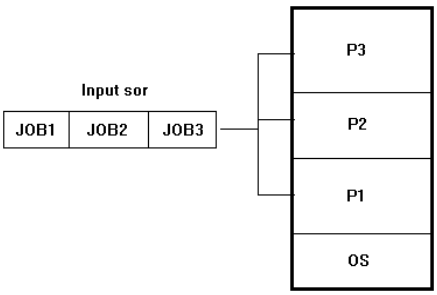
7.2. ábra. Valós címzés, fix partíciók

Multiprogramozás fix méretű partíciókban, szeparált input sorokkal

A memóriának az operációs rendszer által el nem foglalt részét fix méretű részekre, *partíciókra* osztották. Mindegyik partíciónak saját input sora volt, a programokat (job-okat) az egyes partíciókba kellett fordítani-linkelni. A programozó tehát tudta, hogy milyen címen kezdődik és meddig tart egy-egy partíció, a compiler-linker csakis a partícióba tartozó címeket generál. Cím és memória kötődés a link során történik. A memória kezelés sémája a 7.2. ábrán látható.

Valós címzés fix partíciókkal, közös input sorral

A szeparált input sorokat használó rendszer hátránya volt, hogy maradhattak üres partíciók, noha voltak munkák, csak éppen nem az üres partícióba fordították, linkelték azokat. Ezt kiküszöbölendő, a fejlődés következő lépcsője a 7.3. ábrán látható séma volt: a partíciók fix méretűek, de egyetlen közös input sorral. Bármelyik partíció kiürülve azonnal fogadhatta a következő munkát, ami még belefért.

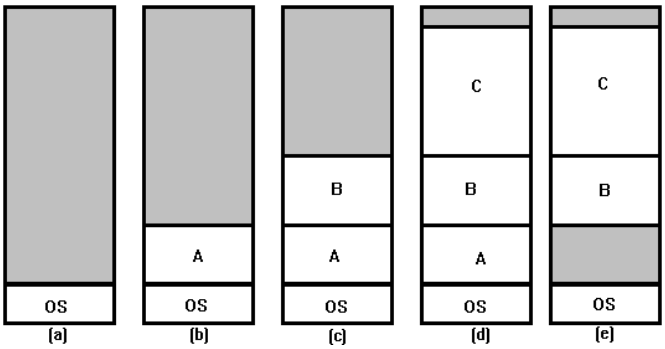


7.3. ábra. Fix partíciók közös input sorral

Megoldandó probléma volt ennél a rendszernél az *áthelyezés* (relocation): miután a programozó nem tudhatta, programja melyik partícióba fog kerülni, nem linkelhette programját a partíciókhoz tartozó címhatárok közé. A megoldás a következő volt: a fordító-taszképítő eltolás címeket (offset) generált, melyek a program 0 kezdő címétől számítódtak. A végleges címet előállíthatták ezek után két módon is: egyik szerint a program betöltése (load) során

minden címet átírtak az eltolás értékhez hozzáadva a partíció kezdőcímét, így előálltak a betöltött programban a partícióhoz tartozó címek. Másik megoldás szerint - mikor is minden címzés egy szegmensregiszter és egy eltolás értékből adódik -, a betöltés során nem változtatták meg a címeket, hanem a partíció kezdőcímét betöltötték a szegmensregiszterbe. Futás során így is a partícióhoz tartozó címek keletkezhettek.

A másik megoldandó probléma a védelem volt: hogyan ellenőrizzék, hogy egy-egy program ki ne címezzen a partíciójából. Nos, a betöltés során a címeket átíró módszer esetén az átírással egyidőben lehetett ellenőrizni minden címzést. A szegmensregiszteres címzéses megoldáshoz más ellenőrzés kellett. Kétféle ellenőrzési módszer is bevált. Egyik szerint a partíciók címeit regiszterekben tartva minden címzés ellenőrződött, vajon e két határ közé esik-e. Vegyük észre, hogy ez dinamikus jellegű, a gépi instrukciók végrehajtása közben történik az ellenőrzés. Másik megoldás szerint - ilyen volt a híres 360-as rendszer - a memória minden 2 blokkja kapott 4 bites védelmi kódot, ugyanakkor a PSW-ben is volt 4 bites védelmi kód. Ha a címzés során e két kód nem egyezett, bekövetkezett a címzés védelem megsértése kivételes esemény.



7.4.ábra. Változó méretű partíciók

Multiprogramozás változó méretű partíciókkal

Erre is van tipikus példa: a CDC Cyber rendszere.

A lényeg itt is az, hogy a fordító-taszképítő valós, eltolás címeket generál (kötődés itt még nincs), a rendszer memóriáját azonban nem osztjuk fel előre partíciókra. Az input sorba került munkák a szabad területről, vagy a már korábban kialakult, a rendszer élete során dinamikusan változó méretű partíciókból igényelnek memória partíciókat. Vannak tehát "szabad" partíciók, és vannak foglaltak. A 7.4. ábrán azt mutatjuk be, hogy A, B, C stb. különböző méretű munkák ilyen sorrendben "belépve", az ábrán láthatóan befejeződve hogyan *partícionálják* a memóriát (kötődés a betöltés vagy a futás során történhet).

A megoldandó feladatok ennél a rendszernél a *relocation* és a *védelem* feladatain (amit az előzőekhez hasonlóan kellett végezni) kívül:

- A munkák helyigényét előre meg kell mondani, hogy ellenőrizhető legyen, befér-e egy - egy nem használt "partícióba."
- Valamilyen stratégia kell az elhelyezésre, ha több üres helyre is befér a munka.
- Idővel szegmentálódik a memória. Szükség van az üres szomszédos szegmensek összevonására, időnként pedig az összes üres terület egyetlen területre való összevonására: a Memory Compaction-ra.

Az elhelyezés stratégiák a következők lehetnek:

- First Fit (Next Fit) stratégia.
- Best Fit stratégia.
- Worts Fit stratégia.

A stratégiák minősítése függ attól, hogy milyen módszerekkel tartják nyilván a memória foglaltságot. Elvileg ez lehet bit térképes, vagy láncolt listás. Az utóbbi volt az elterjedt.

A *best fit* stratégia lényege az, hogy a soron következő munka abba a szabad memória partícióba kerül, amibe a legkisebb veszteséggel befér. Az elgondolás mögötte az, hogy így lesz a legkisebb a veszteség. Érdekes tapasztalat volt azonban, hogy az elgondolás nem ad jó eredményeket. Kikeresni a szabad területek láncolt listáján a „legjobban passzoló" területet időigényes, ugyanakkor a memória szegmentálódik, keletkeznek benne kisméretű szabad partíciók, amikbe később nem is lehet semmit tölteni, csakis memória összevonás után használhatók: éppen az ellenkezőjét érjük el, mint amit akartunk, lesz veszteség.

A *worst fit* stratégia szerint szintén végig kell keresni az összes szabad területet, és abba kell betölteni a munkát, amiben a "legtágasabban" fér el. Időigényes ugyan a keresés, de kevésbé szegmentálódik a memória, nem lesznek nagyon kicsi üres szegmensek.

Egész jó eredményeket szolgáltat a *first fit*, és még jobb a *next fit* stratégia: az első (vagy a következő) szabad területre töltsük a munkát, amibe belefér. Elmarad az időigényes keresés és tapasztalatok szerint nem szegmentálódik túlságosan a memória.

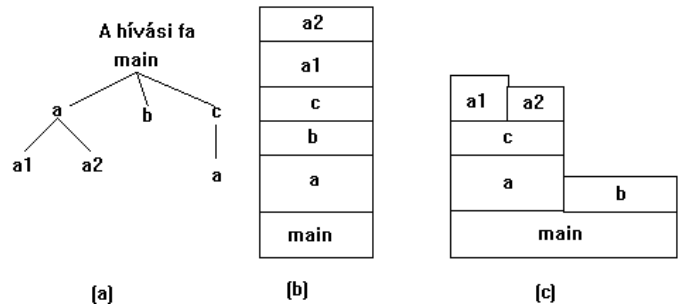
Ennél a memóriamenedzselési sémánál már felmerült a ki-besöprés megvalósítása! Ezzel alkalmassá vált a rendszer az időosztásra (time sharing). Azok a munkák, amik blokkolódtak (vagy az operátor blokkolta őket), kisöprődhettek másodlagos tárolóterületre: nagyon gyors dobtárakra, esetleg diszkekre. Persze, ekkor azonnal felmerült a kérdés, mekkora legyen a kisöprési terület, minden munka (job) számára külön szervezzék, vagy egy közös, de elég nagy területet biztosítsanak erre. Különböző megoldások voltak erre a különböző rendszerekben.

7.2. A virtuális címzés koncepció

A koncepció kialakításának kiváltó oka az volt, hogy egyre nagyobb programokat írtak a programozók, nagyobbakat, mint a rendelkezésre álló fizikai memória. Eleinte a megoldás az átfedés (*overlay)* technika volt. *Átfedésekkel* a valós címzésű rendszerekben is lehetett nagy programokat futtatni.

Az overlay lényege

A programokat rendszerint szubrutinokból (eljárásokból, függvényekből) állítjuk össze, és valószínű, hogy egy-egy adott időben nem hívunk meg minden lehetséges rutint. Azaz, előre összeállíthatjuk a rutinok hívási fáját: ezen ágak az egy időben egymást hívó rutinok. Nézzük pl. a 7.5. ábrát!



7.5. ábra. Az overlay magyarázata

A hívási fa szerint egy időben - legrosszabb esetben - meg lehet hívva a *main-c-a-a1* rutin-sorozat, de sohasem lesz egyidőben meghívva a *b* rutin az *a* rutinnal, vagy a *c*-vel. Ugyanígy látható az is, hogy az *a1* rutin sohasem lesz egyidőben meghívva az *a2* rutinnal, ezek ugyanarra a kezdőcímre is betölthetők, ha dinamikusan tudjuk őket hívásuk előtt betölteni. Az ábrán a (b) ábrarészen bemutatjuk, hogy mennyi lenne a helyfoglalás, ha minden rutin egyidőben be lenne töltve a memóriába. A (c) jelű ábrarészen viszont azt mutatjuk, hogy az egész programnak az élete során ennél kisebb memóriaterület is elég lenne, hiszen sohasem hívódhat

meg egyszerre minden rutin. Az overlay technika alkalmazása során úgy *linkelték* össze a programokat, hogy szükség esetén a rutinok betöltődhettek, felülírva a nem szükséges rutinok memória területét. Ráadásul, a linkerek képesek voltak a betöltést "automatizálni", azaz, aha a linkernek megmondta a programozó a hívási fát, akkor a linker beírta a programba a betöltési utasításokat.

Ma, a virtuális címzések kialakulásával nincs szükség az *overlay*-ezésre.

A virtuális memória lényege

A virtuális memória koncepció a processzek számára azt a képzetet kelti, hogy igen nagy címtartományt és memóriát kezelhet. A koncepció szerint minden processznek igen nagy virtuális címtartománya lehet, ami egy vagy több résztartományból állhat. A virtuális címek virtuális memória cellákat címeznek. A virtuális cellákat a memóriamenedzselés biztosítja: a cellákat vagy a fizikai memória cellái, vagy másodlagos memória (többnyire diszkek) cellái adják.

A taszképítés (linkelés) során virtuális címeket generálnak, a processzek kontextusában (a kódrészekben) virtuális címeket találunk. A processzek futása során dinamikus címleképzés van: a virtuális címeket a buszra kiadható fizikai címekké kell leképezni. A leképzés a processz szemszögéből transzparens, azaz a processznek „nem kell törődni” azzal, vajon a hivatkozott virtuális címhez tartozó cellát pillanatnyilag a fizikai memória cellája adja, vagy az a másodlagos tároló cellája-e. Utóbbi esetben ugyanis a memóriamenedzselés kibemozgatást is végeztet, gondoskodik arról, hogy a virtuális memória másodlagos táron található része bekerüljön a fizikai memóriába.

Legyen a virtuális címtartomány V. Az egyes gépeken a fizikai memória mérete kötött. A fizikai címtartomány az R. Igaz az, hogy

$$V >> R$$

(Olvasata: V jóval nagyobb, mint R).

Mint említettük, a futó processzek szövegében virtuális címek szerepelnek. Amikor a CPU egy instrukciót végrehajt, annak címe virtuális cím, és annak címrészében virtuális címet kap. Az instrukció végrehajtása során a V-beli virtuális címet le kell képeznie R-beli fizikai címre, és a leképzett cím adható ki a buszra. Ez a címleképzés dinamikus! Minden egyes gépi instrukció feldolgozása közben megtörténik a címleképzés. Fontos, hogy gyors legyen ezért a leképzés végrehajtásában a memóriamenedzselő kernel szoftvert segíti a hardver: az MMU (Memory Management Modul).

$$V_{address} \ \underline{\text{Dynamic map}} \ R_{address}$$

Lássuk be, nem elég az „egyszerű leképzés”. Mivel a V >> R, még egy processznél is (ráadásul több processzt is kezelhet a rendszer) előfordulhat, hogy a leképzés sikertelen: a virtuális cella pillanatnyilag csak a másodlagos tárolón van. Úgy szokták mondani, a leképzett cím nem érvényes (not valid), ha van egyáltalán leképzett cím, a leképzett R-beli címen éppen nem az adott processz kontextusához tartozó információk vannak Gondoskodni kell tehát arról, hogy a másodlagos tárolóról az információ bekerüljön a fő memóriába. Hogy beférjen, esetleg onnan valamit ki is kell írni. Tehát van adatmozgatás is a dinamikus leképzésen kívül, a szükségnek megfelelően.

Két dolog szerencsére segít. Az egyik: a programok lokalitása. (Emlékezzünk a tavaly tanultakra!) Nem szükséges, hogy a teljes címtartományhoz tarozó kontextus egy időben benn legyen, akkor is tud futni a processz. A másik: bár a lehetséges címtartomány igen nagy, valójában a processzek virtuális címtartománya ennél kisebb szokott lenni. Nem könnyű olyan programot írni, aminek a kódrésze pl. óriási, sok-sok programsort kell ahhoz leírni. Az adatrészek is akkor lesznek nagyok, ha pl. óriási méretű táblázatokkal dolgozunk, különben nagyon sok egyedi változót kellene deklarálnunk. Valójában tehát három dologról beszélünk:

- egyik a lehetséges virtuális címtartomány. Ez valóban nagy lehet, hiszen pl. 32 bites címzésnél ez 4 Gbyte.
- A másik a processz tényleges virtuális címtartománya. Bár ez is nagy, biztos kisebb az előzőnél. A nagysága a másodlagos memória (a page terület, vagy a swap terület) becsléséhez szükséges, lássuk be, hogy valahol mégiscsak kell tárolni a processzek kontextusát! A processz tényleges virtuális címtartománya nem feltétlenül folytonos. Másrészt a processzek életük során növelhetik, vagy csökkenthetik virtuális címtartományukat is.
- A harmadik a fizikai címtartomány. Ez sajnos, kicsi szokott lenni, még ha ma egy munkaállomás központi memóriája szokás szerint legalább 32Kbyte, vagy több. Szuperszámítógépek központi memóriája lehet 1-2 Gbyte is! Ebből valamennyi *permanens*, azaz nem lapozható/söpörhető ki: pl. a kernel részei, amik mondjuk éppen a memóriamenedzseléshez kellenek. Mindenképp kicsi ez a memória, hiszen ezen osztozik a kernel permanens részén kívül minden processz.

Attól függően, hogy a dinamikus címleképzésben a leképzett memóriablokk mérete fix, vagy változó, beszélhetünk lapozós (paging), vagy szegmentálós rendszerekről.

7.3. A lapozó rendszerek (Paging Systems)

A lapozós rendszerekben fix méretű blokkokban történik a címleképzés, szükség esetén a kibemozgatás.

A virtuális memória egyforma méretű *lapokra* (page) van felosztva. Egy virtuális cím

$$v = (p,o)$$

formájú, ahol: p a lap címe, o (offset) eltolás a lapon belül.

Feltéve, hogy a processz virtuális címtartománya egydimenziós (pl. egy régióba tartozik) és ezen belül folyamatos, a V címtartomány 0 – n közötti bájtokban mért címeket tartalmazhat. Ugyanez a címtartomány lapokban is mérhető: ekkor 0 – p_n lapcímekkel rendelkezik. Az o (offset) értéktartomány természetesen át kell fedje a lapot.

A valós memória a lapokkal egyező méretű *lapkeretekre* (page frame) van felosztva. A valós cím

$$r = (p',o)$$

formájú, ahol p' a lapkeret címe, o az eltolás a lapkereten belül.

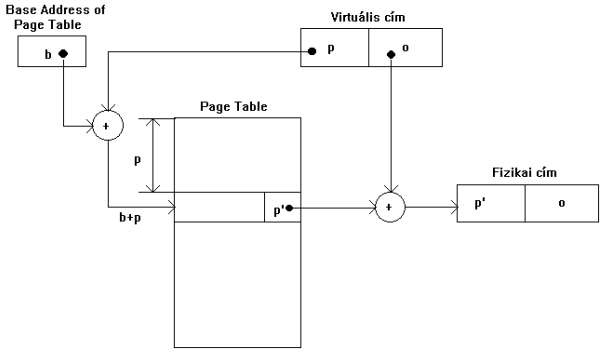
7.3.1. A laptáblák (Page Map Table)

Minden egy dimenziós címtartományú processz számára biztosítandó egy *laptábla* (több régiós processzek számára régiónként szükséges egy-egy laptábla). A laptábla kezdő címét a processz dinamikus kontextusában egy regiszter tartalmazza (Base Address of Page Table Register). A laptábla egy-egy bejegyzést tartalmaz egy-egy laphoz: tehát olyan hosszú, hogy

a processz minden lapjához lesz egy bejegyzése. A hossza tehát a processz tényleges bájtban mért virtuális címtartományából és a lapok méretéből kiszámítható.

A laptábla egy bejegyzése rögzít:

- egy jelzést, hogy van-e a laphoz lapkeret rögzítve, a lap benn van-e a fizikai memóriában, érvényes-e (valid/present-absent bit).
- Védelmi maszkot, azaz, hogy a lap írható, vagy csak olvasható. A bejegyzés a védelem mellett a kilapozásnál is használható: nem írt lapokat ugyanis nem kell feltétlenül kilapozni, egyszerűen eldobható, hiszen nincs rajta változás.
- Módosítás jelzőt. Az írható lapoknál lehet jelentősége: ha nem módosították, szintén nem kell kilapozni, csak eldobni. (modifed bit, dirty bit).
- Végül a lapkeret címét. Az érvényes lapoknál ebből a mezőből vehetjük a címet.
- Egyes rendszerekben leírót a másodlagos tárolóhoz (cím és típus). Belapozásnál hasznos ez az információ (egyes rendszerekben ez a lapkeret cím mezőben van valójában).



7.6.ábra. Címleképzés lapozós rendszerben

Ezek után a dinamikus címleképzést lapozás esetén a 7.6. ábra mutatja be. A virtuális címből veszik a lapcímet és összeadják a laptábla kezdőcímével. Az így kapott „cím” a laptábla p-edik lapjához tartozó bejegyzést címezi. (Úgy is mondhatjuk, a p lapcím indexeli a laptábla bejegyzést). Veszik a laptábla bejegyzést és az állapotbitek szerinti ellenőrzéseket végeznek. Ezekből a legfontosabb a címleképzés szempontjából az érvényesség ellenőrzése (valid/ present-absent bit ellenőrzés). A mennyiben érvényes a lap, veszik a laptábla bejegyzésből a p' lapkeret címet, továbbá a virtuális címből az o eltolás értéket, és képezik ezek összegét: ez az eredményül kapott valós cím, ami kiadható a buszra.

Belátható, hogy a két „összeadás” (lapcím és laptábla kezdet, illetve lapkeret cím és eltolás) hardveres „segítséget” igényel, e nélkül a címleképzés gyorsasága nem lenne megfelelő (itt is látható a az architektúra függőség).

7.3.2. A laphiba (Page Fault)

Ha a dinamikus címleképzés során a laptábla p-edik bejegyzésében a valid/present-absent bit azt jelzi, hogy a kérdéses laphoz nincs lapkeret hozzárendelve, kivételes esemény, *laphiba* következik be.

Emlékezzünk a kivételes eseményekre (exeption condition)! Ezek egy gépi instrukció során következnek be, lekezelésük után a kérdéses instrukció újból végrehajtódik. Nos, így van ez laphiba esetén is. A laphiba lekezelő rutin a másodlagos tárolóról "belapozza" (paging in) a kérdéses lapot. Ehhez keres egy szabad lapkeretet, és oda lapozza be. Ha nem talál szabad lapkeretet, akkor kiválaszt valamilyen, különben érvényes laphoz tartozó lapkeretet, azt „kilapozza”, egyben a lapkerethez tartozó lap laptábla bejegyzésében érvénytelenséget jegyez fel. Az így felszabadult lapkeretbe most már belapozhatja a kérdéses lapot, egyben érvényessé is teszi. A laphiba kezelő ezután visszatérhet: a laphibát okozó instrukció most újból végrehajtva már hiba nélkül leképezheti a címet. A „kilapozza” szót az előbb több okból tettük idézőjelbe. Egy nem módosított lap kilapozása ugyanis nem jár tényleges kiírással (azaz jóval

„olcsóbb)! Másrészt egyes rendszerekben a laphiba kezelő maga nem lapoz ki, hanem a kilapozásra kiválasztott lapot-lapkeretet valamilyen listára helyezi (valamely más processzre hagyva a tényleges kilapozást), és a szabad lapkeretek listájáról választ belapozáshoz lapkeretet.

Lássuk be, bár a jelenség megnevezése a laphiba, egészen normális dologról van szó, természetes dolog, hogy laphibák keletkeznek. Legfeljebb az lehet baj, ha igen gyakori egy processz számára a laphiba, mert akkor a vezérlés menete nagyon lassan halad előre, szinte csak ki-belapozással foglakozik a rendszer!

Most már minden alapkonstrukciót ismerünk. Kérdések illetve követelmények fogalmazódnak meg bennünk:

1. A processzenkénti legalább egy (esetleg több) laptábla mérete gond lehet. Hogy lehetne azt csökkenteni, vagy a nagy méretet kezelni?
2. A címleképzést segíti a hardver, de mégis jó lenne azt gyorsítani. Lehetséges ez?
3. A laphiba kezelő is gyors kell legyen. Itt jó kilapozási algoritmusokat kell találni.
4. Végül egy processznél a laphiba gyakoriság is gond lehet. Hogy tudnánk a processzek szükséges lapjait benntartani a memóriában, hogy ne legyen nagy a Page Fault ráta?

7.3.3. Laptábla méret kezelés

A laptábla méretét a processz mérete és a lapméret határozza meg. Természetesen írhatnánk kisebb programokat is, de most ne így keressük a megoldást.

A jól megválasztott lapméret természetesen egy jó lehetőség. Sajnos, néha ebben a hardver korlátoz: emlékszünk, hogy a memóriamenedzseléshez erős hardver támogatásunk is van, néha a hardver nem engedi, hogy a rendszergazda nagyobb lapméretet válasszon, ezzel a laptábla méretét csökkentse.

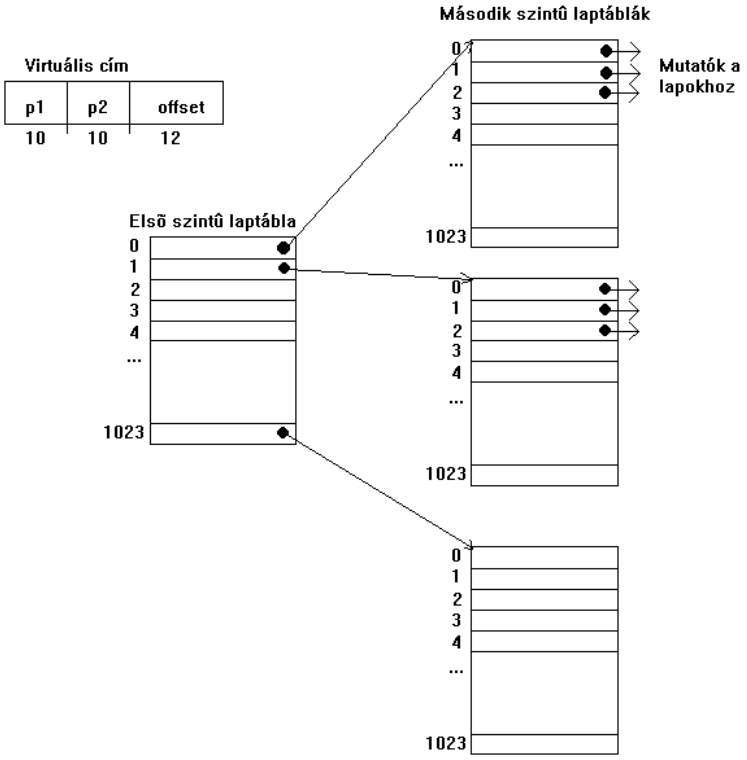
Tipikusan ilyen a helyzet a VAX processzoroknál: ott 512 bájtosak lehetnek a lapok, ezzel a 32 biten képezhető 4 Gbájtos címtartományt 8 Mbyte bejegyzés-számú laptáblával lehet átfogni.

A **VAX megoldás** erre a gondra a következő: Minden processz a laptábláját a saját címtartományának 2 Gbyte és 3 Gbyte közötti részében (ez az ún. System címtartomány) tartja, és maga a laptábla is kilapozható. A System címtartomány laptáblája permanens (maga a System terület azonban nem permanens!), és ennek címét minden processz egy MMU regiszterben tartja. A saját laptáblájának a címét egy másik MMU regiszterben feljegyzi. Mikor egy processz egy virtuális címét le kell képezni, veszik a címből a virtuális lapszámot (ez itt 21 bit hosszú), a p-t, és eltolják 2 bittel (mert a laptábla bejegyzés 4 byte hosszú). A kapott érték és a processz laptábla kezdőcím (MMU regiszterből) összege egy *A* virtuális cím, valahová a 2 G és a 3 G közé mutat: ahol is a processz laptáblája kezdődik. Ezt az *A* virtuális címet megnézik, vajon érvényes -e, a lapja bennvan-e. Ha nincs, előbb a laptáblát be kell lapozni. Feljegyezhető ez a cím a másik MMU regiszterbe, és az igazi címleképzés végre is hajtható. (Összegezve, a VAX egy címleképzés során kétszer is lapozhat, előbb a saját laptábláját lapozza be, majd a tényleges lapot, amire a hivatkozás történt.)

Más processzorok más megoldásokat kínálnak.

Többszintű laptáblák.

Képezzük a 32 bites virtuális cím lapcímrészét két 10 bites mezőből, és 12 bites eltolásból (7.7. ábra). A p1 mező index az első szintű laptáblában, aminek a mérete most már csak 1024 bejegyzés.



7.7. ábra. Többszintű laptáblák.

Egy bejegyzésében mutató van a második szintű laptáblák kezdőcímére. A második szintű laptáblákat a p2 címmező indexeli: a p2 és az első szintű laptábla pointerének összege a második szintű laptábla egy bejegyzésére mutat. A második szintű laptáblák bejegyzéseiben vannak a lapkeret címek: ezek összeadva az eredeti virtuális cím 12 bites offset mezejével a valós címet adják. Vegyük észre, hogy a laptáblák összmérete kisebb lehet! Adott esetben nincs is szükség 1024 darab második szintű laptáblára, hiszen a processz virtuális memóriájának valószínűleg kisebb a mérete, mint a lehetséges címtartomány!

Többszintű laptáblákkal dolgozik pl. a SUN SPARC processzora. Itt háromszintű laptáblák vannak, 8,8,6 bites lapcím mezőkkel, 12 bites eltolás mezővel. A Motorola 68030-as processzora négyszintes laptáblájú. Kétszintes laptáblákat (10, 10, 8 bites mezőkkel) támogatnak az Intel processzorok.

Az invertált laptáblák

A Hewlett Packard és az IBM System 38-as rendszerek ilyen megoldással "csökkentik" a laptáblák méretét.

Az elgondolás az, hogy a címzés bitszélesség növekedésével nem tud versenyt tartani a laptábla méret csökkentés, hiszen már vannak 64 bites címszélességű, sőt, 128 bites címszélességű rendszerek: Hiába "többszintesítünk", óriási méretűek lesznek a laptáblák. A fizikai memória mérete viszont nem növekszik ilyen gyorsan (sajnos), ezért jobb kiindulni a lapkeretektől.

Az *invertált laptáblákban* lapkeretenként vannak bejegyzések: méretük akkora, hogy *minden lapkeretnek legyen egy bejegyzése*.

Egy bejegyzés itt az állapot és védelmi maszkon kívül tartalmazza, hogy mely processz (pid) mely lapja (p) van pillanatnyilag a lapkeretben. A leképzéshez meg kell nézni az invertált tábla bejegyzéseit, hogy az adott processz adott lapja benn van-e egy lapkeretben. Ez a "megnézés" természetesen nem lineáris keresést jelent, a keresés gyorsítására hash (hasításos)eljárást alkalmaznak.

Ha az adott processz adott lapja egy lapkeretben megtalálható, a címleképzés sikeres. Ha nem, laphiba következett be, az kiszolgálandó, valami kilapozandó, a kért lap belapozandó.

Az olyan rendszerek, amelyek invertált laptáblával dolgoznak, mindig rendelkeznek az MMU-ban asszociatív gyorsító tárral is. Ezzel el is jutottunk a következő kérdéskörhöz: hogyan lehet tovább gyorsítani a címleképzést.

7.3.4. Címleképzés gyorsítása asszociatív tárral. (Translation Lookaside Buffer, TLB)

Az ilyen rendszerek MMU-jában létezik egy kisméretű asszociatív tár, egy tábla (TLB), a következő bejegyzésekkel:

- virtuális lapcím,
- érvényesség bit,
- módosítás bit,
- védelmi maszk (rw bitek),
- lapkeret cím.

