Knapp Gábor – Adamis Gusztáv

Operációs rendszerek

LSI INFORMATIKAI OKTATÓKÖZPONT

Knapp Gábor – Adamis Gusztáv

Operációs rendszerek

Javított, bővített kiadás

Nyitott rendszerű képzés – Távoktatás –
oktatási segédlete
Felsőoktatási Tankönyv

LSI Informatikai Oktatóközpont
A Mikroelektronika Alkalmazásának
Kultúrájáért Alapítvány

Budapest, 2004

Bírálta: **Dr. Csopaki Gyula**
egyetemi docens

Lektorálta: **Ribényi András**
főiskolai docens

A könyv megrendelhető, illetve megvásárolható az
LSI Informatikai Oktatóközpont
1037 Budapest, Bécsi út 324.
Telefon: 436-6520
Fax: 436-6521

ISBN 963 577 219 X

Kiadó: **LSI Oktatóközpont**
Felelős vezető: **Dr. Kovács Magda**
Témafelelős: **Flier István**

Szó-Kép Kft.
Bp. Késmárk u. 24.

Tartalomjegyzék

ELŐSZÓ I

ELŐSZÓ A MÁSODIK KIADÁSHOZ I
ELŐSZÓ AZ ELSŐ KIADÁSHOZ II
A KÖNYVBEN HASZNÁLT JELEK III

1. BEVEZETÉS 1

1.1 A SZÁMÍTÓGÉPEK FELÉPÍTÉSE 2
1.1.1 HARDVER MEGKÖZELÍTÉS 2
1.1.2 FUNKCIONÁLIS MEGKÖZELÍTÉS 7
1.2 AZ OPERÁCIÓS RENDSZEREK FEJLŐDÉSE 9
1.2.1 A KEZDETEK 9
1.2.2 KÖTEGELT FELDOLGOZÁS 12
1.2.3 MULTIPROGRAMOZÁS (TÖBBFELADATOS RENDSZEREK) 15
1.2.4 INTERAKTÍV RENDSZEREK 18
1.2.5 SZEMÉLYI SZÁMÍTÓGÉPEK 20
1.2.6 FELDOLGOZÁSI MÓDOK ÖSSZEFOGLALÁSA 22
1.3 A JELEN ÉS A KÖZELJÖVŐ TENDENCIÁI 22
1.3.1 A UNIX OPERÁCIÓS RENDSZER 23
1.3.2 TÖBBPROCESSZOROS RENDSZEREK 24
1.3.3 ELOSZTOTT RENDSZEREK 26
1.3.4 OPERÁCIÓS RENDSZER MINDENHOL 27
1.4 ALAPFOGALMAK 27
1.4.1 FOLYAMATOK 28
1.4.2 ERŐFORRÁSOK 31
1.4.3 AZ OPERÁCIÓS RENDSZEREK MEGHATÁROZÁSA 33
1.4.4 AZ OPERÁCIÓS RENDSZEREK SZERKEZETE, SZOLGÁLTATÁSAI 34
1.5 VIRTUÁLIS GÉPEK 40
1.5.1 „VIRTUÁLIS" KERNEL 41
1.5.2 „VÉKONY" KLIENSEK 43
1.6 A LINUX TÖRTÉNETE 44
1.6.1 UNIX KEZDETEK 44
1.6.2 A LINUX SZÜLETÉSE 45
1.6.3 RENDSZERMAG, RENDSZER, DISZTRIBÚCIÓ 46
1.7 ÖSSZEFOGLALÁS 47
1.8 ELLENŐRZŐ KÉRDÉSEK 47

2. A FELHASZNÁLÓI FELÜLET 49

2.1 A FELHASZNÁLÓ ÉS A RENDSZERMAG 50
2.1.1 KÜLSŐ ERŐFORRÁSOK 50
2.1.2 BELSŐ ERŐFORRÁSOK 51
2.2 A PROGRAMOZÓI FELÜLET 51
2.2.1 A FORRÁSKÓD ELKÉSZÍTÉSE 52
2.2.2 FORDÍTÁS 52
2.2.3 SZERKESZTÉS 53
2.2.4 BETÖLTÉS, DINAMIKUS KÖNYVTÁRAK 54
2.3 KARAKTERES FELHASZNÁLÓI FELÜLET 55
2.3.1 PROGRAMKEZELÉS 55
2.3.2 A PARANCSÉRTELMEZŐ EGYÉB FUNKCIÓI 58
2.4 GRAFIKUS FELHASZNÁLÓI FELÜLETEK 59
2.4.1 AZ ABLAKOZÓ RENDSZER MŰKÖDÉSE 60
2.4.2 A GRAFIKUS FELÜLETEK JELLEMZŐI 61
2.5 SEGÉDPROGRAMOK, ALRENDSZEREK 64
2.6 EGY FELHASZNÁLÓBARÁT FELÜLET JELLEMZŐI 65
2.7 A LINUX FELHASZNÁLÓI FELÜLETE 66
2.8 ELLENŐRZŐ KÉRDÉSEK 68

3. ÁLLOMÁNYOK, KATALÓGUSOK 69

3.1 FÁJLNEVEK 71
3.2 FÁJLOK JELLEMZŐI 73
3.3 KÖZVETETT HIVATKOZÁSOK 75
3.4 KATALÓGUSOK (DIRECTORY) 76
3.4.1 KATALÓGUS NÉLKÜL 76
3.4.2 EGYSZINTŰ KATALÓGUS 77
3.4.3 KÉTSZINTŰ KATALÓGUS 78
3.4.4 TÖBBSZINTŰ (HIERARCHIKUS) FÁJL RENDSZER 79
3.5 HOZZÁFÉRÉSI JOGOK 80
3.5.1 HOZZÁFÉRÉSI JOGOK TÍPUSAI 81
3.5.2 JOGOK NYILVÁNTARTÁSA 82
3.6 FÁJLOK ELHELYEZÉSE 83
3.6.1 FOLYTONOS KIOSZTÁS 84
3.6.2 LÁNCOLT ELHELYEZÉS 86
3.6.3 INDEXTÁBLA ALKALMAZÁSA 87
3.7 MŰVELETEK ÁLLOMÁNYOKKAL, KATALÓGUSOKKAL 89
3.8 A FÁJLRENDSZEREK JÖVŐJE 91
3.9 ÁLLOMÁNYKEZELÉS A LINUXBAN 92
3.9.1 LOGIKAI ÁLLOMÁNYKEZELÉS 92

3.9.2 KLASSZIKUS FÁJLKEZELÉS 92
3.9.3 A VIRTUÁLIS FÁJLRENDSZER 94
3.9.4 EXT2FS - A LINUX LEMEZES FÁJL RENDSZERE 94
3.9.5 PROC - A NEMLÉTEZŐ FÁJLOK RENDSZERE 96
3.9.6 BIZTONSÁG 96
3.10 ELLENŐRZŐ KÉRDÉSEK 98

4. HÁTTÉRTÁRKEZELÉS 101

4.1 HÁTTÉRTÁROLÓK FELÉPÍTÉSE 102
4.1.1 MÁGNESSZALAGOK 103
4.1.2 MÁGNESLEMEZEK 104
4.1.3 OPTIKAI TÁROLÓK 106
4.2 ESZKÖZMEGHAJTÓK 108
4.2.1 A LEMEZ ESZKÖZMEGHAJTÓJÁNAK FELÉPÍTÉSE 109
4.2.2 LEMEZÜTEMEZÉS - A MEGHAJTÓ „FELSŐ" OLDALA 111
4.2.3 A CÍMSZÁMÍTÁS - AZ ESZKÖZMEGHAJTÓ „ALSÓ" OLDALA 114
4.2.4 MEMÓRIA TERÜLETEK KIVÁLASZTÁSA 115
4.3 AZ ADATTÁROLÁS OPTIMALIZÁLÁSÁNAK MÁS MÓDSZEREI 119
4.3.1 BLOKKMÉRET OPTIMALIZÁLÁSA 119
4.3.2 ADATTÖMÖRÍTÉS 121
4.3.3 MEGBÍZHATÓSÁG, REDUNDANCIA 123
4.4 KORSZERŰ TÁROLÓ ARCHITEKTÚRÁK 126
4.4.1 NAGY TÁROLÓRENDSZEREK JELLEMZŐI 126
4.4.2 A TÁROLÓRENDSZEREK MEGBÍZHATÓSÁGA 128
4.4.3 HIERARCHIKUS TÁROLÓ ARCHITEKTÚRÁK 130
4.5 ELLENŐRZŐ KÉRDÉSEK 132

5. ERŐFORRÁSKEZELÉS 135

5.1 AZ ERŐFORRÁS KEZELŐ 136
5.2 ERŐFORRÁS FOGLALÁSI GRÁF 137
5.3 HOLTPONT 138
5.4 KIÉHEZTETÉS 139
5.5 PÉLDA - A VACSORÁZÓ BÖLCSEK 141
5.6 HOLTPONT KEZELŐ STRATÉGIÁK 141
5.6.1 HOLTPONT MEGELŐZŐ STRATÉGIÁK 143
5.6.2 HOLTPONT FELSZÁMOLÁSA 154
5.7 KÖZÖS ERŐFORRÁSOK 157
5.8 ELLENŐRZŐ KÉRDÉSEK 165

6. FOLYAMAT- ÉS PROCESSZORKEZELÉS 167

6.1 FOLYAMATOK LÉTREHOZÁSA 167
6.2 MŰVELETEK FOLYAMATOKKAL 169
6.2.1 VÁRAKOZÁSI SOROK 169
6.2.2 KÖRNYEZETVÁLTÁS 171
6.3 A FOLYAMATOK ALAPÁLLAPOTAI 171
6.4 FELFÜGGESZTETT ÁLLAPOT 173
6.5 PROCESSZORÜTEMEZÉS 174
6.5.1 ELŐBB JÖTT, ELŐBB FUT (FCFS) 176
6.5.2 LEGRÖVIDEBB ELŐNYBEN (SJF) 178
6.5.3 KÖRBEN JÁRÓ ALGORITMUS (RR) 180
6.5.4 PRIORITÁSOS ÉS PREEMPTÍV MÓDSZEREK 182
6.6 A LINUX FOLYAMATKEZELÉSI MEGOLDÁSA 184
6.6.1 A KERNEL FOLYAMATAINAK SZINKRONIZÁLÁSA 184
6.6.2 CPU ÜTEMEZÉS 185
6.7 ELLENŐRZŐ KÉRDÉSEK 186

7. MEMÓRIAKEZELÉS 189

7.1 VALÓSÁGOS TÁRKEZELÉS 189
7.1.1 RÖGZÍTETT CÍMZÉS 190
7.1.2 ÁTHELYEZHETŐ CÍMZÉS 190
7.1.3 ÁTLAPOLÓ (OVERLAY) MÓDSZER 191
7.1.4 TÁRCSERE (SWAPPING) 192
7.1.5 ÁLLANDÓ PARTÍCIÓK 193
7.1.6 RUGALMAS PARTÍCIÓK 195
7.1.7 LAPOZÁS (PAGING) 196
7.2 VIRTUÁLIS TÁRKEZELÉS 201
7.2.1 A VIRTUÁLIS TÁRKEZELÉS ALAPJAI 203
7.2.2 LAPKIOSZTÁSI ELVEK 207
7.2.3 LAPCSERE STRATÉGIÁK 209
7.2.4 HOGYAN CSÖKKENTHETI A PROGRAMOZÓ A LAPHIBÁK SZÁMÁT? 216
7.2.5 A CÍMSZÁMÍTÁS GYORSÍTÁSA ASSZOCIATÍV TÁRRAL 217
7.3 TÁRVÉDELEM, SZEGMENTÁLÁS 220
7.3.1 A FOLYAMATOK LOGIKAI EGYSÉGEINEK VÉDELME 221
7.3.2 A FOLYAMATOK VÉDELME EGYMÁSTÓL 224
7.3.3 AZ OPERÁCIÓS RENDSZER VÉDELME - PRIORITÁSOK 225
7.4 GYORSTÁRAK (CACHE MEMÓRIÁK) 228
7.5 TÁROLÓ HIERARCHIA 229
7.6 LINUX MEMÓRIAKEZELÉS 231
7.6.1 A FIZIKAI MEMÓRIA KEZELÉSE 231

7.6.2 VIRTUÁLIS MEMÓRIA 232
7.6.3 PROGRAMOK BETÖLTÉSE 233
7.7 ELLENŐRZŐ KÉRDÉSEK 234

8. A PÁRHUZAMOS PROGRAMOZÁS ALAPJAI 235

8.1 BEVEZETÉS 235
8.2 A PRECEDENCIAGRÁF 236
8.3 FORK - JOIN UTASÍTÁSPÁR 238
8.4 PARBEGIN - PAREND UTASÍTÁSPÁR 242
8.5 ELLENŐRZŐ KÉRDÉSEK 250

FELHASZNÁLT IRODALOM 251

Előszó

Előszó a második kiadáshoz

Az operációs rendszerek fejlődése, mint megannyi más informatikai területé, továbbra is nagy ütemben folytatódik. Az előző kiadás óta eltelt egy évben csaknem minden vezető szoftvercég generációváltást hajtott végre, megjelent a NetWare 5, vége felé közeledik a Windows NT 5.0 tesztje, példátlan terjedésnek indult a Linux. A változások többnyire az elosztott rendszerek irányába mutatnak, így könyvünket is e szellemben igyekeztünk átformálni.

Alapvető változás, hogy szakítottunk az operációs rendszerek tárgyalásának hagyományos sorrendjével, és a felhasználók számára közvetlenül tapasztalható kezelői felület és állománykezelés témaköreinek tárgyalása után mélyedünk el fokozatosan az operációs rendszerek magjában zajló folyamatokban. A korszerű, többszálú operációs rendszerek megértéséhez nélkülözhetetlen témakörökkel, a párhuzamos programozás, valamint a közösen használt erőforrások tárgyalásával is kiegészült a könyv.

Az elméleti részekben tanultakat a függelék esettanulmányai segítenek elhelyezni a valóságos környezetben. A bemutatott valóságos operációs rendszerek száma reményeink szerint az aktuális igényeknek megfelelően a jövőben tovább bővül.

A könyvben megnöveltük a példák és a kérdések számát annak érdekében, hogy az önálló tanulást jobban támogassuk, a nehezebben érthető részeket több oldalról próbáltuk megvilágítani. A fontos, illetve hosszabb tanulmányozást igénylő bekezdéseket a lap szélén figyelemfelhívó jelekkel láttuk el.

Dr. Adamis Gusztáv
adamis@tmit.bme.hu

Knapp Gábor
knapp@gdf-ri.hu

Előszó az első kiadáshoz

Azt szokták mondani, hogy nem is igazi tudomány az, amely nem úgy kezdődik, hogy „már az ókori görögök is megmondták"... Be kell vallani, hogy az operációs rendszerekről az ókori görögök nem mondtak semmit. Azonban éppen az a gondolkodásmód, amelyet az antik tudósok hagytak ránk, tette lehetővé a technika hihetetlen fejlődését, az elektromos, majd elektronikus gépek, automaták, majd a számítógépek és az operációs rendszerek megjelenését.

A gondolkodásmód lényege, hogy függetlenül attól, hogy egy természeti jelenség megértésére vágyunk (analízis), vagy egy probléma megoldását lehetővé tévő szerkezetet szeretnénk készíteni (szintézis), a folyamatot részekre kell bontani, a részeket további részekre, egészen addig, amíg olyan alapegységekhez nem jutunk, amelyek már könnyedén leírhatók vagy elkészíthetők, és az egészet azután ezekből az egyszerű részekből rakhatjuk össze. Az európai történelem során ez a szemlélet rendkívül eredményesnek bizonyult, de felszínre került a legnagyobb hátrány is: ha túlságosan előtérbe kerülnek a *részletek*, néha feledésbe merül az *egész*.

Az operációs rendszerek tárgyalása során alapvetően a részekre bontó, analizáló módszert fogjuk alkalmazni, de megpróbáljuk ezt úgy tenni, hogy ne vesszen el az egész, vagy ha háttérbe is szorulna egy kis időre, végül a kis részek újra egyetlen egységgé álljanak össze.

A könyv első részének feladata a szükséges előismeretek összefoglalása, a fogalmak történeti megalapozása. A második és harmadik rész az operációs rendszerek legfontosabb funkcióit ismerteti, míg az utolsó, negyedik fejezet a legutóbbi időkben az érdeklődés középpontjába került feladatokat, a párhuzamos feldolgozás, és az elosztott rendszerek egyes kérdéseit elemzi. Minden fejezetet összefoglaló kérdések és feladatok zárnak, melyek segítségével mindenki ellenőrizheti, hogy kellőképpen elsajátította-e azokat az ismereteket, melyekre a további részek építenek.

Az apró részletek kimerítő tárgyalása nem célunk, az operációs rendszerek készítése manapság a nagy szoftver óriások kiváltsága. Egyes vélemények szerint egy operációs rendszer akkor jó, ha működése észrevehetetlen, kezelése magától értetődő. Ez így is van egészen addig, amíg a számítógép hardver konfigurációja meg nem változik, vagy amíg nem próbáljuk olyan feladatra rávenni, amire nem, vagy csak korlátozottan alkalmas. Mint számítógépes szakembereknek, meg kell

válaszolnunk, vagy legalábbis meg kell értenünk az ilyen esetekben felvetődő kérdéseket!

A könyvben használt jelek

Összefoglalás. Az előző fejezet vagy alfejezet legfontosabb fogalmai, néhány szóban

Kérdések. Áttekintő kérdések az ismertetett anyagrészhez. Segítségükkel meggyőződhetünk, hogy nem siklottunk-e el egy-egy rész felett.

Kiegészítés. A jel mellett található rész nem tartozik szorosan az anyaghoz, de segít annak jobb feldolgozásában.

Tanulmányozandó. Az így jelölt bekezdéseket nem elég csak egyszer elolvasni, megértésükhöz alaposabb tanulmányozásra van szükség.

Definíció. A meghatározásokat, törvényszerűségeket tartalmazó bekezdéseket jelöltük így.

Fogalom magyarázatok. A fontosabb fogalmak magyarázata, körülírása található a felkiáltójellel jelölt bekezdésekben.

Súlyponti rész. A megjelölt rész különös figyelmet érdemel.

1. Bevezetés

A bevezető fejezet átismétli a korábban tanultakból a számítógépek felépítésének, működésének azon kérdéseit, melyek a továbblépés szempontjából fontosak lehetnek. Az operációs rendszerek funkcionális egységeit, illetve azok feladatait kialakulásuk folyamatában tárgyaljuk, majd megismerkedünk azokkal a tendenciákkal, melyek az új operációs rendszerek fejlesztőit jelenleg foglalkoztatják. A fejezet végére az olvasó az operációs rendszerek tárgyalásához szükséges fogalmak megfelelő mélységű meghatározásának birtokába jut.

Az operációs rendszerekről előző tanulmányaink alapján annyit azért már sejthetünk, hogy valami vezérlő program félék, amelyek könnyebbé teszik a felhasználó életét azáltal, hogy a hardver vezérlésének feladatát átveszik tőle, tehát a „meztelen" hardver és a felhasználók között helyezkednek el. Az operációs rendszerek fogalmának pontos meghatározása elég nehéz feladat. Nem is biztos, hogy érdemes egyetlen nyakatekert definíciót adni, mivel alapvetően más oldalról közelíti meg a feladatokat a hardver gyártó, a rendszerprogramozó (azaz az operációs rendszer készítője), a programozó, és maga a felhasználó.

Ebben a fejezetben először összefoglaljuk a számítógépek működésének legfontosabb elemeit, majd megismerkedünk az operációs rendszerek kialakulásának történetével, azokkal az indokokkal, melyek egy-egy változást előidéztek. A funkciók ismeretében a fejezet végén visszatérünk az alapfogalmak pontosabb, legalábbis a továbblépéshez elegendő mélységű definiálásra.

1.1 A számítógépek felépítése

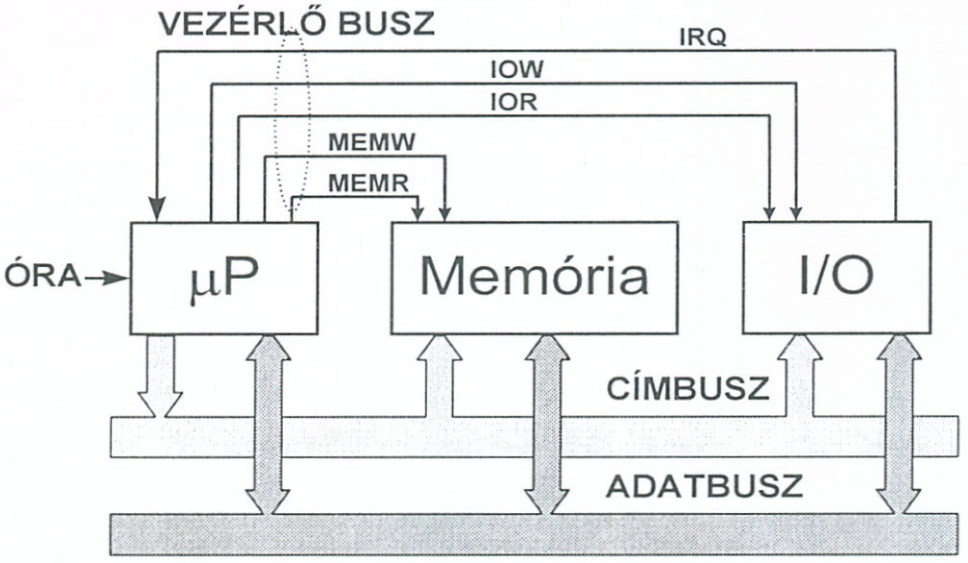
1.1.1 Hardver megközelítés

A számítógépek nagyon sokat változtak az elmúlt évtizedekben, azonban ez a változás kapacitásukat, sebességüket érintette elsősorban, működési elvükben követték a Neumann János által 1945-ben kidolgozott szabályokat. Ezek közül a legfontosabbak a következők:

Σ

1. *Tárolt program:* Az utasításokat az adatokkal azonos módon, közös nagy kapacitású memóriában, numerikus kódok formájában kell tárolni.
2. *Kettes számrendszer:* Az adatok- és program kódok ábrázolására a kettes számrendszert kell alkalmazni.
3. *Vezérlőegység:* Szükség van egy olyan vezérlőegységre, amely különbséget tud tenni utasítás és adat között, majd önműködően végrehajtatja az utasításokat.
4. *Aritmetikai-logikai egység (ALU):* A számítógép tartalmazzon olyan egységet, amely az aritmetikai műveletek mellett képes elvégezni az alapvető logikai műveleteket is.
5. *Perifériák:* Szükség van olyan ki- és bemeneti egységekre, amelyek biztosítják a kapcsolatot az ember és a számítógép között.
6. *Szekvenciális végrehajtás:* A program utasításait egymás után hajtjuk végre.

A fenti elvek alapján megvalósított számítógépek felépítése a következő:



1.1 ábra Számítógépek blokkvázlata

A **CPU** (**C**entral **P**rocessing **U**nit, központi egység, processzor) tölti be a vezérlőegység és az aritmetikai-logikai egység feladatát. Napjainkban a legtöbb számítógépben egyetlen mikroprocesszor látja el ezt a funkciót. A CPU értelmezi és hajtja végre az utasításokban kódolt aritmetikai és logikai műveleteket, vezérli az adatforgalmat a memória és a perifériák között.

A **Memória** szavanként címezhető tárolóegység, melynek rekeszei tárolják az utasításokat és az adatokat egyaránt. Az, hogy egy rekesz tartalma adat vagy utasítás, csak értelmezés kérdése, hiszen az ábrázolás módja azonos. A memóriáknak gyorsan olvashatóknak és írhatóknak kell lenniük, hiszen hozzáférési idejük alapvetően meghatározza az utasítássorozat végrehajtásának sebességét. Az ideális sebesség csak félvezető memóriák alkalmazásával érhető el, ezek viszont drágák, és kikapcsoláskor tartalmukat elvesztik. A mágneslemezek kapacitása már lényegesen jobb, a tápfeszültség megszűnése után is megőrzik adataikat, sebességük viszont több nagyságrenddel elmarad a félvezető tárakétól. A sebesség és a kapacitás közötti optimum keresése és megvalósítása a számítógépek használhatóságának egyik legfontosabb kérdése.

A **Perifériák** alapvető feladata a kapcsolattartás a külvilággal, a felhasználóval. Számtalan típus képzelhető el. A felhasználókkal való közvetlen kapcsolattartásra szolgál a billentyűzet, az egér, a monitor, nyomtató, rajzgép, szkenner stb., a hosszú távú archiválást a mágnesszalagos egységek és optikai lemezek szolgálják. A perifériák

közül az operációs rendszerek szempontjából kiemelkedő fontosságúak a mágneslemezes háttértárak, mint az adat- és programtárolás alapvető eszközei. Ugyanakkor a mai számítógépekben a perifériák nem kizárólag be- és kiviteli célokat szolgálnak, hanem bizonyos monoton, de sokszor kritikus időzítésű feladatok elvégzésével, „átvállalásával" támogatják a processzor vezérlési funkcióit. Ebbe a körbe tartoznak pl. a megszakítás- és DMA vezérlők, az intelligens diszk- és más perifériavezérlők stb.

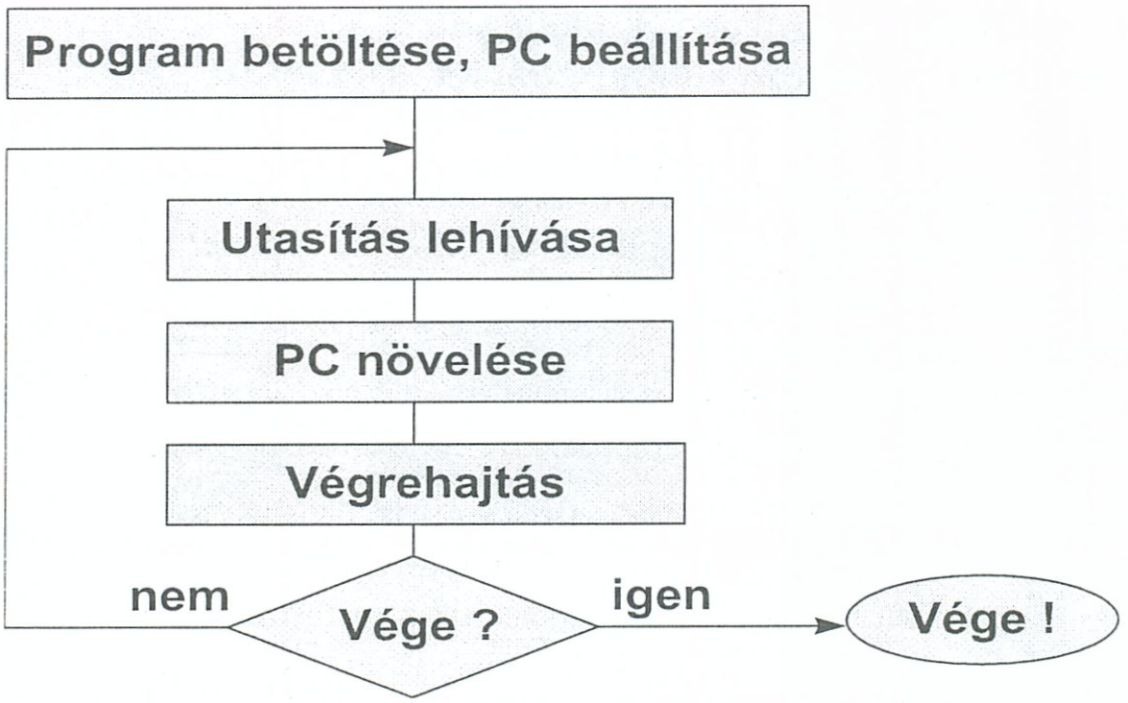
A **Busz** (vagy másik nevén Sín) biztosítja a funkcionális egységek közötti kapcsolatot. Kissé leegyszerűsítve a dolgot, felfogható egy vezetékkötegként, melyen az adatok, címek és vezérlőjelek eljuthatnak a címzettjükhöz. A buszon keresztül kommunikáló eszközöknek természetesen közös „nyelven" kell beszélniük, azaz meghatározott jelszinteket, sebességet, kódolást kell használniuk (protokoll).

A tárolóegységek elnevezései mind a magyar, mind a nemzetközi szakirodalomban meglehetősen nagy változatosságot mutatnak. Könyvünkben, elsősorban a funkciót szem előtt tartva a következő elnevezéseket használjuk:

Memória: az a tárolóegység, ahol a programozó által közvetlenül címezhető módon (például egy assembly utasítással) az aktuálisan végrehajtott utasítások és azok operandusai találhatók, attól függetlenül, hogy ez a funkció fizikailag hogyan került megvalósításra (lásd később: valós memória, virtuális memória). Használatos elnevezések még: main storage, memory, operatív tár, főtár, központi tár.

Háttértár: az a mágneses és/vagy optikai elven működő tárolóegység, mely a programok, adatok hosszabb távú megőrzésére szolgál, de amelyekről programok közvetlenül nem futtathatók. Ezek az egységek a processzorhoz perifériaként csatlakoznak.

Az utasítások és adatok tehát a memóriában találhatók, ezeken kell a CPU-nak műveleteket végeznie. Egy feladat elvégzésére szolgáló utasítások sorozata a program. Egy program futása során a processzor az utasításszámláló regiszterében található cím alapján a memóriából utasítást hív le (fetch), értelmezi (decode), majd végrehajtja azt (execute), és az utasításszámlálót a következő utasítás címére állítja. A memória és a processzor együttműködésének folyamatát írja le az úgynevezett Neumann-ciklus.



1.2 ábra Neumann-ciklus

A Neumann-ciklusban a perifériák nem szerepelnek, pedig a végső cél mindig az, hogy a külvilág egy eseményére (például emberi beavatkozás, vagy egy mérőműszer, érzékelő jele) szintén egy, a külvilág számára értelmezhető választ adjunk (üzenet kiírása vagy egy motor bekapcsolása). A külvilággal való kapcsolattartás pedig a perifériák feladata. A perifériákkal történő kommunikáció háromféleképpen történhet:

1. **Lekérdezéses átvitel** (polling): A processzor folyamatosan lekérdezi a perifériákat, hogy szükségük van-e adatátvitelre. Ha egy periféria adatátvitelt kér, a processzor lebonyolítja a szükséges műveletet, majd a következő perifériához fordul. A módszer legnagyobb hátránya, hogy a processzor az állandó kérdezgetések miatt folyamatosan foglalt, és ráadásul az idő nagy részében felesleges munkát végez, hiszen a lekérdezések döntő részében a perifériák nem igényelnek átvitelt.
2. **Megszakításos átvitel** (**I**nterrupt **ReQ**est - **IRQ**): A periféria a számára kijelölt megszakítás kérő vonalon értesíti a megszakítás vezérlőn keresztül a processzort, ha adatátvitelt igényel. A kérés elfogadása esetén a CPU egy időre félreteszi éppen végzett munkáját, kiszolgálja a perifériát, majd folytatja ott, ahol abbahagyta. A processzor ez esetben nincs teljesen kiszolgáltatva a perifériának,

viszont a programok közötti átkapcsolás, a visszatéréshez szükséges információk elmentése adminisztrációt, szervezést igényel, időt vesz el.

3. **Közvetlen memória átvitel** (**D**irect **M**emory **A**ccess - **DMA**): DMA esetén a memória és a periféria közötti átvitel a processzortól függetlenül, önálló vezérlő segítségével történik. A processzor egy pillanatig sem foglalt (ez nem mondható el azonban a buszról), mindössze az átvitel megkezdése előtt a kezdő memóriacímet, és az átadandó blokk méretét kell közölnie az autonóm vezérlővel.

A háromfajta átviteli mód közül a programozó választhat. Legkönnyebb dolga a lekérdezéssel van, legnehezebb (de egyben leghatékonyabb) a DMA programozása. Hasonlítsuk össze a bemutatott eljárásokat abból a szempontból, hogy az egyes esetekben melyik egység a kezdeményező, és melyik a végrehajtó!

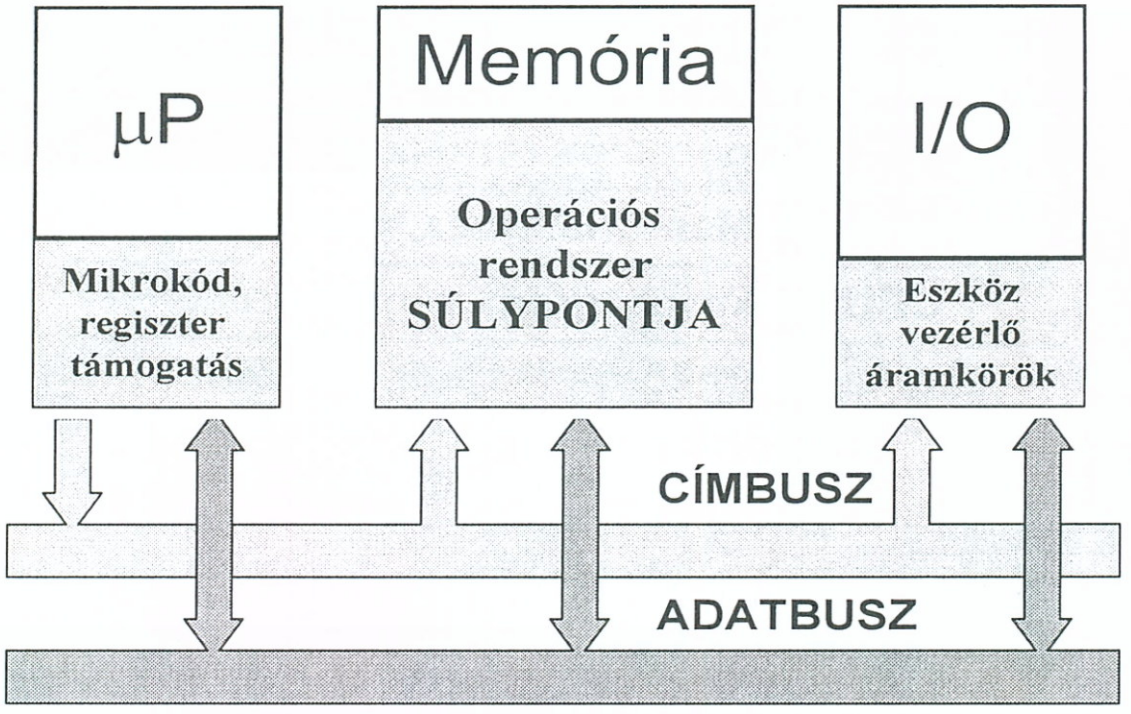
	Kezdeményező	Végrehajtó
Polling	CPU	CPU
Interrupt	Periféria	CPU
DMA	Periféria	DMA vezérlő

1.3 ábra Adatátviteli módszerek összehasonlítása

A DMA azért is játszik különösen fontos szerepet az operációs rendszerek megjelenésében és fejlődésében, mert ez volt az a technika, amely – önállóan, a processzortól függetlenül működő rendszer lévén – lehetőlehetővé tette többfeladatos, multiprogramozott rendszerek készítését.

De hol helyezkedik el az operációs rendszer? Jelenlegi meghatározásunk szerint az operációs rendszer egy program, amely a hardver kezelését hivatott megkönnyíteni. Ha program, akkor a helye a memóriában van. Az operációs rendszereknek hardverkezelő funkciójukból következően azonban vannak erősen hardver-specifikus funkciói is, amelyeket a működési sebesség javítása érdekében célszerű hardver eszközökkel megvalósítani. Az operációs rendszerek támogatására egyes feladatokat a processzor (pl. speciális regiszterek és címzési módok, mikroprogram tár)

illetve a perifériák (pl. **IDE** – **I**ntegrated **D**evice **E**lectronics - vezérlő) egyes áramkörei láthatnak el.

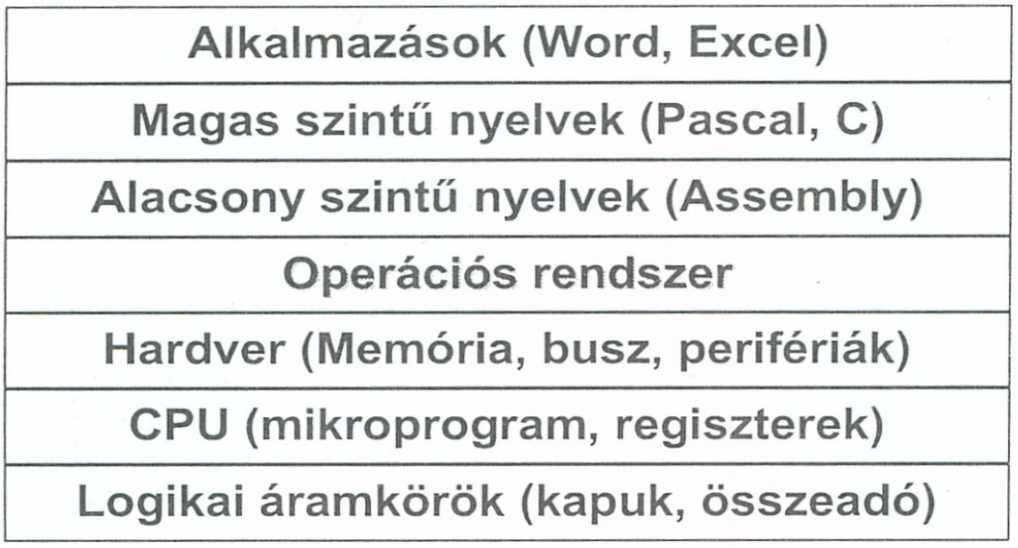


1.4 ábra Számítógép blokkvázlat, operációs rendszerrel

1.1.2 Funkcionális megközelítés

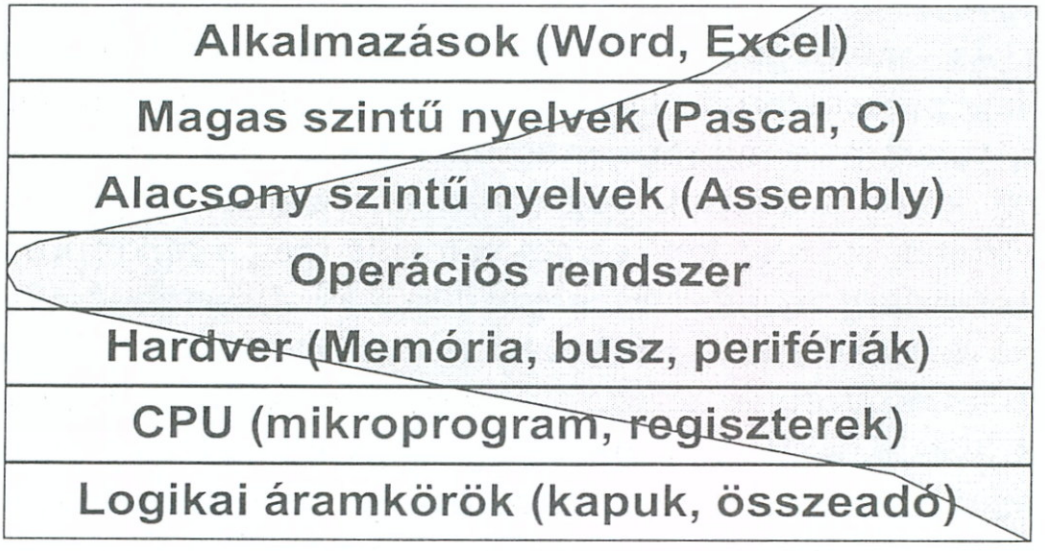
A számítógépek felépítésének, az operációs rendszer elhelyezésének egy másik megközelítésében azt vizsgáljuk, hogy mi szükséges ahhoz, hogy egy alkalmazás, például egy táblázatkezelő program futhasson.

Legelőször is szükséges egy csomó logikai áramkör, amelyekből felépíthetők a processzorok alapegységei, az ALU, a vezérlőegység és a regiszterek, valamint a mikroprogram tár. A mikroprogram tár tartalma, azaz a gépi kódú utasítások végrehajtását vezérlő program alkotja a következő szintet. Az alsó két réteget valósítja meg a mikroprocesszor. A mikroprocesszorhoz memóriát és perifériákat illesztve építhető fel a számítógép, a „lelketlen vas". A negyedik lépcső a hardverhez legközelebb álló szoftver réteg, az operációs rendszer, amely többek között a hardver vezérlésének nehézségeit hivatott elfedni. Az ötödik és hatodik szint a programozók birodalma, a gépi kódú (assembly), valamint a magas szintű nyelvek tartománya. A programozási nyelvek segítségével végre elkészülhetnek az alkalmazások, a hetedik szinten.



1.5 ábra A hétszintű logikai modell

A fenti modellben az operációs rendszernek külön szintet adtunk, tehát látszólag teljesen egyértelműen a helyére került. De így van-e valójában? Gondoljuk csak meg, hogy nem az operációs rendszer dolga-e például az az alkalmazás szintű feladat, hogy a felhasználó által begépelt parancsokat értelmezze? És nem az operációs rendszert támogatják-e a processzor egyes regiszterei vagy speciális üzemmódjai? A későbbiekben még sok szó esik ezekről a kérdésekről, azonban ha bizonyítani még nem is tudjuk, de talán érzékelhető, hogy hiábavaló volt az erőlködés. Az operációs rendszer nem zárható be egy, a logikai sémában éppen félúton lévő dobozba, csápjai a legalacsonyabbtól a legmagasabb szintig nyúlnak.



1.6 ábra Az operációs rendszer helye az alkalmazás hierarchiában

A funkcionális megközelítés előre vetíti az operációs rendszerek meghatározásának egy további nehézségét. Nem elég, hogy több megközelítési mód létezik, ráadásul bizonyos mértékig önkényesnek is kell lenni az operációs rendszerek határainak definiálásában.

1.2 Az operációs rendszerek fejlődése

1.2.1 A kezdetek

Az 1940-es években megjelent első elektronikus számítógépeknek nem volt operációs rendszerük. Az 1944-ben készített jelfogós Mark I, és az 1946-os elektroncsöves ENIAC már bináris elven működött, de a mai értelemben tulajdonképpen nem is volt igazi számítógép, programozásához huzalok százait kellett átdugdosni. Az első tárolt program elvén működő Neumann-féle számítógép 1949-ben, Angliában készült, és a vezetékek cseréje helyett kapcsolók segítségével bitenként lehetett programozni.

Ki készítette az első számítógépet? A számítógépek kialakulásának folyamata a történelem előtti időkbe, az abakuszhoz nyúlik vissza. A jelentősebb változások azonban csak a XIX. században kezdődtek Charles Babbage és Lady Lovelace munkássága nyomán. Ezek a nem lebecsülendő eredmények azonban inkább csak előzménynek tekinthetők, az igazi jogelődöt az 1930-as évek végétől megjelenő elektronikus szerkezetek között kell keresnünk.

Egészen a 60-as évek végéig a probléma teljesen egyszerűnek tűnt. Mindenki elfogadta, hogy a számítógépek kifejlesztésében nagy szerepe volt az ABC (Atanasoff-Berry Computer) alkotóinak, John Atanasoff-nak, Clifford Berry-nek, az első mikrovezérlő készítőjének, Konrad Zuse-nek, az ENIAC (Electronic Numerical Integrator And Calculator) készítőinek, John Maurchly-nak és Presper Eckert-nek, valamint a nagy matematikusnak, a magyar származású John von Neumann-nak.

A bonyodalmat a pénz okozta - mint oly sok más esetben. Amikor az amerikai hadsereg a 40-es évek közepén elhatározta, hogy ballisztikai számítások gyors és pontos végrehajtására szolgálatába állítja az ENIAC-ot, az üzleti élet is megmozdult. Sperry Rand kapcsolt először, és 1951-ben a Mauchly-Eckert párostól megvásárolta a szabadalmi jogokat az ENIAC-ra és az összes olyan működési elvre, mely annak felépítésében jelentős szerepet játszott. De mit ér egy szabadalom, ha nem származik belőle haszon? A Sperry Rand Corporation tehát ostromolni kezdte az időközben gyarapodó számítógép gyártókat, hogy fizessenek szerzői jogdíjat az alapvető elemek, elsősorban az aritmetikai logikai egység és a memória frissítő áramkörök után. Az akció természetesen nem találkozott a versenytársak osztatlan tetszésével, különösen az akkoriban már nagyobb üzleti sikereket elkönyvelő Honeywell találta sérelmesnek a dolgot.

Honeywell ügyvédei nagy kutatásba kezdtek, és hamarosan bizonyítékokat szolgáltattak arra, hogy az Iowa Állami Egyetemen dolgozó Atanasoff és Berry már 1939-ben épített működő prototípust, 1942-ben pedig már általános célú számítógéppel rendelkezett, mely tartalmazta a két kiemelt részegységet is. Tehát legalább 3 évvel megelőzték az ENIAC-ot! A kapcsolat az ABC és az ENIAC között még szorosabbnak bizonyult, ugyanis kiderült, hogy Mauchly 1941-ben konzultált Atanasoff-fal, és néhány megoldás éppen ezután a találkozás után jelent meg az ENIAC projektben.

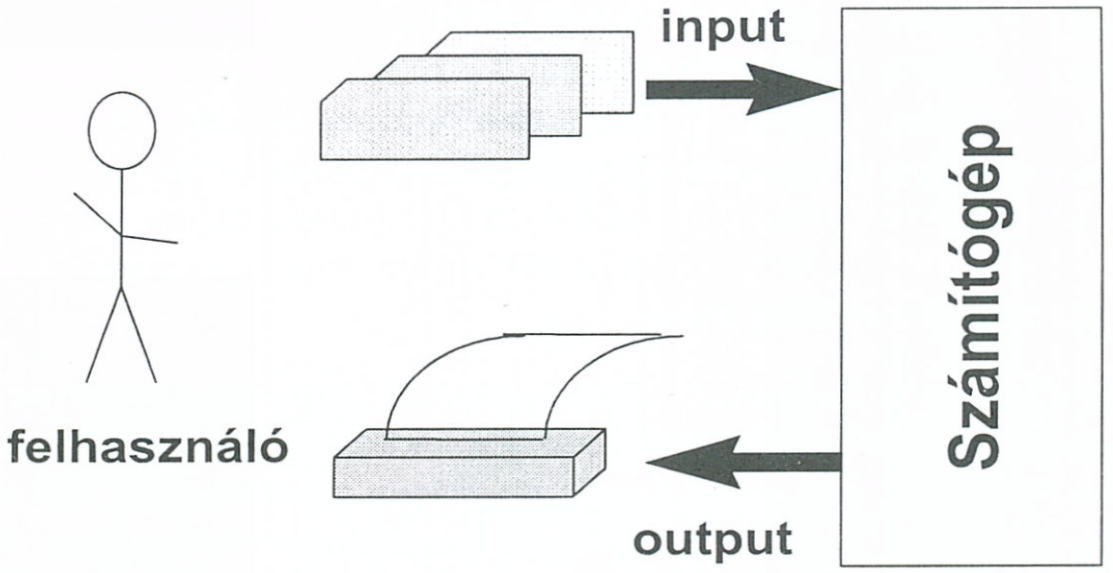
Végül 1973-ban, több évi pereskedés után a bíróság megsemmisítette az ENIAC szabadalmat, és kimondta, hogy Atanasoff munkásságának döntő szerepe van az ENIAC elkészítésében is.

John Vincent Atanasoff (1903-1995) volt tehát az elektronikus számítógép atyja? Talán helyesebb, ha megmaradunk a kezdeti bizonytalanságnál. A számítógép születésénél a körülmények legalább olyan súllyal estek latba, mint a szellemi kapacitás. Ha Atanasoff és Berry szabadalmaztatják találmányaikat és nem kénytelenek szolgálni a II. világháborúban, ha Zuse nem áll a hitleri Németország szolgálatában és nem vonják el figyelmét a konkrét hadi alkalmazások, ha az amerikai hadsereg és az üzleti élet nem az ENIAC-ra figyel fel, az általános célú, Neumann-elvet követő számítógépek hőskora kissé máshogy alakult volna, de az eredmény, a tárolt programmal működő elektronikus számítógép mindenképpen megszületik.

(J.S. Warford: Computer Science c. könyvének melléklete alapján)

A mechanikus programozási módszer nemcsak a felhasználónak volt kellemetlen, a felhasználók kényelmi szempontjai eltörpültek amellett, hogy az óriási és hihetetlenül drága gép (melynek mellesleg két meghibásodás közötti élettartama órákban volt mérhető) kihasználatlanul állt az idő legnagyobb hányadában. A hatékonyság növelése érdekében ekkor kezdtek alkalmazni kártyaolvasókat a bevitel gyorsítására, és a bináris programozás következtében óhatatlanul előforduló hibák számát nagyban csökkentették első szimbolikus nyelvek, az *assembly*-k.

Az 1950-es években a felhasználói kör szélesedni kezdett, azaz a „bitvadászok" mellett megjelentek a valóságos matematikai problémákat megoldani szándékozó igazi felhasználók. Az első magas szintű, algoritmusokra optimalizált programozási nyelv, a FORTRAN megjelenésével a számítógép által kezelhető feladatok köre is bővült. A felhasználó a számára biztosított gépidőben szemtől szembe került a számítógéppel, azt csinált vele, amit akart, pontosabban amit tudott (open shop).



1.7 ábra Számítógép használat a kezdetekben: „open shop"

Egy tipikus programfuttatás akkoriban a következőképpen zajlott:

1. A felhasználó a jó előre lefoglalt gépidő kezdetén, hóna alatt a félméteres lyukkártya csomaggal megérkezett.
2. A konzolírógépen begépelte a jelszavát és ezzel törölte a tár korábbi tartalmát.
3. Beletette a FORTRAN fordító kártyacsomagját, majd a saját fordítandó programját a kártyaolvasóba, és megnyomta a „betöltés" gombot.
4. Ha minden jól ment, kisvártatva a kártyalyukasztón új kártyák sorakoztak, sikerült a fordítás!
5. Újra a tár törlése következett és most már a lefordított program és az adatok betöltésére, majd futtatására kerülhetett sor.
6. Ha ekkor is minden kedvezően alakult, kattogni kezdett a konzol írógép, megszületett az eredmény.

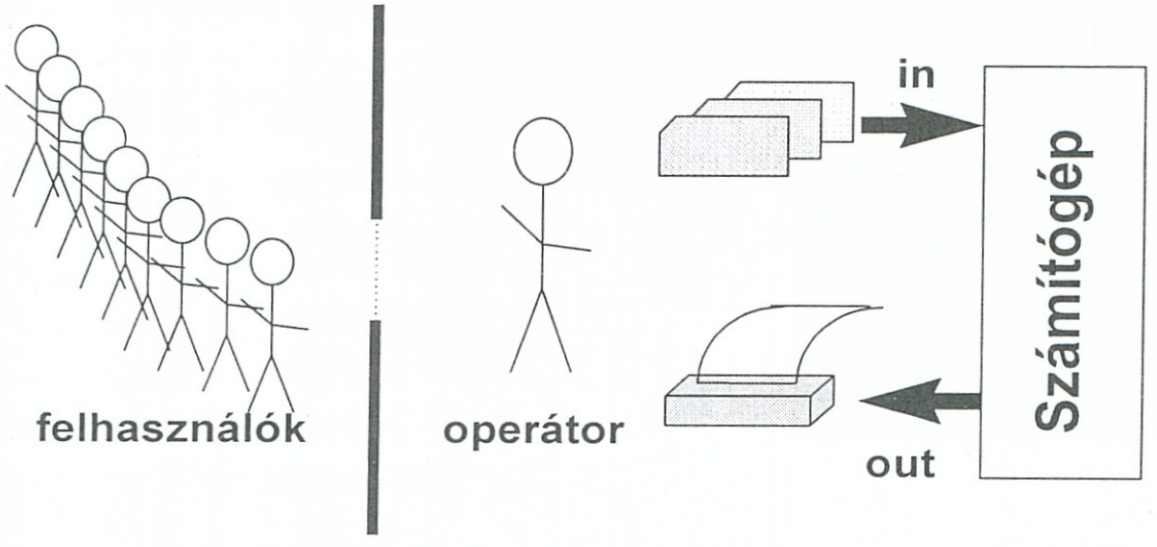
Ha jól ment, ha kedvezően alakult... És ha mégsem? Könnyen előfordulhat, hogy összecserélődött két lyukkártya, vagy véletlenül becsúszott egy nem kívánt karakter, vagy egyszerűen csak az algoritmus volt rossz! Javításra nem volt lehetőség, mert már az ajtóban toporgott a következő kliens. Újabb gépidő foglalás, újabb napok...

Ha minden azonnal sikerült, akkor sem volt ideális a helyzet, sem a felhasználó, sem a gép üzemeltetője szempontjából. A programbetöltés felgyorsult ugyan, de a gyakorlatlan programozók néha több órás küzdelmének eredménye mindössze néhány perces processzor használat volt.

1.2.2 Kötegelt feldolgozás

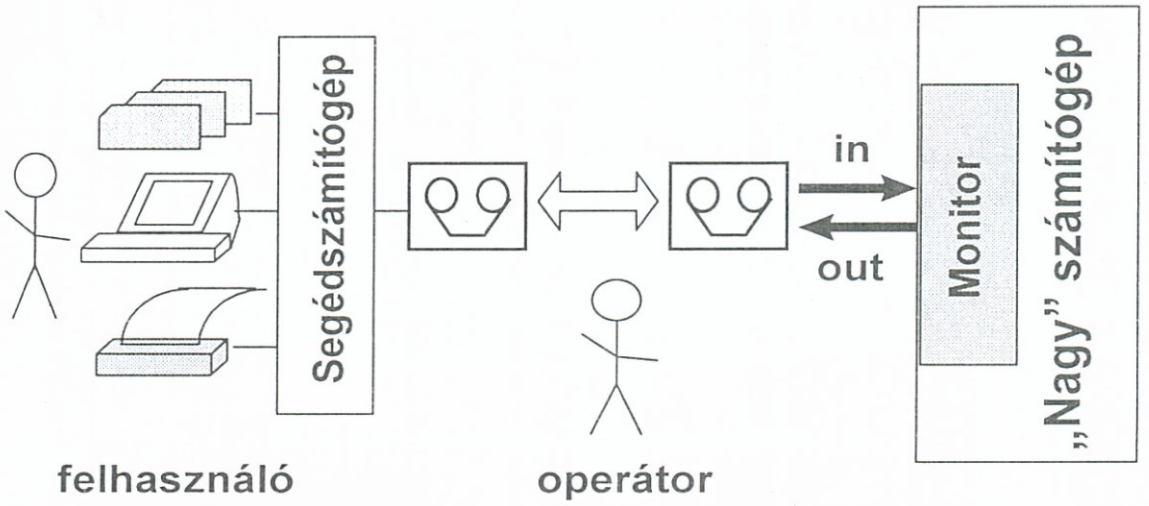
A hatékonyság növelése érdekében a kezelők profizmusát növelni kellett, a műveletek közötti átfedéseket pedig csökkenteni. Mindkét problémára megoldást jelentett egy szakképzett **operátor** alkalmazása, aki nem esett kétségbe az első hibától, a beérkezett munkákat rendszerezni tudta. Így nem kellett például naponta ötvenszer betölteni a fordító programot, pazarolva a drága gépidőt, mindössze a fordításra váró munkákat kellett szépen sorba szedni. Ez a szervezési feladat úgy volt kivitelezhető, ha az operátor több felhasználó munkáit összegyűjtötte, és a gép számára legkedvezőbb sorrendnek megfelelően futtatta őket. A felhasználók előtt így bezárult a számítógéptermek ajtaja, a számítógéppel csak az operátor közvetítésével érintkezhettek (closed shop, operator driven shop).

Az egyes feladatokat leíró kártyakötegek utasításait egymás után, szép sorban hajtotta végre a számítógép, ezért ezt a feldolgozási módot **kötegelt (batch)** feldolgozásnak nevezzük. A munka hatékonyabb lett, de a számítógépek sebessége tovább nőtt, így újra az ember lett a szűk keresztmetszet.



1.8 ábra Operátor vezette géptermek: „closed shop"

Az operátor mechanikus munkáját (és persze lassúságát és tévedéseit) kiküszöbölendő a General Motors laboratóriumában létrehozták az első operációs rendszert, illetve rendszerecskét. A számítógép vezérlését egy állandóan a memóriában tartózkodó programra, a **monitor**ra bízták, az operátor csak a perifériákat kezelte. A kártyaolvasási folyamat gyorsítását a mágnesszalagos egységek megjelenése tette lehetővé.



1.9 ábra Mágnesszalag és segédszámítógép alkalmazása

A felhasználók lyukkártyán érkező, majd később terminálon begépelt munkáit először egy kicsi és buta segédszámítógép (satellite), a nagy géptől függetlenül (off-line) összegyűjtötte egy szalagra, és ezt a szalagot vitte az operátor a nagy, okos számítógéphez, így az adatbeviteli idő töredékére csökkent.

A mágnesszalag alkalmazása még egy jelentős előnnyel járt. Bár a szalagon lévő utasítások ugyanúgy sorosan voltak elérhetők, mint a kártyák esetében, azonban a szalag visszatekerhető, sőt adott pozícióra állítható volt az operátor közreműködése nélkül. A népszerű fordító programok és könyvtárak kártyáit nem kellett tehát minden alkalommal külön betölteni, mindössze a kellő pillanatban utasítani kellett a szalagolvasó egységet, hogy álljon a megfelelő pozícióra.

A szalagra vitt munkák (job) elválasztására, valamint a végrehajtani kívánt művelet közlésére speciális utasításokat kellett adni a számítógépnek. A gépi kód szintű ASSEMBLY és a magas szintű FORTRAN mellett új nyelvcsalád jött létre, a **parancsnyelvek** (command language, command interpreter, job control language) családja. Az

előzőleg leírt fordítási folyamat kártyái a következőképpen követték egymást (mielőtt a szalagra kerültek...)

```
$ JOB NÉV       # a munka neve
$ FTN           # a FORTRAN fordító indítása
...
...             # FORTRAN programsorok
...
$ LOAD          # a lefordított program betöltése
$ RUN           # futtatás
...
...             # a program bemenő adatai
...
$ END           # a munka vége
```

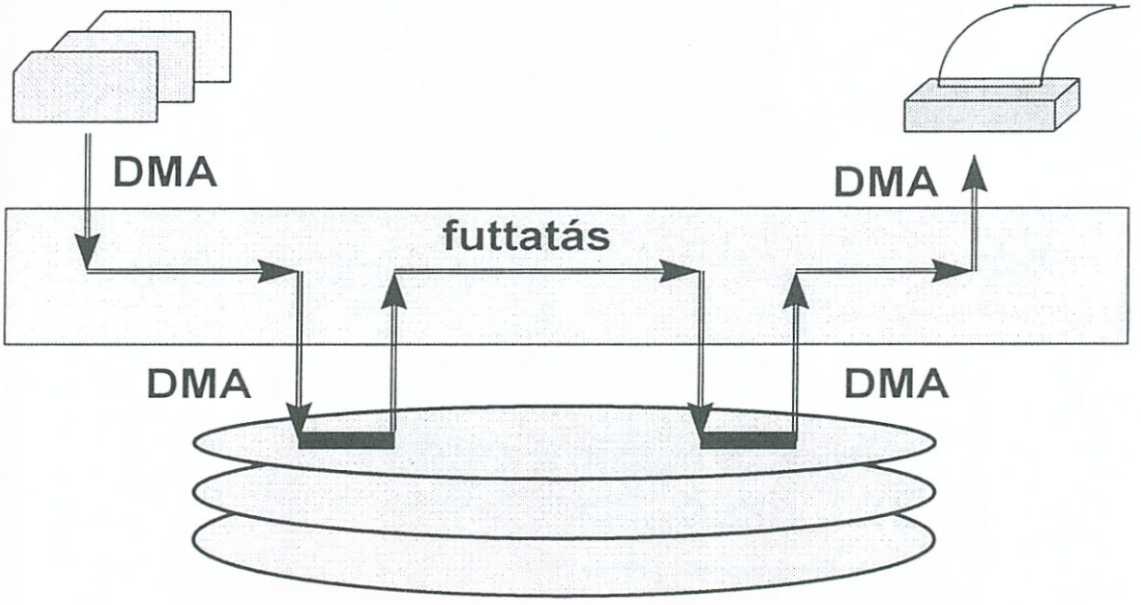
1.10 ábra A Job Control Language

Az 1960-as évekre a programok közvetlen futtatásának folyamatából már kimaradtak a felhasználók, az operátorok és a lassú, mechanikus perifériák. A processzorok gyorsulása azonban további kihívást jelentett. Az IBM 360-as család 1964-es bejelentésével már az addig gyorsnak tekintett mágnesszalagos egységek rontották a leginkább a kihasználtságot, a CPU gyakran kényszerült várakozni.

A CPU és a perifériák közötti sebességkülönbség áthidalására egy olyan gyors átmeneti tároló szolgálhat, amely a futó program által beolvasandó adatokból (és mint speciális adatból, magából a programból) annyit tárol, amennyit csak lehetséges, és a kimenő adatok gyors rögzítésére és megőrzésére is alkalmas egészen addig, amíg a lassú mechanikus kimeneti eszközök azok feldolgozására készen nem állnak. Ez az eszköz a gyors, látszólag tetszőleges elérésű mágneslemez volt. Az intelligens periféria anélkül is képes volt adatátvitelre, (pl. közvetlen memória hozzáférés, DMA útján), hogy a processzor állandóan rajta tartotta volna a szemét. A periféria a művelet befejezését megszakítás kéréssel jelezhette a központi egységnek.

Az ilyen rendszerek a lefordíthatatlan **SPOOL** rendszer nevet kapták. (A SPOOL eredetileg a **S**imultaneous **P**eripheral **O**perations **O**n **L**ine rövidítése volt, azaz párhuzamos perifériaműveletek végzésének

képességére utalt, azonban hallatán a legtöbben már a kezdetektől a mágnesszalagok csévéjére a „spulni"-ra gondoltak.).



1.11 ábra Spooling rendszer

Az operációs rendszerek történetében mérföldkőnek számított az a felismerés, hogy a korszerű mágneslemezes perifériák nemcsak a sebességen, de a munkaszervezésen is képesek javítani:

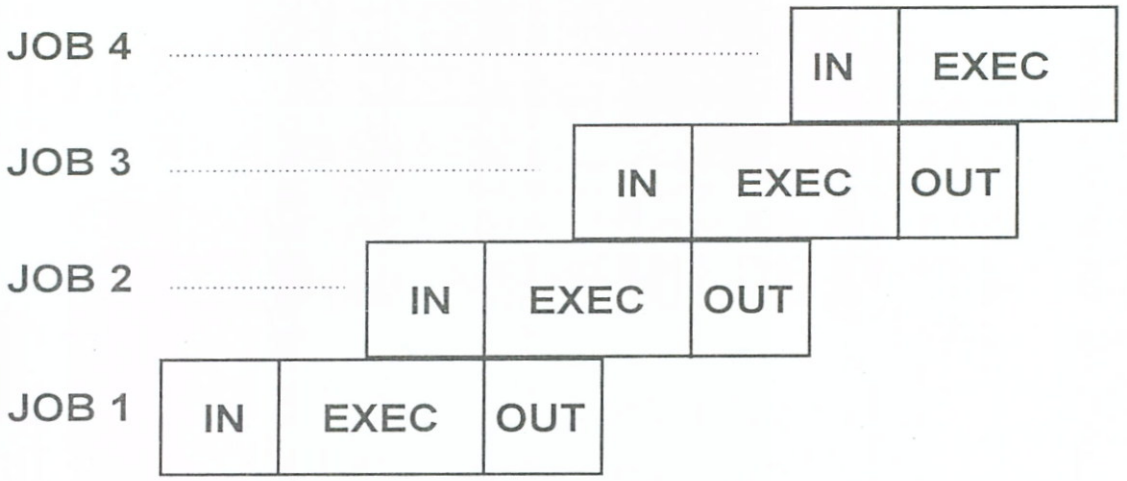
1. Az a tény, hogy a központi egység által gyorsan elérhető helyen, a mágneslemezen egyszerre több munka is volt, lehetővé tette, hogy a CPU válogathasson a futásra várakozó munkák között, különbséget tehessen azok fontossága, vagy a kedvezőbb kihasználtság szempontjából.

2. A mágneslemez alkalmazásának további előnye, hogy az adatok tárolhatók rajta újbóli felhasználás (pl. programok) vagy további feldolgozás (pl. eredmények) céljából. Ezeket az állományokat, vagy más néven fájlokat az azonosíthatóság miatt névvel kell ellátni, ha túl sok van belőlük, katalogizálni kell őket, de ez már egy későbbi fejezet tárgya...

1.2.3 Multiprogramozás (Többfeladatos rendszerek)

A spooling rendszertől már csak egy kis logikai lépés vezet ahhoz a felismeréshez, hogy ha egy rendszerben két (vagy több) önállóan működni képes eszköz van (esetünkben a processzor és a mágneslemez egység), a munkafolyamatok párhuzamosítására is lehetőség kínálkozik. A be- és kiviteli műveletek alatt „unatkozó" központi egység hatékonyságát úgy lehetett növelni, hogy a számítógép egyszerre több

munkát kapott. Tehát amíg a processzor az egyik program számítási lépéseit hajtotta végre, a memória egy másik területére betöltődhetett az újabb munka, illetve az előző munka eredményeinek lemezre mentése alatt a processzor már az új feladaton dolgozhatott.

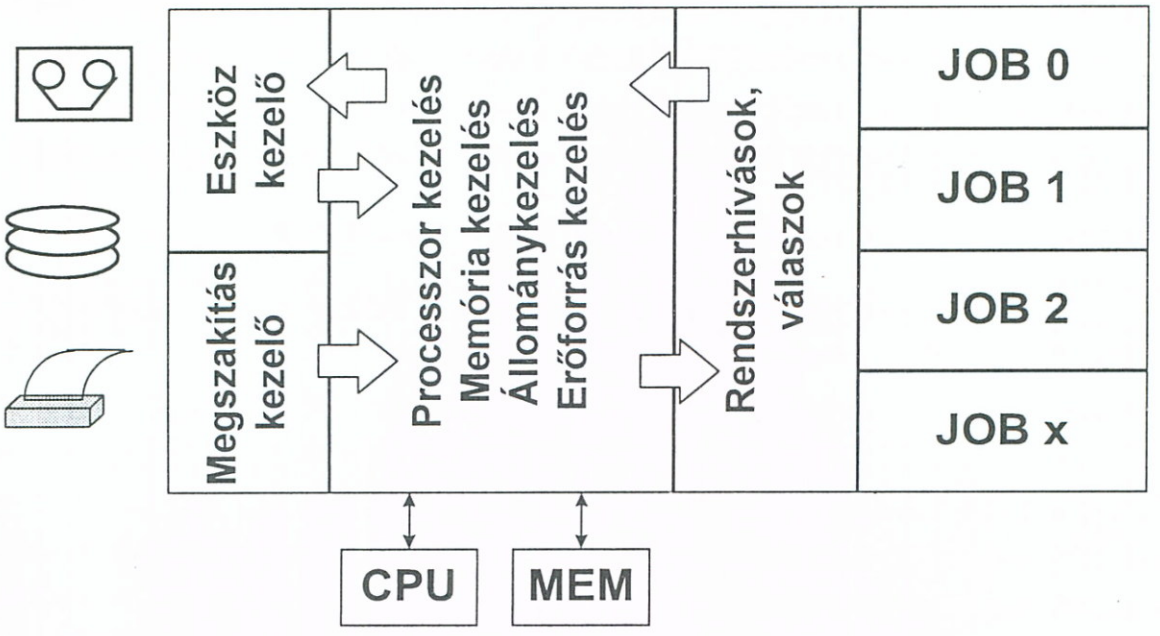


1.12 ábra Átlapolt rendszerek

Egy ilyen átlapolt rendszer azonban csak akkor működik igazán hatékonyan, ha az egyszerre végrehajtott munkák együttes periféria-idő igénye pontosan megegyezik az együttes CPU idő igénnyel. Ez a feltétel azonban a legritkább esetben teljesül. Egy munkára vagy a periféria igény a jellemző (periféria-korlátozott, peripheral bound) vagy a CPU igénye az uralkodó (CPU korlátozott, CPU bound). A kétféle program megfelelő aránya akkor biztosítható optimálisan, ha nem csak kettő, hanem több programot tartunk a „kezünk ügyében". A több program egyidejű végrehajtása mellett szól az is, hogy a munkák nemcsak a futás elején és végén igényelhetnek ki- és beviteli műveleteket, hanem a futás közben is, így ha az egyik program éppen töltődik, a másik pedig perifériára vár, a CPU újra feladat nélkül marad. A megoldás tehát kézenfekvő: több feladatot kell egyszerre kiszolgálni!

A többféle periféria megjelenése, valamint a több feladat egyidejű kezelése komoly, minőségi változtatásokat követelt meg a számítógép működését vezérlő programtól, a monitor, mint kezelői felület mellett kialakultak a vezérlőprogram által ellátandó alapfunkciók, amelyeket összefoglaló néven rendszermagnak nevezünk. A kezelői felület, a **burok (shell)** és a **mag (kernel)** együttese alkotja az **operációs rendszert (operating system)**. Az operációs rendszereknek a hardver, illetve a szoftver oldalról nézve következő feladatokat kellett ellátniuk:

1. **Eszközkezelők (Device Driver)** A felhasználói programok elől el kell fedniük a perifériák különbözőségét, egységes kezelői felületet kell biztosítani.
2. **Megszakítás kezelés (Interrupt Handling)** Alkalmas kell legyen a perifériák felől érkező kiszolgálási igények fogadására, megfelelő ellátására.
3. **Rendszerhívás, válasz (System Call, Reply)** Az operációs rendszer magjának ki kell szolgálnia a felhasználói alkalmazások (programok) erőforrások iránti igényeit úgy, hogy azok lehetőleg észre se vegyék azt, hogy nem közvetlenül használhatják a perifériákat. Erre szolgálnak a programok által kiadott rendszerhívások, melyekre a rendszermag válaszokat küldhet.
4. **Erőforrás kezelés (Resource Management)** Az egyes eszközök közös használatából származó konfliktusokat meg kell előznie, vagy bekövetkezésük esetén fel kell oldania.
5. **Processzor ütemezés (CPU Scheduling)** Az operációs rendszerek ütemező funkciójának a várakozó munkák között valamilyen stratégia alapján el kell osztani a processzor idejét, illetve vezérelnie kell a munkák közötti átkapcsolási folyamatot.
6. **Memóriakezelés (Memory Management)** Gazdálkodnia kell a memóriával, fel kell osztania azt a munkák között úgy, hogy azok egymást se zavarhassák, és az operációs rendszerben se tegyenek kárt.
7. **Állomány- és háttértárkezelés (File and Storage Management)** Rendet kell tartania a hosszabb távra megőrzendő állományok között.
8. **Felhasználói felület (User Interface)**, A parancsnyelveket feldolgozó monitor utódja, fejlettebb változata, melynek segítségével a felhasználó közölni tudja a rendszermaggal kívánságait, illetve annak állapotáról információt szerezhet.

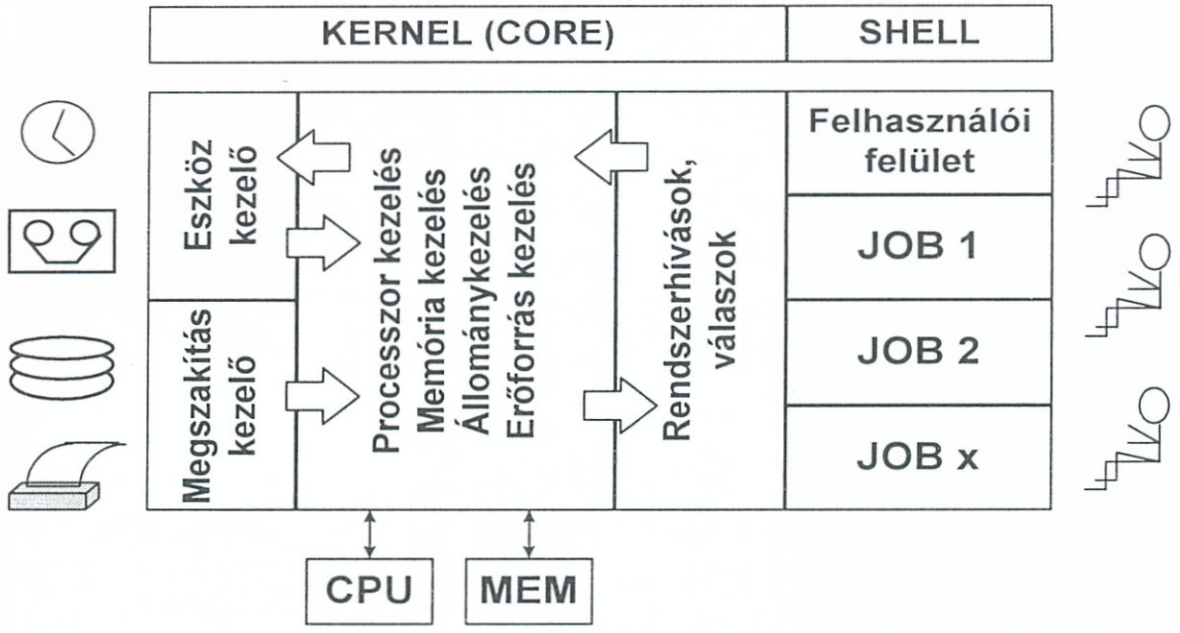


1.13 ábra Az operációs rendszerek alapfunkciói, a kernel

Az egyes munkák tehát (ebben a modellben) egymástól függetlenek, de ugyanazt a processzort, ugyanazt a memóriát és ugyanazokat a perifériákat használják. A zavartalan működés csak abban az esetben lehetséges, ha az operációs rendszer a hardver és az egyes munkafolyamatok között **áthatolhatatlan falat** alkot, minden memória- és perifériaművelet csak az ellenőrzése mellett hajtható végre.

1.2.4 Interaktív rendszerek

Az operációs rendszerek fejlődésének következő lépése az INTERAKTÍV multiprogramozás megjelenése volt. Az eddigi rendszerek a programok futtatását jól támogatták, azonban a programok fejlesztését alig. A kötegelt rendszerek korában a programozó közvetlenül nem befolyásolhatta a program futását, nem ellenőrizhette a részeredményeket, nem állíthatta le a folyamatot egyes kritikus lépéseknél. A programfejlesztés hatékonysága ugrásszerűen emelkedett, amikor a lyukkártya és a mágnesszalag helyett az interaktív terminál vált az elsődleges program- és adatbeviteli eszközzé. A számítógépek felhasználási területe is lényegileg változott meg. A számítógépek gigantikus számológépekből az információkezelés eszközeivé váltak.



1.14 ábra Interaktív rendszerek

Az interaktivitás lehetőségének megteremtése természetesen az operációs rendszert sem hagyta érintetlenül:

1. **Válaszidő.** Az operációs rendszernek emberi mércével is elfogadható válaszidővel kellett reagálnia a felhasználói beavatkozásokra, ez pedig az eddigi órák, napok helyett másodperceket jelentett.
2. **Időosztás.** A számítógépnek nemcsak akkor vannak adminisztratív feladatai, ha egy program éppen írni vagy olvasni akar, hanem időről időre foglalkoznia kell a terminálok előtt ülő türelmetlen felhasználókkal is, a perifériák között megjelenik az *óra*, amely az idő felosztását vezényli (időosztás, time sharing).
3. **Felhasználói felület**. A kötegelt rendszerekben alkalmazott parancsnyelvet ki kellett váltania egy olyan parancsértelmezőnek (command interpreter), amely lehetővé teszi, hogy a felhasználó közölhesse óhajait a számítógéppel. Nem árt, ha a felhasználó és a gép kommunikációs felülete „felhasználóbarát".
4. **Felhasználói adminisztráció**. Az operációs rendszernek a munkafolyamatokon kívül a felhasználókat is könyvelnie kell. Felvetődnek a felhasználói jogosultsággal kapcsolatos biztonsági kérdések. Igényként jelentkezik az egy gép termináljai előtt ülő felhasználók egymás közötti kommunikációja.

Az interaktív rendszerek speciális esetének tekinthető VALÓS IDEJŰ (real time) rendszerek lényegében abban különböznek az interaktív rendszerektől, hogy egy kiszolgálás kérésre adott válasz egy szigorúan meghatározott időn belül meg kell érkezzen. Egy felhasználó – szomorúan ugyan – de elvisel néhány másodperces várakozást, de ugyanez nem mondható el egy számítógép által vezérelt mérőrendszerről vagy gyártósorról.

Az interaktív rendszerek a kötegelt rendszerek *mellett* jelentek meg, nem kiszorítva azokat. A személyi számítógépek esetén ugyan ma dominál az interaktivitás, azonban nagyobbacska gépeknél a kötegelt futtatásnak (természetesen a többfeladatos kötegelt futtatásnak) mindmáig van létjogosultsága.

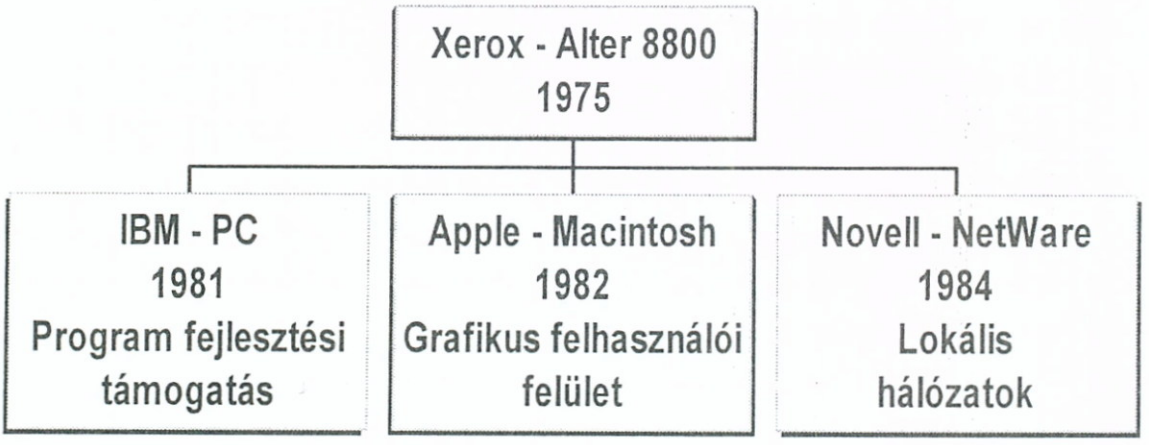
1.2.5 Személyi számítógépek

Az interaktív rendszerek már a 60-as évek közepén megjelentek. Nem egészen 20 évvel a tranzisztorhatás felfedezése után az operációs rendszerek mindmáig alapvető kérdései már felvetődtek, jórészükre megoldás is született. Eddig minden fejlesztés a drága processzoridő jobb kihasználása érdekében történt. Egy másik iparág, a mikroelektronika fejlődése, melynek legnagyobb ugrása éppen erre az időszakra esett, teljesen megváltoztatta a szemléletet. 1959-ben megjelent az első integrált áramkör és a 60-as évek végén már olyan OLCSÓ és megbízható mikroprocesszorok kerültek a piacra, melyek teljesítőképessége vetekedett nagy és drága elődeiével. Megkezdődött a kicsi, de sokak számára elérhető SZEMÉLYI SZÁMÍTÓGÉPEK (**P**ersonal **C**omputer – **PC**) széleskörű elterjedése. A felhasználóktól már nem lehetett elvárni a profizmust, barátságos, könnyen használható kezelői felületet kellett számukra kialakítani. A 70-es években a hangsúly áttevődött a *processzor* fontosságáról a *felhasználó* fontosságára.

A személyi számítógépek első példányai, amelyek az Intel 8080-as processzorára épültek, 1975-ben születtek, a Xerox laboratóriumában, Új Mexikóban. A gép neve Alter 8800 volt, kissé még nehézkesen volt programozható, de bármilyen hihetetlen, szinte minden korszerűnek mondható eszközzel el volt már látva. Az Alter 8800 egérrel vezérelt grafikus felülettel rendelkezett, biztosította a **WYSIWYG** (**W**hat **Y**ou **S**ee **I**s **W**hat **Y**ou **G**et - Ami a képernyőn, az a nyomtatón) funkciót, valamint az egyes gépek Ethernet hálózaton kommunikálhattak

egymással! Akkoriban sok kicsi cég kezdett számítógépeket gyártani, de a könyörtelen versenyben lényegében csak két cég maradt talpon, az IBM (1981-től) és az Apple (1982-től). Mindkettő *a felhasználói felületnek köszönhette eredményeit*. Az IBM a jól kezelhető, megbízható operációs rendszert támogatta erősebben, az Apple a grafikus felületeket, és a magas szintű alkalmazásokat részesítette előnyben. Az IBM PC sikere jórészt a BASIC programozási nyelv és a DOS operációs rendszer megjelenéséhez kötődik. (Mindkét szoftver döntő szerepet játszott a mindmáig piacvezető óriás, a Microsoft birodalom kialakulásában.)

A processzorok teljesítményének növekedésével a személyi számítógépek egyre többet tudtak, egyre inkább kezdtek felvetődni ugyanazok a kérdések, amelyeket már a nagygépek fejlődésében nyomon követhettünk. A 90-es évek elejétől, a Windows 3.1 megjelenésével a grafikus kezelői felület általánossá vált, sőt a Windows már több feladatot tudott egy időben kezelni. Az erőforrások megosztásának igényére a személyi számítógépek esetén a Novell adott először megoldást, kialakultak a lokális hálózatok (LAN), a hálózati funkciókat ellátó gépek, a szerverek pedig már több interaktív felhasználó több feladatát futtatták.



1.15 ábra Személyi számítógépek születése

Eljutottunk tehát ugyanoda, ahová a nagygépek jutottak, de a két fejlődési irány között van egy nagyon lényeges különbség. A nagygépes világban a többfeladatos, erőforrásokat megosztó rendszerek a hardver lehető legjobb kihasználása érdekében jöttek létre, a mikroszámítógépek világában ugyanerre az eredményre a felhasználók minél jobb kiszolgálásának célja vezetett.

1.2.6 Feldolgozási módok összefoglalása

Az operációs rendszerek általában a fenti feldolgozási módok valamelyike szerint kezelik a folyamatokat. Ugyanabban a rendszerben, egyidejűleg is jelen lehet mindhárom!

Kötegelt

- Legfontosabb a hardver kihasználása
- Minimális adminisztráció
- Az ütemezést a folyamatok maguk vezérlik (azaz környezetváltás a befejezéskor vagy olyan erőforrás igénylésénél, amely éppen nem érhető el – kb. kooperatív multitasking)

Időosztásos

- A felhasználók a fontosak (rövid válaszidő)
- Interaktív rendszerek (preemptív multitasking)
- Biztonsági és kommunikációs problémák

Valósidejű

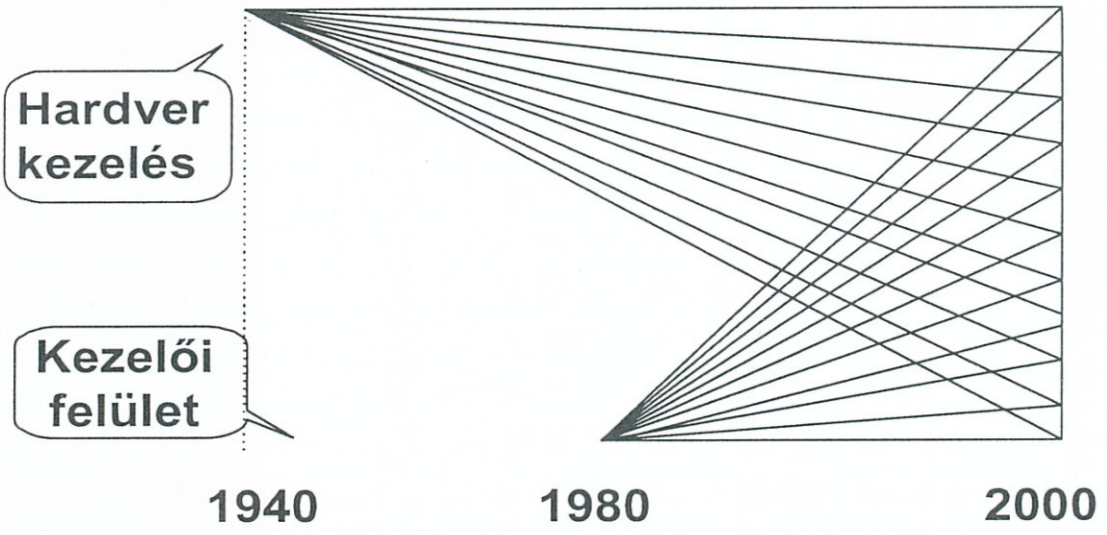
- Véges időkorláton belül kiszolgál maximális terhelés esetén is
- Folyamatvezérlés megvalósítására szolgál
- Az esemény a kezdeményező

1.3 A jelen és a közeljövő tendenciái

A számítástechnika fejlődési iránya tehát a hetvenes évek közepétől kettévált. A személyi számítógépek és a nagygépek fejlődése külön utakat járt, a két ág azonban a nyolcvanas évek végétől újra összefonódni látszik.

