Operációs rendszerek mérnöki megközelítésben

Benyó Balázs

Ez a könyv a Művelődési és Közoktatási Minisztérium támogatásával a Felsőoktatási Pályázatok Irodája által lebonyolított felsőoktatási tankönyvtámogatási program keretében jelent meg.

Copyright © 2000 Panem Kiadó

Tartalom

Támogatók
Előszó
1. Bevezetés
- 1.1. Mit nevezünk operációs rendszernek?
- 1.2. Az operációs rendszerek története
- 1.2.1. Korai rendszerek
- 1.2.2. Batch rendszerek
- 1.2.3. Multiprogramozott rendszerek
- 1.2.4. Időosztásos rendszerek
- 1.2.5. Személyi számítógépek
- 1.2.6. Elosztott rendszerek
- 1.2.7. Valósidejű rendszerek
- 1.2.8. Nyílt rendszerek
- 1.2.9. Napjaink rendszerei
- 1.3. Rendszermodell és rendszerarchitektúra
- 1.3.1. Az operációs rendszer és környezete
- 1.3.2. Funkciók
- 1.3.3. Csatlakozási felületek
- 1.3.4. Számítógép-architektúrák
- 1.3.5. Belső szerkezet
- 1.3.6. Működés

2. Az operációs rendszer mint absztrakt, virtuális gép
- 2.1. Folyamatok és szálak
- 2.2. Folyamatokból álló rendszerek
- 2.2.1. Folyamatok létrehozásának indokai
- 2.2.2. Független, versengő és együttműködő folyamatok
- 2.2.3. Folyamatok születése és halála
- 2.2.4. Folyamatok együttműködése
- 2.2.5. Folyamatok szinkronizációja
- 2.2.6. Folyamatok kommunikációja
- 2.2.7. Holtpont
- 2.2.8. Éhezés
- 2.2.9. Klasszikus konkurens problémák
- 2.2.10. Folyamatokból álló rendszerek leírása nyelvi szinten
- 2.3. Tárak
- 2.3.1. Tárhierarchia
- 2.3.2. A logikai memória

2.3.3. A háttértár-fájlok
2.3.4. Tárak tulajdonságai
2.4. Készülékek és külső kapcsolatok
2.4.1. A külső eszközök fő típusai
2.4.2. Készülékmodell az alkalmazói felületen
2.4.3. Készülékmodell a kezelői felületen
2.5. Védelem és biztonság
2.5.1. Védelem
2.5.2. Biztonság
3. Multiprogramozott operációs rendszerek
3.1. Bevezetés
3.2. A számítógép-architektúra
3.2.1. Az egyprocesszoros von Neumann struktúrájú számítógép-architektúra
3.2.2. Többprocesszoros, szorosan csatolt számítógéprendszerek
3.3. Folyamatkezelés
3.3.1. A folyamatmodell leképezése a fizikai eszközökre
3.3.2. Processzorütemezés
3.3.3. Ütemezés többprocesszoros rendszerekben
3.4. Tárkezelés
3.4.1. A főtár megosztása a folyamatok között
3.4.2. Virtuális tárkezelés
3.4.3. Fájlrendszerek
3.4.3.1. Az állományok tárolása a lemezen
3.5. Készülékkezelés
3.5.1. A kernel B/K-alrendszere
3.5.2. Háttértárak kezelése
3.6. Operációs rendszerek kezelői felülete
3.6.1. Az X Window-rendszer
4. Hálózati és elosztott rendszerek
4.1. Bevezetés
4.2. Hálózati architektúra
4.2.1. Alapfogalmak
4.2.2. A hálózatok topológiája
4.2.3. A hálózatok típusai
4.2.4. A hálózati kommunikáció rétegei
4.2.5. Címzés és forgalomirányítás
4.3. Hálózati jellegű szolgáltatások
4.3.1. Telnet: távoli terminál
4.3.2. FTP: fájlátvitel
4.4. Elosztott szolgáltatások
4.4.1. Jellemzők
4.4.2. Elosztott fájlrendszerek
4.4.3. Folyamatkezelés
4.4.4. Időkezelés és koordináció elosztott rendszerekben
4.4.5. Elosztott rendszerek biztonsági kérdései
5.UNIX

- 5.1. Bevezetés
- 5.2. A UNIX rövid története
- 5.3. Belső szerkezet és működés
- 5.3.1. Szerkezet
- 5.3.2. Folyamatkezelés
- 5.3.3. Ütemezés
- 5.3.4. Szinkronizáció
- 5.3.5. Folyamatok közötti kommunikáció (interprocess communication)
- 5.3.6. Állományrendszer implementációk
- 5.3.7. Teljes folyamatok háttértárra írása (swapping)
- 5.3.8. Igény szerinti lapozás
- 5.4. Hálózati és elosztott szolgáltatások a UNIX-ban
- 5.4.1. A TCP/IP protokoll család
- 5.4.2. A SUN Network File System (NFS)
- 5.5. POSIX
- 5.5.1. Alapfogalmak, felépítés
- 5.5.2. POSIX környezet
- 5.5.3. Hordozható alkalmazások
- 5.5.4. Folyamatkezelés
- 5.5.5. Állománykezelés
- 5.5.6. Jelzéskezelés
- 5.5.7. Terminálkezelés
- 5.6. A LINUX-RENDSZER
- 5.6.1. A Linux fejlődésének állomásai
- 5.6.2. A Linux felépítése és működése

6. A Windows NT operációs rendszer
- 6.1. A Windows NT kialakulása
- 6.1.1. Az NT-vel szemben támasztott követelmények
- 6.1.2. A Windows NT, a Windows 95 és a Windows 98 összehasonlítása
- 6.1.3. Az NT felépítésének fő jellemzői
- 6.1.4. Az NT objektumorientált szemlélete
- 6.2. A Windows NT felépítése
- 6.2.1. HAL
- 6.2.2. Kernel
- 6.2.3. Készülékkezelők (device driverek)
- 6.2.4. Executive
- 6.2.5. Rendszerfolyamatok
- 6.2.6. Szolgáltatások
- 6.2.7. NTDLL.DLL
- 6.2.8. Alrendszerek
- 6.3. A Windows NT belső mechanizmusai
- 6.3.1. Megszakítás- és kivételkezelés
- 6.3.2. Objektumkezelés
- 6.3.3. Szinkronizáció
- 6.3.4. Lokális eljáráshívás
- 6.4. Folyamatok kezelése és ütemezése

6.4.1. A Windows NT folyamatmodellje
6.4.2. Folyamatok kezelése a Windows NT-ben
6.4.3. Szálak kezelése az NT-ben
6.4.4. Szálak ütemezése
6.5. Memóriakezelés
6.5.1. Memória manager felhasználói interfésze
6.5.2. Memóriafoglalás
6.5.3. Osztott elérésű memória
6.5.4. Memóriavédelem
6.5.5. Copy-on-Write
6.5.6. Memória foglalása
6.5.7. A memória mérete
6.5.8. Címtranszformáció
6.6. A Windows NT fájlrendszere (NTFS)
6.6.1. Elvárások az NTFS-sel szemben
6.6.2. Az NTFS további előnyös tulajdonságai
6.6.3. Az NTFS által használt adattípusok, adatszerkezetek
6.6.4. Fájlok elérése NTFS alatt
6.6.5. File ReKord
6.7. Biztonsági alrendszer
6.7.1. A biztonsági alrendszer komponensei
6.7.2. Az objektumok védelme
6.7.3. A biztonsági auditálás
6.7.4. A logon
7.Kérdések, feladatok
7.1. Bevezetés
7.2. Az operációs rendszer mint absztrakt, virtuális gép
7.3. Multiprogramozott operációs rendszerek
7.4. Hálózati és elosztott rendszerek
7.5. UNIX
7.6. Windows NT operációs rendszer
Tárgymutató
A
B
C, CS
D
E, É
F
G, GY
H
I
J
K
L
M
N

O,Ö
P
R
S, SZ
T
U
V
W
X
Z
Irodalom
Az ábrák listája
1.1. ábra. A számítógépes rendszer egymásra épülő rétegei
1.2. ábra. (a)On-line (b) off-line perifériás műveletek blokkvázlata
1.3. ábra. A korai batch programok tipikus szerkezete
1.4. ábra. A spooling technika
1.5. ábra. A multiprogramozás alapelve
1.6. ábra. Az operációs rendszer kontextdiagramja
1.7. ábra. Egyszerű mikrogép architektúrája
1.8. ábra. személyi számítógép architektúrája
1.9. ábra. Szuperszámítógép-architektúra
1.11. ábra. A UNIX és az OS/2 rétegszerkezete
1.13. ábra. Kliens-szerver szerkezetű rendszer
1.14. ábra. Vezérlési szálak különböző típusú rendszerhívások esetén
1.15. ábra. Be-/kivitel lefolyása multiprogramozott rendszerben
1.16. ábra. A parancsértelmező szerkezete
2.1. ábra. Folyamatok együttműködése PRAM szerint működő közös memórián
2.2. ábra. Folyamatok együttműködése üzenetváltással
2.3. ábra. Kommunikáció direkt megnevezéssel
2.4. ábra. Kommunikáció indirekt megnevezéssel
2.5. ábra. Kommunikáció asszimetrikus megnevezéssel
2.6. ábra. Kommunikáció üzenetszórással
2.7. ábra. Erőforrásfoglalási gráf
2.8. ábra. Irányított kört tartalmazó gráf holtponttal és holtpont nélkü
2.11. ábra. Potenciális kérések az erőforrásfoglalási gráfon
2.12. ábra. A termelő-fogyasztó probléma
2.13. ábra. Az írók-olvasók problémája
2.14. ábra. Az étkező filozófusok problémája
2.15. ábra. Adatfolyamok illesztésének problémája
2.16. ábra. Precedenciagráf
3.1. ábra. Von Neumann struktúrájú számítógép egyszerűsített tömbvázlata
3.2. ábra. Tárhierarchia
3.3. ábra. Multiprocesszoros rendszerek logikai tömbvázlata
3.4. ábra. Sorállási diagram
3.5. ábra. Multiprocesszoros rendszerek logikai tömbvázlata
3.6. ábra. Felfüggesztett állapotokat kiegészített állapotmeneti diagram

3.7. ábra. Folyamatleírok láncolása
3.8. ábra. Példa a körülfordulási idő és az időszelet hosszának összefüggésére Round–Robin-rendszerekben
3.9. ábra. Többszintű ütemezés (a) statikus (b) visszacsatolt többszintű sorok
3.10. ábra. Logikai-fizikai címtranszformáció a felhasználói programok többlépcsős feldolgozása során
3.11. ábra. Dinamikus címleképezési módok (a) bázisrelatív címzés (a – a program fizikai kezdőcíme, b – virtuális (logikai) cím, r – fizikai cím), (b) utasításszámláló relatív címzés
3.12. ábra. Késleltetett betöltési módok (a) dinamikus betöltés (b) dinamikus könyvtárbetöltés (c) átfedő programrészek
3.13. ábra. Egypartíciós memória szervezés
3.14. ábra. Többpartíciós memória szervezés
3.15. ábra. Átlapolt tárcsere
3.16. ábra. Címtranszformáció szegmensszervezés esetén
3.17. ábra. Címtranszformáció lapszervezés esetén
3.18. ábra. Címtranszformáció lapszervezés esetén kétszintű laptábla használatával
3.19. ábra. Laphiba kezelésének folyamata
3.20. ábra. Bélády-anomália
3.21. ábra. A multiprogramozás hatása a CPU-kihasználtságra
3.22. ábra. A lokalitás hatása a laphiba gyakoriságra
3.23. ábra. Rendszeregyensúly biztosítása a laphiba-gyakoriság mérése alapján
3.24. ábra. Rendszerstruktúra
3.25. ábra. Mágneslemez-egység felépítése
4.1. ábra. Teljesen és részlegesen összekapcsolt hálózat
4.2. ábra. Hierarchikus és csillag összekapcsolás
4.3. ábra. Gyűrű és kettőzött gyűrű hálózat
4.4. ábra. Sín típusú hálózat
4.5. ábra. A hálózati rétegek és protokollok kapcsolata
4.9. ábra. Bully algoritmus. P_2 kordinátorrá választása P_4 és P_3 meghibásodása után. Majd P_4 koordinátorrá választása annak újraindulása után
4.10. ábra. A gyűrű választási algoritmus működése. A választást most a 19-es csomópont kezdeményezte
5.1. ábra. A hagyományos UNIX-rendszerek belső szerkezete
5.2. ábra. A modern UNIX-ok egy lehetséges szerkezeti felépítése
5.3. ábra. Végrehajtási mód és környezet
5.4. ábra. A folyamatok állapotátmeneti gráfja
5.5. ábra. Láncolt lista egy közbülső elem lefűzése előtt
5.6. ábra. Láncolt lista egy közbülső elem lefűzése közben
5.9. ábra. A folyamatok prioritásának tárolása
5.11. ábra. Call-out függvények láncolt listás ábrázolása
5.12. ábra. Call-out függvények időkerekes ábrázolása
5.15. ábra. A szuperblokkban tárolt inode lista
5.17. ábra. Az open és dup rendszerhívások hatása az állománykezeléssel kapcsolatos adatszerkezetekre
5.18. ábra. A UNIX-állományrendszer könyvtárstruktúrája
5.20. ábra. Az FFS könyvtárszerkezete

5.21. ábra. A vnode szerkezete
5.22. ábra. A vnode-ot is tartalmazó állományrendszer hozzáférés logikai sémája
5.23. ábra. Több állományrendszer használata vfs-sel
5.24. ábra. Folymat háttértárra írása és visszatöltése
5.25. ábra. Tartománybővítés háttértárra írással
5.28. ábra. A tartománymodell és a virtuális memóriakezelést támogató adatszerkezetek kapcsolata
5.29. ábra. A virtuális memóriakezelést támogató adatszerkezetek kereszthivatkozásai
5.30. ábra. A laphibát okozó lap állapotai
5.31. ábra. A módosult adatszerkezetek a lap allokálás után
5.32. ábra. A módosult adatszerkezetek a lap szabad listán történő megtalálása után
5.33. ábra. Egyszerű példa a copy-on-write technika alkalmazására. Az (a) ábra azt az állapotot mutatja, amikor az A folyamat és két gyermeke osztoznak egy lapon, míg a (b) ábrán már az egyik gyermek (B) írás miatt külön lapot használ
5.34. ábra. A hardver, szoftver érvényességi és a szoftverből szimulált hivatkozás bit állapota (a) memóriahivatkozás előtt, (b) memóriahivatkozás után
5.35. ábra. Egy IP-adatcsomag vázlatos felépítése
5.36. ábra. Példa az XDR adatábrázolásra
5.37. ábra. Az RPC működése
5.38. ábra. Az RPC-kérés felépítése
5.39. ábra. Az RPC-válasz felépítése
5.40. ábra. Az NFS működésének sémája
5.41. ábra. Távoli fájl elérése NFS segítségével
5.42. ábra. A legfontosabb POSIX.x szabványok
5.43. ábra. Alkalmazások megfelelősége
5.44. ábra. Trigraph karakterek
5.45. ábra. POSIX jelzéskezelés
5.46. ábra. A Linux-rendszer komponensei
6.1. ábra. Az NT történetének fő állomásai
6.2. ábra. Operációs rendszerek összehasonlítása
6.3. ábra. A Windows NT felépítése
6. 4. ábra. A trapkezelő működése
6.5. ábra. IRQL tábla az Alpha processzornál
6.6. ábra. IRQL tábla az Intel x86 processzoroknál
6.7. ábra. A Windows NT folyamatmodellje
6.8. ábra. Folyamatokhoz és szálakhoz tartozó adatstruktúra
6.9. ábra. Az executive szálblokk felépítése
6.10. ábra. Egy szál állapotai a Windows NT-ben
6.11. ábra. Copy-on-write memórialapok írást megelőzően
6.12. ábra. Copy-on-write memórialapok írást megelőzően
6.13. ábra. A logikai címtér felépítése
6.14. ábra. A fizikai memória mérete különböző memóriamodellek esetén
6.15. ábra. A virtuális cím felépítése
6.16. ábra. A címtranszformáció menete x86-os processzorok esetén
6.17. ábra. Egy lemezírási művelet kiszolgálása
6.18. ábra. A lemezműveletek adminisztrálására használt log fájl szerkezete

6.19. ábra. Réteg szerkezetű device driver struktúra
6.20. ábra. LCN és VCN egymáshoz rendelése
6.21. ábra. NTFS metadata információk
6.22. ábra. A virtuális cím felépítése
6.23. ábra. Egy tipikus file rekord
6.24. ábra. Attribútumok rezidens tárolása
6.25. ábra. Attribútumok nem rezidens tárolása
6.26. ábra. A logonban részt vevő komponensek
A táblázatok listája
1.10.ábra. Nevezetes operációs rendszerek rétegszerkezete
1.12.ábra. A virtuális hardver megvalosító rendszer és közös kernel
2.9.ábra. Többpéldányos erőforrások pillanatnyi allokációja és igénylése
2.10.ábra. Többpéldányos erőforrások maximális igény előrejelzésével
2.17.ábra. Elérési mátrix statikus védelmi tartományokkal
2.18.ábra. Elérési mátrix dinamikus védelmi tartományokkal
4.6.ábra. A centralizált rendszer rétegstruktúrája
4.7.ábra. Elosztott rendszerek réteg struktúrája
4.8.ábra. Az RPC-üzenetek felépítése
5.7.ábra. A UNIX prioritási tartományai
5.8.ábra. Az óramegszakításhoz kötődő ütemezési tevékenységek
5.10.ábra. Ütemezési példa
5.13.ábra. Jelzések
5.14.ábra. Tipikus UNIX-jelzések és értelmezésük
5.16.ábra. A példa állomány lemezblokkjainak elhelyezkedése
5.19.ábra. /etc könyvtár egy lehetséges tartalma
5.26.ábra. A laptábla-bejegyzés által tárolt információk. A bejegyzést a folyamat virtuális címei címzik. Jelölések: Age – öregítés bit(ek), C/W – copy-on-write bit, Mod – módosítás bit, Ref – hivatkozás bit, Val – érvényességi bit,
Protection – védelmi bitek. all
5.27.ábra. A diszk blokk leíró által tárolt információk. A bejegyzést a folyamat virtuális címei címzik

Támogatók

A *Kempelen Farkas Felsőoktatási Digitális Tankönyvtár vagy más* által közreadott digitális tartalom a szerzői jogról szóló 1999. évi LXXVI. tv. 33.§ (4) bekezdésében meghatározott oktatási, illetve tudományos kutatási célra használható fel. A felhasználó a digitális tartalmat képernyőn megjelenítheti, letöltheti, elektronikus adathordozóra vagy papírra másolhatja, adatrögzítő rendszerében tárolhatja. A *Kempelen Farkas Felsőoktatási Digitális Tankönyvtár vagy más* weblapján található digitális tartalmak üzletszerű felhasználása tilos, valamint kizárt a digitális tartalom módosítása és átdolgozása, illetve az ilyen módon keletkezett származékos anyag további felhasználása is.

A jelen digitális tartalom internetes közreadását a Nemzeti Kutatási és Technológiai Hivatal 2006-ban nyújtott támogatása tette lehetővé.

Kutatás-fejlesztési Pályázati és Kutatáshasznosítási Iroda

NKTH
Nemzeti Kutatási és Technológiai Hivatal

Előszó

A számítástechnika és informatika minden képzeletet felülmúló sebességű és szélességű terjedése megköveteli olyan tankönyvek megjelenését, amelyek folyamatosan karbantarthatók, és lépést tartanak a mai kor gyorsan változó követelményeivel. Operációs rendszerekkel foglalkozó könyvekből a világpiacon bőség van. A téma szinte már klasszikusnak számít – persze csak az igen fiatal szakterületen belül. A magyar nyelvű szakirodalom ebben a témakörben szegényesebb. Operációs rendszerekről magyar nyelven mindeddig inkább csak egy-egy részterületet lefedő szakkönyv, vagy pedig egy-egy felsőoktatási intézmény speciális – szűkebb – igényét kielégítő jegyzet került kiadásra. Pedig a témakör fontossága aligha vonható kétségbe. Nehezen képzelhető el, hogy egy számítástechnikával vagy informatikával kapcsolatba kerülő mérnöknek ne legyenek operációs rendszerekre vonatkozó általános, és a legelterjedtebben használt konkrét rendszerekre vonatkozó speciális ismeretei. Ebből következik, hogy a témakörnek az informatikai képzések tanterveiben megfelelő súllyal jelen kell lennie, és általában jelen is van.

Ez a tankönyv azzal a céllal született, hogy a fenti hiányt valamelyest pótolja. Ennek megfelelően igyekeztünk áttekintést adni mind a „klasszikus”, mind az „elosztott” operációs rendszerek alapvető céljáról, működési elveiről, funkcióiról, azok lehetséges megvalósítási módjairól. Az elméleti eredmények mellett hangsúlyt fektettünk ezek megvalósítási lehetőségeire is, így a gyakorlati megvalósítások és eredmények bemutatását két olyan konkrét operációs rendszeren – a UNIX és Windows NT operációs rendszereken – keresztül taglaljuk, melyek napjaink, illetve a közeljövő várhatóan legelterjedtebb és legnépszerűbb operációs rendszerei lesznek.

A Budapesti Műszaki Egyetem Villamosmérnöki és Informatikai Kar műszaki informatika szakán „Operációs rendszerek” című tantárgy az önálló szak indulásától – 1991–1992-től – szerepel alaptárgyként, de a témakör már sokkal korábban megjelent a villamosmérnökök képzési programjában. A szerzők mögött tehát jelentős oktatási tapasztalat áll. Ugyanakkor az „Operációs rendszerek” c. tantárgy keretében felölelt anyag – a szakterület kihívásának megfelelően – az 1997/98-as tanévtől kezdődően megújult, kibővült a korábbiakhoz képest. Könyvünk alapját e tárgy tananyaga képezi. A tankönyv nem követi pontosan az előadássorozat menetét, számos helyen bővebb annál, illetve az előadásokon el nem hangzó témákat is tárgyal, míg más témák, melyek az előadásokon részletesebben szerepelnek, nem, vagy csak érintőlegesen jelennek meg a könyvben. Igyekeztünk az elmúlt években elért néhány fontos eredményt is bemutatni, azonban a témakör sokrétűsége és gyors fejlődése miatt ez messzemenően nem teljes.

A könyv szemléletét és a tárgyalás mélységét a tárgy képzésben elfoglalt helye és szerepe határozta meg. A tárgyból következő leíró jelleget a mérnöki konstruktivitással úgy próbáltuk összeegyeztetni, hogy igyekeztünk megmutatni a miérteket is. Így a felmerülő problémák mellett a lehetséges válaszokat is elemezzük, majd az esettanulmányok konkrét megoldásait elhelyezzük a tervező elvi mozgásterében. Ugyancsak a mérnöki modellalkotó szemléletet kívánjuk erősíteni a virtuális gép és a megvalósítások tárgyalásának szétválasztásával.

A tankönyv hét fejezetből áll. A Bevezetés (1. fejezet) az operációs rendszerek funkcióit, típusait, illetve felépítését ismerteti. A fejezetben igyekszünk rámutatni a hardver-szoftver fejlődés kölcsönös és folyamatos egymásra hatására. Itt alapozzuk meg azt a rendszermodellt, valamint a hozzá szorosan kapcsolódó rendszerarchitektúrát is, amelyre építve a későbbiekben a működést részletesen tárgyaljuk.

A 2. fejezet az operációs rendszerek absztrakt, virtuális gép megközelítését fejti ki. A fejezetben a fő hangsúly az operációs rendszerek egyik alapfogalmára – a folyamatokra – továbbá a szálakra és a folyamatokból álló rendszerek alapkérdéseire (folyamatok együttműködése, szinkronizáció, kommunikáció, holtponthelyzetek, védelem, biztonság) helyeződik.

A 3. fejezet az absztrakt fogalmak klasszikus, egyprocesszoros rendszeren történő megvalósítását tárgyalja. A multiprogramozott operációs rendszerekben alapvető folyamatkezelés (folyamatállapotok, állapotátmenetek, ütemezés), tárkezelés (társzervezés, címek kötése, virtuális tárkezelés), háttértárkezelés (szervezés, kezelés, adattárolás biztonsága) és állománykezelés kérdéseit részletezi.

A hálózati és elosztott rendszerek témájával a 4. fejezet foglalkozik. A hálózatok alapfogalmainak, a hálózattopológiák és kapcsolatok alaptípusainak, valamint a név- és címkezelés legfontosabb kérdéseinek bemutatása után az elosztottságnak a minőségi jellemzők, tervezési szempontok, állományrendszer, folyamat- és időkezelés és biztonság terén jelentkező hatásait tárgyaljuk. Részletes ismertetésre kerül egy kliens–szerver-modellre épülő hitelesítő rendszer is.

Az 5. fejezet a UNIX operációs rendszerről ad áttekintést. A korábban megismert elvek és legfontosabb funkciók megvalósításának bemutatása során fő célunk nem a pillanatnyilag ismert legfrissebb UNIX-implementáció pontos ismertetése, hanem inkább a megvalósítások mérnöki elemzése volt. Kitérünk a fejlesztés gondolatmenetére, az annak során felmerülő problémákra, azok lehetséges megoldási lehetőségeire és az implementációk által megvalósított elvekre, azok előnyeire illetve hiányosságaira.

A 6. fejezet a Microsoft cég újgenerációs operációs rendszerével, a Windows NT-vel foglalkozik. Ez, a korábbi Windows-rendszerektől struktúrájában is különböző operációs rendszer egy új technológiát (NT – New Technology) képvisel és ténylegesen új, korszerű megoldásokat alkalmaz.

A tankönyv utolsó, 7. fejezete ellenőrző kérdéseket és feladatokat tartalmaz az ismertetett anyag egészére vonatkozóan. Elsősorban a mérnökhallgatókra gondoltunk, az anyag elsajátításának ellenőrzését igyekeztünk megkönnyíteni a számukra.

Mindamellett, hogy a könyvet elsősorban tankönyvnek szántuk, úgy véljük, szélesebb olvasótábor számára is bátran ajánlhatjuk. Az informatikus szakembereknek, de az informatika eszközeivel megismerkedni szándékozó szakmán kívülieknek is ígérhetünk érdeklődésüket kielégítő mondanivalót. Az olvasáshoz némi számítástechnikai előismeret (számítógépek felépítése, programozás) hasznos, de nem elengedhetetlen.

A könyv szándékaink ellenére nyilván számos hiányosságot és hibát tartalmaz. Reméljük, hogy a témakört tanuló hallgatók ezeket észrevéve hozzásegítenek a kijavításukhoz.

A könyv megszületéséhez való hozzájárulásukért köszönet illeti mindazokat, akik korábban publikálták eredményeiket, és ezzel lehetővé tették, hogy az igen bőséges nemzetközi irodalomra támaszkodva alakítsuk ki azt a szemléletet és megközelítésmódot, amit oktatási céljaink eléréséhez a legmegfelelőbbnek tartottunk. Hallgatóinknak is köszönjük a konzultációk során feltett kérdéseket, amelyek eredményeként az anyag sokat csiszolódott.

Ugyancsak köszönet illeti a tárgy oktatásában résztvevő tanszékek vezetőit, Arató Pétert és Péceli Gábort a munkafeltételek biztosításáért, Vajk Istvánt az önálló tárgy első változatának kidolgozásában való közreműködéséért, Szigeti Szabolcsot és Mann Zoltán Ádámot értékes anyaggyűjtő munkájukért, a Panem Kiadó munkatársait a konstruktív és rugalmas munkakapcsolat kialakításáért, és végül, de nem utolsó sorban családtagjainkat, hogy elviselték időnként jelenlétünket, időnként hiányunkat az írás során.

A könyv megszületéséhez nagymértékben hozzájárult az a támogatás, amelyet a Művelődésügyi Minisztérium Felsőoktatási Tankönyvtámogatási Pályázata nyújtott.

Budapest, 1999. október 31.

Kóczy Annamária és Kondorosi Károly

szerkesztők

1. Bevezetés

Tartalom

1.1. Mit nevezünk operációs rendszernek?
1.2. Az operációs rendszerek története
  1.2.1. Korai rendszerek
  1.2.2. Batch rendszerek
  1.2.3. Multiprogramozott rendszerek
  1.2.4. Időosztásos rendszerek
  1.2.5. Személyi számítógépek
  1.2.6. Elosztott rendszerek
  1.2.7. Valósidejű rendszerek
  1.2.8. Nyílt rendszerek
  1.2.9. Napjaink rendszerei
1.3. Rendszermodell és rendszerarchitektúra
  1.3.1. Az operációs rendszer és környezete

1.3.2. Funkciók
1.3.3. Csatlakozási felületek
1.3.4. Számítógép-architektúrák
1.3.5. Belső szerkezet
1.3.6. Működés

Az operációs rendszerek alapvető részét képezik bármelyik számítógépes rendszernek. Egy operációs rendszer olyan program, amely közvetítőként működik a felhasználó és a számítógép hardvere között. Elsődleges célja, hogy biztosítsa azt a környezetet, amelyben a felhasználó kényelmesen és hatékonyan végre tudja hajtatni a saját programját. A másodlagos cél a számítógép hardverjének minél hatékonyabb használata. Mindehhez az is szükséges, hogy az operációs rendszerek megfelelő szolgáltatásokat nyújtsanak a programoknak és a programok felhasználóinak azzal a céllal, hogy a programfejlesztési feladat könnyebbé váljék.

A számítógépes rendszerek lényegében három fő komponensre oszthatók: hardver, rendszerprogramok (melyek egészét vagy részét képezi az operációs rendszer), valamint alkalmazói programok.

A hardver – a központi feldolgozó egység (CPU: Central Processing Unit), a memória, valamint a bemeneti/kimeneti (I/O: Input/Output) egységek – szolgáltatják az alapvető számítási erőforrásokat. Az alkalmazói programok a felhasználók számára segítik elő saját feladataik megoldását. A rendszerprogramok leglényegesebb része az operációs rendszer, amely a számítógépes rendszer helyes működését biztosítja, vagyis vezérli és koordinálja a hardver felhasználását a különböző alkalmazói programok között. Az operációs rendszer pontos meghatározása nem egységes, a különböző iskolák más-más megközelítései révén eltérő nézetek terjedtek el.

A fejezet további részében először az operációs rendszer lehetséges értelmezéseit elemezzük. A következő alfejezetben az operációs rendszerek fejlődéstörténetét vizsgáljuk részletesebben. Végül pedig az 1.3. alfejezetben a számítógép rendszerek modelljével és architekturális kérdéseivel foglalkozunk.

1.1. Mit nevezünk operációs rendszernek?

Van egy olyan mondás, miszerint „Szoftver nélkül a számítógép csak egy össze-vissza gubancolt ócskavas-halom”. Bárki, aki dolgozott már számítógéppel – bármilyen rövid ideig is – tisztában van ennek a mondásnak az igazságával. A legtöbb ember számára a „**szoftver (SW: Software)**” szó **alkalmazói (application)** vagy másszóval **felhasználói (user) szoftver**-t jelent, azaz olyan programokat, amelyek segítségével tervezhetünk egy űrhajót, kiszámíthatjuk az adónkat, mozijegyet vásárolhatunk vagy játszhatunk. Ez azonban az igazságnak csak egy része.

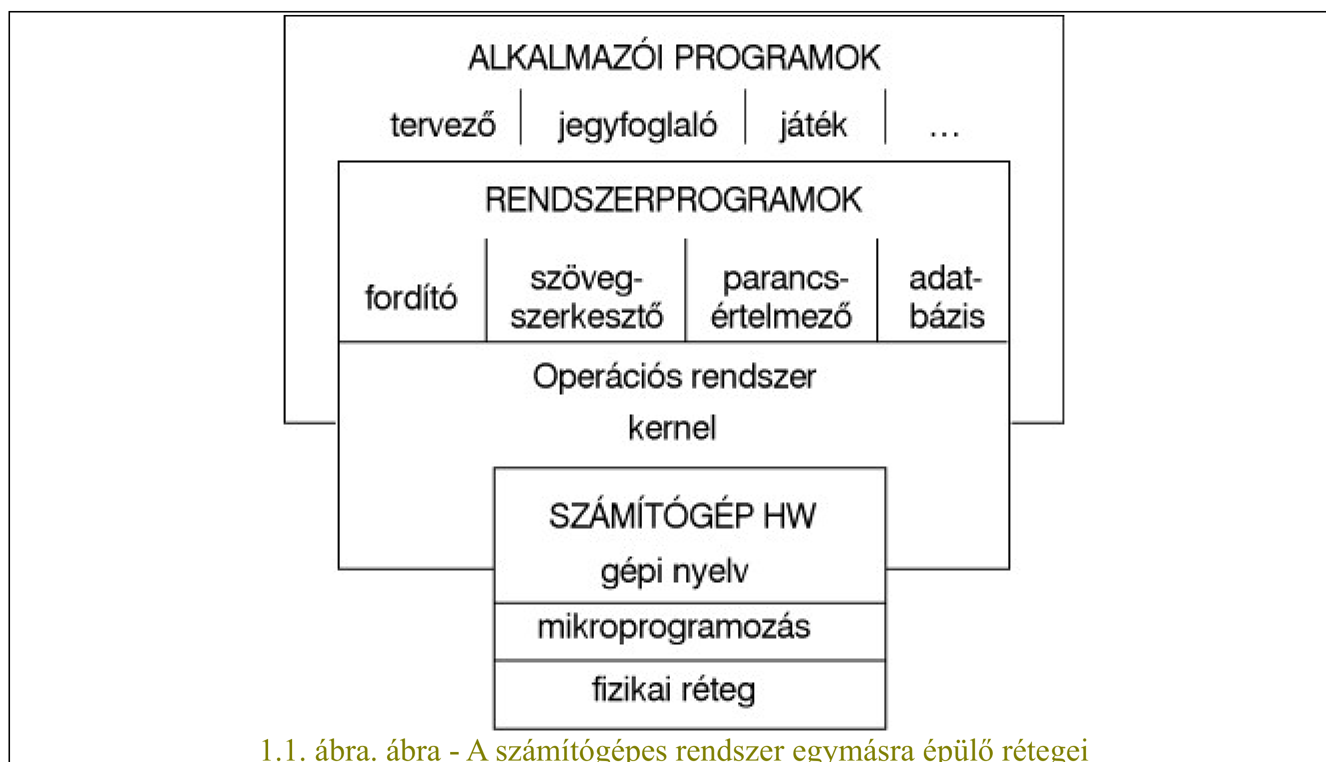
Képzeljük el, hogy odaültetnek bennünket egy olyan számítógép elé, amely tényleg csak a puszta hardverből **(HW: Hardware)** áll, semmilyen szoftver nem található rajta: Hogyan lennénk képesek az említett feladatokat megoldani? De még az olyan egyszerű dolgok is gondolkodást igényelnének, mint két szám összeadása, nem is beszélve a szorzásról, osztásról.

Szerencsére azonban a helyzet a valóságban – legalábbis a számítógép történetének nagyobbik felében – ennél sokkal jobb, mert a szoftvernek van egy általában nem mindig látott, de mindig használt **rétege (layer)**, amely a felhasználó (itt ne csak emberi felhasználókra gondoljunk, a rendszer szempontjából felhasználó lehet valamilyen gép vagy akár egy másik számítógép is) és a hardver között helyezkedik el, olyan környezetet teremtve, amely lehetővé teszi az alkalmazó számára a számítógép *kényelmes* és *hatékony* használatát. Ezt hívják **operációs rendszernek (operating system)**.

Az operációs rendszer fogalmát nemigen tudjuk pontosan meghatározni, különböző irányzatok más-más „szigorúsággal” állnak a kérdéshez. Abban mindenki egyetért, hogy az operációs rendszer része a **mag (kernel)**, vagyis az a program, amely feltétlenül szükséges a gép működéséhez és amely közvetlenül vezérli azt. Régebben ezt úgy is meg lehetett közelíteni, hogy a kernel az a program, amely *állandóan fut*. A **dinamikus kernel** megjelenése óta azonban ez utóbbi definíció már nem állja meg teljesen a helyét. A másik véglet szerint az operációs rendszerbe tartoznak azok a – tulajdonképpen felhasználói – **rendszerprogramok (system program)** is, amelyek a gép általános felhasználását segítik, például a fordítók, szövegszerkesztők, parancsértelmező.

E könyv a nézetek különbözőségét figyelembe véve megpróbál inkább az operációs rendszer két szintjéről beszélni. A belső, szűkebben értelmezett szint a kernelt takarja, míg egy tágabb szint alatt a köznapi értelemben vett operációs rendszert értjük – azaz mindazokat a programokat, amelyeket egy felhasználó a gép vásárlásakor megkap. Lényeges különbség a kettő között, hogy a kernel *el van rejtve*, „*védve*” *van* (hardvervédelem) a felhasználó elől, azaz ehhez a felhasználó nem fér hozzá, nem tudja módosítani és – elrontani. Bizonyos rendszerprogramokat célszerű, ha elérhet a felhasználó, mert így azokat úgy tudja alakítani, hogy használatuk számára a legkényelmesebb legyen. A hardver- és szoftverrétegek egy lehetséges egymásra épülését az 1.1. ábra mutatja.

Az alkalmazói programok tehát a felhasználó problémáit oldják meg, míg a rendszerprogramok a számítógép saját tevékenységét irányítják. Ezen belül az operációs rendszer *felügyeli* az összes **erőforrást (resource)**, továbbá *biztosítja* azt *a környezetet*, amelyben az alkalmazói programok futhatnak. *Vezérli* mind a programok, mind az erőforrások működését, valamint meggátolja a számítógép hibás felhasználását.



1.1. ábra. ábra - A számítógépes rendszer egymásra épülő rétegei

A hardver *hatékony* használatának biztosítása tekintetében az operációs rendszer feladatát – erőforrás kezelés – „alulról”, a hardver oldaláról fogalmaztuk meg (ezt hívják *„bottom-up view”*-nak). Erőforrás alatt mind hardver-, mind pedig szoftvererőforrásokat értünk (CPU-idő, memória, perifériák, állományok, adatok stb.), melyeket a felhasználók céljaik elérése érdekében használni szeretnének. A sok, sokszor ütköző vagy ellentmondó igény között az operációs rendszernek kell döntenie és az igényelt erőforrásokat felhasználóhoz rendelnie úgy, hogy *hatékony, biztonságos és igazságos működést biztosítson*.

A *kényelmes* használat biztosítása tekintetében – megkönnyíti a programozást – „felülről”, a felhasználó oldaláról *(„top-down view”)* közelítünk az operációs rendszerhez. Az operációs rendszer önmagában nem végez (kívülről nézve) hasznos tevékenységet, de olyan környezetet biztosít, amelyben a felhasználói programok képesek hasznos munka végzésére. Ezt úgy is tekinthetjük, hogy az operációs rendszer egy olyan réteget képez a hardver fölött, amely *elrejti* annak körülményességét és bonyolultságát a programozó elől, és *kibővíti* a számítógép hardver szolgáltatásait. Így a felhasználó egy sokkal kellemesebb tulajdonságokkal rendelkező **virtuális gépet (virtual machine, extended machine)** lát.

Manapság a *felhasználóbarát (user friendly)* virtuális gép, illetve környezet biztosítása egyre növekvő fontosságú alapkövetelmény minden operációs rendszerrel szemben. Hosszú időt igényelt azonban, amíg eljutottunk ide. A következő fejezetben ezt a történeti fejlődést tekintjük át röviden, hangsúlyozva azt, ahogyan a hardver- és szoftvertechnikai előrelépések kölcsönösen egymásra hatottak, kedvező lökést adva a továbbfejlődésnek.

1.2. Az operációs rendszerek története

Az operációs rendszerek történetét a számítógépek történetével párhuzamosan, azzal szoros összefüggésben tárgyaljuk. A fejlődés menetének felvázolása során vegyük észre, hogy minden kornak megvolt a „legégetőbb” problémája, amely mellett a többi – bármilyen komoly volt is – eltörpült. Amint azonban sikerült megoldást találni erre, azonnal a következő időszerű nehézség megoldását célozták meg. Így például, mint látni fogjuk az alábbiakban, az első generációs (elektroncsöves) gépek legnagyobb gondja a megbízhatatlanság volt, amely mellett nemigen számított a rossz gépkihasználás. Amint azonban megjelentek a második generációs „megbízható” (tranzisztoros) gépek, a gépidő jobb kihasználásának kérdése került előtérbe. Vagy egy másik példát említve, amint napjainkhoz közeledve eljutottunk ahhoz, hogy a nagysebességű, megbízható gépek kihasználása „optimálissá” vált, majd bizonyos eszközök olcsósága miatt egyre kevésbé volt fontos azok minél jobb kihasználása, a felhasználó kényelmének kiszolgálása lett a „legsúlyosabb” megoldandó probléma.

1.2.1. Korai rendszerek

Charles Babbage (1792–1871) nevéhez fűződik az a szerkezet, amely tulajdonképpen a digitális számítógép ősének tekinthető. Az általa „elemző gépnek” (analytical engine) nevezett mechanikus szerkezetet azonban akkor soha nem sikerült megépíteni, mert abban az időben nem tudták az alkatrészeket megfelelő pontossággal előállítani. (Az eredeti tervek alapján megépítették, és 1991-ben bemutatták a gépet, amelyik azóta a London Science Museumban látható.)

Hatalmas előrelépést jelentett, hogy a II. világháború befejeződése környékén több egyetemen, illetve kutatóintézetben sikerült működő számítógépet építeni – például az Egyesült Államokban H. Aiken (Harvard), J. P. Eckert és W. Mauchley (University of Pennsylvania), Németországban pedig K. Zuse nevéhez fűződnek sikerek. Igazán ismertté azonban Neumann János (1903–1957) magyar származású, Amerikában élt matematikus és H. H. Goldstine irányítása alatt épült (1945) „első” számítógép vált. Az EDVAC (Electronic Discrete Variable Calculator, diszkrét változós elektronikus számológép) hosszú ideig mintául szolgált a többi tárolt programú, elektronikus programvezérlésű számítógép építéséhez. A von Neumann által megfogalmazott alapelv (mely szerint a programokat és adatokat ugyanabban a tárban, ugyanolyan formában tárolták és csak környezetük alapján különböztették meg) miatt nevezik ma is a hagyományos számítógépeket von Neumann struktúrájú gépeknek.

Az első számítógépek 1500 szorzás vagy 15 000 összeadás elvégzésére voltak képesek másodpercenként legfeljebb 12 jegyű decimális számnak megfelelő bináris számokkal. A tervezést, építést, programozást, működtetést, javítást egyetlen csapat végezte. A gépek elektroncsövesek voltak, ami teljesen megbízhatatlanná tette a működést, mivel a nagyjából 20 000 elektroncső közül nagy valószínűséggel kiégett valamelyik az egyszerű programok futása alatt. Programnyelveket nem ismertek ebben az időben, így mindent gépi kódban oldottak meg. A programokat dugaszolós kapcsolótáblán rögzítették és ezeket csatlakoztatták a számítógépbe. További problémát jelentett, hogy nem ismerték előre a futáshoz szükséges időt, így a gépidőfoglalást becslés alapján végezték, ami azzal járt, hogy – jobbik esetben – kihasználatlan maradt a drága gépidő egy része –

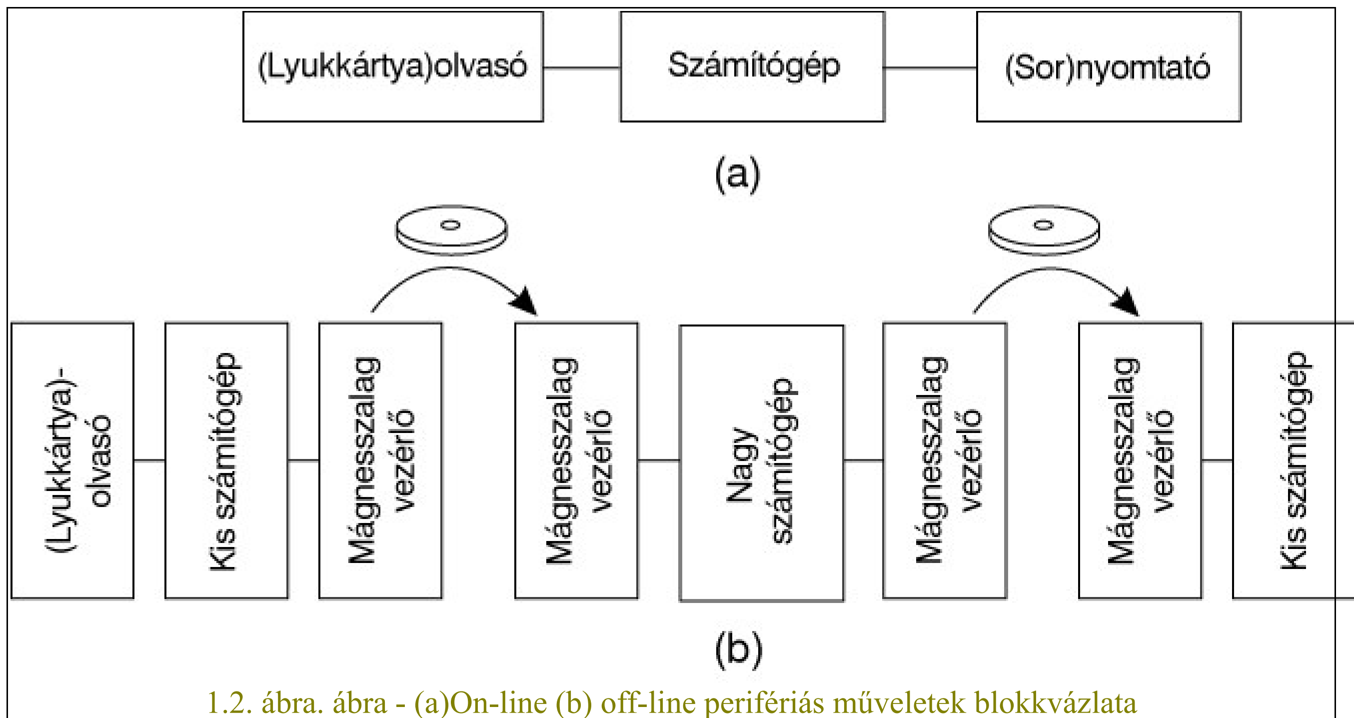
rosszabbik esetben pedig –, az eredmények kiszámítása előtt meg kellett szakítani a futást, és a gépet át kellett adni a következő foglalónak. Technikai előrelépést jelentett, amikor az ’50-es években megjelentek a lyukkártyák.

Ebben az időben számítógép segítségével olyan egyszerű számítási feladatokat oldottak meg, mint szinusz, koszinusz táblázatok kiszámítása.

1.2.2. Batch rendszerek

A számítógépek II. generációját (1955–1965) már tranzisztorokból építették, így megoldódott az eddigi legégetőbb probléma – az elektroncsövek miatti megbízhatatlan működés. Azonban ezek a gépek rettenetesen drágák voltak. Működtetésükhöz speciális légkondicionált géptermek kellettek. A korszakhoz fűződik a külön professzionális operátori csapat megjelenése, amely a gép kiszolgálását és a programok futtatását végezte. A programozás akkori menete szerint a programokat papíron írták (ekkor már ismertek programnyelveket, például assemblereket, Fortrant), majd ezeket lyukkártyán kilyukasztották. Az elkészült lyukkártyákat leadták egy teremben, ahonnan azokat az operátorok vitték a géphez, majd a szükségnek megfelelően betöltötték a fordítót stb. Az eredményeket papírra nyomtatták (**on-line** módon, azaz a beolvasás és az eredmények kinyomtatása nem volt időben szétválasztva a programok futásától), melyeket egy újabb teremben lehetett átvenni. A programok előkészítési ideje tehát roppant nagy volt, a drága gépek drága gépidejének egy jó része azzal telt, hogy az operátorok a gépterem és kiszolgáló termei között sétáltak, miközben maga a számítógép kihasználatlanul állt. Ezért aztán következő fejlődési lépésként a „felesleges” időveszteségek csökkentését tűzték ki célul.

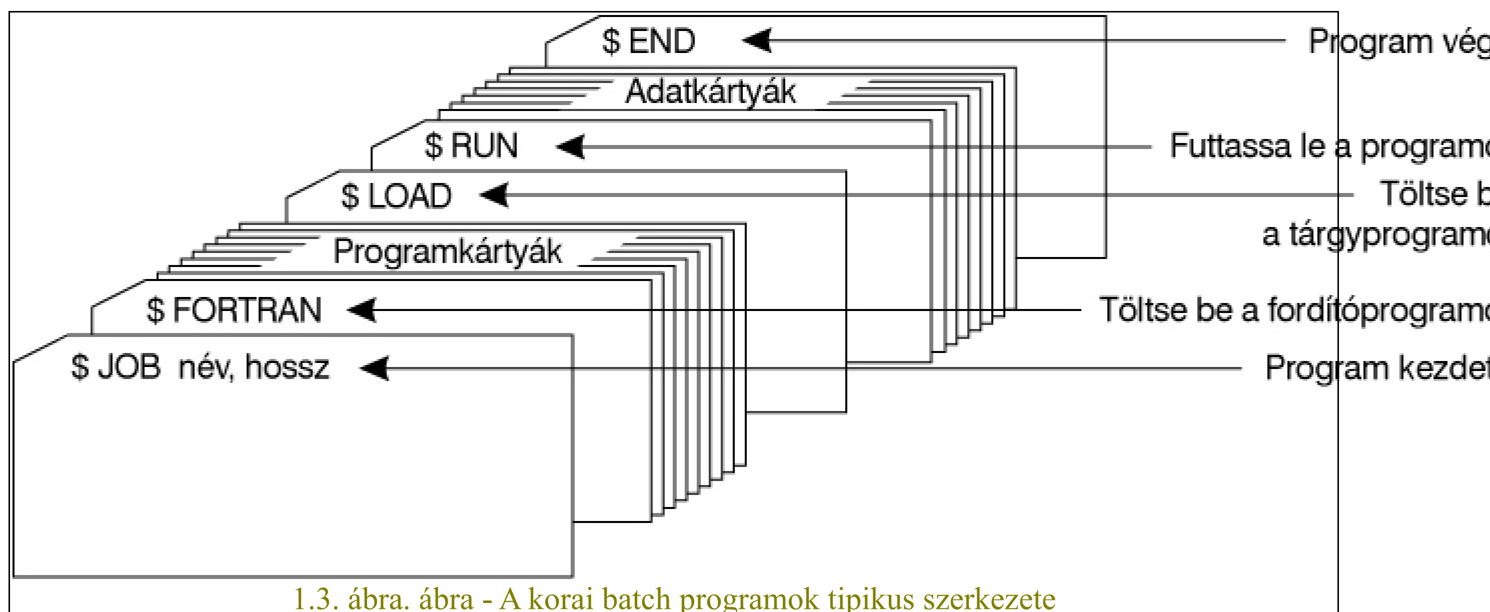
A megoldást a **kötegelt** vagy **batch feldolgozás** megjelenése jelentette, amely szerint az operátorok összeválogatták a „hasonló” munkákat (például mindegyik Fortran fordítót igényelt), majd ezeket egymás után futtatva több korábbi lépést csak egyszer kellett elvégezni. A mágnesszalagok megjelenése (az összeállított köteget a lyukkártyákról egy szalagra másolták) tovább könnyítette és gyorsította ezt a folyamatot. Hasonlóképpen, a jobb gépkihasználást segítette, hogy az on-line perifériás műveletekről áttértek az **off-line perifériás műveletekre** (1.2. ábra), azaz a lyukkártyák beolvasását, valamint az eredmények nyomtatását a programok futtatásától elválasztva, általában egy külön erre a célra kifejlesztett kis számítógép segítségével végezték.



1.2. ábra. ábra - (a)On-line (b) off-line perifériás műveletek blokkvázlata

A programok futása között eltelő „üres” időt (amíg az operátor észrevette, hogy egy munka (job) befejeződött és elindította a következőt) tovább csökkentette, amikor sikerült kifejleszteni egy olyan felügyelő programot – **egyszerű monitort (resident monitor)** –, amely egy-egy munka befejeződése után *automatikusan* beolvasta a következőt és megkezdte annak végrehajtását. *Vagyis megjelentek az első operációs rendszerek*. A jobleírásokat a könnyebb felismerhetőség kedvéért speciális karakterrel kezdődő *vezérlő utasításokkal* egészítették ki, amelyek a monitor számára hordoztak információt. (Így minden jobleírás a munka kezdetét és végét jelző utasítással kezdődött, illetve végződött; külön utasítás adta meg, hogy mikor és milyen fordítóprogramot kell betölteni, mikor lehet a tárgyprogramot betölteni [azaz hol a programrész vége] elkezdődhet a program futtatása stb.)

Az egyszerű monitorok megjelenése egyben azt is jelentette, hogy a korábban egy területként kezelt memóriát a továbbiakban két alapvető részre, egy monitor és egy felhasználói területre osztották.



1.3. ábra. ábra - A korai batch programok tipikus szerkezete

Az off-line perifériás műveletek alkalmazásának legnagyobb előnye abban rejlett, hogy a fő számítógép működése nem függött többé a lassú kártyaolvasó és sornyomtató működésétől, hanem a sokkal gyorsabb mágnesszalag vezérlőtől. Összességében azonban ez a megoldás csak akkor gyorsította a munkák befejeződését, ha nagyjából azonos idejű CPU- és perifériás műveletet tartalmaztak.

Az új technika elterjedése azt is eredményezte, hogy a programok nem az eredeti perifériákat, hanem **logikai B/K-készülékeket** használtak. Ezt hívjuk **készülékfüggetlen** vagy **perifériafüggetlen programozásnak (device independence)**. Szabványos felületű perifériameghajtókat kezdtek alkalmazni egyrészt, hogy biztosítsák, hogy a programok ne vegyék észre a „cserét”, másrészt viszont a programokat átírás nélkül lehetett használni akkor is, ha a rendszerben kicserélték a perifériákat.

A fenti megoldásnak minden előnye mellett hátránya volt, hogy több számítógépet (esetenként hármat vagy akár többet is) igényelt és a szalagok cseréje még mindig nehézkesen, operátori segédlettel történt. A probléma megoldására megalkották az autonóm perifériavezérlőket. Az autonóm perifériavezérlők lehetővé tették a **pufferelést (buffering)**, azaz, hogy a B/K-műveletek a perifériák és a CPU között elhelyezkedő pufferen keresztül hajtódnak végre megszakítás, illetve blokkos átvitel segítségével. Tehát például a beolvasás egy pufferbe történik és a CPU innen veszi az adatot, majd annak feldolgozásával egyidőben újabb perifériás művelet hajtódhat végre *(átlapolt feldolgozás)*.

A *hibakeresés (debugging)* továbbra is komoly nehézségekbe ütközött, ugyanis ezt nyilvánvalóan csak a programozó tudta elvégezni. Mivel azonban nem volt közvetlen kapcsolatban a számítógéppel, ezért off-line módon, a hibás befejeződéskor kiíratott (dumping) memória és regiszter tartalmak segítségével végezte.

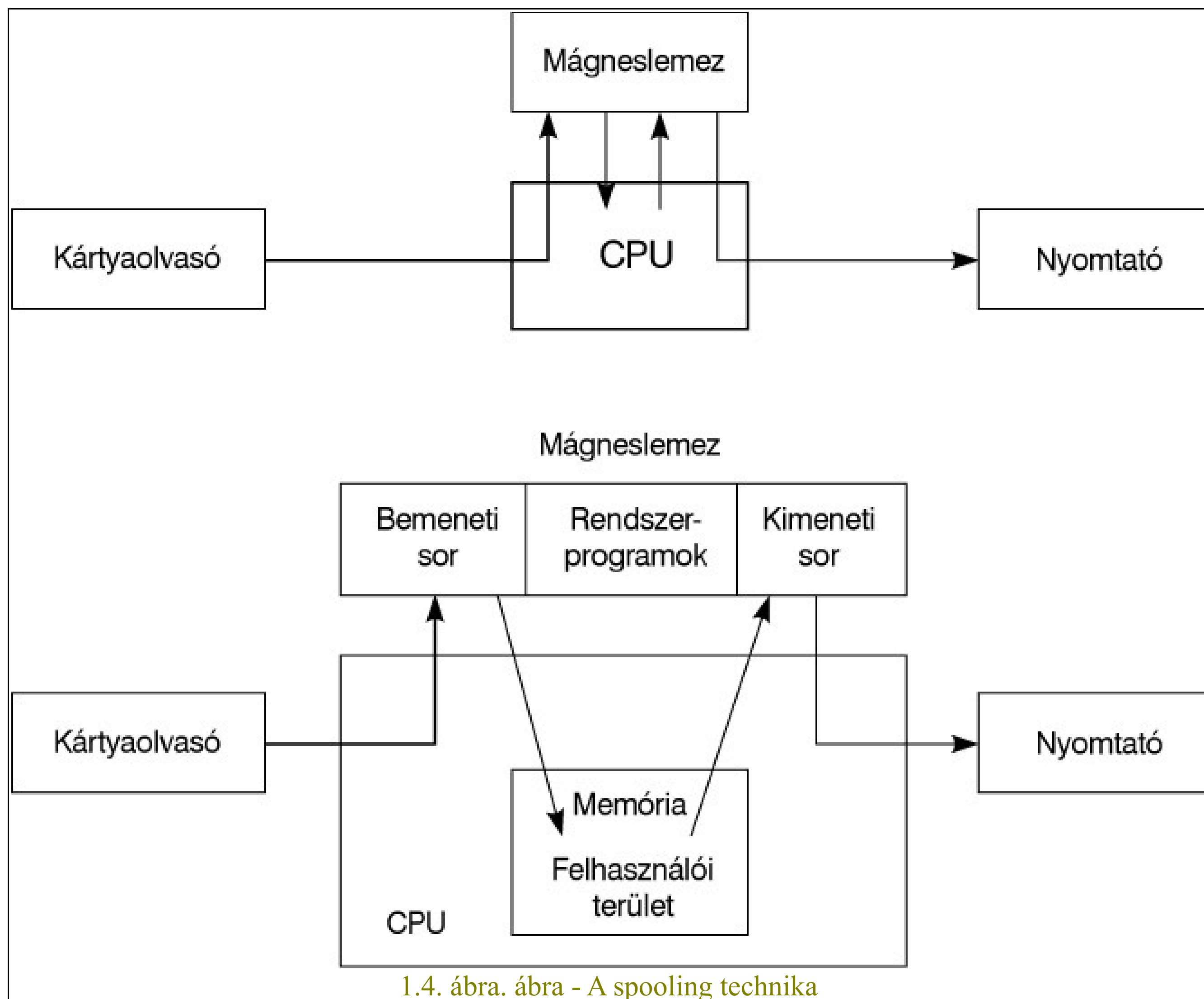
Ebben az időben a számítógép segítségével olyan tudományos és mérnöki számításokat is el tudtak már végezni, mint például parciális differenciálegyenletek megoldása.

A II. generációs gépek hatására a ’60-as évek elejére két teljesen független, egymástól eltérő fejlődési irány alakult ki: szószervezésű nagy tudományos komputerek és karakterszervezésű kis perifériás gépek, ami a fejlesztést igen megdrágította. Következő lépésként tehát ezen kellett segíteni.

A kérdést a számítógépek harmadik generációjánál (1965–1980, hardver: integrált áramkörök) az IBM oldotta meg, a System/360 számítógépcsalád bevezetésével. A család különböző méretű (memória, gyorsaság, perifériák, ár, teljesítmény), de egymással kompatíbilis gépekből állt, így *elvileg* bármely gépre írt szoftver futtatható volt bármelyiken. A gyártmánycsalád és szabványos egységek bevezetése a hardverfejlesztés költségeinek csökkenése mellett egyéb előnyökkel is járt, ugyanakkor azonban nagyon drága és bonyolult operációs rendszert igényelt.

Újra előtérbe került tehát az alapvető kérdés, a költségcsökkentés kérdése. Ehhez komoly lökést jelentett a nagykapacitású, gyors és *véletlen hozzáférésű (random access)* mágnesdobok és mágneslemezek megjelenése. Az ún. **spooling technika (simultaneous peripherial operation on-line)** ezeket a tárakat mint egy hatalmas méretű puffert használja, és egyszerre nemcsak egy, hanem több munkát is lemezre tölt. Ez a megoldás lényegében az off-line perifériás műveletek egyetlen gépen történő megvalósítása. Lehetővé válik a perifériás és CPU-műveletek teljes szétválasztása úgy, hogy *több munka is átlapolódhat*, azaz például az egyik munka végrehajtásával egyidőben egy másik munka beolvasása és egy harmadik eredményeinek kivitele folyhat. Ezzel mind a perifériák, mind pedig a CPU kihasználtsága jelentősen megnőtt (1.4. ábra).

A véletlen hozzáférésű háttértárak megjelenésének legnagyobb jelentőségű következménye azonban a **multiprogramozás** lehetősége volt.

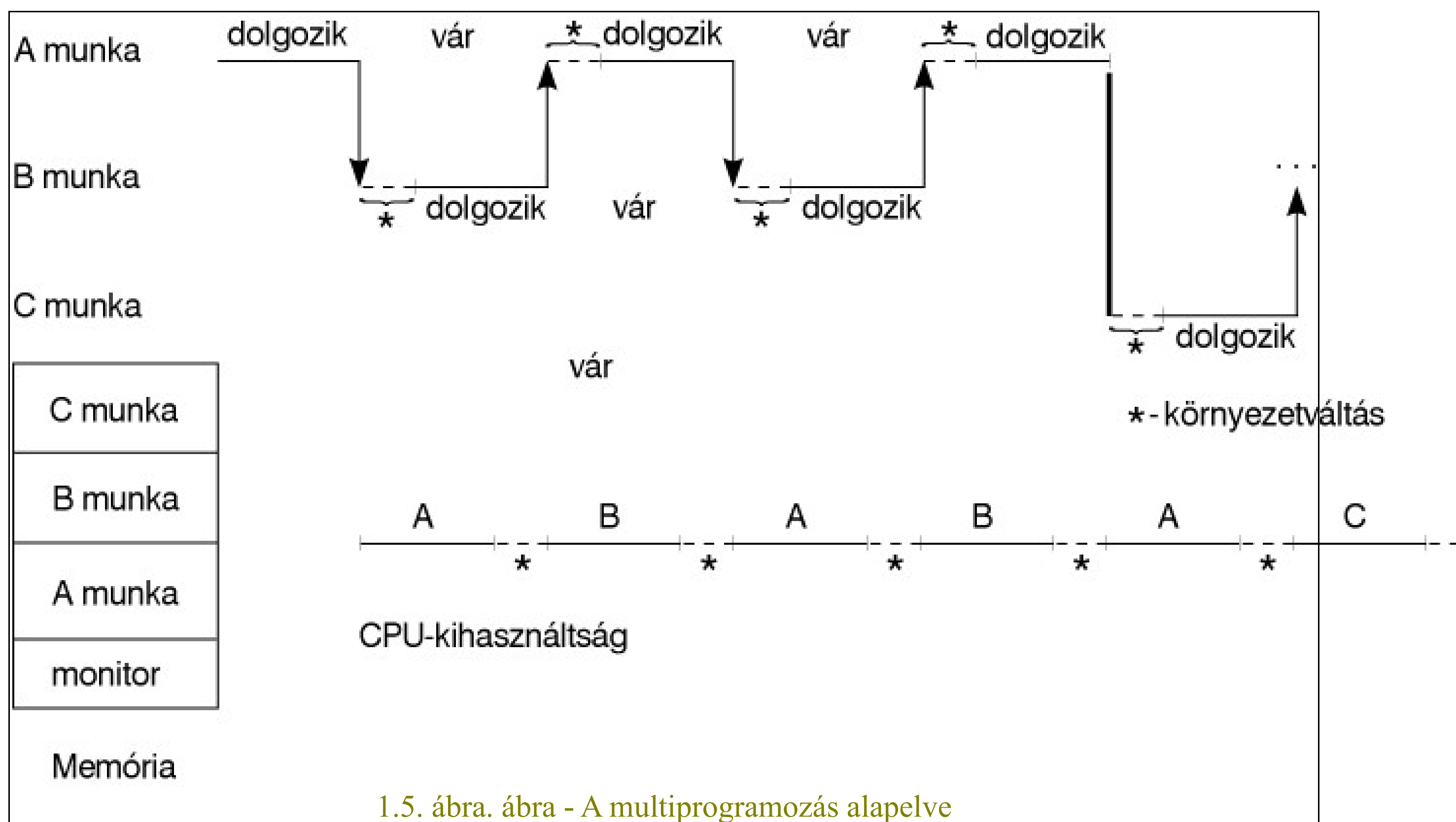


1.4. ábra. ábra - A spooling technika

1.2.3. Multiprogramozott rendszerek

A multiprogramozás megjelenése előtt a lassú perifériák korlátozták a CPU kihasználtságát: a CPU-nak meg kellett várnia a perifériás műveletek befejeződését, mielőtt a következő munka végrehajtásába belekezdhetett. Ráadásul a szekvenciális hozzáférésű tárak miatt a munkákat csak érkezési sorrendjükben lehetett feldolgozni.

A véletlen hozzáférésű tárakról az operációs rendszer tetszőleges sorrendben választhatja ki a következő végrehajtandó munkát, és egy speciális adatszerkezet (nyilvántartás) a „*job pool*” révén úgy *ütemezheti*, hogy a CPU kihasználtsága közel 100%-os legyen. (A későbbiekben látni fogjuk, hogy a gyakorlatban legjobb esetben is csak nagyjából 90%-os kihasználtság érhető el, mivel annak biztosításához, hogy a munkák később tovább futtathatók legyenek, szükséges a környezetük elmentése, illetve visszaállítása.) Egy-egy munka addig fut, amíg várakozni nem kényszerül. Ekkor az operációs rendszer egy másik, futni képes munkát választ ki és indít el. Amint a félbehagyott munka várakozási feltétele teljesül, újra felkerül a futtatható folyamatok listájára, és az operációs rendszer a következő ütemezési alkalommal kiválaszthatja futásra (1.5. ábra).



1.5. ábra. ábra - A multiprogramozás alapelve

A számítógép és a perifériák jó kihasználása mellett a multiprogramozás teljesen új feladatok elé állította a rendszertervezőket. A munkák kiválasztása *ütemezést* igényel. Természetesen egyszerre több programot is a tárban kell tartani, a memória felhasználói területe több munka között oszlik meg, vagyis megjelenik a *tárgazdálkodás* igénye. A CPU mellett a többi *erőforrás felhasználását* is *koordinálni* kell, gondoskodni kell az egyes programok és memóriaterületük *védelméről* a többiekkel szemben stb.

A III. generációs számítógépek elég gyorsak és nagy kapacitásúak voltak és lényegében sikerült megoldani az optimális kihasználtság kérdését is. Így a legégetőbb problémát az jelentette, hogy továbbra sem volt közvetlen kapcsolat a programozó és a gép között, vagyis nehéz volt a programokat módosítani, illetve hibát keresni bennük. Ezért ebben az irányban történtek fejlesztések, aminek hatására létre is jöttek az első olyan rendszerek, amelyeknél közvetlen *on-line* kapcsolat volt a felhasználó és a számítógép között.

1.2.4. Időosztásos rendszerek

Az **időosztásos (time-sharing, multitasking) rendszerek** közvetlen, *interaktív* kommunikációt biztosítanak a felhasználó és programja, valamint az operációs rendszer között. Ezeknél a rendszereknél általában minden felhasználónak külön beviteli eszköze (terminál, konzol) van, melyen keresztül on-line módon adhat parancsokat és kaphat a rendszertől válaszokat. Bár a CPU egyszerre több párhuzamosan dolgozó felhasználó között meg van osztva, azok úgy dolgoznak a számítógépen, mintha az kizárólag hozzájuk tartozna. Ezért fontos elvárás, hogy a rendszer *válaszideje* értelmes tűréshatáron belül legyen. Általában a háttérben valamilyen batch-rendszer is fut, hogy az „üresjáratok” idején se legyen tétlen a gép.

Ehhez a korszakhoz kötődik az MIT, a Bell Laboratorium és a General Electric közösen megkezdett fejlesztése. Terveikhez az ötletet a városi elektromos hálózatok adták, mely szerint minden lakásban megtalálhatók a rendszerhez csatlakozó dugaszoló aljzatok, melyekhez csatlakoztatni lehet a különböző elektromos készülékeket. Ennek mintájára olyan számítógéphez csatlakozó hálózati rendszert szerettek volna létrehozni – először Bostonban, majd másutt is – melyben mindenki számára lehetővé válik – otthonról – a nagy, drága, közös erőforrásokhoz való hozzáférés, vagyis az egy hatalmas közös számítógéphez, valamint adatbázisokhoz stb. való kapcsolódás terminálokon keresztül. Bár a terv abban az időben megvalósíthatatlan maradt, a MULTICS (MULTiplexed Information and Computing Service) néven ismertté vált rendszer híressé vált újszerű és maradandó ötletei miatt. Hogy mást ne említsünk, a projekt pozitív és negatív tapasztalatai egyaránt jelentősen hatottak a UNIX operációs rendszer fejlesztőire. A UNIX pedig még ma is az egyik uralkodó operációs rendszer a miniszámítógépeken és a munkaállomásokon.

1.2.5. Személyi számítógépek

Az LSI (Large Scale Integration) áramkörök megjelenése jelentősen lecsökkentette a számítógép hardver költségeit, így a IV. generációs számítógépek (1980–1990) szinte mindenki számára elérhetők (megvásárolhatók) voltak. Emiatt általánossá vált az *egy felhasználó–egy gép* struktúra, vagyis, hogy minden egyes felhasználó külön számítógéppel rendelkezik. Ezért is hívjuk ezeket az olcsó kisgépeket **személyi számítógépnek (PC: personal computer)**.

A személyi számítógépek megjelenése átütő változást hozott a számítástechnikában. Olyan felhasználók és olyan alkalmazási területek számára vált elérhetővé az eszköz, amelyek korábban fel sem merültek, vagy ahol az ötletek korábban megvalósíthatatlannak bizonyultak (lásd MULTICS projekt). A sokszor egészen más érdeklődésű felhasználók új követelményt fogalmaztak meg a rendszerekkel szemben, mivel nem tudtak és nem is akartak tudni a számítógépek működési elveiről, részleteiről. A *felhasználóbarát (user friendly) szoftver* éppen ezt, azaz a részletek elrejtését és a kényelmes programozási felület biztosítását jelenti. A korra legjellemzőbb operációs rendszerek az MS-DOS (PC-re) és a UNIX (munkaállomásra), amelyek később a felhasználóbarátság jegyében, a feldolgozási sebességben és tárkapacitásban mérhető teljesítményparaméterek rohamos javulásának lehetőségével élve, elsősorban a *felhasználói felületek* tekintetében jelentős fejlődést mutattak (Windows, X-Window).

A személyi számítógépek minden kényelme és előnye mellett azonban a ’80-as évek közepétől igényként merült fel, hogy kapcsolatot lehessen teremteni más – szintén PC-t használó – felhasználókkal, illetve használni lehessen drága, egy-egy ember által már nem megvehető erőforrásokat. Ezért a PC-ket **hálózatba** kezdték kötni. A hálózattal összekötött (és így más felhasználók által is elérhető) személyi számítógéprendszerek rávilágítottak az ezen gépeken futó rendszerek legnagyobb hibájára, nevezetesen, hogy mivel a PC-ket alapvetően úgy tervezték, hogy egyetlen felhasználó használja, ezért semmilyen illetéktelen hozzáférés elleni védelemmel nem látták el őket. Gondoljunk csak például a vírusokra, hogy mennyire esetleges még ma is a PC-k védelme ezekkel szemben.

A ’80-as évek közepe két jelentős technológiai fejlődést hozott. Egyrészt bekövetkezett a mikroprocesszorok minden eddiginél nagyobb teljesítmény- növekedése, másrészt megjelentek a **lokális hálózatok (LAN: local area network)**. Ez utóbbiak általában 10–100 összekötött gépet tartalmaztak, *uniformizált információáramlást* téve lehetővé közöttük. A korábbi központi (centralized) rendszerek (egy CPU, a memóriája, perifériái, terminálok stb.) mellett tehát megjelentek az **elosztott (distributed) rendszerek** (több CPU, közös, illetve külön memóriák stb.).

1.2.6. Elosztott rendszerek

A feldolgozóképesség elosztottsága (decentralizálás) igen sok *előnyös tulajdonságot* hordoz a centralizált rendszerekkel szemben. Hogy csak néhányat említsünk: a *sebesség növekedése*, a *funkciók térbeli elosztása* (a feladathoz igazodó, térben elosztott rendszerstruktúra alakítható ki), a *megbízhatóság növekedése* (egy-egy gép meghibásodása esetén a többi továbbra is működőképes, feladatait az esetek többségében át tudják venni a többiek), a *fejlesztés lehetősége* (kis lépésekben is), az *adatok és eszközök megosztása* (közös adatbázis, illetve drága perifériák közös használata), a *kommunikáció lehetősége* (felhasználók és programok között, elektronikus levelezés, e-mail stb.), *flexibilitás* (terhelésmegosztás, optimalizálás).

Külön ki kell emelni az ár/teljesítmény arányra vonatkozó gazdasági szempontokat. A modern mikroprocesszorok megjelenése előtt Grosch törvénye adta meg, hogy a CPU teljesítménye az ár négyzetével arányos, vagyis kétszeres árért négyszeres teljesítményt lehetett elérni. A mikroprocesszorokra azonban ez már nem érvényes. Kétszeres árért lényegében ugyanazt a CPU-t lehet megvásárolni, valamelyest gyorsabb órával (vagyis a teljesítmény arányaiban alig növekszik). A teljesítménynövelés lehetőségét a több CPU alkalmazása hordozhatja.

Az elosztott rendszerek természetesen *problémákat is felvetnek*, melyek közül a három legfontosabbnak maga a *szoftver* (az elosztottság és párhuzamosság miatt felmerülő problémák komplexitása minőségi ugrást jelent, ezért a korábbiakhoz képest egészen más jellegű operációs rendszerekre lenne szükség, melyekről azt mondhatjuk, hogy bizonyos értelemben még ma is kezdeti stádiumban vannak), maga a *hálózat* (megfelelő átviteli sávszélesség és minőség biztosítása, túlterhelés és egyéb problémák), és a *biztonság* (könnyebb az illetéktelen hozzáférés a „titkos” adatokhoz).

Az elosztott rendszerek fejlesztése során két alapvető cél megvalósítása lebegett a tervezők szeme előtt. Egyrészt olyan rendszereket próbáltak létrehozni, amelyek lehetővé teszik, hogy sok felhasználó tudjon egymás mellett dolgozni és egymással kapcsolatot tartani. A másik cél pedig olyan rendszerek megvalósítása, melyek egy adott problémát a részfeladatok párhuzamosításával maximális sebességgel oldanak meg. Bizonyos szerzők a kétféle rendszer megkülönböztetésére az előbbi esetben használják az **elosztott rendszer** elnevezést (további elnevezések**: hálózati (network)** vagy multikomputeres **rendszerek**), az utóbbiakra pedig a **párhuzamos (parallel) rendszer** elnevezés az elterjedt. A multikomputeres rendszerekben az egyes gépek az esetek többségében saját, külön memóriával és órával rendelkeznek, vagyis **lazán csatoltak (loosely coupled)**, míg a

multiprocesszoros rendszerek többségére a közös memória és óra használat **(szorosan csatolt, strongly coupled** rendszerek) a jellemző.

Az architektúrától függetlenül különbséget kell tennünk a rendszerek között az **átlátszóság (transparency)** alapján. Ennek hiánya esetén *a felhasználóknak tudniuk kell, hogy a rendszerben több számítógép is van*, melyeken saját operációs rendszer fut, és az alkalmazók beléphetnek felhasználóként távoli gépekre is. Az ilyen rendszerek operációs rendszerei nem térnek el alapvetően az egyprocesszoros rendszerek operációs rendszereitől, mindössze *egy hálózatvezérlő (network controller)* kiegészítés szükséges a helyes működtetéshez.

A fentiekkel ellentétben az *átlátszó rendszereket a felhasználó úgy látja, mint egy tradicionális egyprocesszoros rendszert*. Vagyis, bár általában tisztában van vele, hogy a rendszerben több processzor is működik, a feladatok szétosztását az operációs rendszer automatikusan végzi, azaz a felhasználó nem tudja, hogy egy-egy munka melyik processzoron fut, egy fájl melyik számítógép lemezén tárolódik. A feladatok jellegéből következően itt egészen másfajta operációs rendszer szükséges a helyes működtetéshez.

1.2.7. Valósidejű rendszerek

Egészen röviden említjük a IV. generációs számítógép rendszerek egy további fajtáját, a **valósidejű (real-time) rendszereket**.

Azokat a rendszereket, amelyekkel szemben a környezeti, *valós időskálához kötött* idő-követelményeket támasztunk, valósidejű rendszereknek nevezzük. Előírhatjuk például, hogy a rendszer egy környezeti eseményre mennyi időn belül reagáljon, vagy milyen időzített akciókat hajtson végre. Az elnevezés utal arra a különbségre, hogy egy általános célú számítógéprendszer szokásos feladatai „időtlenek” (például egy job végrehajtása a lefuttatás időpontjától függetlenül ugyanazt a helyes vagy hibás végeredményt adja), vagy esetleg belső időt használnak (például szimulációs feladatok esetén).

A valósidejűség szorosan kapcsolódik más tulajdonságokhoz is. A valósidejű rendszerek leggyakrabban célrendszerek (ipari folyamatfelügyelő, -irányító, orvosi rendszerek stb.), melyek célhardveren futnak. Speciális célú operációs rendszereik bizonyos időkorlátok betartásával kell hogy működjenek. Érzékelők (sensor) segítségével észlelt, a *környezetben (külvilágban) történt változásokra adott időn belül válaszolniuk kell*, vagyis a rendszernek garantálnia kell valamilyen előírt időkorláton belüli válaszidőt.

A valósidejű rendszerek két alapvető fajtája: a szigorúbb feltételeket teljesítő **kemény valósidejű rendszerek (hard real-time)** biztosítják, hogy *a kritikus munkák befejeződnek időben*, míg a **lágy valósidejű (soft real-time) rendszerek** csak azt garantálják, hogy *a kritikus munkák prioritással futnak*.

A valósidejű operációs rendszerek tárgyalása mind mennyiségében, mind témájában messze meghaladja e könyv lehetőségeit, így ezekkel a rendszerekkel a továbbiakban legfeljebb utalásszerűen foglalkozunk.

1.2.8. Nyílt rendszerek

A hálózatok széles körű terjedése és ezzel együtt új és drága eszközök széles körű hozzáférhetősége új igények megfogalmazását vonta maga után a rendszertervezők felé. A számítógépek V. generációja (napjainkban) olyan számítógéprendszerekben működik, melyek fizikailag is nagy távolságokon keresztül, az egész világot behálózva, szinte mindenhonnan hozzáférhetők. A fizikai, kulturális, technikai különbségek áthidalása szükségszerűen maga után vonja a *kommunikáció és a csatlakozási felületek egységesítését, világszabványok alkalmazását.* Ennek jegyében egy nemzetközi szervezet, az ISO (International Standards Organization) ajánlásokat fogalmaz meg a rendszerek egységes működéséért (OSI: Open System Interconnection), melyeket célszerű mindenkinek betartani, aki piacképes termékeket akar létrehozni. Így léteznek szabványok az operációs rendszerekre vonatkozóan *(open operating system standards)*, a felhasználói felületekre vonatkozóan *(open user interface standards)*, az alkalmazásokra vonatkozóan *(open user application standards)*, és a kommunikációra vonatkozóan *(open communication standards)*.

1.2.9. Napjaink rendszerei

Napjaink rendszerei multiprogramozott rendszerek. Megtalálhatjuk közöttük a IV. generációtól kezdődően mindegyik említett rendszerfajtát. A rendszerek általában interaktív rendszerek, bár találkozhatunk (az esetek többségében nem tiszta) batch rendszerekkel is. Ez utóbbiak azonban különböznek a korai batch rendszerektől, szerveződési elvük nem az azonos munkafázisok szükségszerű csoportosítását jelenti. Elnevezésük azonban megmaradt, mert ezekben a rendszerekben előre összeállított, vezérlő információkkal ellátott munkák futnak, és a programok futásába nem lehet interaktívan beavatkozni.

Az operációs rendszerek kezelői felületét napjainkban az **ablakozó technika** jellemzi. Ez a felhasználók számára lényegesen barátságosabb, áttekinthetőbb munkalehetőséget nyújt. Ugyanakkor az ablakozó felület mellett – elsősorban a távoli belépések számára, továbbá a parancsértelmező szövegmanipulációs lehetőségeinek és az operációs rendszer *batch fájlokkal* történő programozási lehetőségének megtartása érdekében – a hagyományos, parancssor-orientált kezelői felület is megtalálható (lásd UNIX és származékai).

A számítógépek és az operációs rendszerek területe a technika talán legdinamikusabban fejlődő területének tekinthető. Így, bár az operációs rendszerek történeti taglalását itt abbahagyjuk, maga a történet nem fejeződött be. Éppen nemrégiben jelentették be egy elveiben, működésében teljesen új, beágyazott celluláris neurális hálózatokat *(cellural neural network, CNN)* alkalmazó, három dimenziós rácsszerkezetű, analóg, univerzális, tárolt programú számítógép létrehozását, mely a biológiai szinopszisok terjedéséhez hasonló jelterjedési elv alkalmazásával működik, minden eddiginél nagyobb műveleti sebességet és számítási kapacitást biztosítva. Elterjedése esetén nyilvánvalóan forradalmi változásokat fog hozni a számítástechnikában és az operációs rendszerek fejlődésében. És ki tudja, mi minden van még előttünk?

1.3. Rendszermodell és rendszerarchitektúra

Miután megismerkedtünk az operációs rendszerek rövid történetével, most – megmaradva a bevezetőhöz illő áttekintő szinten – váltsunk nézőpontot.

Közelítsünk az operációs rendszerhez a szoftverrendszerek szokásos tervezési szemléletével. Először húzzuk meg a rendszer határait, vizsgáljuk meg a tipikus környezeti kapcsolatokat, és a rendszer viselkedését a környezet szereplőivel való kapcsolatban (rendszermodell). Ezt követően pedig foglalkozzunk azzal a kérdéssel, hogy a modell milyen belső szerkezetekkel valósítható meg (rendszerarchitektúra).

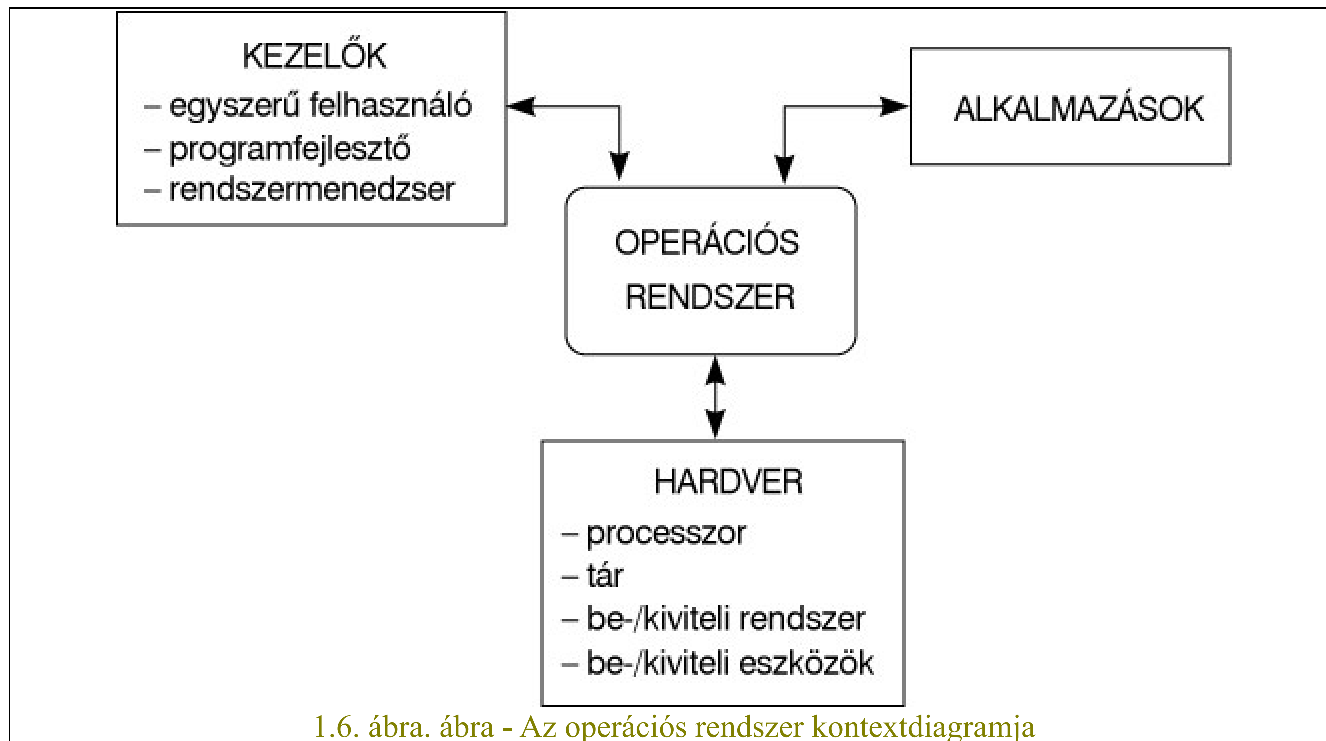
1.3.1. Az operációs rendszer és környezete

A rendszermodell felállításához az 1.6. ábra legyen kiindulási alapunk, amelyik az operációs rendszer ún. kontextdiagramját mutatja be.

A kontextdiagram alapján az operációs rendszer fő környezeti kapcsolatai:

- a kezelők (operátorok),
- az alkalmazói programok,
- a számítógéphardver.

Az operációs rendszernek működése során ezek felé kell csatlakozási felületet nyújtania, és alapvető funkcióit a környezeti kapcsolatokban mutatott viselkedésével jellemezhetjük.



1.6. ábra. ábra - Az operációs rendszer kontextdiagramja

1.3.2. Funkciók

Mint korábban megállapítottuk, az operációs rendszernek két alapvető feladata van:

- *kényelmesen* használható virtuális gép megvalósítása a kezelők és az alkalmazások felé,
- a számítógéphardver *hatékony és biztonságos* működtetése.

A kényelmesen használható virtuális gép azt jelenti, hogy a hardver részleteit el kell fedni, a felhasználók számára áttekinthető modelleket kell kínálni, és azokat meg kell valósítani a fizikai rendszeren. Másként fogalmazva a fizikai rendszer kényelmes és célszerű absztrakcióit kell implementálni az adott számítógép-konfiguráción. Ilyen absztrakciók például az önálló logikai processzorral és tárterülettel rendelkező, egymástól védett, de egymással kommunikálni tudó, párhuzamosan végrehajtódó *folyamatok,* a fizikai címtartomány méretét meghaladó címezhető tártartomány *(virtuális tár)*, a *fájl* és a katalogizált *fájlrendszer,* ahol megoldott a biztonságos és védett adattárolás stb.

A számítógéphardver hatékony működtetése azt jelenti, hogy az alkalmazások futtatását úgy kell koordinálni, hogy a fizikai eszközök kihasználtsága minél jobb legyen, és a felhasználó által előírt prioritások teljesüljenek. A hatékonyság követelményét ugyanis elméletileg megfogalmazhatjuk egy optimumkritériumként, ahol a célfüggvényben különböző minőségi paraméterek szerepelhetnek (például processzorkihasználás, válaszidő, átfutási idő, áteresztőképesség stb., illetve ezen paraméterek statisztikai jellemzői). Az egyes paraméterek jelentősége rendszerenként különböző lehet, illetve egyéb, speciális szempontok is megjelenhetnek.

A biztonságos működtetés pedig azt jelenti, hogy az operációs rendszer korrekt megoldásokat tartalmaz a megosztott erőforrás-használatból eredő problémákra (például közös fizikai processzor használata, a fizikai tár kiosztása, ugyanazon készülékre indított több, egyidejű be-kiviteli művelet végrehajtása), valamint elfedi a tranziens hardverhibákat (például hibatűrő, hibajavító kódolás adattárolások, adatátvitelek esetén).

1.3.3. Csatlakozási felületek

A következőkben az operációs rendszer kapcsolódási felületeit jellemezzük.

1.3.3.1. Kezelői (operátori) felület (Operator Interface, User Interface)

A kezelői felület ember–gép kapcsolat. Arra szolgál, hogy az operációs rendszer ezen keresztül működtethető legyen, illetve működéséről a felhasználó tájékoztatást kapjon. A kezelői felület kialakításának tipikus fizikai eszköze a képernyőt, billentyűzetet és esetleg egeret (grafikus pozicionáló eszköz) tartalmazó munkahely. Kevésbé elterjedten – egyelőre főként speciális rendszerekben – már az operációs rendszerek kezelői felületén is találunk egyéb eszközöket, például hang be- és kimeneteket.

A rendszert kezelő felhasználók a következő jellegzetes csoportokra oszthatók: *egyszerű felhasználókra, alkalmazásfejlesztőkre,* és *rendszermenedzserekre*.

Az *egyszerű felhasználók* jellegzetes tevékenysége, hogy programokat (alkalmazásokat) futtatnak, amelyek segítségével szokásos napi feladataikat látják el (például irodában dokumentumkezelés, táblázatszerkesztés;

automatizált üzemben diszpécser-feladatok stb.). Az egyszerű felhasználók jelentős része elsősorban az alkalmazásokat és azok kezelői felületét használja, az operációs rendszerrel közvetlenül alig kerül kapcsolatba. Számukra az operációs rendszerben a főszerepet a katalogizáltan nyilvántartott fájlok játsszák, amelyek között vannak futtatható programokat tartalmazó, végrehajtható fájlok, továbbá vannak az alkalmazások által kezelt, adatokat tároló fájlok. Érdekli őket, hogy tudnak-e futtatni egyidejűleg több alkalmazást, és ezek az alkalmazások tudnak-e információt cserélni egymással. Az egyszerű felhasználó elvár bizonyos megbízhatóságot, védelmet és biztonságot a rendszertől, ennek érdekében tudomásul veszi, hogy bejelentkezés, azonosítás, és néhány egyéb rendszabály betartása szükséges. A rendszabályokat azonban hajlamos lazán és fegyelmezetlenül kezelni, amint azok számára a legkisebb kényelmetlenséget okozzák.

Az egyszerű felhasználó számára tehát az operációs rendszer olyan gép, amelyik egy felhasználói körnek lehetőséget ad adat- és programfájlok védett és rendezett tárolására, valamint alkalmazások futtatására. A rendszer legfontosabb szolgáltatásai: bejelentkezés, a rendelkezésre álló alkalmazások áttekintése, alkalmazások indítása, esetleges együttműködése és leállítása, fájlműveletek (másolás, mozgatás, törlés, tartalomfüggő feldolgozás).

Az *alkalmazásfejlesztők* részletes ismeretekkel rendelkeznek az operációs rendszer alkalmazói programok számára nyújtott szolgáltatásairól és bizonyos mélységig ismerik az operációs rendszer belső működését. Ezeket az ismereteiket a programkészítésben használják fel. Mint kezelők, az általuk készített alkalmazások tesztelésével, teljesítményelemzésével is foglalkoznak. Egyes operációs rendszerek nyújtanak ehhez eszközöket (például debug módú futtatás), gyakoribb eset azonban, hogy a fejlesztést támogató speciális alkalmazás (például PASCAL, C, C++ nyelvek integrált fejlesztői környezete) tartalmazza a megfelelő szolgáltatásokat. Mindenképpen szükségesek azonban olyan eszközök, amelyek az egyes programok és a hozzájuk tartozó erőforrások (memóriaterület, processzor, perifériák) aktuális állapotát, az erőforrás-használatok időtartamát, az ezekre vonatkozó statisztikákat láthatóvá teszik. Ezek akkor is az operációs rendszer szolgáltatásaira támaszkodnak, ha a fejlesztést támogató alkalmazás teszi őket elérhetővé az alkalmazásfejlesztő számára.

Az alkalmazásfejlesztő számára tehát az operációs rendszer olyan gép, amelyik a programok számára meghívható eljárásokat biztosít, amivel

- leveszi a válláról a hardver pontos és részletes ismeretének és programozásának terhét,
- lehetőséget ad arra, hogy együttműködő programokkal oldjon meg egy feladatot,
- a programok együttműködéséhez ellenőrzött eszközöket ad,
- megszervezi a programok együttfutását és erőforrás-használatát,
- megfigyelhetővé teszi a programok futása közben kialakuló rendszerállapotokat.

Az alkalmazásfejlesztőknek nyújtott szolgáltatások tehát már bizonyos mértékű bepillantást engednek az operációs rendszer belső szerkezetébe, a futó programokhoz rendelt fizikai eszközök állapotát is láthatóvá, esetleg közvetlenül módosíthatóvá teszik.