Láthatólag ezek kissé bővítve ugyanazok az információk, amelyek egy szokásos laptáblában is rögzítettek szoktak lenni. Valóban azok, de most egy asszociatív tárbeli bejegyzésben, mely tárra jellemző a tartalom szerint - a bejegyzéseiben párhuzamos - keresés. Vagyis mikor egy gépi instrukcióbeli virtuális címet le kell képezni, a virtuális cím lapcímét párhuzamosan, egyszerre minden TLB bejegyzésben keresi az MMU. Egyben nézi, érvényes-e (valid), milyen a védelme. Ha találata van, azonnal adódik a lapkeret címe! Elmaradhat tehát a hosszadalmas, a laptáblákat kezelő leképzés.

A TLB bejegyzések száma - architektúrától függő – de nem túl nagy. A programok lokalitása mégis valószínűsíti a találatot. Ha azonban nincs találat az asszociatív tárban, akkor megy a szokásos laptáblás keresés: kikeresik a laptábla bejegyzését, ott nézik az érvényességet, laphibát generálnak, ha szükséges. Ha a szokásos laptáblás leképzés során nincs laphiba, az asszociatív tár egy bejegyzését „frissítik” a megfelelő adatokkal; arra gondolva, hogy a lokalitás miatt hamarosan újra szükség lesz a lapra. Ha itt, az asszociatív tárban nincs hely, akkor valamit innen "kivág", ilyenkor persze néznie kell a módosítás bitet, szükség esetén a rendes laptábla bejegyzésébe is be kell írnia a módosítás tényét.

Gondot jelent persze, hogy a szokásos laptábla felkeresés során bekövetkező laphiba esetén szükséges a kilapozás. A kilapozás esetleg az asszociatív tárban is igazítást igényel. Ilyenkor szokás szerint teljesen újratöltik az asszociatív tárat a közönséges laptáblák adataiból.

Lássuk be, az asszociatív tár nem helyettesíti a szokásos rendes laptáblákat kezelő mechanizmusokat! Azok tehát megvannak, akár invertált formában, akár normál formában, az asszociatív tár alkalmazása csak további gyorsítást eredményezhet, elsősorban a programok lokalitására építve.

Asszociatív tárral rendelkeznek a MIPS cég R2000, R3000 stb. processzorai, így működnek tehát a mi Indigóink is. Van egy kisméretű asszociatív tára az Intel processzoroknak is.

7.3.5. Kilapozási algoritmusok (Page Replacement Algorithms)

A laphiba kezelő gyorsítás is szerepelt, mint a megoldandó feladat. Tulajdonképpen itt két problémával kellene foglalkozni:

- meg lehetne-e mondani előre, mely lapokra lesz a közeljövőben szükség;
- mely lapokat lapozzuk ki, ha a lapkeretek "elfogytak".

Sajnos, az első probléma megoldhatatlan. Voltak ugyan kísérletek, melyekben próbafuttatás során feljegyezték a lap igénybevételi sorrendeket, és soron következő futtatásoknál ezt a sorrendet használták az előre való belapozás vezérlésére, de használható megoldás ebből nem

születhetett. Általános célú rendszereknél az ilyen optimálás teljességgel lehetetlen. A probléma megoldását hagyták az igény szerinti lapozási, majd a munkakészlet (working set) koncepció stratégiákra, ezeknél jobbat nem sikerült eddig csinálni (e két problémakörre visszatérünk).

Azon viszont érdemes gondolkodni, mik legyenek a kilapozandó lapkertek. Itt találhatók reális, megvalósítható algoritmusok.

Mielőtt ezekre rátérnénk, általános megfontolások is hozhatók. Olyan fogalmakat fogunk használni (és megkülönböztetni), mint a lapok belapozási ideje (vagy sorrendje), a lapok hivatkozási ideje (vagy sorrendje), a lapok hivatkozási gyakorisága (hivatkozások száma). Azok a lapok, melyek nem írhatók, vagy ha írhatók is, nem módosítottak, kezelhetők külön a kilapozás során. Ezeket valójában nem szükséges ténylegesen kilapozni, mert tartalmuk nem változott, elegendő őket "eleresztени". A következőkben ismertetett algoritmusokat ezek a tények határozzák meg.

A FIFO kilapozási algoritmus

Ennél a belapozási sorrend a meghatározó; minél régebben lapoztak be egy lapot, annál esélyesebb a kilapozásra. Az operációs rendszer ehhez az algoritmushoz a lapokat láncolt listán tartja, a lista elején a régebben belapozott lapokat, a végén a legfrissebben belapozott lapot. Kilapozni mindig a lista elejéről választ lapot. A mögöttes elgondolás az, hogy a "régi" lapokra már nincs szükség.

Amellett, hogy a lista tárolás elég erőforrás-igényes, az elgondolás is téves. egyáltalán nem biztos az, hogy a régen belapozott lapra nincs a következőkben szükség. (Igaz ugyan, hogy ha szükség van, akkor belapozódva, a lista végére kerülve sokáig nem fog kilapozódni, addig, míg újra a lista elejére nem kerül, de lehet hátrányos az elgondolás.)

Második esélyes FIFO

Tartsuk a lapokat körkörösen láncolt listán, az egyszerű soros lista helyett, és tartsunk nyilván a lapokra egy hivatkozás bitet. A körkörös listához tartozzék egy "óramutató", valójában a körkörös lista elejét mutató pointer. Látjuk majd, hogy az óramutató a listán „körbejárhat”.

A hivatkozás bit itt akkor billen be, ha tényleges hivatkozás volt a lapra.

Amikor kilapozásra van szükség, vizsgáljuk azt a lapot, amire a körkörös listán az óramutató éppen mutat (azaz a lista „elején” lévő lapot). Ha ennek a lapnak a hivatkozás bitje bebillentett állapotú, ne lapozzuk ki, hanem töröljük a hivatkozás bitjét, és léptessük a listán tovább az óramutatót. Ezzel a lap rajtamarad a listán, tulajdonképpen adtunk neki egy második esélyt: ha az óramutató a körön körbejárva újra rámutat, és közben nem volt hivatkozás a lapra, akkor az menthetetlenül ki fog lapozódni. Ha a mutatott lap hivatkozás bitje törölt, akkor viszont a lap azonnal ki fog lapozódni, helyébe a kívánt lap belapozódik (azaz felvevődik a körkörös listára, hivatkozási bitje bebillentett állapotú), az óramutató pedig továbblép a körön.

(Ezt az algoritmust szokás *óra* (clock) *algoritmusnak* nevezni, nem véletlenül.)

Mostanában nem használatos lapok (Not Recently Used pages) NRU (Least Recently Used) LRU

A programok lokalitásának elvéből kiindulva azok a lapok lehetnek kilapozásra esélyesek, melyeket mostanában nem használtak. A kérdés az, hogyan tudjuk ezt a tényt rögzíteni? Va-

lamilyen módon nyilván kell tartani, hogy mikor használták legutóbb a lapot. Vagyis nem azt rögzítjük, mikor lapozódott be, azt sem, hogy milyen gyakran hivatkoztak rá, hanem azt, hogy mikor hivatkoztak rá, esetleg azt is, mikor módosítottak a lapon legutóbb. Az NRU lapok kilapozódhatnak, az LRU lapok lehetőleg nem: az NRU és LRU algoritmusok igen közeli rokonok, majdhogynem ugyanazok.

Nem egyszerű és nem is olcsó a megvalósítás! Minden lap használatának (és módosításának) időbélyege tárterület-igényes, a "rendezés", vagyis a régen használt lapok kikeresése nem könnyű!

Az igazi megoldáshoz kétszeresen láncolt listán tartják nyilván a lapokat, a lista elején a legutóbb használtat. A lista végéről lapoznak ki szükség esetén. A lista közepén szereplő lapot, ha hivatkoznak rá, mozdítani kell a lista elejére.

Közelítő és sokkal "olcsóbb" megoldás a következő:

Legyen minden lapra nyilvántartva 2 biten a

- R - referenced - hivatkozott jelzés,
- M -modified - módosított jelzés.

Az M bit bebillen, ha módosítás történt a lapon.

Az R bit minden óramegszakításnál billenjen:

R = 1, ha az előző időszelet alatt hivatkoztak a lapra,

R =- 0, ha nem hivatkoztak.

Ezzel a lapok 4 osztályba eshetnek:

Osztály	R	M
0	0	0
1	0	1
2	1	0
3	1	1

Az NRU algoritmus szerint a legalacsonyabb nem üres osztályból véletlenszerűen (vagy FIFO sorrend szerint) kiválasztva lapozzuk ki a lapokat. Ebben implicite benne van az, hogy a nem módosítottak "lapozódnak ki" inkább. A módszer legnagyobb előnye, hogy egyszerű, könnyen megvalósítható, nem foglal sok helyet a nyilvántartás.

A fenti közelítő megoldást javíthatjuk: a hivatkozásokat nemcsak az utolsó óraintervallumban, hanem egy adott - nem hosszú - visszamenőleges időintervallumban nyilvántartva további "rendezést" vihetünk be.

Például 8 biten (1 bájton) tartsuk nyilván a hivatkozásokat minden laphoz. Adott időintervallumban minden lap referencia-bájtját léptetjük 1 bittel jobbra (jobb szélső bit kicsordulva elvész), a hivatkozott lapoknál 1-et, a nem hivatkozott lapoknál 0-t beléptetve a bal szélső bitre. Ezzel tulajdonképpen nyilvántartódik az utolsó 8 időintervallumban a lapok "hivatkozási históriája", egy rendezés a mostanában (az utolsó 8 időintervallumban) legkevésbé, vagy leginkább használt lapokra. A kisebb bájt érték a kilapozásra esélyes lapokat (mostanában nem használt lapok), a nagyobb bájt-érték mostanában inkább használt lapokat jelez.

Mostanában legkevésbé használt (Least FrequentlyUsed) LFU algoritmus (Not Frequently Used) NFU

Az alapgondolat: az utóbbi időkben gyakrabban használt lapoknak legyen nagyobb esélyük a bennmaradásra. Nem a lapok hivatkozási ideje, sorrendje, hanem a *hivatkozási frekvencia* a rendezőelv. Nagyon jó a gondolat, csak elég drága lehet a megvalósítása. Lehetne pl. láncolt listán tartani a lapokat, elején (vagy éppen a végén) azt, amire leggyakrabban hivatkoztak, de nagyon időigényes volna a lista karbantartása! Nem is a felfűzés a listára, hanem a lista közepéről a levétel, egyáltalán a lap megtalálása a listán gondot jelent.

Más megoldás is szóba jöhet persze, pl. minden lapra nyilvántartunk egy számlálómezőt, amit növelünk, ha a lapra hivatkozás történt. A kilapozásra esélyesek azok a lapok, melyeknél a számlálómező kicsi. Rögtön jelentkezik két gond. Egyik: elég nagy számlálómezőt kell biztosítani, pl. a laptábla bejegyzésében, ami növeli a laptábla méretét (emlékszünk, ez probléma volt már). A másik a processzek időszerűségének változásával függ össze: lehetnek lapok, melyeket gyakran használtunk a múltban, mostanában viszont nem használatosak. Amíg a többi lap számlálója "felnövekszik" erre a szintre, addig nem tudnak becsületesen versenyezni! Ha kilapozódnak, belapozódnak, újra kezdik a számlálást. Lehetnek anomáliák ebből.

A tiszta LFU, NFU ezért nem is használatos.

A számlálómezős megoldást lehet azonban alkalmazni, ha ráteszünk még egy "öregedés" (aging) algoritmust. Növekszik a számlálómező minden laphivatkozással, az óraintervallumokban azonban célszerű *aging* konstanssal beszorozva öregbítjük. Így az időszerűségüket elvesztő lapok számlálómezeje csökken, esélyt adva a frissebb lapoknak a versenyre.

7.3.6. Igény szerinti lapozás és a munkakészlet modell

A hivatkozási lánc fogalma (Reference String)

A processzek futásuk közben memóriahivatkozási sorozatot generálnak. Minden memóriahivatkozásnak megfelel egy specifikus virtuális lapcím. A processzek memóriahivatkozásai tehát jellemezhetők virtuális lapcímek sorozatával: ezt a lapcím sorozatot nevezzük *a processz hivatkozási láncának*, azaz *reference string*-nek. A hivatkozási lánc tehát lapcím lista, a processz a futása közben e lista szerinti sorrendben igényli a lapokat.

A lapozó rendszerek jellemezhetők a

- a processzek hivatkozási láncával;
- a kilapozási algoritmussal;
- a lapkeretek számával.

Gondot jelent persze, hogy a processzek hivatkozási láncát nehéz előre megjósolni (pedig jó lenne előre belapozni a közeljövőben szükséges lapokat, vagy nem engedni kilapozni azokat, még ha a kilapozó algoritmusunk ilyen döntést is hozna); és gondot jelent az is, hogy több processz élhet egy időben.

Az igény szerinti lapozás (Demand Paging)

A lapozás legegyszerűbb és legtisztább formája szerint, amikor egy processz indul, egyetlen egy lapja sincs a memóriában. Amint a CPU be akarja hozni az első instrukciót, bekövetkezik a laphiba, aminek hatására a kernel behozza az első lapot a hivatkozási láncból. További laphibák generálódnak (pl. a globális változók miatt, a verem miatt), melyek hatására a hivatko-

zási lánc egyre több lapja belapozódik. Egy idő után a processz elegendő lapja benn van a memóriában, a laphiba gyakorisága csökken. Ezt a stratégiát nevezik *igény szerinti lapozásnak* (Demand Paging), hiszen egy lap csak akkor kerül be, ha igény merül fel rá, előre nem lapozunk a stratégia szerint. A stratégia kidolgozói remélik, hogy dinamikus egyensúly állhat be a processzek bennlévő lapjai és a felmerülő igények között.

A munkakészlet (Working Set) fogalom

Egy processz azon lapjainak összessége, melyeket egy adott "pillanatban" használ a processz *munka-lapkészlete* (Working Set). Ha minden lapja a *munkakészlethez* tartozik, nem következik be rá laphiba. Ha a fizikai memória kicsi, a munkakészlethez kevesebb lap tartozhat, ekkor be fog következni előbb-utóbb laphiba. A processzek futásuk közben jellemezhetők a "pillanatnyi" *laphiba-gyakorisággal*, a Page Fault rátával. A fent említett igény szerinti lapozásnál kezdetben nagy a laphiba gyakoriság, és remény szerint később ez csökken.

Lássuk be, ez a "pillanatnyi" fogalom meglehetősen ködös fogalom. A laphiba gyakoriság nem egy pillanatnyi helyzetet jellemez, hanem egy idő intervallumot, a pillanatnyi (current) időtől visszaszámítva valamennyi órajelnyi időt, egy idő-ablakot. Esetleg a munkakészlet úgyis megadható, hogy a processz hivatkozási láncának egy része ez, visszamenve a láncon valameddig. Nem nehéz megoldani, hogy a rendszer tartsa nyilván a processzek munkakészletét, a hivatkozási láncukat valameddig visszamenve. (Persze, ehhez megfelelő kilapozási algoritmus is tartozik!)

A Working Set modell (Dennis, 1970)

Tartsuk nyilván és menedzseljük a processzek munkakészletét. Biztosítsuk, hogy ez a memóriában legyen, mielőtt a processzt futni hagyjuk. Ez a koncepció csökkenteni fogja a laphiba gyakoriságot. Előre való belapozás (Prepaging) is megvalósítható: lapokat előre belapozunk, mielőtt a processzt futni engedjük.

Kérdések sora merül fel bennünk persze, mi is valójában a munkakészlet (lapok vagy lapkeretek készlete), mekkora legyen egy-egy processz munkakészlete, milyen legyen a kilapozási stratégia ekkor stb.

Lokális és globális stratégiák.

Előzőekben már beszéltünk a kilapozási algoritmusokról. A munkakészlet modell kapcsán felmerül bennünk, hogy ha egy processz egy új lapját kell belapozni, és emiatt kilapozásra is szükség van, akkor a kilapozás ugyanennek a processznek a munkakészletéből történjen (lokális stratégia), vagy más processzek munkakészlete is figyelembe vevődjön (globális stratégia).

A *lokális stratégia* annak az elgondolásnak felel meg, hogy minden processz kap valamennyi (rögzített számú) lapkeretet, amivel gazdálkodhat. Itt, ha a processz lapkeret-készlete nő, a laphiba gyakorisága csökkenni fog és fordítva. A *globális stratégiában* a processzeknek nincs rögzített lapkeret számuk, az mindig változik.

Egészítsük ki a lokális stratégiát a következőképpen: a processzek ne rögzített számú lapkeretet kapjanak, hanem azt változtassuk bizonyos határok között. A változtatáshoz használjuk a laphiba gyakoriságot, ami mérhető. Ha nagy a laphiba gyakoriság, növeljük a lapkeretek számát, ha alacsony a laphiba, akkor csökkentsük. Újradefiniáljuk a munkakészlet fogalmat is: a *pillanatnyilag processzhez rendelt lapkeretek készletét*, nevezzük ezentúl munkakészletnek. A "pillanatnyilag" benn lévő lapok teljesen kitöltik a munkakészletet. A laphiba gyako-

risággal tehát bizonyos lapkeret egységekkel növeljük a munkakészletet, persze csak egy adott határig, illetve csökkentjük bizonyos lapkeret számokkal, megint csak egy alsó határig. A processzek munkakészletei között így beállhat egyensúly, egymás rovására növelik, csökkentik a készleteiket, attól függően, hogy milyen a laphiba rátájuk. Tulajdonképpen a laphiba rátákat tartjuk határok között.

Még egy gondolatot vessünk fel: ha sok processz él egy időben, mindegyiknek magas lehet a laphiba gyakorisága, mert a munkakészletek összessége (a Balance Set) nem növekedhet. Ilyenkor legjobb lenne egyes processzeket teljesen kisöpörni a memóriából, helyet adva a többinek a racionális futásra. A következtetésünk tehát az, hogy a lapozás mellet is jó volna a szegmentálás és ki-besöprési koncepció!

7.4. A szegmentálás

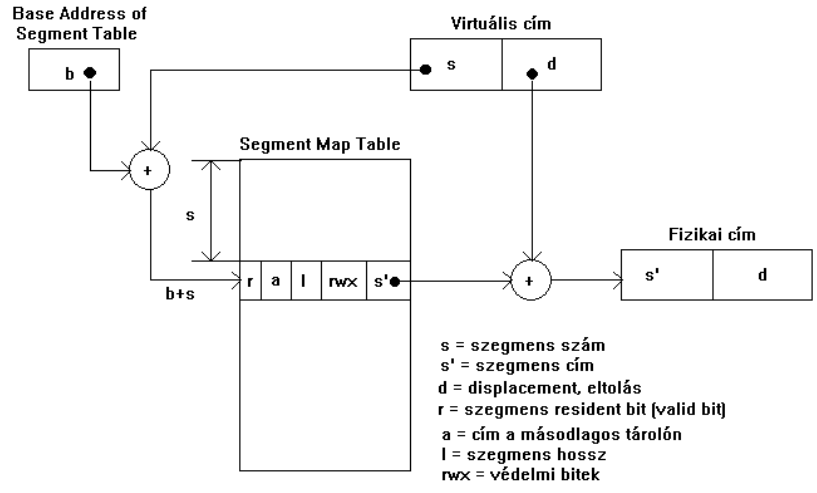
A virtuális memória - ahogy eddig tárgyaltuk - "egydimenziós" volt: a virtuális címek 0-tól mehettek valameddig, a címtartományban a címek követték egymást (lehettek ugyan a címtartományban "hézagok").

Sokszor jó lenne valamilyen "többdimenziós" címzési rendszer:

- külön címtartomány, 0 - valameddig, a program kódnak;
- külön címtartomány, 0 - valameddig, az adatoknak;
- külön címtartomány, 0 - valameddig, a vermeknek;
- külön címtartomány, 0 - valameddig, az osztott könyvtári rutinoknak stb.

Szegmenseket képzelünk el, ezeknek külön címtartományaik vannak, mindegyik 0-tól kezdődik, és persze nem egyforma méretűek. A védelmük is különböző lehet. Minden szegmens a címek lineáris sorozata, 0-tól különböző méretekig. A szegmensek hossza akár változhat is, növekedésük, csökkenésük azonban egymástól független.

A tiszta szegmensenkénti címleképzés



7.8. ábra. Szegmensenkénti címleképzés

Ez valamelyest hasonlít a laponkénti címleképzéshez. A laptábla helyett itt szegmenstábla van, és a leképzés során nem egyforma méretű blokkokban képzünk le, ezért a szegmenstábla soraiban a szegmens hossza is szerepel (7.8. ábra.)

A leképzés itt is dinamikus, instrukciónként történik, és az MMU itt is támogatja a leképzést. Ha a szegmens nincs a memóriában, be kell söpörni (swapping in). Ehhez szükség esetén helyet kell csinálni: más szegmensek kisöprődnek (swapping out). Belátható az is, hogy a szegmentálós rendszereknél kisebb gond a szegmenstábla méret. Valószínűtlen, hogy túl sok szegmensből álljon egy processz.

Milyen rendszerek lehetnek?

Vannak tiszta lapozó (pure paging) rendszerek.

Vannak tiszta szegmentáló, ezzel ki-besöprő rendszerek.

Vannak szegmentáló-ki-besöprő és ugyanakkor lapozó rendszerek is. Ekkor a ki-besöprés során gyakori, hogy a teljes kontextus ki-besöprődik, helyet biztosítva a többi processz számára a lapozáshoz, moderálandó azok laphiba gyakoriságát.

A kombinált rendszerekben a szegmenstábla bejegyzésében nem a szegmens címre mutat az s', hanem a szegmenshez tartozó laptábla kezdetére: itt tehát szegmensenként vannak a laptáblák.

Általában a felfüggesztett (suspended) processzek esélyesek a kisöprésre, vagy azok, amelyeknek hosszú idő óta nincs aktivitásuk. Lehetnek ilyenek bizonyos démon processzek, melyekhez hosszú idő óta nem jött kérelem, de lehetnek terminálhoz kötött processzek is, melyek már hosszú idő óta blokkoltak terminál inputon (pl. elment a felhasználó vacsorázni...)

7.5. A „swap” eszköz/fájl struktúra

A mai rendszerekben a kisöprési/kilapozási terület, a másodlagos tároló lehet egy partíció, vagy egy fájl valamilyen fájlrendszerben. Mindkettőt úgy foghatjuk fel, hogy lapméretű blokkok sorát tartalmazzák. A szóhasználat szerint – akár szegmensenkénti, akár laponkénti leképzésről van szó, akár fájl, akár partíció a másodlagos memória – vegyesen kilapozási fájlról, kilapozási eszközről, kisöprési fájlról vagy kisöprési eszközről beszélünk (swap/paging device/file).

Az ami érdekes, a kilapozási eszköz szabad területének menedzselése.

A klasszikus Unix esetét nézzük. Adott tehát a kilapozási partíció, ami 0 – n blokkok együttese. Ezen a 0-ik blokkon fenntartható a szabad területek térképe. A térkép bejegyzésekből áll, egy-egy bejegyzés cím+hossz érték-párral szabad területet jelez.

Egy processz memória allokálása során foglalni kell hozzá kisöprési területet. A foglalás first-fit stratégiával történik, a szabad terület elejére, egy bejegyzés „igazításával” (ami lehet egy bejegyzés eltüntetése is.

Nagyon fontos megjegyezni, hogy a processzek kódja és inicializált konstans adatai (ezek nem változtatható, nem írható részek) részére nem kell a kisöprési területen helyet biztosítani! Szükség esetén e szegmensek (vagy ezen részek lapjai) a végrehajtható fájlból (executable file, image file) behozhatók!

A memória felszabadítás során a kisöprési területet is fel kell szabadítani. Ez a felszabadítás a szomszédos szabad területek összefűzésével történik: ha előtte és utána is szabad területek vannak, a két bejegyzésből egy bejegyzést csinálnak; ha előtte vagy utána van szabad terület, a bejegyzést igazítják, megnövelve a szabad terület méretet; végül, ha előtte és utána is foglalt területek vannak, akkor beszúrnak új bejegyzést a térképen .

Érdemes tanulmányozni a Linux kisöprési eszköz struktúrát is! Itt is a szabad terület menedzselés az érdekes: nos, ez itt bit térképes a nyilvántartás!

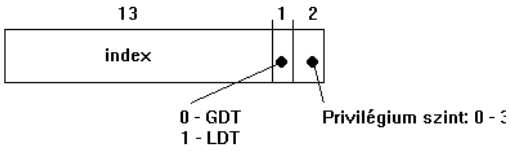
7.6. Szegmentáció és lapozás a 386/486/stb. processzornál

A híres 386/486-stb.-os processzornál 16K független szegmens lehetséges. Ennyire ritkán van szükség. Még egy előnyös tulajdonsága: egy-egy szegmens 10^9 32 bites szót tartalmazhat, ez már ki is használható.

A címszámításhoz a processzor két táblát tart fenn:

- LDT - Local Descriptor Table, processzenkénti tábla;
- GDT - Global Descriptor Table, egy van belőle, a rendszer címtartományhoz.

Az LDT segíti a processzek kód, adat, verem stb. leírását. A GDT a rendszert, a kernelt magát írja le.



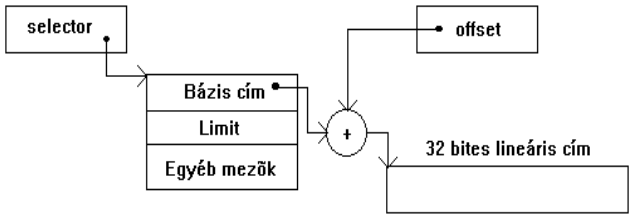
7.9. ábra. A szelektor felépítése

A processzornak van 6 szegmens regisztere. Ismerős a CS, DS, SS, ES stb. Ezek 16 bitesek, és egy-egy szegmens *kiválasztóját* (selector) tartalmazhatják. A *selector* felépítése az alábbi ábrán látható:

A szelektor 13 bites indexe belépési pont az LDT/GDT-be. (Ebből következően a ?DT táblák sorainak száma 213, és mivel egy-egy sorban 8 bájtos descriptor található, kiszámíthatjuk maximális méretüket.)

Az instrukciókban a címek formája: szelektor, eltolás párok:

cím = (selector,offset)



7.10. ábra. A lineáris cím kiszámítása

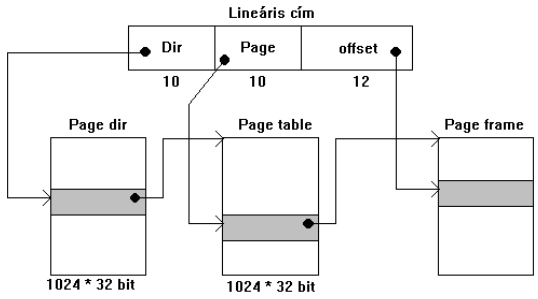
A dinamikus címleképzés során a selector érték betöltődik valamelyik, a megfelelő szegmensregiszterbe (pl. instrukció címzésnél a CS regiszterbe). Ugyanakkor a megfelelő leíró táblából (Descriptor Table) az indexelt leíró betöltődik a CPU mikroprogramjának egy regiszterébe (ezek után a *descriptor* gyorsan elérhető). (Most nem tárgyaljuk azt az esetet, amikor a szegmens ki van "lapozva", mindenesetre ekkor "trap" következik be, a szegmens belapozódik.) A 7.10. ábrán látható címleképzés történik ekkor.

A 8 bájtos descriptor szerkezete elég bonyolult. Benne a

- bázis cím 32 bit lehet, de a felülről való kompatibilitás miatt 16, ill. 24 biten is tud dolgozni.
- A limit mező max. 20 bites. Vagy ténylegesen a szegmens méretét tartalmazza bájtokban, vagy 4K-s lapjainak számát (ebből: a szegmens maximális mérete kiszámítható).
- Az egyéb mezőket nem részletezzük.

Amit nagyon fontos megjegyezni: az eddigi címszámítás eredménye egy 32 bites *lineáris cím*.

A processzornál lehet engedélyezett, vagy lehet letiltott a lapozás. Ezt a tényt 1 biten a globális kontroll regiszterben tárolják. A lapozás-letiltást a 286-oshoz való kompatibilitáshoz is használják.



7.11. ábra. A fizikai címszámítás

Ha a lapozás letiltott, a kiszámított lineáris cím a fizikai cím, kiadható a buszra.

Ha a lapozás engedélyezett, a lineáris cím virtuális címnek vevődik, ekkor folytatódik a címleképzés, méghozzá kétszintű virtuális cím - fizikai címleképzéssel (7.11. ábra).

Megjegyezzük még, hogy a processzornak van egy kisméretű asszociatív tára is (a Dir-Page kombinációkra), ami a virtuális-fizikai címleképzést gyorsítja.

Lássuk be a következőket:

- Megőrizték a felülről való kompatibilitást (Paging disabled).
- Lehetséges a tiszta lapozás (Minden szegmensregiszter ugyanazt a szelektort kapja. Engedélyezett a lapozás, mindig ugyanarra a laptáblára, működhet az asszociatív tár).
- Lehetséges a tiszta szegmentálás (Különböző értéket kapnak a szegmensregiszterek, de letiltott a lapozás).
- Lehetséges a szegmentálás-lapozás kombináció (Külön szegmensregiszterek, engedélyezett lapozás, asszociatív tár kihasználva).

7.7. A Windows NT memóriamenedzselése

7.7.1. Általános jellemzés

Az NT memóriamenedzselése virtuális, TLB-t is használó kétszintű laptáblás lapozós (ugyanekkor invertált laptáblához hasonló laptábla adatbázist is használó), munkakészletet kezelő (azt igazító), lokális másodesélyes FIFO kilapozásos algoritmussal rendelkező, igény szerinti belapozó.