A XXI. század elején járunk, a processzorok teljesítménye elképzelhetetlenül nagy, az áruk elképzelhetetlenül alacsony. Miért ne lehetne egy olcsó személyi számítógépbe is többet tenni belőle? És miért kell egy nagygépnek fizikailag is nagynak lennie? Nem lehetne a korszerű alkatrészek felhasználásával kisebbé és olcsóbbá tenni? És ha már olcsóbb és elérhető, nem lehetne a kezelői felülete is hasonló? És különben is, ha már van számítógép-hálózatunk, miért ne használhatnánk a régi megszokott személyi számítógépünkről a nagygépek szolgáltatásait?

Az idő (de természetesen elsősorban a piac) megadta a válaszokat.



1.16 ábra Napjaink tendenciái

A nagygépek kistestvérei, a munkaállomások, kényelmes, általában grafikus felhasználói felületet kaptak, és a féltve őrzött számítógép termekből kikerültek az irodák, üzemek, tantermek asztalaira, azok mellé a hagyományosan barátságos személyi számítógépek mellé, melyeket jelentősen megnövekedett teljesítményük avatott felnőtté. A különbség elmosódni látszik.

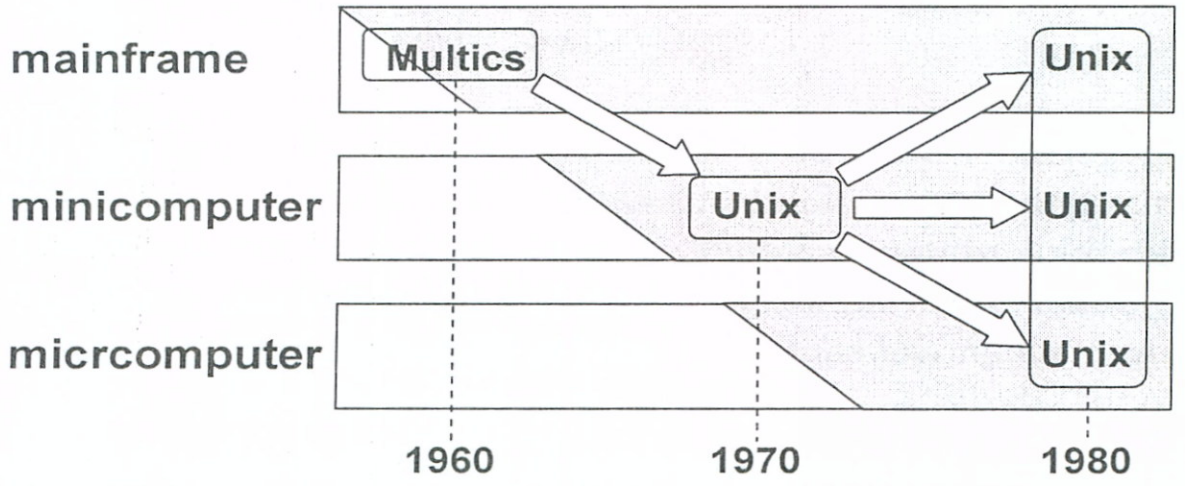
A számítógépek hálózatba szervezése is nemrégiben még elképzelhetetlen méreteket öltött, a világhálóra, az Internetre sokmillió számítógép csatlakozik, méretre, operációs rendszerre, gyártmányra való tekintet nélkül. A hálózatok, amellett, hogy lehetővé teszik a globális erőforrás megosztást, azaz minden feladatot az arra leginkább alkalmas számítógép, vagy számítógép csoport végezhet el, a biztonság területén sok problémát vetnek fel, amelyekre a jövő operációs rendszereinek megnyugtató választ kell találniuk. A külvilág felé nyitott rendszereket kell kialakítani.

Ha a számítógépek felépítése hasonló, ugyanazokat a feladatokat kell ellátniuk, ráadásul együtt is kell tudniuk működni, kézenfekvő, hogy az operációs rendszereiknek is hasonulniuk kell egymáshoz. Kell egy közös nevezőt találni.

1.3.1 A Unix operációs rendszer

Eddig méltatlanul nem esett szó arról az operációs rendszerről, amely a számítástechnika óriási változásait lényegében változatlanul követte a

hatvanas évektől a mai napig. Ez az operációs rendszer a Unix. Fennmaradását rugalmasságának köszönheti, minden konfigurációhoz, minden feladathoz jól igazítható. Külön előnye, hogy nagy kiterjedésű hálózatok fejlesztésénél is jelen volt, így az összekapcsolható rendszerek által támasztott követelmények szinte észrevétlenül épültek bele. A fentiek alapján nem túlságosan meglepő, hogy számítástechnikában, vagy helyesebben az információ technológiában kialakult két legfontosabb szabvány, a rendszerek összeköthetőségét biztosító OSI modell, illetve az alapvető rendszerspecifikációkat megadó POSIX szintén a Unix-hoz kötődik. Napjaink operációs rendszerein végigtekintve (AIX, IRIX, HP-UX, Windows NT, SUN-OS, Linux stb.) elmondható, hogy elkezdtek hasonlítani Unix-hoz, ezen keresztül pedig egymáshoz.



1.17 ábra A Unix terjedése, jelentősége

A fenti ábrából is látható, hogy a Unix az operációs rendszerek között különleges státuszt foglal el annak ellenére, hogy vannak nála többet tudó, és hatékonyabb operációs rendszerek. A Unix fejlődése szinte a kezdetektől végigkísérte az operációs rendszerek történetét, ez a rendszer volt a

- Minden mai operációsrendszer **közös őse**,
- Képes volt rugalmasan **alkalmazkodott** a mindenkori igényekhez
- Jelentős szerepe volt a **szabványosítás**ban

1.3.2 Többprocesszoros rendszerek

Ha nem is volt olyan látványos a haladás a nagyszámítógépek világában, a fejlődés azért nem állt meg. A legjelentősebb változást a többprocesszoros rendszerek megjelenése okozta. A több processzor egyidejű használata nagyobb megbízhatóságot eredményezett, és megteremtette a lehetőséget – a programágak párhuzamos végrehajtása által – a feldolgozás gyorsítására is. Ez utóbbi azonban mind a programozókat, mind az operációs rendszerek készítőit nagy kihívás elé állította, meg kellett oldaniuk a folyamatok közötti kommunikáció, illetve szinkronizálás kérdéseit. A többprocesszoros rendszerek klasszikus megvalósításában a processzorok ugyanazt a memóriát, ugyanazokat a perifériákat használják és a rendszerbuszon keresztül kommunikálnak egymással, tehát közöttük igen szoros a kapcsolat (szorosan csatolt rendszerek - tightly coupled systems). Az előnyök a következők:

1. **Megnövekedett átbocsátó képesség**. N darab processzor párhuzamos alkalmazásától azt várjuk, hogy a feladatok végrehajtási ideje N-ed részére csökken, azaz a rendszer N-szer annyi feladatot képes adott idő alatt elvégezni. Nem szabad azonban megfeledkezni a megnövekedett adminisztrációval, szinkronizációval járó többlet időről sem.
2. **Erőforrás megtakarítás.** Azoknál a feladatoknál, ahol a CPU jelenti a szűk keresztmetszetet, szükségtelen minden perifériát és memóriát többszörözni.
3. **Megbízhatóság.** Ha több funkcionális egység ugyanazt a feladatkört tölti be, egyikük meghibásodása nem okoz katasztrófát, némi teljesítmény csökkenés árán a rendszer működőképes marad. Ez a tulajdonság további lehetőséget is rejt magában. A fennmaradó rész detektálhatja a hibát, kezdeményezheti annak megszüntetését. Az ilyen rendszereket **hibatűrő rendszereknek** (fault tolerant systems) nevezzük.

A több processzoros rendszerek lehetnek **szimmetrikusak** (**SMP** - **S**ymmetric **M**ulti**P**rocessing), vagy **aszimmetrikusak**. Szimmetrikus esetben minden processzor egyenértékű, mindegyiken az operációs rendszer egy másolata fut, minden folyamatot bármelyik processzorra rá lehet bízni. Aszimmetrikus rendszereknél az egyes processzorok feladata előre rögzített, például az egyik lehet a főnök, a másik végezheti a

lebegőpontos számításokat, a harmadik kezelheti a perifériákat és így tovább.

Meg kell jegyezni, hogy a szimmetria, illetve aszimmetria kérdését sem a hardver, sem a rendszerszoftver nem döntheti el önmagában. Egy háromdimenziós képalkotásra tervezett processzor semmiképpen nem lehet egyenértékű egy általános célú mikroprocesszorral (itt a hardver felépítése akadályozza a szimmetrikus működést), és fordítva, egy hardver szempontból tökéletesen szimmetrikusan felépített rendszer sem lehet SMP, ha az operációs rendszer ezt nem úgy kezeli (a szoftver korlátoz).

1.3.3 Elosztott rendszerek

Több processzor szolgálatait úgy is igénybe lehet venni, ha azok egymással csak laza kapcsolatban vannak (lazán csatolt rendszerek - loosely coupled systems). Minden processzornak saját memóriája van, saját perifériáikkal rendelkezhet, saját operációs rendszere van, tehát egy önálló számítógép. A processzorok közötti kapcsolat valamilyen kommunikációs csatornán (telefon vonalon, lokális hálózaton) történhet. Az elosztott rendszerek tipikus példái a számítógép-hálózatok, melyekben az egyes processzorok, illetve más erőforrások egymástól földrésznyi távolságra is lehetnek. Az elosztott rendszerek alkalmazása mellett a következő érvek szólnak:

1. **Rugalmasság**. Az elosztott rendszerek komponensei lehetnek a legkülönbözőbb méretű, kiépítettségű, más-más gyártótól származó számítógépek, így minden feladathoz a legmegfelelőbb összeállítás alakítható ki.
2. **Erőforrás megosztás.** A hardver erőforrásokon kívül igen nagy szerepe van az adatbázisok, információs bázisok elosztott használatának. Például nem kell minden gépre felmásolni a moziműsort (ezzel tároló területet és időt takarítunk meg), ráadásul az adatokat a keletkezés helyén frissíthetik, így mindig aktuális marad.
3. **Sebességnövekedés.** Nagyobb, számításigényes feladatra igénybe vehetjük egy nagyszámítógép processzorait, vagy a terhelést megoszthatjuk több, kicsi számítógép között. A munkák érkezésének megfelelően a számítási kapacitás átcsoportosítható.

4. **Megbízhatóság.** Egy több lábon álló, szimmetrikus rendszerben az egyik komponens meghibásodása nem befolyásolja alapvetően az egész rendszer működését, mindössze némi sebesség csökkenéssel számolhatunk. Abban az esetben azonban, ha a meghibásodott számítógép valamilyen kritikus feladatot lát el (például kizárólag ez szolgálja ki a terminálokat) a hiba az egész rendszer működését leállíthatja.
5. **Kommunikáció.** Az összekapcsolt gépek között adatcsere történhet akár állományok továbbítása, akár elektronikus levelezés formájában.

Az említett előnyök mellett beszélni kell azonban a biztonság kérdéséről is. Az egyes állományok, hardver erőforrások jó esetben a nagy közös feladat hatékony végrehajtását szolgálják, azonban nem tisztességes szándékú felhasználók áldozatául is eshetnek. Az elosztott rendszerek témakörének egyik legfontosabb kérdése a hozzáférések szabályozása.

1.3.4 Operációs rendszer mindenhol

Ha csak röviden is, de meg kell említeni egy nagyon új irányzatot, melynek körvonalai is még csak homályosan látszanak, de hatása a személyi számítógépek elterjedéséhez mérhető lehet.

Eddig csak egyre okosabb számítógépekről volt szó olyan operációs rendszerrel, mely kényelmes kezelhetőséget biztosít, jól gazdálkodik a hardverrel és lehetővé teszi az együttműködést más hasonló szerkezetekkel. Miért ne lehetne ellátni ilyen funkciókkal egy telefont, egy kávéfőzőt, egy fénymásolót, vagy egy televíziót? Miért ne lehetne a háztartási és irodai gépeknek is operációs rendszere? És ha lehet, akkor miért ne lehetne az HASONLÓ a számítógépekéhez. A legjobb megoldás keresése folyik. A próbálkozások egyik első példája a **Microsoft At Work** operációs rendszer, melyet intelligens irodai kommunikációs rendszerek (számítógép + telefon + fax + fénymásoló + szkenner + nyomtató) számára fejlesztettek ki.

1.4 Alapfogalmak

Az operációs rendszerek történetének rövid áttekintése után nagy vonalakban kiderült, hogy mit várunk el egy operációs rendszertől, így definiálhatjuk a leírásukhoz szükséges fogalmakat, és magára az

operációs rendszerre is megkísérelhetünk többé-kevésbé szabatos meghatározást adni.

1.4.1 Folyamatok

Egy új fogalom kellett bekerüljön a számítástechnika szótárába, a FOLYAMAT fogalma. Eddig meglehetősen pongyolán használtuk a munka, feladat, program, folyamat kifejezéseket egy-egy utasítássorozat megjelölésére, azonban ezentúl pontosabban kell fogalmaznunk, éles különbséget kell tenni a program és a folyamat között:

A **program** egy algoritmust megvalósító utasítások sorozata, függetlenül attól, hogy azok magas szintű nyelven, vagy akár bináris gépi kódban vannak ábrázolva és tárolva.

A **folyamat (task, process, job)** egy éppen végrehajtás alatt lévő program. Egy program végrehajtása során több folyamatot is létrehozhat, ugyanaz a program több folyamat formájában is megjelenhet.

Folyamat
Végrehajtás alatt lévő, „élő" program

A folyamat tehát egy „életre kelt" program. Az elkövetkezőkben a program kifejezést csak olyan kódra használjuk, amely valamelyik háttértárolón várakozik arra, hogy elindítsák, az operációs rendszer felügyelete alá kerüljön, azaz folyamattá válhasson.

Egy program végrehajtásában több különböző folyamatból is részt vehet. Gondoljunk például egy levelező programra. A keretrendszer az első folyamat, mely a felhasználó kívánsága szerint létrehozhat egy szerkesztő folyamatot, majd egy levélküldő folyamatot. A létrehozó folyamat a szülő (parent process), a létrehozott a gyerekfolyamat (child process). Természetesen a szülőnek is létre kell jönnie valamikor, ezt általában maga az operációs rendszer vállalja magára, de a gyerekfolyamatok is létrehozhatnak további folyamatokat.

A másik példa legyen egy fordítóprogram. A rendszerben több felhasználó is tevékenykedik egyszerre, előfordul, hogy közülük több is programot szeretne fordítani. Ilyenkor ugyanazon programkód alapján jön létre több, egymástól független folyamat.

Multiprogramozott környezetben, ha a processzorok száma kevesebb a folyamatok számánál (ez pedig általában igaz), a folyamatok nem mindig futhatnak, olykor pihenni kényszerülnek. A tétlenség idejének leteltével, a folyamat tovább haladhat, azonban ahhoz, hogy tudja, hogy hol is tartott, az operációs rendszernek néhány alapvető információt kell róla tárolni.

A **folyamatleíró (-vezérlő) blokk** (**P**rocess **C**ontrol **B**lock – **PCB**, **T**ask **S**tate **S**egment – **TSS**) azonosítja egyértelműen a folyamatot, tartalmazza a folytatáshoz szükséges adatokat (a konkrét tartalma az adott rendszertől függ):

- a folyamat azonosítóját
- a folyamat állapotát
- a programszámláló állását
- a regiszterek tartalmát
- a folyamathoz tartozó memóriaterületek adatait (elhelyezkedését, hozzáférési szabályait stb.)
- a használt perifériák, állományok jellemzőit

A folyamatok fogalmának egy másik megközelítése a folyamatleíró blokkon alapul. Eszerint a folyamatok olyan programok, melyek rendelkeznek folyamatleíró blokkal, azaz az operációs rendszer felügyelete alá kerültek.

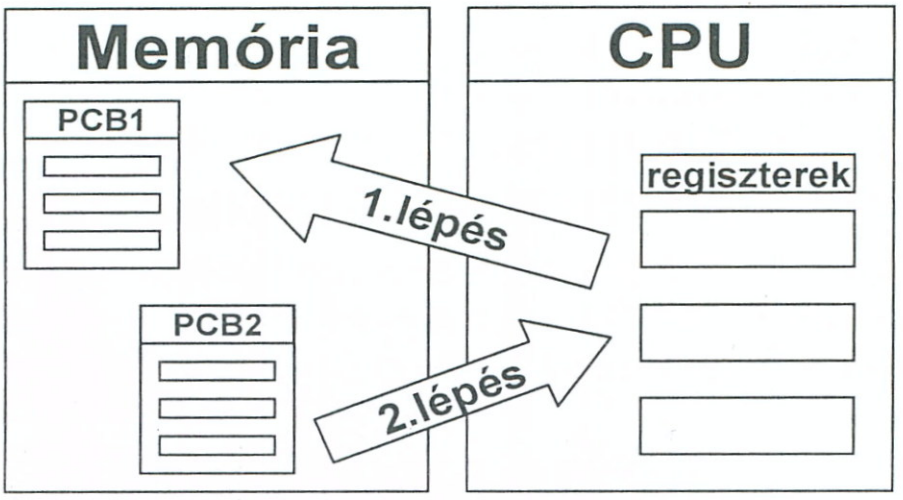
Folyamat
Olyan program, melyeknek van folyamatleíró blokkja

§

A folyamatok közötti váltás e tábla alapján történik. Minél nagyobb ez a tábla, annál lassabban. A biztonság és a sebesség szempontjai, mint mindig, itt is ellentmondóak.

Nézzünk például egy bankot, mely ügyfeleinek - természetesen megfelelő kamatjövedelem megszerzése érdekében - kölcsönöket ad. Amikor új ügyfél (folyamat) jelentkezik, a tisztviselő egy űrlapon (folyamatleíró blokk) rögzíti a bank számára fontos adatokat, a nevét, lakcímét, születési évét, és ugyanezen a lapon követik nyomon a hitel törlesztését is. A bank érdekelt abban, hogy minél gyorsabban forgassa a pénzét, azaz minél több ügyfelet szolgáljon ki, tehát az adminisztrációt, a felvett adatok számát igyekszik csökkenteni. Abban is érdekelt azonban, hogy pénzét biztonságban tudja, ehhez azonban a nyilvántartott adatok számának növelése szükséges (kezesek, azok adatai stb.).

A folyamatok közötti átkapcsolás a **környezetváltás** (context switching). Környezetváltáskor tehát a CPU egy másik folyamat utasításainak végrehajtásába kezd. Átkapcsoláskor el kell menteni minden olyan (PCB-ben tárolt, aktualizált) adatot, amely az éppen félbeszakított folyamat folytatásához szükséges.



1.18 ábra A környezetváltás lépései

A **szálak** (thread) abban hasonlítanak a folyamatokhoz, hgy ezek is egymással párhuzamosan futhatnak, de nem közvetlenül az operációs rendszer felügyelete alatt állnak, hanem egy-egy folyamaton belül működnek. A szálak nem rendelkeznek saját környezettel, hanem közösen használják a szülőfolyamat erőforrásait. Mivel nem az operációs rendszer felügyelete alatt állnak, nyilvántartásukhoz kevesebb adat is és mivel nem rendelkeznek saját környezettel, a köztük lévő átkapcsolás is sokkal gyorsabb, sokszor csak az utasításszámláló és az adatregiszterek mentését igénylik. Az adatcsere lehetőségének biztosítása érdekében a szálaknak általában valamely memória területen osztozniuk kell. E memóriaterület védelméről és az átkapcsolások lebonyolításáról a szálak szülőfolyamatjának kell gondoskodnia, ezt az operációs rendszer nem tudja segíteni, (hisz számára a folyamatok az egységek, a folyamaton belüli szálakról nem is tud.) Szálakat csak ott szerencsés használni, ahol a gyors működés kritikus, és a kisebb megbízhatóságot kellő odafigyeléssel, kimerítő teszteléssel pótolni lehet, mint például az operációs rendszer magjában, a kernelben. A szálakat gyakran nevezik „könnyű" folyamatoknak (light weight process), mert sebesség szempontjából sokkal hatékonyabbak, mint a hagyományos folyamatok.

Hasonlítsuk össze a folyamatok és szálak legfontosabb jellemzőit!

Folyamatok:

- Közvetlenül az operációs rendszer felügyelete alatt állnak
- Környezetváltáskor minden paraméterük mentendő a PCB-be
- Saját erőforrásokkal rendelkeznek, biztonságosak
- Szálakat hoz(hat)nak létre

Szálak:

- Folyamatok felügyelete alatt állnak
- Gyors átkapcsolás (váltáskor minimális adatot kell menteni – PC, regiszterek)
- Veszélyes, mert nincs saját erőforrása (a szálak közösen használják a szülő folyamat erőforrásait)
- Csak jól tesztelt, biztonságos környezetben célszerű használni!

1.4.2 Erőforrások

Erőforrásnak nevezünk minden olyan dolgot, amely szükséges lehet egy folyamat futásához. A **memóriaterület** és a **processzoridő** alapvető erőforrások, amelyek minden folyamat számára nélkülözhetetlenek, hiszen a folyamatnak lennie kell valahol, és az utasításait is végre kell hajtsa „valaki". A folyamatok jó része ezeken kívül használ a felhasználókkal való kapcsolattartás céljából **ki- és bemeneti eszközöket,** de erőforrás lehet egy **állomány** vagy egy **postafiók** is, melyen keresztül a folyamatok egymással kommunikálhatnak. A erőforrás fogalma tehát szintén azok közé a fogalmak közé tartozik, amelyeket precízen meghatározni nem a legkönnyebb, maradjunk tehát az eredeti megfogalmazásnál:

Erőforrás
Minden, ami egy folyamat végrehajtásához szükséges (memória, processzor, perifériák, állományok stb.)

§

Abból a szempontból, hogy egy-egy erőforrást a folyamatok milyen módon használnak, az erőforrások két csoportra oszthatók.

Az egyik csoportra az jellemző, hogy a folyamattól az erőforrás különösebb következmények nélkül ideiglenesen elvehető, azaz a folyamat futása megakad ugyan, de a megszakítás okának megszűnte után az eredeti állapot visszaállítható, és a végrehajtás úgy folytatódhat, mintha mi sem történt volna. Az ilyen erőforrásokat **elvehető** (megszakítható, **preemptive**) erőforrásoknak nevezzük. Tipikus elvehető erőforrás a processzor és a memória, hiszen a folyamatleíró blokk minden olyan információt tartalmaz, mely szükséges az eredeti állapot visszaállításához. Az ilyen erőforrásokkal az operációs rendszer szabadon rendelkezhet saját stratégiái szerint.

A másik erőforrás típus **nem elvehető** (nem megszakítható, **non-preemptive).** Ez a hozzáférési mód olyankor szükséges, ha egy folyamat egy erőforráson olyan műveleteket végez, amelyet nem szeretne, ha valaki megzavarna. Normális esetben egy ilyen erőforrás csak akkor szabadul fel, ha az őt használó folyamat önként mond le róla. Ilyen erőforrások lehetnek például állományok, nyomtatók, mágnesszalagos egységek, vagy bizonyos memóriában tárolt adatblokkok is. Nézzünk egy-két példát:

- Egy raktárkészletet tartalmazó adatbázis rendezése közben nem lenne szerencsés, ha közben a bolti eladók módosíthatnák az egyes tételeket.
- Hogy nézne ki egy olyan levél, melynek a nyomtató az egyik sorát egy gyászjelentésből, a másikat egy ünnepi beszédből venné?
- A folyamatleíró blokkok egy folyamat életútja alatt szintén kizárólag a rendszerfolyamatok által módosíthatók.
- Egy programszerkesztési folyamat félbeszakítása azt eredményezheti, hogy egyes változók címei már a helyükre kerültek, míg másoké még nem. Egy ilyen program lehet, hogy futóképes marad, de mit fog vajon csinálni?

Összefoglalva:

- **Elvehető** (megszakítható, preemtive) erőforrás az, amely a folyamatoktól az erőforrás vagy a folyamat károsodása nélkül ideiglenesen elvehető. Ilyen erőforrás például a *memória* (tartalma háttértárra menthető, majd visszaállítható, a *processzor* (regiszterkészlete menthető).

- **Nem elvehető** (nem megszakítható, non-preemptive) erőforrás esetén az erőforrás használat félbeszakítása az erőforrás vagy a folyamat sérülésével jár. A nem elvehető erőforrások tipikus példája a *nyomtató* (ha több folyamat összehangolatlanul használja) vagy egy *adatbázis indexállománya*.

Az, hogy egy erőforrás elvehető-e vagy sem, természetesen függ a szituációtól is. Például, ha egy folyamat nem elvehetőként foglal egy erőforrást, de valami kritikus helyzet áll elő, mely az egész rendszer működését veszélyezteti, az operációs rendszer olykor a nemesebb cél érdekében közbeavatkozik, és a kisebbik rosszat választja, az erőforrás sérülése árán is elveszi azt a folyamattól.

1.4.3 Az operációs rendszerek meghatározása

A *folyamatok* és *erőforrások* fogalmának körüljárása után megkísérelhetjük az operációs rendszerek definiálását is:

§

Operációs rendszer - Erőforrás szemlélet
A folyamatok egy olyan csoportja, amely a felhasználói folyamatok között elosztja az erőforrásokat

A fenti meghatározás alapján, a folyamatok szemszögéből nézve, az operációs rendszer feladata minden folyamat számára a szükséges erőforrások biztosítása, mégpedig igazságos módon úgy, hogy egyik folyamat se szenvedjen indokolatlan hátrányokat. Az erőforrások oldaláról nézve, az operációs rendszernek gondoskodnia kell az erőforrások minél hatékonyabb kihasználásáról.

A hatékony, megbízható és igazságos erőforrás elosztás csak úgy valósítható meg, ha az operációs rendszer minden szálat a kezében tart, a felhasználókat, illetve felhasználói folyamatokat megfosztja a hardver közvetlen kezelésének minden jogától. Ez nagyon szigorúan hangzik, de van a dolognak egy kellemesebb, optimista szemlélete is, amelyre egy újabb definíció építhető.

§

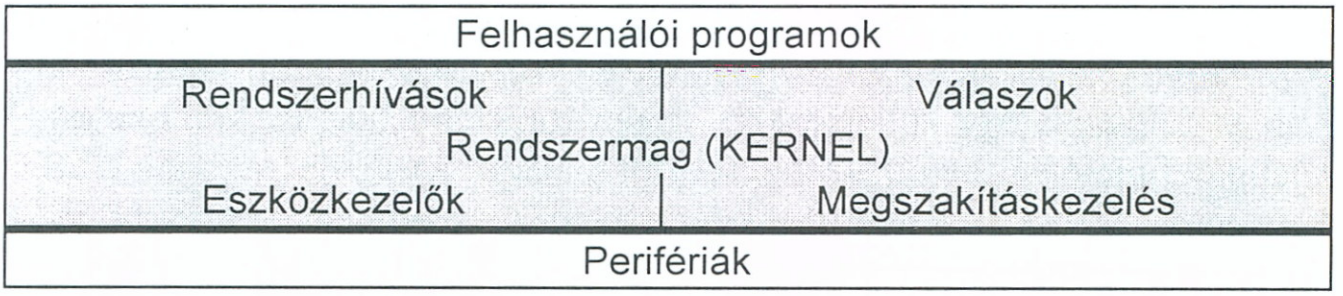
Operációs rendszer – Felhasználói szemlélet
A folyamatok egy olyan csoportja, amely megkíméli a felhasználókat a hardver kezelés nehézségeitől és kellemesebb alkalmazói környezetet biztosít

Mindkét megközelítésben kiemelkedően fontos az egyszerű kezelhetőség (felhasználó barátság, egyszerű programfejlesztés, skálázhatóság, menedzselhetőség, sok funkció, rugalmasság), a hatékonyság (sebesség, kapacitás, bővíthetőség) és a biztonság (naplózás, hozzáférés védelem, stabilitás, megbízhatóság, bizalmasság, hibatűrő képesség). Ezek a szempontok azonban ellentmondóak, egyszerre a legritkább esetben javíthatók. A biztonság növelése például majdnem mindig együtt jár a hatékonyság és/vagy a kényelem romlásával. A szolgáltatások növekedésével járó teljesítménycsökkenést ma még jól palástolja a hardver képességeinek elképesztő (másik oldalról nézve viszont pazarló) növekedése.

1.4.4 Az operációs rendszerek szerkezete, szolgáltatásai

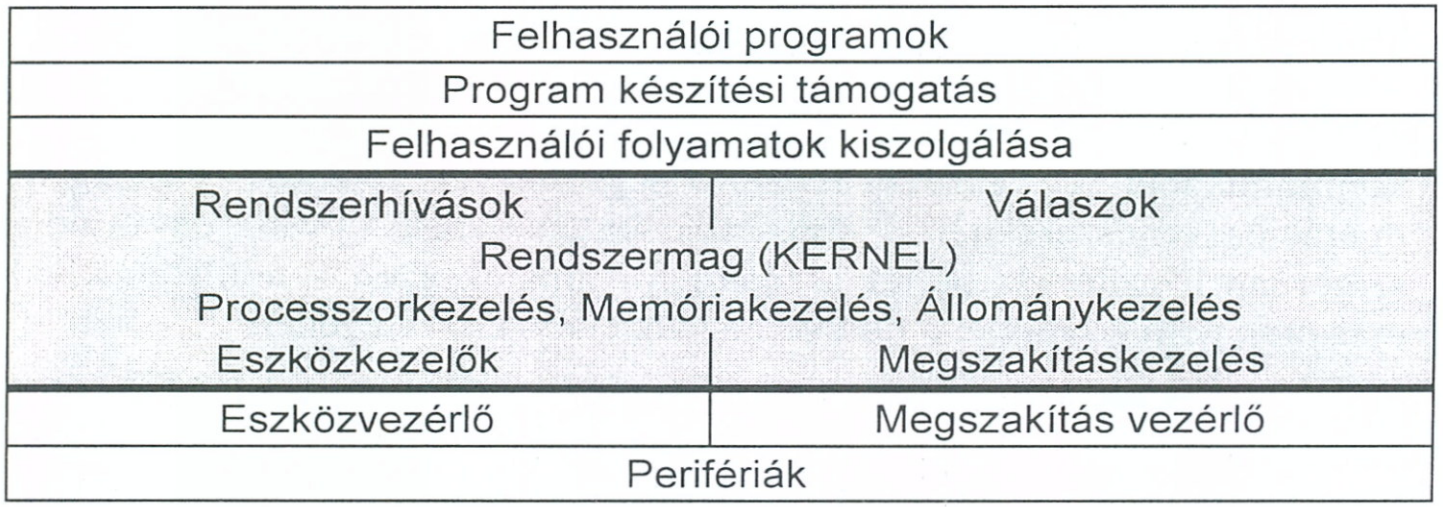
Bár az előző pontban ismertetett két definíció lényegében ugyanannak a dolognak a kétféle megfogalmazása, az utóbbi, a felhasználó felől közelítő szemlélet kissé tágabb értelmezést tesz lehetővé. Az operációs rendszerek alapfeladata, a felhasználói folyamatok (és ezen keresztül a felhasználók) igényeinek kielégítése, különböző szinteken valósulhat meg, az operációs rendszerek határai elmosódnak. Leghelyesebb, ha a bonyolult, összetett feladatrendszert megpróbáljuk részekre, rétegekre bontani, és azután ki-ki kedvére eldöntheti, hogy az egyes funkciókat az operációs rendszerek részének tekinti vagy felhasználói programnak.

Térjünk vissza az operációs rendszerek megjelenésénél alkalmazott modellre.



1.19 ábra Felhasználói folyamatok, kernel, hardver

Az a szint tehát, ami az operációs rendszert operációs rendszerré teszi a hardver és felhasználói folyamatok között helyezkedik el. Ez a rendszer magja (az ábrán sötétebb árnyalattal jelölve), a kernel, mely köré mind a hardver felé, mind a felhasználó felé rétegek sora rakódik. Egészítsük ki az egyszerű modellt, és egyre bővülő körökben vegyük sorra az egyes szintek feladatait!



1.20 ábra Az operációs rendszerek további rétegei

A réteges felépítés lényege, hogy az egyes rétegek meghatározott, jól definiált interfészeken keresztül kapcsolódnak egymáshoz, tehát egy réteg cseréje (például egy új periféria típus megjelenése) nem igényli az egész operációs rendszer átírását.

1.4.4.1 Rendszermag (KERNEL)

A rendszer magjának feladata az erőforrások elosztása és kezelése, a felhasználói folyamatok igényeinek kielégítése, adminisztrálása. A legfontosabb erőforrások, a processzor és a memória, a számtalan többi lehetséges erőforrás közül a háttértárolón lévő állományokat érdemes kiemelni. A **rendszermag** önmaga is folyamatok sokasága. Ezeket az ún. rendszerfolyamatokat feladatukon kívül az is megkülönbözteti a felhasználói folyamatoktól, hogy ezek általában a rendszer bekapcsolásakor jönnek létre, és futásuk a rendszer leállításáig tart.

A kernel hozza létre a felhasználói folyamatokat, elkészíti a folyamatleíró blokkot, memóriaterületet biztosít a végrehajtandó kódnak, illetve az

adatterületnek, majd gondoskodik – többek között – a processzor idő elosztásáról, a folyamatok sorrendjének meghatározásáról.

A rendszermag feladatai közé tartozik a felhasználói folyamatok elválasztása, védelme egymástól, illetve az illetéktelen beavatkozásoktól. A védelemnek szinte minden utasítás végrehajtásnál ellenőrzési feladatai vannak, ezért a sebességnövelés érdekében általában hardver támogatást igényel.

1.4.4.2 Rendszerhívások, válaszok

A felhasználói folyamatok és az operációs rendszer magja között a kommunikáció a **rendszerhívások** segítségével történik. Az egyes operációs rendszereknél a szabályok különböznek ugyan, de mindig szigorúan rögzítettek. A felhasználói folyamatok ezen a felületen kívül más úton nem érhetik el a hardverhez közelebb eső rétegeket.

A rendszerhívások megvalósítására az egyszerű ugróutasítástól kezdve sokféle módszer létezik. Leggyakrabban egy kitüntetett gépi kódú utasítás, egy szoftver megszakítás szolgál erre a célra, mely a vezérlést a rendszermag egy jó meghatározott pontjára adja.

A rendszerhívásokat (és ezzel az operációs rendszerek védelmét) általában a processzorok is támogatják. A processzornak ezekben a rendszerekben létezik egy korlátozottabb üzemmódja, a **felhasználói üzemmód** (user mode), melyet a felhasználói programok használnak és egy teljes utasításkészletet támogató üzemmódja a **rendszer üzemmód** (kernel mode, privilegized mode, system mode, supervisor mode), melyet csak az operációs rendszer használhat.

Ha egy felhasználói folyamat olyan utasítást ad ki, melyet a processzor felhasználói üzemmódjában nem hajtható végre (ilyenek például az hardver kezelő utasítások), az utasítás „csapdába" esik (trap), és a vezérlés az operációs rendszerhez kerül. Az operációs rendszer ilyen esetben vagy hibajelzést ad, vagy megpróbálja „kitalálni", mit szeretett volna a programozó elérni, és a maga eszközeivel végrehajtja a feladatot. (Ilyen működés fordul elő például gyakran, amikor öreg DOS-os programokat Windows környezetben futtatunk.)

A rendszerhívások kiszolgálása a következőképpen történik:

1. A felhasználói folyamat legfontosabb paraméterei elmentődnek.

2. A kernel megfelelő folyamatára kerül a vezérlés.
3. A paraméterek átadásra kerülnek a vermen (stack), a regisztereken vagy valamely közösen használt memóriaterületen keresztül.
4. A processzor *rendszermódba* kapcsolódik át. (Ezt gyakran már a rendszerhívó utasítás maga megteszi).
5. Elindul a megfelelő rendszerfolyamat, végrehajtja a kívánt feladatot.
6. A válaszok vagy hibakódok valamely paraméterátadásra szolgáló területre kerülnek.
7. A processzor visszatér *felhasználói módba*.
8. A megszakított folyamat visszakapja a vezérlést.

1.4.4.3 Eszközkezelők, megszakításkezelés

A perifériák felől az operációs rendszer magját az **eszközkezelőkön** (device driver), illetve a **megszakítás** (interrupt) rendszeren keresztül lehet megközelíteni.

Érdemes ennél a pontnál kicsit elidőzni, visszaemlékezni a fejezet első oldalainak egyik ábrájára. Arra a gondolatra egyrészt, hogy az operációs rendszer hatékonysága jelentősen növekedhet, ha megfelelő hardver támogatást kap, másrészt arra, hogy az operációs rendszer határai meglehetősen elmosódottak. Így van ez a hardverhez közeli rétegekben is. Az *eszközkezelő* funkcióinak egy részét szinte mindig a perifériára integrált *eszközvezérlő* végzi el (IDE vezérlő, SCSI vezérlő), a *megszakításkezelés*ben pedig mindig részt vesz egy előfeldolgozást végző áramkör, a *megszakítás vezérlő* (IBM kompatíbilis számítógépeknél az i8259, vagy ezzel azonos funkciójú áramkör). A továbbiakban főleg a funkciót, és nem annak konkrét megvalósítását tárgyaljuk, így megmaradunk az „eszközkezelő", illetve „megszakítás kezelő" kifejezéseknél.

A külön **eszközkezelők** létjogosultságát az indokolja, hogy a folyamatok kezelése a kernelre olyan feladatokat ró, hogy annak nincs ideje a különböző perifériák speciális tulajdonságaira figyelni, azokat egységes felületen keresztül szeretné kezelni. Ez a szemlélet még az operációs rendszerek hőskorából származik, amikor a kártyaolvasó helyét úgy kellett átvennie a mágnesszalagnak, hogy közben a programkódnak ne kelljen változnia, az egy általános perifériára hivatkozhasson. Az eszközkezelők alkalmazása mellett szóló másik érv az, hogy az operációs rendszerek és a perifériák általában nem együtt fejlődnek, ugyanazon operációs rendszernek egyre újabb, fejlettebb perifériákkal kell együttműködnie, illetve ugyanazon perifériákat több operációs rendszer is

kezelheti. Az eszközkezelők készítésének feladata megoszlik a hardver gyártók és a rendszerprogramozók között. Nem ritka, hogy egy periféria a hozzá tartozó eszközkezelő egy részét hardver szinten, ROM-ban tartalmazza.

A perifériák az operációs rendszer figyelmét **megszakítás kéréssel** (interrupt request) hívják fel magukra. Ilyen kérésre kerülhet sor például egy hardver elem meghibásodásakor, egy adatátvitel megkezdésekor vagy befejezésekor. A megszakításkezelés folyamata nagyon hasonló a rendszerhívások kezeléséhez, azonban a kiszolgáló rutin kiválasztása némileg eltérő.

A processzorok általában egy megszakítás vezetékkel rendelkeznek, ezen keresztül jelez az összes periféria. A megszakítás kérés kezelésének első feladata a forrás meghatározása. Legegyszerűbben ez történhet a perifériák végigkérdezésével (polling). Sokkal hatékonyabb azonban a PC-kben is alkalmazott vektoros megszakításkezelés, de ez hardver támogatást igényel. A megszakítás vezérlő áramkör ez esetben több bemenettel rendelkezik, és minden bemenetéhez tartozik egy regiszter, vagy memóriaszó, mely tartalmazza a kiszolgáló rutin címét. A címeket tartalmazó vektort az operációs rendszer tölti fel betöltődéskor, de bizonyos esetekben futás közben is változtathatja.

A megszakítások tehát olyan eseményeket jeleznek, amelyre az operációs rendszernek lehetőség szerint azonnal kell reagálnia. A megszakításoknak eredetük szerint több típusát különböztetjük meg:

- **Hardver megszakítás** (Interrupt Request – **IRQ**)
  Egy periféria jelezheti így egy régen várt adat megérkezését, de megszakítást okoz a rendszer órája is. A hardver megszakítások speciális esete a **nem maszkolható megszakítás** (Non Maskable Interrupt – **NMI**), amely súlyos hardver hiba, például a memória hibája, vagy a tápfeszültség kimaradás esetén keletkezik. Nevéből is látszik, hogy ezt a típusú megszakításkérést mindig el kell fogadni.
- **Kivétel (Exception)**
  A kivételeket maga a processzor generálja, ha valamilyen hibát, például nullával való osztást kellene végeznie, vagy a címszámításnál tapasztal valamilyen komoly hibát. Ilyen kivétel a már bemutatott **csapda (Trap)** is, amely akkor keletkezik, ha egy felhasználói

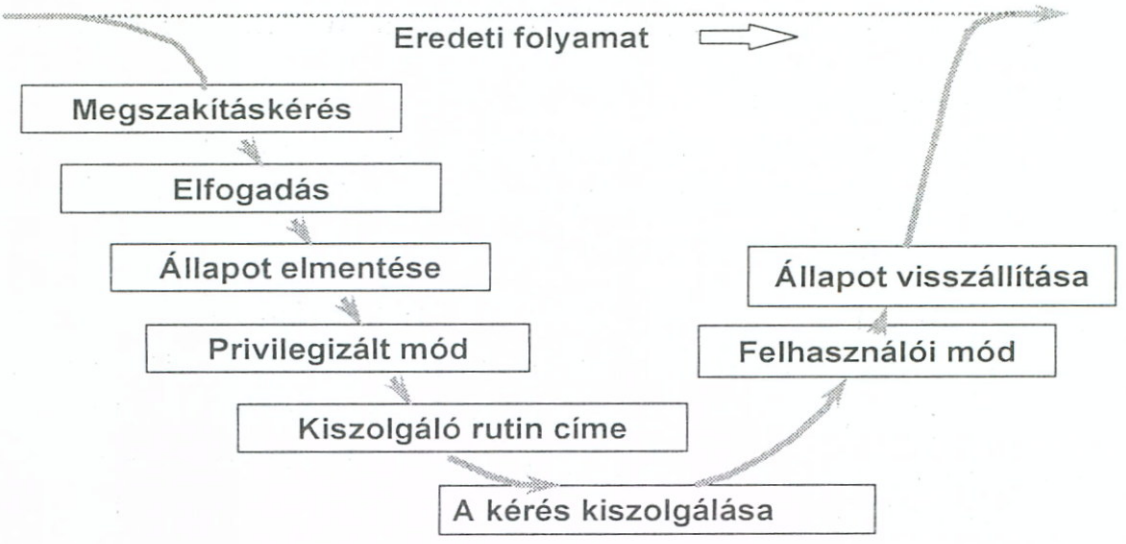
folyamat közvetlenül az hardverhez fordul, azaz olyan utasítást próbál végrehajtani, amihez nem lenne joga (önálló hardver kezelés).

- **Szoftver megszakítás**
  Olyan utasítás, amely végrehajtása a „hagyományos" megszakítások kezelésére hasonlít. Általában segítségükkel vehetők igénybe a rendszerhívások.

A megszakításokhoz legtöbb esetben prioritási szintek rendelhetők. Magasabb prioritású kérések megszakíthatják az alacsonyabb szintű kérések kiszolgálását. A megszakítások általában letilthatók, de ezzel az operációs rendszerek csak indokolt esetben élnek, hiszen fontos adatokat veszthetnek el ezáltal.

A megszakítások kezelése eredetüktől függetlenül, lényegében azonos módon történik. A kezelőprogramok általában rövidek, kevés erőforrást használnak. A megszakításkezelés forgatókönyve többnyire a következő:

1. Megszakításkérés érkezik.
2. A processzor befejezi az éppen végzett műveletet, majd, ha éppen nincs letiltva az adott szintű megszakítás, elfogadja a kérést, ellenkező esetben várakoztatja.
3. A processzor elmenti a futó folyamat állapotát a PCB-be.
4. A CPU privilegizált (kernel) üzemmódba kerül, és letiltódik az összes olyan megszakítás, melynek prioritása kisebb vagy egyenlő az érkezett megszakításéval.
5. A központi egység megállapítja a megszakításkérés helyét, és a megszakítási vektortáblából kikeresi a megfelelő kiszolgáló rutin címét.
6. A kiszolgáló rutin fut.
7. A CPU visszatér felhasználói (user) üzemmódba, és engedélyezi a letiltott megszakítási szinteket.
8. A processzor visszaállítja a megszakított folyamat állapotát, ezzel visszaadva a vezérlést.

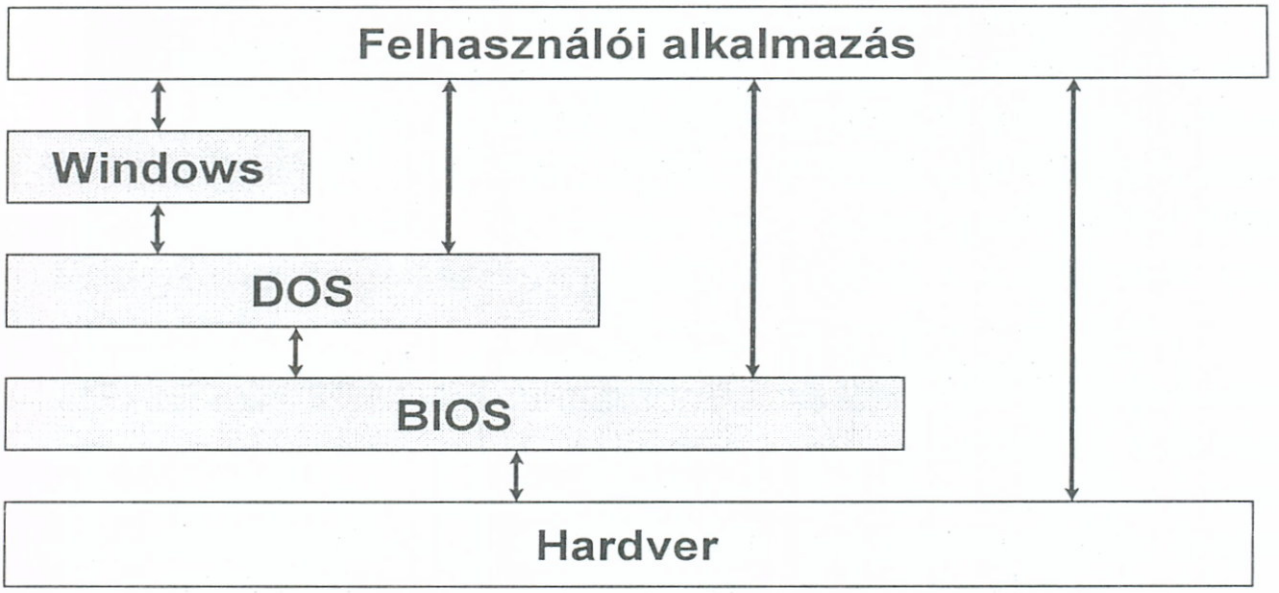


1.21 ábra Megszakítások kezelése

A megszakítás okának jellegétől függően természetesen mindig más és más történhet. A megszakítást kezelő rutin igen közel áll az operációs rendszer magjához, így annak adatait is átírhatja, ezzel befolyásolva például a folyamatok végrehajtási sorrendjét. Nem szabad elfelejteni, hogy a megszakítás kezelő rutin privilegizált módban fut, így kényekedve szerint bármit megtehet!

1.5 Virtuális gépek

Az operációs rendszer, amely a hardver kezelését magára vállalja, olyan felületet biztosít, amely úgy viselkedik, mintha egy látszólagos (virtuális) számítógépen futnának programjaink. De hol érdemes meghúzni az operációs rendszer magjának felső, a felhasználói folyamatok felőli határvonalát? Ahhoz, hogy a kérdést egy kicsit kimerítőbben megválaszolhassuk, nézzük meg a jó öreg DOS operációs rendszert a Windows 3.1-gyel kiegészítve, amint éppen egy Windows alkalmazást futtat.



1.22 ábra DOS + Windows alkalmazás hardver kezelése

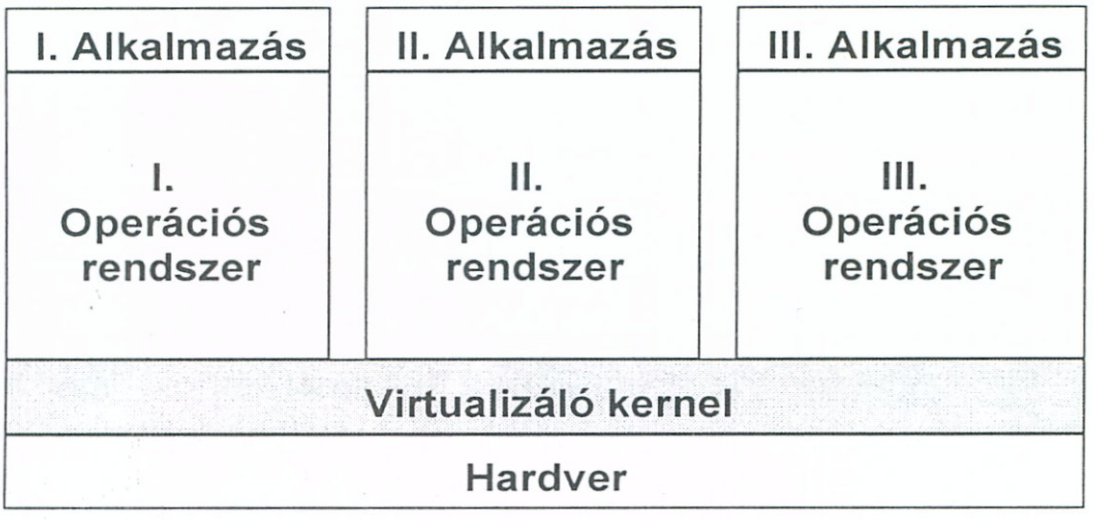
Az ábra bal oldalán kis jóindulattal ráismerhetünk az operációs rendszerek rétegszerkezetére. A felhasználói alkalmazás a Windows rendszerhívásait használja, a Windows a DOS-ra támaszkodik, ez utóbbi viszont a BIOS-on keresztül működteti a hardvert. A jobb oldalon viszont szörnyűséget láthatunk! Az alkalmazás közvetlenül a hardverrel van kapcsolatban! A két szélsőség között az összes lehetséges variáció elképzelhető. Ez ebben az esetben nem olyan nagyon veszélyes, hiszen egy egyfelhasználós rendszerről van szó, ahol az alkalmazásokból ugyan több is lehet, de egy-egy alkalmazás futása alatt korlátlan úr, csak akkor adja át a vezérlést egy másiknak, ha úgy látja jónak (cooperative multitasking). A Windows függvények használata kényelmes, könnyű így szép programot készíteni, de ezért a kényelemért a futási sebesség csökkenése az ár. A hardver közvetlen programozása gyors, de nagyon keserves.

Többfelhasználós rendszereknél, multiprogramozott környezetben az operációs rendszerre, mint áthatolhatatlan falra szükség van, de hogy annak hol legyen a felhasználói programok felőli határa, az elhatározás kérdése.

1.5.1 „Virtuális" kernel

Az egyik szélsőséges megoldást az IBM VM operációs rendszerének példáján mutatjuk be. A virtualizáló kernel nem valósít meg nagyobb bonyolultságú rendszerhívásokat, felhasználói interfészén úgy viselkedik,

mintha maga volna a hardver, azzal a nem lényegtelen különbséggel, hogy több folyamat is működhet rajta párhuzamosan. Éppúgy, ahogy a kernel sem valóságos a szó klasszikus értelmében, a felhasználói felület sem az. A virtuális gépen futó folyamatok egymástól gyakorlatilag teljesen függetlenek, mindössze kissé lassabban futnak, mintha valóban egyedül lennének.



1.23 ábra Az IBM VM vázlatos felépítése

Mi az előnye egy ilyen rendszernek? Mint az az ábrából is látszik, egyszerre futhat rajta több „igazi" operációs rendszer! Párhuzamosan fejleszthető az új operációs rendszer, míg a régi alkalmazásokat futtat, melyek valóságos adatokon, élesben dolgoznak. Ebből fakadó további előny, hogy régebbi rendszerekre írt programok számára binárisan kompatíbilis, megfelelő környezet biztosítható. Gyakori igény napjainkban az alapvetően Intel processzorokra íródott DOS alkalmazások futtatása olyan processzorokon (PowerPC, Alpha), melyek már nyomokban sem emlékeztetnek a 8086-ra.

Annak illusztrálására, hogy ilyen előnyök mellett miért nem készül minden operációs rendszer a klasszikus virtuális gépek mintájára, két példa szolgálhat:

1. A virtualizáló kernelnek a processzor üzemmódjainak tekintetében is követnie kell a valódi viszonyokat. A folyamatok, azaz a különböző operációs rendszerek, viszont csak felhasználói módban futhatnak. Ha tehát a folyamat privilegizált módba kapcsol, a virtualizáló kernelnek virtuális privilegizált üzemmódot kell szimulálnia, ezt azonban a

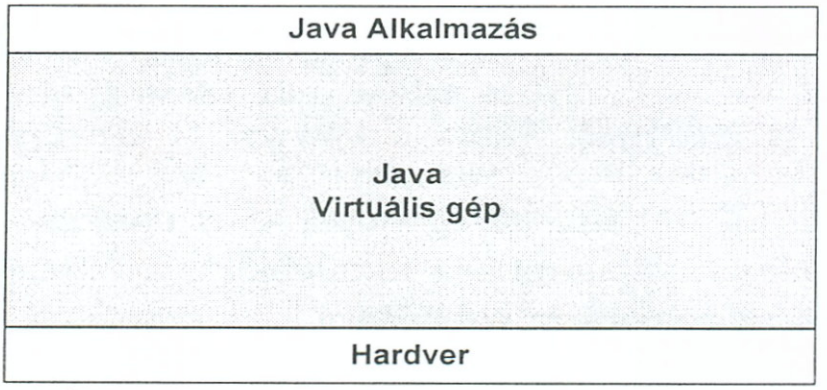
továbbiakban valóságos privilegizált módra kell váltania, ha feladatát el akarja látni.

2. A másik példa a lemezkezelés bonyolultsága. A virtuális gép alatt működő valóságos gép lemezeit ideális esetben úgy kellene elosztani a folyamatok között, hogy mindegyik, a többiektől függetlenül az egészet használni tudja. Ez persze nem lehetséges, de az állandó partíciókra való felosztás sem járható út, mivel a folyamatok száma változhat.

A kernel határának alacsony szinten történő megállapítása tehát kedvező az operációs rendszerek fejlesztői, illetve a régi programok alkalmazói számára, megvalósítása azonban közel sem probléma mentes, nem is beszélve arról, hogy elvész az operációs rendszerek egyik alapvető szerepe, a felhasználó megkímélése a hardver kezelésétől.

1.5.2 „Vékony" kliensek

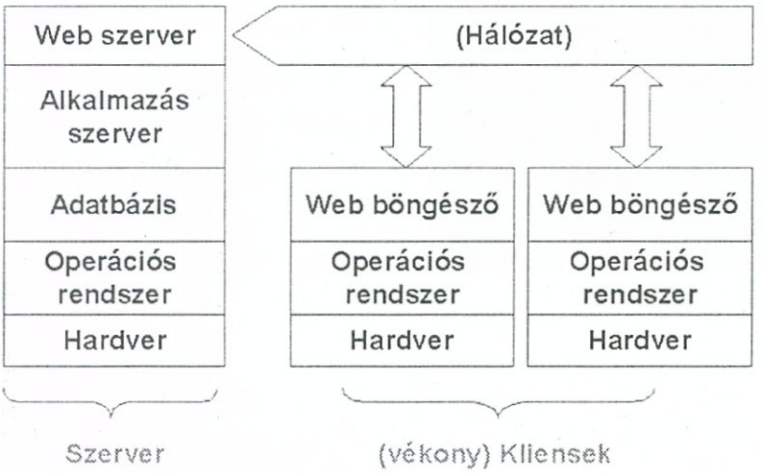
A másik véglet a Sun által kifejlesztett Java alapú rendszer egy lehetséges megvalósítása. A Java magas szintű, objektum orientált programozási nyelv, könnyedén hozhatunk általa létre grafikus objektumokat, ablakokat, képeket. A fordítás után keletkező, úgynevezett bájtkódok (bytecode) értelmezésére létezik már hardver megoldás, ennek széleskörű elterjedésével azonban feltehetően meg kell várnunk a hálózati számítógép (**N**etwork **C**omputer - **NC**) mindennapossá válását. Szoftver úton azonban minden processzor alkalmassá tehető Java appletek futtatására.



1.24 ábra A JAVA virtuális gép felépítése

A magas szintű utasítások bonyolult, összetett operációs rendszert igényelnek. Az ilyen rendszerek természetesen lassabbak, mint például egy C-ben, vagy Assembly-ben írt program, viszont a szabványos felületnek köszönhetően ma már szinte minden processzoron, minden operációs rendszer rendelkezik ezzel az interfésszel.

A gyakorlatban működő nagyobb, korszerű rendszerek már nem nagyon hasonlítanak a hagyományos személyi számítógépekhez, a személyi számítógép (gyakran hordozható kivitelben) gyakran csak a többé-kevésbé intelligens terminál szerepét tölti be. Tipikus szolgáltatási rendszert mutat a következő ábra:



1.25 ábra Korszerű alkalmazás környezet

A klienseken már elegendő szinte csak a böngésző jelenléte, a szerver az, amelyik megvalósítja a funkciók többségét. A hálózat létfontosságú – és a felhasználó ismét eggyel több technikai huncutságnak van kiszolgáltatva. (A szerver oldalon jól felismerhető az adatbáziskezelő, az üzleti logikát megvalósító alkalmazás, és a megjelenítést biztosító web szerver által alkotott háromrétegű modell. És ha már itt tartunk, bár az ábrán ez nem látszik, meg kell említenünk az egyes rétegek közötti kommunikáció már manapság is uraló XML-t is.)

1.6 A Linux története

A Linux példája jól mutatja be egy operációs rendszer kialakulásának folyamatát, fejlődését, valamint – a mindenkori felhasználói igényeknek megfelelően – összetevőinek bővülését.

1.6.1 Unix kezdetek

1965-től az AT&T Bell Laboratórium is részese volt annak a fejlesztésnek, melynek célja egy óriási, univerzális, mindenre alkalmas operációs rendszer, a MULTICS

megalkotása volt. 1969-ben világossá vált, hogy a készülő rendszer túl bonyolult, ezért túl drága, és mérsékelten hatékony lenne, így a Bell Laboratórium visszavonult. Egy csoport azonban, Ken Thompson vezetésével a megszerzett tapasztalatokat felhasználva, de reálisabb célokat maga elé tűzve olyan rendszeren kezdett dolgozni, mely alkalmas a programfejlesztés hatékony támogatására. Az első Unix változat assembly nyelven íródott a PDP 7, majd a PDP 11 miniszámítógépeken futott. Az egyfelhasználós, egyfeladatos operációs rendszer 16 kB memóriát foglalt el, a felhasználói programoknak 8 kB maradt, a háttértár kapacitása 512 kB volt. Milyen is volt a kezdeti Unix? Kidolgozták a fájl rendszert, a folyamatkezelést, a parancsértelmezőt, és a rendszerre egy assembler tette fel a koronát. Mivel elsősorban a programfejlesztés támogatása volt a cél, és nem az alkalmazások futtatása lebegett a szerzők szeme előtt, a hatékonyság, a hardver kihasználása, a biztonság és megbízhatóság szempontjai nem voltak éppen a legfontosabbak.

A Multics-ra rímelő Unix nevet Brian Kernighen adta az új operációs rendszernek. Dennis Ritchie segítségével, aki 1973-ban csatlakozott, átírták a Unix-ot C nyelvre, ami lehetővé tette, hogy a rendszert viszonylag kis erőfeszítéssel más gépekre is adaptálni lehessen, így nagyban elősegítette a rendszer elterjedését.

Az AT&T jelképes összegekért az egyetemek rendelkezésére bocsátotta a Unix-ot, annak forráskódjával együtt, megnyitva ezzel a folyamatos újítás, fejlesztés – és mellesleg a számtalan Unix változat kialakulásának – lehetőségét is. 1980-ban Microsoft és a kaliforniai Berkeley egyetem is a fejlesztők sorába állt, és a Unix hamarosan elnyerte lényegében mai formáját. Az új változatok (AT&T System V, Microsoft Xenix, Berkeley BSD) sokkal megbízhatóbbak lettek, támogatták több folyamat futtatását, a folyamatok közötti kommunikációt, a lokális hálózatokat, és az alkalmazott igény szerinti lapozás jelentősen megnövelte a programozók által használható címtartományt.

1.6.2 A Linux születése

Az 1980-as évek végén a személyi számítógépek piacán is megjelentek a 32 bites processzorok (intel 386), melyek kellően nagy teljesítményűek voltak ahhoz, hogy egy finn diák, Linus Torvalds 1991-ben hozzáfoghasson egy új Unix változat elkészítéséhez, melyet Linux névre keresztelt.

1991-ben a Linux 0.01 forráskódja az interneten szabadon elérhető volt, így fejlesztő kedvű számítógépbarátok széles köre kezdett az új operációs rendszer kiteljesítésén munkálkodni. A fejlesztés fő célpontja a kernel volt, hiszen a Linux, bár korrekt folyamat- és memóriakezelést valósított meg, nem tudott hálózatot kezelni, a perifériáknak csak egy nagyon kis hányadát támogatta, és a fájlkezelés területén is hagyott még kívánni valót maga után.

1994. májusában jelent meg a Linux 1.0. A legfontosabb változást a hálózati protokollok, és a hálózati kommunikáció megvalósítása jelentette, de számos hardver támogatással is bővült a Linux eszköztára: (természetesen az alapvető perifériák mellett) ismerte már az egeret, a CD-ROM-ot, a hangkártyát, a modemet, a hálózati kártyát. A siker hatására nemsokára megjelent az 1.1 változat, de annyi hibát tartalmazott, hogy a

felhasználók inkább visszatértek az előzőhöz. A Linux 1.1 emlékét őrzi az a hagyomány, hogy azóta a páratlan sorszámú Linux-ok mindig egy kísérleti változatot jelölnek.

1996. júniusában, a Linux 2.0 lényegesen javított memória- és fájlkezelő rendszert alkalmazott, és már kitört a PC-k világából, elkészültek a MIPS, Alpha, PowerPC, Motorola, Sparc változatok is. A megújult kernel már szálakat (threads) alkalmazott, és lehetőséget biztosított dinamikusan betölthető modulok használatára.

Az Linux életútja során megszerzett rugalmassága, stabilitása (és persze nem utolsó sorban ingyenes terjesztése) tette lehetővé, hogy napjainkra a hálózati szoftverek piacán elért részesedése elérte az összes többi unix változat együttes részesedését.

1.6.3 Rendszermag, rendszer, disztribúció

A Linux esetében érdemes különbséget tenni a rendszermag (*kernel*), a rendszer (*system*), valamint a teljes, összeállított szoftver csomag, a disztribúció (*distribution*) között.

A *kernel* teljesen eredeti alkotás, amely fölött egy fejlesztői csoport gyámkodik, és biztosítja integritását. Ugyancsak szigorúan stabil a fáljrendszer nevezéktana annak érdekében, hogy a rendszer összetevői között a kompatibilitás biztosítható legyen.

A kernellel ellentétben a *rendszer* tartalmaz olyan elemeket is, amelyek más pojektekből kerültek ide. A Linux rendszer gyarapodott például az MIT X-Window grafikus felületével, a GNU projekt C fordítójával, a hálózatmenedzselő komponenseket pedig eredetileg a BSD unix számára készítették.

A Linux *disztribúció* alapja a kernel, de ez kiegészül egy – a rendszer komponensekből, illetve hasznos unix alkalmazásokból (web böngésző, web szerver, szövegszerkesztő stb.) készített – szoftver gyűjteménnyel, ezen kívül több segédprogramot tartalmaz a telepítés, karbantartás megkönnyítésére. Az úttörő Slackware disztribúciót számos más csomag követte, manapság a legismertebbek a Red Hat, a Debian és a Caldera Linux névre hallgatnak.

A Linux szabad, tehát ingyenes szoftver, azonban nem mentes minden kötöttségtől. A licenc politika (GNU General Public License) legfontosabb eleme, hogy a Linux-ból senki nem húzhat anyagi hasznot és nem korlátozhat másokat abban, hogy az általa létrehozott terméket tovább fejleszthesse. Ha tehát valaki elkészít egy saját unix változatot, azt nem adhatja el, köteles a programmal együtt annak forráskódját is ingyen vagy jelképes összegért biztosítani.

A Linux tehát a Unix hagyományok megtartásával összeállított olyan többfeladatos, többfelhasználós operációs rendszer, amely rendelkezik a teljes Unix eszköztárral. Kialakulásának körülményeiből adódóan máig tervezési alapelve a lehető legjobb hardver kihasználás, a legújabb változatok is futnak egy 4 MB-tal felszerelt 386-os PC-n. A másik alapelv a szabad terjesztés, fejlesztés és az ingyenesség.

A Linux fejlesztésének legfontosabb jelenlegi célja a szabványosítás. annak érdekében, hogy se a programozókat, se a felhasználókat ne érjék különösebb meglepetések, a

Linux-ra készült (általában szintén ingyenes) alkalmazások gond nélkül fussanak az eltérő változatokon is.

1.7 Összefoglalás

A fejezetben megismerkedhettünk az operációs rendszerek fő funkcionális egységeivel, valamint a folyamatok, erőforrások fogalmával. Az egyes feladatok szükségszerű kialakulásának okait elemezve eljutottunk az operációs rendszerek réteges szerkezetének megismeréséig. Láthattuk, hogy az operációs rendszerrel való kommunikáció eszközei a felhasználói oldalról a rendszerhívások, a hardver oldalról a megszakítások, kerülőút a legtöbb rendszerben nem létezik. Összefoglalva, az operációs rendszerek legfőbb feladata a felhasználók minél magasabb szintű kiszolgálása, amit – ha egy kicsit mélyebbre nézünk – azáltal tudnak megvalósítani, hogy a versengő folyamatok között lehetőleg „igazságosan" és hatékonyan osztják el a véges számú erőforrást. Az egyes operációs rendszerek közötti különbségek abban mutatkoznak, hogy ezt a feladatot milyen feladat-környezetre optimalizálva, milyen eredményesen, mennyi erőforrás igénnyel oldják meg.

1.8 Ellenőrző kérdések

1. Ismertesse a kötegelt, az időosztásos és a valósidejű feldolgozási módok lényegét és fő jellemzőit!
2. Ismertesse a megszakítások típusait, a megszakításkezelés lépéseit!
3. Ismertesse az operációs rendszerek feladatait! Milyen fő részekből áll a rendszermag?
4. Mennyiben javítja a rendszerek jellemzőit több processzor alkalmazása? Mi a különbség a szimmetrikus, és az aszimmetrikus elrendezések között?
5. Mi a folyamatleíró blokk (PCB) szerepe? Milyen információ található benne?
6. Mi az összefüggés és a különbség program és folyamat között? Hogyan tartják nyilván az operációs rendszerek a folyamatokat?

7. Miért előnyösek a többfeladatos rendszerek? Milyen árat kell fizetni ezekért az előnyökért?
8. Mik a rendszerhívások, miért van rájuk szükség? Ismertesse a rendszerhívások kiszolgálásának lépéseit!
9. Mik a szálak (thread) és mi a kapcsolatuk a folyamatokkal? Melyek a szálak alkalmazásának előnyei, illetve hátrányai?
10. Ismertesse a legfontosabb Neumann-elveket! Mi a Neumann-ciklus?
11. Milyen módszerekkel kommunikálhat a CPU a perifériákkal? Értékelje őket!
12. Melyek az interaktív operációs rendszerek legfőbb jellemzői?
13. Milyen előnyökkel és hátrányokkal jár az elosztott rendszerek alkalmazása?
14. Ismertesse a „virtuális gép" koncepciót! Miért tekinthető az operációs rendszer virtuális gépnek?
15. Mutassa be az erőforrások, a felhasználói folyamatok és az operációs rendszer viszonyát!
16. Miért fontos a „felhasználói", és a „rendszer" üzemmód különválasztása többfeladatos rendszereknél?

2. A felhasználói felület

A felhasználói felületet ismertető fejezet áttekintést kíván nyújtani mind a felhasználó, mind a programozó lehetőségeiről. Tárgyaljuk mind a karakteres, mind a grafikus kezelői felületek működését, elsősorban a programindítással kapcsolatos funkciókat. Megismerhetjük a programok készítésének alapvető mozzanatait, valamint egy jó program kezelői felületével szemben támasztott legfontosabb követelményeket.

A legelső dolog, amivel a felhasználó találkozik, az operációs rendszerek felhasználói felülete.

Az eddigiekben láttattuk, hogy az operációs rendszerek szinte minden mozzanatot felügyelnek. Kiszolgálják, ütemezik a folyamatokat, biztosítják számukra a megfelelő erőforrásokat. A mai rendszerek túlnyomó többsége interaktív, tehát a felhasználó is igényli, hogy befolyásolhassa a rendszer működését. Miért ne biztosíthatnák ezeket a szolgáltatásokat nemcsak a felhasználói folyamat, hanem közvetlenül a felhasználók számára is?

Van még ezen kívül egy egyáltalán nem elhanyagolható szempont. Lehet, hogy egy operációs rendszer sokat tud, de egyet biztosan nem. Nem tudja kitalálni, hogy egy felhasználó mit akar, milyen programot szeretne indítani. A felhasználói felületre tehát már csak a vezérlés lehetőségének biztosítása érdekében is feltétlenül szükség van.

A felhasználói felület definiálása újból feleleveníti a tankönyvünk elején is felvetett problémát. Hol az operációs rendszer határa? Mit tekinthetünk az operációs rendszer részének, és mit alkalmazásnak, felhasználói programnak. A határok kérdése nemcsak a szolgáltatások fajtáit, hanem azok szintjét is érinti. A felhasználói felület a programozó számára a rendszerhívások és válaszok szintje, a végfelhasználó számára a billentyűzetről vagy a grafikus felületről kiadható, az operációs rendszer

számára adott parancsok, és az ezek hatására a monitorra érkező üzenetek szintje.

A felhasználói felület az ellátandó feladatok szempontjából az alábbi részekre bontható:

- programindítás, kapcsolat a folyamatokkal
- a rendszermag szolgáltatásainak közvetlen felhasználói elérése
- a rendszermag programozói felülete
- alapvető segédprogramok.

2.1 A felhasználó és a rendszermag

A felhasználók általában nem kerülnek közvetlenül kapcsolatba a rendszermaggal, azonban – elsősorban a személyi számítógépek esetén – lehetnek kivételek. A számítógép bekapcsolásakor a hardver ellenőrzése után az első esemény az operációs rendszer betöltése. Ugyanaz az operációs rendszer sok különböző felépítésű gépen futhat, ezért bizonyos esetekben ezeket a beállításokat minimális szoftver támogatással, kézzel kell elvégezni, máskor az operációs rendszer magas szintű támogatást (próbál) biztosítani.

Megjegyzendő – de a felhozott példákból is látható – hogy a felhasználói beavatkozás lehetősége elsősorban a számítástechnika hőskorának rendszereire volt inkább jellemző. Az újabb rendszerek bonyolultsága, összetettsége miatt kézi beállításokra kizárólag csak a legfelkészültebb, számítógépük hardver elemeit és az adott operációs rendszert egyaránt kimerítően ismerő felhasználók vállalkozzanak!

2.1.1 Külső erőforrások

A kézi beállítások tipikus példái az MS-DOS illetve a Windows 16 bites verziói, amelyek hosszú életútjuk során gyakorlatilag változtatás nélkül túlélték a nyomtatók, monitorvezérlők és más perifériák generációváltásait. A hosszú élet titka ez esetben az alkalmazkodóképesség volt. A DOS nyílt rendszer abban az értelemben, hogy felépítése és rendszerfelülete széles körben publikált, alkotói biztosították a lehetőséget a hardver gyártók számára, hogy a rendszerhez jól illeszkedő eszközvezérlő programokat készítsenek és lehetővé tették azt is, hogy ezek az eszközvezérlők szervesen beépülhessenek a rendszermagba. A rendszermag felépítése tehát végérvényesen a betöltődéskor dőlt el, mindössze egyetlen szöveges állomány vezérletével

(CONFIG.SYS, illetve SYSTEM.INI). A fájlban felsorolt eszközkezelők, az operációs rendszer belső meghajtó programjaival együtt alkotják a rendszermagot.

Automatikus beállítások támogatását tűzte ki célul a Windows 95 a nehezen lefordítható *plug and play* (PnP) eljárás kidolgozásával. A módszer lényege, hogy az operációs rendszer és a PnP támogatására felkészített hardver elem egy közös nyelven kommunikálhat, az eszköz az operációs rendszer kérdéseire megfelelő válaszokat ad, azonosítja magát és megadja a számára szükséges kiszolgáló rutinok adatait, illetve az operációs rendszer jelöl ki számára megszakítást és címtartományt. Az elv sokat ígérő, de a protokollt kezdetben nem minden gyártó alkalmazta kellő szinten, ami olykor meglehetősen megnehezítette a telepítést végzők életét. Az ilyen esetek nyomán született a szakmai zsargonban az eljárás „új" neve: *plug and pray* (telepítsd és imádkozz).

Félautomatikus megoldásokra is akad példa. A NetWare *scan for new devices* (új eszközök keresése) funkciója operátori paranccsal kezdeményezhető.

2.1.2 Belső erőforrások

Az operációs rendszerek alapvető erőforrása a **memória**. A memória lényegében minden gép esetén azonos módon viselkedik, de a különböző felhasználási módok, várható terhelések függvényében szükség lehet beállításokra. A leglátványosabb az eredetileg a 8086-os processzor 1 MB–os címtartományára készített DOS változása, mivel itt minden 1 MB fölötti címzési lehetőség a hangolás következménye. A kiterjesztett (**e**X**t**ended **M**emory **S**ystem – **XMS**) és kibővített memória (**E**xpanded **M**emory **S**ystem – **EMS**) megkülönböztetés a 32 bites processzorok és operációs rendszerek korában már elavultnak tekinthető, de még rengeteg alkalmazás működik, mely ezeket aktívan használja.

A **memóriakezelés** módosításának másik aspektusa (a címzési tartomány módosításán kívül) az átmeneti tárolók, a lemezgyorsító tárak kialakításának és vezérlésének kérdése. Az **adatátvitel gyorsításra** szolgáló memóriaterületek javítják a rendszer tulajdonságait, a javulás ára azonban az operációs rendszer méretének növekedése, tehát a felhasználói folyamatok rendelkezésére álló területek csökkenése.

Az optimalizálás a külső erőforrások használatához hasonlóan történhet kézzel, önműködően vagy valamilyen közbülső eljárással.

A DOS esetén a betöltődéskor kell megadni az **átmeneti tárolók** illetve **fájl leíró táblák** számát (FILES=80, BUFFERS=15) a Windows 95 ebbe nem sok beleszólást enged. A **Netware** szükség esetén **automatikusan foglal le memória területeket**, de az átmeneti tárolók minimális és maximális számát, valamint létrehozásuk és megszüntetésük paramétereit kézzel kell beállítani.

2.2 A programozói felület

Az operációs rendszerek egyik feladata a **hardver elrejtése a felhasználók elől**. A rendszermag a felhasználói folyamatok oldaláról

úgy látszik, mintha egy olyan számítógép (virtuális gép) lenne, amely olyan speciális erőforrás kezelő utasításokkal rendelkezik, amelyek csak az operációs rendszerre jellemzők, az aktuális hardver környezettől függetlenek. A felhasználói folyamatok írói, a programozók, a hardver kezelést csak a kernel szolgáltatásainak igénybe vételével végezhetik. Ilyen feltételek mellett természetes, hogy az operációs rendszer készítőinek, a rendszer legjobb ismerőinek a rendszermag funkcióinak megvalósításán kívül támogatnia kell a felhasználói alkalmazások készítését is. A rendszerhívások jól definiált rendszeréhez az assembly szintű programozók közvetlenül, a magasabb szintű nyelvek kedvelői összetettebb eljárásokon keresztül férhetnek hozzá.

2.2.1 A forráskód elkészítése

A **forráskód** elkészítésére szinte minden rendszer biztosít egy **szövegszerkesztőt** (editor), amely a konzolról begépelt szöveget egy állományba menti. A programozási nyelv megválasztása erősen függ az elvégzendő feladattól.

2.2.2 Fordítás

A forráskód alapján azt a gépi kódú programot, amely az adott processzor által ismert utasításokat, valamint a rendszerhívásokat megvalósító szoftver megszakításokat tartalmazza a **fordító program** (compiler) készíti el. Az így keletkező **tárgykódú** (**obj**ect – OBJ) modul az ugrások, változók abszolút címei helyére általában még a modul elejéhez viszonyított relatív címet rendel úgy, mintha feltételezné, hogy a program modul a 0 címtől töltődne be a memóriába. Az abszolút címek már csak azért sem szerepelhetnek a tárgykódban, mert egy program rendszerint több, külön fordított modulból áll. Az egyes program részleteket természetesen készítheti a felhasználó, de az operációs rendszer is tartalmazhat előre elkészített, statikus **rendszerkönyvtárakba** (**LIB**rary – LIB, **D**ynamic **L**inked **L**ibrary – DLL) rendezett általános, több alkalmazás számára is használható, elsősorban periféria kezelő, vagy a felhasználói felület kialakítását segítő rutinokat.

A modern operációs rendszereknek csak a leginkább hardver közeli részei íródnak gépi kódban, nagy részüket a **kifejezetten e célra kifejlesztett C nyelven készítik**, így, ha más nem is, de a C fordító az operációs rendszer születésével egy időben már rendelkezésre áll.

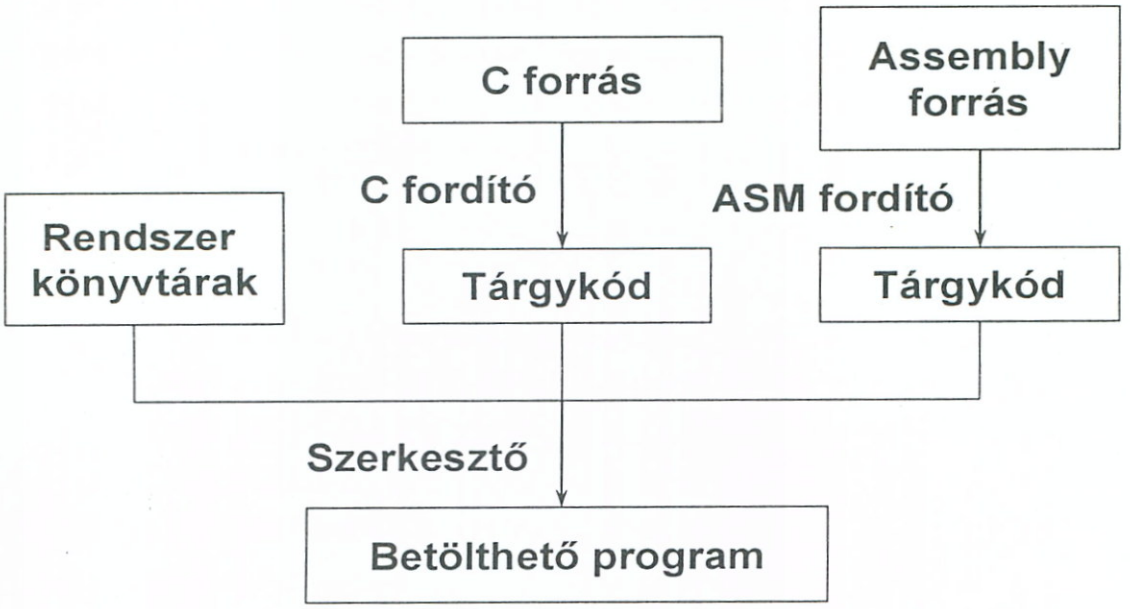
A kernel programozói szempontból egy függvény-, vagy eljárás könyvtárnak tekinthető, a programok készítéséhez szükséges információ leírás és segédprogramok összefoglaló neve a legtöbb esetben API (**A**pplication **P**rogramming **I**nterface).

Az alkalmazások írói számára biztosított rutinok száma általában erősen függ a szolgáltatások szintjétől. A DOS kb. 100 eljárása alapvető operációs rendszer funkciók elérését teszi lehetővé (például egy karakter kiírása az adott képernyő pozícióra), míg a Windows több mint 1000 függvénye a felhasználók által közvetlenül megtapasztalt felület kialakítását is támogatja (például ablak létrehozása menüvel, eszköztárral). A grafikus felületek egyedülálló népszerűségének éppen ez az egyik fő oka: a programozók – többek között saját jól felfogott érdekükben – az operációs rendszer által kínált magas szintű lehetőségeket használják fel, így mind a látvány, mind a működési mód hasonló lesz a legkülönbözőbb feladatú alkalmazások esetén is. Ez a programozási stílus a végfelhasználó számára is nagyon előnyös. Elég egy programot megtanulni, az összes többi szinte magától értetődő, a lényegtől nem vonják el a figyelmet a technikai részletek.

2.2.3 Szerkesztés

A **szerkesztő** (linker) feladata a tárgykódú modulok címeinek összehangolása, a kereszthivatkozások feloldása, a **betölthető program** (**EXE**cutable – **EXE**) előállítása. (A szerkesztőt gyakran *kapcsolat szerkesztő*nek nevezik, azért, hogy véletlenül se lehessen összekeverni a *szövegszerkesztő*vel.) A szerkesztett program még mindig nem tartalmazhat konkrét memória címeket. Mivel a rendszerben egyszerre több folyamat fut, előre nem tudható, hogy az elkészített program a memória mely tartományára kerül. Az operációs rendszer dolgát mindenesetre célszerű megkönnyíteni úgy, hogy amennyiben a processzor lehetővé teszi, a címeket egy, a program kezdetén beállítandó alaphoz (bázis) képest adjuk meg.

A programkészítés fent vázolt menetét a korszerű integrált fejlesztőrendszerek észrevehetetlenné teszik. Látszólag egy lépésben végzik, sőt program nyomkövetési támogatással (Debugger) - lépésenkénti végrehajtás, változók, memória tartalom kiírása - egészítik ki.



2.1 ábra Betölthető program készítése

2.2.4 Betöltés, dinamikus könyvtárak

A **betöltő** (loader) program már az operációs rendszer magjához tartozik, feladata a végrehajtható program elhelyezése a memóriában, a bázis cím kitöltése a megfelelő értékkel, a folyamat leíró blokk elkészítése. A betöltéssel válik egy program folyamattá.

Az így elkészített, illetve betöltött program hátránya, hogy pazarló módon minden részlete egyszerre kerül a memóriába, akkor is, ha ez nem szükséges. További hátrány, hogy azok a modulok, amelyeket több program is használ, mint például a rendszerkönyvtár elemei, annyiszor kerülnek betöltésre, ahány program használni kívánja őket. Az eddigi statikus szemlélettel szemben a problémát a **dinamikusan szerkeszthető könyvtárak** (Dynamic Link Library – DLL) segítségével lehet megoldani. A dinamikus könyvtárak csak akkor kerülnek a memóriába, ha azokra hivatkozás történik, bekerülésük után viszont több program is használhatja egyszerre őket.