A *rendszermenedzser* feladata az operációs rendszer üzemeltetése, valamennyi ezzel kapcsolatos probléma megoldása. Ez magában foglalja egyrészt a *rendszergenerálás* feladatát, ami azt jelenti, hogy a rendszert a rendelkezésre álló hardverhez és az ellátandó feladatokhoz illesztett kiépítésben kell telepíteni. Másrészt *adminisztrációs* feladatokat, ami a rendszer felhasználóinak, a rendszerhez kapcsolódó alkalmazásoknak a nyilvántartását, jogosultságaik kiosztását, a rendszer üzemeltetési szabályainak, biztonsági előírásainak meghatározását, azok betartásának felügyeletét jelenti. Harmadrészt *hangolási* feladatokat, ami a hardver lehetőségeit, a tipikus alkalmazásokat és a rendszer statisztikáit figyelembe véve azoknak a rendszerparamétereknek beállítását jelenti, amelyek az üzemeltetés hatékonyságát befolyásolják (pufferméretek, ütemezési és kiosztási algoritmusok stb.). Negyedrészt *rendszerfelügyeletet* takar, ami a zavartalan, folyamatos működés biztosítását, az esetleges rendellenességek észlelését, elhárítását, az ennek érdekében szükséges időszakos feladatok ellátását (karbantartások, mentések stb.) jelenti.

A rendszermenedzser számára tehát az operációs rendszer olyan gép, amelyik a hardver adott célú hatékony alkalmazását segíti, és amelynek a hardverhez és a feladatokhoz illeszkedő, megfelelő telepítése, behangolása, használatának adminisztrációs és általános üzemeltetési feladatai rá hárulnak.

A rendszermenedzser részletes és alapos ismeretekkel rendelkezik mind az operációs rendszerről, mind a környezetről, és a feladatok ellátásához olyan beavatkozási lehetőségekre van szüksége, amilyen a többi felhasználónak nincs. Ezért a rendszermenedzser számára gyakran külön fizikai eszköz (rendszerkonzol) áll rendelkezésre.

A kezelői felület lehet *szöveges* vagy *grafikus*. A grafikus felületek köny-nyebben áttekinthetők, inkább nevezhetők felhasználóbarát megoldásnak. A szöveges felület előnye ezzel szemben, hogy a kezelő távoli terminálról, kis sávszélességű összeköttetésen keresztül is működtetni tudja a rendszert, és könnyebben megvalósítható a készülékfüggetlenség.

A szöveges kezelői felület lehet *parancsnyelvű* (és általában egyben parancssor-orientált), vagy *menürendszerű* (és általában egyben képernyő-orientált). A menürendszer felhasználóbarát, amennyiben mentesíti a kezelőt a parancskészlet és a gyakran igen bonyolult paraméterezés megtanulása alól. A parancsnyelv előnye ismét csak a készülékfüggetlenség könnyebb megvalósítása, továbbá a parancsok összefűzésének lehetősége (lásd batch üzemmód).

A kezelői felületek általában *interaktív* működtetésre adnak lehetőséget. A kezelő egy beavatkozását a rendszer azonnali reakciója követi. Rákattintunk egy ikonra, kiválasztunk egy menüpontot, vagy begépelünk egy parancsot, mire a rendszer a megfelelő művelet végrehajtásával reagál. Általában a végrehajtás befejezését követően fogad el a rendszer újabb kezelői beavatkozást vagy parancsot *(szinkron működés).* Vannak rendszerek

(a multiprogramozott, vagy multitaszk rendszerek általában ilyenek), amelyek megengedik, hogy az előző parancs befejeződése előtt újabb parancsot indíthassunk *(aszinkron működés)*.

Gyakran hasznos, ha az operációs rendszernek szóló parancsokat egy parancssorozattá fűzhetjük össze. Ezt a parancssorozatot tárolhatjuk például egy fájlban *(batch file, shell script)*, és bármikor egyetlen parancsként végrehajtathatjuk a rendszerrel. Ily módon tulajdonképpen az operációs rendszernek szóló programot állítottunk össze. Úgy is fogalmazhatunk, hogy az operációs rendszer a kezelő számára ugyanúgy egy programozható gépnek látszik, mint a processzor a programozó számára.

Ugyanazon operációs rendszernek többféle kezelői felülete is lehet. Általában a parancsnyelvű felület az alap, amire ráépülhetnek menürendszerű, illetve grafikus felületek. A *batch*, illetve *script* készítésének lehetősége a parancsnyelvű felületen van meg. Az interaktív, illetve a grafikus felületen ilyen „programok” például a WinWord-ből ismert tanítási (makrorögzítés) technikával lennének létrehozhatók, az operációs rendszereknél azonban ez nem terjedt el.

1.3.3.2. Alkalmazási (programozói) felület (Application Interface, Program Interface)

A számítógéprendszeren futó, adott feladatokat megoldó programok valamilyen programozási nyelven (pl. FORTRAN, PASCAL, C, C++ stb.) íródnak. Előkészítési időben (fordítás, szerkesztés) történik meg a program átalakítása a processzor által végrehajtható formára (gépi kód). A program készítője általában feltételezi, hogy a program valamilyen operációs rendszer felügyelete alatt fut, az operációs rendszer pedig kész, előre programozott megoldásokat tartalmaz például a be-/kiviteli műveletekre, az időkezelésre, a dinamikus tárigények kielégítésére, a programok együttműködésének és információcseréjének megoldására. Ezek a műveletek a szubrutinhíváshoz hasonló, de attól néhány tekintetben különböző módon, *rendszerhívásokkal* hajthatók végre. A futtatható program tehát nem zárt abban a tekintetben, hogy saját maga tartalmazza valamennyi, a futása során végrehajtódó gépi utasítását, hanem rendszerhívó utasításokat is tartalmaz, amelyek hatására az operációs rendszer lép működésbe, az operációs rendszer részét képező program kezd futni. Az operációs rendszer által végrehajtott műveletet követően – előbb-utóbb, esetenként jelentős kitérőkkel – általában a hívást követő utasítással folytatódik a hívó program végrehajtása.

Maga a rendszerhívás a legtöbb operációs rendszer és a legtöbb processzor esetén egy *programozott megszakítás* előidézésével történik meg. A szubrutinhívástól – amelyre a legtöbb processzor külön gépi utasítást tartalmaz – abban tér el, hogy végrehajtásakor a visszatérési cím mentése és vezérlésátadás mellett a processzor működési módját (védelmi állapotát) is meg kell változtatni, *felhasználói* (user) módból a privilegizált utasítások végrehajtását is megengedő *rendszer* (system) módba kell átkapcsolni.

A rendszerhívás programozott megszakítással történő megoldása amellett, hogy megoldja a működési mód átváltását, a lehető legnagyobb mértékben függetleníti is egymástól a hívó programot és az operációs rendszert. Az operációs rendszerből a hívó csak egyetlen belépési pontot ismer. Információcsere a hívó és az operációs

rendszer között csak paraméterátadással történik (általában még a végrehajtandó műveletet is átadott paraméter jelöli ki).

Az alkalmazások számára az operációs rendszer egy olyan gép, amelyik kiterjeszti a processzor utasításkészletét. A számítógép és az operációs rendszer együtt egy kiterjesztett utasításkészletű, új gépet alkot, amelyik a processzor utasításkészletén kívül tartalmazza az operációs rendszer műveleteit is. Ez a kiterjesztett gép a futtatható programok rendelkezésére áll.

Természetesen a programok írásakor és fordításakor is számíthatunk erre a kiterjesztett utasításkészletre, noha a használt programnyelv az operációs rendszer felületét gyakran elfedi. A programozó számára a programozási nyelv – hacsak nem assembly nyelvű programozásra gondolunk – eleve magasabb szintű műveleteket enged meg, mint a processzorok gépi nyelve. Ezek a műveletek vagy fordításkor fejtődnek ki gépi utasítássorozattá, vagy a nyelv ún. futtató rendszere tartalmazza a komplex műveleteket megvalósító eljáráskönyvtárat, és a fordító a komplex műveletet megfelelően paraméterezett szubrutinhívássá fordítja. Az operációs rendszert tehát a nyelvi szint elfedi a programozó elől. A rendszerhívások nem közvetlen nyelvi utasítások formájában használhatók, hanem a nyelv által megengedett – általában eljáráskönyvtárakkal megvalósított – komplex műveletekbe épülnek be. A rendszerhívást még akkor is be kell illeszteni a nyelv szintaxisába, ha a nyelvi utasítás pontosan az adott rendszerhívás megfelelő paraméterekkel történő meghívását eredményezi, és semmivel sem többet. Ezt a szintet ragadja meg például a POSIX-szabvány, amelyik az operációs rendszer programozói felületét a C nyelvi elérés megadásával definiálja.

Az alkalmazási felület nyelvi szintű megragadása abból a szempontból is előnyös, hogy a felület ezzel processzorfüggetlenné válik. Az elterjedt programnyelvek, különösen a C nyelv, fordítóprogramjai sokféle processzorra tudnak kódot generálni. A C nyelvű forrásprogram így hordozható, hiszen a rendszerhívásokat is a megfelelő processzor figyelembevételével generálja a fordító.

1.3.3.3. Hardverfelület (Hardware Interface)

A hardver fejlődésének köszönhetően egyre újabb, nagyobb teljesítményű processzorok, be-/kiviteli eszközök, továbbá összekapcsolási módok, architektúrák alakulnak ki. Az operációs rendszerek működtetik a hardvert, igyekeznek hatékonyan kihasználni a hardver lehetőségeit. Ebben a pontban csak a kapcsolódási felület jellegét tárgyaljuk, azonban az operációs rendszer és a hardver szoros és összetett kapcsolatrendszere miatt külön pontot szánunk a számítógép-architektúrák tárgyalására.

Az operációs rendszer és a hardver kapcsolódási felülete több ponton valósul meg.

- Az operációs rendszer maga is program, ami az adott számítógéprendszeren fut, tehát a processzor és az architektúra lehetőségei (utasításkészlet, regiszterkészlet, a rendszerhívás megvalósítása, megszakítási rendszer, címzési módok, be-/kiviteli rendszer) azok, amiket az operációs rendszer felhasználhat.

- Az operációs rendszer kezeli a hardvereszközöket és egyben gazdálkodási feladatokat is ellát. Az alkalmazások számára tárat, processzorhasználatot, lemezterületet biztosít, végrehajtja az alkalmazások által kezdeményezett be-/kiviteli műveleteket. Az architektúra ismerete beépül az operációs rendszert alkotó programokba, meghatározza, hogy az operációs rendszer milyen erőforrás-kezelési és -gazdálkodási feladatokkal foglalkozzon, és ezek során milyen megoldások használata célszerű. Hatékony géphasználat ugyanis nyilvánvalóan csak a működtetett hardver tulajdonságainak ismeretében érhető el.
- Az operációs rendszernek kezelnie kell a rendszerhez kapcsolódó be-/kiviteli eszközöket, amelyek igen heterogén csoportokból kerülhetnek ki, és a rendszer élettartama során új eszközök beillesztésére is sor kerülhet.

Az elsőként említett kapcsolódási pont egyszerűen azt jelenti, hogy az operációs rendszernek az adott processzoron, illetve rendszerarchitektúrán kell futnia, ezekhez illeszkednie kell a működőképesség érdekében. Az elterjedten alkalmazott operációs rendszereknek ezért külön-külön verziójuk van a különböző processzorokra. Egy-egy géptípus különböző kiépítésű konfigurációihoz pedig telepítéskor illeszthetők az operációs rendszerek (telepítés, rendszergenerálás). A tervezők szokásos törekvése, hogy az operációs rendszer processzortól, illetve kiépítéstől függő kódrészeit lehetőleg külön modulokba helyezzék (ez a UNIX és a Windows NT esetén is megfigyelhető).

A második kapcsolódási pont tekintetében is a fenti alapelv érvényesül. Kialakult a kezelendő erőforrások egy egységes absztrakciója (például tár, processzoridő, lemezterületek, fájlok stb.), amelyek gyakorlatilag valamennyi rendszer esetén használhatók. Ezek kezelésének algoritmusai is jelentős részben közösek lehetnek. Az algoritmusoknak azonban vannak hardverfüggő részei (például tárallokáció a lehetséges címzési módokhoz igazítva, környezetváltás a konkrét regiszterkészlet figyelembevételével), amelyek ugyancsak a hardverfüggő modulokba kerülhetnek.

A harmadik kapcsolódási pont a be-/kiviteli eszközök csatlakoztatási felülete. Míg egy rendszer életében a processzor- és architektúra-váltás meglehetősen ritka, készülékcsere, vagy új készülék beiktatása sokkal gyakoribb. Egy új eszköz hozzákapcsolása a rendszerhez egyrészt megfelelő hardvercsatlakoztatás, másrészt megfelelő kezelő program *(driver)* segítségével történik. A készülékek választéka gyakorlatilag korlátlan, minden készülék rendelkezhet olyan specialitásokkal, amelyek csak rá jellemzőek. Az operációs rendszert gyakorlatilag lehetetlen felkészíteni arra, hogy minden elképzelhető készüléket tudjon kezelni. Ezért a készülékeknek is kialakult egy egységes absztrakciója, aminek lényege, hogy a rendszer a készülékeket azonosítani tudja, és egységes módon tud velük párbeszédet folytatni. A párbeszéd a készülékkezelő programokon *(device driver)* keresztül valósul meg. A készülékkezelő programok tehát az operációs rendszer többi részéhez egységes felületen csatlakoznak, de ismerik a készülék részletes működését, és az operációs rendszer egységes felületén kapott parancsokat a specialitások ismeretében hajtatják végre a konkrét eszközökön. (Például egy mágneslemez egység különböző módon, különböző vezérlőegységeken keresztül

kapcsolódhat a rendszerhez. Az operációs rendszer számára ez az egység adatblokkok sorozatát tároló készülék, amire kiadható például egy *„olvass blokkot”* parancs. A készülékkezelő program ismeri a vezérlő és a lemezegység közötti munkamegosztást, tudja, hogy ha nincs bekapcsolva a lemezegység motorja, akkor azt először neki kell-e elindítania és aztán megvárnia a megfelelő fordulatszám elérését, majd elvégeztetnie az olvasást, avagy ezeket a vezérlőegység saját hatáskörében elintézi, és elég egyszerűen továbbadni az olvasás parancsot a vezérlőegységnek.)

Érdekes „munkamegosztás” van az operációs rendszerek szállítói és a készülékgyártók között. Az operációs rendszer szállítója általában mellékeli a nagyobb készülékgyártók tipikus eszközeihez a kezelőprogramokat. Természetesen nem tud felkészülni minden készülékre. Más oldalról a készülékszállítók mellékelik az eszközhöz tartozó kezelőprogramokat különböző operációs rendszerekhez.

Összességében tehát a be-/kiviteli készülékek csatlakoztatása kettős szabványosítást igényel, egyrészt egy szabványos *hardvercsatlakoztatást*, másrészt a készülékkezelő program számára egy szabványos *szoftvercsatlakoztatást*. Az operációs rendszer kezelői felülete pedig megfelelő *eljárást* kínál a rendszermenedzser számára a készülék és a kezelőprogram csatlakoztatására.

1.3.4. Számítógép-architektúrák

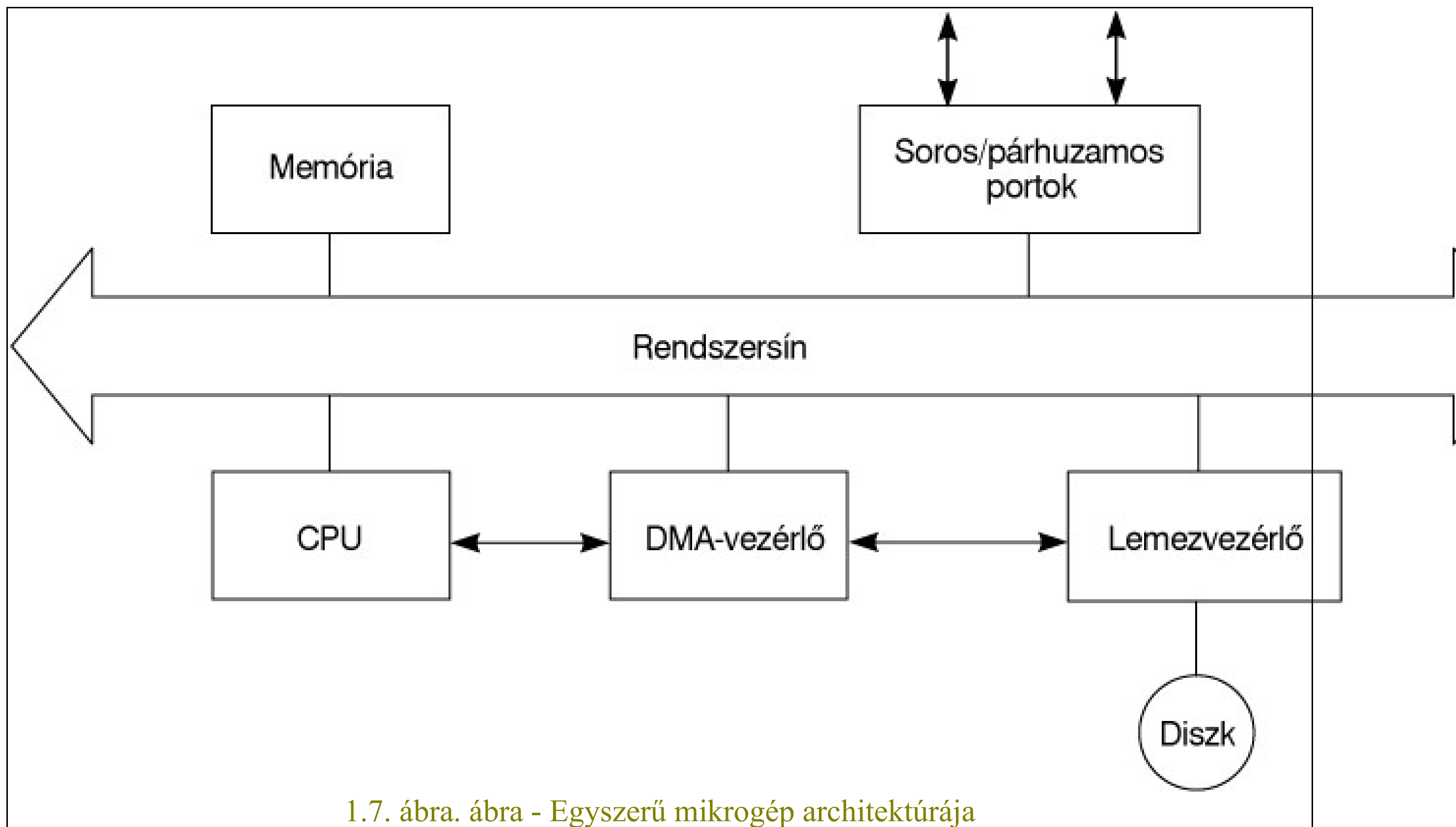
A mikrokontrollerektől a szuperszámítógépekig és elosztott számítási rendszerekig a különböző méretű, teljesítményű és szerkezetű hardverek széles skálája az, amit operációs rendszerrel kell működtetni. Meglehetősen nehéz megtalálni ezekben a rendszerekben azokat a közös vonásokat, amelyek meghatározzák, hogy mit kell tennie az operációs rendszernek a hatékony működtetés érdekében.

Illusztrációként három jellegzetes hardverfelépítést mutatunk be, egy egyszerű mikrogépet, egy tipikus személyi számítógépet, valamint egy szuperszámítógépet.

1.3.4.1. Egyszerű mikrogép

Az egyszerű mikroszámítógép felépítését az 1.7. ábra mutatja.

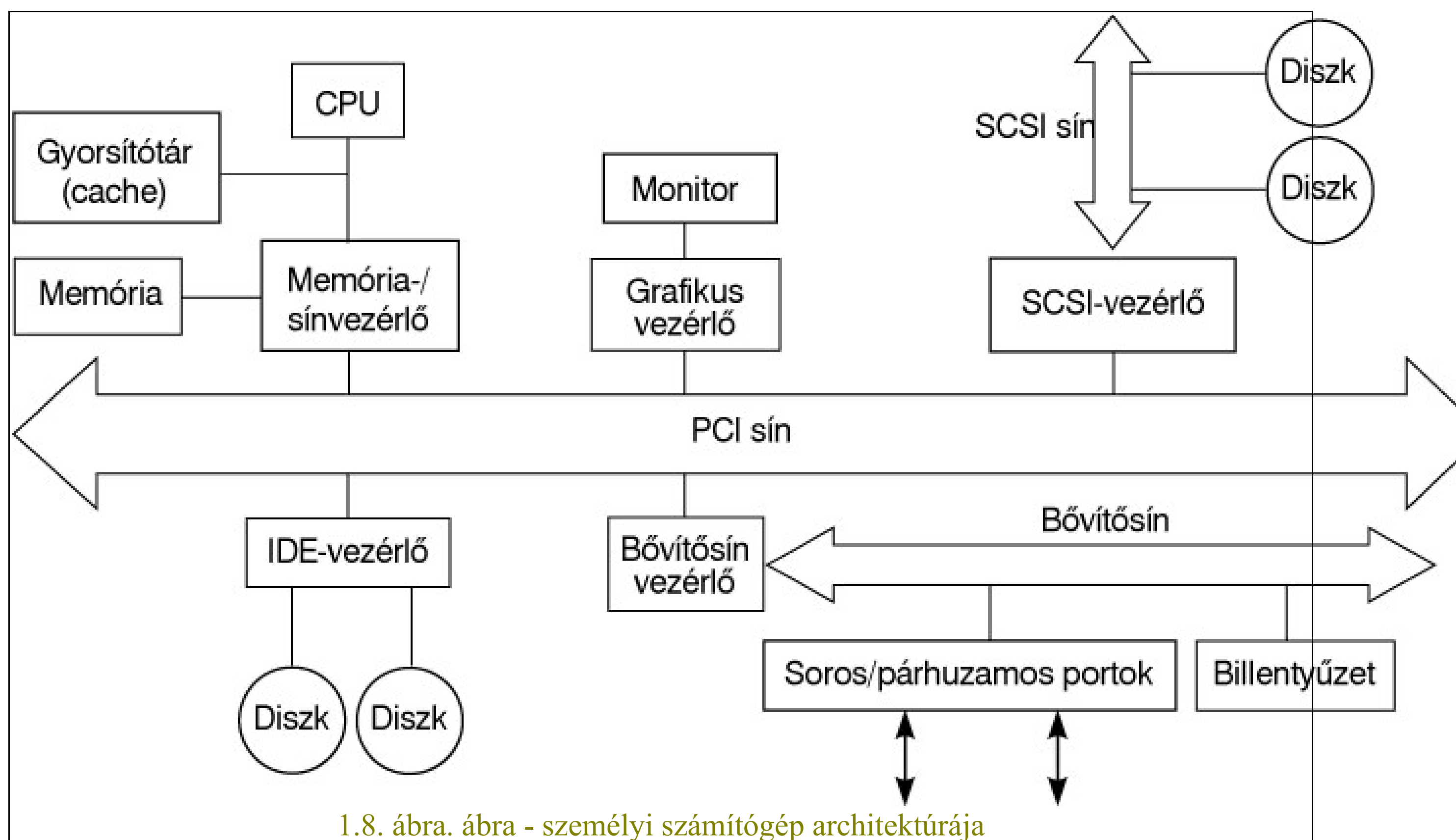
Az egyszerű mikrogép egyetlen sínnel rendelkezik, amelyet mind a CPU-memória, mind a B/K (beviteli/kiviteli) forgalom terhel. A CPU egyben a sínvezérlő szerepét is betölti. Esetleges kezelőszervek a soros/párhuzamos portokra csatlakoztathatók. A DMA-vezérlő képes önállóan szervezni a lemez – memória közötti blokkátvitelt, az adatforgalom azonban a rendszersínt veszi igénybe. A rendszersín használatának szervezésére a DMA-vezérlő és a CPU között külön jelforgalom van. A sín lehetőséget ad megszakításkérések továbbítására is.



1.7. ábra. ábra - Egyszerű mikrogép architektúrája

1.3.4.2. Jellegzetes személyi számítógép

Egy jellegzetes személyi számítógép architektúráját mutatja be az 1.8. ábra. Az architektúra lényegesen bonyolultabb, és olyan megoldásokat tartalmaz, amelyeket korábban csak a nagyszámítógépekben alkalmaztak. A gyorsítótár megfelelő találati arány esetén lehetővé teszi, hogy a memória a CPU-utasítás végrehajtásával egyidejűleg a B/K-adatforgalom számára is rendelkezésre álljon. A grafikus vezérlő saját videomemóriával és intelligens processzorral rendelkezhet, aminek hatása, hogy a grafikus műveletek nem vesznek igénybe nagy sávszélességet a PCI-sínen.

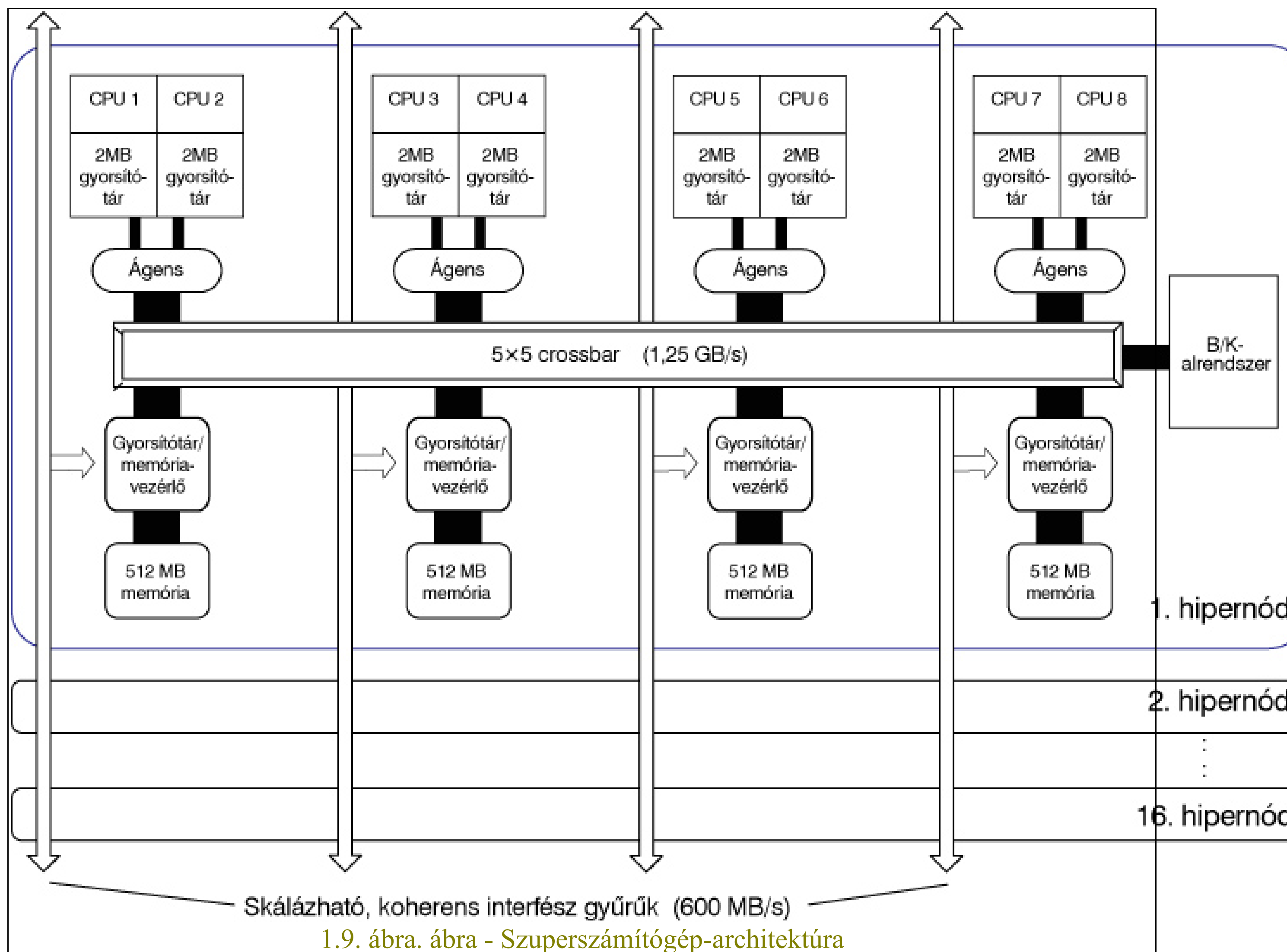


1.8. ábra. ábra - személyi számítógép architektúrája

A bővítősín általában lassabb külső készülékek csatlakoztatására ad lehetőséget, külön vezérlő hajtja meg, biztosítva befelé a megfelelő hatékonyságú sínfoglalást. Természetesen a sín megszakításkéréseket is tud továbbítani. Két további vezérlő (IDE és SCSI) ad lehetőséget jellegzetesen lemezegységek csatlakoztatására, ugyancsak autonóm blokkátviteli képességekkel.

1.3.4.3. Szuperszámítógép

Az 1.9. ábra a Convex Exemplar számítógépcsalád belső felépítését mutatja, amelyik lehetővé teszi, hogy 128 processzor (HP PA-RISC típusúak) hatékony együttműködésével alakuljon ki igen nagy teljesítményű számítási rendszer. Az építkezés hierarchikus, kettős processzoregységekből négy építhető egy *hipernódba*. Minden CPU saját gyorsító utasítás- és adattárral *(cache)* rendelkezik. A *hipernódhoz* egy B/K-alrendszer is tartozik. A crossbar kapcsolóhálózat a processzorpárok, valamint a B/K-alrendszer hatékony és átlapolt kapcsolatát teszi lehetővé a négy memóriablokkal. A memóriablokkok tartalmazzák a *hipernód* saját adatterületét, a közös tár egy adatterületét, valamint a *hipernódok* kapcsolatának hatékonyságát fokozó belső hálózati gyorsítótárak területét. A *hipernódokat* négy SCI *(Scalable Coherent Interface)* szabvány szerint kialakított gyűrű kapcsolja össze. A rendszerben valamennyi processzor számára rendelkezésre áll a memóriában egy közös címtartomány, amely valójában egy elosztott, koherens gyorsítótárral támogatott memóriával valósul meg.



1.9. ábra. ábra - Szuperszámítógép-architektúra

Ha általános következtetéseket akarunk levonni a bemutatott architektúrákból, azt állapíthatjuk meg, hogy egy bizonyos számítási teljesítmény alatt az architektúrák megtartják a gépi utasításkészlet szintjén a Neumann-modell szerinti szekvenciális utasítás-végrehajtást, de legalábbis ennek a látszatát, és ennek, valamint az ezzel átlapolódó B/K-műveleteknek a támogatására adnak egyre hatékonyabb eszközöket. A valódi működés elosztottá és többszálúvá válik, azonban ez a programozó elől rejtve marad.

A nagyobb teljesítménykategóriákban már a gépi utasítások szekvenciális végrehajtásának látszata sem marad meg, a magasabb (nyelvi, operációs rendszer) szintű fogalmak hatékony megvalósítása a cél. A szuperszámítógépek egyik alkalmazási stílusa, hogy a magasszintű nyelveken íródott programokban a fordítók megkeresik a párhuzamosítható műveleteket (például vektor- és mátrixműveletek) és az adott hardverarchitektúrán hatékonyan futó kódot generálnak. A másik alkalmazási stílus a műveletvégrehajtás párhuzamosítását a programozó kezébe adja és speciális operációs rendszerekkel és programozási nyelvekkel támogatja.

1.3.5. Belső szerkezet

Az operációs rendszerek meglehetősen nagy és bonyolult rendszerek. Létrehozásuk során – mai szemmel – a nagy és bonyolult rendszerek valamilyen fejlesztési módszertanát célszerű alkalmazni, amelyek lényeges alapelve a rendszer részekre bontása (dekompozíció), és az általánosítás, azaz a részletek elrejtése (absztrakció). A módszeres fejlesztés eredményeként általában egy jól strukturált (áttekinthető, a feladat fogalomrendszeréhez igazodó) rendszer jön létre. Egy zavaró tényező azonban nem hagyható figyelmen kívül: az operációs rendszerek futási idejű hatékonysága (minél kisebb tár és minél kevesebb processzoridő felhasználása a feladatok teljesítéséhez) fontos szempont (még ma, a kissé indokolatlanul felfokozott teljesítmény- és tárigények korában is). A tiszta struktúra és az adott esetre vonatkozó maximális hatékonyság pedig általában ellentmondó követelmények.

Említésre méltó, hogy a fejlesztési módszertanokra vonatkozó igény (és a szoftverkrízis tudatosodása) jelentős részben a korai multiprogramozott operációs rendszerek fejlesztési problémáinak köszönhető. (Különösen nagy szerepe volt a legendás IBM 360 sorozatra készült, univerzális, OS 360 operációs rendszer hibáinak.) Ezek a rendszerek monolitikus szerkezetűek voltak, azaz egységes rendező elv szerinti belső struktúrájuk nem ismerhető fel.

A későbbi rendszerekben általában felismerhető a strukturált programozási szemlélet eredményeként kialakult rétegszerkezet, illetve a moduláris programozási elvek szerinti modulszerkezet. Napjainkra a hordozhatóság és a hardver lehetőségeinek hatékony kihasználása vált a belső szerkezet kialakításának fő szempontjává.

1.3.5.1. Rétegek és modulok

A strukturált programozás alapelveinek következetes betartása (eljárásabsztrakción alapuló lépésenkénti finomítás) rétegszerkezetű programrendszert eredményez. A rétegek hierarchikusan egymásra épülnek. Minden réteg úgy fogható fel, hogy az alatta lévő réteget – mint **virtuális gépet** – használva egy bonyolultabb virtuális gépet valósít meg a felette elhelyezkedő réteg számára. Ebből következik, hogy minden réteg csak a közvetlen alatta elhelyezkedő réteg „utasításkészletét” használhatja.

A moduláris programozás dekompozíción alapuló megközelítése szerint a nagy programokat szét kell vágni külön kezelhető részekre, modulokra. A modulok belseje a külvilág (többi modul) számára rejtett, a modulból csak az általa mutatott csatlakozási felület érhető el más modulok számára (interface). A modulokat lehetőleg úgy kell kialakítani, hogy a közöttük lévő csatolás minimális, a modul összetartó ereje (kohézió) pedig maximális legyen. Tapasztalatok szerint a maximális kohéziót és minimális csatolást egy-egy komplex adatszerkezetet és a rajta végezhető műveleteket összefogó modul eredményezi, amelyik eljárás-interfésszel és paraméterátadással tart kapcsolatot más modulokkal. Lényegében a ma legelterjedtebb objektumorientált dekompozíció is ezen az elven alapul. Mindemellett más, gyakorlati szempontok indokolhatják a modulok más elvek szerint történő kialakítását (például a rétegszerkezet mentén történő modularizálás, adatmodulok, vezérlőmodulok, interfészmodulok kialakítása stb.).

Illusztrációként bemutatjuk a THE és a VENUS operációs rendszerek rétegszerkezetét (1.10. ábra). A THE operációs rendszert (elsősorban kötegelt feldolgozásra) E. W. Dijkstra (a strukturált programozás és a szemafor atyja) és munkatársai fejlesztették ki és publikálták 1968-ban. 1970-ben Brinch Hansen javasolta elsők között a rendszermag (kernel, nucleus) kialakítását az operációs rendszer alapfunkcióinak megvalósítására. A két munka nyomán fejlesztették ki Liskov és munkatársai az inkább időosztásos üzemmódra szánt VENUS operációs rendszert, amelynek alsó öt rétegét mikroprogramozottan valósították meg (1972).

6. réteg	Felhasználói programok
5. réteg	Készülékkezelés és ütemezés
4. réteg	Virtuális tár
3. réteg	B/K-csatornák
2. réteg	CPU-ütemezés
1. réteg	Parancsértelmező
0. réteg	Hardver

1.10.ábra. táblázat - Nevezetes operációs rendszerek rétegszerkezete

5. réteg	Felhasználói programok
4. réteg	B/K-pufferelés
3. réteg	Operátori konzol kezelése
2. réteg	Tárkezelés
1. réteg	CPU-ütemezés
0. réteg	Hardver

1.12.ábra. táblázat - A virtuális hardver megvalosító rendszer és közös kernel

A rétegszerkezet kialakítása messze nem egyértelmű, hiszen például a háttértárat kezelő program maga is hasonlóan működhet, mint egy felhasználói program, mivel működése során várakozásokra kerülhet sor. Ezért indokolt őt a CPU-ütemező réteg fölé helyezni. Ugyanakkor a CPU-ütemező bonyolultabb rendszerekben dönthet úgy, hogy egy programot kisöpör a memóriából háttértárra, amihez szüksége van a háttértár-kezelő szolgáltatásaira. Ez a szempont éppen fordított réteghierarchiát indokolna. Úgy is fogalmazhatnánk, hogy a réteghierarchia és az ésszerű, funkcionális modulszerkezet ortogonálisan alakul ki. A rétegszerkezet másik problémája, hogy a szigorú hierarchia betartása sok üresjáratot okoz. Egy mélyebb rétegben megvalósított szolgáltatás használata ugyanis több rétegen át érdemi funkció nélkül végrehajtott eljáráshívásokat és paraméterátadásokat eredményez, ami rossz hatékonyságra vezet.

A réteges szerkezet problémái miatt a későbbi operációs rendszerek már nem törekedtek tiszta rétegszerkezetre, hanem kevés réteget és inkább több funkcionális modult alakítottak ki.

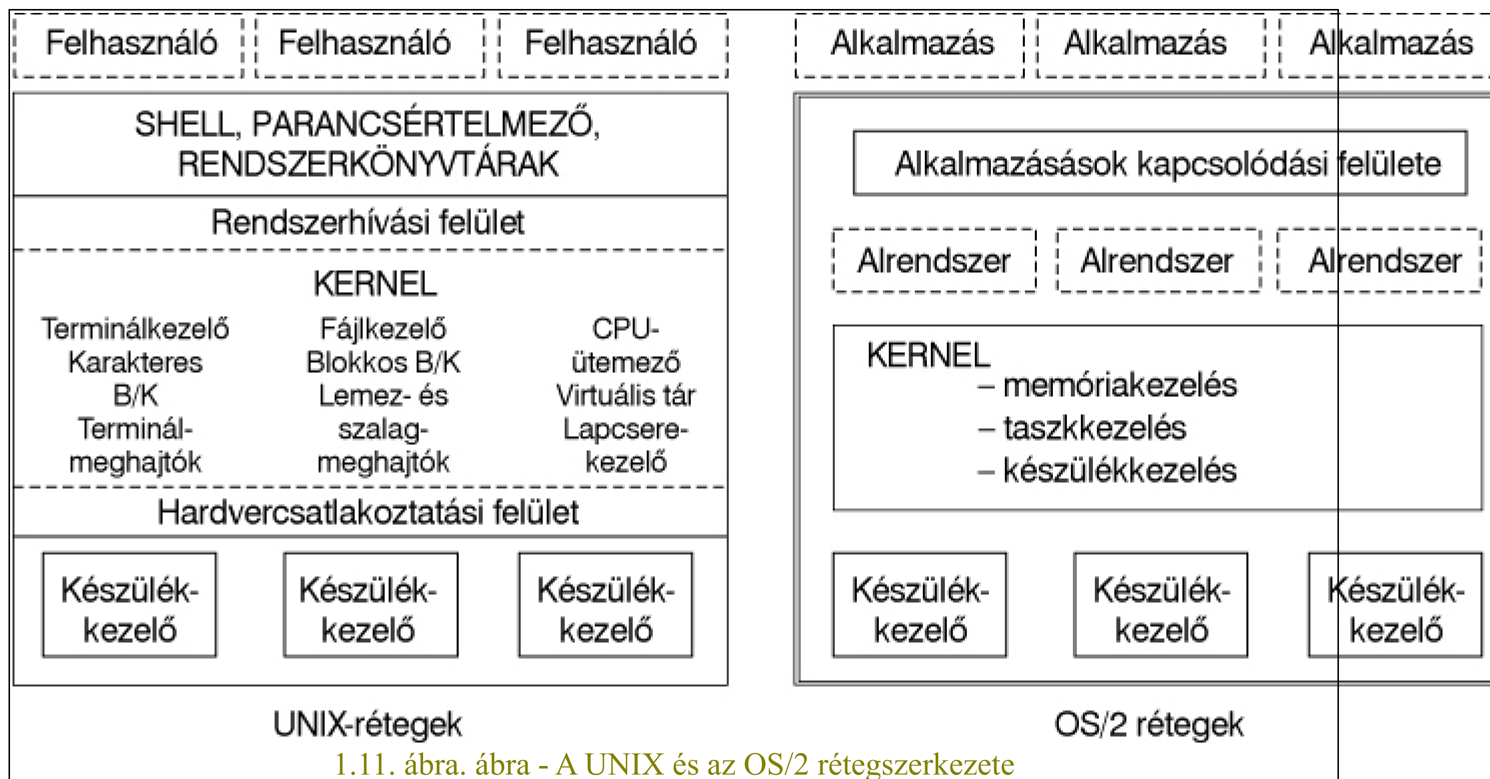
1.3.5.2. Tipikus modulok

Az operációs rendszerek feladataiból adódóan számos rendszerben a tipikus funkcióknak megfelelő modulszerkezetet találunk. A szokásos alapvető funkciócsoportok a következők:

- folyamatkezelés,
- tárkezelés,
- fájlkezelés,
- be-/kivitel kezelése,
- háttértár kezelése,
- hálózatkezelés,
- védelmi és biztonsági rendszer,
- parancsértelmezés,
- felhasználóbarát kezelői felületek.

A fenti funkcionális modulok megjelenése mellett a rétegszerkezet abban nyilvánul meg, hogy

- a hardverfüggő részek egy alsó, hardverközeli rétegbe kerülnek;
- az operációs rendszer alapfunkciói a rendszermagban (kernel) valósulnak meg, amely egyre inkább csak a minimálisan szükséges funkciókat tartalmazza (mikrokernel);
- a feltétlen szükségest meghaladó kényelmi szolgáltatások a felhasználói programokhoz hasonló rendszerprogramokba (rendszerfolyamatokba, alrendszerekbe) kerülnek;
- megjelenik a felhasználói programokból érkező rendszerhívások fogadó felületét megvalósító réteg.



1.11. ábra. ábra - A UNIX és az OS/2 rétegszerkezete

Illusztrációként bemutatjuk a UNIX rendszerek és az OS/2 operációs rendszer belső szerkezetét, de a fentiekhez hasonló tendencia figyelhető meg a Windows NT rendszer egymást követő verzióin is (1.11. ábra).

1.3.5.3. Virtuális hardver

A kernelkoncepció lényege, hogy a felhasználói és a rendszerprogramok számára a processzor utasításkészletének kiterjesztésével a kernel egy virtuális gépet jelenít meg. Mind a felhasználói, mind a rendszerprogramok futásuk során tehát egyrészt a processzor utasításkészletét (eltekintve a privilegizált utasításoktól), másrészt a kernel által nyújtott, rendszerhívással elérhető műveleteket használhatják. A rendszerben egyetlen kernel és több aktív program van jelen, a programok tehát mind a valódi processzort, mind a kernelt közösen, megosztottan használják. Ez azt jelenti, hogy nem csak a valódi processzort, hanem a kernelt is multiprogramozottan kell használni, azaz a processzor állapotának elmentéséhez hasonlóan a kernel állapotát is menteni kell, amikor egyik programról a másik futtatására vált át a rendszer.

Az IBM kutatóinak ötlete volt, hogy a kernel alatt elhelyezkedő rétegbe olyan funkció kerüljön, amelyik egyszerűen a fizikai eszközök megosztott használatára ad lehetőséget, nem kiterjesztett, hanem pontosan a fizikai processzorral és eszközökkel azonos felületen. Így minden felhasználó egy saját virtuális hardvert lát, amelyen elvileg tetszőleges kernelt, operációs rendszert futtathat (általában egyfelhasználós, interaktív üzemmódban) (1.12. ábra). A megoldást először az IBM 360/67-es számítógépre dolgozták ki (1970), kereskedelmi forgalomba pedig az IBM VM/370 operációs rendszerben került (1979).

Alkalmazás	Alkalmazás	Alkalmazás

Kernel		
Hardver		
Alkalmazás	Alkalmazás	Alkalmazás
Kernel	Kernel	Kernel
Virtuális hardver		
Hardver		

2.9.ábra. táblázat - Többpéldányos erőforrások pillanatnyi allokációja és igénylése
Közös kernel Virtuális hardver privát kernelekkel

A virtuális hardver gondolatának továbbvitele napjainkban ismét népszerű. A felhalmozódott szoftverkincs felhasználásának egyik problémája a hardver-kompatibilitás hiánya. Ezen is segíthet a virtuális hardver vagy hardveremuláció. Jellegzetes példák a *Sun* és *DEC* processzorain futó operációs rendszerek virtuális *Intel* CPU-ja, az *Apple Macintosh* virtuális *Motorola 68000* processzora. A JAVA-programok kompatibilitásának kulcsa ugyancsak a *JVM (Java Virtual Machine)* virtuális gép, amelyik bájtkód szinten definiált utasításkészletet képes végrehajtani.

Ebbe a sorba illeszkedik az MIT (Massachusetts Institute of Technology) kutatóinak *exokernel* koncepciója (1995), amelyik ugyancsak a legalsó szintre helyezi a hardver multiplexelésének feladatát, és az egyes felhasználóknak a kívánt hardver-erőforrásokhoz biztonságos, védett hozzáférést nyújt. Ez a szint azonban semmiféle további kiterjesztett funkciót nem valósít meg a hardverhez képest.

1.3.5.4. Kliens–szerver szerkezet

A kernel minimalizálása és a rendszerszolgáltatások rendszerprogramokkal történő megvalósítása a szerkezeti tisztaság és a hatékonyság jó kompromisszumának látszott. Találkozott ez a tendencia a hardver lehetőségeinek bővülésével, többek között azzal, hogy egyetlen rendszeren belül több processzor működik. Ez egyrészt több CPU-t jelenthet, másrészt speciális (aritmetikai, sínvezérlő, készülékvezérlő stb.) processzorokat. A multiprogramozás nyújtotta *látszólagosan* párhuzamos végrehajtással szemben a több procesz-szor lehetőséget ad a programok *valóban* párhuzamos végrehajtására. A hardverfüggetlenségre való törekvés jegyében megnőtt azoknak a *modelleknek* a jelentősége, amelyeket követve *olyan programokat hozhatunk létre, amelyek változtatás nélkül, hatékonyan futtathatók akár egyetlen processzoron multiprogramozottan, akár több processzoron, multiprocesszálással*.

Ezeknek a követelményeknek megfelel a **kliens–szerver-modell**. Lényege, hogy adott funkciókért egy program felelős, amelyik a funkciók ellátásához szükséges erőforrásokat is kezeli. Ennek a programnak (szolgáltató, szerver) a szerepe a többiek (ügyfél, kliens) kiszolgálása az adott funkciók tekintetében. Az ügyfél üzenetet küld a szolgáltatónak, ami tartalmazza a kért műveletet és a szükséges paramétereket, és megvárja a szolgáltató válaszát. A szolgáltató fogadja az ügyfelek kéréseit, elvégzi a kért műveleteket és az eredményekről válaszüzenetben tájékoztatja az ügyfeleket.

Ragadjuk ki egy operációs rendszer néhány funkcióját, például a fájlkezelést, a folyamatkezelést és a felhasználóbarát kezelői felület nyújtását (például grafikus terminálon). Bízzuk ezeket a feladatokat egy-egy szolgáltatóprogramra! Szükség esetén a szolgáltatók természetesen egymás szolgálatait is igénybe vehetik. Az így kialakuló szerkezetet az 1.13. ábra mutatja. A rendszerben a kernel elsődleges szerepe az üzenetek pontos és biztonságos közvetítése. A szolgáltatók

1.13. ábra. ábra - Kliens-szerver szerkezetű rendszer

változtatás nélkül működnek akkor is, ha a szerverek külön-külön processzoron futnak. Természetesen a kernelnek ilyenkor ismernie kell a szereplők elhelyezkedését, és az üzenetközvetítés során használnia kell a processzorok közötti adatátvitel eszközeit.

1.3.6. Működés

Ez a pont az operációs rendszerek működését, azaz jellegzetes belső vezérlési szerkezetét tárgyalja. A programokkal kapcsolatban a vezérlési szerkezet az egyes utasítások végrehajtási sorrendjének előírását jelenti. Arra keressük tehát a választ, hogy mikor mit csinál a program.

A hagyományos programok esetén megszoktuk, hogy az utasítások végrehajtása a processzor működésének megfelelően, sorosan történik, azaz egy utasítás befejezése után kezdődik a következő utasítás végrehajtása. A programnak van kezdete, ahonnan a végrehajtás elkezdődik, és – esetleg adatfüggő útvonalon – halad a program végéig. A processzort a program végrehajtásának menetéből csak egy megszakításkérés elfogadása tudja kizökkenteni. Ilyenkor a processzor úgy tér át automatikusan a megszakításhoz rendelt kiszolgáló program végrehajtására, hogy a megszakított programot később folytatni tudja. A megszakítási rendszer célja és értelme az, hogy a rendszer valamilyen külső, esetleg belső történésre gyorsan tudjon reagálni.

Az operációs rendszer működésének vizsgálatakor legyen az a kiinduló állapotunk, hogy a CPU éppen egy felhasználói program utasításait hajtja végre. Az operációs rendszer ilyenkor nyugalomban van. Mi hozhatja működésbe?

1. A futó program *rendszerhívást* hajt végre.
2. A processzor külső *megszakításkérést* fogad el.
3. Hibamegszakítás *(exception)* következik be.

Bár a rendszerhívás a hívó program szándéka szerinti vezérlésátadás, megvalósítása a legtöbb rendszeren programozott megszakítással történik. Így mondhatjuk, hogy az operációs rendszert mindig megszakítás hozza működésbe, azaz működését a megszakítást kiváltó *események vezérlik*. Az egyes pontokban tárgyalt megszakítások azonban néhány fontos tulajdonságban különböznek, ezért indokolt a külön csoportba sorolásuk.

Az operációs rendszernek van néhány olyan alapfunkciója, amelyik látszólag elemi, de működése nem korlátozódik egyetlen rendszerhívás, vagy megszakítás hatására végrehajtott utasítássorozatra, hanem vezérlési szerkezetében bonyolultabb megoldást igényel. Ezek közül a B/K-műveletek és a kezelői parancsok végrehajtását, valamint az időmérést tárgyaljuk.

A működés két sajátos szakasza a rendszerindítás és rendszerleállás folyamata. Az operációs rendszer funkcióit általában „állandósult” állapotban tárgyaljuk, azonban mindennapi tapasztalatainkból is tudjuk, hogy mind az indítás, mind a korrekt leállítás olyan folyamat, amelynek során a rendszer speciális üzemállapotban működik, és ezek a műveletek elég időigényesek lehetnek. Röviden ezzel a kérdéssel is foglalkozunk.

1.3.6.1. Rendszerhívások kezelése

A **rendszerhívás** általában programozott megszakítást okoz. Hatására az adott processzor megszakítási rendszerének mechanizmusa szerint az operációs rendszer egy meghatározott belépési pontjára kerül a vezérlés. Az elvégzendő műveletet, valamint annak paramétereit a rendszer konvenciói szerint átadott paraméterekkel jelöli ki a hívó program. Az operációs rendszer a műveletet kijelölő paraméter szerint elágazva végrehajtja a műveletet, majd az esetek többségében visszatér a hívó programhoz.

Néhány esetben azonban nem a hívó programhoz való visszatérés történik. Ilyen esetek lehetnek:

- a program befejeződését jelző rendszerhívás, amikor a program által használt erőforrások felszabadíthatók, esetleg új program (például a job következő lépése) betöltése és indítása következhet,
- erőforrás-igénylés (például munkaterület a tárban vagy lemezen, kizárólagos periféria- vagy fájlhasználat stb.), amit a rendszer éppen nem tud kielégíteni, emiatt a hívónak várakoznia kell,
- várakozás másik program jelzésére vagy üzenetére, ami még nem érkezett meg, emiatt a hívónak várakoznia kell,
- adott időtartamú késleltetés vagy adott időpontra való várakozás,
- újraütemezés kérése, azaz lemondás a futásról esetleges fontosabb feladatokat megoldó programok javára,
- B/K-művelet indítása, aminek végrehajtása alatt a hívó várakozik.

Amikor egy rendszerhívás következtében a hívó program várakozásra kényszerül, az operációs rendszer megőrzi a program állapotát, de a várakozás idejére más program végrehajtásáról intézkedik. A hívás helyére

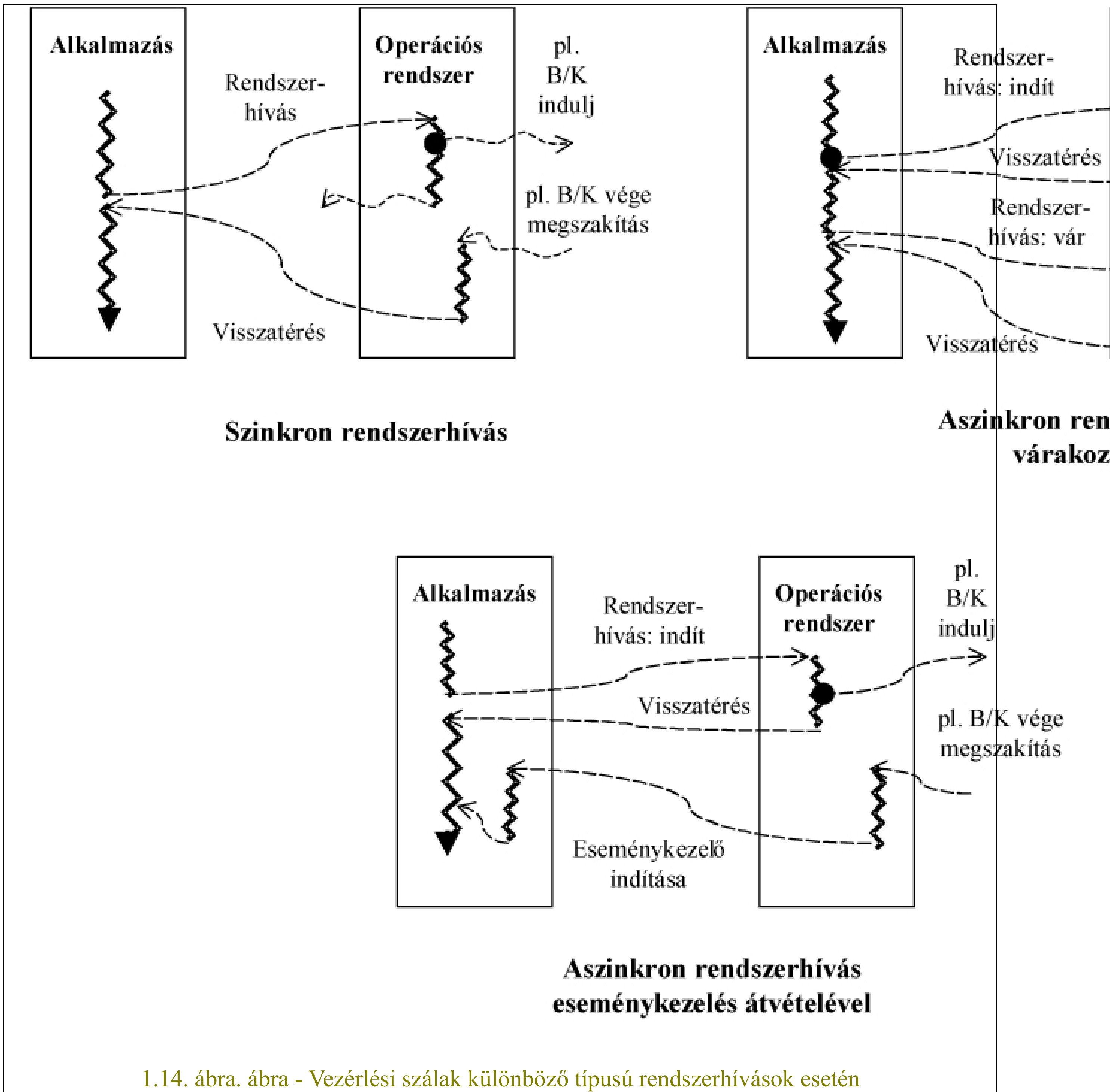
való visszatérés tehát késleltetve lesz mindaddig, amíg a várt feltétel nem teljesül, és a rendszer úgy nem dönt, hogy ismét az adott programot futtatja.

A hívó program számára a várakozás a legtöbb esetben nem jelent a szekvenciális végrehajtástól eltérő működést, számára logikailag a várakozás mindössze annyit jelent, hogy a rendszerhívás „lassan” hajtódik végre (szinkronművelet).

Néhány rendszer lehetővé tesz aszinkron rendszerhívásokat is, jellegzetesen a B/K-műveletek, valamint a programok együttműködését szervező műveletek esetén. Az aszinkron B/K-rendszerhívás a kért B/K-művelet elindítása után visszatér a hívóhoz, a hívó program a B/K-művelettel párhuzamosan folytatódhat, és külön rendszerhívás szolgál az átvitel befejeződésének lekérdezésére, illetve az arra való várakozásra.

Más rendszerek arra adnak lehetőséget, hogy egy program megadjon egy olyan belépési pontot a saját címterében, amelyikre a rendszer valamilyen esemény bekövetkezésekor (például B/K-művelet befejeződése, másik program jelzésének megérkezése) adja át a vezérlést. Így a felhasználói program tulajdonképpen egy külső eseményre induló ágat, azaz egy megszakításkezelés jellegű programrészt is tartalmaz, amelyik azonban természetesen felhasználói módban fog végrehajtódni.

A rendszerhívások hatására kialakuló jellegzetes vezérlési szálakat az 1.14. ábra szemlélteti.



1.14. ábra. ábra - Vezérlési szálak különböző típusú rendszerhívások esetén

1.3.6.2. Be-/kivitel végrehajtása

A multiprogramozás a CPU kihasználásának javítását úgy éri el, hogy egy program B/K-műveleteinek idején más programot futtat a processzoron. Természetesen a megvalósítás feltétele, hogy a hardverarchitektúra adjon lehetőséget a B/K-műveletek és a programutasítások valódi párhuzamos végrehajtására (B/K-vezérlőegységek, pufferelés, megszakítási rendszer).

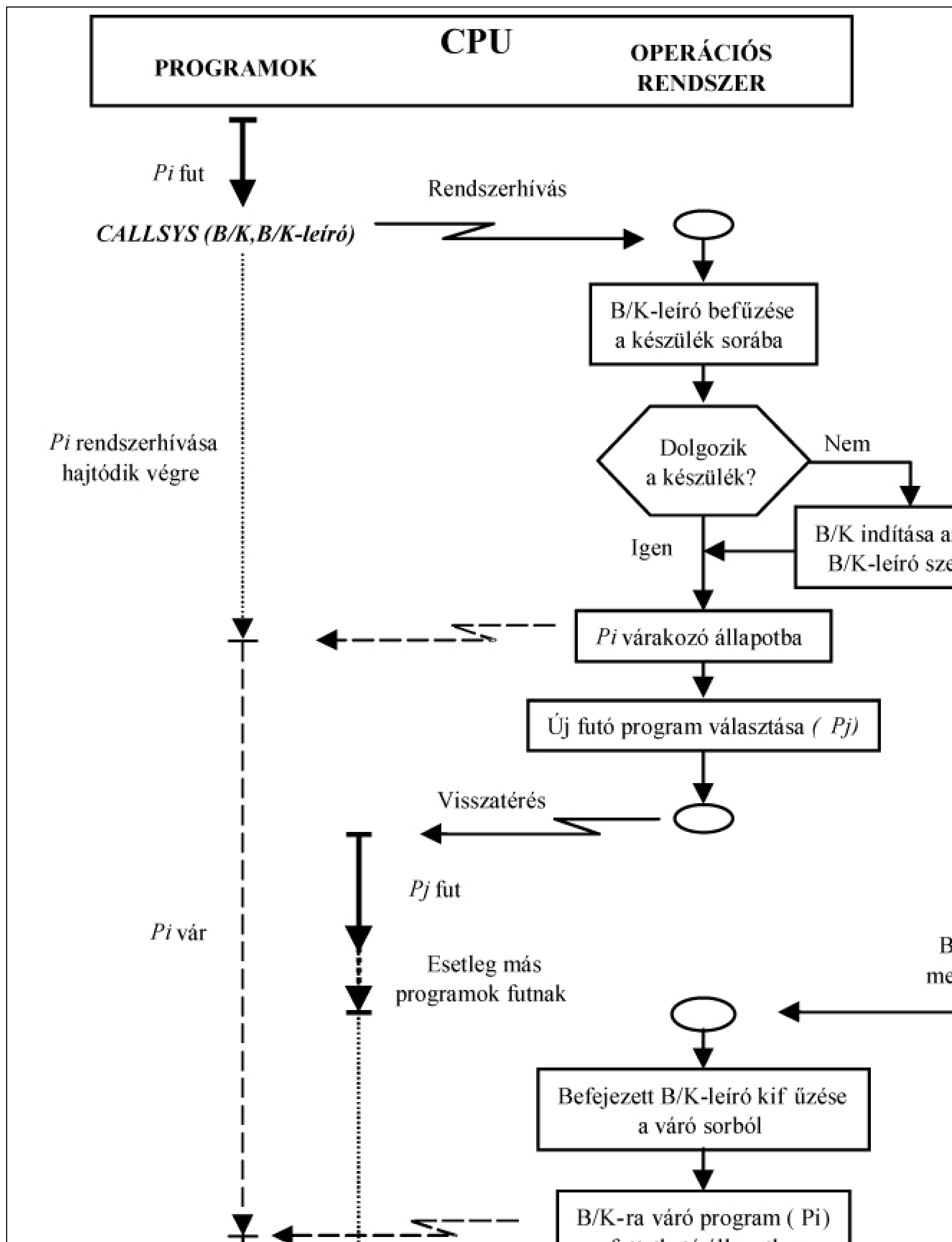
A B/K-műveletekre rendszerhívások állnak rendelkezésre. Vizsgáljuk a szinkron rendszerhívások esetét, tehát a hívó program akkor folytatódik (a rendszerhívást követő utasításon), amikor a B/K-művelet már befejeződött. A rendszerhívás hatására működésbe lépő operációs rendszer a kapott paramétereknek megfelelően elindítja a B/K-műveletet, például úgy, hogy felprogramozza és elindítja a B/K-készülék vezérlőegységét. Ettől kezdve a B/K-művelet az illesztési módnak megfelelő szervezésben (megszakításosan, DMA-használattal stb.) megy a maga útján. Az operációs rendszer megjegyzi, hogy az átvitel befejeződésére a hívó program vár, majd választ egy másik futtatható programot, és a „visszatérés rendszerhívásból" műveletet úgy hajtja végre, hogy ez a kiválasztott program folytatódjon.

A B/K-művelet befejeződését a B/K-vezérlő megszakításkéréssel jelzi. A megszakításkérés hatására ismét az operációs rendszer lép működésbe, észleli, hogy a B/K-művelet befejeződött.

(Megjegyzés: Ha a B/K-művelet karakterenkénti, vagy bájtonkénti megszakítással zajlik, akkor az operációs rendszer – a készülékhez rendelt megszakítási rutin – karakterenként működésbe lép, és programozottan viszi át a következő karaktert a memória és a B/K-készülék adatregisztere között. Ilyenkor az átvitel közben egyszerű „visszatérés megszakításból" művelettel zárul az operációs rendszer aktuális futama. Ha a megszakítási rutin észleli, hogy vége az átvitelnek, azaz éppen az utolsó megszakításra reagált, akkor kell végrehajtani azokat a lépéseket, amelyeket például DMA-szervezés esetén az egyetlen, átvitel végét jelző megszakítás kiszolgálásakor.)

Ez egyben azt jelenti, hogy a befejeződött B/K-műveletre várakozó program folytatható, és ezt az operációs rendszer megjegyzi. Nem jelenti azonban azt, hogy a várakozó programot azonnal folytatni kell, hiszen a B/K-megszakítás egy másik, futó programot szakított meg, a CPU tehát nem volt tétlen. Az, hogy végül mikor folytatódik a várakozó program, az operációs rendszer további döntésétől függ.

Némi bonyodalmat okoz, hogy egy program, amelyik egy másik program B/K-műveletével párhuzamosan éppen fut, elindíthat egy újabb B/K-műveletet ugyanarra a készülékre. Ilyenkor a rendszerhívás végrehajtásakor az operációs rendszer nem tud azonnal intézkedni az átvitel indításáról, hiszen a készülék még foglalt, vezérlőjének regiszterei még az előző átvitel adatait tárolják, átírásuk hibát okozna. A megoldás az, hogy az átvitel paramétereit meg kell jegyezni, a hívót ugyanúgy várakoztatni kell, mintha az átvitel azonnal indítható lett volna, és új futtatható programot kell választani. Az átvitel a megjegyzett paraméterekkel akkor indítható el, amikor a készülék felszabadul (ezt az operációs rendszer észleli az előzőekben elmondottak szerint).



CPU
PROGRAMOK
OPERÁCIÓS RENDSZER
Pi fut
Rendszerhívás
CALLSYS (B/K,B/K-leíró)
B/K-leíró befűzése a készülék sorába
Pi rendszerhívása hajtódik végre
Dolgozik a készülék?
Nem
Igen
B/K indítása az els ő B/K-leíró szerint
Pi várakozó állapotba
Új futó program választása (Pj)
Visszatérés
Pj fut
Pi vár
Esetleg más programok futnak
B/K vége megszakítás
Befejezett B/K-leíró kif űzése a váró sorból
B/K-ra váró program (Pi)

1.15. ábra. ábra - Be-/kivitel lefolyása multiprogramozott rendszerben

Továbbgondolva a lehetőségeket beláthatjuk, hogy a fenti ráindítás többszöröződhet. Egy adott pillanatban tehát nemcsak a CPU-ra várakozhat több program, hanem egy B/K-készülékre is több átviteli kérelem. Az operációs rendszernek ezért a B/K-készülékekre is várakozási sort kell szerveznie.

Egy B/K-rendszerhívás végrehajtásának menete a fentiek szerint a következőképpen foglalható össze:

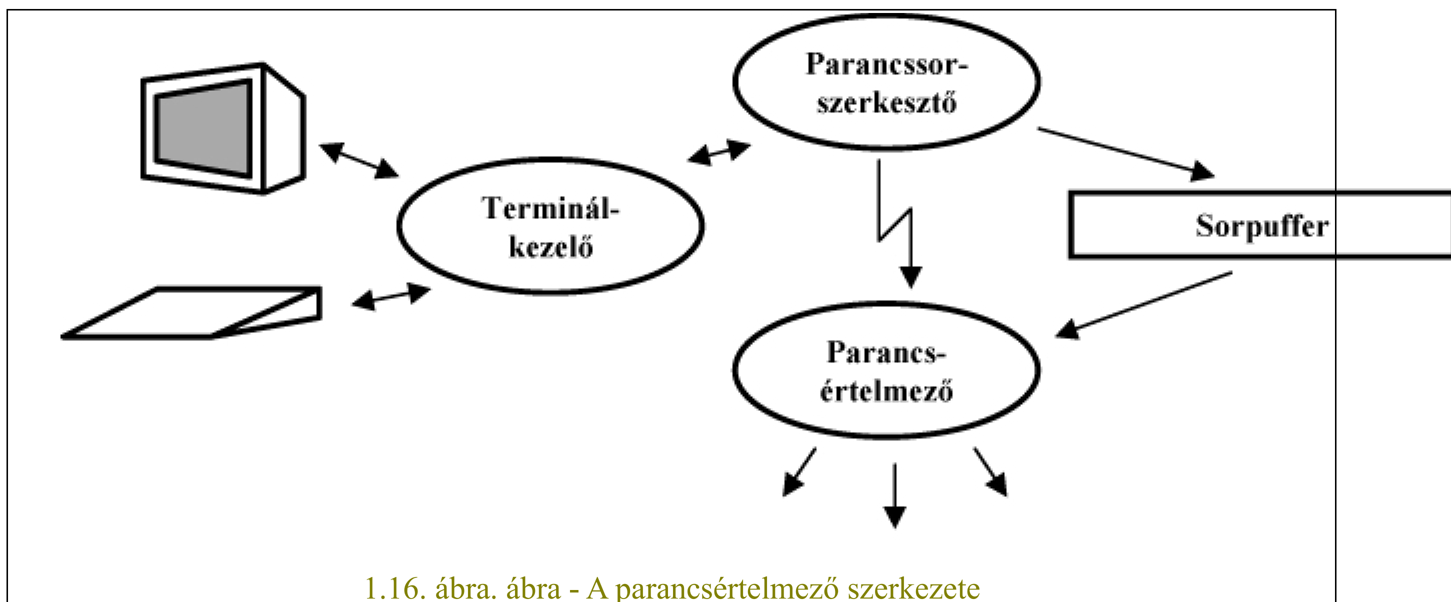
- az operációs rendszer működésbe lép a rendszerhívás hatására, és a következőket teszi:
    - a B/K-kérelmet az indításhoz szükséges paraméterekkel a készülék várakozási sorába helyezi,
    - ha a készülék szabad, elindítja a B/K végrehajtását,
    - választ egy futtatható programot, és oda tér vissza,
- az operációs rendszer B/K-megszakításokat fogad el, amelyekre reagálva:
    - B/K végét jelző megszakítás esetén futtathatóvá teszi a befejeződött B/K-ra várakozó programot,
    - ha van várakozó B/K-művelet a készülékre, elindítja a következő B/K-műveletet,
    - ütemezési stratégiájától függően vagy új programot választ futásra, vagy visszatér a megszakított programra,
- a B/K-rendszerhívást kiadó program akkor folytatódik, ha ismét futtathatóvá vált, és az operációs rendszer őt választotta futtatásra.

A folyamatot az 1.15. ábra szemlélteti.

1.3.6.3. Kezelői parancsok

A kezelői felületen az operációs rendszernek szóló parancsok érkezhetnek, valamint a kezelőnek szóló információk jelennek meg a rendszer állapotáról. A legegyszerűbb kezelői felületet a *parancsértelmező* nyújtja, jellegzetesen interaktív, karakteres eszközökön (terminálon). Egyfelhasználós, valamint interaktív, időosztásos rendszerekben az operációs rendszernek szóló parancsokat a felhasználó ugyanarról a terminálról adja, amelyik egyben az általa indított alkalmazásokkal való párbeszéd eszköze is. Ebben az esetben az operációs rendszerrel, illetve az alkalmazással folytatott dialógus elkülönítését meg kell oldani (időbeli elkülönítés, képernyő felosztása, ablakozás stb.).

Az egyszerű parancsértelmező felépítését és működését az 1.16. ábra mutatja.



1.16. ábra. ábra - A parancsértelmező szerkezete

Az operációs rendszer működése közben a terminál billentyűzetének megszakításkéréseit a terminálkezelő fogadja, leolvassa a leütött billentyű kódját, és átadja a parancssor-szerkesztőnek, amelyik egy belső tárban gyűjti és a képernyőn visszajelzi a beírt szöveget. A sorszerkesztő egyszerű javításokra (karakter visszatörlés, felülírás, beszúrás) lehetőséget ad. A parancs összeállítását követően egy érvényesítő végjel (szokásosan ENTER) leütésével jelzi a kezelő, hogy kéri a parancs végrehajtását. Ilyenkor a parancssor-szerkesztő átadja a belső tárat a parancsértelmezőnek, amelyik elemzi a sort (megkeresi és elkülöníti a paramétereket), és egy prioritási sorrend szerint megkísérli értelmezni és végrehajtatni a parancsot. A prioritás azt jelenti, hogy a parancsértelmező egy értelmezési sorrendet követ. Ha a parancs értelmes a rendszermag számára, akkor egy rendszerhívással elindítja a végrehajtást, ha nem, megkísérli rendszerprogramként értelmezni, ha így sem, akkor a parancsot egy alkalmazás nevének tekinti, és megpróbálja azt indítani.

A parancs végrehajtásának befejeződése után a tárolt parancssor törlődik és ismét a parancssor-szerkesztő kezd működni.

Ez a működés igen rugalmas abban a tekintetben, hogy az alkalmazások a rendszerparancsokkal azonos módon hajtathatók végre, azaz a kezelői felületen az operációs rendszer kiterjesztésének tűnnek.

A parancsvégrehajtás alapesetben szinkron szervezésű, azaz a parancs befejeződése előtt nem adható ki újabb parancs. Multitaszkos rendszerekben azonban lehetőség van aszinkron parancsindításra, amikor a kezelő egy parancs kiadását követően, annak befejeződése előtt újabb parancsokat adhat a rendszernek. Ilyenkor a parancsokat végrehajtó programok multiprogramozottan hajtódnak végre.

A bemutatott, legegyszerűbb kezelői felület sem része a kernelnek a legtöbb esetben, és egy operációs rendszerhez akár több kezelői felület is lehet használatban (lásd UNIX shell).

1.3.6.4. Külső megszakítások kezelése

A külső **megszakítás** egy számítógéprendszerben olyan eszköz, amelyik lehetővé teszi, hogy a rendszer gyorsan reagáljon egy előre nem látható időpontban bekövetkező külső jelzésre. A hardver a rendszerek nagy többségében prioritásos, vektoros megszakításkezelést biztosít.

Egy külső megszakítás elfogadásakor a processzor általában működési módot vált (rendszermódba kapcsol), és a megszakítási vektor által meghatározott címen folytatja a programvégrehajtást. Ezen a címen az operációs rendszer megfelelő része helyezkedik el és reagál a megszakításkérésre. A módváltás miatt fontos, hogy a teljes megszakítási rendszert – beleértve a vektortáblákat, a megszakítási programok címkiosztását is – az operációs rendszer kezelje, mert ellenkező esetben kijátszható lenne a védelmi rendszer. Ebből következik, hogy a megszakítási programok csak olyan reakciókat tartalmazhatnak, amelyek az operációs rendszer írásakor ismertek.

Sok esetben hasznos lehet, ha egyes felhasználói programokat, esetleg programrészleteket megszakítás hatására tudunk végrehajtatni. A rendszerek egy része ezért lehetőséget ad ilyen megoldásokra, azonban ezt áttételek beiktatásával teszi úgy, hogy a felhasználói programrészlet ne rendszer-módban fusson.

Külső megszakításkérések egy számítógéprendszerben jellegzetesen a B/K-készülékek működése során keletkeznek, amikor a készülék, vagy készülékvezérlő programozott kezelést igényel (például kész egy újabb adategység küldésére vagy fogadására, befejezte az átvitelt stb.).

1.3.6.5. Időmérés

Gyakorlatilag minden operációs rendszer tartalmaz időkezelési, időmérési funkciókat. Különösen fontos ez a lehetőség a valósidejű rendszerekben.

Szokásosan a rendszer vezet egy naptárat, amely kezelői paranccsal, vagy programmal lekérdezhető, illetve beállítható, és legalább másodperc felbontású. A további időkezelési feladatok között szerepelhet az egyes felhasználók processzoridő-felhasználásának nyilvántartása, késleltetett, illetve adott időpontra időzített programindítás, időzített jelzések, valamint egyes műveletekhez, eseményekhez rendelt időkorlát előírása a végtelen várakozások kiküszöbölésére.

Korrekt időmérés egy rendszerben csak hardverórával valósítható meg. Fizikailag az óra szorosan integrálódik a processzorhoz, rendszertechnikailag azonban általában egy speciális B/K-készülékként kezelhetjük. Az óra általában egy pontos frekvenciájú kvarc-oszcillátor impulzusait számolja és felprogramozással beállítható, hogy hány impulzusonként kérjen megszakítást. Így egy igen pontos periódusidejű megszakításkérő egységhez jutunk, amely-lyel szemben elvárás, hogy megszakításait a periódusidőn belül lekezelje a rendszer. Ezeknek a megszakításoknak a számlálására építhető fel a további, szoftveres időkezelés. A megszakítás periódusideje meghatározza az operációs rendszer időkezelésének felbontását. Speciális esetekben, ha egyes funkciók finomabb felbontású időkezelést igényelnek, a rendszer lekérdezhetővé teheti a fizikai óra számlálóját, vagy B/K-készülékként további programozható órák iktathatók a rendszerbe.

Minden időmérés elvi problémája, hogy az időpontot jelentő szám folyton növekszik, és előbb-utóbb minden tároló túlcsordul (lásd 2000. év probléma). A felbontás és az átfogás megválasztása egy-egy rendszerben több szempontot mérlegelő tervezői kompromisszum eredménye.

1.3.6.6. Hibamegszakítások kezelése (kivételkezelés)

A számítógéprendszer működése során előfordulhatnak olyan hibák, amelyeket a hardver észlelni képes, de amelyek nem okozzák feltétlen a teljes rendszer működésképtelenségét. Ilyenek lehetnek egyrészt azok a tranziens hibák, amelyek esetleg a művelet ismétlésével javíthatók, illetve a programok olyan hibái, amelyek csak futási időben fedhetők fel, és végrehajthatatlan műveletet eredményeznek (például nemlétező memóriacímre való hivatkozás, nullával osztás, privilegizált utasítás végrehajtásának kísérlete felhasználói módban stb.). Ezekben az esetekben a hardver nem vált működésképtelenné, de a program továbbfuttatása feltehetőleg értelmetlen. A processzorok ezért ilyen esetekben **hibamegszakítást** *(trap, exception)* generálnak, aminek hatására a program folytatása helyett egy megszakítási program indul el, ami az operációs rendszer része.

Az operációs rendszer a hibamegszakítás feldolgozása során

- visszaléphet a megszakított utasításra, ha tranziens hiba valószínűsíthető,
- hibajelzést küldhet a kezelői konzolra,
- abortálhatja a hibát okozó programot,
- elindíthatja a felhasználói program hibakezelő részét (kivételkezelő, exception-handler),
- diagnosztizáló programot indíthat.

Bizonyos esetekben a hibamegszakítás rendszerére komplex rendszerszolgáltatások építhetők. Ilyen például a későbbiekben részletezett virtuális tárkezelés, ahol a nemlétező címre való hivatkozás által okozott hibamegszakítás indítja el a keresett információ memóriából való betöltésének folyamatát.

A hibamegszakítások és a külső megszakítások között fontos különbség, hogy a hibamegszakítás *utasítás közben* keletkezik, és az aktuális utasítás nem tud befejeződni, míg külső megszakítást a processzor mindig *két utasítás között* fogad el, amikor a processzor az utasítás-végrehajtási ciklus kezdőpontján van. Ha az operációs rendszer a hibamegszakítást okozó program folytatása mellett dönt, a program tehát vagy a megszakítást okozó utasítás egy közbülső fázisától folytatható (ehhez meg kell őrizni az utasításon belüli processzorállapotot), vagy az utasítás ismétlésével, amihez nyomtalanul törölni kell az előző, félbemaradt végrehajtási kísérlet hatását (utasítás visszaállítás, rollback).

A kezelői konzolra küldött hibajelzések is okozhatnak problémákat. Sok esetben ugyanis a futó alkalmazás kezelőeszköze és az operációs rendszer kezelőeszköze ugyanaz a terminál. Különösen valósidejű, illetve beágyazott rendszerek esetén az alkalmazás kezelői felületén a felhasználó – esetleg például egy technológiát irányító diszpécser – az alkalmazás saját gondolatkörében mozog és dolgozik – például éppen technológiai

folyamatábrán követ egy gyártási folyamatot. Az operációs rendszer – miután írója nem készülhetett fel minden lehetséges alkalmazásra – csak uniformizált, szűkszavú, többnyire csak kézikönyvek és részletes szakismeret birtokában értelmezhető hibaüzeneteket tud előállítani és kiírni. Egy ilyen üzenet a diszpécsernek nem igazán informatív, és könnyen pánikhelyzetet okozhat. Hasonló rendszereknél ezért előnyös, ha a felhasználói programban megadható az a pont (kivételkezelés, exception), ahova az operációs rendszer a hibamegszakítás során visszalép, és ahol az alkalmazásfüggő hibakezelés a saját állapot- és változótérben programozható. A folyamat hasonló az 1.14. ábrán bemutatott eseménykezelés-átvételhez.

1.3.6.7. Rendszerindítás és leállás

Az operációs rendszer két sajátos működési állapota a rendszerindítás (boot) és a rendszerleállítás (shut-down) folyamata. Aki kapcsolt már be UNIX vagy Windows NT operációs rendszert futtató számítógépet, tudja, hogy bizony mind az indítás, mind a leállítás folyamán az amúgy villámgyors rendszer jócskán „molyol” valamin, mire végre feláll a rendszer, vagy mire kikapcsolhatjuk a gépet.

A bajok egyik forrása az, hogy a számítógépek memóriái ma általában felejtő memóriák, azaz kikapcsoláskor, vagy tápkimaradáskor tartalmuk törlődik. A rendszerben tápkimaradásokat és egyéb katasztrófákat leginkább „túlélő” (perzisztens) tároló a mágneslemez. Bekapcsoláskor tehát az üres memóriában fel kell építeni a rendszert, és ehhez igen sok lemezműveletre van szükség. Kikapcsoláskor ennek a fordítottja játszódik le. A memória és a háttértár (lemez) kezelésében ugyanis a rendszerek igen sokféle hatékonyságnövelő trükköt alkalmaznak (pufferelés, virtuális tár, swapping stb.), aminek következménye azonban az, hogy a lemezen tárolt fájlrendszer és a memóriakép sem lesz minden pillanatban konzisztens. Ha lehetőség van a „rendezett” visszavonulásra (kikapcsolási folyamat), akkor a lemez konzisztenciája és az üzemállapot lemezre írása biztosítható, így a következő indítás várhatóan problémamentes lesz. Ugyanakkor egy váratlan tápkimaradás egy nagyobb rendszert igen kellemetlenül érint, a szünetmentes tápegységek ezért ilyenkor nélkülözhetetlenek.

A másik időigényes bekapcsolási műveletsorozat tulajdonképpen biztonsági és kényelmi okok miatt szükséges. A korszerű operációs rendszerek ugyanis az indítás során feltérképezik a rendelkezésre álló hardvert és ellenőrzik annak működőképességét. Ezzel jelentős adaptivitás érhető el (plug and play), és megelőzhetők a futás közbeni kellemetlen meglepetések. Ebben a fázisban építi fel a rendszer a konfigurációnak megfelelően a készülékkezelőket és a fizikai készülékeket összekapcsoló rendszertáblákat, a megszakítási vektortáblát, amelynek bejegyzései a külső megszakítások kezelését a megfelelő készülékkezelő rutinra irányítják. A rendszer – ahol lehetséges – ki is próbálja a beállításokat, megszólítja a kapcsolódó eszközöket, és ellenőrzi a reakciókat.

2. Az operációs rendszer mint absztrakt, virtuális gép

Tartalom

2.1. Folyamatok és szálak
2.2. Folyamatokból álló rendszerek
    2.2.1. Folyamatok létrehozásának indokai
    2.2.2. Független, versengő és együttműködő folyamatok

2.2.3. Folyamatok születése és halála
2.2.4. Folyamatok együttműködése
2.2.5. Folyamatok szinkronizációja
2.2.6. Folyamatok kommunikációja
2.2.7. Holtpont
2.2.8. Éhezés
2.2.9. Klasszikus konkurens problémák
2.2.10. Folyamatokból álló rendszerek leírása nyelvi szinten
2.3. Tárak
2.3.1. Tárhierarchia
2.3.2. A logikai memória
2.3.3. A háttértár-fájlok
2.3.4. Tárak tulajdonságai
2.4. Készülékek és külső kapcsolatok
2.4.1. A külső eszközök fő típusai
2.4.2. Készülékmodell az alkalmazói felületen
2.4.3. Készülékmodell a kezelői felületen
2.5. Védelem és biztonság
2.5.1. Védelem
2.5.2. Biztonság

Mint az előző fejezetben megállapítottuk, az operációs rendszer egyik fő feladata, hogy egy kényelmesen kezelhető virtuális gépet jelenítsen meg a felhasználói és a programozói felületen. Ebben a fejezetben ennek a virtuális gépnek a fontosabb jellemzőit tárgyaljuk.

A tárgyalás során először most is szoftvertervezői szemlélettel közelítünk az operációs rendszerhez. Egy szoftverrendszer tervezése során a rendszert három nézetből kell leírni:

- *adatnézetből*, azaz fel kell térképezni és le kell írni a szereplőket és kapcsolataikat (objektumok, objektummodell, adatszerkezetek),
- a *műveletek, funkciók* oldaláról, azaz meg kell határozni az egyes szereplők által, vagy rajtuk végezhető műveleteket (funkcionális modell),
- a *viselkedés, vagy vezérlés* oldaláról, azaz le kell írni a működést, azt, hogy melyik művelet mikor hajtódik végre.

Fő rendező elvünk az „adatnézet” lesz, azaz sorra vesszük a legfontosabb fogalmakat, amelyeket az operációs rendszer megjelenít. Ezek a főszereplők a *folyamatok,* a *tárak,* a *készülékek* és a *felhasználók*. Természetesen az egyes pontokon belül foglalkozunk a fogalmakhoz kötődő funkciókkal és viselkedéssel is.

A fejezet végén foglalkozunk az operációs rendszerek legfontosabb *minőségi jellemzőivel*, ezek közül kiemelten a *védelem és biztonság* kérdésével.

2.1. Folyamatok és szálak

A korábbiakban megmutattuk, hogy a különböző típusú rendszerek (kötegelt, időosztásos, valósidejű stb.) egyaránt a multiprogramozás elvén értek el kielégítő hatékonyságot. A multiprogramozás alkalmazása azt jelentette, hogy a kötegelt feldolgozást végző rendszerekben egyidejűleg több *munka* végrehajtása van folyamatban, az időosztásos rendszeren egyidejűleg több *felhasználó* kiszolgálása van folyamatban, a valósidejű rendszerekben pedig egyidejűleg több külső eseményre reagáló *feladat* végrehajtása lehet folyamatban.

A feldolgozás egységeinek megnevezése a különböző rendszerekben más és más – például *munka (job), feladat (task)* –, azonban az operációs rendszer szintjén felmerülő problémák jórészt közösek. A megvalósítás közös problémái vezettek a valamennyi rendszerben egyaránt használható *folyamat* fogalmának kialakulásához, a szakirodalomban azonban a job, task, process kifejezések gyakran egymás szinonimájaként fordulnak elő. A folyamatmodellel szemben a több processzort tartalmazó, multiprocesszáló rendszerek megjelenése újabb követelményt támasztott, mégpedig azt, hogy a modell legyen használható mind multiprogramozott, mind multiprocesszáló rendszerek esetén.

A **folyamat (process)** tehát a multiprogramozott operációs rendszerek alapfogalma. A kifejezést azonban a köznyelv is, és más szakterületek is elterjedten használják. Folyamaton általában műveletek meghatározott sorrendben történő végrehajtását értjük. A folyamat tehát elkezdődik és befejeződik, közben pedig minden részművelet végrehajtása csak akkor kezdődhet el, ha az előző részművelet végrehajtása már befejeződött. A folyamat tehát önmagában szekvenciális. A folyamatok azonban haladhatnak egymással időben párhuzamosan. A folyamat kiterjedésének megválasztása elvileg önkényes. A folyamat egy szakasza kiválasztható és önmagában is folyamatként vizsgálható, illetve egymást követő folyamatok egyesíthetők, és egyetlen nagyobb folyamatként vizsgálhatók.

A folyamathoz kötődő további általános fogalom a *szál* fogalma (például a cselekmény több szálon fut). A műveletek sorrendjét szemléletesen úgy is meghatározhatjuk, hogy egy képzeletbeli fonalra egymás után felfűzzük a műveleteket jelképező gyöngyszemeket. Ezzel a hasonlattal a folyamat egy szálon történő végighaladásnak felel meg.

Az operációs rendszerekkel kapcsolatban a *folyamat* és a *szál* kifejezés természetesen szűkebb, konkrétabb értelemben használatos.

Az egyes rendszertípusok természetes feldolgozási egységei (egy job végrehajtása, egy felhasználó kiszolgálása, egy technológiai eseményre való reagálás) folyamatként vizsgálhatók. Ezen folyamatok közös eleme, hogy mindegyikben programok egymást követő végrehajtása történik. A program betöltése és végrehajtása az operációs rendszer egy tipikus művelete, amelyik valamennyi rendszertípusra jellemző. A program több szempontból is összetartozó feldolgozási egység (jól meghatározható funkciója van, többnyire egy fájlban tároljuk, végrehajtáskor egy címtartományt rendelünk hozzá stb.). Egy program végrehajtása az esetek többségében sorrendi, a programon belüli párhuzamosítás speciális esetekben fordul elő. Ezeknek a szempontok-

nak az alapján indokolt a különböző típusú rendszerek közös *folyamat* fogalmát a programok szintjén megragadni.

Az operációs rendszerek körében tehát *a folyamat egy program végrehajtása.* Pontosabban, a fogalom inkább tárgyiasult értelemben használatos, azaz *a folyamat egy végrehajtás alatt álló program*. A „végrehajtás alatt álló" úgy értendő, hogy végrehajtása már megkezdődött, de még nem fejeződött be. Így akár multiprogramozott, akár multiprocesszáló a rendszer, egyidejűleg több folyamatot kezel.

Egy program egy végrehajtása egy utasítássorozat végrehajtását jelenti. Ugyanannak a programnak – az adatfüggő elágazások miatt – többféle végrehajtása létezhet. A programkód alapján valamennyi lehetséges végrehajtás előállítható. Felvehetünk egy irányított gráfot, amelynek csomópontjai az utasítások, irányított élei pedig az utasításokat a végrehajtás lehetséges sorrendjében kötik össze (folyamatábra, vezérlési gráf). Egy végrehajtás jellemezhető egy *vezérlési szállal*, amelyik az irányított gráfon a program belépési pontjától egy befejeződési pontig vezető útnak felel meg.

A fentiek alapján a folyamatról a következő logikai modellt alkothatjuk:

Minden folyamathoz tartozik egy logikai processzor és egy logikai memória. A memória tárolja a programkódot, a konstansokat és a változókat, a programot a processzor hajtja végre. A programkódban szereplő utasítások és a végrehajtó processzor utasításkészlete megfelelnek egymásnak. Egy utasítás végrehajtását – néhány kivételtől eltekintve – oszthatatlannak tekintjük, azaz a folyamat állapotát csak olyan időpontokban vizsgáljuk, amikor egy utasítás már befejeződött, a következő pedig még nem kezdődött meg. A programvégrehajtás egy vezérlési szál mentén, szekvenciálisan történik, alapvetően az utasítások elhelyezkedésének sorrendjében, ettől speciális utasítások esetén van eltérés. A processzornak vannak saját állapotváltozói (programszámláló, veremmutató, regiszterek, jelzőbitek stb.), amelyek értéke befolyásolja a következő utasítás végrehajtásának eredményét.

A memória a RAM-modell szerint működik, azaz

- tárolórekeszekből áll,
- egy dimenzióban, rekeszenként címezhető,
- csak rekeszenként, írás és olvasás műveletekkel érhető el,
- az írás a teljes rekesztartalmat felülírja az előző tartalomtól független új értékkel,
- az olvasás nem változtatja meg a rekesz tartalmát, tehát tetszőleges számú, egymást követő olvasás az olvasásokat megelőzően utoljára beírt értéket adja vissza.

A folyamatot egy adott pillanatban leíró információk (a folyamat állapottere):

- a memória tartalma, azaz a végrehajtandó programkód, valamint a változók pillanatnyi értéke,

- a végrehajtó processzor állapota (programszámláló állása, további regiszterek, jelzőbitek stb. értéke).

Az operációs rendszer feladata, hogy a fizikai eszközökön (fizikai processzor, fizikai memória) egymástól elkülönítetten (védetten) létrehozza és működtesse a folyamatoknak megfelelő logikai processzorokat és memóriákat. A logikai processzor utasításkészlete a fizikai processzor utasításkészleténél egyrészt szűkebb, mivel nem tartalmazza a privilegizált utasításokat, másrészt ugyanakkor bővebb is, mivel az operációs rendszer szolgáltatásait is tartalmazza (lásd be-/kiviteli műveletek stb.). A logikai memória a fizikai tár akár nem összefüggő címtartományára, sőt esetleg háttértár címekre lehet leképezve, ráadásul a leképezés a végrehajtás során esetleg változhat is.

Feltűnhet, hogy ebből a modellből hiányzik a külvilággal tartott kapcsolat eszközrendszere, azaz a be-/kivitel. A tárgyalás ezen szintjén a B/K-műveleteket a logikai processzor utasításkészlete részének tekintjük, hiszen a B/K-műveletekre rendszerhívások állnak rendelkezésre. Valójában a be-/kivitel hatékony szervezése az operációs rendszerek legbonyolultabb funkciói közé tartozik.