7.7.2. Memória allokáció

Az NT taszkok (többfonalas processzek) 32 bit szélességen 4GB lineáris címtartományt látnak. Ennek felső 2GB-ja rendszer címtartomány (kernel módban elérhető), alsó 2 GB-ja a felhasználói módú címtartomány. A címtartományt 4 KB-os lapokra osztják: ez 12 bit szélességű eltolás (offset) értékkel lefedhető. A 4 GB címtartományt persze nem használja tejesen: vannak "lefoglalt" címtartomány szakaszok, melyeket a Virtual Address Descriptorok tartanak nyilván. Memória allokáció során éppen új címtartomány szakaszt vesz fel egy-egy taszk: új deszkriptor bejegyzéssel (kezdő-vég címmel), és csak amikor ténylegesen használnánk is ezt az új címtartományt (committing), akkor inicializálják a laptábla bejegyzéseket (kétszintű memória allokáció).

Különleges memória allokáció a "leképzett fájl" (Mapped File) objektum. Az objektumot beillesztik a létrehozó taszk virtuális címtartományába (ez a nézet: view), ezután a taszk úgy látja fájlt, mintha teljes egészében a memóriában lenne, betöltéséről és a módosítások lemezre írásáról a memóriamenedzser gondoskodik. Egy taszk több nézetet is létesíthet ugyanahhoz az objektumhoz, több taszk is készíthet saját nézetet megosztott objektumhoz. A "klasz-

szikus" közös memória pedig felfogható a leképzett fájl speciális esetének: az objektum mögött a lapozó fájl van.

Most jegyezzük meg a következőket is: a dinamikus címleképzés során először éppen a deszkriptorok segítségével az ellenőrződik, hogy cím lefoglalt memóriaterületre esik-e, vagy sem. Ha nem, azonnal keletkezik az érvénytelen címre hivatkozás kivételes esemény, nem kell végignézni a TLB-t, a laptáblázatokat a címleképzéshez. Úgy is mondhatjuk, a laptáblák már csak a lefoglalt címtartományokat tartják nyilván.

7.7.3. A címleképzés első szintje: TLB

A 32 bites lineáris cím virtuális cím. Leképzése a processzorban megvalósított, ezért processzszorfüggő "címleképzés megkerül táblázat" (Translation Lookaside Buffer) vizsgálatával kezdődik. A TLB, mint tudjuk, asszociatív áramkör, úgy képzelhetjük el, mint egy kétoszlopos táblázatot. egyik oszlopa maga a virtuális cím (annak 20 legnagyobb helyiértékű bitje), a másik pedig a hozzájuk tarozó laptábla bejegyzések. A címképzés során a processzor a cím legnagyobb helyi értékű 20 bitjét párhuzamosan összeveti az első oszlop bejegyzéseivel, és találat esetén azonnal kapja a laptábla bejegyzéseket. Ha nincs találata a TLB-ben, akkor indítja a szokásos laptábla keresési eljárást.

7.7.4. Kétszintű laptáblák, prototípus laptábla

- A lineáris cím szokásosan 2 szintű laptáblás megoldással képezhető valós címre. Az első szintű laptábla neve itt: *laptábla katalógus (page directory)*. A virtuális cím első 10 bitje indexeli: mérete tehát 1024 bejegyzés, egy bejegyzés 4 bájt. Minden taszknak ilyen méretű saját laptábla katalógusa van. Egy-egy bejegyzése egy-egy második szintű laptábla kezdőcímét tartalmazza. A második 10 bit indexeli a második szintű laptáblákat, ezek a neve egyszerűen: *laptábla*. Méretűk 1024 bejegyzés, és csak annyi laptábla tartozik egy-egy taszkhoz, amennyi szükséges. Egy-egy bejegyzés tárolja a lap státusz-védelmi bitjeit és *lapkeretet*, vagy pedig *prototípus laptáblát* (Prototype Page Table), vagy *másodlagos tárolót* indexel. A védelmi bitek:
- *Érvényes* (valid) lap (érvényes lap bejegyzése lapkeretet vagy prototípus laptáblát indexel, érvénytelené a lap helyét a másodlagos tárolón);
- *Csak olvasható* (read only) lap;
- *Teljes jogú* (read/write) lap;
- *Csak futtatható* (execute only) lap (csak speciális processzoroknál van értelme);
- Figyelt (guarded) lap (Elérésük kivételes eseményt generál. Pl. verem vége figyelhető így.);
- *Tiltott* (no access) lap (elérésük eseményt generál);
- *Zárolt* (locked) lap (Nem lehet kilapozni ezeket);
- *Módosításkor másolt* (copy on write) lap.

A státusz-védelmi bitek értelezéséből az utolsót kell tovább magyarázni. Ez a védelmi módszer az osztott memóriahasználatban a "halogató technika" egy formája. Vegyünk egy példát. Az NT tejesíti a POSIX taszk-kreáció előírást: a gyermek taszk kontextusa majdnem teljesen egyezik a szülőjével (v.ö. Unix fork), azaz a gyermek kód és adatszegmenseit látni kell a gyermek címtartományához rendelve is. Ha a gyermek nem módosítja kontextusát (kódját valószínűleg egyáltalán, sokszor az adatait sem), akkor a "másolás" fölösleges. Jobb megoldás, ha egyáltalán nem másolunk, hanem mindkét taszkban módosításkor másolt státusszal ellátott bejegyzésekkel a laptáblákban közös lapkeretekre hivatkozunk. Ha ezek után bármelyik taszk mégis módosít valamely megosztott lapon, a memóriamenedzser lemásolja számára a kérdéses lapot új lapkeretbe, és a laptáblázat bejegyzéseket felfrissíti, kitörölve a módosí-

táskor másolt bejegyzést, a másik taszkban meghagyva az eredeti lapkeret hivatkozást, a módosító taszkban feljegyezve a másolatra történő hivatkozást.

Szintén az osztott memóriakezelés problémaköréhez tartozik a nem halogatott memória megosztás: akár a leképzett fájl objektum, akár a klasszikus osztott memória objektum esete. Ez az NT a *prototípus laptábla* segítségével kezeli.

Ha úgy tetszik, a *prototípus laptábla* egy további szint a laptábla rendszerben. Több taszk által használt lapoknál (shared pages: egy lapkeretre több laptábla bejegyzés is mutatna) a laptáblák nem közvetlenül mutatnak a lapkeretre, hanem a prototípus laptáblán keresztül: ilyenkor ez a tábla tárolja a védelmi biteket, a státuszt. A fent már említett "leképzett fájl" objektumok, ezek speciális eseteként vehető klasszikus memóriaosztás kezelése történik a prototípus laptábla segítségével, szerencsére a programozó számára transzparensen.

7.7.5. A lapkeret adatbázis

A fizikai memória nyilvántartására (pl. a szabad lapkeretekkel való gazdálkodásra), a ki- és belapozás segítésére az NT memóriamenedzsere az invertált laptábla koncepció szerinti laptáblát, *lapkeret adatbázist* (Page Frame Database) is fenntart.

A lapkeret adatbázis is egy táblázat: lapkeretenkénti bejegyzésekkel. Mérete tehát a fizikai memóriától függ. Egy-egy bejegyzése információkat tárol a keretek állapotáról, valamint "visszamutatót" laptábla bejegyzésre (amiből kiderül, melyik taszk használja a lapkeretet, melyik lapját tárolva benne. Az állapotinformációk:

- *Érvényes* (valid) keret: használatban lévő keret, van benne leképzett lap;
- *Szabad* (free) keret: egy taszk sem használja, kioszthatóó. jegyezzük meg: ha egy taszk terminálódik, az általa használt keretek szabaddá válnak és ez fel is jegyződik a lapkeret adatbázisban.
- *Nullázott* (zeroed) keret: szabad keret, kitöltve 0-kkal. (A C2 biztonsági szabványnak megfelelő memóriakezelést tesz lehetővé: ne lehessen szabaddá vált lapokról információkat szerezni);
- *Készenléti* (standby) keret: tulajdonképpen felszabadított keret, de még megtalálhatók rajta a lapot-lapkeretet korábban használó taszk adatai. Az ilyen kereteket még "visszakérhetik" viszonylag olcsón: ha úgy tetszik második esélyt adva a keretnek-lapnak.
- *Módosított* (modified) keret: a készenlétihez hasonlóan már felszabadított (lemondott róla a taszk, vagy erőszakkal elvették tőle) keret, de újrafelhasználása előtt azt a lemezre kell írni.
- *Hibás* (bad) keret: megbízhatatlanul működő keretekre (vagy keretekre, melyeket ki akar vonni a gazdálkodásból) a memóriamenedzser ráírhatja ezt a bejegyzést.

A lapkeret adatbázisban a keretek státuszának feljegyzése mellett 5 *láncolt listán* is vannak nyilvántartások. Létezik a:

- szabad keretek láncolt listája;
- nullázott keretek láncolt listája;
- készenléti keretek láncolt listája;
- módosított keretek listája;
- hibás keretek listája.

Az NT memóriamenedzsere folyamatosan figyeli a szabad, a nullázott és a készenléti listán található elemek számát, és ha ez bizonyos érték alá csökken, a másodlagos tárolóra írja a módosított kereteket és a készenléti listára átteszi azokat. A módosított keretek mentését a

módosított lap író (Modified Page Writer) rendszertaszk végzi. Ha még így is kevés a szabad-, nullázott-, készenléti keretszám, további tevékenységek is történnek (taszkok munkakészletének csökkentése: trimmelés, keretek erőszakos felszabadítása stb., lásd később). Miután a fizikai memória nyilvántartásának kulcsa a lapkeret adatbázis, többprocesszoros rendszereknél külön gondoskodni kell ennek védelméről. Forgózár (spinlock) védi az adatbázist a kritikus szakaszokra, sorbaállás következhet be (hiába van több processzor), ezért memóriamenedzser ezzel kapcsolatos kritikus szakaszait nagyon hatékonyra kellet írni.

7.7.6. A címleképzés a kétszintű laptáblákkal

Tételezzük fel, hogy a processzor a TLB-ben való keresésben sikertelen volt, ekkor indítja a szokásos laptábla-rendszer szerinti leképzést. A taszkhoz tartozó lapkatalógust a cím első 10 bitje tartalmával indexelve kikeresi a megfelelő laptábla kezdőcímet, ezt a táblát indexelve a második 10 bittel kikeresi a laptábla bejegyzést. Itt a státuszt vizsgálva, ha az érvényes, veszi a mutatót a lapkeretre (vagy prototípus laptábla bejegyzésre). A lapkeret címből és az eredeti virtuális cím eltolás értékéből adódik a valós cím, kiadható a buszra. Közben a védelmek is kezelhetők, szükség esetén a lapkeret adatbázis módosítható.

Amennyiben a laptábla bejegyzés érvénytelen státuszú, kiemelhető belőle a kérdéses lap másodlagos tárolón való helyére utaló mutató és laphiba következik be.

7.7.7. A laphibák kezelése, kilapozás, munkakészlet kezelés

A laphiba kezelő egy-egy taszk számára végzi munkáját, nem nagy hiba tehát, ha a szóhasználatunkban nem a kezelőt, hanem a taszkot fogjuk rendre említeni.

Az NT memóriamenedzser minden taszk számára munkakészletet (working set) biztosít, minden taszk bizonyos számú lapkeretet kap. A készletnek van maximális és minimális értéke. A rendszerállapottól függően a készlet csökkenthet (automatic workig set trimming), növekedhet (magas laphiba ráta a taszkon és van elegendő szabad keret). Egy-egy taszk lokális FIFO kilapozási algoritmussal "gazdálkodik" a munkakészletével: szüksége esetén a legrégebben betöltött lapját "szabadítja" fel. A "felszabadítás" valójában a készenléti vagy módosított listára való áttétel: ez azt jelenti, hogy a gyakran használt lapokat a "felszabadítás" után azonnal vissza is kérheti, vagyis a lapok kapnak egy második esélyt ilyen módon. (Az igazi felszabadítást valójában a módosított lapíró processz végzi, szükség esetén.) Miután a taszk "felszabadított" keretet, a nullázott-, szabad-, készenléti listáról pótolja munkakészletét: választ keretet és abba belapozhatja lapját.

A virtuális memória kezelő tehát, ha úgy érzi, aktiválja a módosított lapíró processzt, ami a módosított státuszú kereteket kilapozza, utána azokat átteszi a készenléti listára. Ha ez sem segít, átnézi, van-e olyan taszk, melynek munkakészlete nagyobb, mint a taszkhoz tartozó minimális érték. Ha vannak ilyenek, ezeknek a munkakészletét csökkenti. ha ezután sincs elegendő memória, valamennyi taszkra elvégzi a kurtítást: kényszeríti a taszkokat a "felszabadításra". Ha a memóriakrízis megszűnik, az egyes taszkok laphiba gyakoriságát figyelve kezdi növelni azok munkakészlet méretét. Taszkok terminálódása esetén azok munkakészleteit megszünteti, a kereteiket szabad listára teszi. Taszkok születése esetén biztosít számukra munkakészletet.

7.7.8. Laphiba kezelés, belapozási algoritmus

Alapvetően szükség szerinti (demand paging) algoritmus szerint történik a belapozás, azzal a kis módosítással, hogy a lokalitás elvét is figyelembe véve a szükséges lapokat közrefogó (néhány) lapot is belapozzák egyúttal.

8. Az I/O rendszer, eszközök, másodlagos tárolók, fájlrendszerek

Az operációs rendszer I/O alrendszerének egyik feladata, hogy a felhasználók (alkalmazásfuttató, programozók stb.) elől elrejtse a hardver eszközök különbségeit, specialitásait, kényelmesen lehessen az eszközöket forrásként vagy nyelőként használni, az eszközökre, eszközökről adatokat továbbítani. Másik feladata az eszközök menedzselése, a processzek számára erőforrásként biztosítani az eszközöket, azok szolgáltatásait, esetleg ütemeznie kell az eszközökhöz való hozzáférést, védeni kell az eszközöket, konkurrens vagy kizárólagos hozzáféréseket megkülönböztetve, védelmi tartományokat nyilvántartva. Az I/O eszközök között kitüntettettek a blokkorientált (struktúrált, diszkes) eszközök. Ezek ugyanis másodlagos tárolóként használhatók, akár virtuális memória kisöprési, kilapozási területeként, akár fájlrendszer hordozójaként. Nézzük először, hogyan is "látjuk" különböző szemszögekből az eszközöket, a fájlrendszert.

8.1. Az I/O, eszközök, fájlrendszer különböző szemszögekből

8.1.1. A felhasználó látásmódja

A felhasználó az eszközöket és a fájlokat *szimbolikus neveiken* ismeri. A felhasználói kapcsolattartó rendszerben ezeket a szimbolikus neveket használja. Korszerű operációs rendszerekben a *hierarchikus fájlrendszert* lát. Ehhez ismeri a *jegyzék* (katalógus, directory) fogalmat, az *ösvény* (path) fogalmat, a *gyökér jegyzék* (root directory) fogalmat, *munkajegyzék* (working directory) fogalmat stb. A fájlrendszer "látásához" három dolgot ismer:

- fájlok együttesét;
- jegyzék struktúrát, ami információkat ad a fájlok csoportosítására;
- logikai eszközt (partíciót, diszket), amin a fájlrendszer elhelyezkedik.

A felhasználó számára a fájl a legkisebb egység a másodlagos tárolón, amit kezelni szokott (ritka, hogy a diszk struktúrát, a blokkokat, még ritkább, hogy oldalakta, sávokat, szektorokat kezeljen).

A kapcsolattartó felület segítségével képes kezelni az eszközöket, a fájlokat: *másolhat* (copy), *mozgathat* (move), *törölhet* (delete, remove) fájlokat, előállíthatja azokat valamilyen segédprogrammal, fejlesztővel stb. A kapcsolattartó parancsai magas szintű "utasításkészletet" biztosítanak az eszközök, fájlok kezeléséhez.

A felhasználó az eszközök, fájlok kezelésében ismeri az eszköz-és fájlvédelmi koncepciókat, a *tulajdonossági-* és *védelmi kategóriákat*, ezeket használja, beállítja stb. Lát egyéb *attribútumokat* is: pl. készítési, utolsó elérési, vagy módosítási dátumokat stb.

Bizonyos operációs rendszerekben a felhasználó lát *fájl szervezési módokat* (file organisation) is: azaz nemcsak a fájlneveket, a nevekhez kapcsolódó attribútumokat, a névhez tartozó adatokat, hanem a fájl struktúráját is. Ezekről az operációs rendszer nézőpontja tárgyalása során kicsit többet is szólunk.

Ez a látásmód a felhasználói látásmód.

8.1.2. A programozó látásmódja

A folyamat (process) szemszögéből minden I/O eszköz vagy fájl egy *csatorna* (stream). A csatornát a folyamat *megnyitja* (open, fopen, create stb. rendszerhívások): ezzel belső azonosítót rendel hozzá. Ez az azonosító lehet egy egész: *fájl-leíró*, lehet egy *fájlpointer* stb. A megnyitás az azonosító definiálása, egyben a *csatorna leképzése* egy az operációs rendszer

számára is ismert eszköznévre, fájlnévre. A csatorna "megszüntethető" a *lezárásával*: az explicit *close*, *fclose* stb. rendszerhívásokkal. A legtöbb operációs rendszerben a nyitott csatornák lezáródnak a folyamat terminálódásával.

A folyamatok számára a nyitott csatornák byte-, vagy rekord-források, -nyelők. A csatornákba, a csatornákból byte-ok, rekordok mozgathatók, szekvenciálisan, vagy közvetlen eléréssel. A mozgatott adatmennyiség függ a csatornához tartozó (a leképzett) fájl, vagy eszköz *szervezettségétől* (organisation), az *elérés módját* is befolyásolhatja a szervezettség. A leggyakoribb adatátvivő rendszerhívások a *read* és a *write* rendszerhívások, de ismerünk más rendszerhívásokat is: *put, get, putchar* stb. A legtöbb operációs rendszer a csatornákhoz biztosít egy *fájl pozíció indikátor* mechanizmust is, ami a nyitott csatornán az adategység pozícióját jelzi. Ennek "mozgatása" (seek) a soros szervezésű fájlokon is lehetővé teszi a *direkt elérést*.

A Unix operációs rendszerek fájlszervezése hallatlanul egyszerű: itt a fájlok bájtok sorozataként megvalósítottak. Nincs különösebb *szervezettség*, az elérés soros, ill. a pozíció indikátor mozgatása segítségével direkt lehet. Az input/outputtal kapcsolatos rendszerhívások a következők:

Nyitó, záró rendszerhívások:

`open(), pipe(), socket()` descriptor köti össze a *stream*-et, ami leképződik egy *fájl*ra, *eszközre*.

`close(), shutdown()`

Adatátvivő rendszerhívások:

`read(), write()` átvitel egy *leíróval* azonosított *folyamból(ba)*, *egy user address space*-beli címről/címre

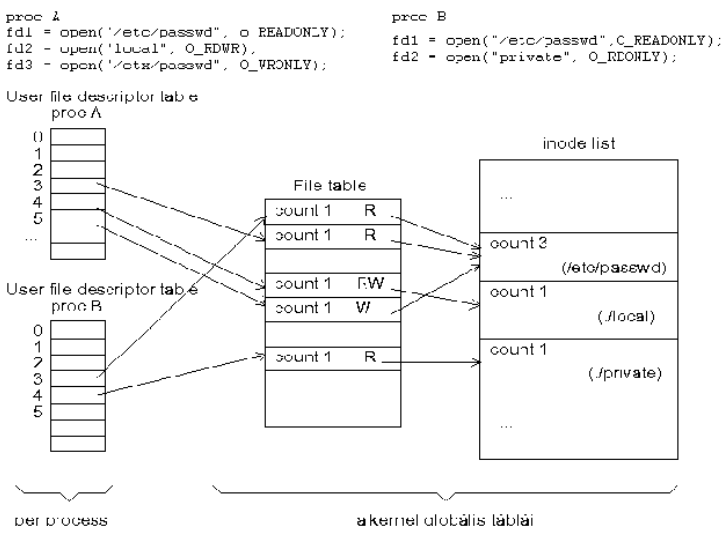
seek() *pozíció indikátor* beállítása

System call-ok jegyzékekre:

`mkdir(), rmdir() és társai` *descriptor* tartozik ezekhez is.

System call-ok a file system kezeléshez:

`mknod() és társai` *descriptor* tartozik ezekhez is.



8.1. ábra. Adatstruktúrák, miután két processz megnyitott fájlokat

Végül egy ábrán bemutatjuk az adatstruktúrákat, melyek azt a helyzetet mutatják, amikor két Unix processz fájlokat nyitott (az ábrán a processz szemszögéből indulunk ki).

8.1.3. Az operációs rendszer látásmódja

Az OS-ek egyik legfontosabb feladata az összes I/O eszköz

- vezérlése,
- könnyen kezelhető interfész biztosítás ezekhez,
- védelem, menedzsment biztosítása ezekhez.

Az operációs rendszer "látásmódja" ezért meglehetősen bonyolult. Tárgyalni fogjuk a kernel I/O-val foglalkozó részének szerkezetét, az eszközök kezelését, a fájlrendszer kialakításának lehetőségeit. Az általános tárgyalás mellett a Unix implementációt kicsit részletezni is fogjuk.

8.2. Alapelvek

- Az I/O szoftver rétegekbe szervezett (Egy felsőbb réteg magasabb szintű absztrakciót biztosít, az alacsonyabb réteg pedig szolgáltatást a felső réteg számára).
- Biztosítani kell az eszközfüggetlenséget (Eszköz változtatásnál ne kelljen módosítani a programot).
- A hibakezelés célszerűen legyen elosztva (A hibákat kezeljük minél közelebb a hardverhez. Tranziens hibákkal ne foglalkozzanak a felsőbb rétegek).
- Szinkronitás - asszinkronitás összeillesztése (A felhasználói processz szinkron ír/olvas, míg maga a transzfer aszinkron, megszakítás vezérelt).
- Osztható (sharable) és dedikált eszközök is kezelhetők legyenek. A holtpont problémák kerülendők!

8.2.1. Az I/O szoftverek szokásos rétegződése

Emlékezzünk az architektúrákra! Az eszközök a buszra csatlakozó

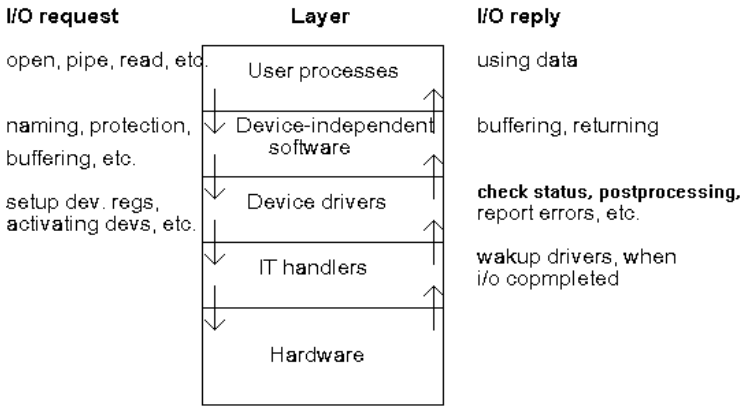
- controller/adapterből, és az ezekhez csatlakozó
- fizikai eszközökből állnak.

Ezekkel a device driver-eknek és bennük az interrupt handler-eknek van közvetlen kapcsolatuk.

Az eszközmeghajtók (Device drivers)

Az I/O alrendszer legalsó részét, az eszköz drivereket egy rutinkészlet (set of routines) és táblázatok, pufferek (tables, buffers) alkotják Miután a kernel részei: a rendszer címtartományához tartoznak (Systen Virtual Address Space). Legfontosabb feladatuk az adatmozgatás (mozgattatás) a központi memória (rendszerint a központi memóriában képzett buffer) és a kontroller (a kontroller buffere) között, továbbá parancskiadás a kontroller számára, hogy

az működjön. További fontos feladatuk olyan események kezelése, melyeket egy-egy kontroller kelt, amivel jelzi, hogy kész van a kapott feladatával.



8.2. ábra. Az I/O szoftverek rétegződése

Az eszköz driver-ek rutingyűjteményt képeznek, jól meghatározott struktúrával.

Alapvetően három részből állnak:

Autokonfigurációs és inicializáló rutinokból. melyek egyszer hívódnak, a driver betöltésekor, indulásakor. Egy monolitikus rendszernél ez a rendszerindításkor történik, dinamikusan betöltődő drivereknél a load után. Feladatuk: tesztelik az eszközöket, vizsgálják azok jelenlétét, inicializálják az eszközöket (pl. felpörgetik stb.). Felülről, call jelleggel hívódnak.

Második rutincsoportot az I/O kérelmeket kiszolgáló rutinok alkotják. Pl. olvass valamennyi bájtot és tedd a memóriába, olvass blokkot, vagy írj blokkot stb. Felülről, call jelleggel hívódnak (Ezek jelentik a driver felső rétegét).

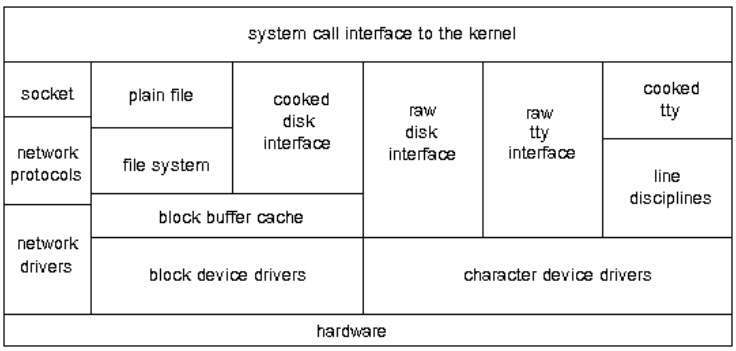
A harmadik csoportot az megszakítás kiszolgáló rutinok alkotják, Ezek "alulról" hívódnak, aszinkron módon. (Ezek az alsó réteghez tartoznak).

Az OS I/O alrendszere tehát feltétlenül tartalmaz device-driver komponenseket, minden konkrét eszközfajtához saját drivert. A rendszergazdáknak kell gondoskodnia arról, hogy minden konkrét eszköznek meglegyen a drivere (monolitikus rendszernél a rendszerképbe linkeltek legyenek a driverek), mindegyik inicializálódjon a normál használat előtt.

Viszonylag egyszerű a szerkezete a Unix I/O alrendszerének, ezért azt elemezzük tovább.

8.2.2. A UNIX kernel I/O struktúrája

Emlékezzünk a kernel funkcionális szerkezetére! Ebből kiemelve az I/O alrendszer a 8.3. ábrán látható.



8.3. ábra. A Unix kernel I/O szerkezete

Az eszközöket, természetesen, szokták osztályozni, csoportosítani.

Vannak karakterorientált, vagy más néven struktúrálatlan eszközök (character oriented devices). Ilyenek a terminálok, soros vonalak, nyomtató portok, analog/digital átalakítók stb. Struktúrálatlannak mondjuk ezeket, ami tulajdonképpen

azt jelenti, hogy ezek az eszközök képesek fogadni/adni egy struktúrálatlan karakter/bájt sorozatot.

Ha jobban meggondoljuk, ezen a bájt-folyamon lehet azért struktúráltság!

Például egy karakteres terminálból(ba) jövő (menő) karakter-sorozat lehet sorokra (lines) tördelt. A karaktersorozatban a sorvég karakterek struktúrálnak. Bevitel esetén a terminál egy sor pufferbe gyűjtheti a karaktereket, és a sorvég leütése után küldheti a teljes sort a rendszerbe. Hiába van azonban ez a struktúráltság, az adatelérés itt csakis szekvenciális lehet, azaz át kell jutni az előző karaktereken (bájtokon), hogy bizonyos karaktereket (bájtokat) elérjünk. Maga az adattovábbítás pedig változó hosszúságú sorokban (blokkokban) történhet, határesetben 1 karakter (bájt) továbbítása is lehetséges. Sor (blokk) továbbítása esetén elképzelhető, hogy a továbbított sort még egy - az eszköz driver fölötti - rutinkészlet (line disciplines) is feldolgozza: ez vezérlő karaktereket, karakterszekvenciákat kezelhet (pl. TAB karakter kiterjesztése helyköz karakterekre, DEL karakter kezelés stb.). Ezt a kezelést igényelhetjük, de el is kerülhetjük: ha igény van az eredeti, "raw" bájtfolyamra, megkaphatjuk azt, és majd feldolgozza azt a felhasználói programunk.

Az eszközök másik nagy csoportját a blokkorientált (block oriented devices), vagy struktúrált eszközök képezik. Tipikus példájuk a diszk eszközök. Mit jelent itt a blokkorientáltság?

Az ilyen eszközöknél blokknyi egységekben történik az adattovábbítás (azaz pl. 1 bájtért is be kell hozni a teljes blokkot), blokk egységben történik az adatrögzítés. Egy-egy blokk feldolgozása "random" jellegű is lehet. Minden blokknak van címe (diszk eszköznél pl. a fej-cilinder-szektor címhármas adhatja). Ez adja a blokk struktúrát, ezért mondjuk ezeket struktúrált eszközöknek.

Vessünk most egy pillantást a Unix I/O ábrára!

Látjuk a "legalsó", hardver közeli komponenseket, a két eszközosztály driver-eit. És mi van felettük? Milyen funkciókat biztosítanak a driver-ek fölötti téglalapokhoz tartozó komponensek? Nézzük ezeket jobbról.

8.2.3. Az eszközök kezelése

A character device drivers feletti line diciplines + cooked tty komponensek a karakteres eszközökre (tipikusan terminálokra, soros vonalakra) struktúrált elérést biztosítanak.

A line disciplines rutinkészlet

- sorokba (line) rendezi az inputot,
- feldolgozza a DEL és KILL karaktereket,
- echózik (hacsak a terminál nem teszi ezt),
- TAB-ot kiterjeszti helyközökké,
- jelzéseket (signals) generál (pl. terminál vonal hangup),
- ú.n. "raw " módban szűri a karaktereket
- stb.

A cooked tty nyitó/záró, író/olvasó és kontrolláló (ioctl) rutinokból áll, tipikusan ezek hívhatók - persze, a diszpécseren át - a felhasználói programokból.