2.3 Karakteres felhasználói felület

A parancsértelmezők az interaktív rendszerek kialakulásával jelentek meg, a parancsnyelvek továbbfejlesztéseként. Feladatuk az operációs rendszer szolgáltatásainak biztosítása az interaktív felhasználó számára. A funkció többféle elnevezésével találkozhatunk attól függően, hogy a feladatkör mely részét tekinthetjük elsődlegesnek. Gyakori a **shell** (burok, héj) elnevezés (Unix, DOS) mely arra utal, hogy a felhasználói felület burokba zárja, eltakarja a rendszer magját, a kernelt. A **command interpreter** (parancs értelmező) kifejezést azok használják, akik számára az operációs rendszernek adott parancsok kiszolgálása a leglényegesebb. Olykor találkozhatunk még a **monitor** (felügyelő) névvel, ha a folyamatok kezelése, nyomon követése az elsődleges. A grafikus rendszerek (pl. a Windows-család tagjai) a feladatkört több részre osztják, így létezik program, fájl-, nyomtató-, feladat (task) kezelő moduljuk.

A továbbiakban a megjelenés formáitól eltekintünk, a funkció megjelölésére a „shell" elnevezést használjuk. A shell-ek tulajdonképpen olyan felhasználói programok, amelyek az operációs rendszer lelkéhez közel állnak. Működésük nem igényel kivételes, privilegizált módot. Bizonyos esetekben – ha egy felhasználó mindig kizárólag ugyanazt a programot futtatja – a klasszikus shell el is maradhat, a felhasználó számára a felületet maga a felhasználói program jelenti.

Más esetben az egyes felhasználók, vagy alkalmazási módok más-más felhasználói felületet kedvelnek. A többféle felület a Unix-nál természetes (C, Bourne-shell), de ha egy kicsit tágabb értelmezést is megengedünk, shell-nek tekinthető a DOS esetén a Norton Commander, a PCShell vagy a DOS-Shell, Windows esetén az Intéző (korábban Fájlkezelő) és a Feladat kezelő (korábban Programkezelő) is.

A shell alapvető feladatai tehát:

- programindítás, programkezelés
- egyéb, operációs rendszer funkciók felhasználói szintű biztosítása (általános értelemben vett fájlkezelés).

2.3.1 Programkezelés

Mint láttuk, ez az a feladat, amely a felhasználó nélkül megoldhatatlan. Létrehozhatnak ugyan futó folyamatok is további folyamatokat, de a

közös ősnek valamikor emberi beavatkozással kellett megszületnie. A programokkal kapcsolatos műveletek lényegében négy részből állnak.

1. A betöltendő állomány kiválasztása.
2. A program számára a megfelelő környezet biztosítása.
3. A folyamat futásának megfigyelése, szabályozása.
4. Vezérlési szerkezetek megvalósítása.

2.3.1.1 Program indítása

A programok indítása általában nem más, mint a gépi kódú utasítás sorozatot tartalmazó, betölthető állomány nevének megadása. Egyes rendszerekben, illetve speciális betöltési mód esetén a program nevét meg kell előznie a **run** (fut) vagy a **load** (betölt) utasításoknak. A névnek természetesen egyértelműnek kell lennie, azaz valami módon tartalmaznia kell a fájl elérési útvonalát is. Gyakran használt programok vagy bonyolult fájlrendszer esetén ez meglehetősen kényelmetlen lehet, ezért bizonyos egyszerűsítő lehetőségeket vehetünk igénybe.

Közvetett fájl elérés. Lényege, hogy ugyanazon állományhoz több helyről is hivatkozhatunk. A hivatkozás maga is egy fájl, azonban egy elég kicsi fájl, melynek tartalma mindössze a betöltendő állomány neve, pontos helye, esetleg paraméterei. A közvetett elérésre példa a Unix link (kapcsolat), a Windows parancsikon, illetve a DOS és a NetWare parancsfájlja (BAT illetve NCF kiterjesztéssel). A valódi állományra hivatkozó indító fájl elhelyezhető az alapértelmezett katalógusban, vagy egyéb, könnyen megtalálható helyen.

Keresési útvonalat is megadhatunk a legtöbb esetben. Ilyenkor az operációs rendszer, ha a hivatkozott programot nem találja meg az aktuális katalógusban, egy előre adott rend szerint veszi sorba a megjelölt katalógusokat addig, amíg a keresett állományra nem bukkan. A Windows és a DOS a PATH (ösvény) nevű rendszerváltozóban tárolja a böngészendő katalógus listát, a NetWare külön meghajtókat (search drive – keresési meghajtó) definiál erre a célra.

Láncolt programfuttatás lehetséges, ha az említett parancsfájl több sort is tartalmaz. A technika nagyon emlékeztet a kötegelt feldolgozásnál említett parancsnyelvek világára, ami nem túlságosan meglepő. Az interaktív rendszerek a korábbi rendszerek emlőin nevelkedtek, azok

eszközeiből többet átvettek. Például a DOS kötegelt (batch), a NetWare a parancs (command) illetve a Unix (script) állományaiban foglalt utasításokat csaknem úgy hajtja végre, mintha egyenként gépeltük volna be őket.

Automatikus programbetöltésre akkor van szükség, ha néhány program indítását minden bekapcsoláskor vagy belépéskor amúgy is elvégeznénk. A láncolt programfuttatásnál tárgyalt speciális állományok módszere ez esetben is alkalmazható, de az általános parancsfájloktól ezeket valahogy meg kell különböztetni. A munka kezdésekor lefuttatandó programokat a DOS esetén a gyökérkönyvtárba elhelyezett AUTOEXEC.BAT állomány, a Windows esetén a WIN.INI fájl LOAD, illetve RUN változói tartalmazzák, míg a NetWare a felhasználói adatbázisban tárolja a szükséges adatokat. (A Windows rendszerekben az automatikus indításnak számos rejtett, nehezen felderíthető lehetősége is van, ez igencsak megnehezíti a „nem kívánt látogatókkal", a vírusokkal, kémprogramokkal folytatott küzdelmet.)

2.3.1.2 Program környezet beállítása

A programok futását befolyásoló, módosító paraméterek összességét nevezzük a program környezetének. (Nem szerencsés, de a magyar szaknyelv ugyanezt a kifejezést használja a program környezetre (environment) és a folyamat környezetre (context). A két dolog természetesen összefügg, de NEM azonos! A környezeti változók adatokat biztosítanak a létrejövő folyamat számára.)

Az adatok lehetnek:

- **paraméterek**, azaz például azon állományok nevei, amelyeken a programnak műveleteket kell végeznie.
- **kapcsolók** (switch, flag, option), melyek a működést pontosítják vagy akár alapvetően megváltoztatják.
- **átirányítási adatok** (redirection), melyek azt jelzik, hogy a program a bemeneti paramétereit az alapértelmezett forrás – általában a billentyűzet – helyett honnan kérje, illetve kimenetét a legtöbbször használt monitor helyett melyik fájlba irányítsa.
- **környezeti változók** (environment variables), az operációs rendszer beállításait jelző paraméterek. Míg az előző adattípusok a

parancssorban adhatók meg, az operációs rendszer környezeti változóinak beállítására külön utasítások szolgálnak (pl. SET), melyek gyakran az automatikusan lefutó parancsállományban kapnak helyet.

2.3.1.3 A folyamat futásának ellenőrzése

Az ellenőrzés lehetősége személyi számítógépeken általában nagyon korlátozott, mindössze az éppen aktív folyamatok listázására van lehetőség. Többfeladatos rendszereknél azonban lehetőség van a futó folyamatok megszüntetésére vagy szüneteltetésére is (pl. Unix Kill, illetve Windows Feladatkezelő). A nagyobb, többfelhasználós operációs rendszerek viszont igen részletes információt is szolgáltatnak, pl. a Windows Feladatkezelője, a NetWare Monitora az egyes folyamatok által használt processzor időt és memória területet is nyilvántartja.

2.3.1.4 Vezérlési szerkezetek

A parancsnyelv általában nem csak program indításra, illetve paraméterezésre alkalmas, hanem összetett, feltételhez kötött és ciklus utasítások is megadhatók vele. Szélsőséges esetnek tekinthetők a kezdeti személyi számítógépek, melyeknél a parancsfordítót maga a BASIC nyelv alkotta (command interpreter = BASIC interpreter). Ha korlátozottabb mértékben is, de mind a DOS, mind a Unix parancsértelmezője alkalmas utasítások egymás utáni végrehajtására (szekvencia), feltételes elágazás (szelekció) megvalósítására, illetve ciklus szervezésére (iteráció), így program-szerű fájlok készítésére.

2.3.2 A parancsértelmező egyéb funkciói

A parancsértelmező a programindításon kívül leginkább állományokkal, katalógusokkal kapcsolatos műveleteket végez, az egyéb feladatok (pl. idő, dátum, lekérdezés) szinte elenyészők. A fájlokkal kapcsolatos műveletek sem mindig tartoznak a parancsértelmező kompetenciájába, gyakran külön segédprogramok formájában valósulnak meg (Unix) vagy külön Shell-t alkotnak (Windows fájl kezelő). Ha az adott operációs rendszer egyes funkciókat a parancsértelmező részeként, míg másokat külön segédprogramonként valósít meg, megkülönböztetünk *külső* és *belső* parancsokat.

Az előzőekben volt már szó arról, hogy egy rendszerben több shell is lehet, akár egyidejűleg is, a shell-ek azonban egymásra is épülhetnek. A

NetWare munkaállomás kliens szoftvere a parancsértelmező tetején képez újabb réteget, és a munkaállomás operációs rendszerének szóló parancsokat egyszerűen továbbadja, míg a szervernek címzett utasításokat (a megfelelő rétegeken keresztül, de a parancsértelmező kikerülésével) egyenesen a szerver felé továbbítja.

2.4 Grafikus felhasználói felületek

A felhasználó és a számítógép párbeszéde kezdetben a lyukkártyákra rögzített, majd a konzolírógépen begépelt parancsok segítségével történt, a közeli jövőben talán a kimondott szavak, majd a gondolatok irányíthatják gépeink működését, azonban jelenleg a leginkább elterjedtek, a legnépszerűbbek a grafikus felhasználó felületek. Az IBM kompatíbilis gépeken a Microsoft Windows különböző variációival találkozunk, az Apple a MacOS grafikus felületét használja, a Unix, és általában a nagygépes operációs rendszerek az X.11 szabványon alapuló X-Window rendszert valósították meg.

A karakteres világ parancsértelmezője nem sokban különbözik egy egyszerű felhasználói programtól. Egyfeladatos rendszerekben (single task, pl. DOS) a parancsértelmező fut, amikor éppen semmi más nem történik; háttérbe vonul, amikor egy felhasználói program kap vezérlést, és annak végeztével újra előbukkan, és várja a következő parancsot.

Többfeladatos, karakteres felületet használó rendszerekben (multitask, pl. Unix) ugyan futhatnak folyamatok a háttérben, azonban ezek működését csak meglehetősen nehézkesen lehet nyomon követni, illetve befolyásolni. Milyen jó lenne, ha minden folyamatra nyithatnánk egy ablakot, amelyen keresztül megfigyelhetjük a történéseket, beavatkozhatunk, vagy ha úgy tartja kedvünk, az ablakot be is csukhatjuk (ez persze nem jelent azt, hogy a folyamat is leáll)!

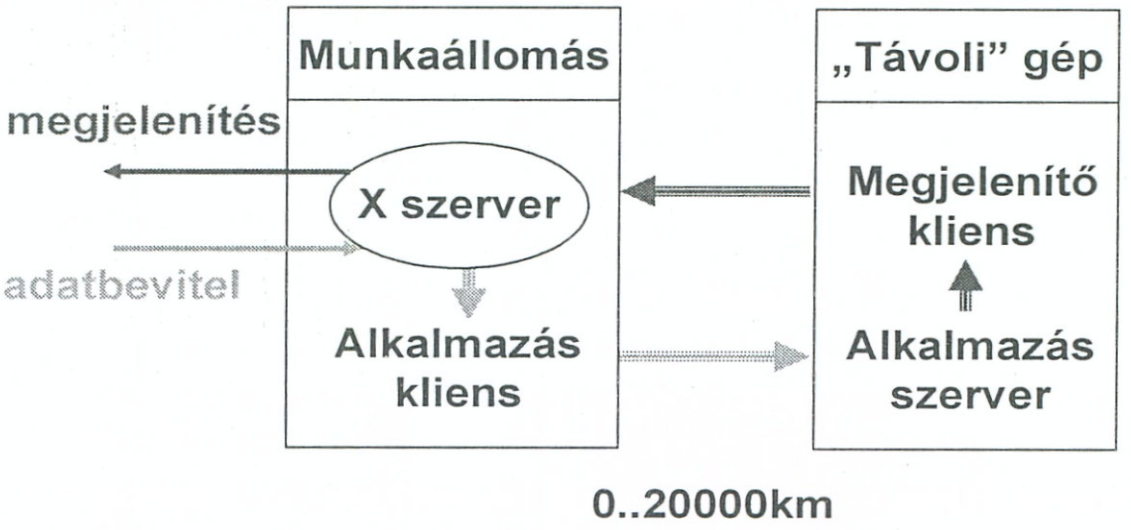
Az ablakozó technikák tehát a többfeladatos rendszerekhez kötődnek, fejlődésük kezdete a 80-as évek elejére tehető.

2.4.1 Az ablakozó rendszer működése

Egy szép színes grafikus képernyő kezelése meglehetősen bonyolult feladat, ezért az ilyen eszközt alkalmazó operációs rendszerben ezt a funkciót egy speciális folyamat csoport, egy alrendszer látja el. Ilyen rendszer az X-Window, és a Microsoft Windows, avagy legújabban a

Novell NetWare GUI-ja is (**G**raphical **U**ser **I**nterface – Grafikus Felhasználói Felület).

A GUI feladata, hogy az őt „használó" folyamat számára biztosítsa a grafikus bevitel, illetve megjelenés lehetőségét. A grafikus interfész tehát egy **szolgáltatást** (szerver, server) nyújt a hozzá forduló **ügyfelek** (kliens, client), a folyamatok számára. A rendszer működése a szolgáltató és ügyfelei közötti, meghatározott szabályoknak (protokoll) eleget tevő üzenetváltáson alapul. Az üzenetvezérelt működés szempontjából teljesen érdektelen, hogy a futó alkalmazások ugyanazon a számítógépen futnak-e, amelyiken a grafikus szolgáltató a hozzájuk tartozó ablakot megjeleníti. A két szélsőséges eset a Windows, ahol az alkalmazások és a GUI ugyanazt a hardvert használják, illetve az X-terminál, ahol a felhasználói interfészt biztosító gép nem is alkalmas másra, így szükségképpen minden kiszolgált ügyfél egy másik gépen kell fusson. Hogyan is működik ez?



2.1 ábra Az X-szerver és az X-kliens együttműködése

Nézzük, hogyan is történik a kommunikáció az alkalmazást futtató és a felhasználói interfészt biztosító számítógép között! Tegyük fel, hogy a Munkaállomásnak nevezett számítógépen már fut az X-szerver, és egy olyan ügyfél program, amely egy távoli gép szintén már futó alkalmazásával áll kapcsolatban (gondoljunk például egy web böngészőre és egy web szerverre). (Már most érdemes megfigyelni, hogy mindkét kommunikáló gép kliens és szerver funkciót egyaránt ellát!)

1. A munkaállomás előtt ülő felhasználó egerével rákattint egy menüpontra vagy ikonra, azaz egy **eseményt** (event) idéz elő.

2. Az ablakkezelő rendszer, az X-szerver az egér pozíciójából megállapítja, hogy a felhasználó melyik alkalmazással kíván kommunikálni. (Billentyűzet használata esetén némi kompromisszum árán a feladat szintén megoldható, erre rövidesen visszatérünk.)
3. Az X-szerver gondoskodik arról, hogy az üzenet eljusson ahhoz a távoli géphez, amelyen a megfelelő alkalmazás szerver fut.

(A távoli gép üzenetkezelő rendszere fogadja és értelmezi az érkezett üzenetet, majd eljuttatja az alkalmazás szerverhez. Az alkalmazás szerver végrehajtja a parancsot, és válaszát a kommunikációs interfészen keresztül visszajuttatja az X-szerverhez.)

4. Az X-szerver fogadja az alkalmazástól érkező megjelenítési információt, és végrehajtja a szükséges változtatásokat (például egy újabb almenüt jelenít meg az alkalmazásszervertől kapott menüpontokkal).

Ez az üzenet feldolgozási ciklus folytatódik folyton–folyvást.

2.4.2 A grafikus felületek jellemzői

A grafikus felhasználói felületen tehát minden alkalmazáshoz tartozik egy vagy több ablak. Azok a területek, amelyek az ablakok szélein kívülre esnének, levágódnak. Az ablakok egymást részben vagy egészben eltakarhatják, az ablakok mozgatásáért, illetve méretváltoztatásáért teljes egészében az ablakozó rendszer felel, a kiszolgált alkalmazásnak erről nincs tudomása. Ha az ablakok sorrendje megváltozik, azaz valamelyik ablak a képernyőn takart helyzetből láthatóvá válik (ez például egy egér kattintással, vagy egy segédprogrammal érhető el), a grafikus interfész egy speciális üzenetet küld az alkalmazásnak (kitakarás, expose), melyben kéri őt, hogy a hiányzó részt újra küldje el.

Az alkalmazások ablakainak lehetnek gyermekei, azaz olyan ablakok, amelyeket a szülő alkalmazás hozott létre. Ezek az ablakok csak a szülő ablakon belül mozoghatnak, azt nem hagyhatják el.

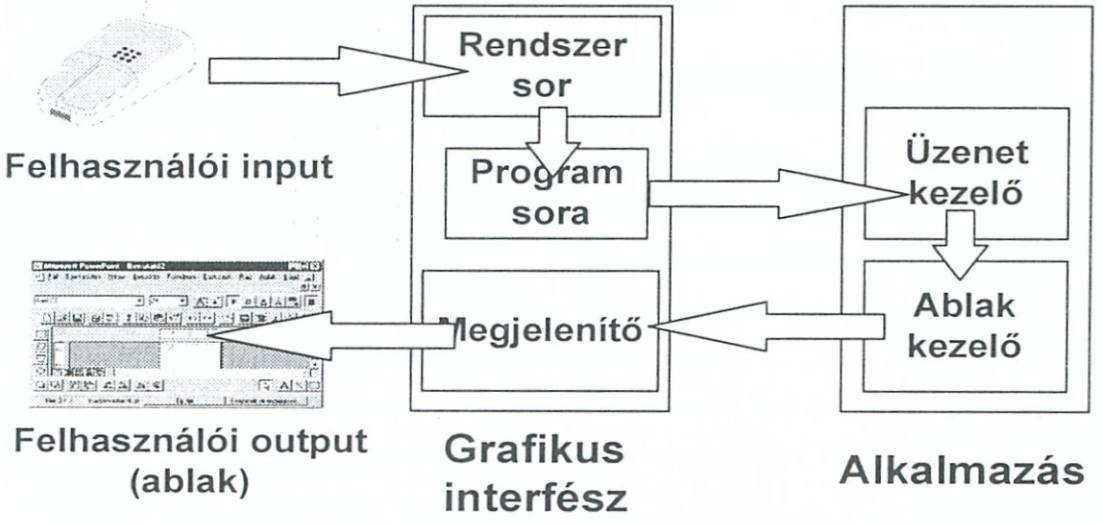
A bemutatott működési modellből már levezethetők azok a követelmények, amelyeket egy grafikus interfészt biztosító rendszernek teljesítenie kell:

- A multitask környezet miatt megoldandó az **események címzettjeinek felismerése** (aktív ablak, fókusz).
- Az alkalmazás oldalon **eszközfüggetlen működés** biztosítandó annak érdekében, hogy az alkalmazás többféle környezetben is futtatható legyen.
- A kommunikációs csatorna tehermentesítése érdekében az adatforgalom csökkentendő a kliens és szerver között amennyire lehetséges. (A távolság igen nagy is lehet!)

2.4.2.1 Az események címzettjének felismerése

Események akkor következnek be, ha az egér, billentyűzet, vagy egyéb beviteli eszköz állapotában változás áll be, illetve, ha a rendszer állapota megváltozik (például lejár az időszelet).

Az üzenet először a rendszer üzenetsorába kerül. Az egér aktuális koordinátáiból egyértelműen megállapítható, hogy mely ablak felett jár, így azonosítható az az alkalmazás, amelynek a kattintással küldött üzenet szól, így továbbítható az üzenet a megcímzett program üzenetsorába.



2.2 ábra Üzenetvezérlési ciklus

Nem ilyen egyszerű a helyzet a billentyűzettel, mivel a billentyűzethez nem rendelhető pozíció. Az ablakozó rendszereknél ezért mindig van egy kitüntetett (jól láthatóan megkülönböztetett) ablak, amelyikhez a rendszer a billentyű leütéseket rendeli, erre az ablakra irányul a figyelem (input focus). A kitüntetett ablak egérrel vagy segédprogrammal megváltoztatható.

Megjegyzendő, hogy az üzeneteket az alkalmazás csak abban az esetben dolgozza fel, ha az számára értelmes, egyéb esetekben eldobja.

2.4.2.2 Eszközfüggetlen működés

Az ablakozó rendszerre íródott alkalmazások, azért, hogy bármilyen kiépítettségű rendszeren futni tudjanak, nem használhatják ki az adott gépen alkalmazott perifériák speciális tulajdonságait, illetve beállításait. Az ügyfelek eszközfüggetlen üzeneteket küldenek, amelyeket az ablakozó rendszer a maga legjobb tudásának megfelelően végrehajt. Az optimális működés érdekében általában van arra is lehetőség, hogy az alkalmazás lekérdezhesse az őt kiszolgáló grafikus felület paramétereit (színek száma, képernyő felbontása stb). A színek kezelésében is az ablakot szolgáltató grafikus interfészé a fő szerep, a program csak ennek palettájáról válogathat.

Az eszközfüggetlen működés természetesen csak kompromisszumok árán biztosítható. Ha a vélt vagy valós sebesség igények, vagy a grafikai megoldások ezt nélkülözhetetlenné teszik, az alkalmazásfejlesztők hajlamosak lemondani a szabványos eszközökről akkor is, ha ezzel veszélyeztetik a többi alkalmazás, vagy akár az operációs rendszer biztonságát (ne feledjük: assembly szinten minden megtehető).

2.4.2.3 Az adatforgalom csökkentése

Különösen abban az esetben, amikor a szolgáltató és az ügyfél a világ két ellentétes pontján található, nagyon fontos, hogy a lassú és túlterhelt csatornán (például az interneten) a lehető legkevesebb adatnak kelljen átmennie. Ennek érdekében az ügyfél nem minden apró kívánságát küldi el az ablakozó rendszernek, hanem több utasítást csokorba fogva. „Hálából" a szolgáltató minden lehetséges dolgot megpróbál önállóan ellátni, például kezeli az egérmozgást, tárolja az ideiglenesen nem látható ablakrészek tartalmát, hogy azt ne kelljen mindig a távoli ügyféltől kérnie. A leghatékonyabb segítség azonban az, hogy a grafikus interfész magas szintű objektumokkal rendelkezik, az ügyféltől mindössze ezek paramétereit kéri. A Microsoft Windows objektumait mindenki jól ismeri: párbeszéd ablakok, rádiógombok, listák, menük stb.

Gyakran arra is van lehetőség, hogy az alkalmazás hozza létre a kiszolgálón ezeket az objektumokat a program indításakor (grafikus környezet, graphic context), és futás közben már csak ennek paramétereit küldözgeti. Tipikus példa erre a betűtípusok letöltése.

2.5 Segédprogramok, alrendszerek

Már a programfejlesztői támogatás esetén is nehéz eldönteni, hogy hol húzzuk meg az operációs rendszer határait, még inkább így van ez a felhasználó által közvetlenül használt programok esetén.

A felhasználói programok leggyakrabban használt példája az interaktív rendszerekben használt tolmács program, a parancs interpreter. A parancsnyelvet gyakran nevezik **shell**-nek (héj, burok), mivel a felhasználó csak ezen keresztül érintkezhet a maggal, a kernellel. A parancsnyelv nem más, mint a rendszerhívások, illetve rendszerhívások sorozatának kiadására szolgáló magas szintű eszköz, ezért vitathatatlanul az operációs rendszerhez tartozik, annak ellenére, hogy azonos funkciók ellátására más shellek is írhatók.

A gyakran használt operációs rendszer szintű feladatok megoldására szolgáló **segédprogramok** (lemezformázás stb.) szintén közel állnak az operációs rendszerhez.

Gyakori, hogy egy speciális felhasználói igényt nem az operációs rendszer módosításával elégítenek ki, hanem egy, a segédprogramok körét kiegészítő, úgynevezett **alrendszerrel** bővítik az operációs rendszer magas szintű szolgáltatásait. A programfejlesztési támogatás is felfogható alrendszerként, de egyre terjednek a grafikus és adatbázis alrendszerek is. Az alrendszerek alkalmazásának előnye, hogy nem kell módosítani hozzá az operációs rendszert, hátrányuk, hogy egy újabb szint ékelődik a felhasználó és az operációs rendszer közé. Az első interaktív rendszerek is ilyen alrendszerként kerültek megvalósításra.

A segédprogramok, alrendszerek leggyakoribb működési területei a következők:

- **Állományok kezelése**. Az állományok másolása, áthelyezése, átnevezése, törlése, valamint a katalógusok létrehozása mellett, a segédprogramok között megtalálhatók a lemezek partícionálására, formázására szolgálók is. Tipikus segédprogram a Norton Commander, mely a leggyakrabban használt 10 fájl művelet végrehajtását teszi hallatlanul kényelmessé.
- **Programfejlesztés**. Szinte valamennyi operációs rendszer tartalmaz egyszerű szövegfájlok előállítására alkalmas programot (editor),

valamint szerkesztőt (linker). A Unix alapú rendszereknek gyakran szerves része a C fordító is.

- **Adatbázis kezelés**. A számítógépes rendszerek jó része adatbázisokat kezel. Ennek támogatására olykor az operációs rendszerek külön eszközöket is tartalmaznak, néha, pl. az AS400 esetén szinte elválaszthatatlanul a többi funkciótól.
- **Kommunikáció.** A hálózatok kezelése napjainkban egyre fontosabb. Az alapvető hálózati rétegeket megvalósító folyamatok ma már szinte kivétel nélkül az operációs rendszer szerves részét alkotják.

A segédprogramok, alrendszerek és felhasználói programok lényegében nem különböznek egymástól, gyakorlatilag csak azok eredete, súlya vagy funkciója dönti el, melyik programot vagy program csoportot melyik kategóriába soroljuk.

Az alrendszerek tehát az operációs rendszerek ideiglenes kiegészítőinek is tekinthetők, így várható jövőjük meglehetősen szomorú. Vagy azért szűnnek meg, mert az általuk biztosított funkció olyan fontosnak bizonyul, hogy érdemes beépíteni az operációs rendszerek későbbi generációiba, vagy azért, mert nem váltják be a hozzájuk fűzött reményeket.

2.6 Egy felhasználóbarát felület jellemzői

Legyen szó egyszerű karakteres felületről, akár a legcsillogóbb grafikus rendszerről, az interaktív programokkal illetve kezelői felületekkel szemben támasztott követelmények már régóta azonosak. Az alábbiakban röviden felsorolt tulajdonságok szem előtt tartása természetesen nemcsak az operációs rendszerek kezelői felületénél, hanem minden program készítésénél hasznosak.

- **Könnyű legyen megtanulni**. A menüszerkezetek szinte magukért beszélnek, míg egy billentyű kombinációt általában sokáig tart begyakorolni.

- **Méretezhető legyen**. A kezdő felhasználó kapjon sok segítséget, de az örökös javaslatok, tanácsok ne háborgassák állandóan a tapasztalt felhasználót.

- **Lehessen visszavonni** következmények nélkül a tévedésből kiadott, hibás utasításokat.
- **Meg lehessen szakítani** az elindított műveleteket végrehajtás közben is. (És a megszakításnak természetesen ne legyenek káros következményei.)
- **Legyen többszintű súgó (help) rendszer**. Az ismerkedőknek szóló tankönyvtől (tutorial) a részletes technikai leírásig (technical reference) minden szintet érdemes megvalósítani. A súgó rendszerben könnyen lehessen keresni, egyszerűen legyen előhívható. Meghívása esetén csak azok a részek legyenek láthatók, amelyek az adott szituációra alkalmazhatók.
- **Használata hasonlítson a nyelvhez.** Az utasítások legyenek igék, a paraméterek főnevek.
- **Minden utasításra legyen válasz!** Nagyon elbizonytalanító érzés, ha begépelünk egy parancsot vagy az egérrel rákattintunk egy ikonra és látszólag (vagy valójában) semmi sem történik. Ide tartozik az az eset is, ha egy folyamat hosszabb időt vesz igénybe, legyen valami jele annak, hogy éppen fut.
- Hasonló funkciókat hasonló módon lehessen végrehajtani akkor is, ha a programok feladata alapvetően különbözik.

2.7 A Linux felhasználói felülete

A Unix rendszerekben a felhasználói felületet egy éppen olyan program biztosítja, mint bármelyik másik alkalmazás, hagyományos neve **shell**. Nem része a **kernel**-nek, tehát nem kell állandóan a memóriában tartózkodnia. A klasszikus felület karakter orientált, legnépszerűbb képviselői a Bourne shell (bsh), C shell (csh) és Korn shell (ksh) névre hallgatnak, de gyakran születnek új, speciális változatok is. A Linuxban például legtöbbször a *bsh* módosított, barátságosabb változatát, a Bourne Again Shell-t (*bash*) valósítják meg. A shell segítségével futtathatók azok a rendszerprogramok, melyek minden Unix változatban megtalálhatók. Ezek a programok a megfelelő rendszerhívások által végzik tevékenységüket, azonban maga a rendszerhívás a felhasználók számára rejtve marad. A rendszerprogramok többsége a fájlokkal, illetve katalógusokkal kapcsolatos műveleteket végez. A leggyakrabban használt funkciók a következők:

```
mkdir       katalógus létrehozása
rmdir       katalógus törlése
cd          az aktuális katalógus váltása
pwd         az aktuális katalógus kiíratása
```

```
ls          a katalógus tartalmának listája
cp          fájl másolása
mv          fájl áthelyezése
rm          fájl törlése
cat         fájl tartalmának listázása
```

Meg kell említeni, hogy a funkciók többségét külön programok valósítják meg, csak elvétve található olyan, amit a shell tartalmaz.

A programok indítását tehát a shell végzi. Ha a várakozási jel (prompt) után a program nevét gépeljük be, az az előtérben fog futni, a shell-t megvalósító program várakozik. A program nevének végére illesztett '&' jel azt jelenti, hogy a shell a program indítás után azonnal visszakapja a vezérlést, nem kell várakoznia, tehát a program a háttérben futhat. A programoknak lehetnek kapcsolói, illetve paraméterei.

A programok bemenő adatai egyszerű esetben a billentyűzetről származnak, a kimenő adatok, illetve hibajelzések a monitorra kerülnek (*standard input, output, error*). A szabványos bemenet és kimenet azonban szövegfájlba irányítható, sőt másik program is szolgálhat forrásként, illetve célként. Ez utóbbi esetben egy speciális átmeneti fájl, a *pipe* (csővezeték) biztosítja az adatok átadását. Minkét esetre nézzünk egy-egy példát!

`$ ls > dirfile`	az aktuális katalógus listázása fájlba
`$ cat nevsor|sort|more`	a *nevsor* nevű fájl tartalmának listája a képernyőre ABC-sorrendben, laponként

Az olyan programokat – mint például a *sort* és a *more*, amelyek a szabványos bemenetről fogadják adataikat, és némi átalakítás után a szabványos kimenetre küldik, szűrőknek (filter) nevezzük.

A shell lehetőséget ad programok írására is. Az ilyen programok neve *shell script,* céljuk olyan utasítás sorozatok végrehajtása, melyekre gyakrabban van szükség. Ilyen például a keresési útvonal (*search path*) megadása.

A felhasználói felület állandósága tulajdonképpen meglepő, hiszen a kernellel ellentétben egy új shell megírása nem különösebben nehéz. Az önkéntes konzervativizmust azonban a napjainkban rohamosan terjedő és egyre népszerűbb X-Windows grafikus felületek is megőrzik.

A felhasználói, illetve programozói felület legfontosabb elemeit ismerhettük meg. A fordító, szerkesztő programok feladatának, az ablakozó rendszerek működének ismeretében jobban megérthetjük a korszerű alkalmazások készítésének lépéseit. A shell beállítási

Σ

lehetőségeinek feltárása lehetővé teszi operációs rendszerünk finomabb „hangolását", hatékonyabb alkalmazását.

2.8 Ellenőrző kérdések

1. Milyen lépésekben történik egy program elkészítése és betöltése?
2. Definiálja 1-2 mondattal a következő fogalmakat:
   a. betöltő program (loader)
   b. közvetett fájl elérés
   c. keresési útvonal
   d. láncolt programfuttatás
   e. automatikus programbetöltés
3. Mit nevezünk egy program környezetének? Milyen jellemzők tartoznak ide?
4. Hogyan támogatják az ablakozó rendszerek az adatforgalom csökkentését, az eszközfüggetlen működést?
5. Ismertesse az ablakozó rendszerek működési elvét a kliens-szerver modell alapján!
6. Mik az alrendszerek? Mi az alkalmazásuk előnye és hátránya? Milyen fontosabb alrendszereket ismer?
7. Melyek egy programfejlesztői alrendszer fontosabb elemei? Hogyan segítik a programfejlesztést?
8. Mi a fordító, a szerkesztő és a betöltő programok fő feladata?

3. Állományok, katalógusok

A mágneslemezek alkalmazása lehetővé tette a programok és adatok hosszabb távú tárolását. Az állományok számának szaporodásával azonban ezek elnevezése, rendszerezése is szükségessé vált. A következő fejezet az állományokhoz kapcsolódó műveleteket, tulajdonságokat, illetve – többfelhasználós környezetben – az állományokhoz és katalógusokhoz rendelhető hozzáférési jogokat ismerteti, valamint a fájlok fizikai elhelyezésének főbb módszereit mutatja be.

Az állomány (vagy más néven fájl) olyan tárolásra szánt adatcsoportot jelöl, melynek leglényegesebb tulajdonsága az, hogy egy választott név segítségével együttesen kezelhető. Nem kell tehát a felhasználóknak azzal foglalkozniuk, hogy állományaik egy lemez melyik sávján, azon belül is melyik blokkban helyezkednek el, hanem nyugodtan rábízhatják magukat az operációs rendszerre, mely a megadott név alapján előkeresi számukra a megfelelő programot vagy a kívánt adatokat.

Állomány (Fájl)
Az adatok egy olyan csoportja, melyre együttesen, névvel hivatkozhatunk

§

A fájl fogalma alatt általában tárolt adatokat értünk, de léteznek olyan operációs rendszerek (például a Unix), ahol ezt a szemléletet minden külső adatfolyamra, így a képernyő tartalomra és a billentyűzetre is általánosították. A fájl fogalom kiszélesítése a programozó számára rendkívül kellemes, ezért egyes elemeit minden operációs rendszerben megtalálhatjuk. A programozó dolga nagyban leegyszerűsödik, mivel nem kell törődnie azzal, hogy programja milyen perifériáról kap majd bemenő adatokat, vagy eredményeit nyomtatóra, képernyőre, fájlba kell

kiírnia, vagy csupán át kell adnia egy másik programnak, elegendő egy fájlra hivatkozni.

A mágneses háttértárak feladata az operatív tár kiegészítése, méretének látszólagos megnövelése, ha ez szükséges (lásd virtuális memória), illetve a felhasználók által létrehozott állományok megőrzése, hogy tartalmuk kikapcsoláskor ne vesszen el, hanem később is elérhető legyen.

A háttértárakon tárolt adatok céljuk szerint három csoportba sorolhatók:

Σ

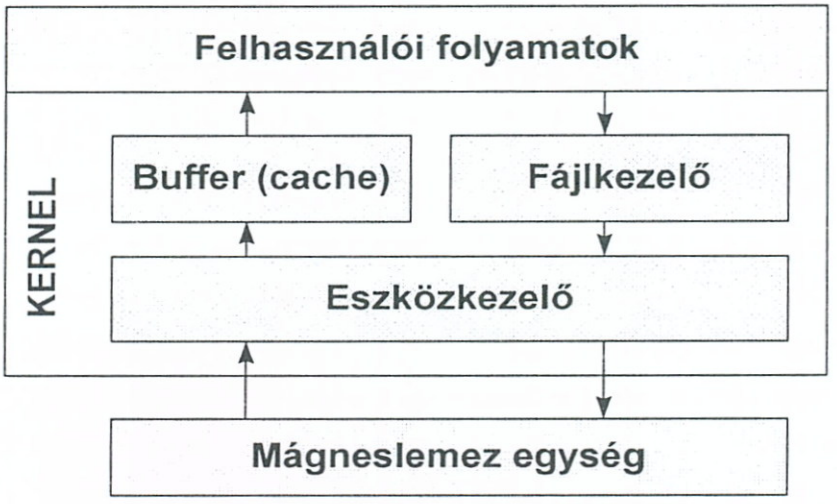
1. **Ideiglenes állományok**, melyeket az operációs rendszer saját működésének támogatására hoz létre. Ilyenek például a memória kezelő által készített, az operatív memóriába éppen nem betöltött lapok vagy szegmensek, illetve cserefájlok.
2. **Felhasználói állományok**, melyekre nevükkel hivatkozhatunk, azaz a klasszikus értelemben vett fájlok, a felhasználói adatok, programok tartós tárolására.
3. **Adminisztratív állományok**, melyek ahhoz szükségesek, hogy az operációs rendszer számára a felhasználók által létrehozott állományok kezeléséhez, megtalálásához szükségesek. Ezek szerkezete, tartalma a felhasználók elől általában rejtett.

A fájlok tartalmuktól függően nagyon sokfélék lehetnek. Az aktuális operációs rendszer szabályai döntik el, hogy ez a különbözőség a névben vagy felépítésben is megjelenik-e, vagy csupán értelmezésben. A Unix esetében például nincs elnevezésbeli megkülönböztetés, de a DOS alapú rendszerek kitüntetik figyelmükkel a végrehajtható bináris programokat (EXE, COM), illetve a parancsnyelvi programokat (BAT).

A felhasználói állományok többnyire állandóak, azaz létrehozásuktól kezdve megmaradnak a háttértárolón egészen addig, amíg megszüntetésükről kifejezetten nem intézkedünk. Mivel a felhasználói folyamatok a hardverrel közvetlenül nem érintkezhetnek, ezért ha egy folyamat valamilyen háttértárolón lévő bemenő adatra vágyik, vagy eredményeit tárolni szeretné, egyéb lehetőség híján, egy rendszerhívás által az operációs rendszer magjához fordul.

A rendszermag azon részét, amely a fájlokkal kapcsolatos műveleteket végzi, **fájlkezelőnek** nevezzük. (Ez nem tévesztendő össze a gyakran azonos nevű felhasználói programmal.) A fájlkezelő az a folyamat, mely

a nevekből a logikai blokkszámokat kialakítva a felhasználói folyamatok kéréseit az eszközvezérlőnek továbbítja.



3.1 ábra Felhasználói folyamatok kiszolgálása

3.1 Fájlnevek

A felhasználói folyamatok fájlnevek segítségével hivatkoznak a kívánt adatcsoportokra. Ezeket a neveket általában maguk a létrehozó folyamatok adják, de előfordulhat az is – ideiglenes állományok esetén –, hogy a folyamat az operációs rendszert kéri fel névadásra, mely a többi elnevezés ismeretében ügyel az egyediségre.

A fájlneveket alkotó karakterek lehetséges halmaza, a név lehetséges hossza, valamint szerkezete az aktuális operációs rendszer függvénye. A nevek több összetevőből állhatnak, melyeket valamilyen speciális karakter választ el egymástól. A leggyakoribb a két összetevőből álló név, ahol az első az egyedi, a fájl tartalmára utaló rész, a második rész a fájl jellegére (szöveg, program, adatbázis) utal. Az elnevezésekre vonatkozó szabályok rendkívül változatosak, nézzünk néhány példát!

Az **MS-DOS**, a személyi számítógépeken klasszikusnak számító operációs rendszer nagyon szigorú szabályokat követel meg. A fájlnév két komponensből áll, az első rész, a név, minimum 1, maximum 8 karakterből állhat, az elválasztó pont után következő kiterjesztés legfeljebb 3 karakterből állhat (itt alsó határ nincs, azaz el is maradhat). A fájlneveket alkotó karakterek az ASCII kódtábla nagybetűi, számai és

egyes speciális karakterei (pl. ~!@#$%^&_-{}) közül kerülhetnek ki. A DOS nem tesz különbséget kis- és nagybetűk között, az operációs rendszer minden karaktert automatikusan nagybetűvé konvertál. Az újabb DOS változatok megengedik ugyan ékezetes karakterek használatát is, de ezek használata az egyes gépek eltérő konfigurációja, valamint az említett karakter konverzió miatt hord némi veszélyeket magában.

A Windows 95 speciális kettős elnevezésrendszert használ, részben a dokumentum-orientált szemlélet, részben a DOS kompatibilitás miatt. A Windows 95-ben minden fájlnak két neve van, egy rövid, és egy hosszú. A rövid fájlnevek követik a hagyományos DOS konvencióit, azaz 8+3 karakteres felépítésűek. Minden fájlhoz tartozik azonban egy legfeljebb 250 karakter hosszúságú, tetszőleges karakterekből álló név is, ugyancsak maximum három karakteres kiterjesztéssel. A Windows 95 alkalmazások ezt a hosszú nevet látják, ebből képezik a rövid nevet. Az első nyolc karakter automatikus választása azonban nem ad kielégítő megoldást, hiszen lehet sok olyan hosszú név is, melynek eleje megegyezik. A Windows 95 úgy oldja meg ezt a kérdést, hogy a hosszú fájlnévből eltávolítja szóköz karaktereket, majd az első 6 karakter után egy elválasztó jelet (~), azután egy sorszámot ad.

A **Unix** fájlnevek hossza maximum 255 karakter lehet, és tetszőleges számú, a DOS-hoz hasonlóan *ponttal* elválasztott összetevőből állhatnak. A Unix megkülönbözteti a kis- és nagybetűket, így a `FILE`, a `file` és a `File` három különböző állományt jelöl. A speciális, vagy ékezetes karakterek használatára nagyjából a DOS szabályai érvényesek, annak veszélyeivel együtt, azaz nem szerepelhetnek olyan karakterek, melyek az operációs rendszer számára valamilyen speciális jelentéssel bírnak.

A **Windows NT** alapú rendszerek a Unix-hoz hasonlóan alkalmasak a hosszú fájlnevek használatára. A Unicode alkalmazása miatt a speciális karakterek mellett az ékezetes betűk is többé-kevésbé veszélytelenül szerepelhetnek. A DOS-szal való kompatibilitás igénye miatt azonban a kisbetű-nagybetű különbség (bár a megjelenésben látható), de mégsem következetes. Szintén a DOS-os hagyomány eredménye lehet, hogy az állományhoz a fájlnév utolsó szekciójában (azaz az utolsó pont után) lévő karaktersort az operációs rendszer kiterjesztésnek tekinti (és általában nem is jeleníti meg), és ez alapján társít alkalmazást az állományhoz. (A kiterjesztés elrejtése és a társítás együtt néha igen veszélyes. A vírusok gyakran használják álcázásként a kettős kiterjesztést (pl. worm.txt.exe).

A korábbi operációs rendszerek némelyike (például a DEC VMS) verziókövetést is lehetővé tett. A fájlnév és a kiterjesztés után pontosvesszővel elválasztva jelent meg a verzió (pl. `RAJZ.DOC;6`). Ez a szolgáltatás ismét kezd fontossá válni, azonban az operációs rendszerek támogatása híján alrendszerekben jelenik csak meg.

Látható, hogy az ismertetett elnevezési szabályok közül a legegyszerűbb a DOS szabályrendszere. Az úgynevezett „8+3" szabály alkalmazása kényelmetlen (nem könnyű olyan nevet adni, amely kellő részletességgel kifejezi a tartalmat), azonban kétségtelen előny, hogy minden operációs rendszer „érti". Ha biztosak akarunk lenni abban, hogy a publikálásra szánt fájlunkat az internetre kapcsolt, a legkülönbözőbb operációs rendszerekkel működtetett számítógép felhasználója láthassa, érdemes önkorlátozással élni, és a jó öreg „8+3" konvenciót alkalmazni.

A fájlokra a felhasználói folyamatok általában a pontos nevük alapján hivatkoznak, de lehetséges a név egy részlete alapján is keresni egy, esetleg több, a mintára illeszkedő állományt. A többféle tartalommal is kitölthető, úgynevezett helyettesítő karakterekre (egyéb elnevezésük wildcard, joker, metakarakter) példa a DOS esetén a '?' mely tetszőleges karakter egyetlen előfordulását jelenti, valamint a '*', mely tetszőleges tartalmú és hosszúságú karakterlánc helyett állhat. A Unix a fenti lehetőségeken kívül ismeri a karaktercsoport lehetőségét is, melynek lehetséges értékeit szögletes zárójelek között kell megadni. Az '[ABC]' kifejezés helyett például állhat egy 'A', egy 'B' vagy egy 'C' karakter.

3.2 Fájlok jellemzői

A fájlokhoz a nevükön kívül egyéb információk is tartoznak, melyeket részben az operációs rendszer ad, részben a felhasználó. A többlet adatok a különböző operációs rendszerekben a fájlnevekhez hasonló nagy változatosságot mutatnak.

Az **utolsó módosítás időpontja** mindig szerepel az adatok között. Újonnan alkotott, még nem módosított fájl esetén ez a paraméter természetesen megfelel a létrehozás időpontjának. A **fájl mérete** szintén nagyon fontos információ, általában bájtokban adják meg. Több felhasználós rendszerek esetén rögzítésre kerül a fájl **tulajdonosa** is.

A fájl állapotára, tulajdonságaira utaló jelzőbiteket összefoglaló néven **attribútumoknak** nevezzük. (Az alábbiakban a DOS és a Unix által

használt jellemzők egyvelegét ismertetjük, melyek együtt talán képet adhatnak a lehetséges funkciókról.) Az attribútumok jelezhetik az archiválást végző program számára, ha egy fájl megváltozott az utolsó mentés óta, azaz *archiválandó* (archive needed), az operációs rendszer többi folyamata számára, hogy a fájl *csak olvasható* (read only), *rendszerfájl* (system), *rejtett állomány* (hidden). A speciális tulajdonságú, adminisztrációs fájlokat is attribútumok különböztetik meg a felhasználói állományoktól, különböző jelzőbitjei vannak a *katalógusok*nak (directory) (amiket nemsokára tárgyalunk), a *szimbolikus hivatkozások*nak (link), az ideiglenes, *adatcsere fájlok*nak (pipe).

Néha (például a Unix esetében) a felhasználók **hozzáférési jogait** is a fájlok jellemzői között találjuk, azaz itt kerül szabályozásra, hogy ki írhatja, olvashatja az állományt, vagy – programfájl esetén – ki hajthatja vére azt. A hozzáférési jogosultságokat tartalmazó információ gyakran a felhasználókhoz rendelődik, vagy külön rendszerállományt alkot, mint azt a későbbiekben láthatjuk.

Az attribútumok és a hozzáférési jogok között a leglényegesebb különbség az, hogy míg az attribútum az állomány önálló jellemzője, a hozzáférési jog egy felhasználó vagy felhasználói csoport és egy fájl viszonyát határozza meg.

Nem említettük eddig a fájlnéven kívüli legfontosabb, a **fájl fizikai elhelyezkedésére** vonatkozó információt, melyet fontossága miatt külön pontban tárgyalunk.

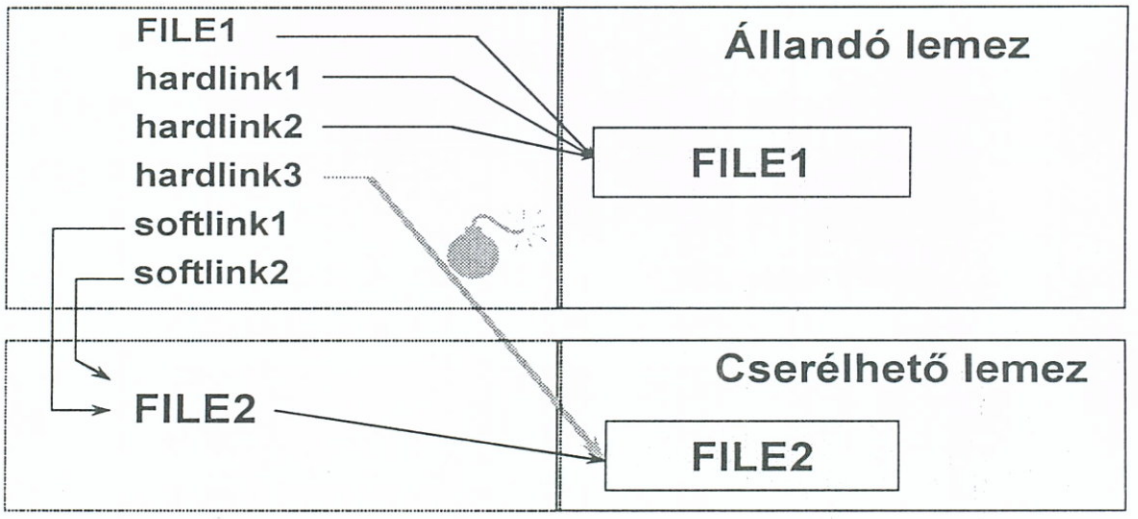
A legfontosabb állomány jellemzők tehát:

- Fájlnév
- Fájlméret
- Létrehozás ideje
- Utolsó módosítás (hozzáférés, nyomtatás) ideje
- Attribútumok
- Tulajdonos
- Hozzáférési jogok
- Fizikai elhelyezkedés

3.3 Közvetett hivatkozások

Közvetett hivatkozásokról (láncolás, link, alias) akkor beszélünk, ha egy fájlhoz nem csak egy elnevezés segítségével juthatunk el, tehát például egy programot a különböző felhasználók különböző neveken érhetnek el. A módszert a Unix valósítja meg következetesen, de a szemlélet nem idegen a Windows-tól sem.

A közvetett hivatkozások használatának szembeötlő előnye a helytakarékosság, teljes másolatok helyet csak a nevek szaporodnak. A másik előny, hogy a fájlok többféleképpen csoportosíthatók, egy fájl több katalógusban is megjelenhet, és a fizikai állomány változása azonnal érvényesül minden csoportban.



3.2 ábra Fájlok láncolása

A **merev láncolás** (hard link) a hivatkozott fájl fizikai címét tartalmazza. A mereven láncolt állományok használata jelentős többlet feladat az operációs rendszer számára. Nyilván kell tartania, hogy egy-egy fájl fizikai megvalósulására hány hivatkozás mutat, és csak akkor szabad a fájlt törölni, amikor már egy sem mutat rá. Merev láncolás esetén, mivel a fájlnevek ugyanarra a fizikai lemezcímre mutatnak, a hivatkozó fájloknak természetesen, ugyanazon az eszközön kell lenniük. Merev láncolás esetén a hivatkozott (fizikai) állomány elmozdítása (például lemeztömörítés esetén) a hivatkozásokat működésképtelenné teszi.

A **lágy láncolás** (soft link) a fizikai cím helyett a hivatkozott fájl nevét tartalmazza, lehetővé téve azt, hogy a fájl akárhol (például egy hordozható eszközön, floppyn) is előfordulhasson. A lágy láncolás további előnye, hogy az adatállományok tömörítése, vagy alapvető,

fizikai címeket is érintő átstrukturálása esetén is működőképes marad. Lágy láncolásnak tekinthető például a Windows parancsikonja.

Az indirekt hivatkozás bevezetése adminisztratív problémákat is felvet. Kié a fájl? Kinek kell fizetnie a hely foglalásáért, ki törölheti azt?

3.4 Katalógusok (directory)

A katalógus olyan speciális, adminisztratív célokat szolgáló fájl, amely a lemezen lévő fájlok adatainak listáját tartalmazza. Magát a katalógusfájlt a felhasználók általában közvetlenül nem láthatják, tartalmukat rendszerhívások segítségével érhetjük el.

§

Katalógus (könyvtár, directory)
Olyan speciális állomány, melynek tartalma a fájlok nevét és jellemzőit tartalmazó rekordok listája

(Az eredeti angol elnevezést, a *directory*-t magyarul gyakran *könyvtár*nak is nevezik, de tartalmilag helyesebb, és a függvény- vagy szubrutin könyvtáraktól való megkülönböztetést is jobban szolgálja a *katalógus* szó.)

Ha egy folyamat egy fájlra hivatkozik, az operációs rendszer először a katalógust vizsgálja meg, hogy létezik-e egyáltalán ilyen fájl. Pozitív válasz esetén jön a következő ellenőrzés, hogy a felhasználónak, illetve az általa indított folyamatnak van-e joga a kívánt művelethez. Amennyiben a fájl is létezik, és a jogosultságok is rendben vannak, akkor kezdődhet meg a katalógusban lévő, a fizikai elhelyezkedésre utaló információ alapján a művelet végrehajtása. (Megjegyzendő, hogy Unix alapú rendszereknél a hozzáférési jog megléte a katalógusinformáció alapján általában még nem állapítható meg, tehát ha a katalógus olvasásához van joga, az állomány neve annak a számára is látható, akinek semmiféle joga nincs az adott állományhoz.)

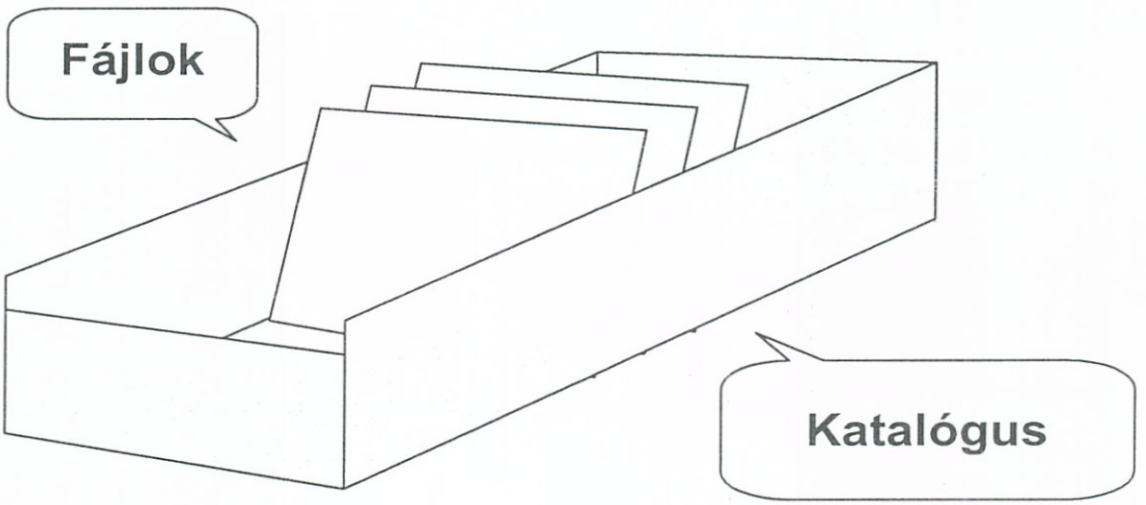
3.4.1 Katalógus nélkül

Soros hozzáférésű média, például mágnesszalag esetén, ahol a blokkok mérete sem állandó, nehéz elképzelni könnyen kezelhető, a fájlok alapvető adatait tartalmazó katalógus állományt. Igaziból azonban nincs

is szükség rá. Adatainkhoz úgyis csak az azt megelőző adatok végigolvasásán keresztül, tehát viszonylag lassan juthatunk. Az elérési idő javítása érdekében a szalagok a fájlok között szüneteket (inter file gap) tartalmaznak. Az adatok teljes hiányát az olvasófej gyors tekercselés közben is felismeri, kicsit lassíthat, amíg megállapítja, hogy elérte-e már célját, és ha nem, szaladhat tovább.

3.4.2 Egyszintű katalógus

Ha egy rendszer csak egyetlen, rendszerszintű katalógust kezel (manapság ez már kuriózum), egyszintű katalógusról beszélünk.



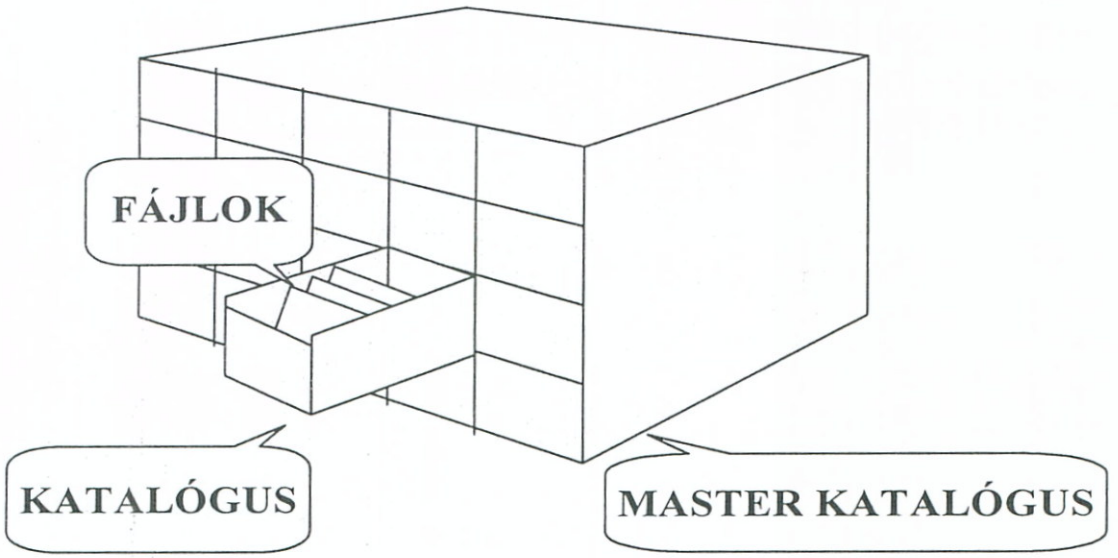
3.3 ábra Egyszintű katalógus

Az egyszintű rendszerekkel az a legnagyobb baj, hogy minden fájlnak a megkülönböztethetőség miatt teljesen egyedi neve kell, hogy legyen. A szigorú fájlnév képzési szabályok megkeseríthetik a felhasználók életét. Ha azonban több komponensből álló, hosszú nevek is adhatók, a helyzet jelentősen javul.

További hátrány, hogy egy-egy fájl megkereséséhez legrosszabb esetben az egész katalógust végig kell olvasni. Ezen a katalógus valamilyen célszerű rendezésével lehet javítani. A fájlnevek szerint névsorba rendezett katalógusokban az igen hatékony bináris keresés alkalmazható, de használatos az a módszer is, hogy a legutóbb használt fájlok mindig a katalógus elejére kerülnek, hátha legközelebb is ezeket keresik.

3.4.3 Kétszintű katalógus

Egy második szint bevezetése sokkal áttekinthetőbb rendszert eredményez, a legtöbb feladat így már megoldható. Minden felhasználó kaphat egy saját katalógust, míg a közösen használt fájlok külön katalógusba kerülhetnek. Az egyes katalógusokat, a katalógusok katalógusa, a *fő* (master) vagy más néven *gyökér* (root) katalógus fogja össze. Mivel ebből eszközönként csak egyetlen egy van, nem is szükséges neki nevet adni.



3.4 ábra Kétszintű katalógus

Az egyes szinteket egymástól, illetve a fájlnévtől a hivatkozásokban általában egy '\' vagy '/' karakter választja el. A *nagy* nevű katalógusban található *diploma.doc* nevű fájl teljes elérési útja a következő:

/nagy/diploma.doc

A névtelen gyökér katalógust követi az első törtvonal, következik a katalógus neve, majd újabb törtvonal után a fájl neve. A fenti hivatkozás egyértelmű, de a 'nagy' nevű katalógusban csak egyetlen 'diploma.doc' nevű fájl lehet. Ha más felhasználó is a diplomamunkáján dolgozik, ugyanazon a számítógépen, illetve lemezen, saját katalógusában szintén adhatja a 'diploma.doc' nevet, például

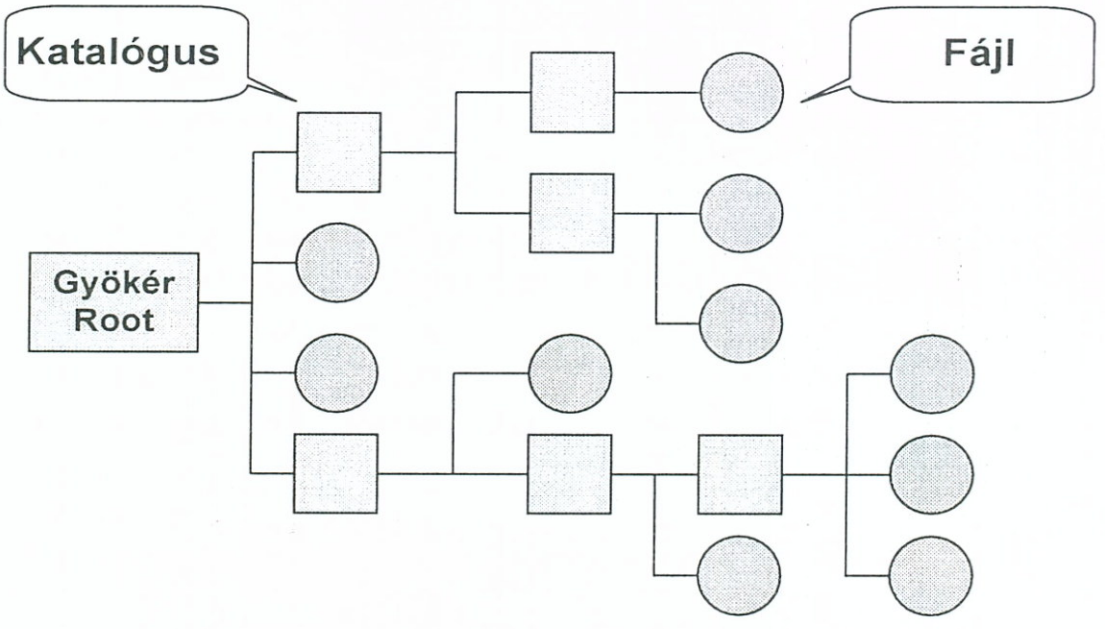
/papp/diploma.doc

A fenti példák szerinti hivatkozás működik, *abszolút* pontos, a két felhasználónak elvileg egymáshoz semmi köze, lehet, hogy nem is ismerik egymást. Nem kell tudniuk, egymás fájlneveiről, sőt általában

nincs is joguk más katalógusában nézelődni. Miért követeljük hát meg szegény felhasználóktól, hogy két törtvonal között minden egyes fájl hivatkozásnál leírják a nevüket?

3.4.4 Többszintű (hierarchikus) fájl rendszer

A kétszintű rendszerről nem nehéz általánosítani a többszintű, többnyire fa struktúrát alkalmazó rendszerekre. A hierarchikus rendszer kiindulópontja a gyökér (root), mely tartalmazhat fájlokat és alkatalógusokat (subdirectory), ez utóbbiak szintén tartalmazhatnak fájlokat és alkatalógusokat és így tovább. Az elrendezés hasonlít egy fejre állított fára (innen az elnevezés), melynek gyökere a gyökérkönyvtár, ágai az alkatalógusok, levelei a fájlok. A hierarchikus felépítés a fájlok keresésének idejére is kedvező hatással van, mivel a lineáris rendszerekkel szemben itt alkalmazható a lényegesen gyorsabb bináris keresés.



3.5 ábra Fa struktúrájú katalógus

Mivel az egymásba ágyazott katalógusok száma elvileg nem korlátozott (a gyakorlatban persze igen) a kétszintű katalógusnál felvetődött, a hivatkozás egyszerűsítésére vonatkozó kérdések itt még fokozottabban jelentkeznek. Az eddig alkalmazott hivatkozási módszer a legalsó szinttől indult:

§

Abszolút hivatkozás

A fájl megadásának az a módszere, ahol a **gyökér** katalógustól kezdődően az összes közbülső katalógus nevének felsorolása után jutunk el a fájlhoz

Az aktuális vagy munkakatalógus (current/working directory) fogalmának bevezetésével egyszerűsíthető a hivatkozás, az eddigi, a gyökérkatalógustól induló *abszolút címzéssel* szemben alkalmazható az aktuális katalógushoz, illetve aktuális meghajtóhoz viszonyított *relatív címzés*.

§

Relatív hivatkozás

A fájl megadásának az a módszere, ahol a gyökér katalógus helyett az aktuális katalógus a kiinduló pont.

Az abszolút és relatív hivatkozás között a legszembetűnőbb formai különbség, hogy az abszolút hivatkozás mindig egy elválasztó karakterrel, jelen esetben törtvonallal kezdődik.

Fa struktúrát alkalmazó rendszerekben az egyes katalógusoknak lehetnek szülei és gyermekei. A szülőkkel a szorosabb kapcsolat természetes, ezért igazi nevén felül, már csak szülői mivoltából is indokolt külön névvel illetni. A szülő katalógus neve általában (a gyermek szemszögéből) '..' (azaz két darab pont), az aktuális katalógus relatív neve '.' (azaz egy darab pont).

3.5 Hozzáférési jogok

Az egyes fájlok tartalma lehet bizalmas, vagy fontos, nem módosítható. Ezen felül nem is lenne helyes, ha egy nagyobb, többfelhasználós rendszerben mindenki minden fájlt láthatna, kénye–kedve szerint törölhetne vagy módosíthatna, sőt felhasználók többségét kifejezetten zavarná, ha az óriási fájl dzsungelből kellene mindig kikeresnie a saját adatait. A fájlokhoz való hozzáférést tehát szabályozni kell. (A továbbiakban a legkisebb egységnek a fájlt tekintjük, de megjegyzendő,

hogy adatbázisok esetén szükség lehet rekord vagy mezőszintű szabályozásra is.)

A felhasználói jogosultságok szemléltetésénél alkalmazott modell egyszerű. Vannak:

- **Felhasználók (vagy Csoportok),** akik valamiféle fájl elérési lehetőséggel rendelkeznek, vagy éppen nem rendelkeznek
- **Állományok**, amelyek használatát szabályozni kívánjuk
- **Hozzáférési jogosultságok,** amelyek megszabják, hogy egy-egy felhasználó mit kezdhet egy fájllal

3.5.1 Hozzáférési jogok típusai

A legutóbbi pont némi magyarázatra szorul. Egyáltalán milyen hozzáférési jogosultságok lehetnek? A felhasználói jogok rendszere, de legalábbis elnevezésük operációs rendszerenként változnak, azonban az alábbi – leginkább a NetWare jogosultságaira emlékeztető – lista ismeretében mindegyikben könnyű lesz eligazodni.

- **Olvasás** (**R**ead - **R**) Az olvasási joggal rendelkező felhasználó a fájl tartalmát megtekintheti, beolvashatja, de nem módosíthatja, nem törölheti.
- **Írás** (**W**rite - **W**) A már létező fájl módosítható, de nem törölhető. Általában értelme az olvasási joggal együtt van igazán, de elképzelhető, hogy egy felhasználó számára csak egy dokumentumhoz való hozzáfűzés engedélyezett az eredeti tartalom ismerete nélkül.
- **Létrehozás** (**C**reate - **C**) Létrehozható egy fájl, és természetesen akkor írhatunk is bele, de csak egyszer, a pillanat megismételhetetlen. Másodszori megnyitás egyéb jogok hiányában nem lehetséges. Ilyen jogosultsággal rendelkeznek a felhasználók például egymás postafiókjában. Ha már bedobtuk a levelet a postaládába, hiába kapkodunk a fejünkhöz.
- **Végrehajtás** (e**X**ecute - **X**) A fájl, illetve helyesebben a program az operatív memóriába tölthető, és futtatható. Olvasási jog hiányában másolásra nincs lehetőség. A jogosultság helyes alkalmazása megoldást kínálhat szerzői jogi problémákra is.

- **Törlés** (**E**rase - **E**) A fájl a katalógusból törölhető. A hangsúly ez esetben a katalógus szón van, ugyanis ez nem jelenti feltétlenül azt, hogy az adatok fizikailag is megsemmisülnek, a tartalom végérvényesen elveszett. A legtöbb operációs rendszer rendelkezik olyan lehetőséggel, mely lehetővé teszi a véletlenül törölt fájlok visszaállítását.
- **Jellemzők módosítása** (**M**odify - **M**) A fájl neve, létrehozásának időpontja, tulajdonosa megváltoztatható, új értéket kaphatnak az attribútumok. Jellemzők módosításának joga önmagában még nem elegendő a hozzáférési jogok változtatására.
- **Hozzáférés módosítása** (**A**ccess control - **A**) Az e joggal rendelkező felhasználó az adott fájlra nézve beállíthatja, vagy törölheti az eddig felsorolt jogosultságokat a többi felhasználó, vagy akár önmaga számára.

Általában a lehetőségek együttes szabályozását is megengedik az operációs rendszerek. Az összes jog beállítására (ALL) és törlésére (-ALL) is lehetőség van.

Bár ezt eddig nem emeltük ki, a jogosultságok értelemszerűen vonatkozhatnak fájlokra és katalógusokra egyaránt, azonban bizonyos korlátokkal. A *Létrehozás* természetesen nem lehet fájlra vonatkozó jog, hiszen az még nem is létezik, a végrehajtás pedig értelmetlen, ha katalógus állományról van szó.

3.5.2 Jogok nyilvántartása

A felhasználókat és a fájlokat tehát a jogosultságok kapcsolják össze. A nyilvántartásnak lehetőség szerint olyannak kell lennie, mely nagyon gyors keresést tesz lehetővé, hiszen erre minden egyes fájlhivatkozásnál szükség van.

Elvileg az a legegyszerűbb, ha a felhasználók adatait tartalmazó adatbázisban helyezzük el azon állományok és a hozzájuk kapcsolódó jogok listáját, amelyekhez az adott felhasználó hozzáférhet. A módszer azonban a gyakorlatban alkalmazhatatlan, hiszen egy új állomány létrehozójának kellene rendelkeznie a hozzáférési jogok beállításáról, tehát más felhasználók adatainak módosításáról, amihez pedig nem lenne szerencsés mindenkinek lehetőséget adni.

Hasonlóan egyszerű módszer, ha az állományok adatai között tartjuk nyilván a jogosultsággal rendelkező felhasználók, illetve csoportok azonosítóit, illetve jogait. Ilyen rendszert használ például a Unix. A módszer hátránya, hogy nem lehet kifinomult hozzáférési jogrendszert alkalmazni, hiszen akkor az állományok jellemzőinek listája lenne túl hosszú, ezáltal a hozzáférés sebessége lassulna.

A leggyakrabban alkalmazott eljárás az, amikor a hozzáférési jogokat külön állományban (pontosabban adatbázisban) tárolják (**A**ccess **C**ontrol **L**ist – **ACL**). Ez az adatbázis minden hozzáférési kísérletnél könnyen lekérdezhető, és csak azokról a felhasználó–állomány viszonyokról kell információt tartalmaznia, amelyek eltérnek a rendszer alapértelmezett beállításaitól.

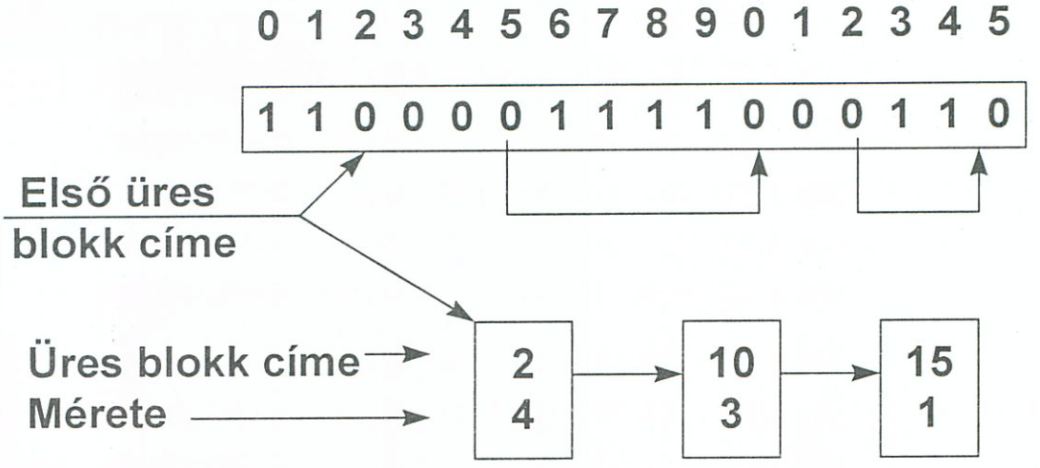
3.6 Fájlok elhelyezése

A fájlok adatai között szerepelt egy paraméter, ami arra utalt, hogy fizikailag hol található maga a fájl a lemezen. A lemezeken az alapértelmezett tárolási egységet **blokknak** nevezzük. Egy blokk mérete néhány száz bájt (tipikusan 512 vagy 1024). A meghajtó egy blokknál kisebb egységgel nem képes műveletet végezni. Egyes operációs rendszerek (például a DOS, és az első Windows változatok) még nagyobb egységekkel, **klaszterekkel** dolgoznak, melyek mérete 1..64 blokk is lehet. A továbbiakban – az általánosság megszorítása nélkül – blokkokban számolunk.

Amikor egy folyamat egy új fájl létrehozását határozza el, általában a lemezen már vannak fájlok, és a foglalt blokkok között kisebb-nagyobb üres területek. Az elkészítendő fájl mérete általában nagyobb, mint egy blokk, tehát a tárolásra több blokkra is szükség van. Az operációs rendszer feladata, hogy megpróbálja kiválasztani a lehető legjobb elhelyezési módot és a későbbi hozzáférés lehetősége érdekében a fájl adatai között, a katalógusban tárolja a visszakereséshez szükséges információt.

Az egyes blokkok foglalt vagy szabad állapotát legegyszerűbb esetben az úgynevezett *foglaltsági tábla* mutatja, melyet az operációs rendszer a lemez katalógusainak, egyéb adminisztrációs állományainak felhasználásával állít elő és tart karban. Ennek a táblázatnak annyi eleme van, ahány blokkja a lemeznek, és az egyes blokkok betöltöttségétől

függően egyeseket vagy nullákat tartalmaz. Célszerű, gyorsabb keresést tesz lehetővé, ha az egyszerű bittérkép mellett vagy helyett egy, a szabad helyek méretét és kezdőcímét tartalmazó láncolt listát is készítünk, és természetesen a bittérképpel együtt folyamatosan karbantartjuk.



3.6 ábra Szabad helyek nyilvántartása

Az operációs rendszer fájlkezelője tehát utasítást kap, hogy egy adott méretű állományt helyezzen el. Az elhelyezésre, illetve az ezzel kapcsolatos információ megadásának módjára háromféle technikát ismertetünk.

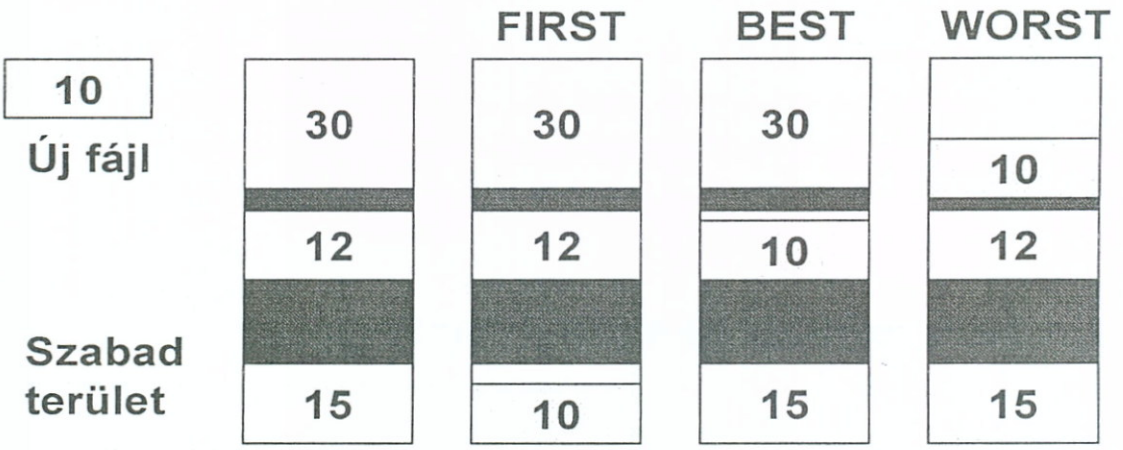
3.6.1 Folytonos kiosztás

Legegyszerűbb, és a későbbi használat szempontjából is kedvező, ha a fájl blokkjai folytonosan helyezkednek el, így a leggyorsabb a beolvasás és az írás. Az operációs rendszer a foglaltsági tábla alapján felderíti, hogy mely szabad területek alkalmasak az állomány befogadására. Gyakran több ilyen terület is van, dönteni kell.

1. A **legelső alkalmas** (First Fit) hely felhasználása a leggyorsabb ugyan, de nem mindig kedvező. Lehet, hogy az éppen elhelyezendő fájl számára, ha szűkebben is, de máshol is lett volna hely, és egy következő állomány egyetlen lehetőségét vettük el.
2. Megkereshetjük azt a szabad tartományt, melynek mérete csak minimálisan haladja meg az állomány méretét, azaz a fájl a **legjobban illeszkedik** (Best Fit). A módszer hátránya, hogy számításigényes,

végig kell nézni az összes szabad helyet, mielőtt a döntésre sor kerülne.

3. A létező legnagyobb helyre is gondolhatunk, melyben az elhelyezett állomány mellett a legtöbb szabad hely marad, azaz a **legrosszabbul illeszkedik** (Worst Fit). E mellett az elhelyezés mellett a legnagyobb az esélye annak, hogy egy újabb fájl is befér majd a fennmaradó szabad blokkokba, ugyanakkor viszont előbb-utóbb elfogynak a nagy helyek. Ez utóbbi miatt általában ez a stratégia adja a legrosszabb helykihasználást, míg – ha a fájlok véletlenszerűen „keletkeznek" - a Firs Fit és a Best Fit stratégia hatékonysága közel azonos, de a First Fit gyorsabb.



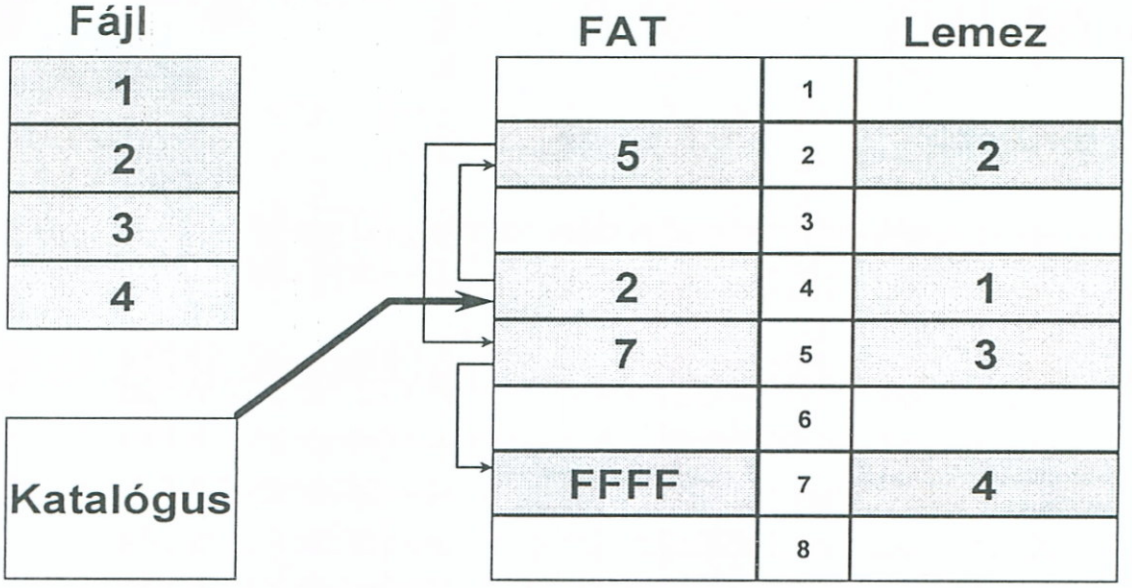
3.7 ábra Állományok folytonos elhelyezése

Bármelyik módszert választjuk, ha ragaszkodunk a folytonos elhelyezéshez, óhatatlanul szétdarabolódik a rendelkezésre álló, még szabad hely. Könnyen előfordulhat, hogy, bár összességében van elegendő üres blokk a lemezen, újabb fájl mégsem helyezhető el, mivel a szabad helyek nem alkotnak kellő méretű összefüggő tartományt. Ezen lehet segíteni a lemez rendezésével, töredezettségének megszüntetésével (defragmentálás, garbage collection), de ez meglehetősen időigényes munka, másrészt a merev láncolást is tönkreteszi. A másik hátrány, hogy nem ismerhető előre a fájl végleges mérete. Túl precíz illeszkedés esetén a fájl méretének minimális növekedése is csak az egész állomány áthelyezésével lehetséges (ha egyáltalán lehetséges). További nagy probléma, hogy ha a fájl belsejéből ki akarunk törölni egy blokkot, az azzal jár, hogy az összes maradék blokkot egy blokkal előre kell mozgatni.

A fájl helyének nyilvántartása viszont egyszerű, olvasása gyors. Mindössze a kezdő blokk sorszámát kell megadni a katalógusban, a blokkok száma már számítható a fájl méretéből.

3.6.2 Láncolt elhelyezés

Ha lemondunk az egyszerű nyilvántartásról, és egy újabb táblázatot is igénybe veszünk, sokkal rugalmasabb módszerhez jutunk. A katalógusban itt is csak a fájl kezdő blokkjának címét kell megadni, az összes többi adatot a fájl elhelyezési tábla (**F**ile **A**llocation **T**able – **FAT**) tartalmazza. A táblázatnak ugyanannyi eleme van, mint ahány blokk a lemezen, és egy-egy elem mérete akkora, hogy lehetővé teszi az összes blokk megcímzését (például 256 blokk esetén 8 bites cím elegendő, de 4 milliárd blokk esetén már 32 bit szükséges). Minden rekesz tartalma a fájl következő blokkjára mutató sorszám, ha van következő blokk, FFFF (azaz csupa egyes), ha ez volt az utolsó blokk.



3.8 ábra Láncolt elhelyezés

Az eljárás mentes a folytonos elhelyezésnél tapasztalható töredezés veszélyétől. A szabad helyek az utolsó blokkig kihasználhatók, a fájlok mérete a lemez fizikai határáig növekedhet. Az üres helyek keresésére sem kell időt és energiát szánni, az első szabad blokknál lehet kezdeni. Ennél a módszernél nem probléma a fájl méretének növelése, illetve csökkentése.

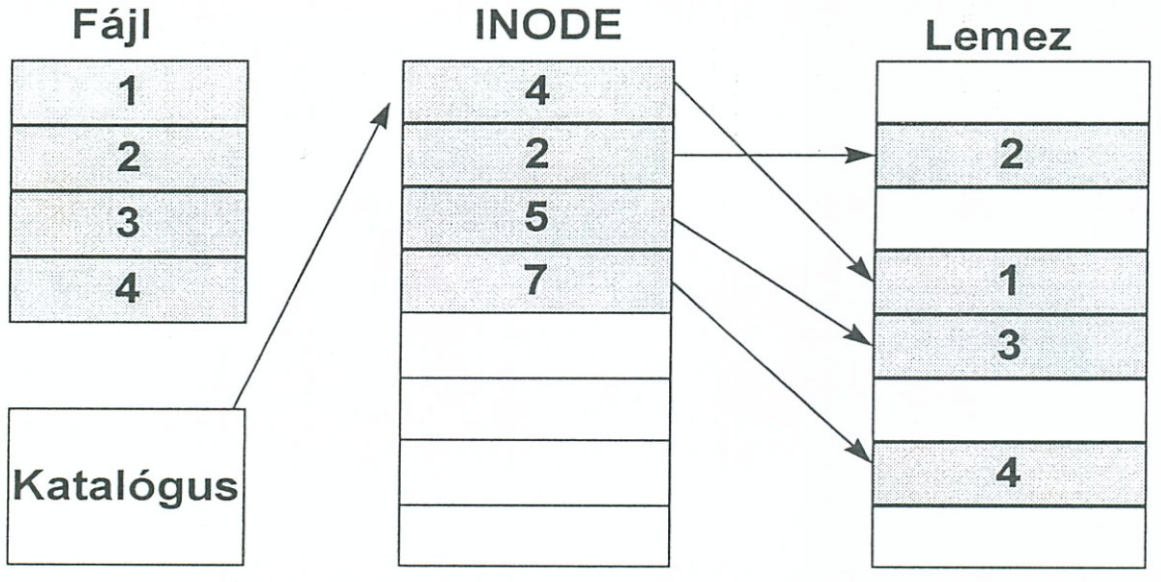
A FAT meglehetősen nagy táblázat lehet, és szerepe a biztonságos tárolásban döntő. A FAT sérülése esetén nagy valószínűséggel a kettészakadt fájlt visszaállítani nem lehet. A láncolási módszert alkalmazó operációs rendszerek, például a DOS, a Windows és a NetWare a biztonság kedvéért két ilyen táblázatot tartanak fenn. Mindkét táblázatot ellenőrző összeggel látják el annak érdekében, hogy az esetleges sérülés azonnal észrevehető legyen. A kisebb hibák már az ellenőrző összeg segítségével javíthatók, de ha nagyobb a baj, még akkor is rendelkezésre áll a másolat.