A folyamat logikai modellje mind multiprogramozott, mind multiprocesz-szoros rendszerek esetén használható. A multiprogramozott rendszerek jellemzője, hogy egyetlen fizikai processzoron, valamint a hozzá tartozó memórián és perifériákon kell egyidejűleg több folyamatot végrehajtani, azaz több logikai processzort és memóriát működtetni. Ezt úgy is felfoghatjuk, hogy az operációs rendszer a logikai-fizikai leképezés mellett a folyamatok számára erőforrásokat biztosít, azaz erőforrás-kiosztó, illetve gazdálkodási feladatot is ellát.

Multiprocesszoros rendszerek esetén az egyes folyamatoknak megfelelő logikai processzorokat több fizikai processzorra lehet szétosztani. Az erőforrás-gazdálkodás itt több processzorral, valamint a hozzájuk tartozó memóriával való gazdálkodást jelenti. Összefoglalóan:

- Az operációs rendszerek körében a folyamat egy végrehajtás alatt álló – „életre kelt” – program.
- A folyamat létrejön (megszületik), amikor a végrehajtás megkezdődik, és megsemmisül (meghal), amikor a végrehajtás befejeződik.
- A folyamatot egy memória–processzor együttes működésével modellezhetjük, amelynek állapotleírói a memóriatartalom (a végrehajtandó kód és a változók értéke), valamint a processzor állapota (a programszámláló és a regiszterek értéke).

Egyes operációs rendszerek lehetőséget adnak **szálak (thread)** létrehozására is. A szálak lényegében párhuzamos végrehajtású, közös memóriát használó programrészek a folyamatokon belül (egy program végrehajtása több szálon futhat). A szálaknak saját logikai processzoruk van (a CPU-ért ugyanúgy versenyeznek, mint a folyamatok), azonban memóriáik nincsenek védetten elkülönítve, közös logikai memóriát használnak, azaz a kódon és változókon osztoznak. Emiatt – és ez a szálak alkalmazásának gyakorlati jelentősége – az operációs rendszer lényegesen gyorsabban tud végrehajtani egy átkapcsolást a szálak között, mint a folyamatok között.

A szál és a folyamat megkülönböztetésére használatos a **pehelysúlyú** *(lightweight)* és **nehézsúlyú** *(heavyweight)* folyamat elnevezés is. A továbbiakban nem teszünk éles különbséget a folyamatok és szálak között, a szálakat általában önálló folyamatoknak tekintjük.

2.2. Folyamatokból álló rendszerek

Egy számítógéprendszeren belül általában egyidejűleg több folyamat van jelen. Ezek a folyamatok különbözőképpen keletkezhettek és különböző kapcsolatok lehetnek közöttük. Ha a rendszer célja csak a hatékony erőforrás-kihasználás, a folyamatok kezelése az operációs rendszer belügye maradhat. A felhasználó (az egyszerű felhasználó és a programfejlesztő) nem is találkozik folyamatokkal, számára csak futtatandó programok, esetleg *jobok* léteznek. A folyamatkezelésben csak az operációs rendszert készítő programozók és a rendszermenedzserek érintettek. Más a helyzet, ha a felhasználó is létrehozhat olyan folyamatokat, amelyek együttműködve oldanak meg egy feladatot. Ilyenkor a kezelői és programozói felületen is meg kell jeleníteni a folyamatkezelést, valamint a folyamatok együttműködésének szervezését segítő funkciókat. Azokat az operációs rendszereket, amelyek a felhasználó számára is lehetővé teszik a folyamatkezelést, *multitaszkos* rendszereknek szokták nevezni.

2.2.1. Folyamatok létrehozásának indokai

Láttuk, hogy történetileg a multiprogramozás kialakulását a processzorkihasználás javítása motiválta. Emellett más szempontok is indokolhatják, hogy egy rendszerben több folyamatot hozzunk létre. Melyek tehát az indokok?

- *Hatékonyabb erőforrás-kihasználás.* Ez a processzorkihasználás javításának egy általánosabb megfogalmazása. A processzoron kívül lehetnek a rendszerben más, nagy értékű erőforrások, amelyeket egyetlen folyamat csak működésének egy rövidebb szakaszában használ, a többi időben indokolt a használatot mások számára lehetővé tenni. Egy feladatmennyiség egy adott erőforráskészlettel akkor oldható meg a leghatékonyabban, ha lehetőleg minden eszköz (erőforrás) minden pillanatban valamilyen hasznos feladatot végez, és nem tétlen.
- *A feladat-végrehajtás gyorsítása.* Az erőforrások hatékony kihasználása általában a feladatmegoldást is gyorsítja. Ezen túlmenően a feladatoknak kereshetjük azokat a megoldásait, amelyek párhuzamosan megoldható részfeladatokra bontják az eredeti problémát. Ha ilyenkor ezeket a részfeladatokat az erőforrások többszörözésével, valódi párhuzamossággal, folyamatokként tudjuk végrehajtani, további jelentős gyorsítást érhetünk el.
- *Többféle feladat egyidejű végrehajtása.* A felhasználók számára hasznos és kényelmes lehet, ha a számítógépet egyidejűleg többféle célra használhatják (például egy hosszadalmas számítás közben szövegszerkesztés végezhető).
- *Rendszerstrukturálás.* Bizonyos feladatokra könnyebb olyan szerkezetű programrendszert készíteni, amelyiken belül több folyamat működik együtt, mindegyik a feladat egy leválasztható

részén dolgozik, csak meghatározott pontokon cserélnek információt. Ilyen feladatok elsősorban a valósidejű rendszerek körében fordulnak elő, ha a rendszernek a környezet többféle, egymástól függetlenül bekövetkező eseményére kell reagálnia (például irányítórendszerek, több kezelőt interaktívan kiszolgáló rendszerek).

A folyamatok létrehozása mellett szóló érvek mellett meg kell említeni egy lényeges ellenérvet is. A folyamatokból álló rendszer fejlesztése – elsősorban a tesztelés fázisában – lényegesen nehezebb és bonyolultabb feladat, mint a szekvenciális programoké. A folyamatok aszinkron futása miatt ugyanis a rendszer egy adott lefutása gyakorlatilag reprodukálhatatlan, a szisztematikus tesztelés és hibakeresés ezért igen nehéz.

2.2.2. Független, versengő és együttműködő folyamatok

A rendszer folyamatai egymáshoz való viszonyukat, a köztük lévő csatolás erősségét tekintve lehetnek *függetlenek, versengők* vagy *együttműködők*.

A *független folyamatok* egymás működését semmilyen módon nem befolyásolják. Végrehajtásuk teljes mértékben *aszinkron*, azaz egymással párhuzamosan is végrehajtódhatnak, de a végrehajtás egymáshoz viszonyított sebességéről semmilyen feltételezést nem tehetünk. Számunkra az ilyen folyamatok gyakorlatilag érdektelenek, hiszen ezek külön-külön, önálló programokként vizsgálhatók.

A *versengő folyamatok* nem ismerik egymást, de közös erőforrásokon kell osztozniuk. Versengő folyamatok alakulnak ki például egy kötegelt feldolgozást végző rendszerben, a véletlenszerűen érkező, egymást nem ismerő felhasználói *jobok* feldolgozásakor. Az egyes *jobok* részeként elinduló programok ugyanazon a számítógépen hajtódnak végre egy multiprogramozott operációs rendszer felügyelete alatt. Ezeknek a folyamatoknak nem kell tudniuk még azt sem, hogy őket multiprogramozottan fogják végrehajtani, ezért programkódjuk ugyanolyan, mintha egy soros feldolgozást végző rendszerre írták volna. A több egyidejűleg működő folyamat helyes és hatékony futtatásának problémáit az operációs rendszeren belül kell megoldani (például minden folyamatnak külön memóriaterülete legyen, a nyomtatásaik ne gabalyodjanak össze, lehetőleg minden erőforrás munkában legyen stb.). Ezt a bonyolult erőforrás-kiosztási, gazdálkodási, védelmi és koordinációs feladatot az operációs rendszereken belül gyakran *együttműködő* folyamatokkal oldják meg. Az operációs rendszer saját, belső folyamatait *rendszerfolyamatoknak*, a többit *felhasználói folyamatoknak* nevezzük. A korrekt és biztonságos erőforrás-kezelés a folyamatok aszinkron futásának korlátozását, *szinkronizálását* igényli (például, ha egy folyamat nyomtatni akar, amikor egy másik folyamat éppen összetartozó adatsorozatát nyomtatja ki, meg kell várnia, amíg a másik folyamat nyomtatóhasználata befejeződik).

Az *együttműködő folyamatok* ismerik egymást, együtt dolgoznak egy feladat megoldásán, információt cserélnek. Az együttműködő folyamatokból álló rendszert tervszerűen bontottuk folyamatokra, amelyektől ezért a tervező szándéka szerinti kooperatív viselkedés várható el. Ezekben az esetekben a folyamatok egymástól való védelmének jelentősége kisebb, párhuzamosan működő programrészekként szálak is alkalmazhatók. Együttműködő folyamatok esetén a folyamatok leírása és az együttműködés műveletei a programkódban is

megjelennek, azaz a logikai processzor utasításkészletében szerepelnie kell az ehhez szükséges utasításoknak (például folyamat indítása, erőforrás kizárólagos használatának kérése, üzenetküldés egy másik folyamatnak stb.). Valójában ezeket a műveleteket az operációs rendszer hajtja végre. Az együttműködéshez szükséges funkciókon kívül természetesen a versengő folyamatoknál már említett erőforrás-kezelési feladatokat is el kell látnia az operációs rendszernek.

Együttműködő folyamatok munkájukat információcsere útján tudják összehangolni. A cserélt információ esetenként egyetlen bitnyi, csupán jelzés jellegű, máskor akár több megabájt is lehet.

2.2.3. Folyamatok születése és halála

Mindeddig nem foglalkoztunk azzal a kérdéssel, hogy *hogyan jönnek létre és hogyan szűnnek meg* a rendszert alkotó folyamatok. Vizsgáljuk most ezt a kérdést az egyszerűség kedvéért egyetlen fizikai processzort tartalmazó, azaz multiprogramozott rendszerre.

A Neumann-elven működő számítógépek soros feldolgozást végeznek. Amikor egy ilyen számítógépet bekapcsolunk, elindul egy rendszerépítési folyamat (boot, inicializálás). Ezt egy ősfolyamatnak tekinthetjük, amelyik létrehozza a rendszert alkotó többi folyamatot. A rendszerépítés eredményeként már egy működésre kész operációs rendszer alakul ki, amelyik maga is több folyamatból állhat (rendszerfolyamatok).

A működésre kész operációs rendszerben például interaktív kiszolgálás esetén minden terminálhoz tartozhat egy rendszerfolyamat, amelyik felhasználói parancsokat fogad és hajt végre. Kötegelt feldolgozás esetén elindulhat egy *jobkészletet* kezelő folyamat, amelyik *jobok* végrehajtását kezdi meg és *jobonként* egy újabb folyamatot indít. Valósidejű rendszer esetén az operációs rendszer saját felépülése után létrehozza és inicializálja a felhasználói rendszert alkotó folyamatokat.

A rendszerfolyamatok tehát újabb, felhasználói folyamatokat hoznak létre, a felhasználói folyamatok pedig – bizonyos típusú rendszerekben – maguk is létrehozhatnak további felhasználói folyamatokat a logikai processzor megfelelő utasításának végrehajtásával (például *fork, create*).

A folyamatok általában saját jószántukból fejezik be működésüket a logikai processzor megfelelő (például *exit*) utasításának végrehajtásával. Bizonyos esetekben (például működési hibák) azonban szükség lehet arra, hogy más folyamat szüntessen meg egy folyamatot (például *kill*).

Azokat a rendszereket, amelyek működése során – a felépülés és inicializálás kezdeti szakaszától eltekintve – nem jönnek létre és nem szűnnek meg folyamatok, *statikus* rendszereknek nevezzük, szemben a *dinamikus* rendszerekkel, amelyekben működés közben bármikor születhetnek és megszűnhetnek folyamatok.

A folyamatok eredetét szükség esetén a szülő–gyermek viszonyokkal, azaz egy fa struktúrával írhatjuk le (processzgráf). Dinamikus rendszerben a fa természetesen folyamatosan követi a folyamatok születését és halálát. Sok operációs rendszer a szülő–gyermek viszonyokra építi erőforrás-gazdálkodási és jogosultsági filozófiáját. Ennek egyik megközelítése a hierarchikus erőforrás-gazdálkodás, amikor a gyermek folyamatok

csak a szülő erőforrásaiból részesülhetnek és nem létezhetnek önállóan, csak amíg szülőjük is létezik. Egy másik megközelítés a globális erőforrás-gazdálkodás, amikor a rendszer valamennyi folyamata létrejötte után egyenrangú, önálló szereplő, és versenyezhet a teljes erőforráskészletből való részesedésért.

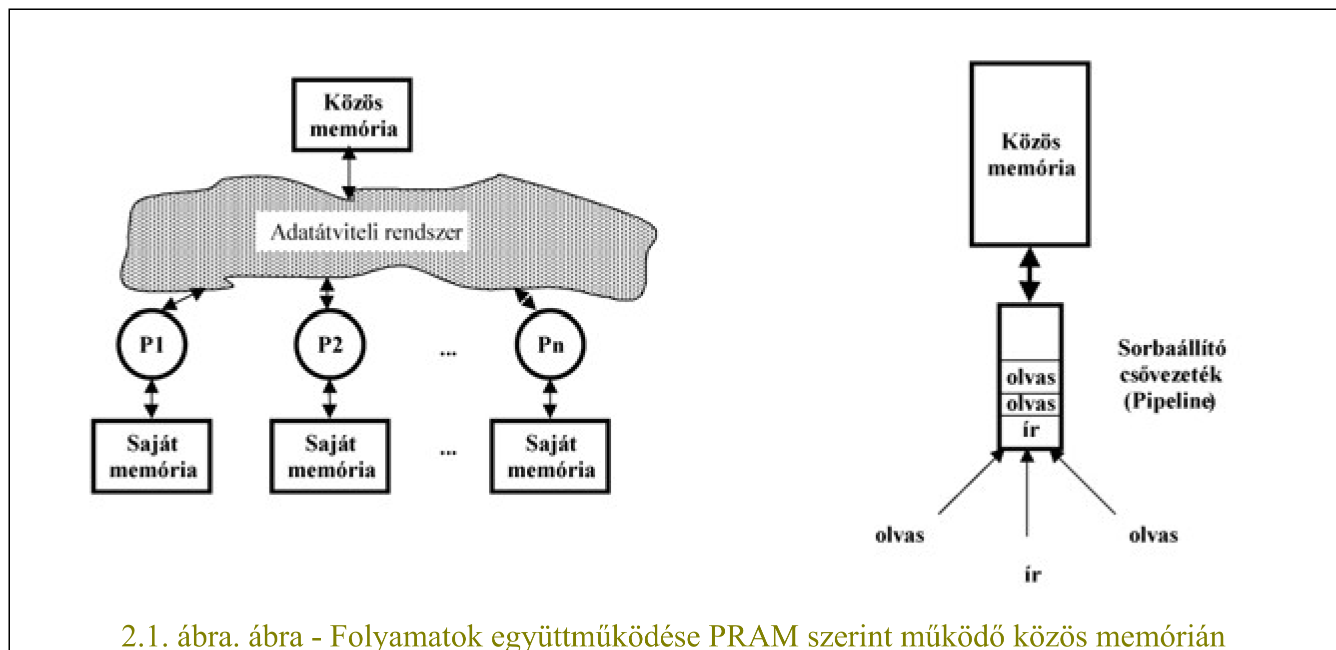
2.2.4. Folyamatok együttműködése

A folyamatok együttműködése információátadással valósul meg. A folyamatok közötti információcserének két alapvető módja alakult ki, a *közös memórián* keresztül, illetve az *üzenetváltással* történő adatcsere (később mindkettőt részletesen tárgyaljuk). Valamennyi konkrét megoldás – a legegyszerűbb, processzorok közötti jelzőbittől a nagy adatbázisokon végzett tranzakciókig vagy a földrészeket átfogó hálózati alkalmazásokig – erre a két alapsémára vezethető vissza.

2.2.4.1. Együttműködés közös memórián

Közös memórián keresztül történő adatcsere esetén az együttműködő folyamatok mindegyike saját címtartományában lát egy közös memóriát. A közös memória elérését (a közös memória címtartományára kiadott *olvasás* vagy *írás* művelet végrehajtását) valamilyen adatátviteli rendszer (kapcsolóhálózat, sín stb.) teszi lehetővé.

A folyamatok párhuzamos futása miatt a közös memóriát egyidejűleg több folyamat is írhatja, illetve olvashatja. Ilyen esetekre a RAM-modell nem határozza meg a memória működését, ezért a közös memóriát a RAM-modell kiterjesztésével kapott **PRAM** *(Pipelined Random Access Memory)* modell szerint kell kialakítani.



2.1. ábra. ábra - Folyamatok együttműködése PRAM szerint működő közös memórián

A PRAM-modell szerint működő memóriát több processzor írhatja és olvashatja egyidejűleg. Az *olvas* és *ír* műveletek egyidejű végrehajtására a RAM-modellhez képest az alábbi kiegészítések vonatkoznak:

- *olvasás-olvasás* ütközésekor mindkét olvasás ugyanazt az eredményt adja, és ez megegyezik a rekesz tartalmával,
- *olvasás-írás* ütközésekor a rekesz tartalma felülíródik a beírni szándékozott adattal, az olvasás eredménye vagy a rekesz régi, vagy az új tartalma lesz, más érték nem lehet,
- *írás-írás* ütközésekor valamelyik művelet hatása érvényesül, a két beírni szándékozott érték valamelyike írja felül a rekesz tartalmát, harmadik érték nem alakulhat ki.

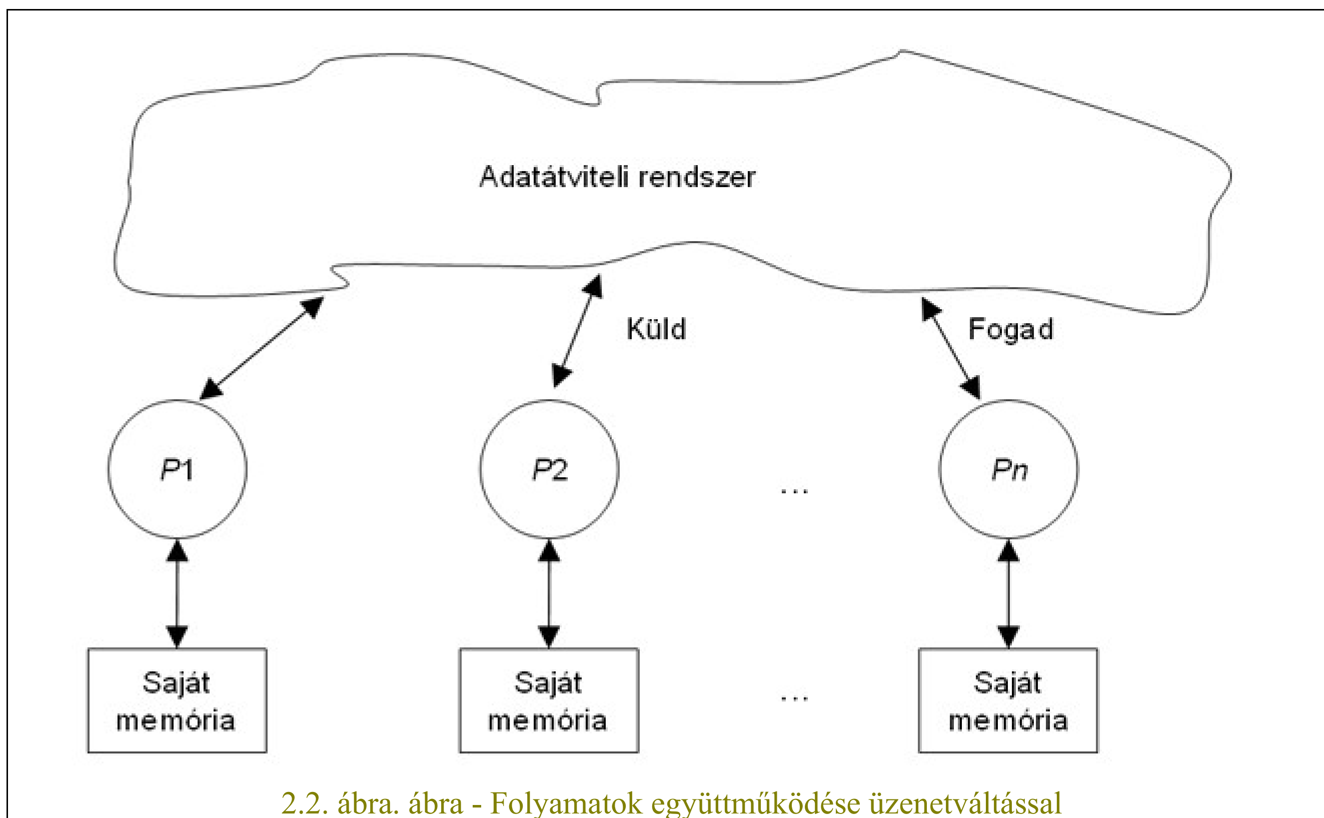
Ezek a szabályok lényegében azt jelentik, hogy az egyidejű műveletek nem interferálhatnak, azaz nem lehet közöttük zavaró kölcsönhatás. Hatásuk olyan lesz, mintha valamilyen előre nem meghatározható *sorrendben* hajtódnának végre (ezt tükrözi a *pipelined* elnevezés, arra utalva, hogy a memóriához egy sorosítást végző csővezetéken jutnak el a parancsok). Másként fogalmazva, ezek a műveletek a modell szintjén oszthatatlanok (atomiak).

A közös memória használatával történő adatcsere helyes lebonyolításához a PRAM-modell szerint működő memória mellett a folyamatok működésének összehangolása is szükséges (például az adat fogadója akkor olvassa el az adatot, ha a küldő már elhelyezte azt; összetett adatok átadásakor félig átírt rekordot ne kezdjen olvasni a fogadó stb.). Ez ismét a folyamatok szabadon futásának (aszinkronitásának) korlátozását jelenti, azaz *szinkronizációt* igényel.

2.2.4.2. Együttműködés üzenetváltással

Üzenetváltásos adatcsere esetén a folyamatoknak nincs közös memóriája. Az adatátviteli rendszer most a logikai processzorokat kapcsolja össze. Rajta keresztül a folyamatok üzeneteket tudnak küldeni, illetve fogadni. Az üzenetküldésre, illetve fogadásra a folyamatok logikai processzorainak utasításkészletében megfelelő utasítások állnak rendelkezésre. Legyenek ezek a *Küld (Send)* és a *Fogad (Receive)* műveletek.

A *Küld(<cím>,<folyamat>)* művelet végrehajtásakor a műveletet végrehajtó folyamat elküldi a saját memóriájának megadott címén tárolt adatot a megadott folyamatnak, a *Fogad(<cím>,<folyamat>)* művelet pedig végrehajtásakor a megadott folyamattól érkező üzenetet a saját memória megadott címén tárolja.



2.2. ábra. ábra - Folyamatok együttműködése üzenetváltással

Míg gyakorlatilag valamennyi közös memóriás információcserét alkalmazó megoldás a PRAM-modellre alapul, az üzenetközvetítésre nincs egyetlen elfogadott modell. Ha csak a működés logikájával foglalkozunk, és olyan lényeges kérdéseket nem is érintünk, mint az átviteli sávszélesség, az összeköttetés megbízhatósága, az üzenetszerkezet, az átviteli közeg sajátosságai, a kapcsolatépítés esetleges dinamikája, akkor is számos tulajdonságot találunk, amelyekben az egyes megoldások eltérhetnek, és amelyek ismerete fontos a felhasználók (rendszertervezők, programozók, üzemeltetők) számára. A folyamatok kommunikációs műveleteinek tulajdonságaival ezért külön pontban foglalkozunk.

2.2.5. Folyamatok szinkronizációja

A közös erőforrások használata, valamint a folyamatok közötti közös memóriás információcsere biztonsága és helyessége érdekében a folyamatok aszinkron „szabadonfutását” esetenként korlátozni kell. A műveletvégrehajtásra vonatkozó időbeli korlátozásokat nevezzük szinkronizációnak. A korlátozások alapesetei a következők:

Kölcsönös kizárás (mutual exclusion)

Több folyamatban lehetnek olyan utasítássorozatok **(kritikus szakaszok)**, amelyek egyidejű (konkurens) végrehajtása nem megengedett, azaz ezeknek az utasítássorozatoknak a végrehajtása kölcsönösen ki kell, hogy zárja egymást.

Kölcsönös kizárás szükséges jellegzetesen a megosztottan használt erőforrásokon végzett oszthatatlan műveletsorozatok esetén (például, ha egy folyamat megkezdett egy nyomtatást, ki kell zárni, hogy más folyamat

közbenyomtathasson egy-egy sort, vagy a közös memóriában tárolt, összetett adatszerkezetek kezelésekor, az adatbázis-tranzakciók végrehajtásakor meg kell akadályozni, hogy egyes folyamatok félig átírt, inkonzisztens rekordokat lássanak). Ezeknek a műveletsorozatoknak az idejére az adott erőforrás használatára kizárólagosságot kell biztosítani a folyamat számára.

Kicsit pontosabb fogalmazásban: egy folyamatokból álló rendszerben egy K kritikus szakasz folyamatok végrehajtási szakaszainak egy S_k halmazát jelenti, amely S_k halmaz bármely két $s_{k,i}$ és $s_{k,j}$ elemének átlapolt végrehajtása tilos. Ha egy folyamat $s_{k,i} \in S_k$ szakaszának végrehajtása megkezdődik, azt mondjuk, hogy a folyamat *belépett* a K kritikus szakaszba. Hasonlóan $s_{k,i}$ végrehajtásának befejeződésekor azt mondjuk, hogy a folyamat *kilépett* a K kritikus szakaszból. Egy folyamat csak akkor léphet be a K kritikus szakaszba, ha egyetlen más folyamat sem tartózkodik éppen K-ban. Ellenkező esetben meg kell várnia, amíg a bent tartózkodó folyamat elhagyja K-t. Ha egy folyamat elhagyja K-t, az esetleges belépésre várakozó folyamatok közül egyetlen léphet be K-ba.

Fentiekből következik, hogy egy rendszerben több, egymástól független kritikus szakasz létezhet. Minden kritikus szakaszhoz az egymást kizáró végrehajtási szakaszok egy halmaza tartozik. Egy K kritikus szakaszba való belépésnek nem feltétele az, hogy más *(L, M, N stb.)* kritikus szakaszokban éppen tartózkodik-e folyamat és melyik. Az is lehetséges, hogy egy folyamat egyszerre több kritikus szakaszban tartózkodik (például több kizárást igénylő erőforrást használ egyidejűleg).

Megjegyezzük, hogy a folyamat szekvenciális végrehajtása miatt (a szálakat önálló folyamatnak tekintve) egy folyamaton belül a kölcsönös kizárás bármely két utasítás között teljesül, folyamaton belüli két utasítássorozat kölcsönös kizárásának előírása értelmetlen.

A kritikus szakaszokat a programozók definiálják, a belépési szándékot és a szakaszból való kilépést a kódban elhelyezett műveletekkel jelölik. Versengő folyamatok esetén ez nem más, mint kizárólagos használat kérése, illetve ennek feloldása bizonyos eszközökre [például *Lefoglal(nyomtató), Felszabadít(nyomtató)*]. A műveletek rendszerhívásokat jelölnek, és futási időben az operációs rendszer hozza meg a döntést a folyamat továbbengedéséről vagy várakoztatásáról. Együttműködő folyamatok elvileg bármilyen egyeztetett megoldást használhatnának a probléma megoldására. A gyakorlatban számukra is kialakultak az operációs rendszer szinkronizációs rendszerhívásai, amelyek már nem kötődnek konkrét eszközhasználathoz, csak koordinációs szerepük van (lásd később).

Egyidejűség (randevú)

Különböző folyamatokban elhelyezett műveletekre előírhatjuk, hogy azok várják be egymást és „egyidejűleg" hajtódjanak végre. Kellően finom időléptékben természetesen az egyidejűség értelmezése problematikus. Az egyidejűség pontosabban úgy értendő, hogy egyik folyamat sem léphet túl a benne egyidejű végrehajtásra kijelölt végrehajtási szakaszon mindaddig, amíg valamennyi többi folyamat meg nem kezdte a saját kijelölt szakaszának végrehajtását.

Az egyidejűség tipikus alkalmazása az átmeneti tárolót nem tartalmazó adatátvitel *Küld* és *Fogad* műveleteinek végrehajtása, valamint a távoli eljáráshívás kiadása és elfogadása (lásd később).

Előírt végrehajtási sorrend (precedencia)

Különböző folyamatok kijelölt műveletei között végrehajtási sorrendet írhatunk elő. Legyen például *P1* folyamat egy utasítása *S1*, *P2* egy utasítása pedig *S2*. Az *S1 => S2* precedencia előírása azt jelenti, hogy *S1* végrehajtásának be kell fejeződnie, mielőtt *S2* végrehajtása megkezdődik. Ha az egyébként aszinkron futású *P1* és *P2* úgy hajtódna végre, hogy *S2* hamarabb kerülne sorra, *P2*-nek meg kell várnia, míg *P1*-ben *S1* végrehajtása befejeződik.

A precedencia előírása jellegzetesen annak biztosítására használatos, hogy egy folyamat csak akkor kezdje felhasználni a másik folyamat által előállított adatokat, amikor azok már rendelkezésre állnak.

2.2.5.1. Megoldások a PRAM-modell keretei között (szoftvermegoldások)

A szinkronizáció igénye először a multiprogramozott operációs rendszeren belül a kölcsönös kizárás problémájának formájában vetődött fel. A multiprog-ramozott rendszer folyamatainak kézenfekvő együttműködési módja a közös memória használata. A probléma megoldását ennek megfelelően a PRAM-modell szerinti közös memóriát használó folyamatokra keresték – további hardveres támogatás nélkül. A szinkronizáció ebben a felfogásban a folyamatok felelőssége. A folyamatok egyezményes adatszerkezeteket használnak a probléma megoldására, és a szinkronizációs pontokon egyezményes műveletsorozatot (protokoll) hajtanak végre. Ezért említi az irodalom ezeket szoftvermegoldásokként.

(Megjegyzés: a szoftvermegoldások bemutatását elsősorban didaktikai szempontok indokolják, hiszen a mai megoldások hardvertámogatásra épülnek.)

Lássuk tehát a kölcsönös kizárás problémájának megoldási kísérleteit!

A kölcsönös kizárás megoldásaival szemben a következő általános elvárásokat támasztjuk:

- minden körülmények között teljesüljön a kölcsönös kizárás,
- a belépési protokoll döntéshozatala csak a belépésben érdekelt folyamatok részvételét igényelje, azaz a többi folyamat nyugodtan haladhasson anélkül, hogy foglalkozniuk kellene a kérdéssel,
- véges időn belül minden belépni szándékozó folyamat bejuthasson (ettől esetenként eltekinthetünk).

Kézenfekvő megoldásnak tűnik kritikus szakaszonként egy *foglaltságjelző bit* elhelyezése a közös memóriában *szabad* kezdőértékre állítva. A kritikus szakaszba belépni szándékozó folyamat teszteli a jelzőt, ha szabad, akkor átállítja foglaltra és belép a kritikus szakaszba, kilépéskor pedig visszaállítja szabadra. A folyamatok tehát a következőképpen működnek (a programrészleteket Pascal-szerű pszeudo-kóddal illusztráljuk):

var közös_jelző:{foglalt,szabad}:=szabad *Valamennyi folyamat:* … belépés: **olvas** (közös_jelző) **if** foglalt **then goto** belépés **ír** (közös_jelző,foglalt) ***<kritikus szakasz>***

kilépés: **ír** (közös_jelző,szabad)

…

A megoldás problémája, hogy mivel a közös_jelző a közös memóriában helyezkedik el, kis valószínűséggel bár, de előfordulhat, hogy több folyamat olvassa egyidejűleg, és így többen is szabadnak találják. Ekkor egyidejűleg többen léphetnek be a kritikus szakaszba, ami nyilvánvalóan hibás.

Megjegyezzük, hogy az algoritmus nemcsak a folyamat általános modelljét (amikor minden folyamatnak külön processzora van) feltételezve hibás, hanem multiprogramozott megvalósításban is. Ekkor ugyan a párhuzamos olvasás kizárt (hiszen az egyetlen processzor egyidejűleg csak egyetlen utasítással foglalkozik), azonban az olvasás és visszaírás közé beékelődhet más folyamat végrehajtása, amelyik ugyancsak olvashatja, és szabadnak találhatja a jelzőt.

Néhány további zsákutcát átugorva, amelyek vagy ugyancsak nem biztosítják a kölcsönös kizárást, vagy csak felváltva engedik be a két folyamatot, vagy holtpontot okozhatnak, lássunk egy igazi megoldást!

A probléma egy helyes megoldása két folyamatra (Peterson, 1981): legyen a két folyamat *P*1 és *P*2. Legyen mindkét folyamatnak egy-egy jelzője a belépési szándékra, valamint egy változó, amelyik megmutatja, hogy egyidejű belépési szándék esetén melyik léphet be.

var jelző:**array**[1..2]**of**{foglal,szabad}:=szabad következő: {1,2} ***P1 folyamat:*** ... **ír** (jelző[1],foglal) **ír** (következő,2) belépés1: **olvas** (jelző[2]) **if** szabad **then goto** belép1 **olvas** (következő) **if** 2 **then goto** belépés1 belép1: ***<kritikus szakasz>*** kilép1: **ír** (jelző[1],szabad) … − − − − − − − − − − − *P2 folyamat:* … **ír** (jelző[2],foglal) **ír** (következő,1) belépés2: **olvas** (jelző[1]) **if** szabad **then goto** belép2 **olvas** (következő) **if** 1 **then goto** belépés2 belép2: ***<kritikus szakasz>*** kilép2: **ír** (jelző[2],szabad) … − − − − − − − − − − − −

Figyeljük meg, hogy a belépésre vonatkozó döntésben a következő változónak csak akkor van szerepe, ha a másik folyamat jelzője nem szabadot mutat, azaz egyidejű belépési szándék veszélye áll fenn. Ilyenkor legrosszabb esetben a következő változóra kiadott írás is egyidejű, de az írási verseny valahogy eldől. A következő változó csak egy értéket vehet fel, mégpedig a későbbi írás eredménye (a vesztes) jelöli ki a *másik* folyamatot belépőnek, ami akkor is helyes, ha a másik folyamat már korábban is a szakaszban tartózkodott.

A megoldás követése már két folyamatra sem valami egyszerű, több folyamatra pedig csak a gyakorlatban alig használható bonyolultságú általános megoldás adható. Elsőként Dijkstra publikált megoldást *n* folyamatra 1965-ben, majd több más javaslat is született. Talán a legáttekinthetőbb Lamport megoldása (1974) a pékség és egyéb sorállásra alkalmas üzletek és szolgáltatóhelyek működésének analógiájára *(bakery algorithm)*.

Bakery algoritmus adatszerkezetek közös tárban: **var** számot_kap: **array** [0..n-1] **of** *Boolean* := *false*; (jelzi, hogy egy folyamat éppen sorszámot kap) sorszám: **array** [0..n-1] **of** *integer* := 0; (tárolja a sorszámokat)

Pi folyamat: **belépési protokoll:** *Sorszámot kér:* számot_kap[i]:=*true*; sorszám[i]:=max(sorszám)+1; számot_kap[i]:=*false*; *Amíg van nála kisebb sorszámú, helybenjár:* **for** j:=0 **to** n-1 **do begin while** számot_kap[j] **do** üres_utasítás;

while sorszám[j]ą0 **and** (sorszám[j],j)< (sorszám[i],i) **do** üres_utasítás; **end; kilépési protokoll:** sorszám[i]:=0;

Az algoritmus szerint a belépni szándékozó folyamatok először sorszámot kérnek. Mivel a sorszámosztás a közös memóriában tárolt *sorszám* tömb maximális elemének kiválasztását igényli, nem atomi PRAM-művelet. Ezért folyamatonként egy jelző *(számot_kap)* véd attól, hogy eközben a folyamat sorszámát vizsgálhassa egy másik folyamat. Az ellen azonban nincs védelem az algoritmusban, hogy két folyamat azonos sorszámot kapjon.

Miután a folyamat megkapta a sorszámát, végignézi a többi folyamat sorszámait, és ha az övé a legkisebb, belép a kritikus szakaszba. A vizsgálat közben kivárja, míg egy éppen sorszámot kérő folyamat megkapja a számot, csak utána hasonlítja össze a sajátjával. Valahányszor talál olyan folyamatot, amelyik kisebb sorszámot kapott, kivárja, amíg az a folyamat elhagyja a szakaszt, csak azután lép a következő folyamat sorszámának vizsgálatára. Így a ciklus végére érve biztosan beléphet a szakaszba, mert kivárt minden korábban érkezőt, és ezek egy esetleges következő belépési szándék esetén már csak nagyobb sorszámokat kaphattak.

Mivel két folyamatnak lehet azonos sorszáma, az algoritmus a sorszámok összehasonlítása helyett a sorszámból és a folyamat azonosító számából álló rendezett számpárokat *(sorszám[i],i)* hasonlítja össze, így egyező sorszámok esetén is egyértelmű a döntés a kisebb azonosító számot viselő folyamat javára.

A szoftvermegoldások bonyolultsága más irányokba terelte a probléma megoldásán fáradozó kutatókat. Olyan eszközöket dolgoztak ki, amelyek – a PRAM-modellt is kiterjesztve – a processzor utasításkészletébe épülve támogatják a folyamatok szinkronizációjának megoldását (hardvertámogatás). A következőkben ezek közül ismertetünk néhányat.

2.2.5.2. A PRAM-modell kiterjesztése

A kritikus szakaszhoz rendelt foglaltságjelző bit logikailag egyszerű és jó ötlet. A problémát az okozza, hogy a jelző értékét többen is kiolvashatják, mielőtt az első olvasó *szabad*ról *foglalt*ra állította volna. Megoldódik a probléma, ha a PRAM-modellt úgy módosítjuk, hogy a jelzőkre vonatkozóan az *olvas* és *ír* utasításokon kívül bevezetjük az *OlvasÉsÍr (TestAndSet)* utasítást. Az utasítás kiolvassa és az *olvas* utasításhoz hasonlóan visszaadja egy jelző értékét, majd foglaltra állítja azt. A művelet ugyanúgy oszthatatlanul (sorosítva) hajtódik végre, mint az *olvas* és *ír* utasítások. Ez azt jelenti, hogy egy szabad jelzőre több *OlvasÉsÍr* utasítást egyidejűleg végrehajtva ezek közül csak egy ad vissza *szabad* értéket, a jelző végső értéke pedig természetesen *foglalt* lesz.

Az *OlvasÉsÍr* utasítás segítségével a kölcsönös kizárás megoldása:

var közös_jelző:{foglalt,szabad}:=szabad

Valamennyi folyamat:

… belépés: **OlvasÉsÍr** (közös_jelző) **if** foglalt **then goto** belépés **<*kritikus szakasz*>** kilépés: **ír** (közös_jelző,szabad) …

Az *OlvasÉsÍr* utasításhoz hasonlóan a *Csere (Swap)* utasítás bevezetésével is megoldható a probléma. A *Csere* utasítás a közös memória egy rekeszének és a folyamat saját memóriája egy rekeszének tartalmát cseréli meg ugyancsak oszthatatlanul. A kölcsönös kizárás megoldása a *Csere* utasítással:

var közös_jelző:{foglalt,szabad}:=szabad *Valamennyi folyamat:* **var** saját_jelző:{foglalt,szabad} … saját_jelző:=foglalt belépés: **Csere** (közös_jelző,saját_jelző) **olvas** (saját_jelző) **if** foglalt **then goto** belépés **<*kritikus szakasz*>** kilépés: **ír** (közös_jelző,szabad) …

Az *OlvasÉsÍr* vagy a *Csere* utasítások valamelyikét egy ideje már a fizikai processzorok utasításkészlete tartalmazza. Ezek megvalósítása olyan, hogy többprocesszoros rendszerben is garantálja az oszthatatlanságot például a memóriasín több műveletre kiterjedő lefoglalásával (lásd *LOCK* az Intel processzorcsalád, *read-modify-write* ciklus a Motorola processzorcsalád esetén). Multiprogramozott rendszerben (egy processzor) az oszthatatlanságot a megszakítások tiltásával is elérhetjük, így szimulálhatók az oszthatatlan *OlvasÉsÍr* vagy a *Csere* utasítások.

A fenti megoldások nem garantálják, hogy a kritikus szakaszba belépni szándékozó folyamatok véges időn belül bejutnak a szakaszba. Ez ugyanis azon múlik, hogy a versenyző *OlvasÉsÍr* vagy *Csere* utasítások milyen sorrendben kerülnek a PRAM sorosító csővezetékébe. A foglaltságjelző ciklikus tesztelésére alapozottan ez a probléma ismét csak igen bonyolultan oldható meg. Áttekinthető megoldáshoz akkor jutunk, ha a várakozó belépési szándékokat nyilvántartjuk, és a belépések ütemezését meghatározottá tesszük. Minden szempontból célszerűbb tehát, ha a folyamatok önszervező belépési protokolljai helyett a szinkronizáció problémájának korrekt, a hardver lehetőségeihez igazodó, tisztességes ütemezést biztosító megoldását logikailag a folyamatokat végrehajtó virtuális processzorra, realizációját tekintve az operációs rendszerre bízzuk.

2.2.5.3. Szinkronizációs eszközök az operációs rendszer szintjén

A kiterjesztett PRAM-modell felhasználásával az operációs rendszer szintjén összetettebb szinkronizációs eszközök hozhatók létre. Ezek közül a következőkben a szemafort, az erőforrást és az eseményt tárgyaljuk.

Szemafor

A szemafor, mint univerzális szinkronizációs eszköz használatára Dijkstra tett javaslatot 1965-ben. A szemafor egy speciális változó, amelyet csak a hozzá tartozó két, oszthatatlan művelettel lehet kezelni.

Az *általános szemafor* esetén a szemaforváltozó egész (integer) típusú. A két művelet elnevezése többféle lehet, például *Belép* és *Kilép,* vagy *Vár (Wait)* és *Jelez (Signal).* Leggyakrabban azonban az eredeti definícióban szereplő jelölést *(P* és *V)* használja a szakirodalom, így a továbbiakban mi is ezt követjük.

A *P(s)* művelet hatása egyenértékű azzal, mintha a folyamathoz tartozó logikai processzor a következő programrészletet hajtaná végre oszthatatlanul (definíciós program):

while s<1 **do** *üres_utasítás*;

s:=s-1;

(az *üres_utasítás* művelet nélküli továbblépést jelent, *s* pedig a közös memóriában lévő szemaforváltozó)

A *V(s)* művelet definíciós programja:

s:=s+1;

Hangsúlyozni kell, hogy mindkét művelet oszthatatlan, valamint azt is, hogy a szemaforváltozó más utasításokkal (írás, olvasás) nem érhető el. Azt is meg kell jegyeznünk, hogy a definíció a véletlenre bízza, hogy a várakozó folyamatok közül melyiknek sikerül a továbbhaladás, amikor egy másik folyamat *V* műveletet hajtott végre.

Vegyük észre, hogy az általános szemafort *k* kezdőértékre állítva a rendszer *k* darab *P* műveletet végrehajtó folyamatot enged tovább, a továbbiakat azonban várakoztatja. Ezután csak akkor enged tovább folyamatot a *P* rendszerhívásról, ha más folyamat *V* műveletet hajtott végre. Ezzel egy olyan *általánosított* kritikus szakaszt hoztunk létre, amelyiken belül egyidejűleg *k* darab folyamat tartózkodhat.

Precedencia és kölcsönös kizárás megvalósítására alkalmas a *bináris szemafor*, amelynek szemaforváltozója csak két értéket vehet fel (0 és 1, vagy *foglalt* és *szabad*). Kölcsönös kizárásra *1* kezdőértékű bináris szemafor használható, amelyre a kritikus szakaszba belépni kívánó folyamat *P* műveletet, a kritikus szakaszból kilépő pedig *V* műveletet hajt végre. Precedencia megvalósításához *0* kezdőértékű bináris szemafor használható. Az előbb végrehajtandó műveletet követően a folyamat *V* műveletet hajt végre a szemaforra, a másik folyamat pedig a később végrehajtandó művelet előtt *P* műveletet.

Erőforrás

Az erőforrás, mint szinkronizációs eszköz egy logikai objektum, amelyet egy folyamat lefoglalhat és felszabadíthat. A lefoglalás és a felszabadítás között a folyamat kizárólagosan használhatja az erőforrást, azaz erre a szakaszára és más folyamatok ugyanezen erőforrásra vonatkozó foglalási szakaszaira kölcsönös kizárás valósul meg. Az erőforrásokat egy név vagy egy sorszám azonosítja. A programozók megegyezésén, illetve rendszertervezői döntésen múlik, hogy egy-egy erőforrást milyen tartalommal használnak. A *Lefoglal(<erőforrás>)* és *Felszabadít(<erőforrás>)* műveletek egyenértékűek egy *s* bináris szemaforra kiadott *P(s)* és *V(s)* műveletekkel. Mint a szemafornál sem, az erőforrásra várakozó folyamatok esetén sem meghatározott, hogy melyikük kapja meg a felszabaduló erőforrást, és haladhat tovább. Csak az biztos, hogy egyetlen ilyen folyamat lesz.

Esemény

Az esemény egy pillanatszerű történés a rendszerben, amelyre folyamatok várakozhatnak. Az esemény bekövetkezése valamennyi rá várakozó folyamatot továbbindítja. Az eseménnyel így egy összetett precedencia valósítható meg, ahol több, különböző folyamatokban elhelyezett műveletre ugyanazt az előzményt írjuk elő.

Két folyamat esetén egy esemény jelzése és az arra való várakozás egyenértékű egy szemaforra kiadott *V*, illetve *P* műveletekkel. Több folyamat esetén azonban lényeges különbség, hogy a szemaforra kiadott *V* művelet hatására csak egyetlen várakozó folyamat indulhat tovább, míg az esemény bekövetkezése valamennyi várakozó folyamatot továbbindítja. Míg a szemafor és az erőforrás felszabadítása az objektum állapotában megőrződik akkor is, ha nem várnak rá, és egy későbbi foglalás várakozás nélkül sikeres lesz, az esemény jelzése általában hatástalan, ha nincs rá várakozó, csak a várakozás megkezdése után bekövetkező esemény indít tovább folyamatot.

A bemutatott szinkronizációs eszközök közül a szemafor és az erőforrás több várakozó folyamat közül egy továbblépését teszi lehetővé, azonban ennek kiválasztását a véletlenre bízza. Ez a megoldás sok esetben nem kielégítő. Az egyik fő érv a megoldás ellen, hogy nincs garancia arra, hogy minden várakozó véges időn belül tovább tud lépni. Egy másik – az előzőnek egyébként ellentmondó – érv a véletlenre hagyatkozás ellen, hogy amennyiben fontossági sorrendet (prioritást) állítunk fel a folyamatok között, a véletlenszerű továbbindítás ezt nem érvényesíti. Felmerül tehát annak az igénye, hogy a választás meghatározott algoritmus szerint történjen. Ennek megoldása a várakozó folyamatok sorbaállítása, és ütemezett továbbengedése. A szemaforokhoz és erőforrásokhoz így *várakozási (ütemezési) sorokat* rendelhetünk, amelyeket a rendszer meghatározott (ütemezési) algoritmus szerint kezel. A leggyakoribb az érkezési sorrend szerinti (FIFO), illetve a prioritásos ütemezés.

2.2.6. Folyamatok kommunikációja

Mint láttuk, a folyamatok együttműködésének másik alapmodellje a közös memóriás együttműködés mellett az üzenetváltásos együttműködés, azaz a folyamatok közötti kommunikáció. Az üzenetváltásos együttműködés akkor került előtérbe, amikor realitássá vált több, adatátviteli vonalon, vagy hálózaton keresztül összekapcsolt számítógép együttműködése.

A kommunikáció alapsémáját a 2.3. ábra mutatja be. Azonkívül, hogy a logikai processzor utasításkészletében szerepel *Küld* és *Fogad* művelet, a kommunikáció működésével, tulajdonságaival kapcsolatban számos nyitott kérdés maradt, és megállapítottuk, hogy nincs egyetlen tiszta modell, amelyet a megoldások követnek.

Néhány a legkézenfekvőbb kérdések közül:

- Hogyan nevezhetjük meg a partnert? A partnert látjuk, vagy egy közbülső objektumot (csatornát, postaládát)? Egyetlen partner veheti egyszerre az üzenetünket, vagy több is? Csak egyetlen partnertől várhatunk üzenetet egy adott pillanatban, vagy többektől is?
- Amikor a *Küld* művelet befejeződött, meddig jutott az üzenet? Már ott van a partnernél, vagy csak útjára bocsátottuk (lásd levél bedobása a postaládába)? Honnan fogjuk megtudni, hogy rendben megérkezett-e?
- Hogyan működik a *Fogad* művelet? Mi történik, ha hamarabb akarunk fogadni egy üzenetet, mint azt elküldték? Kell-e visszaigazolást küldenünk, vagy ezt a rendszer automatikusan elintézi?

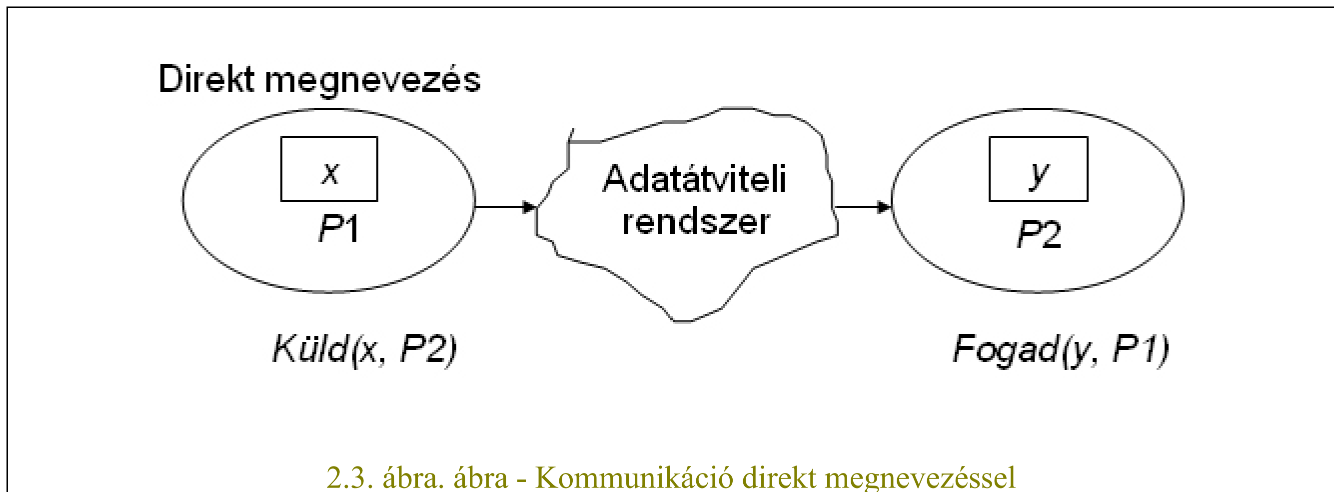
A következőkben a felmerülő kérdések közül hárommal foglalkozunk részletesebben:

- Milyen megnevezést használhatnak a kommunikáló partnerek, hogy egymásra találjanak?
- Milyen szemantikai konzisztenciának kell fennállni a küldés és a fogadás között?
- Milyen járulékos (implicit) szinkronizációs hatása van a kommuniká-ciónak?

2.2.6.1. A partner megnevezése

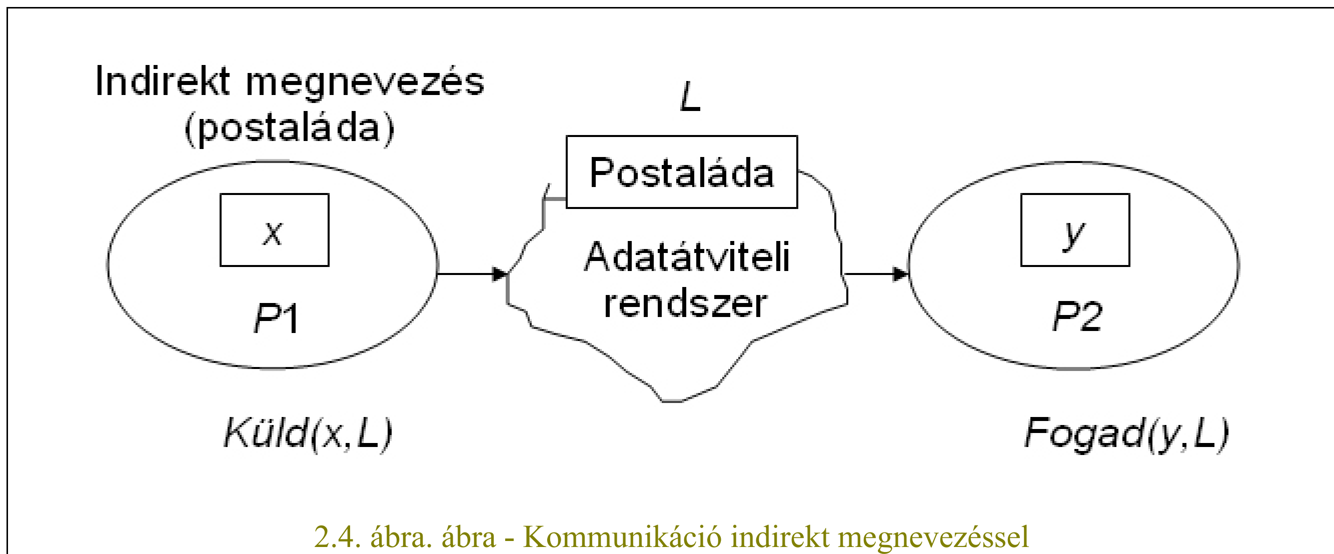
Megnevezés tekintetében beszélhetünk közvetlen (direkt), közvetett (indirekt), valamint aszimmetrikus kommunikációról, illetve megnevezésről, továbbá csoportkijelölésről és üzenetszórásról.

A *közvetlen (direkt)* kommunikáció két folyamat között zajlik, mind a *Küld*, mind a *Fogad* művelet megnevezi a partner folyamatot (2.3. ábra). P1 elküldi a saját címtartományában tárolt *x* változó értékét P2-nek, aki azt saját *y* változójába teszi el. (Ha a változók közös címtartományban lennének, egyszerűen *y:=x* értékadást használhatnánk, így azonban kommunikációs műveletek szükségesek.)



2.3. ábra. ábra - Kommunikáció direkt megnevezéssel

Közvetett (indirekt) kommunikáció esetén a partnerek nem egymást nevezik meg, hanem egy közvetítő objektumot, (például *postaládát,* vagy *csatornát*). A postaládán keresztül bonyolódó kommunikációt a 2.4. ábra szemlélteti.



2.4. ábra. ábra - Kommunikáció indirekt megnevezéssel

A *postaláda* egy általában véges, de elméletileg esetleg korlátlan befogadóképességű, az üzenetek sorrendjét megtartó (FIFO) tároló, amely a *Küld-Fogad,* (betesz-kivesz) műveletpárral kezelhető. A *Küld(<cím>,<postaláda>)* művelet a saját címtartományban elhelyezkedő üzenetet a postaláda következő szabad tárolóhelyére másolja. Ha a postaláda tele van, ürülésig várakozik. A *Fogad(<cím>,<postaláda>)* művelet a postaládában legrégebben várakozó üzenetet kimásolja a megadott, saját címtartománybeli címre, helyét pedig felszabadítja.

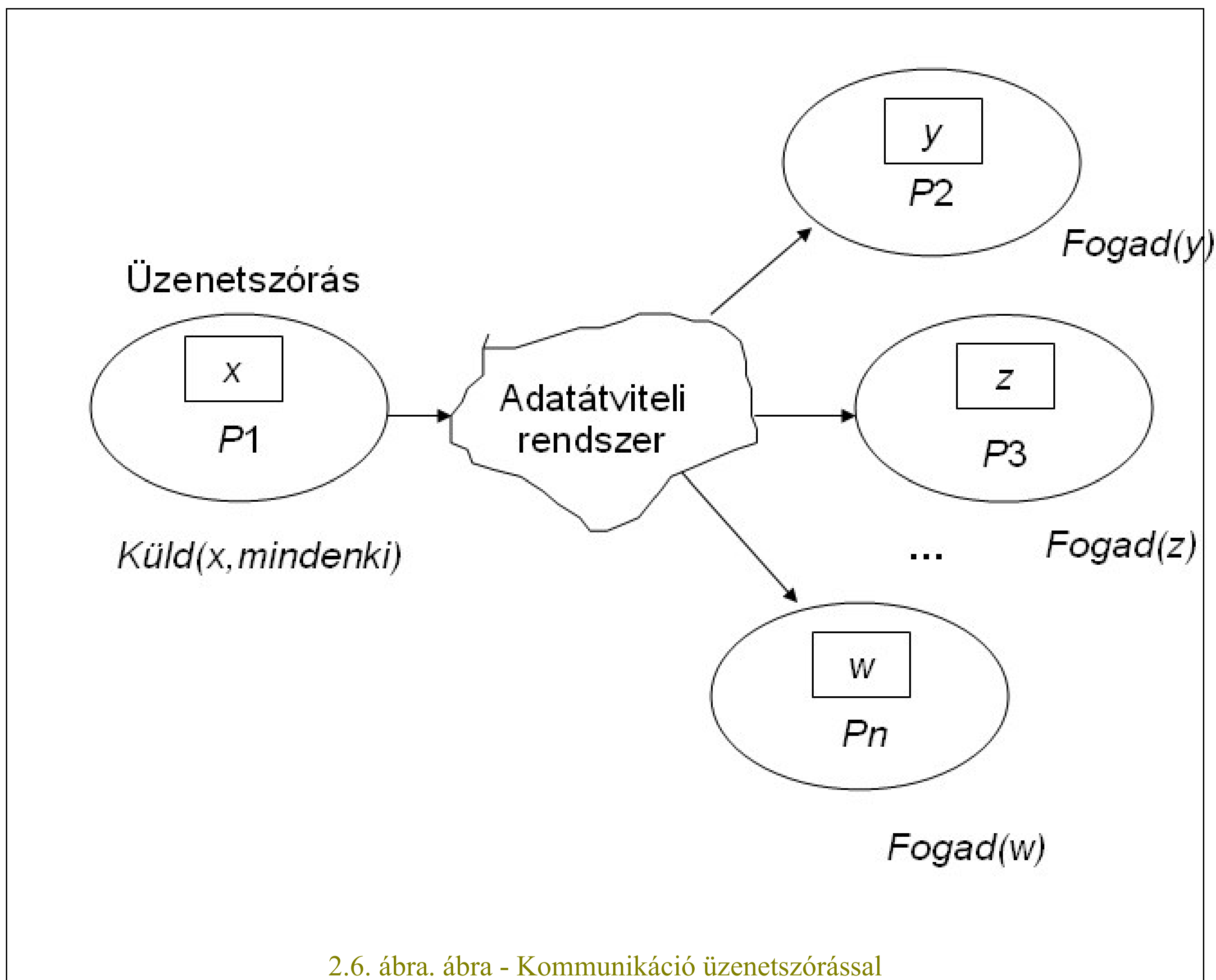
A *csatorna* olyan kommunikációs objektum, amelyik két folyamatot kapcsol össze. A csatorna lehet egyirányú (szimplex), osztottan kétirányú, azaz egyidejűleg egyirányú, de az irány változtatható (félduplex), vagy kétirányú (duplex). Tartalmazhat 0, véges vagy végtelen kapacitású átmeneti tárolót. Egy véges tárolókapacitású duplex csatorna egyenértékű két véges befogadóképességű postaládával, amelyeket csak két folyamat használ, egyiket egyik irányú, másikat az ellenkező irányú adatcserére.

A postaláda és a csatorna lazítja a kommunikáló folyamatok közötti csatolást, lehetővé teszi, hogy olyan folyamatok is információt cseréljenek, amelyek nem ismerik egymást.

2.5. ábra. ábra - Kommunikáció asszimetrikus megnevezéssel

Aszimmetrikus megnevezés esetén az egyik folyamat, az adó vagy vevő, megnevezi, hogy melyik folyamattal akar kommunikálni, a másik partner viszont egy saját *be-/kimeneti kaput* (*port*) használ, amelyiket – ha csak egy van – nem is kell megneveznie. Ha az üzenet vevője használ bemeneti kaput, akkor a műveletek alakja *Küld(<cím>,<folyamat>), Fogad(<cím>)*. Ez a megoldás azokban az esetekben hasznos, amikor a fogadó folyamat nem ismeri a küldőt, de a küldő ismeri a fogadót, például egy *ügyfél* folyamat szolgáltatási kérelmet küld egy *szolgáltató* folyamatnak *(kliens–szerver modell)*. A fordított irányú aszimmetria alkalmazására pedig egy feladat lebontását és szétosztását végző menedzser folyamat és több, munkáért versengő végrehajtó folyamat kapcsolata lehet példa. A küldő a kimeneti portjára küldi az elvégzendő feladatot tartalmazó üzenetet, a fogadó pedig az első ráérő feldolgozó lesz *(farmer–worker modell)*.

Csoportkommunikáció esetén az üzenet küldője folyamatok (esetleg kommunikációs objektumok) egy csoportját nevezheti meg vevőként. Ilyenkor egyetlen üzenetküldő művelet végrehajtása azt eredményezi, hogy az üzenetet a csoport valamennyi tagja megkapja. Az *üzenetszórás (broadcasting)* logikailag a csoportkommunikáció azon esete, amikor az egyetlen művelettel elküldött üzenet a rendszer valamennyi folyamatához eljut (2.6. ábra).



2.6. ábra. ábra - Kommunikáció üzenetszórással

A csoportkommunikáció lehetősége lényegesen egyszerűsíti a küldő műveleteit, ha több folyamattal akarja ugyanazt az információt közölni. Bizonyos típusú fizikai átviteli közegeken a csoportkommunikáció igen hatékonyan valósítható meg (például sín-topológiájú összeköttetések, rádiókommunikáció).

2.2.6.2. Szemantikai konzisztencia

Most azt szeretnénk tisztázni, hogy mi történik a kommunikációs műveletek végrehajtásakor, milyen hatása van a műveleteknek az egyes folyamatok állapotterére (változókra, kommunikációs objektumokra), milyen minimális konzisztencia-feltételt kell betartani a műveletek megvalósításakor.

Az üzenetváltásos modell akkor került előtérbe, amikor több számítógépet kapcsoltak össze egy rendszerré adatátviteli vonalon, vagy hálózaton keresztül. Ilyenkor fel kell készülni olyan rendkívüli esetekre is, hogy a partner folyamatot futtató számítógépet esetleg kikapcsolták, vagy a nagyobb távolságot áthidaló átvitel közben az adatok sérülnek, hiszen a hibalehetőség lényegesen nagyobb, mint az egyetlen chipben, dobozban, esetleg egyetlen kártyán elhelyezkedő egységek közötti átvitel esetén. Az üzenettovábbítás műveleteivel szemben ezért

elvárás, hogy a műveletet végrehajtó folyamat ne fagyjon be sem *átviteli hibák*, sem a *partner folyamatok kiesése* esetén, és a kommunikációs műveletek helyes vagy hibás végrehajtásáról szerezzen tudomást. A műveletek *helyességének visszajelzésére* az egyik szokásos megoldás egy *állapotkód* visszaadása, amelynek egyik értéke a helyes végrehajtás, további értékei pedig az előforduló hibakódok lehetnek. Egy másik megoldás lehet a logikai processzor szintjén megvalósított *hiba-megszakítás*, ami a folyamat végrehajtását a hiba felderítéséhez és a folyatáshoz szükséges információk tárolását követően egy hibakezelési pontra (exception handler) irányítja. A *partner folyamat kiesését* szokásosan a műveletekre megszabott *időkorlát (time-out)* figyelésével észlelik. A kommunikációs művelet az időkorlát elérésekor hibajelzéssel akkor is befejeződik, ha a partner válaszának hiánya, vagy a kommunikációs közeg hibája miatt az adatcsere még nem történt meg. A folyamatok programozásához megfelelő rugalmasságot nyújt, ha lehetőség van a kommunikációs műveletekre vonatkozó időkorlát paraméterként történő megadására.

A hibajelzéssel és időkorláttal kiegészítve közvetlen kommunikáció esetén a műveletek a következő alakúak: *Küld(<cím>,<folyamat>,<időkorlát>, <hibakód>),* illetve
Fogad(<cím>,<folyamat>,<időkorlát>,<hibakód>).

Az időkorlát lehetséges értékei között célszerű a *0* és a <*végtelen*> értéket is megengedni. A *0* várakozás például akkor hasznos, ha egy fogadó folyamat működésének egy pontján csak akkor akar üzenetet fogadni, ha azt már elküldték, egyébként van más teendője. A <*végtelen*> várakozás pedig az olyan folyamatok esetén szükséges, amelyiknek nincs teendője, amíg valaki egy üzenettel el nem indítja.

A *Küld* művelet szemantikáját tekintve elsősorban az a kérdés merül fel, hogy okozhat-e várakozást a művelet, és mi az, ami az adatcseréből a művelet befejeződésekor, azaz a folyamat továbblépésekor már megtörtént.

A lehetséges megoldásváltozatok közül az egyik véglet: a *Küld* művelet akkor fejeződik be, amikor az adatcsere teljes egészében befejeződött, a küldött üzenet rendben megérkezett és a helyére került. Ez a megoldás általában a *Küld* művelet várakozását is okozhatja, a végrehajtás során ellenőrzések, esetleg hibajavítások is történhetnek. A megoldás ahhoz hasonló, mint amikor tértivevényes levelet küldünk valakinek, és addig nem lépünk tovább, amíg a tértivevény aláírva vissza nem érkezett.

A másik véglet az lehet, hogy a *Küld* befejeződésekor az üzenet csupán bekerült a kommunikációs rendszerbe, de további sorsáról semmit sem tudunk. Ilyenkor a *Küld* művelet általában sohasem várakozik. Ez a megoldás ahhoz hasonló, mint amikor valakinek úgy küldünk levelet, hogy egyszerűen bedobjuk a postaládába, és továbbmegyünk.

A két véglet között számos köztes megoldás létezhet (például az üzenet elhagyta azt a processzort, amelyiken a küldő folyamat fut stb.).

Valamennyi megoldás esetén be kell tartani azt a minimális konzisztencia-feltételt, hogy amennyiben nincs hibajelzés, az elküldött üzenetet tartalmazó terület a küldő folyamat saját memóriájában (a <*cím*> tartalma) a

Küld befejeződése után – akár a következő utasítással – felülírható legyen. Ennek már nem szabad hatással lennie az elküldött üzenetre.

A *Fogad* művelet megvalósításai általában várakozást okozhatnak abban az esetben, ha még nem érkezett üzenet. A művelet egyszeri végrehajtása pontosan egy üzenetet helyez el a fogadó folyamat saját memóriájába akkor is, ha esetleg több várakozó üzenet van a folyamat számára a kommunikációs rendszerben. Az üzenetek érkezési sorrendben fogadhatók, minden üzenet csak egyszer, azaz a fogadás törli az üzenetet a kommunikációs rendszerből. Postaláda, illetve bemeneti kapu használata esetén kérdés, hogy a fogadó tudomást szerez-e arról, hogy ki az üzenet küldője. Erre általában szükség van, hiszen az üzenetre gyakran választ kell küldeni. Ha a kommunikációs rendszer ezt az információt nem közli automatikusan, a folyamatoknak kell gondoskodniuk arról, hogy a küldő azonosítható legyen az üzenet tartalma alapján.

2.2.6.3. Járulékos (implicit) szinkronizáció

A kommunikációs műveletek általában a kommunikációban résztvevő folyamatok szabadonfutásának korlátozását okozzák. Az egyes műveletekkel járó szinkronizációs mellékhatás elsősorban a kommunikációs rendszer átmeneti tárolójának (puffer) kapacitásától függ.

Tárolás nélküli átvitel esetén a kommunikációs rendszer csak közvetít, de a *Küld* és *Fogad* műveleteknek be kell várnia egymást ahhoz, hogy az információcsere megtörténhessen. A két folyamat ezen utasításaira *egyidejűség* érvényes (a Küld és a Fogad *randevúja*).

Véges kapacitású tároló alkalmazása bizonyos keretek között kiegyenlíti a küldő és fogadó folyamatok sebességingadozásait. A fogadó folyamat várakozik, ha üres az átmeneti tároló, a küldő folyamat pedig akkor, ha nincs szabad hely a tárolóban. Ugyanazon üzenet elküldésének meg kell előznie az üzenet fogadását, tehát ezen műveletekre *sorrendiség (precedencia)* érvényesül. Emellett egy *rövid távú* szinkronizációs hatás is érvényesül, magának a tárolónak a kezelése általában *kölcsönös kizárással* valósul meg, azaz két folyamat ugyanazon kommunikációs objektumra hivatkozó kommunikációs műveleteinek megvalósítási részleteit szemlélve egyes szakaszokra kölcsönös kizárás áll fenn.

Végtelen kapacitású tároló (csak modellben létezik) esetén a küldő folyamatnak sohasem kell várakoznia, egyébként a véges kapacitású tárolóra elmondottak érvényesek.

2.2.7. Holtpont

Amikor az első multiprogramozott rendszereket üzembe állították, időnként nehezen reprodukálható, ezért nehezen megkereshető és megmagyarázható hibákat tapasztaltak. Ezek egyik típusa a lyukas kölcsönös kizárásokból (foglaltságjelző bit) származott. A másik jelenségtípus a rendszerek időnkénti „lefagyása”. Ahogy a kölcsönös kizárás korrekt megoldása is alapos analízist, majd elméletileg megalapozott megoldást igényelt, a lefagyási jelenségekkel kapcsolatban is hasonló volt a helyzet. Az elemzések kiderítették, hogy folyamatok bizonyos esetekben úgy akadhatnak össze, hogy csak egymást tudnák továbbindítani, de mivel mindegyik a másikra vár, senki nem tud továbblépni. Az ilyen helyzeteket holtpontnak (deadlock) nevezzük. Holtpont

bekövetkezésének gyakran igen kicsi a valószínűsége. A hiba éppen ezért alattomos, és nehezen reprodukálható. Vegyük hát alaposabb vizsgálat alá a jelenséget!

2.2.7.1. Mi a holtpont?

Ha az együttműködő folyamatok szinkronizációs és kommunikációs műveleteit kellő körültekintés nélkül alkalmazzuk, könnyen előidézhetjük a rendszer folyamatainak teljes vagy részleges befagyását.

Tegyük fel, hogy egy rendszerben két erőforrást (például az *NY* nyomtatót és az *M* mágnesszalag egységet) használ két folyamat a következő módon:

P1 folyamat: ... ***Lefoglal(M)*** <mágnesszalag használata>... ***Lefoglal(NY)*** ... <nyomtató és mágnesszalag együttes használata> ... ***Felszabadít(M)*** <nyomtató használata>… ***Felszabadít(NY)*** ... – – – – – – – – – – – – – –

P2 folyamat: ... ***Lefoglal(NY)*** <nyomtató használata>... ***Lefoglal(M)*** ... <nyomtató és mágnesszalag együttes használata> ... ***Felszabadít(NY)*** <nyomtató használata>… ***Felszabadít(M)*** ... – – – – – – – – – – – – – –

Tegyük fel, hogy a két folyamat együttfutásakor *P1* már lefoglalta *M*-et, de még nem foglalta le *NY*-et, *P2* pedig lefoglalta *NY*-et de még nem foglalta le *M*-et. (Ilyen helyzet kialakulhat, hiszen a folyamatok futására nincs korlátozás, de ugyanezért nem biztos, hogy minden együttfutáskor kialakul.) A továbbiakban *P1* előbb-utóbb le akarja foglalni *NY*-et is, de nem kaphatja meg, mert az már *P2*-é, *P2* pedig le akarja foglalni *M*-et is, de az már *P1*-é. Így mindkét folyamat arra fog várni, hogy a másik elengedje az erőforrását, de addig egyik sem tud eljutni, mert előbb erőforráshoz kellene jutnia. A két folyamat között keresztreteszelés alakul ki, ebből a helyzetből egyik sem tud továbblépni.

Vegyük észre, hogy ha a folyamatok úgy futnak le, hogy valamelyiknek sikerül mindkét erőforrást megszerezni, akkor már nem alakulhat ki holtpont, hiszen akkor ez a folyamat előbb-utóbb fel is szabadítja azokat és ekkor – ha már közben igényelte, várakozás után – a másik is hozzájuk juthat. A holtpont kialakulásának valószínűsége annál nagyobb, minél hosszabb a folyamatok azon szakasza, amikor még csak egyik erőforrást birtokolják. Nem alakulhatna ki holtpont, ha a folyamatok egyetlen művelettel kérnék el mindkét erőforrást.

Vegyük észre azt is, hogy akkor sem alakulhatna ki holtpont, ha mindkét folyamat azonos sorrendben kérné el a két erőforrást.

A fenti, legegyszerűbb esetnél lényegesen bonyolultabban és áttételesebben is összegubancolódhatnak folyamatok amiatt, hogy közös erőforrásaik vannak, illetve üzeneteket várnak egymástól.

Általánosságban azt mondjuk, hogy egy rendszer folyamatainak egy H részhalmaza **holtponton** van, ha a H halmazba tartozó valamennyi folyamat olyan eseményre vár, amelyet csak egy másik, H halmazbeli folyamat tudna előidézni.

Megjegyzések:

- A definíció általános, az esemény nemcsak erőforrás felszabadulása lehet, hanem tetszőleges más valami is, amire egy folyamat várakozni tud.
- A rendszerben lehetnek futó, élő folyamatok a holtponton lévők mellett, tehát nem biztos, hogy a befagyás teljes.
- Nem biztos, hogy a holtpont a folyamatok minden együttfutásakor kialakul, sőt az esetek jelentős részében igen kis valószínűséggel alakul ki. Ezért a jelenség nehezen reprodukálható, alattomos hibaforrás.
- A holtpont egyrészt azon funkciók kiesését okozza, amelyeket a befagyott folyamatok látnak el, másrészt csökkenti a rendszer teljesítőképességét, hiszen a befagyott folyamatok által lekötött erőforrásokhoz a többi folyamat sem tud hozzájutni.

A továbbiakban olyan rendszerekre szorítkozunk, amelyek folyamatai között csak az erőforrásokért folytatott verseny miatt van kapcsolat (ide tartozhat tetszőleges, kölcsönös kizárás jellegű szinkronizáció is).

2.2.7.2. Holtpont erőforrásokért versengő rendszerekben

Az erőforrásokért versengő rendszerekre a következő rendszermodellt állíthatjuk fel:

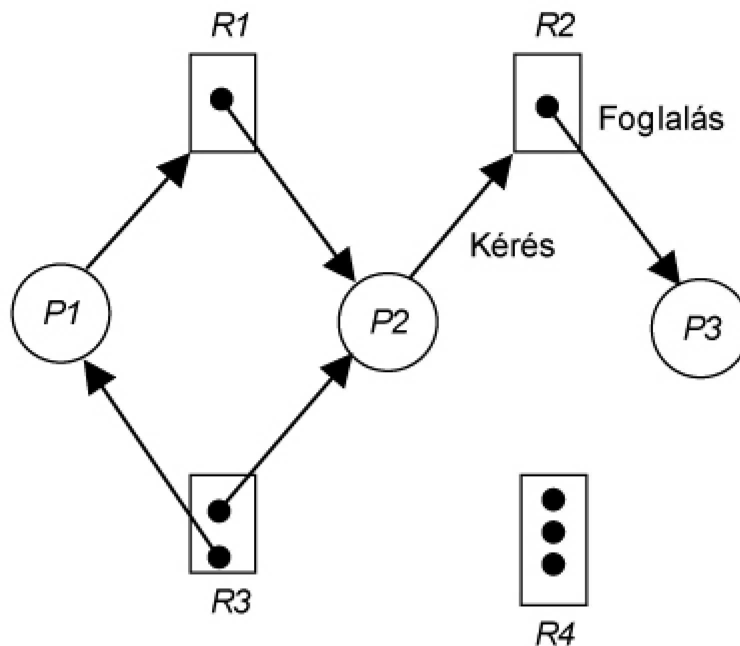
- A rendszerben véges számú és típusú erőforrást kell felosztani az értük versengő folyamatok között.
- Az erőforrások osztályokba (típusokba) sorolhatók. Egyes típusokhoz egyetlen erőforráspéldány tartozik *(egypéldányos erőforrások)*, más típusú erőforrásokból több példány áll rendelkezésre *(többpéldányos erőforrások)*, amelyek használati értékükben azonosak. Egy folyamat számára közömbös, hogy azonos típuson belül melyik erőforráspéldányokat használja. Jellegzetes egypéldányos erőforrások például a speciális perifériák (rajzgép, kisebb rendszerek egyetlen nyomtatója), rendszertáblák, multiprogramozott rendszerek processzora stb. Jellegzetes többpéldányos erőforrások: memória, a rendszer egyenértékű nyomtatói, munkaterület a mágneslemezen.
- Az erőforrásokat használati módjuk szerint csoportosíthatjuk megosztottan használható vagy csak kizárólagosan használható erőforrásokra. A *megosztottan használható erőforrások állapota menthető és visszaállítható (preemptable)*, így esetleg elvehetők egy folyamattól, és később visszaadhatók úgy, hogy a folyamat zökkenőmentesen folytatódhasson (lásd CPU, memória). Ilyen erőforrásokra megfelelő időbeli ütemezéssel (a használat időbeli megosztásával) látszólag párhuzamos használatot lehet szimulálni. A *csak kizárólagosan használható* erőforrások *állapota nem menthető (non-preemptable)*, így nem vehetők el a folyamattól anélkül, hogy a folyamatot visszaküldenénk egy korábbi állapotába (lásd nyomtató).
- A folyamatok az erőforrások használata során a következő lépéseket hajtják végre:

1. *Igénylés.* Az igénylés rendszerhívással történik. Legyen az erőforráskérés művelete a *Kér.* Egy kéréssel általános esetben (többpéldányos erőforrások) akár valamennyi erőforrásfajtából is lehet egyidejűleg több példányt igényelni. Ha a rendszerben *n* fajta erőforrás van, a művelet paramétere egy *n* elemű vektor, amelynek minden eleme az adott erőforrásfajtából igényelt mennyiséget jelöli *(Kér (n1,n2, ...,nn))*. Az igény nem teljesíthető, ha bármelyik erőforrástípusból nem adható ki a folyamatnak az igényelt mennyiség. Ilyenkor a folyamat várakozni kényszerül. Tekintve, hogy az erőforráspéldányok egyenértékűek, a folyamat a szabad példányok bármelyikét megkaphatja. A folyamat azonban a megkapott konkrét példányokat használja, amelyeket tehát most már azonosítani kell. A *Kér* művelet ezért általános esetben erőforrás-típusonként egy-egy tömböt is visszaad, amelyik az odaadott erőforráspéldányok azonosítóit tartalmazza. A felszabadítás – amennyiben nem az adott típus valamennyi birtokolt példányának felszabadítása a folyamat szándéka – a konkrét példányok felszabadítását kell hogy jelentse.
2. Használat.
3. *Felszabadítás.* Általános esetben a *Felszabadít* művelet két formáját célszerű bevezetni. A *Felszabadít (E1,E2,...,Em)* paraméterei erőforrástípusok. A művelet a felsorolt erőforrástípusokból a folyamat által birtokolt valamennyi példányt felszabadítja. A *Felszabadít (V1,V2, ...,Vn)* alakban a paraméterek *Vi* vektorok, amelyek az egyes erőforrástípusokból felszabadítandó példányok azonosítóit tartalmazzák. Amennyiben az erőforrásokra várakoztak más folyamatok, ezek közül valamelyik megkaphatja a felszabaduló erőforrásokat és továbbléphet.

A holtpont kialakulásának szükséges feltételei:

1. *Kölcsönös kizárás:* legyenek olyan erőforrások a rendszerben, amelyeket a folyamatok csak kizárólagosan használhatnak.
2. *Foglalva várakozás:* legyen olyan folyamat, amelyik lefoglalva tart erőforrásokat, miközben más erőforrásokra várakozik.
3. *Nincs erőszakos erőforrás-elvétel* a rendszerben, azaz minden folyamat addig birtokolja a megkapott erőforrásokat, amíg saját jószántából fel nem szabadítja azokat.
4. *Körkörös várakozás:* a rendszerben lévő folyamatok közül létezik egy olyan {*P0*, *P1*, ... *Pn*} sorozat, amelyben *P0* egy *P1* által lefoglalva tartott erőforrásra vár, *Pi* egy *Pi+1*-re, végül *Pn* pedig *P0*-ra vár.

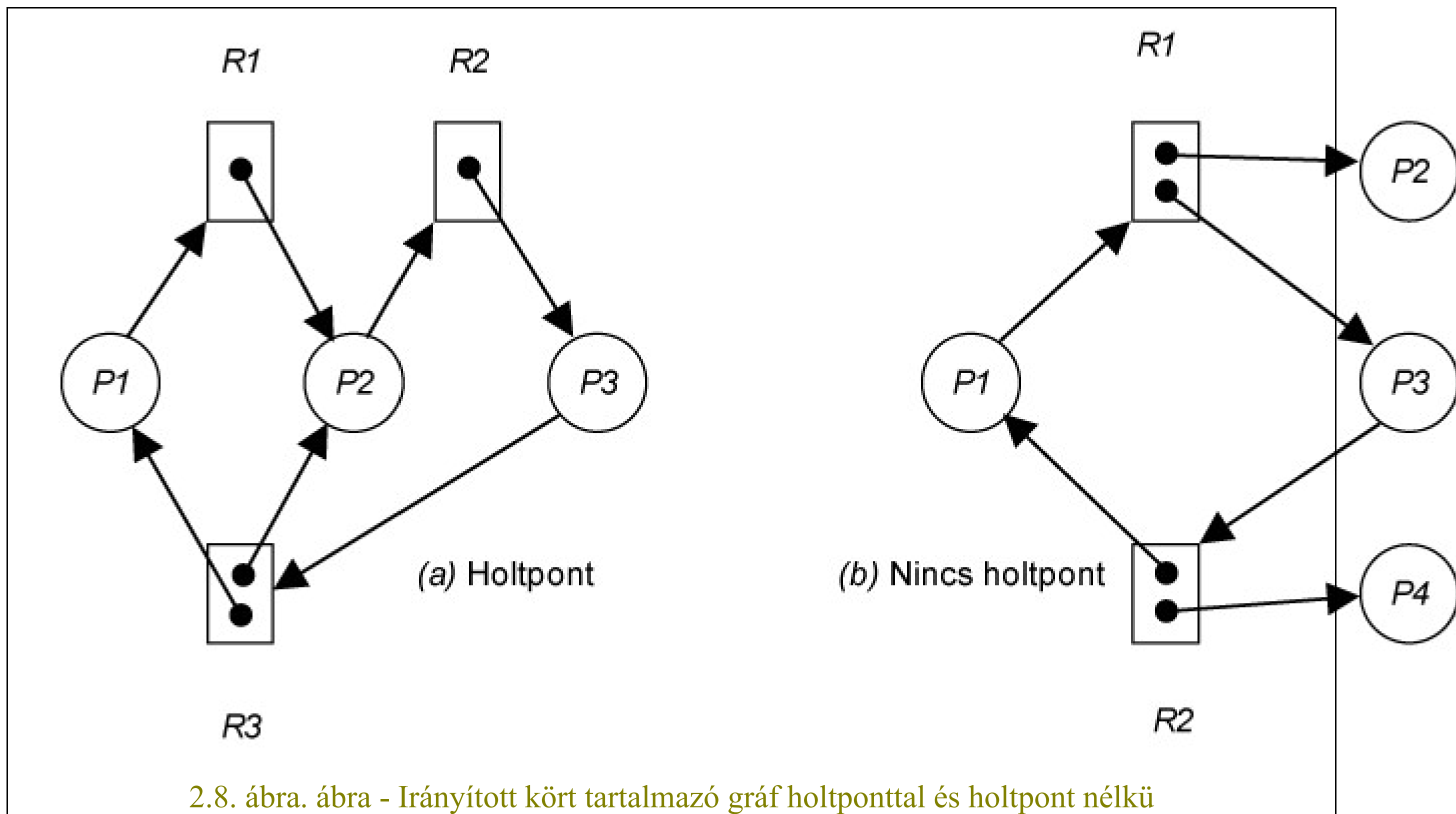
A rendszer pillanatnyi állapotát **erőforrásfoglalási gráffal (resource allocation graph)** modellezhetjük (2.7. ábra).



2.7. ábra. ábra - Erőforrásfoglalási gráf

A gráfnak kétféle csomópontja van: erőforrás (*Ri*, téglalap) és folyamat (*Pi*, kör). Ugyancsak kétféle él található a gráfon. Irányított él vezet *Pi* folyamattól *Rj* erőforráshoz (kérés él), ha *Pi* már kérte *Rj*-t, de még nem kapta meg. Irányított él vezet *Ri*-től *Pj*-hez (foglalás él), ha *Pj* birtokolja (megkapta és még nem szabadította fel) *Ri*-t. Többpéldányos erőforrások esetén a példányokat megfelelő számú ponttal jelezzük az erőforrástípust jelképező téglalapon belül. Annak érzékeltetésére, hogy a folyamat konkrét példányt birtokol, de csak típust kér, a foglalás éleket a konkrét példányt jelképező pontból indítjuk, a kérés éleket pedig csak a téglalap határvonaláig vezetjük.

Az erőforrásfoglalási gráf a rendszer működése során folyamatosan változik, ahogyan új kérések és foglalások történnek. A gráf elemzése alapján következtethetünk holtponthelyzetek fennállására. Körkörös várakozás esetén (holtpont 4. szükséges feltétele) a gráfon is irányított kör van. Abban az esetben, ha minden erőforrás egypéldányos, a gráfon kimutatható kör egyben elégséges feltétel is a holtpont fennállására. Többpéldányos esetben azonban kör jelenléte nem jelenti feltétlen azt, hogy a rendszerben holtpont van. A 2.8. ábra *(b)* oldalán a kör ellenére látjuk, hogy mind *R1*, mind *R2* egy-egy példányát olyan folyamat birtokolja, amelyik nem várakozik (*P2* és *P4*), így van esély rá, hogy felszabadítja az erőforrást. A körben szereplő *P1* és *P3* tehát nem feltétlen egymásra vár, hanem *P2*, illetve *P4* által okozott esemény hatására is továbbléphetnek.



2.8. ábra. ábra - Irányított kört tartalmazó gráf holtponttal és holtpont nélkü

Az ábra *(a)* oldalán ezzel szemben olyan helyzetet látunk, amelyben mindhárom folyamat várakozik, és bármelyiküket csak a három várakozó valamelyike tudná egy erőforrás-felszabadítással továbbengedni.

Miután megalkottuk a holtpontprobléma vizsgálatára alkalmas rendszermodellt az erőforrásokért versengő folyamatok esetére, foglalkozzunk azzal a kérdéssel, hogy mit kezdhetünk a problémával. Háromféle alapállásunk lehet:

1. Nem veszünk tudomást a problémáról és nem teszünk semmit *(strucc algoritmus)*.
2. Észrevesszük, ha holtpont alakult ki, és megpróbáljuk feloldani *(detektálás és feloldás)*.
3. Védekezünk a holtpont kialakulása ellen. Ezen belül is két lehetőségünk van:

   – *megelőzés,* amikor is strukturálisan holtpontmentes rendszert tervezünk, amelyben a szükséges feltételek valamelyikének kizárása miatt eleve nem alakulhat ki holtpont,

   – *elkerülés,* amikor is a rendszer futás közben csak olyan erőforráskéréseket elégít ki, amelyek nem vezetnek holtpontveszélyhez.

2.2.7.3. A strucc algoritmus

Az „algoritmus” elnevezését Tanenbaum és Woodhull könyvéből kölcsönöztük. Első reakciónk valószínűleg az, hogy ez az alapállás meglehetősen hanyag magatartásforma, hiszen ismert hibalehetőséget hagyunk tudatosan a rendszerben. Bizonyos típusú rendszerek esetén – ahol élet- és vagyonbiztonság a tét – nem is engedhető meg az

ilyen tervezői megközelítés. Más, kisebb kockázatú esetekben – ahol megengedhető a „kiszállunk, beszállunk, és akkor biztos jó lesz ...” szemlélet, azaz különösebb veszteség nélkül újraindítható a rendszer – azonban mégis reális lehet a probléma figyelmen kívül hagyása.

Mint a mérnöki praxisban annyiszor, azt kell mérlegelnünk, mekkora a probléma és milyen áron tudjuk megoldani. Az esetek többségében a holtpont kialakulásának valószínűsége meglehetősen kicsi, bár néhány tényező ezt a valószínűséget jelentősen megnövelheti. Minél többféle típusú, de típusonként minél kevesebb erőforrásért minél több folyamat verseng, a holtpont kockázata általában annál nagyobb. Minél hosszabb ideig tartják lefoglalva a folyamatok az erőforrásokat, és minél inkább alkalmazzák a „rákérést”, azaz újabb erőforrások igénylését a már használtak mellé, a holtpont kialakulásának valószínűsége ugyancsak annál nagyobb.

Ugyanakkor látni fogjuk, hogy az *észlelés és feloldás* a kritikus rendszerekben nem jelent minőségi különbséget a probléma figyelmen kívül hagyásához képest, a *megelőzésnek* és *elkerülésnek* pedig hatékonyságromlásban jelentkező ára van.

Végül is tervezői mérlegelés kérdése, hogy egy rendszerben milyen alapállás és milyen megoldás a legmegfelelőbb. Tény, hogy az általános célú, közismert operációs rendszerek általában nem foglalkoznak a problémával, legfeljebb a belső funkciók tervezésekor törekedtek arra, hogy a holtpont kialakulásának valószínűségét csökkentsék. A felhasználó együttműködő folyamatainak holtpontra jutása ellen azonban nincsenek beépített védelmek.

2.2.7.4. A holtpont észlelése

A rendszer futása során több jel mutathat arra, hogy holtpont van. Bizonyos funkciók nem élnek, a rendszer lelassul, parancsokra nem, vagy nagyon lassan reagál. A gyanú felmerülésekor – vagy óvatosságból gyakrabban, például bizonyos időközönként, vagy eseményekhez kötötten is – elindíthatunk a rendszerben egy *holtpontdetektálást* végző programot, amelyik felderíti, hogy vannak-e a rendszerben holtponton lévő folyamatok, és ha igen, melyek azok. Ha vannak, akkor a holtpontot érdemes felszámolni, ami többnyire rombolással jár, de annyival jobb a helyzet, mint a probléma figyelmen kívül hagyása esetén, hogy nem kell a teljes rendszert újraindítani, hanem megúszhatjuk néhány folyamatra kiható beavatkozással.

A detektálás indítása történhet az operációs rendszer által automatikusan, vagy kezelői parancsra. Megjegyzendő, hogy ha nagyon óvatosak vagyunk, akkor az operációs rendszer akár minden erőforráskérés teljesítése után ellenőrizheti, nem vezetett-e holtpontra a kérés teljesítése. Ezzel azonban körülbelül akkora adminisztrációs terhelést (overhead) okozunk a rendszerben a detektáló program gyakori futtatásával, mintha az *elkerülés* érdekében minden kérés teljesítése *előtt* hajtanánk végre ellenőrzéseket, tehát akkor már inkább az elkerülést válasszuk.

Milyen algoritmus szerint működjön a detektáló program? Érdemes megkülönböztetni az egypéldányos és többpéldányos erőforrások esetét, ugyanis egyszerűbb algoritmust alkalmazhatunk, ha valamennyi erőforrásfajtából csak egyetlen példányunk van.

Kezdjük az általános esettel, azaz a többpéldányos erőforrásokkal. Mutassuk be a problémát egy példán. Egy adott fajtájú erőforrásból a rendszerben van 10 példány. A rendszerben négy folyamat (*P1*,...,*P4*) működik. A pillanatnyi helyzetet a 2.9. ábra szemlélteti.

	FOGLAL	KÉR
P1	4	4
P2	1	0
P3	3	4
P4	1	2

2.10.ábra. táblázat - Többpéldányos erőforrások maximális igény előrejelzésével

Az erőforrások közül 9 a FOGLAL oszlop szerint már ki lett osztva, 1 szabad példánya van még a rendszernek. *P1*, *P3* és *P4* várakozik a *KÉR* oszlop szerinti példányra (nyilván egyik sem szolgálható ki, mert nincs elég szabad példány), *P2* pedig fut. Vajon vannak-e a rendszerben holtponton lévő folyamatok?