A raw tty interface rutinkészlet szintén a karakter orientált eszközöket (tipikusan soros vonalak, terminálok stb.) kezeli, csak éppen nem szűrik az inputot, minden bájtot, karaktert eljuttatnak a felhasználói programhoz.

Az ábrából nem feltétlenül jön a következtetés, de vegyük tudomásul, akár ugyanaz az eszköz váltakozva kezelhető a "durva" felületen át és a "finom" (cooked) felületen át.

Tovább az ábrán láthatjuk a "raw disk interface" téglalapot a character device driver felett. Ámbár a diszk tipikusan blokk orientált eszköz, mégis lehetséges hozzá karakter driveren át hozzáférés. Elképzelhetjük ui. a diszket is bájtok struktúrálatlan folyamának, és lehet olyan alkalmazás (felhasználói program), ami szekvenciálisan akarja olvasni/írni ezt a bájtfolyamot (pl. egy diszkmentő/visszatöltő segédprogram, ami teles diszkképet ment). Kétségtelen, hogy mikor egy diszket bájt-folyamként kezelünk, ugyanakkor nem kezelhetjük blokkorientált módon is ...

El is érkeztünk az ábrán a block device drivers fölötti komponensekhez. Ezekről később részletesebben fogunk szólni, most csak annyit, hogy a block buffer cache mechanizmus egy gyorsítótár a diszkek és a memória között. Látjuk azt is, hogy a blokkorientált eszközökre képezhetünk fájlrendszert (fájlrendszereket), és azon át is elérhetjük a diszk (blokkorientált eszköz) blokkjait, emellett - és akár felváltva is - elérhetjük a blokkokat kimondottan a blokkcímeik szerint (cooked disk interface komponensen át).

8.3. Diszkek, blokk orientált eszközök

Pillanatnyira félretéve a buffer cache komponenst, azt látjuk, hogy a felhasználói processzek a diszkek blokkjaihoz két úton is hozzáférhetnek. Egyik út a fájlrendszer kezeléshez tartozó rendszerhívás csoport. Mikor a felhasználói processz fájlból való olvasást kér, az valójában "átalakul" egy diszk blokk behozatalra, diszk blokk igényre. A felhasználói processz a cooked disk interface-en keresztül közvetlenebbül is kérhet adott című diszk blokkot.

A diszk driver-hez a kérelem felülről mindenképp úgy jön, hogy adott diszk adott blokkjára van igény.

A logikai diszk (Logical Disk) fogalma

Sokféle diszk van, különböző olvasófej számmal, sáv (track) számmal, szektor számmal, különböző méretekben, különböző gyártók, interfészek vannak stb. Bonyolulttá válik a driver írók feladat, ha minden változathoz illeszteniük kell a driver-eket. Egyszerűbbek lesznek a driver-ek, ha a logikai diszk modellt használhatják.

A *kontroller/adpter*-ek a *driver szoftver*rel együtt biztosíthatnak egy konzisztens modellt, a ***logical disk modellt.***

E szerint a *logical disk* blokkok sora **0**-tól **n**-ig sorszámozva. A logikai blokkcímet a kontroller "fordítja le" fej-cilinder-szektor címhármasra, a driver szemszögéből nézve a diszk blokkok sorának látható.

Szinte minden fizikai diszk konvertálható egy ilyen ideális modellre.

A gondolat tovább folytatható. Egy fizikai diszk logikailag egymásutáni blokkjai összefoghatók, a diszk partíciók alakíthatók ki rajtuk.

A diszk partíciók

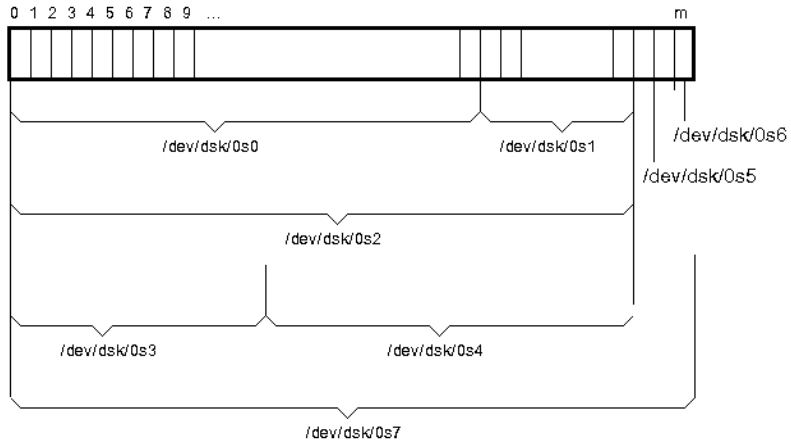
Egy fizikai diszk konvertálható partíciókra. Egy partíció a diszk valahanyádik blokkjától kezdődően valahány egymásutáni blokk. Tudnunk kell, hol kezdődik a partíció, és hogy milyen hosszú. A partíció első blokkja a 0 logikai címet kaphatja, a következő a 1-es címet s.í.t. Ezután a partíció egy blokkjára a relatív logikai címével hivatkozhatunk, a partíció nem más,

mint egy logikai diszk, ami **0** - **n**-ig sorszámozott blokkokból áll. Egy partíció egy logikai diszk-eszköz, kell legyen szimbolikus neve, kell, hogy tartozzon hozzá eszköz driver, ami kezelni tudja.

A partíciók az operációs rendszerekben úgy kezelhetők, mint a diszkek, A partíciókra:

- *file system*-et szervezhetünk,
- kijelölhetjük kilapozási/kisöprési (*swap)* területnek (lásd: memory management),
- kijelölhetjük *boot loader*-nek, innen töltődhet a rendszer,
- kijelölhetjük *alternate block area*-nak (hibás blokk helyett innen allokálhatunk blokkot).

Ahogy említettük, minden partíciónak kell legyen szimbolikus neve és kell hozzá tartozzon eszköz driver, ami kezelni tudja. Egyes operációs rendszerekben az eszköz szimbolikus neveket megszokhattuk, lehetnek azok az abc betűi, konvencionálisan az A: és a B: nevű eszközök floppy diszkeket, a C:, D: s.í.t. szimbolikus nevek további diszkeket - akár logikai diszkeket, partíciókat - jelölhetnek. de mi a helyzet a Unixban? Itt nem ismerünk hasonló szimbolikus neveket!



8.4. ábra. Partíciókra osztás Unixban

A Unixban az eszközök szimbolikus neveit az ún. speciális fájlok hordozzák! A speciális fájlok szokásosan a /dev jegyzékben (vagy ebből kiinduló aljegyzékekben) vannak bejegyezve. Egy-egy speciális fájlra való hivatkozáskor valójában a hozzá tartozó eszközre hivatkozunk. Maga a speciális fájl nem hosszú, csak 2 számot tartalmaz: a *major device number*-t, ami az eszközt vezérlő kontrollert (adaptert), és a *minor device number*-t, ami a kontroller által vezérelt eszközt (akár partíciót) azonosítja. A két azonosító szám együtt nemcsak az eszközt azonosítja, hanem az eszköz kezeléséhez szükséges *eszköz driver* kernel komponenst is!

A partíciókra osztáshoz alacsony szintű szoftverek kellenek, amelyeket a gyártóktól kell beszerezni! A diszk *formattálás*a után lehet *partícionál*ni. A partíciókra osztáskor megmondjuk, hol kezdődnek és milyen hosszúak a partíciók. A partícionálási információk az u.n. *partition table*-ban vannak, ez pedig a fizikai diszk **0** sorszámú blokkján található. Egy partícionált diszk újra partícionálható, de a partíció határok megváltoztatása tönkretehet *file system*-eket!

Egyes operációs rendszerek (pl. Unix) megengedik a partíciók átlapolódását.

Egy Unix-os példa:

```
             Beginning   Size      Purpose
-------------------------------------------------------
/dev/dsk/0s0    0       16002   Root file system
/dev/dsk/0s1    16002   16002   Swap area
/dev/dsk/0s2    32004   25002   /usr file system
```

```
/dev/dsk/0s3     57006    25002    First user file system
/dev/dsk/0s4     82008    64566    Remainder
```

Ökölszabályok a partíció méretek kiválasztására

- A *root file system*-hez: a legfontosabb dolgok itt elférjenek.
- A *swap area*-hoz: ökölszabály: 2 * a központi memória mérete.
- A /usr-hez: elférjenek a közös dolgok, pl. a *man pages* is!
- kezdetben a /dev/dsk/0s3-at és /dev/dsk/0s/0s4-et nem is használjuk. Innen akár át is partícionálhatunk.

Nota bene! Vannak partíció név konvenciók a Unix-okban!

Pl. a System V:

```
/dev/dsk/c0d0s0
         | | |
         | | partíció száma
         | |
         | device száma
         |
        driver-adapter/controller azonosító
```

Foglaljunk össze

Az egyes *speciális fájl*okhoz vagy diszkazonosítókhoz tartoznak *logikai diszkek, partíciók*. Mind *0* - *n*-ig számozott blokkokat tartalmaznak. A partíciók mérete adott a bennük szereplő blokkok számával.

Sokszor már a nevükből, de mindenképp a tartalmukból tudjuk (a kernel a tartalmukból tudja), milyen *driver-adapter/controller-partíció-eszköz*-ről van szó.

Láttuk: átlapolás lehetséges.

Láttuk továbbá, hogy partícióváltoztatás csak korlátozottan lehetséges.

A fizikai eszköz *0***.** sorszámú blokkjában van a partíció tábla.

A partíciók szerepe különböző lehet (file system, swap, boot, alternate).

A *block address* a blokkok címe a partícióban: a blokkok sorszáma. *0* és *n* közötti. (block number = block address).

8.4. A fájlrendszerek

A *fájl*: valamilyen szempontból összetartozó adatok névvel ellátva. Vannak névkonvenciók.

8.4.1. Fájl struktúrák, fájl organizáció

Három általános struktúra lehetséges:

- A fájl bájtok sora. Tulajdonképpen *nincs struktúráltság*, ill. a processzek struktúrálhatnak, ha akarnak.
- A fájl rekordok sora. A rekordok lehetnek állandó, vagy változó hosszúságúak, ezen belül un. blokkoltak. A *rekord struktúráltság* a diszken, partíción, szalagon rögzített, nem a processzek struktúrálnak, hanem az OS I/O alrendszere.

- *Indexelt szervezésű rekordokból* áll a fájl. A rekordok nem feltétlenül egyforma hosszúságúak, van bennük egy vagy több kulcsmező - rögzített pozíción -, amik segítségével gyors kikeresésük lehetséges.

Unix-ban az első, MS-DOS-ban az első és a második, VAX/VMS alatt mindhárom organizáció lehetséges.

8.4.2. A fájl elérések

Általánosan kétféle lehet:

- *szekvenciális* vagyis *soros* elérés, ami mindhárom organizációnál lehetséges. Ez tulajdonképpen azt jelenti, hogy ha egy bájtot, vagy rekordot el akarunk érni, az előtte álló bájtokat, rekordokat végig kell olvasni, vagy legalább is át kell lépni.
- *random*, vagyis *véletlenszerű* elérés, ami azt jelenti, hogy egy byte vagy rekord elérése független a többi bájttól, rekordtól. A Unix ezt a *seek* rendszerhívással biztosítja. Más operációs rendszerek a fix hosszúságú szekvenciálisan szervezett rekordokhoz, ill. az indexelt szervezésű rekordokhoz a közvetlen - random - elérést biztosítják.

8.4.3. Fájl típusok

Osztályozhatjuk a fájlokat a tartalmuk szerint is. Így lehetnek:

- közönséges (regular) fájlok, amik tovább is osztályozhatók (text fájlok, binary fájlok stb.)
- *jegyzékek* (directories), amik bejegyzéseket tartalmaznak további fájlokról.
- bizonyos operációs rendszerekben *FIFO* jellegű fájlok, *mailbox*-ok,
- a Unix-ban un. *speciális fájl*ok, amik tulajdonképpen eszközöket azonosítanak.
- *könyvtárak* (libraries), melyek *tagokat* (members) tartalmaznak, amik maguk lehetnek szövegek, tárgy modulok, végrehajtható programok stb.

8.4.4. Fájl attribútumok

A fájloknak nemcsak nevük és adataik vannak, hanem további kapcsolódó információik: pl. készítési dátumuk, utolsó módosítási vagy elérési dátumuk, tulajdonosuk, csoporttulajdonosuk, védelmi maszkjuk, írás/olvasási engedélyük stb. is jellemzik őket. Ezeket nevezhetjük az attribútumaiknak.

8.5. Fájlrendszer implementációk

Blokk-struktúrált eszközökre (logikai diszkekre - partíciókra) szervezhetünk *fájlrendszer*t.

Tulajdonképpen három dolgot kell megoldani:

- hogyan rögzítsük, hogy egy adott fájlhoz mely blokkok és milyen sorrendben tartoznak,
- hogyan tartsuk nyilván a logikai diszken a szabad blokkokat, hogyan keressünk ezekből, ha foglalni akarunk, vagyis hogyan menedzseljük a blokkokat a partíción,
- végül, hogyan rögzítsük a fájl attributumokat, főképpen milyen legyen a jegyzék szerkezet.

8.5.1. Blokkhozzárendelés fájlokhoz

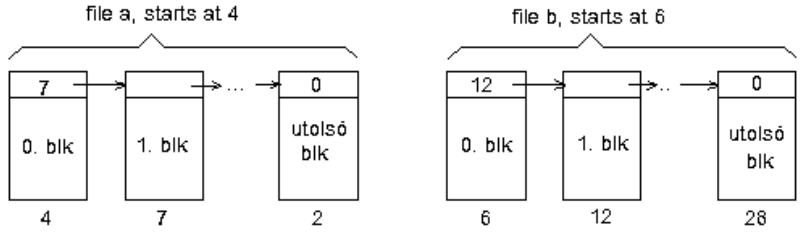
8.5.1.1. Folyamatos allokáció

Egyszerű séma, egymásutáni blokkokat foglalunk a fájl számára, annyit, amennyit az el fog foglalni. A fájl tartalom keresése során csak a kezdő blokk címét kell megadni pl. a fájl nevét tartalmazó jegyzékben. Nyilvántartják ilyenkor a hosszat, vagyis az utolsó blokk címét is.

Gond: fájl allokáció során elegendő összefüggő területet kell találni, fregmentálódik a diszk (*compaction* kell, a *gap*-ek nem használhatók ki), nehézkes a hozzáfűzés (*append)*. Korszerű operációs rendszerekben nem használják már.

8.5.1.2. Láncolt lista allokáció

Minden fájl "tartalma" a diszk blokkok láncolt listája. Így nem lesz fregmentáció. A fájl nevét tartalmazó jegyzék bejegyzésébe az első blokk mutatója, esetleg a fájl hossza bejegyzendő, az első blokkban megtalálható a következő blokk címe (mutatója) s.í.t., az utolsó blokkban a mutató NULL pointerként jelzi, hogy nincs tovább.



8.5. ábra. Láncolt lista allokáció

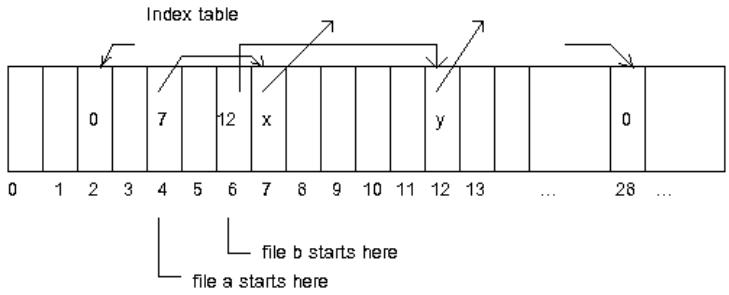
Gond: soros olvasás még elfogadható, de random keresés nagyon lassú lehet, mert mindenképp végig kell menni a láncolt listán. Másrészt a blokkokból elvesznek egy kicsit a pointerek, márpedig mi szeretjük a kettő hatványai méretű adat blokkokat.

Nem szokásos ez a módszer.

8.5.1.3. Láncolt lista allokáció index-táblával

Emeljük ki a blokk mutatókat egy indextábla mezőibe. Az indextábla a partíció kötött (megegyezés szerinti) helyén található tábla, annyi bejegyzéssel, ahány blokk van a partíción. Egy az egy megfeleltetés van a táblamezők (pointermezők) és a blokkok között: az i-edik táblabejegyzés az i-edik blokkhoz tartozik. Ha egy fájl kezdődik az i. blokkon, folytatódik a j., k. stb. blokkokon, akkor: a jegyzékben a neve mellett szerepeljen i, az i. pointermezőben szerepeljen j, a j. pointermezőben a k, stb.

Az indextábla egy bejegyzése (pl. az i. mezőbe írt j érték) kettős információt hordoz: maga az index (i) azt mondja, hogy az i-edik blokk a fájl blokkja. A mező tartalom (a k) pedig azt mondja, hogy a soron következő blokk a k-adik blokk.



8.6. ábra. Indextábla

Jó, ha az indextábla egészéről, vagy legalább részletéről *in-core* másolat van a memóriában (gyorsítás).

Tulajdonképpen az MS-DOS és végső soron a VAX/VMS is ezt a módszert használja.

Gond: nagy blokkszámú diszkek igen nagy indextáblát igényelnek. Javítható ez, ha néhány blokkot csoportba (un. *cluster*-ba) foglalunk, és a cluster-ek láncolt listájával dolgozunk. Ilyenkor persze romlik a diszk blokkok kihasználtsága, mivel nem valószínű, hogy a fájljaink mérete *cluster*-ek szorzata.

8.5.1.4. I-bögök, f-bögök alkalmazása

A Unix fájlrendszer tervezésénél kialakított koncepció - később részletezzük - szerint minden fájlhoz tartozik egy bög (node), egy adatstruktúra, ami a különböző fájl attribútumok mellett a fájl egymásutáni blokkjainak elhelyezkedését is rögzíti. A Unix i-bögök a partíció meghatározott helyén találhatók, együttesen alkotják az i-listát. A jegyzék bejegyzésben megadva az i-bög indexét (az i indexet) megragadható az i-bög, ezzel az egész fájl. Más operációs rendszerekben kissé hasonló koncepcióval f-bögök segítségével kezelhetők a fájlok blokkjai (HPFS és NTFS fájlrendszerek).

8.5.1.5. Jegyzék implementációk

Emlékszünk, a jegyzék maga is egy fájl, ami bejegyzéseket tartalmaz más fájlokról. A bejegyzésekben a fájlok attribútumait tárolhatjuk.

Miután a jegyzék is fájl, blokkok tartoznak hozzá is. Blokkjain belül vannak a bejegyzések, ezek lehetnek állandó, vagy változó hosszúságúak, az implementációtól függően. A bejegyzésekben az attribútumok között a legfontosabb a fájlnév. Sokszor van keresés a név alapján. A bejegyzések struktúrája befolyásolja a keresést. Ez lehet:

- lineáris,
  - nem rendezett bejegyzéseken, amik között nem foglalt bejegyzések is lehetnek (a törölt fájlokra);
  - rendezett, "hézagok" nélküli bejegyzéseken, ahol is gyorsabb keresési módszerek is megvalósíthatók;
- hash táblás: a szokásos szekvenciális bejegyzések mellett egy hash táblát is implementálnak, ami a gyorsabb keresést segíti.

Az esettanulmányok során különböző jegyzék implementációkkal fogunk megismerkedni.

8.5.2. A szabad diszkterület menedzselési lehetőségei

A fájlrendszer implementációkat tervezők másik gondja, hogyan menedzseljék a szabad diszkterületet, hogyan tartsák nyilván a szabad és a foglalt blokkokat, hogyan igényelhet rendszerhívás blokkokat, fájl törlésnél hogy adható vissza blokk a szabad blokkok mezejébe.

Alapvetően két megoldás szokásos.

8.5.2.1. Bit vagy mező térkép a szabad, ill. foglalt blokkokról

A partíció meghatározott (konvencionális helyén található) területe a bit/mező térkép. Bit/mező bejegyzések vannak a térképen, éppen annyi, ahány blokk található a partíción. Egy az egy megfeleltetés van a bitek/mezők és a diszk blokkjai között: az i-edik blokkhoz az i-edik bit/mező tartozik. A szabadság vagy foglaltság jelzésére a megfelelő bit 0 vagy 1 értékű, a megfelelő mező 0, vagy a foglaltságot jelző egész bejegyzésű. A bit-térkép - gyorsítási célokból - in-core memória másolattal kezelhető. Nagy diszkeken egy-egy bithez/mezőhöz cluster rendelhető.

Gyakorlatilag ezt a módszert használja a VAX/VMS, a HPFS, az NTFS és az MS-DOS FAT fájlrendszere is.

Az MS-DOS un. FAT (File Allocation Table) táblája az index tábla és a mezőtérkép összevonása. Egy-egy bejegyzése három információelemet is hordozhat:

- az i-edik bejegyzés i indexe azt mutathatja, hogy az i-edik blokk (cluster) a fájl blokkja (már amennyiben a bejegyzés tartalom nem nulla);

- a bejegyzés k értéke megmondja, hogy a soron következő blokk a k-ik blokk (vagy ha az EOF jelzés, akkor itt a vége);
- a nem nulla k érték egyben azt is mondja, hogy az i-edik blokk foglalt, speciális nem nulla k érték azt mondhatja, hogy ez egy hibás blokk. Egy 0 bejegyzés pedig éppen azt mondja, hogy az i-edik blokk szabad.

8.5.2.2. Láncolt lista a szabad blokkokról

Fix helyen lévő blokk tartalmaz bejegyzéseket szabad blokkokról, annyiról, amennyit képes egy blokk nyilvántartani, továbbá ugyanez a blokk egy következő blokk pointerét is tartalmazza, ami további szabadblokkokat tart nyilván s.í.t. Például 1K méretű blokkok és 16 bites diszk blokk szám esetén egy blokk 511 szabad blokkot és még egy blokk számát - ami a következő elem a listán - nyilvántarthat. 20M -ás diszk esetén 40 blokk elegendő az összes szabad blokk nyilvántartására.

Gyakorlatilag ezt a módszert használja a Unix, azzal a kiegészítéssel, hogy maguk a szabad blokkok használatosak a láncolt lista tárolására.

8.6. Unix fájlrendszer implementáció

8.6.1. Összefoglalás, alapok

Foglaljuk össze, mit tudtunk meg eddig:

A felhasználó lát egy *hierarchikus fájl-rendszert*, benne j*egyzékek*et, *fájl*okat (fájl neveket), attribútumokat stb.

Tudjuk, hogy vannak *eszközök,* amiket *speciális fájlok* reprezentálnak és láttuk ezek *kezelését.*

A processzek látnak a nyitott adatfolyamaikat azonosító leírókat (*descriptor)*, amik valamilyen módon összekötődnek az i-bögökkel, ezen keresztül fájlnevekkel, eszköz azonosítókkal.

A kernel lát i-bögöket, eszközöket és *buffer*-eket. Ezekből különösen érdekes a *block devices buffer cache.*

Eléggé *egységes interface*-nek tűnik ez.

Nem tudjuk azonban:

- Mi az i-bög (*i-node)*? Hogyan kötődik a *fájl-rendszer system* elemeihez, a fájlokhoz?
- Mi a szerepe a *buffer cache*-nek?

Továbbá:

- A *descriptor* fogalom azt súgja: a Unixban minden fájl. Ha ez igaz (Igaz!), akkor a *fájl-rendszer*be vannak integrálva az eszközök, ugyanakkor a *blokk orientált eszköz*re van szervezve a *fájl-rendszer.*

Lehetséges ez? És hogyan?

8.6.2. A Unixban minden fájl

A Unixban minden fájl és egy *fájl-rendszerbe* rendezett. Vannak

- közönséges fájlok,

- jegyzékek,
- speciális fájlok (eszközöket takarnak),
- fifo-k (pipe-ok) stb.

A /dev jegyzék tartalma

E jegyzékben a *speciális fájl*ok vannak feljegyezve. Kapcsolatokat biztosítanak az *eszközök*höz. Lehetnek aljegyzékek is: `/dev/dsk/*`

Általában a nevük már mutatja, milyen eszközökről van szó. Tartalmuk (elég rövidek!):

- *major device number* azonosítja a controller/adapter-t, és egyben a *device drivert* a kernelben.
- *minor device number* azonosítja az eszközt, ami a controller/adapter-hez kapcsolódik (ne feledjük, egy controller/adapter több fizikai eszközt is kezelhet).

Általános szabály:

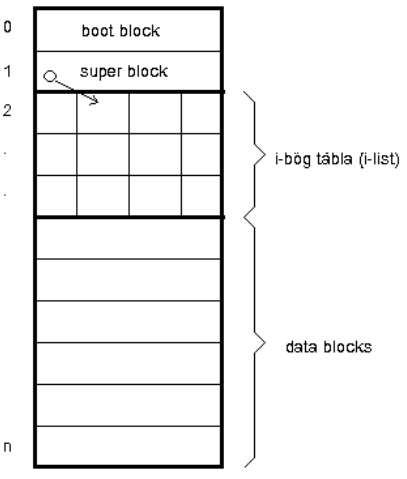
> Egy *rendszerhívás*, aminek argumentuma *speciális fájl*hoz kötődő leíró, a megfelelő eszközt kezeli. (Ez egyszerű és érthető a karakter orientált eszközökre.)
>
> Blokk orientált eszközre *fájl-rendszer* szervezhető.

A ***logical disk modell*** itt is megvan. E szerint a *logical disk* blokkok sora **0**-tól **n**-ig sorszámozva. Minden fizikai diszk konvertálható egy ilyen ideális modellre. Tartozik hozzá egy speciális fájl (pl.: `/dev/dsk/0s0`). Ez megoldja a *driver-adpter/controller*-device azonosítást.

8.6.3. Inode (Information node) (i-node) (i-bög)

Az *i-bög* (*i-node*) egy bejegyzés egy *információs táblában* (*i-list*), fájlok fontos jellemzőit - többek között az elhelyezkedésükre vonatkozó információkat tartalmaz. Az *i-bög* (*i-node*) leír egy fájlt. A UNIX rendszerben minden fájlnak egyedi *i-bög*e van.

Az *i-bög tábla* (*i-list*) *i-bög*ök tömbje. Az *i-bög tábla* a *logikai diszk*en található, de a kernel beolvassa ezt a táblát a memóriába (*in-core i-node list*) és azon manipulál.



8.7. ábra. Egy logikai diszk struktúrája

Az *i* index ehhez a táblához. (Néha az *i-bög* kifejezésen is ezt az indexet értjük, azonban ez nem zavaró, mert az *i* index az i-bög táblában (*i-list*-ben) az *i-bög*höz.)

Az *i* indexek értéktartománya: i = 1 - valameddig;

Történelmi okokból:

- i = 1 : bad blocks
- i = 2 : root i-node

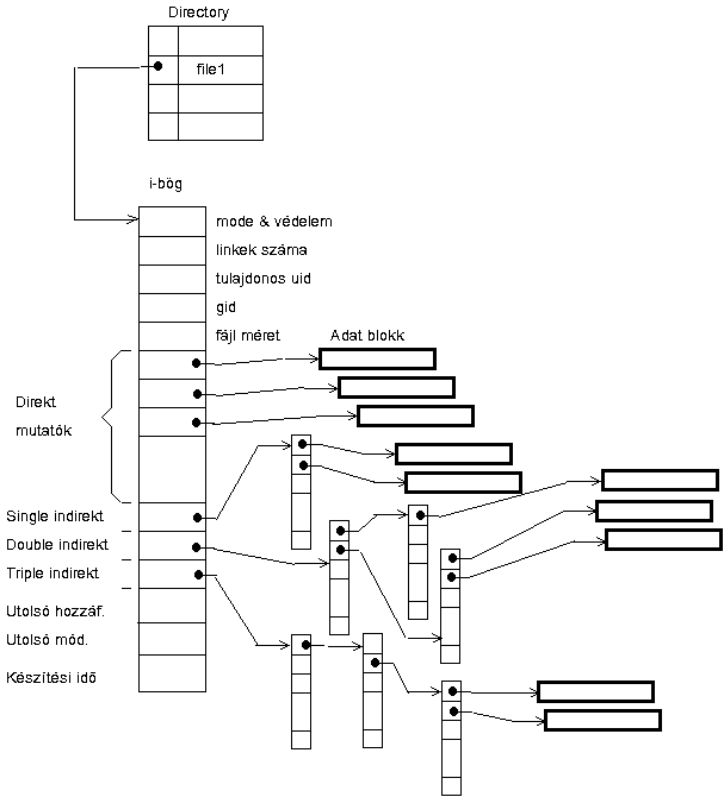
Ma már: a szuperblokkba (*superblock)* van bejegyezve a gyökér i-bög.

Egy logikai diszk struktúrája a 8.7 ábrán látható.

Az i-bögök (i-node-k) tartalma

Az i-bög egy részét a `<sys/stat.h>` -ban definiált `stat` struktúra írja le. A mezők többségének értelmezése a megjegyzésekben megtalálható. Ismeretlen típusok (pl. `ino_t`, `dev_t`) definíciói a `<sys/types.h>`-ban találhatók. Egy fájl *i-bög*ét a `stat()` és az `fstat()` rendszerhívásokkal lehet lekérdezni. A tartalom (lásd: 8.8. ábra.):

- mode és védelmi maszk (a mode a fájl típusára jellemző kód: szokásos fájl, jegyzék, FIFO fájl, speciális fájl- e az adott i-böghöz tartozó fájl; a védelmi maszk a különböző védelmi tartományokban való fájlelérést rögzíti. Amikor ls -l paranccsal listát készítünk, az első mező -rwxr-x--- mintája ebből a bejegyzésből adódik).