A FAT további korlátja, hogy csak a szekvenciális hozzáférést támogatja, azaz, ha például egy fájl 10. blokkját akarjuk elérni, ahhoz végig kell követni az első 9 blokk helyét, ami lassú lehet.

Régebben komoly hátrány volt a megcímezhető merevlemez méretének korlátos volta is, azonban ez a 32 bites FAT (és az ennek kezelését megfelelő sebességgel ellátni képes feldolgozó kapacitás) kialakításával gyakorlatilag megszűnt.

3.6.3 Indextábla alkalmazása

Rugalmas elhelyezési lehetőségekhez jutunk, ha egy óriási táblázat, a FAT helyett sok kicsit használunk, minden állományhoz külön-külön egyet. A katalógus tartalmazza a fájlhoz tartozó kicsi táblázat, az **indextábla** (i-node) címét, az indextábla pedig az adott fájl blokkjainak a címét.

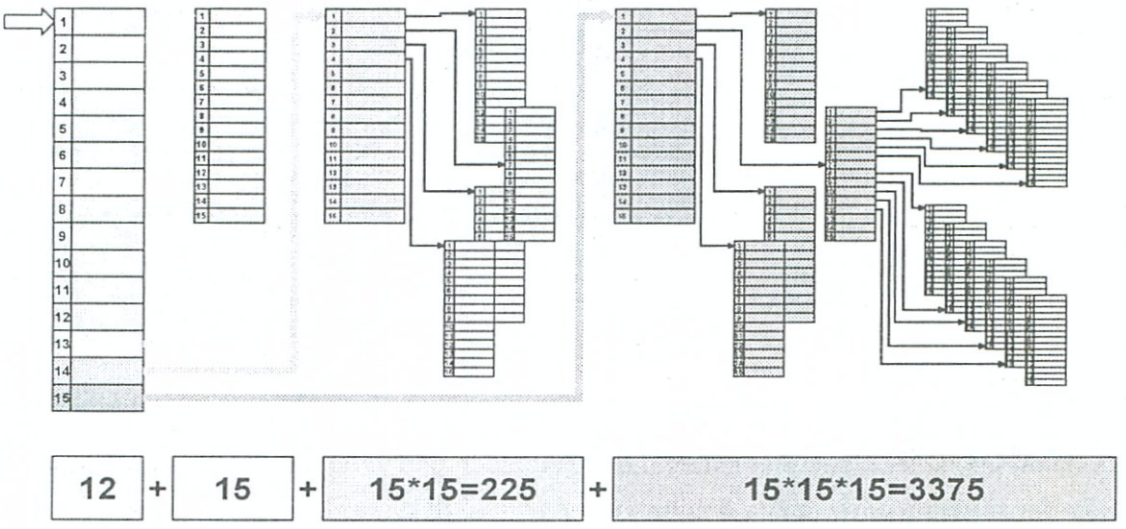


3.9 ábra Indexelt elhelyezés

A módszer előnye a töredezettség elkerülése mellett az, hogy az elhelyezési információ gyorsan elérhető, kevésbé sérülékeny a tárolás is. Hátránya, hogy valamiféle becslésre van szükség arra nézve, hogy mekkora legyen az indextábla, azaz mekkorák lesznek fájlok? Ha túlságosan nagy a tábla, pazarló, ha túlságosan kicsi, akkor ez határozná meg a maximális fájlméretet, ami megengedhetetlen. A Unix operációs rendszer kombinált módszer alkalmazásával oldotta meg ezt a problémát.

A katalógus csak egy címet, egy indextábla címét tartalmazza, amely 15 címzésre szolgáló rekeszből áll. (Ezen felül természetesen ott vannak az egyéb állomány jellemzők, jogok tárolására szolgáló mezők is.) Az első 12 rekesz a fájl első 12 blokkjának sorszáma. Amennyiben ez kevésnek bizonyulna, a 13, rekesz egy újabb indexre mutat, mely további 15 blokk nyilvántartását teszi lehetővé. Ha a fájlunk még 27 blokknál is nagyobb, az első indextábla 14. rekesze egy indirekt indextáblát címez, amelynek tartalma további 15 indextábla címe, és így tovább...

Az i-node-ot alkalmazó fájlrendszerek nagy fájlok blokkjainak elhelyezését tehát több tábla szinttel írják le. Az elsődleges, másodlagos és harmadlagos táblák a Linux esetén (az előző példabeli 15-tel szemben) 256 eleműek, így, ha kihasználják az összes lehetőséget, akkor 1 kB-os blokkméret esetén maximálisan 2^{24} * 1kB= 16GB méretű fájlt képesek tárolni és még ilyen nagy fájlok esetében is csak három táblázatban kell keresni, hogy egy blokkot megtaláljunk.



3.10 ábra Indexelt elhelyezés nagy állományoknál

Az indexelt leképzés egyik további előnye, hogy lehetővé teszi a közvetlen elérést is, azaz ha most a fájl 10. blokkját akarjuk elérni, annak a címét egyszerűen megtudhatjuk az indextábla 10. sorából.

Megjegyzendő, hogy a korszerű fájlrendszerek (például az NTFS) az állományok elhelyezésénél szintén az i-node technikát használják.

3.7 Műveletek állományokkal, katalógusokkal

Az állományok, katalógusok felépítésének megismerése után összeállíthatjuk a rendszermag megfelelő szolgáltatásainak rendszerét. A műveleteket a folyamatok, illetve a programozók szempontjából célszerű csoportosítani.

- **Állomány létrehozása**: a katalógusba új bejegyzés kerül, az új állomány számára az operációs rendszer megfelelő mennyiségű szabad blokkot keres.
- **Katalógus létrehozása**: különleges státuszát mutató attribútumán kívül semmiben sem különbözik a fájlok készítésétől.
- **Keresés a katalógusban** a fájl neve alapján az állomány jellemzőinek, fizikai elhelyezkedésének adatait keressük, a leggyakoribb katalógus művelet. Hatékony és gyors végrehajtása érdekében célszerű a katalógust rendezett állapotban tartani.
- **Állomány megnyitása**, ha a folyamat egy állományhoz fordul, a megnyitás során az operációs rendszer ellenőrzi a kívánt művelethez szükséges jogosultság meglétét. Ha minden rendben van, létrehozza a fájl leíró táblát (**F**ile **C**ontrol **B**lock – **FCB**). A folyamat a továbbiakban ezen a struktúrán keresztül végzi műveleteit. A megnyitás célja lehet
    - **írás**: a fájl eredeti tartalma törlődik, az új adatok a régiek helyére kerülnek
    - **olvasás**: a fájl a művelet után változatlan marad
    - **hozzáfűzés**: az állomány eredeti tartalma nem sérül, az újabb adatok a meglévők után kerülnek
    - **írás/olvasás**

  Az adatok értelmezése szerint

- **bináris** a beolvasott bájtok pontosan megfelelnek a fájlban tároltaknak
- **szöveg** az olvasás a *fájl vége* karakternél leáll

Az elérés módja szerint

- **sorrendi** (szekvenciális): az aktuális pozíciót tartalmazó mutató az adatok írásával/olvasásával automatikusan növekszik
- **közvetlen** (random): az írás vagy olvasás helyét meghatározó mutatót a program állítja.

- **Pozícionálás állományokban:** A mutató helyének beállítása. Speciális, gyakran használt pozícionáló utasítás a fájl elejére állítás (rewind). Hozzáfűzés esetén a mutató automatikusan a fájl végére ugrik, onnan indul.
- **Írás/olvasás:** Adatátvitel a memória és az állomány között a fájlmutató által meghatározott pozíciótól.
- **Állomány kezelése:** Hatására az átmeneti tárolóban lévő adatok rögzítésre kerülnek, az FCB megszűnik.
- **Állomány, katalógus törlése**. A szülő katalógus megfelelő bejegyzésének ellátása a törölt állapotot mutató jelzővel (NEM fizikai törlés!)

Katalógusokon végzett, de tulajdonképpen állományokra vonatkozó művelet a fájlok jellemzőinek, hozzáférési jogainak módosítása, illetve (szerencsés esetben) a véletlenül törölt fájl visszaállítása.

Az aktuális pozíciót az operációs rendszer az FCB-ben tárolja. Érdemes rávilágítani az FCB és a PCB viszonyára!

Az operációs rendszer a PCB-vel azonosítja a folyamatokat. Ebben – többek között – minden használt erőforrás is be van jegyezve, így az állományok is. Az állományoknak viszont túl sok jellemzője van ahhoz, hogy mindegyik magában a PCB-ben szerepeljen, ezért az operációs rendszer a használt fájlokhoz külön táblázatokat hoz létre, és csak az e táblákra mutató adatokat tárolja a PCB-ben.

PCB (Process Control Block): A *folyamat* folytatásához szükséges adatok

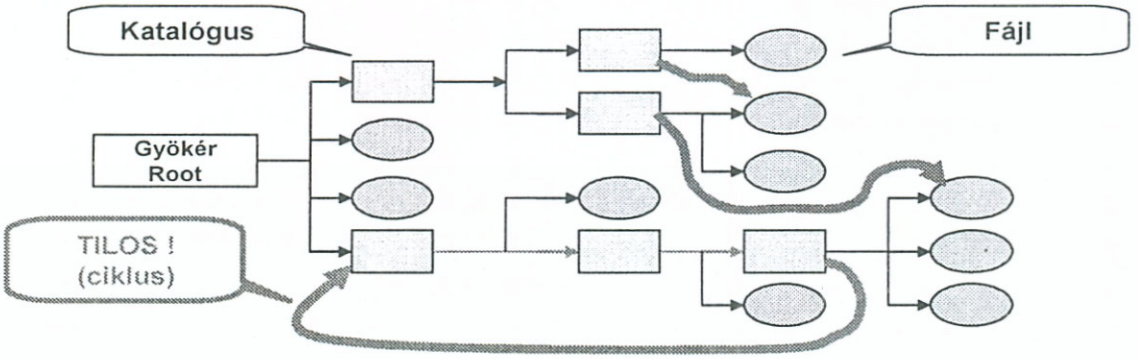
- Többek között: az FCB-kre mutató adatok

FCB (File Control Block): A *fájl művelet* folytatásához szükséges adatok

- Logikai azonosító
- Aktuális pozíció (hol tartok az olvasással?)
- Megnyitási mód (írás, olvasás, bináris, szöveg)
- Zárolási adatok (ne piszkáljon bele senki!)
- Változásjelző stb. (el kell menteni, mert megváltozott a tartalom!)

3.8 A fájlrendszerek jövője

Ha megfigyeljük napi munkánkat, és például a Microsoft operációs rendszereit, nyilvánvalóvá válik, hogy egyre kevésbé az állományokra, azok logikai



elhelyezkedésére, kiterjesztésére, hanem azok tartalmára figyelünk. Ez a fájlrendszerek módosulását, legalábbis a klasszikus fájlrendszerek háttérbe szorulását indukálhatja.

Bár a hierarchikus katalóguselrendezés jelenleg a legáltalánosabban elterjedt, ezek általánosításaként elvileg lehetséges aciklikus (azaz körbejárható hurkot nem tartalmazó) gráf jellegű katalógusokat készíteni, melyek bonyolultabbak ugyan, de sok előnyük is van. Hasonlóan a láncolás vagy parancsikonok által megvalósított közvetett fájl eléréshez, rugalmasabb, takarékosabb szervezést, többszempontú csoportosítást tesznek lehetővé.

3.11 ábra: Acilikus gráf

Az aciklikus gráf elve egyes alkalmazásokban, alrendszerekben, illetve az adatbáziskezelők szintjén már megjelent (IBM Lotus Notes, Asciental Media 360).

Nemcsak a szervezésben, de az állományokról tárolt információ tekintetében is változásokra számíthatunk. A hagyományos katalógusok – elsősorban történeti okokból – csupán a tároláshoz, legegyszerűbb kereshetőséghez, rendezéshez szükséges leíró információt tartalmazzák. Az újabb és újabb operációs rendszerek egyre több adat tárolását teszik lehetővé, azonban az adatok köre rögzített (az állományokról készült metaadatbázis sémája állandó). A felhasználó általában nem definiálhat új keresési szempontokat, de még ha meg is tehetné, az egyéni megoldások a hordozhatóság, kompatibilitás ellen hatnának.

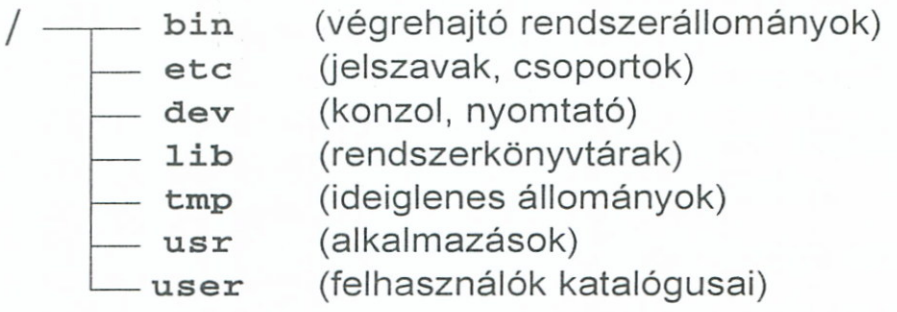
Miért ne tartalmazhatná maga az állomány valamilyen szabványos módon a saját kísérőinformációit? A jövőben várható (valószínűleg XML alapú technológián alapuló, a szemantikus web tartalmi elemeit alkalmazó) önleíró állományrendszerek megjelenése.

3.9 Állománykezelés a Linuxban

3.9.1 Logikai állománykezelés

A Linux által megvalósított állománykezelés a felhasználói oldalról nézve megegyezik a többi Unix változatban alkalmazottal.

A fájl a Unix felfogás szerint nem más, mint bájtok sorozata. A fájlrendszer hierarchikus, azaz az állományok a könnyebb áttekinthetőség érdekében katalógusokba vannak szervezve. A fa struktúra kiinduló pontja a gyökér (*root*) katalógus. Ha a megnyitni kívánt fájlt a gyökérhez képest szeretnénk megadni, tehát *abszolút címzést* alkalmazunk, az elérési utat törtvonallal ('/', *slash*) kell kezdeni. Relatív címzésnél a kiinduló pont az aktuális katalógus, alkalmazható a '.' szimbólum az aktuális, illetve a '..' szimbólum a szülő katalógus jelölésére. Ugyancsak relatív címzésnél hasznos szimbólum a '~' (*tilde*), amely a felhasználó bejelentkezési katalógusát jelöli. Mindkét esetben a fájlhoz vezető útvonal által érintett katalógusokat egymástól a '/' karakterrel kell elválasztani. Az egyes alkalmazásokkal ellentétben a kernel általában nem feltételez semmiféle rendezettséget, struktúrát a fájlon belül, ez alól csupán a katalógusok a kivételek. Unix specialitás az is, hogy az eszközkezelők, például a konzol, a nyomtató, a hálózat felülete is logikailag bájt folyam (*stream*), azaz fájl formájában jelentkezik. A perifériák tehát a felhasználó számára úgy jelennek meg, mintha a */dev* katalógusban lévő állományok lennének. Az alábbi ábra tipikus Unix katalógus szerkezetet mutat.



Az állomány- és katalógusnevekre nézve a jelenlegi Linux nem alkalmaz semmiféle szigorú kikötést. A kis- és nagybetűk mindig különbözőek!

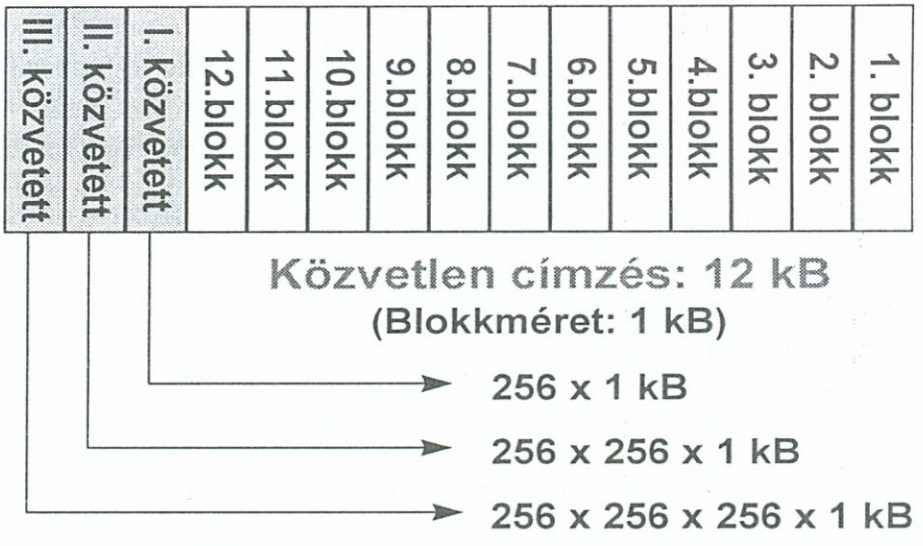
3.9.2 Klasszikus fájlkezelés

Az állományok hagyományosan valamilyen háttértáron, elsősorban mágneslemezeken helyezkednek el. A fizikai háttértárak logikai eszközökre oszthatók, azaz partícionálhatók. Az egyes logikai partíciók első szektorában található az operációs rendszert betöltő program, a *boot block*, amelyet a partícióhoz rendelhető legfontosabb paramétereket (kapacitás, blokkméretet stb.) tartalmazó *superblock* követ (lásd DOS

boot record, partíciós tábla). A betöltő programok közül természetesen csak egyetlen egyre van szükség, de a biztonság kedvéért több is szokott lenni belőlük.

A **katalógusok** szerepe a Linux-ban csupán az *állománynevek* és *inode* sorszámok egymáshoz rendelése, a bejegyzés típusáról, adatairól ennek alapján még semmit sem lehet tudni. A superblock-ot közvetlenül követi a gyökér katalógus (*root directory*), amely az adott partíción elhelyezkedő fájl rendszer kiinduló pontja. A katalógusok elhelyezkedés szempontjából nem különböznek a közönséges fájloktól, csak tartalmuk más kissé. A bejegyzések (változó hosszúságú fájlnevekről lévén szó) tartalmazzák a fájlnév hosszát, magát a fájlnevet, valamit a fájl jellemzőit leíró indextáblára (*inode*) mutató 4 bájtos pointert. A keresés a katalógusokban lineáris.

Az **inode** tartalmaz minden olyan adatot, amit a katalógusba bejegyzett névvel jelölt objektumról a rendszernek ismernie kell. Ebből derül ki, hogy az adott objektum milyen típusú (fájl, katalógus vagy egész fájl rendszer), melyik felhasználó tulajdona, mely felhasználói csoportnak vannak különleges jogai és mik ezek a jogok, mekkora a mérete, mikor jött létre, mikor módosították utoljára, hány katalógus bejegyzés hivatkozik rá és természetesen azt, hogy az adatokat tartalmazó blokkok hol helyezkednek el. A Linux 12 közvetlen, és háromszintű közvetett mutatót használ egy fájl blokkjainak azonosítására.



Látható, hogy a harmadik szintű közvetett címzésre még 1 kB-os blokkméret esetén is csak igen ritkán kerül sor, hiszen egy 64 MB-os fájl már két szinttel megcímezhető.

Érdekes eset, amikor az inode egy fájlrendszert ír le. Ennek a lehetőségnek a következménye, hogy a Unix egyetlen katalógus hierarchiában képes megjeleníteni különböző fizikai vagy logikai eszközöket, függetlenül attól, hogy ezek ugyannak a számítógépnek a perifériái, amelyen a felhasználó dolgozik, vagy egy másik számítógép egy lemeze. A fájlrendszer beillesztését egy katalógus szerkezetbe *mount*-olásnak nevezzük. Logikailag a beillesztett fájlrendszer gyökér katalógusa kerül az inode táblába, így a távoli gép winchesterének tartalma úgy látszik, mintha saját gépünk egy katalógusa lenne.

3.9.3 A virtuális fájlrendszer

A Linux – mint már láttuk – minden adatfolyamot fájlként kezel attól függetlenül, hogy annak forrása vagy célja egy program lemezen tárolt kódja, vagy például a konzol billentyűzete vagy monitora, esetleg egy hálózati csatorna. További megoldandó probléma, hogy fejlődése során a Unix-Linux sok fejlődési állomáson ment keresztül, sokféle fájlrendszerrel kellett kompatíbilisnek maradnia.

A Linux a problémát a virtuális fájlrendszer (**V**irtual **F**ile **S**ystem - **VFS**) alkalmazásával oldotta meg. A VFS objektum orientált szemlélettel definiálja az állományok jellemzőit és azt a szoftver réteget, amely a fájlokkal végezhető műveleteket valósítja meg. Háromféle objektum létezik, a *fájl-*, *inode-* és *file rendszer* - objektum. Az objektumok minden egyes képviselője tartalmaz egy mutatót, amely arra a táblázatra mutat, amely az adott objektumra vonatkozó műveleteket végző függvényeket tartalmazza. A fájlrendszert használó folyamatoknak tehát ettől kezdve nem kell foglalkozniuk azzal, hogy egy fájl a lemezen, vagy egy másik gépen, vagy egy I/O csatornán érhető el, vagy az adott rendszer milyen fizikai fájl szerkezet valósít meg, bízhatnak benne, hogy az adott objektum által mutatott függvények a kívánt műveletet jól fogják elvégezni. (Természetesen a fizikailag különböző rendszerekhez tartozó függvények kódja más-és más, csak a nevük és paramétereik azonosak.)

Fájlrendszer objektumot rendel az operációs rendszer minden olyan eszközhöz, amin fájlok találhatók, vagy amin keresztül fájlok érhetők el. Ennek legfontosabb feladata az, hogy egy folyamat kérésére visszaadja a hivatkozott állomány inode sorszámát, mivel a fájlrendszer–inode számpár a VFS számára egyértelműen azonosítja az állományt.

Az **inode objektum** a fájl egészét jellemzi. Tartalmazza a hozzáférési jogokat, a fájl legfontosabb jellemzőit, illetve a fizikai elhelyezkedésre vonatkozó adatokat. (Természetesen az inode objektumban található a fájlműveleteket végző függvények listája is.) Egy inode objektumot több folyamat is felhasználhat, ezért ezek gyakran a folyamat lefutása után is a gyorsító tárban maradnak, hátha más folyamatok is használni tudják.

A **fájl objektum** általában csak egy folyamathoz tartozik, az állomány pillanatnyi felhasználásának paramétereit tartalmazza. Ez az objektum tárolja az aktuális mutatót, mely megmondja, hogy hol tart éppen a szekvenciális olvasás, azt, hogy a megnyitás írási vagy olvasási szándékkal történt-e, valamint itt tartja nyilván az operációs rendszer a hozzáférések gyakoriságát, hogy megállapítsa, érdemes-e gyorsító tárat alkalmazni a működés optimalizálásának érdekében. Mivel a fájl objektumok egy konkrét folyamat futásához rendelhetők, a futás befejeztével – további feladat híján – megszűnnek.

A katalógusok kezelése kissé különbözik a klasszikus fájlokétól, de nem annyira, hogy külön objektum típust lenne érdemes definiálni. A katalógusokra vonatkozó speciális függvények a megfelelő *inode* objektumon keresztül érhetők el.

3.9.4 EXT2FS - A Linux lemezes fájl rendszere

A Linux lemezegységei a *ext2fs* (Second Extended File System – második bővített fájl rendszer) fájlrendszer szerint szerveződnek. Az elnevezés jól tükrözi a többszöri

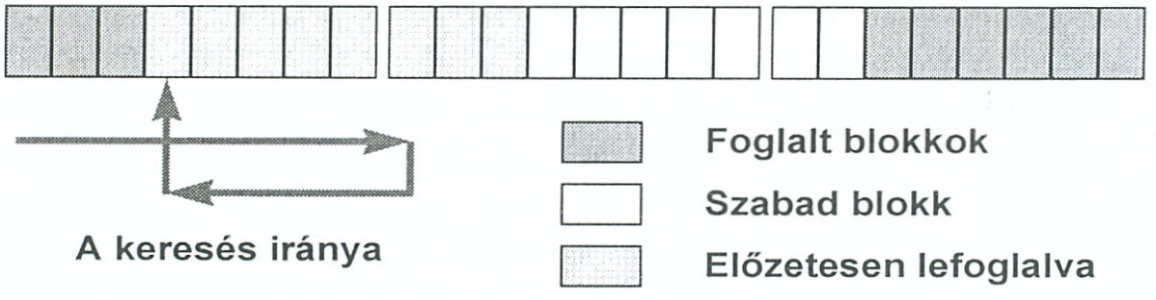
bővítést, módosítást. A fájlrendszer célja, hogy minél jobban kihasználja a rendelkezésre álló tároló kapacitást, és minél gyorsabban legyen képes kiszolgálni a folyamatok igényeit. A nagyobb sebesség érdekében nagy blokkméretek kialakítása szükséges, mivel így a fájlműveletekkel járó rendszertevékenység (címzés stb.) részaránya a lehető legkisebb. A nagy blokkméret viszont magában rejti a töredezettség (*fragmentation*) lehetőségét, ami rossz helykihasználással jár. Az *ext2fs* – éppúgy mint a többi fájlrendszer – e két ellentmondó szempontot igyekszik optimálisan kielégíteni.

A Linux nem alkalmaz a töredezettség csökkentése érdekében változó méretű blokkokat, hanem eleve kis blokkmérettel dolgozik. Az alapértelmezett méret 1 kB, de a rendszer támogatja a 4 kB-os és 8 kB-os blokkok alkalmazását is (természetesen fájlrendszerenként csak egyfélét).

Annak érdekében, hogy az átvitel sebessége is megfelelő legyen, az egymáshoz fizikailag közeli blokkok csoportokba (*block group*) szervezettek. Ez a megoldás emlékeztet a DOS clustereire, amely szomszédos blokkokat von össze, és nagyon hasonlít a klasszikus Unix-ok cilinder csoportjaira (*cylinder groups*), melyek szomszédos a cilinderek blokkjait tartalmazzák. Az *ext2fs* arra törekszik, hogy az azonos állományokhoz tartozó blokkok azonos csoportban, sőt lehetőleg folytonosan helyezkedjenek el. Nézzük, hogyan érhető el ez a cél!

Ha egy folyamat létre szeretne hozni egy állományt először ki kell választani egy blokk csoportot. Adatfájlok esetén a legkívánatosabb az a csoport, ahol az adatfájlhoz tartozó *inode* is található. Katalógusok esetén viszont a cél éppen ellentétes. Az *ext2fs* arra törekszik, hogy a katalógusokat egyenletesen terítse el a lemezen, egyrészt a terhelés elosztás, másrészt az adatbiztonság növelése miatt.

A blokk csoporton belül a szabad helyeket egy bit térkép tartja nyilván, a keresés ebben a táblázatban történik. Ha új fájlról van szó, a blokkcsoport eleje a kiinduló pont, ha már meglévő fájl bővítéséről, a keresés az eredeti fájl utolsó blokkjától indul. A rendszer a műveletet két lépcsőben végzi. Először egy egész bájtot, azaz nyolc összefüggő üres blokkot keres. Ha sikeres volt a művelet, annak érdekében, hogy lehetőleg ne maradjon kihasználatlan töredék, az algoritmus bitenként visszafelé lépked egészen addig, amíg szabad blokkokat talál.



A nyolc összefüggő blokkot a rendszer előzetesen lefoglalja a folyamat számára, és a felesleg csak a fájl lezárása után szabadul fel. Így nem fordulhat elő, hogy az egyidőben futó folyamatok felváltva írogassanak, így töredezettséget idézzenek elő.

Ha nincs összefüggő tartomány, nincs más hátra, blokkonként kell összeszedni a szükséges helyet, sőt, néha más blokkcsoportokat is igénybe kell venni.

3.9.5 PROC - A nemlétező fájlok rendszere

A fájl fogalom általánosításaként jött létre a Unix fejlődése során egy olyan fájlrendszer, melynek elemei sehol sem tárolódnak, hanem a felhasználó folyamatok kérésére generálódnak. Eredeti célja a programok tesztelésének elősegítése volt. Minden aktív folyamathoz tartozott egy katalógus, melynek neve a folyamat azonosítója (illetve annak ASCII megfelelője), tartalma pedig a folyamat futására jellemző adatok halmaza volt. A Linux a koncepciót jelentősen továbbfejlesztette azáltal, hogy a gyökér katalógusban szövegfájlként tette elérhetővé a fontosabb rendszer-statisztikákat, a betöltött eszközkezelők adatait.

3.9.6 Biztonság

A Linux biztonsági rendszere – a többi operációs rendszerhez hasonlóan – két pilléren nyugszik.

3.9.6.1 A felhasználók azonosítása

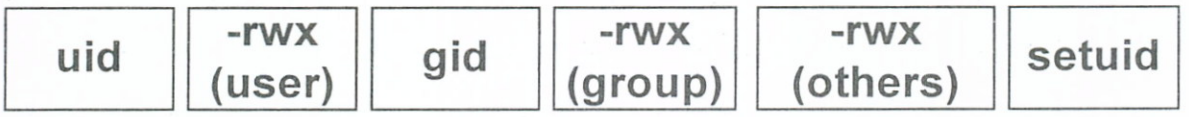
A **felhasználók azonosítása** (Authentication) biztosítja, hogy senki sem kerülhet kapcsolatba a rendszerrel egészen addig, amíg be nem bizonyítja, hogy neki ott jogai vannak. Az azonosítás jelszó segítségével történik. A felhasználó által adott jelszót a rendszer egy kvázi véletlen számmal kombinálja, majd egy egyirányú transzformációnak veti alá, és az eredményt egy jelszó fájlban tárolja. Az átalakítás azért egyirányú, hogy semmiképpen se lehessen találgatással visszakövetkeztetni az eredeti jelszóra. Bejelentkezéskor ugyanez a folyamat ismétlődik, és az eredmény összehasonlításra kerül a jelszófájl tartalmával.

A biztonság nagyban függ a jelszó hosszától, az elérhető kvázi véletlen számok mennyiségétől, a transzformáció bonyolultságától, és nem utolsó sorban a jelszó fájl védettségétől. A kezdeti Unix-ok bizony hagytak kívánnivalókat ezen a téren, de ezeket a hibákat jórészt már kijavították.

3.9.6.2 Hozzáférési jogok

A **hozzáférési jogok** (Access control) szabályozzák, hogy melyik felhasználó melyik objektumokkal, és milyen műveleteket hajthat végre. A szabályozás az objektumok széles körére vonatkozik, a fájlokra éppúgy, mint a folyamatokra, vagy egy megosztott memória szegmensre. Linux alatt háromféle hozzáférési jog létezik. Egy objektum lehet olvasható (**r**ead – **r**), írható (**w**rite – **w**), végrehajtható (**e**xecute – **x**), illetve az előző három bármely kombinációja. A Unix a felhasználókat a felhasználói azonosító (**u**ser **i**dentifier – **uid**) alapján, a felhasználói csoportokat a csoportazonosító (**g**roup **i**dentifier – **gid**) szerint különbözteti meg. Létezik még egy speciális csoport (*others*), melynek a rendszer minden felhasználója a tagja. Minden objektumnak van egy *uid* és (legalább)

egy *gid* mezője, valamint három, a három különböző felhasználói szinthez (*user, group, others*) kötődő jogosultságokat leíró mezője.



Például ha egy felhasználónak egy állománnyal valami célja van, az operációs rendszer megvizsgálja, hogy a felhasználó azonosítója megegyezik-e az objektum leíróban tárolt *uid*-vel. Ha igen, az azt jelenti, hogy az objektum éppen az aktuális felhasználó tulajdona (*owner*), és így őt az *uid*-hez rendelt jogok illetik meg. Ha a példánkban szereplő felhasználó benne van abban a csoportban, amelyet az állomány *gid* mezője tartalmaz, rá a csoportjogok vonatkoznak, egyébként be kell érnie a mindenki számára biztosított *others* lehetőségekkel.

Az egyetlen speciális jogokkal rendelkező felhasználó a **root** (olykor *superuser*-nek is szólítják), akinek mindig mindenhez joga van. Kizárólag a *root* jogosultságú folyamatoknak van joga elérni a fizikai memóriát, a fontosabb rendszerfájlokat, csak neki van joga privilegizált módban futó folyamatokat indítani.

Van azonban egy olyan lehetőség is, amelynek segítségével közönséges folyamatok is – legalábbis egy feladat erejéig – különleges jogokkal rendelkezzenek. Erre akkor lehet szükség, ha egy felhasználó éppen a jelszavát kívánja megváltoztatni, vagy a nyomtatási-, esetleg az elektronikus levelezési sorban kíván egy állományt elhelyezni, illetve minden esetben, amikor rendszerszintű állománnyal akad dolga. Ezt a jogosultságokat leíró tábla **setuid** mezője teszi lehetővé. Ha ez a bit a tulajdonos által engedélyezett, az elindított folyamat, amikor a kritikus, magasabb privilégiumokat igénylő részhez ér, birtokolhatja a tulajdonos jogait (*effective uid*), majd visszakapcsolhat a felhasználó által elérhető jogosultsági szintre (*real uid*).

Az előző fejezetből megismerhettük az állományok létrehozásának célját, a különböző rendszerekben alkalmazott elnevezési szabályokat, a fájlok tulajdonságait, a hozzájuk rendelhető felhasználói jogokat, illetve a velük végezhető műveleteket. Kitértünk a katalógus szerkezetekre, illetve a fájlrendszeren belüli eligazodás lehetőségeire. A fájlok fizikai elhelyezésének tárgyalásánál tárgyaltuk a folytonos, láncolt, valamint az indexelést alkalmazó módszert

Σ

3.10 Ellenőrző kérdések

1. Ismertesse a hierarchikus katalógusrendszerben a relatív fájl elérés módját! Milyen speciális elnevezéseket és jelöléseket ismer?
2. Mi a fájl, milyen adatait tartja nyilván az operációs rendszer? Milyen elnevezési szabályokat ismer? Mi az előnye, ill. hátránya a „8+3" szabály betartásának?
3. Mi a katalógus (directory) célja? Milyen katalógus elrendezéseket ismer? Mik ezek előnyei illetve hátrányai?
4. Milyen fájl hozzáférési jogokat ismer? Mi a különbség az hozzáférési jogok és az attribútumok között?
5. Milyen közvetett fájl elérési módokat ismer? Mikor nem alkalmazható a „hard link"?
6. Mutassa be a láncolt fájl elhelyezés (FAT) elvét! Melyek a legfontosabb előnyei, illetve hátrányai?
7. Mutassa be az indexelt fájl elhelyezés (INODE) elvét, előnyeit, hátrányait!
8. Milyen módon tartják nyilván az operációs rendszerek a fájlok közötti szabad területeket? Hasonlítsa össze a tanult módszereket!
9. Milyen stratégiák alapján helyezhetünk el egy fájlt a lemezre (meglévő fájlok közé) folytonos kiosztás esetén? Értékelje a tanult módszereket!
10. Hasonlítsa össze a folytonos, a láncolt és az indexelt fájl elhelyezés előnyös és hátrányos tulajdonságait!
11. Folytonos, láncolt és indexelt fájl elhelyezési stratégiáknál milyen problémát okozhat a nagy fájlok elhelyezése? Milyen módszerekkel küszöbölhető ki?
12. Definiálja az abszolút fájl elérés fogalmát! Melyek a hátrányai? Mikor használható ez a módszer?
13. Mi az eltérés a DOS alapú és a Unix alapú rendszerek fájlkezelése között?
14. Milyen adatok találhatók indexelt fájl elhelyezés esetén az INODE táblában?

15. Mi a FAT-et alkalmazó fájlrendszerek legnagyobb korlátja? Hogyan lehet ezen túllépni?
16. Mi az FCB (File Control Block) szerepe? Van-e valami kapcsolatban a PCB-vel (Process Control Block)?

4. Háttértárkezelés

Az állományok jellemzőinek megismerése után azokat az eszközöket vesszük sorra, amelyek ezek tárolására szolgálnak. A háttértárak, ezen belül is különösen a mágneslemezek működésének tárgyalása egyúttal például is szolgál az általános eszközkezelésre.

A processzor és a memória mellett a számítógépek nélkülözhetetlen építőelemei a ki- és bemeneti (input/output) egységek, melyek alapvető feladata egyrészt a külvilággal való kapcsolattartás, másrészt az adatok, programok tartós, állandó tárolása későbbi felhasználás céljából.

A perifériáknak rengeteg típusa ismert, és kis túlzással az is állítható, hogy mindössze annyi a közös bennük, hogy a felhasználók egyikük működésével sem szeretnének túlzottan szoros kapcsolatba kerülni, a kezelést örömmel engedik át az operációs rendszereknek. A felhasználó szempontjából az lenne a legkellemesebb, ha a perifériák fizikai különbözősége ellenére használatuk (természetesen csak a felhasználói folyamatok szemszögéből) egységes lehetne.

A külvilág legfontosabb képviselője a felhasználó, a maga sajátos, a számítógépek működésétől alapvetően eltérő emberi gondolkodásával és kommunikációs lehetőségeivel, de a számítógépek közötti adatcsere szerepe is óriási, és napjainkban is rohamosan nő. A háttértárak fontossága nem kérdéses. Az összes periféria bemutatására nincs lehetőségünk, tehát választani kell egy alkalmas képviselőt, de melyiket?

1. Az emberi tényező szerepét már láttuk a felhasználói felület tárgyalásánál, a hálózatok fontosságára pedig az elosztott rendszereknél utaltunk.
2. A mágneses háttértárak végigkísérték a számítógépek fejlődését, szerepük ma is elsőrendű.
3. A perifériakezelés egységesítésében a példát a lemezeken tárolt állományok kezelése szolgáltatta.

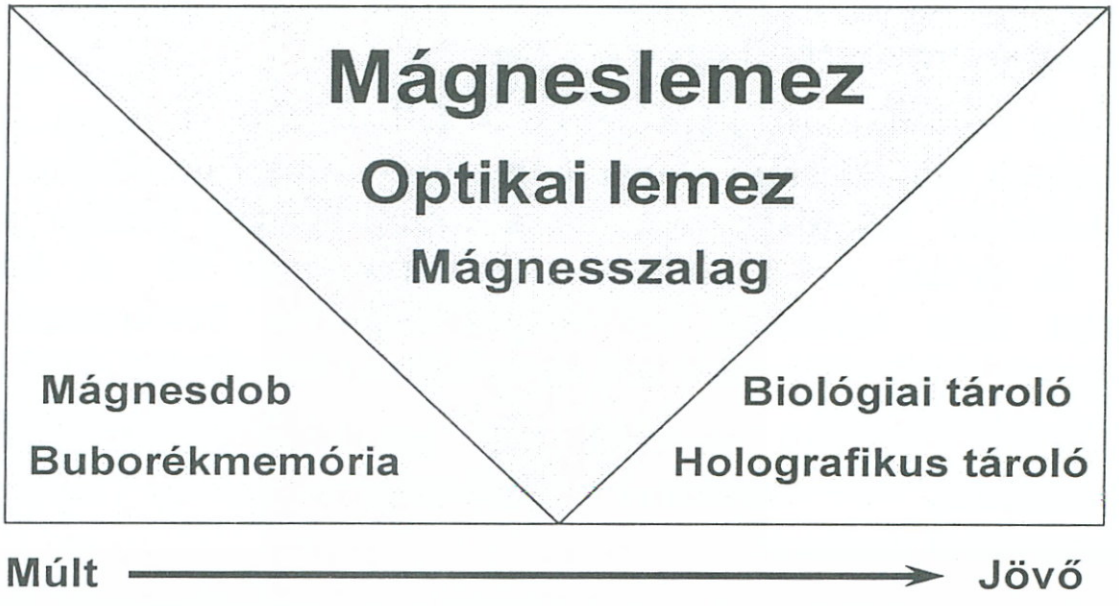
4. A lemezek kezelése igényli a leggyorsabb, legösszetettebb működést, ezért az itt felhasznált megoldások biztosan alkalmazhatóak a többi perifériánál is.

A következőkben tehát csak az operációs rendszerek háttértár kezelésére koncentrálunk.

4.1 Háttértárolók felépítése

A számítógépek megjelenése óta számtalan háttértár gyanánt alkalmas eszközt fejlesztettek ki. Mindegyikük célja olyan tárolási forma megvalósítása, melynek mérete jelentősen meghaladja az operatív tár méretét, és nem felejti el tartalmát a tápfeszültség kikapcsolásakor. Némelyek, mint a mágnesdob, vagy a buborékmemória már elfoglalták jól megérdemelt helyüket a műszaki múzeumok polcain, mások, mint a holografikus, vagy biológiai elven működő tárolók még nem hagyhatták el szülőszobájukat, a kísérleti laboratóriumokat.

A jelenleg alkalmazott háttértárak közül kiemelkedően a legjelentősebbek a mágneslemezes tárolók, de bizonyos feladatkörökben még tartják magukat a mágnesszalagok, és feltörekvőben vannak az optikai elven működő háttértárak. A továbbiakban ezekkel a típusokkal foglalkozunk.



4.1 ábra Háttértár típusok

4.1.1 Mágnesszalagok

A mágneses háttértárolók legelső formája a mágnesszalag volt. Adattároló kapacitásuk a lemezes tárolókét korábban jóval meghaladta, jelenleg azok nagyságrendjébe esik. Alkalmazásuknál fontos szempont, hogy mágnesszalagok esetén a lejátszó/felvevő eszköz ugyan drága, de az adathordozó ára töredéke a hasonló kapacitású merevlemezek árának, és az éppen parkoló szalag energiát sem fogyaszt.

Felépítéséből adódóan soros hozzáférésű, tehát mielőtt beolvashatnánk a kívánt adatokat, végig kell haladni az azt megelőzők felett, így egy-egy adatcsoport elérési ideje akár percekben is mérhető (nem is beszélve a szalagcsere idejéről). A soros elérés további hátránya, hogy bármilyen adatmódosítás esetén a teljes állományt újra kell rögzíteni. Ha azonban a megfelelő szalag már a megfelelő helyre ért, az adatátviteli sebesség nagyságrendileg megegyezik a lemezeknél tapasztalhatóval. A mágnesszalagok feladata felépítésükből adódik: hatalmas mennyiségű, *összefüggő* adatok tárolása:

- Archiválás, adatmentés (akár teljes rendszerek)
- Nagy tömegű adat átvitele (például adatbázisok)
- Sok összefüggő adat tárolása (például videó állományok)

Σ

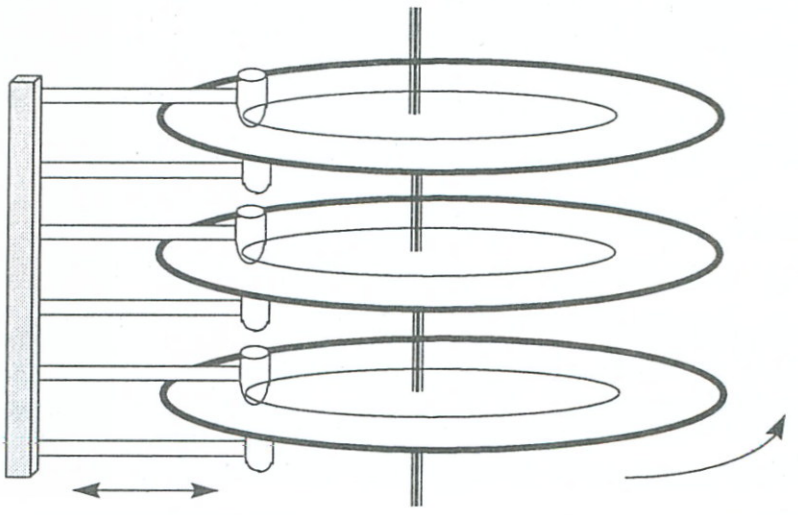
1. **Archív tárolóként** olyan adatok, programok tárolására szolgál, melyekre ritkán van szükség. Az archív tárolás egy esetére, az **adatmentés** (backup) céljaira is ideális megoldás. Például egy hálózat állapotát minden este elmenthetjük egy szalagra, majd másnap egy másikra, és a két szalagot felváltva használjuk, akkor minimális költséggel igen nagy adatbiztonságot érhetünk el.
2. **Adatok átvitele számítógépek között**, ha tekintélyes mennyiségről van szó, még manapság, a hálózatok korában is mágnesszalagok segítségével a leggyorsabb. Például az Interneten elfogadhatónak számító 10 kB/s átviteli sebesség mellett egy 20 GB-os adatmennyiség átvitele csaknem egy hónapig tartana, míg egy repülő a szalagokkal a világ legtávolibb pontjáról is egy nap alatt célba ér.
3. **Nagy adatmennyiség (például videó) tárolása.** Bonyolult rendszerek szimulációja, gyakran igényli a közbülső állapotok tárolását, ezek nemritkán igényelnek gigabájtos kapacitást. A rengeteg adatra mindig csak együtt, egyszerre van szükség, ezért a szalagos tároló ideális erre a célra.

A piacon számos megoldás van jelen, jellemzőik nagyságrendileg megegyeznek (DAT, AIT, DLT, LTO). A korábbi lineáris szalag párhuzamos sávjai helyett a mai mágnesszalagok rögzítési módja leginkább a videómagnókéra hasonlít.

Kapacitásuk 16-200 GB, amely a beépített tömörítő algoritmusokkal még fokozható. Átviteli sebességük 6-12 MB/mp, elérési idejük 30-60 mp.

4.1.2 Mágneslemezek

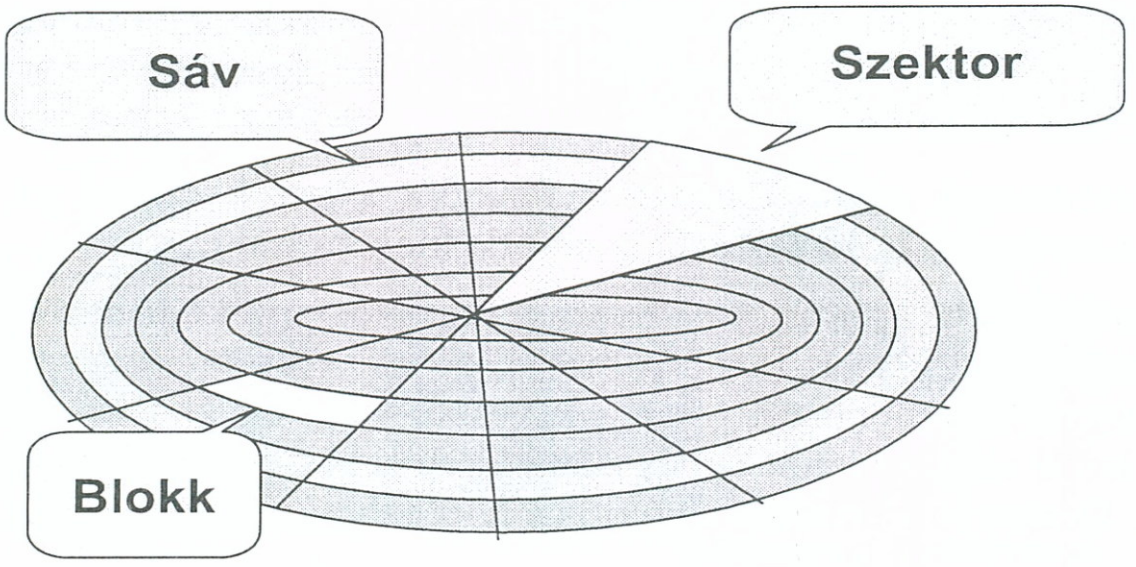
A mágneslemezek (winchester, **H**ard **D**isk **D**rive - **HDD**) a leggyakrabban alkalmazott, leguniverzálisabban használható háttértároló eszközök. Segítségükkel nagy adatátviteli sebesség (10-160 MB/s) érhető el, igen nagy kapacitásúak (20-200 GB) és viszonylag olcsók. Közös tulajdonságuk, hogy mágnesezhető réteggel borított, 1,5-5,25 hüvelyk átmérőjű korongokból állnak. A lemez gyorsan forog (3600, 5400, 7200, 10000 fordulat/perc), a koncentrikus körök, a *sávok* (track) mentén tárolt adatokat sugárirányban mozgatható *olvasó/író fejek* olvassák, illetve rögzítik. Legtöbbször egy tengelyen több lemez is található (winchester), ilyenkor a közös, fésűszerű mechanikával mozgatott fejek száma mindig kétszerese a lemezek számának. Az egymás alatt elhelyezkedő sávokat együttesen *cilindernek* nevezzük. A lemezeket teljesen zárt doboz védi a legapróbb szennyeződésektől is. A fejek a lemezfelület fölött mikronnyi távolságra légpárnán úsznak.



4.2 ábra A winchester felépítése

A sávokon kívül egy-egy lemezoldal, mint egy torta szeletei, *szektorokra* is oszlik. A sávok és a szektorok metszéspontjainál kialakuló ívekben, a

blokkok jelentik a legkisebb átvihető adatmennyiséget. A blokkok nemcsak adatokat tartalmazhatnak, hanem sorszámot, vagy ellenőrző összeget is, esetleg jelezhetik az operációs rendszer számára az adott blokk hibás voltát. A szektorok és blokkok előkészítése, ellenőrzése szoftver úton, a formázás során történik.



4.3 ábra A lemezoldalak felosztása

A lemezek logikailag gyakran több, egymástól független kötetből (partíció, volume) állnak. A kötetekre vonatkozó speciális információt, a kezdő sáv címét és a méretét a lemezcsoportok első sávja, azaz az első lemez legkülső sávja, a *partíciós tábla* tartalmazza. A partíciós táblában található meg az az információ is, hogy melyik kötet az aktív, azaz melyik tartalmazza az éppen betölteni kívánt operációs rendszert.

A blokkok tipikus mérete 0,5-64 kB. Egy blokk tartalmának átviteléhez szükséges időt három tényező befolyásolja:

1. **Fejmozgási idő** (seek time), az az idő, amíg a fej eléri a kívánt blokkot tartalmazó sávot. Értéke a 10 ms nagyságrendjébe esik.
2. **Elfordulási idő** (latency time) a kiválasztott blokkot tartalmazó szektor fej alá kerülésének ideje, mely legrosszabb esetben egy körülfordulási idő, átlagosan szintén kb. 10 ms.
3. **Adatátviteli sebesség** (transfer time) a blokk adatainak továbbításához szükséges idő. 3600/perc-es fordulatszámmal, 63 szektorral számolva az az idő, míg a fej egy blokk fölött tartózkodik, kb. 0,25 ms. Ha a

blokkméret 512 byte, a szükséges átviteli sebesség kb. 2 MB/s. Ez az érték a fordulatszám növelésével, a párhuzamos működéssel és a bitsűrűség növelésével jelentősen fokozható.

A blokkok címzéséhez tehát három adatra van szükség. Meg kell adni a lemezoldal, a sáv és a szektor sorszámát. A lemezek a leggyakrabban címzett perifériák, kezelésükhöz – különösen a felhasználói folyamatok szintjén – nem igazán kellemes, ha mindig három adatot kell megadni. Az operációs rendszerek azonban nem hiába alakultak ki, átveszik ezt a kellemetlen feladatot.

A merevlemezek minden háttértár funkciót képesek ellátni az adatok, programok tárolásától a virtuális memória kialakításáig.

A winchesterekhez lényegében teljesen hasonlóan működnek a hajlékony lemezes *floppy lemezek*, illetve a merev, de *cserélhető lemezes* tárolók. A leglényegesebb különbség, hogy a kevésbé, vagy egyáltalán nem védett felület miatt az adatbiztonság, és adatátviteli sebesség egyaránt jelentősen romlik. Ezeket az áldozatokat pótolja a hordozhatóság.

4.1.3 Optikai tárolók

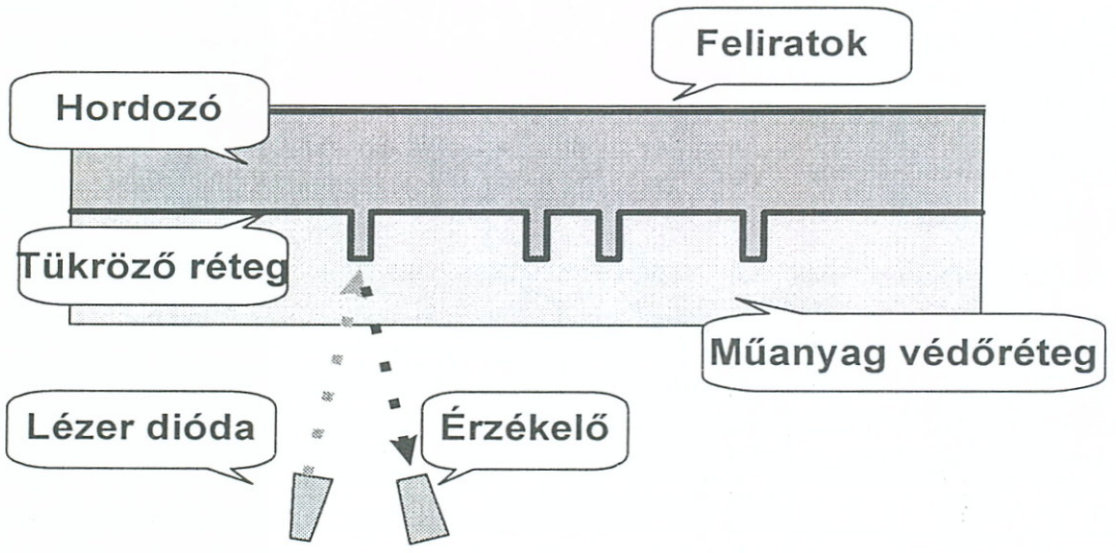
Az adattárolás alapvetően más formáját valósítják meg az optikai lemezek. A 8 vagy 12 cm átmérőjű műanyag korongokon 650-1300 MB, sőt a legfrissebb technológia a **DVD** (**D**igital **V**ersatile **D**isk) alkalmazásával 17 GB adat is tárolható. Általában csak olvashatóak (CD-ROM), de léteznek egyszer, vagy többször írható változatok is. Az elérési idő, illetve adatátviteli sebesség kezdetben megegyezett a hangrögzítésben alkalmazott értékekkel (1x, egyszeres sebességű CD), azaz nem sokkal volt különb a hajlékony mágneslemezeknél mérhető 300 ms, illetve 150 kB/s értékeknél. A többszörös (2x–32x) sebességű CD-k paraméterei arányosan jobbak elődeinél.

Az optikai lemezek alkalmazásának legnagyobb előnye, hogy nagy tömegben való gyártásuk nagyon olcsó, ezért kiválóan alkalmasak lexikonok, könyvek, nagyméretű programok kereskedelmi forgalmazására. Kis méretük és megbízhatóságuk alkalmassá teszik őket archiválási célokra.

A hangtechnikából átvett **CD** (**C**ompact **D**isk) felületén az egyes biteket (a hagyományos hanglemezhez hasonlóan) spirális vonalban elhelyezkedő, 0,5 μm átmérőjű kiemelkedések (pit) és a köztük lévő sík

területek (land) reprezentálják. A bitek a mágneslemezektől eltérően, ahol a belső sávokban az adatsűrűség növekedett, itt egyenletes távolságokra, 1,6 μm-re követik egymást. Az egyenletes adatátviteli sebesség elérése érdekében ezért az optikai lemezek fordulatszáma függ az olvasó fej aktuális helyzetétől (5-10 s^{-1}, illetve ennek többszörösei). (Megjegyzendő, hogy az olvasás belülről kifelé történik.)

Az olvasás elve az, hogy a lemezt letapogató infravörös lézersugár a felületről visszaverődve különböző időben (fázisban) érkezik vissza az érzékelőhöz, attól függően, hogy kiemelkedés, vagy térköz haladt el alatta.



4.4 ábra Optikai lemezek olvasása

Az írható lemezek kiolvasása hasonlóképpen történik, azonban az írás más fizikai elveken alapul. Ez utóbbi esetben lézerrel felmelegített speciális anyag mágnesezettségét változtatják meg, és az ezzel szerencsésen együtt járó törésmutató csökkenés (terjedési sebesség növekedés) jelentkezik úgy az olvasó számára, mintha a fény rövidebb utat tett volna meg.

4.2 Eszközmeghajtók

Az eszközmeghajtók az operációs rendszer magjának a részei, és mint ilyenek, feladatuk egyrészt a hardver hatékony kihasználása, másrészt a felhasználó, illetve felhasználói folyamat kiszolgálása úgy, hogy a hardver részletei minél inkább a háttérben maradjanak. Annak érdekében, hogy megállapíthassuk, hogy mit kell tudnia a lemezeket vezérlő

eszközmeghajtónak, vessük össze a felhasználók felől érkező igényeket a hardver igényeivel.

A **felhasználói folyamatok**, ha valamilyen perifériára van szükségük, egy rendszerhívást adnak, melyet a kernel egy folyamata fogad, és ad tovább az eszközvezérlőnek. A kérés természetesen érkezhet a rendszermagon belülről is, például az egyik legaktívabb lemezhasználó folyamat a virtuális memória kezelője. Az igazán kellemes környezet az, ha minden periféria egységes felülettel kezelhető. Ez nemcsak a felhasználók érdeke, hanem az operációs rendszerek tervezőié is, mivel a perifériák gyorsan változnak, és igen sokfélék, öngyilkosság lenne olyan operációs rendszert létrehozni, mely teljes egészében egy perifériára épít. A rétegszerkezet megvalósításával elérhető, hogy a hardver változásával csak jól meghatározott rétegekhez kelljen hozzányúlni. Tehát a lemezhez forduló rendszerfolyamat szempontjából az a legkedvezőbb, ha neki elegendő lenne az alábbi adatstruktúrát kitöltenie:

Eszköz típusa	Lemez, szalag, CD, nyomtató, stb.
Eszköz azonosítója	Az azonos típusú eszközök megkülönböztetése
Az adat kezdőcíme az eszközön	Azon blokk címe, melynek tartalmát írni vagy olvasni akarjuk
A memóriacím	Ahová a beolvasott adatok kerülnek, vagy ahonnan a kiírandók származnak
Az adatok mennyisége	Az átviendő blokkok száma
Írás vagy Olvasás	Az adatáramlás irányának meghatározása
Visszatérés	Annak a kernel folyamatnak az azonosítója, melyet értesíteni kell a művelet befejezésekor

4.5 ábra Az átvitelhez szükséges adatok

A **lemezegység** ezzel az igénnyel szemben egy számhármast vár, mely megadja, hogy a kívánt blokk melyik lemezoldal melyik szektorának melyik sávjába található. Az átvitel másik végpontjával, a memóriával, illetve az értesítést váró folyamattal a hardver alaphelyzetben semmit sem tud kezdeni:

Fej sorszáma [H]	Melyik lemez, melyik oldalán található a keresett blokk?
Szektor sorszáma [S]	Melyik szektorban van az adat?
Cilinder sorszáma [C]	Melyik cilinderen van a blokk?
Írás vagy Olvasás	Az adatáramlás irányának meghatározása

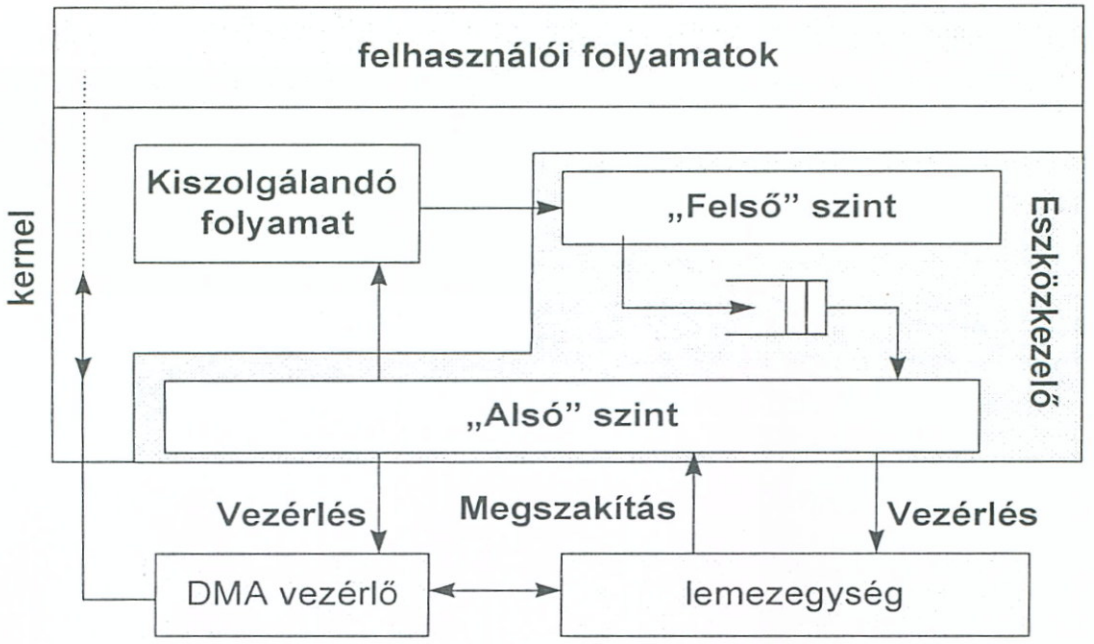
4.6 ábra A lemezegység számára szükséges adatok

A lemez eszközmeghajtójának a feladata ebben a megközelítésben nem más, mint a két táblázat közötti konverzió hatékony megvalósítása.

4.2.1 A lemez eszközmeghajtójának felépítése

A réteges felépítési elvnek megfelelően az eszközmeghajtó további funkcionális egységekre bontható. Legegyszerűbb, ha különválasztjuk a folyamatokkal kapcsolatot tartó részt, és az eszközzel kommunikáló részt. Egy lemezegységhez rengeteg kérés futhat be, ezért a két egység között várakozási sort kell kialakítani. Az alábbi példában az egyszerűség kedvéért egyetlen kérő folyamat szerepel.

A működés tehát nagy vonalakban a következő. A kiszolgálásra váró folyamat a 4.5 ábrán megadott adatstruktúrát átadja az eszközmeghajtó „felső" moduljának. A felső szint a kérést a várakozási sorba helyezi, azonban a várakozó folyamatok és a lemezegység állapotának ismeretében a sorban a hatékonyság, vagy prioritási szempontok figyelembevételével némi módosítást is végezhet. Az „alsó", hardver közeli szint a várakozási sorból kiemeli a soron következő feladatot, majd a lemezegységet a megfelelő blokkra pozícionálja, a DMA vezérlőt pedig feltölti a memóriatartomány kezdőcímével, az adatok mennyiségével és az adatátvitel irányával. (Egyszerűbb eszköz, kisebb adatmennyiség esetén nem lenne feltétlenül szükség DMA technikára, esetleg regiszterek is elegendőek lennének.) Az adatátvitel végét a hardver egy megszakítással jelzi az eszközmeghajtónak, az pedig a jó hírt továbbadja a kérő folyamatnak.



4.7 ábra A lemez eszközmeghajtó működése

Mit csinál a művelet közben a kérő folyamat? Két dolgot tehet. Vagy türelmesen várakozik, vagy a kérés átadása után azonnal folytatja egyéb teendőit.

Szinkron átvitelnek nevezzük a műveletet akkor, ha a kérő folyamat a művelet befejezéséig a várakozási sorban tartózkodik. Ekkor a folyamat nem érzékeli az adatátvitel sebességét, hiszen mikor újra futhat, már minden kért adat rendelkezésére áll.

Az **aszinkron átvitel** a másik alternatíva. Ez esetben a folyamat nem várakozik, viszont a folyamatnak is, az operációs rendszernek is komoly gondot okoz, hogy az éppen feltöltés alatt álló területek, és a források sem változhatnak az átvitel alatt. Külön gond, hogy egy folyamat az átvitel alatt akár be is fejeződhet, és nincs, aki felszabadítsa a foglalt memóriarészt.

Meg kell jegyezni, hogy az ábra szigorú határvonalai ellenére a hardver és a szoftver közötti átmenet elmosódott. Az eszközök gyakran igen nagy hardver támogatást adnak a fenti feladatok ellátásához, sőt gyakran maga az eszközmeghajtó, vagy annak „alsó" része is az eszközön elhelyezett ROM-ban található.

A vázlatos áttekintés után nézzük az egyes szinteket részletesebben.

4.2.2 Lemezütemezés - a meghajtó „felső" oldala

A felső szint feladata tehát a kérés átvétele, vizsgálata, és elhelyezése a várakozási sorban. Ha más, magasabb szempont nem szól bele a dologba, a kiszolgálás sorrendjét csak a várakozási idő (a fejmozgás) optimalizálásának célja vezérli. A kérések teljesítésének átlagos ideje mellett annak szórása is fontos paraméter. Kis várakozási idő nagy szórással jelentheti azt, hogy egyes folyamatok csak megengedhetetlenül lassan juthatnak adatokhoz.

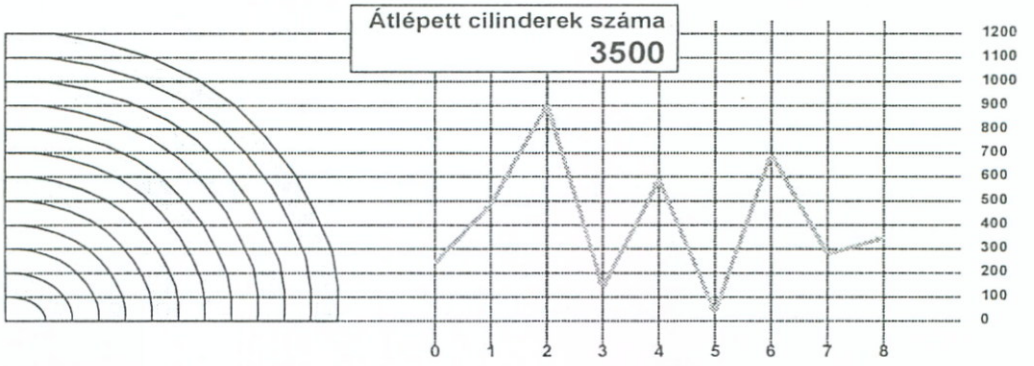
Azt különösebb elemzés nélkül is beláthatjuk, hogy a választott módszer alkalmazásától függetlenül a lemezek középső sávjai vannak a legkedvezőbb helyzetben, mivel bárhol álljon is a fej, éppen átlagosan középre juthat el a leghamarabb. A megállapítás következménye, hogy a gyakran használt adatok, például a virtuális memória lapjai célszerűen itt kell elhelyezkedjenek.

A különböző lemezütemezési algoritmusok más-más felhasználási módra szolgálnak. Példaképpen egy-egy tipikus tesztsorozat esetén vizsgálhatjuk meg viselkedésüket. Az alacsony szintű lemezütemezők lelki világába sem az alkalmazásfejlesztő, sem a felhasználó nem láthat bele, de a példából érzékelhető, hogy egy rosszul megválasztott algoritmus akár háromszorosára is növelheti ugyanazon kérések kiszolgálásának idejét!

4.2.2.1 Sorrendi kiszolgálás

A **sorrendi kiszolgálás** (**F**irst **C**ome **F**irst **S**erved - **FCFS**) a legegyszerűbb stratégia, amely igaziból nem is stratégia, a folyamatokat érkezési sorrendjükben szolgálja ki. Minden folyamat szóhoz jut, de közel sem optimális módon. Az algoritmus nem törődik a fej pillanatnyi állásával, így szerencsétlen esetben elképzelhető, hogy az idő legnagyobb része a fej mozgatásával telik el. Az FCFS módszer továbbfejlesztésének tekinthető a „felszedő" (pick up) eljárás, amely alapvetően sorrendi, de ha a fej mozgása közben egy kérés kielégíthető, az mintegy mellékesen megtörténik.

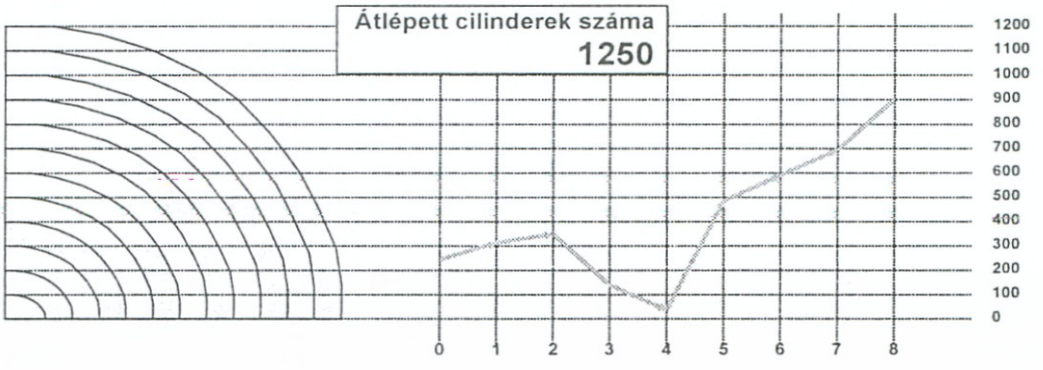
Eredeti sor	250	500	900	150	600	50	700	300	350
Átrendezett	250	500	900	150	600	50	700	300	350



4.8 ábra FCFS lemezütemezés

A **legkisebb elérési idő** módszere (**S**hortest **S**eek **T**ime **F**irst - **SSTF**), azt a kérést részesíti előnyben, amely kiszolgálása a legkisebb fejmozgással kielégíthető, azaz amelyhez tartozó adatblokkok az aktuális fejpozícióhoz a lehető legközelebb vannak. Ez a módszer eredményezi általában a legkisebb átlagos várakozási időt, de a várakozási idő szórása nagy, mivel fennáll a veszélye annak, hogy egyes, a sűrű kérések helyétől távol eső blokkokat igénylő folyamatokra csak bizonytalan idő múlva kerül sor (ha mindig van „közeli" kérés, a „távoli" kérésekhez sosem jutunk el - kiéheztetés).

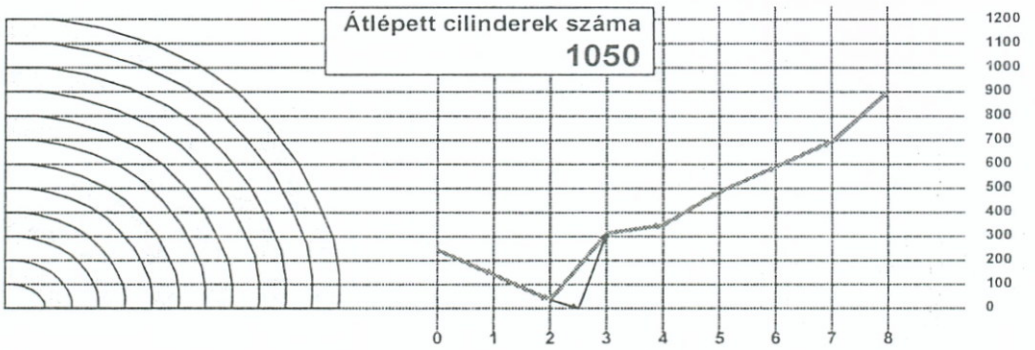
Eredeti sor	250	500	900	150	600	50	700	300	350
Átrendezett	250	300	350	150	50	500	600	700	900



4.9 ábra SSTF lemezütemezés

Pásztázó (Scan, Look) módszer alkalmazásánál a fej állandó mozgásban van, sorban elégíti ki a mozgási irányába eső kéréseket. A pásztázás iránya csak akkor fordul meg, ha az eredeti irányban már nincs kiszolgálandó kérés, illetve a fej elérte a szélső sávot. Az algoritmus gyengéje, hogy a rossz ütemben érkező, azonos, szélső fej pozícióra irányuló kérelmek kielégítése csak egy teljes oda-vissza mozgás elvégzése után lehetséges, így a szórás meglehetősen nagy. (A középső sávok fölé közel periodikusan érkezik a fej, míg a szélsők esetében egymáshoz időben közel kétszer érkezik a fej, majd nagyon sokáig nem - azaz amíg eljut a lemez túlsó felére és onnan vissza. De itt nincs kiéheztetés, legkésőbb egy oda-vissza mozgás után a kérés biztos kielégítésre kerül.)

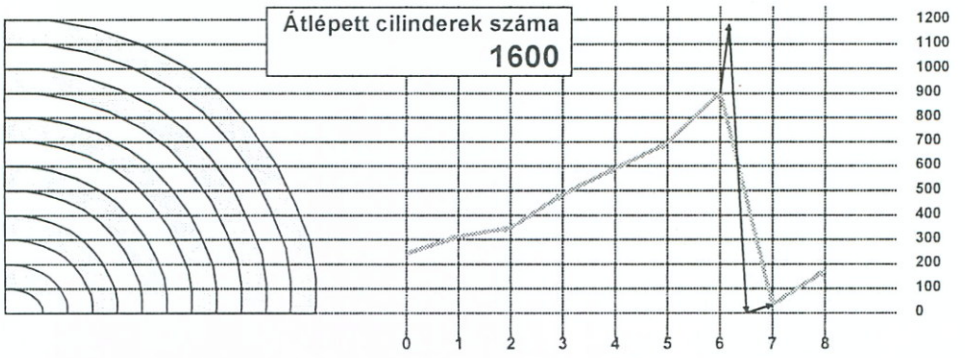
Eredeti sor	250	500	900	150	600	50	700	300	350
Átrendezett	250	150	50	300	350	500	600	700	900



4.10 ábra SCAN lemezütemezés

Az **egyirányú pásztázás** (**C**ircular **S**can, **C-Scan**) az előző utóbb említett problémáját küszöböli ki azáltal, hogy adatokat csak az egyik mozgásiránynál továbbít a fej. Ha a legtávolabbi kérés is kielégítésre került, a fej a legelső kérésre ugrik vissza, majd újra kezdi útját. A visszatérési idő alatt lehetőség van arra, hogy a vezérlő összeállítsa a következő pásztázás „menüjét", azaz a fej hasznos irányban való mozgása során nem kell gondolkodnia, így növelhető a sebesség.

Eredeti sor	250	500	900	150	600	50	700	300	350
Átrendezett	250	300	350	500	600	700	900	50	150



4.11 ábra C-SCAN lemezütemezés

Az alábbi összehasonlító táblázat adatai csalókák, minden a folyamatok kéréseinek sorrendjéről, érkezési gyakoriságától és eloszlásától függ. Az alkalmazandó optimális eljárás az adott környezet függvénye. Lehetséges a kérések sűrűségétől függően esetenként más-más módszer alkalmazása is.

Algoritmus	Várakozási idő	Várakozási idő szórása
Sorrendi – FCFS	Erősen függ a kérések sorrendjétől	
Legrövidebb idejű – SSTF	kicsi	nagy (kiéheztetés)
Pásztázó – SCAN	közepes	közepes
Egyirányú pásztázó – C-SCAN	közepes	kicsi

4.12 ábra Lemezütemező algoritmusok

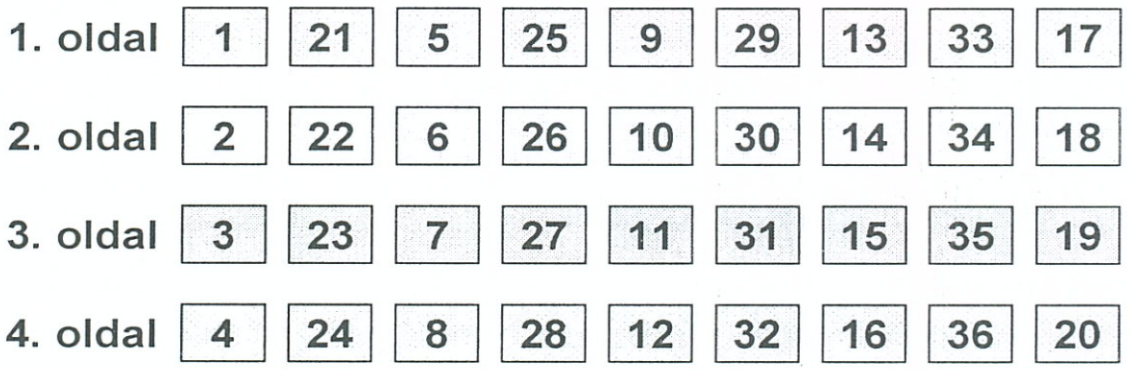
4.2.3 A címszámítás - az eszközmeghajtó „alsó" oldala

A folyamatok lineáris címzést szeretnek, a lemezegységek vektoros, három paraméterből állót. Az eszközmeghajtó egyik legfontosabb feladata a konfliktus feloldása. A blokksorszámot [b] a fej [h], a cilinder [c] és a szektor [s] sorszáma alapján a következő képlettel számíthatjuk ki (C a cilinderek, S a szektorok száma):

$$b = h * C * S + c * S + s$$

Ez az összefüggés sem túl barátságos, de még rontja a helyzetet, hogy ennek az inverzére van szükség, azaz a *b* ismeretében kell számítanunk a *h, c* és *s* paramétereket. Ha már ennyit kell számolni, kérdés, hogy nem lehetne-e valahogy ezt is az optimalizálás szolgálatába állítani?

A blokkokra sorszámuk alapján hivatkozhatunk, azonban ez a számozás, éppen az elérési és átviteli idők javítása érdekében, nem mindig követi az első ránézésre logikusnak tűnő rendszert. A számozásnál a geometriai jellemzők helyett az elérési idők dominálnak, így például az egymás alatti blokkok kapnak szomszédos számokat, hiszen ezek olvasása nem igényel sem lemez, sem fejmozgást, tehát igen gyors. Ha az adatátviteli sebesség a szűk keresztmetszet, lehetséges, hogy az egy sávon belüli szomszédos blokkok gyorsabban követik egymást, mint ahogy az előző blokk adatai feldolgozásra kerülnének. Ilyenkor a számozásnál egy vagy több blokk kimarad. Ez a közbeékelődő (interleave) technika. Az alábbi ábra egy 9 szektoros, 2 lemezből (4 lemezoldalból) álló, 1:2 inteleave-t használó lemezegység legkülső cilinderének számozását mutatja.



4 oldal, 9 szektor, 1:2 interleave

4.13 ábra Példa a blokkok számozására

4.2.4 Memória területek kiválasztása

Eddig kissé elhanyagoltuk azt a kérdést, hogy hová kerülnek, vagy honnan, a memória mely területéről származnak a perifériáról érkező, vagy oda irányuló adatok. A DMA vezérlő természetesen csak valóságos,

fizikai címekkel tud valamit kezdeni, hardver eszköz lévén számára értékelhetetlenek a virtuális memória lehetőségei.

4.2.4.1 Szinkron és aszinkron átvitel

Láttuk, hogy amikor egy folyamat lemezről (vagy hálózatról, vagy más perifériáról) kér adatot, vagy írni szeretne oda, a programozó döntésétől függően aszimmetrikus és szimmetrikus adatátvitel között választhat. Az elnevezések kissé megtévesztők. A lényeges különbség abban van, hogy az átvitelért elsősorban az operációs rendszer vagy a folyamat (azaz a programozó) a felelős.

Ha egy folyamat **aszinkron átvitelt** indított el, azaz az adatátvitel elindítása után nem kerül azonnal a várakozó listára, lehet olyan fizikai memória területe, melyet az átvitel rendelkezésére tud bocsátani. Láttuk azonban, hogy ez rengeteg adminisztratív kötelességgel jár. A folyamatnak vigyáznia kell, hogy az érintett területekre ne írjon, onnan ne olvasson, az operációs rendszernek pedig arra, nehogy időközben más folyamatnak adja át a választott tartományt.

Aszinkron átvitel esetén tehát a kérés (rendszerhívás) átadása után

- A folyamat **tovább fut**,
- **Megtartja erőforrásait** (az átvitel az FOLYAMAT területére történik),
- A biztonságért a **folyamat** a felelős.

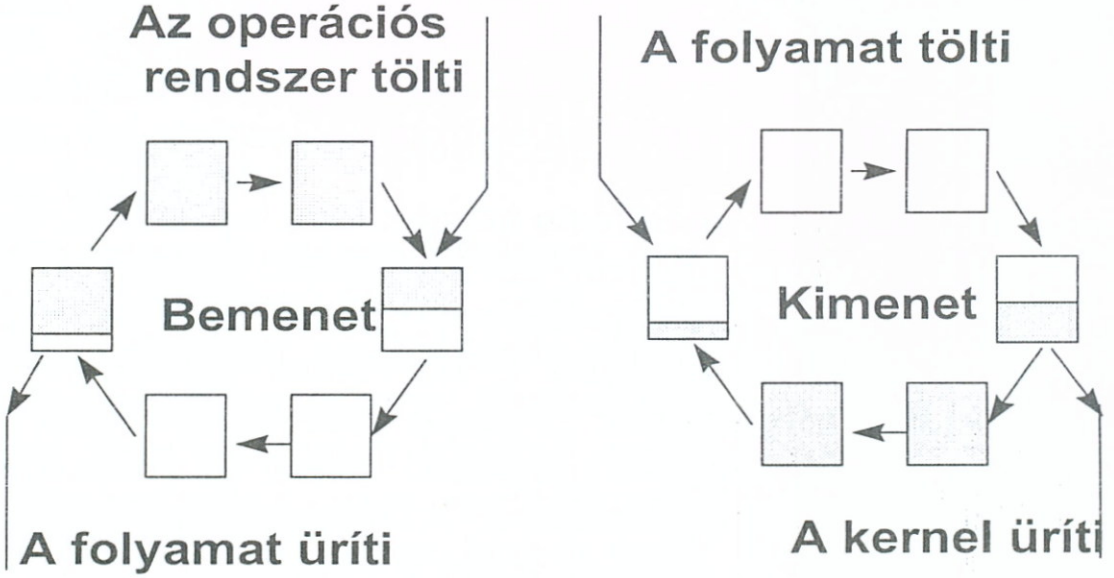
Szinkron átvitel esetén az átvitelt igénylő folyamat az adatátvitel kérést kifejező rendszerhívás után várakozik annak befejezéséig. Ilyenkor nincs többlet adminisztráció, viszont memória terület sincs, hiszen az operációs rendszer elveszi a várakozó folyamatoktól lapjaikat és átadja azokat más, futni képes folyamatoknak. Nincs más hátra, az operációs rendszernek kell átmenetileg területet biztosítania a várakozó folyamat adatainak, majd a művelet befejezése után át kell azokat másolnia a folyamat területére.

Szinkron átvitel esetén tehát a kérés (rendszerhívás) átadása után

- A folyamat **felfüggesztődik**,
- **Elveszti erőforrásait** (átvitel az OS területére),
- A felelősség a biztonságért az **operációs rendszeré.**

4.2.4.2 Átmeneti tárak (Buffer pool)

A felhasználói folyamat szintjén kínál megoldást az **átmeneti tárak** tömbjének (buffer pool) kialakítása a felhasználói folyamat memóriaterületén. Bufferek segítségével úgy valósítható meg aszinkron adatátvitel, hogy az sem a folyamat, sem a rendszermag számára nem jelent sok többlet terhet. A másik előny, hogy a folyamat maga tudja eldönteni, hogy mennyi területre van szüksége, és nem kell versenyeznie az operációs rendszer buffereiért. Általában külön bufferek kialakítása célszerű az írási és az olvasási műveletek megvalósítására. A nyilvántartást tovább egyszerűsíti, ha az egy blokknyi méretű tároló helyeket körkörösen alakítjuk ki.