P2 nyilván nincs holtponton, hiszen fut. *P1*, *P3* és *P4* várakozik, de vajon egymásra várnak-e, vagy van még esélyük a továbblépésre? A válasz némi elemzést igényel. Meg kell vizsgálni a továbblépési esélyeket. Pillanatnyilag csak *P2* fut, ő fel tud szabadítani erőforrásokat, legfeljebb annyit, amennyi van nála. Ez 1 példány, ezzel együtt a rendszernek 2 szabad példánya lesz. Ez mire elég? *P1*-nek nem, *P3*-nak nem, de *P4*-nek igen. Így *P4*-nek van továbblépési esélye. Ha továbblép, arra is van esély, hogy felszabadítsa a nála lévő erőforrásokat. Már nála van egy, a továbblépéshez kap kettőt, tehát három példányt tud felszabadítani. Mivel a rendszernek nincs további szabad példánya, ebben az esetben összesen ez a 3 lesz szabad. Sajnálatosan ez sem *P1*, sem *P3* számára nem elég, így innen már nincs továbblépési lehetőség. Az elemzés végeredménye tehát az, hogy *P1*-nek és *P3*-nak nincs továbblépési esélye, *P1* és *P3* tehát holtponton van.

Az elemzésnek ezt a gondolatmenetét általánosítja a Coffman és szerzőtársai által 1971-ben publikált algoritmus több erőforrástípusra.

Legyen a folyamatok száma *N*, az erőforrástípusok száma *M*.

Tároljuk a szabad erőforrások számát a *SZABAD* nevű, *M* elemű vektorban, az egyes folyamatok által már lefoglalt erőforrások számát a *FOGLAL, N×M* elemű mátrixban, a várakozó kéréseket a *KÉR, N×M* elemű mátrixban.

Jelentse *FOGLAL[i]* a *Pi* folyamat által az egyes erőforrásfajtákból foglalt mennyiséget tároló vektort, azaz a *FOGLAL* mátrix i-edik sorát, hasonlóan *KÉR[i]* pedig a *Pi* folyamat egyes erőforrástípusokra vonatkozó várakozó kéréseit, az a *KÉR* mátrix i-edik sorát.

Az algoritmus alapgondolata, hogy szisztematikusan megkeresi azokat a folyamatokat, amelyeknek továbblépési esélye van, ha végül maradnak olyan folyamatok, amelyeknek nincs, azok holtponton vannak. A keresés minden lépésében olyan folyamatot keres, amelynek várakozó kérései kielégíthetők a rendelkezésre álló erőforrásokkal. Ha van ilyen folyamat, akkor az kivehető a további vizsgálatból (van továbblépési esélye), a rendelkezésre álló készlet pedig megnövelhető a folyamat által birtokolt erőforrásokkal (hiszen van rá esély, hogy a továbbinduló folyamat ezeket felszabadítja), és a következő folyamat a megnövelt készlettel kereshető. Az algoritmus úgy zárul, hogy nem talál újabb folyamatot. Ennek egyik lehetséges oka, hogy elfogytak a folyamatok – ilyenkor nincs holtpont a rendszerben; másik lehetséges oka, hogy nincs köztük olyan, amelyiknek elég az éppen rendelkezésre álló készlet – ilyenkor a megmaradt folyamatok holtponton vannak.

Az algoritmus egy *GYŰJTŐ* nevű, *M* elemű vektort használ a továbbléptethető folyamatoktól visszakapott erőforrások akkumulálására, valamint egy *TOVÁBB* nevű, *N* elemű logikai változó vektort azoknak a folyamatoknak a kipipálására, amelyeket továbbléptethetőnek talált.

Az algoritmus 2. lépésében a *KÉR[i] ≤ GYŰJTŐ* reláció (két vektor összehasonlítása) akkor igaz, ha *KÉR[i],[j] ≤ GYŰJTŐ[j]* fennáll minden *j*-re (*j*=1,2,...,*M*), azaz a kérést valamennyi erőforrásfajtából ki lehet elégíteni.

A Coffman-féle holtpontdetektáló algoritmus 1. Kezdőértékek beállítása: GYŰTŐ := SZABAD TOVÁBB[i] := *hamis* minden i-re (i=1,2,...,N) **2. Továbblépésre esélyes folyamatok keresése: Keress i-t**, amelyre (TOVÁBB[i] = *hamis* ÉS KÉR[i] ≤ GYŰJTŐ); **Ha van ilyen i, akkor** GYŰJTŐ := GYŰJTŐ + FOGLAL[i]; TOVÁBB[i] := *igaz*; ismételd a 2. lépést; **Egyébként** folytasd a 3. lépéssel **3. Kiértékelés: Ha** TOVÁBB[i] = *igaz* minden i-re (i=1,2,...,N), akkor *NINCS HOLTPONT*; **Egyébként** A Pi folyamatok, amelyekre TOVÁBB[i] = *hamis*, *HOLTPONTON VANNAK*.

A fenti algoritmus természetesen *egypéldányos erőforrások* esetén is működik, azonban ilyenkor egyszerűbb, kedvezőbb komplexitású algoritmus is használható. Egypéldányos erőforrások esetén holtpont van a rendszerben, ha az erőforrásfoglalási gráfon kör alakul ki, és azok a folyamatok vannak holtponton, amelyek a kör csomópontjaiban találhatók. Az irányított gráfok kördetektáló algoritmusai hatékonyabbak az általános esetre bemutatott Coffman-algoritmusnál, ezért egypéldányos esetben inkább ezek használata célszerű.

2.2.7.5. A holtpont feloldása

Ha a holtpont már kialakult, általában nem számolható fel veszteségek nélkül. Ahhoz, hogy a holtponton lévő folyamatok továbbléphessenek, illetve az általuk foglalt erőforrások felszabaduljanak, valamilyen erőszakos megoldáshoz kell folyamodni. A lehetséges megoldásokat közelíthetjük a folyamatok, illetve az erőforrások oldaláról, de az eredmény minkét esetben hasonló. Ha a folyamatok oldaláról közelítünk, ki kell lőni (abortálni kell) egy vagy több folyamatot a holtponton lévők közül, ezeket egyszersmind megfosztjuk valamennyi erőforrásuktól. Az erőforrások oldaláról közelítve a feloldáshoz erőszakkal kell elvenni erőforrásokat egy vagy több folyamattól. Ennek következménye – hacsak az erőforrás állapota nem menthető – az, hogy az érintett

folyamatokat vagy kezdőpontjukra kell visszaállítani (ez egyenértékű az abortálással), vagy legalábbis egy olyan állapotba kell visszaküldeni, ahol még nem használták az erőforrásokat, és ahonnan ismételhető a végrehajtásuk (rollback). A holtpont feloldását végezheti a rendszer kezelője manuálisan, vagy megkísérelheti az operációs rendszer automatikusan.

A holtpont feloldásakor – bármelyik oldalról közelítünk is – a következő problémákkal találkozunk:

- *Dönteni kell, hogy radikális, vagy kíméletes megoldást válasszunk-e.* A radikális megoldás, ha valamennyi, a holtpontban érintett folyamatot felszámoljuk. Ez biztosan megszünteti a holtpontot és nem merül fel kiválasztási költség. Mérlegelés esetén meg kell vizsgálni, hogy a választott folyamatok felszámolása elegendő-e a holtpont megszűnéséhez. Ezen túlmenően az áldozatok kiválasztásához szempontrendszert kell felállítani, és ennek megfelelő döntési algoritmust kell végrehajtani, ami időt és erőforrásokat igényel. A mérlegelendő szempontok között szerepelhet például, hogy
    - nincsenek-e menthető állapotú erőforrások, amelyek elvétele esetén nincs szükség abortálásra, vagy visszaállításra,
    - a lehető legkevesebb folyamatot kelljen felszámolni,
    - milyen a folyamatok prioritása,
    - mekkora részét végezték már el feladataiknak (ez veszne kárba).
- *Biztosítani kell a folyamatok visszaállíthatóságát.* Ha arra gondolunk, hogy egy folyamat létrehozhat és módosíthat fájlokat, egyéb maradandó változásokat okozhat a rendszerben, sőt holtpontra jutásakor félkész, inkonzisztens állapotot hagyhat maga után, az sem nyilvánvaló, hogy egy folyamat következmények nélkül abortálható és újraindítható. Közbenső visszaállítási pontok létrehozásához pedig legalább a végrehajtáskor érintett utolsó visszaállítási ponthoz tartozó állapotot tárolni kell. Ezt a problémát az operációs rendszer önmaga gyakorlatilag nem tudja megoldani, a folyamatokat is fel kell készíteni arra a lehetőségre, hogy sor kerülhet újraindításukra, vagy visszaállításukra.

2.2.7.6. A holtpont megelőzése

Ha a holtpont kialakulása nem engedhető meg egy rendszerben, védekezni kell ellene. Ennek egyik módja, hogy gondosan tervezünk, és valamelyik szükséges feltétel kiküszöbölésével strukturálisan holtpontmentes rendszert hozunk létre, ezzel *megelőzzük* a holtpont kialakulását. Az így tervezett rendszernek futás közben nem kell foglalkoznia az erőforrások kiadásakor a pillanatnyi helyzethez igazodó döntéshozatallal, hiszen működése során eleve nem alakulhat ki holtpont.

Vegyük sorra, milyen lehetőségeink vannak arra, hogy a holtpont kialakulásának szükséges feltételeit kiküszöböljük!

Kölcsönös kizárás

A kiküszöbölésre gyakorlatilag nincs esélyünk. Bizonyos mozgásterünk van abban, hogy csökkentsük a kizárólagosan használt erőforrások számát. Például egy rekord vagy fájl esetén, amelyeket egyébként indokoltan nyilvánítunk erőforrásnak, megengedhetjük az olvasások egyidejűségét. Egy másik lehetőség a kizárólagos használati szakaszok oszthatatlan műveletként történő megvalósítása, és ezeknek az oszthatatlan műveleteknek a sorosítása. Erre példa az operációs rendszerekben gyakran alkalmazott fájlonkénti nyomtatás. A folyamatok a nyomtató hosszú távú lefoglalása helyett egy saját használatú fájlban tárolják a nyomtatandó adatokat, majd a fájlt küldik el a nyomtató várakozási sorába. A sort egy önálló nyomtatófolyamat dolgozza fel fájlonként sorosan, így a fájlok között nincs konfliktus, és nem szükséges a nyomtató erőforrásként történő lefoglalása. Finomabb léptékben a folyamatok közös memóriájának PRAM-modellje is ezt az elvet követi, de alkalmazhatjuk a megoldást rekordok felülírása esetén is. Megjegyzendő, hogy a kölcsönös kizárás problémáját ezek a megoldások nem száműzik a rendszerből, csupán az időtávot csökkentik, és a megoldás felelősségét a rendszer egy jól kézbentartható területére korlátozzák. A nyomtató sorába való beláncolás, vagy a PRAM csővezetékébe való parancselhelyezés ugyanis maga is igényli, hogy rövidebb távú kölcsönös kizárások érvényesüljenek.

Foglalva várakozás

Foglalva várakozás akkor alakul ki, ha egy folyamat már birtokol erőforrást, és ekkor kér újabbat, amire várakoznia kell. Ez a helyzet elkerülhető, ha minden folyamat betartja azt a szabályt, hogy az egyidejűleg szükséges valamennyi erőforrását egyetlen rendszerhívással kéri el. A szabály betartásával megelőzhető a holtpont, de ára az erőforrás-kihasználás jelentős romlása. Ha ugyanis a folyamatnak van olyan futási szakasza, amelyiken több erőforrást is használ egyidejűleg, akkor ezek mindegyikét már akkor le kell foglalnia, amikor az elsőre szüksége van. Így a többit akkor is leköti, amikor még nem használja őket.

Nincs erőszakos erőforráselvétel

Ezt a feltételt menthető állapotú erőforrások esetén küszöbölhetjük ki problémamentesen, ellenkező esetben az erőforrás elvétele egyben a folyamat abortálását, vagy legalábbis egy korábbi állapotba való visszaállítását okozza.

Az egyik erőszakos megoldás az erőforrást kérő folyamatot bünteti. Amelyik folyamat olyan erőforrást kér, amire várnia kellene, várakozásba is kerül, egyben valamennyi már birtokolt erőforrását is elveszti, és csak akkor folytatódhat, ha ezeket is, és a kérteket is egyidejűleg vissza tudja kapni.

A másik megoldás az erőforrást kérő folyamatot igyekszik előnyben részesíteni. Ha a folyamat kérése nem elégíthető ki a szabad készletből, a rendszer a már amúgy is várakozó folyamatoktól elvett erőforrásokkal igyekszik teljesíteni a kérést. Ha így sem sikerül, csak akkor kell a kérő folyamatnak várakoznia, ami persze azt jelenti, hogy közben tőle is vehetnek el erőforrást. A folyamat akkor folytatódhat, ha a kért és az esetleg közben elvett erőforrásokat is egyszerre megkaphatja.

Körkörös várakozás

Ez a feltétel kiküszöbölhető, ha a folyamatok mindegyike betart egy viselkedési szabályt, amelyik kicsit bonyolultabb, mint az „egyszerre kérem az erőforrásaimat”, de kevésbé rontja az erőforrás-kihasználást.

A körkörös várakozás az erőforrásfoglalási gráfon keletkező körnek felel meg. Egy ilyen kör alakja: $Pi \rightarrow Rj \rightarrow Pi+1 \rightarrow Rj+1 \rightarrow \ldots. Pi+n \rightarrow Rj+n \rightarrow Pi$

Tegyük fel, hogy a folyamatok megegyeznek az erőforrástípusok sorszámozásában. Viselkedjen minden folyamat úgy, hogy csak nagyobb sorszámú erőforrást kérhet azoknál, amelyek már a birtokában vannak. Vegyük észre, hogy ekkor a gráf minden útja csak növekvő sorszámú erőforrásokon át vezethet (a folyamatokba befutó élek kisebb sorszámú erőforrástól indulnak, mint ahova a folyamatból kivezető élek befutnak), így ezek az utak sohasem záródhatnak, nem alakulhat ki kör az erőforrásfoglalási gráfon.

2.2.7.7. A holtpont elkerülése

A holtpont elleni védekezés másik lehetősége a megelőzés mellett az elkerülés. Az elkerülés alapelve, hogy a rendszer minden erőforrásigény kielégítése előtt mérlegeli, hogy nem vezet-e holtpontveszélyre a kérés teljesítése, másszóval, fennmarad-e a *biztonságos* állapot. Így lehetséges, hogy egy kérést akkor sem elégít ki, és a kérő folyamatot várakoztatja, ha egyébként elegendő számú szabad példány áll rendelkezésre a kérés teljesítéséhez. Az elkerülés tehát a védekezés egy dinamikus formája, amelyik futási idejű helyzetelemzést igényel. Nem vezet be az erőforrás-kihasználást rontó megkötéseket, ugyanakkor adminisztrációs teljesítményveszteséget okoz.

Kérdés, hogy milyen algoritmussal dönthető el, hogy egy kérés kielégítése esetén biztonságos marad-e a rendszer állapota. Ismét csak érdemes megkülönböztetni az egypéldányos, illetve többpéldányos erőforrások esetét. Kezdjük ismét az általános, *többpéldányos* esettel és egy egyszerű példával!

A rendszerben négy folyamatunk van (*P1*,...,*P4*), és egy erőforrástípusból tíz példányunk. Az aktuális helyzetet a 2.10. ábra mutatja. Tudjuk, hogy futásuk során az egyes folyamatoknak egyidejűleg legfeljebb hány példányra lesz szükségük *(MAXIMÁLIS IGÉNY)*. A már megkapott mennyiséget a *FOGLAL* oszlop mutatja, a *MÉG KÉRHET* oszlop pedig a két előző különbsége, legfeljebb ekkora igénnyel jelentkezhet a továbbiakban egy-egy folyamat. Ennek alapján a rendszernek még 3 szabad példánya van. Pillanatnyilag *P3* jelentkezett 2 példányért, és *P4* 1 példányért. A szabad mennyiségből mindkét kérés kielégíthető. Vizsgáljuk meg, mi történik, ha kielégítjük mindkét kérést!

	MAXIMÁLIS IGÉNY	FOGLAL	MÉG KÉRHET	KÉR
*P*1	7	3	4	0
*P*2	7	0	7	0
*P*3	8	1	7	2
*P*4	6	3	3	1

2.17.ábra. táblázat - Elérési mátrix statikus védelmi tartományokkal

*P*3 sora a táblázatban ekkor 8,3,5,0 lesz, *P4*-é pedig 6,4,2,0. A rendszernek nem marad szabad példánya. Mi történhet ezután? A legrosszabb esetben a következő pillanatban minden folyamat bejelenti a még kérhető maximális igényét, azaz *P1* 4, *P2* 7, *P3* 5, *P4* pedig 2 példányt kér. A rendszernek nincs szabad példánya, egyik kérést sem tudja kielégíteni, tehát holtpont alakul ki. Mindkét kérés kielégítése tehát veszélyes, a rendszer holtpontra juthat.

Most tegyük fel, hogy csak *P4* kérését teljesítjük. Ekkor a táblázatban *P3* sora 8,1,7,2 marad, és a folyamat várakozik, *P4* sora pedig 6,4,2,0 lesz. A rendszernek marad 2 szabad példánya. Tegyük fel most is, hogy a következő pillanatban a legrosszabb eset lép fel, azaz minden folyamat jelentkezik a még kérhető maximumra vonatkozó igényével, azaz *P1* 4, *P2* 7, *P3* további 5, tehát összesen 7 (ez úgy értendő, hogy ha megkapja a várt 2 példányt, a következő pillanatban kérhet újabb 5-öt), *P4* pedig 2 példányt kér. Mit tehet most a rendszer? *P4* kérését teljesíteni tudja a 2 szabad példánnyal. *P4* már nem kérhet többet, feltételezhetjük, hogy lefut és felszabadítja a nála lévő valamennyi erőforrást. Ekkor a rendszernek 6 szabad példánya lesz. A még várakozó folyamatok közül ki tudja elégíteni *P1* kérését (4), *P1* is lefuthat, és felszabadítja erőforrásait. Ezután a rendszernek már 9 szabad példánya lesz. Ezzel már *P2* vagy *P3* kérése is teljesíthetővé válik, bármelyiket teljesítjük először, a folyamat lefuthat és újabb erőforrások szabadulnak fel, tehát az utolsónak hagyott folyamat igénye is teljesíthető lesz. Megállapíthatjuk tehát, hogy ha a rendszer továbbra is kellő óvatosságot tanúsít, akkor nem alakul ki holtpont. Az az állapot tehát, amelyik az eredeti helyzetből *P4* kérésének teljesítésével kialakul, *biztonságos*.

A bemutatott gondolatmenetet követi a problémát több erőforrásfajtára általánosan megoldó bankár-algoritmus (Dijkstra, 1965). A név onnan származik, hogy a probléma hasonló a bankok erőforrás-kihelyezési problémájához. Hitelezők fordulnak a bankhoz beruházásaik finanszírozásához szükséges hitelekért. Közlik a maximális hiteligényüket, amelyet a bank a megvalósítás ütemében folyósít. A bank véges tőkével rendelkezik. A hitelt csak az az ügyfél tudja visszafizetni, aki be tudta fejezni a beruházást, ehhez pedig szüksége van arra, hogy előbb-utóbb hozzájusson az igényelt hitel teljes összegéhez. Ha a bankár nem elég óvatos, ott állhat egy csomó félkész beruházással és kiürült kasszával, a hitelek következő részletét nem tudja folyósítani, ügyfelei pedig nem tudnak törleszteni. Hogy viselkedjen a bankár, hogy elkerülje az ilyen helyzeteket (kicsit életszerűtlenül tételezzük fel, hogy az állam segítségére sem számíthat)?

Legyen a folyamatok száma *N*, az erőforrástípusok száma *M*.

A folyamatoktól elvárjuk, hogy előzetesen bejelentik erőforrástípusonként a maximális igényeiket. Ezt a *MAX* nevű, *NxM* elemű mátrixban tartjuk nyilván.

Tároljuk a szabad erőforrások számát a *SZABAD* nevű, *M* elemű vektorban, az egyes folyamatok által már lefoglalt erőforrások számát a *FOGLAL, NxM* elemű mátrixban. Jelöljük a *MAX – FOGLAL* mátrixot *MÉG* névvel. *MÉG* ugyancsak *NxM* méretű, és elemei azt jelölik, hogy egy folyamat egy erőforrásból még legfeljebb mekkora igényt nyújthat be.

Jelentse *FOGLAL[i]* a *Pi* folyamat által az egyes erőforrásfajtákból foglalt mennyiséget tároló vektort, azaz a *FOGLAL* mátrix *i*-edik sorát, hasonlóan *MÉG[i]* a *MÉG* mátrix *i*-edik sorát.

A folyamatok várakozó kéréseit a *KÉR* mátrix (*NxM* elemű, *i*-edik sora a *Pi* folyamat várakozó kérésének adatait tartalmazza, jelölése *KÉR[i]*) tárolja.

Az algoritmus arra ad választ, hogy egy kiválasztott folyamat várakozó kérése kielégíthető-e úgy, hogy a rendszer biztonságos állapotban maradjon.

Az algoritmust két szinten tárgyaljuk. Az első szint ellenőrzi a kérést, és ha érdemes vele foglalkozni, módosítja a nyilvántartást a kérés teljesítése után kialakuló értékekre. Ezután megvizsgálja, hogy ez az állapot biztonságos-e. Ha igen, a kérés teljesíthető, a folyamat megkapja az erőforrásokat. Ha nem, a folyamatnak várakoznia kell, a nyilvántartást pedig vissza kell állítani a vizsgálat kezdetén fennálló értékekre.

Bankár-algoritmus (első szint)**: 1. A kérés ellenőrzése: Ha** KÉR[i] > MÉG[i] **akkor** *STOP***;** (HIBA, a folyamat hazudós) **Ha** KÉR[i] > SZABAD **akkor** VÉGE; (nincs elég, várni kell) **2. A nyilvántartás átállítása az új állapotra:** SZABAD := SZABAD **–** KÉR[i]; FOGLAL[i] := FOGLAL[i] **+** KÉR[i]; **3. Biztonságosság vizsgálata 4. Ha nem** BIZTONSÁGOS **akkor állapot visszaállítása:** SZABAD := SZABAD + KÉR[i]; FOGLAL[i] := FOGLAL[i] **–** KÉR[i]; VÉGE; (várni kell) **Egyébként kérés teljesítése;** VÉGE; **Biztonságosság vizsgálata** (2. szint, előző 3. pont): **B1. Kezdőértékek beállítása:** GYŰJTŐ := SZABAD LEFUT[i] := *hamis* minden i-re (i=1,2,...,N) **B2. Továbblépésre esélyes folyamatok keresése: Keress i-t**, amelyre (LEFUT[i] = *hamis* ÉS MÉG[i] ≤ GYŰJTŐ); **Ha van ilyen i, akkor** GYŰJTŐ := GYŰJTŐ + FOGLAL[i]; LEFUT[i] := *igaz*; ismételd a B2. lépést; **Egyébként** folytasd a B3. lépéssel **B3. Kiértékelés: Ha** LEFUT[i] = *igaz* minden i-re (i=1,2,...,N), akkor *BIZTONSÁGOS* **Egyébként** *NEM BIZTONSÁGOS;* (A Pi folyamatok, amelyekre LEFUT[i] = *hamis*, holtpontra juthatnak)

A második szinten a biztonságosság vizsgálatát mutatjuk be. Az algoritmus alapgondolata, hogy szisztematikusan megkeresi azokat a folyamatokat, amelyek a legkedvezőtlenebb esetben is le tudnak futni. A legkedvezőtlenebb eset az, ha az adott pillanatban mindenki a még kérhető maximális igénnyel jelentkezik. A keresés minden lépésében olyan folyamatot keres, amelynek a *még kérhető maximális* igénye is kielégíthető a rendelkezésre álló erőforrásokkal. Ha van ilyen folyamat, akkor az kivehető a további vizsgálatból (biztosan le tud futni), a rendelkezésre álló készlet pedig megnövelhető a folyamat által birtokolt erőforrásokkal (hiszen a folyamat lefutása után ezek felszabadulnak), és a következő folyamat a megnövelt készlettel kereshető. Az algoritmus úgy zárul, hogy nem talál újabb folyamatot. Ennek egyik lehetséges oka, hogy elfogytak a folyamatok – ilyenkor az állapot biztonságos, hiszen minden folyamat biztosan le tud futni; másik lehetséges oka, hogy nincs köztük olyan, amelyiknek elég az éppen rendelkezésre álló készlet – ilyenkor az állapot nem biztonságos, hiszen a legkedvezőtlenebb esetben a maradék folyamatok holtpontra juthatnak.

A biztonságosságot vizsgáló algoritmus egy *GYŰJTŐ* nevű, *M* elemű vektort használ a lefutó folyamatoktól visszakapott erőforrások akkumulálására, valamint egy *LEFUT* nevű, *N* elemű logikai változó vektort azoknak a folyamatoknak a kipipálására, amelyeket továbbléptethetőnek talált.

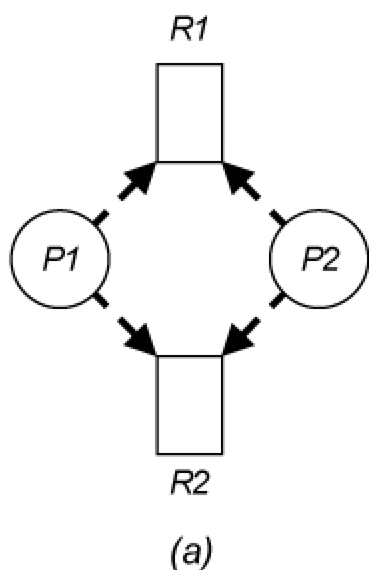
Az algoritmusban a vektorok összehasonlítása (például *MÉG[i] ≤ GYŰJTŐ* reláció) akkor igaz, ha *MÉG[i],[j] ≤ GYŰJTŐ[j]* fennáll minden *j*-re (*j*=1,2,...,*M*), azaz a kérést valamennyi erőforrásfajtából ki lehet elégíteni.

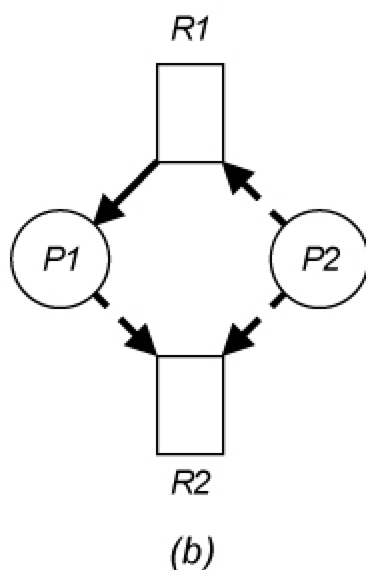
Az algoritmus emlékeztet a korábban tárgyalt Coffman-féle holtpontdetektáló algoritmusra, a két algoritmus megszületésének időrendje azonban tárgyalási sorrendünkkel ellentétes.

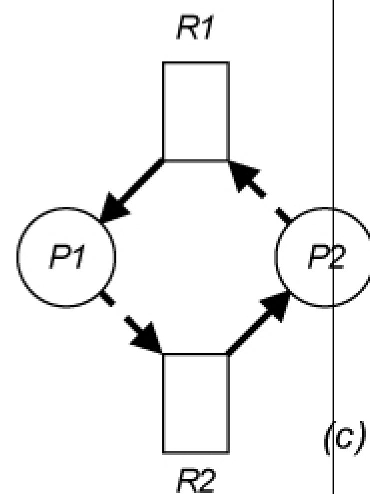
A detektáló algoritmusokhoz hasonlóan *egypéldányos erőforrások* esetére ismét csak érdemes az erőforrásfoglalási gráfhoz fordulni, mert ott hatékonyabb algoritmusokat találhatunk.

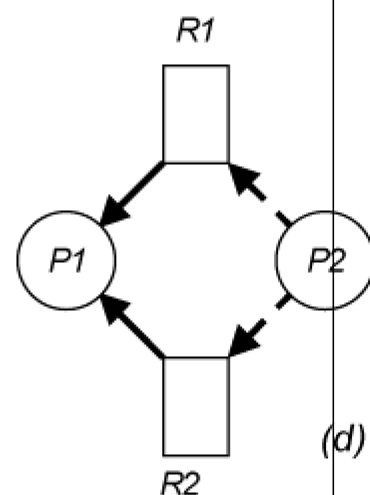
A biztonságosság vizsgálatához szükséges előzetes információ egypéldányos esetben azt jelenti, hogy fogja-e használni a folyamat az adott erőforrást. Ennek jelölésére bevezethetünk a gráfon egy új éltípust, az ún. *lehetséges kérés* élet. Ez az él folyamattól erőforráshoz vezet, és jelöli, hogy egy folyamat kérheti az adott erőforrást. A biztonságosság vizsgálatakor a legrosszabb eset az, ha valamennyi lehetséges kérés fellép, azaz a lehetséges kérések valódi kérésekké válnak. Ha tehát egy kérés teljesítését mérlegeljük, kísérletképpen módosítjuk az erőforrásfoglalási gráfot, mintha odaadtuk volna az erőforrást (egy kérés él iránya megfordul és foglalás éllé válik). Ha most a lehetséges kérés éleket is figyelembe véve kört találunk a gráfon, a kérés teljesítésekor kialakuló állapot nem biztonságos, a kérést nem szabad teljesíteni.

Az elmondottakat a 2.11. ábra szemlélteti két folyamat és két erőforrás egyszerű esetére. A lehetséges kérés éleket szaggatott vonal jelzi. A kezdeti állapotot az *(a)* ábra mutatja. Először *P1* kéri *R1*-et. A vizsgálathoz a rendszer bejegyzi a *(b)* ábrának megfelelően az *R1* ® *P1* foglalás élet. A gráfon láthatóan nincs kör, a kérést a rendszer teljesíti.









2.11. ábra. ábra - Potenciális kérések az erőforrásfoglalási gráfon

Ezután két lehetséges továbblépést mutat a *(c)* és a *(d)* ábra. Ha *P2* kéri *R2*-t, a *(c)* ábra szerinti helyzet alakul ki a kérés teljesítése esetén. A gráfon kör van, az állapot nem biztonságos, a kérést nem szabad kielégíteni. *P1* ugyanis bármikor kérheti *R2*-t, *P2* pedig *R1*-et, és ekkor a rendszer holtpontra jutna. Ha viszont *P1* kéri *R2*-t, a *(d)* ábra szerinti helyzet áll elő, a gráfon nincs kör, a rendszer biztonságos állapotban marad, a kérés teljesíthető.

2.2.7.8. Kombinált stratégiák

A holtpontprobléma kezelésének ismertetett módszerei kombináltan is alkalmazhatók. A rendszer erőforrásai például osztályokba sorolhatók. Az egyes osztályok sorszámozottak, a folyamatok betartják, hogy különböző osztályokhoz tartozó erőforrásokat csak az osztálysorrend szerint foglalnak le. Az egyes osztályokon belül más-más megelőzési vagy elkerülési stratégia is használható.

Egy ismert rendszer például a következő megoldást alkalmazta.

Az erőforrásokat négy osztályba sorolták:

- *Belső erőforrások* (például rendszertáblák, leírók stb.). Mivel ez a rendszerfolyamatokat érinti, a megoldás a rendszertervező kezében van, aki betartathat egy egyezményes foglalási sorrendet, így az osztályon belül is alkalmazható a sorrendezésen alapuló megelőzés.
- *Memória.* Menthető állapotú erőforrás (háttértárra másolás, visszatöltés), erőszakos elvétel használható megelőzésre.
- *Készülékek és fájlok.* Az osztályon belül elkerülés alkalmazható a használati igény előzetes bejelentése alapján.
- *Munkaterület a lemezen.* Általában ismert méretű igények vannak, egyszerre kell kérni a szükséges méretet, nincs „rákérés”.

Az osztályok között a felsorolás szerinti foglalási sorrendet kellett betartani.

2.2.7.9. Kommunikációs holtpontok

Holtponthelyzet nemcsak erőforráshasználat miatt alakulhat ki, hanem a folyamatok tetszőleges olyan együttműködése során, amelyik a folyamatok körkörös várakozására vezethet.

Tételezzünk fel például egy kliens-szerver architektúrájú rendszert, ahol az ügyfelek és a szolgáltatók is folyamatok. Ebben a rendszerben az ügyfél küld egy kiszolgálást kérő üzenetet a szolgáltatónak, majd vár a válaszra. Ha a szolgáltatói és ügyfél szerepeket nem sikerül statikusan elkülöníteni, előfordulhat, hogy egy folyamat egyik kapcsolatában ügyfél, egy másikban azonban szolgáltató. Az is előállhat, hogy egy szolgáltatónak egy ügyfél kiszolgálása közben ügyfélként kell más szolgáltatóhoz fordulnia. Előfordulhat, hogy ilyenkor az ügyfél-kiszolgáló lánc záródik, tehát egy kiszolgálást kérő folyamat a választ csak akkor kaphatja meg, ha a közben – esetleg több áttételen keresztül – hozzá, mint kiszolgálóhoz érkező kérésre válaszol. Ha a folyamat erre nincs felkészítve, ebben az esetben holtpont alakul ki. Az ehhez hasonló problémák kezelésében segítenek a folyamaton belül kialakítható szálak, amelyek különösen a szerver szerepet (is) betöltő folyamatok programozásakor hasznosak.

A kommunikáció által okozott várakozások ugyancsak szemléltethetők irányított gráfokkal, amelyek csomópontjai folyamatok, irányított élei pedig a várakozó folyamattól vezetnek ahhoz a folyamathoz, amelyiknek az üzenetet küldenie kellene. Ilyen várakozási gráf (wait-for graph) egyébként az erőforrásfoglalási gráf redukálásával is létrejöhet. A kördetektáló algoritmusok értelemszerűen itt is használhatók.

2.2.8. Éhezés

A holtpont mellett a folyamatok működésének egy másik zavara lehet az éhezés. Az éhezés azt jelenti, hogy egy várakozó folyamat nincs ugyan holtponton, de nincs rá garancia, hogy véges időn belül továbbindulhasson.

A szinkronizációs eszközök tárgyalásakor bevezettük az *ütemezés* fogalmát. Azokban az esetekben, amikor több várakozó folyamat közül kell továbbindulót választani, a választást gyakran nem célszerű a véletlenre bízni, hanem meghatározott algoritmus szerint kell dönteni. A rendszer a szemaforokhoz, erőforrásokhoz, egyéb

szinkronizációs objektumokhoz várakozási (ütemezési) sorokat rendel, amelyeket meghatározott algoritmusok szerint kezel (így még az esetleges véletlen kiválasztás is tudatossá tehető).

Az ütemezés leggyakoribb alapalgoritmusai az érkezési sorrend szerinti (FIFO) és a prioritásos ütemezés.

Egy ütemezést *tisztességesnek (fair)* nevezünk, ha garantálja, hogy egy várakozási sorból minden folyamat véges időn belül továbbindulhat, ameny-nyiben a rendszerben véges számú folyamat működik és a rendszerben nincs holtpont vagy hibás folyamat (amelyik például nem enged el egy megszerzett erőforrást). Ellenkező esetben az ütemezés *tisztességtelen (unfair)*.

Tisztességes ütemezés például az érkezési sorrend szerinti kiszolgálás, tisztességtelen a statikusan rögzített prioritás szerinti kiszolgálás (ilyenkor nincs garancia arra, hogy egy alacsony prioritású folyamat valaha is hozzájut az erőforráshoz, ha mindig van nála magasabb prioritású igény).

Megjegyzések:

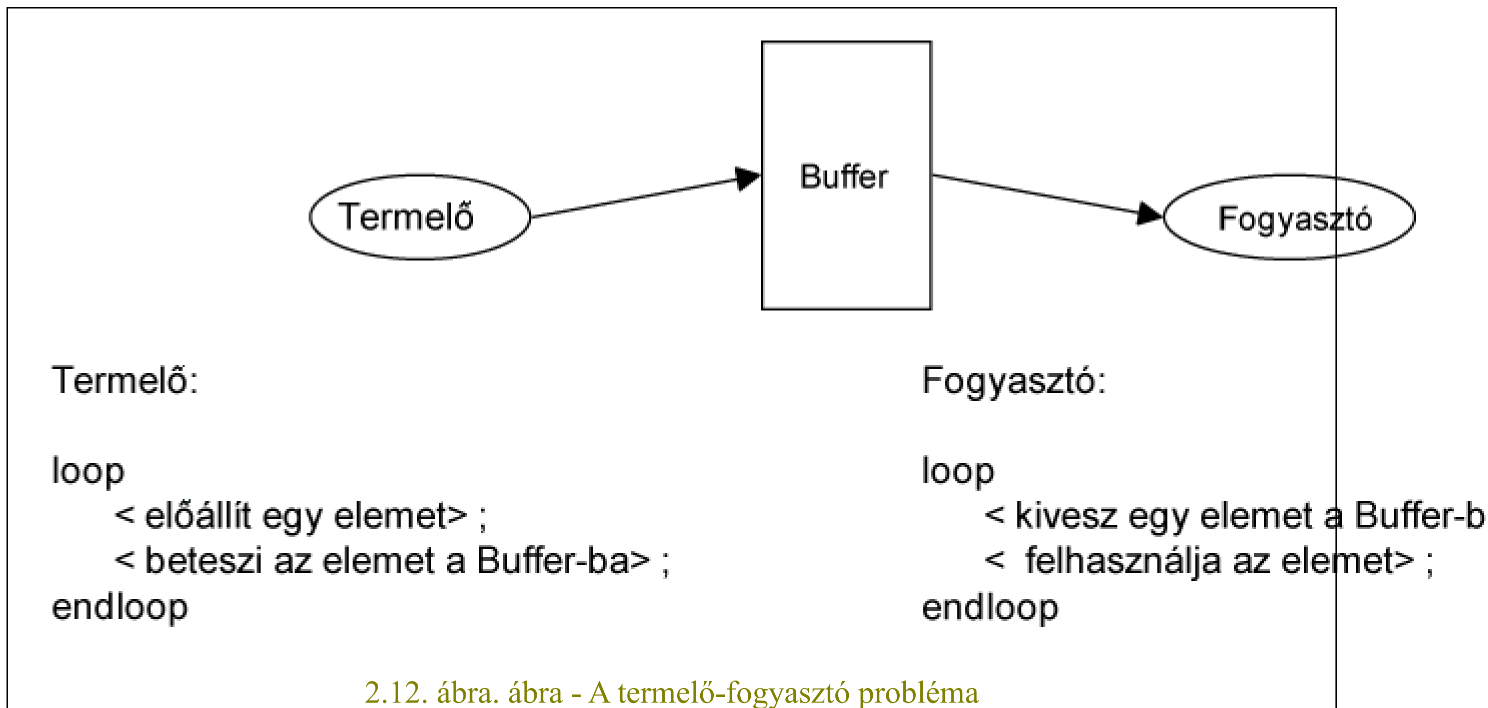
- Az éhezés nem jelent holtpontot.
- Külső eseményre (például kezelői beavatkozás) való meghatározatlan idejű várakozás nem jelent sem éhezést, sem holtpontot.
- Tisztességtelen ütemezés tudatosan is alkalmazható, csak számolni kell a kevésbé fontos funkciók kiesésével a terhelés (a futtatandó folyamatok száma) növekedésekor.

2.2.9. Klasszikus konkurens problémák

Tekintsünk át néhány olyan feladatot, amelyek megoldása jellegzetesen együttműködő folyamatokra vezet! Tapasztalatok szerint a folyamatok használatával megoldható gyakorlati problémák többsége visszavezethető ezeknek a klasszikussá vált feladatoknak valamelyikére. Éppen ezért ezek a problémák egy-egy újabb együttműködési modell, szinkronizációs, vagy kommunikációs eszköz kidolgozása esetén gyakran szerepelnek mint teszt és benchmark esetek.

2.2.9.1. Termelő–fogyasztó probléma

A rendszerben egymással párhuzamosan egy termelési folyamat, valamint egy fogyasztási folyamat zajlik. A *Termelő* saját ritmusában és algoritmusa szerint előállít valamit (például adatokat), amit egy raktárba tesz *(Buffer)*.



2.12. ábra. ábra - A termelő-fogyasztó probléma

A *Fogyasztó* saját ritmusában, saját algoritmusa szerint működve felhasználja a termelő által előállított terméket (adatokat), a soron következőt mindig a raktárból véve elő.

A *Termelő* és a *Fogyasztó* sebességére nincs kikötésünk. A *Buffer* kiegyenlítő hatású, kisebb sebességingadozások esetén nem akadnak fenn a folyamatok, legfeljebb a *Termelő* lelassulása esetén a raktárkészlet fogy, a *Fogyasztó* lelassulása esetén pedig növekszik. Mivel a *Buffer* kapacitása véges, elvárjuk, hogy ha megtelt a *Buffer*, a *Termelő* várjon a következő elem betételével, amíg felszabadul egy hely, illetve ha üres a *Buffer*, a *Fogyasztó* várjon, amíg érkezik egy elem. Ugyancsak elvárjuk, hogy az elemeket a *Fogyasztó* ugyanabban a sorrendben dolgozza fel, ahogyan a *Termelő* előállította azokat.

Hogyan tudnánk olyan programot írni egy adott programozási nyelvet és operációs rendszert feltételezve, amelyik végrehajtásakor a *Termelő* és a *Fogyasztó* párhuzamosan hajtódhat végre, és a *Buffer* kezelése is helyes lesz?

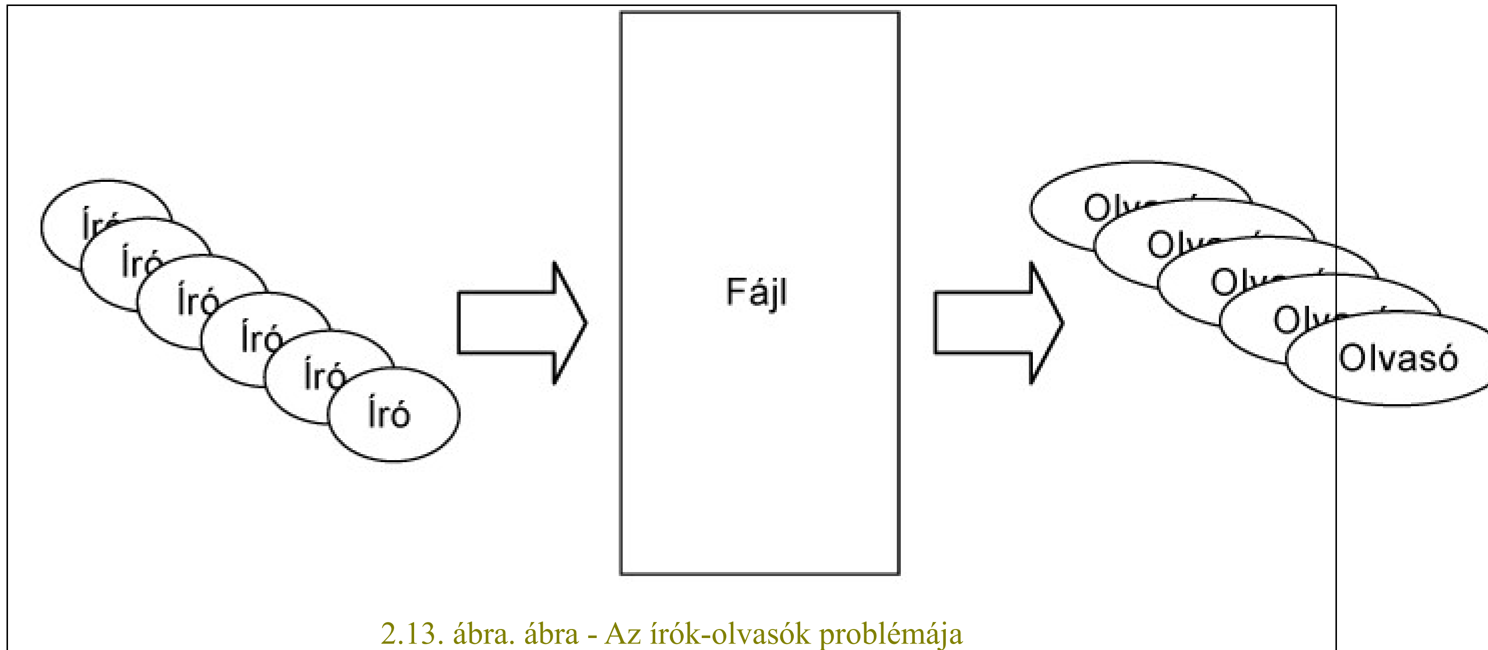
2.2.9.2. Írók–olvasók problémája

Egy adatszerkezetet (például egy hirdetőtáblát vagy egy fájlt) többen kívánnak írni is és olvasni is. Az írások és olvasások csak bizonyos rendszabályok betartása esetén nem zavarják össze egymást:

- egyidejű olvasásokat tetszőleges számban megengedünk, mert nem zavarják egymást,
- írás és olvasás nem folyhat egyidejűleg, mert aki olvas, félig átírt dolgokat látna, amiből valami zagyvaság jöhet ki,

- több írás nem folyhat egyidejűleg, mert ha két írás átlapolódhatna, a végeredmény egyik fele az egyik írás eredménye lehet, a másik fele a másiké, ami nagy valószínűséggel ismét csak zagyvaságot eredményez.

Ennek megfelelően elvárjuk, hogy írás csak akkor kezdődhessen, ha sem írás, sem olvasás nincs folyamatban, olvasás viszont kezdődhet ha más olvasások már folyamatban vannak.



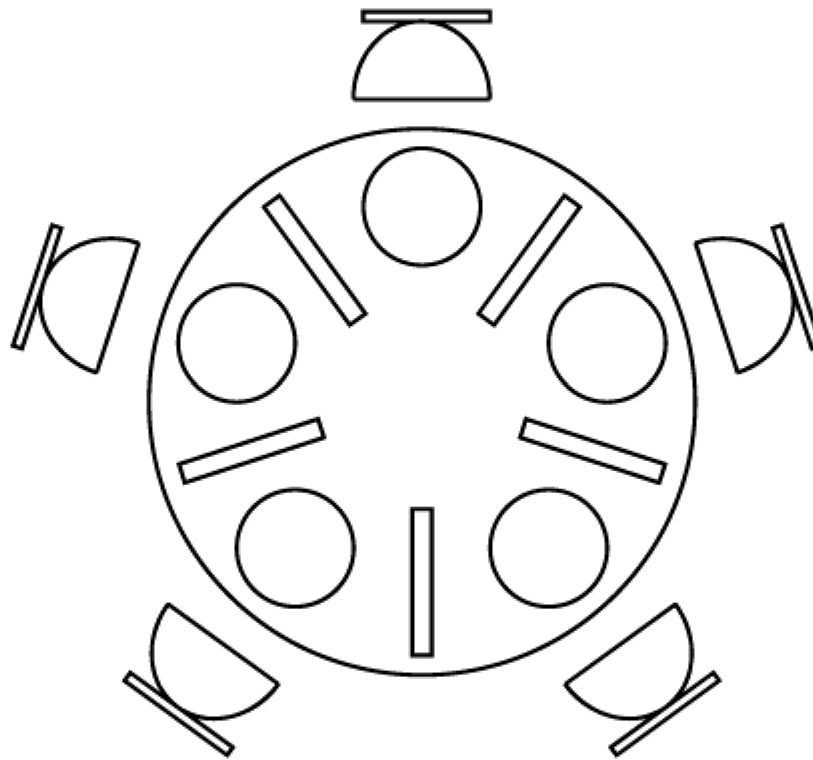
2.13. ábra. ábra - Az írók-olvasók problémája

Megjegyezzük, hogy ésszerű lehet még egy kikötés annak elkerülésére, hogy a folyamatosan érkező olvasók miatt az írni szándékozók a végtelenségig ne jussanak szóhoz: ha van várakozó író, akkor újabb olvasó már ne kerüljön sorra, amíg a várakozó írók nem végeztek (írók–olvasók II. probléma).

Hogyan tudnánk olyan programot írni, amelyik kifejezi az írók és olvasók párhuzamos működését és biztosítja a megadott szabályok betartását (nyelv, operációs rendszer)?

2.2.9.3. Az étkező filozófusok problémája

Egy tibeti kolostorban öt filozófus él. Minden idejüket egy asztal körül töltik (2.14. ábra). Mindegyikük előtt egy tányér, amiből sohasem fogy ki a rizs. A tányér mellett jobb és bal oldalon is egy-egy pálcika található a rajz szerinti elrendezésben.



2.14. ábra. ábra - Az étkező filozófusok problémája

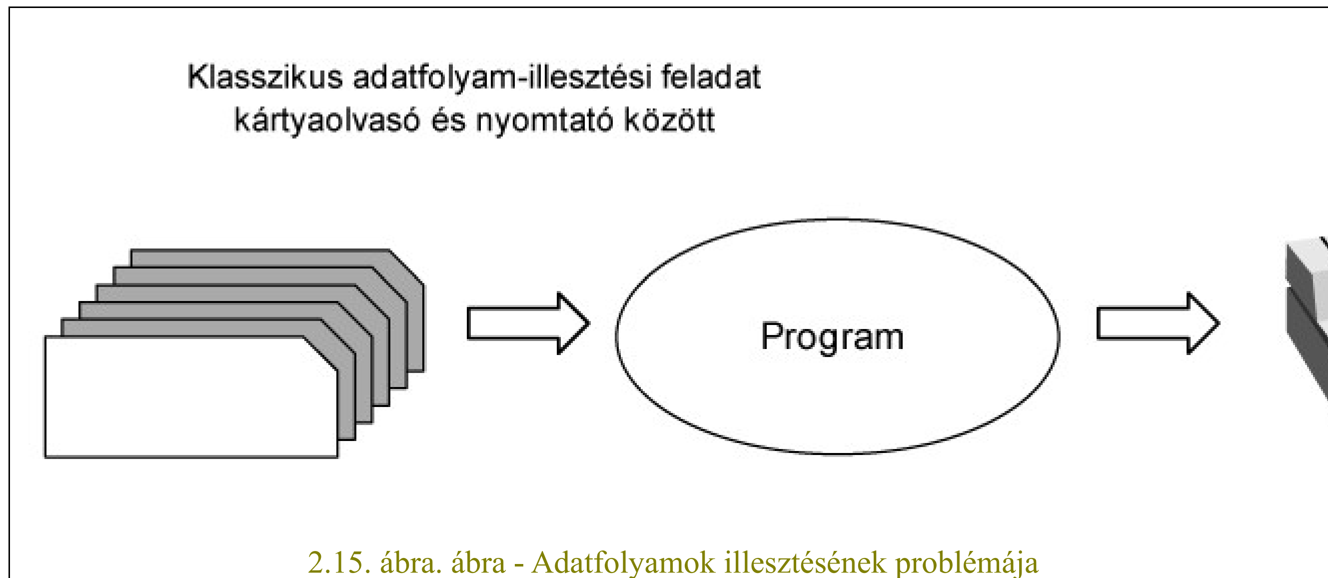
A filozófusok életüket az asztal melletti gondolkodással töltik. Amikor megéheznek, étkeznek, majd ismét gondolkodóba esnek a következő megéhezésig. És ez így megy az idők végezetéig. Az étkezéshez egy filozófusnak meg kell szereznie a tányérja melletti mindkét pálcikát. Ennek következtében amíg eszik, szomszédai nem ehetnek. Amikor befejezte az étkezést, leteszi a pálcikákat, amelyeket így szomszédai használhatnak.

Hogyan kell viselkedniük a filozófusoknak, hogy ne vesszenek össze a pálcikákon, ne kerüljenek olyan megoldhatatlan probléma elé, amin fenn-akadva többé nem tudnak sem enni, sem gondolkozni (például, ha mindegyikük felveszi mondjuk a bal pálcikát és nem teszi le, az holtponthelyzet), és egyikük se haljon éhen, azaz ha enni akar, ez egy idő után a leggyámoltalanabbjuknak is sikerüljön? Hogyan tudnánk olyan programot írni, amelyik leírja a filozófusok viselkedését?

2.2.9.4. Adatfolyamok illesztése

Ez a probléma a CPASCAL *(Concurrent Pascal)* nyelv bevezetésének szokásos mintapéldája. Eredetileg arról szól, hogy adott egy kártyaolvasó és egy nyomtató. A kártyaolvasóba helyezett kártyák egy hosszabb szöveget tartalmaznak, amit ki kell nyomtatni. Maga a szöveg egy bekezdésekre tagolt karakterfolyam, amelyben az új bekezdést speciális karakter jelzi (NL). Egy kártya 80 karaktert tud tárolni, a kártyaolvasóról tehát nyolcvan karakteres blokkokban érkezik az adatfolyam. A szöveget lapokra tördelve, lapszámozással ellátva, a bekezdéseket külön sorban kezdve, soronként 132 pozíciót használva kell kinyomtatni.

A probléma általánosabban úgy fogalmazható meg, hogy egy beérkező, sajátosan strukturált és ütemezett adatfolyamot egy más struktúrájú és ütemezésű adatfolyamként kell továbbítani. Modern változatban könnyen fogalmazhatnánk hasonló feladatot szövegfájlok kezelése, vagy hálózati adatátviteli rendszerek protokolljainak illesztése témakörében.



2.15. ábra. ábra - Adatfolyamok illesztésének problémája

A kérdés ismét az, hogy hogyan írhatunk olyan programot, amelyik maximális sebességgel tudja működtetni mindkét oldalon a készülékeket, és minél tisztább szerkezetben tudja megoldani az eltérő struktúrák illesztését. (Ennél a feladatnál még a folyamatokra bontás sem adott. Ha a feladat megoldásával próbálkozunk, érdemes kísérletet tenni először egyetlen szekvenciális programmal, azaz egyetlen folyamattal, majd legalább két, egy olvasó és egy nyomtató folyamatra való felbontással.)

A bemutatottakon kívül további klasszikus problémákat is találhatunk az irodalomban. Valamennyinek lényeges eleme, hogy több aktív szereplő (több folyamat) együttműködését kell megoldani és leírni. Ajánljuk az olvasónak, hogy vágjon neki, és próbáljon kedvenc programnyelvén (persze annak pszeudo változata is megteszi) megoldási kísérleteket tenni a fenti feladatokra. Egyelőre tételezze fel, hogy írhat olyan programrészleteket, amelyek majd valahogy folyamatként, egymással párhuzamosan fognak végrehajtódni. Válasszon valamilyen együttműködési modellt (közös memória vagy üzenetváltás), és használja a már bemutatott szinkronizációs, kommunikációs eszközök közül azokat amelyeket jónak lát. Tételezze fel, hogy ezek rendszerhívásként elérhetőek.

2.2.10. Folyamatokból álló rendszerek leírása nyelvi szinten

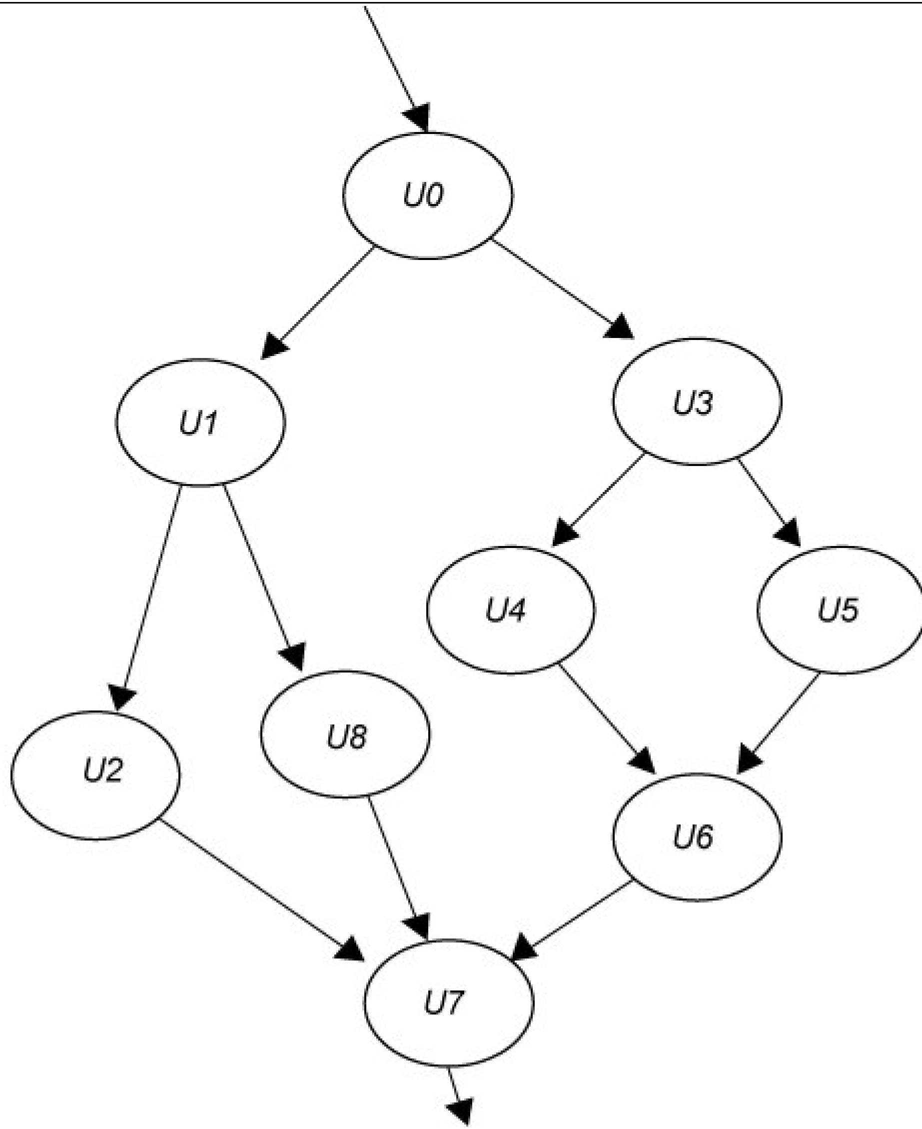
Ahhoz, hogy az előző pontban bemutatott feladatokat megoldó programokat írhassunk, megfelelő *nyelvi eszközök* szükségesek annak kifejezésére, hogy mely programrészletek lesznek önálló folyamatok, azaz hajtódhatnak végre párhuzamosan, valamint milyen műveletekkel oldják meg a kommunikáció és a szinkronizáció problémáit. Ezek nélkül csak „kézi” módszereink vannak: egyenként megírjuk a programokat,

valahogyan elindítjuk őket folyamatként az operációs rendszer kezelői felületéről, és a szinkronizációs, kommunikációs rendszerhívásokat megpróbáljuk a nyelvi interfészen keresztül elérni.

Konkurens (párhuzamosan végrehajtható ágakat tartalmazó) programok készítésének igénye először az operációs rendszerek programozásakor vetődött fel, azonban a multiprogramozott, majd a multitaszkos rendszerek megjelenésekor az alkalmazásfejlesztők ilyen irányú igénye is megjelent. A nyelveknek egyrészt tehát a *párhuzamosság leírására*, másrészt a *folyamatok együttműködésének leírására* kellett modelleket és eszközöket adniuk. A következőkben az erre kialakult megoldásokat mutatjuk be röviden.

2.2.10.1. A párhuzamosság leírása

Tegyük fel, hogy egy feladat megoldása során olyan műveleteket használunk fel, amelyeket már nem tudunk, vagy nem akarunk tovább bontani, részletezni (elemi folyamatok). A megoldáshoz meg kell adnunk azt is, hogy ezek az elemi műveletek milyen vezérléssel (milyen sorrendben, milyen feltételek mellett stb.) hajtandók végre. Ha folyamatokban gondolkozunk, lehetnek ezek között párhuzamosan végrehajthatók, és lehetnek olyanok, amelyek csak egy másik elemi folyamat után hajthatók végre. Ahhoz képest, hogy akár minden elemi folyamat párhuzamosan futhatna, egy sereg precedenciát írunk elő. A végeredményt egy **precedenciagráfon** ábrázolhatjuk (2.16. ábra).



2.16. ábra. ábra - Precedenciagráf

A precedenciagráfon bármelyik irányított úton elhelyezkedő elemi folyamatok *(Ux)* egyetlen folyamattá is összevonhatók. Bármely két utasítás, amelyet nem köt össze irányított út, párhuzamosan is végrehajtódhat, azaz konkurens.

A precedenciagráf szemléletesen kifejezi a párhuzamosságot, de síkban kell ábrázolni, míg a programszövegek lineáris sorozatok.

A programnyelvi leírásra az első javaslat a **fork-join** operátorok alkalmazása volt. A *fork C* utasítás hatására a végrehajtás párhuzamosan elágazik, az egyik szál a következő utasításon, a másik a *C* címkén folytatódik. A *join N* utasítás pedig *N* szál összefuttatását oldja meg. Az utasítás több szálon is végrehajtódhat. Minden végrehajtása eggyel csökkenti az utasításhoz rendelt számláló értékét, amelynek kezdőértéke *N* volt. Az utasítás minden őt

végrehajtó szálat elnyel, amíg a számláló le nem nullázódik, csak azt engedi tovább, amelyik a nullára csökkenést előidézte.

A fork-join páros a goto-hoz hasonló anomáliákhoz vezetett (áttekinthetetlen vezérlési szerkezet), így helyette egy strukturált utasítás bevezetését javasolták, a **parbegin-parend** párt. Ez a megoldás valóban áttekinthetőbbé és biztonságosabbá tette a nyelvi leírást, de nem írható le vele tetszőleges precedenciagráf, csak járulékos szinkronizációs műveletekkel.

A megvalósított első konkurens nyelvben, a CPascalban (és később több másikban is, például MODULA, ADA) az explicit folyamatdeklaráció (process declaration) használatos annak kifejezésére, hogy egy programegység folyamatként, azaz más hasonló programegységgel párhuzamosan hajtódhat végre. A folyamatdeklarációk igen hasonlóak az eljárásdeklarációkhoz.

2.2.10.2. Az együttműködés nyelvi modelljei

A párhuzamos végrehajtású programrészek az egygépes rendszerekben közös memórián működhettek együtt – megfelelő szinkronizációt feltételezve. A *szemaforra* alapozott szinkronizációt a párhuzamosságot megengedő magasszintű nyelvekben nem találták elég biztonságosnak, hiszen például egy kölcsönös kizárás megoldása esetén a programozó következmények nélkül elfelejtheti a használatát, vagy ami legalább olyan kellemetlen, elfelejtheti a felszabadítását. A szemafor és a védett objektum között csak a programozó(k) tudatában van kapcsolat, így az objektumot bármelyik folyamat nyugodtan használhatja a szemafor lefoglalása nélkül.

A kölcsönös kizárás megoldására az első javasolt magasszintű eszköz a **kritikus régió** volt. A javaslat szerint a változódeklarációkban jelölni kellett a védett közös változókat *(shared)*, az ilyen változókra pedig csak a változóhoz tartozó kritikus régióban hivatkozhat a program (ha *X shared*-ként deklarált: *region X do...<X használata>...od*). A megoldás lehetővé teszi a fordítási időben történő ellenőrzést az esetleges régión kívüli hivatkozások felderítésére.

A kritikus régióval meglehetősen nehézkesen kezelhetők azok a helyzetek, amelyekben egy feltételvizsgálattól teszik függővé a folyamat a szakaszba való belépést, de már magának a feltételvizsgálatnak az elvégzéséhez is a védett változót kell használni. Erre kínált megoldást a **feltételes kritikus régió.**

A következő jelentős lépcsőfok a *CPascal nyelvi modellje*, amelyik az absztrakt adatszerkezetek adatrejtési elveit *(encapsulation)* és a konkurens végrehajtás lehetőségét ötvözte. Bevezette a **monitor** rendszertípust. A monitor három fő összetevőből áll. Vannak *privát*, kívülről el nem érhető *adatai*, ezeken saját, *kívülről is hívható eljárásai* tudnak csak műveletet végezni, valamint van egy *inicializáló kódrésze*. A kívülről hívható eljárásokra garantált, hogy kölcsönös kizárással futnak le, és várakozási sor szerveződik rájuk. Az eljárásokban a feltételes kritikus szakasz funkciójának megvalósítására egy speciális változótípust *(queue)* vezettek be, amely *delay* és *continue* műveletekkel kezelhető. Egy *delay(x:queue)* művelet várakozásba helyezi a hívó folyamatot, és lehetővé teszi, hogy más lépjen be a kritikus szakaszba. A folyamat addig vár, amíg valaki végre nem hajt egy

continue(x:queue) műveletet. Megjegyzendő, hogy a *continue* hatása nem őrződik meg, ha nincs várakozó, a jelzés elszáll.

Két maradandó hatású nyelvi modell – nem konkrét nyelv – született a ’70-es években: az *együttműködő folyamatok* **CSP (Communicating Sequential Processes)** és az *elosztott folyamatok* **DP (Distributed Processes)** modell.

Mindkettőt az motiválta, hogy megjelentek az egymással adatátviteli vonalakon, vagy hálózaton összekapcsolt számítógépek, és a folyamatmodellt és a folyamatok együttműködésének modelljét lehetőleg úgy kellett kialakítani, hogy azonos processzoron futó, és különböző processzorokon futó folyamatokra is alkalmazható legyen.

A *CSP-modell* üzenetváltással együttműködő folyamatokat kezel, bevezeti az **őrzött parancs (guarded command),** az **alternatíva,** és az **ismétléses alternatíva** szerkezeteket, amelyekkel a több belépési ponttal rendelkező, konkurens hívásoknak kitett *szerver* típusú folyamatok működése is leírható, amelyek nem tudják előre, hogy a következő hívásuk honnan érkezik.

A *DP-modell* hasonló alapokról indul, a folyamatok közötti kommunikációt azonban az eljáráshívás és paramétercsere szemantikájára alapozza **(távoli eljáráshívás, remote procedure call)**.

A ’80-as évek elején érett szabvánnyá az **ADA** nyelv, amely szintetizálta a kor valamennyi elméleti eredményét. A párhuzamos programrészek létrehozására taszkok deklarálhatók, amelyek a *távoli eljáráshívás* elvén működhetnek együtt. A szerver típusú folyamatok a CSP-modellben javasolt alternatív szerkezetet *(select)*, és az őrzött parancsoknak megfelelő szerkezetet *(when F => accept <entry-call>)* használhatják a többfelől várható hívások fogadására.

A további nyelvek a párhuzamosság és az együttműködés alapsémái tekintetében már nem hoztak átütő újdonságot. Napjainkban a legelterjedtebben a C nyelvi könyvtárak formájában megvalósított *PVM (Parallel Virtual Machine)*, vagy *MPI (Message Passing Interface)*, valamint a *Java* nyelv modelljei használatosak a programszinten párhuzamos végrehajtás területén.

2.3. Tárak

Annak a virtuális gépnek, amelyet az operációs rendszer nyújt az alkalmazások és a kezelők számára, eddig elsősorban a processzorával, illetve processzoraival ismerkedtünk. Egy valós, vagy virtuális gép hasonlóan fontos alkotórészei a tárak.

2.3.1. Tárhierarchia

Egy számítógépes rendszerben a tárak hierarchikus rendbe szervezettek. A hierarchia legalsó szintjén a fizikai processzor regiszterei helyezkednek el, fölötte sorrendben az operatív tár (memória), a háttértárak vagy másodlagos tárolók, valamint a külső tárak vagy harmadlagos tárolók találhatók. Minél magasabb hierarchiaszinten van egy tár, általában annál nagyobb mennyiségű adat befogadására alkalmas, viszont –

gyakran a hosszabb keresési idő következtében – annál lassabb működésű, és annál nagyobb egységekben címezhető. Ugyancsak különböznek az egyes szintek a tárolt információ élettartama és tárolási biztonsága tekintetében is. A processzorregiszterekben tárolt adatok élettartama néhány utasítás, a memóriatartalomé jó esetben a program futási ideje. A programokat túlélő, maradandó adatokat fájlok tárolják, egy aktualitási időszakban a másodlagos tárakon, majd archiválva, harmadlagos tárolókon.

A több tárolási szint hatékony kezelése a rendszerek működtetésének egyik legfontosabb problémája. Azt az ellentmondást, hogy a műveletvégzésben résztvevő adatoknak a processzor regisztereiben kell elhelyezkedniük, míg a számítógépes rendszerben kezelt adat- és programmennyiség általában még a háttértárakra sem fér el egyidejűleg, csak az adatok *tárolószintek közötti folyamatos mozgatásával* lehet feloldani. A szintek közötti adatmozgatás hagyományos megoldása, hogy a processzor utasításonként esetleg többször is a memóriához fordul (egyrészt magáért az utasításért, másrészt adatokért), a programokat végrehajtás előtt a háttértárról a memóriába kell tölteni, a háttértáron tárolt adatokat csak fájlműveletekkel (a be-/kiviteli rendszeren keresztül) lehet elérni, a harmadlagos, külső tárakból pedig a személyzet közreműködésével kell kikeresni és a megfelelő perifériába helyezni a megfelelő adathordozókat.

Az egyes hierarchiaszintekhez más-más elérési, hivatkozási módok tartoznak. Az operatív tár címeire az utasítások meghatározott címzési módokkal és numerikus címértékekkel hivatkoznak. A háttértárakon tárolt fájlokra ezzel szemben nevekkel, amelyekhez csak futási időben – a fájlok megnyitásakor – kötődnek konkrét blokkcímek. A memória bájtonként, szavanként címezhetően érhető el, a fájlok ezzel szemben gyakran szekvenciális elérésűek. A tárolószintek közötti adatmozgatások történhetnek explicit módon, valamelyik alkalmazás megfelelő műveletének végrehajtásával. A rendszerekben azonban igen gyakran rejtett átjárások vannak a szintek között, amelyek célja a hatékonyság, vagy a kényelem fokozása.

A rejtett átjárások két módja fordul elő:

- az alacsonyabb szint elérési módjait terjesztik ki a magasabb szintű tárra (virtualizálás),
- a magasabb szintű tár elérési módját használva valójában egy alacsonyabb szintű tárat kezelnek (gyorsítótár, cache).

A két mód megvalósítása végül hasonló részproblémákra vezet, mindkettőre hasonló fogalmak (találat, betöltési, kiírási stratégiák stb.) alkalmazhatók.

A virtualizálás természetesen a magasabb szintű tár bevonása miatt lassabb működést eredményez, mint a valódi alacsonyabb szint elérése, ám igen kényelmes, hiszen az alacsony szint felhasználható címtartománya sokszorosára nőhet.

A gyorsítótár két szint közé épül be, vagy az alacsonyabb szintű tárolóban kerül kialakításra a magasabb szinten elhelyezkedő adatmennyiség egyes részeinek átmeneti tárolására. Az eredetileg magasabb szinten tárolt adatokra vonatkozó műveletek végrehajtásakor a művelet végrehajtója először megvizsgálja, hogy az adat nincs-

e a gyorsítótárban. Ha ott van (találat), akkor a művelet gyorsan végrehajtható. Ha nincs ott, akkor adatcserét kell végrehajtani, a magasabb szintről be kell hozni az adatot a gyorsítótárba.

Jellegzetes gyorsítótárak:

- a processzorokba épített hardver-gyorsítótárak *(utasítás- és adatcache)*, amelyek a memóriában tárolt adatok egy munkában lévő részét teszik gyorsan elérhetővé a processzor számára,
- a fájlok egy részének tárolására a memóriában kialakított átmeneti tárterületek *(buffer-cache)*.

A felhasználó által manuálisan kezelt gyorsítótárak:

- a memóriában kialakított, több fájl tárolására alkalmas virtuális diszkek (RAM-diszk, elektronikus diszk),
- a harmadlagos tárak fájlrendszereit tároló mágneslemez területek.

A gyorsítótárak és a magasabb szintek közötti adatcsere általában a felhasználó elől rejtetten, egyidejűleg nagyobb adatmennyiségeket mozgatva történik. A hatékonyságnövekedés oka a feldolgozás lokalitási tulajdonsága, azaz ha egy adatra (beleértve a program utasításait is) a feldolgozás során szükség van, akkor nagy valószínűséggel a környezetében tárolt adatokra is szükség lesz. A gyorsítótár megfelelő méretezésével és hatékony adatcsere-algoritmusokkal 80–99%-os találati arány is elérhető.

Az operációs rendszerek hatáskörébe a tárhierarchia szintjei közül a memória, valamint a háttértárakon és harmadlagos tárolók aktuális adathordozóján kialakított fájlrendszerek tartoznak, a felsorolt gyorsítótárak közül pedig a *virtuális memória* és a fájlrendszerek gyorsítótárai *(buffer-cache)*. Ezek közül a fájlrendszerek kezelése – tekintve, hogy perifériákon történő adattárolásról van szó – a be-/kiviteli rendszerre épül.

2.3.2. A logikai memória

Mint korábban bemutattuk, a folyamathoz tartozó virtuális gép a procesz-szoron kívül egy logikai memóriát tartalmaz. A logikai memóriát a folyamat egy RAM, illetve közösen használt memória esetén PRAM-modell szerint működő tárnak látja. Az egyetlen folytonos címtartomány helyett már a logikai szinten is érdemes három önálló címtartományt elkülöníteni, mivel ezek viselkedése és kezelése eltérő. A folyamathoz tartozó programkódot és esetleg konstansokat tároló *kódterületet*, az adatokat, változókat tároló *adatterületet*, valamint a *verem (stack) területét*. A kódterületet általában csak olvassák a folyamatok, ezért közös használata kevesebb problémát okoz. Mérete általában már végrehajtás előtt ismert. Az adatterület változókat tárol, a folyamatok írják is, de mérete ugyancsak ismert. A verem ezzel szemben amellett, hogy változókat tárol, dinamikusan növekedhet, és a mérete bizonyos esetekben – például iteráció adatfüggő mélységű rekurzív eljáráshívások esetén – nem határozható meg előzetesen. A folyamatoknak ezért általában fel kell készülniük a verem túlcsordulását jelző hibamegszakításra.

2.3.3. A háttértár-fájlok

A folyamatok a másodlagos tárolókon a futásukat túlélő adatok tárolására névvel azonosítható fájlokat hozhatnak létre. A fájl tartalmát a folyamatok állítják elő és használják fel, az operációs rendszer a tartalommal általában nem foglalkozik, a fájlt egyszerű bájtfolyamként kezeli.

A fájlok, mint tárolási egységek, egészükben is lehetnek műveletek tárgyai, illetve természetesen műveletek szükségesek a fájlban tárolt adatok kezelésére is. A másodlagos tárolókon elhelyezkedő igen nagyszámú fájl megtalálása, egyértelmű azonosítása rendezett nyilvántartási rendszert igényel, ami leggyakrabban egy **hierarchikus könyvtárszerkezettel** valósul meg. A hierarchikus könyvtárszerkezet azt jelenti, hogy a könyvtárak tartalmazhatnak fájlokat, valamint további könyvtárakat (alkönyvtárak), és ez valamennyi szinten lehetséges. A könyvtárhoz tartozó fájlokról egy **katalógusfájl (directory)** tartalmazza a szükséges adatokat. A tárolási hierarchia a legtöbb esetben egyetlen közös gyökérből indul, más esetekben a legmagasabb szinten több **kötet (volume)** helyezkedhet el, ami gyakran egyben egy fizikai egységet is jelöl. A tárolási hierarchiában a fájlt *elérési útvonala* és *neve* együttesen azonosítja, így csak az adott könyvtáron belül szükséges egyedi neveket használni.

2.3.3.1. Fájlmodellek

A fájlban tárolt adatok elérésére háromféle fájlmodell (elérési módszer, *access method*) használatos:

Soros elérésű *(sequential)* fájl

A fájl fogalma az operációs rendszerekben a mágnesszalag analógiájára alakult ki. A szalagon tárolt információt csak a tárolt byte-ok sorrendjében lehetett olvasni vagy írni. Sok információfeldolgozási feladathoz a soros hozzáférés elegendő. A soros elérés modelljéhez tartozik egy ún. fájlpointer, ami az aktuális elérési pozíciót tárolja.

Közvetlen elérésű *(direct)* fájl

A tárolt adatelemeket (például bájtokat) közvetlenül, sorszámuk alapján el lehet érni. Az átviteli eljárásoknak paraméterként meg kell adni az adatelem állományon belüli címét (sorszámát).

Indexelt, index-szekvenciális elérésű fájl *(index sequential access method*, *ISAM)*

Ez a modell a fájlban tárolt adatok szerkezetének ismeretét is feltételezi. Ha a fájl rekordokat tárol, akkor a rekordokat kijelölt mezőik (kulcs) tartalma alapján érhetjük el. A keresés megkönnyítésére egy segédfájl (indexfájl) szolgál, amelyik a fájlban előforduló kulcsértékek szerint rendezve tárol egy-egy mutatót a megfelelő rekordra. Ez a modell nem minden operációs rendszerben áll rendelkezésre.

2.3.3.2. Fájlműveletek

A műveletek egy része a fájl egészére vonatkozik, más része a fájl adatainak kezelését teszi lehetővé.

- Adatelérés, írás vagy olvasás

- közvetlen hozzáférésnél a művelethez meg kell adni az információ címét, pozícióját,
- soros hozzáférésnél az átvitel során az aktuális pozíciót a rendszer növeli és tárolja, eldöntendő, hogy megengedjük-e az átvitel során az állomány szimultán írását és olvasását is, és ha igen, akkor soros elérésnél közös vagy külön-külön fájlmutatót használunk-e.

- Hozzáírás (append), bővítés. Az állomány végéhez új információt írunk. Az állomány mérete növekszik.
- Pozicionálás. A soros hozzáférésnél használt pozíciót a soros átvitel mindig növeli, pozicionálásnál viszont azt az állomány tetszőleges pontjára állíthatjuk. Gyakori eset a pozíció visszaállítása az állomány elejére, a mágnesszalag analógiát sugalló visszacsévélés (rewind). A pozicionálás műveletével lehet a közvetlen hozzáférést megvalósítani.
- Megnyitás (open). Az átviteli műveletek hatékonyabbak, ha nem kell mindegyiknél az állományt a neve alapján a nyilvántartásokból megkeresni. Ezért a modell szintjén is megjelenik a megnyitás művelete, hiszen a további műveletek már más (nem név szerinti) hivatkozási módot igényelnek. A megnyitás műveletének szerepe:
    - az állomány megkeresése, fizikai elhelyezkedésének meghatározása, a további elérést segítő adatszerkezetek felépítése,

– hozzáférési jogosultságok ellenőrzése,

– az állományon elvégezhető műveletek (írás, olvasás, hozzáírás) megadása,

– osztott állománykezelés szabályozása,

– fájlmutató kezdeti beállítása.

Az állomány sikeres megnyitása után az operációs rendszer visszaad egy olyan értéket (mutató vagy logikai perifériaszám), amely a további műveletekben a név helyett használható az állomány azonosítására.

- *Lezárás (close)*. Jelzi a folyamat részéről az adatelérési műveletsorozat befejezését. A megnyitáskor elhelyezett zárak feloldódnak, a fájl esetleg a másodlagos tárra még ki nem írt (pufferelt) adatai kiíródnak.
- *Végrehajtás (execution)*. Az operációs rendszer létrehoz egy új folyamatot, a programfájlt betölti a folyamat tárterületére és elindítja.
- *Létrehozás (create)*. Az állomány létrehozásakor egy új katalógusbejegyzés is létrejön, a rendszer szükség esetén az állományhoz szabad blokkokat foglal le.
- *Törlés (delete)*. A törlés megszünteti a katalógusbejegyzést, felszabadítja az állományt tároló területet a másodlagos táron.

2.3.4. Tárak tulajdonságai

Az előzőekben a tárhierarchia egyes szintjein elhelyezkedő adatok címzése és elérése szerint tettünk különbséget a memória és a háttértár között. Emellett a táraknak más fontos tulajdonságai is lényegesek, és hozzátartoznak a virtuális gép jellemzéséhez.

2.3.4.1. Működési sebesség

A tárak működési sebessége az **adatelérési idővel (access time)** jellemezhető, ami egy olvasási, illetve írási művelet végrehajtási ideje. A fizikai memória esetén az elérési idő címfüggetlennek vehető. A modellben is ezzel a feltételezéssel élhetünk, bár a rejtett átjárások (virtuális tárkezelés) miatt egyes hivatkozások végrehajtási idői között jelentős különbségek lehetnek, és az átlagos érték kedvezőtlenebb a fizikai tárra jellemző időknél. A fájlok fizikai tárolói jelentős keresési idővel rendelkeznek (lemezek fejmozgása, elfordulása). A gyorsítótáras kezelés (buffer cache) miatt a modell kedvezőbb átlagos végrehajtási időket vehet figyelembe.

2.3.4.2. Címezhető adategység

A fizikai eszközök eltérő felépítése, működése ellenére a modell bájt felbontásban teszi elérhetővé mind a memóriában, mind a fájlokban tárolt adatokat. Fájlok esetén ez a bemutatott soros, illetve direkt elérés valamelyike lehet.

2.3.4.3. Tárolási biztonság

Az egyes tárak robusztussága, működésbiztonsága igen eltérő lehet. A ma általánosan használt memóriák (elektronikus tárak) legkellemetlenebb tulajdonsága, hogy tápellátás kimaradás esetén felejtenek. A mágneses elven működő háttértárak ilyen szempontból kedvezőbbek, „nem felejtők”, ugyanakkor előfordulnak lemezsérülések, vagy tranziens felírási, leolvasási hibák.

A rendszertervező, alkalmazásfejlesztő általában három tárkategóriában gondolkozik: *felejtő, nemfelejtő* és *stabil* tárakban. A felejtő tár kisebb rendszerzavarok esetén is elveszíti tartalmát. A nemfelejtő kisebb sokkokat túlél (például tápkimaradás), a stabil tár pedig elvileg mindent túlél, gyakorlatilag igen nagy valószínűséggel lehet számítani az adatok megőrzésére az újraindításig. A közelítőleg stabil tárak megfelelő redundanciával kezelt háttértárakon alakíthatók ki.

2.4. Készülékek és külső kapcsolatok

A virtuális gép fontos objektumai a folyamatok és tárak mellett a készülékek, illetve az egyéb külső kapcsolatok. Valamennyi olyan eszközt, amelyik a fizikai architektúra szerint perifériának minősül, igyekeznek egységesen kezelhetővé tenni, közös modellt találni számukra. Ez nem könnyű, mert ezek az eszközök igen sokfélék. Két fő irány figyelhető meg, amelyek – ha kicsit mélyebbre nézünk – nem is különböznek olyan lényegesen egymástól. A két irány a fájlok és a többi készülék viszonyának meghatározásában különbözik. Az egyik irány valamennyi külső eszközt a *fájl* absztrakció alá sorol be, és speciális fájloknak tekinti például a nyomtatót, a terminált és a többi perifériát, kezelésükre – esetleg speciálisan paraméterezhető – fájlműveleteket nyújt. A másik irány

központi fogalma a *logikai periféria*, amelyik egy absztrakt be-/kiviteli eszköz, és a rendszer működése során a logikai perifériához akár különböző fizikai eszközök, és rendszerint fájlok is rendelhetők. Bármelyik irányból közelítünk is, a meglehetősen vékony fedőrétegek alatt elég hamar előkerülnek az eszközök különbözőségéből származó problémák, ami nem jelenti azt, hogy a készülékeket ne lehetne osztályokba sorolni. Ebben a fejezetben egyrészt egy rendszerezésre teszünk kísérletet, majd a jellegzetes készülékek modelljeit tárgyaljuk, amelyek a felhasználói, illetve programozói felületen megjelennek.

2.4.1. A külső eszközök fő típusai

A számítógépek körül egyre többféle olyan készüléket találunk, amelyekkel közvetlen kapcsolatban van. Ezek a készülékek feladataikat, szerkezetüket, működési elvüket tekintve rendkívül sokfélék lehetnek.

A perifériákat egyrészt *jellegzetes feladatköreik* szerint csoportosíthatjuk, amelyek a következők lehetnek:

- ember–gép kapcsolat,
- gép–gép kapcsolat,
- adattárolás.

Jellemző tendencia, hogy egyre újabb, a korábbiakra alig emlékeztető készülékek jelennek meg, amelyeknek együtt kell működniük a számítógéppel. Ugyanakkor – szerencsére – a készülékek csatlakoztatásának szabványosodása is megfigyelhető. Ugyancsak jellemző tendencia, hogy a perifériák egyre intelligensebb saját vezérlőegységgel rendelkeznek, így a kapcsolatokban is egyre magasabb szintű, a részleteket elfedő modellt láttatnak magukból. Mindennek eredményeként a fenti, funkcionális különbség a kapcsolat alacsony szintű rétegeiben nem ismerhető fel, ott minden eszköz a gép–gép kapcsolat sajátosságai és szabványai szerint kezelhető.

A következő csoportosítási szempont a perifériák *működési sebessége* lehet. Többségük mechanikus alkatrészeket is tartalmaz, ezért a perifériák működési sebessége (például feldolgozott bájt/s mértékegységben) általában lényegesen alatta marad a processzorok sebességének. Különösen azok az eszközök lassúak, amelyek működési sebességét emberi reakcióidők határozzák meg. Más esetekben azonban – főként célrendszerek, speciális alkalmazások gép–gép kapcsolataiban – a külső eszközként kezelt műszerek, érzékelők, egyéb készülékek olyan gyorsak, hogy a számítógéphez való illesztésük speciális megoldásokat igényel.

A készülékek jellegzetes sebességkategóriái (a felsorolás sorrendjében egyre gyorsabbak):

- emberi beavatkozást igénylő készülékek,
- mechanikus, elektromechanikus készülékek,
- elektromos, elektronikus készülékek,
- optikai készülékek,
- részecskefizikai készülékek.

Ismét más csoportosítási szempont a készülékek *vezérelhetősége*. Vannak eszközök, amelyek (természetesen adott sebességi korlátokon belül) a számítógép által meghatározott ütemben képesek adatok küldésére vagy fogadására. Más eszközöknél ellenben éppen az jelenthet gondot, hogy egy hosz-szabb adatfolyamot csak saját ritmusukban képesek kezelni. Ilyenek például a hagyományos rendszerekben a diszkek a lemez folyamatos forgása miatt, vagy a mai rendszerek multimédiás, audio/video jelfolyamait közvetítő eszközök, a külső tv-adás számítógépes fogadásának, vagy számítógépes mozgókép tv-adásba történő közvetítésének eszközei. Ha a számítógép oldalán nem tudunk igazodni a készülék által diktált ütemhez (szinkronhoz), adatvesztés lép fel, és az átvitel egyáltalán nem, vagy csak jelentős időveszteséggel ismételhető meg.

Az operációs rendszer készítőjének szempontjából újabb rendező elv lehet az egyes készülékek *illesztési módja* a számítógéphez. A számítógép hardver általában különböző készülékvezérlők alkalmazásával oldja meg a készülékek és a tár közötti adatcserét. Egyes vezérlők csak egy-egy eszközt felügyelnek, mások egy külön sínrendszeren több eszközt kezelnek. Egyes vezérlők önálló processzorral és mikrokóddal, jelentős memóriával rendelkezhetnek, mások csupán néhány regisztert (adat ki-/bemenet, parancs, státus) és sínillesztő áramköröket tartalmaznak *(portok)*. Egyes eszközök csak B/K-címeket használnak, mások memóriába ágyazott címeket, vagy azokat is. Egyes eszközök állapotjelzőit programozottan kell lekérdezni *(polling)*, mások megszakításkéréssel jeleznek egy-egy eseményt. Egyes eszközök bájtonként vagy szavanként igényelnek processzorműveletet, mások blokkonként cserélnek adatot a memóriával (DMA).

Ez az utóbbi szempont nemcsak az illesztés, hanem a magasabb szintű kezelés során is előtérbe kerül. A készülék és a számítógép közötti *adatfolyam jellege* szerint ugyanis a készülékek *karakteres* vagy *blokkos* átvitellel működtethetők.

A bemutatott csoportosítás is érzékelteti, hogy az operációs rendszer által kezelt készülékek rendkívül sokfélék, ráadásul további, nehezen tipizálható tulajdonságokkal is rendelkeznek. Hogyan lehet ezeket a készülékeket a kezelői és programozói felületen egységesen megjeleníteni?

Az egységes felület kialakításának elve, hogy találjuk meg a lényeges közös vonásokat, és rejtsük el a részleteket, ruházzuk a megfelelő felelősségeket az architektúra alacsonyabb szintjeire. A készülékek esetén elég sok elrejteni való akad, és elég kevés a tipizálható, általánosítható funkció. A készülékkezelő *(device driver)* programokra meglehetősen sok feladat marad.

2.4.2. Készülékmodell az alkalmazói felületen

Az operációs rendszer programjai számára a legfontosabb absztrakció, hogy a készülékre adatokat lehet küldeni, illetve adatokat lehet róla fogadni. Ez megfelel a fájlokra kiadható írás, olvasás műveleteknek, ezért sok rendszer a *fájl fogalmát terjeszti ki* a készülékekre. A készülékek ilyen esetben a fájlokhoz hasonlóan nevükkel azonosíthatók, és beilleszkednek a könyvtári nyilvántartás névhierarchiájába. A fájlok használata előtt meg kell nyitni azokat, ami a készülékek esetén is értelmezhető, hiszen vannak eszközök, amelyek például előzetes motorindítást, kizárólagos használat lefoglalását, vagy kapcsolatfelépítést igényelnek, mielőtt az érdemi

adatátvitelre sor kerülhetne. Ugyancsak értelmezhető a készülékekre a használati jog ellenőrzése. Mint a fájlok megnyitása, a készülékek előkészítése is gyakran rejtetten történik a folyamatok indulásakor, vagy az első hivatkozáskor.

A fájlrendszerre épülve, vagy amellett a rendszerek lehetővé teszik *logikai perifériák* használatát, ami a készülékfüggetlen programozást szolgálja. Egy logikai periféria a rendszerben névvel vagy sorszámmal azonosítható, és ez a név, vagy sorszám használható a be-/kiviteli műveletekben írás esetén a cél, vagy olvasás esetén a forrás kijelölésére. A névhez a kezelő rendelhet konkrét fizikai eszközt, vagy a folyamat az előkészítési szakaszában végrehajtott megnyitásokkal maga végzi el a hozzárendelést (ez a szakasz természetesen nem lesz készülékfüggetlen, de a kód döntő része az marad). A logikai perifériákhoz általában nemcsak készülékek, hanem fájlok is hozzárendelhetők. A logikai perifériákra egyszerű be-/kiviteli műveletek adhatók ki. Lássuk ennek egy modelljét részletesebben!

2.4.2.1. Egyszerű be-/kivitel

A kernel alkalmazói felületén a B/K-műveletek egy adatblokk mozgatására vonatkoznak a memória és valamely logikai periféria között. Ennek a műveletnek a bonyolítására egy indító *(START)*, egy várakozó/tesztelő *(WAIT)* és rendellenességek esetére egy leállító *(STOP)* művelet szükséges, valamint az átvitel paraméterezéséhez egy egyezményes adatstruktúra, a B/K-leíró *(IOCB: Input Output Control Block)*.

Az adatmozgatás paraméterei: *forrás, cél, mennyiség.* A forrás és a cél közül az egyik egy memóriacím, a másik egy logikai periféria név (vagy szám). A mennyiség az átviendő bájtok száma. A műveletek argumentumában a logikai periféria neve és egy mutató szerepel az IOCB-re. Az IOCB tartalma: *parancs*, ami lehet írás, olvasás stb.; *állapotjelző*, amelyik az átvitel sikerét vagy hibáját jelzi vissza, *memóriacím; mennyiség; egyéb paraméterek*, amelyek a periféria által értelmezhetők lehetnek (például diszkek esetén blokkcím).

A *START* a logikai periférianév alapján fizikai készüléket választ ki, a fizikai periféria várakozási sorába fűzi az átviteli paramétereket tartalmazó IOCB-t. Ha a periféria nincs működésben, meghívja a készülékkezelő program szabványos *Start_Device* műveletét (az egységes készülékcsatlakoztatási felület előírja, hogy ilyen műveletnek kell lennie). Ezután mindkét esetben visszatér, azaz *aszinkron* hívásként nem várja meg a B/K-művelet végét.

A *WAIT* művelet ellenőrzi az IOCB állapotjelzőjét. Ha a B/K-művelet még nem fejeződött be, a hívó folyamatleíróját ráfűzi a megadott IOCB-re és a folyamatot várakozó állapotba helyezi, majd CPU ütemezést indít. Ha a B/K- művelet befejeződött, visszatér a hívóhoz. Egyes rendszerekben a művelethez időkorlát rendelhető, amely 0 és végtelen értékekre is beállítható. A 0 érték lehetővé teszi az elindított művelet befejeződésének lekérdezéses észlelését (polling), a végtelen időkorlát pedig a blokkolt műveletek megvalósítását.

A *STOP* művelet használatával programhibák, leállások esetén megállítható az elindított, de még be nem fejeződött B/K-művelet. Ha a készülék még nem kezdte meg a végrehajtást (nem az első az IOCB a sorban),

egyszerűen kifűzi az IOCB-t, ha már megkezdte, a szabványos készülékcsatlakoztatási interfészen keresztül meghívja a kezelőprogram *Stop_Device* eljárását.