8.8. ábra. A Unix i-bög szerkezete

- linkek száma (vajon ez mi?)
- a tulajdonos uid-ja (Minden fájlnak van tulajdonosa, egy felhasználó "belső" azonosítójával rögzítve. ez is megjelenik az ls -l parancsnál).
- a tulajdonos gid-je (a fájl "csoport tulajdonosságát" rögzíti.)
- a fájl mérete.
- 10 db. direkt pointer (a fájl első blokkjait mutatja)
- 3 db. single-double-triple indirekt pointer (nagyobb fájlok blokkjainak közvetett tárolásához).
- 3 db dátum/idő (utolsó hozzáférés, utolsó módosítás, készítés dátuma/ideje)

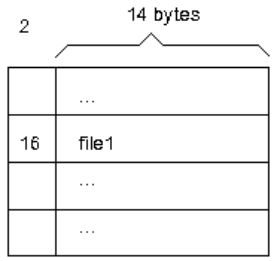
Figyelem!

Speciális fájlhoz tartozó i-bögök (hogy speciális fájl tartozik egy i-böghöz, az kiderül a mode mezőből), nem adatblokkokra mutatnak, hanem

- az első blokkcím tartalma az un. *major-minor device numbers*,
- a maradék 12 mezőt nem használják.

8.5.4. A jegyzékek tartalma

A jegyzékek bejegyzései: *i-index - név* párok

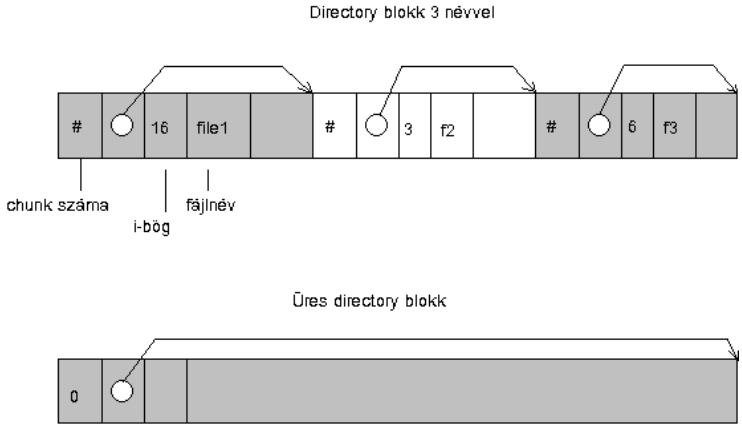


8.9. ábra. SVID jegyzék szerkezet

SVID-ben 16 byte hosszú rekeszek, ami max. 14 karakteres neveket enged meg (8.9. ábra).

BSD-ben un. *chunk*-ok, amik max.255 karakteres neveket engednek meg (8.10. ábra).

A jegyzék implementációból következik, hogy a jegyzékek bejegyzései nem rendezettek, lehetnek bennük üres bejegyzések, bennük a keresés szokásosan szekvenciális.



8. 10. ábra. BSD jegyzék szerkezet

Minden *név* „lefordítható” *i-bög*re. Tanulmányozzuk a *namei* algoritmust.

```
algorithm namei // osveny nevet inode-va alakit
/* Elemzi az "osveny" egy-egy komponenset, minden nevet az
osvenybol inode-va konvertal, az inode segitsegevel megnezi,
hogy az directory-e, vagy sem, ha nem az, visszater
jelezve ezt a tenyt, ha az jegyzek, veszi inode-jat, es vegul
visszater az input osveny inode-javal. Beirtam a "mount point"
keresztezes algoritmusat is amit eloszor nem fontos nezni! */

input:  osveny // path_name
output: locked_inode, vagy no_inode hibajelzes

{  if(osveny a gyokerbol indul)
           working_inode = root_inode; //algorithm iget
   else
           working_inode = current_directory_inode; //iget

   while( van az osvenyben tovabbi nev) {
      komponens =  a kovetkezo nev az inputbol;
      ellenorizd, hogy a working_inode jegyzek-e es
        a hozzáférés engedélyezett-e;
      if(working_inode a gyoker es a komponens "..")
         continue; // vissza while-ba
component_search:
      olvasd a working_inode-val azonositott
         directory-t! Keresd a "komponens"-t benne!;
      if(komponens megfelel egy bejegyzesnek a directory-ban)
{
          vedd az inode-t directory bejegyzesbol;
          if(found inode_of_root and
             working_inode is root and
```

```
                component name is ".."){// crossing mount point
                  get mount table entry for working_inode;
                  release working_inode; // iput
                  working inode = mounted on inode;
                  lock monted_on_inode;
                  increment reference_count of working_inode;
                  goto component_search for "..";
              }
              ereszd el a working_inode-t; //alg. iput
              working_inode = az_elobb_vett_inode; //iget
          }
          else // a komponens nem directory
              return (no inode);
      } // while vege
      return( working_inode);
}
```

Hogyan érhetők el fájlok? (Mindegy, milyen fájl)

- *a gyökér i-bögöt* veszed a szuperblokkból, míg a többi fájlra:
- jegyzékben kikeresed nevét, és veszed a hozzátartozó *i-index*-et.
- az *i-bögöt* kiemeled -- ezzel minden adata megvan

A gyors keresés technikái

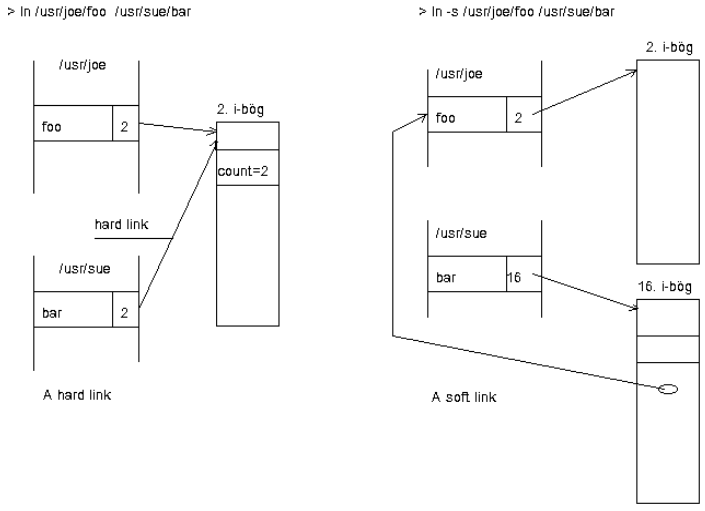
- *in-core i-node-list* mindig a memóriában van!
- memóriában van az *aktuális jegyzék i-böge* (processzenként!).
- memóriában van az aktuális jegyzék *szülő jegyzék*ének *i-böge* is.(processzenként!)

Előnyök ennél az implementációnál

- áttekinthető, egyszerű, megérthető
- kis méretű fájlok gyorsan elérhetőek a közvetlen adat blokk címzés miatt
- nagyon nagy méretű fájlok is kezelhetők
- fontos információk az *i-bög*ben vannak, gyorsan megszerezhetők
- a fájl "mozgatás" (move) (mv) igen gyors lehet: csak a jegyzékekben kell változtatni, az *i-bög* marad!
- könnyű megvalósítani a fájl "link"-et.

8.6.5. A fájl link (ln)

Előfordulhat, hogy már egy létező fájlt - aminek a neve be van jegyezve egy jegyzékbe, van i-böge - szeretnénk egy másik névvel is elérni. Ezt a más nevet lehet, hogy nem is ugyanabban a jegyzékbe szeretnénk bejegyeztetni. Másolhatnánk a fájlt, de ebben az esetben az új fájl tartalma csak pillanatnyilag egyezik az eredetivel, akár a régit, akár az újat változtatjuk, máris nem egyeznek meg a fájlok. Továbbá, a másolással duplázzuk a helyfoglalást. Mi azt szeretnénk, hogy a két (vagy több) név tényleg ugyanarra a fájlra hivatkozzon, bármelyik névvel elérve a fájlt azt módosítjuk, a másik neveiken vizsgálva is lássuk a változtatást. Ha úgy tetszik, az eddigi tisztán hierarchikus fájlrendszert szeretnénk hálóssá tenni. Nos, a Unix fájlrendszer implementáció erre lehetőséget ad a linkelés segítségével.



8.11. ábra. A fájl link

Az un. *hard link* esetén egy új directory bejegyzés készül a megfelelő jegyzékben. Az új jegyzék-bejegyzésben az i index egy már meglévő fájlhoz tartozó i-bögre mutat. Ugyanakkor az i-bögben növekszik a linkszámláló: azt jelzi, hogy ezen i-böggel azonosított fájlra több jegyzék bejegyzés is hivatkozik. Minden más attribútum változatlan. Marad a védelmi maszk, a tulajdonosi és csoporttulajdonosi bejegyzés, ezért az elérési jogok korlátozhatnak! (Még egy másik korlát is lehet: csakis ugyanazon a fájl-rendszeren lévő fájlok kapcsolhatók össze hard linkkel!) Fájl törlés esetén csak a linkek száma csökken, és egy directory bejegyzés tűnik el, ha a linkszám még nem 0, az i-bög továbbra is foglalt. (8.11. ábra.)

A *soft*, vagy *symbolic link* során készül egy új fájlnév bejegyzés, új i-böggel. Az új i-bög hozzáférési maszkja új, a tulajdonossági információk is újak, a blokkpointer mezők tartalma azonban nem a szokásos. A pointerek helyett itt a másik fájl abszolút ösvényneve van feljegyezve, ennek szimbolikus linknek segítségével megkereshető az "eredeti" fájl. Ha az eredeti fájlt törlik, a szimbolikus link "sehova se mutat", hibajelzést kapunk hivatkozáskor. Szimbolikus linkkel különböző fájlrendszerek fájljai is összefűzhetők.

8.6.6. A szuperblokk tartalma

A *szuperblokk* információkat tartalmaz az egész fájlrendszerről. Minden logikai diszknek az első blokkja. Hagyományosan ma is 512 bájt hosszú. Némelyik Unix rendszer másolatokat is őriz a szuperblokkról.

A szuperblokk többek között tartalmazza:

- a fájlrendszer méretét,
- a szabad blokkok számát,
- a szabad blokkok listáját (pontosabban a lista elejét),
- indexet a következő szabad blokkhoz a szabad blokkok listáján,

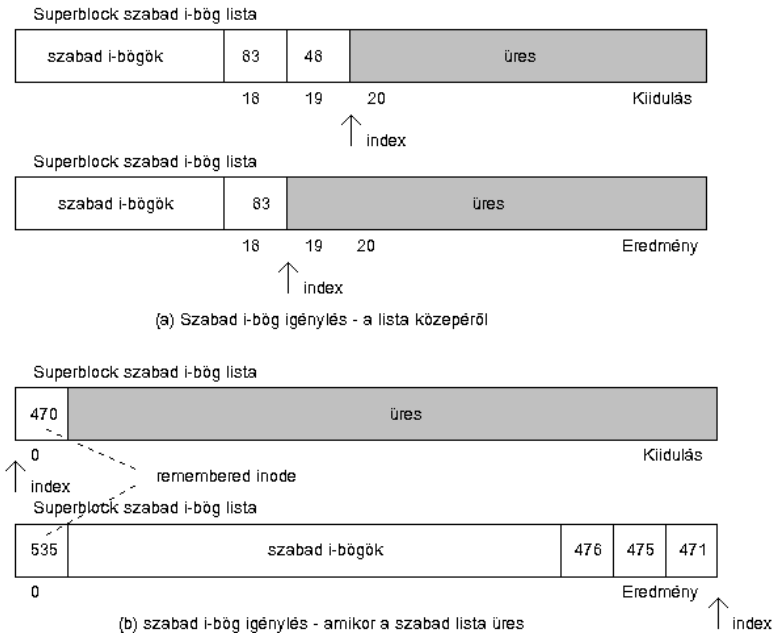
- az i-bög-tábla (i-list) méretét,
- a szabad i-bögök számát,
- a szabad i-bögök listáját (pontosabban valamennyi szabad i-bögről egy listát),
- indexet a következő szabad i-böghöz a szabad i-bögök listáján,
- lock mezőt a két szabad listához,
- egy jelzőt (flag), ami jelzi, hogy történt-e módosítás a szuperblokkban.

Figyelem!

Az un. *mounted file system* szuperblokkja benn van a memóriában is (in-core superblock), ez gyorsítást eredményez. A *sync* parancs segítségével időnként kiírjuk a lemezre is a szuperblokkot. Most megérthetjük, miért nem szabad csak úgy kikapcsolni a Unix-os gépeket: szétesheti a fájlrendszer, ha nincs kiírva az aktuális szuperblokk. (Szétesett fájlrendszert a rendszer-menedzser (superuser) az fsck parancs segítségével rendbehozhat, de fájlok veszhetnek el!).

8.6.7. Algoritmusok fájlkészítéshez, törléshez

8.6.7.1. Fájlkészítéshez a következőt kell tenni:



8.12. ábra. Szabad i-bög igénylés

(Lásd: ialloc,alloc algoritmusok, és a 8.12 ábra).

Allokálni kell egy i-bögöt (ialloc), azt ki kell tölteni, a jegyzék bejegyzést ki kell tölteni.

Allokálni kell a fájl számára szabad blokkokat (alloc), be kell jegyezni ezeket az i-bögbe, és tölthetjük a blokkokat az adatokkal.

A 8.12. ábra (a) része azt mutatja, hogy a szabad i-bög listáról vesszük a következő, a 48-as i-bögöt, az index ezután a következő szabad i-bögre mutat.

A (b) ábrán a szuperblokk szabad i-bög listája kiinduláskor üres (index a 0-ra mutat). Ez persze nem jelenti feltétlenül azt, hogy egyáltalán nincs szabad i-bög! Lássuk be, hogy a szuperblokk szabad i-bög listája nem jegyezheti föl az összes szabad i-bögöt, csak valamennyit ezekből. A ténylegesen szabad i-bögök a diszk i-listája vizsgálatával deríthetők ki: szabadok azok, melyeknek a linkszámlálója 0. Ha tehát a szuperblokk szabad i-bög listája üres, a kernel olvasva a diszket, annyi szabad i-bögöt helyez el a szuperblokk szabad i-bög listáján, amennyit csak tud. Ezt a keresést az un. *remembered inode*-tól kezdi, nem pedig elölről. A feltöltés után az indexet áthelyezi, és a *remembered inode* értéknek feljegyzi a legnagyobb talált i-bög számot. (Kérdés, miért így csinálja? Válasz: nem veszt időt a valószínűleg foglalt i-bögök vizsgálatával, amik az i-bög lista elején vannak.)

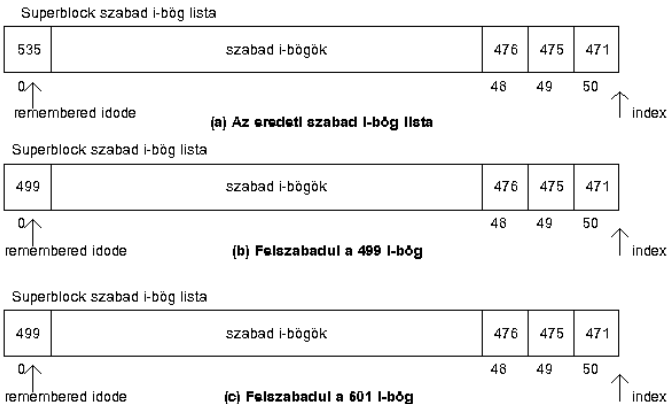
```
algorithm ialloc        // inode-t allokal
```

```
input:  file system
output: locked inode
{  while(not done) {
      if(super block locked) {
         sleep(amig super block szabad nem lesz);
         continue // vissza a while ciklushoz
      }
      if(inode lista a super block-ban ures) {
         lock super block;
         vedd az un. "remembered inode"-t a szabad inode
           kereseshez;
         keress a diszken szabad inode-kat, amig a
           super block teli nem lesz, vagy amig
           nincs tobb szabad inode;
         unlock a super block-ra;
         wake up signal (super block szabad lett);
         if(nem talalt szabad inode-t a diszken)
            return(no inode);
         allitsd be a "remembered inode"-t a kov. kereseshez;
      }
      // vannak inode-k a super block inode listajan
      vegy inode-t a super block inode listajarol;
      // Az iget algoritmus, ami lock-olja is az inode-t!
      if(egyaltalan nincs szabad inode) { // !!!
         ird az inode-t a diszkre;
         ereszd el az inode-t; // iput algoritmus
         continue // vissza while ciklusra
      }
      // van szabad inode
      inicializald az inode-t;
      ird ki a diszkre az inode-t;
      csokkentsd a file system szabad inode szamlalojat;
      return(inode);
   }// while ciklus vege
}
```

A fájl törlés lépései (miután a linkszámláló elérte a 0-t)

Az adatblokkjait szabad listára kell tenni a szuperblokkban.

A fájl i-bögjét szabad listára kell tenni (ifree algoritmus).



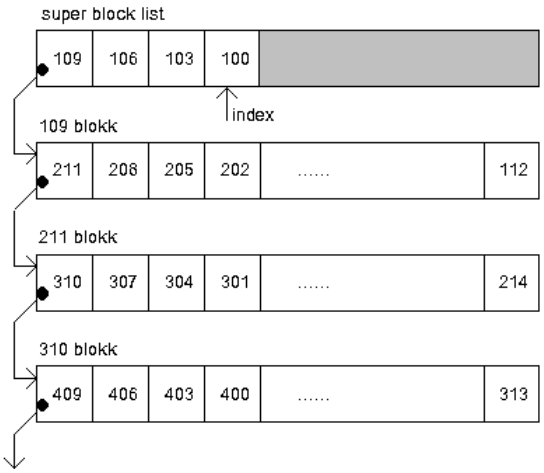
8.13. ábra I-bög számok elhelyezése az i-bög listára

A fájltörlés lépései még egyszerűbbek. Az első lépést később tárgyaljuk, a második lépés az i-bög szabad listára tétele. A szabad lista közepére kerül az i-bög száma, ha a lista nincs tele. Erről ábrát nem is mutatok. Ha a szabad i-bög lista teli van (8.13. ábra), akkor nem is biztos, hogy vesztegeti az időt a felszabadított i-bög megjegyzésével (8.13. ábra (c)). Az ábra (a) része mutatja a kiindulási helyzetet, majd feltételezés szerint először a 499. számú i-bög (b), majd rögtön utána a 601-es i-bög (c) felszabadul. A 499-es a lista elejére kerül, mert kisebb, mint a "remembered inode".

```
algorithm ifree          // inode felszabaditas
input:  file system inode szam
output: semmi
{  noveld a file system szabad inode szamlalojat;
   if(super block locked)   return;
   if(inode lista teli van) {
      if(inode kissebb, mint a "remembered inode") {
         remembered inode = input inode;
         tedd az inode-t a lista elejere;
      }
   } else {// nincs teli a lista, sot, lehet, hogy ures
      tedd az inode-t az inode listara, az index-szel
        jelolt helyre;}
   return;
}
```

8.6.7.2. Blokkok allokálása, felszabadítása

A szuperblokk egy listáján tartalmaz valamennyi szabad blokk bejegyzést, továbbá egy láncolt listán nyilvántartja az összes szabad blokkot (8.14. ábra). Az ábra szerint szabadok a 100, 103, 106 és 109-es blokkok, továbbá azok, amik a 109-esen is szabadnak vannak nyilvánítva (112, ...211), továbbá 211-es blokkon szabadnak jelöltek, és így tovább.

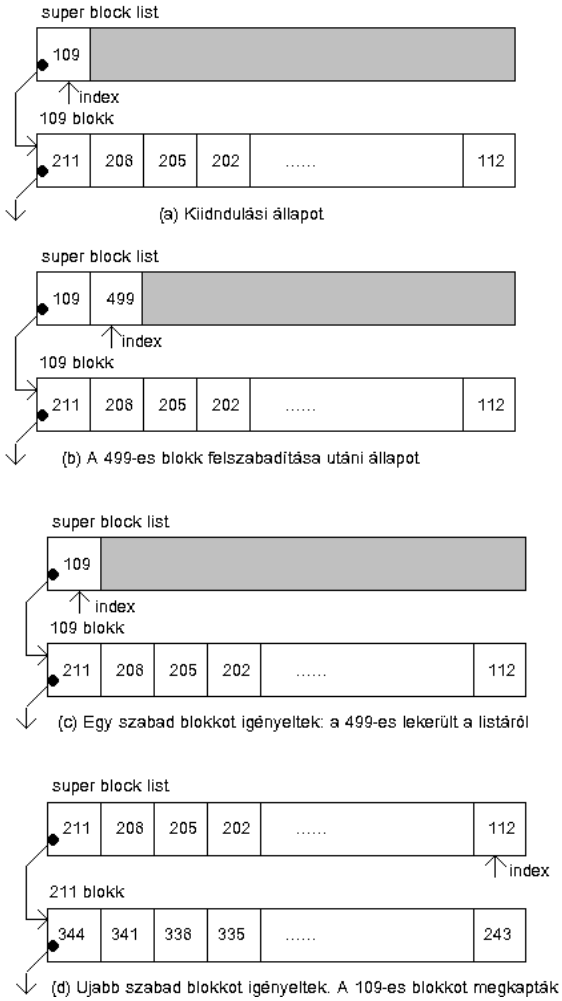


8.14. ábra. A szabad blokkok láncolt listája

Ha blokkokra van szükség, az *alloc* algoritmus szerint jár el a kernel.

```
algorithm alloc // file
system block allokacio
input:  file system number
output: buffer az uj blokk szamara
{  while(super block locked)
      sleep(amig a super blokk szabad nem lesz);
      vegy egy blokkot a super blokk szabad listajarol;
      if(az utolso blokkot vetted) {
         lockold a super blockot;
         olvasd azt a blokkot, amit a szabad listarol
           eppen vettel; // algorithm bread
         masold az e blokkban levo blokk szamokat a
           super blokkba;
         ereszd el a blokk buffert; // alg. brelse
         unlock a super blokkra;
         wake up signal azoknak a processzeknek, akik
           a super blokk unlock-ra varnak;
      }
      vegy buffer-t annak a blokknak, amit elobb vettel
        a super blokk listajarol; // algorithm getblk
      nullazd a buffer tartalmat;
      csokkentsd a szabad blokkok szamlalojat;
      jelezd, hogy modositottad a super blokkot;
      return(buffer);
}
```

Elemezzük a 8.15. ábrát! Kiinduláskor az utolsó szabad blokkra mutat az index (a 109-re). Először felszabadítjuk a 499-es blokkot: beíródik a listába, index eggyel jobbra mozdul (b). Most blokk allokálás következik (c), az előbb felszabadított 499-es blokk az áldozat, és előállt az eredeti helyzet. Következzen megint blokk allokálás: ekkor a 109 blokk listáját behozza a szuperblokkba, és a 109-es blokkot felajánlja, hogy használjuk, a szuperblokk indexét egészen jobbra tolja.



8.15. ábra. Blokk felszabadítás, blokkok igénylése

Láttuk, egyszerű a blokk felszabadítás, ha a szuperblokk lista nincs teli (8.15. (a) ábra). Ha teli van, az újonnan felszabadított blokk *link block* lesz, ebbe beíródik a szuperblokk listája, ennek száma kerül a szuperblokk lista jobb szélső elemébe, index ide mozdul (egyetlen eleme van a listának!).

8.6.8. Fájlrendszer kialakítása, használatba vétele

A rendszer adminisztrátor (superuser) a logikai diszkeken kialakíthat fájl rendszereket.

```
# mkfs diskname size
```

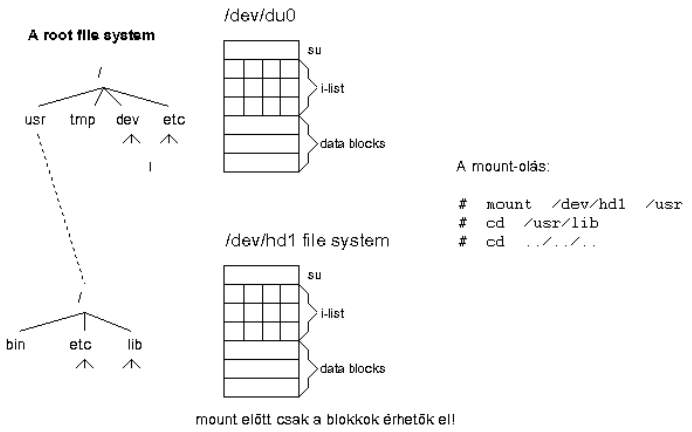
Megadhatja a fájlrendszer méretét, az i-bög tábla (i-list) méretét stb.

Szétesett fájlrendszert az fsck paranccsal „rendbeszedhet”. Ilyenkor elveszhetnek fájlok.

Mindezeket csak *nem mount*-olt eszközökre végezheti!

A rendszer boot során egy *root file system* valamelyik logikai diszkről használhatóvá válik (mountolódik). Lehetnek azonban még más logikai diszkek, melyeken vannak létrehozott fájlrendszerek, de ezek a fájlrendszerek mountolása nélkül nem kezelhetők. Maguk a partíciók blokkosan (a cooked disk interface-en át, vagy karakterorientáltan) kezelhetők, de a rajtuk lévő fájlrendszer nem érhető el.

Tételezzük fel, hogy rendszerbetöltéskor mountolódott a /dev/du0 logikai diszken lévő gyökér fájlrendszer (root file system), ezért „láthatók rajta” a /usr, a /tmp, a /dev, a /etc stb. jegyzékek, ugyanakkor létezik a /dev/hd1 logikai diszk, azon már korábban létrehoztak egy fájlrendszert. (8.16. ábra). Ennek a fájlrendszernek a jegyzékeit (bin, etc, lib), fájljait még nem kezelhetjük. Ahhoz, hogy „lássuk” ezt a fájlrendszert is, használhatóvá kell tennünk a mount paranccsal. A használhatóvá tétel nem más, mint az eredeti fájlrendszer egy üres jegyzékébe a az „új” fájlrendszer (mountolt fájlrendszer) gyökér jegyzékének leképzése: a mountolt fájlrendszer hierarchikus struktúrájának „beillesztése” az eredeti fájlrendszer struktúrájába.



8.16. ábra. Fájlrendszerek mount előtt és után

A mountolást csak a rendszergazda teheti meg.

A mount parancs:

```
#/etc/mount logical-disk üres-directory
```

Hatása: az *üres directory* ezután a *logical-disk* gyökér jegyzékének számít. A *logical-disk, pontosabban az azon lévő fájlrendszer* "hozzáadódik" az eredeti hierarchikus fájlrendszerhez! Az új fájlrendszer "használhatóvá válik".

Ezután pl. kiadható a következő parancs:

```
> cd /usr/lib # legyen a munkajegyzek a lib
```

A *mount*-olás ténye a /etc/mnttab (mount table)-ba bejegyződik! Ennek a táblának egy-egy bejegyzése egy-egy *mountolási pont* (mount point).

A mountolási pontokat bárki lekérdezheti az argumentum nélküli mount paranccsal.

```
> mount
```

A rendszergazda *dismount*-olhat (azaz megszüntetheti a mountolt fájlrendszer gyökér jegyzékének egy jegyzékbe való leképzését), ha nem „foglalt” a *mountolási pont*.

A mount tábla (/etc/mnttab) egy bejegyzése a 8.17. ábrán látható. Mielőtt megnézzük, foglaljuk összse az elnevezéseket.

Az eredeti fájlrendszer: root file system, original file system, mounted on file system.

Ennek egy üres jegyzékére mount-olhatunk. Ez elérhető kell legyen.

Mountolt eszköz: mounted device, mounted file system.

A mountolt eszköz speciális fájlneve: a partíció speciális fájlneve.

Ez a fájlnév - és a hozzá tartozó i-bög index - az eredeti fájlrendszerben a /dev jegyzékben van valahol. Ebből vehető a

A mountolt eszköz logikai száma: logical device number = major + minor device number.

A mountolt eszköz gyökér jegyzéke, ennek i-böge: root directory of mounted device (file system), i-node of it.

Ez az mkfs hívással elkészült szuperblokkba be van írva, ott van a fájlrendszeren.

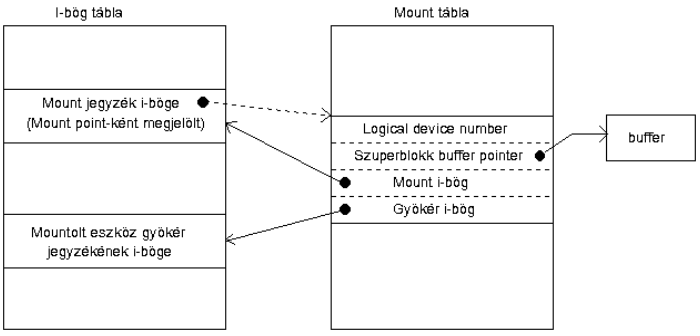
A mount jegyzék: mounted on directory.

Ez az eredeti fájlrendszer egy üres jegyzéke. Úgy is hívjuk, hogy **mountolási pont.**

A mount jegyzék i-böge: mounted on i-node.

Ez az eredeti fájlrendszer i-böge, az i-bög táblába be van jegyezve. Emlékezzünk arra is, hogy az i-bög tábla másolata az in-core memóriában van!

A buffer cache: Ezt a fogalmat később tárgyaljuk. Mindenesetre annyit most erről, hogy pointerekkel mutathatunk bufferekre, a bufferekben adat blokkok lehetnek, többek között lehetnek eszközök szuperblokk másolatai is.



8.17. ábra. Adatstruktúrák mount után

Ezek után egy bejegyzés a mount táblába a következő elemeket tartalmazza:

- a mountolt eszköz logikai száma (logical device number of mounted file system)
- buffer pointer, ami a mountolt eszköz szuperblokkjára mutat (pointer to a buffer containing the file system super block)
- a mountolt eszköz gyökér jegyzék i-bögére mutató pointer (pointer to the root inode of the mounted file system)
- a mount jegyzék i-bögére mutató pointer (pointer to the inode of the mounted on directory)

Fontos! A mount rendszerhívás nemcsak a /etc/mnttab-ot tölti, hanem a *mount jegyzék*be (*mounted on directory*-ba) bejegyzi, hogy ez egy *mount point*.