4.14 ábra Körkörös átmeneti tárolók

4.2.4.3 Lemezgyorsítás (Disk caching)

Szinkron átvitelnél tehát feltétlenül szükség van az operációs rendszer támogatására, azonban ebből kis ráfordítással még előny is kovácsolható.

Az adatátvitel esetén mindig gondot okoz, ha a kommunikáló eszközök sebessége különbözik, így van ez a lemezegységek esetén is. A következőkben a különböző lehetséges esetekre szolgáló megoldásokat foglaltuk össze. Fontos kiemelni, hogy a buffer alkalmazása csak bizonyos adatmennyiségig, adatcsomag méretig megoldás. A gyakorlatban természetesen a módszerek kombinációjával találkozhatunk.

Azt az időt, amely alatt a kívánt adatok megjelennek az adott memória területen, lényegében két tényező határozza meg:

A **beolvasási sebességet**, a mágnesesen rögzített jelek megtalálásához szükséges időt a meghajtó mechanikai jellemzői korlátozzák (aláfordulási idő, fejmozgás).

Az **átviteli sebességet**, azaz a lemezmeghajtó átmeneti tárolója és a memória közötti átvitel idejét elsősorban az elektronikus jellemzők határozzák meg.

A két sebesség viszonya határozza meg az optimalizálás technikáját, eszközeit:

- Beolvasási > Átviteli → Közbeékelő (interleave) technika
- Átviteli > Beolvasási → Szalag (stripping) technika
- Beolvasási ≠ Átviteli → Buffer (csak „burst" jelleggel)

Ha az operációs rendszer adatterületén hozunk létre buffereket, és azokba nemcsak a kívánt blokkot, hanem az azt követő néhány blokkot is beolvassuk, a lokalitási elv értelmében „előre dolgozunk", mivel feltehetően a folyamat legközelebb a következő blokkokat szeretné olvasni. Írás esetén a folyamatnak elegendő csak az operációs rendszer bufferébe tölteni adatait, kiadni a megfelelő címet, és utána nyugodtan rábízhatja magát a kernel folyamataira. A módszer rokonságot mutat mind a hardver gyorsító tár (cache), mind a virtuális memória lapcsere algoritmusával.

A **rendszerszintű buffer** (Disk Cache) alkalmazásakor a háttértárhoz forduló folyamat először a kernelhez küld rendszerhívást, az azonban nem továbbítja ezt azonnal az eszközvezérlőnek, hanem megnézi, hátha a saját buffereiben megvan a kívánt adat. Ha igen, elvégezhetők a módosítások minden lemezművelet nélkül, azonban a változtatás csak a memóriabeli másolatot érinti! Az operációs rendszernek számon kell tartania a változtatásokat, és időről időre, vagy legalábbis a folyamat megszűnésekor szinkronizálni kell a memóriaképet a valóságos háttértár tartalommal. Egyes megvalósításokban az olvasási és írási bufferelés külön-külön engedélyezhető.

A lemezgyorsító, bufferelő algoritmust néha külön program valósítja meg (MS-DOS, Smartdrive), de gyakran az operációs rendszer szerves része (NetWare, Windows).

4.3 Az adattárolás optimalizálásának más módszerei

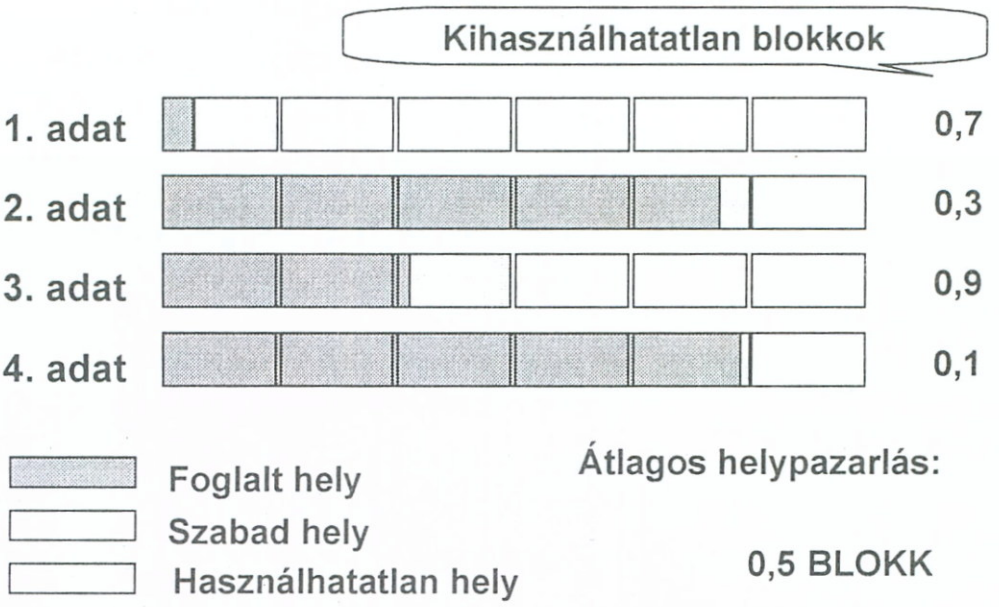
A háttértárak hatékonyságának legfőbb jellemzői: a tárolható adatmennyiség és az átviteli sebesség nagysága, illetve a biztonság. Azaz nagyon sok adatot szeretnénk tárolni, azokat nagyon gyorsan elővenni, ha kell, de úgy, hogy az adatvesztésnek vagy torzulásnak a veszélye minimális legyen. Az alábbiakban tárgyalt elvek és módszerek már nem tisztán hardver megoldások, a szoftver, elsősorban az operációs rendszer támogatását is igénylik.

4.3.1 Blokkméret optimalizálása

A winchestereknél láthattuk, hogy a blokkméret 0,5 kB és 64 kB között változhat. De minek a függvényében? A blokkméret a merevlemezek formattálásánál végleges értéket kap, ezért kialakításánál (ha ez egyáltalán lehetséges) valamiféle előismeretre van szükség. Mi is a probléma lényege?

Az adatállományok mérete általában nem egyezik meg pontosan egy blokk méretével. Vagy kisebb, vagy nagyobb. A lemezegység azonban blokkokat tud kezelni, annál kisebb adategység számára nem létezik. Az adatcsoportok a lemezen annyi blokkot foglalnak le, amennyi még éppen szükséges, azaz a lefoglalt blokkok együttes mérete éppen egyenlő az adatcsoport méretével, vagy nagyobb annál. Az egyenlőségnek nagyon kicsi az esélye, ezért szinte biztos, hogy a legutolsó blokk jó része kihasználatlan, üres marad. Ha az adatcsoportok méretét véletlenszerűnek tekintjük, könnyen belátható, hogy egy-egy adatcsoport átlagosan egy fél adatblokknyi területet hagy kihasználatlanul.

4.15 ábra Adatok tárolása lemezen



Ha az ábra példájával élünk, 1 kB-os blokkméret esetén összesen 2 kB, 64 kB-os blokkméret esetén 128 kB ment veszendőbe. Kézenfekvőnek látszik a megoldás, minél kisebb blokkméretet kell alkalmazni. De túl szép lenne, ha ilyen egyszerű lenne.

Az operációs rendszereknek fenn kell tartaniuk egy táblázatot amelyben adminisztrálják, hogy egy blokk foglalt vagy szabad. Ezt a táblázatot magán a lemezen is tárolni kell, de mivel nagyon sűrűn van rá szükség, működés közben az operatív memóriában is kell tartani egy másolatot belőle. A táblázat blokkonként legalább egy bitet kell tartalmazzon, melynek értéke 1, ha foglalt, 0, ha szabad. Az alábbi példa egy 1 GB-os, azaz 2^{30}-os winchesterről szól.

A lemez kapacitása 1 GB = 2^{30} bájt

Blokkméret		Foglaltsági tábla mérete	
512 bájt	2^9 bájt	$2^{30}/2^9/2^3=2^{18}$	256 kbájt
2048 bájt	2^{11} bájt	$2^{30}/2^{11}/2^3=2^{16}$	64 kbájt
64 kbájt	2^{16} bájt	$2^{30}/2^{16}/2^3=2^{11}$	2048 bájt

4.16 ábra A foglaltsági tábla mérete

Túl kicsi blokkméret választása esetén takarékoskodunk tehát a lemez férőhellyel, de pazarlóan bánunk az operatív tárral. Nagyméretű foglaltsági tábla esetén a keresés is lassabb lesz, terjedelmesebb táblázatot kell átnéznie az operációs rendszernek.

A blokkméret helyes megválasztása tehát nem más, mint optimumkeresés a lemez férőhely, és az operatív memória pazarlása között. Egyes operációs rendszerek nem is teszik lehetővé a módosítást (MS-DOS). Mások (pl. NetWare) a választási lehetőségek felajánlása mellett egy olyan szolgáltatást is nyújtanak, melynek segítségével a fennmaradó helyek is részben hasznosíthatók (block suballocation).

4.3.2 Adattömörítés

Az adattömörítés mind a tároló kapacitás növelésére, mind az adatátviteli sebesség növelésére megoldást kínál. A tömörítés a felhasználó programok szintjén is megvalósulhat (pkzip, arj), azonban most azt az esetet vizsgáljuk, amikor maga az operációs rendszer vállalja magára ezt a feladatot, a visszaállítás beolvasáskor automatikusan megtörténik.

A tömörítésnek sok módja ismert. Léteznek veszteséges tömörítő eljárások, melyek a hang vagy kép esetén igen hatékonyak lehetnek, de természetesen nem alkalmazhatók programok esetén, itt egyedüli megoldás a veszteségmentes tömörítési eljárások használata. Az alábbiakban három egyszerű eljárást ismertetünk vázlatosan, gyakran használják ezek kombinációit is.

1. A **futási hossz kódolás** (Run Length Encoding) olyan esetekben alkalmazható a legjobban, ahol egy adatmezőben nagyon sok egyforma elem van. Lehet például egy adatállomány, melyben sok a nulla érték, és csak olykor fordulnak elő értékes számjegyek. Ha például egymás után 30db '0' karakter következik, egy speciális, ritkán előforduló karakter után (ez általában a ESC karakter) elegendő megadni a 30-as számot és mindössze egyszer a '0'-t. Ezzel az egyszerű módszerrel a jelen példában 90%-os csökkenést értünk el! A helyzet persze nem mindig ilyen szép, a lehetséges tömörítés mértéke erősen függ az adatok jellegétől. Ha véletlenül mégiscsak előfordulna a választott speciális karakter, azt annak kettőzésével jelezhetjük a kicsomagoló algoritmusnak.

2. A **különbségi kódolás** (Difference Encoding) segítségével lassan, fokozatosan változó adatok esetén érhető el jó eredmény. Ilyen lehet egy szép kék égboltot ábrázoló fénykép, illetve annak digitalizált változata. A módszer lényege, hogy nem magát az adatot, hanem csak a változást tárolja. Ha a változás kicsi, kevesebb bit is elegendő, tehát rövidebb az ábrázolási forma. Legyen például egy bájtsorozatunk, egy egyesével növekvő számsorozatot ábrázoltunk 1-től 128-ig. A differenciális kódolás szerint a sorozatból csupa egyes lesz, mely szélsőséges esetben egyetlen biten is ábrázolható, tehát a tömörített ábrázolásra 128/8, azaz 16 bájt elegendő.

3. A **Huffmann-kódolás** az előző, nagyon speciális esetben igazán hatékony módszerekkel szemben széles körben alkalmazható (így működnek a fax készülékek is). A módszer lényege egy kód összerendelés, mely a tömörítetlen adatok között gyakrabban előfordulókhoz rövidebb, a ritkábbakhoz hosszabb kódot rendel. A működés illusztrálására álljon itt a következő példa:

Eredeti szöveg: **KEREKES SZEKEREK MENNEK**

Statisztika, kódolás:

8 db E	00
5 db K	01
2 db R	10
2 db S	1100
2 db N	1101
2 db space	1110
1 db M	11110000
1 db Z	11110001

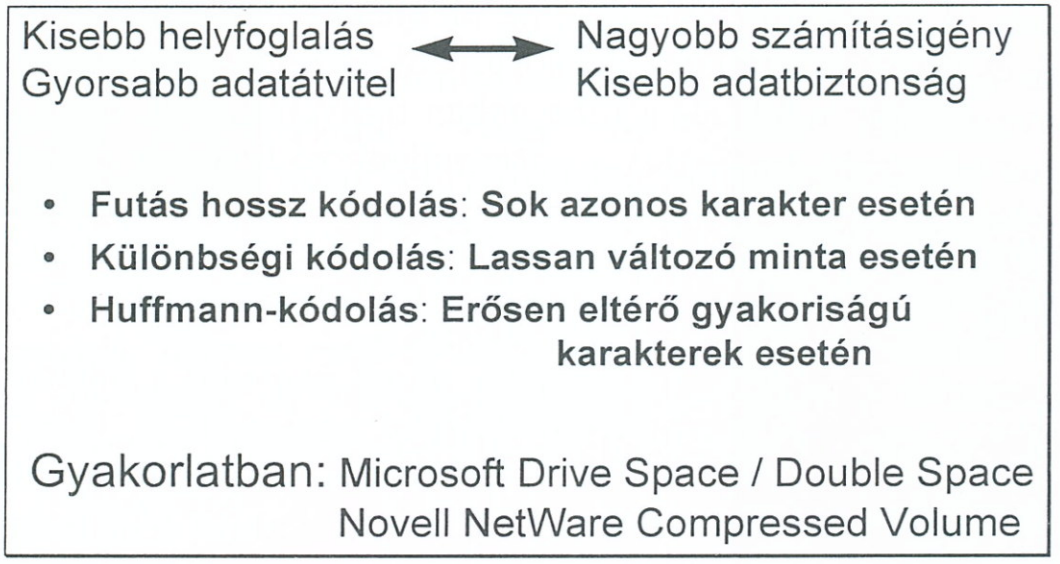
Hatékonyság:

Eredeti szöveg:
184 bit

Kódolt szöveg
70 bit

4.17 ábra Példa a Huffmann-kódolásra

A módszer tehát működik. Figyeljük meg, hogy az 'S', 'R' és 'space' karakterek kódjában szereplő 2 db 1-es, illetve az 'M' és 'Z' 4 db 1-e a dekódoló programnak szóló információ, azt mutatja meg, hogy az utána következő hány bitet kell együtt kezelni. A módszer szembetűnő hátránya a számításigényesség (ezért operációs rendszer szinten csak különleges esetekben alkalmazzák), és a kódtáblát is tárolni kell valahol.



4.18 ábra Tömörítési eljárások

Összefoglalva, a tömörítési módszerek tehát kisebb helyfoglalást és gyorsabb adatátvitelt eredményeznek, mely előnyökért cserébe számításigényesek. További hátrány, hogy a tömörítési eljárások csökkentik a redundanciát – ezért képesek veszteség nélkül tömöríteni – de ezzel csökkentik az adatbiztonságot. Egy sérülés egy hosszabb szakasz helyreállíthatatlan hibájához vezethet.

A gyakorlatban használt ismertebb, operációs rendszer szintű tömörítő módszerek az MS-DOS-hoz kínált Double Space, mely a külön termékként forgalmazott, hardver platformon is létező SpedStore alapján készült. Ennek módosított változata a Drive Space már a Windows95 alatti rendszerekben is működik. A NetWare 4-es verziójától szintén lehetséges a kötetek tömörítése.

Megemlítendő, hogy a tömörítés és titkosítás rokon eljárások, csak céljuk más. Azonban míg a tömörítéssel találkozhatunk az operációs rendszerek szintjén is, a titkosítást szinte kizárólag az alkalmazások szintjén végzik.

4.3.3 Megbízhatóság, redundancia

Minél kevésbé megbízható egy eszköz, vagy általánosabban adatátviteli rendszer, annál nagyobb szükség van az adatok védelmére, a hibák javítására, vagy legalábbis detektálására.

A hibadetektálás lehetséges legegyszerűbb eszköze a paritásbit használata. A **paritásbit** értéke 1, ha az ellenőrzött adatblokkban páros számú 1-es van, 0, ha páratlan. Ha az átvitel során az adat megsérül, vagy már eleve rosszul indult, és a hiba abban nyilvánul meg, hogy egyetlen adatbit az ellenkezőjére fordul, a paritásellenőrző áramkör észreveszi, és módosítást kérhet. Kettős hibát a módszer nem vesz észre. Paritásbitet a mágnesszalagokon kívül általában az operatív tár védelmére is használnak.

Minél nagyobb a redundancia mértéke, annál nagyobb a hibadetektálás, hibajavítás lehetősége. Gondolatkísérletként képzeljük el, hogy minden egyes bájtot megkettőzve tárolunk. Ekkor mind a nyolc bit hibája igen nagy valószínűséggel felderíthető, de a javításra nincs lehetőség, hiszen csak az eltérést figyelhetjük meg, de nem tudjuk, melyik bájt a jó, és melyik a rossz. Növeljük tovább a redundanciát, és adjunk az eddigi kettő mellé egy harmadik bájtot! Ekkor jó eséllyel minden bit javítható is. Könnyen belátható, hogy a biztonság és a tömörség egymásnak ellentmondó irányzatok, nehéz optimumot találni. Az adatok megháromszorozásánál azért vannak hasonló biztonságú, de kevésbé helyigényes módszerek.

Hibajavító kódok alkalmazhatók, ha a várható hibák egymástól függetlenek. Általánosan elmondható, hogy az $n+1$ bitből álló hibajavító kód alkalmas $n+1$ db hiba detektálására, és n db hiba javítására. A hibajavítás alapja általában az úgynevezett Hamming-távolság, azaz azt az adatot tekintjük jónak, amely megfelel a hibajavító bitek állásának, és a rossz adattól a lehető legkevesebb bitben tér el. Az optikai egységek olvasási hibalehetőségei meglehetősen nagyok, ezért a CD-ROM olvasóknál 6 db hibajavító bitet használnak.

Ellenőrző összegek (**C**yclic **R**edundancy **C**heck - **CRC**) használatosak akkor, ha a várható hibák nem függetlenek egymástól, hanem egy adatfolyam egymást követő bitjei sérülnek, például egy hálózati zavar, vagy mechanikai sérülés hatására. CRC-t használnak - többek között - a lemezegységek blokkjainak, illetve a mágnesszalagok rekordjainak védelmére.

A hibajavítás fenti formái működés közben nem igénylik az operációs rendszer közreműködését, ahhoz már csak akkor fordulnak, ha nagy a baj. A lemezegységek például hiba észlelése esetén önállóan újraolvasással kísérleteznek, egészen addig, míg a próbálkozások száma vagy ideje el

nem éri az előírt maximumot, csak akkor fordulnak megszakításkéréssel az operációs rendszerhez.

Eddig az adatok hibáival foglalkoztunk, de előfordulhat, hogy az egész lemezegység meghibásodik. Nagy és fontos adatokat kezelő rendszereknél az ilyen hibák sem okozhatnak válsághelyzetet.

A **lemez tükrözés** (mirroring, duplication) esetén minden lemezből kettő van, és minden lemezzel kapcsolatos művelet mindkét egységen párhuzamosan, azonos módon hajtódik végre. Az operációs rendszer figyeli az eltéréseket, és hiba esetén megkísérli a javítást. A tükrözés elve kiszélesíthető a lemezegységen felül a vezérlőkártyára, sőt az egész gépre is.

A **RAID** (**R**edundant **A**rray of **I**nexpensive **D**isks) a probléma egy másik megközelítése. Ebben az esetben olcsó lemezek együttműködő tömbje végzi az adatok tárolását úgy, hogy az adatblokkok meghatározott rendszer szerint megoszlanak a lemezek között. A RAID detektálja, és jelzi egy egység meghibásodását, azonban az adatok szétterítettsége és redundanciája miatt az egész rendszer maradéktalanul működőképes marad. A kicserélt lemezegységre az adatok automatikusan kerülnek föl. A RAID erős operációs rendszer támogatást igényel, sőt az ilyen egységek általában saját processzorral és operációs rendszerrel is rendelkeznek.

A megfelelő biztonsági minősítésű operációs rendszerek (Novell, Windows NT stb.) mindegyike támogatja mindkét ismertetett módszert.

- **Adatszintű védelem**
  - **paritásbit - egyetlen bithiba**
  - **hibajavító kód - független hibák**
  - **CRC - összefüggő hibák**
- **Eszközszintű védelem**
  - **lemeztükrözés - lemez megkettőzése**
  - **RAID - adatok redundáns elosztása**

4.19 ábra Az adattárolás biztonságának növelése

4.4 Korszerű tároló architektúrák

Egy személyi számítógépnél egy háttértár (merevlemez) meghibásodása nagyon fájdalmas veszteségeket okoz, és általában éppen a legrosszabbkor következik be. Nagyobb rendszereknél egy ilyen meghibásodás sok ezer felhasználó ellehetetlenülését okozhatja, cégek mehetnek tönkre, dolgozók ezrei kerülhetnek utcára (lásd a World Trade Center katasztrófájának következményeit). Ráadásul, a fokozott megbízhatóság mellett olyan nagy tárolókapacitásra van szükség, amely a PC-knél alkalmazott módszerekkel megoldhatatlan. A továbbiakban lássuk ennek a problémakörnek egy-két fontosabb aspektusát!

4.4.1 Nagy tárolórendszerek jellemzői

A nagy rendszereknél is természetesen fontosak a hagyományos szempontok:

- Kapacitás (tera/peta bájt nagyságrendű)
- Adatátviteli sebesség (>100 Mbyte/sec)
- Elérési idő (minimális, de nagyon változó)

Kiemelkedő, talán még a fentieknél is fontosabb szerepet kap azonban a:

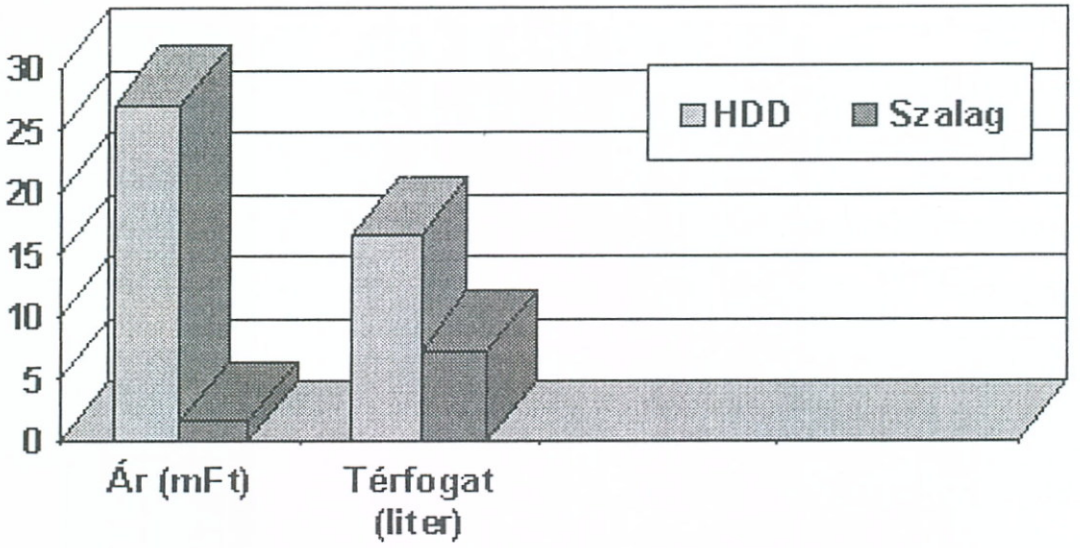
- Megbízhatóság, rendelkezésre állás
- Rugalmas konfigurálhatóság, bővíthetőség
- Menedzselhetőség

A személyi számítógépekből, munkaállomásokból megismert rendszerek ezeket a szempontokat egyáltalán nem, vagy csak jelentős korlátokkal képesek kielégíteni.

- Korlátozott kapacitás: A klasszikus buszrendszerek meglehetősen korlátozott számú eszközt képesek kezelni (IDE: 2, SCSI: 15), ennyi eszközből pedig csak erősen korlátos tárolókapacitás építhető ki.
- Rögzített konfiguráció, platform függőség: A hagyományos lemezkezelés nem teszi lehetővé, hogy egy-egy partíció méretét akár működés közben megnöveljük, illetve csak nagyon korlátozott lehetőségeink vannak arra (pl. Samba), hogy ugyanazt az egységet más-más operációs rendszer alól elérjük.
- Mentés helyi hálózaton (LAN): A mentések általában helyi hálózaton keresztül történnek, ezért azt könnyen túlterhelhetik. Bizonyos – nem is túlságosan nagy – lemezkapacitás felett a hálózat sebessége válhat a mentések sűrűségének, teljességének korlátjává.
- Meghatározott szállítók (nem nyílt rendszer): Egy állandóan bővülő és fejlődő rendszernek nem szabad olyan komponensekből felépülnie, amely az üzemeltetőt néhány szállítóhoz köti. Az igazán rugalmas rendszer nemzetközi szabványokon alapuló protokollok segítségével kell működjön.

Nagy tárolókapacitás megfelelő megbízhatósággal, gazdaságosan ma csak szalagos tárolók segítségével építhető. Az alternatívaként felvetődő a merevlemez tömbök, vagy optikai tárolók jelenleg – és úgy tűnik még egy jó darabig – nem helyettesítik a szalagokat. Nézzünk egy példát! Példánkban 2000GB kapacitást valósítunk meg 36 GB-

os SCSI merevlemezekkel (léteznek már 250GB-os modellek is, de a nagy megbízhatóságot igénylő helyeken még mindig ezeket használják), illetve 25GB-os adatszalagokkal.



4.20 ábra A szalagos és a merevlemezes tárolás összehasonlítása

Az ábra magáért beszél. Árban óriási a különbség, de még helyigényben is a szalagok vezetnek. És akkor még nem is beszéltünk olyan „apró" kérdésekről, mint az áramfelvétel vagy a disszipáció. (A szalagos rendszereknél az éppen nem használt szalagok gyakorlatilag egy polcon tárolódnak, tehát üresjáratban nem fogyasztanak. Igény esetén egy robot helyezi el a megfelelő szalagot egy olvasóban.)

A szalagos egység alkalmazása azonban egy jelentős hátránnyal jár – és itt kap szerepet a munkaszervezés, optimalizálás, azaz az operációs rendszer –, ugyanis a szalagcseréhez, pozicionáláshoz szükséges 20-40 másodperc sok esetben megengedhetetlen.

Mint mindig, ha igencsak eltérő sebességű eszközök közötti adatáramlás optimalizálásáról van szó, gyorsítótár (cache) technikát alkalmazhatunk. A HSM (Hierarchical Storage Management) rendszerek éppen ezt valósítják meg azáltal, hogy figyelik az állományok használati statisztikáit, és a leggyakrabban hivatkozott állományokat merevlemezeken tartják. Ezzel a felhasználói folyamat szempontjából

- A látszólagos tárolókapacitás megegyezik a szalagok összkapacitásával
- Az elérési idő a gyakrabban használt állományok esetén megegyezik a merevlemez által biztosított értékkel.

Az elemek összetevőinek (szalag/szalagolvasó/lemez) megfelelő száma, aránya esetén a rendszer a legtöbb igényt maximális sebességgel elégíti ki, és szinte elképzelhetetlen kapacitást biztosíthat.

4.4.2 A tárolórendszerek megbízhatósága

A megbízhatóság növelésétől azt várjuk, hogy az esetleges meghibásodás a munkát ne tegye lehetetlenné, a hibát automatikusan jelezze. Alapvetően két módszert alkalmazhatunk:

- garantált minőségű (és ezért igen drága) elemeket használunk (és itt a garancia nemcsak az eszközre értendő, hanem annak tartalmára is!)
- olcsóbb elemeket használunk ugyan, de az egyes összetevőket többszörözzük (azaz redundáns rendszer hozunk létre).

Gyakorlatilag csak a második út a járható. A redundancián alapuló megbízható lemezrendszereket RAID (Redundand Array of Inexpensive Disks) nevezzük. Az egyes implementációkat számokkal jelölik. Létezik Raid-0..5, sőt az egyes módszerek kombinációjaként lehetséges például Raid 50 is. Leggyakrabban a Raid-1-et, illetve a szinte egyenértékű Raid-3 vagy Raid-5 valamelyikét alkalmazzák. Nézzük a legfontosabb jellemzőket!

- RAID 1 - a lemeztükrözés
  - Minden adatból fizikailag kettő van,
  - Sérülés esetén az épen maradt lemez működik,
  - kétszeres háttértár kapacitást kell kiépíteni.
- RAID 3 vagy RAID 5
  - A logikai adat blokkokat fizikailag több lemezegységen elosztva tároljuk,
  - a szétosztott blokkokhoz ellenőrző összeget rendelünk, amelyet egy további fizikai eszközön tárolunk.

A RAID-1 nem szorul magyarázatra: hiba esetén az operátor jelzést kap a hibáról, és a rendszer a megmaradt jó lemezével zavartalanul folytatja a működést. Ha a rendszer támogatja a „hot swap", azaz melegtartalék funkciót, a tartalék 3. lemez, amely eddig nem vett részt a folyamatokban, automatikusan a meghibásodott lemez helyére lép.

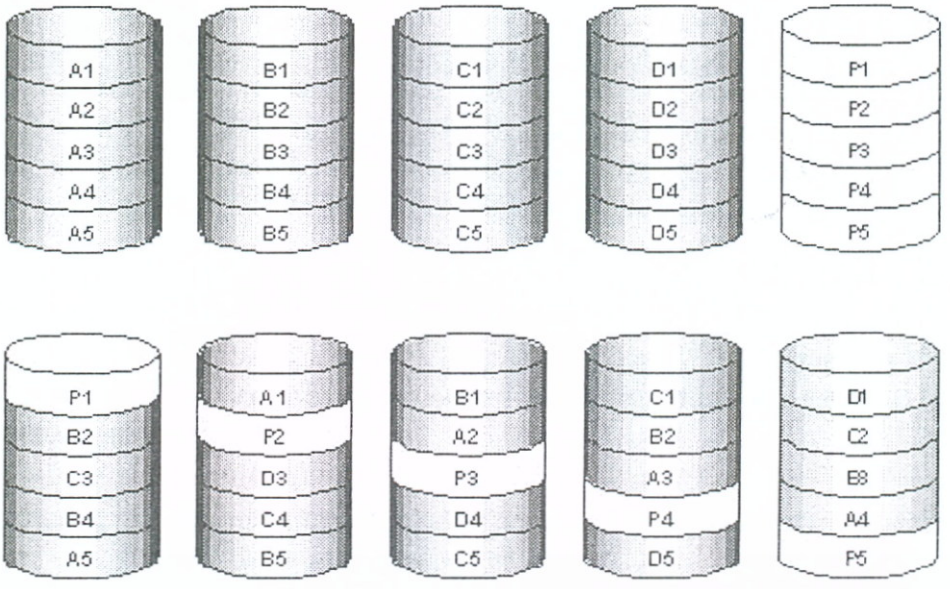
A Raid-3/5 működésének illusztrálására tanulmányozzuk a következő ábrán szereplő példát! Tegyük föl, hogy van négy darab számunk (A, B, C, D), amit szeretnénk nagy biztonsággal megőrizni, és rendelkezésünkre áll egy 5 lemezből álló tárolórendszer, melyen hardver vagy szoftver módszerrel RAID-et alakítottunk ki. Tároljuk a négy számot négy különböző lemezen, az ötödik lemezre pedig írjunk egy ellenőrző összeget (az egyszerűség kedvéért) a négy szám összegét!

Vegyük észre, hogy bármely lemezegység hibája esetén a $P = A + B + C + D$ összefüggés megfelelő átrendezésével annak tartalma visszaállítható.

1.lemez	2.lemez	3.lemez	4.lemez	5.lemez (paritás)
A=35	B=12	C=20	D=71	P=138

A fenti ötletnek kétféle megvalósítása létezik:

- A paritásokat tárolhatjuk mindig ugyanazon a lemezen (RAID-3)
- A paritásokat ciklikusan elosztva tároljuk az öt lemezen (RAID-5)



4.21 ábra RAID-3 és RAID-5 összehasonlítása

A két módszer között az a leglényegesebb különbség, hogy a RAID-3 gyorsabb (de az 5. lemez terhelése jóval nagyobb, mint a többi négy lemezé, a RAID-5 a lemezeknek egyenletesebb terhelést biztosít (de a bonyolultabb elhelyezés miatt lassabb).

Hasonlítsuk össze az ismertetett módszereket!

- RAID-1 (tükrözés)
  - A leggyorsabb, a legnagyobb helyigényű
  - A legdrágább megoldás
  - Tipikus alkalmazás: Rendszerlemezek
- RAID-3 (paritás külön eszközön)
  - Nagy állományok, kevés tranzakció
  - Olcsóbb, mint a tükrözés
  - Tipikus alkalmazás: digitális videó
- RAID-5 (elosztott paritás)
  - Kis állományok, sok tranzakció,
  - A lemezek terhelése egyenletesebb
  - RAID-3-nál lassabb
  - Tipikus alkalmazás: adatbázisok, általános fájlszerver

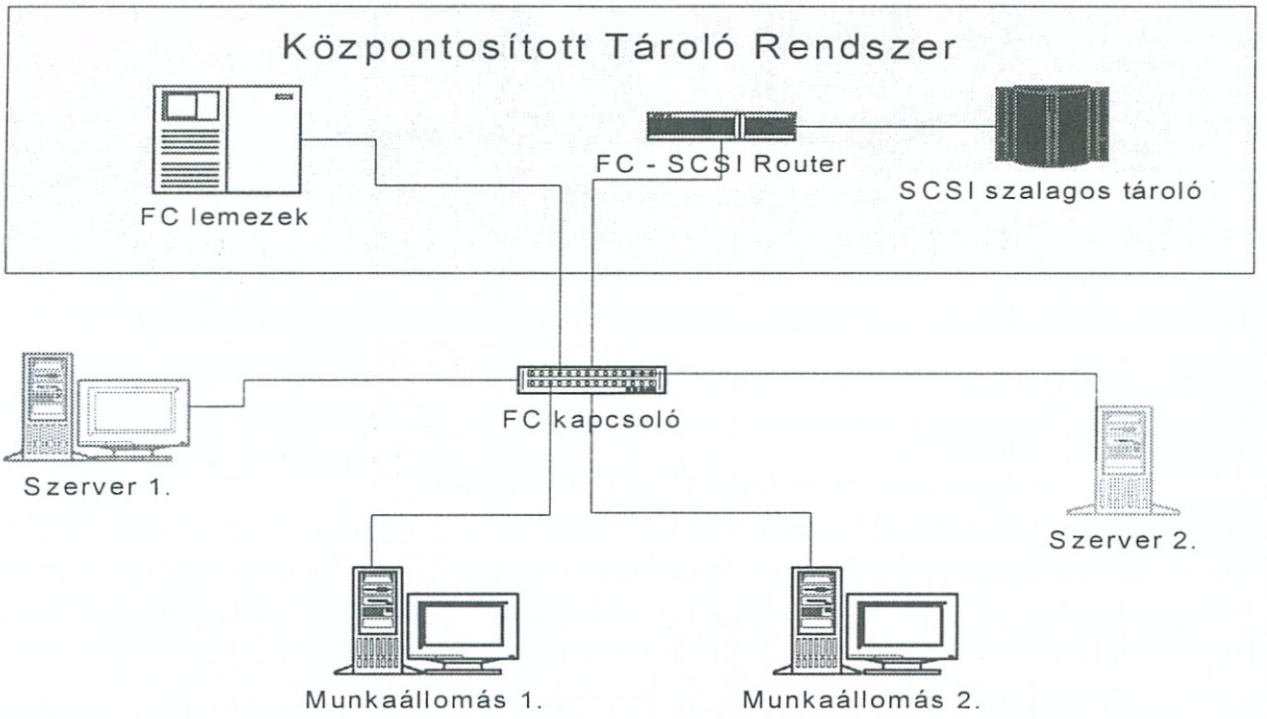
Jelentős változás figyelhető meg a tárolókat összekötő buszrendszerek fejlődésében is. Nagy rendszereknél az IDE interfész elképzelhetetlen, az SCSI korlátozott. A manapság legkorszerűbbnek tartott megoldás az 1-2-4 Gbit/s átviteli sebességű, garantált

sávszélességet biztosító, gyakorlatilag minden gyakrabban használt protokollt támogató Fibre Channel (FC) technológia.

4.4.3 Hierarchikus tároló architektúrák

A hagyományos megoldás szerint a szerverek és a munkaállomások önálló tátolórendszerrel rendelkeznek, közöttük az adatforgalom közönséges helyi hálózaton zajlik. A megoldás működőképes (bár a közönséges 100 Mb/mp Ethernet hálózat „rosszul viseli" az 50 Mb/mp-es garantált sávszélességet igénylő adatfolyamokat), rendkívül rugalmatlan, és nehezen változtatható.

Az alábbi ábrán vázolt, központosított tárolórendszert megvalósító architektúra az ún. SAN (Storage Area Network) más elvekre épül. Minden kapcsolódó eszköz a számára kijelölt méretű tárolóterületet látja, és olyan gyorsan érheti el, mintha az az elérhető leggyorsabb eszközökkel, a saját gépében lenne megvalósítva.



4.22 ábra Storage Area Network (SAN) architektúra

A SAN előnyei kézenfekvők:

- igen rugalmasan konfigurálható,
- gyakorlatilag korlátlanul bővíthető
- védett helyen, az archívumtól akár több kilométerre elhelyezhető
- nem függ az operációs rendszertől
- önállóan, a rendszer többi részétől függetlenül menedzselhető

- a mentések központosítottan megoldhatók.

A SAN elsődleges felhasználása az alkalmazás szerverek háttértárral való ellátása nagysebességű, hibatűrő hálózaton keresztül. Lényeges, hogy a fizikai háttértár terület kerül felosztásra, tehát nem az adatok, az állományok, így jogosultságok definiálásának sem értelme, sem lehetősége nincs. A hozzáférés szabályozás hiánya elég súlyos hátrány, azonban egy, a SAN-hoz nevében is kapcsolódó rendszer ezt a hátrányt is kiküszöböli.

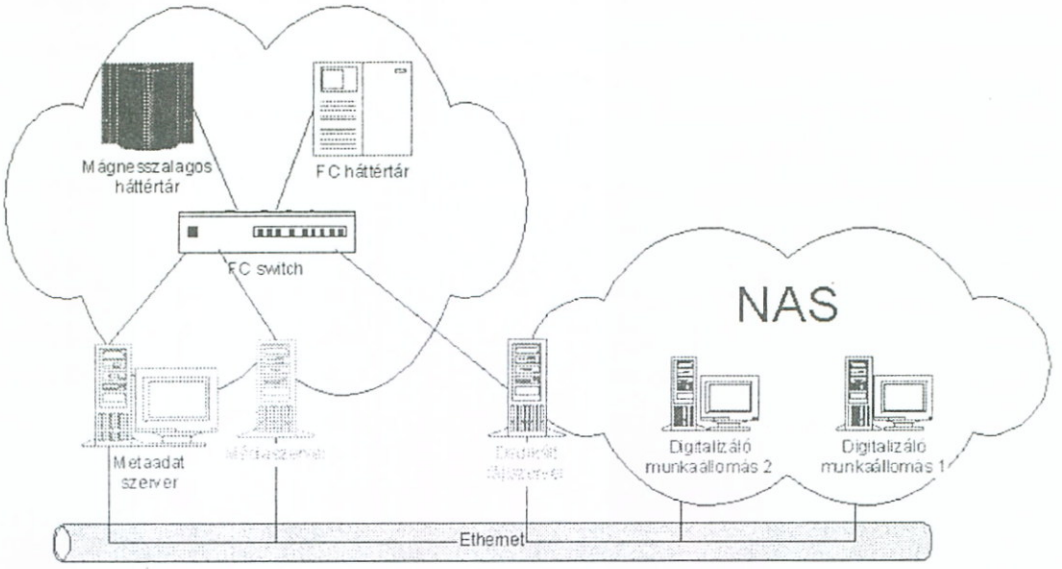
A hátrányokat kiküszöbölendő (és mellesleg a megtanulandó betűszavak számát növelendő) egy más architektúra mellett tették le a voksot. A NAS (Network Attached Storage) feladata az, hogy a kliensek számára a hagyományos kommunikációs hálózaton (LAN, Internet) keresztül elérhetővé és megoszthatóvá tegye a SAN-on, vagy hagyományos háttértáron tárolt adatokat. A NAS lényegében egy dedikált állomány szerver, ami természetesen a jogosultságok kezelését is megoldja.

Hasonlítsuk össze a két megoldást!

SAN	*NAS*
Szerver-háttértár kapcsolat	Szerver-kliens kapcsolat
Sok platform	Tetszőleges platform
Nagy megbízhatóság, sávszélesség	Nem garantált, osztott sávszélesség
A kapcsolódó szerverek száma néhány száz	Tetszőleges számú kliens
Maximális kiterjedés 10-20 km	Tetszőleges távolság áthidalható
Közvetlen háttértárat igénylő alkalmazások: média szerver, adatbázis szerver	Állomány szerver alapú alkalmazások
Csatorna orientált átvitel	Csomag orientált átvitel

4.23 ábra SAN és NAS összehasonlítása

A két rendszer nemcsak egymás mellett, egymást kiegészítve működhet, hanem egészen össze is mosódhat. Ilyenkor a NAS szerver háttértára természetesen a SAN része (az ábrán is így jelöltük), azonban a kliensek kéréseit, ha lehetséges, nem a LAN-on keresztül, hanem a kliensek FC kapcsolatát kihasználva (az ábrán nem jelöltük) a SAN-on keresztül elégítjük ki. A LAN-on ebben az esetben csak a kérések továbbítódnak, a jogosultságok ellenőrzése folyik, és az eléréshez szükséges információt kapják a kliensek. A bonyolult, kifinomult megoldásokhoz természetesen speciális, nagyon drága vezérlő szoftverek szükségesek.



4.24 ábra A Network Attached Storage (NAS) architektúra

Érdekesség, hogy a SAN és NAS (és általában a hárombetűs betűszavak) bűvöletében a hagyományos (PC-kben is alkalmazott) architektúra is új nevet kapott: DAS (Direct Attached Storage).

Σ

Az előző fejezet alapján képet alkothattunk arról, hogy milyen feladatai vannak egy eszköz kezelése kapcsán az operációs rendszernek. Kitértünk az egyes háttértár típusok fontosabb tulajdonságaira, részletesen elemeztük a lemezkezelő működését, funkcióit. A fejezet végén felhívtuk a figyelmet az adattömörítés, illetve adatbiztonság által támasztott ellentmondó követelményekre.

?

4.5 Ellenőrző kérdések

1. Ismertesse a mágneslemezek felépítését! Milyen paraméterekkel jellemezhetők? Milyen részekből áll egy blokk megtalálásának ideje?
2. Ismertesse a tanult adattömörítési elveket! Mikor melyik módszer használata kedvező? Milyen előnyei és hátrányai vannak a tömörítésnek?
3. Mi a lemezütemezők célja? Sorolja fel és jellemezze a tanult algoritmusokat!

4. Milyen elven működnek az optikai tárolók? Hasonlítsa össze jellemző paramétereit a mágneslemezekével!
5. Mi az interleave (közbeékelődő) technika? Hogyan kell elhelyezni a merevlemezeken a blokkokat, ha a cél a leggyorsabb átviteli sebesség biztosítása?
6. Ismertesse a lemezmeghajtó felépítését és működését!
7. Hasonlítsa össze a lemezegység és a memória közötti szinkron és aszinkron adatátvitelt!
8. Hogyan függ össze a blokkméret és a tárolási veszteség nagysága mágneslemezeknél?

5. Erőforráskezelés

A fejezet az erőforrás kezelés általános kérdéseivel foglalkozik. Az operációs rendszer egyik alapfeladatát, az erőforrások elosztását többféle stratégia szerint végezheti el. A különböző módszerek előnyeik mellett veszélyeket is rejtenek magukban (holtpont, kiéheztetés), melyek megelőzésére vagy kezelésére fel kell készülni. Részletesen ismertetjük a holtpont megelőzésére szolgáló eljárásokat. A többfeladatos rendszerekben gyakran előfordul, hogy több folyamat egymással kommunikál, gyakran közösen használt memória területeken keresztül. A közösen használt erőforrások kezelésének bemutatásával zárul a fejezet.

A számítógépekben többféle ún. erőforrás található. Ezeket az erőforrásokat különféleképpen csoportosíthatjuk.

Vannak a *hardver erőforrások*, ezek közé tartoznak például a processzor, a memória, a nyomtató és az egyéb perifériák. A másik nagy csoportot a *szoftver erőforrások* alkotják. Ezek közé a különböző közösen használható programok, adatállományok, adatbázisok tartoznak. Ma a számítógépek árának egyre kisebb részét jelenti a hardver és egyre nő a szoftvertermékek súlya.

Egy másik csoportosítás szerint beszélhetünk *„hagyományos"*, illetve *operációs rendszer által létrehozott erőforrások*ról. A hagyományos jelzővel azokat az erőforrásokat illetjük, amelyek az operációs rendszer nélkül is léteznek, például nyomtatók, szövegszerkesztők, hogy egy-egy hardver és szoftver példát is lássunk. Az operációs rendszer saját magának, illetve a futó folyamatoknak a vezérlésére többféle táblázatot, adatstruktúrát stb. is létrehoz. Ilyenek például a már látott fájl leírótáblák, lemez adatblokkok, a perifériaműveletek gyorsításához használt pufferek, illetve a későbbiekben részletesen ismertetendő folyamatvezérlő blokkok (PCB), laptáblák, szegmensleíró táblák stb. Az operációs rendszereknél szintén egyre inkább előtérbe kerül ezek fontossága és minél hatékonyabb használata, hiszen ezáltal lehet gyorsítani a rendszer működését.

Vannak olyan erőforrások, például a processzor és a memória, melyek használata a folyamatok között *(időben) megosztható* (sharable) és amelyek az őket használó folyamatoktól – természetesen az elvétel szabályait betartva – bármikor *elvehetők*, használatuk *megszakítható* (preemptive).

Vannak azonban olyan erőforrások, melyek *nem megoszthatók* (non-sharable) a folyamatok között, és ha már egyszer használatba vettük őket, a megkezdett művelet befejezéséig *nem megszakíthatók, nem elvehetők* (non-preemptive). Ilyen erőforrások lehetnek például a nyomtatók, mágnesszalagos egységek.

Az elkövetkezőkben a processzoridővel és a memóriával ellentétben olyan erőforrásokról lesz szó, melyek

- Nem megoszthatóak (non-sharable) más folyamatokkal;
- Használatuk nem megszakítható (non-peemptive), a folyamatok sorban egymás után használhatják őket.
- Korlátozott számúak (pl. egy rendszer 3 nyomtatójából nem lehet 4-et lefoglalni).
- Egész számúak (nincs 1,5 nyomtató).
- Egyenrangú elemekből álló osztályokra bonthatók, azaz a folyamatok számára érdektelen, hogy a csoport mely tagja áll rendelkezésükre.

5.1 Az erőforrás kezelő

Az erőforrás kezelő (resource manager) a rendszermag azon része, amely az erőforrások elosztásáért és lefoglalásáért felelős. Ha egy folyamat erőforrást igényel, az erőforrás kezelő dönti el, hogy a kérés kielégíthető-e. Pozitív döntés esetén az erőforrás „tulajdonjoga" bejegyződik a folyamatleíró blokkba, és az erőforrás is hozzárendelődik az őt kérő folyamathoz. Használat után a folyamat egy újabb rendszerhívást ad, melynek hatására az erőforrás kezelő felszabadítja az erőforrást. Egyes esetekben a folyamat (például valami egészen más jellegű hiba miatt) megszűnik, mielőtt felszabadította volna erőforrásait. Ilyenkor az erőforrás kezelő - jobb híján - feltételezi, hogy a folyamat rendben hagyta az erőforrásokat, és maga végzi el a felszabadítást.

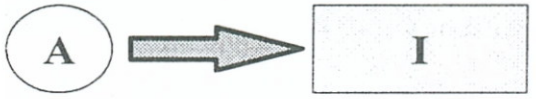
Előfordulhat azonban, hogy az erőforrás igényt nem lehet kielégíteni. Ha a folyamatnak nincs joga a választott erőforrás használatához vagy az éppen nem működőképes, az elutasítás tartós vagy végleges. A folyamat ekkor egy hibaüzenetet kap a rendszermagtól. A másik esetben, ha az igény jogos, de a kért erőforrás éppen foglalt, a folyamat az igény kielégítéséig az erőforrás-várólistára kerül.

Az erőforrás kérés nem feltétlenül jelenti azt, hogy a folyamat egyben a processzoridőt is elveszti, de az ütemezők gyakran használják fel ezt a kedvező alkalmat a folyamatok közötti váltásra.

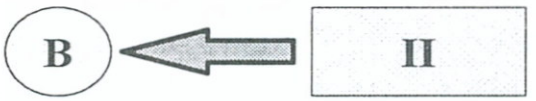
Összefoglalásul megállapíthatjuk, hogy az erőforrás kezelő gondoskodik a számítógép (rendszer) erőforrásainak – a futó folyamatok igényei alapján történő – hatékony, gazdaságos elosztásáról, illetve az erőforrások használatáért vívott versenyhelyzetek kezeléséről.

5.2 Erőforrás foglalási gráf

Az **erőforrás foglalási gráf** (resource allocation graph) szemléletes segédeszköz az erőforrás igények és foglaltságok állapotának ábrázolására. A gráf a folyamatok (körök) és az erőforrások (téglalapok) viszonyát mutatja. Ha egy folyamat erőforrást igényel, ezt az állapotot a folyamatot ábrázoló körtől az igényelt erőforrást ábrázoló téglalapra mutató nyíl jelzi. Ha egy erőforrás egy folyamat birtokában van, a nyíl iránya fordított, azaz az erőforrástól mutat a folyamat felé.



Az „A" folyamat igényli az „I" erőforrást



A „B" folyamat birtokolja a „II" erőforrást

5.1 ábra Erőforrás foglalási gráf

5.3 Holtpont

Az erőforrás kezelésnek a következő módszere végtelenül egyszerűnek tűnik: ha van szabad erőforrás, kielégítjük a kérést, ha nincs, a folyamat

várakozni kényszerül. Ez a probléma leginkább **liberális** megközelítése, azonban korlátai elég hamar megmutatkoznak.

Nézzünk egy példát! Van két folyamatunk, **A** és **B**, melyek mindegyike két mágnesszalagos egységet igényel. Például az egyikről töltik be a végrehajtandó algoritmust, a másikról az adatokat. A két szalag használata nem egyszerre kezdődik, de van olyan időszak, amikor egyszerre mindkettőre szükség van. Először mindkettő az egyiket igényli (előzékenyen gondolva a többi folyamatra), és csak egy bizonyos idő után a másikat. A rendszerben pontosan két szalagos egység van, **I** és **II**. A jelenet például a következőképpen zajlik le:

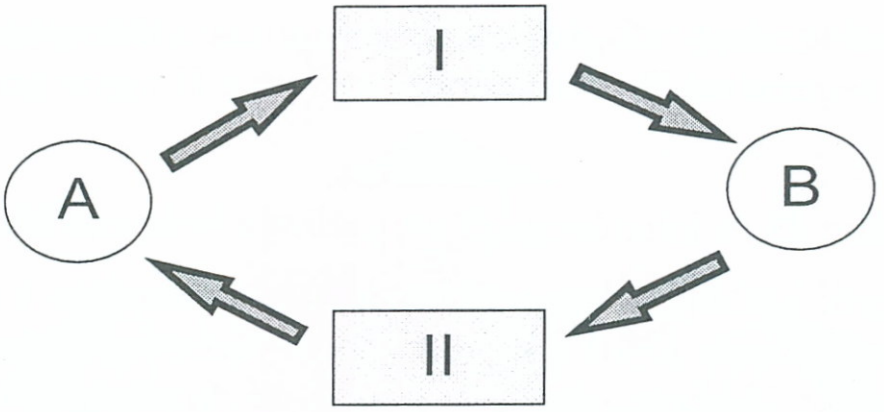
1. Tegyük fel, hogy a rendszer mindkét szalagos egysége szabad.
2. Az **A** folyamat igényel egy szalagos egységet.
3. Az erőforrás kezelő lefoglalja az **I** szalagot az **A** folyamat számára.
4. A **B** folyamat igényel egy szalagos egységet.
5. Az erőforrás kezelő lefoglalja a **II** szalagot a **B** folyamat számára.
6. Az **A** folyamatnak szüksége van a másik szalagos egységre is, kéri azt.
7. Az erőforrás kezelő - szabad erőforrás híján - várakoztatja **A**-t.
8. A **B** folyamatnak szüksége van a másik szalagos egységre is, kéri azt.
9. Az erőforrás kezelő - szabad erőforrás híján - várakoztatja **B**-t.

§

Holtpont - Deadlock

Több folyamat egy olyan erőforrás felszabadulására vár, amit csak egy ugyancsak várakozó folyamat tudna előidézni.

Mind **A** mind **B** várakozik, és mivel éppen a másikra várakozik, a helyzet reménytelen. Ez az állapot kapta a **HOLTPONT** (deadlock) nevet. Holtpont esetén a folyamatok körkörösen egymásra várakoznak, az erőforrás foglalási gráfban a nyilak mentén körbejárható zárt görbe, hurok jelenik meg.



5.2 ábra Holtponti erőforrás foglalási gráf

5.4 Kiéheztetés

A túlzottan liberális stratégia tehát egy nehezen kezelhető problémához vezetett, melyet végeredményben az okozott, hogy egyszerre több folyamat versenyezhetett ugyanazért az erőforrásért. Ha megszüntetjük a párhuzamosságot, a probléma megelőzhető.

A leginkább **konzervatív** megoldás az, ha megtiltjuk, hogy egyszerre több folyamat is rendelkezzen erőforrással. Ez esetben az erőforrással rendelkező folyamat sohasem várakozik, minden igénye kielégíthető, a többiek pedig türelmesen várják, hogy befejeződjön, mert akkor ők juthatnak az összes erőforráshoz. A fenti példánk ez esetben a következőképpen működik:

1. Tegyük fel, hogy a rendszer mindkét szalagos egysége szabad.
2. Az **A** folyamat igényel egy szalagos egységet.
3. Az erőforrás kezelő lefoglalja az **I** szalagot az **A** folyamat számára.
4. A **B** folyamat igényel egy szalagos egységet.
5. Az erőforrás kezelő várakoztatja **B**-t, hiszen **A**-nak már adott egy erőforrást.
6. Az **A** folyamatnak szüksége van a másik szalagos egységre is, kéri azt.
7. Az erőforrás kezelő lefoglalja a **II** szalagot is az **A** folyamat számára.
8. Munkája végeztével **A** felszabadítja az **I** és a **II** egységet.
9. Az erőforrás kezelő lefoglalja az **I** szalagot a **B** folyamat számára.
10. A **B** folyamatnak szüksége van a másik szalagos egységre is, kéri azt.

11. Az erőforrás kezelő lefoglalja a **II** szalagot is a **B** folyamat számára.
12. Munkája végeztével **B** felszabadítja az **I** és a **II** egységet.

A módszer működik, a lehetősége is megszűnt a holtpont kialakulásának.

A módszer egyik hátránya, hogy többnyire a rendszerben a rendelkezésre álló erőforrások száma jóval nagyobb, mint amit egy folyamat igényel, így az erőforrások jó része kihasználatlanul áll. Ez még a kisebbik baj, a biztonságért a rendszer lelassulásával fizetünk.

A másik hátrány sokkal súlyosabb. Amíg az egyik folyamat használhatja az összes erőforrást, a várakozó folyamatok elszaporodhatnak a várakozási sorban, a várakozási sor hosszúra nyúlhat. Az erőforrások felszabadulása után az erőforrás kezelő valamilyen algoritmus alapján dönt arról, hogy melyik folyamat következik. Ha ez a stratégia nem kellően demokratikus, vannak kiemelt és háttérbe szorított folyamatok (prioritásos módszer), előfordulhat, hogy egy folyamat hiába áll sorba, mindig akad egy, amelyik megelőzi. Ilyenkor még csak megbecsülni sem lehet, hogy a hátrányos helyzetű folyamat mikor futhat, mivel ez függ a beérkező folyamatok jellemzőitől. Ez az állapot a **KIÉHEZTETÉS** (starvation). A gyakorlatban kiéheztetés-veszélyes stratégiát soha nem szabad használni!

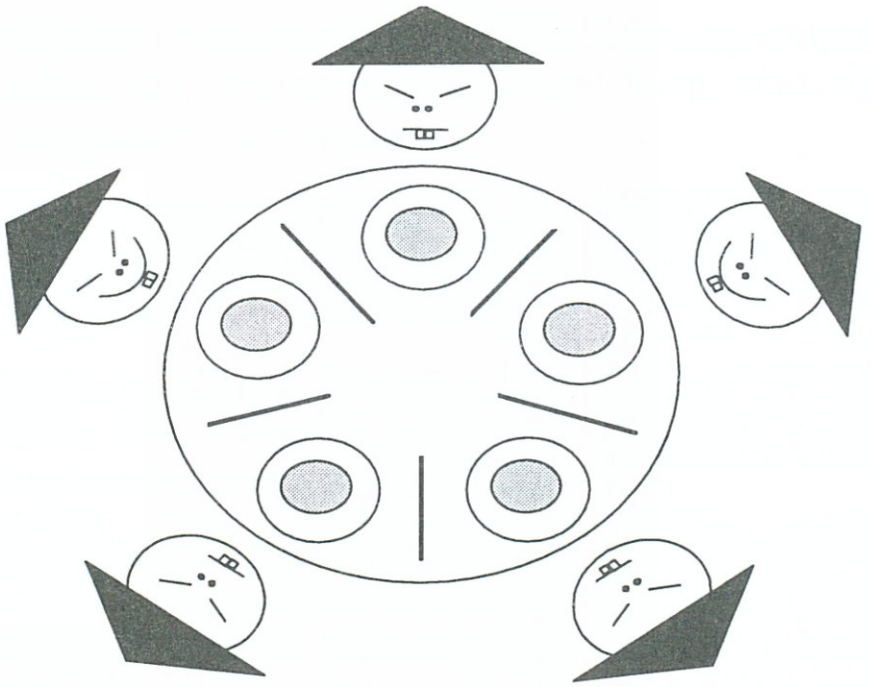
§

Kiéheztetés - Starvation
Egy folyamat – az erőforrás kezelő stratégiája miatt – beláthatatlan ideig nem jut erőforráshoz.

5.5 Példa - A vacsorázó bölcsek

Öt kínai bölcs ül egy kerek asztal körül. Mindegyik előtt ott egy tányér rizs és a szomszédos tányérok között egy-egy pálcika. Az evéshez két pálcika kell, ezért egy bölcsnek mind a jobb-, mind a baloldali pálcikát meg kell szereznie, hogy elkezdhessen enni. Ha egy bölcs eszik, akkor

egyik szomszédja sem tud, hiszen legalább az egyik pálcikája hiányzik hozzá.



5.3 ábra Kínai bölcsek

Ha mindegyik bölcs egyszerre jut arra a döntésre, hogy először a bal-, majd a jobboldali pálcikát veszi kézbe, mindegyiknek csak egy pálcika jut, senki sem tud enni, HOLTPONT-ra jutnak.

Ebben az esetben feltehetően születik megoldás, hiszen bölcsekről van szó. Ez azonban az egymásról mit sem tudó folyamatok esetén nem várható. Egy felsőbb erőnek, az operációs rendszernek kell rendet teremtenie, például úgy, hogy egyiktől elveszi a pálcikát és a szomszédjának adja.

5.6 Holtpont kezelő stratégiák

A szélsőséges liberalizmus és a szélsőséges konzervativizmus tehát egyaránt veszélyekkel jár, valamilyen közbülső megoldást kell találni. Alapvetően kétféle stratégia választható: vagy **meg kell előzni a holtpont kialakulását**, vagy folyamatosan figyelni kell, hogy kialakult-e holtpont, és ha igen, meg kell tenni a megfelelő lépéseket a **holtpont felszámolására**. Látni fogjuk, hogy a holtpont megelőzése kevesebb veszteséggel jár, mint a már kialakult holtpont megszüntetése, de sajnos,

nem mindig tehető meg, ezért kell a holtpont felszámolásával is foglalkoznunk majd.

Először foglaljuk össze, mi is vezethetett az előző példákban a holtpont kialakulásához:

§

1. Vannak olyan erőforrások, amelyek nem megoszthatók: egyszerre csak egy folyamat használhatja őket, azaz **kölcsönös kizárás van** a rendszerben.
2. Az erőforrásokra várakozó folyamatok **várakozás közben lefoglalva tartanak erőforrásokat.**
3. Vannak olyan erőforrások, amelyek nem preemptívek, azaz **erőszakkal nem elvehető („nem rabolható") erőforrások.**
4. Az erőforrásokra várakozó folyamatok körkörösen egymásra várnak, azaz az első folyamat egy olyan erőforrásra vár, amit a második birtokol, a második olyanra, amit a harmadik birtokol, stb., míg végül az utolsó folyamat olyan erőforrásra vár, amit az első birtokol, vagyis a rendszerben **ciklikus várakozás van**.

Ahhoz, hogy a rendszerben holtpont kialakuljon, **e négy feltétel egyidejű teljesülése szükséges!**

5.4 ábra Holtpont kialakulásának feltételei

Nézzük meg egy olyan példán, amellyel biztosan már mindenki találkozott, hogy ezek a feltételek tényleg holtpontot eredményeznek! Egy útkereszteződésbe mind a négy irányból behajt egy-egy autó és egymástól egyik sem tud továbbmenni. Itt az erőforrásoknak az

útkereszteződés „negyedrészei", míg a folyamatoknak az autók felelnek meg.

Az első feltétel teljesül, hiszen az útkereszteződés egy darabján egyszerre csak egy autó lehet. (Ha lenne legalább egy hely, ahol két autó elfér egymás tetején, megszűnne a holtpont.)

A második feltétel is teljesül, hiszen míg az autók várakoznak, lefoglalva tartják az úttest egy-egy részét. (Ha legalább egy autó az útkereszteződés előtt állt volna meg, mindenki elmehetne.)

A harmadik feltétel is teljesül, hiszen - legalábbis civilizált országban - nem száll ki három autó vezetője és nem tolja vissza a negyediket. Ha viszont mégis megtennék, megszűnne a holtpont.

Végül a negyedik feltétel is szükséges, hiszen ha csak három autó érkezett volna a kereszteződéshez, szép sorban el tudnának menni egymás után, azaz a holtpont kialakulásához szükség van arra, hogy a várakozási lánc bezáródjon.

5.6.1 Holtpont megelőző stratégiák

Most, miután láttuk, hogy milyen feltételek teljesülésére van szükség a holtpont kialakulásához, vizsgáljuk meg, hogy hogyan előzhető meg kialakulása. Könnyű belátni, hogy ha a **négy feltétel legalább egyikének a kialakulását megakadályozzuk, nem alakul ki holtpont!**

A feltételeket alaposabban megvizsgálva kiderül, hogy az első és a harmadik feltétellel nem tudunk mit kezdeni, hiszen a rendszerünkben vagy vannak kölcsönös kizárást igénylő, illetve erőszakkal el nem vehető erőforrások, vagy nincsenek. Tehát csak a másik kettőre, a várakozás közbeni foglalásra és a körkörös várakozásra érdemes koncentrálnunk.

5.6.1.1 Egyetlen foglalási lehetőség (One-shot allocation)

Az egyetlen foglalási lehetőség stratégia a várakozás közbeni erőforrás lekötést tiltja meg azáltal, hogy a folyamatok csak egy lépésben (célszerűen induláskor) foglalhatják le az összes szükséges erőforrást. (Természetesen ez csak akkor lehetséges, ha azok rendelkezésre is állnak. Ha bármelyik erőforrás is hiányzik, a folyamat várakozó listára kerül.) Az eljárás filozófiájából következik, hogy az a folyamat, amelyik már rendelkezik erőforrással, többletigényt nem nyújthat be, a további kéréseket az operációs rendszer elutasítja.

§

Egyetlen foglalási lehetőség

Csak az a folyamat foglalhat erőforrást, amelyik még egyetlen eggyel sem rendelkezik

Holtpont nem alakulhat ki, mivel a futó folyamatok nem kényszerülnek várakozni, mindenük megvan, a várakozó folyamatok pedig nem rendelkeznek erőforrásokkal. Az eljárás nem sokkal rugalmasabb, mint a konzervatív eljárás volt, bár itt nemcsak egy folyamat rendelkezhet erőforrással.

A módszer további előnye az, hogy egy folyamat, ha már megszerezte az összes szükséges erőforrást, soha többé nem kell, hogy erőforrásra várakozzon, azaz gyorsan lefuthat.

A módszer nagy hátránya, hogy az erőforrás kihasználás szempontjából nagyon pazarló, hiszen a folyamatok olyankor is kénytelenek foglalva tartani erőforrásokat, ha pillanatnyilag nincs rájuk szükség, de a későbbiekben még kelleni fognak. További hátrány, hogy nem mindig mérhető fel előre egy folyamat erőforrás igénye. A legkedvezőtlenebb tulajdonság azonban a kiéheztetés veszélye. Ha egy folyamat sok és népszerű erőforrást igényel, előfordulhat, hogy soha nem jön el az a pillanat, amikor mindegyik egyidejűleg áll rendelkezésre, így a folyamat bizonytalan ideig várakozni kényszerül, éhezik.

Megjegyzés: a módszernek létezik egy „finomított" változata is. Eszerint egy folyamat futása során többször is igényelhet ugyan erőforrásokat, de csak azzal a feltétellel, hogy *előzőleg az összes birtokolt erőforrását elengedte*. Ennek a változatnak előnye a nagyobb rugalmasság, és ezáltal az erőforrások valamivel jobb kihasználtsága. Többé viszont nem lesz igaz, hogy egy futó folyamatnak nem kell erőforrásra várnia – tehát lassabban futnak a folyamatok –, valamint többször adódhatnak kiéheztetés-veszélyes szituációk, hiszen, ha egy folyamat egy új erőforrás igénylése miatt elengedte az összes addig birtokolt erőforrását, akkor többnyire csak akkor folytathatja működését, ha az új mellett az összes régit is visszaszerezte, de ki tudja, hogy ez mennyi idő alatt sikerül neki.

5.6.1.2 Rangsor szerinti foglalás (Hierarchical allocation)

A rangsor szerinti foglalás a ciklikus várakozás lehetőségének kiküszöbölésével alkalmas a holtponthelyzetek megelőzésére. Lényege, hogy az erőforrások osztályaihoz egy-egy növekvő sorszámot rendelünk úgy, hogy a leggyakrabban használt erőforrások kapják a legkisebb sorszámokat. Ha egy folyamatnak egy osztályon belül több erőforrásra van szüksége, azokat csak egyszerre igényelheti. A megelőzési stratégia azon az elven alapul, hogy a folyamatok csak rangsor szerint növekvő sorrendben igényelhetnek erőforrásokat.

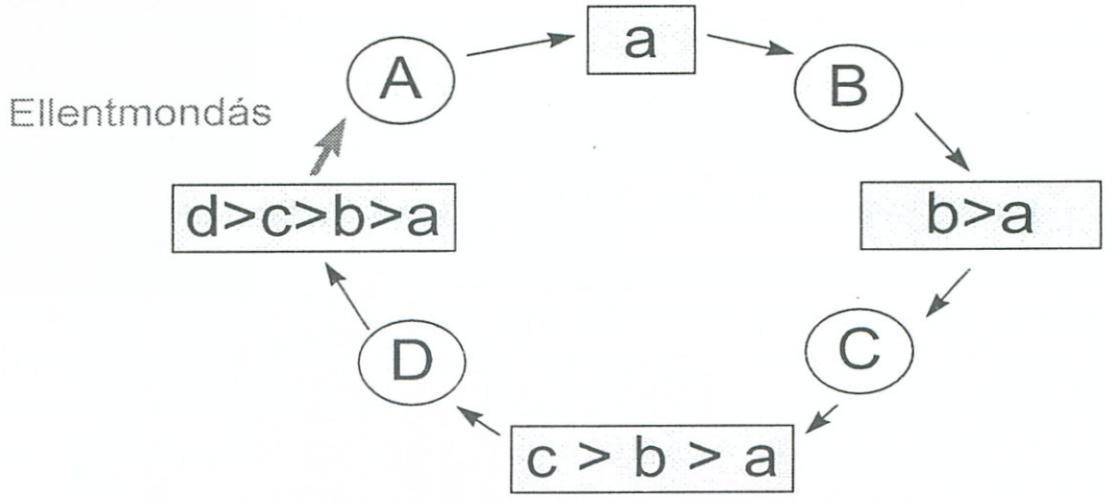
Rangsor szerinti foglalás
Egy folyamat csak olyan osztályból igényelhet erőforrást, melynek sorszáma magasabb, mint a már birtokolt erőforrások sorszáma

§

Az algoritmus működőképessége nem látható be azonnal, de viszonylag könnyen bebizonyítható, hogy ha a fenti feltételek ellenére holtpont alakulna ki, az csak úgy lenne lehetséges, ha valamelyik folyamat megsértette volna a rangsor szerinti foglalás elvét (indirekt bizonyítás).

Nézzük a következő erőforrás foglalási gráfot!

Tegyük fel, hogy a holtpont már kialakult, az **A** folyamatnak szüksége lenne a **B** folyamat által birtokolt **a** erőforrásra. A **B** folyamat viszont egy olyan **b** erőforrásra vár, amelynek sorszáma a rangsor elvnek megfelelően nagyobb kell legyen a már birtokolt **a**-énál. A hurok mentén folytatva a gondolatmenetet egészen az **A** folyamat által foglalt erőforrásig, megállapítható, hogy a lefoglalt erőforrások sorszáma **a**-hoz képest folyamatosan növekszik, tehát az **A** folyamat által már birtokolt erőforrás magasabb sorszámú, mint az igényelt. Az **A** folyamat tehát nem tartotta be a játékszabályokat. Mivel a gondolatmenet során nem hibáztunk, és végül ellentmondáshoz jutottunk, következik, hogy a kiindulási feltétel hamis, azaz holtpont NEM alakulhatott ki.



5.5 ábra Rangsor szerinti foglalás

A módszer hatékonyságát jelentősen befolyásolja a sorszámok kiadásának módja. Az lenne jó, ha a számokat olyan sorrendben rendelnénk az erőforrásokhoz, amilyen sorrendben azokat a folyamatok általában igénylik, de ez speciális, ún. feladat-orientált rendszereket kivéve általában előre nem mondható meg. (Különösen igaz ez az interaktív rendszerekre.) Ha egy folyamatnak a meglévőknél alacsonyabb sorszámú erőforrásra van szüksége, fel kell szabadítania erőforrásokat egészen addig a szintig, míg az igénylési feltételek nem teljesülnek.

5.6.1.3 A bankár algoritmus (Banker's algorithm)

A bankár algoritmus úgy kerüli el a holtpont kialakulásának lehetőségét, hogy a rendszert mindig ún. **biztonságos állapotban** tartja. Egy állapotot akkor tekintünk biztonságosnak, ha **létezik a folyamatoknak legalább egy olyan sorrendje, amely szerint haladva az összes folyamat maximális erőforrás igénye kielégíthető**.

Az erőforrás kezelő tevékenysége hasonlít a bankáréhoz – innen az elnevezés –, aki a lehető legtöbb kölcsönt szeretné nyújtani (meglévő pénzét egy olyan kuncsaftnak adva, akitől – ha kamatostul visszakapja – annyi pénzt zsebel be, hogy abból fedezni tudja egy másik kuncsaft kölcsönigényét, stb.) a csődbe jutás veszélye nélkül.

Az algoritmus *csak akkor működik, ha a folyamatok indulásukkor tudják, hogy a különböző erőforrásokból egyszerre maximálisan hányat fognak igényelni*, és ezt az operációs rendszernek – egy rendszerhívás által – be is jelentik. (Természetesen a megvalósíthatóság érdekében az igények

nem haladhatják meg a rendszerben ténylegesen meglévő erőforrások számát.) Ez a módszer legnagyobb hátránya: vannak ugyanis olyan folyamatok, amelyek erőforrás igénye előre nem tudható.

Az algoritmus úgy működik, hogy egy beérkezett erőforrásigény teljesítése előtt az erőforrás kezelő kiszámolja, hogy ha az igényt *teljesítené*, akkor a rendszer biztonságos állapotban maradna-e. Ha igen, teljesíti az igényt; ha nem, akkor a folyamat várakozó listára kerül. Ha egy folyamat visszaad erőforrásokat, akkor az erőforrás kezelő végignézi a várakozó listán levő folyamatokat, hogy most már kielégíthető-e valamelyik igénye.

Bankár algoritmus
Sohase elégítsünk ki egy igényt, ha az nem biztonságos állapotot eredményez

§

Az algoritmus biztosítja a holtpontmentességet, ugyanis az erőforrás kezelő mindaddig nem elégíti ki a várakozó folyamatok igényét, amíg nem biztosítható, hogy az újabb erőforrások lefoglalása által előálló új állapot biztonságos, azaz a futó folyamatok erőforrás igényét valamilyen sorrendben ki tudjuk elégíteni, és ezáltal azok le tudnak futni.

De mi is a nem biztonságos és mi a biztonságos állapot? Nézzünk egy egyszerű példát! Tegyük fel, hogy egy rendszerben, amelyben összesen 12 erőforrás található, két folyamat fut (F1 és F2). Az erőforrás kezelő nyilvántartása alapján az F1 folyamat legfeljebb 6, az F2 folyamat legfeljebb 11 erőforrást igényelhet. Mindkét folyamat 4-4 erőforrást már le is foglalt, tehát a szabadon maradt erőforrások száma szintén 4. Táblázatosan összefoglalva:

Aktuális állapot:
Összesen 12 db erőforrás
F1 és F2 folyamatok

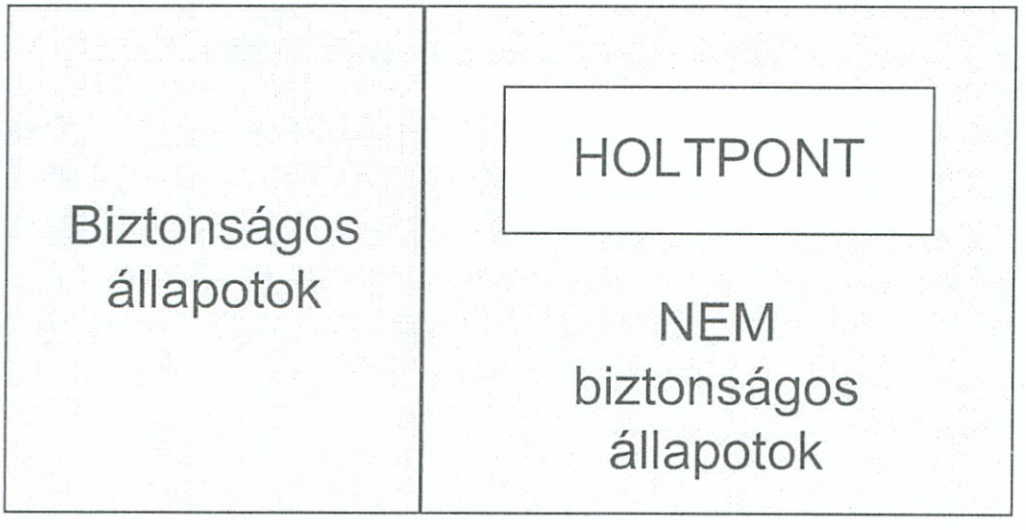
	Foglal	Max. igény	Várható igény
F1	4	6	2
F2	4	11	7
Szabad	4	= 12-4-4	

Mi történik, ha például a F2 folyamat előáll azzal az igénnyel, hogy azonnal kéri mind a 7 fennmaradó, előre bejelentett erőforrását? A szabad erőforrásokból a kérés nem elégíthető ki, tehát F2 várakozó listára kerül. Az F1 folyamat azonban maximális igényének felhasználása esetén is zavartalanul lefuthat. Miután F1 lefutott, és *felszabadította mind a 6 erőforrását*, F2 számára is jut elegendő, így folytathatja a munkát. A folyamatok tehát az {F1, F2} sorrendben **biztonságosan** befejeződhetnek.

§

Biztonságos állapot
Egy rendszer állapota akkor biztonságos, ha létezik legalább egy olyan sorrend, amely szerint a folyamatok erőforrás igényei kielégíthetőek

Meg kell jegyezni, hogy ha egy rendszer állapota nem biztonságos, nem jelenti azt, hogy a holtpont kialakul, hanem mindössze azt, hogy LEHET, HOGY kialakul! Azaz az algoritmusunk úgy garantálja a holtpont-mentességet, hogy már egy, *a holtpontinál tágabb szituáció – a nem biztonságos állapot – kialakulását is megakadályozza.* Tehát az algoritmusunk ilyen értelemben *túlzott biztonságra tör.*



5.6 ábra: Biztonságos és Nem biztonságos állapotok

A bankár algoritmus feladata, hogy megakadályozza a nem biztonságos állapot kialakulását, és így egy esetleges holtpont létrejöttét azáltal, hogy minden foglalás előtt elemzést végez, hogy a kérés teljesítése esetén is biztonságos marad-e a rendszer állapota.

Folytassuk az előző példát! Érkezzen az előzőleg leírt biztonságos rendszerünkbe egy F3 folyamat, amely bejelenti, hogy legfeljebb 8 erőforrásra lesz szüksége, de abból máris 2-t kér. Van elegendő szabad erőforrásunk, azonban az algoritmus előírja, hogy csak akkor szabad kielégíteni az igényeket, ha a változás eredményeként születendő új állapot biztonságos marad.

Aktuális állapot:

Összesen 12 db erőforrás

F1 és F2 folyamatok

	Foglal	Max. igény	Várható igény
F1	4	6	2
F2	4	11	7
Készlet	4	= 12-4-4	

Várakozó „F3" folyamat - BEENGEDHETŐ ?

F3	2	8	6

Tehát nézzük meg, hogy ha kielégítenénk F3 igényét, biztonságos állapotban maradunk-e. Ha igen, F3 megkapja a kért két erőforrást, ha nem, nem adjuk oda neki, F3 várólistára kerül. Ha feltesszük, hogy kielégítettük F3 igényét, akkor a következő táblázatot kapjuk:

Tegyük föl, hogy beengedjük...

	Foglal	Max. igény	Várható igény
F1	4	6	2
F2	4	11	7
F3	2	8	6
Készlet	2		

Az „F1" folyamat lefuthat, mivel várható igénye nem nagyobb a szabad erőforrások számánál

A táblázatból látható, hogy F1 két erőforrást igényelhet még, és pont ennyi szabad erőforrás van is. Adjuk oda ezt a kettőt F1-nek! Ha az F1 lefutott, akkor felszabadítja azt a két erőforrást, amit most kapott meg, valamint azt a négyet is, amit eddig birtokolt. Azaz miután az F1 lefut, a szabad erőforrások száma 6 lesz.

Az „F1" lefutott, foglalt erőforrásai felszabadultak...

	Foglal	Max. igény	Várható igény
F2	4	11	7
F3	2	8	6
Készlet	6		

Az „F3" folyamat lefuthat, mivel várható igénye nem nagyobb a szabad erőforrások számánál

A 6 szabad erőforrásból F3 igénye kielégíthető, az elkezdhet futni. Miután a F3 lefutott, a szabad erőforrások száma 8 lesz, amiből a F2 7-es igénye bőven kielégíthető, így az is elkezdhet futni, majd lefutása után az összes erőforrás szabad lesz. Az állapot **biztonságos** marad, hiszen a folyamatok erőforrás igénye {F1, F3, F2} sorrendben kielégíthető, tehát F3 megkaphatja a kért két erőforrást.

Az „F3" lefutott, foglalt erőforrásai felszabadultak...

	Foglal	Max. igény	Várható igény
F2	4	11	7
Készlet	8		

Az „F2" folyamat lefuthat, mivel várható igénye nem nagyobb a szabad erőforrások számánál

Az állapot a „F3" beengedése után is biztonságos

ENGEDJÜK BE !

Nézzük meg mi történik, ha az F3 folyamat nem 8, hanem 9 erőforrást jelent be maximális igényként! Látszólag apró a változás, mégis jelentős a különbség az előző esethez képest.

Az F1 folyamat lefutása után a szabad erőforrások száma 6, míg az F2 és F3 folyamatok maximális erőforrás igénye egyaránt 7!

Az „F1" lefutott, foglalt erőforrásai felszabadultak...

	Foglal	Max. igény	Várható igény
F2	4	11	7
F3	2	9	7
Készlet	6		

Az állapot „F3" beengedése után

NEM BIZTONSÁGOS

Nem feltétlenül alakul ki holtpont, hiszen elképzelhető, hogy a folyamatok nem kívánják kihasználni maximális lehetőségeiket vagy mielőtt az egyik folyamat kérné a maximumát, a másik visszaad néhány olyat, amit most használ, de ebbe az operációs rendszernek már nincs beleszólása, vagyis az állapot NEM BIZTONSÁGOS! A bankár algoritmusnak tehát kötelessége várakozó listára irányítania a F3 folyamatot - hiszen, ha az igényét kielégítenénk, az nem biztonságos állapotot eredményezne.

Nézzünk egy kicsit bonyolultabb példát! Most a rendszerünkben egy helyett három erőforrás van. Bankár hasonlattal élve, most a bankárunknak egyszerre három különböző pénznemben kell egymás után kihelyezni a pénzét a várakozó kuncsaftok között úgy, hogy végül visszakapjon mindent.

Azaz most a rendszer akkor lesz biztonságos állapotban, ha találunk legalább egy olyan sorrendet, amely szerint a folyamatok igényeit ki tudjuk elégíteni *egyszerre mindhárom erőforrásból*!

Nézzük tehát a példát!

Egy rendszerben háromféle erőforrás van, E1, E2, E3.

Az E1-ből 10 darab, az E2-ből 5 darab, míg az E3-ból 7 darab van.

A rendszerünkben 5 folyamat fut. Kérdés: biztonságos-e a következő állapot holtpontmentesség szempontjából?

	MAX. IGÉNY			FOGLAL		
	E1	E2	E3	E1	E2	E3
F1	7	5	3	0	1	0
F2	3	2	2	3	0	2
F3	9	0	2	3	0	2
F4	2	2	2	2	1	1
F5	4	3	3	0	0	2

Nézzük meg, hogy mit jelentenek a mátrixok! Például a MAX. IGÉNY mátrix F3 sorának és E1 oszlopának metszéspontjában lévő 9-es szám azt jelenti, hogy a F3-as folyamat az E1-es erőforrásból maximum 9-et igényelhet egyszerre, míg a FOGLAL mátrix ugyanezen helyén lévő 3-as szám azt, hogy ez a folyamat a 9 lehetséges E1-ből 3-at már lefoglalt és használ.Tehát ez az, amit tud az operációs rendszer.

Az algoritmus indulásához még két dolgot ki kell számolni: az egyik az, hogy pillanatnyilag az egyes erőforrás fajtákból hány szabad van (az egyszerűség kedvéért hívjuk ezt KÉSZLET-nek), míg a másik az, hogy az egyes erőforrásokból az egyes folyamatok még mennyit igényelhetnek (ezt az egyszerűség kedvéért a továbbiakban „várható igény" helyett egyszerűen csak IGÉNY-nek hívjuk majd).

Kezdjük először a KÉSZLET kiszámításával! Azt tudjuk, hogy a rendszerben hány erőforrás van *összesen* (10, 5, illetve 7). Azt is ki tudjuk számolni, hogy a folyamatok az egyes erőforrásokból *összesen* mennyit foglalnak. Nyilván ennek a két számhármasnak a *különbsége* lesz a szabad erőforrások száma, azaz a KÉSZLET. Nézzük tehát: az E1-ből a F1 0-t, a F2 3-at, a F3 is 3-at, a F4 2-t, míg a F5 0-t foglal, vagyis összesen 0+3+3+2+0=8-at foglalnak a folyamatok, tehát 10-8=2 szabad az E1-ből. Mit csináltunk? Összeadtuk a FOGLAL mátrix első oszlopának az elemeit, majd az összeget kivontuk az E1 erőforrások összes számából. Csináljuk meg ezt a másik két fajta erőforrásra is!