Az alkalmazások általában nem közvetlenül ezt a felületet használják, hanem a nyelv, illetőleg a rendszer könyvtári csomagjainak eljárásait. Az egyszerű B/K-műveletek lehetővé teszik, hogy a magasabb szintű, nyelvi és könyvtári műveleteket akár szinkron (várakozó blokkolt), akár aszinkron (továbbfutó, nem blokkolt) változatban kialakíthassák. A blokkolt megoldás a *START* után azonnal *WAIT* hívást tartalmaz, így a folyamat általában várakozó állapotba kerül. A nem blokkolt megoldás csak elindítja az átvitelt, majd a folyamat tovább futhat, és alkalmas ponton lekérdezheti az átvitel eredményét. A magasszintű műveletek szívesebben alkalmaznak blokkolt megoldást, mert a vezérlési szál ilyenkor nem ágazik el a B/K-indítás következtében, és a programkód könnyebben érthető és követhető. Ha nem blokkolt megoldás szükséges, akkor is szívesebben hoznak létre inkább új szálat a folyamaton belül, és azon a B/K-műveletet akkor is blokkoltan hajtják végre.

2.4.2.2. Fájlműveletek

A legtöbb eszközre a szekvenciális fájl modellje jól illeszthető, hiszen az adatáramlás bájtfolyam jellegű.

A diszkes fájlok tipikus műveleteit az előző fejezetben áttekintettük. Megnyitáskor a fájlra névvel kell hivatkozni, a további read, write, seek műveletekben azonban már a megnyitáskor kapott számmal.

A virtuális tárat használó rendszerek gyakran lehetővé teszik a *fájlok memóriába ágyazott kezelését*. Egy *leképező* művelet helyet foglal a fizikai tárban a fájl memóriában tartandó részeinek és visszaadja azt a virtuális címtartományt, amelyben a fájl látszik. Ezt követően a fájl adataira memóriareferenciákkal hivatkozhatunk.

A további fizikai eszközök (például nyomtató, billentyűzet, modem, audio és video berendezések) read, write műveleteinek egy része ugyancsak blokkonkénti, más része karakterenkénti B/K-rendszerhívást okoz. Egyes eszközökre csak olvasás (billentyűzet, CD), vagy csak írás (nyomtató) értelmezhető, ettől eltekintve a fájlként történő kezelés a felhasználói felületen könnyen értelmezhető.

2.4.2.3. Grafikus eszközök kezelése

A grafikus eszközöket az alkalmazások általában egy grafikus könyvtár eljárásaival kezelik. Ezek az eljárások raszter- vagy vektorgrafikai modellt valósítanak meg, és alakzatrajzoló, törlő, mozgató, továbbá állapotbeállító (háttérszín, rajzolószín, vonaltípus, kitöltés stb.) műveleteket tesznek lehetővé. A könyvtári eljárások a monitorok grafikus memóriáját, vagy annak egy részét általában memóriába ágyazottan látják. Bizonyos eszközökben a könyvtári eljárások programja és a grafikus memória külön célprocesszorral együtt a grafikus vezérlő kártyáján helyezkedik el. Ezt az eszközt az operációs rendszer csak néhány B/K-címen, vagy memóriába ágyazottan kezelhető regiszterként látja. Ennek ellenére a felhasználó számára logikailag ugyanaz a felület jelenik meg mint a hagyományos megoldásoknál, a műveletek végrehajtása azonban lényegesen gyorsabb lehet.

2.4.2.4. Terminálkezelés

A terminál egy karakteres be-/kiviteli egység, amelyik képernyőt és billentyűzetet tartalmaz, és általában a számítógéptől távolabb helyezkedik el, ezért adatátviteli vonalon vagy hálózaton csatlakozik a számítógéphez. Az alkalmazások számára a terminál sororientált vagy képernyőorientált kezelésére állnak rendelkezésre műveletek. Az adatátvitel jellegzetesen karakterenkénti és nem transzparens, azaz egy adott kódrendszert (benne vezérlőkaraktereket is) használ. A terminálok igen sokféle szabvány szerint működhetnek, különösen a billentyűzet-kiosztás és a vezérlőkarakterek elhelyezése a kódtáblában lehet sokféle. Az operációs rendszerek ma már valamennyi, a fontosabb szabványoknak megfelelő kezelést választhatóan tartalmazzák, ám ezek közül választani sem egyszerű felhasználói feladat.

2.4.2.5. Hálózati kapcsolatok kezelése

A hálózati kapcsolatok kezelésére jellegzetes megoldás a logikai csatlakozók (socket) fogalmának bevezetése (lásd UNIX, Windows NT). Az alkalmazások létrehozhatnak csatlakozókat, segítségükkel becsatlakozhatnak távoli címekre, figyelhetik, hogy más rácsatlakozott-e az ő csatlakozójukra. A kapcsolat létrejötte után a csatlakozó egy indirekt kommunikációs objektumként működik. A szerver oldal – ahova több csatlakozón érkezhetnek hívások – munkájának támogatására nyújtják a rendszerek a *select* műveletet, amely viszszaadja azon csatlakozók listáját, amelyiken van várakozó üzenet.

A csatlakozók mellett igen változatos további kommunikációs modellek is megjelennek a rendszerekben (postaláda, üzenetsor, pipe, stream stb.).

A készülékekbe épített vezérlők képességeinek fejlődésével számos készülék a folyamatok közötti kapcsolatokhoz egyre inkább hasonlatos módon kezelhető.

2.4.3. Készülékmodell a kezelői felületen

Az *egyszerű felhasználó* kezelői felületén a készülékeket is az alkalmazások közvetítik.

Az *alkalmazásfejlesztő* számára az alkalmazói felület modellvilága bír elsődleges jelentőséggel, azon túl pedig az eszközök konfigurálása, a logikai, fizikai készülékek összerendelésének lehetőségei.

Újdonságot az eddigiekhez képest leginkább a *rendszermenedzser* számára nyújtandó készülékmodellek jelentenek. Neki kell ugyanis végrehajtani új készülékek csatlakoztatását a rendszerhez, továbbá elvégezni azok megfelelő beállítását. Ezen a szinten a rendszerek némelyike valóban egy széles, modellezett eszközválasztékot kínál, ahol tipikus funkciójú készülékek legfontosabb paraméterei beállíthatók, és a konkrét eszközpéldányok hozzárendelése is megtörténhet a megfelelő modellhez. Például egy modemre beállítható, hogy milyen adatkapcsolati protokollt használjon, használjon-e paritásbitet, meddig várjon a tárcsahangra stb., és az is, hogy mindez melyik fizikai csatlakozóra kapcsolt készülékre vonatkozik.

2.5. Védelem és biztonság

Védelemnek nevezzük eljárásoknak és módszereknek azon rendszerét, mely lehetőséget teremt a számítógép erőforrásainak programok, folyamatok, illetve felhasználók által történő elérésének szabályozására. A védelem fogalmát fontos megkülönböztetnünk a rendszer *biztonságának* (vagy biztonságosságának) fogalmától. A rendszer biztonsága annak a mértéke, hogy mennyire lehetünk bizonyosak a számítógépes rendszer, illetve a rendszerben tárolt adatok sérthetetlenségében. Míg a védelem szigorúan a rendszer belső problémája, kizárólagosan csak a rendszeren belüli objektumokkal foglakozik, addig a rendszer biztonságának biztosításakor figyelembe kell venni a rendszer működési környezetét, amelyből a rendszerünket potenciális támadások érhetik. Természetesen a két témakör szervesen összefügg, hiszen minden biztonsági rendszer feltételez egy megfelelően működő védelmi mechanizmust, amire építhet.

2.5.1. Védelem

A védelemről beszélve célszerű szétválasztani a védelem *módszerét* és a védelem *szabályrendszerét*. A védelem szabályrendszere meghatározza, hogy *mit* kell tenni a rendszer zökkenőmentes biztonságos használatához, míg a védelem módszere (vagy mechanizmusa) lehetőséget teremt a szabályozás megvalósítására, vagyis a rendszerobjektumok kezelésének *módját* határozza meg.

2.5.1.1. Védelmi tartományok

Védelmi szempontból a számítógépes rendszereket tekinthetjük úgy, mint objektumok, illetve az objektumokat használó folyamatok halmazát. Az objektumok ebben a modellben lehetnek mind szoftver komponensek (például fájlok, programok, szemaforok stb.), mind pedig hardver komponensek (például CPU, memória, nyomtató stb.). Az objektumok egyedi névvel érhetők el a rendszerben, és mindegyiken az objektumra jellemző műveletek végezhetők. Például egy fájlt a fájl nevével és az elérési útjával azonosítunk. A fájlon végezhető tipikus műveletek, például olvasás, írás, végrehajtás. A rendszermodellt általánosan tekinthetjük úgy, hogy az objektumok absztrakt adattípusok, amin végezhető műveletek függenek az adott objektum típusától.

A folyamatok működése a fenti modellben gondolkodva nem más, mint a fent definiált absztrakt adattípusok példányain végzett műveletek sorozata. A rendszert biztonságossá tehetjük, ha az objektumokon végzett műveletek végrehajtását jogosítványokhoz kötjük. Természetesen egy folyamat végrehajtásához szükségünk van a folyamat által használni kívánt objektum elérését lehetővé tevő jogosítványra, ami az objektumon végzendő műveletet engedélyezi. A rendszer abszolút biztonságosan működik, ha minden pillanatban minden folyamat pontosan csak azokkal a jogosítványokkal rendelkezik, ami feltétlenül szükséges az aktuálisan végzett művelet végrehajtásához, hiszen ha más objektumot próbálna elérni a folyamat, vagy az adott objektumot nem megengedett módon használná, a számítógép védelmi rendszere azonnal jelezne. (A szabályozásnak ezt a formáját ún. *need-to-know* szabályozásnak – szükséges, hogy ismerje, elérje szabályozásnak – nevezik.) Ez az ideális szabályozás a valóságban nem valósítható meg elsősorban a szükséges tetemes rendszeradminisztráció miatt.

Az objektumok elérésének szabályozására az ún. *védelmi tartományok* szolgálnak. A védelmi tartomány nem más, mint jogosítványok gyűjteménye objektumokon végezhető műveletek végrehajtására. Egy védelmi tartomány tartalmazhat jogosítványt egyazon objektum többféle művelettel történő elérésére, illetve egy objektum-művelet páros akár több különböző tartományban is szerepelhet. A védelmi tartományokat a legegyszerűbben táblázatos formában, az ún. elérési mátrix segítségével ábrázolhatjuk. A 2.17. ábra egy statikus elérési mátrixot mutat. A táblázatból kiolvashatjuk például, hogy csak a B tartományban tartózkodó folyamat jogosult nyomtatni a nyomtatón, vagy azt, hogy az A tartományban tartózkodó folyamat két fájlt olvashat: az *adat.txt*-t és a *help.bat*-ot.

Egy rendszerben használhatunk

- *dinamikus* és
- *statikus*

		Objektumok			
		adat.txt	doc.doc	help.bat	nyomtató
Tartományok	A	olvasás		olvasás	
	B				nyomtatás
	C		olvasás	végrehajtás	
	D	olvasás, írás		olvasás, írás	

2.18.ábra. táblázat - Elérési mátrix dinamikus védelmi tartományokkal

védelmi tartományokat attól függően, hogy egy folyamathoz tartozó védelmi tartomány változik-e a folyamat végrehajtása során. Ha nem változhat, statikus, míg az ellenkező esetben dinamikus védelmi tartományról beszélünk.

Dinamikus védelmi tartományok használatakor szabályozni kell a védelmi tartomány váltásának illetve megváltoztatásának módját. Több módszer létezik ennek megvalósítására.

- *Védelmi tartomány váltás (switch).* A legegyszerűbb módszer, hogy definiálják, mely védelmi tartományból mely védelmi tartományba léphet át egy folyamat. Ennek ábrázolása lehetséges az előbb megismert elérési mátrixban is. A 2.18. ábra első sorában például láthatjuk, hogy az A jelű tartományból csak a B jelű tartományba léphetünk át, a B jelű tartományból a C és a D jelű tartományokba léphet a folyamat, míg a C jelű tartományból nem lehetséges a védelmi tartomány megváltoztatása.
- *Elérési jogosítványok másolása (copy).* Lehetséges, hogy a rendszer engedélyezi egy bizonyos objektumon történő művelet végrehajtására vonatkozó jogosítvány másolását egy adott védelmi tartományban. Ez azt jelenti, hogy az adott védelmi tartományban futó folyamat jogosult az általa birtokolt, egy adott művelet elvégzésére szóló jogosítványt odaadni más védelmi tartományoknak.

- *Objektumok tulajdonlása (owner).* Az előző másolási jogot általánosíthatjuk. Kijelölhetünk egyes védelmi tartományokat, melyek ún. tulajdonosi joggal rendelkeznek egy adott objektum felett, ami azt jelenti, hogy az általuk birtokolt objektumon végezhető bármelyik műveletre adhatnak jogosítványt egy másik tartománynak.

	Objektumok				Tartományok			
	adat.txt	doc.doc	help.bat	printer	A	B	C	D
A	olvasás		olvasás			váltás		
B				nyomtatás			váltás	váltás
C		olvasás	végre-hajtás					
D	olvasás írás		olvasás írás		váltás			

4.6.ábra. táblázat - A centralizált rendszer rétegstruktúrája

2.5.1.2. Elérési mátrix ábrázolása és kezelése

Egy valós rendszerben igen sok objektum és nagyon sok védelmi tartomány létezhet, azonban egy-egy tartomány általában csak néhány objektum elérésére tartalmaz jogosítványokat. Az elérési mátrix ennek megfelelően egy igen nagy, de ritka mátrix, melynek optimális tárolására és kezelésére számos különböző módszert használnak.

Globális tábla

A globális tábla a legegyszerűbb tárolási mód. Egyszerűen a rendszer tárolja a <tartomány, objektum, műveletvégzési jog> hármasokat, vagyis egyszerűen egy listába gyűjti az elérési mátrix kitöltött elemeit. (A 2.17. ábra első sora alapján például a következő két elemet tárolná a rendszer: <A, adat.txt, olvasás>, <A, help.bat, olvasás>.) A módszer hibája, hogy az adódó lista igen hosszú egy valós rendszerben, ami hosszadalmassá teszi a táblázaton végzett műveleteket (keresés, beszúrás, változtatás). A műveletek elvégzését gyakran a háttértár elérése is meghosszabbítja, hiszen nagy táblázat nem fér be teljesen a memóriába.

Objektumelérési lista

Ennél a módszernél az elérési mátrixot lényegében az oszlopai mentén feldaraboljuk, és az oszlopok tartalmát tároljuk. Van egy listánk a rendszerben létező összes objektumokról, és minden objektumhoz tároljuk a hozzá tartozó <tartomány, műveletvégzési jog> párokat.

Tartományok jogosítványainak listája

Ennél a módszernél az elérési mátrixot a sorai mentén daraboljuk fel, és a sorok tartalmát tároljuk. A rendszerben létezik egy lista a védelmi tartományokról, és minden tartományhoz tároljuk a hozzá tartozó <objektum, műveletvégzési jog> párokat.

Zár–kulcs (lock–key) módszer

A zár–kulcs módszer átmenet az előző két módszer között. Minden objektumhoz tartoznak egyéni bitminták. Ezek töltik be a zár szerepét. Hasonlóan, minden védelmi tartomány rendelkezik egy vagy néhány bitmintával. Ezek töltik be a kulcs szerepét. Ha egy tartomány bitmintája egyezik az objektum bitmintájával, vagyis a tartomány egyik kulcsa illeszkedik az objektum zárjába, azt jelenti, hogy a tartományt birtokló folyamat elvégezheti a kulcshoz tartozó műveletet.

Mint említettük, a globális tábla módszer nehézkes, így a gyakorlatban ritkán használják. A objektum elérési lista használata gyorsítja a jogosultság ellenőrzését és annak elérését az objektum alapján. A tartományok jogosítványainak tárolása viszont a tartományok szerinti elérést támogatja, ami hasznos lehet egy folyamat jogainak meghatározásakor. Látható, hogy más-más szituációkban használható ki a két módszer előnyös tulajdonsága, emiatt ez utóbbi módszereket gyakran kombinálják.

Vegyünk egy példát. Tételezzük fel, hogy a védelmi tartományok egyszerűen folyamatokhoz kötődnek, minden folyamatnak van egy védelmi tartománya. Ha egy folyamat először kezdeményezi egy objektum elérését, ellenőrzi a rendszer az objektum elérési listájában, hogy a folyamat végezheti-e az adott műveletet. Ezek után a rendszer ad egy elérési jogosítványt a folyamatnak, aminek segítségével az objektumon történő további műveletvégzéskor már igazolni tudja a jogosultságát. Ezt a módszert használja például a UNIX-rendszer fájlok elérésekor.

A fájlok használatát minden esetben a fájl megnyitásának kell megelőznie, mikor is a folyamat köteles megadni a fájl adatain kívül a fájlon végzendő művelet típusát is. A megnyitás során az operációs rendszer ellenőrzi a folyamat fájlra vonatkozó jogosultságait. Ha a folyamat számára engedélyezett műveletet kíván végezni, az operációs rendszertől kap egy fájlleírót, melynek használatával további ellenőrzés nélkül érheti el a kívánt fájlt. Természetesen a fájlleíró csak a kért művelettípus végrehajtására alkalmas, a rendszer minden alkalommal ellenőrzi ezt.

Az elérési mátrix ábrázolási módszerek közül a zár–kulcs mechanizmus a leghatékonyabb módszer. Ennek használata rugalmas, a kulcsok egyszerűen adhatók át a tartományok között, a zár egyszerű megváltoztatásával megvonhatók jogosultságok, illetve a jogosultság ellenőrzése is gyors, hiszen nem szükséges nagy tömegű adatot mozgatni. Ez a módszer előnyei miatt gyorsan terjed a modern rendszerekben.

2.5.2. Biztonság

Egy számítógépes rendszer biztonsága, mint korábban már meghatároztuk, annak a bizonyosságnak mértéke, hogy a rendszer, illetve a rendszerben tárolt információ nem sérülhet meg vagy illetéktelenül nem kerülhet ki a rendszerből. A védelemmel ellentétben, aminek biztosítása szigorúan a rendszer belső problémája, a rendszer biztonságának meghatározásakor szükség van a rendszer környezetének a vizsgálatára is. Míg a védelem esetén programhibák illetve programok nem kívánt mellékhatásai, vagyis véletlen események ellen kell a folyamatokat

megvédeni, addig a biztonság biztosításakor feltételezni kell szándékos és rosszindulatú támadásokat is a rendszer ellen.

A számítógépes rendszerek biztonságáról beszélve nemcsak technikai, hanem rendszerszervezési, rendszeradminisztrációs kérdésekkel is foglalkoznunk szükséges. Technikailag tökéletes biztonságú rendszerben tárolt adatok is könnyűszerrel elérhetővé válnak egy támadó számára, ha nem biztosítunk megfelelő fizikai védelmet a rendszernek. Ha például a rendszer adattárolói a megfelelő védelem hiányában egyszerűen kivehetők a rendszerből és egy másik, védelem nélküli helyen lemásolhatók, a rendszer biztonsága elégtelen, holott az technikailag tökéletes.

A rendszer biztonságát ért sérülések oka lehet szándékos és véletlen. A szándékos támadásoknak három fajtáját különböztetjük meg:

- adatok illetéktelen olvasása,
- adatok illetéktelen módosítása,
- adatok tönkretétele.

A biztonsági rendszer feladata, hogy a szándékos támadások illetve véletlen sérülések ellen védelmet nyújtson. Mivel a gyakorlatban abszolút biztonságos rendszert igen nehéz létrehozni, ezért a gyakorlatban alkalmazott védelmi rendszerek célkitűzése az, hogy a támadó dolgát annyira nehezítse meg, hogy a támadás „költsége” nagyobb legyen, mint a sikeres támadás eredményeként remélt haszon.

2.5.2.1. A felhasználók azonosítása

A számítógépes rendszerhez történő jogosulatlan hozzáférés megakadályozásának első lépése a felhasználók azonosítása. A felhasználók azonosítására több módszer létezik:

- felhasználó azonosítása személyes tulajdonságai, például ujjlenyomata, retinamintázata, aláírása, kezének erezete stb. alapján
- felhasználó azonosítása annak birtokában lévő tárgyak, például kulcs, azonosító kártya stb. alapján
- felhasználó azonosítása csak általa ismert információ, például név, jelszó, esetleg algoritmus alapján.

A személyes tulajdonság és a tárgyak alapján történő azonosítás esetén a számítógépes rendszer valamilyen speciális periféria (aláírás-scanner, ujjlenyomat digitalizáló, kártyaolvasó stb.) segítségével végzi a felhasználó azonosítását.

A számítógépes rendszerekben a jelszóval történő azonosítás a legelterjedtebb elsősorban azért, mert ez a módszer valósítható meg a legegyszerűbben. A jelszóval történő azonosítás tipikusan két problémát vet fel: ne lehessen a jelszót egyszerűen kitalálni, illetve annak megadásakor azt „kihallgatni”. Gyakori, hogy a

felhasználók mások által egyszerűen kitalálható jelszót használnak: például saját login nevüket, házastársuk nevét, születésük dátumát stb. A jelszó kihallgatása történhet egyszerűen annak begépelésekor, de gyakori támadási forma volt a korai hálózati rendszerekben a hálózaton kódolatlanul továbbított jelszók begyűjtése is.

A jelszó kitalálása ellen a következőképen védekeznek a rendszerek:

- „Nehezen kitalálható” jelszó választására kényszerítik a felhasználót.
- Jelszó gyakori cseréjére kötelezik a felhasználót. A jelszó érvényessége időben korlátozott a rendszerben, az érvényesség lejártával a rendszer kéri a felhasználót jelszavának megváltoztatására. Általában a rendszerek tárolják a korábban már használt jelszavakat, és ezeket nem engedik megint használni.

Ha nehezen kitalálható jelszót szeretnénk választani, akkor érdemes figyelembe venni, milyen tipikus támadási stratégiák léteznek egy felhasználó jelszavának megfejtésére. Az egyik ilyen támadási stratégia, hogy a támadó a felhasználók által gyakran használt jelszavakkal, illetve a felhasználó nevéből, esetleg más személyes adataiból generált kulcsszavakkal próbálkozik.

Ennek a támadásnak kiküszöbölésére szolgálnak az ún. jelszóellenőrző programok. A rendszer új jelszó megadásakor vagy valamilyen gyakorisággal rendszeresen lefuttatja a jelszóellenőrző programot. Ez mechanikusan végigpróbálja a felhasználó nevéből, ismert adataiból kikövetkeztethető legkézenfekvőbb, illetve a rendszerben mások által gyakran használt jelszavakat. Ha ilyet talál, felszólítja a rendszer a felhasználót jelszavának megváltoztatására.

A másik gyakori támadási forma, az ún. szisztematikus támadás, amikor a támadó egy szótárból módszeresen végigpróbálgatja az összes benne szereplő szót. Ezt a támadást nagyban segítheti, ha valamilyen módon ismertté válik a jelszó hossza és ez a hossz rövid.

Az ilyen támadások meghiúsítása érdekében a rendszerek általában megszabják a legrövidebb elfogadható jelszó méretét és megkövetelik legalább néhány nem alfanumerikus karakter (szám vagy jel) illetve nagybetű használatát.

A szisztematikus támadást nehezíti, ha gyanúsan sok, rövid időn belül ismétlődő hibás belépési kísérlet esetén a rendszer – esetleg átmeneti időre – letiltja a további próbálkozást.

Az előbbieken túl fontos figyelembeveendő szempont amikor a rendszer nehezen megfejthető jelszavak választására készteti a felhasználókat, hogy a jelszó ne legyen annyira nehezen memorizálható, hogy azt a felhasználó ne tudja megjegyezni. Ezzel ugyanis arra ösztönzi a rendszer a felhasználót, hogy a jelszavát valamilyen formában külön tárolja. Ez igen veszélyes lehet, hiszen ebben az esetben a védtelen helyen tárolt jelszó könnyedén illetéktelen kezekbe kerülhet.

A jelszó kihallgatásának megakadályozása a következőképpen történhet:

- A legegyszerűbb alapvető védekezés, hogy a jelszó megadásakor az nem jelenik meg a képernyőn.

- A felhasználók jelszavait a rendszer csak a rendszergazda által elérhető védett állományban, vagy kódoltan tárolja. Gyakran alkalmazott megoldás, hogy a rendszer a jelszavakat kódolja olyan, nem invertálható kóddal, mely esetén az eredeti jelszavak a kódolt állomány alapján nem állíthatók vissza. A rendszer egy felhasználó belépésekor a megadott jelszót ugyanazzal az eljárással kódolja, és a kódolt jelszót hasonlítja össze az általa tárolt változattal. Ebben az esetben a kódolt jelszavakat tartalmazó fájl lehet akár mindenki számára elérhető, hiszen abból az eredeti jelszó nem fejthető meg. Ilyen módszert alkalmaz a UNIX-rendszerek többsége a felhasználók jelszavainak tárolására. Annak oka, hogy az utóbbi időkben fejlesztett UNIX-rendszerek esetén a jelszavakat tartalmazó fájl mégsem publikus az, hogy az ilyen tárolás megkönnyíti a korábban már említett szótárt használó szisztematikus támadást, ha a használt kódolás algoritmusa a támadó számára ismert.

A jelszavak lehallgatásának megakadályozása érdekében fejlesztették ki a felhasználó által ismert algoritmussal történő azonosítást. Ebben az esetben létezik egy, a felhasználó és a rendszer által ismert algoritmus, ami általában két, terminálon begépelhető betűsorozatot rendel egymáshoz. A rendszer belépéskor egy véletlen betűsorozatot ad a felhasználónak. A felhasználónak az algoritmust alkalmaznia kell a megadott sorozaton, és annak eredményével kell válaszolnia. Ebben az esetben az esetleges támadó akár tudhatja is a két karaktersorozatot, de azt nem tudja többet használni, mert a rendszer mindig más és más betűsorozatot használ egy felhasználó beléptetésekor. Ez az azonosítási eljárás természetesen akkor működik jól, ha minden felhasználó esetén különbözik a használt algoritmus. Ezt úgy oldja meg a rendszer, hogy az algoritmusnak két bemeneti paramétere van. Az egyik paramétert a rendszer az első belépéskor közli a felhasználóval, és minden alkalommal ezt a paramétert használja a felhasználó azonosításakor a rendszer által minden belépéskor külön generált változó paraméterrel együtt.

2.5.2.2. A rendszer biztonságát növelő általános módszerek

A rendszer elleni támadások megakadályozásának általános módszere a veszélyeztetett pontok figyelése *(threat monitoring)*. Folyamatosan vizsgálja a rendszer a felhasználók tevékenységét és a gyanús aktivitást jelzi a gép rendszergazdájának vagy a felhasználónak. Gyakori például, hogy a felhasználót belépésekor tájékoztatják arról, hogy mikor használta utoljára a rendszert, illetve az azóta eltelt időben hány sikertelen – hibás jelszavú – belépési kísérlet történt. Gyakori megoldás, hogy a hibás jelszó megadásakor a rendszer mesterségesen lelassul, és egyre hosszabb ideig váratja a felhasználót a jelszó begépelése után, így nehezítve meg a szisztematikus támadást. Még szigorúbb, de gyakori megoldás, hogy a rendszer néhány sikertelen belépési próbálkozás után a felhasználó belépési lehetőségét meghatározott ideig letiltja.

Az esetleges betörések diagnosztizálására szolgál az ún. aktivitás naplózás *(audit logging)*, mely során a rendszer feljegyzi az egyes erőforrásokhoz történő hozzáférés időpontját, a műveletet végző felhasználót és az erőforráson végzett műveletet. Ez a módszer ugyan nem véd az illetéktelen hozzáférésektől, de az elkövető utólagos azonosítását megkönnyíti.

A rendszer publikus, vagyis több felhasználó által is elérhető csatornán továbbított üzeneteinek lehallgatását megakadályozó módszer a kódolás vagy rejtjelezés *(encryption)*, melynek célja, hogy az üzenetet olyan módon alakítsa át, kódolja, hogy az értelmezhetetlen legyen a csatornát figyelő támadó számára. Az üzenet valódi címzettje valamilyen többletinformáció (ún. kulcs) segítségével képes az üzenet eredeti tartalmát visszaállítani. A rejtjelezés elsődleges felhasználói a katonai alkalmazások, azonban napjainkban a civil szférában is rohamosan terjed a rejtjelezés használata.

A rejtjelezéssel rokon terület az üzenet illetve partner hitelesítés, ami a kommunikációs partner illetve az általa küldött üzenet hitelesítésére szolgál. Egy publikus csatornán érkező üzenet címzettje nem lehet biztos az üzenet küldőjének személyében, illetve abban, hogy az elküldött üzenetet nem változtatta-e meg valaki. Az Interneten például nagyon egyszerű olyan e-mailt küldeni, amelyben nem a tényleges írója van küldőként megnevezve. Az ilyen támadások kivédése érdekében az üzeneteket többletinformációval látják el. Ez lehet a küldőt azonosító ún. elektronikus aláírás *(electronic signature)*, vagy más, a tényleges üzenetből és a küldőt azonosító információból generált kódolt üzenetrész.

3. Multiprogramozott operációs rendszerek

Tartalom

3.1. Bevezetés
3.2. A számítógép-architektúra
    3.2.1. Az egyprocesszoros von Neumann struktúrájú számítógép-architektúra
    3.2.2. Többprocesszoros, szorosan csatolt számítógéprendszerek
3.3. Folyamatkezelés
    3.3.1. A folyamatmodell leképezése a fizikai eszközökre
    3.3.2. Processzorütemezés
    3.3.3. Ütemezés többprocesszoros rendszerekben
3.4. Tárkezelés
    3.4.1. A főtár megosztása a folyamatok között
    3.4.2. Virtuális tárkezelés
    3.4.3. Fájlrendszerek
    3.4.3.1. Az állományok tárolása a lemezen
3.5. Készülékkezelés
    3.5.1. A kernel B/K-alrendszere
    3.5.2. Háttértárak kezelése
3.6. Operációs rendszerek kezelői felülete
    3.6.1. Az X Window-rendszer

Ebben a fejezetben azokat a problémákat és megoldási elveket tárgyaljuk, amelyek a modellszinten felvázolt operációs rendszer – az absztrakt virtuális gép – egyetlen processzort tartalmazó hardverarchitektúrán történő megvalósítása során merülnek fel.

Több processzor többféleképpen alkothat rendszert. Ezek egy része nem alkalmaz minőségileg új megoldásokat és elveket a multiprogramozáshoz képest. Az ilyen rendszerek egyszerűen a CPU többszörös erőforrásként történő figyelembevételével, vagy összekapcsolt, önálló multiprogramozott alrendszerekként

kezelhetők. Az első esetben a CPU-ütemezés vesz figyelembe néhány új szempontot, a második esetben pedig az alrendszerek külső kapcsolatként látják egymást. A rendszerek más része azonban néhány szolgáltatásban minőségileg is újat nyújt. Ezekkel az ún. hálózati és elosztott rendszerekkel a következő, 4. fejezet foglalkozik.

Ugyancsak ebben a fejezetben szerepelnek a tárak, készülékek, kezelői felületek megvalósításának azok a megoldásai, amelyek a szokásos eszközökre építenek, és a multiprogramozott rendszerekben alakultak ki. Természetesen ezek a megoldások a hálózati és elosztott rendszerekbe is beépülnek.

3.1. Bevezetés

Az 1.2. alfejezetben az operációs rendszerek története kapcsán már szó volt a multiprogramozott rendszerek létrejöttének okairól, illetve néhány szóban arról is, hogy milyen új kihívásokat jelentett e rendszerek megjelenése, és előnyeik mellett milyen új feladatok megoldása elé állították a rendszerfejlesztőket. Röviden szeretnénk tehát csak összefoglalni azokat az indítékokat, amelyek a mai multiprogramozott rendszerek kialakulásához vezettek, valamint vázolni azokat a – később részletesen tárgyalt – problémákat, amelyek megoldása szükséges feltétele volt a multiprogramozott operációs rendszerek megbízható és helyes működésének.

A félvezető technika és a számítógép architektúrák fejlődése során a CPU sebessége olyan nagymértékben megnövekedett, hogy felvetette a lassú perifériás műveletek eredményeire várakozás idejének esetleges kihasználását. Megoldást erre egy másik program futtatása jelenthetett. Vagyis az, hogy az operációs rendszer „egyszerre” több munkát futtat. A multiprogramozott rendszerek létrejöttében az alapvető gazdaságossági szempontok mellett természetesen szerepet játszottak olyan további szükséges technikai feltételek is, mint a véletlen hozzáférésű tárak megjelenése vagy a tárkapacitás megnövekedése.

A multiprogramozott rendszerekben a következő lépések alapján történik a folyamatok futtatása:

- Az operációs rendszer nyilvántartja és tárolja a futtatandó folyamatokat.
- A futtatható folyamatok közül az operációs rendszer választ.
- A futásra kiválasztott folyamat addig fut, amíg várakozni nem kényszerül, illetve bizonyos rendszereknél amíg el nem veszik tőle a futás jogát.
- Az operációs rendszer megjegyzi a várakozni kényszerülő folyamat várakozási okát, és elmenti azt a környezetet, amely a későbbi továbbfutáshoz szükséges. Ezután kiválaszt egy másik, futni képes folyamatot (beállítja a környezetét) és elindítja.
- Ha a félbehagyott folyamat várakozási feltétele teljesül, az operációs rendszer a következő alkalommal, amikor az éppen futó folyamat várakozni kényszerül, újra elindít(hat)ja.

Az alapmodellből és a fenti lépésekből adódóan a multiprogramozott rendszerek a következő új problémákat vetették fel:

- A folyamatok közötti hatékony átkapcsolások megkövetelik, hogy több program egyidőben a központi tárban tartózkodhasson, vagyis a *tárgazdálkodást.*
- A folyamatok között egyszerre több *futásra kész* is lehet, amelyek közül az operációs rendszer a *CPU-ütemezés* során választ.
- A CPU mellett a rendszerben levő többi erőforrás használatát több folyamat is igényelheti. Ennek megfelelően az *erőforrásokat folyamathoz kell rendelni,* a hozzáféréseket koordinálni *(szinkronizálni)* kell, bizonyos erőforrásokhoz *biztosítani* kell a *kizárólagos használat jogát*, a *holtpont helyzeteket kezelni* kell.
- A *védelmi mechanizmusokon* keresztül az operációs rendszernek biztosítania kell, hogy az egyes programok ne zavarják egymás, illetve az operációs rendszer működését.
- A számítógép bonyolultságának növekedése miatt a hardverarchitektúra, és különösen a B/K készülékek részleteit el kell rejteni, és a szolgáltatásokat megfelelően ki kell bővíteni a *felhasználóbarát virtuális gép és környezet* biztosítása révén.
- A hálózatok, elosztott rendszerek megjelenése óta biztosítani kell a megfelelő *kommunikációs felületet, kapcsolatot más számítógépek és programok* felé.

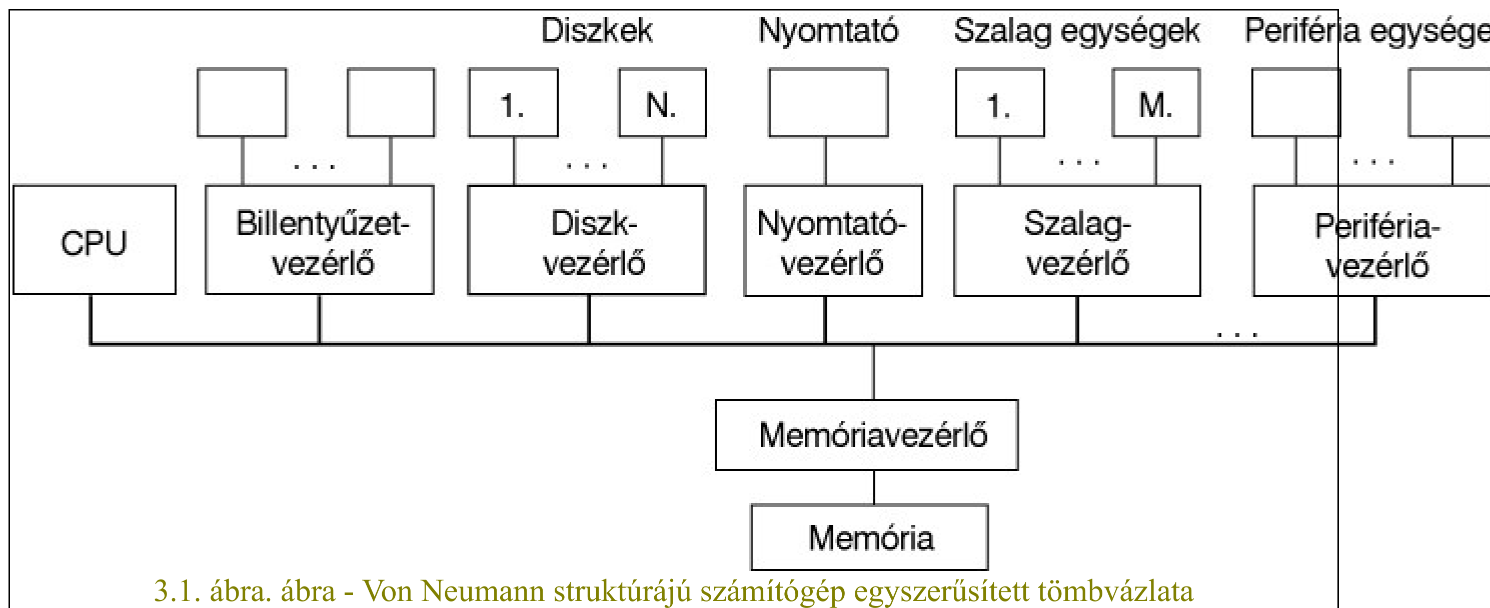
A következő alfejezetekben először annak a hardverarchitektúrának a lényegesebb jellemzőit foglaljuk össze, amelyre a multiprogramozott operációs rendszerek építhetnek, majd az imént felsorolt problémákat és megoldásaikat fogjuk részletesebben vizsgálni.

3.2. A számítógép-architektúra

Az operációs rendszer felépítésének és működésének részletes tárgyalásához szükségünk van számítógép-architektúrákra vonatkozó általános ismeretekre, illetve annak pontosítására, hogy milyen hardver modellt tételezünk fel az operációs rendszerek „alatt”. Mint az 1.3. fejezetben láttuk, a hardverarchitektúra igen széles teljesítményspektrumot foghat át, és igen sokféle lehet. A könyv írása során abból a feltételezésből indultunk ki, hogy az olvasónak megvannak a megfelelő előismeretei e tekintetben, így itt csak ismétlés jelleggel, nagyon nagy vonalakban utalunk azokra a legfontosabb hardversajátságokra, amelyek a multiprogramozott rendszerekben tipikusak, és amelyekre építve az operációs rendszerek megvalósítják feladataikat.

3.2.1. Az egyprocesszoros von Neumann struktúrájú számítógép-architektúra

A **von Neumann struktúrájú**, modern, **általános célú számítógépek** CPU-ból és különböző berendezésvezérlő egységekből állnak, amelyeket közös busz (sín) köt össze. Az egységek közötti információ (utasítás és adat) áramlás a buszon keresztül történik, és ez biztosítja a CPU- és készülékvezérlők memóriához való hozzáférését is. Minden egyes készülékvezérlő egység bizonyos típusú – hozzájuk kapcsolt – készülék(ek) működését és használatát kontrollálja (3.1. ábra). A CPU- és a készülékvezérlők egyidejűleg működ(het)nek.



3.1. ábra. ábra - Von Neumann struktúrájú számítógép egyszerűsített tömbvázlata

3.2.1.1. Bekapcsolási folyamat

A számítógép bekapcsolásakor vagy újraindításakor egy egyszerű indítóprogram (az ún. *bootstrap program*) kezd el futni, amely beállítja (inicializálja) a rendszer kezdeti állapotát, majd betölti az operációs rendszer magját. Ezután az operációs rendszer veszi át a vezérlést, feltérképezi és inicializálja a hardvert, felépíti a rendszertáblákat (például megszakítási vektortábla), elindítja a rendszerfolyamatokat (ha vannak), és elindul(nak) a felhasználói bejelentkezésre váró folyamat(ok). Ebben az állapotában a rendszer *valamilyen eseményre – szoftver vagy hardver megszakításra (interrupt) – vár.*

3.2.1.2. Megszakítási rendszer

A megszakítási rendszer az architektúra egyik legsajátosabb, leginkább processzorcsalád-függő eleme.

Külső hardver megszakítás bármikor érkezhet – általában a buszon keresztül – a CPU-hoz. A szoftver azáltal kezdeményez megszakítást, hogy egy speciális fajta utasítást, *rendszerhívást (system call, monitor call)* hajt végre. Külön osztályát képezik a megszakításoknak a hibamegszakítások, amelyek valamilyen működési rendellenesség esetén következnek be. Összességében tehát nagyon sokféle esemény, például perifériás művelet befejeződése, nullával való osztás, illegális memória hozzáférés stb. eredményezhet megszakítást.

Megszakítás esetén a CPU felfüggeszti az éppen futó tevékenységet, és adott helyre adja a vezérlést. A megszakításkezelés *logikai* lépései szerint az így elindított megszakításkezelő program szükség szerint elmenti a megszakított folyamat környezetét (regiszterek, verem tartalmát stb.), majd a megszakítás okának megfelelő kiszolgáló rutin fut. Befejeződése után visszaállítja a régi környezetet, és visszaadja a vezérlést a korábban futott folyamatnak. A különböző fajtájú megszakítások kezelésére külön rutinok szolgálnak.

A megszakítási rendszerek lehetnek *egy-* és *többszintűek*. Előbbi esetben egy megszakítás kiszolgálása alatt le vannak tiltva az újabb megszakítások. A kifinomultabb rendszerek többszintű megszakítást tesznek lehetővé.

Ilyenkor lehetséges megszakítás fogadása megszakítás kezelés alatt is. A *maszkregiszter* segítségével állítható be, hogy adott interrupt idején melyek a fogadható magasabb prioritású megszakítások.

Az operációs rendszerek ún. *esemény-vezérelt rendszerek*, azaz, ha nem fut éppen valamilyen folyamat, nincs B/K-kiszolgálás, felhasználói kérés, akkor az operációs rendszer tétlenül vár valamilyen eseményre, interruptra.

Fentiek miatt a megszakításkezelés a számítógép-architektúrák kiemelt fontosságú részét alkotja. Gyors működést várunk tőle (hogy ne tartsa fel a folyamatokat), ezért általában úgy valósítják meg a megszakításkezelést, hogy egy rendszertáblában tárolják az ismert, véges számú lehetséges megszakításhoz tartozó kezelőrutin kezdőcímét. A rutinok futási időre optimalizáltak, még a felesleges adminisztrációt okozó eljáráshívásokat is kiküszöbölik, és *egy* eljárás keretében végzik el a megszakításkezelés *összes szükséges lépését*.

3.2.1.3. B/K-struktúra

A perifériás műveletek elindításakor a CPU feltölti a kívánt perifériavezérlő regisztereit, melyek tartalma alapján a vezérlő azonosítja a kívánt szolgáltatást és megkezdi az adatátvitelt egy lokális pufferen keresztül. Az adatátvitel befejeződésekor (vagy ha a puffer megtelik) a készülékvezérlő egy megszakításkéréssel értesíti erről a CPU-t.

A perifériás átvitelt a készülékvezérlő is kezdeményezheti – megszakításkéréssel –, amennyiben a pufferébe adat érkezik kívülről, valamilyen perifériás készülék felől.

A felhasználói folyamatok által kezdeményezett perifériás műveletek fenti folyamata kétféle ütemezéssel történhet. **Szinkron B/K** esetén a folyamat csak a perifériás művelet végrehajtása után kapja vissza a vezérlést. A másik lehetőség, az **aszinkron B/K** szerint a folyamat már a perifériás művelet megindítása után visszakaphatja azt. Ez utóbbi változat hatékonyabb rendszerműködést eredményez.

3.2.1.4. Közvetlen memória hozzáférés, DMA

Nagy sebességű készülékek nagyobb méretű adatblokkjainak a memória és a készülék között történő gyors átvitelére dolgozták ki a **közvetlen memória hozzáférés (DMA: Direct Memory Access)** technikáját. Ilyen – egyszerre elérhető – nagyobb méretű tömböknél a CPU-t erősen megterhelné, és a rendszer működését nagyon lelassítaná, ha az információt – a perifériás készülék nagy sebessége miatt viszonylag kis időközökkel – kis egységekben, például karakterenként, megszakítással vinnénk a CPU-regiszterekbe, vagy rajtuk keresztül a memóriába, illetve innen a perifériavezérlő pufferébe.

Miután a CPU a készülékhez tartozó, megfelelő regisztereket – mutatók, számlálók – beállította a készülékvezérlőn, vagy a DMA-vezérlőn, a készülékvezérlő a pufferéből közvetlenül a memóriába, vagy egy memóriaterületről a saját pufferébe tölt egy (a puffer méretének megfelelően tipikusan 128 byte és 4 Kbyte közötti méretű) adatblokkot anélkül, hogy a CPU-t megzavarná a működésében. Legfeljebb *buszciklust* kell

„*lopjon*” *(cycle stealing)* a CPU elől az átvitelhez. A DMA-átvitel befejeződése után a perifériavezérlő, vagy a külön DMA-vezérlő *blokkonként egyetlen* megszakítással jelzi a művelet befejeződését.

3.2.1.5. Tárak

Az utasításoknak (és a hozzájuk tartozó operandusoknak) a központi tárban kell lenniük a végrehajtáskor. A regisztereken kívül ez az egyetlen olyan memória fajta, amelyhez a CPU közvetlenül hozzá tud férni. Az utasítás-végrehajtási ciklus során az utasítások először (a programszámláló értékének megfelelő címről) az utasításregiszterbe kerülnek, majd dekódolás és az esetleges operandusok beolvasása után végrehajtódnak. Az eredmények a legtöbb esetben a memóriába íródnak vissza.

A von Neumann struktúrájú számítógépeknél az adatok és utasítások nincsenek megkülönböztetve. Ugyanabban a memóriában, ugyanúgy vannak tárolva, csak a környezetük alapján lehet különbséget tenni közöttük.

A fentiekből következően az lenne az optimális, ha az összes programot és adatot a központi tárban tárolhatnánk. Sajnos ez nem lehetséges, mert egyrészt a központi tár *túl kicsi* az összes szükséges program és adat állandó tárolásához, másrészt pedig ez a memória *nem képes megőrizni* (elveszti) a tartalmát a tápfeszültség kikapcsolása vagy kimaradása esetén.

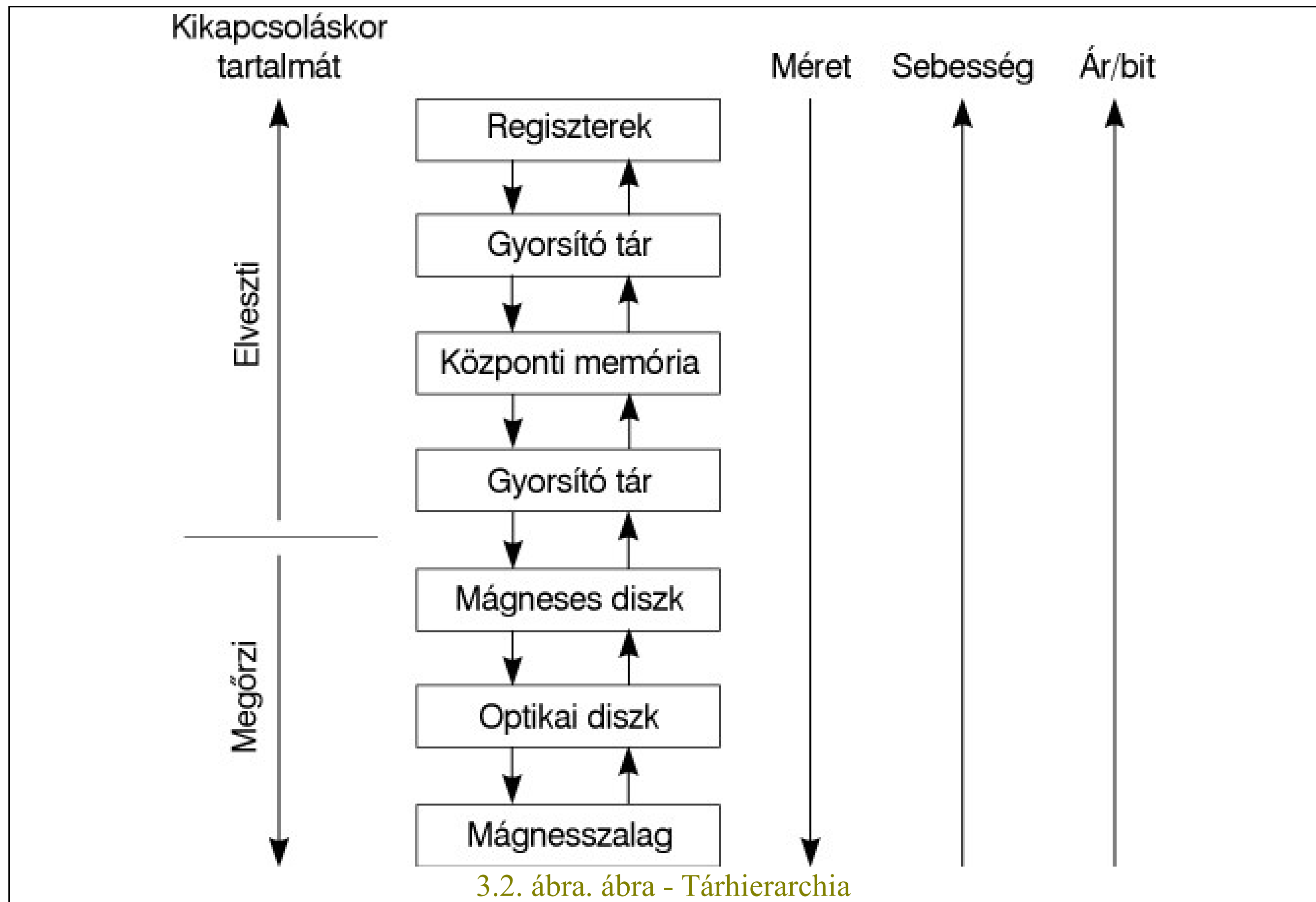
Mindezek miatt megfelelően *nagyméretű* másodlagos tárolási lehetőséget kell biztosítani az információ *állandó* tárolására. Ebben az értelemben a **másodlagos** vagy **háttértárak** a központi memória kiterjesztésének tekinthetők.

A számítógépek adattároló rendszereit sebességük, méretük és áruk alapján hierarchia szintekbe sorolhatjuk (3.2. ábra). A hierarchián lefelé haladva egyrészt csökken a bitenkénti költség (és így nő az elérhető méret), de az elérési idő növekszik. Az effektív elérési sebesség növelése céljából **gyorsítótárakat (cache)** szokás alkalmazni a fő szintek – regiszterek és központi memória, illetve főmemória és háttértárak – között. A CPU és a memória közötti gyorsítótárat általában hardver kezeli. Ezek gyakran a processzorhoz integrált gyors és viszonylag kisméretű tárak. A memória és a háttértár közötti gyorsítótárakat pedig az operációs rendszer valósítja meg. A gyorsítótárakban a mögöttes memóriaszint leggyakrabban, vagy a közeljövőben várhatóan használt információi találhatók, amelyek a rájuk való hivatkozás százalékos arányának (találati arány) megfelelően javítják az átlagos hozzáférési időt.

A regiszterek, a központi memória és a gyorsítótárak a tápfeszültség kikapcsolásakor elveszítik a tartalmukat, míg a háttértárak megőrzik azt.

A leggyakrabban használt másodlagos tárak a lemezegységek (mágneses, illetve optikai) – ezek blokkonként címezhető *véletlen hozzáférésű (random access)* memóriák –, valamint a mágnesszalag, amely *szekvenciális hozzáférésű*. A mágneses diszkek még ma is igen elterjedten használtak, ezért az ezeket érintő adatforgalom hatékonysága (gyorsasága) fontos tényező lehet a rendszer optimalizálásban. Az optikai diszkek, bár lassabbak a mágneses diszkeknél, elérhető áruk és egyéb kiváló tulajdonságaik miatt egyre inkább terjedőben vannak. A

mágnesszalagok sok szempontból elvesztették már a jelentőségüket lassúságuk és hozzáférési módjuk merevsége miatt, azonban az adatsérülésekkel szembeni robusztusságuk, és nagy tárolási kapacitásuk okán ma is használják őket igen nagyméretű adathalmazok *hosszú távú* tárolására.



3.2. ábra. ábra - Tárhierarchia

3.2.1.6. Védelem

Védelmi kérdésekkel később részletesen foglalkozunk, ezért itt éppen csak megemlítjük, hogy a mai számítógép rendszerek komoly hardver támogatást nyújtanak különböző védelmi mechanizmusok megvalósításához.

Védelemről alapvetően három szinten beszélhetünk. Ezek a CPU-védelem, az operációs rendszer védelme és a felhasználói programok, területek védelme. Az utóbbi kettő az, amely szorosan kötődik az operációs rendszerek témájához, így ezekről a későbbiekben részletesen fogunk beszélni. Az első, azaz a CPU-védelem kérdése túlmutat a könyv által tárgyalni kívánt témákon, ezért csak a teljesség kedvéért említjük, hogy ide tartozik például a hurokba került programoknak egy időzítő által okozott megszakítása *(timer IT)*, watchdog processzorok működése stb.

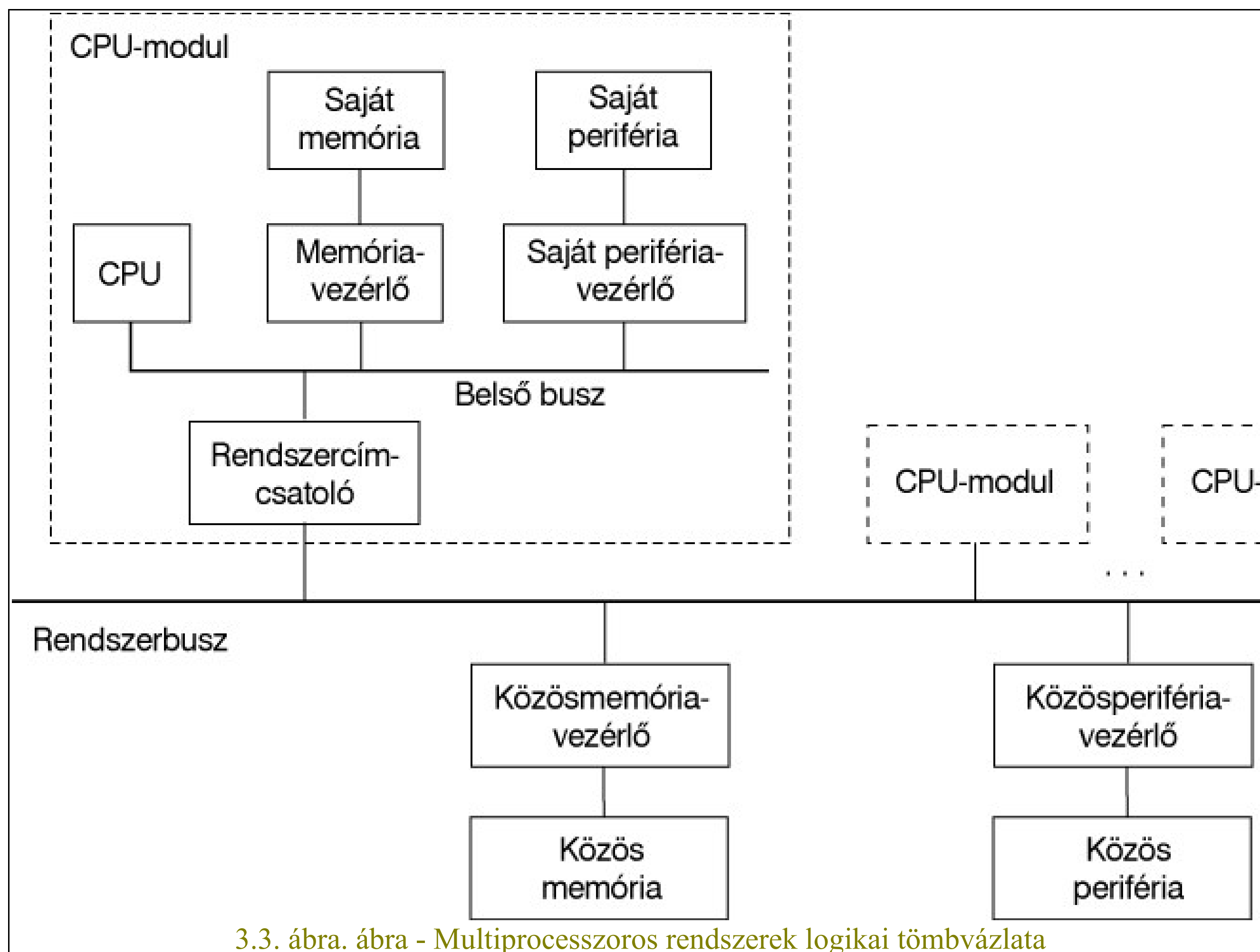
Az operációs rendszer szintjén megvalósítandó védelmek alapfeltétele, hogy a CPU két működési módban tudjon dolgozni, amelyek közül az egyik, az utasításkészlet és az alapvető erőforrások teljes körű használatát biztosító ún. privilegizált (rendszer, kernel stb.) mód, csak az operációs rendszer számára legyen fenntartva.

Mind a védelmet, mind egyéb rendszerfunkciókat további hardver megoldások támogathatják (például memóriakezelő egységek, asszociatív tárak, speciális környezetváltó utasítások stb.), amelyeket az egyes funkciók tárgyalásakor részletezünk.

3.2.2. Többprocesszoros, szorosan csatolt számítógéprendszerek

Többprocesszoros, szorosan csatolt (ún. **multiprocesszoros**) **rendszerekben** a korábbiakkal ellentétben nem egy, hanem több CPU található, és az egyes processzorok között a kapcsolat közös erőforráso(ko)n keresztül valósul meg. A szorosan csatolt rendszerek *közös memóriát és órát*, és az esetek nagy részében *közös operációs rendszert* használnak. (Természetesen ezen felül mindegyik processzorhoz tartozhatnak saját erőforrások, például saját memória is, amelyeket közvetlenül csak az adott processzor ér el.)

A rendszer moduljait egy rendszersín köti össze, amelyen keresztül a közös információforgalom bonyolódik, illetve amelyen keresztül a közös memóriához hozzá lehet férni (3.3. ábra). Ha a CPU-modulok azonos felépítésűek és tulajdonságúak, akkor **homogén rendszerről** beszélünk, ha



3.3. ábra. ábra - Multiprocesszoros rendszerek logikai tömbvázlata

eltérőek is lehetnek, akkor a rendszer **inhomogén**. A CPU-modulok rendelkezhetnek azonos jogokkal (**szimmetrikus** esetben), illetve lehetnek közöttük kiemelt jogosultságokkal bírók (**aszimmetrikus** vagy **master-slave** elrendezés esetén).

Az előző, egyprocesszoros esetet vizsgáló részben tárgyalt elvek és megoldások többprocesszoros esetben is alkalmazhatók, érvényesek. A több processzor miatt azonban koordinálni kell a közös rendszersínhez való hozzáférést. A rendszersín vezérlő a memória vagy perifériacím alapján dönti el, hogy saját (belső) erőforráshoz, vagy közös erőforráshoz kell fordulnia (ez utóbbi esetben a rendszersínen keresztül).

A rendszersínhez való hozzáférést egy sínvezérlő *(arbiter)* egység felügyeli. Több, egyidejű igény fellépése esetén a vezérlő többféle – egyszerű vagy kombinált – stratégia alapján választhat. Egyik legegyszerűbb módszer, ha egyik irányból indulva a másik felé, a „helyileg” előbbre levő buszt igénylő modul kapja meg a rendszersín használati jogát (daisy chain). Ez a megvalósítás igen egyszerű, azonban komoly hátránya a terjedési késleltetés, valamint az, hogy a rendszerbe kapcsolható modulok számát ez utóbbi sajátság korlátozza.

Több esetleges kérés között a vezérlő választhat prioritás alapján is, ilyenkor a sínhasználatot a legnagyobb prioritású modulnak kínálja fel, amennyiben a rendszerbusz szabad. Kritikus hibát okozhat a prioritáseldöntő meghibásodása.

Azonos CPU-n futó folyamatok a saját belső memóriájukon osztoznak (ha van ilyen), míg a különböző CPU-kon futó folyamatok a közös memórián keresztül kommunikálnak egymással. Emiatt a rendszersín mind teljesítőképessége, mind esetleges meghibásodása miatt erősen befolyásolja a rendszer működését.

A vázolt felépítési és működési struktúra lehetővé tesz további hierarchikus szerveződést is, amikor egy modul belső buszára is több processzort kapcsolunk.

3.3. Folyamatkezelés

A 2. fejezetben tárgyalt folyamatmodell feltételezte, hogy minden folyamat a saját különálló processzorán fut. A multiprogramozott rendszerekben azonban egyetlen processzor több folyamatot futtat. A következőkben azzal foglalkozunk, hogy hogyan teszi mindezt.

3.3.1. A folyamatmodell leképezése a fizikai eszközökre

A folyamatmodell feltételezte, hogy a rendszer minden folyamata saját processzoron fut, és saját memóriája van. A multiprogramozott rendszerben egyetlen fizikai processzoron és egyetlen, lineárisan címezhető fizikai memóriában kell megvalósítani a folyamatok működtetését. A modell leképezésének legfontosabb kérdéseit tárgyaljuk a következőkben.

3.3.1.1. A működés alapjai

A folyamat megismert modellje mind multiprogramozott, mind multiprocesszáló rendszerekben használható. A folyamatok kezelésével kapcsolatos feladatok természetesen különbözőek a két rendszertípus esetén, hiszen a modell megvalósítása a különböző rendszerarchitektúrákon egymástól lényegesen különböző problémákat vet

fel. Általában igaz, hogy a multiprocesszáló rendszerekben a folyamatok száma meghaladja a processzorok számát, emiatt ilyenkor is megoldandó az egyes processzorok multiprogramozott működtetése (és emellett természetesen a processzorok együttműködésével kapcsolatos feladatok is). A folyamatok kezelésének alapvető kérdéseit a következőkben multiprogramozott esetre tárgyaljuk, a multiprocesszálás esetére csak utalunk, illetve egyes kérdéseinek külön fejezetet szánunk.

Egy multiprogramozott operációs rendszer minden időpillanatban folyamatok egy csoportjának végrehajtásával foglalkozik, amelyeket egyetlen fizikai processzoron futtat. A **multiprogramozás foka** a rendszerben egy adott pillanatban jelenlévő – tehát megkezdett, de még be nem fejezett – folyamatok száma.

Az operációs rendszer egyik alapvető feladata az erőforrás-gazdálkodás és –kiosztás. A két alapvető erőforrás a processzor (CPU) és a memória. Ezek kiosztása még együttműködő folyamatok esetén is általában a versengés elvén történik, hiszen a folyamatok kódjában rendszerint nem jelenik meg külön utasítás ezek lefoglalására, illetve felszabadítására.

Amikor egy folyamat létrejön a rendszerben, létrejön számára egy logikai memória és egy logikai processzor, amelyekhez fizikai megfelelőket kell rendelni.

A logikai memória létrehozása a fizikai rendszer tekintetében azt jelenti, hogy a folyamat megkapja a fizikai tár egy részét, ahova a kódja és változói (vagy legalább is azok egy futáshoz éppen szükséges része) megfelelő kezdőértékekkel betöltődnek. Elvileg a logikai memóriához a háttértár valamely területe is hozzárendelhető, az éppen futó folyamat végrehajtandó utasításának azonban mindenképpen a memóriában kell lennie (hiszen a háttértárról való betöltés a processzorsebességhez képest rendkívül lassú). A processzorért emiatt csak azok a folyamatok versenyezhetnek, amelyek következő végrehajtandó kódrésze a memóriában van.

A logikai processzorok látszólag párhuzamosan működnek. Valójában az egyetlen fizikai processzor egyidejűleg egyetlen folyamat utasításait tudja végrehajtani. Ha a fizikai processzor elég gyakran átkapcsol egy folyamat egy részletének végrehajtása után egy másik folyamatra úgy, hogy mindegyik elég gyakran jusson szóhoz, akkor egy durvább időléptékben a folyamatok futása párhuzamosnak látszik. A dolgot úgy is felfoghatjuk, hogy a rendszer legfontosabb egyedi erőforrása a fizikai processzor, amelyet egyidejűleg egyetlen folyamat használhat, és amelyhez tartozik egy várakozási sor, ahol a többi, processzorra váró folyamat tartózkodik. A processzor felszabadulásakor az operációs rendszer választja ki valamilyen algoritmussal a sorban várakozók közül azt a folyamatot, amelyik a következő időszakra megkapja a processzort (CPU-ütemezés).

A folyamatok tipikus szerkezete olyan, hogy egy processzor által végrehajtott utasítássorozatot – **processzorlöket (CPU-burst)** – egy B/K-művelet – **be-/kiviteli löket (I/O-burst)** – követ, és ez a két fázis ismétlődik ciklikusan. A be-/kiviteli művelet végrehajtása a megszakításos, vagy DMA jellegű szervezés miatt gyakorlatilag csak a perifériát (és időszakosan a be/kivitelben résztvevő vezérlő egységeket és adatutakat) köti le és nem veszi igénybe a CPU-t. Egy folyamat be-/kiviteli műveleteinek végrehajtása közben a CPU így más folyamat futtatásával foglalkozhat. Ekkor természetesen lehetséges, hogy a működő perifériára egy közben futó

folyamat újabb be/kiviteli műveletet kezdeményez, és ez többször is ismétlődhet. Tehát a perifériákért is versenyeznek a folyamatok, azokat is egyedi erőforrásoknak kell tekinteni, a rájuk várakozó folyamatokat sorba kell állítani, ütemezni kell stb. Amikor egy be/kiviteli művelet befejeződött, a műveletet kezdeményező folyamat ismét a CPU-ért versenyez, hogy következő processzorlökete végrehajtódhasson.

A rendszer folyamatai különbözhetnek abban a tekintetben, hogy futásuk során milyen arányban használják a CPU-t és a B/K-eszközöket. Vannak folyamatok, amelyeket intenzív CPU-használat, és mérsékelt B/K-igény jellemez. Ilyenek például a numerikus közelítő módszerekkel dolgozó egyenletmegoldások, és minden olyan program, amelyik sok számítást végez viszonylag kevés adattal. Más folyamatok alig használnak CPU-t, de annál intenzívebb a B/K-tevékenységük. Ilyenek például a formátum-átalakítást végző konverziós programok. A CPU-használat és B/K-használat aránya alapján megkülönböztetünk **CPU-intenzív**, illetve **B/K-intenzív** folyamatokat. Az erőforrások jó kihasználásának feltétele, hogy egy adott időszakban a CPU-intenzív folyamatok és a B/K-intenzív folyamatok száma kiegyenlített legyen a rendszerben.

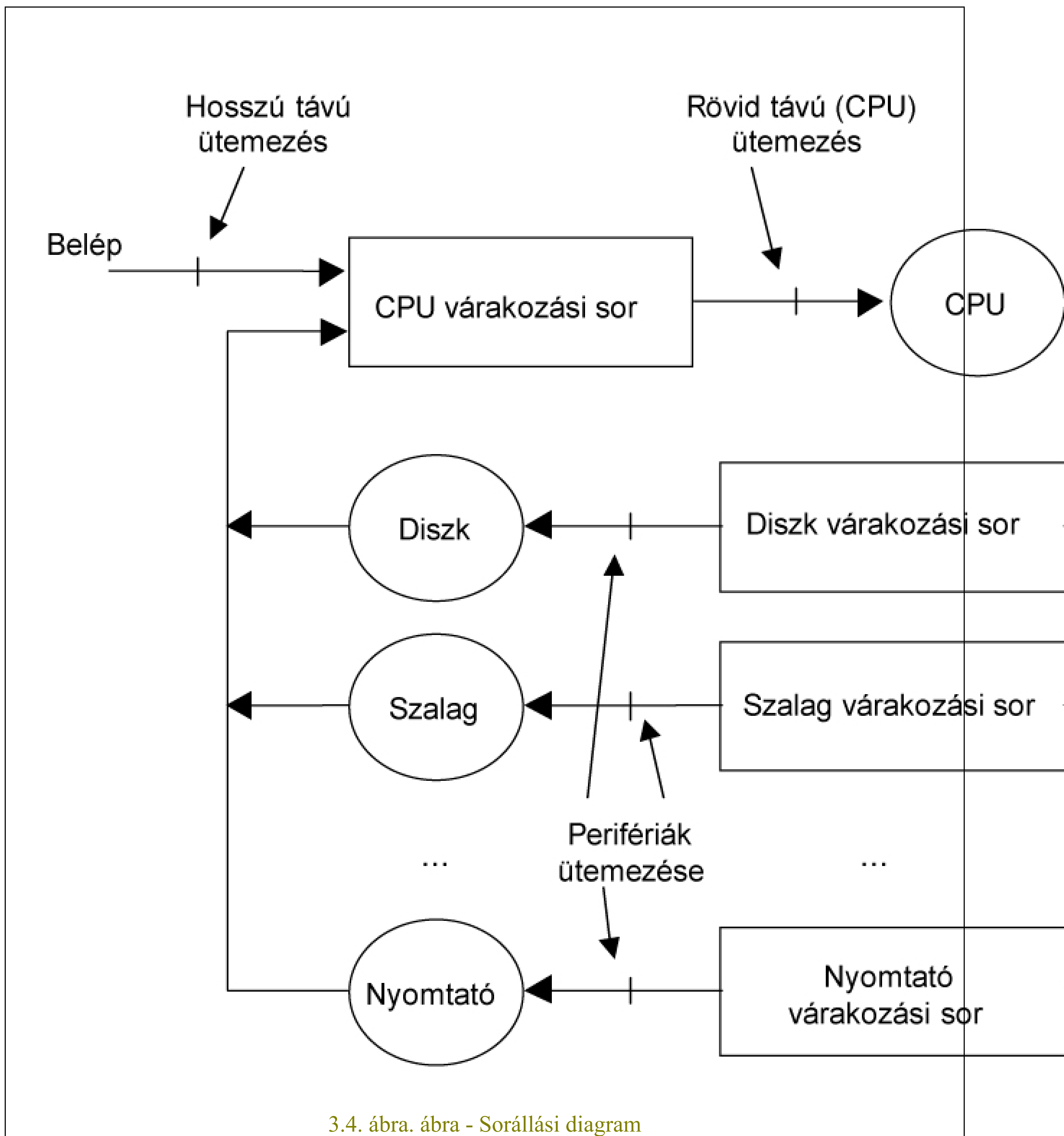
Az operációs rendszer az elvégzendő feladatok végrehajtását valamilyen szempontból optimalizálni igyekszik. Ez lehet például kötegelt feldolgozás esetén az áteresztőképesség, vagy a processzor-kihasználás maximalizálása, egy feladat átlagos átfutási idejének minimalizálása stb., interaktív rendszerek esetén az átlagos válaszidő minimalizálása, valósidejű rendszerek esetén a határidő-mulasztások minimalizálása, és még sok más jellemző. A megfogalmazott célokhoz igazítják azokat az algoritmusokat, amelyek döntenek új folyamatok beengedéséről a rendszerbe (új *job* indulása, új felhasználó belépése), a multiprogramozás fokának változtatásáról (növelés vagy csökkentés), az egyes folyamatoknak kiosztott memóriaterület méretéről, és amelyek ütemezik az erőforrásokat.

Az operációs rendszer folyamatkezelését két modellel szemléltetjük. Az egyik egy tömegkiszolgálási modell, az ún. **sorállási modell** (grafikus ábrázolása a **sorállási diagram**), amelyik a folyamatok erőforrás-használatára koncentrál, a másik egy **állapotmodell** (grafikus ábrázolása az **állapotdiagram**), amelyik a folyamatok végrehajtásának menetére koncentrál.

3.3.1.2. Sorállási modell

A sorállási modell korlátos erőforráskészletért versengő folyamatok rendszerének elemzésére alkalmas. A modell a sorállási diagramon szemléltethető. A modell addig áttekinthető, amíg azoknak az erőforrásoknak a használatát szemléltetjük vele, amelyeket a folyamatok szekvenciálisan használnak, azaz elengedik a birtokolt erőforrást, mielőtt újat igényelnének.

Az operációs rendszerek tipikus sorállási diagramját a 3.4. ábra mutatja. A diagram a processzorlöket, B/K-löket ciklikusságot szemlélteti azzal a feltételezéssel (ami normális működés esetén általában teljesül), hogy a folyamatoknak közben folyamatosan elegendő memória áll rendelkezésükre.



3.4. ábra. ábra - Sorállási diagram

Az erőforrásokat körök, a várakozási sorokat téglalapok jelölik. A rendszer működését a folyamatok irányított élek mentén történő vándorlása szemlélteti. Az új, induló folyamatok processzorlökettel kezdődnek, ezért a CPU várakozási sorába lépnek be. Normál befejeződés esetén ugyancsak processzorlöketből lépnek ki. Közben a be-

/kiviteli műveleteiktől függően kerülnek át a különböző perifériák várakozási soraiba, illetve lesznek a perifériák használói. Egy be-/kiviteli löket után ismét a CPU várakozási sorába kerülnek vissza.

A modell számításokra is alkalmas változata valószínűségi változókat és eloszlásfüggvényeket használ a terhelés (a folyamatok rendszerbe történő belépésének időbeli gyakorisága) és a kiszolgálás (az erőforrások birtoklásának időszükséglete) jellemzésére. A valószínűségi modell kiértékelése alapján adott struktúrára, adott terhelési és kiszolgálási paraméterek mellett számíthatók a rendszer fontos működési jellemzőinek (például átfutási idő, várakozási idő, sorhosszúság) várható értékei, illetve egyéb valószínűségi jellemzői.

A diagramon jól azonosíthatók az ütemezési pontok, amelyek tipikusan a várakozási sorok kimenetén helyezkednek el. Kivétel a rendszerbe való bejutást szabályozó ún. **hosszú távú ütemező**, amelyik tipikusan a kötegelt feldolgozást végző rendszerekben van jelen, és új *jobok* végrehajtásának megkezdéséről (új folyamatok indításáról) dönt. Az elvégzésre váró munkák közül a választás szempontja, hogy a rendszerben a CPU-intenzív és B/K-intenzív folyamatok aránya optimális legyen (*optimális job-mix* fenntartása). Jó tehát, ha a rendszernek van előzetes információja egy-egy munka CPU-, illetve B/K-igényéről.

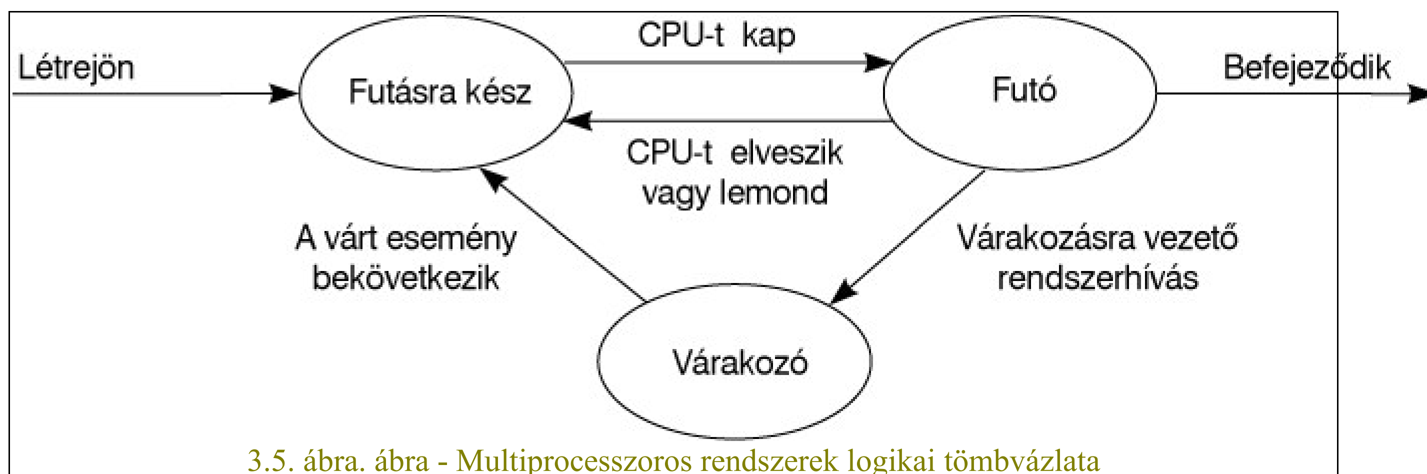
Kitüntetett szerepe van a **CPU-ütemezőnek**, amit általában **rövid távú ütemezőnek** hívnak, és amelynek algoritmusait külön fejezetben tárgyaljuk. (A rövid, illetve hosszú távú elnevezés a futási gyakoriságra utal. A hosszú távú ütemező általában akkor fut le, amikor egy *job* végrehajtása befejeződik. A rövid távú ütemező pedig biztosan lefut, amikor egy processzorlöket befejeződik. A két gyakoriság között több nagyságrend különbség van). A perifériák ütemezése a leggyakrabban érkezési sorrendben (FIFO) történik. Kivétel például a mágneslemez, amelynek hatékony ütemezésével ugyancsak foglalkozunk a későbbiekben.

Néhány rendszerben működik **középtávú ütemező** is, amelynek egyik szerepe a multiprogramozás fokának változtatása jellegzetesen azokban a helyzetekben, amikor a memória válik a rendszer szűk keresztmetszetévé, illetve ez a helyzet megszűnik. A középtávú ütemező – észlelve a memóriaszűkét – egyes folyamatokat *felfüggeszt*, memóriaterületüket háttértárra menti és felszabadítja, átmenetileg kivonja őket az erőforrásokért folytatott versengésből. A felfüggesztett folyamatok memóriaterületét a többi folyamat használhatja. Később, ha a terhelés csökken, a felfüggesztett folyamatok visszatölthetők és folytathatják működésüket. A felfüggesztendő, illetve visszatöltendő folyamatok kiválasztásának szempontja is a CPU-intenzív és B/K-intenzív folyamatok kiegyensúlyozása lehet, ami már nemcsak előzetes információkra, hanem a folyamatok korábbi viselkedésére is alapozhat. A középtávú ütemező akkor is felfüggeszthet folyamatot, ha az várhatóan igen hosszú várakozásra kényszerül. A középtávú ütemező helye nem mutatható meg a diagramon, mert a memória használata és a memóriaigény növekedése más erőforrások (CPU) használata közben lép fel, az erőforrás-használat nem szekvenciális, ezért a modell ennek leírására ebben a formában nem alkalmas.

3.3.1.3. Állapotmodell

Egy folyamat végrehajtásának dinamikáját a multiprogramozott rendszerekben egy hozzárendelt állapotjelzővel és az állapotátmeneti gráffal írhatjuk le.

Az állapotátmeneti gráf legegyszerűbb változatát a 3.5. ábra mutatja be.



3.5. ábra. ábra - Multiprocesszoros rendszerek logikai tömbvázlata

Futásra kész (ready) állapotban vannak azok a folyamatok, amelyeknek következő műveletét a CPU bármikor végrehajthatná. Másszóval a CPU-n kívül minden más erőforrást birtokolnak, amire működésük adott szakaszában szükségük van.

Futó (running) állapotban egy multiprogramozott rendszerben egyidejűleg egyetlen folyamat lehet, amelyiknek az aktuális műveletét a CPU éppen végrehajtja.

Várakozó (waiting, blocked) állapotban vannak azok a folyamatok, amelyek nem tudják használni a CPU-t, mert valamilyen feltétel teljesülésére várakoznak. Ilyen feltétel lehet például egy általuk kezdeményezett be-/kiviteli művelet befejeződése, együttműködő folyamatok esetén valamilyen szinkronizációs vagy kommunikációs művelet végrehajtódása.

Állapotátmenetek

Amikor a folyamat *létrejön*, megkapja a fizikai memória egy területét, ahova betöltődik a kódja, vagy legalább is annak az induláshoz szükséges része, létrejönnek a változói, esetleg más erőforrásokat, indítási paramétereket kaphat. Az operációs rendszer nyilvántartásba veszi, és *futásra kész állapotba* állítja, hogy végrehajtása megkezdőzhessen.

Futásra kész => futó állapotátmenet történik, amikor az operációs rendszer CPU-ütemezője kiválasztja a folyamatot végrehajtásra. Új futó folyamat kiválasztása mindenképpen szükséges, ha az előző folyamat befejezte működését, vagy várakozó állapotba került, hiszen ilyenkor a CPU felszabadul.

Futó => futásra kész állapotátmenet két ok miatt fordul elő.

- Az operációs rendszer elveszi a processzort a folyamattól (*preemptív* ütemezés). Amikor a CPU-ütemező nem érkezési sorrendben hajtja végre a futásra kész folyamatokat, hanem például prioritás szerinti sorrendben, akkor előfordulhat, hogy egy folyamat futása közben egy nála magasabb prioritású másik folyamat átlép várakozó állapotból futásra kész állapotba. Ha a prioritás azonnali érvényesítése fontosabb szempont, mint egyszerűen a CPU munkában tartása,

akkor a futó folyamatot meg kell szakítani, vissza kell léptetni futásra kész állapotba, és a magasabb prioritású folyamatot kell futásra kiválasztani.

- A folyamat lemond a processzorról (újraütemezést kér). Együttműködő folyamatok kooperatív viselkedésének egyik formája, hogy nempreemptív ütemezés esetén egy folyamat hosszú processzorlöketek közben, meghatározott pontokon lehetőséget ad más folyamatok futására.

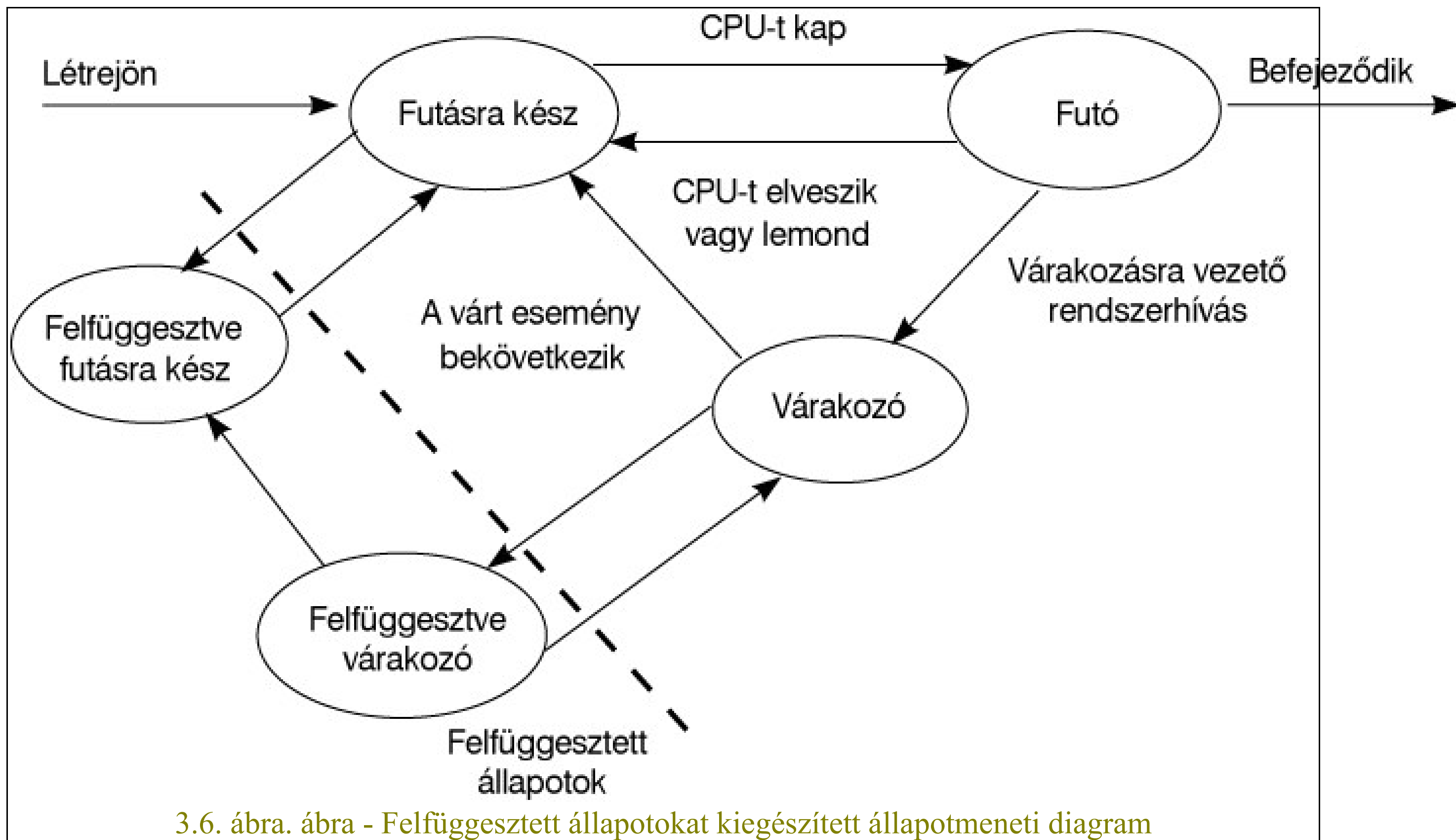
Futó => várakozó állapotátmenet akkor következik be, ha a folyamat olyan műveletet indított, amelynek végrehajtása várhatóan hosszabb (más folyamat futtatására kihasználható idejű) CPU-tétlenséget okozna. Ilyenek a be-/kiviteli műveletek, az erőforrás-foglalások, ha az erőforrás foglalt, a szemaforra kiadott várakozások, ha a szemafor tilosat jelez, az üzenetfogadások, ha az üzenet még nem érkezett meg stb.

Várakozó => futásra kész állapotátmenet történik, ha a folyamat által várt esemény bekövetkezik (például az elindított be/kivitel befejeződik, az erőforrás felszabadul, a szemaforra jelzés érkezik, az üzenet megérkezik stb.).

A folyamatok működésüket akkor *fejezik be*, amikor utolsó utasításuk végrehajtódik. Ez normális esetben egy olyan rendszerhívás (például *exit*), aminek hatására az operációs rendszer felszabadítja a folyamat erőforrásait és törli a folyamatot a nyilvántartásából, esetleg információt ad a folyamat szülőjének a futás eredményéről.

A bemutatott állapotmodell csak a legalapvetőbb eseteket tartalmazza. A gyakorlatban az operációs rendszerek általában bonyolultabbak. Alkalmazzák például a felfüggesztést, ami a folyamat működésének időleges szüneteltetését jelenti, miközben memóriaterülete felszabadul, vagy a folyamat „kilövésének” lehetőségét (végleges eltávolítását általában rendellenes működés miatt), illetve több más várakozó állapotot is bevezetnek. A felfüggesztett állapotokkal kiegészített állapotátmeneti diagram egy változatát mutatja a 3.6. ábra.

Látható, hogy a rendszer várakozó, vagy legfeljebb futásra kész folyamatokat függeszt fel, és a felfüggesztett folyamatokat is áthelyezi *felfüggesztve futásra kész* állapotba a várt esemény bekövetkezésekor.



3.6. ábra. ábra - Felfüggesztett állapotokat kiegészített állapotmeneti diagram

3.3.1.4. Egy megvalósítási séma

A következőkben egy logikai sémát mutatunk be. A legtöbb operációs rendszerben ehhez hasonló megoldásokat találunk, bár a konkrét részletekben jelentős eltérések lehetnek.

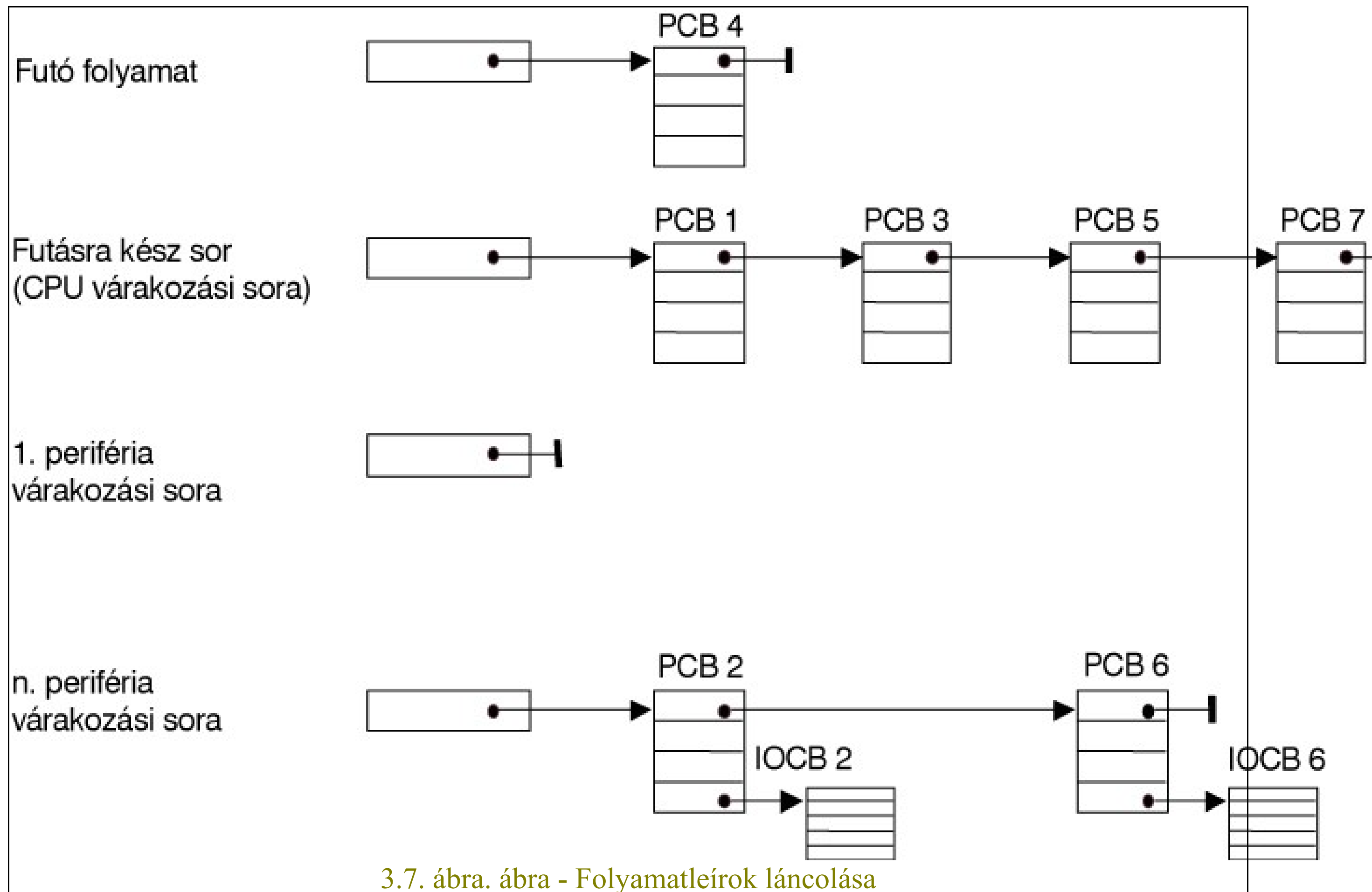
Az operációs rendszer a folyamatok kezeléséhez szükséges információkat egy speciális adatszerkezetben, a *folyamatleíróban (PCB: Process Control Block)* tárolja.

A folyamatleíró tartalma:

- a folyamat azonosítója,
- a folyamat állapota,
- a folyamat szülőjének és gyerekeinek azonosítója,
- a folyamathoz tartozó összes tárterület leírása, mutatók, illetve a virtuális tárkezeléshez szükséges összes adat (címtranszformációs táblák, lemez blokkok leírása stb.),
- a folyamat által használt egyéb erőforrások leírása (például a nyitott állományok leírása),
- a regiszterek tartalma,
- várakozó folyamatoknál a várt esemény leírása,

- ütemezéshez szükséges információk (prioritás, várakozási idő),
- számlázási információk és egyéb statisztikák.

A sorállási modellt az operációs rendszer tipikusan a folyamatleírók láncolt listákra fűzésével valósítja meg (3.7. ábra).