Tanulmányozzák a 8.17. ábrát, a *namei* algoritmust, a *mount* algoritmust, *unmount* algoritmust!

```
algorithm mount
inputs: block special file neve
        "mount point" jegyzek neve
        opcio: (read only)
output: semmi
{   if(not supert user)
        return(error);
    get inode for block special file (algorithm namei);
    make legality checks;
    get inode for "mounted on" directory name (namei);
    if(not directory, or reference count > 1) {
      release inodes (algorithm iput);
      return(error);
    }
    find empty slot in mount table;
    invoke device driver open routine;
    get free buffer from buffer cache;
    read super block into free buffer;
    initialize super block fields;
    get inode of mounted device (iget),
       save into mount table;
    mark inode of "mounted on" directory as mount point;
    release special file inode /* iput */;
    unlock inode of mount point directory;
```

```
}

algorithm umount

input:  special file name of file system
                  to be unmounted
output: none
{  if(not super user)
      return(error);

   get inode of special file (namei);
   extract major,minor number of device being unmounted;
   get mount table entry, based on major,minor number
     for unmounting file system;
   release inode of special file (iput);
   remove shared text entries from region table for
     files belonging to file system; [Bach: Chapter 7.]
   update super block, inodes, flush buffers;
   if(files from file system still in use)
      return(error);
   get root inode of mounted file system
     from mount table;
   lock inode;
   release inode (algorithm iput)/*iget was in mount*/;
   invoke close routine for special device;
   invalidate buffers in pool from unmounted file system;
   get inode of mount point from mount table;
   lock inode;
   clear flag marking it as mount point;
   release inode (iput) /* iget in mount */;
   free buffer used for super block;
   free mount table slot;
}
```

8.6.9. A Unix buffer cache és kapcsolódó algoritmusok

A Unix kernel funkcionális felépítését mutató ábrán felfedezhetjük a blokkorientált eszközök *device driver*-e és a *file system* között elhelyezkedő *buffer cache*-t.

Miután minimalizálni akarták a tényleges diszk hozzáféréseket, ezért a Unix kernelben megvalósítottak egy belső pufferezést: ez a buffer cache. (Különböztessük meg a CPU és a központi memória közötti hardveres cache memóriától! Annak szerepe a CPU és a memória közötti adatelérés gyorsítás, ennek szerepe a diszk és a felhasználói memória közötti adatmozgatás gyorsítása!)

Más operációs rendszerekben is szokásos a disz blokk cache-elés. Mivel a klasszikus Unix buffer cache mechanizmus elég egyszerű, ezen mutatjuk be a buffer cache lényegét (a mai, korszerű Unix-ok már virtuális memóriakezeléssel együtt oldják meg a disz blokk elérés gyorsítását).

8.6.9.1. Általános leírás, alapfogalmak

A buffer cache-ben bufferek vannak. Ezek adatrészében (testében) diszk blokkok találhatók. A kernel a diszkes I/O esetén először a cache-ben keresi a blokkokat. Ha ott megtalálja a keresett blokkot, nem is fordul a diszkhez, onnan szolgáltatja a blokk tartalmát, oda írja az output-ot. Miután van esély arra, hogy egy diszk blokk megtalálható a buffer cache-ben is (lokalitás elv!), teljesítmény növelés lehet az eredmény.

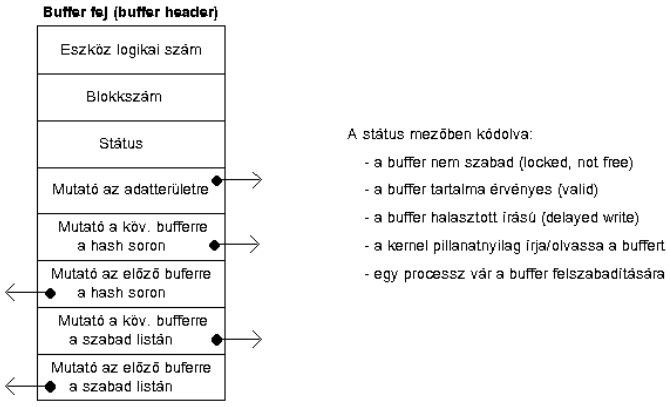
A kernel a rendszer inicializálásakor allokál helyet a buffer cache-nek, a memória méretétől és teljesítmény korlátoktól függő nagyságút. Úgy mondhatjuk, a buffer pool-ban van x számú buffer.

Valamely buffer két részből áll:

- a buffer fejből (buffer header), és
- a buffer adattároló részéből. Ez éppen akkora, mint egy diszk blokk: tartalma egy diszk blokk lehet.

Valójában a buffer adattároló részében a tartalom egy az egy leképzése egy diszk blokknak. Jegyezzük meg, hogy valamely diszk blokk csakis egy buffferbe lehet leképezve! A leképzés persze időleges, valamely buffer egyszer ezt, másszor azt a diszk blokkot tartalmazza.

Nézzük ezek után a buffer szerkezetét. Ebből a buffer fej az érdekes, hiszen az adatterület az egy blokknyi összefüggő memóriaterület (8.18. ábra).



8.18. ábra. A buffer fej szerkezete

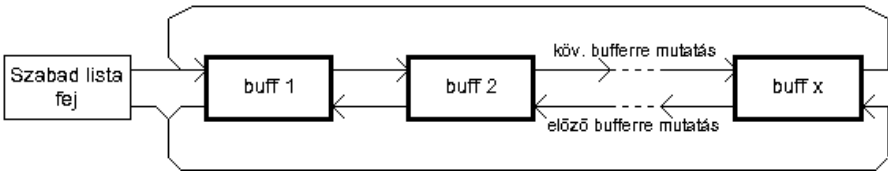
Az *eszköz logikai szám* és a *blokkszám* mezők azonosítják, melyik blokk van leképezve a bufferbe.

A státus mező a buffer állapotát jelzi. Ebből a szabad - nem szabad állapot lehet érdekes: a szabad buffer természetesen tartalmazhat érvényes (valid) adatokat, de ha kell ez a buffer, a kernel veheti, és valamilyen célra, pl. a blokk tartalom kiolvasására, felülírására, de akár másik blokk tartalom betöltésére használhatja. A nem szabad (lock-olt, not free, busy) jelzésű buffert viszont a kernel éppen használja. Az ilyen buffer más célra nem használható. Később magyarázzuk a halasztott írású buffer állapotot és azt az állapotjelzést, hogy a buffer felszabadulására van legalább egy várakozó processz.

A buffer fej többi mezője mutatókat (pointer) tartalmaz. Érthető az adatterületre mutató pointer szerepe: ez a kapcsolat a fej és a tényleges adatterület között. A további mutatók viszont magyarázandók.

A buffer pool-ban lévő bufferek fel lehetnek fűzve két *kétszeresen kapcsolt körkörös listára* (doubly linked circular list). Ez a két lista:

- a bufferek szabad listája;
- a *hash* listák valamelyike.



8.19. ábra. A bufferek szabad listája

A buffer fejben két mutató pár mutatja a listákon a következő és az előző buffert. Nézzük először a szabad listát (8.19. ábra).

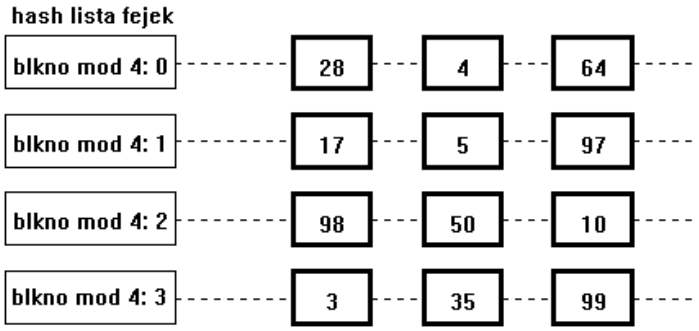
Bár az ábra csak egy téglalappal jelöl egy-egy buffert, az valójában mindig a buffer fejből és a hozzá tartozó adatterületből áll.

Ha a rendszernek egy *szabad bufferre* van szüksége, két eset lehetséges:

- akármelyik szabad buffer megfelel: ekkor a szabad lista elejéről veszi az első buffert, és persze azt leveszi a szabad listáról;
- azonosított szabad bufferre van szüksége: ekkor azt akár a lista közepéről veszi le.

Ha egy buffer felszabadul (persze ettől függetlenül még lehet benne "valid" adat), akkor általában a szabad lista *végére* illeszti a buffert. Néha az *elejére* (látjuk majd mikor), de sohasem a közepére. Ez tulajdonképpen egy legkevésbé az utóljára használt (least recently used) algoritmus: ha a rendszer egy buffert allokált egy diszk blokkhoz, akkor nem használja ezt a buffert, amíg más bufferek nem voltak utóbb használva.

Amikor a kernel egy diszk blokkot keres, először megnézi, hogy benn van-e a buffer mezőben (buffer pool). A keresés alapja az *eszköz logikai szám* és a *blokkszám* kombinációja. Hogy a keresés gyors legyen, hogy ne kelljen a teljes buffer mezőt végignézni, a diszk blokkokat tartalmazó buffereket úgynevezett *hash listákon*[2] (hash queues) tartja. Minden olyan buffer, ami diszk blokkot tartalmaz, rajta van valamelyik hash listán.



8.20. ábra. Bufferek a hash listákon

A 8.20. ábrán - egyszerűsítve - bemutatjuk, hogy néhány diszkblokk hogyan helyezkedik el a hash listákon. Itt a hash függvény a *blokkszám moduló 4*. Ez egyszerűsítés: az eszköz logikai számával nem foglakozunk. A *blkno mod 4* függvény egy-egy *hash lista fejét* címezi, maga a blokk a fejtől induló kétszeresen kapcsolt körkörös listán van. Az ábrán a kétszeres pointerezést a pontvonal jelzi csak. Az ábrán a buffereket jelképező téglalapokba a blokkszámot írtuk be, nem a buffer valamilyen azonosítóját, de ez talán érthető: így egyszerűbb az ábrázolás.

A sorokban a bufferek száma dinamikusan változik. A hash függvény azért ilyen egyszerű, hogy gyorsan kiszámítható legyen.

---

[2] hashing: adott kulcs szerinti gyors keresést lehetővé tevő technika. A kulcs értékét felhasználva egy ún. hash függvénnyel kiszámítják az objektum helyét.

Leszögezhetjük: a buffer cache-ben lévő diszk blokkok bufferei mindig rajta vannak egy és csakis egy hash listán. Nem számít, hogy a listán hol, csakis az, hogy melyik listán.

Hozzáfűzzük: valamely hash soron lévő buffer lehet a szabad listán is ugyanekkor, ha az állapota szabad! (A buffer fejek megfelelő pointerei lehetővé teszik, hogy a szabad listán is rajta legyen egy-egy buffer.)

8.6.9.2. Buffer allokálása a buffer mezőből

Képzeljük el, hogy olvasni akarunk a diszkről egy blokkot. Ekkor adott a diszk logikai száma és a blokkszám. (Honnan adódik? Az i-bögből például!) Ekkor a kernel megnézi, hogy az blkno-nak megfelelő buffer a buffer pool-ban van-e: végignézi a megfelelő hash listát. Ha ott megtalálja, megnézi még azt is, "valid"-e, és gyorsan visszatér a buffer címével. Ha nincs a blokk a buffer a pool-ban, allokálni kell neki egy szabad buffert, és ebbe be kell olvasni a blokkot.

Ha írni akarunk (ekkor is a diszk-szám és blokkszám a kiinduló adat), akkor allokálni kell egy buffert, abba már lehet írni, és idővel ez majd ki is kerül a diszkre.

Fontos algoritmus tehát az ún. *buffer allokáló algoritmus* (getblk).

Az algoritmus 5 tipikus forgatókönyvet (scenario) mutat.

A kernel megtalálja a blokkot a hash listáján és ennek buffere szabad.

Nem találja a blokkot a hash listáján, így allokál egy buffert a szabad listáról.

Nem találja a blokkot a hash listáján és megkísérel allokálni neki egy buffert. Talál is ilyet, de az "delayed write" - halasztott írás állapotú. Ekkor elindítja ennek az aszinkron írását, és allokál egy másik buffert.

Nem találja a blokkot a hash listáján, ugyanakkor a szabad lista üres. Ekkor blokkolódik (sleep), amíg egy buffer szabaddá nem válik.

A kernel megtalálja a blokkot a hash listáján, de ennek buffere pillanatnyilag foglalt. Ekkor is blokkolódik (sleep), míg a buffer szabaddá nem válik.

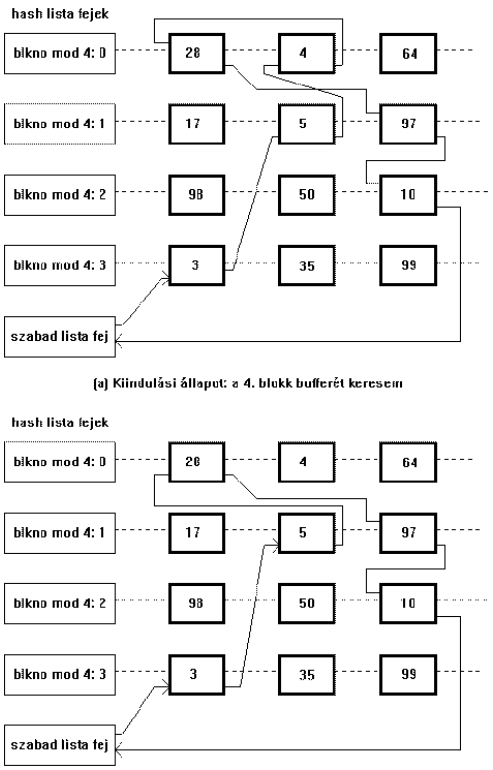
```
algorithm getblk

input:  file system number
                block number
output: locked buffer that can now be used for block

{
    while(buffer not found) {
        if(block in hash queue) {
            if(buffer busy)     /* scenario 5 */ {
                sleep(event buffer becomes free);
                continue; /* back to while loop */
            }
            mark buffer busy;   /* scenario 1 */
            remove buffer from free list;
            return(buffer);
        }
```

```
        else    /* block not on hash queue */
        {
            if(there are no buffers on free list) {
                        /* scenario 4 */
                sleep(event any buffer becomes free);
                continue;       /* back to while loop */
            }
            remove buffer from free list;
            if(buffer marked for delayed write) {
                        /* scenario 3 */
                asyncronous write buffer to disk;
                mark it to be old;
                continue;       /* back to while loop */
            }
            /* scenario 2 - found a free buffer */
                remove buffer from old hash queue;
                put buffer onto new hash queue;
                return(buffer);
        }
    }
}

algorithm brelse

input:  locked buffer
output: none

{
    wakeup all procs: event waiting for any buffer to
        become free;
    wakeup all procs: event waiting for this buffer to
        become free;
    raise processor execution level to block interrupts;
    if(buffer contents valid and buffer not old)
        enqueue buffer at end of free list;
    else
        enqueue buffer at beginning of free list;
    lower processor execution level to allow interrupts;
    unlock buffer;
}
```

Mielőtt folytatnánk a *getblk* elemzését, térjünk ki röviden arra, mi történhet a getblk után, amikor is az visszaad egy *lock*-olt buffer címet:

- a kernel olvashat blokkot a diszkről az adott bufferbe,
- írhat adatot a bufferbe, esetleg a bufferből a diszkre.

Mindkét esetben kell lock-olni a buffert, amíg vele foglakozik. A buffer használata után viszont a *brelse* algoritmussal "elereszti" a buffert. A *brelse* algoritmusban láthatjuk a megszakítás letiltást (block interrupt). Ennek oka, hogy nemcsak direkt hívása lehet a *brelse*-nek, hanem azt hívhatja az *aszinkron I/O interrupt handler* is. El kell kerülni az egymásba ágyazott brelese-ket.

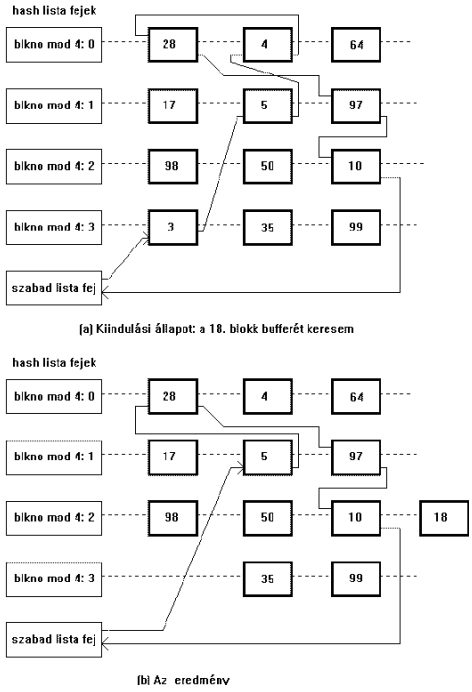


8.21. ábra. 1. forgatókönyv.

Nézzük most végig a *getblk* algoritmus egyes forgatókönyveit. A soron következő ábrák a bufferek indexeit nem mutatják, helyettük mindenütt a bufferek tartalmát mutató blokkszámokat írtuk be.

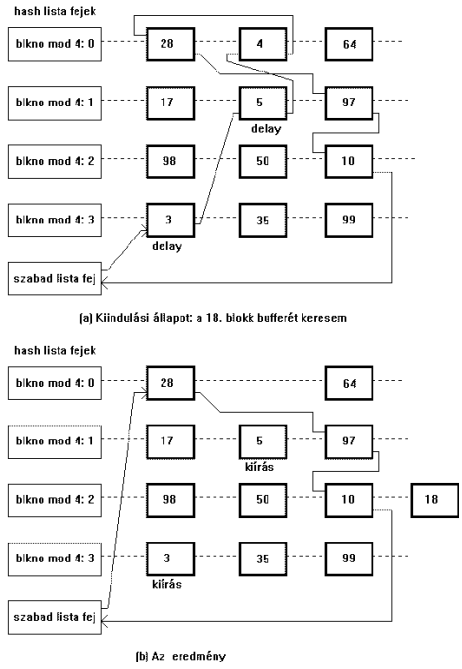
A 8.21. ábrán kiinduláskor (a), 12 buffer van a poolban, ezekből 6 a szabad listán. Szükség van a 4. számú blokkra (és az szabad is: 1. forgatókönyv). Az eredmény a (b) ábrán látható, a *getblk* visszatér a 4. blokkot tartalmazó buffer címével. A 4. blokk természetesen rajta maradt a hash listáján, de most nem szabad (locked).

A 2. forgatókönyv követhető nyomon a 8.22. ábrán. Itt is az (a) ábrarész a kiindulás, és a 18. számú blokkra volna szükségünk, ami nincs benn a buffer cache-ben. Ekkor a kernel leveszi a szabad listáról az első buffert (ami most a 3. blokkot tartalmazza éppen), áthelyezi a 18. blokk hash listájára: a 3-as listára. Utána a *getblk* visszatér ennek a buffernek a címével. Az eredményt a (b) ábrarészen láthatjuk.

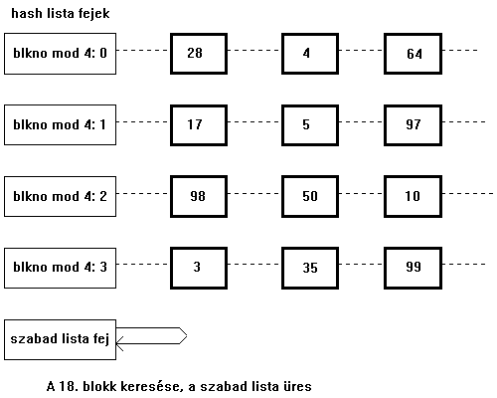


8.22. ábra. 2. forgatókönyv.

A 3. forgatókönyv nyomon követhető a 8.23. ábrán. Tételezzük fel, hogy a 3. és az 5. blokk a szabad listán van, de mindkettő állapota *halasztott írású* (delayed write), azaz a tartalmuk még nem került ki a diszkre (a. ábrarész). Ugyanekkor a 18. blokkra szükségünk van, ami viszont nincs a hash listáján. Ekkor a *getblk* leveszi először a 3. blokk, majd az 5. blokk bufferét a szabad listáról, és elindít rájuk egy aszinkron kiíratást a diszkre. Folytatásként a 4. blokk bufferét levéve a szabad listáról áthelyezi az új hash listára. Láthatjuk az eredményt is az ábrán.

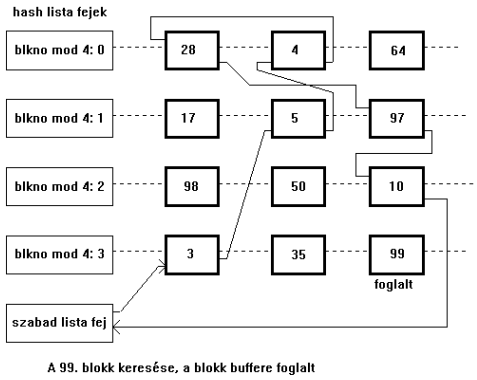


8.23. ábra. A 3. forgatókönyv



8.24. ábra. A 4. forgatókönyv

A 8.24. ábra a 4. forgatókönyvhöz tartozik. Tegyük fel, hogy az A processz számára keres a kernel buffert, akármelyik jó lenne. A szabad lista viszont üres. Ekkor az A processz blokkolódik, amíg szignált nem kap, ami jelzi, hogy szabadult fel buffer. A *brelse* kézbesítteti majd ki ezt a szignált, akár közvetlen hívással, akár a megszakítás kezelőből történő hívással. Jegyezzük meg, hogy a "felébredt" A processz újra kell keressen szabad buffert, nem allokálhat közvetlenül a szabad listáról, mert más processzek is várhatnak szabad bufferre, a versenyhelyzet kialakulhat.



8.25. ábra. Az 5. forgatókönyv

Végül az 5. forgatókönyvet magyarázza a 8.25. ábra. Tételezzük fel, A processznek kell a 99. blokk. Látható, van is buffere, de az foglalt (*locked*). Mit jelent az, hogy *locked*? Például egy másik, a B processz olvastat bele, és az olvasás befejésére vár: ezen blokkolt (ezen "alszik"). Ilyenkor tilos másik buffer keresni a blokknak, hiszen sohasem lehet egy blokk két bufferhez rendelve! Ekkor az A processz megjegyezve, hogy ez a buffer kellene neki, blokkolódik: *sleep on demand of that buffer*. Mindkét processz blokkolt tehát, mindkettő ugyanannak a buffernek a felszabadulására vár. Végül is az I/O megszakítás kezelőből hívódó *brelse* fogja felszabadítani a buffert, szignált küldve mindkét processznek erről. Ekkor az A és a B versenyezni fog a bufferért, egyik meg is kapja majd, a másik pedig újból várhat a buffer felszabadulására.

8.6.9.3. Írás, olvasás a diszkre, a diszkről

Miután a buffer allokáló és elerésztő algoritmusokat (getblk, brelse) végignéztük, könnyen megérthetjük az írás/olvasás kiszolgálását. A diszkről való olvasás algoritmusa a *bread* algoritmus.

```
algorithm bread /* block read */

input:  file system block number
output: buffer containing data

{
        get buffer for block; /* algorithm getblk */
        if(buffer data valid) /* valoban az a blokk van
                              benne? */
                return(buffer);
        initiate disk read;
        sleep (event disk read complete);
        return (buffer);
}
/* Ne feledjuk: a szinkron olvaso interrupt handler nem ad
brelse-t! Azt a processz kell adja, amikor a buffer-bol
atvette az adatokat! */
```

A *bread* algoritmusban látható, hogy ha a kért blokk benn van a buffer cache-ben, a buffer azonosítójával azonnal visszatér az algoritmus. Ha viszont nincs a buffer cache-ben, kezdeményeződik a diszk olvasás.

Sokszor előfordul - pl. mikor szekvenciálisan olvasunk egy fájlt -, hogy egymásután több blokkot akarunk beolvasni. A teljesítmény növelhet *breada* algoritmus (read-ehead). Ebben az első blokk olvasás - ha nincs a bufferban - szinkron, a második blokk olvasás aszinkron. (Ne felejtsük, az aszinkron olvasás *brelse*-t ad.)

```
algorithm breada        /* block read and read ahead */

input:  (1) file system block number for immediate read
        (2) file system block number for asynchronous read
output: buffer containing data for immediate read
{
        if(first block not in cache) {
                get buffer for first block /* alg. getblk */;
                if(buffer data not valid)
                        initiate disk read:
        }
        if (second block not in cache) {
                get buffer for second block) /* getblk */;
                if(buffer data valid)
                        release buffer /* algorithm brelse */;
                else
                        initiate disk read;
        }
        if(first block was originally in cache) {
                read first block /* algorithm bread */;
                return (buffer);
        }
        sleep(event first buffer contains valid data);
        return(buffer);
}

/* Gyorsitasi celok miatt elore olvas. Ebben az elso olvasas
szinkron -nem ad brelse-t -, a masodik asszinkron: ad brelse-
t! */
```

Nézzük most a diszkre való írás algoritmusát! (*bwrite*).

```
algorithm bwrite        /* block write */

input:  buffer
output: none

{
        initiate disk write;
        if(I/O synchronous)
        {
                sleep(event I/O complete);
                release buffer /*algorithm brelse */;
```

```
        }
        else if (buffer marked for delayed write)
                mark buffer to put at head of free list;
}
```

A felhasználói processz szemszögéből nézve

```
write  getblk  buffert tölts  bwrite
```

szekvenciát kell végrehajtani.

Ha a *write* szinkron, a hívó processz blokkolódik, amíg az I/O végre nem hajtódik. Ha a *write* aszinkron, a kernel elindítja az írást, de nem várja meg annak teljesítését. A buffert viszont csak akkor ereszti el, amikor az I/O befejeződött. Külön eset az ún. *halasztott írás* (delayed write). Különböztessük meg az *aszinkron írástól*! A halasztott írás esetén a buffer állapot (state) bejegyzést kap erről a tényről, és utána a *brelse*-vel "eleresztődik", a tényleges kiíratás nem kezdődik meg. A kernel csak akkor fogja a buffert ténylegesen kiírni, ha a 3. forgatókönyv szerint a buffert reallokálni kellene másik blokk számára. Ez előnyös lesz olyan esetekben, amikor a buffer tartalmát egymásután többször változtatják: nincs közben időigényes diszkre való kiíratás. Azaz halasztott írás esetén a tényleges kiíratás addig halasztódik, amíg csak lehet.

8.6.9.4. A buffer cache előnyei, hátrányai

A bufferezésnek sok előnye van, sajnos, vannak hátrányai is.

- Egyik legfontosabb előny, hogy a bufferezés egységes kezelést tesz lehetővé. Nem számít, hogy a blokk i-bögöt, szuper blokkot vagy adat blokkot tartalmaz, egyetlen felület van a diszkekhez, és ez egyszerűsíthet sokmindent.
- Másik előny, hogy nincs *elrendezés korlátozás* (alignment restrictio). A hardver megvalósítások sokszor más elrendezést kívánnak a diszken és a memóriában: ilyen lehet pl. 2 bájtos vagy 4 bájtos határra illesztés. Az elrendezést a kernel belsőleg elintézi, nem jelent tehát majd gondot az átvitel (portability) más hardver implementációkra.
- Nagy előny a tényleges diszk-átvitelek redukálása, ezzel a tényleges teljesítmény növelése. A processzek olvasáskor sokszor megtalálják a blokkokat a buffer cache-ben, elkerülhető így tényleges és lassú diszk átvitelek. A halasztott írás lehetőség előnye könnyen belátható. Persze, jól meg kell szabni a buffer mező (buffer pool) méretét!
- Előnynek mondható az is, hogy az egységes interfész segít elkerülni a holtpont helyzeteket.

És most nézzük a hátrányokat is.

- A halasztott írás koncepció hátránya, hogy meghibásodás esetén inkonzisztenciát okozhat. Előfordulhat adatvesztés: a felhasználói processzek *flush* rendszerhívása sem jelenti feltétlenül azt, hogy a diszkre is kikerülnek az adatok, lehet, hogy csak a felhasználói címtartományból a kernel címtartományához tartozó bufferbe kerülnek, ekkor egy rendszerösszeomlás adatvesztést fog okozni.
- A bufferezés során mindig megtörténik a felhasználói címtartományból a kernel címtartományba másolás, onnan a diszkre való írás, vagy fordítva ugyanez az olvasásnál. Nagy adatforgalom esetén az extra másolás csökkenti a teljesítményt, akkor jobb lenne közvetlenül a felhasználói címekre írni. Megbontaná azonban az egységes felületet, ha ezt meg szeretnénk oldani. Kis adatforgalom esetén viszont nagyon valószínű, hogy a bufferezés gyorsít. (És ki tudja azt definiálni, hogy mi a nagy és mi a kicsi adatforgalom?)

9. Rendszermenedzseri feladatok

Ezzel a témakörrel - részletesebben - három választható tárgyuk is foglalkozik majd. Ezek a GEIAL206 Operációs rendszerek menedzselése, a GEIAL207 Biztonság és védelem a számítástechnikában és a GEIAL208 Windows 2000 rendszergazdai ismeretek tantárgyak. Miután ezek választhatók, nem biztos, hogy minden informatikus hallgató felveszi azokat. A legalapvetőbb rendszermenedzseri feladatokat azonban illik minden informatikusnak ismerni, ezért lerövidítve összefoglaljuk és megismerjük azokat e tárgyban is. Az érintett rendszermenedzseri feladatokat Unix operációs rendszerbeli feladatokon mutatjuk be, azon gyakoroltatjuk.