E2-ből szabad: 5-(1+0+0+1+0)=5-2=3,

míg E3-ból szabad: 7-(0+2+2+1+2)=7-7=0.

Tehát az induló készlet: {2,3,0}.

A másik feladatunk a folyamatok további várható IGÉNYének a felmérése. Tudjuk, hogy a folyamatoknak mi a MAXimális IGÉNYe, illetve azt, hogy pillanatnyilag hány erőforrást FOGLALnak. E két mátrix elemeinek a *különbsége* lesz az IGÉNY mátrix:

	MAX. IGÉNY			–	FOGLAL			=	IGÉNY		
	E1	E2	E3		E1	E2	E3		E1	E2	E2
F1	7	5	3		0	1	0	F1	7	4	3
F2	3	2	2		3	0	2	F2	0	2	0
F3	9	0	2		3	0	2	F3	6	0	0
F4	2	2	2		2	1	1	F4	0	1	1
F5	4	3	3		0	0	2	F5	4	3	1

Miután megvannak a kiinduló adataink, kezdhetjük a keresést. Ha megnézzük az IGÉNY mátrixot, látható, hogy a {2,3,0} KÉSZLET-ből csak az F2 igényét elégíthetjük ki. Ha F2 lefut, visszaadja a most kapott {0,2,0} erőforrást valamint *az eddig foglalt {3,0,2}-t is.* Tehát az új

KÉSZLET = {2,3,0}-{0,2,0}+{0,2,0}+{3,0,2}={2,3,0}+{3,0,2}=**{5,3,2}**

Ebből a KÉSZLETből a megmaradt F1, F3, F4 és F5 folyamatok közül az F4 vagy az F5 igénye kielégíthető. Válasszuk mondjuk F5-öt! Ha F5 lefut, az új KÉSZLET = {5,3,2}+{0,0,2}={5,3,4} lesz. Ebből F4 igénye teljesíthető. Lefutása után az új KÉSZLET={5,3,4}+{2,1,1}={7,4,5} lesz.

Ebből például F1 igénye kielégíthető. F1 lefutása után az új KÉSZLET a következő lesz: {7,4,5}+{0,1,0}={7,5,5}. Ez viszont a megmaradt F3-nak is elég, lefutása után a KÉSZLET: {7,5,5}+{3,0,2}={10,5,7}, vagyis az összes erőforrás szabad lesz. Tehát a folyamatok erőforrás igénye például F2, F5, F4, F1, F3 sorrendben kielégíthető, azaz a kiinduló állapotunk **biztonságos** volt. (Megjegyzés: a számítás során többször kerültünk olyan helyzetbe, hogy a pillanatnyi készletből több folyamat igényét is kielégíthettük volna. Tehát jelen példánkban több más „jó" sorrend is lett volna. Azonban ezeket nem kell megkeresni, hiszen, ha *legalább egy* sorrendet találunk, az állapot biztonságos!)

Σ

Foglaljuk össze a lépéseket még egyszer!

1. Az induló KÉSZLET meghatározása: az összes erőforrás számából kivonjuk a FOGLAL mátrix egyes oszlopainak összegét.

2. Az IGÉNY mátrix meghatározása: a MAX. IGÉNY mátrix elemeiből kivonjuk a FOGLAL mátrix elemeit.

3. Megnézzük, hogy van-e olyan folyamat, amely igénye a készletből kielégíthető: keressünk olyan sort az IGÉNY mátrixban, amelynek elemei nem nagyobbak a KÉSZLET elemeinél.

4. Ha nincs ilyen sor, az állapot NEM BIZTONSÁGOS.

5. Ha volt ilyen sor, akkor az új KÉSZLET-et megkapjuk, ha az eredeti KÉSZLET elemeihez hozzáadjuk a kiválasztott folyamat FOGLAL sorának elemeit.

6. Ha van még folyamatunk, visszamegyünk a 3. pontra.

7. Ha nincs több folyamatunk, az állapot BIZTONSÁGOS.

Σ

Az algoritmus értékeléseként elmondható, hogy

- mivel a nem biztonságos állapot nem jelenti egyben a holtpont kialakulását, az algoritmus a rendszert túlbiztosítja, de az így fellépő csökkenés az erőforrások kihasználtságában sokkal kisebb mértékű, mint a korábbi algoritmusoknál volt;
- nem minden esetben valósítható meg, mivel előre tudni kell, hogy maximum hány erőforrást igényelnek a folyamatok;
- bonyolult, időigényes számítást igényel.

5.6.2 Holtpont felszámolása

Mint láttuk, sajnos a holtpont megelőző stratégiák mindegyikének komoly elvi és/vagy gyakorlati problémái vannak, tehát ezek csak speciális körülmények között alkalmazhatók. Ezért beszélnünk kell arról is, hogy mit tegyünk akkor, ha a rendszerben holtpont már kialakult.

5.6.2.1 Holtpont detektálása

Először is képesnek kell lennünk észrevenni, detektálni, ha egy holtpont kialakult. Ehhez az operációs rendszernek folyamatosan nyilván kell tartania az erőforrások szétosztását és a ki nem elégített igényeket. Ezen adatokból kiindulva időnként egy ún. *holtpont detektáló algoritmust* kell lefuttatni. Ezen algoritmusok két leggyakrabban használt változata közül az egyik a bankár algoritmushoz nagyon hasonló (csak nem kell tudni hozzá a folyamatok maximális igényeit, ami a problémát jelentette), míg a másik a már említett erőforrás foglalási gráfot ellenőrzi, hogy nem tartalmaz-e hurkot.

A következő kérdés az lehet, hogy mi az az „időnként", amikor a holtpont detektáló algoritmust futtatni célszerű. Az egyik válasz az lehet, hogy minden olyan esetben futtassuk le ezt, amikor egy erőforrás igényt nem tudunk azonnal kielégíteni. Bár így vehetjük észre leghamarabb a holtpontot, de ilyen eset nagyon gyakran történik, ezért ez a megoldás - az algoritmusok nagy számításigénye miatt - nagyon lelassítaná rendszerünk működését. Tehát nincs más hátra, a biztonságból áldozni kell: futtassuk a holtpont detektáló algoritmusunkat viszonylag ritkán, mondjuk adott - jó nagy - időnként periodikusan. Ezzel meg nyilván az a baj, hogy ez alatt az idő alatt esetleg több holtpont is kialakulhatott és akár elég régóta blokkolhatja is a folyamataink működését.

5.6.2.2 Holtpont megszüntetése

Az első gondolatunk a holtpont megszüntetésére - vagy más néven a *rendszer holtpontból való felélesztésére* - az lehet, hogy az összes, a holtpontban lévő folyamatot szüntessük meg. Ez tagadhatatlanul elég egyszerű megoldás, de hatalmas veszteséggel jár, hiszen sok folyamat munkáját kell újra kezdeni, illetve nem is biztos, hogy meg lehet ismételni működésüket. Próbáljunk meg finomítani az ötleten: valamilyen szempont alapján válasszunk ki először *egyet* a holtpontban résztvevő folyamatok közül, szüntessük meg azt, majd ellenőrizzük, hogy a *maradék* még mindig holtpontban van-e. Ha nem, „olcsón megúsztuk", ha igen, válasszunk még egy áldozatot, stb. Könnyű belátni, hogy ilyenkor legrosszabb esetben is egy folyamat „élve marad", míg az előbb mindet „kivégeztük". Az ár persze az, hogy a vezérlés bonyolultabb lett.

Milyen szempontok alapján választhatunk áldozatot a holtpontban lévő folyamatok közül? Több szempontot is figyelembe vehetünk, de sajnos

sok szempont egymásnak ellentmond, így nem lehet optimális megoldást találni, több-kevesebb kompromisszumot kell kötnünk.

Áldozat kijelölési szempontok:

- Melyikkel hány erőforrást nyerek - célszerű olyan folyamatot választani, amely sok erőforrást használt, mert valószínű, hogy az így nyert erőforrás mennyiség elég lesz a többi futásához;
- Hány további erőforrást igényel még - célszerű olyan folyamatot kiválasztani, amely még sok erőforrást fog igényelni, hiszen valószínűleg ez hajlamos lesz arra, hogy újabb holtpontot okozzon;
- Mennyi már elhasznált CPU időt ill. I/O munkát vesztek a megszüntetéssel - célszerű olyan folyamatot választani, amellyel keveset, hiszen így kevesebb munkát kell majd újra elvégezni;
- Mennyi idő van még hátra a futásából - nem célszerű olyan folyamatot kiválasztani, amely már majdnem kész;
- Ismételhető / nem ismételhető folyamat-e - nem célszerű olyan folyamatot választani, amely munkája nem ismételhető meg;
- A folyamat prioritása - célszerű kis prioritású folyamatot választani;
- Megszüntetése hány további folyamatot érint - egy olyan folyamat megszüntetése, amely sok más folyamattal dolgozott együtt, az együttműködő folyamatok megszüntetését vagy legalábbis - ha a megszüntetett folyamatunk megismételhető volt - hosszas várakoztatását okozhatja.

A módszerünket tovább finomíthatjuk: lehet, hogy nincs is szükség a folyamat teljes megszüntetésére, elég ha néhány *erőforrást elveszünk tőle*. Ez persze veszteséget jelent, de sokszor sokkal kevesebbet, mint ha a folyamatot teljesen megszüntetnénk. Meg kell azonban jegyeznünk, hogy sok esetben egy nem elvehető erőforrás elrablása a folyamat működésében helyrehozhatatlan hibát okoz, míg máskor - különösen, ha a folyamatban sokféle hibakezelési rutin található - ez nem okoz (nagy) problémát. Az áldozat kijelölési szempontok ilyenkor is a fentiekhez hasonlók lehetnek.

Sokszor azonban még erre sincs szükség, elég ha a holtpontban lévő folyamatokat – vagy közülük néhányat – egy olyan korábbi állapotból folytatunk, amikor még nem volt holtpont. Ehhez az kell, hogy a

folyamatok állapotát periodikusan elmentsük, ezek az ún. *ellenőrző pontok* (check points). Így lesz a legkisebb a felélesztési veszteség, de ez a módszer hatalmas tárigényt igényel (ilyenkor használhatók például a nagykapacitású mágneslemezek) és sok plusz idővel jár, ezért általában csak a különlegesen nagy megbízhatóságot igénylő rendszerek esetében alkalmazzák.

Fontos megemlítenünk azt, hogy akármilyen módszer szerint is járunk el a holtpont megszüntetésénél, figyelembe kell vennünk azt is, hogy ugyanazt a folyamatot nem választhatjuk akárhányszor áldozatul, hiszen ez a folyamat *kiéheztetését* okozná.

Végezetül meg kell említeni a holtpontkezelés két legnagyobb veszélyét.

Az egyik az ún. *alulszabályozás*, vagyis az, ha olyan túlbiztosításra törekszünk, hogy ezáltal ugyan holtpont nem fog kialakulni, de a rendszer lehetőségeit távolról sem tudjuk kihasználni, például nem működtetünk párhuzamosan eszközöket, feleslegesen várakoztatunk folyamatokat stb. Ezt okozhatja például az egyetlen foglalási stratégia.

A másik probléma az ún. *túlszabályozás*, azaz az, ha olyan bonyolult, számítás- és/vagy memóriaigényes algoritmust választunk, hogy maga az algoritmus futtatása lassítja le katasztrofálisan a rendszert.

5.7 Közös erőforrások

Közös erőforrásoknak azokat az erőforrásokat nevezzük, amelyeket egynél több folyamat szeretne egy időben használni. Az ilyen erőforrások vezérlésekor különösen a nem megosztható, más néven *kölcsönös kizárást* (**mut**ual **ex**clusion – mutex) *igénylő* erőforrások esetén merülnek fel nehézségek. Hiszen nem megengedhető, hogy például egy nyomtatót úgy használjon két folyamat, hogy egy sort az egyik, egy sort a másik nyomtat. Nézzük meg mit lehet tenni az ilyen és hasonló problémák elkerülésére!

Általánosságban a kölcsönös kizárást igénylő erőforrások kezelését végző programrészeket **kritikus szekciónak** nevezzük. Azt kell majd biztosítani, hogy egy kritikus szekcióban mindenképpen csak egy folyamat tartózkodhasson.

A példa, amelyen a vizsgálódásainkat végezzük az ún. *termelő-fogyasztó probléma*. Ennek a legegyszerűbb változata a következő. Tegyük fel,

hogy van két folyamatunk, mely egy közös memóriaterületen, az ún. **postaládán** keresztül kommunikál egymással. Az egyik folyamat – az ún. **termelő folyamat** – adatokat ír a közös memóriaterületre, míg a másik folyamat – az ún. **fogyasztó folyamat** – ezeket az adatokat kiolvassa onnan. Plasztikusabban fogalmazva, a termelő leveleket tesz a postaládába, míg a fogyasztó kiveszi azokat onnan.

5.7 ábra Termelő és fogyasztó folyamatok

Nyilvánvaló, hogy a postaláda ebben a példában egy kölcsönös kizárást igénylő erőforrás, hiszen amíg például a termelő nem fejezte be a levél beírását, a fogyasztó nem olvashatja azt ki onnan, illetve fordítva, amíg a fogyasztó nem fejezte be az előző levél kivételét, nem teheti be a termelő a következőt.

Hogyan lehet ezt vezérelni? A megoldást a vasutasok már régen kitalálták. Mivel igen kellemetlen következményekkel tud járni, ha egy adott sínszakaszon egyszerre két vonat is tartózkodik, ezért a megfelelő helyre elhelyeznek egy szemafort, amely ha pirosat mutat, nem lehet behajtani az adott szakaszra. Az ekkor érkező vonatnak várnia kell, míg az előző el nem ment és szabadot nem mutat a szemafor. Ez az elv itt is segít.

Rendeljünk az adott postaládánkhoz egy **szemafort**. Ez a szemafor egy változó a memória egy olyan helyén, amelyet mind a termelő, mind a fogyasztó elér. (Tipikusan például a postaláda eleje vagy vége.) Mondjuk azt, hogy ha a szemafor értéke 0, akkor tilosat mutat, ha értéke 1, akkor szabadot. Mivel ez a szemafor kétféle dolgot tud jelezni - később majd lesznek komplikáltabb szemaforok is - ezért ezt **bináris szemafor**nak hívjuk. Ezek után nézzük meg például, hogy mit kell tennie a termelő folyamatnak, ha használni akarja a postaládát!

A termelő folyamat programja:

1. Ki kell olvasni a szemafor értékét.

2. Meg kell vizsgálni, hogy szabad volt-e (azaz az értéke nem 0-e).
3. Ha a szemafor szabad volt, akkor tilosra kell állítani, hogy más ne tudjon majd „behajtani", amíg dolgozik a termelő.
4. Ha nem - természetesen legalább egy kis várakozás után, hogy időközben legyen esély arra, hogy a szemafort valaki szabadra állítsa - vissza kell menni az 1. lépésre.
5. A termelő írhat a postaládába.
6. Ha befejezte, a szemafort szabadra kell állítani, hogy más is jöhessen.

És a fogyasztó folyamat programja?

1. Ki kell olvasni a szemafor értékét.
2. Meg kell vizsgálni, hogy szabad volt-e (azaz az értéke nem 0-e).
3. Ha a szemafor szabad volt, akkor tilosra kell állítani, hogy más ne tudjon majd „behajtani", amíg dolgozik a fogyasztó.
4. Ha nem - természetesen legalább egy kis várakozás után, hogy időközben legyen esély arra, hogy a szemafort valaki szabadra állítsa - vissza kell menni az 1. lépésre.
5. A fogyasztó olvashat a postaládából.
6. Ha befejezte, a szemafort szabadra kell állítani, hogy más is jöhessen.

5.8 ábra Szemafor

Látható, hogy – a postaláda használati módjától (írás illetve olvasás) eltekintve – a két program azonos.

Az ugyan túl szép lenne, ha a megoldás ilyen egyszerű lenne, de nem is sokkal bonyolultabb. A fentiek majdnem mindig jól működnek, kivéve azt az esetet, ha a két folyamat közel egyszerre akar a postaládához férni. Tegyük fel, hogy a termelő jött egy picivel előbb, kiolvasta a szemafor

értékét. Azonban míg vizsgálja, hogy szabad-e, jön a fogyasztó, és az is kiolvassa a szemafort. Ezután hiába állítja tilosra a szemafort a termelő, ez már késő, hiszen a fogyasztó is szabadnak találta a jelzést. Gondolkozzunk egy kicsit! Mi okozhatta a problémát? Az, hogy a szemafor kiolvasása, vizsgálata és lefoglalása nem egy művelet volt, hanem egy megszakítható műveletsor. Ha azt biztosítani tudnánk, hogy az 1.-4. lépések megszakíthatatlanok legyenek, azaz ún. **oszthatatlan művelet** legyen, meglenne a megoldás. Hiszen akkor a picivel később érkező fogyasztó folyamat már a termelő által beállított tilos jelzést találná. Ez azonban minden esetben hardver támogatást igényel.

Sok számítógép utasításkészletében találunk olyan utasítást, amelyek a fenti vagy ahhoz nagyon hasonló műveletsort valósítanak meg. Ezt az utasítást sokszor „**TEST AND SET**", azaz „vizsgáld meg és állítsd be" utasításnak hívják. Ez az utasítás tipikusan úgy működik, hogy egy paraméterül megadható memóriaterület – itt lesz a szemafor – értékét kiolvassa egy regiszterbe, majd a memóriaterületre beírja a tilos értéket. Ezután a regiszterben megvizsgálhatjuk a szemafor értékét. Ha az szabad volt, akkor a szemafort az előbb tilosra állítottuk. Míg ha eleve tilos volt, a tilos érték ismételt beírásával azt nem változtattuk meg.

Egyszerűbb esetekben – például ha nincs multiprogramozás és csak megszakítási eljárások dolgoznak együtt egymással vagy a főprogrammal – az 1. lépés elé elhelyezett DI – disable interrupt, megszakítás letiltás – és a 4. lépés után tett EI – enable interrupt, megszakítás engedélyezés – utasítások használata is megoldás lehet, míg igazi többprocesszoros rendszereknél a memóriához hozzáférést lehetővé tévő sín megfelelő ideig tartó ún. *lezárása* (lock) a tipikus megoldás.

Bonyolultabb esetekben a szemafor szabadra állítása is több utasításból állhat, ezért célszerű ezt is oszthatatlan műveletként megvalósítani.

Mivel a szemaforok lefoglalása és felszabadítása nagyon gyakori feladat többfolyamatos rendszerekben, ezért sok operációs rendszer ehhez támogatást nyújt, mégpedig az ún. **P és V primitívek** formájában.

A P primitív egy szemafor lefoglalását végzi - azaz az 1. – 4. pontokat - (tehát benne van az esetleg szükséges várakozási ciklus is!), míg a V primitív a szemafor szabaddá tételét – azaz a 6. pontot – intézi. (A primitív a számítástechnikában oszthatatlan, „atomi" műveletet jelöl, míg a P és V betűk a megfelelő műveletek holland kezdőbetűi, ugyanis a

problémát egy holland úriember – Dijkstra – ismerte fel először és ő adott rá megoldást.) Formájukat tekintve a P és a V primitívek egy-egy függvénynek felelnek meg, amelyeknek egy paraméterük van, az állítani kívánt szemafor neve.

Nézzük meg ezután, hogy mit kell tennie a termelő és a fogyasztó folyamatnak! (Tegyük fel, hogy a postaládát vezérlő szemafor neve „S".)

P(S);
ÍRÁS A MEMÓRIÁBA
V(S);

P(S);
OLV. A MEMÓRIÁBÓL
V(S);

5.9 ábra P és V primitívek használata

Mielőtt tovább mennénk, beszéljünk meg egy-két „különlegesebb" esetet!

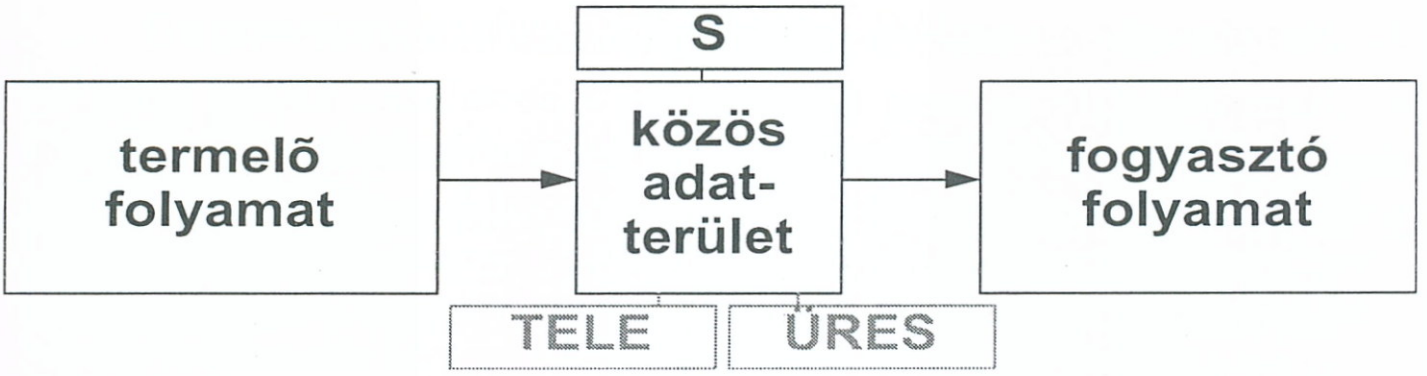
Egyrészt, könnyű belátni, hogy a megoldásunk akkor is működik, ha nem *egy* termelő és *egy* fogyasztó folyamatunk van, hanem akár az egyikből, akár a másikból, akár mindkettőből több is hozzá akar férni a postaládához. (Hasonlóképpen, mint ahogy egy útkereszteződésnél egy jelzőlámpa több autót is tud vezérelni, legfeljebb az átjutási idő, – azaz a számítástechnikai példában a postaládára várakozási idő – megnőhet.) Sőt az sem jelent problémát, ha vannak olyan folyamatok is, amelyek hol termelőként, hol fogyasztóként viselkednek.

Másrészt azt sehol sem használtuk ki, hogy az erőforrásunk egy *memóriaterület* volt, hasonlóan vezérelhető tetszőleges, kölcsönös kizárást igénylő erőforrás – például egy nyomtató – is, csak arra kell vigyázni, hogy a szemafor vagy egy olyan memóriaterületen legyen, amit *mindazon folyamatok használhatnak, akik magát az erőforrást is* vagy magában az erőforrásban – ha az erőforrás gyártója gondolt erre – például egy vezérlőregiszter formájában.

Harmadszor pedig gondoljunk bele a következőbe: ha a szabad érték = 1, a tilos érték pedig = 0, akkor a szemafor tilosra állítása azt jelenti, hogy 1-

es helyett írunk be 0-t, míg a szabadra állítása azt jelenti, hogy 0 helyett teszünk be 1-est. Vagyis az első esetben *eggyel csökkentettük*, míg a második esetben *eggyel növeltük* a szemafor korábbi értékét. Ha a P primitívünk ténylegesen *csökkenti eggyel* a szemafor értékét – és nem *beírja a 0-t* – míg a V primitívünk *növeli eggyel* a szemafor értékét – és nem *beírja az 1-est* –, akkor bináris szemaforok esetén kicsit bonyolultabban ugyan, de ugyanazt a hatást érjük el, mint az eredeti verzióval. (Megjegyzés: ez például az a már említett bonyolultabb eset, amikor a V primitív is több utasításból áll, hiszen tipikusan egy memóriában tárolt számértéket nem lehet növelni, csak úgy, ha növelés előtt behozzuk az akkumulátor regiszterbe a változó értékét, ott hozzáadunk egyet, a növelés után pedig visszaírjuk a memóriába.) Viszont ez a csökkentő-növelő megoldás nem csak *bináris*, hanem a több értéket is felvehető szemaforok kezelésére is jó lesz, amint azt rövidesen látni fogjuk.

Hát akkor menjünk tovább! Képzeljük el, hogy most egy olyan postaládánk van, amelybe nem egy, hanem több, mondjuk *N* levél fér el. Nyilvánvaló, hogy a postaládához való hozzáférést úgy lehet gyorsítani, hogy mielőtt harcba szállnánk a postaláda használati jogáért, megnéznénk, hogy az általunk kívánt művelet egyáltalán elvégezhető-e. Ha mi például egy termelő folyamat vagyunk és látjuk, hogy minden hely betelt, felesleges küzdenünk a postaládáért, hiszen ha meg is szereznénk, úgysem tudnánk oda levelet betenni, viszont mások elől - például egy fogyasztó elől, aki nekünk helyet tudna csinálni egy levél kiolvasásával... - csak feleslegesen foglalnánk a postaládát. Jobb lenne, ha ilyenkor várakoznánk, majd egy kicsivel később megnéznénk, hogy lett-e már hely. Ennek megvalósítására rendeljünk a postaládához két újabb, ún. **torlódásvezérlő szemafort**. Az egyik szemafort hívjuk mondjuk TELE-nek, és mutassa ez a postaládában lévő levelek, azaz *tele helyek számát*, míg a másikat ÜRES-nek, mely mutassa a postaládában lévő *üres helyek számát*. Ezek a szemaforok többértékű, azaz **nembináris szemaforok** lesznek, hiszen 0 és a postaláda kapacitását jelző N közötti értékeket vehetnek fel (a 0-t és az N-et is beleértve).



5.10 ábra Torlódásvezérlő nembináris szemaforok

Nézzük meg, hogy most mit kell csinálni a termelőnek!

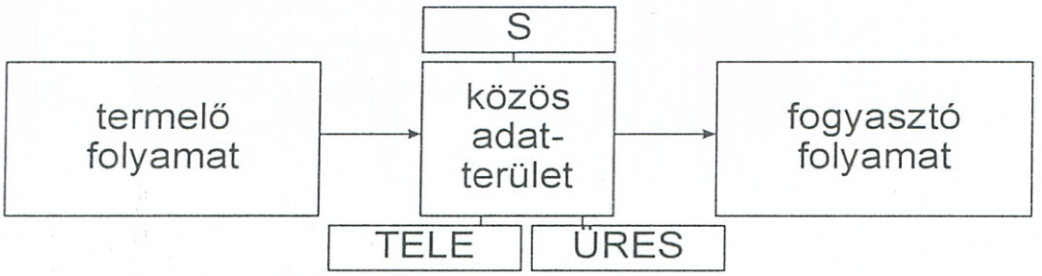
1. Meg kell néznie, hogy van-e üres hely. Ha nincs, (azaz ÜRES=0), akkor várakozik, ha volt, akkor lefoglal magának egy üres helyet az ÜRES értékének eggyel csökkentésével. DE HISZEN PONT EZT CSINÁLJA A MÓDOSÍTOTT **P** PRIMITÍV!
2. Miután lefoglalt egy helyet - versenybe száll a postaláda használati jogáért. (Hiszen elképzelhető, hogy van ugyan lefoglalt helye, de éppen valaki más dolgozik a postaládában.) Ezt már megbeszéltük korábban.
3. Használhatja a postaládát, azaz betehet egy levelet oda.
4. Használat után gyorsan felszabadítja a postaládát, hiszen hátha már más is várakozik rá. Ezt szintén láttuk a korábbiakban.
5. Végül, mivel befejezte a levél írását, jelzi, hogy eggyel megnőtt a postaládában lévő levelek száma a TELE szemafor eggyel való növelésével. DE HISZEN PONT EZT CSINÁLJA A MÓDOSÍTOTT **V** PRIMITÍV!

Hasonlóan a fogyasztó programja:

1. Meg kell néznie, hogy van-e levél a postaládában. Ha nincs, (azaz TELE=0), akkor várakozik, ha volt, akkor lefoglal magának egy levelet a TELE értékének eggyel csökkentésével. DE HISZEN PONT EZT CSINÁLJA A MÓDOSÍTOTT **P** PRIMITÍV!
2. Miután lefoglalt egy levelet - versenybe száll a postaláda használati jogáért. (Hiszen elképzelhető, hogy van ugyan lefoglalt levele, de éppen valaki más dolgozik a postaládában.) Ezt már megbeszéltük korábban.

3. Használhatja a postaládát, azaz kivehet egy levelet onnan.
4. Használat után gyorsan felszabadítja a postaládát, hiszen hátha már más is várakozik rá. Ezt szintén láttuk a korábbiakban.
5. Végül, mivel befejezte a levél olvasását, jelzi, hogy eggyel megnőtt a postaládában lévő üres helyek száma az ÜRES szemafor eggyel való növelésével. DE HISZEN PONT EZT CSINÁLJA A MÓDOSÍTOTT V PRIMITÍV!

Nézzük meg mindezt a P és V primitívek használatával a 6.11. ábra segítségével! (A hozzáférést vezérlő szemaforunkat - az előző példához hasonlóan - hívjuk most is „S"-nek!)



P(ÜRES);	P(TELE);
P(S);	P(S);
ÍRÁS A MEMÓRIÁBA	OLV. A MEMÓRIÁBÓL
V(S);	V(S);
V(TELE);	V(ÜRES);

5.11 ábra P és V primitívek használata nembináris szemaforokkal

Foglaljuk össze, hogy milyen feltételek mellett működik a megoldásunk!

- A SZABAD = 1, a TILOS = 0.
- A P primitív *eggyel csökkenti,* a V primitív *eggyel növeli* a paraméterül kapott szemafor értékét.
- Mielőtt a működést elkezdtük, a következő *kezdőértékekre állítottuk be* (azaz így *inicializáltuk*) a szemaforokat:

  S = SZABAD (hiszen kezdéskor senki nem használja a postaládát)

  TELE = 0 (hiszen kezdetben nincs levél)

ÜRES = N (hiszen kezdetben nincs levél, azaz minden hely üres)

Ebben a fejezetben mélyebben megismerkedhettünk az erőforrás kezelés általános szabályaival, fontosabb stratégiáival. Láttuk, hogy a nem kellőképpen megválasztott módszer hogyan vezethet kiéheztetéshez vagy holtponthoz. A holtpont megelőző stratégiák közül részletesen tárgyaltuk a bankár algoritmus működését.

A fejezet a közösen használt erőforrások problémáinak tárgyalásával zárult.

5.8 Ellenőrző kérdések

1. Mi az erőforrás? Hogyan csoportosíthatók az erőforrások? Írjon példákat!
2. Milyen holtpont felszámolási módokat ismer?
3. Mit lehet tenni egy létrejött holtpont felszámolása érdekében?
4. Melyek lehetnek az áldozat kijelölés szempontjai holtponti állapotban lévő folyamatok megszüntetése esetén?
5. Mutassa be a holtpont megelőzési stratégiák működési elvét!
6. Milyen holtpont megelőzési módszereket ismer? Jellemezze őket röviden!
7. Mi a holtpont? Ismertesse a holtpont kialakulásának feltételeit!
8. Mi a kiéheztetés? Hogyan függ össze a kiéheztetés veszélye és az alkalmazott erőforrás-kezelő stratégia?
9. Mikor mondjuk azt, hogy egy állapot (holtpont szempontjából) biztonságos?
10. Ismertesse a bankár algoritmus működési elvét!
11. Mutassa be a bankár algoritmus alkalmazásának előnyeit, hátrányait és korlátait!
12. Mi az erőforrás foglalási gráf? Hogyan jelentkezik a holtpont az erőforrás foglalási gráfon?

13. Mi a postaláda? Hogyan használható a P és a V primitív postaláda vezérlésére?
14. Mit csinál a P és V primitív? Részletezze felépítésüket!
15. Mi a kölcsönös kizárás? Adjon példát olyan esetre, ahol a kölcsönös kizárást biztosítani kell! Hogyan lehet ezt megvalósítani?
16. Milyen esetekben van szükség nembináris szemaforra? Hogyan általánosítható a P és V primitív ennek kezelésére?

6. Folyamat- és processzorkezelés

Az erőforrás kezelés általános ismertetése után áttérünk az egyik legfontosabb erőforrás, a processzor kezelés tárgyalására. Ennek érdekében részletesebben áttekintjük a folyamatokkal kapcsolatos műveleteket, majd példákon keresztül mélyedünk el a leggyakoribb processzorütemezési algoritmusokban.

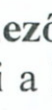
A folyamatok és a processzor kapcsolata igen szoros, a folyamatok minden utasítását a processzor hajtja végre. Egy folyamat elindításától befejezéséig folyamatosan igényli a processzor közreműködését. Rendszerünkben több folyamat is fut, ezért a precíz működés érdekében pontos időzítésre van szükség. A processzor kihasználtsági foka, különösen régebben, amikor a hardver igen drága volt, elsőrendű fontossággal bírt, de manapság sem elhanyagolható. A felhasználók szempontjából azonban nem ez a kulcskérdés. A felhasználó akkor érzi jól magát, és akkor elégedett a számítógép működésével, ha az a programjait a lehető legrövidebb idő alatt végrehajtja. A kihasználtság és a sebesség tehát egyaránt fontos, az operációs rendszereknek mindkét feltételt teljesíteniük kell.

A következőkben egy folyamat sorsát követjük végig, és megvizsgáljuk azokat a folyamatokat, amelyek a végrehajtást vezérlik, annak sebességét befolyásolják. Az idővel való gazdálkodást **ütemezésnek** (scheduling) nevezzük. Az ütemezés során a folyamatok állapota változik meg. Attól függően, hogy milyen állapotok között történik váltás, az ütemezők több szintjét definiálhatjuk.

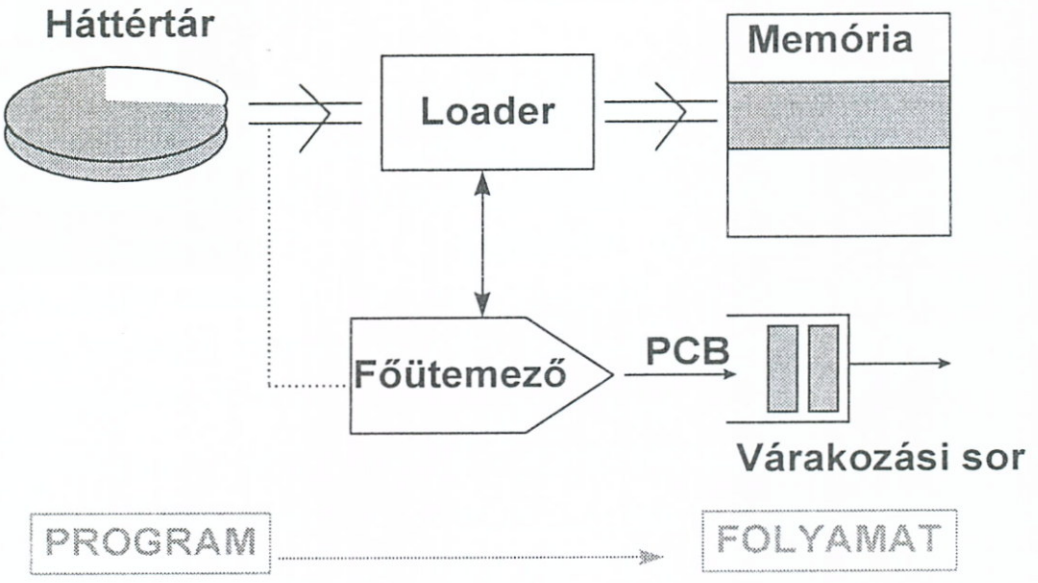
6.1 Folyamatok létrehozása

Amikor a felhasználó úgy dönt, hogy lefuttat egy programot, kevéssé valószínű, hogy a processzor azonnal az ő feladatának végrehajtásába kezd. Már sok folyamat lehet a rendszer felügyelete alatt, és bizony a

többi felhasználó is szorgalmasan termeli a programokat. Tehát már a processzor közelébe kerülésért is meg kell küzdeni. Tegyük fel, hogy olyan számítógépen dolgozunk, ahol - legalábbis részben - a felhasználói programok futás előtt egy háttértárra kerülnek, és az operációs rendszer fenntartja magának a jogot, hogy eldöntse, hogy melyik program kerülhet a memóriába, azaz válhat folyamattá.



A **főütemező** (high-level scheduler) vagy **magas szintű ütemező** választja ki a háttértáron lévő programok közül azt, amelyik az operációs rendszer közvetlenebb felügyelete alá kerülhet, elkezdődhet a végrehajtása, azaz folyamattá válhat. A főütemező működésére az operációs rendszer időléptékével mérve viszonylag ritkán van szükség, ezért *hosszú távú* ütemezőnek (long-term scheduler) is nevezik. A választás a processzor kihasználás szempontjából ideális akkor lenne, ha olyan program kerülne a futó folyamatok közé, amely a többi folyamattal együtt teljesítené azt a követelményt, hogy az összes folyamatra nézve a processzor-igény időtartama megegyezzen a periféria-igény időtartamával. Ez a feltétel azonban általában nem biztosítható, hiszen nem áll rendelkezésre elegendő információ a program igényeiről, így a főütemező kénytelen az érkezési sorrend, vagy az előre meghatározott prioritás alapján dönteni. Interaktív rendszerek esetén még ennyit sem lehet tenni, mert a felhasználó önkényesen, kénye-kedve szerint dönthet, hogy melyik programját indítja. Tisztán interaktív rendszereknél magas szintű ütemező - feladat híján - nincs is, legalábbis döntési jogkörrel nem rendelkezik. Az operációs rendszernek azonban mindenképpen rendelkeznie kell egy olyan komponenssel, amely a futni készülő programokból folyamatokat készít: betölti őket a memóriába és *folyamatleíró blokkot* (**PCB** - **P**rocess **C**ontrol **B**lock) rendel hozzájuk. (Jelenlegi vizsgálatunkban a memória méretét végtelennek tételezzük fel, a betöltő (Loader) programnak is szabad kezet adunk, tehát nem foglalkozunk azzal még, hogy hogyan és hová kerül a program, lényeg, hogy bent legyen.)



6.1 ábra Folyamatok születése

A PCB *minden* információt tartalmaz, ami a folyamat futásához szükséges. Az operációs rendszer szempontjából a PCB maga a folyamat, a folyamat állapota kizárólag attól függ, hogy az operációs rendszer éppen mit kezd a folyamatleíró blokk adataival. Ha az adatok a processzor regisztereibe kerülnek, akkor a folyamat fut, egyébként a PCB feltehetően valamelyik várakozási sor egyik elemét alkotja.

6.2 Műveletek folyamatokkal

6.2.1 Várakozási sorok

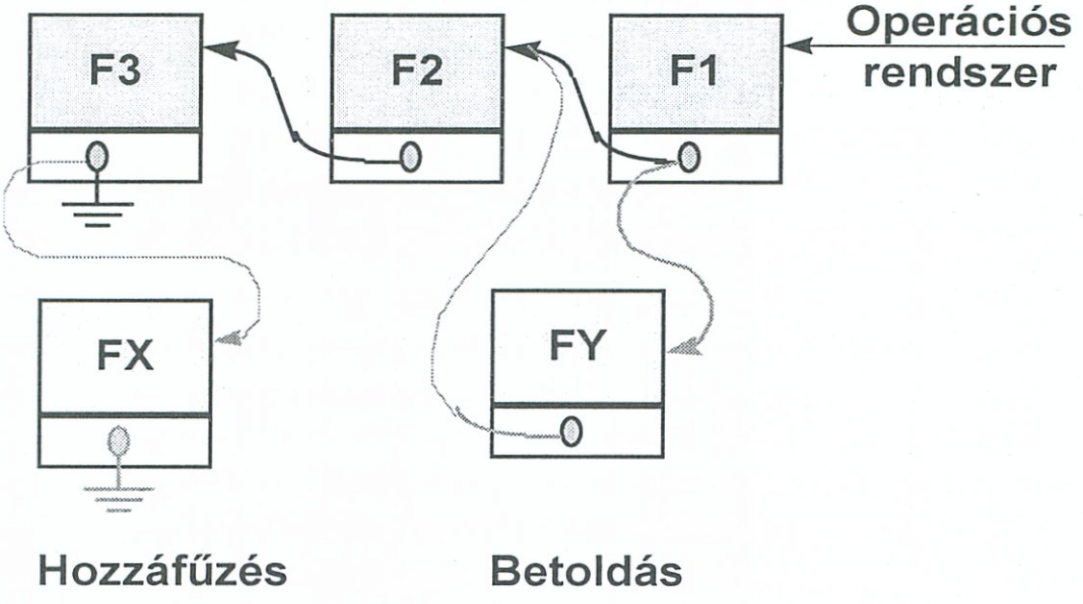
A **várakozási sor** (queue) a leggyakoribb olyan adatstruktúra, mellyel az operációs rendszerek ütemezésével kapcsolatban találkozhatunk. Egyszerre csak egy folyamat futhat, a többi valahol várakozik.

A várakozási sor egy olyan lista, melynek minden eleme egy adatból és egy mutatóból áll. Az adat jelen esetben önmaga is rekord, maga a PCB, a mutató pedig egy másik, azonos típusú listaelemre mutató pointer. A listát a következő Pascal deklaráció definiálja (az operációs rendszereket sohasem írják Pascalban, de működésük leírásához igen gyakran használatos a Pascal szintaxisa):

```
type
   PCB = record
              ProgramCounter: integer;
              ProcessorState: integer;
              Registers: array 1:16 of integer
              {egyéb adatok}
         end;
   PCBpointer = ^PCB;
   Lista = record
              Process: PCB;
              NextProcess: PCBpointer;
         end;
```

6.2 ábra A folyamatok listájának felépítése

A lista adattípus kiválóan alkalmas előre nem meghatározható számú elem rendezett tárolására. Könnyű hozzáfűzni, beékelni elemet, könnyű az elemek sorrendjét megváltoztatni. Az operációs rendszernek mindössze az első adat címét kell ismernie, és máris mindent tud a folyamatokról.



6.3 ábra Várakozási sor módosítása

6.2.2 Környezetváltás

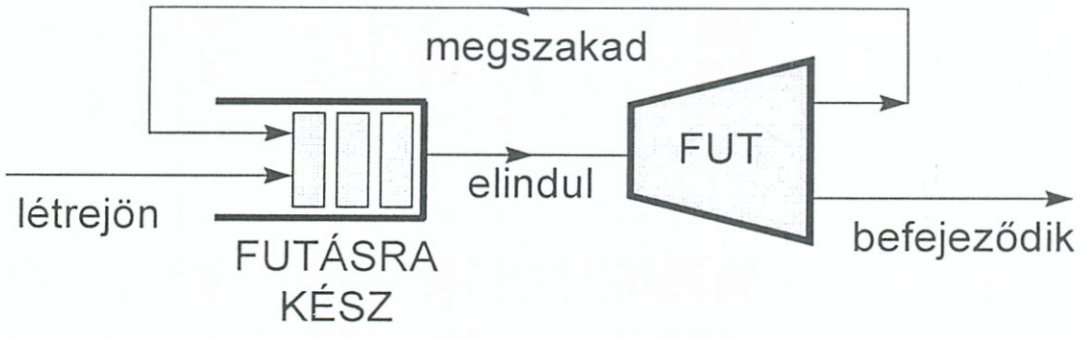
Az operációs rendszer olyan folyamatok összessége, amelyek felhasználói folyamatokkal végeznek műveleteket. A felhasználói folyamatok, illetve az őket reprezentáló folyamatleíró blokkok tehát a rendszerfolyamatok változóinak tekinthetők. Az operációs rendszer működése úgy is felfogható, hogy annak folyamatai az egyik várakozási sorból kiveszik a folyamatot, valamit csinálnak vele, és beteszik egy másikba, esetleg az eredeti sor végéhez fűzik.

A PCB-k, illetve a benne foglalt adatok képezik az operációs rendszer folyamatainak környezetét, kontextusát (context). Ha a rendszerfolyamat egy másik felhasználói folyamattal akar foglalkozni, a környezetét kell átkapcsolni. Ez a művelet a **környezetváltás** (context switching).

A környezetváltás előtt természetesen az előző folyamat állapotában beállt változásokat, regisztertartalmat, a környezeti változók értékét, illetve a használt perifériák állapotát a lecserélni kívánt folyamat PCB-jében regisztrálni kell.

6.3 A folyamatok alapállapotai

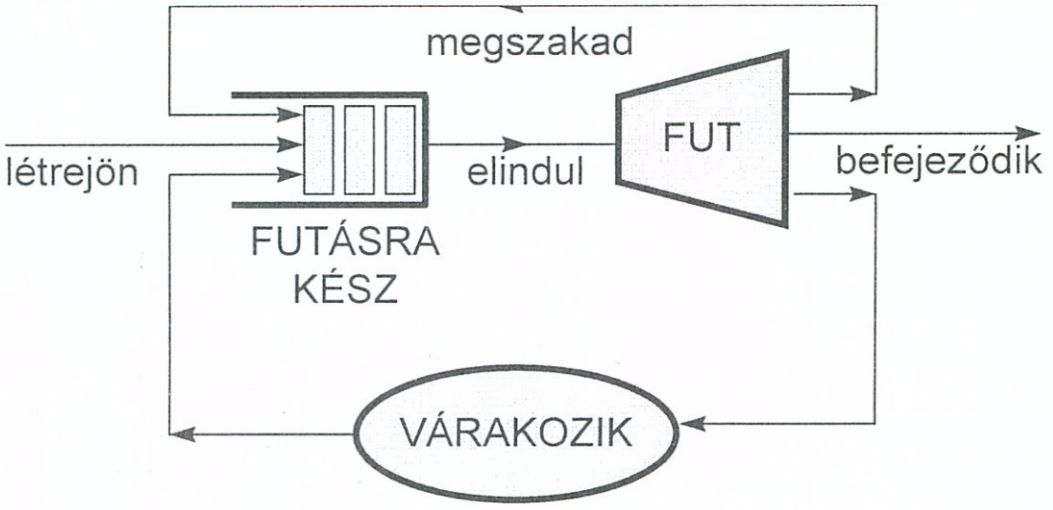
Mikor a főütemező elvégezte feladatát, és létrehozta az új folyamatot, az a *futásra kész* folyamatok sorába kerül. Innen az operációs rendszer egy folyamata (az alacsony szintű ütemező, nemsokára részletesen megvizsgáljuk) kiveszi, és átadja futtatás céljából a processzornak, azaz *elindul*.



6.4 ábra Folyamatok állapotai I.

A folyamat mindaddig futhat, amíg be nem fejeződik, vagy az operációs rendszer el nem veszi tőle a lehetőséget, például azért, mert lejárt a neki szánt idő, vagy magasabb prioritású, fontosabb folyamat érkezett. A futás

megszűnéséért azonban nem mindig tehető felelőssé az operációs rendszer. Ha a folyamat például azért akad el, mert adatra kell várnia egy külső egységről, vagy éppen nyomtatásba kezdett, teljesen jogos, hogy átengedje a helyét addig egy másik türelmetlenül várakozó folyamatnak. Blokkvázlatunk egy újabb állapottal bővül:



6.5 ábra Folyamatok állapotai II.

A folyamatok tehát várakozó állapotba kerülnek amíg a várt esemény be nem következik, utána visszatérnek a futásra kész folyamatok sorába. A 6.5. ábrán az egyszerűség kedvéért csak egy állapottal ábrázolt várakozás általában várakozási sorok csoportja, melyekben külön-külön állnak sorba a lemezre, nyomtatóra, hálózatra stb. vágyó folyamatok.

Egy folyamat tehát az időzítés szempontjából alapvetően három alapállapotot vehet fel, az állapotok között négyféle átmenet lehetséges.

Az alapállapotok:

1. **FUTÁSRA KÉSZ** (ready): Ebben az állapotban - a processzoron kívül - minden erőforrás a folyamat rendelkezésére áll. A folyamatok létrejöttüket követően a FUTÁSRA KÉSZ állapotba kerülnek.
2. **FUT** (running): A processzor annak a folyamatnak az utasításait hajtja végre, amelyik ebben az állapotban van.
3. **VÁRAKOZIK** (blocked): Ha futó folyamat olyan erőforrást igényel, amelyik pillanatnyilag nem áll rendelkezésre, vagy a további futásához szüksége van egy másik folyamat által szolgáltatandó eredményre, ebbe az állapotba kerül.

Az állapot átmenetek:

1. **Elindul** (dispatch): A központi egység felszabadulása esetén az alacsonyszintű ütemező a FUTÁSRA KÉSZ állapotban lévő folyamatok közül választja ki azt, amelyik a FUT állapotba kerülhet.
2. **Megszakad** (timer runout): Ha a futó folyamat számára biztosított idő lejár, visszakerül a FUTÁSRA KÉSZ állapotba. (Nem mindegyik operációs rendszer teszi lehetővé, hogy egy folyamatot megszakítsunk, ehhez a rendszernek, illetve a folyamatnak *megszakíthatónak* (preemptív) kell lennie).
3. **Vár** (wait, block): Amennyiben olyan erőforrásra van szüksége, amely éppen foglalt, a FUT állapotból a VÁRAKOZIK állapotba jut.
4. **Feléled** (awake): A várt esemény bekövetkezése esetén a folyamat FUTÁSRA KÉSZ állapotba kerül, vagyis újra beáll a processzorra várakozó folyamatok sorába.

6.4 Felfüggesztett állapot

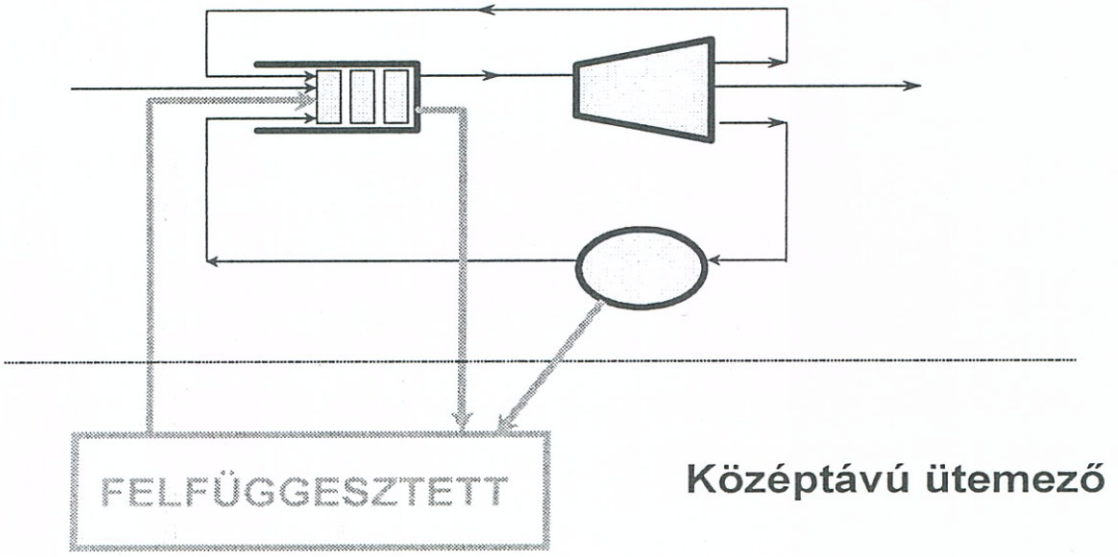
Az alapállapotok tárgyalásánál a **futás megszakadásának** lehetséges okaiért egyrészt a futó folyamatot (befejeződik, perifériára vár), másrészt az operációs rendszer kifejezetten a processzor kezelésével foglalkozó rendszerfolyamatát, az alacsony szintű ütemezőt tettük felelőssé. Léteznek azonban más okok is.

Nagyobb rendszereknél indokolt egy olyan mechanizmus megvalósítása, mely **folyamatosan figyeli a rendszer** terhelését, és ha a normális működés zavarait észleli, beavatkozik. Milyen zavarokról lehet szó? Ha túlságosan sok folyamat került futásra kész állapotba, előfordulhat, hogy egyiknek sem jut megfelelő mennyiségű erőforrás, és a processzor idejének javát rendszeradminisztrációval, például környezetváltásokkal tölti.

A **középtávú**, vagy más néven **közbenső szintű** ütemező (medium-term scheduler) ilyenkor beavatkozik, és egyes folyamatok felfüggesztésével, vagy a prioritási szintek időszakos változtatásával helyreállítja a hatékony működés feltételeit. Például egy gyakran futó, magas prioritású folyamat prioritását ideiglenesen csökkenti vagy egy kiéheztetéstől szenvedő, alacsony prioritású folyamat prioritását ideiglenesen növeli. Sok esetben a holtpont detektálásáról és a holtpontot okozó folyamatok

felfüggesztéséről is a közbülső szintű ütemező gondoskodik. A folyamatok által elfoglalható állapotok száma eggyel növekszik:

A **felfüggesztett** (suspended) folyamatok időlegesen kiszállnak az erőforrásokért folyó versenyből, de folyamatok maradnak, és futásuk a helyzet jobbra fordulása után bármikor folytatható.



6.6 ábra Felfüggesztett állapotok

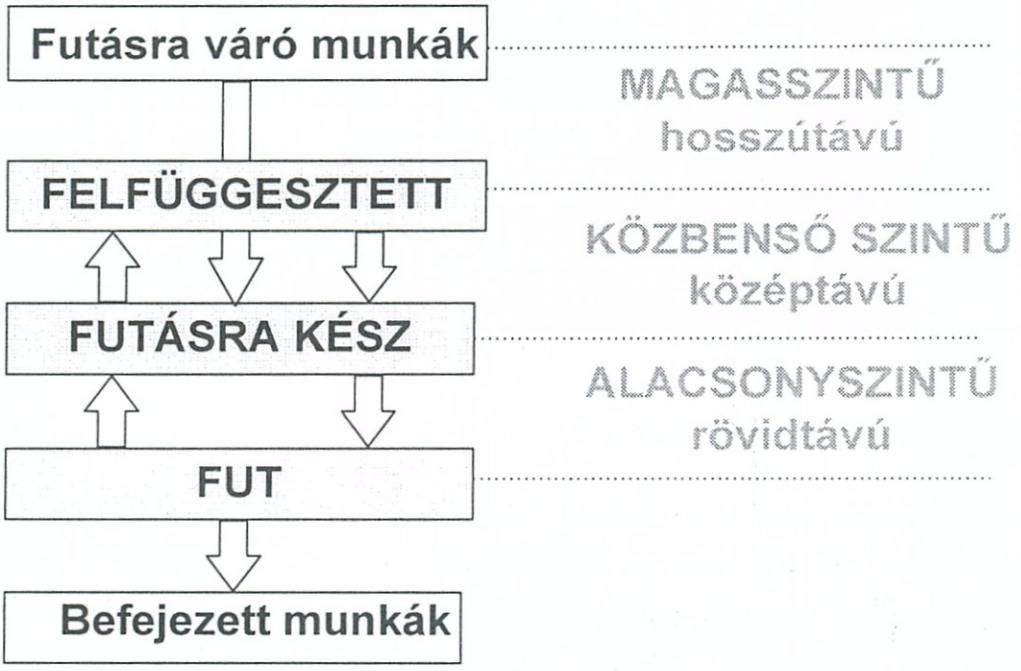
6.5 Processzorütemezés

Nem volt szó még arról, hogy ha nem kell perifériára várni, a rendszer is megfelelően viselkedik, és megszakítás sem jön éppen, akkor milyen folyamat osztja el a processzoridőt a futásra várakozó, kizárólag processzor hiányban szenvedő folyamatok között? A keresett folyamat az operációs rendszerek talán legaktívabb komponense, az **alacsony szintű ütemező** (rövid távú ütemező, short-term scheduler, low level scheduler, diszpécser).

Az alacsony szintű ütemezés célja, hogy az operatív tárban lévő, kiszolgálásra váró, azaz futásra kész folyamatok számára biztosítsa a processzor használatának jogát, azaz tegye lehetővé az azokban található utasítások végrehajtását. Az alacsony szintű ütemezővel szemben támasztott legfőbb követelmény a gyorsaság, hiszen ha a folyamatok

közötti átkapcsolás túl hosszú időt venne igénybe, a rendszer hatékonysága jelentősen csökkenne.

Volt már *magas-, közbenső* és *alacsonyszintű*, illetve más megközelítésben *hosszú-, közép- és rövidtávú* ütemezőnk. Az alábbi ábra az egyes ütemezési szintek viszonyát, ha nem is teljesen precízen, de feltétlenül szemléletesen mutatja be:



6.7 ábra Ütemezési szintek

Az alacsony szintű ütemező feladata tehát, hogy a processzort a futásra kész folyamatok között igazságosan és hatékonyan ossza el. Az igazság és a hatékonyság mennyiségileg nehezen kezelhető fogalom, ezért precízebb, mérhető jellemzőket célszerű bevezetni. Sokféle statisztika készíthető, azonban talán az alábbiak a legjellemzőbbek:

1. **Várakozási idő** (waiting time, missed time): megadja, hogy a folyamat mennyi időt töltött tétlen várakozással.
2. **Átfutási idő** (penalty ratio): a folyamat érkezésétől annak befejezéséig eltelt idő.
3. **Válaszidő** (response time) az az idő, amely a folyamat rendszerbe érkezésétől (indításától) az első futás kezdetéig telik el. Interaktív rendszereknél nagyon fontos, hogy a felhasználók megnyugodjanak, programjuk elindult, nincs nagyobb baj.

A fenti paramétereken kívül mindhárom esetben nagyon tanulságos a *szórás*, illetve a folyamat igényére vonatkoztatott *arányszám* is.

$T_{ÉRKEZÉS}$	**A folyamat a futásra kész sorba érkezik**
T_{KEZDET}	**A futás kezdőidőpontja**
$T_{BEFEJEZÉS}$	**A folyamat elhagyja a rendszert**
$T_{IGÉNY}$	**A folyamat processzoridő igénye**

$$T_{VÁRAKOZÁS} = T_{BEFEJEZÉS} - T_{ÉRKEZÉS} - T_{IGÉNY}$$

$$T_{VÁLASZ} = T_{KEZDET} - T_{ÉRKEZÉS}$$

$$T_{ÁTFUTÁS} = T_{BEFEJEZÉS} - T_{ÉRKEZÉS}$$

6.8 ábra Ütemezési statisztikák

6.5.1 Előbb jött, előbb fut (FCFS)

A legegyszerűbben megvalósítható ütemezési módszer. Semmi más nem kell hozzá, mint egy jól ismert várakozási sor. A folyamatok egyszerűen érkezési sorrendben kapják meg a processzort (**F**irst **C**ome **F**irst **S**erved - **FCFS**), és azt egészen addig maguknál tarthatják, amíg be nem fejeződnek, vagy egy periféria művelet miatt várakozni nem kényszerülnek.

A

Előbb jött, előbb fut - FCFS
A folyamatok érkezési sorrendben futhatnak
Előny: egyszerű, biztos
Hátrány: a folyamatok érkezési sorrendjétől nagymértékben függő várakozási idő

§

módszer működéséből fakadó előnye, egyszerűsége mellett az, hogy biztosan mindegyik folyamat sorra kerül előbb–utóbb. De az is látszik, hogy inkább utóbb. Ha egy nagy futási idejű folyamat keveredik a rendszerbe, folyamatok egész sorát tarthatja fel (kamion hatás). A többi erőforrás is feltehetően kihasználatlanul marad, hiszen a rövidebb

munkák periféria igényei a hosszú folyamat futása alatt kielégítést nyertek, azok visszakerültek a futásra várók sorába. Az eljárás relatíve a hosszú folyamatoknak kedvez, nekik *processzorigényükhöz képest* viszonylag keveset kell várniuk.

Az alábbi példában az eljárás minősítésére az átlagos várakozási időt választottuk. Az algoritmus bemenő adatai a tapasztalatból vett, becsült értékek vagy egyszerűen véletlen számok lehetnek. A számítás elvégzéséhez ismerni kell az egyes folyamatok érkezési idejét, valamint processzor igényét. Példánkban (és a többi módszer leírását követő példákban) a következő adatokkal dolgozunk (az idő mértékegysége most érdektelen):

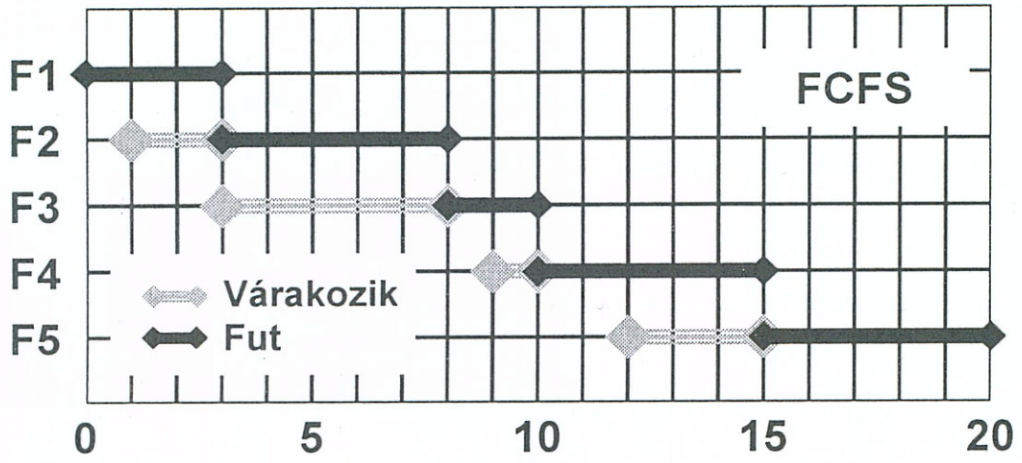
Folyamat	Érkezési idő	Processzor igény
F1	**0**	**3**
F2	**1**	**5**
F3	**3**	**2**
F4	**9**	**5**
F5	**12**	**5**

A feladat megoldásához a táblázatot ki kell bővíteni néhány oszloppal.

Folyamat	Érkezési idő	Processzor igény	Kezdési idő	Befejezési idő	Várakozási idő
F1	**0**	**3**	**0**	**3**	**0**
F2	**1**	**5**	**3**	**8**	**2**
F3	**3**	**2**	**8**	**10**	**5**
F4	**9**	**5**	**10**	**15**	**1**
F5	**12**	**5**	**15**	**20**	**3**
		Összes várakozási idő:			**11**
		Várakozási idők átlaga:			**11:5=2,2**

A várakozási idő a kezdési idő és az érkezési idő különbsége. Ennél a módszernél egy folyamat csak akkor kezdődhet, ha a másik már befejeződött. Az átlagos várakozási idő az egyes folyamatok várakozási idejének összege, osztva a folyamatok számával (jelen esetben 5). Itt azonban meg kell jegyeznünk, hogy a valóságban nem igaz az, hogy egy

folyamat azonnal elkezdődhet, amint egy másik befejeződött. Hiszen a „régi" folyamat állapotának mentése, a következő folyamat kiválasztása, állapotának betöltése időt igényel. Tehát a mi számításainkból adódó várakozási értékeknél a valóságban rosszabb a helyzet, a várakozási idő a **környezetváltások számával** arányosan nő. Nézzük meg ugyanezt a példát egy idődiagramon:



6.5.2 Legrövidebb előnyben (SJF)

Míg az előző módszer relatíve a leghosszabb folyamatoknak kedvez, ez az ütemezési eljárás előnyben részesíti a rövid processzoridő igényű munkákat (**S**hortest **J**ob **F**irst - **SJF**) azaz a *várakozó folyamatok közül* mindig a legrövidebbnek adja oda a processzor használati jogát. Ha több egyforma idejű folyamat is lenne, közülük az futhat, amelyik előbb érkezett (FCFS).

§

Legrövidebb előnyben - SJF
A folyamatok közül először a legrövidebb fut
Előny: a legrövidebb átlagos várakozási időt adja
Hátrány: a hosszú folyamatoknál kiéheztetés léphet fel!

Az FCFS módszernél a folyamatok processzor igényét csak a példa feladat miatt kellett ismerni, itt azonban nagyon fontos, hiszen ettől függ az alacsony szintű ütemező döntése! Ez a módszer egyik gyengéje, hiszen a folyamatok processzorigénye sokszor nem ismerhető előre, a kívánt időtartam csak statisztikailag, vagy a programozó becslése alapján

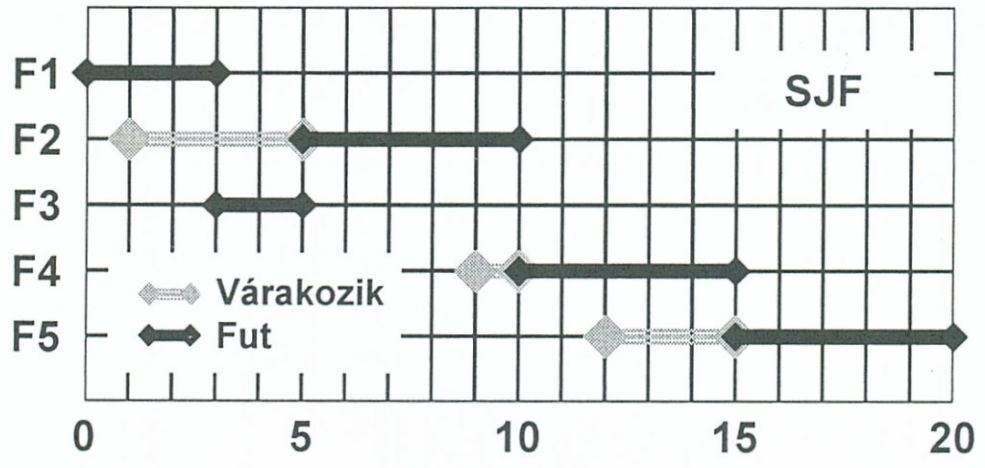
állapítható meg. A következő probléma, hogy ha a rendszerben folyamatosan érkeznek a munkák, előfordulhat, hogy egy hosszú folyamat sohasem kerül sorra, mindig lesz egy rövidebb, amelyik elé vág (kiéheztetés).

Az SJF algoritmus garantálja a legrövidebb átlagos várakozási időt, (de miért örüljön ennek egy örökre kizárt hosszú folyamat?) A számítási példa egyszerű, azonban bonyolultabb esetekben célszerű külön oszlopot fenntartani az egyes folyamatok befejezésekor várakozó folyamatoknak, illetve közülük a legrövidebbnek. A példánk az SJF algoritmusra alkalmazva:

Az SJF algoritmusnak elképzelhető preemptív változata is, azaz ha egy folyamat futása közben érkezik egy nálánál rövidebb időigényű folyamat, azonnal megkapja a vezérlést.

Folyamat	Érkezési idő	Processzor igény	Kezdési idő	Befejezési idő	Várakozási idő
F1	0	3	0	3	0
F2	1	5	5	10	4
F3	3	2	3	5	0
F4	9	5	10	15	1
F5	12	5	15	20	3
	Összes várakozási idő:				8
	Várakozási idők átlaga:				1,6

És következzen az erre az esetre érvényes időidiagram!



6.5.3 Körben járó algoritmus (RR)

A Körben járó (**R**ound **R**obin - **RR**) algoritmus az „hátulsó pár előre fuss" elvet valósítja meg. Interaktív rendszerekben, ahol a felhasználók a terminálok előtt ülve várják programjuk eredményeit, nem engedhető meg az, hogy egy folyamat addig használja a processzort, amíg akarja (az előző két módszernél ezt tételeztük fel). Azért, hogy mindegyik folyamat viszonylag egyenletes sebességgel folyamatosan előre haladhasson, az RR módszer használatakor egy bizonyos *időszelet* (time slice) eltelte után az ütemező elveszi a futó folyamattól a processzort (az algoritmus tehát, az eddigiekkel szemben, preemptív). Az addig futó folyamat a várakozási sor végére kerül, a következő folyamat futhat, de ő is csak maximum egy időszeletnyiig.

§

Körben járó - RR
Minden folyamat egy adott időszeletig futhat, majd újra sorba kell állnia
Előny: demokratikus, a legrövidebb a válaszideje
Hátrány: több környezetváltást igényel

Minden egyes folyamatváltásnál környezetváltásra is sor kerül, ami az időszelet nem megfelelő megválasztása esetén túlsúlyba is kerülhet. A megnövekedett adminisztrációt már az egyszerű példa kapcsán is megtapasztalhatjuk, most már végképp kevés az egy táblázat. A folyamatok adatai a régiek, de mivel egy-egy folyamat nem egy lépcsőben fut le, a sorok száma már több kell legyen, mint a folyamatok száma, sőt előre nem is lehet tudni, hogy mennyi. Az időszelet hossza legyen 4 egység !

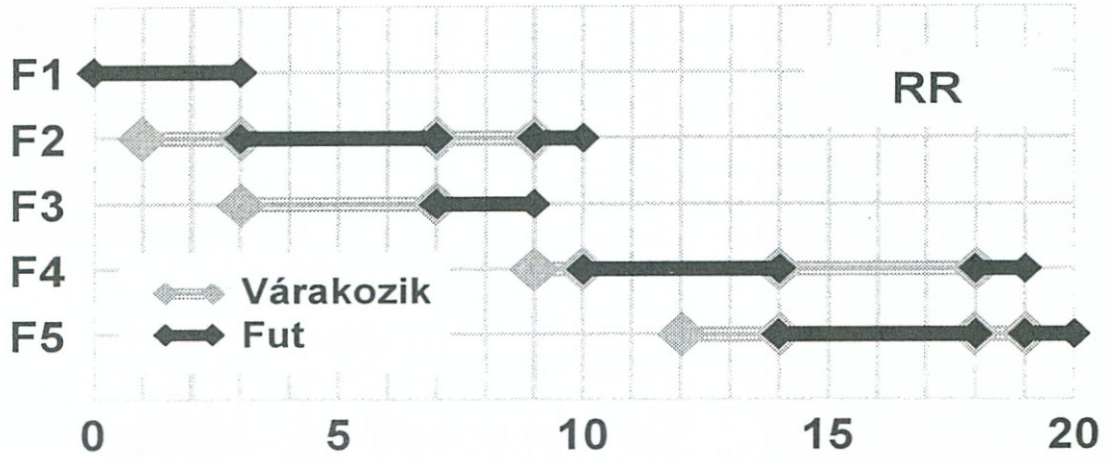
Folyamat	Érkezési idő	Processzor igény	Befejezési idő	Várakozási idő
F1	0	3	3	0
F2	1	5	10	4
F3	3	2	9	4
F4	9	5	19	5
F5	12	5	20	3

Folya-mat	Érk. idő	Kezd. igény	Kezd. idő	Bef. idő	Vár. idő	Maradék igény	Befejezéskor várakozó folyamatok
F1	0	3	0	3	0	0	F2,F3
F2	1	5	3	7	2	1	F3,F2
F3	3	2	7	9	4	0	F2,F4
F2*	(7)	(1)	9	10	2	0	F4
F4	9	5	10	14	1	1	F5,F4
F5	12	5	14	18	2	1	F4,F5
F4*	(14)	(1)	18	19	4	0	F5
F5*	(18)	(1)	19	20	1	0	-
Összes várakozási idő:					**16**		
Várakozási idők átlaga:					**3,2**		

A várakozási idők átlaga az összes várakozási idő osztva a *folyamatok számával*, jelen példánkban 16/5=3,2.

Megjegyezzük, hogy a táblázatban *-gal jelöltük meg azokat a folyamatokat, amelyeket felfüggesztettek, majd újra sorra kerültek. Ilyenkor az érkezési idő oszlopban a felfüggesztés ideje, míg a kezdeti igény oszlopban a maradék idő szerepel. Azért, hogy ezt megkülönböztessük a tényleges érkezési időtől és kezdeti igénytől, zárójelbe tettük.

Érdemes itt is megnézni, hogy az idő függvényében mi is történt:



Az eddig ismertetett algoritmusoknál sem közömbös, de itt különösen indokolt megemlíteni, hogy a környezetváltásra fordított idő figyelembe vétele nélkül a folyamatok érkezését és igényét a legnagyszerűbben leíró adatokkal számolva is torz eredmény születik.

6.5.4 Prioritásos és preemptív módszerek

A valós rendszerekben a folyamatok általában nem egyforma fontosságúak, hanem fontosság szerint rangsorba állíthatjuk őket, azaz *prioritást* rendelhetünk hozzájuk. A prioritás lehet *belső* (ez esetben a prioritást az operációs rendszer határozza meg), illetve lehet *külső* (ez esetben a prioritási szintet általában a felhasználó jelöli ki). Egy tipikus belső prioritást használó algoritmus az SJF, ott a folyamatok annál fontosabbak, minél rövidebbek. A prioritásos módszerek értékelésére az SJF-nél elmondottak általánosíthatók: előnyük, hogy az a szempont, ami alapján a prioritást meghatároztuk az ütemezés során érvényesíthető, ugyanakkor legnagyobb hátrányuk, hogy az alacsony prioritású folyamatok számára mindig *kiéheztetés-veszélyesek*. Az erőforrás kezelésnél tanultuk, hogy kiéheztetés-veszélyes algoritmusokat nem szabad használnunk, ezért a prioritásos módszereknél mindig szükség van olyan módosításra, amely a kiéheztetés lehetőségét megszünteti. A gyakorlatban általában két fő módszer terjedt el.

Az egyik módszer azon alapszik, hogy ha – tipikusan a közbenső szintű ütemező – detektálja, hogy bizonyos folyamatok már régóta várakoznak, akkor ideiglenesen módosítja a prioritási szinteket: vagy a fontos folyamatok prioritási szintjét csökkenti (ez a ritkább) vagy a kevésbé fontos, kiéheztetés alatt álló folyamatok prioritási szintjét növeli (ez a gyakoribb).

A kiéheztetést elkerülő másik módszer azon alapszik, hogy – megint tipikusan a közbenső szintű ütemező – figyeli, hogy egymás után hányszor kapta meg magas prioritású folyamat a processzort, és ha ez egy bizonyos számot meghalad, akkor mindenképpen egyszer (vagy néhányszor) alacsonyabb prioritású folyamatot választ, még akkor is, ha van várakozó magas prioritású folyamat is. Ez a kiválasztási arány elég szélsőséges lehet (például 256:1 vagy még nagyobb), hiszen a magas prioritású folyamatokat előnyben kell részesíteni, de a lényeg, hogy a kis prioritású folyamatok, ha lassan is, de folyamatosan előrehaladhatnak.

A prioritás az RR módszerrel is kombinálható, mégpedig kétféleképpen. Az egyik, a gyakrabban alkalmazott módszer szerint, a folyamatok időszelete – prioritási szintjüktől függetlenül – azonos. (Ezt egyszerűbb hardver úton megvalósítani, és mint tudjuk ütemezésnél a legfőbb szempont a gyorsaság!) Ilyenkor nem egy, hanem – a prioritási szinteknek mefelelően – több várakozási sor van. A folyamatok az időszeletük lejártakor a prioritásuknak megfelelő sor végére kerülnek vissza. Csak akkor választhatunk egy alacsonyabb prioritású sorból, ha a magasabb prioritású sor(ok) üres(ek). A kiéheztetést itt úgy szokták elkerülni, hogy ha vannak hosszú ideje várakozó alacsony prioritású folyamatok, akkor a magas prioritású folyamatok időszeletük lejárta után ideiglenesen nem a „saját" sorukba, hanem eggyel lejjebbi sorba kerülnek vissza.

A másik módszer esetén csak egy várakozási sor van, de az időszelet nagysága függ a folyamat prioritásától, azaz minél fontosabb egy folyamat, annál hosszabb az időszelete. Ilyenkor kiéheztetés ki sem alakulhat, hiszen egy megszakított folyamat mindig az egyetlen várakozási sor végére kerül vissza, azaz biztos, hogy csak akkor kerülhet újra sorra, ha az összes többi, előtte várakozó folyamat már sorra került. Ugyanakkor viszont a prioritás-függő időszeletet hardver úton nehezebb megvalósítani, másrészt ha sok alacsony prioritású folyamatunk van, akkor a fontosaknak is sokáig kell várniuk, ezért ezt a módszert viszonylag ritkán alkalmazzák.

Az ütemezési módszereknél használt másik tulajdonság a preempció. Ez azt jelenti, hogy ha az éppen futó folyamatnál fontosabb érkezik, akkor az azonnal *megszakítja* az éppen futót. Ezáltal a prioritással kifejezni szándékozott szempont jobban érvényesíthető, azaz a fontos folyamatok még inkább előtérbe helyezhetők, természetesen a kevésbé fontosak kárára, hiszen most már az sem garancia a folyamat lefutására, ha

megkapta a processzort, mert bármikor jöhet egy nála fontosabb, amelyik elveheti azt tőle (azaz itt még inkább előtérbe kerül a kiéheztetés!).

Tipikus preemptív algoritmus az RR módszer. Az RR módszer tulajdonképpen az FCFS preemptív változata, hiszen a futtatás alapvetően a várakozási sorba érkezés sorrendjében történik, de a futó folyamatnál bárki fontosabbá válik, ha az időszelet lejárt. Az RR értékelésénél megállapítottak szintén általánosíthatók az összes preemtív algoritmusra: azaz a kifejezni szándékozott szempont hatékonyabban érvényesíthető, de a környezetváltások száma megnő, hiszen sokszor lesz olyan, hogy egy futó folyamatot amiatt kell felfüggesztenünk, mert egy nála fontosabb érkezett.

6.6 A Linux folyamatkezelési megoldása

Ütemezés alatt általában a CPU idő elosztását értjük, azonban többfeladatos rendszerekben, amilyen a Linux is, más vonatkozásai is vannak a kérdésnek. A felhasználói folyamatok a kernel rutinjain (és ezen keresztül a többi erőforráson) is osztoznak, ezt az összehangolási feladatot is meg kell oldani.

6.6.1 A kernel folyamatainak szinkronizálása

A kernel folyamatai akkor jutnak szerephez, ha egy folyamat ezt kifejezetten kéri (rendszerhívás, *system call*), vagy közvetve, ha például egy felhasználói folyamat laphibát idéz elő (csapda, *trap*), illetve ha egy hardver eszköz kér kiszolgálást (megszakítás, *interrupt*).

A probléma abból adódik, hogy a kernel műveletei közösen használt adatstruktúrákon hajtódnak végre. Ha egy rendszerfolyamat éppen egy műveletet végez, például a háttértárról olvas a memóriába egy állományt, egy másik, hasonló jellegű, ugyanarra a fájlra vonatkozó kérés miatt nem szakíthatja meg anélkül, hogy egyes adatok (például a fájl mutató) épségét ne veszélyeztetné. Meg kell tehát oldani a nem megszakítható kódrészletek, a kritikus szekciók problémáját.

A Linux a legegyszerűbb megoldást választotta: a kernel folyamatai nem megszakíthatók (*non-preemptive*). Ha éppen egy kernel folyamat fut, amikor a rendszeróra elindítaná CPU ütemező folyamatot, az ütemezés nem indul meg azonnal, hanem csak egy jelzőbit értéke (*need_resched*) mutatja a rendszernek, hogy amint lehetséges, ütemeznie kell.

A felhasználói folyamatok tehát nem okozhatnak gondot. A hardver oldalról érkező megszakítások szintén nem veszélyesek, hiszen sohasem idézik elő közvetlenül a végrehajtási sorrend változását, csak jelzőbitek értékeit állítják be, ezek értékeit pedig az ütemező veszi majd figyelembe legközelebbi futásakor. Probléma csak akkor fordulhat elő, ha maga a kernel folyamat kényszerül futását felfüggeszteni például egy laphiba

miatt, ez azonban gondos programozással elkerülhető: kritikus szekcióban nem szabad felhasználói memóriát használni.

6.6.2 CPU ütemezés

Mikor az éppen futó folyamat valamilyen okból várakozni kényszerül, vagy egyszerűen csak elérkezik az időszelet vége, az alacsonyszintű ütemezőnek el kell döntenie, a várakozó folyamatok közül melyiket futtassa a továbbiakban. A Linux két algoritmust használ, mindkettő megfelel a POSIX előírásainak. Az egyik, a preemptív időosztás az egyszerű batch, vagy interaktív folyamatok számára szolgáló demokratikus módszer, míg a másik, a valós idejű folyamatok számára kialakított szigorú prioritásos rendszer. A folyamatok típusa létrejöttükkor dől el, az erre vonatkozó bejegyzést a folyamatleíró tábla (PCB) tartalmazza.

Az **időosztásos (*time-sharing*) rendszer** ütemezője olyan algoritmus szerint működik, melyben minden folyamat rendelkezik egy bizonyos számú kredit ponttal, és az a folyamat indulhat, amelynek a pontszáma a legmagasabb. Nézzük, hogyan alakulnak ki a pontszámok!

A folyamatok indulásukkor kapnak egy prioritásukkal arányos kiinduló kredit pontszámot. A rendszer órája által periodikusan indított rendszerfolyamat az éppen futó folyamat pontszámát eggyel csökkenti, a futó folyamatok tehát öregednek. Mikor egy folyamat pontszáma eléri a nullát, felfüggesztésre kerül. Ha már egyetlen futóképes folyamat sincs a rendszerben (azaz minden folyamat várakozó, vagy felfüggesztett állapotban van), egy olyan folyamat indul, amely újra ellátja a folyamatokat kredit pontokkal (*recrediting*). A pontszámok a következő algoritmus alapján képződnek:

$$\text{Új kredit} = \frac{\text{Régi kredit}}{2} + \text{Prioritás}$$

Az algoritmus tehát egyszerre veszi figyelembe a folyamat előéletét és fontosságát. Ha egy folyamat nagyon sokat fut, hamar kimeríti a készleteit (és felfüggesztésre kerül), a sokat várakozó folyamatok pedig szépen csendben gyűjtögetik a pontjaikat, és amikor bekövetkezik a várva várt esemény nagy lendülettel (és magas pontszámmal) indulhatnak újra. A módszer tehát megpróbál igazságosan és demokratikusan bánni mind a processzoridő (CPU bound), mind a perifériák (pripherial bound) által korlátozott folyamatokkal. A külön megadott prioritás lehetővé teszi a rendszer finomhangolását. A háttérben futó folyamatokhoz célszerűen alacsonyabb, míg az interaktív folyamatokhoz magasabb prioritás rendelhető.

A **valós idejű (*real time*) feldolgozás**t a Linux szoftver úton valósítja meg. A valós idejű folyamatok mindig elsőbbséget élveznek a többi folyamattal szemben a processzoridő elosztásakor, azonban a szoftver megvalósításból fakadóan az éppen futó interaktív folyamat befejeződését meg kell várniuk.

A Linux kétféle algoritmust valósít meg, ezekhez külön várakozási sor tartozik. Az algoritmusok a várakozó folyamatok közül azt választják, amelyiknek a prioritása a

legmagasabb, ha pedig azonos prioritások között kell dönteniük, a régebben érkezett folyamat részesül előnyben. A kétféle lehetőség között mindössze az a különbség, hogy míg az egyik (non-preemptive, FIFO jellegű) a folyamatra bízza, hogy mikor fejezi be a munkáját, a másik (preemptív, RR jellegű) algoritmus az időszelet kimerítése után elveszi a CPU-t a futó folyamattól.

A **szimmetrikus többprocesszoros rendszereket** (Symmetric Multi Processing – **SMP**) a Linux a 2.0 változattól kezdve támogatja, a további rendszerfejlesztés is nagy mértékben ebben az irányban halad. A legnagyobb nehézséget az okozza, hogy a Linux filozófiája szerint a kernel egységes, monolitikus, annak adatstruktúráihoz csak a kernel folyamatai férhetnek hozzá, azok is csak szigorú szabályok szerint. A legegyszerűbb, – a 2.0-ban megvalósított – módszer ezt úgy biztosítja, hogy kernel folyamat egyszerre csak egy processzoron futhat. A nagy hátrány abból fakad, hogy a folyamatok többsége intenzíven használja a kernel rutinjait, így általában az egyik processzor tétlen.

Σ *A folyamatokkal foglalkozó rész végigkövette a folyamatok egész életútját születésüktől megszűnésükig. Megismerhettük a nyilvántartásukat végző folyamatleíró blokkot (PCB), a várakozási sorok szerepét betöltő listák (queue) feladatát, illetve a folyamatok sorsát igazgató ütemezők funkcióját. Részletesen, példák formájában is foglalkoztunk a három legegyszerűbb rövid távú ütemezési algoritmussal, az FCFS, SJF illetve az RR módszerrel.*

?