3.7. ábra. ábra - Folyamatleírók láncolása

A 3.7. ábrán a *P4* folyamat fut, a *P1*, *P3*. *P5*. és *P7*. folyamat futásra kész, az 1. periféria szabad, az *n*. periféria a *P2* folyamat által kezdeményezett be-/kiviteli műveletet hajtja végre (a perifériák érkezési sorrendben szolgálják ki a folyamatokat, ezért nincs külön mutató arra, hogy melyik folyamat műveletét hajtja végre éppen a periféria), de már a *P6* folyamat is kiadott rá műveletet. A *P2* és *P6* folyamat várakozó. A be-/kiviteli műveletek paramétereinek tárolására a rendszer *be-/kiviteli leírókat (IOCB, Input-Output Control Block)* használ. Az IOCB tartalmazza a művelet kijelölését (például olvasás, írás), a tárterület címét, ahova, vagy ahonnan a B/K-művelet végrehajtandó, szükség esetén a B/K készülékre vonatkozó egyéb adatokat (például mágneslemez egység szektorcíme), az átviendő adatmennyiséget, állapotjelzőket stb., azaz a művelet végrehajtásához szükséges valamennyi adatot. Az IOCB-k a B/K-műveletet kezdeményező folyamat PCB-jére fűződnek fel, így az elindított B/K-műveletek és az azokra várakozó folyamatok összetartozásának nyilvántartása is megoldott.

A perifériák mellett hasonló várakozási sorok szervezhetők szemaforokra, logikai erőforrásokra és más szinkronizációs, kommunikációs objektumokra.

Látható, hogy a bemutatott megoldásban a folyamat állapotát a PCB-ben tárolt állapotjelző mellett az is jellemzi, hogy a hozzá tartozó PCB éppen melyik sorban található.

Amikor a rendszer átkapcsol egy másik folyamat futtatására, a futó folyamat teljes állapotterét menteni kell, hogy a végrehajtás folytatható legyen (logikai memória, logikai processzor), továbbá az új futó folyamat utoljára elmentett állapotterét kell elővenni és visszaállítani, hogy annak végrehajtása folytatódjon. Ezt a műveletet **környezetváltásnak (context switching)** nevezik. Tekintve, hogy a memóriát a folyamat tartósan birtokolja, a memóriakép megőrződik a folyamat várakozó állapotában is, így csak a logikai processzor átkapcsolása szükséges. Ez egyrészt a fizikai processzor regisztereinek, másrészt az operációs rendszer belső változóinak (rendszertáblázatok, memóriakezelési információk, periféria hozzárendelések stb.) elmentését, és az új folyamatnak megfelelő beállítását jelenti. A mentés a PCB-ben fenntartott megfelelő területre történik.

A be-/kiviteli műveletek végrehajtásának menete a következő:

- a folyamat (általában rendszerhívások segítségével) kitölt egy IOCB-t, amiben megadja a művelet paramétereit,
- B/K rendszerhívást hajt végre az IOCB paraméterként történő átadásával,
- az operációs rendszer
  - hozzáláncolja az IOCB-t a folyamat PCB-jéhez,
  - a folyamat PCB-jét befűzi a periféria sorába,
  - ha a sor üres volt, az IOCB paramétereivel indítási parancsot ad a perifériának,
  - a folyamatot várakozó állapotba helyezi,
  - CPU-ütemezést hajt végre, azaz kiválasztja a következő futó folyamatot és környezetet vált,
  - visszatér (a visszatérés a környezetváltás miatt az új folyamatra történik).

A CPU és a perifériák párhuzamosan működnek. Az átvitelek végét megszakítások jelzik, amelyeket az operációs rendszer kezel (a megszakítások kiszolgáló programjait hardverütemezésű rendszerfolyamatoknak tekinthetjük).

Egy be-/kiviteli megszakítás kezelésekor az operációs rendszer:

- az átvitel helyes vagy hibás eredményére utaló jelzést ír a periféria várakozási sorának elején álló PCB-hez láncolt IOCB-be,

- a sor elején álló PCB-t kifűzi a sorból, átteszi a *futásra kész* sorba, a PCB-ben a folyamat állapotjelzőjét *futásra kész*-re állítja,
- ha van még várakozó folyamat a periféria sorában, a következő IOCB paramétereivel indítási parancsot ad a perifériának,
- ha a CPU-ütemezés preemptív, CPU-ütemezést hajt végre,
- visszatér.

Ezzel a szervezéssel a B/K-rendszerhívást kiadó folyamatnak a rendszerhívást követő utasítása akkor hajtódik végre, amikor az átvitel már befejeződött, és a CPU-ütemező őt választotta ki futásra.

3.3.1.5. Tétlen ciklusok kiküszöbölése

A szinkronizációs és kommunikációs műveletek ugyancsak okozhatják a folyamatok várakozását. A szemafor *P(s)* műveletének korábban (a 2.2.5.3. pontban) bemutatott *definíciós* programja:

while s<1 **do** *skip*;

s:=s-1;

Ez a program az oszthatatlanság mellett még azzal a feltételezéssel is él, hogy minden folyamatot külön processzor hajt végre, amelynek nincs más dolga, mint ciklikusan tesztelni a szemaforváltozót, amíg csak szabadnak nem találja. Multiprogramozott rendszerben ezek a tétlen ciklusok – az ún. **foglalva várakozás (busy waiting)** – használhatatlanok (gyakorlatilag még multiprocesszoros esetben is azok, ezért nevezzük a programot csupán definíciós programnak), hiszen a ciklikus tesztelés feleslegesen kötné le a processzort és a memóriát.

A szemaforkezelést ezért az operációs rendszer hatáskörébe kell utalni *P* és *V* rendszerhívásokkal, amelyek megvalósítása például a következő lehet.

Feltételezzük, hogy az operációs rendszer folyamatkezelő rendszerhívásai között van *Elalszik* művelet, amelyik a hívó folyamatot várakozásba helyezi és CPU-ütemezést indít, valamint *Felébreszt(p)* művelet, amelyikkel egy folyamat futásra kész állapotba tudja tenni a *p* várakozó folyamatot.

Feltételezzük továbbá, hogy a nyelv rendelkezik listakezelő műveletekkel *(Felfűz, Lefűz)*.

A szemafor megvalósításának pszeudo kódja:

type *semaphore* = **record** érték:*integer*:=1; várósor: **list of** folyamat; **end**; *P(s):* s.érték:=s.érték-1; **if** s.érték < 0 **then begin** *Felfűz*(s.várósor); *Elalszik*; **end**; *V(s):* s.érték:=s.érték+1; **if** s.érték < 1 **then begin** *Lefűz*(p,s.várósor); *Felébreszt*(p); **end**;

Megjegyzések:

- A szemafor *s* változójának pillanatnyi értéke tárolja a szemaforra várakozó folyamatok számát (negatív előjellel).

- A szemafor ütemezési elve (a várakozók közül a továbbinduló kiválasztása) a listakezelő műveletek algoritmusában van elrejtve.

3.3.2. Processzorütemezés

A multiprogramozott operációs rendszerek alapjának a CPU-ütemezés tekinthető. A rövid távú ütemezés feladata annak eldöntése, hogy a processzort melyik futásra kész folyamat kapja meg. Mivel, mint azt az előző fejezetben láttuk, a rövid távú ütemező gyakran fut, ezért gyorsnak kell lennie, hogy a CPU-idő ne az ütemezéssel teljék. Ezért ez az ütemező része a kernelnek és állandóan a memóriában van.

A rövid távú ütemezők két alapvető fajtáját különböztethetjük meg. **Preemptív (preemptive) ütemezésről** beszélünk, ha az operációs rendszer elveheti a futás jogát az éppen futó folyamattól, „futásra kész” állapotúvá teheti, és a CPU-t egy másik folyamatnak adhatja, azaz egy másik folyamatot indíthat el.

Nem preemptív (non preemptive) ütemezésről akkor beszélünk, ha az operációs rendszer nem veheti el a futás jogát a folyamattól, azaz a futó folyamat addig birtokolja a CPU-t, amíg egy általa kiadott utasítás hatására állapotot nem vált, azaz csak maga a folyamat válthatja ki az új ütemezést, például ha befejeződik, valamilyen erőforrásra, eseményre vár vagy önként lemond a futás jogáról.

Ütemezés következhet be, ha

- a futó folyamat befejeződik,
- egy folyamat felébred, futásra késszé válik,
- a futó folyamat várakozni kényszerül (valamilyen esemény bekövetkezésére), illetve,
- a futó folyamat önként lemond a futás jogáról vagy pedig elveszik tőle.

Az első és a harmadik esetben az ütemezés mindig környezetváltással jár, hiszen a következő futó folyamat egészen biztosan nem a korábban futott lesz. A másik két esetben előfordulhat, hogy az ütemezőnek nem kell másik folyamatot kiválasztania.

Ha történik ütemezés a második esetben (folyamat felébred), illetve a negyedik eset bizonyos eseteiben (elveszik a CPU-t a futó folyamattól), akkor egészen biztos preemptív az ütemező, egyébként lehet ilyen is és olyan is. Maga az ütemezés úgy történik, hogy az ütemező valamilyen algoritmus szerint kiválaszt egy futásra kész folyamatot a futásra kész sorból (ready queue).

Mielőtt azonban belemennénk a rövid távú ütemezési algoritmusok tárgyalásába, nézzük meg, hogyan lehet ezeket összehasonlítani, illetve milyen követelményeket fogalmazhatunk meg az ütemezési algoritmusokkal szemben.

3.3.2.1. Az ütemezési algoritmusok összehasonlítása

A különböző ütemezési algoritmusok különböző tulajdonságokkal rendelkeznek. Adott helyzetekben alkalmazott algoritmusok kiválasztásánál ezen tulajdonságok hatását kell mérlegelni és a célnak legmegfelelőbbet kiválasztani.

A leggyakrabban alkalmazott összehasonlítási mértékek a következők:

- **Központi egység kihasználtság (CPU utilization).** Alapvető cél, hogy a CPU lehetőleg minél több időt fordítson „hasznos” munka végzésére. A CPU-kihasználtság azt mutatja, hogy a CPU-idő hány százaléka fordítódik ténylegesen a folyamatok utasításainak végrehajtására. A kihasználtságot csökkenti, ha a CPU henyél *(idle)*, azaz nincs olyan folyamat, amelyik futhat, illetve amikor rendszeradminisztráció, ütemezés stb. történik (rezsi, overhead):

$$\frac{\Sigma \text{CPU} - \text{idő} - \Sigma(\text{Henyélés, adminisztráció})!}{\Sigma \text{CPU} - \text{idő}} \times 100[\%]$$

A kihasználtság tipikus értékei 40–90% közé esnek.

- **Átbocsátó képesség (throughput).** Az egy időegység alatt elvégzett munkák számát mutatja.

Elvégzett munkák száma / Idő

Az átbocsátó képesség tipikus értékei nagyon széles tartományban szórnak, a feladatok jellegétől függően (például ...,1/h, ..., 10/sec, ...).

- **Körülfordulási idő (turnaround time).** Egy-egy munkára vonatkozóan a rendszerbe helyezéstől a munka befejeződéséig eltelt időt mutatja.

Σ (Végrehajtási idő + Várakozási idő)

A fenti két összetevő közül a végrehajtási idő nem függ az ütemezéstől, ezért az ütemezési algoritmusok jellemzésére alkalmasabbnak tűnik a várakozási idő.

- **Várakozási idő (waiting time).** Értéke azt mutatja, hogy egy-egy munka összességében mennyi időt tölt várakozással. A várakozó és futásra kész állapotokban töltött időn kívűl ide számítódik a felfüggesztett állapotok ideje, valamint a hosszú távú ütemezésig eltelő előzetes várakozás is:

Σ (Várakozó + Futásra kész + Felfüggesztett + Hosszú távú ütemezésig eltelt) idő

- **Válaszidő (response time).** Időosztásos rendszerekben nagyon fontos, hogy a felhasználók érezzék, hogy a rendszer reagál a parancsaikra. A válaszidő az az idő, amely az operációs rendszer kezelői felületének – esetleg egy felhasználóval kommunikáló folyamatnak – adott kezelői parancs után a rendszer első látható reakciójáig eltelik, vagyis amennyi idő alatt a rendszer válaszolni képes.

3.3.2.2. Az ütemezési algoritmusokkal szemben támasztott követelmények

Az ütemezési algoritmusokkal szemben az operációs rendszerekben különböző – gyakran egymásnak ellentmondó – követelmények fogalmazhatók meg. Az alábbiakban néhányat felsorolunk ezek közül.

- Valamilyen célfüggvény szerint legyen optimális. (Ez a követelmény messze túlmutat az operációs rendszereken, hiszen bármilyen mérnöki tervezés esetén fontos annak a megfogalmazása, hogy az adott rendszer milyen célra készül, és ennek megfelelően lehet törekedni a cél szerinti optimum elérésére.)
- Legyen korrekt, azaz minden folyamat kapjon egy bizonyos CPU-időt, kezeljen azonosan bizonyos folyamatokat.
- Egyes folyamatoknak biztosítson prioritást.
- Kerülje a kiéheztetést.
- A viselkedése legyen „jósolható”, azaz terheléstől függetlenül becsülhető legyen például a várható maximális körülfordulási idő, a futtatási költség stb.
- Legyen minimális a rezsiidő. (Bár ez nem mindig eredményez jobb teljesítményt!)
- Legyen minimális a batch munkák várakozási ideje.
- Legyen maximális az átbocsátó képesség.
- Részesítse előnyben azokat a folyamatokat, amelyek fontos (népszerű) erőforrásokat foglalnak (hiszen az ilyen folyamatok sok más folyamatot várakozásra kényszeríthetnek).
- Részesítse előnyben a kihasználatlan erőforrásokat igénylő folyamatokat.
- Növekvő terhelés esetén a rendszer teljesítménye „elegánsan”, azaz fokozatosan romoljon le és ne omoljon össze hirtelen (graceful degradation). (Napjainkban ez a követelmény egyre fontosabb alap-szemponttá válik, és megint csak azt kell mondanunk, hogy az operációs rendszereken túlmenően, szinte minden összetett, orvosi, diagnosztikai, folyamatirányítási, kommunikációs, beágyazott stb. rendszer kapcsán az egyik elsődleges tervezési szempontként merül fel.)

Jól látható, hogy a fenti elvárások közül több egymásnak ellentmondó, így együttesen nem teljesíthető. Ezért is olyan fontos minden rendszernél megfogalmazni azokat a célokat, amelyek kiemelt fontosságúak az adott esetben, hogy ennek megfelelően kiválaszthatók legyenek azok a szempontok, amelyek mentén optimalizálni akarunk.

3.3.2.3. Ütemezési algoritmusok

Egyszerű ütemezési algoritmusok

Legrégebben várakozó (FCFS: First Come First Served). Nem preemptív algoritmus, amely a legrégebben várakozó folyamatot választja ki futásra. Megvalósítása igen egyszerű, a futásra kész folyamatok egy várakozási sor végére fűződnek fel, az ütemező pedig mindig a sor legelején álló folyamatot kezdi futtatni.

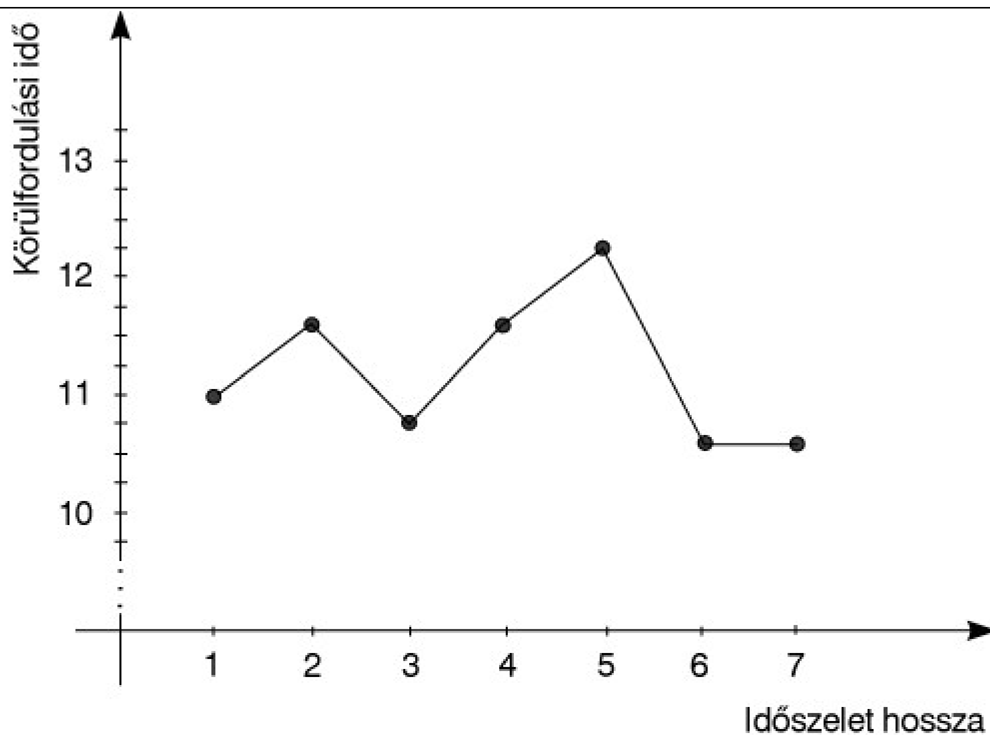
Ennél az algoritmusnál igen nagy lehet az átlagos várakozási idő, mivel egy-egy hosszú CPU-löketű folyamat feltartja a mögötte várakozókat (ezt hívjuk *konvoj hatásnak*). Képzeljünk el egy rendszert egy hosszú CPU-löketidejű és több rövid CPU-löketidejű folyamattal. Az első folyamat futása alatt az összes többi várakozni kényszerül és a perifériák is tétlenek lesznek. Később, amikor a hosszú CPU-löketidejű folyamat perifériás műveletre vár, a helyzet megfordulhat, azaz a rövid folyamatok futása után a CPU lehet tétlen.

Körbeforgó (RR: Round Robin). Preemptív algoritmus, amely az időosztásos rendszerek alapjának tekinthető. Minden folyamat, amikor futni kezd, kap egy *időszeletet (time slice).* Ha a CPU lökete nagyobb ennél, akkor az időszelet végén az ütemező elveszi a folyamattól a processzort, és a futásra kész állapotúvá tett folyamatot a futásra kész sor végére állítja.

Ha a CPU-löket rövidebb az időszeletnél, akkor a löket végén a rendszer folyamatait újraütemezzük, és a futó folyamat időszelete újraindul.

Ezeknél az algoritmusoknál nehéz az időszelet megfelelő méretének a meghatározása. Túl hosszú időszelet esetén ugyanis az algoritmus átmegy FCFS-algoritmusba, míg túl kicsire választás esetén megnő a környezetváltások száma, ami a rendszer hasznos teljesítményét jelentősen lerontja. A statisztikai vizsgálatok alapján ökölszabályként alkalmazható, hogy a CPU-löketek kb. 80%-a legyen rövidebb az időszeletnél.

Egy másik érdekes kérdés az időszelet hossza és a körülfordulási idő közötti összefüggés. Első várakozásainkkal ellentétben az esetek jelentős részében nem monoton függvényt kapunk válaszul. Példaként próbáljuk felrajzolni (környezetváltások nélkül) a fenti összefüggést egy olyan rendszerben, ahol négy futásra kész folyamat van, a következő CPU-löketidőkkel: 6, 3, 1 és 7 időegység (3.8. ábra).



3.8. ábra. ábra - Példa a körülfordulási idő és az időszelet hosszának összefüggésére Round–Robin-rendszerekben

Prioritásos ütemezési algoritmusok

A prioritásos algoritmusok közös sajátossága, hogy a futásra kész folyamatok mindegyikéhez egy, a futás kívánatos sorrendjére utaló számot – **prioritást (priority)** rendelünk. Az algoritmusok pedig a futásra kész folyamatok közül a „legnagyobb” prioritásút választják ki következő futtatandónak.

A prioritás hozzárendelése történhet az operációs rendszeren kívüli tényezők (például a folyamat saját kérése vagy operátori beavatkozás) által – ilyenkor **külső prioritásról** beszélünk, vagy történhet az operációs rendszer által – ilyenkor **belső prioritásról** beszélünk.

A prioritások lehetnek időben állandóak – **statikus prioritás** – vagy (az operációs rendszer által módosítottan) időben változóak – **dinamikus prioritás**.

A következőkben ismertetett algoritmusok esetében a prioritást a CPU löketidő határozza meg.

Legrövidebb (löket)idejű (SJF: Shortest Job First). Nem preemptív algoritmus, amely a futásra kész folyamatok közül a legrövidebb *becsült* löketidővel rendelkező folyamatot választja ki következőnek futásra.

Az algoritmus kiküszöböli a FCFS-nél tapasztalható konvoj hatást, és ennél az algoritmusnál optimális az átlagos várakozási és körülfordulási idő.

Komoly gondot jelenthet azonban az, hogy a löketidők általában nem ismertek. Ilyenkor az operációs rendszer becslésekből indul ki, vagy a folyamatot elindító felhasználó „bevallását” tekinti kiindulásnak. Előbbi esetben a

folyamat előző viselkedése alapján, a korábbi löketidők – általában exponenciális – átlaga szolgál a prioritás alapjául.

Ha a felhasználó által megadott értékeket vesszük alapul, akkor figyelembe kell venni azt is, hogy a felhasználók „nem érdekeltek” a löketidők pontos megadásában, hiszen tisztában vannak vele, hogy kisebb érték vallása esetén előnyösebb lesz a besorolásuk (bár a rajtakapott hazugságokat büntetheti is az operációs rendszer „hátrasorolással”).

Ismételt futás esetén azonban a helyzet sokkal jobb, hiszen a rendszerstatisztikákból már ismertek lehetnek a löketidők.

Legrövidebb hátralevő idejű (SRTF: Shortest Remaining Time First). Ez az algoritmus az SJF preemptív változata. Ha egy új folyamat válik futásra késszé, akkor az ütemező megvizsgálja, hogy az éppen futó folyamat hátralevő löketideje, vagy az új folyamat löketideje kisebb, és a rövidebbet indítja el.

Egy futó folyamat megszakításához és egy másik elindításához környezetváltásra is szükség van, ami szintén időt igényel, így ezt is figyelembe kell venni, amikor egy futó folyamat megszakítása mellett döntünk.

Az algoritmus használata az előzőével azonos problémákat vet fel.

Legjobb válaszarány (HRR: Highest Response Ratio). A prioritásos algoritmusok nagy hátránya a kiéheztetés veszélye. (Vagyis, hogy a kis prioritású folyamatok elől a nagyobb prioritásúak „ellopják” a CPU-t.) Ennek kivédése az **öregítés (aging)** révén történhet, amikor a rendszer a régóta várakozó folyamatok prioritását fokozatosan növeli.

Ezt az elvet alkalmazza az SJF-ből kiindulva a HRR-algoritmus. A folyamatok kiválasztásánál a löketidő mellett, a várakozási időt is figyelembe veszi. A prioritás alapjául a

(Löketidő + *k** Várakozási idő)/Löketidő

összefüggés szolgál (*k* egy alkalmasan megválasztott konstans).

Többszintű algoritmusok

A többszintű algoritmusok sajátossága, hogy a futásra kész folyamatok nem egy, hanem több sorban várakozhatnak. Minden sorhoz prioritás van rendelve és az ütemező csak akkor választ ki futásra kisebb prioritású sorból folyamatot, ha a nagyobb prioritású sorok mind üresek. Az egyes sorokon belül különböző kiválasztási algoritmusok működ(het)nek.

Statikus többszintű sorok (SMQ: Static Multilevel Queues). Ennél az algoritmusnál a folyamatokhoz statikus prioritás rendelődik, azaz minden folyamathoz az elindulásakor rendelünk valamilyen prioritást (vagyis valamilyen kritérium alapján egy adott várakozási sorba soroljuk), amely a folyamat élete során nem változik (3.9.*(a)* ábra).

Egy lehetséges példa a folyamattípusok prioritás besorolására a következő:

- rendszerfolyamatok (ezekhez célszerű a legmagasabb prioritást rendelni, hiszen közvetlen hatással vannak a rendszer működésére),
- interaktív folyamatok (hogy biztosítani tudjunk a felhasználók számára elfogadható válaszidőt),
- interaktív szövegszerkesztők (kevésbé kritikusak, mint az előzőek),
- kötegelt feldolgozás (általában akkor futnak, ha „van idő"),
- rendszerstatisztikákat gyűjtő folyamatok (a későbbi futtatásokhoz, rendszerfejlesztésekhez stb. hasznos, illetve szükséges információkat gyűjtik össze, azonban ezek általában nincsenek közvetlen hatással a *pillanatnyi* rendszer működésre, így alacsony lehet a prioritásuk).

A statikus prioritásra épülő algoritmusok komoly hátránya az alacsony prioritással rendelkező folyamatok nagy várakozási ideje, kiéheztetése. Erre példaként említjük azt a világot bejárt esetet, amely szerint egy 1973-ban kikapcsolt IBM számítógépen az operátor talált egy 1967-ben elindított és még mindig futásra váró folyamatot.

Visszacsatolt többszintű sorok (MFQ: Multilevel Feedback Queues). Az előző algoritmusnál mutatott probléma megoldása a már említett öregítés vagy valamilyen más, ezt teljesen vagy részlegesen kiváltó technika alkalmazása lehet. Az MFQ-ütemezésnél a folyamatokhoz dinamikus prioritás rendelődik. Ez az ütemezés tehát dinamikus, vagyis a folyamatok a prioritás hozzárendelésüknek megfelelően dinamikusan átkerülhetnek egyik sorból a másikba.

Az egyes sorokon belül általában preemptív algoritmusokat – például növekvő időszeletű RR-t – alkalmaz, kivéve a legkisebb prioritású sort, amelyen belül az ütemezés történhet egy egyszerű FCFS által.

Az algoritmus a folyamatokat futásuk alapján különböző maximális CPU löketidejű osztályokba sorolja és a rövid löketidejű folyamatokat részesíti előnyben. A folyamatok elindulásukkor általában a legnagyobb prioritású sorba lépnek be. Ha azonban túllépik az időkorlátot, azaz futásukhoz nem elegendő egy időszelet, akkor az operációs rendszer csökkenti a prioritásukat, és eggyel lejjebbi (kisebb prioritású, ám nagyobb időszeletre, illetve FCFS-re épülő) sorba kerülnek át.

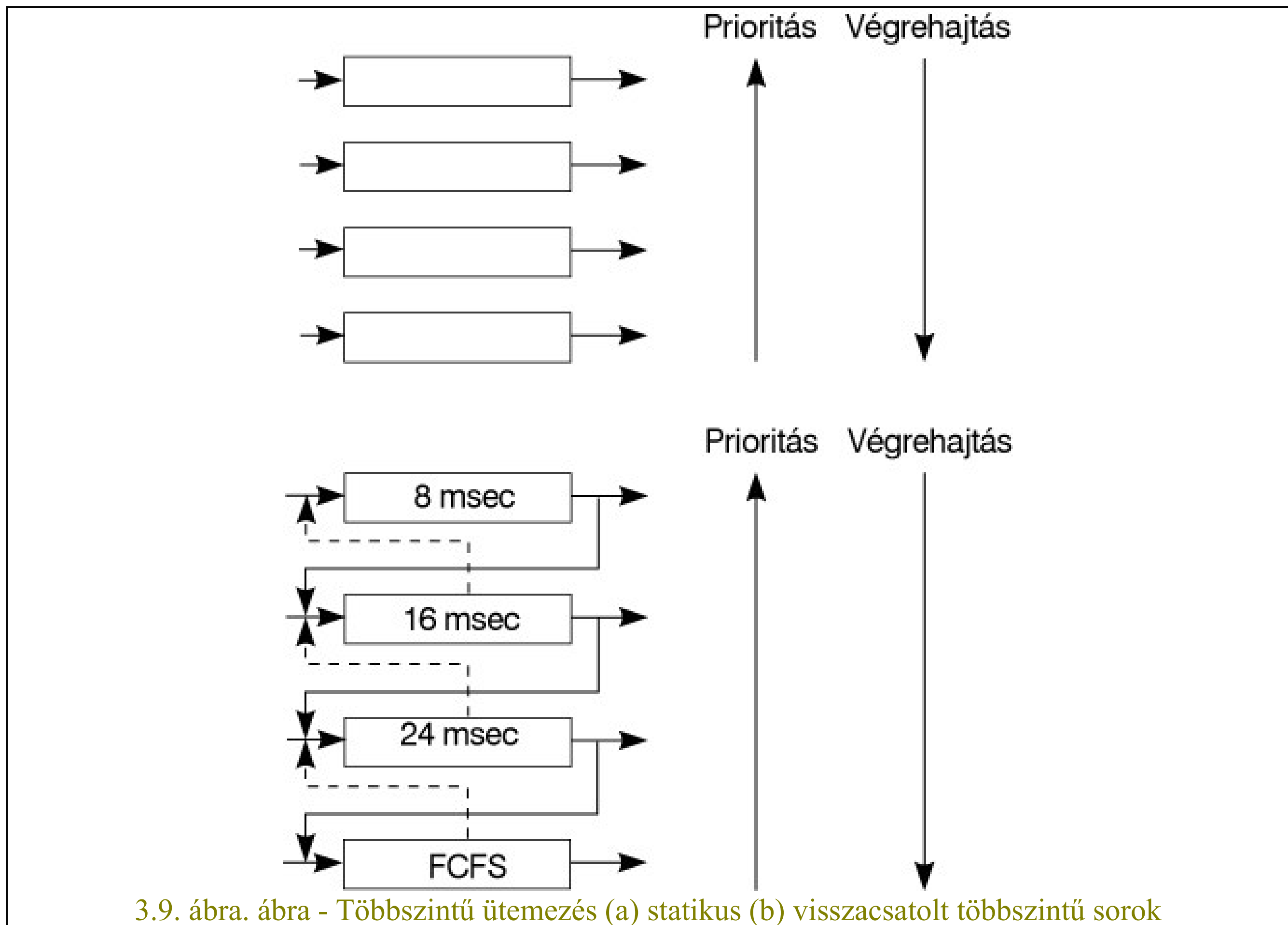
Hogy feloldjuk a maximális löketidőn alapuló besorolásból következő merevséget (vagyis hogy a folyamatok ne hordozzák egész életük során azt a „büntetést", amelyet akácsak egyszeri időszelet túllépés miatt is kaptak), célszerű lehet időről időre felülvizsgálni a folyamatok besorolását, és ha szükséges, például az átlagos löketidő mérése alapján, a prioritásokat módosítani (növelni).

Hasonlóképpen, a régóta várakozó folyamatok prioritását az operációs rendszer megnövelheti *(aging)*, és nagyobb prioritású sorba sorolhatja át (3.9.*(b)* ábra).

Az MFQ-ütemezőket a következő szempontok alapján lehet osztályokba sorolni:

- Hány sorban várakoznak a futásra kész folyamatok?

- Milyen ütemezési algoritmus(oka)t használunk az egyes sorokon belül?
- Hogyan kaphat nagyobb prioritást egy folyamat?
- Mikor csökken egy folyamat prioritása?
- Hová lépnek be az elindított folyamatok?



3.9. ábra. ábra - Többszintű ütemezés (a) statikus (b) visszacsatolt többszintű sorok

3.3.2.4. Az ütemezési algoritmusok „jóságának” értékelése

Egy mérnök számára a „mi van benne”, „hogyan működik” és „miért így” kérdések mellett soha nem kerülhető el annak megválaszolása sem, hogyha több lehetőség közül választhatunk, akkor „melyik a jobb”, illetve a meglevő „hogyan tehető jobbá”. Ez utóbbi kérdések megválaszolása a legtöbb esetben igen bonyolult, vagy egyenesen lehetetlen. Hiszen láttuk korábban, hogy a „jóság” követelményei összetettek, sőt egymásnak ellentmondó elemeket tartalmaznak. Vagyis nem lehet általánosságban egyértelmű választ adni. A konkrét esetekben azonban – alapos elemzés után – sokszor megadhatók a prioritások, vagyis az egyes elvárások fontossága. Így már közelebb kerülhetünk ahhoz, hogy megadjuk, hogy egy-egy konkrét esetben mit tekintünk „jónak”, „optimális” működésnek.

Példaként az időosztásos, interaktív rendszereket említjük, ahol az elfogadhatóan kis válaszidő követelménye az egyik legfontosabb az elvárások között. Más rendszerek esetében ez a követelmény nem számít fontos tervezési szempontnak. Vagyis a jóság kérdését mindig a céllal szoros összefüggésben kell vizsgálnunk.

Az ütemezési algoritmusok jóságának mérésére többféle lehetőség van.

Analitikus vizsgálat

A **determinisztikus modellezés** analitikus eszközt kínál az algoritmusok viselkedésének elemzésére. Sajnos a rendszerek terhelése számos véletlenszerű paraméterrel jellemezhető, így a determinisztikus modellek inkább esetek elemzésére, mint jósági jellemzők számítására alkalmasak. Előre összeállított adatokból – folyamatok száma, löketidők hossza – kiindulva, és az algoritmusokat a **sorbanállás elmélet** és **sztochasztikus modellezés** segítségével kezelve lehet következtetéseket levonni, azonban bármennyire is egyszerű a vizsgálati módszer, a kapott eredmények nem általános érvényűek, hanem csak szűk körre lesznek igazak.

Szimuláció

Az ütemezési algoritmusokat számítógépes modell segítségével is vizsgálhatjuk, mely során korábbi tapasztalatok vagy mérések alapján meghatározott eloszlású véletlen számokkal, illetve konkrét mért számértékekkel jellemezhetjük a folyamatokat.

A nyert eredmények érvényessége a modell és az adatok helyességétől, valamint az elvégzett szimulációk mennyiségétől függ.

Implementáció

Elvileg lehetőségünk van az algoritmusok implementálására és így teljesítményük *valós körülmények közötti mérésére*. Ez a módszer a legmegbízhatóbb a felsoroltak közül, azonban értelemszerűen egyben a legköltségesebb is, így a gyakorlatban nem igen használják.

3.3.3. Ütemezés többprocesszoros rendszerekben

Az előző részben az egyprocesszoros rendszerekben történő ütemezés kérdéseivel foglalkoztunk. Napjainkban azonban egyre gyakoribbak az olyan, több processzort tartalmazó – szorosan csatolt – rendszerek, ahol igényként merül fel, hogy a futásra kész folyamatok a rendszer bármelyik szabad processzorán elindulhassanak, és így a feladatok elvégzése minél rövidebb időt igényeljen, illetve a rendszeren belül egyenletes terheléselosztást biztosítsunk a processzorok között.

A többprocesszoros rendszerekben történő processzorütemezés értelemszerűen bonyolultabb feladat az egyprocesszoros esetnél. Az előzőekhez hasonlóan itt sem létezik „legjobb” megoldás, azonban a korábban tárgyalt elvek és sajátságok az esetek egy részében itt is alapvetően igazak lesznek.

Különbséget kell tennünk heterogén és homogén rendszerek processzor-ütemezése között. A **heterogén rendszerekre** az jellemző, hogy a rendszerbe épített CPU-k különbözőek lehetnek nemcsak méret, sebesség,

felépítés, funkció stb. tekintetében, hanem például az utasításkészlet szempontjából is. Ilyen esetekben a lefordított folyamatok kötötten, csak a nekik megfelelő gépi utasításkészlettel rendelkező processzoro(ko)n lesznek képesek futni.

Hasonlóképpen csak bizonyos processzorokon indíthatók el azok a folyamatok, amelyek valamilyen speciális, az adott processzorhoz belső buszon keresztül csatlakoztatott eszköz használatát igénylik. Ez mind homogén, mind pedig heterogén rendszereknél kötöttséget jelent.

A **homogén rendszereknél**, tehát ahol a beépített CPU-k funkcionalitás szempontjából egyformák, általában nagyobb szabadságunk van az ütemezésnél. A futásra kész folyamatok a rendszer bármelyik szabad processzorán elindulhatnak. Alapvető cél a terheléselosztás biztosítása, tehát ahelyett, hogy a futásra kész folyamatok CPU-khoz rendelt külön sorokban várakoznának (és így könnyen előfordulhatna, hogy egy CPU tétlenül áll, míg egy másik futásra kész sorában több folyamat is várakozik), célszerű közös várakozási sort (sorokat) létrehozni, ahonnan mindegyik processzor veheti a rajta következőként futó folyamatot.

A közös várakozási sor(ok)on alapvetően kétféle ütemezési megközelítést alkalmazhatunk**. Szimmetrikus multiprocesszoros rendszerekben** minden egyes CPU saját külön ütemező algoritmust futtat, amely a közös sorból választ. A sorok hibamentes megosztott használatához pedig biztosítani kell a hozzáférésre vonatkozó kölcsönös kizárást.

Az **aszimmetrikus multiprocesszoros rendszerek** megkerülik ezt a problémát azzal, hogy egyetlen ütemező fut egy kiválasztott processzoron és ez az egyetlen ütemező osztja szét a feladatokat a szabad CPU-k között. Értelemszerűen ez a *master-slave* felépítés sokkal egyszerűbb adathozzáférést eredményez.

3.4. Tárkezelés

A multiprogramozott rendszerekben a CPU-t több folyamat *megosztottan* használja. Ahhoz, hogy megfelelő működési sebességet tudjunk biztosítani, egyszerre több folyamatot is a **központi tárban (main storage, memory)** kell tartanunk, hiszen, mint korábban láttuk, a háttértárak elérési ideje – gyorsító tár alkalmazása ellenére is – sokkal nagyobb a főtárénál, így nagyon lassú lenne, ha környezetváltásnál a háttértárról kellene behozni, illetve a háttértárra kellene kivinni a folyamatokat. Vagyis a központi tár igen fontos erőforrás, melyért több folyamat verseng, és szervezése, kezelése az operációs rendszerek tervezését, megvalósítását és teljesítményét befolyásoló egyik legfontosabb tényező.

3.4.1. A főtár megosztása a folyamatok között

A korábban tárgyalt tárhierarchia és általános társzervezési elvek folytatásaként ebben a részben a „klasszikus" tárkezelés két legfontosabb kérdését, a program címeinek kötését és a tárallokációt vizsgáljuk meg részleteiben.

3.4.1.1. A program címeinek kötése

A CPU közvetlenül csak a központi tárhoz tud hozzáférni, így a folyamatok aktuálisan végrehajtandó utasításának és az operandusoknak az operatív tárban kell elhelyezkedniük. Ez azt jelenti, hogy a programokat,

adatokat végrehajtás előtt a másodlagos (háttér) tárakból be kell tölteni a központi (fizikai) memória valamilyen címére.

Egy programhoz általában lineáris, folytonos címtartományt szoktunk képzelni, amely 0-tól kezdődően (a program eleje) a program maximális címéig terjed. Ezt hívjuk **logikai címtartománynak (logical address space)**. A mai rendszerekben a programok végrehajtása gyakorlatilag soha nem a 0. címtől kezdve, azaz a logikaival egyező **fizikai címtartományban (physical address space)** történik. Egy program a megírása (ahol értelemszerűen a logikai címtartományban vagy szimbolikus elnevezésekben gondolkodunk) és a végrehajtása (itt viszont a konkrét fizikai szó vagy byte címekre van szükség) között különböző fázisokon mehet keresztül (fordítás, kapcsolatszerkesztés, betöltés stb.). Az egyes lépések alatt a címek különböző módon lehetnek reprezentálva.

Egy-egy program lefordításakor, szerkesztésekor a fordító és a kapcsolatszerkesztő program első menetben általában a logikai címtartományban ad értékeket a szimbolikus hivatkozásoknak. Ugyanakkor a végrehajtáshoz már a fizikai címekre van szükség. A **logikai és fizikai címtartomány közötti megfeleltetés, leképezés (mapping)** elvileg bármelyik lépés alatt elvégezhető, de valamikor biztosan meg kell tenni. Nézzük meg tehát, hogy mikor (a program előkészítésének melyik fázisában) milyen lehetőségeink vannak a leképezés elvégzésére (3.10. ábra).

3.10. ábra. ábra - Logikai-fizikai címtranszformáció a felhasználói programok többlépcsős feldolgozása során

Statikus logikai-fizikai címleképezés

A címkonverzió történhet **fordítás** közben **(compile time)**. Ha ismeretesek a program fizikai címei, akkor a leképezés elvégezhető a fordítás alatt. Ha a későbbiek során a program (fizikai) kezdőcíme megváltozik, akkor azt újra le kell fordítani. Éppen ezért, ezt a megoldást – merevsége miatt – csak speciális esetekben, a ROM (Read Only Memory, csak olvasható tár) memóriába kerülő programok esetén használják, egyébként pedig nem a végleges fizikai cím, hanem logikai cím rendelődik a tárgykódban a hivatkozásokhoz.

A végleges fizikai címek kötésére a következő lehetőség **szerkesztés** közben adódik **(link time)**. Szerkesztésre akkor van szükség, ha egy program több, egymástól függetlenül lefordított modulból áll, amelyek hivatkoznak más modulban definiált logikai címre. A kapcsolatszerkesztő (linker) program feladata a modulok közötti kereszthivatkozások feloldása, és a modulok egymás mögé helyezése. Ennek eredményeképpen a tárgykódban a logikai címek helyére fizikai címek helyettesítődhetnek, azonban az esetek többségében csak a modulok logikai címtartományainak egyesítése történik meg, és egy **áthelyezhető kódot (relocatable code)** kapunk eredményül.

Ha a végleges címkonverzió **betöltés** közben **(load time)** történik, akkor a korábban kapott áthelyezhető kód címhivatkozásait módosítja a betöltő program (loader) az aktuális címkiosztás szerint.

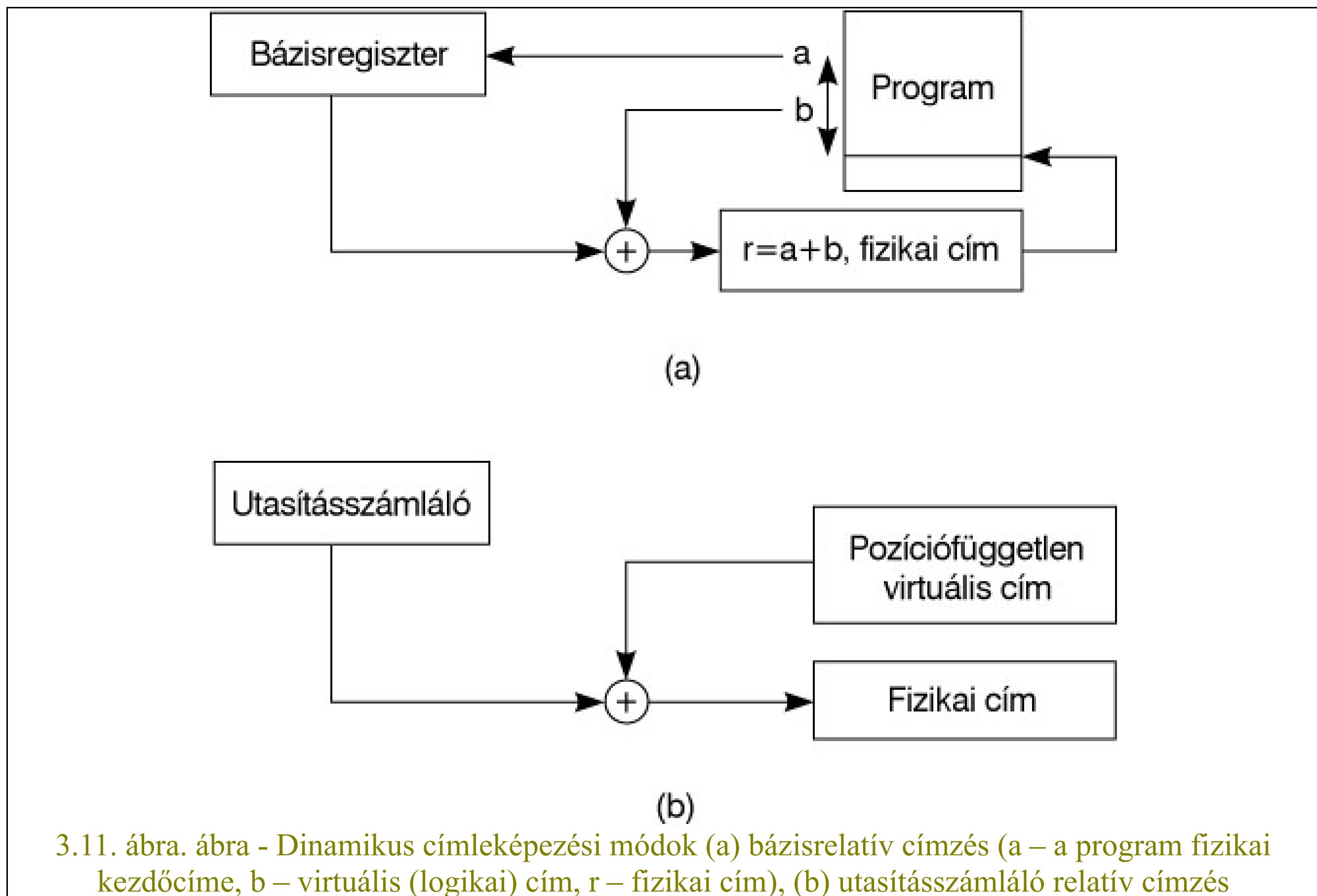
Az eddig felsorolt megoldások ún. **statikus leképezést** valósítanak meg, azaz a fizikai címek hozzárendelése a program élete során egyszer, végrehajtásuk előtt következik be, és az értékek többé nem változnak.

Dinamikus logikai-fizikai címleképezés

A tárgazdálkodás szempontjából igen előnyös lehet, ha a program módosítás nélkül futtatható tetszőleges szabad tárterületre töltve, és helye akár végrehajtás közben is megváltoztatható. Ilyenkor **dinamikus címleképezésről** és dinamikus áthelyezhetőségről beszélünk. Dinamikus címleképezésnél a program memóriaképe logikai címeket tartalmaz, és a konkrét fizikai cím-hozzárendelés csak az **utasítás-végrehajtás** során következik be (speciális hardverelemek segítségével).

A legegyszerűbb dinamikus címleképezést a **bázisrelatív címzés** valósítja meg. Ha a program valamennyi címhivatkozása bázisrelatív címzési módú, a program tetszőleges helyre betölthető, a bázisregisztert a betöltési kezdőcímre állítva a program végrehajtható (3.11.*(a)* ábra). Ekkor annak sincs akadálya, hogy a program végrehajtásának megszakadása után a teljes programot egy új kezdőcímre töltsük vissza, vagy másoljuk át, és így folytatódjék a végrehajtás.

A bázisrelatív címzéshez igen hasonló (akár annak alesetének is tekinthető) dinamikus címleképezési mód az utasításszámláló-relatív címzés, melynek eredményeképpen **pozíciófüggetlen kódot (PIC: Position Independent Code)** kapunk, amelyet ugyancsak végrehajthatunk tetszőleges címre töltve (3.11.*(b)* ábra).



3.11. ábra. ábra - Dinamikus címleképezési módok (a) bázisrelatív címzés (a – a program fizikai kezdőcíme, b – virtuális (logikai) cím, r – fizikai cím), (b) utasításszámláló relatív címzés

Késleltetett betöltés

Jobb tárkihasználás érhető el **késleltetett betöltés** alkalmazásával, vagyis ha a programok indításakor nem a teljes programot, hanem csak egyes részeit töltjük a tárba, míg más részek futás közben, szükség esetén töltődnek be. Az ide sorolható módszerek az esetek többségében dinamikus címleképezést igényelnek.

Dinamikus betöltésnek (dynamic loading) hívjuk azt a módszert, ha a programhoz tartozó egyes eljárások – áthelyezhető formában – a háttértáron vannak, és csak a meghívásukkor töltődnek be. Ilyenkor a futás elindításakor csak a főprogram kerül a memóriába, majd egy-egy eljárás meghívásakor először azt kell ellenőrizni, hogy a meghívott eljárás a memóriában van-e. Ha nem, akkor egy speciális programrész betölti a kívánt rutint, majd átadja a vezérlést az eljárásnak (3.12.*(a)* ábra).

A módszer előnye, hogy a programok összméretüknél általában lényegesen kisebb tárterületet igényelnek, a ténylegesen meg nem hívott eljárások soha nem töltődnek be. (A program részei között sokszor találunk olyan eljárásokat – például hibakezelő rutinokat – melyekre csak ritkán van szükség, így felesleges azokat minden esetben betölteni a memóriába.)

A dinamikus betöltést a programozónak magának kell megszerveznie, a megvalósításhoz az operációs rendszer nem, legfeljebb a nyelvekhez tartozó fordító, szerkesztő és futtató programok adnak támogatást.

A késleltetett betöltés következő, az előbbi módszer módosított – az operációs rendszer által is támogatott – változata a **dinamikus könyvtárbetöltés** vagy **dinamikus kapcsolatszerkesztés (dynamic linking).** A programban használt rendszerkönyvtárak eljárásai helyett a szerkesztő csak egy **csonkot (stub)** tesz be a programba, mely valamilyen hivatkozást tartalmaz egy könyvtárra és ezen belül az adott eljárásra. A csonk első meghívásakor a rendszer a kívánt eljárást betölti a tárba, majd az eljárás fizikai kezdőcímét a hívó eljárásban szereplő logikai címhez rendeli. Az ezután következő hívások már a betöltött eljárást fogják – időveszteség nélkül – meghívni (3.12.*(b)* ábra). (A könyvtárakat természetesen pozíciófüggetlenül kell kódolni.)

A módszer előnye a csökkentett tárfelhasználáson kívül az is, hogy a könyvtárak módosítása (hibajavítás, újabb, javított változatra cserélés) esetén nem szükséges az összes, a módosított könyvtárat használó programot újrafordítani. Az azonban előfordulhat módosított könyvtárak esetén, hogy a memóriába ugyanannak az eljárásnak különböző verziói (egymástól eltérő példányai) töltődnek be, attól függően, hogy a futó folyamatok módosítás előtt vagy után hívták meg az eljárást. Működési problémát okozhat, ha a változtatások a korábbival *inkompatíbilis* új verziót eredményeznek.

A dinamikus kapcsolatszerkesztést alkalmazó korszerű rendszerek annyiban különböznek a fenti sémától, hogy a betöltött eljárások más folyamatok számára is rendelkezésre állnak, azaz a csonk első meghívásakor a rendszer figyelembe veszi azt is, ha más folyamat már meghívta az eljárást, és így a kívánt eljárás már a tárba töltődött. Ilyenkor ez a fizikai cím rendelődik a hívó eljárás logikai címéhez. A megoldás felveti az újrahívhatóság problémáját (amikor több folyamat egyidőben is végrehajthatja ugyanazt az eljárást), melyről a fejezetben később részletesebben is szó lesz.

A központi tár fizikai méreténél nagyobb méretű programok írását és futtatását teszi lehetővé az **átfedő programrészek (overlay)** technikája. A módszer kifejlesztésénél abból a felismerésből indultak ki, hogy a programoknak csak egy részét teszik ki azok a programrészek, illetve adatok, amelyekre a teljes futási idő alatt szükség van, így elegendő ezeket állandóan a memóriában tartani. A programok fennmaradó része pedig „szétvágható” olyan részekre, amelyek időben elkülönülten – azaz egyidőben csak az egyik – dolgoznak. Ezeket szükség esetén egyesével töltjük be, majd használatuk után újra a háttértárba menthetjük (ezt sok rendszer nem teszi meg), hogy a felszabaduló területre egy másik overlay részt tölthessünk (3.12.*(c)* ábra). (A módszer tehát tulajdonképpen annyiban különbözik a dinamikus betöltéstől, hogy itt a betöltött részek nem maradnak bent a memóriában állandó jelleggel a további futás idejére.)

Az átfedés számára fenntartott tárterület a legnagyobb programrészlet hosszával egyezik meg. Az overlay technika szempontjából az átfedő programrészek a háttértáron tartalmazhatnak abszolút címeket is (így a módszer nem igényel feltétlenül dinamikus címleképezést), hiszen ugyanarra az ismert kezdőcímre töltődnek be.

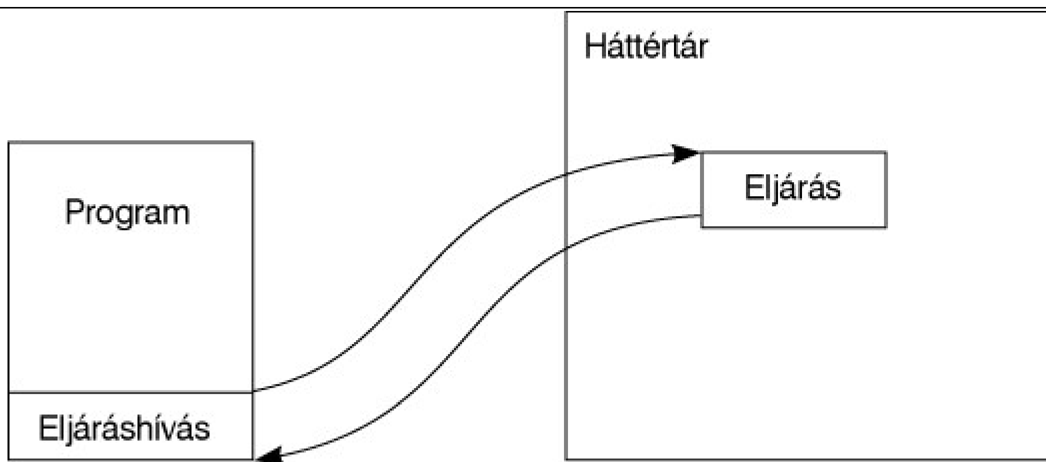
A dinamikus betöltéshez hasonlóan ez a módszer sem igényel különleges támogatást az operációs rendszertől. Egyszerű fájlrendszer alkalmazásával megoldható az átfedő programrészek memóriába olvasása, illetve háttértárra írása, azonban az overlay részek tervezése a program szerkezetének és futás közbeni viselkedésének alapos ismeretét igénylik a programozótól, ezért manapság ritkán használják, csak ott, ahol a fizikai memória

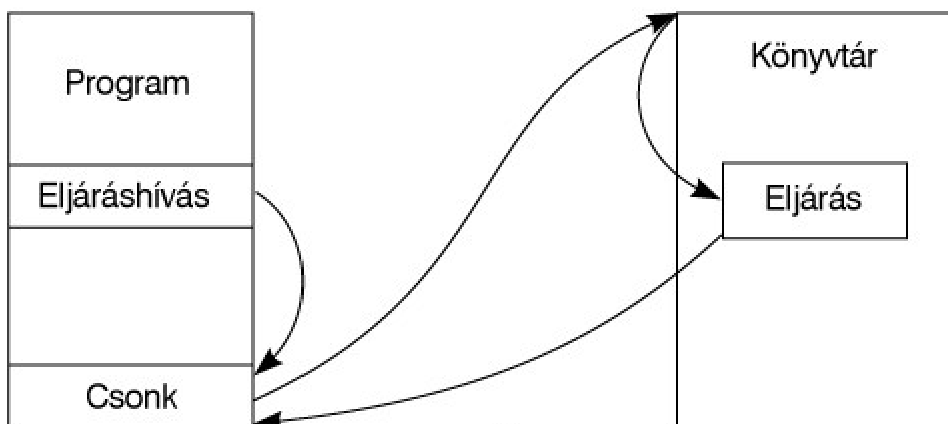
mérete ezt szükségessé teszi, és más, korszerűbb automatikus módszerek (például virtuális tárkezelés) nem valósíthatók meg.

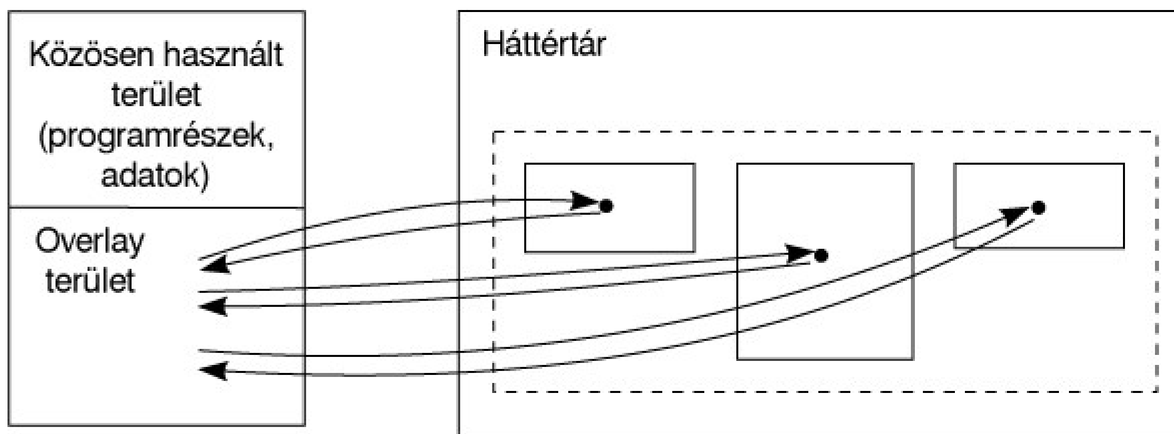
3.4.1.2. Társzervezési módszerek

A számítógépek történetének korai időszakában a memória viszonylag drága erőforrásnak számított. Ezért a rendszertervezők komoly erőfeszítéseket tettek a központi tár használatának optimalizálására. A mai tárak ára egyre alacsonyabb, ezért a „gazdaságos" tárkihasználás némileg mást jelent, mint korábban. (A „gazdaságosság" mögött húzódó *költségfüggvényben* a hangsúlyok eltolódtak a mai élet kihívásai által diktált követelmények – például felhasználói kényelem, nagy komplexitású, beágyazott rendszerek összetett feladatai, valósidejű működés – felé.)

Társzervezés alatt azt a módot értjük, ahogyan a központi tárat a felhasználók (folyamatok) között megosztjuk. Az „optimális" tárhasználatra való törekvés során felmerült kérdések közül az első megválaszolandó mindjárt az volt, hogy egyszerre csak egyetlen, vagy több felhasználó is használhassa-e a főmemóriát. Később az vetődött fel, hogy ha egy időben párhuzamosan több felhasználó között osztjuk meg a memóriát, akkor vajon mindegyikhez ugyanakkora területet rendeljünk-e, vagy különböző méretű részekre, **partíciókra (partition)** osszuk azt. És ezeknek a partícióknak a hossza állandó vagy pedig változó legyen-e? Eldöntendő az is, hogy a programok a (felhasználói) memórián belül bárhol, vagy csak bizonyos részeken futhatnak-e, továbbá, hogy a folyamatokhoz rendelt fizikai memória területnek egybefüggőnek kell-e lennie, vagy össze nem függő részekből is állhat.







3.12. ábra. ábra - Késleltetett betöltési módok (a) dinamikus betöltés (b) dinamikus könyvtárbetöltés (c) átfedő programrészek

A fenti kérdésekkel is érzékeltethető, egyre táguló lehetőségek a történeti fejlődés során nyíltak meg. Mindegyik változatra találhatunk működő rendszert, ezért ennek a pontnak további részében azt tekintjük át, hogy ezek milyen sajátosságokkal rendelkeznek, és hogyan is lehet megvalósítani őket.

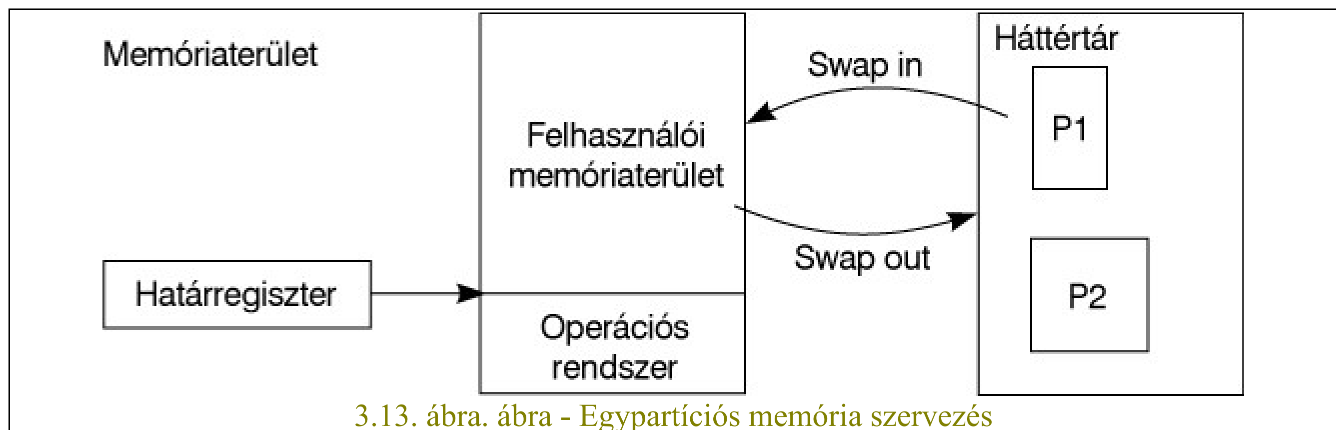
Egypartíciós rendszer

Az **egypartíciós rendszereknél** egyidőben egyszerre egyetlen folyamat használhatja a központi tárat, így az operációs rendszeren – és a speciális tárterületeken, mint például a megszakításvektorok, a periféria-címtartományok – felüli folytonos címtartományon teljes egészében ez a folyamat futhat.

A betöltő a program indításakor azt az első szabad címre hozza be. Ha a program futása során az operációs rendszernek több tárra van szüksége, akkor azt vagy a program által nem használt területről – a tár másik végéről – „lopja" el, vagy a programot kell áthelyezni, ami hardvertámogatás nélkül nehézkes.

Az operációs rendszer védelmére elegendő egyetlen regiszter, amely a program legkisebb címét tartalmazza. A folyamat futása közben – felhasználói módban – a tárkezelő hardver minden egyes kiadott memóriacímet összehasonlít a határregiszter értékével, és ellenőrzi, hogy minden hivatkozás a tárolt cím felett legyen. Rendszerhíváskor a processzor olyan működési módba kerül át (rendszer vagy kernel mód), ahol ez a védelem kikapcsol, és az operációs rendszer a teljes tárat eléri. A határregiszter értéke csak rendszermódban változtatható meg.

A folyamatok váltása a központi tár és a háttértár közötti a későbbiekben részletezett tárcsere (swapping) segítségével történhet.



3.13. ábra. ábra - Egypartíciós memória szervezés

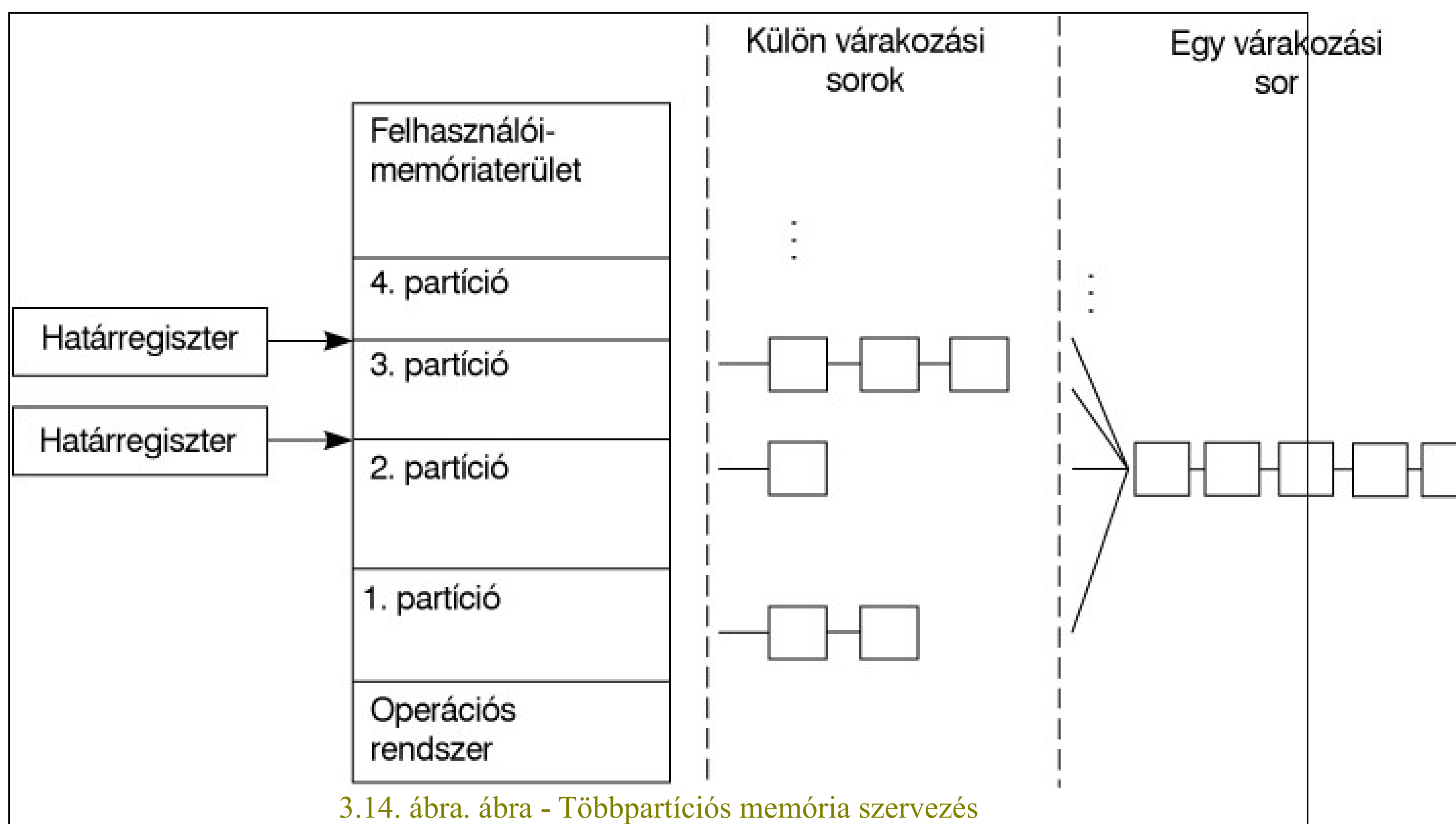
Többpartíciós rendszer

Bár swapping segítségével egypartíciós rendszerekben is megvalósítható valamiféle multiprogramozás, azonban ez a mágneslemezes átvitel nagy időigénye miatt jelentősen megnöveli a környezetváltási időt, így legfeljebb a lemeznél még sokkal lassabb periferiális műveletek esetén, vagy a felhasználói esélyegyenlőség biztosítása érdekében volt érdemes átkapcsolni más folyamat végrehajtására. A hatékony multiprogramozás megkövetelte, hogy egy időben egyszerre több folyamat is a tárban tartózkodjék.

Multiprogramozás rögzített partíciókkal

A korai rendszerekben a felhasználói tárterületet olyan különböző méretű partíciókra osztották, melyek mérete a rendszer működése során nem változhatott **(rögzített partíciók, fixed partition)**. A programok a méretüknek megfelelő partícióban futtathatók. Minden partícióban egyetlen folyamat tartózkodhat. Így a multiprogramozás fokát alapvetően a partíciók száma korlátozza.

Egy-egy folyamat befejeződése után az operációs rendszer egy következő folyamatot választ ki futásra a szabad partícióhoz, vagy az egyetlen közös várakozási sorból (ilyenkor nyilván figyelembe kell venni, hogy a programok csak a maximális memóriaigényüknek megfelelő méretű partícióban futtathatók), vagy pedig a már eleve ehhez a partícióhoz kötött várakozási sorból (ilyenkor a hosszú távú ütemező a belépő folyamatokat közvetlenül a különböző partíciókhoz tartozó várakozási sorokba sorolja) valamilyen – általában a legjobb illeszkedést kereső – stratégia szerint (3.14. ábra).



3.14. ábra. ábra - Többpartíciós memória szervezés

Ez a tárkezelési mód nagyon egyszerű, ugyanakkor nagyon merev, így végeredményben rossz tárkihasználást biztosít. A folyamatok nem használják ki teljesen a partícióban rendelkezésükre álló tárterületet, minden partícióban a benne futó folyamathoz rendelt, ám kihasználatlan memóriaterület, „lyuk” marad. Ezt hívjuk **belső tördelődésnek (internal fragmentation)**.

Multiprogramozás változó méretű partíciókkal

Fejlettebb operációs rendszerekben a folyamatok – igényeiknek megfelelő – **változó méretű partíciót (variable partition)** kapnak. Így a partíciókon belül nincs kihasználatlan memóriaterület.

A belépő folyamatok számára az ütemező egy kellően nagy, folyamatos, szabad memóriaterületet keres, melyet két részre oszt. A szükséges nagyságú területet a folyamathoz rendeli, a maradék pedig továbbra is szabad marad. Ezzel sikerül kivédeni a belső tördelődést.

Változó méretű partíciók használata esetén a problémák akkor kezdődnek, amikor folyamatok befejezik a futásukat és az operációs rendszer felszabadítja az általuk foglalt területet. Így lyukak keletkeznek az allokált memória- részek között. Ezek ugyan hozzárendelhetők más folyamatokhoz, azonban szinte biztos, hogy előbb-utóbb a folyamatokhoz rendelt tárterületek között akkora szabad területek maradnak, melyek egyenként túl kicsik a folyamatok számára, és így kihasználatlanok maradnak. Ezt a jelenséget hívjuk **külső tördelődésnek (external fregmentation)**.

A külső tördelődés, ha meghalad egy bizonyos mértéket, jelentősen késleltetheti újabb folyamat elindítását, mindamellett, hogy *összességében* elegendő szabad terület lenne ehhez. Azonban ezek a területek nem alkotnak folytonos címtartományt. Ezen segíthet a **szabad helyek tömörítése (compaction, garbage collection)**, azaz a tár egyik végére rendezése. Ilyenkor az operációs rendszer úgy helyezi át a partíciókat, hogy azok érintkezzenek egymással, és így összefüggő szabad területet kapjunk.

A módszer ugyan megoldja a problémát, azonban nagyon időigényes, hardver támogatást (dinamikus áthelyezhetőséget) igényel, megzavarhatja a rendszer működését (például egy interaktív rendszer válaszideje váratlanul jelentősen megnövekedhet), és a tárterületek mozgatásához az áthelyezési információkat meg kell őrizni.

Mindezek miatt nem egyértelmű, hogy érdemes-e alkalmazni. Összességében jobban járhatunk, ha ehelyett inkább várakozunk, amíg más futó folyamatok is befejeződnek, vagy pedig a tárterület lefoglalása során próbáljuk optimalizálni a kihasználást.

A külső tördelődés csökkentésére különböző stratégiákat dolgoztak ki a folyamatokhoz rendelendő tárterület kiválasztására. Az operációs rendszer a betöltendő program számára a szabad területek közül az alábbiak szerint választhat.

Az **első megfelelő (first fit)** algoritmus a tár elejéről indulva az első elegendően nagy területet foglalja le a folyamat számára. Az algoritmus igen gyors, eredményeképpen a memória mintegy 30%-a marad kihasználatlan. (Ezt hívják **50%-os szabálynak**, mivel a folyamatok által lefoglalt területek összegének mintegy fele marad kihasználatlan.)

A **következő megfelelő (next fit)** algoritmus annyiban különbözik az előzőtől, hogy a keresést nem a tár elején, hanem az utoljára lefoglalt tartomány végétől kezdi. Hatásfoka hasonló az előzőéhez.

A **legjobban megfelelő (best fit) algoritmus** a legkisebb, még elegendő méretű területet foglalja le. A mögöttes filozófia szerint, minél kisebbek a megmaradó lyukak, összességében annál kevesebb lesz a kihasználatlan memóriaterület. A módszer az előzőeknél lassabb, és bár első pillanatra hihetetlennek tűnik, de némileg rosszabb memória kihasználást eredményez az előzőeknél.

A **legrosszabban illeszkedő (worst fit)** algoritmus az előzővel épp ellentétben, a legnagyobb szabad területből foglal, mert abban bízik, hogy a megmaradó „nagy” lyuk elegendően nagy lesz egy további folyamat számára. Szimulációs vizsgálatok alapján azt mondhatjuk, hogy ez a legrosszabb hatásfokú algoritmus (a memória terület kb. 50%-a marad kihasználatlan).

A statisztikai vizsgálatok eredményeképpen tehát megállapíthatjuk, hogy hatásfokát tekintve az első három algoritmus nagyjából azonos kihasználtságot eredményez. Figyelembe véve, hogy az első megfelelő algoritmus megvalósítása a legegyszerűbb és ez működik a leggyorsabban, mindenképpen ez tűnik a legjobbnak.

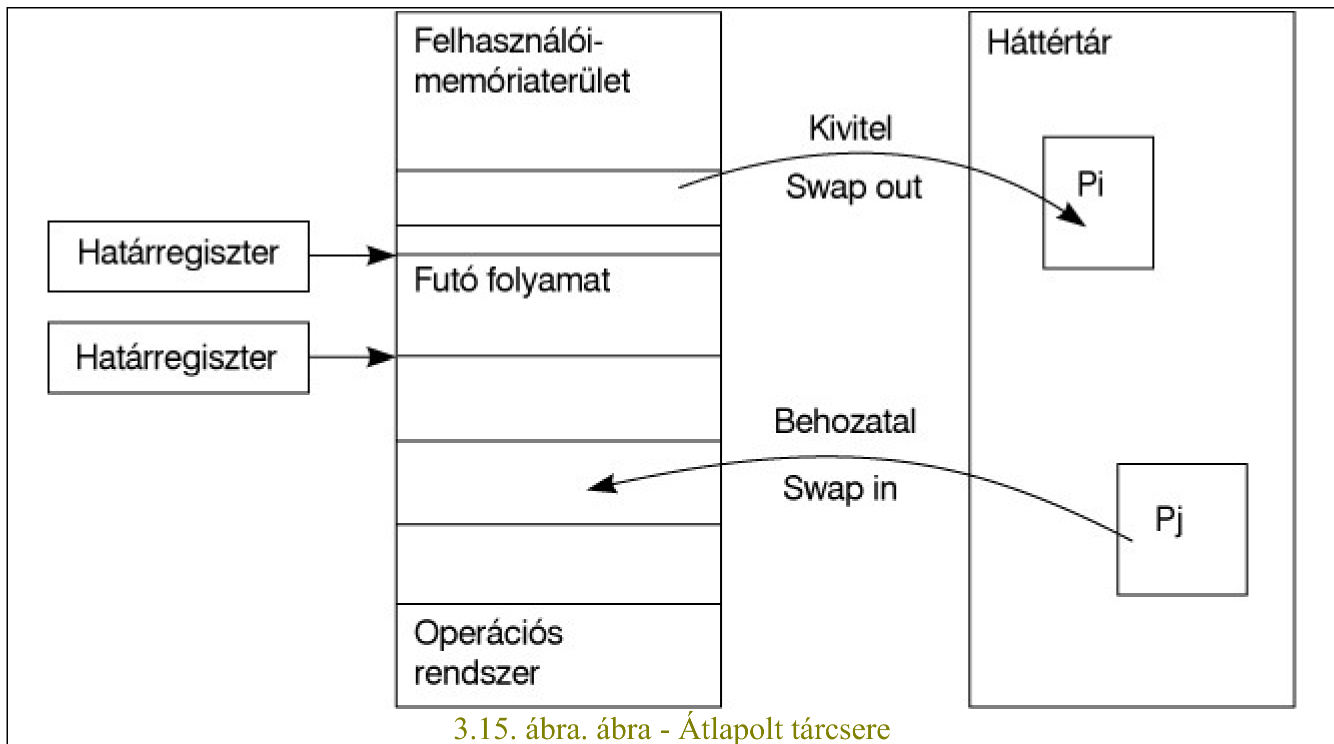
A többpartíciós rendszerek megjelenésével nemcsak az operációs rendszert, hanem más folyamatok tárterületét is védeni kell a hibás (vagy rosszindulatú) folyamatok működésének következményeitől. Ezért a védelem két határregisztert használ, amelyekbe a folyamat futásakor, a hozzá tartozó címtartomány határait tölthetjük. Az egyes címzéseknél a hardver a tartományból való kicímzést figyeli.

Tárcsere

A folyamatoknak a háttértárról a memóriába való bevitelét, illetve a memóriából a háttértárra való kivitelét a **tárcsere (swapping)** technika segítségével oldhatjuk meg. Ennek során az operációs rendszer egy-egy folyamat teljes tárterületét a háttértárra másolja, így szabadítva fel a területet más folyamatok számára. Ehhez természetesen az operációs rendszernek pontosan ismernie kell a folyamatok aktuális tárigényét.

A tárcsere a perifériás átvitel miatt időigényes – egyszerre nagy tárterületet kell mozgatni –, így jóval hosszabb időt vesz igénybe, mint egy szokásos környezetváltás. A tárcserével töltött idő jelentősen megnövelheti a futás idejét, ezért mindenképpen célszerű a tárcsere idejét viszonylag alacsonyan tartani az effektív futási időkhöz képest (például Round–Robin-ütemezést alkalmazó rendszer tervezésekor az időszelet hosszának megválasztásánál figyelembe kell venni a swapping idejét és arányát), valamint lehetőség szerint minimalizálni kell a tárcserék számát és optimalizálni magát a műveletet.

Az ütemezőnek célszerű a tárban levő futásra kész folyamatok közül – ha van ilyen – választani. A tárterület elmentésénél pedig figyelhetünk arra is, hogy egy változatlan, a háttértáron meglevő tartományt nem kell újra kiírni. További gyorsítást tesz lehetővé az **átlapolt tárcsere (overlapped swapping)**, vagyis amikor az éppen futó folyamat végrehajtásával párhuzamosan történik egy másik folyamat kivitele, illetve egy harmadik behozatala (3.15. ábra).



3.15. ábra. ábra - Átlapolt tárcsere

Tárcserét korszerű rendszerek a várhatóan hosszabb ideig várakozni kényszerülő folyamatoknál, vagy a rendszer túlterheltségét (például feladatok, azonos igények torlódása) megelőzendő alkalmaznak. Tárcserére lehet szükség új folyamat létrehozása *(fork)* esetén (ha nincs elég hely a szülő és a gyerek számára), ha egy folyamat tárigénye – például a verem mérete – megnő, illetve ha egy másik – nagyobb prioritású – folyamatnak van szüksége a memóriaterületre.

Ha szükség van további memóriaterületre, akkor el kell dönteni, melyik folyamat legyen az áldozat. Ehhez figyelembe vehetjük a folyamatok állapotát, prioritását, futási, illetve várakozási idejét stb. Behozatalnál pedig azt kell megválaszolni, hogy a háttértáron levők közül mikor és kit hozzunk be, szintén a fenti szempontok esetleges figyelembevételével. Mindkét esetben figyelnünk kell azonban arra, hogy a folyamatokat feleslegesen ne pakolgassuk, valamint, hogy a választási algoritmus ne vezessen éhezésre.

Statikus logikai-fizikai címleképezésnél egy további megkötést jelent, hogy a háttértárra kivitt folyamatokat ugyanarra a memóriaterületre kell behozni, amelyet korábban elfoglaltak. Dinamikus – futás közbeni – címtranszformáció alkalmazása esetén a folyamatok minden további nélkül betölthetők az előzőtől eltérő memóriaterületre.

Az eddigi társzervezési megoldásoknál abból a feltételezésből indultunk ki, hogy a folyamatok fizikailag is folytonos memóriaterületet használnak. A korszerű tárkezelő hardverek azonban lehetővé teszik, hogy a folyamatok folytonos logikai címtartományának nemfolytonos fizikai címtartományt feleltessünk meg [ezt hívjuk **mesterséges folytonosságnak (artificial continuity)**]. Ezzel megszüntethető a külső, és jelentősen csökkenthető a belső tördelődés.

A tárcserék ideje erősen függ az átvitt információ mennyiségétől, azaz az átvitt memóriaterület nagyságától. Ezért a tárcsere idő lecsökkentésének kézenfekvő módjaként kinálkozik, hogy ne a teljes folyamatokat, hanem csak egy részüket kelljen mozgatni a memória és a háttértár között. Ehhez az igazi segítséget azok a speciális hardverelemek jelentették, amelyek lehetővé tették, hogy a futó folyamatnak ne a teljes címtartománya, hanem annak csak egy része tartózkodjék a központi tárban. Ilyenkor előfordulhatnak olyan hivatkozások, amelyekhez nem tartozik érvényes fizikai cím, mivel a hivatkozott cím nincs a memóriában. Megfelelő hardver-szoftver összjáték esetén azonban a rendszer képes arra, hogy a felhasználó számára észrevétlenül a szükséges programrészletet a háttértárból a központi tárba töltse, és a korábbi hivatkozást – ezúttal már sikeresen – végrehajtsa.

A következő korszerű társzervezési módszerek közös jellemzője, hogy egyrészt mesterségesen folytonos címtartományt használnak (azaz a fizikai címtartomány általában több, egymástól elkülönült memóriablokkból áll), másrészt lehetőséget adnak arra, hogy a folyamatok tárterületének csak egy része tartózkodjék egyidőben a memóriában. A szabad tárterületek rugalmas felhasználhatósága érdekében ezek a módszerek futás közbeni címleképezést alkalmaznak.

A leképezéshez szükséges többletinformációk mennyiségének elviselhető szinten tartása céljából a gyakorlati címtranszformációk tartományonként megőrzik a megfeleltetés folytonosságát. Egy-egy összefüggő logikai címtartományt **(blokk)** egy-egy önmagában összefüggő, de tetszőleges fizikai címtartományra képeznek le.

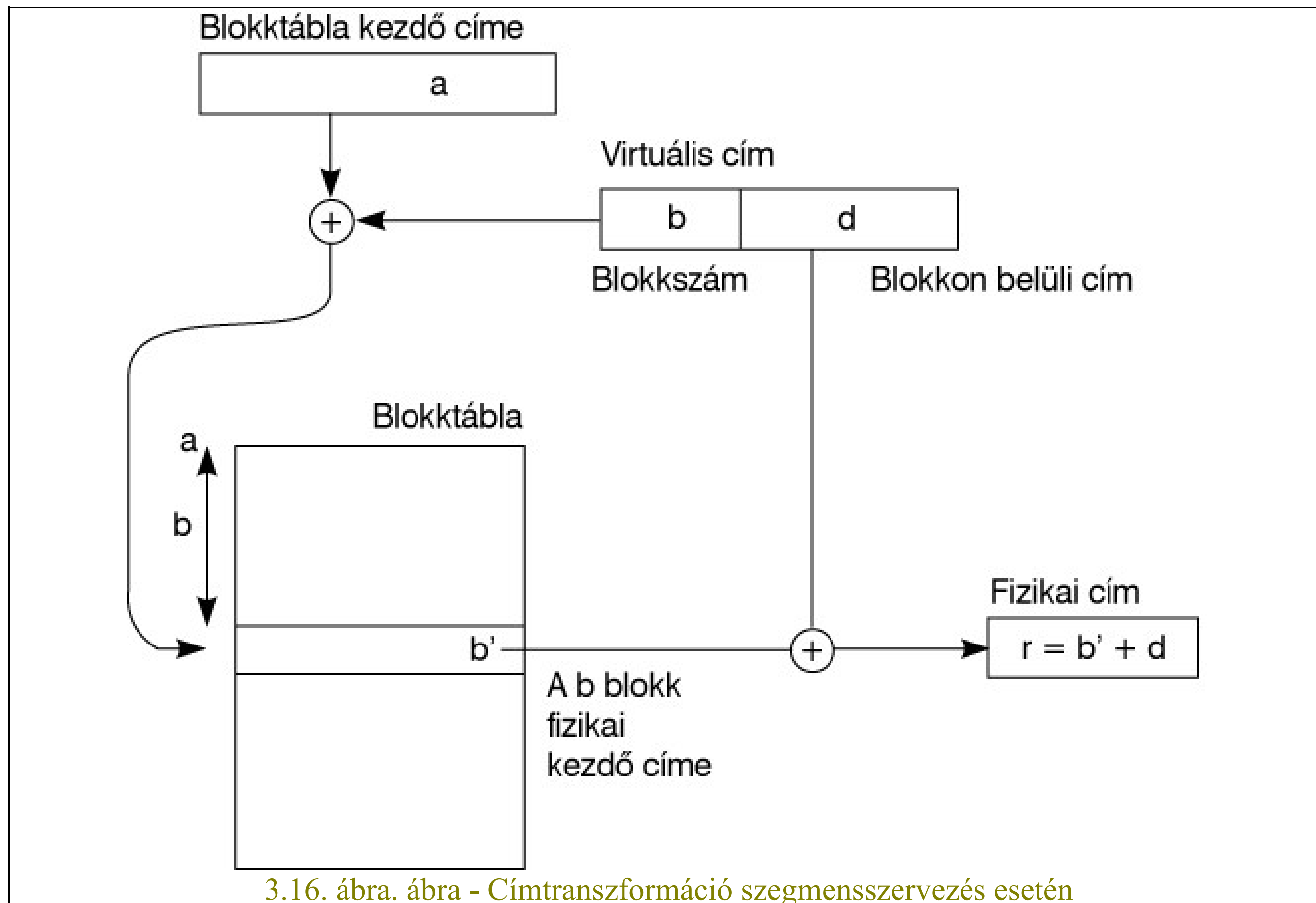
A megfelelő sebességet hardver támogatás biztosítja. Minden egyes folyamathoz tartozik egy ún. **blokktábla (block map table)**, amely a logikai cím–fizikai cím összerendeléseket tartalmazza (a folyamatonkénti külön blokktáblára azért van szükség, mert a folyamatok logikai címtartománya fedi egymást, de fizikai címtartományuk el kell különüljön). A logikai címek <b, d>, azaz <blokkcím, blokkon belüli eltolás (displacement)> formában adottak. A blokktábla minden egyes blokkhoz megadja a blokk fizikai kezdőcímét. A hardverrel támogatott címleképezés mindig egy meghatározott módon elérhető (aktuális) blokktábla használatával történik. A futó folyamathoz tartozó blokktáblát az operációs rendszer teszi aktuálissá környezetváltáskor. Szokásos megoldás, hogy a címképző egység egy mutatón (pointer) keresztül éri el a blokktáblát, mert ilyenkor az aktuálissá tétel igen gyors lehet, hiszen csak a mutató átállítását igényli.

Szegmensszervezés

A programozók a programjaikat nem egyetlen összefüggő, folyamatos logikai címtartománynak látják, hanem sokkal inkább úgy, mint több, különböző funkcionális részegység együttesét, amelyek egy-egy részfeladatot oldanak meg (például a főprogram, egy-egy eljárás, a verem stb.). A részegységek fizikai elhelyezkedése pedig közömbös a felhasználó számára.

A **szegmensszervezés (segmentation)** ezt a látásmódot tükrözi, a cím-transzformációnál a logikai **szegmenseket (segment)** felelteti meg egy-egy blokknak. Az értelemszerűen *különböző hosszúságú* blokkok

önmagukban folytonos fizikai címtartományt foglalnak el, a címképzés a blokktáblás sémát követi (3.16. ábra). Belső tördelődés nem lép fel.



3.16. ábra. ábra - Címtranszformáció szegmensszervezés esetén

A blokktáblában a szegmensek hosszát is tárolni kell. Ezzel biztosítható, hogy a folyamat ne címezhessen ki a szegmensből. A címképző hardver a szegmens területén kívülre mutató *túlcímzést* figyeli, és ha ilyen előfordul, hibamegszakítást generál (segment overflow fault).

Ahhoz, hogy a szegmentált címzési mód segítse a szegmensenkénti tárcserét is, a blokktábla egy bitje (residency bit) azt mutatja, hogy a szegmens a memóriában van-e vagy sem. Ha olyan szegmensre történik hivatkozás, amelyik nincs a tárban, ennek a bitnek az állása alapján a címképző hardver hibamegszakítást generál (missing segment fault). Természetesen a tárcserék lebonyolításához tárolni kell azt az információt is, hogy a háttértáron hol található meg a szegmens.

A szegmensszervezés lehetőséget ad arra, hogy több folyamat egy-egy logikai szegmense ugyanarra a fizikai címtartományra legyen leképezve, vagyis, hogy a folyamatok közös eljárásai egyetlen példányban tárolódjanak a memóriában **(osztott szegmenshasználat, segment sharing)**.

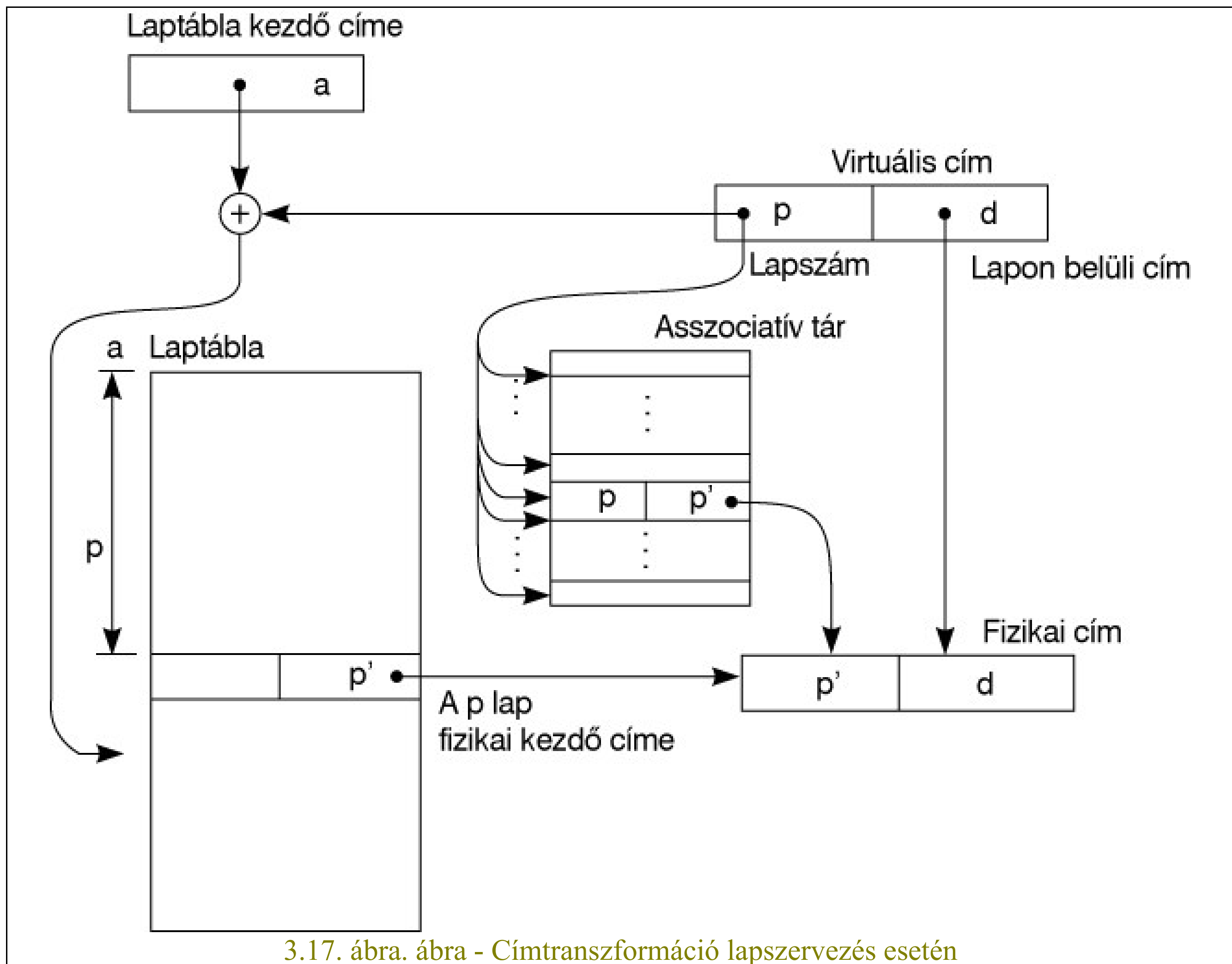
Szegmensszervezés esetén könnyen megvalósítható **a hozzáférések szabályozása (access control)**. A folyamatoknak az egyes szegmensekhez különböző hozzáférési módokat engedélyezhetünk. Ezzel a folyamat tárterületét védhetjük saját működésének hibáitól, illetve osztott szegmenshasználat esetén a különböző

folyamatoknak különböző hozzáférési jogosultságokat adhatunk. Szokásos hozzáférési jogok az **olvasási jog** (**read access**, ilyenkor a folyamat a szegmens területét olvashatja), **írási jog** (**write access**, a folyamat a szegmens területére írhat, az ott levő értékeket módosíthatja) és a **végrehajtási jog** (**execute access**, a szegmensben gépi utasítások vannak, amelyeket a folyamat végrehajthat). A módosítható adatterületek helyes osztott használatához a kölcsönös kizárást a folyamatoknak kell biztosítania, ehhez a hardver nem nyújt támogatást.

Lapszervezés

A szegmensszervezésnél használt változó méretű blokkok megoldást jelentenek a belső tördelődésre, azonban óhatatlanul fellép a külső tördelődés jelensége. Ez utóbbi megszüntetéséhez azonos, rögzített méretű blokkok és nekik megfelelő **fizikai memória keretek (frame)** használata szükséges (vagyis, hogy a folyamatok tárterületét ilyen egységekben allokáljuk). Az azonos méretű blokkokra a **lap (page)**, az ennek megfelelő társzervezésre pedig a **lapszervezés (paging)** elnevezést használjuk. A külső tördelődés megszüntetésének ára a belső tördelődés megjelenése. A folyamatok utolsó lapja átlagosan csak félig lesz tele, a tördelődési veszteség tehát átlagosan folyamatonként fél lap. A lapméretet az egyes rendszerekben egymásnak ellentmondó szempontok mérlegelése alapján választják meg. A lapméret csökkentésével csökken a belső tördelődésből származó tárveszteség. Ugyanakkor nő a laptábla mérete, és az adategységre vonatkoztatott átlagos keresési (címzési, lappangási) idő növekedése miatt csökken a memória és a háttértér közötti adatátvitel sebessége (ugyanolyan adatmennyiséget több blokkban kell átvinni).