9.1. Összefoglalás

A rendszermenedzser legfontosabb feladatai:

- a rendszer installálása, hangolása (setting up), méretre alakítása, karbantartása (updating), erőforrás kezelés, kontrol: újabb eszközök, perifériák felvétele, levétele (connecting devices) [ezt most nem részletezzük].
- A rendszer indítása, leállítása (startup-shutdown) [röviden foglalkozunk vele].
- Konfigurációs fájlok karbantartása, daemonok indítása, adott időben futtatandó parancsok indítása (crontab), kommunikációs beállítások stb.
- A felhasználók menedzselése (felvétel, törlés, jogosultságok kiosztása stb.) [röviden foglalkozunk vele, majd a NIS rendszer koncepcióval is].
- A fájlrendszer mentése, visszaállítása (backup, restore, recovery) [röviden foglalkozunk vele].
- A fájlrendszer integritás biztosítása (fsck) [röviden foglalkozunk vele], szabad terület (free space) karbantartás.
- Naplózások, események figyelése (monitoring), statisztikák készítése, számlázás.

Szinte mindegyik tevékenység kapcsolatos a biztonsággal. Nagy rendszereknél a rendszermenedzser mellett külön biztonsági menedzsert (security manager) foglalkoztatnak. A fontossága miatt a kockázati és biztonsági kérdéseket is összefoglaljuk a rendszermenedzseri feladatok érintése előtt.

9.2. Kockázatok és védelem

Általános szabály: a biztonságossá tételnek ára van. Ezért figyelembe kell venni

- a gép (rendszer) fontosságát,
- a biztonsági munkák mennyiségét,
- a biztonsági komponensek hatását a felhasználókra.

Egyensúlyt kell keresni, hogy a biztonsági komponensek ne legyenek bosszantóak, hátráltatóak.

A fenyegetések osztályai

I. Fizikai fenyegetések

- Természeti csapások (tűz, földrengés stb.)
- Jogosulatlan hozzáférések laboratóriumokhoz, gépekhez, berendezésekhez (betörés, kulcsmásolat készítés, beléptető rendszer kijátszása stb.).

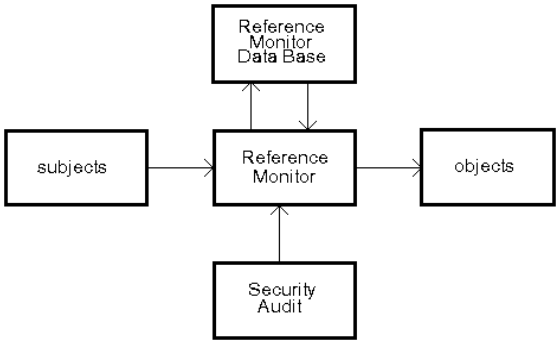
II. Logikai fenyegetések

- Felhasználók felelőtlensége, tévedése (pl. meggondolatlan törlések: del *.*).
- Jogosult szolgáltatás megtagadás valamilyen probléma miatt. (Pl. garantált válaszidőt meghalad a szolgáltatás, és ennek jogi, gazdasági következményei vannak, eszköz tönkremenetel miatt adatvesztés és biztonsági másolatok nincsenek stb.).
- Jogosulatlan hozzáférések erőforrásokhoz, információkhoz, szolgáltatásokhoz. Ezen belül érdemes külön kezelni a következő osztályokat:
  - felhasználók kíváncsisága, jogosultsági határainak kipróbálása, a biztonsági háló lyukainak keresésére tett próbálkozások;
  - behatolás károkozási szándékkal, a biztonsági háló lyukainak felhasználása kártevő módon: copyright sértés, információ eltulajdonítás, kémkedés, személyiségi jogok sértése, "gépi idő" lopás, információk törlése, baktérium-vírus-worms programok készítése stb.

A felhasználók tévedései ellen alig szoktak központilag szervezetten védekezni, bár a rendszeres, központilag szervezett mentések itt is segíthetnek (ezek célja azonban más). Vegye mindenki figyelembe a régi közmondást: Don't put all your eggs in one basket! Célszerű biztonsági másolatokat készíteni és azokat "más " helyen őrizni!

A következőkben a védelemmel kapcsolatos koncepciókat, alapfogalmakat tekintjük át.

9.2.1. A hivatkozás figyelés (Reference Monitor) koncepció



9.1. ábra. A hivatkozás figyelés koncepció

A Reference Monitor koncepció a 60-as években kidolgozott koncepció, a többfelhasználós számítógéprendszerek "hozzáféréses" típusú védelmi problémáinak megoldására. A koncepció lényege az alábbi (9.1) ábrán foglalható össze.

Subjects: (szubjektumok): aktív entitások, melyek objektumokat tudnak "elérni". Ide tartoznak a felhasználók, a processzek, taskok, job-ok.

Objects (objektumok): passzív entitások, erőforrások, szolgáltatások. Ilyenek pl. a számítógépek, CPU-k, a memóriák, eszközök, fájlok, programok stb.

Reference Monitor Data Base: definiált biztonsági követelmények, azaz mely szubjektumok, kinek az érdekében, mely objektumokhoz, hogyan férhetnek hozzá.

Security Audit (biztonsági figyelő, watchdog): a hozzáférési kísérletek (sikertelen/sikeres) naplózása, riasztás.

Reference Monitor: a rendszer központi eleme. Bármilyen szubjektum, ha egy objektumhoz akar férni, csakis a Reference Monitor-on keresztül férhet hozzá. Ez azonosítja a szubjektumot (authentication), és leellenőrzi a Reference Monitor Data Base-n át a hozzáférés jogosultságot (authorisation).

A hivatkozás figyelés koncepció - egyetlen közös adatbázissal - sohasem valósult meg, de részei szinte minden rendszerben megtalálhatók.

9.2.2. További alapfogalmak

Az azonosítás (authentication) fogalma

A szubjektumok (a felhasználók és a felhasználók nevében eljáró processzek) azonosítandók. Az azonosítás célja megmondani, hogy a szubjektum mely *védelmi tartományba* (protection domain) tartozik.

A felhasználók azonosítására vannak *külső és belső azonosítási technikák.* Pl. külső azonosítási lehetőség mágneskártyás vagy vonalkódos engedélyeztető rendszer, gépet, szobát lezáró kulcs stb. Belső azonosítási technika pl. a *jelszós* (password) védelem, vagy néhány, csakis a felhasználó által ismert információ (pl. gyermekkori betegség neve stb.) lekérdezése egy rövid dialógusban.

Jellegzetes problémakör a jelszós azonosítás problémaköre, ez ugyan egy kiváló azonosítási lehetőség, de egyben egy lehetséges lyuk a biztonsági hálón.

A hozzáférési jogosultságok (authorisation) fogalomköre

Objektumok (erőforrások) elérése, ezekhez való hozzáférések privilégiumainak gyűjtőneve az *authorisation*. Példákon keresztül talán könnyebb megérteni.

Fájlokat (ezek objektumok) lehet

- olvasni r (read),
- írni, újraírni w, rw (write, rewrite), lehet tartalmukhoz
- hozzáfűzni a (append), lehet azokat
- törölni d (delete),
- végrehajtani x (exec).

Számítógépeken, CPU-n lehet alkalmazásokat, rendszerprogramokat futtatni.

Eszközökhöz is lehetnek hozzáférések, nagyrészt hasonlóak a fájlokhoz való hozzáférésekhez (r, w, rw, a, d). Üzenetsorokba lehet üzeneteket elhelyezni, onnan kiolvasni, lehet üzenetsort megszüntetni: ezek is jelezhetők a w, r, d betűkkel.

A védelmi tartomány (Protection Domain)

Az Oxford számítástechnikai értelmező szótár [Novotrade, 1988] - további magyarázatokkal az alábbi definíciót adja: a védelmi tartomány a védett erőforrások hozzáférési privilégiumainak összessége.

Meglehetősen absztrakt fogalom, ezért körüljárjuk.

MS-DOS védelmi tartomány

Nézzünk egy egészen konkrét példát: az MS-DOS *command.com* programja fut. Különösebb autentikáció nem volt, az MS-DOS elindításával "beléptünk" a védelmi tartományába. A *command.com* védelmi tartománya a legtöbb fájlhoz törlési jogot ad. de az *IO.SYS* és az *MSDOS.SYS* fájlokhoz ebben a tartományban nincs törlési jog. A *del* paranccsal (ez része a *command.com*-nak) nem lehet ezeket törölni. Ugyanebben a tartományban a *PRN* eszközhöz sincs *delete* jog, *írás* jog viszont van hozzá!

Unix példa a védelmi tartományokra

Az *uid* és *gid* - melyek meghatározzák (authentication) ki vagy te és mely csoportba tartozol - védelmi tartományokat is meghatároznak. Tulajdonképpen két tartományt

- az uid-dal azonosított tartományt (veled azonosított védelmi tartományt),
- a gid-del azonosított tartományt (a csoportod védelmi tartománya).

Hány védelmi tartomány van a Unix-ban?

Elég sok, ahogy ez az eddigiekből következik:

- ahány felhasználói számlaszám létezik,
- ahány csoport létezik,

legalább annyi védelmi tartomány van.

Lehetségesek az uid/gid párokkal meghatározott védelmi tartományokon kívüli tartományok?

Igen! A védelmi tartomány általánosabb fogalom, ha valahogy azonosítható és a hozzáférési jogok valahogy rögzíthetők, akkor máris van védelmi tartomány. Pl. a *zeus dialup* jelszava külön védelmi tartományt biztosít a *zeus* CPU-i elérése számára. Egy CPU felhasználói mód - kernel mód váltása is védelmi tartomány váltás: itt az autentikáció a szabályozott módváltás (trap fogalom), az authorizáció pedig a szélesebb címtartomány elérési, a nagyobb instrukciókészlet használati jogosultság biztosítás.

A processzek egy vagy több védelmi tartományba futnak. Az *authentikáció* célja éppen megmondani, mely védelmi tartomány(ok)ban fut egy folyamat.

Az erőforrások (objektumok)egy vagy több védelmi tartományhoz tartoznak. A védelmi tartomány megmondja a hozzáférési jogokat.

9.2.3. Hozzáférési jogok listája, jogosultsági listák

A védelmi hálók leírásának két, különböző szervezettségű formája lehetséges. Az egyik a *hozzáférési jogok listája* (Access Control List, ACL), rövidebben *hozzáférési lista* (Access List), a másik a *jogosultsági listák* (Capatibility List, CL) formája.

Analógiát keresve a jogosultság úgy tekinthető, mint egy meghívó egy bálra, a meghívót bemutatva beléphetünk, míg a hozzáférési lista egy bál meghívottainak névsora, a belépéskor azt ellenőrzik, rajta vagyunk-e a listán.

Hozzáférési lista

A forma lényege, hogy maga az objektum - hozzákapcsolt attribútumokkal - tárolja védelmi tartomány azonosítási lehetőséget és az objektumhoz való hozzáférési jogokat. Ha úgy tetszik, a lista "sorokból" áll. Sorai az objektumok, hozzá felsorolva, mely védelmi tartományban milyen hozzáférési jogok vannak:

```
CPU1:   (PD1: x,  PD2: x, ...  PDi: x, ...  PDn: x)
  ...
mem1:   (PD1:rwx)
  ...
file1: (PD1:rwx, PD2: rw)
file2: (PD1: r,  PD2: rw, ... PDi: r, ...  PDn: r)
...
```

A hozzáférések ellenőrzéséhez a folyamatokról csak annyit kell tudni, hogy az melyik védelmi tartományban fut. Ennek a formának előnye, hogy az ACL viszonylag rövid lista,

többnyire könnyen kezelhető. Hátránya, hogy a lista sorok változó hosszúságúak lehetnek, fájl rendszerre nagyon hosszúak és itt már nehezen kezelhetőek.

ACL jellegű a Unix (fájl) védelmi rendszere (később látjuk, hogy egyszerűsített a forma), a Windows NT védelmi hálója, a VAX/VMS fájl- és eszköz védelmi rendszere (ez is egyszerűsített), a bitmintákkal védett valós címzésű memória partíciókkal dolgozó memória menedzsment memóriavédelme (IBM 360) stb.

A jogosultsági lista

Ez a forma - meglehetősen régi - egyre nagyobb jelentőségű: osztott rendszerekben, hálózatokban lehetetlen a védelmi tartományokat (akár a szegregációval egyszerűsített ACL-eket is) tárolni.

Az Oxford szótár definíciója: a jogosultsági lista az engedélyezett műveletek jegyzéke.

A forma: sorok az egyes védelmi tartományokhoz, bennük felsorolva az objektumok és a hozzáférési jogok:

```
PD1: (CPU1: x, mem1: rwx, file1: rwx, file2: r, ... )
PD2: (CPU1: x, file1: r,  file2: rw, ...)
 ...
PDi: (CPU1: x, file2: r, ...)
 ...
PDn: (CPU1: x, file2: r, ...)
```

A jogosultság (capatibility) fogalmat kell megértenünk, ez azt jelenti, hogy egy processz mit csinálhat az objektummal. Ha nincs jogosultsága, akkor nem érheti el. A processzhez kapcsolódnak a jogosultságok, a processznek van jogosultsági listája (hiszen a processz egy vagy több védelmi tartományban fut, azok jogosultságai vannak a processzhez rendelve). Tehát nem az objektumok attribútumai tárolják a hozzáférési kódokat, hanem a processzekhez rendelik a jogosultságokat.

Tipikus példa erre a felhasználói mód - kernel mód váltás: a processzhez más jogosultság (capatibility) kapcsolódik a váltással. Másik tipikus példa a valós címzésű, memória partíciókkal gazdálkodó rendszerek esetén a partíció határokat regiszterekben tároló memória védelmi rendszer: miután a regiszterértékek a procesz kontextusához tartoznak, a processzhez kapcsolódik a "jogosultság" ellenőrzési információ.

Egy processz átadhatja a "jogosultságát" egy másik processznek, ekkor a másik processznek is lesz jogosultsága a objektumhoz, a jogosultság felíródik a processz jogosultsági listájára. Természetesen ekkor a szokásos jogosultságokon (rwx, d, a) kívül további "jogosultságokkal" (jogosultság átadási jog: grant, jogosultság átvételi jog: take) is kell foglakozni, ezek explicite vagy implicite benne kell legyenek a rendszerben.

A Unix egyszerűsített fájl-hozzáférési modellje

A Unix-ban minden fájl, mondhatjuk, a fájlvédelemmel általánosan megoldanak egy sor védelmi problémát. Az eszközök a speciális fájljaik segítségével kezelhetők, a csövek - bármilyen is az implementációjuk - szintén a fájlrendszeren keresztül védettek. A Unixban a szubjektumok mindig processzek.

Minden fájlhoz az i-bögben (i-node), egyéb objektumokhoz, pl. üzenetsorokhoz (msg), szemaforokhoz (semaphores) stb. valahol tárolják

- az objektum tulajdonosát (uid),
- a csoporttulajdonost (gid),
- és implicite ezzel a többieket (others), akiknek nincs tulajdonosi relációja az objektumhoz.

Ezzel tulajdonképpen a védelmi tartományokat szegregálják. Az így szegregált védelmi tartományokhoz tartozó hozzáférési jogokat (rwx) is az objektumhoz rendelve tárolják: a fájloknál az i-bögben, egyéb objektumoknál valahol.

Ez a hozzáférési lista tehát nem hosszú, könnyen elfér az i-bögben. Hátránya viszont, hogy nem olyan flexibilis, mint a teljes ACL rendszer, egy fájlnál nem lehet különböző csoportokat - ezzel védelmi tartományokat - azonosítani, egy fájl csakis egy csoporthoz tartozhat.

A lényeg tulajdonképpen az, hogy a védelmi tartományok a tulajdonossági kategóriákon keresztül azonosítottak:

	PD1	**PD2**	**PD3**
	owner	group-ownership	no-ownership
	uid	gid	others
file1:	rwx	r--	---

Ebből az látható, hogy egy fájl három védelmi tartományba tartozik.

Mint láthatjuk, a hozzáféréseknek csak három kategóriája van: rwx. A szokásos fájloknál, eszközöknél, csöveknél nem nehéz a hozzáférések értelmezése:

- r read: olvasható a fájl, de nem változtatható, nem is törölhető, nem futtatható, még akkor sem, ha ez különben futtatható program, vagy burok program.
- w write: engedély a változtatásra. beleírhatsz akár az elejétől kezdve (ezzel átírhatod, hozzáfűzhetsz, törölheted. Jegyezzük meg, hogy egy szövegfájl, amire csak w engedélyünk van nem tölthető be egy szövegszerkesztőbe: hiányzik az r jog, de hozzáfűzhetünk adatokat.
- x execute: engedély a futtatásra. Bináris fájloknál ez elegendő is a futtatásra. Burokprogramoknál a futtatáshoz szükséges az r jog is, mert a burok olvasni is akarja a fájlt.

A jegyzékekre vonatkozó rwx jogosultságok értelmezése:

- r olvashatja a jegyzéket, ezzel listázhatja a tartalmát, pl. ls paranccsal. Fájlnév behelyettesítéshez (dzsóker használathoz) szükséges jog. A jegyzékbe bejegyzett fájlokhoz azonban a puszta r jog nem enged hozzáférni.
- w írhatja a jegyzéket, azaz "betehet és kivehet" fájlokat e jegyzékbe, jegyzékből. Cserélhet a fájlneveket.
- x itt nem futtatást engedélyez, hanem hozzáférést magukhoz a fájlokhoz, amik ebben a jegyzékben vannak. A cd parancshoz szükséges. Csak x joggal r nélkül hozzáférhetünk a bejegyzett fájlokhoz, ha tudjuk a teljes nevüket.

A védelmi rendszer működéséhez tudni kell, hogy a hozzáférni akaró processz milyen védelmi tartományokban fut.

A Unix-ban a processzeknek is van

- valós és effektív tulajdonosa (uid-dal azonosítva),
- valós és effektív tulajdonos csoportja (gid-del azonosítva).

A valós és effektív tulajdonosok többnyire egybeesnek, de setuid koncepció (lásd később) szerint különbözhetnek is. A védelmi tartományokat, melyben egy processz fut, az effektív tulajdonosságok határozzák meg, ezzel a processz két védelmi tartományban fut.

A védelmi politika

1. Először a hozzáférni akaró processz effektív tulajdonosági kódja és a fájl tulajdonossági kódja összevetődik. Ha itt egyezés van, akkor a fájltulajdonoshoz rendelt jogosultság (rwx bitminta) szerint biztosított a hozzáférés.

2. Ha az előző összevetésben nincs egyezés, akkor a csoport tulajdonosságok vetődnek össze (a processz effektív csoport tulajdonos kódja!). Egyezés esetén a fájl i-bögjéből a csoporthoz tartozó bitminta szerinti a hozzáférés.

3. Ha az előző két összevetés "sikertelen", akkor az others bitminta szabja meg a hozzáférést.

Ez azt jelenti, hogy tulajdonos hozzáférését a fájlvédelmi minta tulajdonos része határozza meg, hiába van nagyobb jog akár a csoport, akár az others bitmintában. Egy fájl, aminek védelmi mintája a következő

```
---r--rwx
```

nem elérhető a tulajdonos számára, a csoport tagjai ugyan olvashatják, de nem írhatják, mialatt mindenki más, aki nem tartozik a csoporthoz teljes hozzáféréssel rendelkezik. Ezért ez a hozzáférési lista nem valami hasznos!

A setuid koncepció (D. Ritchie)

Minden processz rendelkezik

- valós tulajdonosi, csoporttulajdonosi azonosítóval, és
- effektív tulajdonosi, csoporttulajdonosi azonosítóval.

A legtöbb esetben a valós és az effektív tulajdonosságok ugyanazok. A valós tulajdonosságot a processz a szülőjétől örökli, vagy a szülője állítja be neki.

Gyakran szükség van arra, hogy különleges többlet-jogokat biztosítsunk processzeknek valamilyen szabályozott módon. Például, ha jelszót változtatunk a passwd segédprogrammal, be kell írnunk a jelszavakat tartalmazó fájlba, amire persze nekünk nem lehet írási jogunk, mert akkor bárkinek a jelszavát átírhatnánk, kitörölhetnénk stb. Hogy megoldják ezt a problémát, Ritchie javaslatára, a kernel megengedi, hogy olyan processzeket kreáljunk, amelyeknek többletjoguk van. Bizonyos végrehajtható fájlok ún. setuid/setgid (Set User Identification/Set Group ID) engedéllyel rendelkeznek (lásd man ls). Amikor egy ilyen programot futtatunk, a keletkezett processz valós tulajdonosa mi leszünk (valós csoport tulajdonosa a mi csoportunk), hiszen a mi shell-ünkből indítottuk, annak tulajdonosságait örökli. Az effektív tulajdonosa/csoport-tulajdonosa viszont az az uid/gid lesz, ami a betöltendő program fájlhoz tartozik. A fenti példában passwd futtatható program tulajdonosa a root (superuser), ez egyben setuid program. Ha futtatjuk, a processz valós tulajdonosa mi vagyunk, effektív tulajdonosa viszont a root. Mivel a jelszavakat tartalmazó fájl tulajdonosa szintén a root, a processz kicserélhet jelszavunkat, írhatja a fájlt.

Gyakorló feladatok

1. Tanulmányozzák az on-line manual-ban az umask, chmod és chown parancsokat!

2. Gyűjtsenek ki setuid engedéllyel rendelkező programokat.

3. Állítsanak be fájljaikra különböző, célszerű védelmi mintákat, ellenőrizzék a hozzáféréseket.

Összefoglalás

Léteznek védelmi tartományok (Protection Domains). Ezek valamilyen módon azonosítottak.

A processzek - tulajdonképpeni szubjektumok - védelmi tartományokban futnak. Az objektumok (passzív elemek, fájlok, jegyzékek, eszközök stb.) különböző elérésekkel (rwx stb.) kezelhetők.

A védelmi tartományok és az elérési jogok vagy hozzáférési lista (ACL) jelleggel (objektumokhoz kötött hozzáférési jogok védelmi tartományonként), vagy jogosultsági lista (CL) jelleggel (processzekhez kötött tartományokkénti jogosultságok) rögzítettek. A két forma ugyanazt az információtartalmat biztosítja.

A védelmi tartomány nagyon általános fogalom. Védelmi tartomány váltás van felhasználói mód - kernel mód váltásnál. Védelmi tartományok vannak egy számítógéprendszer eléréséhez, a kapcsolatépítéshez, az ülés létesítéséhez: a "belépéshez" stb.

Általános szabály: egy védelmi tartományba bejutni csak szabályozott módon lehet. Legtöbbször az azonosító/jelszó mechanizmuson keresztül, de vannak más mechanizmusok is.

Sok rendszerben a szokásos (ordinary) védelmi tartományok azonosítása a tulajdonossági kategóriákon keresztül történik, de vannak más módszerek is, melyek kiegészítik az előzőt.

9.3. Rendszerindítás, leállítás

Egy többtaszkos, esetleg többfelhasználós operációs rendszer indítása rendszerint bonyolultabb feladat, mint egy egyszerű gép bekapcsolás. Persze, ha jól installálták, jók az alapbeállítások (setup), jó indító (startup) konfigurációs burok programokat írtak hozzá, akkor lehet, hogy egyszerű az indítás. Nézzük át, mi is történik a *startup* során, miket kellhet a rendszermenedzsernek csinálni!

A gép bekapcsolása után az operációs rendszer indulása több fázisban történik. a fázisok során:

1. A hardver ROM rutinok futnak.
2. Az ún. *boot loader* fut.
3. Az operációs rendszer kernel inicializálódik.
4. A "kézből" induló processzek keletkeznek.

A ROM rutinok teszteléseket végeznek, ellenőrzik a memóriát, egyéb hardver komponenseket. Lehetséges, hogy interaktív menüt kínálnak, esetleg az intelligens kontrollerek felprogramozását segítik. Ezeket a gép szállítója adja a géphez. Végül kezdeményezik a boot-olást.

Maga a boot-olás rendszerint két fázisban történik: *first stage boot* és *second stage boot*.

A first stage boot során a kijelölt *partíció* 0. blokkjáról betöltődik a *kezdeti betöltő* (initial boot) program, és elindul. (Nem boot-olható partíció, eszköz kezdeti betöltő programja egy üzenetet ír ki, hogy nem rendszer diszkről akartunk boot-olni.) A kezdeti betöltő program akármilyen operációs rendszer betöltését kezdeményezheti, azzal, hogy indítja a *second stage boot* programot. Néha "be van drótozva" a second stage boot neve, helye a first stage boot-ba, néha bekéri a nevet a kozolról (ekkor már a konzol eszköz is ismert).

A second stage boot program lehet része egy operációs rendszer fájl-rendszerének: ott egy fájl, adott névvel, de speciális helyen van, hogy a kezdeti boot program - ami rövid program, elfér egy blokkon - felismerhesse. Lássuk be, hogy a két boot program együttműködő kell legyen.

A second stage boot-ba "be lehet drótozva" az operációs rendszer kerneljének neve, de az is lehet, hogy adhat készenléti jelet (prompt) a konzolra (rendszerint ez a kettőspont (colon :)), kérve a kernel nevét, helyét stb. Ezek természetesen operációs rendszertől függő dolgok. Ha kéri, akkor meg kell adni a diszkvezérlő nevét (a driver program része a second satge boot-nak), a partíciót, egy eltolás (offset) blokkszámot (az átvizsgálandó partíció kezdetétől számítva honnan keressen) és a kernel nevét.

Egy példa erre:

```
: rm (0,0) unix
```

ahol

- rm: a kontroller azonosító (driver-t azonosítja),
- 0: első partíció (a drive száma),
- 0: kezdetétől keress,
- unix: ez a kernel neve, töltsd be.

A kernel - miután betöltődött -indul: inicializálódik. Ezzel tulajdonképpen felkészíti a hardvert és a szoftvert használatra. Pl. ellenőrzi és belső tábláiba feljegyzi a rendelkezésére álló memóriát a virtuális memóriakezeléshez, inicializálja az eszköz driver-eket.

Az inicializált kernel végül kreál processzeket. "Kézből" készül a 0. pid-ű processz, SVID Unixnál ez a swapper, más, újabb Unix-oknál a sched. Nem találunk swapper, vagy sched futtatható program-fájlt a fájl-rendszerben, az a kernel része.. A swapper/sched feladata lesz az ütemezés. Most azonban első tevékenységeként elkészíti az 1. pid-ű processzt, az init processzt.

Az init processz lesz általában minden más processz szülője. Állapotai (states): boot, normál, powerfail. Konfigurációs fájlja a /etc/inittab (SVID rendszerekben), vagy a /etc/ttys (BSD rendszerekben). Szignálok hatására az init a konfigurációs fájljához fordul, az abban meghatározottaknak megfelelően cselekszik.

Boot állapotban az init dátum/idő/időzóna ellenőrzést, beállítást végeztethet; megkérdezheti, akarunk-e fájlrendszer ellenőrzést (fsck) végeztetni; feldolgozza az rc (run command) szkript(ek)et. Ez készít mount táblát (ha még nincs), és mount-olja a fájlrendszereket; letisztítja a /tmp jegyzéket; daemonokat indít (pl. a cron daemont, hálózati, számlázási daemonokat stb.).

A boot állapothoz a single user level tartozik (lásd később).

Normál állapotban a terminál vonalon jövő "jelentkezés" hatására gyermek processzt kreál, abba betölti a getty futtatható programot, ez később magára tölti a login programot, a login pedig a felhasználó kezdeti programját: rendszerint egy burkot (ezért lesz az ülésünk kezdeti burkának szülő processze az init). A normál állapothoz a multi user level tartozik.

Az init *powerfail állapotába* lép olyan rendszereken, melyek képesek az áramkimaradást észlelni. Ilyenkor veszélyhelyzeti eljárásokat (emergency procedures) hajthat végre az init. (Pl. a sync parancsot a fájl-rendszer integritás megőrzésére.)

A Unix futási szintek (Run Levels)

Az init működése és a /etc/inittab szerkezetének megértéséhez meg kell ismerkednünk a Unix System III-ban bevezetett, az SVID-ben továbbfejlesztett futási szint koncepcióval (a BSD Unix-ok kicsit különböznek).

Az SVID-ben a futási szint két értelemben is használatos. Az *egyik értelemben* létezik:

- egyfelhasználós szint (single user level), jele: s, S. Az init boot állapotához ez a szint tartozik.
- többfelhasználós szint (multi user level), jele: 2. Ez tartozik az init normál állapotához.

Az egyfelhasználós szint startup során érdekes, ezen a szinten történhet a fájl-rendszer integritás ellenőrzés, a fájl-rendszerek mountolása (jóllehet, átléphet a startup során a rendszer ezen a szinten). Egyfelhasználós szintre válthat a rendszermenedzser, ha karbantartási feladatokat akar végezni.

A *futási szintek a másik értelemben* 0-6 numerikus szintjelzést kapnak: ez a szint értelem csakis az inittab bejegyzéseinek kiválasztására valók. A numerikus szint koncepciót használhatja a rendszermenedzser a vonalak (portok) elérésének kontrollálására.

Az inittab szerkezete, az init működése

Az init működését három dolog vezérli: a pillanatnyi állapota (state), a pillanatnyi futási szint (run level) és az esemény jelzése (signal), ami bekövetkezett. Az init e három információ szerint, beolvasva az inittab-ba, kiválasztja a megfelelő sort, és aszerint cselekszik.

Az inittab egy tipikus sora a következő szerkezetű:

```
label:run level:action: program to start
```

Példa:

```
co:2:respawn:/etc/getty console co_9600
```

A *címke* (label) mező szerepe a számlázásnál, monitorozásnál jön elő: programok indításánál feljegyződik, melyik inittab sorral történt az indítás.

A *run level* mező szerepe az ellenőrzés: ha a pillanatnyi szintnek megfelel a mező, akkor a sor negyedik mezejében lévő program indulhat (vagy folytatódhat, ha létezik). Ha nem felel meg: a sorhoz kapcsolódó processz 20 másodpercen belül hangup szignált kap, hogy terminálódjon; ha nem létezik, nem is indul. Itt számítanak a numerikus szintek is. A mező lehet üres (ugyanaz, mintha minden szintet felsorolnánk), lehet szint-jelek felsorolása.