6.7 Ellenőrző kérdések

1. Ábrázolja a folyamatok lehetséges állapotait, valamint az átmeneteket! Milyen hatások idézhetik elő az egyes állapot átmeneteket?
2. Hasonlítsa össze a tanult alacsony szintű ütemezési algoritmusokat!
3. Ismertesse az alacsony szintű CPU ütemezés feladatát! Milyen paraméterekkel jellemezhetők az egyes algoritmusok?
4. Mi a kapcsolat az egyes (folyamat) ütemezési szintek között?

5. Mi a környezetváltás (context switching)? Mi idézhet elő környezetváltást? Milyen műveleteket kell ilyenkor az operációs rendszernek végrehajtania?
6. Mi a magas szintű, illetve a közbenső szintű ütemező feladata?
7. Ismertesse a körbenjáró (round robin) algoritmus alapgondolatát! Mi a fő előnye, illetve hátránya?
8. Mi az előnyük illetve a hátrányuk a prioritásos algoritmusoknak? Hogyan kezelhető a prioritás időosztásos rendszerekben?

7. Memóriakezelés

Az operatív tárral foglalkozó fejezet a memóriakezelés korábban alkalmazott technikáin keresztül jut el a napjainkban szinte egyeduralkodó virtuális tárkezelés problémáihoz. Tárgyaljuk a memóriakezelés optimalizálására alkalmas stratégiákat, valamint a többfolyamatos rendszerekben elkerülhetetlen védelmi kérdéseket. A fejezet végén kitérünk a bonyolult és sok műveletet igénylő algoritmusok hardver által történő gyorsításának lehetőségeire.

Mind a processzor-, mind a memóriakezelés abból a tényből kell kiinduljon, hogy a futásra váró és futó programok azt feltételezik, hogy egyedül ők vannak a világon, egyáltalán nem törődnek azzal, hogy van például operációs rendszer, illetve más folyamatok is futni merészelnek. A folyamatok általában azt is fegyelmen kívül hagyják, hogy mennyi memória van a gépben valójában.

Pedig az erőforrások hatékony kihasználása, és a folyamatok közötti gyors átkapcsolás érdekében több folyamatnak kell egyidejűleg a memóriában tartózkodnia. Az operációs rendszer magjának, a kernelnek, azon belül is a memóriakezelőnek a feladata a memória elosztása a folyamatok között, a mindenkori állapot adminisztrálása és szükség esetén a háttértárak igénybe vétele, mégpedig oly módon, hogy a folyamatok se egymást, se az operációs rendszert ne sérthessék meg.

7.1 Valóságos tárkezelés

A *valóságos tár* kifejezés kis magyarázatra szorul. Nem minden tár valóságos? Természetesen az, de létezik a tárkezelésnek egy olyan - nemsokára ismertetendő - módszere is, melyet kidolgozói virtuális tárkezelésnek neveztek el. A valóságos tár elnevezés a virtuális tártól való megkülönböztetést szolgálja. A kétféle módszer között a leglényegesebb eltérés az, hogy a valós tárkezelés esetében az éppen végrehajtott folyamathoz tartozó programutasításoknak és adatoknak teljes egészében

az operatív memóriában kell tartózkodniuk, míg virtuális tárkezelés esetén csak az *éppen végrehajtás alatt álló rész* van az operatív tárban, a program és az adatok többi része a háttértáron található.

7.1.1 Rögzített címzés

Amíg egy számítógépen csak egy felhasználó dolgozhatott és ő is csak egyetlen, viszonylag kicsi programot futtathatott, a memóriakezelés különösebb gondot nem okozott. Az operációs rendszer állandó területen, például a memória legelső, legkisebb című rekeszein helyezkedett el, a felhasználói program használhatta az operációs rendszer végétől egészen a legnagyobb címig az egész memóriát. A program változóinak, illetve vezérlésátadásainak címe így már fordítás közben meghatározható volt.

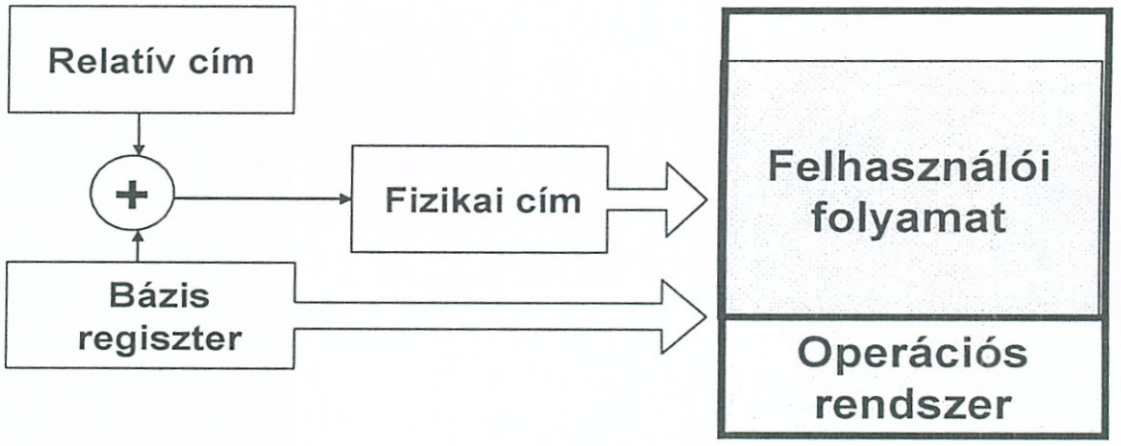
A tárvédelem is elég egyszerű ebben az esetben, mindössze azt kell figyelni, hogy egy adott, az ún. **határregiszter**ben tárolt címnél kisebbet a felhasználói program ne érhessen el, mert az már az operációs rendszer fennhatósága. A felhasználói program az operációs rendszert rendszerhívások által érte el. A rendszerhívás első dolga, hogy a processzort védett üzemmódba kapcsolja, így biztosítva az operációs rendszer számára az egész memória elérését. Természetesen a határregiszter tartalmának változtatása is szigorúan az operációs rendszer feladata volt.

7.1.2 Áthelyezhető címzés

Az első korlát, amibe a rögzített címzésű rendszerek beleütköztek, az volt, hogy az operációs rendszer mérete nem bizonyult állandónak. Ez önmagában még nem lett volna baj, de a programozók is hamar megelégelték, hogy egy-egy operációs rendszer módosítás után mindig módosítaniuk kellett programjaikat. Amikor azonban megjelentek az olyan operációs rendszerek, melyek tranziens résszel is rendelkeztek, azaz egyes részeik csak akkor töltődtek be a memóriába, ha szükség volt rájuk, a szorgalmas programozóknak is bealkonyult, hiszen a felhasználói program rendelkezésére álló memória címtartomány a program végrehajtása során is változhatott.

A megoldás viszonylag egyszerű volt. A programok fordításakor a fordítóprogram már nem közvetlen fizikai címeket generált, (azaz nem azt mondta meg például, hogy ugorj el a 2345H címre), hanem a program elejéhez képesti *relatív címeket* (tehát azt mondta meg például, hogy

ugorj el a program *elejéhez képesti* 1234H címre). Ahhoz, hogy ezekből a relatív címekből fizikai címeket kapjunk, most már csak azt kellett tudni, hogy hol kezdődik a program a memóriában. Ha ezt tudjuk, akkor ehhez a kezdőcímhez hozzá kell adni a programban szereplő *relatív cím*et vagy más néven *eltolást* (displacement) és megkapjuk a keresett memóriarekesz fizikai címét. Ennek a módszernek a támogatására találták ki az ún. **bázisregisztert**. Ez a regiszter tartalmazza a program kezdő- vagy más néven **báziscímét**. A processzor minden memória műveletnél automatikusan hozzáadja a bázisregiszter tartalmát az utasításban szereplő címhez és az így kapott összeg lesz az a fizikai memóriacím, amihez fordul. Már csak egy kérdés maradt hátra: ki határozza meg a program kezdőcímét? Természetesen, ez az operációs rendszer feladata, hiszen az operációs rendszer mindig tudja magáról, hogy éppen meddig ér és hol kezdődik a szabad hely a memóriában, így a felhasználói programot az első szabad címtől kezdve töltheti be, ezt a kezdőcím értéket pedig *a program betöltésekor* beírja a bázisregiszterbe.

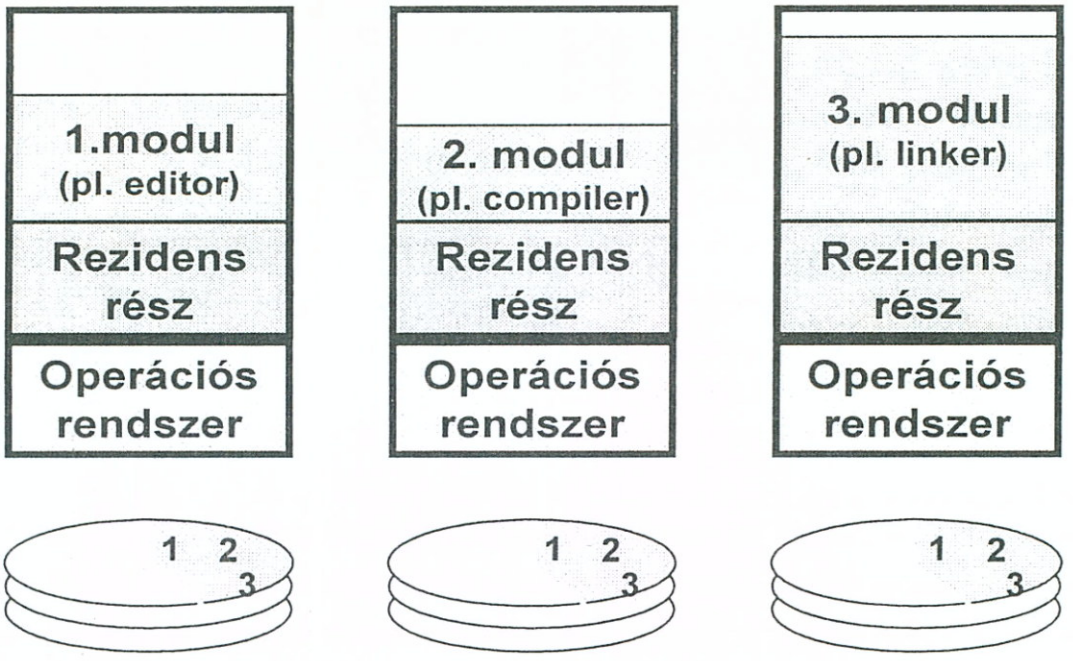


7.1 ábra Áthelyezhető címzés

7.1.3 Átlapoló (overlay) módszer

Az eddigiekben olyan kicsi programokról volt szó, amelyek teljes egészükben belefértek a memóriába. A programozók természetesen nem elégedtek meg ezzel, nem voltak hajlandók fejet hajtani a gyarló földi korlátok előtt, nagyobb programokra vágytak. Ezért azonban meg kellett dolgozniuk. Úgy kellett a feladatokat szervezni, hogy az olyan méretű blokkokból álljon, melyek külön-külön már elhelyezhetők legyenek, azaz beférjenek a rendelkezésre álló memóriaterületre. Ezzel a átlapoló (overlay) technikával elérhető volt, hogy a memóriában állandóan csak a programrészek közötti átkapcsolást végző modulnak kelljen tartózkodnia,

a többiek közül hol az egyik, hol a másik rész került a memóriába, a többi a háttértáron várakozott.



7.2 ábra Átlapoló (overlay) technika

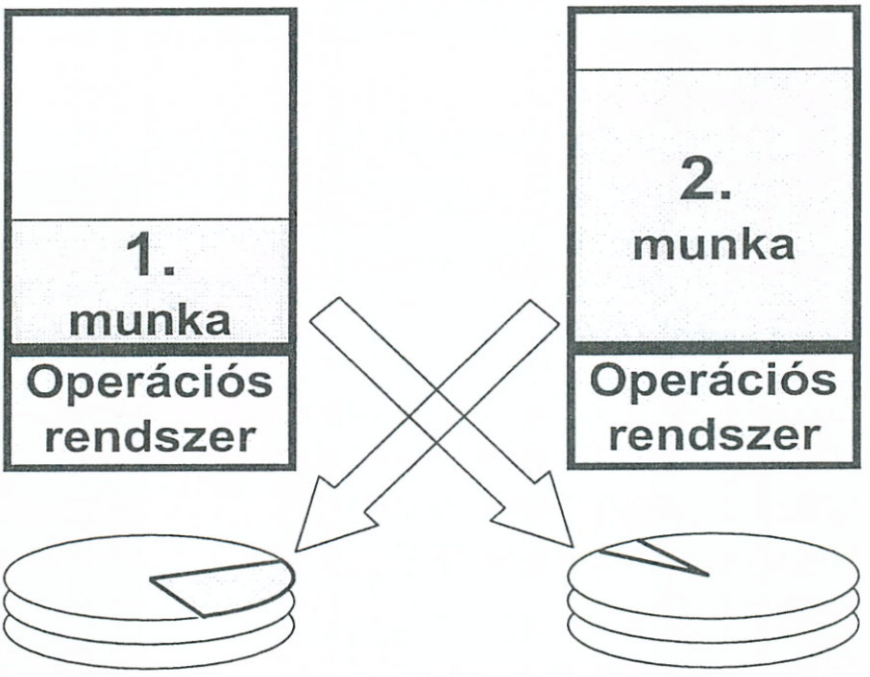
Az ilyen programozáshoz az operációs rendszer semmiféle támogatást nem adott, mindenről a programozónak kellett gondoskodnia, de megérte: a program mérete meghaladhatta a memória méretét.

Még egy nagyon fontos és előremutató dolog történt. A háttértár, amely eddig csak a programok tárolására szolgált, először vesz részt a program futtatásában aktív szereplőként.

7.1.4 Tárcsere (swapping)

A következő kihívást az jelentette, amikor egy időben több felhasználó programjának futtatását kellett biztosítani, persze egymástól függetlenül. A feladat megoldásának legegyszerűbb és legősibb módja, ha minden felhasználó kap egy bizonyos időszeletet, majd ha az lejárt, az általa használt egész memóriaterületet az operációs rendszer a háttértárra másolja, és onnan betölti a következő felhasználóhoz tartozó memóriatartalmat. Így mindegyik felhasználó úgy dolgozhat, mintha az egész memória az övé lenne. A memóriatartomány ki-be másolását **tárcserének** (swapping) hívjuk, a másolás eredményeképpen keletkező állományt **cserefájl**nak (swap file). A módszer nagy hátránya, hogy lassú,

hiszen az időszelet lejártakor a *teljes* memóriaterületet a háttértárra kell másolni, illetve onnan betölteni, ami nagyon sokáig tart!



7.3 ábra Tárcsere

A processzor ütemezéshez hasonlóan itt sem mindegy, hogy mekkora az a bizonyos időszelet, mert ha túl kicsi, akkor a másolgatásra fordítódik az idő legnagyobb része. Segíteni lehet a dolgon úgy, ha az operációs rendszer elég okos, és csak azokat a memóriarészeket mozgatja, amelyek változtak, de ennek menedzselése bonyolult.

A címzés és a tárvédelem szempontjából semmi lényeges változás nem történt, az operációs rendszernek sem az átlapoló, sem a tárcsere technika esetén nem kell többet tudnia, mint az áthelyezhető címzés esetén.

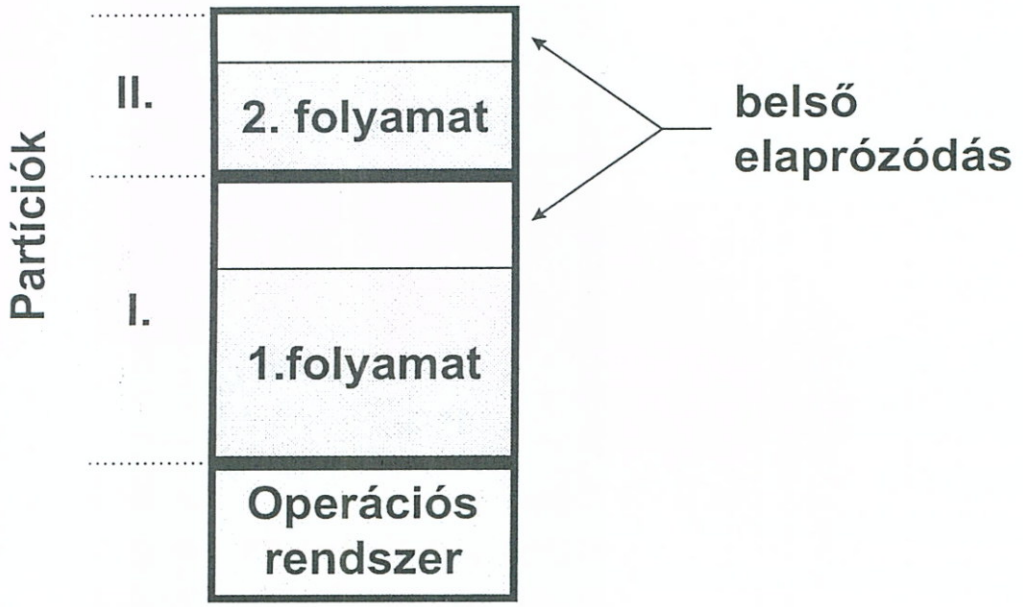
7.1.5 Állandó partíciók

Mi történik azonban, ha az éppen futó folyamat várakozni kényszerül, például felhasználói adatbevitelt vár? Tegyük félre és hadd jöjjön a másik? A másik folyamathoz tartozó memóriaterület betöltése viszont időigényes, ki tudja, hogy megéri-e? Ha állandóan több folyamat tartózkodhatna a memóriában, egyszerű lenne a dolog, a várakozás idejére csak át kéne gyorsan kapcsolni a másikra, és az máris futhatna.

Ha a felhasználói folyamatok számára rendelkezésre álló memóriaterületet egymástól független részekre, partíciókra osztjuk, több

program memóriában tartására is lehetőség nyílik. Egy-egy partíció úgy viselkedik az őt birtokló folyamat számára, mintha teljesen önálló memória lenne, a partíción belül lehetőség van az átlapoló vagy a tárcsere technika alkalmazására is.

Mekkora legyen azonban egy partíció? Ha túl kicsi, esetleg nem fér bele folyamat. Ha túl nagy, akkor - mivel a partíciót csak egy folyamat használhatja -, sok kihasználatlan üres hely marad *a partíció területén belül*. Ezt a jelenséget nevezzük **belső elaprózódásnak** (internal fragmentation). A gyakorlatban az a módszer terjedt el, hogy becslések és statisztikák alapján a memóriában több különböző méretű partíciót alakítottak ki.



7.4 ábra Partícionált rendszer

A partíciókat alkalmazó rendszerekben az operációs rendszerekre már komoly feladat hárul. Ismerniük kell a partíciók méretét és annyi bázisregiszter-határregiszter párt kell számon tartaniuk és folyamatosan vizsgálniuk, ahány partíció van. A másik feladat a partíciók kijelölése a folyamatok számára. Bármilyen tudományosan történt is a partíciók méretének megválasztása, a folyamatok csak azért sem jönnek a megfelelő sorrendben. Az operációs rendszernek döntést kell hoznia, hogy a futásra váró programok közül melyik kapjon meg egy felszabaduló partíciót. Ha az első érkező kapja (FCFS - előbb jött előbb fut), akkor fennáll a veszélye annak, hogy kicsi program nagy partíciót foglal el, ami rossz tárkihasználást eredményez. A másik megvalósított stratégia szerint az operációs rendszer külön várakozási sort tart fenn a

különböző memóriaigényű programoknak. Ez utóbbi esetén viszont például lehetséges, hogy a nagyobb partíció üresen áll, míg a kisebbért vérre menő küzdelem folyik: az eredmény újra csak rossz memória kihasználás.

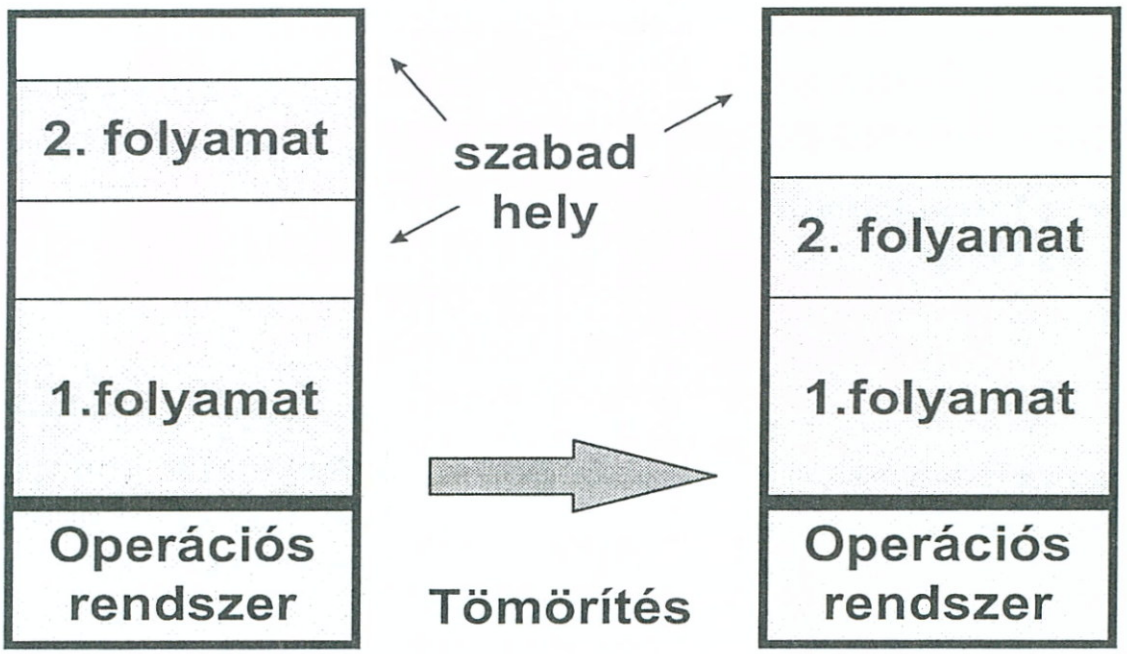
7.1.6 Rugalmas partíciók

Az állandó partíció méretháttányait jórészt kiküszöböli, ha a partíciók számát és nagyságát nem rögzítjük szigorúan, azok rugalmasan alkalmazkodhatnak az aktuális feltételekhez. A teljesen szabadon kezelt méret és kezdőcím azonban túl bonyolult címszámítást igényelne, ezért a partícióméretet célszerű valamely kettő hatvány egész számú többszörösére választani (pl. 2048). Az operációs rendszer ebben az esetben nyilvántartja a partíciók foglaltságát, és az érkező folyamatoknak a befejezett folyamatok után fennmaradó lyukakból választ helyet, persze, csak ha van olyan szabad terület, ahová az befér.

A rugalmas partíciók módszere (majdnem) kiküszöböli a belső elaprózódást (egy kis maradék hely mindig lesz, hiszen a programok mérete nem pontosan egyezik meg a minimális partícióméret – a fenti példa szerint 2048 bájt – egész számú többszörösével. De könnyű belátni, hogy a fenti példában még a legrosszabb esetben is az egy folyamatra eső belső elaprózódás csak maximum 2047 bájt lehet, ami a mai tárméretek mellett elhanyagolható). „Cserében" viszont megjelenik egy új veszély. Ugyanis, ha egy folyamat befejeződik, akkor a helyére betöltendő új folyamat memóriaigénye nem biztos, hogy megegyezik az előzőével. Nyilvánvaló, hogy ha az új folyamat memóriaigénye nagyobb, mint a régié volt, akkor az ide nem tölthető be, viszont ha kisebb, akkor az új folyamat vége után marad egy kis szabad hely, amely viszont általában már túl kicsi, hogy oda más folyamatok beférjenek. Azaz előbb-utóbb a folyamatok által használt memóriaterületek között viszonylag kicsi, de összességében akár jelentős méretű lyukak alakulnak ki. Ezt hívjuk **külső elaprózódásnak** (external fragmentation). Könnyen előfordulhat ugyanis, hogy egy folyamat betöltéséhez összességében lenne elég szabad hely, de az nem folytonosan, hanem az egyes aktív partíciók között teljesen szétdarabolódva áll rendelkezésre. Megoldást jelenthet, ha olykor egy tömörítő algoritmust futtatunk (garbage collection), mely a memóriában lévő folyamatokat folytonosan egymás után rendezi, természetesen úgy, hogy közben módosítja a hozzájuk rendelt bázis- és határregisztereket is. Bár egy jól átgondolt, a várakozó programok jellemzőit is figyelembe

vevő algoritmus itt is csodát tehet, jelentősen lerövidítve a tárrendezéshez szükséges időt, de a tömörítő algoritmus mindig nagyon lelassítja a rendszer működését. Ez különösen interaktív rendszerek esetén nagy probléma: egy türelmetlen felhasználó esetleg a tömörítés alatt elkezdi nyomkodni a reset gombot.

Rugalmas partíciók esetén az „újonnan érkező" folyamat memóriabeli elhelyezésére hasonló stratégiák alkalmazhatók (First, Best, Worst Fit), mint amit a lemezkezelés során a folytonos fájl elhelyezésnél már megismertünk, az ott tárgyaltakhoz hasonló problémákkal.



7.5 ábra Rugalmas partíciók és tömörítés

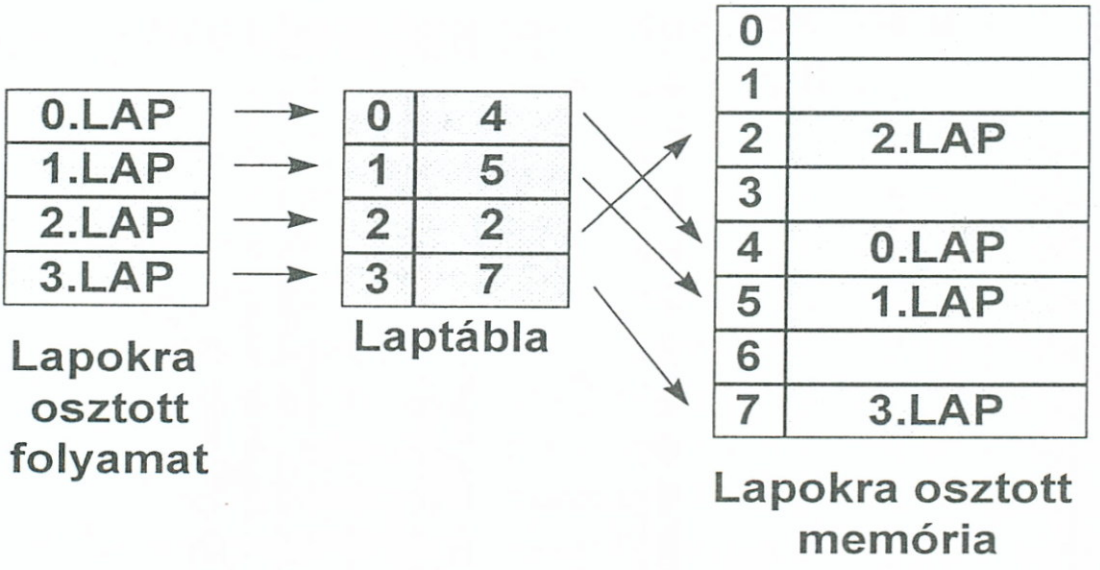
7.1.7 Lapozás (paging)

Mi okozta az elaprózódást az állandó és a rugalmas partíciók használata esetén? Az, hogy a programjaink nem egyforma memóriaterületet igényeltek, viszont ragaszkodtunk ahhoz, hogy **folytonosan** helyezzük el őket a memóriában. Mivel azon nem tudunk változtatni, hogy a folyamataink különböző méretű memóriát igényelnek, próbáljunk meg segíteni úgy, hogy lemondunk arról, hogy a folyamataink folytonosan helyezkedjenek el a memóriában.

Osszuk fel a rendelkezésre álló operatív memória területet **egyforma és viszonylag kisméretű** egységekre, úgynevezett **lapokra**. Egy folyamat memóriában való elhelyezésekor most már nem szükséges az, hogy

akkora *összefüggő* szabad memóriaterület álljon rendelkezésre, amennyit a folyamat igényel, hanem elég az, hogy *összességében* legyen ennyi hely. A folyamat első néhány lapját elhelyezzük az első szabad helyen, a következő néhány lapot a másodikban stb.

Igen ám, de felmerül egy új probléma. Mivel az egyes folyamatok most már nem folytonosan helyezkednek el, nem elég már csak azt megjegyezni, hogy hol kezdődnek és milyen hosszúak, hanem sajnos nyilván kell tartani, hogy az egyes részek hol helyezkednek el. Erre a célra szolgálnak a **laptáblák**. Az operációs rendszer minden egyes folyamat betöltésekor létrehoz a folyamat számára egy laptáblát, mely a logikai lapokhoz hozzárendeli a fizikai lapot. A következő ábrán az operatív memória 8 lap méretű és ebből pillanatnyilag a 0., az 1., a 3. és a 6. lapot használják más folyamatok, míg a 2., a 4., az 5. és a 7. lap üres. Mivel összességében 4 üres lap van, betölthetünk egy olyan folyamatot, amely 4 lapot igényel. A folyamat első (0. sorszámú) lapját töltsük be például az operatív memória 4. számú laphelyére, az 1. sorszámút az 5. üres helyre, a 2. sorszámút a 2.-ra és végül az utolsó, 3. sorszámú lapot a 7. helyre. Ennek a nyilvántartásához létre kell hoznunk egy négysoros laptáblát, mely első (azaz a 0. számú) sora azt mutatja, hogy a 0. sorszámú *logikai* lap a 4. sorszámú *fizikai* lap helyére került az operatív memóriában stb. (Megjegyzés: Azt könnyű belátni, hogy igazából a laptábla első oszlopára nincs is szükség, hiszen a laptábla sorainak *sorszáma* egyértelműen utal a logikai lap sorszámára. Itt az ábrán ezt az első oszlopot csak a megértés elősegítésére tüntettük fel.)

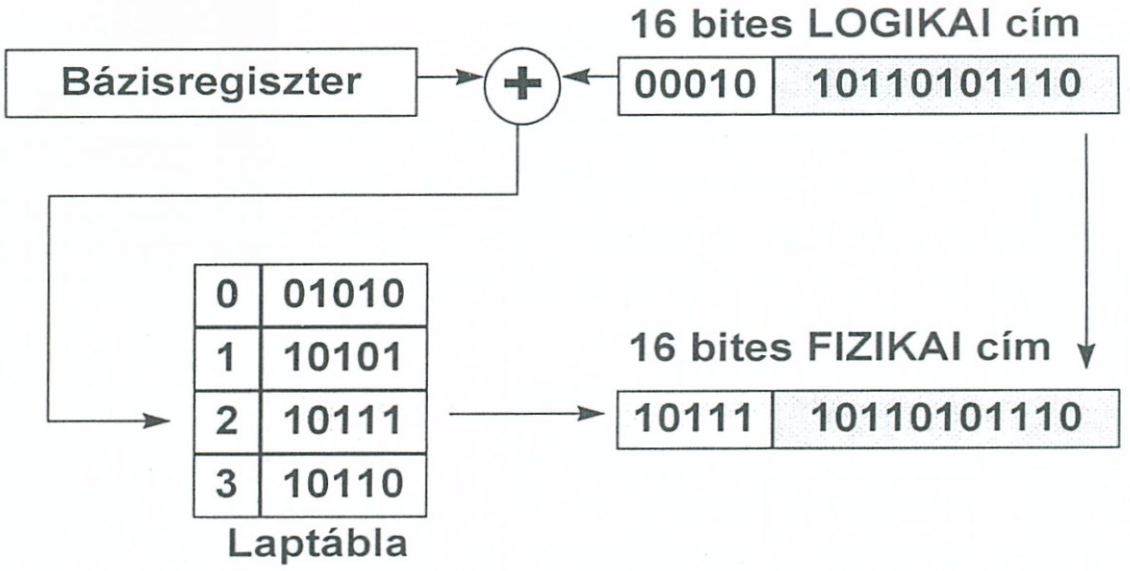


7.6 ábra A laptábla használata

Most már csak egy problémát kell megoldani. Ez pedig az, hogy hogyan találjuk meg a memóriában az egyes utasításokat és adatokat. Hiszen a fordítóprogramunk azt mondja, hogy például ugorjunk el a program elejéhez képesti 16. sorra. Igen ám, de hol van most a 16. sor, hiszen a folyamatunk nem folytonosan helyezkedik el a memóriában? Az egyszerűség kedvéért tételezzük fel, hogy minden lap 10 sorból áll, amelyeket 0 és 9 közötti számokkal látunk el. (Természetesen a valóságban egy lap mérete célszerűen nem a 10 valamelyik hatványával egyezik meg, hanem a 2-évl.) Ezek után könnyű kiszámolni, hogy a 16. logikai sor az 1-es számú logikai lapon helyezkedik el és azon belül ez a 6. számú sor (ezt úgy hívjuk, hogy a lapon belüli **eltolás** értéke 6). Most már csak azt kell tudnunk, hogy hol van az operatív memóriában az 1. számú logikai lap. De hiszen *éppen ezt mondja meg a laptábla 1. sora!* Ha ránézünk a laptáblánkra, látható, hogy az 1. logikai lap az 5. fizikai lap helyén van az operatív memóriában és ezen belül keressük a 6. sort, vagyis a 16. logikai cím a valóságban az 56. fizikai címen található meg.

Nézzük meg általánosan, hogy lapozásnál hogyan történik a **címszámítás**!

A processzor által kiadott **logikai címet** formálisan két részre osztjuk. A cím első része adja meg a **lapszámot**, míg a második része a **lapon belüli eltolást**. Ezek után megnézzük a laptábla „lapszám"-adik sorát, és az ott található számérték megmutatja a lap **fizikai kezdőcímét** az operatív memóriában. Ehhez a számértékhez kell hozzáilleszteni (úgy mondjuk, hogy „hozzáadni", holott ez nem összeadást, hanem hozzáillesztést jelent!) a lapon belüli eltolás értékét.



7.7 ábra Fizikai cím számítása

Nézzünk egy valóságos példát! Tegyük fel, hogy a logikai cím 16 bites, azaz 2^{16} = 65356 sort címezhetünk meg. Legyen egy lap mérete 2^{11} = 2048 sor. Ebből kiszámítható, hogy 2^{16} / 2^{11} = 2^5 = 32 db lapunk van. Azaz a lapok kiválasztására 5, míg a lapon belüli sor kiválasztására (eltolás) 11 bit kell. Például így:

logikai cím: 00010|10110101110 → lapcím: 00010, eltolás: 10110101110.

Tegyük fel, hogy a laptábla 00010 (=2) sorában az 10111 (=23) számérték található, azaz a keresett memóriahely fizikai címe: 10111|10110101110.

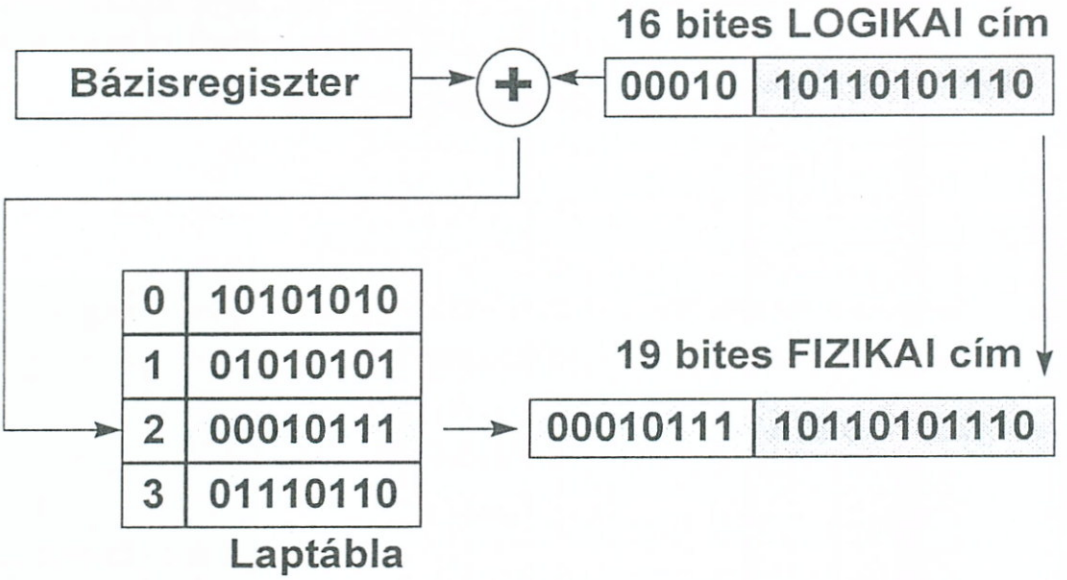
Lapozásnál a címszámítás látszólag egyszerű: a folyamat által kívánt logikai címnek azt a részét, amely a laptábla megfelelő rekeszére mutat, ki kell cserélni a laptábla megfelelő rekeszének tartalmára és már készen is áll a fizikai cím. De hol legyen a laptábla, és hány rekeszből álljon? Adott lapméret esetén tudjuk ugyan a lapok, azaz a szükséges laptábla rekeszek számát, de a memória is (egyszerű bővítéssel), a lapméret is változhat (az operációs rendszer más beállításával), így tisztán hardver eszközökkel nehéz megoldani a címszámítást. A gyakorlatban az terjedt el, hogy egy-egy folyamat létrehozásakor az operációs rendszer megfelelően védett területen létrehozott a folyamat számára egy laptáblát a memóriában, és a laptábla címét a folyamat egyéb adataival együtt a folyamatleíró blokkban tárolta. A címszámító hardver megkapja az éppen futó folyamat laptáblájának címét, ahhoz hozzáadja a logikai lapcímet, és

az így keletkezett mutató által címzett rekeszből veszi ki a lap fizikai kezdőcímét. A dolog működik, de a folyamat minden egyes memóriához fordulása egy újabb memóriához fordulást igényel, tehát a **memória elérés ideje** – még ha a vezérlés idejétől el is tekintünk - **megduplázódik.** (Ezt az elrémítő hatást címszámítást segítő asszociatív memória alkalmazásával kb. 10%-ra lehet csökkenteni. Lásd később, a virtuális tárkezelésnél!)

Láthatjuk, hogy a lapozás alkalmazásával a külső elaprózódást kiküszöböljük, a belső elaprózódás minimalizálását pedig az szolgálja, hogy a lapok viszonylag kicsik. (A folyamatok mérete általában nem egészszámú többszöröse a lapméretnek, így folyamatonként átlagosan egy fél lapnyi terület üres marad.) A belső elaprózódás szempontjából az lenne az optimális, ha a lapméret 1 sor lenne, de akkor a laptábla mérete lenne nagyon nagy. E két szempont közötti kompromisszum eredménye az, hogy a gyakorlatban a lapméret általában 1-4 kB.

A lapozó technika amellett, hogy kiküszöböli az elaprózódást, mintegy melléktermékként egy további lehetőséggel kecsegtet, amelyet főképp régebbi, kis címzési kapacitású processzorok esetén lehet jól használni:

> A processzor látszólagos címzési tartománya könnyedén megnövelhető, hiszen nem kell mást tenni, mint a laptábla szóhosszúságát megnövelni. Az egyszerre kiadható címek száma ugyan nem növekszik, de azok egy szélesebb tartományban helyezkedhetnek el. Például az előző, 16 bites címet alkalmazó példánkkal élve, eredetileg a megcímezhető memória mérete 2^{16} byte, azaz 64 kB lehetett. Ha minden 5 bites logikai lapcímhez 8 bites fizikai lapcímet rendelünk, (azaz a laptáblában nem ötbites, hanem nyolcbites számokat tárolunk), akkor 8-szor annyi, azaz 512 kB memória címezhető. Egy folyamat címtartománya ugyan nem haladhatja meg a 64 kB-ot, viszont egyszerre a memóriában lehet 8 ilyen folyamat!



7.8 ábra Memóriabővítés laptábla segítségével

7.2 Virtuális tárkezelés

Az előzőekben ismertetett valóságos memóriakezelésnek három fontosabb problémája van.

Az egyik az, hogy ma már az egyszerűbb mikroprocesszorok **címzési kapacitása is olyan nagy, amekkorát nem akarunk kiépíteni** (drága, nagy méretű, még komoly rendszerekben is kis kihasználtságú lenne). Például egy 32 bites címzésű mikroprocesszor címtartománya 4 GB, míg a ténylegesen kiépített fizikai tár, azaz a félvezető elemekből kialakított memória mérete számítógép típustól és felhasználási körtől függően manapság 10..100 MB nagyságrendű. A programozó / fordítóprogram azt hiszi, hogy 4 GB memóriát használhat és ennek megfelelő címeket generál, de ténylegesen ennél sokkal kisebb memória áll rendelkezésre, sokkal kisebb fizikai címtartománnyal. A különbség láthatóan a 32 bites gépek esetén is óriási, és akkor még nem is beszéltünk a 64 bites gépekről! Valami olyan címtranszformációs mechanizmust kell kitalálni, amely feloldja ezt az ellentmondást úgy, hogy a programozónak ne kelljen beavatkoznia. (Azaz szakkifejezéssel élve: *átlátszó legyen a felhasználó számára.*)

A másik probléma az, amit az erőforrás kezelésnél már láttunk. Egy multiprogramozott rendszer hatékonysága – egy bizonyos határig – nő, ha minél több folyamat fut párhuzamosan. Igen ám, de ahhoz, hogy sok

folyamat futhasson, sok folyamatot kell betölteni a memóriába, azaz nagyon nagy memóriára van szükség, amit - láttuk - nem akarunk kiépíteni. Szerencsére a folyamatokra igaz az úgynevezett **lokalitási elv**, amely azt mondja ki, hogy **ha egy program adott pontján vagyunk, akkor nagy valószínűséggel, viszonylag hosszú ideig ennek a pontnak egy nem túl tág környezetében fogunk maradni**, azaz kissé pongyolán fogalmazva, a programok végrehajtásuk során nem ugrálnak a memóriában össze-vissza. Ebből viszont az következik, hogy igazából **nincs is arra szükség, hogy a teljes programkódot mindig a memóriában tartsuk**, elegendő a program egy viszonylag kis részét az operatív memóriába tölteni. Persze ez az operatív memóriában lévő rész a végrehajtás során folyamatosan változik, de *egyszerre* sosem kell az egész. Világos, hogy ha nem kell a folyamatok teljes kódját az operatív memóriában tartani, akkor ugyanakkora memóriában több folyamat (folyamatrész) fér el, azaz több folyamat futhat párhuzamosan.

Ez viszont automatikusan megoldja a valóságos tárkezelés harmadik problémáját is, azaz azt, hogy a valóságos tárkezelés esetén egy folyamat betöltése, valamilyen okból történő felfüggesztése esetén háttértárra mentése, majd újra betöltése rendkívül hosszú időt vesz igénybe, hiszen az *egész* programot be/ki kell másolni a processzorhoz képest nagyon lassú háttértárról/háttértárra. Az előző pontból látható, hogy erre sincs szükség, hiszen az induláshoz elég, ha csak az induló utasítás egy viszonylag kis környezetét töltjük be. Hasonló igaz a felfüggesztésre, nem az egész kódot, hanem csak a bent lévő viszonylag kis részt kell a háttértárra menteni. Tehát a **betöltési, felfüggesztési műveletek lényegesen felgyorsulnak**.

A fentiekből látható, hogy az ötlet lényege az, hogy a folyamatok kódjának mindig csak egy része legyen bent az operatív memóriában. Azonban ez - ahogy az már lenni szokott... - újabb problémákat vet fel. Egyrészt nyilván kell tartani, hogy éppen mi van bent az operatív memóriában és hol, másrészt pedig, mivel a folyamatok kódjai csak részben vannak az operatív memóriában, bizony időről-időre előfordul olyan szituáció, hogy egy folyamat kódjának egy olyan részét akarja használni, amely *pillanatnyilag* nincs bent az operatív memóriában, azaz mielőtt a folyamat tovább futhatna, be kell hozni egy újabb részét. (Ezt a szituációt hívjuk *laphibának*.) Természetesen ez viszont azt okozhatja, hogy bizonyos, már bent lévő részek tovább nem kellenek, azokat viszont

ki kell menteni a háttértárra, és ennek helyébe lehet behozni az új részt. Ez pedig időt és komoly vezérlést igényel.

A kérdés az, hogy ilyen esetekben mekkora részt hozzunk be. Hogy elkerüljük a különböző méretű részekből és a folytonos elhelyezésből adódó problémákat, célszerű a lapozási technikánál megismert technológiát felhasználni, persze némi módosítással. Azaz a megoldás tulajdonképpen a lapozási technika olyan módosítása, hogy a lapoknak *egyszerre csak egy részhalmaza van bent az operatív memóriában*, nem pedig az összes lap.

Ez az ún. **virtuális tárkezelés**.

7.2.1 A virtuális tárkezelés alapjai

Először is tisztázzuk, hogy miért hívják a módszert *virtuális* tárkezelésnek! A bevezetőben abból indultunk ki, hogy - bár csak egy viszonylag kisméretű operatív tárunk van - a programozó/fordítóprogram úgy látja, mintha rendelkezésére állna a teljes címzési kapacitásnak megfelelő, folytonosan címezhető memória. Ez az ún. látszólagos vagy más néven **virtuális memória**. Azért virtuális, mert ez ilyen formában nem létezik, ez a terület a valóságban egy háttértáron (például diszken) található és - a korábban megbeszélteknek megfelelően - a diszkvezérlő a folytonos, lineáris címeket átalakítja olyan formájúvá, ahogy azt a diszkek igénylik (lemezoldal, sáv, szektor). De a programozó erről semmit sem tud, az egészet az operációs rendszer intézi.

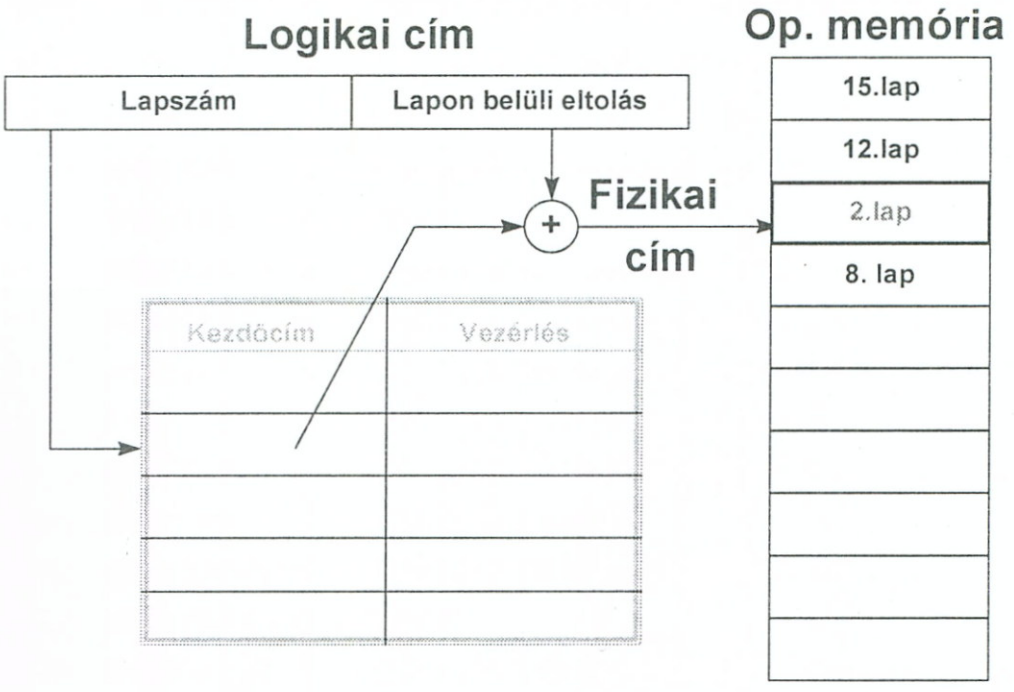
!

Ezek után rátérhetünk a megvalósítás részleteire!

Osszuk fel az operatív memóriát és a virtuális memóriát **is** egyforma méretű, viszonylag kis egységekre, **lapokra**. Az operatív memóriában hozzunk létre minden folyamathoz egy laptáblát, amely minden sorában a lap operatív tárbeli kezdőcíme mellett tartalmazzon még egy olyan vezérlés jellegű bitet is, amely azt mutatja, hogy az adott lap *bent van-e* (present) az operatív tárban vagy nincs.

!

Hogyan történik a címszámítás?



7.9 ábra Lapszervezésű virtuális tár

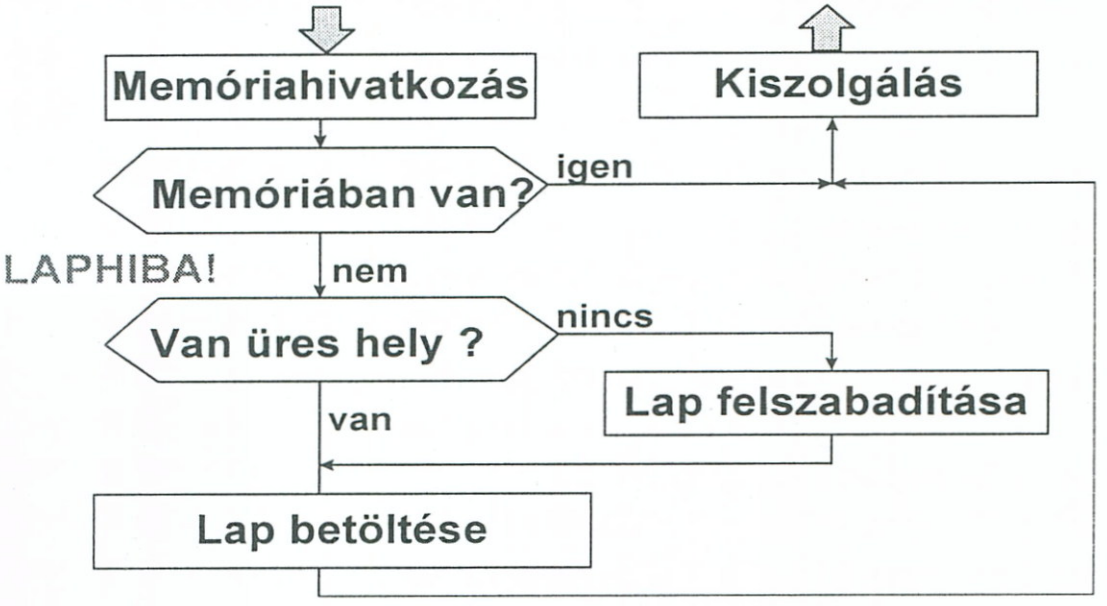
A processzor által kiadott logikai címet most is egy lapszám és egy eltolás részre osztjuk. A lapszám segítségével megnézzük a laptábla megfelelő sorában a present bitet. Ha a present bit azt mutatja, hogy az adott lap az operatív memóriában van, akkor hasonlóan járunk el, mint eddig: a laptábla „lapszám" sorában megtaláljuk a lap operatív tárbeli kezdőcímét és ehhez hozzáadva az eltolás értékét, megkapjuk a keresett fizikai címet.

Ha azonban a present bit azt mutatja, hogy hivatkozott lap még nincs betöltve, azt be kell tölteni a háttértárról. Ilyenkor beszélünk **laphibáról** (page fault), mely nem valamilyen hardver hiba, hanem csak azt jelenti, hogy a hivatkozott lap jelenleg nincs bent az operatív memóriában. A lapok betöltése tehát a folyamatok igényei szerint történik, ezért ezt az eljárást **igény szerinti lapozásnak** (demand paging) nevezzük. (Megjegyzés: egy másik módszer az úgynevezett **előretekintő lapozás**. Ez azt jelenti, hogy egy laphibánál nem csak a kért lapot hozzuk be, hanem a környezetében lévő néhány lapot is. Az oka ennek az, hogy - mint azt már korábban láttuk - a diszkműveletek során a *megtalálási idő* sokkal nagyobb, mint az *átviteli idő*, azaz ha nagy nehezen megtaláltuk a keresett lapot, akkor már majdnem mindegy, hogy egy vagy néhány lapot hozunk-e be, és a lokalitási elv értelmében pedig elég valószínű, hogy nem csak a hivatkozott lapunk fog kelleni a későbbiekben, hanem pl. a

szomszédos is. Ezzel a módszerrel a laphibák száma csökkenthető, bár néha feleslegesen is behozunk lapokat.)

Térjünk vissza a laphibához! Tehát a present bit azt mutatta, hogy a keresett lap nincs bent az operatív memóriában, azt be kell hozni a háttértárról. Igen ám, de hová? A számítógép indulása utáni pillanatokat leszámítva az operatív tárban nincs szabad hely, hiszen már megelőzően a folyamatok igényei alapján teletöltöttük a szükséges lapokkal. Azaz ahhoz, hogy egy új lapot behozzunk, először meg kell határozzuk, hogy melyik bent lévő lap *helyére* hozzuk be az újat. Ennek eldöntésére többféle ún. *lapcsere* algoritmus létezik, amelyeket a későbbiekben részletesen tárgyalni fogunk. Tegyük fel, hogy megvan az „áldozat", de mielőtt behoznánk helyére az új lapot, ki kell mentenünk a háttértárra. Miután a „régi" lapot elmentettük, behozhatjuk az újat. Természetesen a laptáblában a „régi" lap present bitjét „*nincs bent*", míg az „új" lapét „*bent van*" értékűre kell állítani és az „új" lap esetén a kezdőcím mezőt is ki kell töltenünk.

Két dolgot érdemes megfigyelnünk. Egyrészt, ha a folyamat nem hivatkozik egy lapjára, észre sem veszi, hogy az nincs az operatív memóriában. (Mai programok esetén a hibakezelő eljárások a program méretének a felét is kitehetik. De ha a futás során nem lép fel hiba, ezek a lapok egyáltalán nem kerülnek be az operatív memóriába!) A másik fontos dolog az, hogy a lapok betöltéséhez lemezműveletekre van szükség, ahol legalább négy nagyságrenddel (10000-szer!) lassabb eléréssel kell számolnunk. Ha figyelembe vesszük azt, hogy egy új lap behozatala előtt a régi lapot el kell menteni, akkor láthatjuk, hogy egy laphiba esetén két háttértár műveletre van szükség. Ha tehát minden 1001-dik memória hivatkozás eredményez laphibát, az átlagos elérési idő $(1000 * 1 + 1 * 2*10000)/1001 \approx 21$, tehát a folyamatok futási sebességét alapvetően meghatározó memóriaműveletek sebessége kevesebb, mint huszadára csökken! Ez nem nagyon kellemes, mindent el kell követni a sebesség növelése érdekében.



7.10 ábra A virtuális tárkezelés algoritmusa

Nézzük meg, hogy milyen lehetőségek vannak a címszámítás gyorsítására!

Először vizsgáljuk meg, hogy tényleg szükséges-e a kicserélendő lapot minden esetben kimenteni a háttértárra, mielőtt felülírnánk! Kis gondolkozással rájöhetünk, hogy nem. Hiszen, ha addig, amíg a lap az operatív memóriában volt, *nem írtunk rá*, akkor felesleges kimenteni, mert a háttértáron érvényes másolata található (onnan töltöttük be és azóta nem módosítottuk). Tehát, ha tudnánk, hogy egy lapra írtunk-e vagy sem, akkor csökkenteni lehet a kimásolandó lapok számát, vagyis növelni a memória hozzáférés átlagsebességét. Ehhez nem kell mást tenni, mint a laptábla minden sorában kialakítani egy-egy újabb jelzőbitet, amelyet, ha az adott lapra *írunk*, mondjuk 1-esbe állítunk. Ezt a bitet nagyon gyakran „piszkos" (dirty) bitnek hívják, mert azt jelzi, hogy a lap „piszkos"-e, azaz írtunk-e rá vagy sem, míg bent volt az operatív memóriában. Ennek megfelelően, célszerű a kicserélendő lap meghatározásakor figyelembe venni, hogy lehetőleg minél többször „nem piszkos" lapot írjunk felül.

Könnyű belátni, hogy nem mindegy, hogy egy folyamat hány lapot használhat a memóriában, azaz másképpen fogalmazva, hány **kerettel** rendelkezik. Hiszen várhatólag minél több lapot használhat, annál kevesebb olyan eset van, hogy új lapot kell betöltenie. (Megjegyzés: ez nincs mindig így, előfordulhatnak olyan speciális esetek, hogy egy folyamat több laphibát generál akkor, ha több kerete van, mint ha

kevesebb. Ez az ún. Béládi-féle anomália, de ez viszonylag ritka jelenség.)

Ezek után felmerül a kérdés, hogy hogyan határozhatjuk meg, hogy egy folyamat hány lappal gazdálkodhasson? Ennek eldöntésére többféle, ún. **lapkiosztási elv** létezik.

7.2.2 Lapkiosztási elvek

Legkedvezőbb a helyzet akkor, ha a folyamat számára szükséges minden lap az operatív tárba kerülhet. Ezt azonban általában szándékosan kerüljük, még akkor is, ha fizikailag lehetséges lenne. Inkább több folyamat részleteit szeretnénk a memóriában látni, mint egyet teljes egészében. Nem *egy* folyamat futását kell optimalizálni, hanem az összesét együtt!

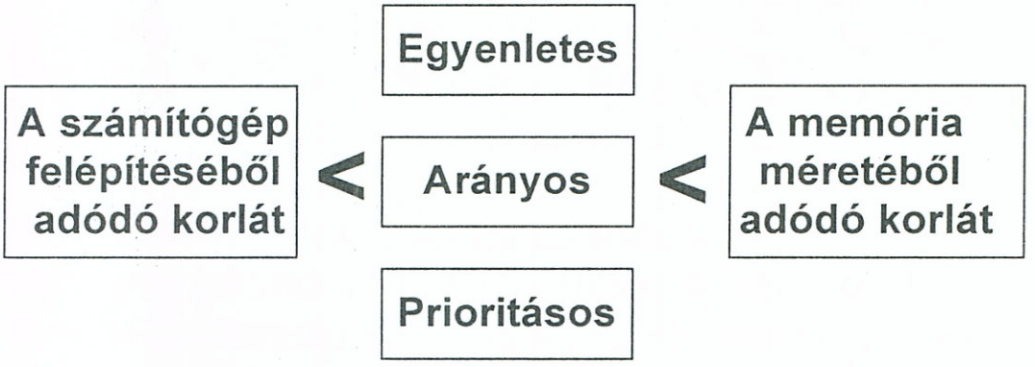
A folyamatok számára biztosítható lapok számának felső korlátja a fizikai tár mérete. Létezik azonban egy alsó korlát is, amelyet az adott processzor utasításkészletének ismeretében határozhatunk meg. Vannak ugyanis olyan utasítások, amelyek végrehajtása során több memóriacímet is használunk. Legrosszabb esetben ezek mindegyike különböző lapon található meg, azaz ha a folyamatnak nincs legalább ennyi lapja, akkor az ilyen típusú utasításokat nem tudja végrehajtani. (A gyakorlatban általában az ún. indirekt címzéses utasítások a legkritikusabbak. Ilyen esetekben maga az utasítás egy adott címen található - 1. lap; az utasítás nem az operandus címét tartalmazza közvetlenül, hanem annak a helynek a címét, ahol az operandus címe található (a „cím címe" - hasonlóan a pointerhez) - 2. lap; és végül maga az operandus a 3. lapon lehet. Ha az utasítások mérete nagyobb is lehet, mint egy memóriarekesz hossza - azaz például bájtos memória esetén több mint egy bájt -, akkor ehhez még egy lapot hozzá kell adni, hiszen elképzelhető, hogy az utasítás első bájtja egy lap utolsó helyén található, míg a második bájt a következő lap elején. Hasonló a helyzet, ha a címek és/vagy az operandusok is több bájtosak lehetnek. Azaz ilyenkor a minimális lapszám 6.) Az alsó és a felső korlát által meghatározott két szélső érték között az operációs rendszer dönt valamilyen elv alapján.

Legegyszerűbb esetben a rendelkezésre álló lapokat **egyenletesen** osztjuk el a folyamatok között, azaz minden folyamat ugyanannyi kerettel gazdálkodhat. Ez a legegyszerűbb, de egyben a legkevésbé kifinomult megoldás is, hiszen lehetnek kis folyamatok, amelyek dúskálnak a

lapokban, és nagyok, amelyek állandóan laphibákkal kénytelenek küszködni.

Jobb, igazságosabb elosztást tesz lehetővé, ha a folyamatok virtuálismemória-igényeinek figyelembe vételével döntünk. Azaz egy folyamat minél nagyobb virtuálismemória-területet használ, annál több keretet kap. Ez esetben **arányos** lapkiosztásról beszélünk.

Ha a folyamatok között vannak eltérő prioritásúak, azaz különböző fontosságúak, akkor a magasabb rendűek joggal követelhetnek extra előnyöket, tehát a hasonló igényű, de alacsonyabb prioritású folyamatoknál több lapot. Ez a **prioritásos** elosztás, mely az azonos fontosságú folyamatokat természetesen arányosan, vagy egyenletes elosztással kezeli.



7.11 ábra Keretek számának meghatározása

! Ha a rendelkezésre álló lapok száma egy folyamat számára a futás során állandó, **lokális elosztásról** beszélünk. Az operációs rendszer azonban rendelkezhet úgy is, hogy nem oszt ki minden lapot, csak a minimálisan szükségeseket, a fennmaradó szabad lapokból a folyamatok dinamikus igényeit elégíti ki. Ez a módszer a **globális** lapkiosztási stratégia. Ez utóbbi előnye, hogy rugalmasabb, hátránya, hogy az „ügyesebb", szerencsésebb folyamatok a gyengébbek rovására garázdálkodhatnak, azaz olyan sok lapot begyűjthetnek, hogy a többieknek nem jut a minimálisnál több.

! Ha egy folyamat bármely okból olyan kevés laphoz jut, hogy csaknem mindig laphibát okoz, fut ugyan, de, mint láttuk, futása akár tízezerszer is lassabb lehet, mint normális esetben. Ezt a jelenséget nevezik **vergődésnek** (trashing).

A vergődés megelőzésére alkalmazott módszerek közül a legelterjedtebb a laphibák gyakoriságának figyelésén alapul. Tapasztalatok, szimulációk vagy számítások alapján meghatározható egy kívánatosnak tartott laphiba gyakorisági tartomány. Ha egy folyamat egy adott minimális értéknél kevesebb laphibát generál időegység alatt, akkor ez azt jelenti, hogy túl sok lapja van, tehát az operációs rendszer elvesz tőle egyet; viszont, ha egy adott maximális számnál több laphibát okoz, akkor kevés lapja van, tehát az operációs rendszer ad neki még egyet.

A vergődés veszélye lokális lapkiosztási stratégia mellett lényegesen kisebb (hiszen ilyenkor lehetséges, hogy ugyan egy folyamat vergődik, de mivel a lapkeretek száma állandó, a többi folyamatot „békén hagyja", míg globális kiosztás esetén a vergődő folyamat elvesz lapokat a többiektől, így a többi is vergődni kezd.)

7.2.3 Lapcsere stratégiák

Egy következő pont, ahol az operációs rendszer be tud avatkozni a laphibák számának csökkentésébe és ezen keresztül a memóriához való átlagos hozzáférési időnek a csökkentésébe az az új lapok helyének, azaz a lecserélendő lapoknak a kiválasztása.

A lapcsere algoritmusok minősítésére mesterséges vagy tapasztalati úton laphivatkozási sorozatokat (ún. **referencia stringeket**) alkalmaznak. A szimuláció eredménye a különböző módszerek hatékonyságára jellemző laphiba szám.

7.2.3.1 Az optimális stratégia (Optimal - OPT)

Az optimális stratégia szerint azt a lapot kell kicserélni, amelyikre a legtávolabbi jövőben lesz szükség, hiszen így a bent maradó lapokkal a lehető legtovább ki tudjuk elégíteni a lapok iránti igényeket laphiba nélkül.

§

OPT - Optimális stratégia
Azt a lapot kell lecserélni, amelyre a legkésőbb lesz szükség

Az alábbi példa egy 9 lapos (0..8) folyamatról szól, melynek az operációs rendszer három lapnyi helyet, azaz három *keretet* biztosított. A folyamat a

6 8 3 8 6 0 3 6 3 5 3 6 sorrendben hivatkozik lapjaira, és feltételezzük, hogy induláskor egy lap sincs bent. (A többi módszer értékelésénél is ugyanezt a példát használjuk majd.)

A laphibák előfordulási helyét az ábrán *-gal jelöltük. Amelyik oszlopban nincsenek számok, azok azt jelzik, hogy az aktuális lap igénylésekor nem volt laphiba, nem történt semmi változás, az előző oszlop adatai vannak érvényben.

Igényelt lap	6	8	3	8	6	0	3	6	3	5	3	6
1.lap	6	6	6			6				6		
2.lap		8	8			0				5		
3.lap			3			3				3		
	*	*	*			*				*		

Laphibák száma: 3 + 2

7.12 ábra Laphibák száma az OPT stratégia esetén

A laphibák két csoportra oszthatók. Az első három elkerülhetetlen, hiszen valamikor fel kell tölteni az üres kereteket (célszerűen, amíg van üres keret, addig az igényelt lapot az első szabad helyre hozzuk be – ez a rész tehát független az alkalmazott lap*csere* stratégiától), a többi laphiba viszont a lapcsere algoritmus jellemzője. Látható, hogy maga az algoritmus csak további két laphibát eredményezett.

Az algoritmusnak azonban van egy nagy hibája, ez pedig az, hogy egy lapcserénél *előre* kellene tudni, hogy a későbbiekben milyen sorrendben fogunk a lapokra hivatkozni. A döntés a jövőbelátáson alapul, tehát a gyakorlatban megvalósíthatatlan. Ismertetése csupán annyiból célszerű, hogy - mivel bizonyíthatóan ez okozza a legkevesebb laphibát - viszonyítási alapul szolgálhat a többi módszer értékelésénél.

7.2.3.2 Előbb jött - előbb megy (First In First Out - FIFO)

Mivel láttuk, hogy az optimális stratégia - sajnos - megvalósíthatatlan, próbáljunk meg kitalálni olyan algoritmust, amely megközelítőleg olyan

hatékony, mint az optimális, de megvalósítható. Forduljunk megint a már jól ismert lokalitási elvhez. A lokalitási elv azt mondta ki, hogy ha a programunk egy adott pontján tartózkodunk, akkor valószínűleg viszonylag sokáig ennek egy szűk környezetében maradunk. „Lefordítva" ezt a lapozásra: valószínű, hogy a továbbiakban a mostanában behozott lapok kellenek, míg a régebben behozottak nem. Meg is van az alapötlet: tartsuk nyilván, hogy milyen sorrendben hoztuk be a lapokat, és lapcsere esetén a *legrégebben behozottat cseréljük le*.

FIFO - Előbb jött, előbb megy
Azt a lapot kell lecserélni, amely legrégebben van bent a memóriában

§

Ráadásul ez viszonylag egyszerűen megvalósítható, hiszen nem kell hozzá más, mint egy egyszerű várakozási sor, amely olyan hosszú, ahány kerete az adott folyamatnak van. Ezt az ún. **FIFO** (**F**irst **I**n **F**irst **O**ut - amelyik először ment be, az jön először ki) sort használjuk az adminisztrációhoz. Minden frissen behozott lap sorszáma bekerül a FIFO sor végére, ezáltal a sorban már bent lévő lapsorszámok eggyel előre lépnek és a sor másik végén „kipotyog" a legrégebbi lap sorszáma.

Igényelt lap	6	8	3	8	6	0	3	6	3	5	3	6
1.lap	6	6	6			0		0		0	3	
2.lap		8	8			8		6		6	6	
3.lap			3			3		3		5	5	
	*	*	*			*		*		*	*	

Laphibák száma: 3 + 4

7.13 ábra Laphibák a FIFO stratégia esetén

Látható, hogy a FIFO módszer a „kötelező" 3 laphibán felül további 4-et generált, azaz jelen példánkban kétszer annyit, mint az optimális. (Valós esetekben, ahol a keretek száma általában lényegesen több mint 3, azért

nem ilyen rossz az arány, de olyankor is igaz, hogy a FIFO hatékonysága jóval kisebb az OPT hatékonyságánál.)

Tehát a FIFO algoritmus, bár egyszerűen implementálható, de hatékonyságában messze elmarad az optimumtól.

Mi lehet a baj? A lokalitási elv segítségével történő okoskodásunkban ott követtük el a hibát, hogy azt mondtuk, hogy a legrégebben *behozott* lapot cseréljük le. Mert lehet, hogy bár egy lapot régen hoztunk be, de még mindig használunk. Tehát nem a *behozatal*, hanem a legutolsó *használat* idejét kellene számon tartanunk, hiszen az következik inkább a lokalitási elvből, hogy mivel valószínűleg a most használt lap környezetében maradunk egy darabig, a már *régóta nem használt* lapokra valószínűleg nem lesz többé szükség. Ez lesz az LRU módszer, de mielőtt rátérnénk ismertetésére, nézzük meg, hogy tipikusan melyek azok a lapok, amelyeket régen hoztunk be és még mindig használunk. Ezek a lapok általában a program legfontosabb, gyakran használt részeit, alapvető adatait tartalmazzák és főként ilyenek a rendszerprogramok, tehát például az operációs rendszer lapjai. Vagyis a FIFO elv - különösen globális lapkiosztási stratégiával párosítva - az operációs rendszer legfontosabb lapjait állandóan „kilapozza", és ez sokszor nagyobb probléma, mint a kis hatékonysága.

7.2.3.3 Legrégebben használt (Last Recently Used - LRU)

Ezek után térjünk rá a már említett stratégia vizsgálatára. Vagyis azt tartsuk nyilván, hogy egy lapot mikor használtak, és azt a lapot cseréljük ki, amelyet a bent lévők közül a *legrégebben használtunk utoljára*. (**L**ast **R**ecently **U**sed - **LRU** algoritmus.)

LRU - Legrégebben használt
Azt a lapot kell lecserélni, amelyre legrégebben hivatkozott a folyamat

Az eddigi példa az LRU esetére:

Igényelt lap	6	8	3	8	6	0	3	6	3	5	3	6
1.lap	6	6	6			6	6			6		
2.lap		8	8			8	3			3		
3.lap			3			0	0			5		
	*	*	*			*	*			*		

Laphibák száma: 3 + 3

7.14 ábra Laphibák LRU stratégia esetén

A módszer kedvezőnek tűnik, de honnan tudjuk, hogy melyik lapot mikor használta a folyamat? A kérdés csak hardver támogatás segítségével válaszolható meg hatékonyan, ugyanis csak hardver megoldás lehet képes arra, hogy elviselhető időveszteséggel minden egyes memóriahivatkozásnál módosítson egy, a lapok felhasználási sorrendjét tartalmazó, FIFO jellegű listát, vagy a laptábla erre szolgáló mezőjébe bejegyezze a hivatkozás időpontját és laphibánál ezek közül megkeresse a legkisebb értéket.

A módszer tehát viszonylag kevés laphibát eredményez, de cserébe az adminisztrációs terhek szinte elviselhetetlenül növekedtek.

7.2.3.4 Egyéb lapozási stratégiák

Az eddigi lapozási stratégiákkal az volt a baj, hogy vagy jó hatásfokúak, de elvileg (OPT) vagy bonyolultságuk miatt gyakorlatilag (LRU) megvalósíthatatlanok, vagy ugyan könnyen megvalósíthatók, de nagyon sok laphibát okozók (FIFO) voltak. Próbáljunk meg kitalálni olyan stratégiákat, amelyek elfogadható hatékonyság mellett viszonylag egyszerűen meg is valósíthatók!

Mindkét ismertetendő eljárás a laptábla minimális bővítésével jár, a laptábla adatai is tevékeny részt vállalnak a lecserélendő lap kiválasztásában.

7.2.3.5 Második esély (Second Chance - SC)

Az eljárás a FIFO elven alapul, egy kis kiegészítéssel. A FIFO elv legnagyobb problémája az volt, hogy a régóta bent lévő, de még mindig használt lapokat állandóan kilapozta. Próbáljuk meg ezt a hátrányt kiküszöbölni. Minden laphoz - a laptáblába - helyezzünk el egy ún. „hivatkozás" bitet, amelyet, ha a lapot használjuk, automatikusan 1-esbe állítunk. (Ez igény szerinti lapozásnál egyben azt is jelenti, hogy ha a lapot behozzuk, ezt a bitet rögtön 1-esbe állítjuk, hiszen azért hozzuk be, mert hivatkoztunk rá!) Laphiba esetén keressük meg azt a lapot, amely a FIFO sor elején áll. Ha ennek a lapnak a hivatkozás bitje = 1, akkor mégse őt válasszuk áldozatul, hanem tegyük be a FIFO sor végére (mintha most érkezett volna), de 0-ás hivatkozás bittel. Azaz adjunk neki egy újabb esélyt. Ezt folytassuk addig, amíg a FIFO sor elejére egy olyan lap nem kerül, amelynek hivatkozás bitje = 0, és azt cseréljük le.

Nézzük meg az algoritmusunk működését egy példán (mely azért tér el az előzőektől, mert ezen jobban illusztrálható a működés). Itt a táblázat kockáiban a lapsorszám után vesszővel elválasztva a hivatkozás bit értékét tüntettük fel.

Igényelt lap	6	8	3	8	6	0				8	6	
1.lap	6,1	6,1	6,1			6,0	6,0	6,0	0,1	0,1	0,1	0,1
2.lap		8,1	8,1			8,1	8,0	8,0	8,0	8,1	8,0	8,0
3.lap			3,1			3,1	3,1	3,0	3,0	3,0	3,0	6,1
	*	*	*						*	!	!!	*

7.15 ábra Laphibák SC stratégia esetén

Nézzük meg, mi történik! Amikor a 0. lapra hivatkozunk, akkor a 6-os lap van bent legrégebben, de 1-es hivatkozás bittel. Adjunk neki még egy esélyt, vagyis tegyük be a FIFO végére, de 0-ás hivatkozás bittel. Ekkor a FIFO tartalma 8,3,6 lesz, vagyis a 8-as a „legrégebbi" lap. Ő is kap egy új életet, stb. Látható, hogy végül - ugyan elég körülményes módon - itt ugyanazt - a 6-os - lapot cseréltük le, amit a FIFO algoritmus is lecserélt volna. Általánosan igaz az, hogy ha *minden bent lévő lap hivatkozás bitje 1-es*, akkor a második esély stratégia a FIFO elején lévő lapot cseréli le és

az összes többi bent lévő hivatkozás bitjét törli. (Erre külön kis célhardvert is szoktak készíteni, és így egy lépésben megkapható az, ami most végül négy lépésben jött ki.)

A 0. lap behozatala után a 8-ast használjuk. A 8-as bent van, de 0-ás hivatkozás bittel. Ezt gyorsan 1-be állítjuk, hiszen most hivatkoztunk rá. (Ez látható a !-lel jelölt oszlopban.) Ezután a 6-os lapot használnánk, de azt be kell hozni. A FIFO elején a 8-as lap van, de 1-es hivatkozás bittel. Új életet adva neki, betesszük a FIFO végére - 0-ás hivatkozás bittel (!!–lel jelölt oszlop) - és megnézzük, hogy most melyik lap van elöl. A 3-as, annak hivatkozás bitje 0, tehát lecserélhető, ennek helyére hozzuk be a 6-os lapot.

Itt látszik a módszer előnye, hogy végül is a régen behozott, de nem sokkal ezelőtt használt (8-as) lap bent maradt a memóriában.

7.2.3.6 Mostanában nem használt

A **„mostanában nem használt"** lapok cseréjét javasolja az LRU módszer enyhített, könnyebben megvalósítható változata. Mit jelent az, hogy *mostanában*? Az operációs rendszer, ha a folyamat egy lapra hivatkozik, a laptábla erre a célra fenntartott, egy bites mezőjét igazra állítja. Lapcsere esetén, ha lehetséges, azok közül a lapok közül kell választani, melyek „használt" bitje nulla. Ha egy laphoz már legalább egyszer fordultak, a jelzőbit állapota igaz. Hogy egy lap ne maradhasson örökre a tárban, a lapcsere algoritmus lapcserekor az összes lap jelzőbitjét nullázza. A *mostanában* kifejezés tehát azt takarja, hogy az előző lapcsere óta használták vagy nem használták a kérdéses lapot.

Mindössze egyetlen bit kiegészítéssel, és némi hardver támogatással az LRU módszerhez közeli hatékonyságot lehetett elérni. Nem kellett mást tenni, mint a „legrégebben" szóból elhagytuk a „leg" szócskát. E módszernél általában a nullás jelzőbitű lapokból véletlenszerűen választanak.

7.2.4 Hogyan csökkentheti a programozó a laphibák számát?

Meg kell említenünk, hogy a programozók (vagy manapság egyre inkább az intelligens, sokféle szempont szerint optimalizálni képes fordítóprogramok) is sokat tehetnek a laphibák számának csökkentésében. Néhány példa:

- Nem célszerű egy sokat használt ciklust úgy elhelyezni, hogy annak eleje az egyik lapon, míg vége egy másik lapon legyen, érdemesebb esetleg néhány üres (**NOP** - **N**o **op**eration) utasítás beszúrásával a teljes ciklust egy lapra tenni.
- Törekedni kell arra, hogy a program egy adott részén használatos adatokat lehetőleg egymáshoz közel (egy lapon) helyezzük el.
- Célszerű bizonyos helyeken az algoritmus elkészítésénél is figyelembe venni a lapozás tényét. Egy szélsőséges, de ennek ellenére sokszor előforduló példa: Tételezzük fel, hogy egy lap 2 kB, azaz 2048 B méretű. Egy egész számot tipikusan 2 B-on szoktunk ábrázolni, azaz egy lapra 1024 egész szám fér ki. Legyen egy olyan tömbünk, amely 1024 sorból áll és minden sorban 1024 egész szám van. Legyen az a feladatunk, hogy a tömb összes elemét ki kell nulláznunk. Valamint tételezzük fel, hogy a fordítóprogram a tömb elemeit sorfolytonosan tárolja, azaz jelen példánkban az első sort egy lapon, a másodikat a következőn stb. Ha az elemek kinullázását is sorfolytonosan végezzük 1024 laphiba keletkezik. Ám, ha a ciklusunkban csak annyit változtatunk, hogy a két indexet felcseréljük, és az elemeket oszlopfolytonosan akarjuk elérni (azaz először az első sor első elemét, majd a második sor első elemét - amely most egy másik lapon van!! - stb.) akkor 1024 * 1024 (azaz több mint egymillió!) laphibát okozunk!

7.2.5 A címszámítás gyorsítása asszociatív tárral

Ha még emlékszünk, a virtuális tárkezelés legnagyobb hátránya az volt, hogy lassú.

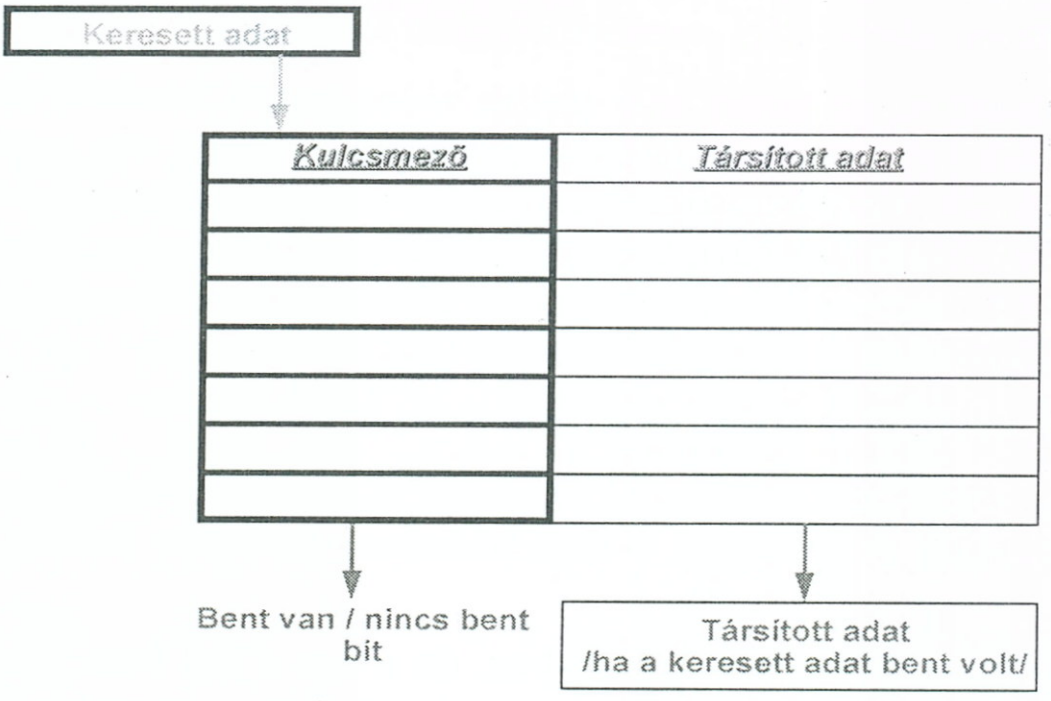
A sebesség növelése érdekében egyrészt igyekeztünk csökkenteni a laphibák számát megfelelően hatékony lapcsere algoritmus használatával, illetve a folyamatokhoz rendelt keretek számának optimális meghatározásával, másrészt a laphiba kezelésének az idejét minimalizáltuk úgy, hogy megfigyeltük, hogy egy kicserélendő lapot módosítottak-e vagy sem, míg az operatív memóriában tartózkodott.

Azonban ne felejtsük el azt, hogy a laptábla alkalmazásával a memória (látszólagos) sebessége akkor is legalább a felére csökken, ha egyáltalán nincs laphiba. Próbáljuk meg tehát csökkenteni a memória-hozzáférés idejét olyankor is, amikor nem volt laphiba!

Mielőtt erre rátérnénk, vizsgáljunk meg egy speciális típusú memória elemet!

A hagyományos memóriák úgy működnek, hogy bemenetükre egy címet kell adnunk, ennek hatására pedig a kimenetükön megjelenik egy adat.

Az úgynevezett *tartalom címezhető memóriák* (content addressable memory) ezzel ellentétben úgy működnek, hogy a bemenetükre egy adatot kell adnunk, a kimenetükön pedig megjelenik, hogy ez az adat benne van-e a memóriában vagy nincs, és mindez egy memória ciklusidő alatt. Ráadásul még ezeknek a memóriáknak a ciklusideje kb. *tizede* a hagyományos memóriák ciklusidejénél, azaz másképpen fogalmazva, kb. *tízszer gyorsabbak.* (Viszont meglehetősen drágák és elég kis méretűek.) Egy ilyen memóriamodulhoz hozzárakhatunk egy hagyományos tárolóelemet, és ilyenkor a kimeneten nem csak az jelenik meg, hogy a bemenetre adott adat (az ún. *kulcsinformáció*) bent van-e a memóriában, hanem ha bent van, akkor egy másik kimeneten megjelenik az adott kulcshoz társított, a hagyományos modulban lévő adat is. Ezért hívjuk ezt a típusú memóriát *asszociatív memóriának.*



7.16 ábra Asszociatív (tartalom címezhető) memória

Hogyan használhatunk egy ilyen memóriaelemet a virtuális tárkezelés címszámításának gyorsítására?

Ismét térjünk vissza a „jó öreg" lokalitási elvhez. Ebből következik, hogy ha - a laptáblán keresztül - kiszámítottuk egy lap kezdőcímét, akkor várhatóan azt a lapot a későbbiekben még újra használni fogjuk, azaz újra ki kell majd számolni a címét. Milyen jó lenne, ha lenne egy gyors tárunk, amely az utoljára használt néhány lap címét tartalmazza! Erre pont alkalmas az asszociatív memória! A kulcsmezőbe (tartalom címezhető rész) helyezzük el a lap *logikai címét*, tehát azt a címet, amit a processzor kiadott, míg a társított részbe helyezzük el a lap *fizikai címét*, azaz a laptábla megfelelő sorát. Ezek után hogyan hajtjuk végre a címszámítást? Ha a processzor kiad egy (logikai) címet, a címszámítást *párhuzamosan* elkezdjük a hagyományos módon, a laptáblán keresztül és az asszociatív tár kulcsbemenetére is ráadjuk. Ha szerencsénk van - és a lokalitási elvből következően általában (a gyakorlatban az esetek több mint 90%-ában) szerencsénk van - ez a cím ott van az utoljára használt néhány lap címei között, vagyis az asszociatív memóriából gyakorlatilag azonnal (kb. egytized operatív memória ciklusidő alatt) megkapjuk. Ekkor leállítjuk a hagyományos címszámítást, és már fordulhatunk az operatív memóriához a keresett adatért/utasításért. Ezzel az ilyen esetekben a több mint kétszeres hozzáférési időt kb. 1,1-szeresre sikerült csökkenteni.