A lap mérete gyakorlati szempontok miatt mindig 2 hatványa (általában 512 és 16 K byte között), ugyanis ez esetben a címtranszformáció második részében (a blokkon belüli eltolás figyelembevételénél) nem szükséges összeadást végezni, hanem elegendő az eltolást egyszerűen a fizikai címben a kisebb helyiértékű bitek helyére másolni (3.17. ábra).

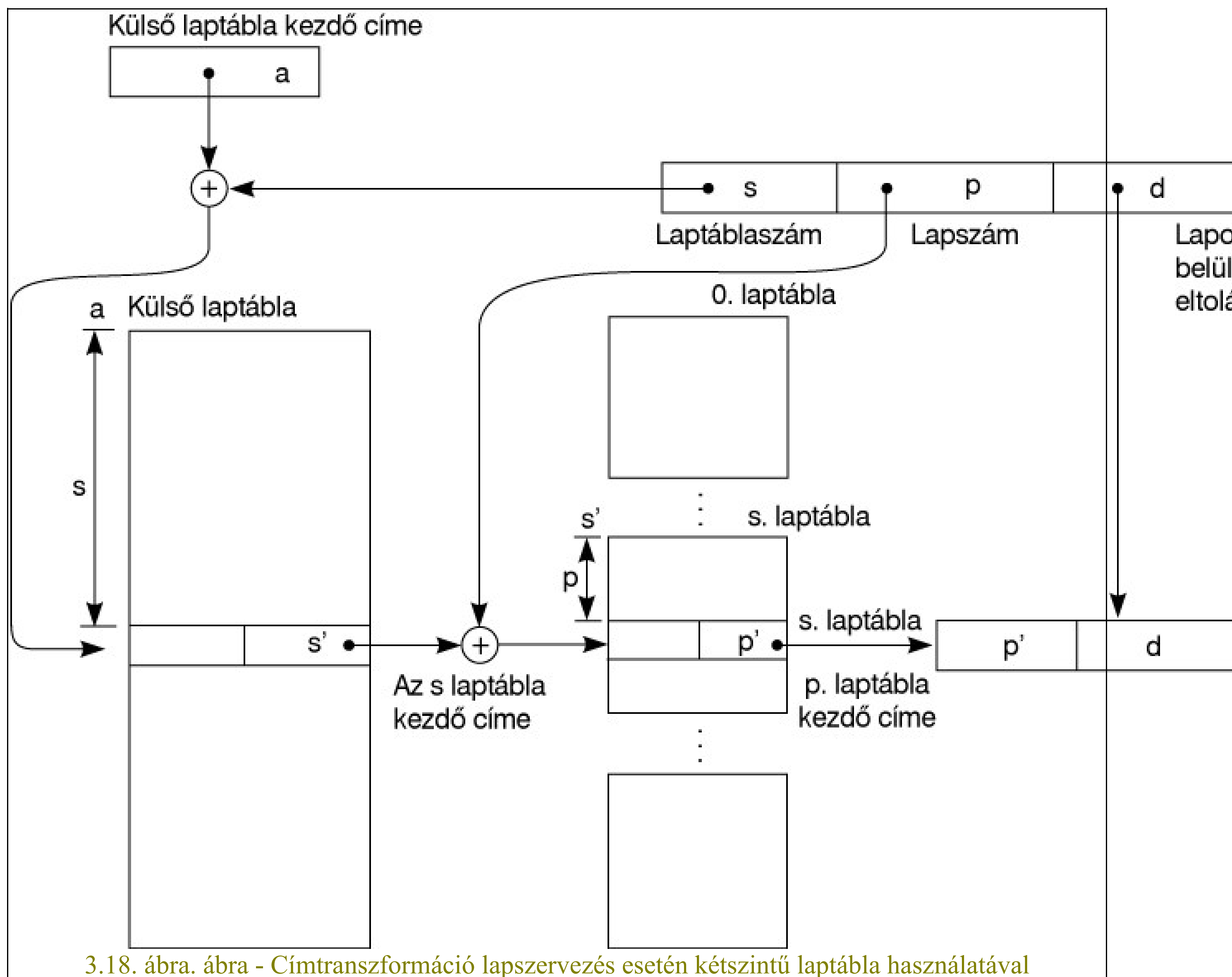


3.17. ábra. ábra - Címtranszformáció lapszervezés esetén

Közvetlen leképezés esetén a folyamathoz tartozó minden lap fizikai címe egy laptáblában van, és a transzformáció mindig innen veszi elő a fizikai lapcímet. Mivel a lapok mérete viszonylag kicsi a programok méretéhez képest, ezért a laptábla mérete meglehetősen nagy lehet, és így nehéz speciális, elkülönített, gyors hozzáférésű tárban tartani. Ha pedig a laptábla a memóriában helyezkedik el, akkor minden egyes laptábla hozzáférést igénylő címtranszformációt magában foglaló memóriaművelet két memória hozzáférést igényel, ami jelentős lassulást okoz.

A laptábla méretének csökkentésére a megoldást a többszintű laptábla alkalmazása kínálja. A modern rendszerek korábban elképzelhetetlen nagyságú logikai címtartomány használatát teszik lehetővé, amely azonban egyben hatalmas méretű laptáblák használatát is jelenti. Gondoljuk csak meg, hogy egy 32 bites logikai cím 4K (2^{12}) byte méretű lapok használata esetén 1 millió (2^{20}) laptábla bejegyzést jelenthet. Ez – bejegyzésenként 4 byte-tal számolva – önmagában 4Mbyte fizikai memóriát igényel. Ekkora méretű táblák memóriában tartása, ki- és bevitele semmiképpen sem célszerű. Ugyanakkor már utaltunk rá, hogy a folyamatok egy adott időintervallumban a teljes címtartományuknak nagy valószínűséggel csak egy-egy részét használják. Így, ha a laptáblát megfelelően osztjuk részekre, elegendő néhány rész-laptáblát a memóriában tartani.

A gyakorlatban – az effektív hozzáférési idő növekedése miatt – legfeljebb két, illetve háromszintű laptáblát szoktak használni. Kétszintű laptábla-szervezés esetén a címképzést a 3.18. ábra mutatja.



3.18. ábra. ábra - Címtranszformáció lapszervezés esetén kétszintű laptábla használatával

A hozzáférési idő problémájára a tipikus megoldást az **asszociatív leképezés**, azaz egy speciális szervezésű, gyors **asszociatív tár (associative registers, translation look-aside buffer)** alkalmazása jelenti, ahol a folyamat bizonyos – gyakran vagy a közeljövőben várhatóan használt – lapjainak logikai és fizikai címei találhatók. Az asszociatív tár gyors hozzáférésű és egyetlen művelettel kiadja a bemenetére juttatott logikai címhez tartozó fizikai címet, tehát lényegében a laptábla gyorsító tárjaként funkcionál.

A sebességben megfelelő asszociatív tárak általában nem elegendően nagyok akár csak egyetlen folyamat teljes laptáblájának befogadására sem, így az asszociatív leképezés megvalósítására **kombinált technikák** alakultak ki. A fizikai lapcím keresése a logikai lapcím alapján egyidejűleg kezdődik mind az asszociatív tárban, mind pedig a direkt laptáblában (ami ilyenkor általában memóriában tárolt). Ha a keresett cím megvan az

asszociatív tárban, ez igen gyorsan – gyakorlatilag még a direkt táblához forduló memóriaművelet megkezdése előtt – kiderül, és a direkt táblához fordulás leáll. Mivel az asszociatív tár csak a laptábla töredékét képes tárolni, tartalmát valamilyen stratégiával folyamatosan frissíteni kell, hogy a várható hivatkozások minél nagyobb valószínűséggel megtalálhatók legyenek benne. Később még részletesen elemezzük azt a megfigyelést, hogy azonos lapra általában több egymást követő hivatkozás történik rövid időn belül. Ezért, ha a hivatkozott lapcím nem található az asszociatív tárban, a direkt leképezés eredményeként kapott logikai-fizikai címpárt érdemes felvenni az asszociatív tárba valamelyik régen használt bejegyzés helyére. Környezetváltáskor az asszociatív tár tartalmát is cserélni kell, hiszen ugyanazon logikai címekhez most új fizikai címek fognak tartozni.

Mivel a programok futásuk egyes időszakaiban a teljes címtartományuknak csak kis részét használják, ezért az asszociatív tárak – kis méretük ellenére – általában 80–99% közötti **találati arányt (hit ratio)** tesznek lehetővé (találati aránynak nevezzük az asszociatív tárban megtalált hivatkozott lapok arányát).

Az egyszerűség kedvéért tegyük fel, hogy a memória hozzáférés ideje 100 nsec, az asszociatív tárban való keresés pedig 20 nsec-t igényel. Asszociatív tár használata nélkül a fizikai memóriacím elérése 200 nsec-t igényel (100 nsec laptábla hozzáférés + 100 nsec memóriakeret elérés). Asszociatív tárral, 90%-os találati arányt feltételezve, az effektív memória hozzáférési idő kb. 128 nsec [0,9*(20 nsec asszociatív tár elérés + 100 nsec memóriakeret elérés) + 0,1*(100 nsec laptábla hozzáférés + 100 nsec memóriakeret elérés)], az elvi 100 nsec helyett.

Az eddigi társzervezési módszerekkel ellentétben lapszervezés esetén az eltolás tekintetében nincs szükség túlcímzés elleni védelemre, hiszen minden kiadható cím lapon belül marad. Túlcímzés csak a lapcímben történhet, ami a laptábla méretének megfelelő határregiszter segítségével ellenőrizhető. A memóriavédelem az egyes fizikai memóriakeretekhez rendelt védelmi bitek segítségével történhet, amelyeket az elviselhető sebesség érdekében hardvernek kell ellenőriznie. A számítógépek többsége azonban nem tartalmazza ezt a hardvertámogatást.

A szegmensszervezéshez hasonlóan szükség van viszont egy olyan bitre (érvényességi, *valid* bit), amely azt mutatja, hogy a lap a memóriában van-e, és természetesen a háttértáron való elhelyezkedés információját is tárolni kell valahol.

A szegmensszervezésű rendszerekhez hasonlóan a lapszervezésű rendszereknél is nagy előny az osztott laphasználat lehetősége. Ilyenkor több folyamat hivatkozhat azonos fizikai lapokra, amelyeket az esetek nagy részében azonos logikai címen is látnak, de ez nem szükségszerű. Képzeljük, mekkora tárigény csökkenést jelent az, hogy a több felhasználó által használt programok kódrészének, illetve a könyvtári eljárásoknak – szövegszerkesztő, fordító, ablakozó, adatbáziskezelő stb. – csak egyetlen példányát kell a memóriában tartani. Ezek a lapok általában olyan eljárásokat tartalmaznak, amelyek kódja végrehajtás során sohasem változik, így több folyamat egyidőben is végrehajthatja ugyanazt az eljárást **(újrahívható, reentrant code)**. Természetesen minden folyamatnak külön regiszter- és adatkészlete kell hogy legyen a *változók* számára. A védelemmel

kapcsolatban korábban mondottakkal ellentétben az ilyen közösen használt lapoknál az operációs rendszer gondoskodik a „csak olvasható” jelleg biztosításáról.

Kombinált szegmens- és lapszervezés

A kombinált szervezés egyesíti mind a szegmens-, mind pedig a lapszervezés előnyeit. A logikai cím három komponensből (szegmenscím, lapcím, eltolás) áll. A lapszervezésből következően nincs külső tördelődés. A szegmens- szervezés miatt a feladat logikai szerkezete tükröződik a társzerkezetben és egyszerűbben megvalósítható az osztott használat. A belső tördelődési veszteség ezzel szemben szegmensenként jelentkezik.

A hozzáférési jogok ellenőrzése és az osztott tárhasználat a szegmensszervezésnek megfelelően történik. A címtranszformáció lényegében első szinten egy laptábla-címeket tartalmazó szegmenstábla, második szinten pedig szegmensenként egy-egy fizikai lapcímeket tartalmazó laptábla segítségével történik, az esetleges asszociatív tár pedig itt címhármasokat tárol, és szegmenscím, lapcím párossal kereshető.

3.4.2. Virtuális tárkezelés

Az előző részben különböző memóriakezelési stratégiákat vizsgáltunk. Mindegyik módszernek az volt az alapvető célja, hogy lehetőleg kellően sok folyamatot tudjunk a memóriában tartani egyszerre, és így megfelelő szintű multiprogramozást tudjunk biztosítani. Azonban a fenti technikák alapvetően megkövetelték, illetve törekedtek arra, hogy a programok végrehajtásuk előtt minél teljesebb terjedelmükben a memóriába kerüljenek. A késleltetett, illetve overlay betöltések, tárcserék, a felhasználók és a programfejlesztési segédeszközök szintjén tették lehetővé a tártakarékos futtatást, általános megoldást nem kínáltak.

Ezzel szemben a **virtuális tárkezelés (virtual memory management)** – a lap-, illetve szegmensszervezésre épülően – olyan szervezési elvek és az operációs rendszer olyan algoritmusainak összességét takarja, mely *megengedi és biztosítja, hogy a rendszer folyamataihoz tartozó logikai címtartományoknak csak egy – a folyamat futásához éppen szükséges – része legyen a központi tárban*, *de ennek ellenére bármelyik folyamat szabadon hivatkozhasson bármilyen, tartományában szereplő logikai címre.*

3.4.2.1. A működés alapjai

A korábbi algoritmusok tárgyalása során láttuk, hogy nem kerülhető meg az, hogy a végrehajtandó utasítások a fizikai memóriában legyenek. Első megközelítésként a teljes logikai címtartományt a fizikai memóriába olvasták. Az átfedő programrészek alkalmazása és a dinamikus betöltés valamelyest lazított a fenti követelményen, azonban e technikák sok óvatosságot és erőfeszítést követeltek a programozótól. A virtuális tárkezelés ezzel szemben a programozó elől is rejtett megoldást kínál, amelyet az operációs rendszer nyújt, és amely teljes szabadságot biztosít akár a fizikai memória méretét sokszorosan meghaladó logikai (virtuális) címtartományok használatában.

A virtuális tárkezelés előzményeit tekintve különböző programok vizsgálata során bebizonyosodott, hogy *nem szükséges*, hogy a programok teljes logikai memóriaterülete a központi tárban legyen, mert a programok nem használják ki a teljes címtartományukat:

- tartalmaznak ritkán futó kódrészleteket (például a hibakezelő rutinok),
- a statikus adatszerkezetek általában túlméretezettek (például vektorok, tömbök, szimbólum táblák definiálásánál a ténylegesen szükségesnél sokkal nagyobb területeket foglalhatunk le),
- a program különböző részei időben egymástól elkülönülten működhetnek (ezt már az overlay technika is kihasználta),
- a programok az összetartozó részek címtartományát is időben egyenetlenül használják, azaz az időben egymáshoz közeli utasítások és adatok általában a térben is egymáshoz közel helyezkednek el **(lokalitás)**.

A korábbi vizsgálatokból az is világossá vált, hogy *nem is célszerű*, hogy a programok teljes logikai memóriaterülete a központi tárban legyen, hiszen

- ekkor a futtatható programok méretét nem korlátozza a központi memória nagysága, azaz a ténylegesen meglevő tárterületnél nagyobb tárigényű (hosszabb) programokat is futtathatunk,
- az egyes folyamatok tárigényének csökkenésével a memóriában tartott folyamatok száma növelhető, azaz növelhető a multiprogramozás foka, és ezzel javítható a CPU-kihasználtság és átbocsátóképesség anélkül, hogy a körülfordulási idő, vagy a válaszidő növekednék,
- a programok betöltéséhez, illetve a folyamatok háttértárba mentéséhez kevesebb perifériás műveletre van szükség, azaz a betöltés/kivitel gyorsabb lesz.

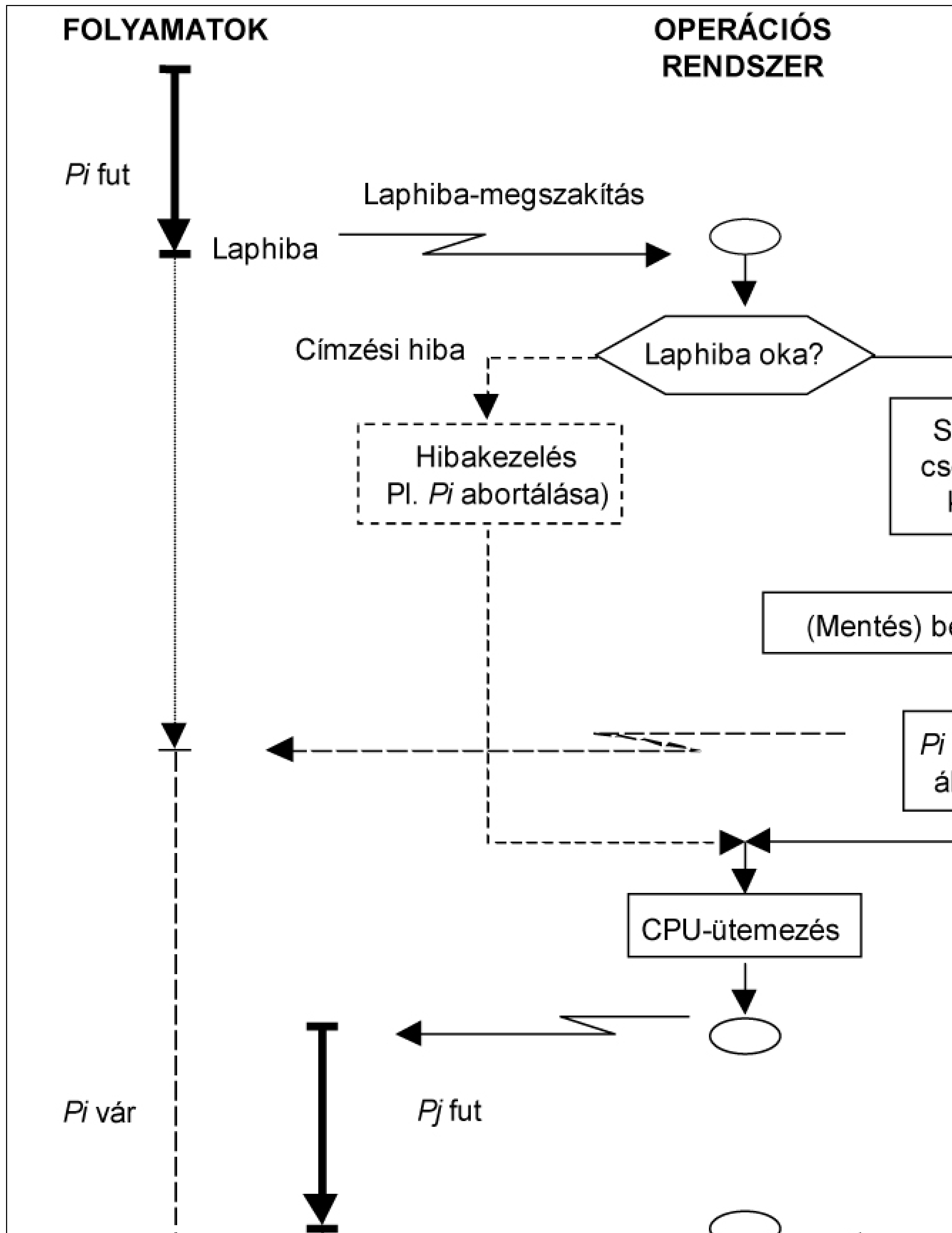
Jól látható tehát, hogy mind a rendszer, mind pedig a felhasználó szempontjából előnyös, ha a folyamatok tárterületének csak egy részét tartjuk egyidejűleg a memóriában.

Amikor egy folyamat érvénytelen – azaz nem a valós memóriában levő – címre hivatkozik, a címképző hardver hibamegszakítást okoz, amelyet az operációs rendszer kezel, és behozza a háttértárról a szükséges blokkot. Ennek logikai lépései a következők:

- Az operációs rendszer megszakítást kiszolgáló programrésze kapja meg a vezérlést, amely
  - elmenti a folyamat környezetét,
  - elágazik a megfelelő kiszolgáló rutinra,
  - eldönti, hogy a megszakítást programhiba (például kicímzés) vagy logikailag érvényes, de nem betöltött blokkra való hivatkozás okozta-e.
- Ez utóbbi esetben

- Az operációs rendszer behozza a kívánt blokkot a központi tárba:
    - a blokknak helyet keres a tárban; ha nincs szabad terület, fel kell szabadítania egy megfelelő méretű címtartományt, ennek tartalmát esetleg a háttértárba mentve,
    - beolvassa a kívánt blokkot.
- Az operációs rendszer átadja a vezérlést a folyamatnak, amely ismételten végrehajtja, vagy folytatja a megszakított utasítást.

A blokk behozatala természetesen nem a folyamat aktív várakozása mellett történik. A perifériás művelet(ek) miatt a blokk behozatal sok időt vesz igénybe (meg kell várni, amíg a perifériás eszköz felszabadul, ki kell várni az átvitelt előkészítő műveleteket, valamint át kell vinni egy blokkot a periféria és a tár között), ezért a központi egység jobb kihasználása érdekében a blokkhiba miatt megszakított folyamatot az operációs rendszer várakozó állapotba helyezi, és más, futásra kész folyamatot indít el. Amikor a kívánt tartomány bekerül a tárba (azaz a folyamat várakozási feltétele bekövetkezik), az igénylő folyamat futásra kész állapotú lesz, és átkerül a futásra kész sorba, ahol megvárja, amíg az ütemező újra elindítja. A folyamatot a 3.19. ábra szemlélteti.



FOLYAMATOK
OPERÁCIÓS RENDSZER
Pi fut
Laphiba-megszakítás
Laphiba
Címzési hiba
Laphiba oka?
Hibakezelés
Pl. Pi abortálása)
(Mentés) betöltés i
Pi várakoz
állapotba
CPU-ütemezés
Pi vár
Pj fut

3.19. ábra. ábra - Laphiba kezelésének folyamata

A virtuális tárkezelés megvalósítása teljesen új feladatok elé állítja az operációs rendszert, amelynek megoldásához megfelelő hardver támogatás szükséges. Legelőször is biztosítani kell, hogy az érvénytelen címre való hivatkozás megszakítást okozzon. Ezenfelül a megszakított utasítások újraindíthatók, vagy folytathatók kell hogy legyenek. A folytathatóság megoldása ritkább, mert utasítások végrehajtása közben fennálló állapotot kellene elmenteni, ami meglehetősen bonyolulttá tenné a processzorokat. Az újraindíthatóság megoldása, bár első pillanatra talán triviálisnak tűnik, bizonyos típusú utasításoknál – például többcímű, blokkmozgató, autoinkremens, illetve autodekremens indirekt című utasítások – ugyancsak nem egyszerű feladat.

A virtuális tárkezelést az IBM/360 számítógépcsalád megjelenésével kezdték elterjedten alkalmazni, és elmondhatjuk, hogy ma már a fejlettebb mikroprocesszorok mindegyike használ valamilyen fajta virtuális tárkezelést. A korszerű hardverek szinte kivétel nélkül lap- vagy kombinált szervezésűek, ezért a virtuális tárkezelés is csaknem mindig lapok mozgatásán alapul, így a továbbiakban mi is lapszervezésű rendszerekkel foglalkozunk.

A virtuális tárkezelés lapcsere algoritmusait egyéb, a blokkokhoz tartozó – később részletesen tárgyalt – bitek is támogatják.

A különböző rendszerek minősítését alapvetően befolyásolja a működési sebességük. A folyamatok futásának sebességét pedig döntően a tárhoz férés effektív ideje határozza meg. Ez virtuális tárkezelés esetén a megfelelő előfordulási valószínűséggel súlyozott memória hozzáférés, illetve (laphiba esetén) lapbehozatali időből számítható. Ha *p* jelöli a laphiba előfordulás valószínűségét, akkor

Effektív hozzáférési idő = (1-p) * Memória hozzáférési idő + p * Laphiba idő.

A laphiba kiszolgálási ideje több, mint öt nagyságrenddel is nagyobb lehet a valós memória hozzáférési időnél (a lemezműveletek időigénye a mai rendszerekben is 10 msec körül van), ezért *p*-nek nagyon kicsinek (10^{-6}–10^{-8}) kell lennie ahhoz, hogy a folyamatok futása ne lassuljon le túlságosan.

A virtuális tárkezelés tárgyalásánál három alapvető kérdést kell megválaszolnunk:

- Melyik lapot hozzuk be a tárba (betöltendő lap kiválasztása)?
- Ha nincs szabad hely a memóriában, akkor melyik lapot cseréljük le (lapcsere, replacement strategy)?
- Hogyan gazdálkodjunk a fizikai tárral, azaz melyik folyamat számára hány lapot biztosítsunk?

A továbbiakban e három kérdést vizsgáljuk meg kicsit részletesebben.

3.4.2.2. Betöltendő lap kiválasztása (fetch-strategy)

Alapvetően két stratégiát követhetünk.

Igény szerinti lapozás (demand paging) esetén csak a laphibánál hivatkozott lapot hozzuk be a tárba. A módszer előnye, hogy a betöltendő lap kiválasztása nagyon egyszerű, továbbá, hogy csak a biztosan szükséges lapok kerülnek be a tárba. Ugyanakkor az új lapokra való hivatkozás mindig laphibát okoz, ami lassítja a működést.

Előretekintő lapozásnál (anticipatory paging) ezzel szemben az operációs rendszer megpróbálja „kitalálni”, hogy a folyamatnak a közeljövőben mely lapokra lesz szüksége, és azokat szabad idejében – amikor a lapcseréhez használt háttértár szabad – betölti.

Ha a jóslás helyes volt, akkor az előre behozott lapok miatt jelentősen lecsökken a laphibák száma, és ezzel a folyamat futási sebessége nagyban felgyorsul. Ha a döntés hibás volt, akkor felesleges lapok foglalják el a tárat.

Mivel manapság a központi tár ára drasztikusan csökken és ezzel párhuzamosan mérete egyre nő, ezért az előretekintő lapozás egyre népszerűbb, hiszen a hibás döntés – azaz a felesleges tárfoglalás – „ára” egyre kisebb.

3.4.2.3. Lapcsere stratégia (replacement strategy)

Lapcsere esetén ki kell választani azt a lapot, amelyiket feláldozzuk az új lap betöltése érdekében. A lapcsere stratégiák alapvető célja az, hogy az optimális esetet közelítsék, azaz azt a lapot válasszák áldozatul, amelyikre a legkésőbb lesz szükség (vagy másképp fogalmazva legtovább nem lesz szükség).

A lapcsere általános esetben két lépésből áll: az áldozat kimentéséből és az új lap betöltéséből. Az algoritmusok hatékonyságát nagyban fokozhatja, ha a mentést – vagyis háttértárra írást – csak akkor végzik el, ha szükséges, vagyis ha a lap tartalma a betöltés óta módosult. Annak nyilvántartása, hogy egy lapra betöltése óta írtak-e, csak hardver támogatással valósítható meg hatékonyan. A támogatás lényege, hogy minden fizikai laphoz tartozik egy jelzőbit – a **módosított bit (modified bit, dirty bit, M bit)**. Ezt a bitet a lap betöltésekor az operációs rendszer törli, a tárkezelő hardver pedig minden, a lapra író memóriaművelet végrehajtásakor beállítja. A bit a laptáblában is elhelyezhető.

Bizonyos algoritmusok igénylik a lapra történő hivatkozások figyelését is, ami ugyancsak hardver támogatással hatékony. A laptáblában erre a célra is fenntarthatunk egy bitet. Ezt a **hivatkozott bitet (referenced bit, used bit, R bit)** a címképző hardver állítja be minden esetben, amikor az adott lapon belüli címre történik hivatkozás. A bitet az operációs rendszer törli adott időnként, vagy eseményhez (például laphiba) kötötten.

Optimális (OPT: Optimal) algoritmus

Az algoritmus *előrenéz* és a lapok *következő használatának idejét* veszi figyelembe. Ennél az algoritmusnál a legkevesebb a laphibák száma. Sajnos azonban a megvalósítása nehézségekbe ütközik, mivel jövőbeni laphivatkozásokra vonatkozó információt igényel (a helyzet az SJF ütemező algoritmushoz hasonló). Az egzakt végrehajtás gyakorlatilag megoldhatatlan, a kódok valamiféle előreolvasását és szimulált végrehajtását igényelné az adatfüggő elágazások figyelembevételével, ami csaknem olyan bonyolult lenne, mint a program valódi végrehajtása. Így csak közelítő megoldásokkal találkozunk. Az optimális algoritmus szerepe pedig az, hogy

összehasonlítási alapként szolgáljon egy-egy eset utólagos értékelésekor, amiből következtetéseket vonhatunk le arra nézve, hogy az alkalmazott algoritmus mennyire közelítette meg az optimumot.

Legrégebbi lap (FIFO) algoritmus

A FIFO-algoritmus úgy próbálja közelíteni az optimálist, hogy *hátranéz* és a *behozatal idejét* figyeli. Az algoritmus azt a lapot cseréli le, amelyik legrégebb óta a tárban van. Megvalósítása egy egyszerű FIFO-listával történhet. Hibája azonban, hogy azokat a lapokat is lecseréli, amelyeket a folyamatok gyakran használnak.

Ennél az algoritmusnál felléphet egy érdekes jelenség, amelyet **Bélády**[1] **-anomáliának** hívunk. Eszerint – várakozásainkkal ellentétben – bizonyos esetekben, ha növeljük a folyamathoz tartozó fizikai memóriakeretek számát, akkor nem csökken, hanem éppen növekszik a laphiba gyakoriság. A 3.20. ábra mutat példát erre.

Laphivatkozások:		0	1	2	3	0	1	4	0	1	2	3	4	0	1
FIFO-algoritmus három memóriakerettel:		0	0	0	3	3	3	4	4	4	4	4	4	0	0
			1	1	1	0	0	0	0	0	2	2	2	2	1
				2	2	2	1	1	1	1	1	3	3	3	3
Laphibák:		•	•	•	•	•	•	•			•	•		•	
FIFO-algoritmus négy memóriakerettel:		0	0	0	0	0	0	4	4	4	4	3	3	3	3
			1	1	1	1	1	1	1	0	0	0	4	4	4
				2	2	2	2	2	2	1	1	1	1	0	0
					3	3	3	3	3	3	2	2	2	2	1
Laphibák:		•	•	•	•			•	•	•	•	•	•	•	•

3.20. ábra. ábra - Bélády-anomália

Újabb esély (SC: Second Chance) algoritmus

Az újabb esély algoritmus a FIFO olyan változatát valósítja meg, amely a sor elején levő lapot csak akkor cseréli le, ha nem hivatkoztak rá. Vagyis *hátranéz* és a *használat tényét* figyeli. Ha hivatkoztak a lapra, akkor a *hivatkozott* bitet törli és a lapot visszateszi a FIFO-lista végére, vagyis a lap kap egy újabb esélyt. Ezzel kiküszöböli a FIFO-algoritmus hibáját és a gyakran használt lapokat nem cseréli le. A *hivatkozott* bit tehát ebben a megoldásban azt jelzi, hogy a lapra történt-e hivatkozás azóta, mióta az operációs rendszer legutóbb megvizsgálta, mint lehetséges áldozatot.

Óra (Clock) algoritmus

Az óra algoritmus csak megvalósításában különbözik az újabb esély algoritmustól. A tárban levő lapok egy körkörös listára vannak felfűzve a betöltés sorrendjében, és egy mutató mutat a legrégebbi lapra. Lapcserénél a kijelölt lap kivitele előtt az algoritmus megvizsgálja az R bitet. Amennyiben egynek találja, úgy nem viszi ki ezt a lapot, hanem törli az R bitet, és a mutató eggyel előbbre lép. A lépések addig ismétlődnek, amíg az algoritmus egy kivihető lapot nem talál (azaz amelynél R=0).

Legrégebben nem használt (LRU: Least Recently Used) algoritmus

Az LRU-algoritmus azt a lapot választja áldozatul, amelyre a folyamatok leghosszabb ideje nem hivatkoztak. Ez az algoritmus közelíti legjobban az optimálist, mivel ugyan *hátrafele* néz, viszont a *használat idejét* veszi figyelembe (azaz a közelmúltból következtet a közeljövőre, ami a lokalitás miatt jó becslést adhat).

Ennél a jó teljesítménye miatt gyakran használni kívánt algoritmusnál gondot okoz, hogy „drága". A használat ideje alapján történő sorbarendezés külön hardvertámogatást igényel, és az algoritmus futási ideje is magasabb az egyszerű algoritmusokénál. A megvalósításra több változatot is kidolgoztak.

- *Számlálóval*. A lapra történő minden hivatkozásnál feljegyezzük annak időpontját (egy logikai óra, vagy számláló állását). Lapcserénél a tárolt időpontok közül a legrégebbit keresi ki az algoritmus. Ez a megvalósítás a lapkiválasztási időt is megnövelheti, hiszen a legkisebb idő megkeresése mellett minden memória hivatkozásnál egy extra memória írási ciklust jelenthet (az időpont tárolása).
- *Láncolt listával.* A memóriában tárolt lapok egy láncolt listában vannak felfűzve, az újonnan behozott lapok a lista végére kerülnek, az algoritmus a lista elején álló lapot választja ki. Címhivatkozásnál a címzett lapot kivesszük a listából és a végére illesztjük.
- *Kétdimenziós tömbbel.* Ennél a megvalósításnál a hardver egy n*n-es (n a lapok számát mutatja) mátrixot használ (inicializáláskor a mátrix a **0** mátrixszal egyezik meg). Az i. lapra történő hivatkozásnál a mátrix i. sorának minden bitjét egyre, majd az i. oszlop minden elemét nullára állítja. A legkisebb bináris értékű sor az LRU-laphoz tartozik.

A bonyolult megvalósítás miatt az LRU-algoritmus helyett sokszor annak – hardvertámogatást nem, vagy alig igénylő – közelítését szokták használni:

Legkevésbé használt (LFU: Least Frequently Used vagy NFU: Not Frequently Used) algoritmus

Ennél az algoritmusnál abból indulhatunk ki, hogy a közelmúltban gyakran használt lapokat a folyamatok a közeljövőben is használni fogják még, és ugyanígy, a közelmúltban ritkán, vagy nem használt lapokra a közeljövőben nem lesz szükség. Ilyenkor az operációs rendszer rendszeres időközönként végignézi a memóriában levő lapokat, és a hozzájuk rendelt számlálóhoz hozzáadja az *R* bit (0 vagy 1) értékét, és egyben

törli az *R* biteket. Az algoritmus a legkisebb számláló értékkel rendelkező – vagyis a legritkábban használt – lapot választja ki kivitelre.

Hátrányt jelent, hogy az algoritmus „nem felejt”, vagyis az egykor gyakran használt lapok még sokáig a memóriában maradnak akkor is, ha már biztosan nem lesz rájuk hivatkozás (például többmenetes fordításnál az egyes menetekhez tarozó lapok). A problémán *öregítéssel* segíthetünk, például úgy, hogy az *R* bitet a legnagyobb helyiértékű bit helyére másoljuk, de előtte a számlálót jobbra léptetve csökkentjük a régebbi hivatkozások súlyát.

A módszer másik problémája, hogy a frissen betöltött (és így biztosan kis számláló értékű) lapokat is könnyen kiteheti újra a háttértárra. Ezért általában a frissen behozott lapokat az első használatig **befagyasztjuk (page locking)** a tárba.

Utóbbi időben nem használt (NRU: Not Recently Used) algoritmus

Az utolsó algoritmus, amelyet megemlítünk, az *R* (hivatkozott) és *M* (módosított) bitek használatán alapszik. A hivatkozottság egy idő elteltével elveszti a jelentőségét, ezért az operációs rendszer rendszeres időközönként törli az *R* bitet. Ugyanakkor az *M* bit értékét őrizni kell, hiszen törlése információ veszteséghez vezetne.

A két bit értéke alapján az algoritmus a lapokat négy csoportba osztja, és lapkivitelnél *hátranézve* és a *használat idejét és módját* is figyelembe véve, a lehető legkisebb prioritású csoportból választ véletlenszerűen.

Prioritás	*R* bit	*M* bit	Megjegyzés
0	0	0	Nem hivatkozott, nem módosított
1	0	1	Nem hivatkozott, módosított
2	1	0	Hivatkozott, nem módosított
3	1	1	Hivatkozott, módosított

Az algoritmusok működhetnek úgy, hogy szükség esetén mindig a folyamat saját munkaterületén belül választanak ki lapot kivitelre – ilyenkor **lokális lapcsere algoritmusról** beszélünk, vagy pedig az egész memórián belül keresnek – **globális lapcsere algoritmus** esetén. Ez utóbbi stratégia sokkal alkalmasabb a terhelésingadozások kiegyenlítésére, azonban könnyen előfordulhat, hogy a „burjánzó” (sokfele hivatkozó) folyamatok kiszorítják a „kicsiket”.

Bármelyik algoritmus használata esetén gyorsíthatjuk a lapcserét, és az eszközök egyenletesebb terhelését érhetjük el, ha egy *periodikusan felébresztett háttérfolyamat, a paging daemon* a háttértár szabad idejében (henyélésekor) a módosított lapokat kimásolja, így ezek esetleges későbbi lecserélésekor azokat nagy valószínűséggel már nem kell újra kiírni.

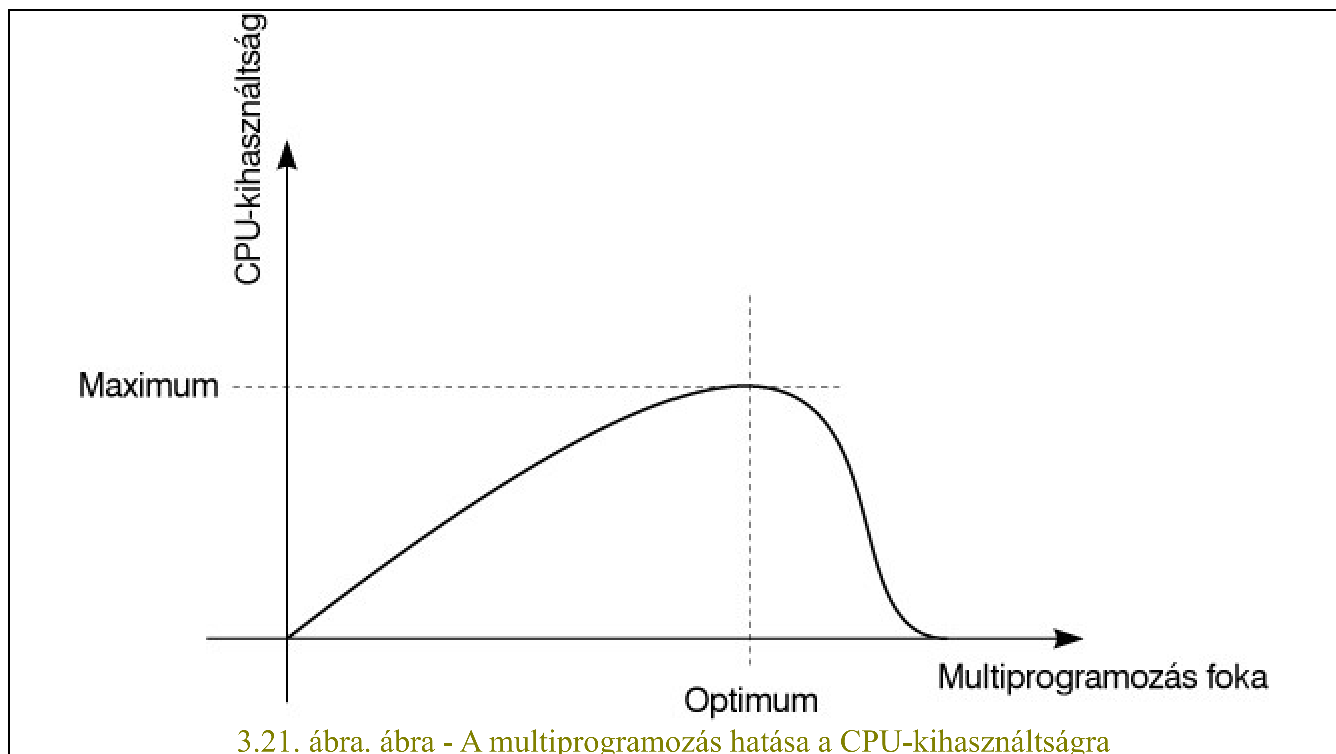
Fenntarthatunk egy előrehozott, kisebb CPU-terhelés idején futtatott áldozatválasztással kialakított listát, amelyikről azonnal leemelhetünk egy lapot, ha cserére van szükség. Ha a listán lévő lapok száma csökken, újabb előzetes áldozatokat választhatunk. Fokozza a hatékonyságot, ha a paging daemon a szabad listára tett

módosított lapokat írja ki először, illetve a listára eleve a kimentést követően kerülnek fel a lapok. Előfordulhat az is, hogy a szabad listára került lapok felülírása előtt újabb hivatkozás történik rájuk. Ezek a hivatkozások még megtalálják a lapokat, ilyenkor természetesen nem kell azokat újra behozni, csupán lekerülnek a szabad listáról.

3.4.2.4. Gazdálkodás a fizikai tárral

A folyamatok lapigénye

A fizikai tárgazdálkodás legalapvetőbb kérdése, hogy hány lapot adjunk az egyes folyamatoknak. A folyamatok szempontjából nézve az a jó, ha minél több lapjuk van a tárban, hiszen nagy valószínűséggel annál kevesebb laphibát okoznak. Túl kevés lap esetén állandóan laphiba lép fel. Ha a rendszerben átlagosan nem tud befejeződni egy laphiba kiszolgálása, mielőtt egy újabb laphiba fellép, a folyamatok felgyűlnek a mágneslemez várakozási sorában, és a CPU-nak nem lesz futtatható folyamata, **CPU-tétlenség** alakul ki. A rendszer teljesítménye leromlik. (Fennáll annak a veszélye is, hogy a hosszú távú ütemező ezt a B/K-intenzív folyamatok túlsúlyaként értékeli, és újabb folyamatokat enged be, ami természetesen tovább rontja a helyzetet.) A gyakori laphibák által okozott teljesítménycsökkenést **vergődésnek (thrashing)** nevezzük. A jelenséget jól szemlélteti a 3.21. ábra, amelyik a multiprogramozás fokának és a CPU-kihasználásnak az összefüggését mutatja virtuális tárkezelést alkalmazó rendszerek esetére.



3.21. ábra. ábra - A multiprogramozás hatása a CPU-kihasználtságra

A rendszer szempontjából nézve, ha egy folyamatnak sok lapot adunk, azt jelenti, hogy kevesebb folyamatot tudunk a tárban tartani, így alacsonyabb lesz a multiprogramozás foka és ezzel várhatóan a CPU-kihasználtság is. A görbe kezdeti szakaszán kevés folyamat van a rendszerben. Minél kevesebben vannak, annál nagyobb annak a valószínűsége, hogy mindegyikük várakozik, a CPU pedig tétlen. A folyamatok számának a

növekedésével a CPU-kihasználás aszimptotikusan közelít az adminisztrációs (például környezetváltási) veszteségekkel csökkentett 100%-os maximumhoz. A folyamatok számának további növekedésekor azonban – mivel az egy folyamatra jutó tárterület csökken – belép a tárkezelés hatása és kialakul a vergődés, a CPU-kihasználás pedig a folyamatok számának további növelésével meredeken leesik. El kell kerülni, hogy a rendszerben ilyen üzemállapot alakulhasson ki, azonban az optimumot lehetőleg meg szeretnénk közelíteni. Fontos tehát, hogy meg tudjuk becsülni, milyen **laphiba gyakoriság (PFF: Page Fault Frequency)** mellett marad még a rendszer egyensúlyban.

Könnyen beláthatjuk, hogy a rendszer egészét tekintve a CPU akkor nem válik tétlenné, ha átlagosan egy laphiba kezelése közben nem következik be újabb laphiba (ha meggondoljuk, ugyanerre a következtetésre jutottunk az effektív memóriahozzáférési idő kiszámítása kapcsán).

A teljes rendszerre vonatkozó egyensúlyi feltételt vonatkoztathatjuk az egyes folyamatokra is, azaz előírhatjuk, hogy két laphiba közötti átlagos futási idejük haladja meg a laphiba átlagos kiszolgálási idejét. Ha ez minden folyamatra teljesül, akkor az egész rendszer is egyensúlyban lesz.

Visszatérve az eredeti kérdésre, célszerű tehát egy folyamatnak annyi lapot adni, amennyi szükséges az egyensúlyhoz, azaz ahány lapra hivatkozik a laphiba kiszolgálás átlagos ideje alatt (ugyanakkor nem sokkal többet, mert akkor leromlik a multiprogramozás foka). Ezt az értéket a **munkahalmaz méretének (working-set size)** nevezzük**.**

Munkahalmaz

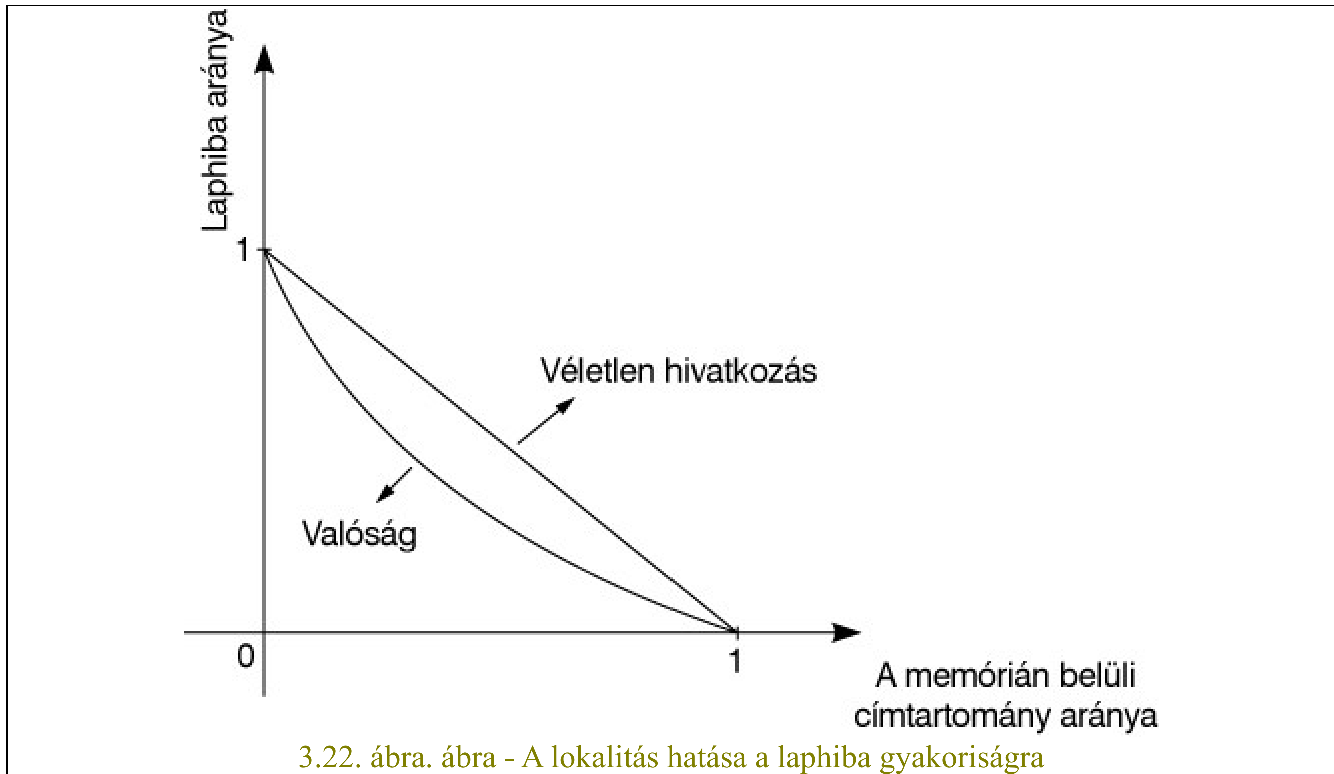
Munkahalmaznak (working set, bizonyos irodalmakban **működő lapkészlet)** nevezzük egy folyamat azon lapjainak a halmazát, amelyre egy adott időintervallumban (munkahalmaz-ablak) – a hossza célszerűen a laphiba kiszolgálási idővel egyezik meg – hivatkozik. A munkahalmaz dinamikus fogalom, tehát időben változik, ami nemcsak a munkahalmazba tartozó lapok, hanem a munkahalmaz méretének a változását is jelenti az időtengely mentén. Pontos mérése, nyilvántartása igen nehézkes, ezért az esetek többségében csak becslésekre hagyatkozunk.

Lokalitás

A munkahalmaz becslésében nagy szerep jut a **lokalitásnak (locality)**. A folyamatok statisztikailag megfigyelhető tulajdonsága, hogy egy időintervallumban a címtartományuknak csak egy szűk részét használják. **Időbeni lokalitás** alatt azt értjük, hogy egy hivatkozott címet a folyamat a közeljövőben várhatóan újra használni fog, míg a **térbeli lokalitás** fogalma azt takarja, hogy az időben egymáshoz közelálló hivatkozások nagy valószínűséggel egymáshoz közeli címekre történnek.

A munkahalmaz becslése során tehát kiindulhatunk abból, hogy a közelmúltra vonatkozó munkahalmaz nem fog lényegesen eltérni a közeljövőben szükséges munkahalmaztól (a korábban tárgyalt lapcsere stratégiák is mind ezen a felismerésen alapulnak). Ez nem jelenti azt, hogy a folyamatok munkahalmazának mérete egyes futási szakaszokban ne változhatna meg jelentősen.

A lokalitás hatással van arra is, hogy hogyan alakul a laphiba gyakoriság a folyamat teljes címtartományából a memóriába töltött hányad függvényében. A laphibák aránya nem lineáris függvénye lesz a memóriába töltött címtartomány arányának (mint ahogy véletlen hivatkozás esetén fennállna), hanem ennél kisebb értékeket kapunk (3.22. ábra).



3.22. ábra. ábra - A lokalitás hatása a laphiba gyakoriságra

Dinamikus lokális tárgazdálkodás

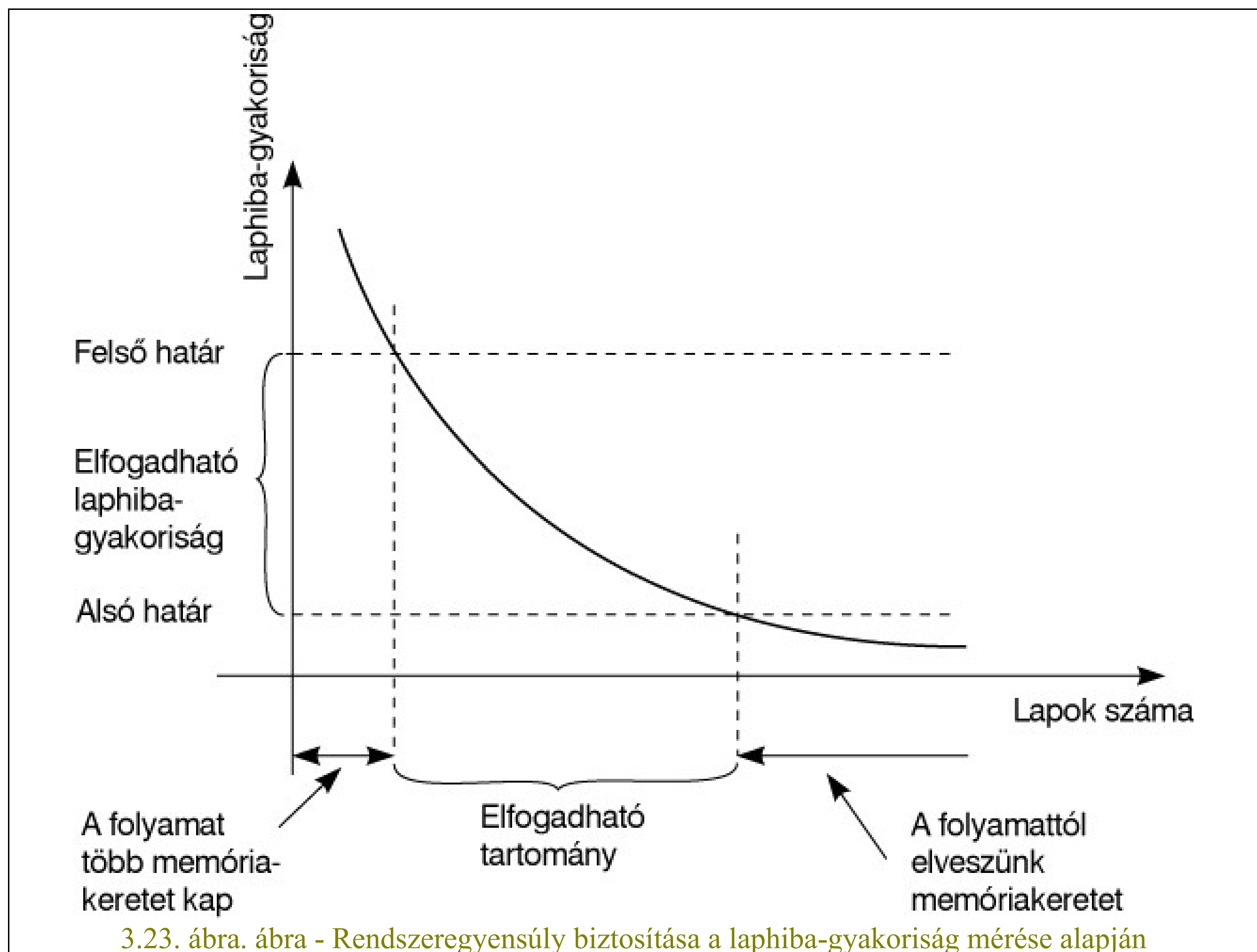
Az optimális rendszerműködéshez tehát el kell kerülni a vergődést, és törekedni kell arra, hogy a folyamatoknak a lokalitás alapján meghatározott munkahalmaza futás közben a központi memóriában legyen. A folyamatokat úgy érdemes elindítani vagy felfüggesztés után újraindítani, hogy egyszerre több lapjukat – a várható munkahalmazt – behozzuk a tárba (másképp induláskor túl gyakori lenne a laphiba). **(Előrelapozás, prepaging)**

Az optimális (legmagasabb fokú) multiprogramozás eléréséhez a legtöbb rendszer lokális lapcsere algoritmust alkalmaz. Ugyanakkor a munkahalmaz tárban tartását – a laphiba gyakoriság mérésén alapuló – a folyamatokhoz tartozó memóriaterület méretének dinamikus változtatásával biztosítja.

A globális lapgazdálkodás (globális lapcsere algoritmusok) kedvezőtlen kölcsönhatást okozhatnának egymástól egyébként független folyamatok között. A statikus lokális tárgazdálkodás ezt megakadályozza, de nem tud alkalmazkodni a folyamatok futás közben változó lapigényéhez. A jó kompromisszumot dinamikus lokális gazdálkodás jelentheti, amelyik alkalmazkodni képes a folyamatok aktuális lapigényéhez.

A folyamatok aktuális lapigényének meghatározására az egyik megoldás a munkahalmaz (illetve méretének) mérése, azonban, mint korábban említettük, ez túl körülményes lenne. Ehelyett inkább a folyamatok *laphiba gyakoriságát*, illetve az ezzel egyenértékű *laphibák között eltelt időt* (interfault time) mérik, ami sokkal könnyebben megoldható.

Ha a laphiba gyakoriság meghalad egy felső határértéket, akkor az operációs rendszer a folyamathoz újabb lapot rendel, vagy pedig – amennyiben nincs több hozzárendelhető szabad memóriakeret, vagy memóriabővében lévő folyamat – a multiprogramozás fokának csökkentése mellett dönt, és felfüggeszti valamelyik folyamatot. Ha pedig a laphiba-gyakoriság alatta marad az alsó határértéknek, akkor elvesz a folyamattól egy lapot (3.23. ábra). Új folyamatot pedig csak akkor lehet elindítani, ha van „elegendő” szabad memóriakeret a számára.



3.23. ábra. ábra - Rendszeregyensúly biztosítása a laphiba-gyakoriság mérése alapján

3.4.2.5. Egyéb tényezők

Lapméret

A rendszer működésére közvetlen hatással van a lapok mérete is. Ezt általában a hardver köti meg, így az itt következőket inkább hardver, mint szoftver rendszerfejlesztők mérlegelik.

Nagyobb lapméretek alkalmazása esetén figyelembe kell venni, hogy kevesebb lappal, így kisebb laptáblával számolhatunk, a perifériás átvitel fajlagos ideje csökken (az átvitel idejének nagyobb részét az előkészítési idő teszi ki), a folyamatok munkahalmazának mérete nő, ugyanakkor nagyobb lesz a belső tördelődés.

Kisebb lapoknál éppen ellenkezőleg, jobban érvényesül a lokalitás (kisebb lesz a munkahalmaz), kisebb lesz a belső tördelődés, viszont a perifériás átvitel fajlagos ideje és a laptáblák mérete is megnő.

Lapok tárba fagyasztása

Az LFU lapcsere algoritmusnál már említettük a lapok tárba fagyasztásának lehetőségét. Erre kifejezetten szükség is van bizonyos esetekben. Például, elindított perifériás műveleteknél az átvitel befejeződéséig a kijelölt címtartományt a tárban kell tartani.

Program struktúra

A virtuális tárkezelés átlátszó, azaz működése el van rejtve a felhasználó elől. Ha azonban a tárkezelés működését és sajátságait – elsősorban a lokalitás hatását – mérnöki szemmel vizsgáljuk, jól látható, hogy maga a programozó is sokat tehet a programok gyorsabb lefutásáért, a rendszer jó teljesítményéért. Csak néhányat említünk a jól alkalmazható „trükkök” közül.

Programíráskor

- többdimenziós tömböket célszerű a tárbeli elhelyezkedésüknek megfelelően bejárni (például keresésnél),
- a programokban célszerű kerülni a nagy ugrásokat,
- az egyszerre használt változókat, az egymást hívó eljárásokat célszerű egymás mellé helyezni.

Fordításkor

- az eljárásokat célszerű egymás mellé helyezni,
- a kód- és adatterületeket célszerű szétválasztani (a kódterület nem változik, így lapcserénél nem kell kimenteni).

3.4.3. Fájlrendszerek

A klasszikus operációs rendszerek felhasználók által leginkább használt absztrakciója a fájl, állomány. Fájlnak a felhasználó, vagy a rendszer szempontjából összetartozó információk perzisztens, a létrehozó programot „túlélő” gyűjteményét nevezzük. A fájlokat a rendszer többnyire valamilyen háttértáron tárolja, amely tartalmát megőrzi még akkor is, amikor a rendszer áramellátását kikapcsolták. Leggyakoribb a közvetlen hozzáférésű perifériás eszközök, mint például a mágneslemez használata, de főleg korábbi rendszerekben fel-feltűntek soros hozzáférésű tárolóeszközök is. Ma ezek az eszközök elsősorban a biztonsági másolatok és a nagyméretű, off-line adatraktárak tárolóiként használatosak.

Egy operációs rendszer általában nagy számú fájlt tárol, kezel. Ezeket egymástól meg kell különböztetni, el kell különíteni. Az egyes állományokat tipikusan egy hozzájuk rendelt név azonosítja, a nagy számú fájlt könnyebb kezelhetőség érdekében könyvtárakba csoportosítják. Az állományok az operációs rendszer felhasználói számára közös, logikai erőforráshalmazt képeznek, az ebből fakadó erőforrás-gazdálkodási feladatok, mint például a hozzáférések ütemezése, koordinálása, szabályozása, korlátozása is a fájlkezelő rendszerek feladata.

A fájlabsztrakció a rendszer használója, programozója számára kényelmes hozzáférést biztosít a fájl tartalmához, elrejtve a tárolás és kezelés konkrét részleteit. A fájlkezelő rendszer tipikusan a következő magas szintű feladatokat valósítja meg:

- hozzáférést biztosít az állomány részeihez, megvalósítja az állományok tartalmának átvitelét a háttértár és a központi tár, illetve a háttértár és egyéb perifériák – például nyomtató – között,
- műveleteket biztosít az egyes fájlokon, illetve a fájlokat összefoglaló könyvtárakon,
- osztott hozzáférések esetén koordinálja, időzíti az egyes folyamatok fájlhasználatát,
- szabályozza a felhasználóknak, vagy azok programjainak a fájlokhoz hozzáférést, különválasztva kezeli az egyes felhasználók állományait és könyvtárait, megakadályozza, hogy a fájlokon illetéktelen felhasználók műveleteket végezhessenek, esetlegesen védi a fájlokban tárolt információkat az illetéktelen olvasások ellen, kódolva, titkosítva annak tartalmát,
- gyakori a fájlokban tárolt információk sérülése, hardver- vagy kezelési hiba miatt bekövetkező elvesztése elleni védelem, a fájlok időszakonkénti megbízhatóbb háttértárra mentése is.

A fájlok kezelését különböző, a konkrét hardvertől egyre távolodó, a logikai szervezéshez egyre közeledő, egymásra épülő programrétegek valósítják meg.

A perifériás eszközhöz legközelebb az ún. készülékkezelő programok (*device driver,* meghajtó) állnak, amelyek közvetlenül a berendezéseket vezérlik. Ezen réteg feladata az adatok központi tár és a periféria közötti átvitelének kezdeményezése, vezérlése, lebonyolítása. A jelenlegi hardverrendszerekben tipikusan megszakításosan vezérelt perifériák megszakításai is ide futnak be.

A meghajtókra épül az elemi átviteli műveletek lebonyolítására szolgáló réteg. Itt történik meg a fájlok tartalmának a periféria által használt címzési rendszerre való leképzése, a periféria által megkövetelt, avagy csak a rendszer teljesítményét javító, az átvitelek számát csökkentő blokkok képzése. Mivel a háttértár elérése még a korszerű, nagy sebességű mágneslemezes tárak esetén is több nagyságrenddel lassabb, mint a központi tár elérése, e réteg gyakori feladata gyorsító tárak (cache) képzése, a gyakran használt blokkok központi tárból történő újrafelhasználásának megvalósítása is. Egyidejű hozzáférés kérelmek esetén a réteg sorba állíthatja a kéréseket, optimalizálva a fizikai erőforrás kihasználását.

Az elemi átvitelekre épül az állományszervezés rétege, amely nyilvántartja a háttértár szabad és lefoglalt területeit, gondoskodik az egyes fájlokhoz tartozó blokkok logikai összefűzéséről, a dinamikus helygazdálkodásról. Ide tartozik új fájlok létrehozásakor, vagy a fájlokban tárolt információ bővülésekor további szabad helyek lefoglalása, illetve állományok törlésekor a korábban lefoglalt területek felszabadítása. Tipikusan ez a réteg biztosít eljárásokat a rendszer programozói számára az állomány tartalmának eléréséhez, módosításához.

A logikai állományszervezés rétege ismeri és kezeli a nyilvántartásokat, az állomány neve alapján megtalálja annak helyét. Ide tartozik az állományok felhasználónkénti hozzáférésének szabályozása, de gyakran a konkurens elérések időbeli ütemezése, szinkronizálása is e réteg feladata.

Vizsgáljuk meg részletesebben az egyes rétegek megvalósításánál használt adatszerkezeteket, algoritmusokat. A készülékkezelők és az elemi átviteli műveletek rétegével itt nem foglalkozunk, azt a készülékkezelésről szóló fejezet részletezi. Az állományok tárolására közvetlen hozzáférésű lemezegységet tételezünk fel.

3.4.3.1. Az állományok tárolása a lemezen

Az operációs rendszer a lemezterületet a hardverhez hasonlóan nagyobb egységekben, blokkokban kezeli. Ez az adatmozgatás, lefoglalás, felszabadítás alapegysége, ennél kisebb területekkel az operációs rendszer nem foglalkozik (néhány kivételtől eltekintve). Egy logikai – operációs rendszer által képzett – blokk tipikusan 1 vagy néhány hardverblokkot – szektort – tartalmaz. A blokkok méretének meghatározását szervezési, tárolási és hatékonysági szempontok befolyásolják. Minél nagyobb blokkokat használunk, annál kevesebb az egységnyi információ átviteléhez szükséges többlet művelet, viszont a nagy blokkokat ki nem töltő információ miatt feleslegesen foglalunk mind a háttértárban, mind a központi tárban területet, jelentős belső tördelődési veszteség lép fel.

A blokkok méretét gyakorta meghatározza az adatszerkezetekben a blokkokat azonosító címek számára fenntartott hely mérete. Főleg kisebb, kezdetlegesebb rendszerekben a blokkok címzésére viszonylag kevés helyet, mondjuk 16 bitet tartottak fenn. Eleinte a 16 biten ábrázolható különböző értékek bőven elegendőnek bizonyultak az összes, a lemezen előforduló fizikai szektor egyenkénti megcímzésére. Azonban a lemezegységek kapacitásának ugrásszerű növekedésével ez a 16 bit hamarosan kevésnek bizonyult, fizikai szektorokat össze kellett vonni az operációs rendszer által egyben kezelt logikai blokkokká, hiszen ezek száma nem haladhatta meg a 64 K-t. Ezért aztán a logikai blokkok mérete egyre növekedett, így gyakran előfordulhatott, hogy egy tipikusan néhány száz bájtból álló szöveges fájl a lemezen több megabájtnyi területet kényszerült elfoglalni. A csapdából természetesen egyszerűnek tűnik a kiút, nagyobb címtartományt kell fenntartani a blokkok számára. Ám ez nem csak a blokkok adminisztrációja számára fenntartott területeket növeli, elvéve a helyet az „értékes” adatoktól, de a háttértárak rohamos fejlődésével a ma még bőségesen elegendőnek tűnő címtartomány 1-2 éven belül szűknek bizonyulhat. Az egyes rendszerváltozatok kompatibilitásának fenntartása is nehezíti a probléma megoldását, hiszen a rendszer belső adatszerkezeteit érintő módosításokat nem könnyű zökkenőmentesen lebonyolítani.

Másik lehetőség a háttértár particionálása, a fizikailag nagy számú blokkot tartalmazó lemezegységek logikailag kisebb egységekre, ún. kötetekre „darabolása”, ám a szétszabdalt tárterület kezelése is problémákat okoz.

A fájlszervezés rétegének a lemez minden egyes blokkját nyilván kell tartania, tudni kell, mely blokkok tartoznak az egyes fájlokhoz, melyek az éppen szabad, fel nem használt blokkok. Ismernie kell a különböző alacsony (például szabad területek), illetve magas szintű (például könyvtárak) nyilvántartások tárolására fenntartott blokkokat is, és gazdálkodnia kell velük.

Alacsony szinten kétfajta adatszerkezetet különíthetünk el, a lemez szabad blokkjainak, illetve az egyes fájlokhoz tartozó blokkoknak a leírását. Mivel mind információtartalmában, mind kezelésében más jellegű nyilvántartással van dolgunk, e célokra más és más adatszerkezetek honosodtak meg.

A szabad blokkok nyilvántartására a különböző adatszerkezetek használatosak:

- *Bittérképes ábrázolás.* A lemez mindegyik logikai blokkjához egy bitben tárolható, hogy szabad-e. Ezekből a bitekből képzett vektort a lemez kijelölt helyén tároljuk. A bitvektoros szervezés esetén egymás melletti szabad blokkok kiválasztása nagyon egyszerű, de a vektor kezelése csak akkor hatékony, ha a teljes vektort a központi tárban tudjuk tartani.
- *Láncolt listás ábrázolás.* A szabad blokkokból „lecsípett” bájtokban egy másik szabad blokk sorszámát, címét tárolhatjuk. Így a szabad blokkokat mintegy láncra fűzzük, csak a lánc elejét kell „kézben tartani” – egy jól definiált helyen tárolni –, innen kiindulva az összes szabad blokk elérhető. Mivel a szabad blokkban értékes információ nincs, ezért a lecsípett bájtok nem okoznak problémát. Az ábrázolás viszont nem hatékony, a lista bejárása lassú, mert több lemezműveletet igényel.
- *Szabad helyek csoportjainak listája.* A fenti ábrázolás teljesítménye könnyen javítható. Ahelyett, hogy minden egyes blokkból csak egyetlen címnyi területet csippentenénk le, kihasználhatjuk a teljes blokkot arra, hogy más szabad blokkok címét tároljuk. Mivel a szabad blokkok száma dinamikusan változhat, az egyik címet a fenti láncolt lista képzésére használjuk, ám a többi *n-1* címnyi terület 1-1 szabad blokkot jelöl ki.
- *Egybefüggő szabad területek nyilvántartása.* A szabad blokkok egyesével történő tárolása helyett egy táblázatban az egymás mellett lévő szabad blokkokról az első sorszámát és a terület hosszát tároljuk. A módszer akkor előnyös, ha a szabad területek átlagos hossza (jóval) nagyobb egynél, valamint ha az allokálásnál egymás melletti szabad területeket akarunk kiválasztani. Ha a táblázat elemei a kezdő blokk szerint rendezettek, az egymás melletti szabad területek összevonása egyszerű.

Az egyes állományokhoz tartozó blokkok tárolására, lefoglalására a következő általános megoldások alakultak ki:

- *Folytonos terület lefoglalása.* A fájlhoz tartozó információt egymás melletti szabad blokkokban tároljuk. A rendszernek csupán az első blokk sorszámát, valamint a blokkok számát kell tárolni az állományt leíró adatok között.

A módszer legnagyobb hátránya, hogy az esetek nagy többségében nem tudjuk előre, hány blokkra lesz szükségünk, így valamilyen algoritmussal a szükséges blokkok számát becsülni kell. Ha ez nem bizonyul elégnek és a lefoglalt blokkokat követő területet már használják, akkor a foglalást végző eljárás hibajelzést ad vissza, vagy megpróbál nagyobb szabad területet keresni, és oda a már felhasznált blokkok tartalmát is átmásolni. Itt is felléphet a külső tördelődés problémája, azaz az összefüggő területek lefoglalása és felszabadítása esetén a szabad területek kis, nehezen használható részekre tördelődnek. Különböző allokációs algoritmusokkal (első illeszkedő, legjobban illeszkedő, legrosszabbul illeszkedő kiválasztása) küzdhetünk a tördelődés ellen, illetve időszakonként tömörítést hajthatunk végre, ami azonban meglehetősen időigényes és körülményes eljárás, ezért viszonylag ritkán használatos. A fájlkezelésben az allokációs algoritmusok kombinált használata is előfordul. Amíg a lemezen nagy szabad területek vannak, általában teljesül a *worst-fit* allokáció mögött álló feltételezés, azaz a maradék használható más célra, legyen hát minél nagyobb. Amikor a lemez foglaltsága növekszik, ez a feltételezés egyre kevésbé teljesül, gyakorlatilag nincs jelentősége, melyik területet allokáljuk *(first-fit, next-fit)*. További telítődéskor már használhatatlan hulladékok keletkeznek, amelyeket minimalizálni érdemes *(best-fit)*. Ha mindemellett még a lefoglalandó terület méretét is figyelembe vesszük, és az átlagos szabad területhossz és a foglalás aránya szerint választunk algoritmust, viszonylag kedvező kihasználást érhetünk el.

A folytonos terület foglalásának előnye, hogy a tárolt információnak mind soros, mind közvetlen elérése egyszerű, gyors. Napjaink rendszerében ez a tárolási forma általában akkor használt, ha előre tudjuk a szükséges blokkok számát, illetve igény az összes blokk egyidejű, gyors tárba, illetve tárból való mozgatása. Tipikus eset a virtuális tárkezelés esetén a tárcsere (swapping) megvalósítása.

- *Láncolt tárolás.* A szükséges blokkokat egyesével allokáljuk, minden blokkban fenntartva egy helyet a következő blokk sorszámának. Általában a rendszer az állományleíróban az első és esetleg az utolsó blokk sorszámát tárolja.

A tárolási mód nagy előnye, hogy ez egy dinamikus adatszerkezet, az állomány mérete szükség szerint tetszőlegesen növekedhet, határt csak a szabad blokkok száma szab. A fenti módszer külső tördelődéstől is mentes, a szabad területeket nem kell tömöríteni.

Hátrány, hogy a láncolt tárolás csak a tárolt információ soros elérést támogatja, a közvetlen eléréshez a listát az elejéről végig kell járni.

Külön problémát jelent, hogy a lánc szervezéséhez szükséges címet a blokkok hasznos területéből kell lecsípni. Így egy-egy blokk központi tárba másolásakor ezeket a darabokat ki kell hagyni. Ezen a problémán segít a láncolt tárolás egy változata, ahol a láncokat az állományoktól elkülönülten, egy **fájl allokációs táblában (FAT)** tároljuk. A táblában a lemez minden blokkjához tartozik egy bejegyzés (pointer), amelyik – amennyiben a

blokk fájlhoz tartozik –- a fájl következő blokkjára mutat. Az egyes állományokhoz csak a lánc első blokkjának sorszámát kell tárolni az állományleíróban, a következő blokkok a FAT-ból megtalálhatók. Ezt az allokációs táblát egyidejűleg a szabad blokkok tárolására is fel lehet használni. A tárolt információ közvetlen elérése is egyszerűbb, nem kell az egyes blokkokat végigjárni, elég e FAT-ban „lépkedni". A FAT, vagy annak jelentős része a tárban tartható, így ez a keresés gyors. A sérülékenység ellen a FAT kettőzött tárolásával védekezhetünk. Ezt a megoldást alkalmazzák az MS-DOS rendszerek.

- *Indexelt tárolás.* Az indexelt ábrázolás alapötlete, hogy az állományhoz tartozó blokkokat leíró címeket külön adatterületre, az indextáblába gyűjtjük. Az állományleíróban az indextáblát tartalmazó blokk(ok) címét kell tárolni.

A módszer előnyei, hogy a közvetlen hozzáférés megvalósítása egyszerű, hiszen az indextábla segítségével minden blokk közvetlenül megtalálható. Lehetőség van „lyukas" állományok tárolására, azaz olyan állományokra, amelyek nem minden blokkja tartalmaz információt. Ebben az ábrázolási formában a nem használt területekhez nem kell lemezblokkot lefoglalni, azok helyét az indextáblában üresen hagyjuk.

Hátránya, hogy a blokkok címeinek tárolásához akkor is legalább egy teljes (index)blokkot kell lefoglalni, ha az állomány csak kevés blokkot tartalmaz. A tárolási mód pazarló. Mivel az indextábla mérete előre nem ismert, ezért meg kell oldani, hogy a táblázat is dinamikusan növekedhessen, például az indexblokkokat láncba fűzzük, hasonlóan a szabad helyek csoportjainak tárolásához, vagy többszintű indextáblákat használunk.

- *Kombinált módszerek.* Egyes rendszerek az állomány hozzáférési módja, vagy mérete függvényében más és más módszert alkalmazhatnak. Például kis állományokat folytonosan, míg nagyokat indexelten tárolhatunk. Vagy soros hozzáférés igénye esetén láncolt, közvetlen hozzáférés esetén összefüggő blokkok tárolását választhatjuk.

Mind a szabad blokkok, mind az állományok tárolásának fent említett módszerei többé-kevésbé érzékenyek a lemezen tárolt nyilvántartások, adatszerkezetek sérülésére. Például egy láncolt blokkos szabad hely nyilvántartásnál egy blokk sérülése a lánc megszakadásához vezethet, ezzel a lánc végén álló blokkok elvesznek a rendszer számára. Még nagyobb problémát jelenthet az egyes állományokhoz tartozó blokkok leírásának elvesztése, megsérülése.

Ezt a problémát az operációs rendszerek általában redundáns adatszerkezetekkel igyekeznek enyhíteni. Például használhatunk kétirányú láncokat, ha a lánc mindkét vége még megvan, egy-egy blokk kiesését könnyen elviselhetjük, mindkét irányból elindulva megkeressük a „szakadást" és összefűzzük a láncot.

A redundáns adatszerkezetek árát a csökkenő értékes tárolási kapacitással és a lassabb kezelő algoritmusokkal fizetjük meg. Egy rendszer tervezőjének gondosan mérlegelnie kell tehát az egyes tényezőket, például azt hogy a perifériás eszköz várhatóan milyen arányú meghibásodásokat produkál, mekkora katasztrófát okoz számunkra egyes szabad blokkok, vagy az állományok blokkjainak elvesztése. A sérülések hatásai kiküszöbölésének egyszerű eszköze a háttértár tartalmának rendszeres, gyakori mentése.

A fájlkezelő rendszernek – vagy az erre szorosan épülő segédprogramoknak – tipikus feladata a különböző lemezen tárolt nyilvántartások konzisztenciájának ellenőrzése, az esetleges sérülések kijavítása. A javítás ügyesen megválasztott adatszerkezetek esetén, bonyolult – nagy lemezegységek esetén lassú – ellenőrző algoritmusokat használva gyakran automatikusan történhet, ám néha a rendszergazda kézi beavatkozására is szükség lehet. Legrosszabb esetben a rendszergazdának egyes blokkok tartalmát „szemrevételeznie" kell, megtippelni, hogy vajon ez szabad blokk lehetett-e, vagy valamelyik – és főleg melyik – állományhoz tartozhatott. Ez a tevékenység szöveges információk esetén még úgy-ahogy sikerrel kecsegtet, de „bináris" állományok esetén nagyon nehezen használható.

3.4.3.2. A fájlok szerkezete

Míg a fentiekben az állományok háttértáron lévő fizikai szerkezetével foglalkoztunk, most a felhasználó szempontjait tükröző belső szerkezetét taglaljuk.

A fájl legáltalánosabb esetben összetartozó bitek sorozatának tekinthető. Napjaink hardver architektúrái tipikusan a bájtot tekintik a tárolás alapegységének, így a fájlokat is felfoghatjuk mint összetartozó bájtsorozatot. Ha ritkán is, de azért találkozhatunk a bájthatárokat nem „tisztelő" – változó bithosszúságú – információtárolási móddal is.

Az így tárolt bit- vagy bájtsorozathoz a felhasználó saját szempontjai szerint szerkezetet, jelentést rendelhet. Ilyenkor az alapegységet általában mezőnek nevezzük, amely rendeltetése szerint egy-egy adott típusú, kódolású adatot tartalmaz. Az összetartozó mezőket gyakran rekordokba foglalhatjuk. Mind az egyes mezők, mind a rekordok tárolás szerint lehetnek kötött, vagy változó hosszúságúak. Ez utóbbi esetben gondoskodni kell az egyes mezők, rekordok határainak felismerhetőségéről, például tárolva a mező hosszát, vagy speciális végjellel jelezve az adategység végét.

Az egyes operációs rendszerek a felhasználói adatszerkezethez különbözőképpen viszonyulhatnak. A leggyakoribb, általános célú fájlrendszerek nem veszik figyelembe a jelentés szerinti adatcsoportosítást, számukra a fájl „csak" egy bájt-, esetleg ritkán bitfolyam. A rendszer csak ennek a folyamnak a manipulálásához ad közvetlen eljárásokat (például olvasd a következő *n* bájtot). Az állományokat kezelő felhasználói programoknak kell a megkapott bájtokhoz jelentést rendelni, elkülöníteni az egyes információ-darabkákat.

Célrendszerekben, vagy univerzális, adatbáziskezelést is integráló általános célú rendszerekben (ilyeneket különösen IBM-rendszereken találhatunk) megtalálható viszont a felhasználói adatszerkezetet figyelembe vevő, azt támogató szervezés is, ahol például létjogosultsága van az olvasd a következő rekordot vagy következő mezőt eljárásnak. Persze itt külön gondot jelent az egyes adatszerkezetek „megismertetése" a kezelő eljárásaival, gyakori, hogy új adatszerkezet, tárolási mód stb. esetén a fájlrendszer adatkezelő eljárásait is bővíteni kell.

Megjegyzendő, hogy még azon rendszerekben is, amelyek a felhasználók fájljait szerkezet nélküli bájtsorozatnak tekintik, vannak a rendszer által ismert és kezelt szerkezetű állományok. Például a futtatható programok kódját tartalmazó fájlok általában rendszerenként különböző, de kötött, szigorúan definiált

szerkezettel rendelkeznek. Bár az operációs rendszer fájlkezelő alrendszere nem ismeri ezek szerkezetét, de az erre épülő betöltő program annál inkább ismeri és használja. Ugyancsak találunk a rendszerekben minimális támogatást egyszerű szövegfájlok, illetve parancsfájlok kezelésére.

3.4.3.3. Könyvtárak

A könyvtár állományok – esetleg más könyvtárak – gyűjteménye, a könyvtár tartalmát a hozzá tartozó nyilvántartás (katalógus, directory) írja le. A könyvtár és nyilvántartás fogalma a szakirodalomban gyakran keveredik.

Az egyes nyilvántartások állományonként egy ún. nyilvántartás bejegyzést tárolnak. A bejegyzések tartalmazzák az egyes állományok attribútumait.

Az állomány legfontosabb attribútuma az azonosítására szolgáló név. Ennek a névnek könyvtáranként egyedinek kell lennie. Míg egyes rendszerek a nevet csak egy karaktersorozatnak tekintik, addig mások a nevet részekre osztják, például névre, típusra (ún. kiterjesztésre), esetleg verziószámra. Ezeket a rendszer külön kezeli, például hivatkozhatunk az adott könyvtár összes adott típusú fájljára, vagy például az egyes fájlok legutolsó, legfrissebb tárolt verziójára.

A nyilvántartás bejegyzés tartalmazhat az állomány fizikai elhelyezkedésére vonatkozó információkat, mint például a hosszát, a hozzá tartozó blokkok leírását, a lehetséges hozzáférés módját. Gyakoriak a logikai attribútumok is, például az állomány típusa (belső szerkezete szerint), tulajdonosa, különböző időpont adatok – létrehozásának, utolsó hozzáférésének, módosításának dátuma, esetleg érvényességének ideje –, felhasználónkénti, vagy felhasználó-csoportonkénti hozzáférés, engedélyezett műveletek.

A nyilvántartás bejegyzéseket tárolhatjuk a lemez speciális, elkülönített területén, de speciális, kötött szerkezetű fájlokban is. Néha előfordul kombinált tárolási forma, például kötet nyilvántartásba kerül a köteten lévő összes fájlt leíró nyilvántartás bejegyzés fizikai, esetleg logikai attribútumai, míg a neve, a könyvtárak szervezésének leírása külön tárolódik.

Az aktuálisan használatban lévő fájlokról az operációs rendszer nemcsak a háttértáron, de a központi tárban is tárol információkat. Ezek a fenti attribútumok – egy része – mellett a nyitott fájlok kezeléséhez szükséges egyéb állapotinformációt is tartalmazzák. Ilyen lehet például soros hozzáférés esetén az aktuális pozíció az állományban, vagy a megengedett műveletek listája, de osztott, konkurens kezelés esetén a szinkronizációhoz szükséges információk (például lock) is ide kerülnek.

A fájlkezelő rendszerek gyakorta szerveznek hierarchiákat az egyes könyvtárakból. A korai rendszerek például kétszintű hierarchiát használtak, a második szinten felhasználóként külön könyvtárba foglalva az egyes fájlokat. Napjainkban elterjedtek a szabadabb, flexibilisebb szerkezetek, például a faszerkezetű könyvtárak, ahol az egyes könyvtárak nemcsak fájlokat, hanem alárendelt könyvtárakat is tartalmazhatnak. Legáltalánosabb a gráfszerű könyvtárszerkezet. Mivel tipikus probléma lehet egy körrel rendelkező gráfban a tárolt elemek bejárása,

kilistázása, ezért a rendszerek igyekeznek biztosítani a körmentes gráfszerkezetet, esetleg kivételt téve a rendszergazdával.

3.4.3.4. Műveletek

A fájlkezelés modellszinten is megjelenő alapműveletein túl a megvalósítás további részműveleteket igényel. A fájlok nyilvántartásának megoldása az egyes rendszerekben igen különböző lehet, így a könyvtárakra vonatkozó műveletek is jelentősen különbözhetnek.

Jellegzetes műveletek a könyvtárakon:

- *Fájl attribútumainak módosítása.* Az állományhoz tartozó egyes logikai információk az állományban tárolt információk módosítása nélkül is megváltoztathatók.
- Új könyvtár létrehozása
- *Könyvtár törlése.* Könyvtár törlésekor eldöntendő, hogy mi történjék a tartalmával. Például egy könyvtár csak akkor törölhető, ha üres, nincsenek benne állományok vagy könyvtárak. Esetleg a törlés törli az összes benne lévő állományt, esetleg rekurzívan törli az összes benne foglalt könyvtárat is.
- *Keresés.* A könyvtárakon végzett egyik leggyakoribb művelet, hogy egy névhez meg kell találni a hozzá tartozó állományt. Keresni lehet a hierarchikus könyvtár szerkezetben, de végezetül a keresés mindig az egy könyvtárban tárolt állományok között történik. A keresés sebessége jelentősen befolyásolhatja a fájlkezelő rendszer teljesítményét. Ezért különböző adatszerkezetekkel igyekeznek a keresést felgyorsítani. A könyvtár hierarchiában történő keresést gyakran hashtáblákkal gyorsítják, míg a nyilvántartás bejegyzések közötti keresést például rendezéssel vagy hashtáblákkal lehet gyorsítani.

3.4.3.5. Osztott fájlkezelés

A fájl lehet olyan erőforrás, amelyet egyidejűleg több folyamat is használni akar. Amennyiben ezek a folyamatok csak olvassák, tartalma nem módosul, így az állomány osztottan is használható. Ha azonban legalább egy folyamat írja az állományt, akkor mint minden erőforrást, ezt is kölcsönös kizárással védeni kell. Kérdés, hogy a kölcsönös kizárást mennyi ideig kell fenntartani.

Az osztott fájlkezelést szabályozhatjuk a teljes fájlra. Ilyenkor az állományt először megnyitó folyamat definiálhatja, hogy a későbbi megnyitási kérelmekből mit engedélyezhetünk: például kizárólagos használatot kérünk, azaz több megnyitást nem engedélyezünk. Esetleg megengedhető, hogy a többi folyamat az állományt csak olvassa; legáltalánosabb esetben pedig egy vagy több folyamat is megnyithatja írási joggal.

Ha az állományt legalább egy folyamat írhatja, a fájlkezelő rendszernek definiálni kell, hogy az olvasó folyamatok az állomány változását mikor veszik észre: azonnal, vagy csak azután, hogy az író folyamat lezárta az állományt, addig a régi tartalmat látják.

Az egész állományra vonatkozó kölcsönös kizárás néha túlzottan, megengedhetetlenül óvatos, hiszen egyéb folyamatok az állományhoz csak lezárása után férhetnének hozzá.

Amelyik folyamat az állományban összetartozó információt akar elérni – írni vagy olvasni –, akkor ez(eke)t, mint önálló erőforrásokat lefoglalja – illetve várakozik, amíg felszabadul(nak) –, majd az átvitel lezajlása után a lefoglalt rekordo(ka)t felszabadítja. Természetesen akár több egyidejűleg írási joggal rendelkező folyamat is létezhet, a kölcsönös kizárás csak egyes kijelölt rekordokra vonatkozik. Az állomány módosítását a többi folyamat a módosított rekordok felszabadítása után azonnal látja.

Sajnos a kölcsönös kizárás miatt itt is felléphet a holtpont vagy a kiéheztetés veszélye.

3.4.3.6. A hozzáférés szabályozása

Mivel az operációs rendszer általában több felhasználó állományait tárolja, kezeli, felvetődik az az igény, hogy jogosulatlan felhasználók ne férhessenek hozzá minden állomány tartalmához, illetve ne végezhessenek rajtuk bizonyos műveleteket.

A hozzáférési jogokat az állomány létrehozója vagy az állomány felett speciális jogokkal rendelkező felhasználó definiálhatja. Megjegyzés: az állományokat folyamatok hozzák létre, a „létrehozó" vagy az a felhasználó, aki a folyamatot elindította, vagy a folyamathoz tartozik egy fix felhasználó. A hozzáférési jogok általában az egyedi fájlokhoz, esetleg egész könyvtárakhoz tartoznak. Néha azonos fájlokat több néven, több hozzáférési úton is elérhetünk, ilyenkor a jogosultságokat rendelhetjük mind a konkrét fájlhoz, mind az elérési útvonalhoz. A jogosultságok azt szabályozzák, hogy a rendszer egyes felhasználói milyen műveletek elvégzésére jogosultak. Amennyiben a rendszernek túl sok felhasználója van, célszerű a jogosultságokat felhasználói csoportokhoz rendelni.

Fájlokhoz rendelhető tipikus jogosultságok: írás, olvasás, hozzáírás, végrehajtás, törlés, míg könyvtárakra vonatkozóan megengedhető módosítás, törlés, listázás, keresés, új állomány létrehozása.

3.5. Készülékkezelés

A számítógépeken futó alkalmazások jelentős része B/K-intenzív, azaz a számítógép feldolgozó képességét elsősorban külső eszközökkel (perifériákkal) folytatott szervezett, rendezett adatcserére használja. A be-/kiviteli eszközök hatékony működtetése ezért az operációs rendszerek egyik legfontosabb feladata. A számítógépeket a legváltozatosabb készülékek veszik körül, amelyek mindegyike speciális kezelést igényel.

Az operációs rendszer feladatai a B/K-kezelésben:

- *Egységes alkalmazói felület kialakítása (application interface).* Ennek keretében el kell takarni a részleteket, minél inkább egységesíteni a műveleteket, lehetővé tenni a logikai perifériakezelést, a műveletek átirányíthatóságát.
- *Egységes csatlakozó felület kialakítása a készülékek számára (device interface).* A perifériák fizikai csatlakoztatása a hardverarchitektúra meghatározott helyein, egyszerűbb, vagy

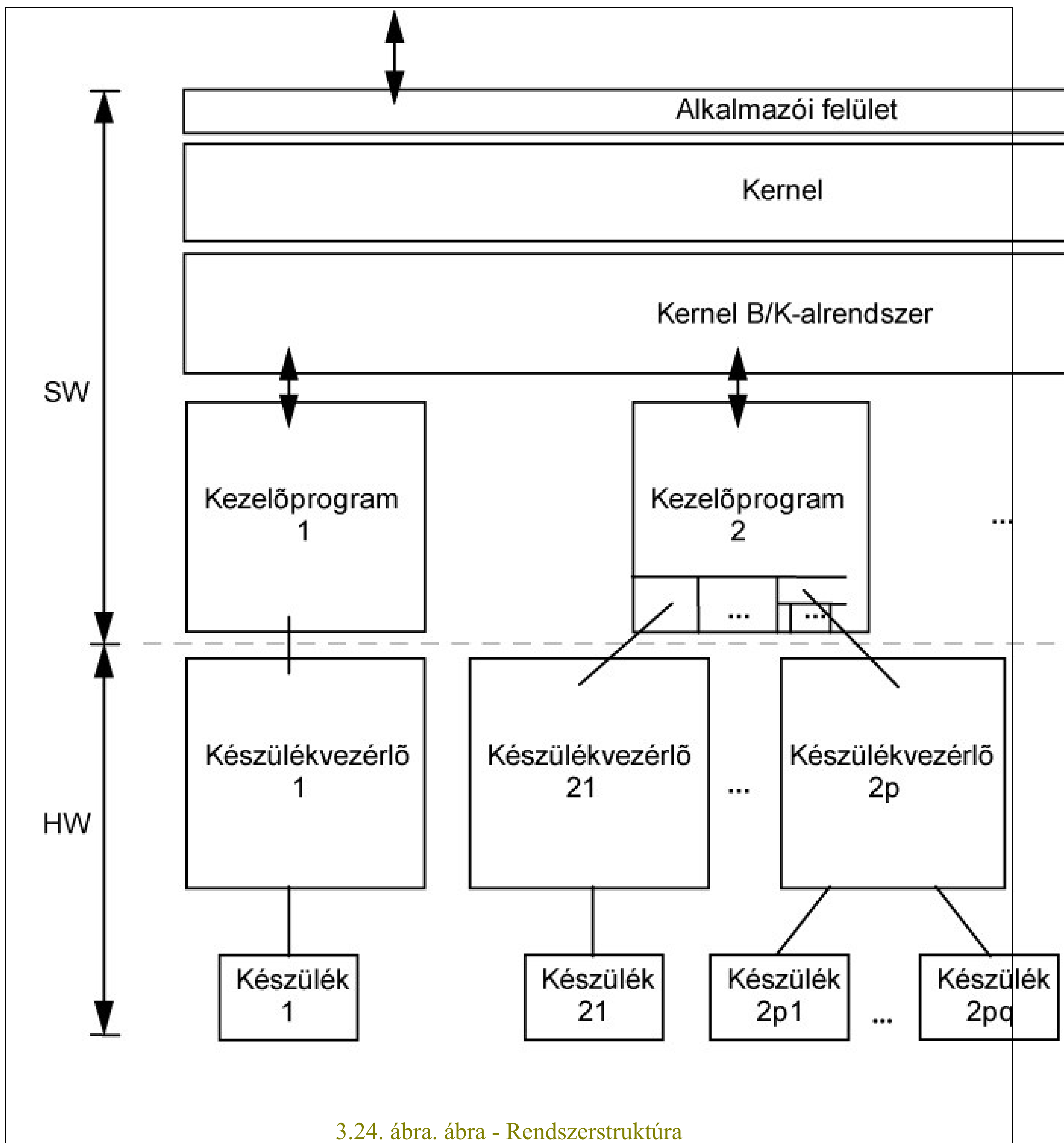
bonyolultabb vezérlők csatlakozási pontjaira történhet. Egyszerűbb eszközök és szabványos B/K-vezérlő esetén az operációs rendszer eleve tartalmazza a vezérlőegység kezelőprogramját, a működésbe helyezéshez csupán néhány paramétert kell megadni. Bonyolultabb, új eszközök esetén az operációs rendszerhez való illesztés egy speciális kezelőprogram (device driver) beillesztését igényli a rendszerbe, amelyik a periféria minden sajátosságát – beleértve a fizikai csatlakoztatás módját és adatait is – ismeri. A kezelőprogram és az operációs rendszer illesztési felületét ezért egységes, specifikált felületként kell kialakítani.