Az *action mező* reprezentálja az akciót. 11 különböző akció lehetséges, néhányat ismertetünk:

- sysinit: Boot állapothoz tartozik. Indítja a programot amikor az init először olvassa az inittab-ot.
- respawn: (Normal state akció.) Indítsd a programot, és indítsd újra mindenkor, ha befejeződött. A getty-hoz használjuk.
- wait: (Normal state.) Indítsd a programot és várd meg a befejeződését, mielőtt a következő sort feldolgoznád.
- off: (normal state.) Ha a program (vagy leszármazottja) él, akkor terminálod.
- once: (Normal.) Indítsd el a programot egyszer, és addig ne indíts újra, míg az (vagy leszármazottja) él.

Tanulmányozzák a /etc/inittab fájlt különböző rendszerekben! Nézzék a manual lapot is!

Az init-tel való kommunikáció

A rendszermenedzser (superuser) küldhet szignált az init-nek akár a telinit, akár az init paranccsal. Tanulmányozzák a manuel-ben.

A rendszer leállítása

A szokásos rendszerzárást a /etc/shutdown burokprogrammal végzi a rendszermenedzser (kiadása előtt a gyökér jegyzék legyen az aktuális jegyzék, az unmont-ok miatt). Grafikus felhasználói felületnél toolchest menüelemmel is zárhat rendszert.

A shutdown

- figyelmezteti a felhasználókat a rendszerzárásra;
- leállítja a daemon processzeket;
- terminálja az aktív processzeket;
- umount-olja a fájlrendszereket;
- single user futási módba állítja a rendszert (init s);
- kiad sync parancsot

Tanulmányozzák a manualben!

Fájlrendszer konzisztencia ellenőrzés (fsck)

Nem megfelelő eszközkezelés, nem megfelelő rendszerzárás esetén a fájl-rendszer "szétesheT". A szétesés leggyakoribb okai:

- áramkimaradás miatti nem normális rendszerzárás;
- kivehető médium (pl. floppy) kivétele umount előtt.

A szétesett fájl-rendszerben a hibák:

- A szuperblokk módosított, de csak az in-core változatában (a diszkre nem íródott ki).
- Vannak blokkok, melyek nem tartoznak fájlokhoz, de nincsenek a szabad listán sem.
- Vannak blokkok, melyek a szabad listán is, és valamelyik fájl i-bögben is be vannak jegyezve.
- Vannak jegyzék (directory) bejegyzések, melyekben az i nem mutat érvényes i-bögre.
- Vannak i-bögök, melyekben a link számok nagyobbak, mint ahány jegyzékből van az i-bögre utalás.

Mi jelzi a szétesést?

A szuperblokk egy mezője. Vagy az a tény, hogy a szuperblokkba bejegyzett i lista méret és a valódi i lista méret nem egyezik. Segédprogramokkal lekérdezhető, vajon egy fájl-rendszer szétesett-e vagy sem, de legjobb minden mountolás előtt az fsck segédprogrammal ellenőrizni, és megpróbálni rendbehozni a fájl-rendszert.

Az fsck indítása (csak szuperuser, és csakis nem mountolt állapotban):

```
# fsck special-file
```

Az fsck hat fázisban fut.

1. fázis: az i-bögök ellenőrzése, adatgyűjtés a további fázisokhoz

Minden i-bögben

- érvényes fájl típus bejegyzés kell legyen;
- nem nulla linkszám kell legyen;
- érvényes blokk mutatók kellenek;
- jó fájl-méret bejegyzés kell legyen.

Ebben a fázisban eljegyződnek az érvényes blokk címek, a link-számok, az i-bög állapotok, az érvényes i-bögök.

2. fázis: az ösvények ellenőrzése

Minden jegyzék bejegyzés ellenőrződik a gyökértől kezdve! Egy directory bejegyzés érvényes i-bögre kell mutasson (adatok az 1. fázisból).

Ellenőrződik, hogy az 1. fázisban gyűjtött hivatkozások és link számok jók-e? (Minden fájl szerepel valamelyik jegyzékben? Megfelelő számú jegyzékben?)

3. fázis: kapcsolat ellenőrzés

Ha valamelyik dir típusú i-bögnek nincs dir-beli bejegyzése (ezt a 2. fázisban felfedeztük), akkor készüljön dir bejegyzés neki!

4. fázis: hivatkozások ellenőrzése

Nem dir típusú i-bögnél is előfordulhat, hogy nincs megfelelő számú dir bejegyzés hozzá. Márpedig a dir bejegyzések össz-számának (2. fázisban rögzítettük) meg kell egyezni a linkek számával, továbbá az összes i-bögök számának egyezni kell a szuperblokkba bejegyzett számmal.

Szükség esetén a nem dir típusú i-bögöknek is készül dir bejegyzés.

5. fázis: a szabad lista ellenőrzése

Végignézzük, hogy a szabad listán érvényes blokk címek vannak-e, ezekből szerepel-e valamelyik i-bögben is (egy blokk vagy a szabad listán, vagy egy i-bög bejegyzésben szerepelhet csak).

6. fázis: szabad lista helyreállítás

Levehető a szabad listáról egy helyes i-bögben címzett, érvényes blokk.

Felvehető a szabad listára az, amire nincs érvényes i-bögben hivatkozás (ne felejtsük, itt már a dir bejegyzések kiegészítettek!).

Gyakorló feladat:

Floppy lemezt mountoljunk, írjunk rá, vagy töröljünk róla valamit, és mielőtt umountolnánk, vegyük ki a lemezt! Ekkor nagy valószínűséggel nem lesz konzisztens a fájl-rendszer a floppyn. Most umontolhatunk, majd visszatéve a floppyt, fsck-val próbáljuk meg rendbetenni!

Fájl-rendszer készítés, használatba vétel

Tételezzük fel, létezik logikai diszk a rendszerünkben (van partíció, vagy diszk, pl. floppy, és van hozzá speciális fájl, driver program). Ekkor az mkfs segédprogram segítségével fájl-rendszert készíthet a szuperuser a logikai diszken.

Az mkfs először elkészíti a fájl-rendszer két részét: a szuperblokkot és az i-listát. Utána az adat blokkokat a szabad listára összegyűjti. Végül elkészíti a gyökér jegyzéket (i-bög = 2), ennek blokkját le is veszi a szabad listáról.

Tanulmányozzuk a man-ban az mkfs-t! Használatához minimálisan a logikai diszk nevét (speciális fájlja nevét), esetleg a méretét (blokkban) kell megadni.

```
# mkfs diskname size
```

Figyelem! Az mkfs felülírja a diszket! Ami volt rajta, elvész.

További argumentumok is adhatók az mkfs-nek: pl. az i-lista mérete megszabható. Az alapértelmezési i-lista méret úgy számítódik, hogy 4K-ként lesz i-bög. azaz átlagosan 4K-s fájlokra számítunk. Ha kevesebbel is beérjük vagy többre van szükségünk, használjuk az mkfs-t így:

```
# mkfs diskname size:inode
```

Az /etc/labelit segédprogrammal címkét adhatunk a diszknek. A címke használata a mountolásnál ellenőrzésre jó: a mount figyelmeztet, ha a fájl-rendszer címkéje és a mount-pont neve nem egyezik.

A lost+found jegyzék

Az új fájl-rendszer elkészítése után célszerű készíteni lost+found jegyzéket, ugyanis az fsck használja ezt! Persze, ezt csak mountolás után tehetjük. A következő parancsok pl. elkészítik és jó engedélyeket adnak e jegyzéknek:

```
# mkdir /ures-dir
# mount diskname /ures-dir
# cd /ures-dir
# mkdir lost+found
# chmod 777 lost+found
```

Most megvan a lost+found a célszerű engedélyekkel. Van benne két bejegyzés is, a . és a .. jegyzék. Van benne hely valamennyi további bejegyzésre (62, feltéve, hogy 1024-esek a blokkok). Miután az fsck több bejegyzési helyet is kívánhat, célszerű a lost+found méretét megnövelni, mielőtt tényleg használatba vennénk a diszket! Ezt úgy szokták csinálni, hogy ciklusban fájlokat készítenek (méretük nem számít!), amiket a lost+found-ba jegyeztetnek be, majd letörlik ezeket a fájlokat. Az eredmény az lesz, hogy a lost+found mérete (a hozzátartozó adat blokk szám megnövekszik. (Az egész játék arra jó, hogy egy későbbi fsck működés közben ne kelljen a lost+found számára blokkokat foglalni, hiszen akkor gond lehet az érvényes-nem érvényes blokkokkal.) (Némely rendszerben az mkfs készít megfelelő lost+found-ot is.)

Gyakorlat:

Floppy lemezen alakítsanak ki fájl-rendszert! Tegyék használhatóvá mountolással! Ellenőrizzék, ha szükséges készítsék el a megfelelő lost+found-ot. Végül használják a fájlrendszert. Gyakorolják az umount-ot is.

9.4. A felhasználók menedzselése

Számlaszámrendszer, ennek menedzselése

A *számlaszám* (account) egy azonosító, nevéből következően

- erőforrás felhasználás számlázására, nyilvántartására stb. szolgál, de ezen kívül jó

- tulajdonossági kategóriák rögzítésére (ezzel védelmi tartományok azonosítására), a védelmi háló kialakítására.

Osztályai

I. használat szerint

- Bejelentkezésre (kapcsolat + ülés létesítésre) szolgáló személyes használatú számlaszámok. Ezek a *szokásos* (ordinary) felhasználói számlaszámok.
- Bejelentkezésre nem szolgáló, de a *tulajdonosságot jelölő* számlaszámok (bizonyos system account-ok).

II. A védelmi háló szerint

- *Korlátozott jogokat biztosító* (restricted) számlaszámok, mint pl. egy titkárnői számlaszám.
- *Szokásos* (ordinary) számlaszámok, pl. fejlesztő munkára, általános használatra.
- *Privilegizált* számlaszámok, melyeket a rendszermenedzserek, biztonsági menedzserek, programrendszer felelősök stb. kaphatnak.

Egy számlaszám komponensei

- A login név: *lname*, amit a rendszermenedzser és a felhasználó közösen, megegyezéssel választ, hogy egyedi legyen.
- A hozzátartozó, változtatható *jelszó* (password), néha több jelszó. Kezdeti értékét a rendszergazda adja, közli a felhasználóval, aki utána megváltoztathatja. Néha kötelezik is a változtatásra.
- Belső azonosító: *uid*, *UIC*. Ezt a rendszergazda választja, rendszerint egy egész szám. Feltétlenül egyedi. Konvenciók lehetnek a kiválasztásánál.
- Csoport név: *groupname* (rendszergazda és felhasználó megegyezve választják, vagy csoport nevek.
- Csoport azonosító: *gid*, *GUI*. Rendszergazda választja. Néha több csoport azonosító kell. Konvenciók lehetnek a választásához.
- A *HOME*/*default* eszköz/jegyzék a bejelentkezési számlaszámokhoz. A rendszergazda választja. Néhol a felhasználó átállíthatja.
- A bejelentkezéskor *induló* processz program fájljának neve a bejelentkezési számlaszámokhoz. rendszerint ez egy burok, de lehet egy alkalmazás is (pl. titkárnőnek).
- Limitek és quoták a számlaszámhoz. *Capatibility list* jellegű! A Unixban ilyen csak közvetve van.
- Általános adatok a felhasználóról: teljes név, szoba szám stb., kommentár jellegű.

A Unix számlaszám rendszerhez tartozó fájlok

Ezeket a bejelentkezéskor használja a rendszer:

- /etc/passwd, ami *szuperuser* tulajdonú, de mindenki által olvasható ASCII fájl.
- /etc/group, *su* tulajdonú, de mindenki által olvasható ASCII fájl.
- /etc/init, az init processz program fájlja. Ez "figyeli a vonalakat", hogy egyéb processzek segítségével a fenti két fájlt használva ellenőrzött ülés létesítést biztosítson.

Sikeres kapcsolat+ülés létesítés után a kapcsolattartó processz fut, melynek tulajdonosa az *lname/uid/gid*-del jelölt személy. Védelmi tartományai a az *uid/gid*-del azonosítottak. Ennek gyermek processzei is ezekben a *protection domain*-ekben futnak, hacsak nincs valami korlátozó/kiterjesztő mechanizmus (setuid/setgid).

A /bin/passwd a jelszó állítására alkalmas program. Lehet "proactive", azaz olyan jelszóállító, ami megkövetel bizonyos szabályokat. Pl.:

- legalább x karakter hosszú legyen a jelszó;
- változatos karakterekből álljon;
- *password aging*: lejárati idővel rendelkezik, régi jelszavakra emlékszik, azokat nem engedi újra;
- jelszó generátor lehetőséget ad;
- stb.

A Unix /etc/passwd fájl sorainak mezői (: a mezőelválasztó):

(A fájlban egy-egy sor egy-egy számlaszám.)

- lname: a bejelentkezési név. Ha véletlenül nem egyedi, az első találati sor lesz az vizsgált sor.
- pwd: titkosított, vagy "shadow" (nem itt tárolt) jelszó. Lehet üres: ekkor nincs jelszó, ami tipikus védelmi lyuk, kerülendő, ellenőrizendő. Lehet letiltást, lejáratot jelző bejegyzés is.
- uid: egy egész szám, a belső azonosító.
- gid egy egész, a csoport belső azonosítója.
- teljes név (kommentár)
- HOME jegyzék (bejelentkezési jegyzék).
- startup program fájl

Ha az *uid* nem egyedi, a tulajdonossági attribútumok nem választhatók szét, gondok jelentkezhetnek. Vannak konvenciók a kiválasztáshoz (melyek CL jellegű védelmi tartomány rögzítésre alkalmasak):

- 0 a rendszergazda (root, superuser) azonosítója;
- -1 használatos a hibás számlaszámokhoz (invalid account);
- -2 az NFS nobody számlaszám;
- 1 - 10 rendszer számlaszámok;
- 11 - 99 fontos, kitüntetett személyek (pl. a uucp számlaszámai);
- 100-60000 szokásos felhasználók számlaszámai.

A gid-re vonatkozóan is vannak konvenciók (pl. a 0 a rendszer csoportja).

A Unix /etc/group fájl sorainak mezői (egy sor egy csoport):

- gname: a csoport név;
- pwd (csoport jelszó, a csoport védelmi tartomány elérésére volna, de nem használják)
- gid: a csoport belső azonosítója;
- bejelentkezési nevek listája, vessző szeparátorral elválasztva.

Lássuk be, összefüggés kell legyen e két fájl között. Lássuk be továbbá, hogy létező jegyzékekre, futtatható programokra való utalások vannak a /etc/passwd fájlban. Ezért a *superuser* (rendszermenedzser) e két fájl karbantartását erre a célra írt shell (esetleg fordított-linkelt végrehajtható) programokkal szokta végezni, amik összehangoltan kezelik a dolgokat. Miután azonban a /etc/passwd és a /etc/group fájlok egyszerű szövegfájlok, a rendszergazda egy szokásos szövegszerkesztővel is karbantarthatja őket. Ügyelnie kell azonban ekkor az összehangolásokra!

A login processz használhat még két fájlt:

/etc/dialups # az "őrzött" eszközök speciális fájlnevei;

/etc/d_passwd # a hozzájuk tartozó jelszók.

Rendszermenedzseri gyakorló feladat

Vegyünk fel új felhasználót.

- Válasszunk neki egyedi lname/uid párt.
- Válasszunk neki gname/gid párt. Mi van, ha nem létezik?
- Válasszunk neki HOME directory nevet. Ha ez a dir nemlétezik?
- Válasszunk neki startup programot.
- vi editorral írjuk be a sorát a /etc/passwd fájlba. Jó, ha biztonsági másolaton dolgozunk, és a végén átnevezzük!
- vi editorral írjuk be, editáljuk a sorát a /etc/group fájlba. Itt is biztonsági másolattal célszerű dolgozni.
- Készítsük el a HOME jegyzékét. Írjuk át ennek tulajdonosát uid/gid-re. Tegyünk bele "startup" fájlokat (.login, .profile stb.)
- Próbáljuk ki, jó-e.

Hogyan oldjuk ezt meg célszerű shell script segítségével?

9.5. A NIS (Network Information Services)

(Korábban Yellow Pages elnevezésű rendszer).

A NIS központosított adatbázis, ami a hálózat használatát hatékonyabbá teszi.

Tipikus példa lehet a hálózati névfeloldás: a hálózati csomópontok neveihez sokszor szükséges hozzárendelni az IP címet. Az egyedi gépeken a /etc/hosts fájl tartalmaz név-IP cím párokat, amiből a névfeloldás lokálisan megoldható. Ha nincs NIS rendszer, akkor ezt a fájlt az egyedi csomópontokon karban kell tartani, állandóad naprakész állapotba kell hozni. Ha viszont telepítünk NIS rendszert, a karbantartást csak egy gépen kell végezni, ez a gép a többi számára szolgáltathatja a karbantartott táblázatot.

Általános keresési szabály fogalmazható meg a NIS rendszer esetén. Általában valamilyen keresett információ előbb a NIS adatbázisban keresendő, aztán a helyi adatok között. Persze vannak fordított keresési sorrendek is, és lehet olyan keresés, amikor a helyi adatbázist meg sem nézik. Mindezt majd látni fogjuk.

A NIS kliens-szerver koncepciója

A szokásos kliens szerver koncepcióval dolgozik a NIS. Mégis, amikor *NIS kliens* kifejezést mondunk, az lehet, hogy egy csomópontot (egy host-ot) jelent a hálózaton, vagy lehet, hogy egy ilyen gépen futó processzt. Ugyanígy, a NIS szerver jelenthet egy gépet, vagy egy szolgáltató processzt ezen a gépen.

Egy NIS kliens gépen futó processz (ez is NIS kliens) küldhet egy *kérelmet* egy NIS szervernek (szerver gépen futó processznek). A kérelemben valamilyen információt igényel a NIS adatbázisból. A szerver processz kezeli az adatbázist, kiveszi a kért információt, és egy *válaszban* elküldi a kliens processznek.

A szerverek hierarchiája

Létezik a NIS rendszerben egy *master server*, és létezhetnek *slave server*-ek. A master server gépen tartják karban a NIS adatbázist. A slave gépek duplikátumokat tartanak az adatbázis-

ból: szerepük tulajdonképpen akkor van, ha valamilyen okból a master nem tudja kiszolgálni a klienstől jövő kérelmet. Ilyenkor a kérelmet valamelyik slave kapja meg, és az szolgál ki.

A NIS képek (maps)

A NIS adatbázis *képekből* áll. Egy *kép* (map) fájlok csoportja. A map-ekben *dbm* adatbázis formában találhatók az adatok, nem ASCII formában: ennek oka a gyorsabb keresés, a hatékonyabb tárolás. (Segédprogram konvertálhatja az ASCII fájlokat *dbm* formába.) Minden *képnek* van neve. A kliens gépeken futó alkalmazásoknak tudniuk kell ezeket a neveket, mert a kérelmeiket a map-ek neveivel adják ki, továbbá tudniuk kell a map-ekben tárolt információk formáját.

A *map*-ekben *keys* és *values* formában szokták az információkat tárolni.

Pl. a hosts.byname képben a keys egyedi gépek (host-ok)nevei; a values ezek IP címei lehetnek.

A NIS tartományok (domain)

Egy NIS tartomány gépek csoportja, melyek ugyanazt a NIS adatbázist használják. A tartományoknak van nevük. A tartományhoz tartozik egy *master server gép*, és tartozhat néhány *slave server*. Ezen kívül a tartományhoz tartozhat valamennyi *kliens gép*. Jegyezzük már most meg, hogy a *szerver gépek* egyben *kliens gépek* is.

A NIS tartomány egybeeshet az Internet tartománnyal, de ez nem kötelező. Sőt, a NIS tartományba tartozó gépek lehetnek különböző hálózatokon is.

A NIS adatbázis "home" jegyzéke a

`/usr/etc/yp`

vagy a

`/usr/etc/yp/domain_name`

jegyzék.

A NIS daemon processzek

Három daemon processz segíti a NIS rendszert. Ezek az ypbind, ypserv és az rpc.passwd processzek. Mielőtt szerepüket, feladatukat tárgyalnánk, foglaljuk össze, hol kell fussanak ezek a daemonok:

daemon	Kliens gép	Slave gép	Master gép
ypbind	x	x	x
ypserv		x	x
rpc.passwd			x

Az összekötés (binding) fogalma

Az összekötéssel "emlékszik" egy processz arra, hogy melyik szerver figyeli és szolgálja ki kérelmét. Az ypbind daemonnak futnia kell a kilens gépeken és a szerver gépeken is, mert az alkalmazások, amik információkat kérnek a NIS adatbázisból, a tartomány akármelyik gépén futhatnak. Az ypbind felelős azért, hogy az alkalmazások "emlékezzenek" arra, melyik ypserv daemont szólíthatják meg kérelmekkel.

Ha egy alkalmazás kér valamilyen információt, ami szokásosan helyi adatbázisban lenne, akkor az alkalmazásból hívott *futásideji könyvtári* (RTL) függvény a *binding* koncepció segítségével folyamodhat a NIS adatbázis információért. Az ypbind daemon segíti az alkalmazást, hogy megkapja, melyik *szerver gép* melyik *portján* folyamodhat az információért.

Az ypserv daemon kezeli a NIS adatbázist. Futnia kell a szervereken. Az ypserv fogadja a kéréseket, kiveszi az információt az adatbázisból és visszaküldi válaszüzenetben az alkalmazásnak.

A *master szerveren* futnia kell a /usr/etc/rpc.passwd daemonnak is. Ez a daemon a NIS adatbázis karbantartására való. Ez engedi meg, hogy a "távoli" felhasználók az yppasswd paranccsal módosíthassák jelszavaikat. a NIS adatbázisban, ypchpass paranccsal módosíthassanak egyéb adatokat ugyanott. Nem futhat a slave servereken, hiszen azokon a NIS adatbázist nem szabad módosítani. A slave-ek csak másolhatják az adatbázist. Ebből az is következik, hogy ha a master server gép meghibásodik mialatt legalább egy slave működőképes, a NIS adatbázis lekérdezése (pl. bejelentkezés) lehetséges, de jelszóváltás ez alatt nem lehetséges!

A NIS adatbázis

Mint említettük, az adatbázis képei *dbm* formájúak. A szokásos ASCII formából a rendszermenedzser a makedbm segédprogrammal alakíthat fájlokat erre a formára. A NIS adatbázisban vannak standard és lehetnek nem standard képek (map-ek). A leggyakoribb standard, alapértelmezés szerinti képek a "becenevükkel" együtt a következők:

Becenév (nick-name)	**Teljes név**
passwd	passwd.byname
group	group.byname
network	network.byaddress
hosts	hosts.bynumbers
protocols	protocols.bynumbers
services	services.byname
rpc	rpc.bynumbers
aliases	mail.aliases
ethers	ethers.byname

Tartalmukat remélhetőleg a nevük segítségével megérthetjük: pl. a passwd kép a NIS-beli passwd fájl, a group a csoport fájl stb.

A kliensek

Ha egy gép kliens (fut rajta az ypbind), akkor a rajta futó alkalmazások bizonyos kérelmek esetén nemcsak a megfelelő helyi fájlokat, vagy a helyi fájlokat egyáltalán nem keresik fel. Nem mindent felsorolva és nem teljes magyarázatot adva, néhány példán bemutatom a kereséseket.

Ha egy kliensen futó processznek (pl. a login processznek) szüksége van információkra a

/etc/passwd, vagy a

/etc/group fájlból, akkor a könyvtári rutin először a helyi fájlt nézi először. Ha a helyi fájlban + (vagy -) karaktert talál, akkor a NIS passwd, ill. group képért is folyamodik.

A /etc/host hely fájlt csak a boot-olás során keresi. Utána az RTL rutinok mindig a NIS-től kérnek információt.

Segédprogramok

Általános NIS adatbázis lekérdező program az ypcat. Argumentumaként NIS kép nevét adhatjuk meg, elegendő csak a becenevet megadni, és az ypcat kilistázza az NIS adatbázis megfelelő map-jének tartalmát (egyes rendszergazdák biztonsági okokból nem engedélyezik az ypcat használatát).

```
$ ypcat mapnév
```

NIS jelszó cserélhető az yppasswd segédprogrammal. Ha NIS tartományhoz tartozik a gépünk, ezt a parancsot kell használnunk a passwd helyett. Egyes rendszergazdák a passwd segédprogramot lecserélik az yppasswd-re, azaz csakis a NIS rendszerbeli jelszóváltást engedik meg.

```
$ yppasswd
```

Ha a NIS passwd fájlban egyéb adatmezőinket akarjuk cserélni, pl. a teljes nevünket, a bejelentkezési jegyzékünket stb., akkor az ypchpass parancsot használjuk. (Egyes rendszergazdák a chpasswd segédprogram nevén is a ypchpasswd-t futtatják számunkra).

Egyéb segédprogramok is vannak, melyek közül többet csak a rendszergazda használhat. Részletesebb információkat kaphatnak a NIS- leíró könyvekből.

Néhány megjegyzés

A NIS rendszer a hálózat használatát hatékonnyá tevő adatbázis. Mint láttuk, segítheti a névfeloldást, a felhasználók menedzselését stb. A rendszergazdáktól függ, hogy milyen információkat szolgáltat. Gyakori például, hogy a felhasználó-menedzselést a NIS-sel oldják meg, a névfeloldást azonban „kihagyják” a NIS-ből, azt a Domain Name System rendszerrel végeztetik. Az is igaz, hogy hálózati rendszerben a hatékony felhasználói menedzselésre a NIS-en kívül is találunk megoldást: X.500-as directory adatbázisban találhatók (többek között) a felhasználói adatok (a számlaszámok), és ldap protokollal érik el a directory struktúrából az autentikációhoz szükséges információkat (bejelentkezési név + jelszó + uid + stb.). A Miskolci Egyetemen a Számítóközpont gépein (gold, silver stb.) még NIS rendszer fut, az iit tartományon belül viszont 2001 szeptemberétől directory adatbázist használunk ldap protokollos lekérdezéssel.

9.6. Archiválás, backup,restore

Fájlok, fájl-rendszerek tönkremehetnek, letörölhetjük őket véletlenül, és ha nem késztettünk rendszeresen mentéseket, rengeteg munkánk veszhet el. A mentés (backup) duplikált másolata fájloknak, fájl-rendszer részleteknek, fájl-rendszereknek., amikről visszatölthetünk (restore, recovery), újra előállítva valamilyen korábbi állapotot.

A mentési eszközök lehetnek szalagok, kazetták, diszkek, CD-k. a továbbiakban a /dev/tape eszköznév a mentési médiumot jelzi, akármi is lehet az.

```
/dev/mt/tps0d4
        tps0d4nr
        tps0d4ns
        ps0d4nrns
```

Ha egy szalagra több backup fájlt is szeretnénk írni, akkor az nr tape. Általában a byte-sorrend cserélődik, ha ezt megakadályozzuk, ns tape-t használunk.

Standard eszközök : /dev/tape /dev/tapens /dev/tapenrns /dev/tapenr Linkeltek a /dev/mt/tpsXdY-ra !

A mentések fajtái lehetnek

- fájlok szerinti (file-by-file) mentések;
- diszk kép másolat (image copy).

Az előbbinél a mentés lassúbb, de könnyebb a visszaállítás (*tar* és *cpio* segédprogramok), az utóbbi gyorsabb, de több munka a visszaállítás (*dd* és *volcopy* segédprogramok).

Kategorizálhatjuk a mentéseket az archivált adatmennyiség szerint is, Így lehet

- teljes mentés (full backup) a teljes fájlrendszer mentése;
- részleges mentés (partial backup) egy adott jegyzék alatti jegyzékrendszer mentése. Gyakran módosított jegyzékrendszer archiválására kisebb mentési területet igényelve, gyorsan is menthetünk.

Készíthetünk növekményes mentéseket (incremental backup) is: ezek azon fájlok másolatai, melyek egy adott idő óta (rendszerint a megelőző mentés óta) változtak (vagy éppen csak a változások feljegyzése). Ez is gyors, kis területet igényel, de egy idő után nehéz a követése.

A rendszermenedzser, vagy a biztonsági menedzser feladata, hogy stratégiát dolgozzon ki arra, hogy

- mikor mentsenek (pl. amikor nincs nagy terhelés),
- milyen gyakran mentsenek (ez a biztonsági követelményszinttől függhet, nyilván gyakrabban mentenek egy banki, vagy katonai környezetben, mint mondjuk egy egyetemen), és nyilvánvaló, hogy nem kell az egyes fájlrendszereket azonos gyakorisággal menteni;
- milyenek legyenek a mentési technikák, a mentés-ellenőrzések (verification), nyilvántartások stb.

A mentési segédprogramokból ismerkedjünk meg eggyel, a tar segédprogrammal.

A tar (tape archive) segédprogram

Tanulmányozzuk a manuel-ban. Kapcsolói:

- c új mentést készít (create a new tape);
- v bővebb információt ad (verbose=fecsegő);
- f a követő argumentum a tape;
- t listázza a tape-n lévő neveket;
- x extract;
- u update;

Egy példa:

```
$ tar cvf /dev/tape /usr2 # uj mentest keszitettünk
$ tar tvf /dev/tape       # listazzuk a neveket a mentesrol
```

A tar egyetlen nagy fájlt készít, ebbe minden fájlt bemásol, rögzíti a fájl-struktúrát is. A mentésből visszaállíthatók az eredeti ösvénynevek. Minden lementett fájlnak 512 bájtos fejrésze van, ezt követik a fájl adatait tartalmazó blokkok (CRC ellenőrzéssel). Képes a tape határo-

kon átlépni (multi-volume). A POSIX szabványnak megfelel. (A POSIX-nak megfele a cpio is, ez is fájlonkénti mésolatkészítő.)

Gyakorlat:

A gyakorlaton formattált floppyra készítsünk mentést, olyan jegyzékből kiindulva, ami ráfér egy lemezre. Utána töröljük le az eredeti fájlokat, hogy meggyőződhessünk a visszaállításról, és állítsuk vissza a mentésről a fájl-rendszert.