De mi van olyankor, ha az asszociatív tárban nincs bent a keresett cím? Ekkor végigjárjuk a hagyományos címszámítási utat, *de ennek eredményét rögtön be is írjuk az asszociatív tárba a legrégebben használt lap címe helyére*. Ezáltal a következő hivatkozáskor már meg fogjuk találni az asszociatív tárban. Ezt az asszociatív tárat gyakran *címszámítást kikerülő tárnak* (**T**ranslation **L**ookaside **B**uffer - **TLB**) nevezik.

Ezek után három különböző szituáció lehet.

1. A keresett lap címe bent van a TLB-ben (ez a leggyakoribb), onnan gyorsan megkaphatjuk és leállítjuk a hagyományos, laptáblán keresztüli címszámítást.

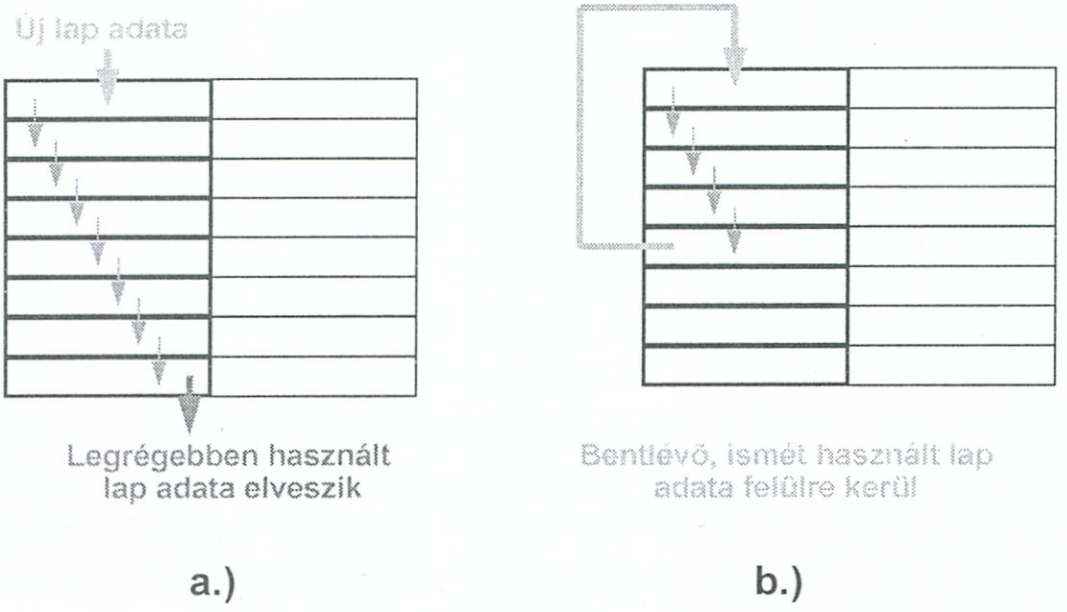
2. A keresett lap címe nincs bent a TLB-ben, de maga a lap bent van az operatív tárban. (Azaz a lapot régóta nem használtuk, de még nem lapozódott ki. Ilyen eset azért lehet, mert a TLB mérete - mint már említettük - meglehetősen kicsi, tipikusan 32, 64 vagy 128 soros, és ezért nem fér bele az összes, operatív tárbeli lap adata.) Ilyenkor végrehajtjuk a

címszámítást a laptáblán keresztül és beírjuk a kiszámított címet a TLB-be is.

3. A keresett lap nincs bent az operatív memóriában (laphiba). Ekkor a lapot be kell hozni a háttértárról, megfelelően módosítani kell a laptáblát, a módosított laptábla segítségével el kell végezni a címszámítást és a kiszámított címet be kell írni a TLB-be is.

Végül azt a kérdést kell megvizsgálni, hogy hogyan tudjuk könnyen biztosítani, hogy a TLB-ben mindig a *legutoljára használt* néhány lap adata legyen bent? (Láttuk, hogy ez okozta a fő problémát az LRU stratégia megvalósíthatóságánál.)

A megoldás az, hogy a TLB amellett, hogy egy asszociatív memória, egy speciális léptető (shift) regiszterként viselkedik. (Ez a viszonylag kis méret miatt tehető meg.) Az új lap adatai a léptető regiszter tetejére kerülnek, a bent lévő adatok pedig eggyel lejjebb lépnek, alul pedig elveszik az eddigi utolsó sor. Ha egy lapra hivatkozunk, akkor a lap adatait tartalmazó sor kerül a léptető regiszter tetejére és az összes eddig felette lévő sor eggyel lejjebb csúszik. (Ilyenkor tehát nincs adatvesztés.) Könnyen belátható, hogy ezzel a módszerrel a TLB tetején mindig a legutoljára használt lap adatai lesznek, míg legalul a legrégebben nem használt lap adatai találhatók meg.



7.17 ábra A TLB működése

7.3 Tárvédelem, szegmentálás

Multiprogramozott környezetben meg kell teremtenünk annak a feltételeit, hogy az egyes folyamatok egymás memóriaterületeit ne zavarják, még olyankor sem, ha például programhibát tartalmaznak. Vagyis ki kell alakítani egy jól működő védelmi rendszert. Azonban a folyamatok közötti szeparálás nem szabad, hogy „tökéletes" legyen, hiszen vannak olyan esetek, amikor több folyamat együtt akar működni, például egy közös adatterületen keresztül. Tehát a védelmi rendszernek meg kell akadályozni a „rosszakaratú" hozzáféréseket, de ellenőrzött keretek között biztosítani kell a folyamatok közötti kommunikációt.

A tárvédelemnek általában három különböző szintjét szoktuk megkülönböztetni:

1. Védeni kell egy folyamat különböző logikai egységeit egymástól.
2. Védeni kell a felhasználói folyamatokat egymástól, de biztosítani kell közöttük az igényelt kommunikáció lehetőségét.

3. Védeni kell az operációs rendszert a felhasználói folyamatoktól.

A következőkben tekintsük át, hogy a gyakorlatban milyen megoldások terjedtek el!

7.3.1 A folyamatok logikai egységeinek védelme

Tipikus programhiba szokott lenni, amikor rosszul használjuk a veremtárat (stack memóriát), és több adatot töltünk be oda, mint amennyit onnan kiolvasunk, ezáltal a veremtárban lévő adatok egy idő után teljesen megtöltik azt, majd elkezdik felülírni a veremtár előtt lévő más adatokat vagy utasításokat (stack overflow - verem túlcsordulás).

Hasonlóképpen veszélyes helyzetet okozhat, ha például egy ugróutasításban rossz címet adunk meg és ezáltal beleugrunk az adatok közepébe és elkezdjük azokat „végrehajtani". Ennek a fordítottja is gyakran előfordul, vagyis amikor egy memóriába író utasításnak adunk meg rossz címet és az adatunkkal véletlenül felülírjuk a programunk valamelyik utasítását.

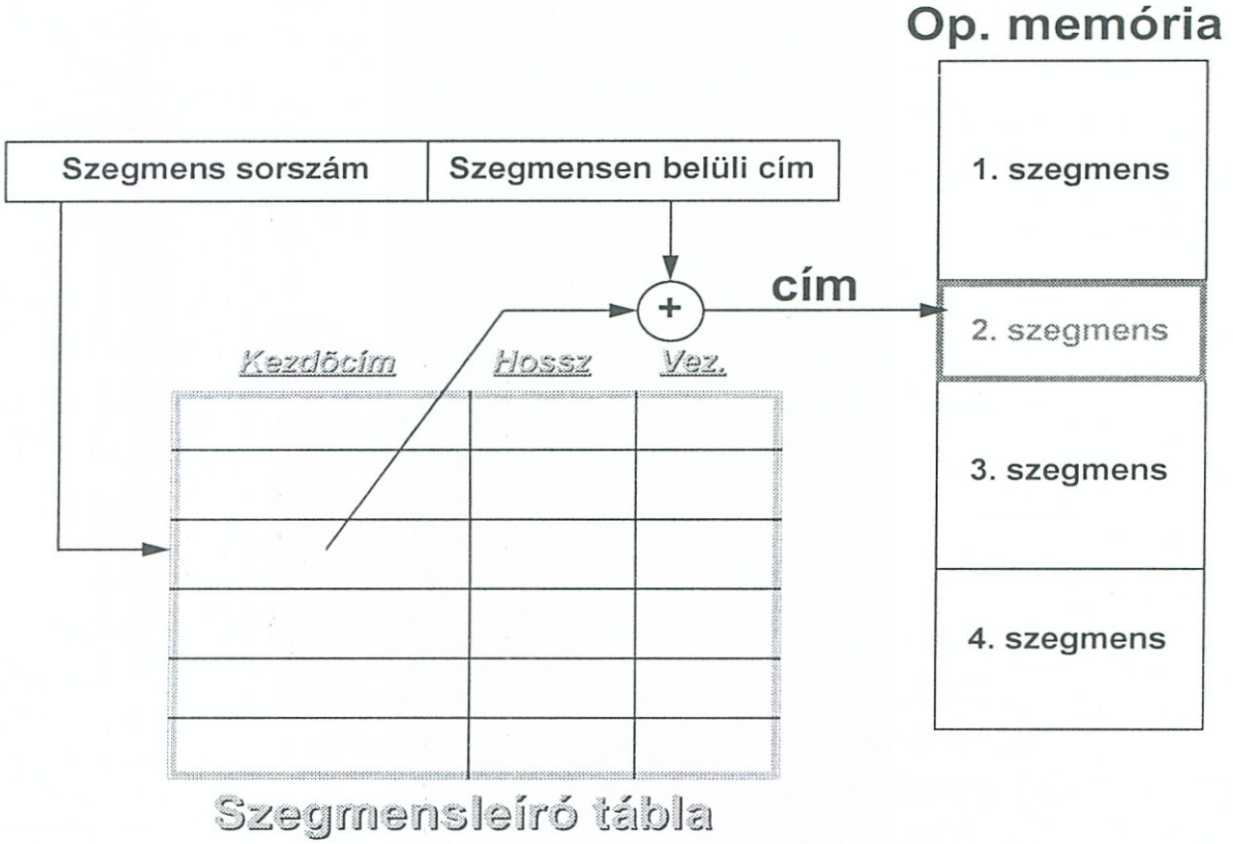
Hogyan lehet ezek ellen védekezni?

Kérjük meg a *programozót* (illetve ma már ezt elsősorban a fordítóprogramok teszik meg helyette), hogy a programírás során határozza meg, hogy hol található a programkód, hol vannak az adatok - sőt kijelölhet több, egymástól független adatterületet is, amelyek más-más célokat szolgálnak -, illetve mondja meg azt, hogy hol és mekkora legyen a veremtár. Másképpen fogalmazva: jelölje ki a program logikai egységeit, azaz *szegmentálja* a programot. Vagyis készítsen egy **kódszegmenst** (az utasítások számára) egy vagy több **adatszegmenst** és egy **veremszegmenst**. Mondjuk meg az operációs rendszernek, hogy milyen szegmenseket készítettünk és kérjük meg arra, hogy – a sebesség érdekében sokszor a hardver segítségével – menet közben mindig ellenőrizze, hogy az általunk definiált szegmenseken belül akarunk-e dolgozni.

Mit kell tudni ehhez az operációs rendszernek?

Először is tudnia kell, hogy milyen szegmenseket definiáltunk, azok hol vannak, milyen típusúak és milyen hosszúak (hiszen a szegmensek mérete nem egyforma!). Ennek a nyilvántartásához készítsünk - a laptáblához hasonlóan - egy ún. **szegmensleíró táblát** (segment descriptor table), amely minden egyes szegmensről éppen a fenti adatokat tartalmazza.

Azaz annyi sora van, ahány szegmenst definiáltunk, minden egyes sorban lesz egy mező, amely megadja a **szegmens kezdőcímét**, a **szegmens hosszát**, a **szegmens típusát**, valamint további néhány - a későbbiekben tárgyalandó szerepű - ún. **védelmi bitet**.



7.18 ábra Címszámítás szegmentáláskor

Amikor pedig a processzor utasításkészletét tervezik, minden - memóriát használó - utasításhoz hozzárendelnek egy szegmenstípust, amilyen szegmensen belül az utasítás által használt címnek lennie kell. (Például: PUSH/POP utasítások: stack szegmens; adatmozgató, műveletvégző utasítások: adatszegmens; ugróutasítások: kódszegmens.)

Már csak egy lépés van hátra: a programozónak/fordítóprogramnak olyan címeket kell használnia, amelyek - logikailag - két részből állnak: az első rész a szegmens „nevét", azaz sorszámát jelenti, míg a második rész mondja meg, hogy a szegmensen belül melyik sort akarjuk elérni (eltolás).

A dolog most már egyszerű: egy utasítás végrehajtásakor a processzor által kiadott címet két részre kell osztani, a felső rész segítségével megkeressük a szegmensleíró tábla megfelelő sorát, ahol először is azt ellenőrizzük, hogy a szegmens típusa megegyezik-e az utasítás által megkövetelttel, majd megnézzük, hogy a cím második része beleesik-e a szegmens kezdete és hossza által meghatározott tartományba, (valamint ellenőrizzük a védelmi biteket - lásd később). Ha bármelyik ellenőrzés során hibát tapasztalunk, nem engedjük az utasítást végrehajtani, egy hibajelzést adunk. Ha nincs hiba, akkor a szegmensleíró tábla kezdőcím értékéhez a processzor által kiadott logikai címben szereplő eltolást hozzáadva, megkapjuk a keresett adat/utasítás fizikai címét.

Láthatjuk, hogy ezzel a módszerrel sikerült kiküszöbölni a fejezet elején említett tipikus hibákat, azaz egy folyamat logikai egységeit meg tudjuk védeni egymástól.

Egy aprócska baj van csak, hogy a megoldás „túl jóra sikerült". A bevezetőben például azt említettük, hogy az általában nagy baj, ha egy adatmozgatás során felülírunk egy utasítást. Ez többnyire igaz, ha *véletlenül* tesszük ezt meg. Ugyanakkor azonban viszonylag gyakran alkalmazott programozói fogás, hogy egy program menet közben *szándékosan* módosítja önmagát. (Így működnek például azok az ún. „mesterséges intelligenciával" rendelkező programok, amelyek „tanulni" képesek.) A szegmentálás bevezetésével az ilyen és hasonló programozói „trükkök" előtt is becsuktuk a kaput. A megoldás viszonylag egyszerű. Be kell vezetni egy olyan utasítást, amely segítségével *egy utasítás erejéig* módosíthatjuk az utasításhoz rendelt szegmens típusát. Mivel ezeket az utasításokat általában a módosítani kívánt utasítások elé kell írni, ezért ezeket *szegmensmódosító prefix utasításoknak* (segment override prefix) hívjuk.

Például:

```
CS: MOV cím, regiszter
```

A MOV utasítás a *regiszter* tartalmát a memóriába a *cím* helyre írja, azt feltételezve, hogy az egy adatszegmens belsejében található. A CS prefix segítségével mondjuk meg azt, hogy most a *cím*-ről azt kell feltételezni, hogy az egy kódszegmensben van.

7.3.2 A folyamatok védelme egymástól

A másik védelmi feladatunk, hogy a folyamatok memóriaterületeit megvédjük egymástól, de biztosítsuk a szükséges folyamatközti kommunikáció lehetőségét.

A megoldás gyakorlatilag már készen van, egy pici ötletre van csak szükség. Ne egy darab szegmensleíró táblánk legyen, amely az összes folyamat összes szegmensét leírja, hanem rendeljünk *egy-egy szegmensleíró táblát minden folyamathoz*. Ezzel a feladatot megoldottuk, hiszen így minden folyamat csak azokhoz a szegmensekhez férhet hozzá, amelyek a *saját* leírótáblájában fel vannak tüntetve, a memória többi részéről, és így értelemszerűen az ott lévő egyéb szegmensekről nem is tudnak.

Egy kérdés maradt nyitva: hogyan tudnak kommunikálni egymással a folyamatok, ha ezt igénylik? A válasz nagyon egyszerű: senki sem mondta azt, hogy az egyes leírótáblákban *csak különböző* szegmensek adatai szerepelhetnek. Ha két folyamat például egy közös adatterületet szeretne használni, akkor erre a közös területre definiáljunk egy adatszegmenst és ennek adatait tegyük be *mindkét* folyamat leírótáblájába. Ezáltal ehhez mindkét folyamat hozzá tud férni, de rajtuk kívül *senki más*.

Itt nyílik lehetőség egy újabb trükkre.

Tegyük fel, hogy például egy bank számlakezelő programját kell elkészíteni. Ilyenkor lesz minden számlatulajdonosnak egy-egy folyamata, valamint egy olyan folyamatunk is, amely a bank tevékenységét írja le. Nyilvánvaló, hogy ahhoz az adatszegmenshez, amely X. számlatulajdonos számlaegyenlegét tartalmazza, X. folyamatának és a bank folyamatának is hozzá kell tudnia férni, - ez tehát egy közösen használható adatszegmens lesz -, de azt csak a banknak szabad megengedni, hogy *módosítsa* az egyenleget, X-nek nem, ő csak *olvashatja*. (De szép is lenne az ellenkezője!) Hogyan lehet ezt megoldani? Egyszerűen a leírótáblában a szegmens adatait ki kell egészíteni egy bittel (vagy komplikáltabb esetekben bitcsoporttal), amely azt mondja meg, hogy hogyan használhatja az adott folyamat a szegmenst - ez a **hozzáférési jog**ot (access rights) szabályozó bit(csoport). Ez a bit(csoport) a már korábban említett védelmi bitek között található. Ezek után az egyenleget tartalmazó szegmens az X. és a bank folyamat

leírótáblájában is szerepelni fog, de az előbbi helyen *csak olvasható*, míg az utóbbi helyen *olvasható és írható* hozzáférési joggal. Természetesen a címszámítás során ezt a bitet is ellenőrizni kell, és ha egy folyamat meg nem engedett módon akar egy szegmenshez fordulni, akkor azt nem szabad megengedni és hibajelzést kell adni.

Az első fejezetben említettük, hogy az operációs rendszer egy csomó, kellemesen használható szolgáltatást nyújt a felhasználóknak, például képernyőre írás, nyomtatás stb. Most már könnyen kitalálható, hogy ezek is egy-egy szegmensben találhatók meg. Ahhoz, hogy ezeket - és más hasonló segédprogramokat - minden folyamat használni tudja, minden egyes folyamat leírótáblájában szerepeltetni kell őket. Ez azonban nagy pazarlás. Ezért azt szokták tenni, hogy létrehoznak egy közös, mindenki által használható, úgynevezett **globális leírótáblát** (**G**lobal **D**escriptor **T**able - **GDT**) a már említett, minden folyamathoz rendelt saját, **lokális leírótáblán** (**L**ocal **D**escriptor **T**able - **LDT**) felül. Megjegyzés: sok esetben a megszakítási rutinok számára is külön leírótáblát (**I**nterrupt **D**escriptor **T**able - **IDT**) szoktak készíteni.

Azt pedig, ugye, már felesleges is említeni, hogy ha minden folyamatnak van egy-egy leírótáblája, akkor annak a helye (kezdőcíme) is része lesz a folyamat azon jellemzőinek, amelyeket egy folyamat felfüggesztésekor a PCB-be el kell menteni?

7.3.3 Az operációs rendszer védelme - prioritások

Ha azt mondjuk, hogy az operációs rendszer is egy - illetve, mivel a gyakorlatban az operációs rendszer is több folyamatból szokott állni, néhány - folyamat a rendszerben lévő folyamatok közül, saját szegmensleírótáblával, tulajdonképpen az előző pontban felvázoltakkal megoldottuk az operációs rendszer védelmét is más folyamatoktól. Azonban az operációs rendszer folyamatainak célszerű speciális jogokat biztosítani, amelyeket más folyamatoknak nem adunk meg (például, tudja módosítani más folyamatok leírótábláit). Ezért célszerű, ha minden folyamathoz egy-egy ún. **prioritási szintet** rendelünk. Emellett a szegmensleíró tábla sorait is egészítsük ki egy-egy új mezővel, amely azt jelzi, hogy *legalább milyen prioritási szinttel kell rendelkeznie* egy folyamatnak ahhoz, hogy az adott szegmenst használja. Ez az ún. prioritás (priority) mező is a védelmi bitek között helyezkedik el. Ezek

után a címszámítás során azt is ellenőriznünk kell, hogy a szegmenst használni akaró folyamat tényleg megfelelő prioritási szintű-e.

Kezdőcím	Hossz	Típus	Prioritás

7.19 ábra A szegmensleíró tábla egy sorának felépítése

Korábban azt tanultuk, hogy egy folyamat leírótáblájában azokat a szegmenseket tüntetjük fel, amelyeket a folyamat használhat. Jogos lehet a kérdés, hogy miért tegyünk ide egy szegmenst egy olyan prioritási szinttel, amely fontosabb, mint a folyamat prioritása, azaz a folyamat nem fogja tudni használni? Magyarázatul nézzünk két egyszerű példát:

Az egyik a globális leírótábla használatával kapcsolatos. Azt mondtuk, hogy a globális leírótáblában olyan szegmenseket sorolunk fel, amelyeket elvileg bármely folyamat használhat. Most, ha minden itt szereplő szegmenshez egy prioritási szintet rendelünk, akkor ez úgy módosul, hogy a szegmenst a rendszerben szereplő bármely, *legalább adott prioritással rendelkező* folyamat elérheti, de más nem.

A másik példa a következő: az operációs rendszerek szempontjából nagyon kényelmes, ha egy adott folyamatra vonatkozó összes információ egy helyen található meg. Ez a hely pedig célszerűen a folyamat lokális szegmensleíró táblája. Azaz tegyük ide azon szegmensek adatait, amelyeket a folyamat használni szeretne - természetesen olyan prioritási értékkel, hogy a folyamat el tudja érni -, de tegyük ide azon a szegmensek adatait is, amelyek az adott folyamatra vonatkoznak, ám csak az operációs rendszer számára kellenek - a legtipikusabb példa a folyamatleíró blokkot (PCB) tartalmazó ún. folyamat- vagy *task állapot szegmens* (**T**ask **S**tate **S**egment - **TSS**). Ezeknek a szegmenseknek pedig adjunk olyan prioritási szintet, hogy csak az operációs rendszer férhessen hozzájuk, maga a folyamat ne.

7.3.3.1 A címszámítás gyorsítása szegmentálásnál

A szegmentálás bevezetésével hasonló probléma merül fel, mint a lapozásnál. Ugyanis ahhoz, hogy a memóriából megkapjunk egy adatot, először a szegmensleíró táblához kell fordulnunk, majd csak ezután érhetjük el a keresett információt. Vagyis a memória hozzáférés ideje

legalább *kétszeresére nő*. Ez szintén elfogadhatatlan, tehát most is tennünk kell valamit a gyorsítás érdekében.

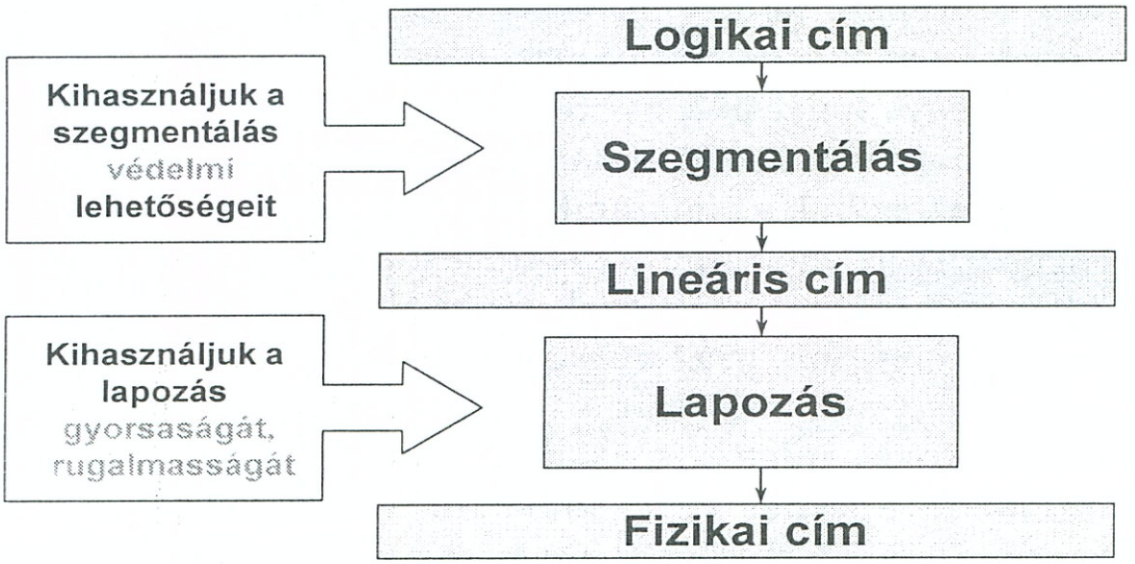
Azonban, mivel a szegmensek *logikailag összetartozó* dolgokat tartalmaznak, ezeken belül a lokalitási elv sokkal erősebb, mint a tulajdonképpen véletlenszerűen szétszabdalt programrészeket tartalmazó lapok esetében.

Itt nem kell olyan bonyolult struktúra, mint amilyen a TLB volt, elegendő, ha kialakítunk egy **kódszegmens regisztert** - amelyben az éppen futó folyamat kódszegmensének az adatai (kezdőcím, hossz, védelmi bitek) vannak, illetve ehhez hasonlóan egy **stack szegmens regisztert** és egy vagy néhány (tipikusan 2 vagy 4) **adatszegmens regisztert**. A címszámításnál hasonlóan járunk el, mint a lapozásnál. Párhuzamosan elkezdjük a címszámítást a szegmensleíró táblán keresztül, illetve megnézzük az adott típusú szegmensregiszter(eke)t. Mivel itt most regisztereket használunk, ezért ezek sebessége még a TLB-énél is több nagyságrenddel nagyobb, azaz gyakorlatilag „nulla idő alatt" megkapjuk a kért szegmenscímet, másrészt - éppen a szegmensen belüli szoros logikai összetartozás miatt - a találati arány gyakorlatilag 100% lesz. Például szinte az egyetlen eset, amikor a kódszegmens regiszter rossz értéket tartalmaz az az, amikor a folyamat elindítása/újraindítása után az első utasítását végrehajtjuk. Ilyenkor a kódszegmens regiszter még az *előzőleg futott* folyamat kódszegmensének adatait tartalmazza, vagyis az új folyamat kódszegmensének adatait csak a leírótáblából kapjuk meg, de ezt mindjárt be is írjuk a kódszegmens regiszterbe és ott most már *mindig megtaláljuk, amíg a folyamatunk fut*. Hasonló a helyzet a folyamat indulását/folytatását követő első veremművelet illetve első néhány adatművelet esetén a verem- illetve adatszegmens regiszterekkel kapcsolatban.

7.3.3.2 Összetett memóriakezelés

A gyakorlatban a szegmentálást és a lapszervezésű virtuális tárkezelést együttesen alkalmazzák. Mint azt láttuk, a szegmentálás *elsősorban a védelmi rendszer része*, azaz segítségével azt lehet eldönteni, hogy egy adott memóriaterülethez valakinek *joga van-e hozzáférni*. Ha a válasz igen, akkor meg kell keresni, hogy ténylegesen *hol van* a keresett memóriarész. Ennek eldöntésére pedig a virtuális tárkezelés szolgál.

Tehát a processzor által kiadott *logikai cím* először a szegmentáló egységbe kerül, majd - ha engedélyezett a memóriaművelet - a szegmensfordítás kimenete lesz a lapozóegység bemenete. (Ilyenkor ezt a címet *lineáris címnek* nevezzük. Ez arra utal, hogy a virtuális memóriát egy nagyméretű, folytonos memóriaként képzeljük el.) Végül a lapozóegység kimenete adja meg a keresett memóriarekesz *fizikai címét* az operatív memóriában.



7.20 ábra A szegmentálás és a virtuális tárkezelés együttes alkalmazása

Ma már az egyszerűbb mikroprocesszorok is nagyon összetett memóriakezelési lehetőséggel rendelkeznek és belső struktúrájukat is úgy alakították ki, hogy a minél gyorsabb memóriaelérés érdekében lehetőleg minél nagyobb hardver támogatást nyújtsanak az operációs rendszerek memóriakezelő funkciói számára. Vagyis a mai processzorokban általában egy komplett memória vezérlő egység (**M**emory **M**anagement **U**nit - **MMU**) is megtalálható.

7.4 Gyorstárak (cache memóriák)

A memória hozzáférést tovább lehet gyorsítani úgy, hogy a processzor és az operatív memória közé egy viszonylag kisméretű, de nagyon gyors tárat, egy ún. **cache memóriát** teszünk. Ez a cache memória mindig az operatív memória legutoljára használt részeinek és környezetüknek a másolatát tartalmazza.

7.21 ábra A cache memória helye

Ha memóriához akarunk fordulni, akkor a keresett adatot az operatív memóriában és a cache-ben egyszerre kezdjük el keresni, és ha szerencsénk van akkor a cache-ban nagyon gyorsan megtaláljuk, majd leállítjuk a keresést az operatív memóriában. (A gyakorlati operatív memória / cache memória méretarányokból következően kb. 90%-os az ún. cache találati arány, azaz csak az esetek kb. egy tizedében kell kivárnunk a lassú operatív memóriát.)

Ha a keresett adat nem volt bent a cache-ban, akkor azt és környezetét behozzuk oda az operatív memóriából.

Könnyű belátni, hogy a cache memória és az operatív memória viszonya és vezérlése nagyon hasonló az operatív memória / virtuális memória viszonyhoz és vezérléshez. Csak mivel a cache memória mérete lényegesen kisebb, ezért ez a tény bizonyos speciális módszerek alkalmazását is lehetővé teszi, tovább gyorsítva a működést.

7.5 Tároló hierarchia

Láthattuk, hogy egy korszerű rendszerben sok, különböző méretű és sebességű tár található, amelyek együttesen alkotják az ún. *tároló hierarchiát.*

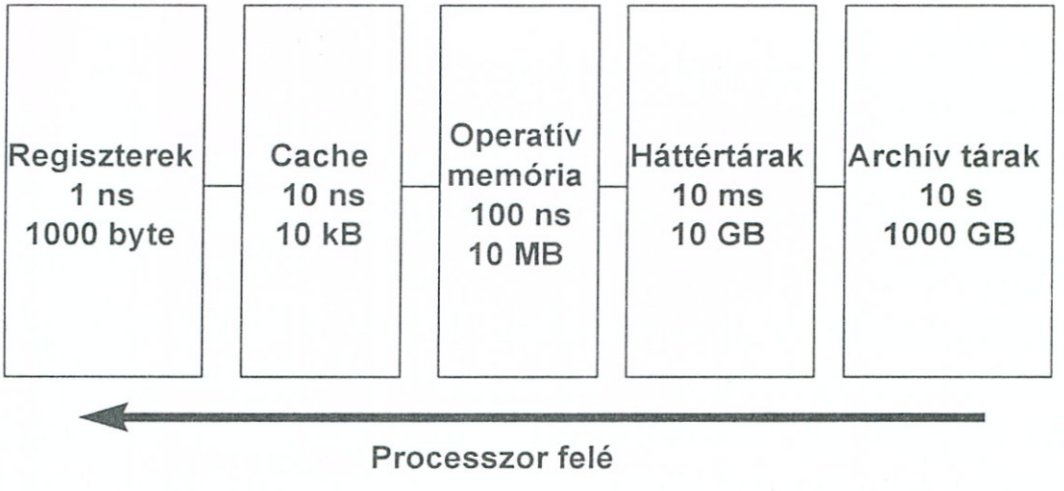
A tárcsere (swapping) és az átfedéses (overlay) technika az operatív tár szervezésébe már bevonta a háttértárat is, kihasználva annak jóval nagyobb méretét, és elszenvedve annak jóval nagyobb elérési idejét. Fokozottan érvényesülnek ezek a hatások a virtuális tárkezelés használata esetén. Léteznek olyan rendszerek is, melyek a programozás szempontjából nem is tesznek különbséget a tároló funkciók különböző fizikai megvalósulása között, hanem egységesen kezelik azokat, az adatáramlás szervezése kizárólag az operációs rendszer (és az azt támogató célhardver) feladata. Érdemes ezért egy kicsit elidőzni ennél a kérdésnél.

Ha több folyamat is van a rendszerben, és a processzor több folyamat utasításait kell végrehajtsa, célszerű szem előtt tartani az úgynevezett **gyorsítótár elv**et. A gyorsítótár elv mögött az a szemlélet húzódik meg, hogy a processzor számára leggyorsabban elérhető helyen azok az adatok legyenek, amelyekre a leggyakrabban van szükség. Az elv megsértése keserves eredményhez vezethet. Ha a processzornak a folyamat futásához szükséges adatokat például mindig a lassú háttértárakról kellene beszereznie, a rendszer fő tevékenysége nem a felhasználói folyamatok végrehajtása, hanem az állandó háttértár-memória adatmozgatás lenne.

§

Gyorsítótár-elv
A leggyakrabban használt adatok legyenek a leghamarabb elérhető helyen

Az adattároló eszközökre sajnos igaz az az állítás, hogy minél nagyobb méretűek, annál lassabban lehet hozzájutni az adataikhoz, másrészről viszont minél gyorsabbak, annál drágábbak is. A processzor regisztereinek nanoszekundum nagyságrendű elérési idejétől a mágnesszalagos egységek néhány perces reakciójáig számos lépcső létezik:



7.22 ábra Minél nagyobb, annál lassabb

A fenti ábra csak nagyságrendeket közöl. A legritkábban használt adatok kerülhetnek a leglassabb, ám legnagyobb tároló kapacitású helyre. Ha egy folyamat egy adatot igényel, természetesen először a cache-hez fordul, és

megnézi, nincs-e ott véletlenül a kívánt paraméter. Ha megvan (**cache találat,** cache hit), jó, lehet használni. Ha nincs (**cache hiány,** cache miss), hiba keletkezik, lehet keresni egy szinttel feljebb. Ha azonban egy lassú tárban megtaláljuk a várva várt adatot, célszerű a kért adatot, valamint - a lokalitási elvnek megfelelően - annak egy környezetét is átmásolni az „eggyel gyorsabb" tárba, és ez a folyamat így mehet egészen a regiszterekig. Az előrenyomuló adatok természetesen helyet igényelnek, kiszorítják a kevésbé használt adatokat, azok a tárhierarchia lassabb régióiba lépnek vissza. Természetesen, ha egy adat felkerült a gyorsabb tárak valamelyikébe, csak akkor kell onnan visszatérnie egy lassabba, ha ez igazán fontos, azaz ha kell a hely más fontosabb adatnak.

A memória látszólagos mérete tehát megegyezik a leglassabb háttértár méretével, az elérési idő viszont igen széles tartományban változik. Tételezzük fel, hogy az átlagos találati valószínűség 95%. Ekkor, ha az igazán kis méretű regiszter tárakat elhanyagoljuk, az átlagos elérési idő:

0,95*10ns + 0,05*(0,95*100ns+0,05*(0,95*10ms+0,05*10s)) = 12,7 μs

Az eredmény eléggé elrettentő, de még a 99% találati valószínűségnél eredményül kapott 1,1 μs sem valami kedvező. De gondoljunk csak egy kicsit bele. Az esetek 99%-ában az elérési idő 10 ns, és csak ha mind az 1000 GB-ot szeretnénk címezni, akkor kapjuk az 1 μs-ot! Az átlag azonban (mint mindig) itt is csal. A kedvező elérési idő csak olyan folyamatok esetén érvényes, ahol a folyamathoz tartozó programkód mérete a cache méretének nagyságrendjébe esik. (Íme egy újabb érv a moduláris programozás mellett!)

7.6 Linux memóriakezelés

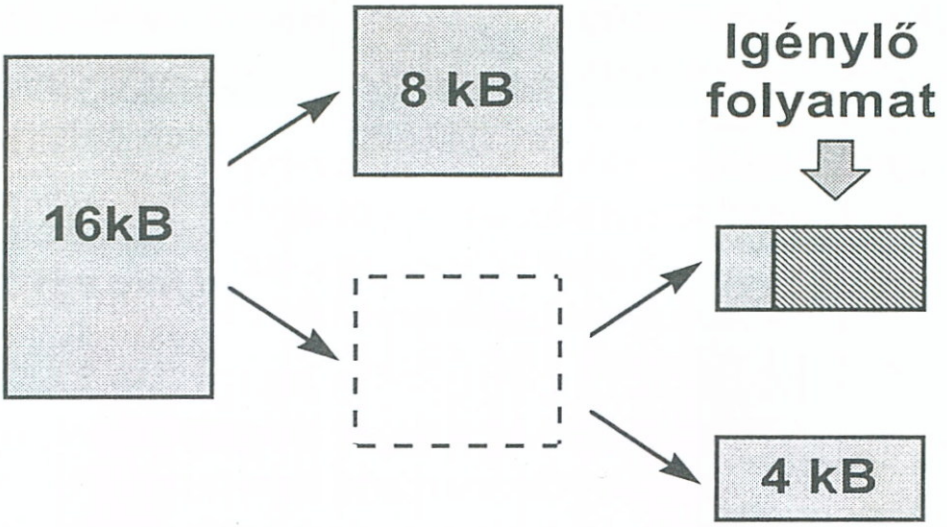
A memóriakezelő két feladatot lát el. A fizikai memória karbantartásáért felelős rész gondoskodik a lapok, lapcsoportok lefoglalásáról illetve felszabadításáról, a másik összetevő a felhasználói folyamatok felől látható virtuális memória kezelését végzi.

7.6.1 A fizikai memória kezelése

A Linux alapvetően a *buddy-heap* algoritmust használja. Ez azt jelenti, hogy ha két szomszédos tartomány szabad, akkor a rendszer ezeket azonnal összevonja egy összefüggő, nagyobb tartománnyá. A szabad helyek nyilvántartását több, független láncolt lista végzi, minden mérethez külön sor tartozik.

Tegyük fel, hogy a minimális lapméret 4 kB, és érkezik egy 2 kB-ot igénylő folyamat, de mind a 4 kB-os területeket, mind a 8 kB-os helyeket nyilvántartó lista üres. A *buddy-*

heap ilyenkor felosztja az első 16 kB-os helyet két 8 kB-os darabra, majd az egyik 8 kB-ost további két 4 kB-os darabra és az egyiket lefoglalja a folyamat számára. A listákról tehát eltűnik egy 16 kB-os összefüggő terület, de megjelenik egy 8 kB-os és egy 4 kB-os, és még a folyamat is megkapta a maga lapját.



Erre az alaprendszerre épülnek aztán a buffer cache, page cache és virtuális memória kezelő alrendszerek.

A *buffer cache* a blokk-orientált eszközökről érkező vagy oda tartó adatok átmeneti tárolására szolgál. A *page cache* bármilyen I/O csatornához kapcsolódhat, tipikusan a hálózattal való kommunikációt segíti, vagy fájlokat helyezhet el benne a rendszer.

A rendszerek természetesen szorosan együttműködnek. Ha például egy folyamat a saját területére kíván átmeneti tárolás (és gyorsabb elérés) céljából egy fájlt beolvasni (*page cache*), az adatok a *buffer cache*-en keresztül érkeznek a lemezről és betöltődés után a virtuális memória kezelő hatáskörébe kerül.

7.6.2 Virtuális memória

A Linux a fizikai memóriát lapokra és szegmensekre (Linux szóhasználattal régiókra) osztja fel. A szegmenseket lapok összefüggő halmazai alkotják, ezekhez rendelhetők olvasási, írási vagy végrehajtási jogok. A szegmensek különböznek abból a szempontból is, hogy kizárólag egy adott folyamathoz tartoznak (*private*) vagy több folyamat közösen használja őket (*shared*). Ha egy olyan lapra írunk, amely egyéni szegmens részét képezi, a rendszer megjegyzi ugyan, hogy megváltozott a lap, de mentésére csak akkor kerül sor, mikor az adott lap lecserélésre kerül. Ha a szegmens megosztott, a változás azonnal megjelenik a háttértáron (is), hogy a többi folyamat is értesülhessen róla.

Ha a folyamat már betöltötte az összes rendelkezésére álló memória lapját és egy újabbat igényel (*demand paging*) a rendszernek ki kell választania a memóriában lévő lapok közül egy áldozatot, melynek helyére az új lap bekerülhet. A Linux az áldozat kijelölésére a második esély (*second chance*) stratégia egy módosított változatát használja. Minden memória laphoz tartozik a laptáblában egy bejegyzés, amelyben a rendszer nyilvántartja a lap életkorát (*age*), azaz méri, hogy az adott lap mennyi ideje tartózkodik már a fizikai memóriában. A bejegyzésben szereplő értéket egy periodikusan

elindított rendszerfolyamat időről időre csökkenti. Ha azonban a felhasználói folyamat a laphoz fordul, egyúttal annak életkorát is megnöveli. Az operációs rendszer ezzel a politikával egyszerre veszi figyelembe azt, hogy egy lap milyen régen van már a tárban, illetve azt, hogy mennyire kelendő. A lecserélendő lap mindig az lesz, melynek életkora a legkisebb.

A fizikai memóriának nem minden területe fölött rendelkezik a virtuális memóriakezelő. A kernel fenntart magának egy védet (protected) tartományt saját céljaira, amelyhez így felhasználói folyamatok nem férnek hozzá. A kernel tehát a valós memóriában fut!

7.6.3 Programok betöltése

A Linuxnak (történeti okokból) több betöltő programja is van. Ha egy új program készül futni, a fájl fejlécéből a rendszer megállapítja, hogy melyik betöltő folyamatot indítsa valamint azt, hogy mekkora és milyen felépítésű memóriaterületre lesz a folyamatnak szüksége. A folyamathoz tartozó lapok nem kerülnek azonnal a fizikai memóriába, de a betöltő folyamat létrehozza a laptáblát, beilleszti a folyamatot a virtuális memóriába, annak egyes részeit feltölti (pl. eltárolja a (virtuális) lemezen a program nevét, paramétereit).

Az előkészítés után a betöltő program beállítja a program fejlécéből kiolvasott értékekre a PCB program számlálóját, majd a további működtetést rábízza az ütemezőre és a virtuális memóriakezelőre. A Linux támogatja a dinamikusan betölthető rendszermodulok használatát. Ezek csak kifejezett kérés esetén kerülnek a memóriába, és akkor sem a kernel területére, hanem úgy viselkednek mint az egyszerű felhasználói folyamatok.

A modern operációs rendszerek mindegyike virtuális memóriakezelést használ. E fejezetben ennek előzményeként bemutattuk az átlapoló, tárcsere, lapozási és szegmentálási technikát. A virtuális memóriakezelés alapkérdése a lapcsere stratégia hatékonysága, ezért ezt részletesen, számítási példák segítségével ismertettük, majd a fejezet végén a címszámítást segítő hardver megoldásokat mutattuk be. A memóriát több folyamat közösen használja, ezért a védelem kérdése is létfontosságú. Ebben a részben kapott helyet a szegmens szervezésen alapuló tárvédelem tárgyalása is.

Σ

7.7 Ellenőrző kérdések

?

1. Mutassa be a címszámítási eljárást lapszervezésű virtuális tár esetén!
2. Ismertesse a számítógépek tárhierarchiájának elemeit, azok jellemzőit!
3. Ismertesse a virtuális tárkezelés lényegét!
4. Milyen előnyös tulajdonságokkal rendelkezik a „második esély" (SC) és a „mostanában nem használt" (NUR) technika? Hogyan működnek?
5. Mi a lényegi különbség a lapszervezésű és a szegmensszervezésű tárkezelés között? Egyszerre is alkalmazható a kettő?
6. Miért van szükség tárvédelemre? Hogyan segíti a szegmens szervezés a védelmi funkciók ellátását?
7. Milyen lapcsere stratégiákat ismer? Jellemezze őket röviden!
8. Mikor mondjuk azt, hogy egy folyamat „vergődik" (trashing)? Hogyan lehet megelőzni, illetve megszüntetni ezt a jelenséget?
9. Mutassa be a címszámítási eljárást szegmentálás esetén!
10. Mi a belső illetve külső elaprózódás? Hol léphetnek fel? Hogyan lehet ellenük küzdeni?
11. Mi a laphiba? Milyen lépésekből áll a laphiba kezelése? Hogyan lehet gyorsítani a laphiba kezelést?
12. Hogyan lehet gyorsítani a címszámítási eljárást lapszervezésű virtuális tár esetén?
13. Hogyan lehet gyorsítani a címszámítási eljárást szegmentálás esetén?
14. Hol és miért van szükség címszámítást kikerülő tár (TLB) használatára? Hogy működik?
15. Mi a lokális és a globális leírótábla? Mi szerepel bennük? Hogyan valósítható meg segítségükkel a folyamatok együttműködése tárvédelem mellett?
16. Értékelje a tanult lapcsere stratégiákat azok gyakorlati megvalósíthatósága szempontjából!

8. A párhuzamos programozás alapjai

Az adatfeldolgozás, számítás gyorsításának egyik lehetősége a párhuzamosítás. A folyamatok ilyenkor több végrehajtási ágra szakadhatnak, melyek mindegyike külön processzoron hajtódhat végre, majd újra egyesülhetnek a megfelelő feltételek esetén. A párhuzamos programozás a hagyományostól eltérő gondolkodásmódot igényel. Ennek eszközeivel ismertet meg a következő fejezet.

8.1 Bevezetés

Amint a korábbi fejezetekben már láttuk, a számítógép működését jelentősen meggyorsíthatjuk, ha a folyamatokat nem egymás után, hanem egymással párhuzamosan hajtjuk végre. Úgy is fogalmazhatunk, hogy az operációs rendszer és az általa végrehajtott folyamatok együttesen egymással versengő, azaz konkurens folyamathalmazt alkotnak. Egyprocesszoros rendszerben ezek a folyamatok alapvetően két különböző típusúak lehetnek. Az egyik csoportot az *erőforrásokért versengő, de egyébként egymástól független folyamatok* alkotják. E folyamatok tevékenysége között nincs kapcsolat, csak az, hogy ugyanazon gépen futva egymás „orra elől" igyekeznek megszerezni az erőforrásokat. (Megjegyzés: egyprocesszoros, multitasking környezetben szigorúan vett teljesen független folyamatok nem lehetnek, hiszen legalább a processzor egy olyan erőforrás, amelyért minden folyamat verseng.) A folyamatok másik csoportját a *kooperáló folyamatok* alkotják. Ezen folyamatokat a közös erőforrásokért vívott versenyen felül még a közös cél érdekében kifejtett tevékenység is összeköti. (Ilyenek voltak például a már korábban megismert termelő-fogyasztó folyamatok.)

Multiprogramozott környezetben tehát nyilvánvalóan szükség van a konkurencia leírására és a folyamatok közötti szinkronizáció megoldására.

A folyamatok közötti *szinkronizáció*nak három fő fajtája lehet. Az egyik a már megismert *kölcsönös kizárás*, azaz az, hogy két folyamat közül egy időben csak az egyik - de mindegy, hogy melyik - futhat. A másik az, amikor két folyamatnak egy dolgot egyszerre kell végrehajtania. Ez az ún. *randevú*. Ilyenkor az „előbb érkező" folyamatnak meg kell várnia a másikat, majd végrehajtásuk - többnyire paraméter-átadás után - újra szétválik. A harmadik tipikus eset a *precedencia*, vagyis a sorrend biztosítása. A precedencia itt azt jelenti, hogy egy bizonyos folyamat adott utasításcsoportjának végrehajtása meg kell, hogy előzze egy másik folyamat másik adott utasításcsoportjának végrehajtását. (Például egy üzenetet előbb be kell tenni egy postaládába és csak ezután lehet kiolvasni onnan.)

A következőkben nézzük meg, hogy - a szűk keretek miatt a teljesség igénye nélkül - milyen fontosabb módszerek, leírásmódok alakultak ki a párhuzamosság jellemzésére!

Mielőtt azonban erre rátérnénk, egy fontos megjegyzés. A mai rendszerekben általában *folyamatok* közötti párhuzamosításról beszélhetünk. Mi azonban - a könnyebb érthetőség és elképzelhetőség érdekében - olyan példákat nézünk meg, amelyben *utasítások* párhuzamos végrehajtásáról lesz szó. Ilyen - legalábbis a klasszikus, Neumann-gépeken - nincs, de példánk felfogható olyan speciális esetnek is, amikor a rendszerünkben nagyon rövid, mindössze egy utasításból álló folyamatok vannak.

8.2 A precedenciagráf

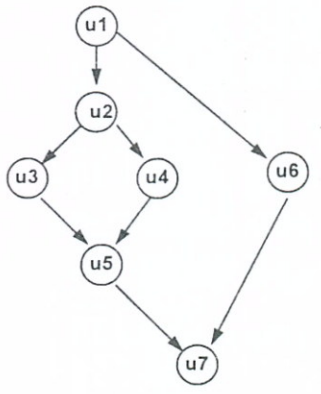
Nézzük meg figyelmesen a következő, u1-u7 utasításból álló programot!

```
u1:   x:=120;
u2:   y:=x/4;
u3:   a:=x-y;
u4:   b:=x+y;
u5:   c:=a+b;
u6:   d:=x-1;
u7:   e:=c*d;
```

Láthatjuk, hogy az u2-es utasítást csak az u1 befejeződése után hajthatjuk végre, hiszen az u2 az y-t az előző, az u1 utasítás eredményéből - x-ből - akarja kiszámolni. Ehhez a művelethez pedig addig nem foghatunk hozzá, amíg az x nincs meg, azaz amíg az u1 végrehajtását be nem fejeztük. Vagyis a bevezetőben említett terminológiával, u1-nek precedenciája van u2 felett, azaz u1 meg kell, hogy előzze u2-t. Viszont egész más a helyzet például az u3 és az u4 között. Az mindkettőre igaz ugyan, hogy csak x és y kiszámítása után hajthatjuk végre, de mivel *egymás* eredményét nem használják, ezek egymással párhuzamosan is végrehajthatók. Nézzük meg végül az u5 esetét! Itt a-ra és b-re is szükségünk van, azaz csak akkor mehetünk tovább, ha mindkettő megvan. Ez a randevú esete.

Nézzünk most egy eszközt, amellyel szemléletesen, könnyen áttekinthetően le tudjuk írni azt, hogy hol lehet párhuzamosítani, hol kell szinkronizálni („bevárni"), illetve hol kell utasításokat egymás után végrehajtani, azaz precedenciát biztosítani. Az eszköz innen kapta nevét is, ez az ún. **precedenciagráf**. A precedenciagráf csomópontjaiban folyamatok (jelen példánkban utasítások) vannak, és egy csomóponttól nyíl (irányított él) mutat egy másik csomópontig akkor, ha az első csomópont által reprezentált folyamat végrehajtása meg kell, hogy előzze a másik csomópont által reprezentált folyamatét. Tehát a nyilak két folyamat végrehajtási sorrendjét határozzák meg. Készítsük el a fenti programunk precedenciagráfját!

```
u1: x:=120;
u2: y:=x/4;
u3: a:=x-y;
u4: b:=x+y;
u5: c:=a+b;
u6: d:=x-1;
u7: e:=c*d;
```



(ha több él találkozik egy csomópontban: szinkronizálás!)

8.1 ábra Precedenciagráf

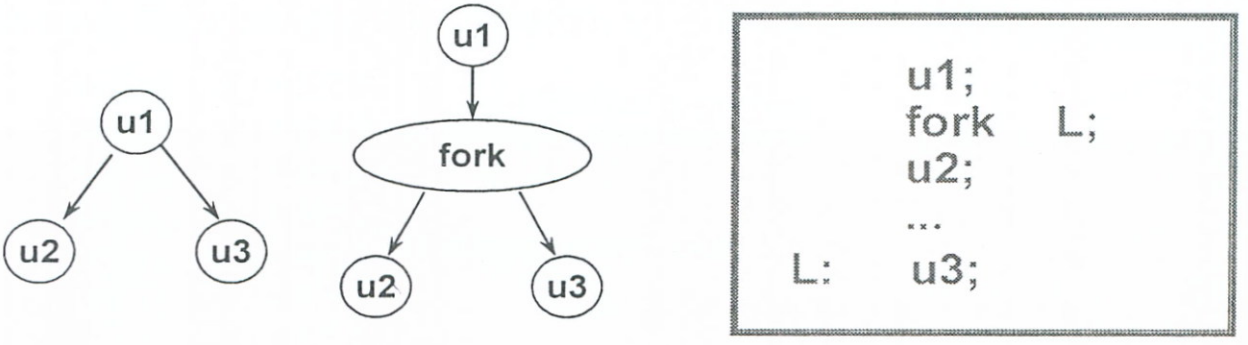
A precedenciagráf tehát egy irányított, körbejárható hurkot (ún. ciklust) *nem* tartalmazó gráf.

A precedenciagráfból ránézésre megállapítható, hogy mely utasítás kell megelőzzön mely más utasítást/utasításokat (ezt mutatják a nyilak), hol van szükség szinkronizációra (u5 és u7 előtt), és végül, hogy hol lehetséges a párhuzamosítás. (Azok az utasítások hajthatók végre párhuzamosan, amelyekre igaz, hogy egyikből se tudunk eljutni a másikba a nyilak mentén; a példában az u6-tal párhuzamosítható az u2, u3, u4 és u5-ből álló utasításnégyes bármelyike, azon belül pedig az u3 az u4-gyel).

A precedenciagráfnak megvan még az a kellemes tulajdonsága, hogy vele bármilyen szituáció leírható, emellett szemléletes is, viszont hátránya, hogy egy számítógép nem ért belőle semmit, tehát programozásra nem alkalmas. Keresnünk kell tehát olyan leírásmódot, amely segítségével a számítógép számára érthetően írhatjuk le a párhuzamosítási lehetőségeket.

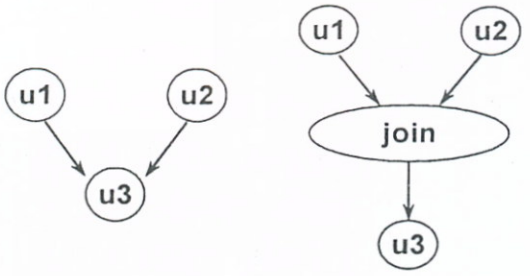
8.3 Fork - join utasításpár

Az első ilyen módszer a **fork** és **join** utasítások használata. A fork jelentése szétágazni, míg a join-é egyesülni. A **fork** utasítás segítségével a végrehajtás *pontosan két*, egymással párhuzamosan végrehajtható ágra osztható szét. Ezek közül az első ág utasításai közvetlenül a fork utasítás után kerülnek felsorolásra, míg a másik ág a fork utasítás paraméteréül megadott *címkénél* kezdődik. (Megjegyzés: Ha a végrehajtást kettőnél több párhuzamos ágra kell bontani, ez egymás után alkalmazott fork utasításokkal oldható meg. Egy ilyen példát majd a későbbiekben látni fogunk.) Az ábrán egy precedenciagráf-részlet, annak fork utasítással való kiegészítése, illetve programja látható.



8.2 ábra A fork utasítás

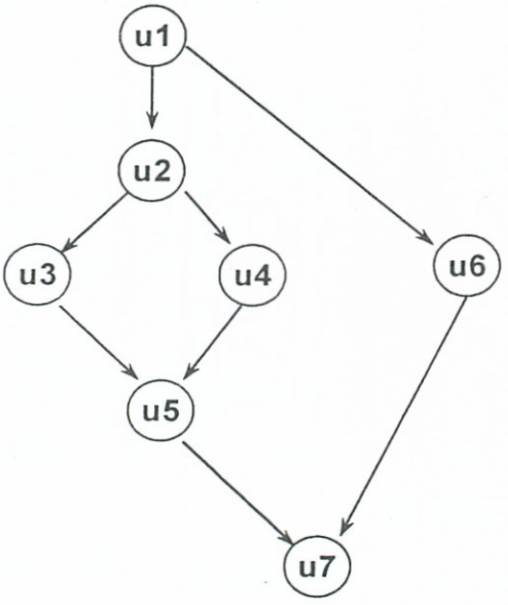
Egymással párhuzamosan futó ágakat a **join** utasítás segítségével egyesíthetünk. A join utasítás tetszőleges számú ágat egyesíthet, az egyesítendő ágak számát az utasítás paraméteréül megadandó *számláló* értéke mutatja meg. A join utasítás a szinkronizálást is megoldja: ha az itt találkozó ágak egyike befejeződik, azaz az ág programjának a végrehajtása után elérjük a join utasítást, akkor a join utasítás számlálójának értékét *eggyel csökkentjük*, majd az ágat felfüggesztjük. Ezt megtesszük mindaddig, míg az utolsó ág is be nem fejeződött - azaz amikor a számláló értéke 0 lesz - és ekkor elkezdődik az egyesített ág futása. Az egyesítendő ágak tehát „bevárják" egymást. Vegyük észre, hogy a leíráshoz a **goto** (azaz ugorj a paraméterül megadott címkére) utasítás használata nélkülözhetetlen.




```
    száml:=2;
    u1;
    goto L;
    u2;
L: join száml;
    u3;
```

8.3 ábra A join utasítás

Ezek után lássuk, hogy az előző fejezetben megszerkesztett precedenciagráf hogyan írható le fork-join utasítások segítségével!




```
     számI2 := 2;
     számI1 := 2;
     u1;
     fork L1;
     u2;
     fork L2;
     u3;
     goto L3;
L2: u4;
L3: join számI1;
     u5;
     gotoL4;
L1: u6;
L4: join számI2;
     u7;
```

8.4 ábra A fork-join utasításpár használata

Nézzük meg a program elkészítését lépésenként!

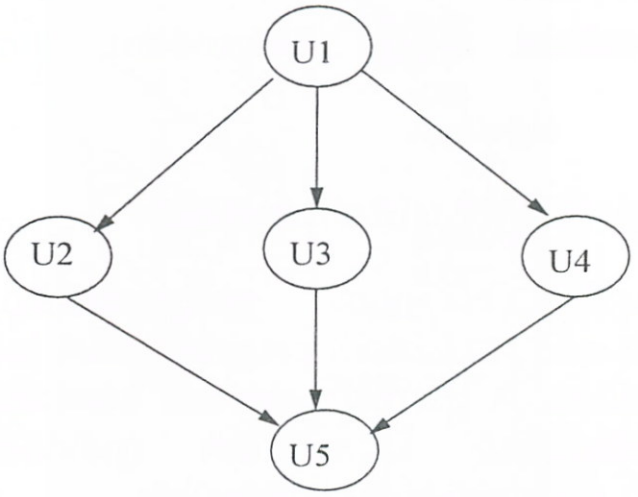
Az első utasítás az u1, majd utána két ágra bomlik a gráf, ide tehát egy fork utasítás kell. A baloldali részgráfot (u2, u3, u4 és u5) írjuk le közvetlenül a fork utasítás után, míg a jobboldali részgráfot (u6) majd máshol kezdjük, mondjuk az L1-es címkénél. Ezért használjuk tehát a fork L1; formát.

A baloldali részgráf az u2-vel kezdődik, majd egy újabb elágazás jön. Ennél is folytassuk a baloldallal, a jobboldali ág leírása pedig kezdődjön mondjuk az L2-vel jelzett sorban. Tehát írhatjuk: fork L2;. Ennek a baloldali részgráfja csak az u3-at tartalmazza. Ezután az u5 jönne, de előtte egyesíteni kell majd a két ágat egy join utasítással. Ezt írjuk majd a jobboldali ág végére, amit viszont még nem tudunk, hogy hol lesz, mondjuk adjuk neki az L3 címkét. Ide kell tehát ugranunk, ezért jön a goto L3; utasítás. Most befejeztük a második fork baloldalát, kezdjük el a jobboldalt! A fork utasításunk azt mondja, hogy a jobboldal majd az L2 címkénél kezdődik, tehát ide kell azt kitennünk, majd jöhet az u4. Ezt követi az ágak egyesítése az u5 előtt. Ide kell ugranunk a baloldali részgráf végéről is, tehát ez a sor lesz megjelölve az L3 címkével. A join utasításnak van egy számláló paramétere, ezt hívjuk pl. száml1-nek. Mivel itt két ágat kell egyesíteni, ezért ennek kezdőértéke 2 lesz, ezt írjuk a program elejére. A join után következhet az u5, és ez a vége az első fork utasítás baloldalának. Ugorjunk még el majd a jobboldali ág végére írandó join utasításra, amelyet tartalmazó sort jelöljük meg mondjuk az L4-es címkével.

Az első fork jobboldala az L1-es címkénél kezdődik. Ez csak az u6-ot tartalmazza, majd jön az egyesítés. Ide kell tehát ugrani a baloldal végéről, vagyis ez a sor kapja az L4 címkét. Megint két ágat kell egyesíteni, használjuk erre a száml2-t, mely inicializálását írjuk a program elejére.

Végül pedig már csak az u7 maradt, és kész is van a programunk.

Nézzünk most meg egy másik példát! Írjuk le a következő precedenciagráfot fork-join utasítások segítségével!



A problémát most az okozza, hogy az U1 utasítás után a gráf *három* ágra szakad, míg a fork utasítással csak *kétfelé* való elágaztatás lehetséges. Megoldás: *két* fork utasítás használata. Az elsővel a gráfot kétfelé ágaztatjuk, míg a másodikkal az egyik ágat (a mi megoldásunkban az elsőt) bontjuk további két részre. Figyeljük meg azt is, hogy mivel az U5 utasítás előtt 3 ág találkozik, a join utasítás *c* számlálójának kezdetben 3-as értéket adunk!

```
        c:=3;
        U1;
        fork L1;
        fork L2;
        U2;
        goto L3;
L2:     U3;
        goto L3;
L1:     U4;
L3:     join c;
        U5;
```

A fork-join utasításpár előnye, hogy segítségükkel bármilyen precedenciagráf leírható – tehát „ereje" megegyezik a precedenciagráféval – viszont a sok ide-oda ugrálás miatt eléggé

áttekinthetetlen programkódot kapunk, amely módosítása, illetve a benne lévő esetleges hibák megtalálása és javítása embert próbáló feladat.

Jó lenne találni valami olyan leírásmódot, amely áttekinthetőbb eredményt ad!

8.4 Parbegin - parend utasításpár

A **parbegin-parend** (bizonyos programozási környezetekben *cobegin-coend*) utasítások strukturált programozási lehetőséget nyújtanak a konkurencia leírásához. A parbegin-parend utasításpár tulajdonképpen konkurensen végrehajtható folyamatok (példáinkban utasítások) zárójelezését jelentik. Az utasításpár használata:

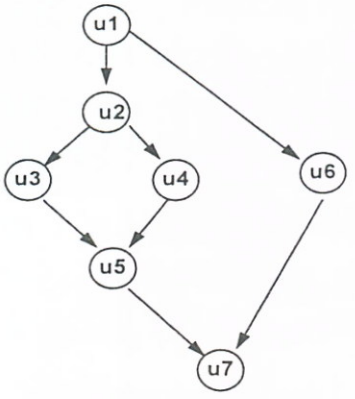
```
parbegin u1; u2; ... parend;
```

Ez azt jelenti, hogy a parbegin és a parend *között* felsorolt u1, u2, ... utasítások *egymással párhuzamosan végrehajthatók.*

Nézzük meg példaként az utolsó fork-join példánál már látott precedenciagráf leírását parbegin-parend utasítások segítségével. A megoldás nagyon egyszerű, hiszen tulajdonképpen éppen az ilyen jellegű gráfok leírásához „találták ki" a parbegin - parend utasításokat:

```
U1;
parbegin
      U2;
      U3;
      U4;
parend;
U5;
```

Nézzünk meg egy kicsit bonyolultabb példát, azt a gráfot, amit a fejezet első ábráján láthattunk!



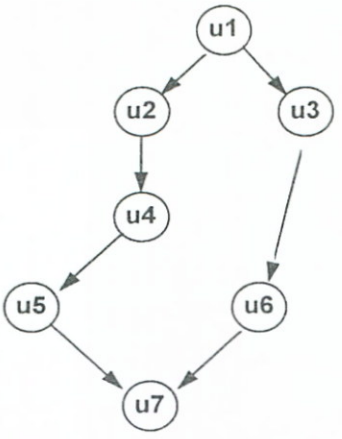

```
u1;
parbegin
   begin
      u2;
      parbegin
         u3;
         u4;
      parend;
      u5;
   end;
   u6;
parend;
u7;
```

8.5 ábra A parbegin-parend utasítások használata

Nézzük meg most is, hogyan született meg az eredményünk! Az első utasítás az u1, majd utána két részre szakad a gráf, tehát a párhuzamosan végrehajtható ágak programjait parbegin és parend közé kell tenni. A baj csak az, hogy az u6-tal párhuzamosan most nem egy utasítás, hanem az u2, u3, u4 és u5 utasítás*csoport* hajtható végre. A megoldás a - például a PASCAL nyelvből már jól ismert - **begin - end** utasítások használata. (Emlékeztetőül: a begin - end arra jó, hogy olyan helyeken, ahol csak egy utasítás állhat, elhelyezhessünk egy utasításcsoportot is. A begin - end közé tetszőlegesen komplikált utasítássort tehetünk, amelyek „kifelé" egy utasításnak „látszanak". Nekünk most pont erre van szükségünk.) Tehát a begin - end közé kell leírni azt, ami az u6-tal párhuzamosítható. Ez a rész az u2-vel kezdődik, majd a gráf ismét kétfelé ágazik. De ez a két ág már csak egy-egy utasítást tartalmaz, ezért most egy parbegin - parend között egyszerűen felsorolhatjuk az u3 és u4 utasításokat, majd következik az u5 utasítás és itt ér véget az u6-tal párhuzamosítható utasításcsoport, azaz ide kell az end utasítás. Az első parbegint lezáró parend után pedig már csak az utolsó, az u7-es utasítást kell leírni ahhoz, hogy kész legyünk.

Látható, hogy ezzel a megoldással áttekinthetőbb, ide-oda ugrálásoktól mentes kódot kapunk.

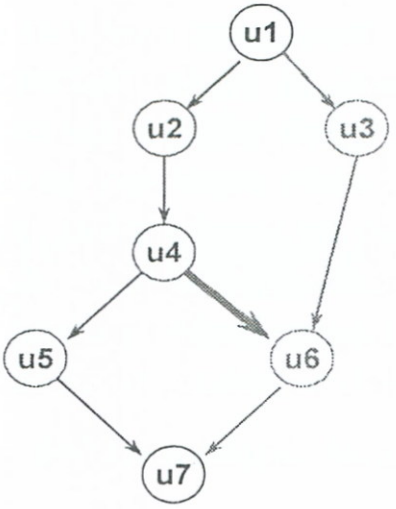
Gyakorlásképpen nézzünk egy másik példát! Azt hisszük, mindenki könnyedén meg tudja oldani önállóan is a feladatot.




```
u1;
parbegin
   begin
      u2;
      u4;
      u5;
   end;
   begin
      u3;
      u6;
   end;
parend;
u7;
```

8.6 ábra A parbegin-parend utasítás használata - gyakorlás

De változtassunk egy kicsit a gráfon! Rajzoljunk be egy nyilat az u4-ből az u6 felé (8.7 ábra)! Most viszont akárhogy küzdünk, nem sikerül a gráf leírása parbegin - parend utasítások segítségével. De nem azért, mert ügyetlenek voltunk, hanem ez a gráf egy olyan típusú gráf, amely *elvileg sem írható le tisztán parbegin - parend használatával.*



8.7 ábra A parbegin-parend utasításokkal leírhatatlan precedenciagráf

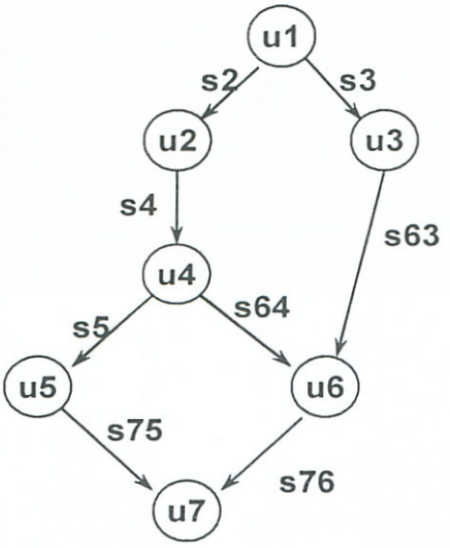
Miért, mi okozza itt a problémát? Az, hogy az u1 után szétváló két ág *nem egy helyen egyesül*, hanem két helyen, u4-u6 illetve u5-u7 között. A parbegin - parend utasítások természetéből fakadóan viszont következik, hogy az egy ponton - a parbeginnél - szétváló ágak egy ponton - a hozzá tartozó parendnél - kell, hogy találkozzanak újra.

Tehát a parbegin - parend utasításokkal szemléletes, áttekinthető kódot kapunk, de sajnos *nem minden precedenciagráf leírásához* alkalmasak, vagyis „erejük" kisebb, mint a precedenciagráfok vagy a fork - join ereje.

Azonban nem kell elkeserednünk! Az a mérnöki életben ritka szerencse ér bennünket, hogy egy teljesen más dologra kitalált eszköz segít a parbegin és a parend problémájának leküzdésében is. Ez az eszköz pedig - az erőforrásokkal foglalkozó fejezetben - már megismert szemaforok illetve P és V primitívek használata. Hogyan használhatók ezek itt?

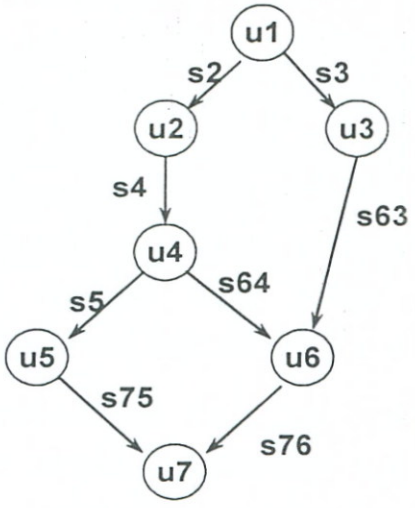
Tételezzük fel, hogy az összes utasításunk párhuzamosan végrehajtható, de tegyünk – az első kivételével – minden egyes utasítás elé egy őt engedélyező szemafort, amelyet induláskor TILOS-ra állítunk. Ezt a szemafort majd akkor engedélyezzük – a V primitív segítségével –, ha az adott utasítást megelőző utasítást végrehajtottuk. Azt pedig, hogy egy utasítás végrehajtását elkezdhetjük-e, a P primitív segítségével tudhatjuk meg, pontosabban, az utasítás végrehajtása előtt a P primitív segítségével addig várakozunk, míg az őt vezérlő szemafort valaki szabadra nem állította, majd a szemafort „lefoglaljuk", azaz újra tilosra állítjuk, végrehajtjuk az utasítást, végül engedélyezzük a következő utasítás(oka)t.

Nézzük ezt meg a fenti példán! Rendeljünk - az első kivételével - az összes utasításhoz egy-egy szemafort! Ezeket a következő ábrán a nyilak fölé írva láthatjuk. A szemaforokat - az áttekinthetőség kedvéért - jelöljük az s betű mellett azzal a számmal, ahányas sorszámú utasítást engedélyezik. Tehát például az s2 szemafor legyen az, amely az u2-es utasítást engedélyezi. Azoknál az utasításoknál, amelyeket egynél több helyről kell engedélyezni - vagyis ahol találkoznak az ágak - a szemaforok nevével azt is mondjuk meg, hogy melyik utasítás *felől* engedélyezik az adott utasítást. Például az s64 jelentése: egy olyan szemafor, mely az u6 utasítást engedélyezi az *u4 végrehajtása után.*



8.8 ábra Szemaforok használata precedenciagráf parbegin/parend utasításokkal való leírásához

Ezek után nézzük meg magát a programot!




```
parbegin
    begin u1; V(s2); V(s3); end;
    begin P(s2); u2; V(s4); end;
    begin P(s3); u3; V(s63); end;
    begin P(s4); u4; V(s5); V(s64); end;
    begin P(s5); u5; V(s75); end;
    begin P(s64); P(s63); u6; V(s76); end;
    begin P(s75); P(s76); u7; end;
parend;
```

8.9 ábra Parbegin-parend program szemaforokkal

Megint nézzük meg, mit jelentenek az egyes sorok!

Először a szemaforokat tilosra állítjuk, ezt végzi el a program elején látható értékadás. Majd parbegin - parend között következnek az utasítások. Mivel egy-egy utasítás végrehajtásánál most több lépést kell megtennünk – általában végrehajtási engedélyre várakozás, utasítás végrehajtás, következő utasítás engedélyezése – ezért van szükség most is a begin - end használatára. Nézzük meg példaként az első két sort!

Az első sorban először végrehajtjuk az u1-et – ez nincs kötve szemafor-feltételhez, hiszen ő a kezdő utasításunk – majd engedélyezzük az őt követő u2 és u3 végrehajtását az s2 és s3 szabadra állításával.

A második sorban először várakozunk, míg az u2-t nem engedélyezik az s2 szabadra állításával, végrehajtjuk az u2-t, majd engedélyezzük az őt követő u4-et az s4 szabadra állításával, stb.

De miért tettük az egészet parbegin és parend közé? A bevezetőben azt mondtuk, hogy azt tételezzük fel, hogy az összes utasítás egymással párhuzamosan végrehajtható. A valóságban azonban nem az utasítások végrehajtása zajlik párhuzamosan, hanem a P primitívek segítségével a *vezérlő szemaforok vizsgálata*. Ez jelenti a módszerünk egyetlen hátrányát: a szemaforok állandó, folyamatos vizsgálata elég sok energiáját leköti a számítógépnek. De ez az az ár, amit fizetnünk kell azért, hogy szemléletes, minden gráfot leírni képes és közvetlenül programozásra alkalmas leírásmódot találjunk a párhuzamosítás leírására.

Hogyan lehet minimalizálni a szemaforok állandó vizsgálatából eredő veszteséget?

Az egyik megoldás a P és V primitívek módosítása. Anélkül, hogy - terjedelmi okokból - a részletekbe bocsátkoznánk, a megoldás elve az lesz, hogy ha egy szemafort tilosnak talál egy folyamat, akkor azt - megfelelő operációsrendszer-hívással a P primitív felfüggeszti („elaltatja"), míg a V primitív – szintén operációsrendszer-hívás segítségével – az adott szemaforra váró „alvó" folyamatokat újraindítja („felébreszti") a szemafor szabadra állításán felül.

A másik betartandó szabály - valós, a példáinkban nézetteknél sokkal nagyobb gráfoknál - pedig az, hogy a gráfoknak csak azon része programozásánál használjunk szemaforokat, ahol ez elkerülhetetlen, ahol viszont nincs rájuk szükség, ott válasszuk a szemaforok nélküli, tisztán parbegin/parend-del operáló leírást. (Hiszen például a 8.6. ábra gráfja leírható lenne szemaforok bevezetésével is, de erre ott, mint láttuk, nincs szükség!)

Összefoglalásul még egyszer megemlítjük, hogy a parbegin/parend utasítások szemaforokkal való kiegészítésével *minden gráf leírható*, tehát megint egy *univerzális eszközhöz* jutottunk.

Végezetül, gyakorlás céljából nézzünk meg egy összetett feladatot:

Adott a következő fork/join leírású programrészlet:

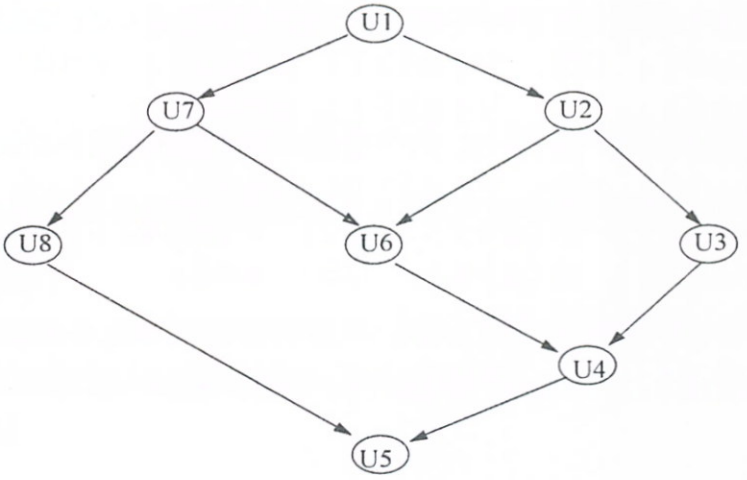
```
      c1:=c2:=c3:=2;
      U1;
      fork L1;
      U7;
      fork L3;
      U8;
      goto L4;
L1:   U2;
      fork L3;
      U3;
L2:   join c1;
      U4;
      goto L4;
L3:   join c2;
      U6;
      goto L2;
L4:   join c3;
      U5;
```

Kérdések:

a.) Rajzoljuk fel a precedenciagráfot!
b.) Mely utasítások hajthatók végre az U3 utasítással párhuzamosan? Indokoljuk meg, miért!
c.) Valósítsuk meg ugyanezt a programot parbegin/parend utasításokkal is!

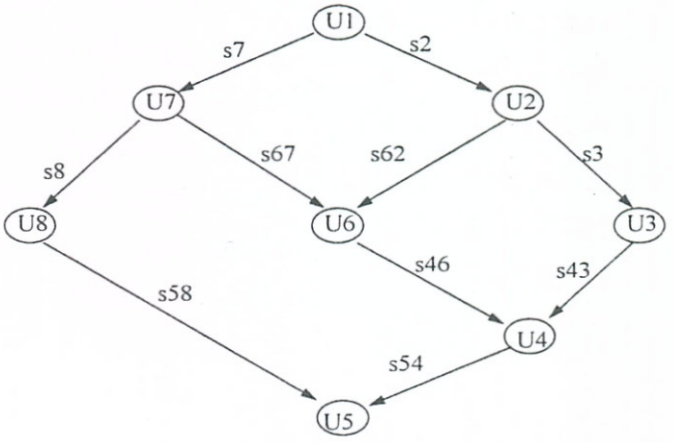
A megoldások:

a.) A precedenciagráf:



b.) A gráfból leolvashatóan U3-mal párhuzamosan a vele „megelőző" illetve „követő" kapcsolatban nem álló utasítások, azaz az U6, illetve az U7(!) és az U8(!) hajtható végre.

c.) Az adott precedenciagráf parbegin/parend utasításokkal közvetlenül nem valósítható meg (hiszen például az U1 után szétváló két ág egynél több helyen, az U6-nál és az U5-nél is egyesül), be kell vezetnünk a megfelelő vezérlő szemaforokat is. A szemaforokat berajzolva a precedenciagráfba:



Magyarázat: például az s2 szemafor az U2 utasítás elvégzését engedélyezi, az s43 szemafor az U4 elvégzését az U3 elvégzése után, míg az s46 az U4 elvégzését engedélyezi az U6 elvégzése után stb.

A parbegin/parend program a berajzolt szemaforokkal:

```
s7:=s2:=s8:=s67:=s62:=s3:=s46:=s43:=s58:=s54:=TILOS;
```

```
parbegin
      begin U1; V(s7); V(s2); end;
      begin P(s7); U7; V(s8); V(s67); end;
      begin P(s2); U2; V(s62); V(s3); end;
      begin P(s8); U8; V(s58); end;
      begin P(s67); P(s62); U6; V(s46); end;
      begin P(s3); U3; V(s43); end;
      begin P(s46); P(s43); U4; V(s54); end;
      begin P(s58); P(s54); U5; end;
parend;
```

Magyarázat: például a *begin P(s46); P(s43); U4; V(s54); end;* sor jelentése:

Az s46 szemafor segítségével megvárjuk, míg az U4 utasítást engedélyezzük az U6 elvégzése után, illetve az s43 szemafor segítségével megvárjuk, míg az U4 utasítást engedélyezzük az U3 elvégzése után. Ha mind a két feltétel teljesül, elvégezhetjük az U4 utasítást, majd ezután az s54 szemafor szabadra állításával engedélyezzük az U5 utasítás elvégzését, stb.

Σ

A párhuzamos programozást ismertető fejezetben azokkal az alapvető programozási technikákkal ismerkedhettünk meg, amelyek lehetővé teszik, párhuzamosan futó folyamatok működésének szinkronizálását. Bemutattuk a szemléletes leírásra kiválóan alkalmas precedenciagráfot, valamint tárgyaltuk a megvalósítására szolgáló fork/join szerkezetet. A nehezen áttekinthető programozási stílus helyett alternatívaként bemutattuk a struktúrált programozáshoz jobban illeszkedő parbegin/parend technikát. A fejezet végén a szemaforok alkalmazása, a parbegin/parend programozási módszer általánosítása kapott helyet.

8.5 Ellenőrző kérdések

?

1. Mutassa be a szinkronizáció három formáját (kölcsönös kizárás, randevú, precedencia)!
2. Mi célt szolgált a precedenciagráf? Melyek előnyei illetve hátrányai?
3. Hogyan készíthető párhuzamos program a fork/join szerkezet segítségével?

4. Mi a fork/join technika alkalmazásának előnye illetve hátránya?
5. Miért áttekinthetőbb a parbegin/parend struktúra, mint a fork/join technika?
6. Mely precedenciagráfok nem írhatók le parbegin/parend szerkezettel?
7. Hogyan szolgálják a szemaforok a parbegin/parend módszer általánosítását?
8. Hasonlítsa össze a párhuzamosan futó folyamatok három leírási módját, a precedenciagráfot, a fork/join és a parbegin/parend szerkezeteket!

Felhasznált irodalom

1. Adamis Gusztáv: Operációs rendszerek példatár, GDF, 1997
2. Ágoston György: Operációs rendszerek - Windows kiegészítő, GDF, 1997
3. Bakos T. – Zsadányi P.: Operációs rendszerek, GDF, Budapest, 1999, ISBN 963-577-098-7
4. Bartók – Nagy - Laufer: UNIX felhasználói ismeretek, OpenInfo, Budapest
5. Benkő – Kiss – Tamás – Tóth: Könnyű a WINDOWS-t programozni? ComputerBooks, Budapest, 1992
6. Benyó B. – Kondorosi K. – Sziray J.: Operációs rendszerek alapjai, Széchenyi István Főiskola, Győr, 1999
7. Benyó B. et al: Operációs rendszerek – Mérnöki megközelítésben, Panem, Budapest, 2000, ISBN 963-545-250-0
8. H. Custer: Inside Windows NT, Microsoft Press, Redmont, 1993, ISBN 1-55615-481-X
9. Cserny László: Mikroszámítógépek, LSI, Budapest, 1997
10. H. M. Deitel: Operating Systems, Addison Wesley, Reading, 1990, ISBN 0-201-18038-3
11. R. Finkel: An Operating Systems Vade Mecum, Prentice Hall, London , 1988, ISBN 0-13-637761-2
12. I. M. Flynn – A. M. McHoes: Understanding Operating Systems, PWS Publishing, 1996, ISBN 0-534-95093-0
13. Jedlovszky Pál: UNIX lépésről lépésre, LSI, Budapest, 1997
14. Kiss - Kondorosi: Operációs rendszerek, Műegyetemi Kiadó, Budapest, 1992
15. Knapp Gábor – Adamis Gusztáv: Operációs rendszerek 1. kiadás, LSI Oktatóközpont, 1999.

16. Kovács Magda: Első lépés a mikroszámítógépek világába, LSI, Budapest, 1997

17. Magyar G. et al: Digitális archívum kézikönyv, kézirat, 2001

18. A. Silberhatz – P. B. Galvin: Operating System Concepts, Addison Wesley, 1998, ISBN 0-201-54262-5

19. A. J. Tanenbaum – A. S. Woodhill: Operációs rendszerek, Panem-Prentice-Hall, Budapest, 1999, ISBN 963-545-1898-X

20. Warford: Computer Science, D.C.Heath & Company, 1991

21. Windows SDK: Programmer's Reference, Microsoft, 1992

22. Windows User's Manual, Microsoft, 1992

ISBN 963-577-251-3