- *A készülékek hatékony működtetése.* Ennek keretében meg kell oldani a perifériák ütemezését és az átvitelek szimultán lebonyolításában a programozottan végrehajtandó feladatokat (megszakítás-kezelések, vezérlőegységek felprogramozása, állapotellenőrzések, hibakezelések, pufferelések stb.).

Ezek a feladatok az operációs rendszer belső szerkezetét is meghatározzák, és jellegzetesen a 3.24. ábrán bemutatott rendszerstruktúrára vezetnek.

A rendkívül heterogén perifériakészlet részletkérdéseitől elvonatkoztatva az alkalmazói felület a be-/kivitelre egységes, néhány paraméterrel leírható műveleteket ad.

A kernel B/K-alrendszere már valamelyest megosztottabb, az ábrán nem jelölt alrétegekre és modulokra osztható, amelyek szükség szerint ütemezéseket, puffereléseket, blokkosításokat végeznek.



3.24. ábra. ábra - Rendszerstruktúra

A rendszerben több azonos típusú eszköz működhet. Ezek illesztési módja, vezérlőegysége eltérő lehet. A legegyszerűbb esetben egy perifériához egy vezérlőegység és ahhoz egy kezelőprogram *(device driver)* tartozik. Egy kezelőprogrammal működtethetünk több vezérlőegységet, azon belül több perifériát. Ilyen esetekben is mindig vannak azonban a kezelőprogramnak legalább is olyan adatszerkezetei, amelyek a perifériákra nézve egyediek.

A perifériák működtetésének részleteit csak a kezelőprogramjuk ismeri. Ezért minden perifériatípushoz külön kezelőprogram tartozik. Egy rendszeren belül a kezelőprogramok és a kernel csatlakozási felülete szabványos, rendszerenként azonban jelentősen eltérő lehet. Ha egy új perifériát vásárolunk és csatlakoztatjuk a rendszerünkhöz, egy megfelelő kezelőprogramot is telepítenünk kell hozzá, amit vagy az operációs rendszer szállítója készített és a rendszerkönyvtárban megtalálható, vagy a periféria szállítója mellékelt a különböző operációs rendszereknek megfelelően több változatban.

Az egységes alkalmazói felületen a készülékekről a 2. fejezetben bemutatott kép rajzolódik ki. Hogyan valósítja meg ezt a modellt a kernel B/K-alrendszere? Ezt mutatjuk be a következőkben, majd a legbonyolultabb készülékek egyikének, a mágneslemeznek a kezelésével foglalkozunk, és megmutatjuk a hatékony kezelés érdekében kialakult megoldásokat.

3.5.1. A kernel B/K-alrendszere

A kernel B/K-alrendszerének jellegzetes feladatai:

- perifériák ütemezése,
- átmeneti és gyorsítótár működtetés,
- perifériahasználat koordinálása,
- hibakezelés.

A perifériák ütemezése annak a végrehajtási sorrendnek a meghatározását jelenti, amelyet a perifériára várakozó átviteli kérelmek végrehajtására alkalmaz a rendszer. Az érkezési sorrendtől való eltérés bizonyos esetekben jelentősen javíthatja a perifériahasználat hatékonyságát. Erre egy példát a következőkben a diszkműveletek ütemezésének bemutatásakor látunk.

Átmeneti tárolást (pufferelést) alkalmaz a kernel, ha

- két eltérő sebességű adatfolyamot kell összekapcsolni (gyakran kettős puffert alkalmazva),
- a periféria számára blokkosítás szükséges és az alkalmazás nem blokkokkal dolgozik, vagy más blokkméretet használ,
- az adatküldő művelet szemantikája csak átmeneti tárolással biztosítható (a művelet visszatérése után az elküldött példány módosítható legyen).

Háttértárak esetén gyorsító tár alkalmazása is jelentősen növelheti a hatékonyságot, ha ismétlődő írások/olvasások jellemzik a perifériahasználatot. (Az átmeneti tároló és a gyorsítótár közötti lényeges különbség, hogy a puffer az egyetlen létező kópiát tárolja az adatról, a gyorsítótár pedig a háttértáron lévő adat egy másolatát.)

A perifériahasználat koordinálása tekintetében az egyszerű B/K-műveletek sorbaállítása csupán ezeknek a műveleteknek az oszthatatlanságát garantálja. Külön megoldást kell találni azoknak a perifériáknak a kezelésére,

amelyek hosszabb távú oszthatatlan műveletekkel kezelhetők hatékonyan. Ilyen például a nyomtató vagy a mágnesszalag, amelyekre az alkalmazások az összetartozó adatokat általában nem tudják egyetlen B/K-művelettel elküldeni.

A probléma egyik megoldása, hogy ezekhez a perifériákhoz egy speciális puffert (spoolt) rendelnek. Egy folyamat által a perifériára küldött adatokat először egy spoolfájlban gyűjtik össze. A kész fájlokat aztán egyenként másolják ki a perifériára. Így a fájlon belüli sorrendek megőrződnek.

Más rendszerek a kizárólagos perifériahasználat kérésére biztosítanak eszközöket (lefoglalás, felszabadítás).

A B/K-műveletek közben előforduló hibákat a kezelőprogramok, vagy az operációs rendszer észleli. Mindkét észlelési szinten megkísérelhető a tranziensnek vélt hibák kiküszöbölése a művelet, vagy részművelet ismétlésével. Más hibatípusok esetén az operációs rendszernek nincs javítási lehetősége. A hibát ilyenkor jelezni kell a magasabb szintek (a hívó) felé. A hibajelzés, azaz a művelet sikertelenségének vagy sikerének visszajelzésére a könnyű kiértékelhetőség érdekében általában 1 bit szolgál. Hibajelzés esetén ezen kívül további kísérő információk segítik a hiba okának behatárolását. A hibakezelés beépített folyamatában gyakran szerepel a kezelő értesítése is. Az operációs rendszer ezt csak alkalmazás-független, általános kijelzésekkel tudja megtenni, ilyen információ megjelenése egy alkalmazás kezelőábráján rendkívül zavaró lehet. A beépített hibajelzéseket ezért a rendszer saját kezelőszervére célszerű irányítani, a hibát pedig jelezni kell az alkalmazás felé is, hogy az alkalmazáshoz illő reakciót tudja lejátszani.

3.5.2. Háttértárak kezelése

A számítógéprendszerekben azért alkalmaznak háttértárat, mert:

- a központi tár fajlagos, egy bitre jutó ára magas, ezért kapacitása viszonylag kicsi,
- a központi tárban az információk a tápfeszültség kikapcsolásával elvesznek,
- minél nagyobb a tárolási kapacitás, gyakorlati okokból annál nagyobb kell legyen az egyedileg megcímezhető információegység mérete.

A hardver fejlődésével a következő típusú háttértárak jelentek meg:

- mágnesszalag,
- mágnesdob,
- mágneslemez,
- optikai adatrögzítés (lemez szervezésű; csak olvasható (például *CD-ROM*), egyszer írható (*WORM*), illetve írható-olvasható lemezek),
- egyéb (kísérleti jellegű) megoldások:
  - mágnesbuborék tár,

- félvezető tárak, (EEPROM, memóriakártya),
- holografikus tárolás.

A háttértárak fizikai kezelése adott esetben igen bonyolult feladat. Az operációs rendszerek törekszenek a különböző típusú tárolóeszközök egységes kezelésére. Az operációs rendszer háttértár-kezelő egysége az operációs rendszer többi komponense felé a háttértárakat függetlenül elérhető adattároló blokkok sorozataként mutatja. A háttértár-kezelők működését a jelenlegi rendszerek legelterjedtebb háttértára esetén, a *mágneslemezes tár (magnetic disc)* kezelésén keresztül mutatjuk be.

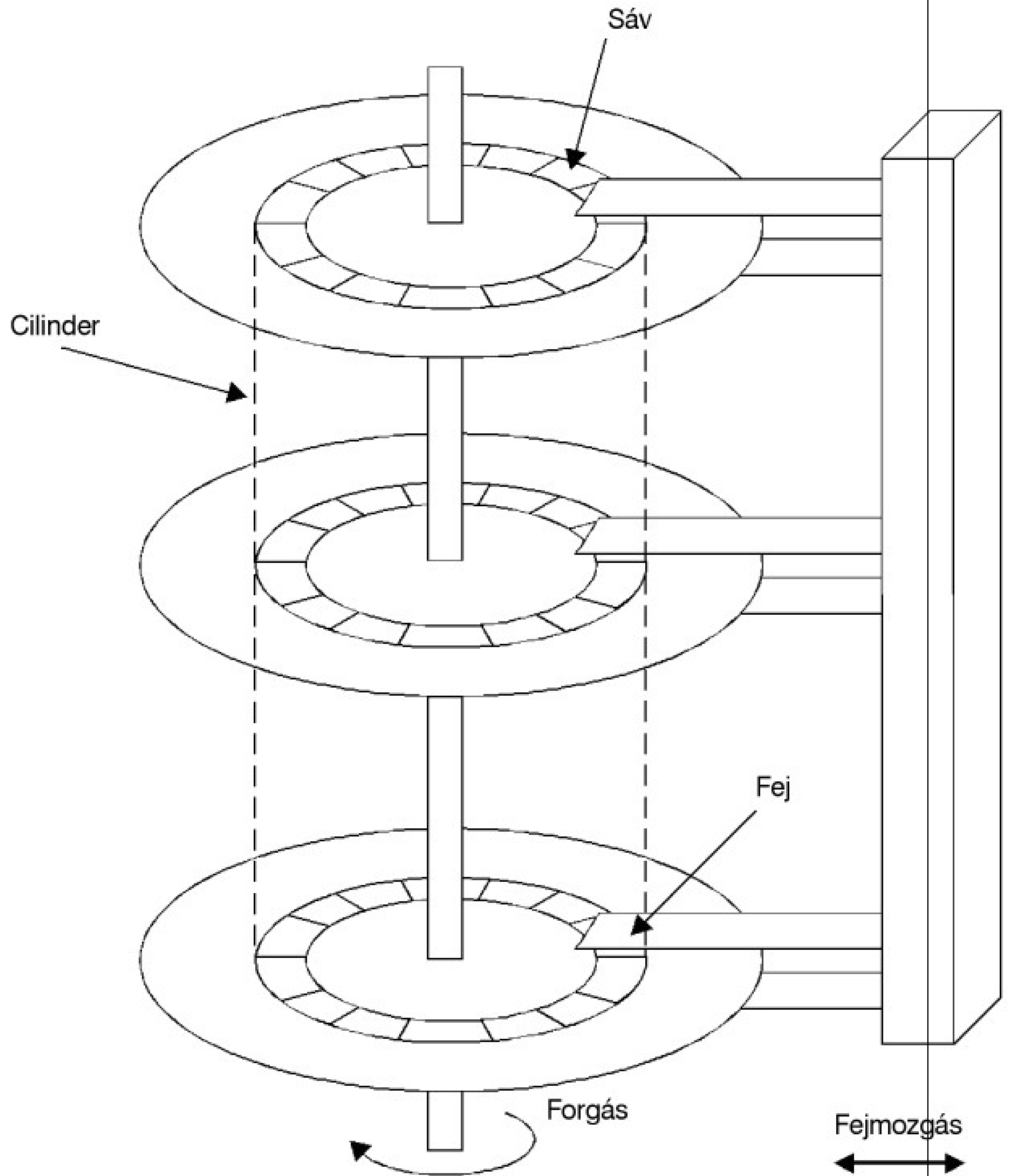
3.5.2.1. A lemezegység fizikai szervezése

A mágneslemezes egység működésének lényege, hogy a tárcsaszerű, forgó mágneses hordozó felett – oldalanként egy – író-olvasó fej mozog, a fejek mozgatását közös mechanika végzi. Általában a lemezek mindkét oldalát használják, a lemez felett és alatt is található fej. A mágneslemez sematikus felépítését a 3.25. ábra mutatja.

A 3.25. ábrán látható elnevezések magyarázata:

- Sáv (track) egy-egy lemezfelület azon területe, amelyet a fej elmozdulás nélkül, a lemez egyetlen körülfordulása alatt elér.
- Cilindernek (cylinder) nevezik a fejtartó szerkezet egy adott pozíciójában leolvasható sávok összességét.
- Egy-egy sávot – általában sávonként azonos méretű, gyakorta az összes sávon azonos számú, azonos szögtartományt elfoglaló – szektorra (sector) osztják. Az információtárolás a szektoron belül bitsoros. A szektor az információátvitel legkisebb egysége, a lemezvezérlő egyszerre egy teljes szektort olvas vagy ír. A modern lemezeken a szektorok elhelyezkedését a vezérlő a sávra felírt speciális mágneses jelekből ismeri fel.

A mágneslemez-egységek lemezei lehetnek fixek, de cserélhető lemezek (például hajlékonylemez, *floppy disc*) is használatosak.



3.25. ábra. ábra - Mágneslemez-egység felépítése

- Az átvitel kiszolgálásának ideje a következő összetevőkre osztható:
    - a *fejmozgási idő (seek time)* az az idő, amíg a fej a kívánt sávra áll,
    - az *elfordulási idő (latency time)* az az idő, amíg a kívánt szektor a fej alá fordul,

- az információ átvitelének ideje (transfer time).

A felsorolt idők között nagyságrendi különbségek vannak, a fejmozgási idő a leghosszabb.

- Szektorok címzése. A lemez szektorait az operációs rendszer lineárisan címzi, a lemezillesztő viszont több komponensű – több dimenziós – címet igényel. A kettő között az összefüggés:

$b = s * (i * t + j) + k$,

ahol s a sávon lévő szektorok száma, t a cilindereken lévő sávok száma, i a kijelölt cilinder, j a fej (lemez felület) száma, k pedig a sávon belüli szektorok száma (i, j, és k értékei 0-tól indulnak). A b eredmény a szektor lineáris, 0-tól induló sorszáma.

3.5.2.2. A lemezműveletek ütemezése

A multiprogramozott rendszerekben egyszerre több folyamat verseng a mágneslemezes perifériákért, egy átvitel lezajlása után több újabb kérés várakozhat kiszolgálásra. Az ütemezési algoritmusok a kérések megfelelő sorrendbe állításával az egyes folyamatok rovására próbálják a várakozási idők csökkentésével a rendszer teljesítményét növelni. Az algoritmusok célja a fejmozgás optimalizálása, vagy az elfordulási várakozás csökkentése.

Az algoritmusok értékelésének szempontjai:

- átbocsátó képesség, időegység alatt lebonyolított átvitelek száma,
- átlagos válaszidő, egy átvitel kérésétől a végrehajtásáig eltelt átlagidő,
- válaszidő szórása.

Elsősorban interaktív rendszerekben fontos szempont, hogy a folyamatok előre látható sebességgel fussanak, a futásuk ne ingadozzon nagyon rajtuk kívül álló okok miatt.

Az algoritmusok értékelésénél az átviteli kérések címének egyenletes eloszlását tételezzük fel.

A fejmozgás optimalizálása

Sokféle algoritmus képzelhető el, az itt következő felsorolás csak néhány ismertebb alaptípust vizsgál.

- **Sorrendi kiszolgálás** (FCFS: First Come First Served). Az átviteli kéréseket érkezésük sorrendjében szolgáljuk ki. Az algoritmus nem törődik a fej mozgásával, kicsi az átbocsátó képessége, nagy az átlagos válaszideje, de ennek szórása viszonylag kicsi.
- **Legrövidebb fejmozgási idő** (SSTF: Shortest Seek Time First). Az algoritmus következőnek azt a kérést szolgálja ki, amelyik az aktuálishoz legközelebb lévő cilinderre hivatkozik – ennek az eléréséhez szükséges a legkisebb fejmozgási idő. Bár teljesítménye az FCFS-nél jobb, de a válaszidők szórása nagy, sőt fennáll a kiéheztetés veszélye: egy távoli cilinderre vonatkozó kérést az újra és újra érkező közeli kérések nem meghatározható ideig késleltethetik.

- **Pásztázó** (SCAN). Az algoritmus a következő kérés kiválasztásánál csak azokat a kéréseket veszi figyelembe, amelyekhez szükséges fejmozgás az aktuális mozgási iránynak megfelelő. A mozgási irány akkor fordul meg, ha az aktuális irányban már nincs több kiszolgálatlan kérés. Az algoritmus teljesítménye jobb, mint az SSTF, a válaszidő szórása is kisebb. A pásztázásból következő sajátossága, hogy a középső cilindereket gyakrabban látogatja, mint a szélsőket.
- **N lépéses pásztázó** (N-SCAN). Egy irányba mozogva csak azokat a kéréseket – közülük is csak *N*-et – szolgálunk ki, amelyek a pásztázás elején már megvoltak. A pásztázás közben érkező kérésekre csak a következő irányváltás után kerül sor. Az algoritmus válaszidejének szórása kisebb a SCAN-nél is, a válaszidő akkor sem nő meg, ha az aktuális cilinderre sok kérés érkezik.
- **Egyirányú** (körforgó) **pásztázó** (C-SCAN: Circular SCAN). A kérések kiszolgálása mindig csak az egyik irányú fejmozgásnál történik, a másik irányban a fej közvetlenül a legtávolabbi kérés cilinderére ugrik. Implementálható a pásztázás közben beérkezett kérések mind menet közbeni, mind a következő pásztázásra halasztott kiszolgálásával. Az algoritmus elkerüli a külső sávoknak a belsőkhöz viszonyított alacsonyabb fokú kiszolgálását.

Egyes rendszerekben a lemez terhelésének függvényében különböző módszereket alkalmaznak, például

- alacsony terhelésnél SCAN,
- közepes terhelésnél C-SCAN,
- nagy terhelésnél az elfordulási idő optimalizálásával bővített C-SCAN.

Az elfordulási idő optimalizálása

Az egy cilinderen belüli kérések a lemezek aktuális pozíciójának, valamint a szektorok sorrendjének – ami nem mindig monoton növekvő *(szektor közbeékelődés, sector interleave)* – ismeretében kiszolgálás előtt sorbaállíthatók.

3.5.2.3. Egyéb szervezési elvek a teljesítmény növelésére

- *Lemezterület tömörítése* (*disk compaction*). Az egymáshoz tartozó blokkokat lefoglaláskor a lemezen fizikailag is egymás mellé igyekszünk elhelyezni, illetve ezt az állapotot egy időnként futtatott rendezőprogrammal elérni. A lemezműveleteknél is megfigyelhető lokalitás következményeként egy folyamat várhatóan az egymáshoz közeli – egymást követő – blokkokat fogja olvasni, így a fejmozgás minimális lesz. A tömörítés nem mindig vezet teljesítményjavuláshoz, hiszen egy multiprogramozott rendszerben egy időben sok folyamat használja a lemezt.
- A gyakran szükséges adatokat SCAN típusú ütemezésnél érdemes a lemez középső sávjain elhelyezni.

- A gyakran szükséges adatokat a lemezen több példányban, különböző sávokon helyezzük el, így minden fejállásnál kiválaszthatunk egy viszonylag közel lévő cilindert, ahol a kívánt adat megtalálható. A módszer elsősorban nem – vagy csak nagyon ritkán – változó adatoknál használható, hiszen valamennyi példány módosítása hosszú időt jelentene, és módosítás közben az adatok konzisztenciájának biztosítása is megoldandó.
- *Több blokk egyidejű átvitele.* Mivel a kérések kiszolgálásának jelentős része a fejmozgásból eredő várakozással telik, ha már a megfelelő pozíción vagyunk, igyekezzünk minél több blokkot egyszerre átvinni és azokat a memóriában tárolni.
- *Blokkok átmeneti tárolása.* A gyakran vagy a közeljövőben várhatóan szükséges blokkokat igyekezzünk a központi-, esetleg a perifériaillesztőben lévő tárban tartani *(disk cache)*.
- Az átmeneti tárolás közben foglalkozni kell a megváltozott tartalmú szektorokkal:
    - a tárban változással egyidejűleg a lemezre is felírjuk *(write through cache)* vagy
    - csak akkor írjuk ki, ha a tárra szükség lesz.
- Ez utóbbi a nagyobb teljesítményű módszer, hiszen egy gyakran változó szektor kiírás előtt újra megváltozhat, viszont kevésbé biztonságos, hiszen a rendszer meghibásodása esetén a szükséges módosítások nem kerültek a lemezre.
- *Adattömörítési (data compression) eljárások használata.* A lemezen az információt tömörített *(compressed)* formában tároljuk, visszaállítása – a programozó számára láthatatlanul – csak a beolvasásakor történik meg. Ily módon csökkenthető a szükséges perifériás átvitelek száma. A tömörítést és visszaállítást a perifériakezelő program vagy célhardver végezheti.

3.5.2.4. Az adattárolás megbízhatósága

- *Adatok mentése (backup)*. A lemez teljes vagy az előző mentés óta megváltozott tartalmát *(incremental backup)* időnként más – általában mágnesszalagos, újabban optikai lemezes – háttértárra kell kimásolni, ahonnan a lemez sérülése, a tárolt adatok véletlen törlése esetén az egész, illetve a szükséges részek visszaállíthatók.
- *Átmeneti tár és a háttértár tartalmának szinkronizálása.* Az átmeneti tárban lévő „fontosabb” változásokat, esetleg időnként az összes változást a lemezre kell írni (*sync* a UNIX-ban).
- Lemezegységek többszörözése:

– kétszerezés *(disk shadowing, mirroring):* az írásokat egyszerre mindkét lemezen elvégezzük, így a két lemez pontosan azonos információt tárol, az egyik kiesése esetén a másik még használható.

– többszörözés: egy nagy kapacitású lemezegység helyett több kicsit használnak, amelyekre az információt hibajavító kódolással – legegyszerűbb esetben paritásbittel – megtoldva szétterítik. Így egység(ek) kiesése esetén a tárolt információ még visszaállítható. *(RAID: Redundant Array of Inexpensive Disks)*.

3.6. Operációs rendszerek kezelői felülete

Az operációs rendszerek kezelői felülete felelős a felhasználóval történő kapcsolattartásért. A kezelői felületek tipikus feladatai a következők:

- felhasználói parancsok bevitele, értelmezése,
- parancsok végrehajtása,
- eredmények, illetve esetleges hibák közvetítése, megjelenítése.

A felhasználók által használható parancsok típusai:

- *Belső parancsok*. Közvetlenül az operációs rendszer kezelői felülete hajtja végre a parancsot.
- *Tárolt (külső) parancsok*. Az operációs rendszer kezelői felülete egy önálló programként megvalósított parancsot hív meg. Ezeket a parancsokat két csoportra oszthatjuk:

– rendszerparancsok: a parancs az operációs rendszerrel együtt szállított parancsok. Minden felhasználó azonos parancsot ér el.

– felhasználói parancsok: a felhasználó önmaga fejlesztette a parancsot.

A kezelői felületek fejlesztésekor a legfontosabb szempont a felhasználóbarát tulajdonság megvalósítása. Ez elsősorban széles körben használt rendszerek esetén okoz gondot, mert a rendszer felhasználói köre nagyon heterogén lehet, vagyis az egyes felhasználók eltérő igényeket támasztanak a rendszer elé.

A kezdő, tapasztalatlan felhasználó igényei általában a következők:

- kis számú és egyszerű parancsok,
- biztonságos parancsok, amelyek nem okozhatnak súlyos és visszaállíthatatlan változást a rendszerben, illetve megerősítést kérnek minden ilyen akció előtt,
- részletes és környezetfüggő segítség minden szituációban.

A tapasztalt felhasználó igényei ezzel szemben:

- hatékony, a rendszer minden lehetőségét kihasználó parancsok,
- konfigurálható parancsok, amelyeket a felhasználó saját igényeinek megfelelően meg tud változtatni,
- a felhasználó által bővíthető parancskészlet.

A fenti igények láthatóan több ponton ellentmondanak egymásnak. Ez természetesen lehetetlenné teszi, hogy egy minden igényt kielégítő felhasználói interfészt készítsünk. A felhasználói interfész fejlesztésénél általában valamilyen kompromisszumot kell kötni a fenti szempontok között. Gyakran alkalmazott megoldás, hogy a felhasználói interfész valamilyen paraméter állításával lehetőséget ad a felhasználó tapasztaltságának beállítására, ezzel biztosítva a felhasználó igényeihez való igazodását.

Az operációs rendszerek és a felhasználó közötti kommunikáció eszközei:

- Nem interaktív rendszerek:

– Job Control Card

- Interaktív karakteres interfészt biztosító rendszerek:

– billentyűzet (keyboard),

– karakteres terminál (display).

- Grafikus be-/kimenetet kezelő rendszerek:

– egér,

– fényceruza,

– érintő képernyő (touch-screen),

– grafikus terminál *(display)*.

- Hang be-/kimenet.

A fenti felsorolás a rendszerek fejlődését is mutatja. A számítógépek kapacitásának gyors fejlődésével az operációs rendszerek mind nagyobb súlyt fektettek a felhasználói felületek fejlesztésére. A fejlődés első lépcsője az interaktív rendszerek megjelenése volt, majd a grafikus interfészt biztosító rendszerek következtek. A mai napig is tart az emberi kommunikációra mindinkább hasonlító kezelői felületet biztosító rendszerek fejlesztése. A beszédfelismerésen illetve beszéd szintézisen alapuló kommunikációt biztosító rendszerek egyre nagyobb teret hódítanak, hiszen alapvető beszédfelismerési funkciókat akár egy mobiltelefon kínálta számítási kapacitással is lehetséges megvalósítani.

A továbbiakban egy konkrét példán, az *X Window*-rendszeren keresztül mutatjuk be egy grafikus interfész felépítésének és működésének részleteit.

3.6.1. Az X Window-rendszer

Az X Window egy olyan rendszer, mely grafikus kimenettel rendelkező alkalmazások felhasználói felületének kialakítására ad lehetőséget. A rendszer fejlesztése 1983–1984-ben kezdődött a Massachusetts Institute of Technology-n (MIT, USA). A fejlesztés célja olyan kommunikációs felület készítése volt, ami azonos kezelői felületet biztosít a hálózattal összekötött, különböző operációs rendszereket futtató számítógépeken. Az X

Window segítségével lehetőség van az alkalmazás és a kezelői felület szétválasztására. Míg a kezelői felületet a helyi gépen futó X szerver jeleníti meg, addig az alkalmazás akár távoli gépeken is futhat.

A rendszer gyors elterjedését számos előnyös tulajdonságán túl segítette, hogy az X Window forráskódja publikus. Az X Window-t elsősorban a UNIX rendszerek támogatják.

Az X Window működésének jellemzője a kliens-szerver modell használata. A szerver egy grafikus terminálon futó folyamat, mely grafikus ki- és bemeneti lehetőséget biztosít a kliens folyamat számára. A szerver kezeli az ún. *grafikus munkahelyet*, melynek részei:

- a képernyő (illetve képernyők),
- a billentyűzet (alfanumerikus bemeneti eszköz), és
- egy grafikus bemeneti eszköz.

A kliens egy grafikus be-/kimenetet igénylő (általában interaktív) folyamat.

3.6.1.1. Az X protokoll

A rendszer magja az X protokoll, amely definiálja a kliens és a szerver együttműködésének módját. Leírja a lehetséges grafikus funkciókat, valamint a megengedett akciókat.

A protokoll kétirányú aszinkron kommunikációt tesz lehetővé, vagyis mind a kliens, mind a szerver küldhet üzeneteket. Az üzenetek küldése után egyik fél sem várakozik visszajelzésre, hanem folytatja működését.

Az üzenetek típusai a következők lehetnek:

- *kérés* (kliens küldi a szervernek),
- *válasz* (szerver küldi a kliensnek),
- *esemény* (szerver küldi a kliensnek),
- *hiba* (szerver küldi a kliensnek).

A protokoll legfontosabb jellemzője, hogy a hálózati kommunikáció mérséklésére törekszik. Ennek módjai:

- A kliens nem egyes üzeneteket, hanem üzenetek összegyűjtött csomagját küldi át a hálózaton.
- A szerver helyben kezel bizonyos egyszerű eseményeket, mint például az egérmozgatást.
- A szerver szoftver-erőforrásokat (grafikus környezetet, betűtípushoz tartozó leírást stb.) hoz létre a kliens kérésére, amiket később a kliens folyamat egyszerű hivatkozással érhet el.

3.6.1.2. Az X Window-rendszer koncepciója

Az *X* környezet alapvető eleme a grafikus, ún. *X munkahely*, mely egy *X display*-ből (*X* megjelenítőből) és az azon levő (egy vagy több) *X screen*-ből (*X* képernyőből), valamint bemeneti eszközökből áll. Az *X* munkahely

egy *karakteres bemeneti* eszközt (általában billentyűzet) és egy pozicionálásra alkalmas *grafikus bemeneti* eszközt kezel. A grafikus bemeneti eszköz leggyakrabban egér, de lehet fényceruza vagy érintő képernyő is.

3.6.1.3. Ablakkezelés

A grafikus képernyő kezelése *ablakok* létrehozásával történik. Az ablak egy téglalap alakú képernyőrészlet, melyben a felhasználó újabb ablakokat nyithat, vagy az egyes ablakok területére rajzolhat.

Az ablakok rendszere hierarchikus. A kliens a működése elején nyit egy ún. gyökér ablakot. Az összes többi ablaka ennek az ablaknak lesz leszármazottja. Az ablakok mozgathatók a képernyőn. A leszármazott ablakok területe csak az ősük ablakfelületén látszik. A kilógó vagy esetleg átlapolódó, egymást fedő ablakoknál a rendszer automatikus vágást alkalmaz. Egy ablakfelület kitakarásakor (láthatóvá válásakor) a szerver ún. *kitakarás eseményt* küld a kliens folyamatnak, lehetőséget adva az ablak tartalmának frissítésére. A kliens külön kérésére a szerver támogathatja az ablak automatikus, szerver által megvalósított frissítését.

3.6.1.4. Bemeneti eszközök kezelése

A bemeneti eszközöket az *X* szerver figyeli. Az eszközök állapotváltozásakor (például egy billentyű vagy egér gombjának leütésekor) a szerver esemény üzenettel értesíti a klienst.

A bemeneti információ elosztására az *X* Window-rendszer bevezette az ún. *input focus* fogalmát. Az *input focus* birtokosa az a kitüntetett kliens folyamat, amelyik a bemeneti eszközök állapotváltozásakor a szerver által értesítendő. Az *input focus*t mindig egy folyamat birtokolja, és a szerver által definiált módon adható át, illetve kérhető el a kliensek között.

A pozicionáló eszköz mozgásáról a kliens az aktív ablak bal felső sarkától számított relatív koordinátákban mért információt kap a szervertől. Karakteres bevitel esetén a billentyűkhöz történő karakter-hozzárendelést a szerver végzi, azonban a kliens kérésére a szerver megváltoztathatja az aktuális karakter-hozzárendelést.

3.6.1.5. Megjelenítő eszköz kezelése

Az *X* Window ún. *raszteres* (*képpontokból*, *pixelekből* álló) grafikus terminált tud kezelni. Az ablakok helyét a képernyő bal felső sarkától számolt derékszögű koordináta-rendszerben tartja számon. Az ablakok és egyéb rajzelemek elhelyezkedését képpontokban méri. Az ablakon belüli rajzelemek elhelyezkedését az ablak bal felső sarkától mért relatív koordináták szerint tárolja.

Rajzoláskor az *X* egyszerű, előre definiált rajzelemek használatát engedi meg. A rajzelemek halmaza azonban bővíthető a felhasználó által. Rajzolni mind a képernyőre, mind pedig (virtuálisan) a memóriába lehetséges. Virtuális rajzolással a kliens karbantarthatja az esetlegesen letakart képernyőjének tartalmát, melyet majd a kitakarás esemény után frissíthet.

Az X Window a színek használatát ún. *palettázással* támogatja. A kliens egy 128 elemű paletta színeit definiálhatja a szerver által biztosított színtartományban, mely általában igen széles. A palettahasználat előnye,

hogy egy palettaszín definiálása után a kliens egy nyolcbites azonosítóval hivatkozhat egy színre, mely szín ábrázolására a szerver 16, 24, vagy akár 32 bitet is használhat. A paletta színein az azonos képernyőn futó alkalmazások osztoznak. Ha egy alkalmazásnak nem elegendő a palettában megmaradt színek száma, kérheti a szervert, hogy biztosítson számára egy külön palettát. Ez a felhasználó számára is szembetűnő, mert ebben az esetben a képernyő színei megváltoznak, attól függően, hogy az önálló palettát használó vagy valamelyik másik, közös palettát használó folyamat birtokolja-e az *input focus*t.

3.6.1.6. A kezelői felület elemei

Egy működő X rendszernek három fő eleme van:

- *Windowing system.* A rendszernek ez az eleme felelős az *X* protokoll megvalósításáért, vagyis ez a rendszer magja. Lebonyolítja a kommunikációt a kliensekkel, elvégzi a megjelenítési funkciókat.
- *Window manager.* A *Window manager* egy kitüntetett kliens folyamat, ami az ablakok felhasználó által történő manipulálását intézi. Minden *Windowing system*hez csak egy *Window manager* kapcsolódhat.

A *Window manager* által támogatott műveletek az ablakokkal:

– mozgatás,

– méretezés,

– zárás,

– ikonizálás,

– menü biztosítása.

A *Window manager* minden ablakot ellát az ablakok kezelését (mozgatását, ikonizálását stb.) megkönnyítő tartozékokkal. A tartozékok kinézete, illetve a biztosított funkciók az adott *X* megvalósítástól függően különbözhetnek.

- *Session manager.* A *Session manager* egy állandóan futó X szerver esetén a felhasználó beléptetését intézi a grafikus képernyőn. Opcionális része a rendszernek. *Session manager* használata esetén a felhasználók nem a megszokott karakteres *login promptot* használják belépéskor, hanem a *Session manager* által biztosított grafikus környezetet.

A *Session manager* funkciója a felhasználó kényelmének növelésén túl annak a lehetőségnek biztosítása, hogy a felhasználó választhasson az adott rendszerben rendelkezésre álló *Window manager*ek között, vagyis lehetővé tegye a felhasználónak a grafikus környezetének beállítását.

4. Hálózati és elosztott rendszerek

Tartalom

4.1. Bevezetés
4.2. Hálózati architektúra
4.2.1. Alapfogalmak
4.2.2. A hálózatok topológiája
4.2.3. A hálózatok típusai
4.2.4. A hálózati kommunikáció rétegei
4.2.5. Címzés és forgalomirányítás
4.3. Hálózati jellegű szolgáltatások
4.3.1. Telnet: távoli terminál
4.3.2. FTP: fájlátvitel
4.4. Elosztott szolgáltatások
4.4.1. Jellemzők
4.4.2. Elosztott fájlrendszerek
4.4.3. Folyamatkezelés
4.4.4. Időkezelés és koordináció elosztott rendszerekben
4.4.5. Elosztott rendszerek biztonsági kérdései

4.1. Bevezetés

Mára a számítógépes hálózatok, az elosztott rendszerek szerves részét képezik életünknek. Ebben nagy szerepet játszott az Internet, ami az elmúlt közel két évtizedes élete során százmilliókat ismertetett meg a számítógépes hálózatokkal és elosztott rendszerekkel. Bár ezen felhasználók nagy része valóban csak felhasználó, azonban ez a hatalmas érdeklődés a fejlesztésnek, az új műszaki megoldások kutatásának is nagy lendületet adott.

A számítógépes hálózatok és elosztott rendszerek gyors terjedésével egy újfajta számítási, feldolgozási módszer alakult ki. Míg kezdetben, a hálózatok alkalmazása előtt a számítási modell nagyon egyszerű volt: az adott feladatot egyetlen gép számítási kapacitásával kellett megoldani, addig a hálózatok megjelenésével a modell bővülhetett, a számítások elvégzésére már több együttműködő processzort, vagy számítógépet tartalmazó elosztott rendszer áll rendelkezésre. Ez a változás persze nem robbanásszerűen ment végbe. Kezdetben a számítógépek összekapcsolásának elsődleges célja az adatok számítógépek közötti továbbítása volt. Azonban nagyon hamar felismerték, hogy az elosztottság több előnnyel is kecsegtet: egyrészt a terheléselosztás megvalósítása, másrészt pedig hibatűrő viselkedés, nagyobb rendelkezésre állás, és más hasonló szolgáltatások területén is nagy előrelépést jelenthet. A ’80-as évek közepére kialakult a ma is egyik legelterjedtebben alkalmazott számítási modell, az elosztott architektúra előnyeit kihasználó *kliens–szerver*-modell. Ebben a megközelítésben a rendszerünkben bizonyos kitüntetett feladatokat egy-egy adott *szerver* végez el a *kliensek* kérésére. Amennyiben például egy állományhoz szeretnénk hozzáférni, akkor azt egy fájlszerver, ha nyomtatni szeretnénk, akkor azt egy nyomtató szerver tudja a számunkra biztosítani. A kliens–szerver-modell mellett megjelentek az objektumorientált számítási modellek is.

Az elosztottság az operációs rendszerekkel szembeni elvárásainkat is módosította. Az elosztott operációs rendszer egyik fő feladata, hogy egy elosztott rendszerben lehetővé tegye annak kényelmes programozását, és ne korlátozza a rajta megvalósítható alkalmazások körét. Ehhez az elosztott rendszerben rendelkezésre álló

erőforrásokat az alkalmazások számára általános, problémaorientált absztrakciókkal írja le. Például a hálózatok és processzorok helyett engedje meg, hogy a felhasználó kommunikációs csatornákban és folyamatokban gondolkodjon.

Egy nyílt elosztott rendszerben az elosztott operációs rendszer mikrokernelek és szolgáltatások halmazából épül fel. Nem lehet éles határvonalat húzni az operációs rendszer és a rajta futó alkalmazások között. Ebben az esetben a határvonal még inkább elmosódott, mint a hagyományos operációs rendszerek esetén.

Az elmúlt években számos kutatás foglalkozott az elosztott operációs rendszerek kérdéskörével. Ezek közül néhány, mint például a *Mach* és a *Chorus*, mind műszaki, mind pedig kereskedelmi szempontból is az érdeklődés középpontjába került, az általuk felvetett műszaki megoldások számos ponton radikálisan új megoldásokat kínáltak. Az *Amoeba*, a *Clouds* és a *V* rendszerek szintén elosztott operációs rendszer kutatás eredményei, ezek elsősorban érdekes műszaki megoldásaikkal hívták fel magukra a figyelmet. Mind az öt projekt egy ún. *minimálkernelre* vagy *mikrokernelre* alapozta a rendszert.

A 4. fejezet az alábbi témákra bomlik.

A 4.2. alfejezet bevezeti az olvasót a hálózatok alapfogalmaiba, ismerteti a topológiák és a kapcsolatok alaptípusait és rámutat a név- és címkezelés legfontosabb problémáira és megoldásaira. A 4.3. alfejezet ismerteti a hálózati alapszolgáltatásokat. A 4.4.1. rész részletesen ismerteti az elosztott rendszerek alapvető jellemzőit, és a tervezésüknél figyelembe veendő legfontosabb tervezési szempontokat. A 4.4.2. részletesen tárgyalja az elosztottság hatását az egyik legfontosabb operációs rendszer komponensre, az állományrendszerre. Bemutatja a *fájlszerver*-modellt és példát ad a megvalósítási lehetőségekre. A 4.4.3. alfejezet rámutat a folyamatkezeléssel szembeni elvárásokra elosztott környezetben, tárgyalja a kliens–szerver folyamatok, a távoli eljáráshívások, proxyk és démonok használatának előnyös tulajdonságait. A 4.4.4. rész az időkezelésnek és a koordinációnak az elosztottságból eredő problémáit és azok megoldásait szemlélteti. Az elosztottság és a nyílt hálózatok egy jelentős problémát állítanak előtérbe: a biztonsági kérdéseket. Az új környezetben át kell gondolni a korábban alkalmazott biztonságpolitikát és biztonságtechnikai eszközöket, és ezeket hozzá kell igazítani az elosztott környezet követelményeihez. A 4.4.5. rész részletesen tárgyalja az elosztott rendszerek biztonsági kérdéseit, és ismertet egy kliens–szerver-modellre épülő hitelesítő rendszert.

4.2. Hálózati architektúra

4.2.1. Alapfogalmak

A számítógép hálózat fogalma általában számítógépek és perifériák (például nyomtató) egy adott halmazának valamilyen eszközzel történő összekötését takarja. A kapcsolat lehet közvetlen (egy kábellel kialakított), vagy közvetett (a hálózati kapcsolat kialakítását segítő eszközökön, például modemen keresztüli).

A hálózatba kapcsolt komponenseket **csomópontoknak** nevezzük. A csomópontokat kommunikációs hálózatok kötik össze, melyek az átvitelt biztosító **vonalakból** és **kapcsolóelemekből** állnak. Az átvitelt biztosító

vonalak neve **csatorna, vonal, vagy trönk**. A kapcsolóelemek vagy a hálózatba kapcsolt gépek részei (például hálózati kártya), vagy önálló speciális berendezések (útvonalválasztók, hidak stb.). A hálózati csomópontok összekapcsolási rendje az ún. **hálózati topológia**.

A hálózati adatcsere (kommunikáció) architektúrája réteges felépítésű: a kommunikáció egyes szintjeit megvalósító algoritmusok egymástól elkülönülnek, közöttük interfészek biztosítják a kapcsolatot. A rétegek egymásra épülnek, a kapcsolatban részt vevő két fél azonos szinten levő rétegei kommunikálnak egymással az alsóbb rétegek szolgáltatásait igénybe véve. Az azonos rétegek közötti kommunikációban használt adatformátumok és párbeszéd szabályok összességét **protokollnak** nevezzük.

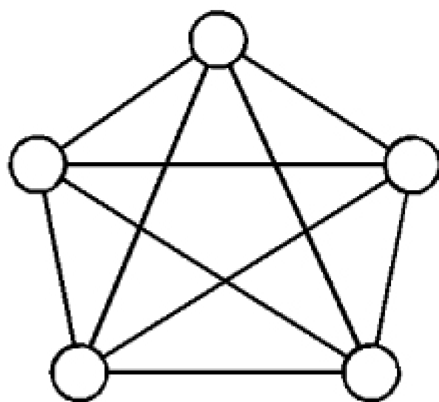
4.2.2. A hálózatok topológiája

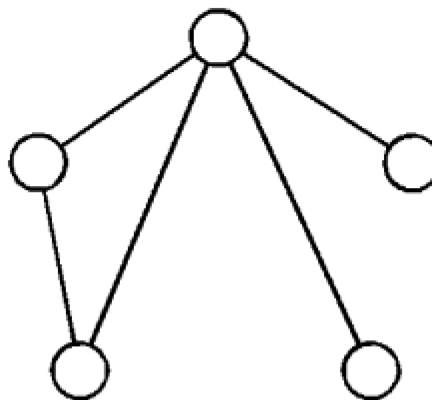
A számítógép hálózatok csomópontjainak összekapcsolását, azaz a hálózat topológiáját többféle szempont szerint lehet értékelni. A csomópontok közötti kapcsolatok kiépítettsége, a kommunikáció sebessége és az adattovábbítás megbízhatósága a legfontosabb szempontok.

A kapcsolat kiépítettsége szerint a hálózatokat két nagy kategóriába soroljuk:

- *Teljesen összekapcsolt (fully connected).* A rendszert alkotó összes csomópont között közvetlen kapcsolat van. A kapcsolat kiépítésének költsége magas, négyzetesen arányos a csomópontok számával. A kommunikáció gyors, hiszen csak egy csatorna kell az üzenetnek a címzetthez juttatásához; megbízható, mivel a közvetlen csatorna meghibásodása esetén két csomópont között sok egyéb, más csomópontokat érintő út van.
- *Részlegesen összekapcsolt (partially connected).* Nincs minden csomópont-pár között közvetlen kapcsolat. A hálózat építésének költsége kisebb, mint a teljesen összekapcsolt esetben, de a kommunikáció is lassabb, egyes üzenetek csak több közvetítő csomóponton keresztül juthatnak el a címzetthez.

A hálózat egyes csatornák meghibásodására érzékeny lehet, a teljes hálózat több részhálózatra eshet szét, amelyekben lévő csomópontok csak egymással tudnak üzenetet váltani, a másik részben lévőkkel nem. A kritikus, könnyen meghibásodó csatornák mellé érdemes kerülőutakat biztosítani.





4.1. ábra. ábra - Teljesen és részlegesen összekapcsolt hálózat

A részlegesen összekapcsolt hálózatok a következő alaptopológiákból épülhetnek fel:

- *Hierarchikus.* A csomópontok közötti kapcsolatok faszerkezetűek, minden csomópontnak – a legfelső kivételével – van egy szülő és néhány gyerek csomópontja. Csak a szülő és a gyerekek tudnak közvetlenül üzeneteket váltani, a testvérek, rokonok csak a megfelelő ősökön keresztül üzenhetnek egymásnak.

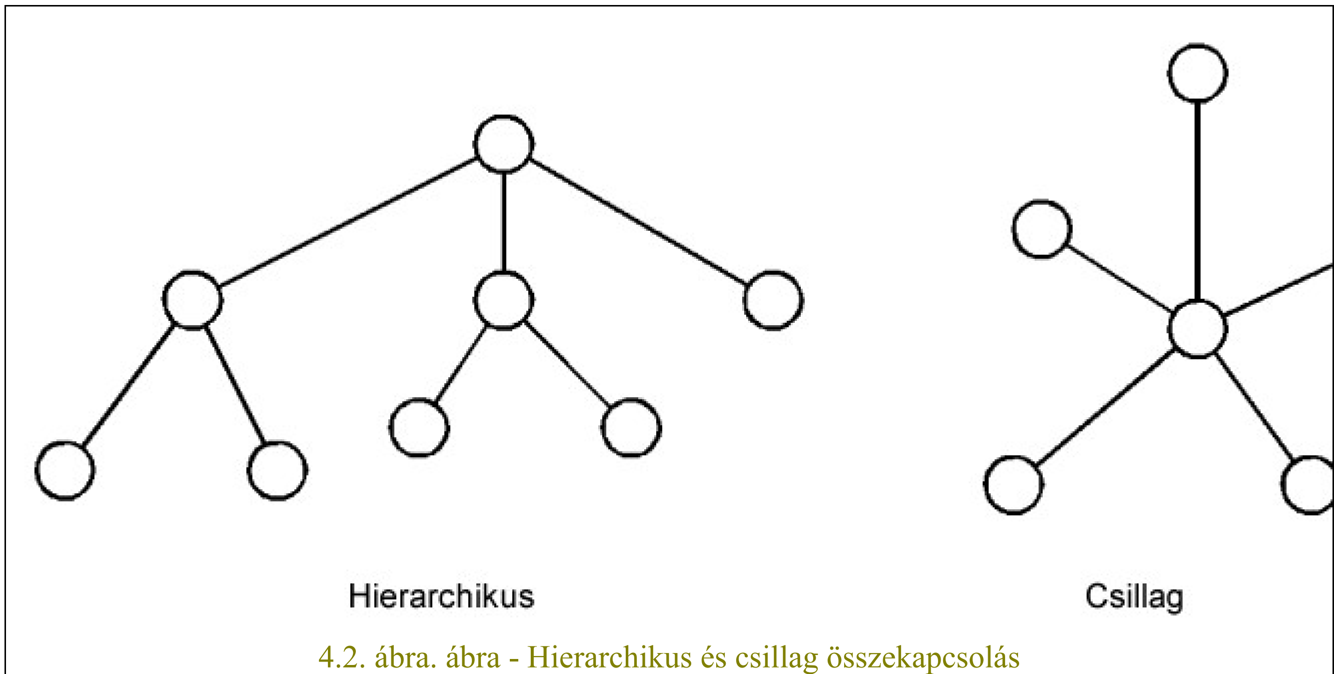
Egy csatorna vagy egy – üzenettovábbító – szülő állomás kiesése esetén a hálózat részekre szakad.

- *Csillag (star).* A csillag hálózatban van egy központi csomópont, amely az összes többi csomóponttal közvetlen kapcsolatban áll, a többiek viszont csak a központi csomóponthoz csatlakoznak, más közvetlen kapcsolatuk nincs.

A kapcsolatok kiépítésének költsége viszonylag kicsi, két állomás közötti kommunikáció viszonylag gyors – csak a központi csomópont közreműködését igényli –, viszont a központ kiesése esetén a csomópontok magukra maradnak, túl sok csomópont üzenetei pedig túlterhelhetik a központot.

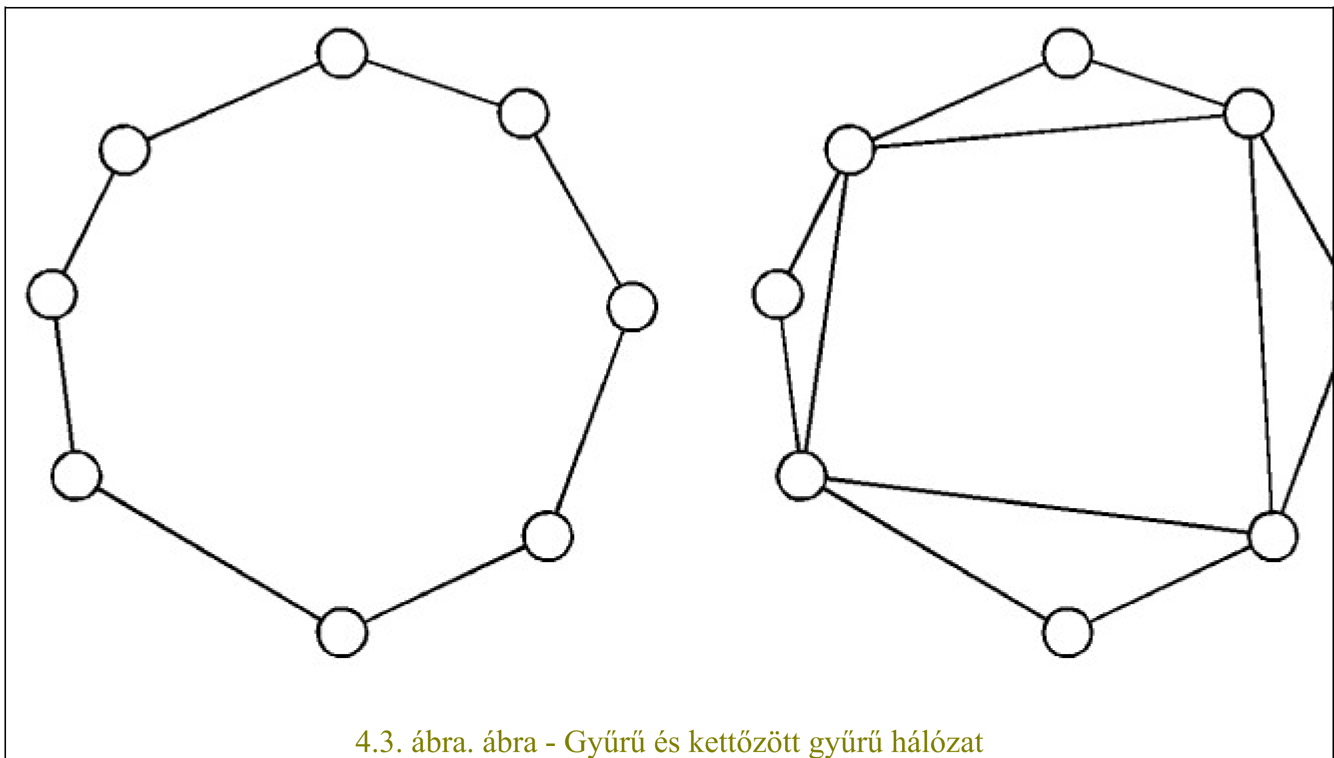
- *Gyűrű (ring).* A gyűrű olyan topológia, ahol minden csomópont pontosan két másikhoz kapcsolódik. Létezik egyirányú – ahol minden csomópont ugyanabba az irányba továbbítja az üzeneteket –, illetve kétirányú gyűrű.

A hálózat kiépítési költsége az állomások számával lineárisan arányos. Az üzenettovábbítás lassú, sok csomópont közvetítésére lehet szükség: legrosszabb esetben egyirányú gyűrű esetén *n-1*, kétirányú gyűrűnél *n/2* átvitel kell.



4.2. ábra. ábra - Hierarchikus és csillag összekapcsolás

Egyirányú gyűrűnél egy, kétirányú gyűrűnél pedig két csomópont kiesése esetén szakad részekre a hálózat. Ez ellen kettőzött gyűrűvel védekeznek. Ezzel együtt a gyűrű hálózatok rendkívül érzékenyek a csomópontok, vagy a köztük lévő kapcsolatok kiesésére.


4.3. ábra. ábra - Gyűrű és kettőzött gyűrű hálózat

- *Vezérjeles gyűrű (token ring).* A gyűrűhöz hasonló kialakítású hálózat, melyben egy hurkolt kábelközpontban történik a gépek összekötése (azaz minden gép egy központi egységbe

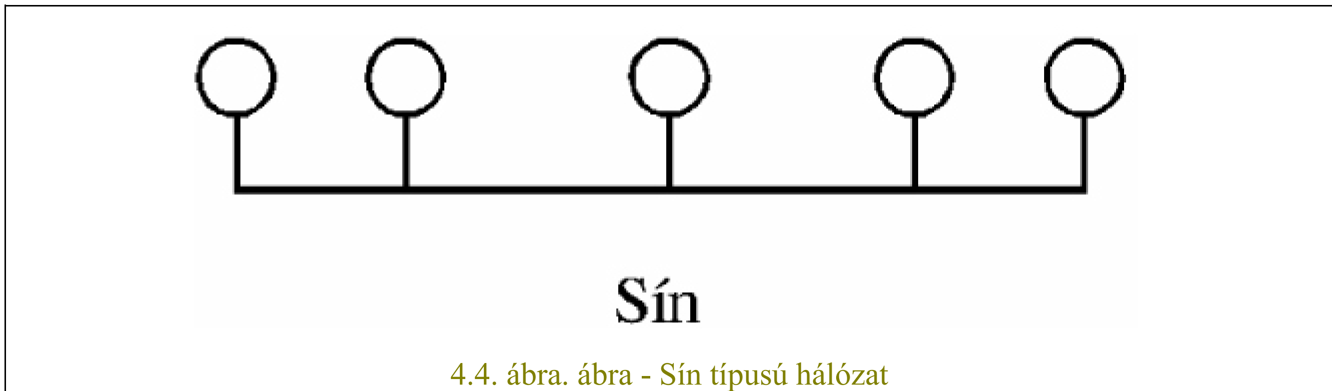
csatlakozik). A vezérjel egy speciális adat, mely állandóan körbejár a hálózatban, és az az állomás kap engedélyt kommunikáció kezdeményezésére, mely az adott pillanatban a vezérjelet birtokolja.

A hálózat kiépítési költség szempontjából a csillag topológiához hasonlít, üzenettovábbítás szempontjából gyűrű topológiájú.

Megbízhatósága lényegesen nagyobb a gyűrű topológiánál, mivel csak egy virtuális gyűrűt tartalmaz, fizikai topológiája csillag.

- *Sín (bus)*. A sín topológiájú hálózatban az összes állomás egy közösen használt kommunikációs csatornára kapcsolódik. Bármelyik két csomópont a csatornán keresztül közvetlenül válthat üzeneteket.

A gyűrű topológiához hasonlóan a hálózat kiépítésének költsége az állomásszámmal lineárisan nő, a kommunikáció gyors, de a csatorna könnyen telítődhet. Egyes állomások kiesése a többit nem érinti, a csatorna meghibásodása viszont katasztrofális.



4.4. ábra. ábra - Sín típusú hálózat

A csillag topológiák váltak a mai lokális hálózatok legelterjedtebb megoldásává, mivel rugalmasak, könnyen bővíthetőek, viszonylag kis költséggel telepíthetőek (különösen a ma elterjedt integrált kábelezési rendszerekben, ahol nem csak a számítógépes, de a telefon és épület-felügyeleti kábelezés is egy rendszer része). A csillag topológia szinte teljesen kiszorította a busz és gyűrű megoldásokat, és egy új topológia kialakulásának alapját képezte: a kapcsolt hálózatokét.

- *Kapcsolt (switched)*. A kapcsolt hálózatok topológiája lényegében csillag alakú, azzal a különbséggel, hogy a központban egy speciális hálózati eszköz, a kapcsoló *(switch)* foglal helyet. A kapcsoló a csomópontok közötti hálózat kialakítását a hálózatban adott pillanatban található csomagoknak megfelelően alakítja ki, a feladókat közvetlenül összekötve a vevőkkel. Ily módon a kapcsolt topológia az állomások között olyan pont-pont kapcsolatot valósít meg, mely dinamikusan változtatható az adatátviteli igényeknek megfelelően.

A hálózat kiépítésének költségét a csillag topológiához képest csak a kapcsolóeszköz beépítése növeli.

A kommunikáció sebessége nagy, mivel egyszerre kevés eszköz használja ugyanazt a vonalat, és az állomások a kapcsolón keresztül pont-pont módon köthetők össze. A kapcsoló képes különböző sebességű vonalak összekötésére is.

Az eddigiekben felsorolt alap topológiákat felhasználva komplex hálózati topológiákat építhetünk. Az alap topológiák kizárólag kis, helyi hálózatok számára alkalmasak, *skálázhatóságuk (azaz a hálózatba kapcsolt gépek számának és a hálózat méretének lehetséges növelése)* erősen korlátozott. Az egyedi igényeknek megfelelően kell egy skálázható, komplex topológiát kialakítani ezen alap építőelemek felhasználásával.

4.2.3. A hálózatok típusai

A hálózatokat kiterjedésük alapján is lehet osztályozni:

- *Helyi hálózat (local area network, LAN).* A helyi hálózatok tipikusan egy, illetve néhány szomszédos épületet fednek le (néhány kilométer kiterjedésűek), átviteli sebességük nagy (másodpercenként 10 Mbit és 1 Gbit között van). A tipikus helyi hálózati topológia sín, gyűrű vagy csillag, a tipikus átviteli közeg a csavart érpáros kábel (UTP).
- *Nagy területű hálózat (wide area network, WAN)*. A nagy területű hálózatok helyi hálózatok összekapcsolásával jönnek létre. A helyi hálózatok közötti kapcsolatok általában nagysebességű (100 Mbit feletti) vonalak, bár nem ritka a modemes, illetve más telefonvonalas összeköttetés sem. Általában elmondható, hogy a nagy hálózatok sebessége az egyéni felhasználók szemszögéből nézve egy-két nagyságrenddel kisebb a helyi hálózatokénál.

4.2.4. A hálózati kommunikáció rétegei

Egy alkalmazás teljes hálózati kommunikációját megvalósító szoftver egy olyan bonyolult rendszer, melyben egyszerre többféle feladattal és hardver architektúrával kell foglalkozni. Emiatt a szoftvert jól meghatározott részkomponensekre kell bontani.

Ezen komponensek egy része a hálózati eszközökben (kapcsolókban, útvonal választókban) található, más részük a hálózatba kapcsolt számítógépek operációs rendszereinek része, míg harmadrészben az adott alkalmazás szoftverébe épülnek be. A kommunikációt végző szoftver ilyen felbontását a számítógép-hálózati kommunikáció réteges szervezése segíti. A rétegek *(layers)* a kommunikációs rendszer egymástól jól elkülönülő, független részei, melyek szabványos interfészekkel kapcsolódnak egymáshoz.

Többféle modell létezik a rétegszerkezet kialakítására, mint például az **Open Systems Interconnection (OSI) Referencia Modell**, vagy az Internet világában használatos **TCP/IP** rendszer.

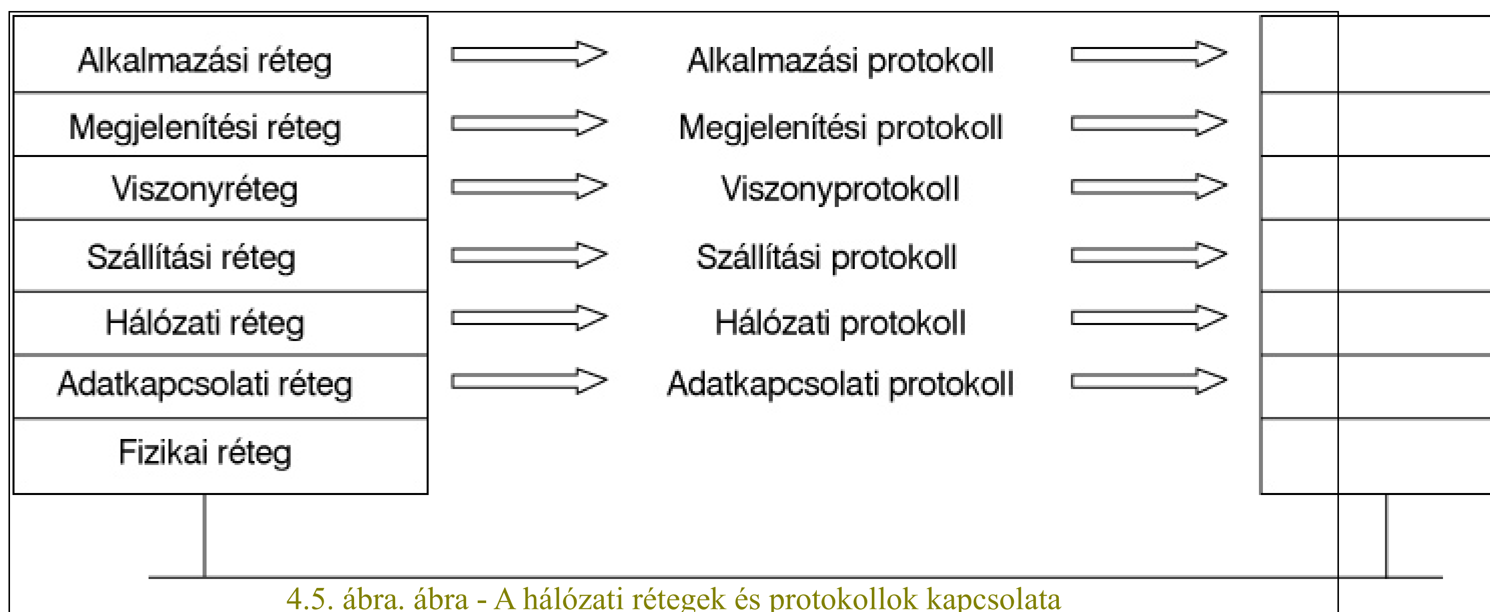
Az OSI-modell hét réteget határoz meg:

- *Fizikai (physical)*. A legalsó szint, az adatok átvitelére szolgáló mechanikai, elektromos, és egyéb jellemzőket leíró modul. Ez a réteg határozza meg az átvitel időzítéseit, a kapcsolat irányát,

felépítésének és lebontásának módját. A réteg többféle fizikai hordozó közeget használhat az elektromos jelektől a fényjeleken át a rádióhullámokig.

- *Adatkapcsolati (data link)*. A fizikai réteg szolgáltatásainak igénybevételével az adatok megbízható átküldését végzi el, azaz észleli és lehetőség szerint javítja a kommunikáció során bekövetkező adatátviteli hibákat. Az átviendő adatokat keretekbe (bájtcsoportokba) szervezi, ellátja őket kiegészítő adatokkal (cím, ellenőrző összeg stb.), és elküldi a vevő felé. A vevőtől fogadja az adatvételt igazoló keretet.
- *Hálózati (network)*. Az adatok fizikai útját határozza meg a célállomásig. Több lehetséges út esetén kiválasztja a legmegfelelőbbet (útvonalválasztás, *routing*).
- *Szállítási (transport).* Feladata a gépek közötti transzparens adatátvitel megvalósítása. A kommunikációban részt vevő gépek elől teljesen elfedi a hálózatot, annak topológiáját, és olyan kapcsolatot biztosít számukra, mintha pont-pont összeköttetésben lennének. Ez a réteg gondoskodik arról, hogy a fogadott adat pontosan megegyezzen az elküldött adattal. Hiba esetén újraküldi az adatot.
- *Viszony (session).* A kommunikációban részt vevő gépek a kommunikáció során valamilyen viszonyban állnak egymással (azaz a kommunikáció nem csak egy üzenet küldését és fogadását, hanem ezek sorozatát is jelenti). A viszony réteg feladata a párbeszéd megszervezése, a helyes szinkronizáció megteremtése, a két fél műveleteinek összehangolása, egymás állapotának megismerése.
- *Megjelenítési (presentation).* A réteg az adatok alkalmazástól független, egységes kezeléséért felelős. Az alkalmazás különböző típusú adatait egységes, közös reprezentációra hozza, szükség esetén tömöríti, illetve titkosítja. Ez az a szint, ahol a fájl- és karakterformátumok egységes, alkalmazástól független ábrázolása kialakul.
- *Alkalmazási (application).* Ez a réteg kapcsolódik a felhasználóhoz, illetve a felhasználói alkalmazásokhoz, mint például az elektronikus levelezés, fájlátvitel, webböngésző stb. Alapvető feladata a hálózaton keresztül érkező információ megjelenítése, illetve a felhasználó által bevitt információ továbbítása az alsóbb rétegek felé.

A TCP/IP, az Internet alapprotokoll családja, szintén réteges felépítésű, de csak öt réteget használ. Az OSI-modellben található rétegek közül néhánynak a feladatait másokkal összevonja, illetve bizonyos rétegekkel (mint a fizikai és az adatkapcsolati) nem foglalkozik. A hálózati réteg protokollja az IP *(Internet Protokol)*, míg a szállítási rétegben a TCP és az UDP protokollokat definiálja. Az e fölötti rétegekben a TCP/IP-rendszer az UDP illetve a TCP protokollokra épít. A megjelenítési és viszony szintek szolgáltatásai a TCP/IP alkalmazási rétegben jelennek meg.



4.5. ábra. ábra - A hálózati rétegek és protokollok kapcsolata

4.2.5. Címzés és forgalomirányítás

A forgalomirányítás *(routing)* feladata annak meghatározása, hogy két állomás közötti üzenetek, amennyiben nincs köztük közvetlen kapcsolat, milyen úton, mely közbülső csomópontokon keresztül haladjanak.

A forgalomirányítást a csomópontok olyan táblázatok alapján végzik, amelyek tartalmazzák a lehetséges útvonalakat és esetleg az útvonalakhoz tartozó egyéb információkat, mint például a csatorna sebességét, költségét, aktuális terhelését. A forgalomirányító táblázatokat időnként aktualizálni kell akár kézzel, akár automatikusan.

A gyakrabban használt forgalomirányítási módszerek:

- *Fix útvonal*. Két adott csomópont között az üzenetváltás mindig rögzített – általában a legrövidebb, legkisebb költségű – útvonalakon történik.
- *Virtuális áramkör (virtual circuit).* Egy kapcsolati viszony *(session)* elején kialakult útvonalon halad az összes üzenet, amíg a kapcsolatot valamelyik fél le nem zárja.
- *Dinamikus forgalomirányítás (dynamic routing).* Minden üzenethez külön keresnek útvonalat. A közbülső állomások üzenetenként döntik el, hogy azt melyik útvonalon adják tovább. A döntést befolyásolhatja a csatornák aktuális rendelkezésre állása, illetve terhelése.

Miután a forgalomirányítás talált egy útvonalat a két állomás között, az ott futó folyamatok üzeneteket váltanak. Az állomások kapcsolódásának módjai a következők lehetnek:

- *Áramkörkapcsolás (circuit switching).* A két folyamat kommunikációjához egy fizikai csatorna tartozik, amelyet más folyamatok a kapcsolat időtartama alatt nem használhatnak. A kapcsolat

kialakítása lassú, de ezután az üzenetváltás a csatorna kapacitását teljesen kihasználva folyhat. Ha a folyamatok ritkán küldenek üzenetet, a csatorna kihasználatlan.

- *Üzenetkapcsolás (message switching).* Minden üzenet csomópontok közötti továbbításához ideiglenesen egy fizikai csatornát rendelnek, amelyet az üzenet átvitele után más folyamatok használhatnak. Az átvitel során a közbülső állomásoknak a teljes – gyakorta nagyon nagy – üzenetet venni és a továbbításig tárolnia kell.
- *Csomagkapcsolás (packet switching).* Az üzeneteket a küldő állomás kis, rendszerint azonos hosszúságú csomagokra bontja. A csomagok önállóan haladnak a hálózaton a céljuk felé, csak a fogadónál állítják belőlük össze újra az üzenetet.

A megoldás jól kihasználja a csatornák átviteli kapacitását, hiszen egymás után több üzenethez tartozó csomagot lehet átvinni, nagy méretű közbülső tárakra sincs szükség. Ezzel szemben minden csomag többlet-információt tartalmaz, az átvitel sebessége lassabb.

4.3. Hálózati jellegű szolgáltatások

A hálózati szolgáltatások jellemzője, hogy a felhasználó tisztában van azzal, hogy a rendszerben több csomópont működik. Számára a távoli és a helyi objektumok kezelése nem homogén.

Ebben a fejezetben a két legalapvetőbb hálózati szolgáltatást, a távoli terminált és a hálózati fájlátvitelt foglaljuk össze. A későbbi fejezetekben további szolgáltatásokat, mint például az elosztott fájlrendszer részletesebben is ismertetünk.

4.3.1. Telnet: távoli terminál

A távoli terminál alapvető célja, hogy hálózaton keresztül távoli belépési lehetőséget és virtuális terminált (billentyűzetet és képernyőt) nyújtson a felhasználók számára. Más szavakkal, bárhol is legyen egy *A* gép előtt ülő felhasználó, mindenhonnan oly módon használhassa a *B* gépet, mintha előtte ülne, annak terminálját használná, feltételezve, hogy *A* és *B* gépek között létezik hálózati összeköttetés. Eredetét tekintve a távoli terminál a soros vonali terminálhozzáférés *(dialup terminal connection)* megfelelője a hálózati környezetben.

A távoli belépés egyik megvalósítása a **Telnet** *(teletype network)* **protokoll**, mely a TCP/IP protokollcsalád része, illetve a protokollt használó **Telnet program**. A fejezet további részében e rendszeren keresztül ismertetjük a távoli terminál elérés legfontosabb tulajdonságait. (Napjainkban a Telnet protokoll helyett az SSH protokoll használata javasolt, amely a távoli terminál-elérést titkosított hálózati kommunikációval oldja meg.)

A távoli terminál elérés megvalósítása látszólag egyszerű feladat, a valóságban azonban a terminálok és számítógépek sokfélesége miatt nem is olyan egyszerű. A következőkben összefoglaljuk a leglényegesebb feladatokat és azok megvalósításának alapelemeit.

4.3.1.1. A Telnet kapcsolat

A távoli terminál kapcsolat lényege két hálózati gép közötti karakteralapú kommunikáció megvalósítása. A *Telnet* a kliens és a szerver között egy **hálózati virtuális terminál** *(network virtual terminal)* kapcsolatot hoz létre. A kapcsolat mindkét vége az adott gépen virtuális terminálként működik, logikai billentyűzetet és képernyőt definiálva. A logikai képernyő karakterek megjelenítésére képes, míg a logikai billentyűzet karaktereket generál. A *Telnet* kliens és szerver oldali programjai felelősek a virtuális és valóságos eszközök közötti megfeleltetésért, a karakterek tényleges megjelenítéséért, illetve fogadásáért. A virtuális terminál koncepció segítségével a *Telnet* rendszer minden olyan fizikai eszközzel képes kommunikálni, melynél a virtuális kódok és a valóságos eszköz közötti leképezés létezik.

A kapcsolatot megvalósító rendszernek gondoskodnia kell a valóságos eszköz sajátosságainak figyelembe vételéről, azaz például a vezérlőkarakterek és terminál jellemzők megfelelő átalakításáról a kapcsolatban részt vevő gépek között.

A *Telnet* a feladatot protokoll szinten oldja meg a terminálok jellemzőit leíró protokoll elemek beiktatásával. Amikor két gép *Telnet* protokollal kapcsolatba lép egymással, a kapcsolat kiépítése során a *Telnet* önmaga meghatározza a kapcsolatra érvényes terminál és kommunikációs jellemzőket. A *Telnet* protokoll része az a képesség, hogy csak azokat a kommunikációs módokat támogassa egy kapcsolatban, melyeket mindkét végállomás képes kezelni. Mindezeket maga a protokoll tisztázza a kapcsolat kiépítése során, megkímélve a kommunikáló feleket a további adat- és vezérlőjel átalakításoktól, értelmezésektől.

Ily módon lehetővé válik, hogy a virtuális terminál csak azokat a vezérlő kódokat fogadja, amelyek a hozzá rendelt fizikai eszközökön is értelmezettek.

4.3.1.2. Szerver- és kliensprogramok

A *Telnet* a szervergépen általában egy állandóan futó folyamattal rendelkezik, amelyik a kliens gépektől beérkező kéréseket fogadja, azok kiszolgálását kezdeményezi. UNIX-rendszerekben ez az ún. *telnetd* program, melyet az *inetd* szuperszerver indít el a beérkező kéréseknek megfelelően. Windows NT- és más PC-alapú rendszerek esetén egy önálló *Telnet* szerveralkalmazás oldja meg a feladatot.

A szerverfolyamat feladata a kommunikációs paraméterek tisztázása (a virtuális terminál meghatározása) után a terminált kiszolgáló folyamat elindítása és hozzárendelése a virtuális terminálhoz. UNIX-rendszerekben ez általában a felhasználó *shell* programja, Windows- és más PC-rendszerek esetében az alap parancsértelmező (pl. DOS prompt).

A kliens oldal általában egy *Telnet* nevű programot futtat, és ezzel kezdeményezi a kapcsolat kiépítését a szerverhez. A *Telnet* kliens elindításához szükség van a szerver IP-címére vagy nevére (ez utóbbi csak akkor használható, ha a rendszer képes a névhez tartozó IP-cím megtalálására). A TCP/IP protokoll családban az alkalmazások gépen belüli címzésére szolgáló portok közül a *Telnet* saját, dedikált portszámmal (23)

rendelkezik. A kliens indításakor ez az alapértelmezés szerinti port, amely újabb paraméter megadásával módosítható.

A *Telnet* kliens program hívása:

telnet [<gépnév vagy IP cím>] [<portszám>]

Az alábbi példák egyenértékűek, feltételezve, hogy a használt gépnév megfelel az IP-címnek.

telnet 152.66.76.1

telnet server.mit.bme.hu

telnet server.mit.bme.hu 23

A 23-as *Telnet* porttól eltérő szám használata esetén természetesen szerveroldalon is az eltérő számú porton kell a *telnetd* programot elindítani.

$ telnet server.mit.bme.hu

Trying 152.66.76.1...

Connected to server.mit.bme.hu.

Escape character is '^]'.

SunOS 5.7

login: labor

Password: ********

Last login: Thu Sep 2 12:41:28 from labpc1.mit.bme.hu

$

Ahogy az látható az előző példa kódjában, a *telnet* parancs kiadása után a *Telnet* program elvégzi a név–IP-cím átalakítást, majd hozzákapcsolódik a szerverhez. Ezek után beállítja a kommunikációs paramétereket a szerver és a kliens között, kiírja a speciális *Telnet* kontrollkaraktert (CTRL+]), végül összekapcsolja a klienst a szerverfolyamat által elindított *login* programmal. A *login* program a felhasználói név és jelszó ellenőrzése után a felhasználó saját *shell* programját indítja el (a *$* jel jelzi a várakozó *shell*t).

A *Telnet* kliensprogramot paraméterek nélkül elindítva az ún. parancsértelmező módot kapjuk, amikor a *Telnet* rendszer beépített parancsait érhetjük el. Ez a parancsértelmező mód a létrejött kapcsolat során (a virtuális terminál használata közben) is elérhető a speciális kontroll karakter leütésével (általában a CTRL+]).

4.3.1.3. A Telnet parancsértelmező

A *Telnet* parancsértelmező része kliens oldali parancsok kiadására, beállítások megváltoztatására szolgál. Ha a kapcsolat már kiépült a szerverig, akkor minden parancs kiadása után a *Telnet* visszatér a virtuális terminálhoz.

Ha a kapcsolat még nem épült ki, akkor folyamatosan parancs módban marad mindaddig, míg kapcsolat kiépítését nem kezdeményezzük. Parancs módban elindíthatunk kliens oldali alkalmazásokat, megváltoztathatjuk az aktuális kliens könyvtárat, speciális vezérlő karaktereket küldhetünk a szervernek, alapállapotba állíthatjuk a kliens–szerver kapcsolatot, illetve megváltoztathatunk egyes kommunikációs paramétereket.

Az alábbi példa szemlélteti a *Telnet* parancsokat:

telnet> <CTRL+]>

telnet> help

Commands may be abbreviated. Commands are:

close close current connection

logout forcibly logout remote user and close the connection

display display operating parameters

mode try to enter line or character mode (‘mode ?’ for more)

open connect to a site

quit exit telnet

send transmit special characters (‘send ?’ for more)

set set operating parameters (‘set ?’ for more)

unset unset operating parameters (‘unset ?’ for more)

status print status information

toggle toggle operating parameters (‘toggle ?’ for more)

slc change state of special charaters (‘slc ?’ for more)

z suspend telnet

! invoke a subshell

environ change environment variables (‘environ ?’ for more)

? print help information

<return> leave command mode

telnet> display

will flush output when sending interrupt characters.

won’t send interrupt characters in urgent mode.

won't skip reading of ~/.telnetrc file.

won't map carriage return on output.

will recognize certain control characters.

won't turn on socket level debugging.

won't print hexadecimal representation of network traffic.

won't print user readable output for „netdata".

won't show option processing.

won't print hexadecimal representation of terminal traffic.

echo [^E]

escape [^]]

rlogin [off]

tracefile „(standard output)"

flushoutput [^O]

interrupt [^C]

quit [^\]

eof [^D]

erase [^H]

kill [^U]

lnext [^V]

susp [^Z]

reprint [^R]

worderase [^W]

start [^Q]

stop [^S]

ayt [^T]

DO ECHO

DO SUPPRESS GO AHEAD

WILL TERMINAL TYPE

WILL NAWS

WILL NEW-ENVIRON

A kapcsolat módosítását, inicializálását, illetve egyéb, a *Telnet* szervert is érintő parancsokat a kliens egy speciális *Telnet COMMAND* csomagban küldi el a szervernek. Ezen csomagok tipikusan pár bájtot tartalmaznak, első a parancs kódja, majd opcionálisan azt követik a paraméterek.

4.3.2. FTP: fájlátvitel

A **fájlátviteli protokoll** *(FTP: file transfer protocol)* a gépek közötti hálózati fájlátvitel segédeszköze. Lehetővé teszi a fájlok mozgatását a kliens és szerver között mindkét irányban, könyvtárak létrehozását, átnevezését, illetve törlését. Ellentétben a *Telnet* programmal, az FTP nem képes távoli programok futtatására, de fájlátvitelre sokkal egyszerűbben használható program.

4.3.2.1. FTP-kapcsolat

A *Telnet* programmal ellentétben az FTP két párhuzamos kapcsolatot épít ki a kliens és a szerver között: egyet az FTP-parancsok és egy másikat az adatok átvitelére. Ennek megfelelően a szerver oldalon a kiszolgáló folyamat két részre bontható, egy protokoll értelmezőre és egy adatátvitel végrehajtóra.

Az FTP esetében a gépek közötti kommunikáció kiépítése lényegesen egyszerűbb: nincs szükség a virtuális terminálok beállítására. Ugyanakkor itt is gondoskodni kell az eltérő reprezentációk megfeleltetéséről.

Az FTP a fájlátvitel többféle formáját is megengedi. A legelterjedtebb a következő két forma: *bináris* és *szöveges*. Bináris FTP-kapcsolat esetén az átküldött fájlok bájthelyesen, változtatás nélkül érkeznek meg. Szöveges kapcsolat esetén az FTP gondoskodik az eltérő szöveges fájlformátumok átalakításáról (például a DOS–UNIX szövegfájlok konverziójáról). A helyes átviteli mód kiválasztása az átvitel előtt a felhasználó feladata. Alapértelmezésben az FTP a szöveges módot használja, ami sok hibára ad lehetőséget.

Az FTP további lényeges jellemzője, hogy az adatátvitel (a rendszer lényegi szolgáltatása) a felhasználó számára is jól követhető, ellenőrizhető. A felhasználó képes az adatkapcsolat beállítására. Erre a *Telnet*hez hasonlóan az FTP-parancs üzemmód alkalmas.

4.3.2.2. FTP kliens- és szerverprogramok

Szerver oldalon az FTP-folyamat szolgálja ki a klienseket. UNIX-rendszerekben ez tipikusan az **ftdp** démon folyamat, míg Windows NT- és más PC-rendszerek alatt egy önálló FTP szerveralkalmazás. A szerverfolyamat a klienskérések fogadása után ellenőrzi a felhasználói nevet és jelszót, majd az ellenőrzött jogosultságok alapján hozzáférést teremt a szervergép fájlrendszeréhez.

A UNIX-rendszerek többségében az FTP-szerviz megtalálható, de alapértelmezésben nem engedélyezett, az *inetd* szuperszerver kezeli. A Windows-alapú rendszerekből hiányzik ez a szolgáltatás, külön szoftver

telepítésével és futtatásával hozható létre. A Windows-alapú TCP/IP programcsomagok többsége tartalmaz ilyen szervert, illetve önálló programként is beszerezhető.

Az FTP-klienst a felhasználó a *Telnet* klienshez hasonlóan, a szerver címének megadásával indíthatja el. Itt is létezik a *Telnet* esetében látott parancs üzemmód, mely azonban itt alapértelmezésnek számít, azaz a kapcsolat kiépítése során is ez az interfész látszik a felhasználó felé. A parancsértelmezőnek adott utasításokkal vezérelhetjük az FTP-kapcsolatot, a fájlátvitelt, és ezek segítségével utasíthatjuk az FTP-szerver folyamatot a szervergép fájlrendszerének módosítására.

A Telnet programhoz hasonlóan az FTP is dedikált portszámot használ, a 21-est. Itt is megadható más portszám a kliensprogram paraméterei között.

Az FTP kliensprogram használata:

ftp [<szerver cím vagy név>] [<portszám>]

Alkalmazási példa:

$ ftp ftp.mit.bme.hu

Connected to ftp.mit.bme.hu.

Name (ftp.mit.bme.hu:labor): labor

331 Password required for labor.

Password: ********

230 User labor logged in.

ftp>

$

A kapcsolat kiépítése után itt is a felhasználói név és jelszó megadása és ellenőrzése következik, ami után a parancsértelmező promptot kapjuk. Általában a helyi és távoli könyvtár, valamint a fájlátviteli mód meghatározása után kezdődik el a tényleges fájlátvitel.

A következő rövid példa egy szöveges fájl átvitelét mutatja a kliensgépről a szervergépre, majd egy bináris fájl átvitelét a szerverről a kliensre.

ftp> ascii

200 Type set to A.

ftp> put leiras.txt

200 PORT command successful.

150 Opening ASCII mode data connection for leiras.txt (4896 bytes).

226 Transfer complete.

local: leiras.txt remote: leiras.txt

4896 bytes received in 1.8 seconds

ftp> bin

200 Type set to I.

ftp> get backup.zip

200 PORT command successful.

150 Opening BINARY mode data connection for backup.zip (4944123 bytes).

226 Transfer complete.

local: backup.zip remote: backup.zip

4944123 bytes received in 9.8 seconds

ftp>

Az FTP általában egy speciális felhasználó azonosítási módot is alkalmaz, mellyel lehetővé válik a valós név és jelszó nélküli, ún. anonim FTP-belépés. Ekkor névként az „anonymous” (vagy esetenként az „ftp”) nevet kell megadni, jelszóként pedig az FTP-kapcsolatot kezdeményező személy e-mail címét. Ilyenkor az FTP-szerver egy speciális könyvtárterülethez engedélyez korlátozott hozzáférést, általában a fájlrendszer egyéb részeinek teljes kizárásával. UNIX operációs rendszer alatt ezt a *chroot()* rendszerhívás segítségével oldja meg a kapcsolathoz rendelt szerver folyamat. A *chroot()* rendszerhívás a fájlrendszert egy adott pontjától kezdve „levágja”, azaz a fa struktúra pont feletti részeinek elérését teljesen megszünteti.

4.3.2.3. Az FTP-parancsértelmező

Az FTP-rendszerben a felhasználó alapértelmezésben a parancsértelmezőt használja helyi és távoli feladatok megoldására, illetve a fájlátvitelre. Az alábbi példa összefoglalja egy tipikus FTP-rendszer parancsszavait.

ftp> help

Commands may be abbreviated. Commands are:

! cr macdef proxy send

$ delete mdelete sendport status

account debug mdir put struct

append dir mget pwd sunique

ascii disconnect mkdir quit tenex

bell form mls quote trace

binary get mode recv type

bye glob mput remotehelp user

case hash nmap rename verbose

cd help ntrans reset ?

cdup lcd open rmdir

close ls prompt runique

ftp>

A parancsok alapvetően három nagy kategóriába sorolhatóak: helyi (kliens) parancsok, távoli (szerver) parancsok, és adatátviteli parancsok. Az átviteli mód meghatározására szolgáló *ascii* és *binary* parancsok kiadása után a *cd* paranccsal beállíthatjuk a távoli, illetve az *lcd* paranccsal a helyi munkakönyvtárat. Ezek után a *put* vagy *get* parancsok segítségével bonyolítható le a kívánt fájlátvitel, majd a *quit* paranccsal szakíthatjuk meg a kapcsolatot és léphetünk ki az FTP kliensprogram parancsértelmezőjéből.

4.4. Elosztott szolgáltatások

4.4.1. Jellemzők

Mint azt korábban láttuk, az **elosztott rendszer** hálózattal összekötött, **elosztott szoftverrel** felruházott autonóm számítógépek összessége. Ebben az architektúrában az elosztott szoftver két fő feladatot lát el: az aktivitások koordinálását és az erőforrások megosztását. Egy elosztott rendszer akkor teljesíti a vele szemben támasztott elvárásokat, ha a felhasználója egyetlen, integrált számítási eszközt lát, az elosztottság számára rejtve marad. Az alábbiakban áttekintjük azokat a jellemzőket, amik alapján egy elosztott rendszert minősíteni lehet.

4.4.1.1. Az elosztott rendszerek legfontosabb jellemzői

Az elosztott rendszerekkel szemben számos követelmény merül fel. Ezek közül a legfontosabbak:

- erőforrás-megosztás,
- nyitottság,
- konkurencia,
- skálázhatóság,
- hibatűrés,
- átlátszóság,
- biztonság.

A továbbiakban részletesen elemezzük ezeket a követelményeket.

Erőforrás-megosztás

Az erőforrások osztott használata mindig jelentős szerepet töltött be a számítógépes rendszerek történetében. Ezt elsősorban gazdasági és rendszerszervezési okokra lehet visszavezetni. A hardver- és szoftverkomponensek fejlődése egyre finomabb megközelítést tett lehetővé az osztott erőforrás használathoz. Míg a ’60-as évekre elsősorban az **időosztásos rendszerek** voltak jellemzőek, ahol a hangsúly elsősorban a **CPU osztott használatára** helyeződött, addig a ’70-es évek elején megjelentek a **többfelhasználós rendszerek**, felvetve egyéb hardverberendezések, mint például a **nyomtatók, lemezegységek** és **perifériák osztott használatának** az igényét, ahol kényelmi és költségkímélő szempontok domináltak. A szoftvereszközök rohamos fejlődése szinte elengedhetetlenné tette a koncepcionális erőforrások, az **adatok osztott használatát**. Manapság komplex szoftverrendszer fejlesztése elképzelhetetlen osztott adat és program használat nélkül. A fejlesztőcsapat együtt használhatja a fejlesztőeszközöket, közvetlen betekintést nyernek egymás munkájába, sőt az osztott használat a bonyolult rendszerek karbantartását is megkönnyíti. A kereskedelmi alkalmazások egyre elterjedtebben alkalmazzák azt a megközelítést, hogy a felhasználók egyetlen aktív adatbázis osztottan használt adatobjektumaihoz férnek hozzá, ahelyett, hogy mindenhol egy-egy saját másolattal dolgoznának, így a kényelmes használat mellett lényegesen egyszerűbb a konzisztencia karbantartás is. A hálózati és elosztott alkalmazások egyre szélesedő területe, amikor a számítógépekkel együttműködő felhasználói csoportok hatékony munkavégzését támogatják. Ez elvezetett egy újfajta munkamódszer kialakulásához, az ún. **számítógéppel támogatott együttműködő munkavégzéshez** (Computer Supported Cooperative Working – CSCW, amit **groupware** néven is ismernek), ahol egy adott nagy komplexitású feladat elvégzésén egyszerre párhuzamosan többen is dolgoznak, és a feladat megoldásához szükséges összes szoftverkomponenst (programokat, modelleket, adatelemeket, adatbázisokat) közösen, osztottan használják.

Míg a többfelhasználós rendszerekben az erőforrások osztott használata nyilvánvaló (egy rendszer, a kernel felügyeli az erőforrásokat), addig elosztott rendszerekben újfajta architektúrára, támogatásra van szükség. Ez az új elem az **erőforrás-menedzser**, aminek az a feladata, hogy biztosítsa egy adott erőforrás optimális és igazságos használatát. Az erőforrás felhasználói az erőforrás-menedzserrel kommunikálnak, tőle kérnek hozzáférést az erőforráshoz. Az utóbbi időben két elterjedt modellt alkalmaznak: a **kliens–szerver-modellt** és az **objektummodellt**.

A kliens–szerver-modellben a szerver egy adott erőforrás menedzsere, a kliensek kéréseken keresztül próbálják az erőforrást használni. Ennél a modellnél nagyon fontos, hogy a kliens–szerver kapcsolat mindig egy adott feladatra vonatkozik, egy másik feladatkörben a szerver is lehet kliens (egy másik szerver számára). Az általános modellben egy tetszőleges számítógép futtathatja a klienst és a szervert (kliens/szerver lehet azonos gépen is). Nagyon fontos, hogy egy elosztott rendszerben egy adott típusú szolgáltatást egyszerre több, egymással ekvivalens szerver is nyújthat, így meg kell különböztetni a szolgáltatást magától a szolgáltatást konkrétan nyújtó szervertől!

Egy rendszerben azonban nem lehet minden erőforrást ezzel a modellel kezelni, bizonyos erőforrásoknak lokálisnak kell maradni. Ezek közül a legfontosabbak a CPU, a memória és a lokális hálózat interfész. Ezeket tipikusan az operációs rendszer, a kernel kezeli.

Az objektum alapú modell hasonló az objektumorientált programozás modelljéhez. Itt minden osztott erőforrást egy objektumként modellezünk, az erőforrást bezárjuk az objektumba. Az objektumok egyedi azonosítókkal rendelkeznek, így mozgathatók. A szolgáltatáskérést egyszerűen az objektumnak küldött üzeneteken keresztül lehet megvalósítani. A modell előnye, hogy egyszerű, rugalmas keretet biztosít az erőforrások kezeléséhez, másrészt az osztott erőforrásokat azonos módon lehet kezelni.

Nyitottság

A nyitottság kérdése a rendszer **bővíthetőségével** foglalkozik. Alapvetően az elosztott modellben azzal a feltételezéssel élünk, hogy **minden erőforrásból korlátlan mennyiség** áll rendelkezésünkre (ha mégsem, akkor a rendszert zökkenőmentesen bővíthetjük). Ebből adódóan egy elosztott rendszerrel szemben természetes elvárás, hogy ha a rendszert valamilyen szempontból bővítjük (például újabb gépeket adunk hozzá, újabb szolgáltatásokat integrálunk, újabb szerverekkel bővítjük), akkor ne kelljen architekturális változtatásokat végrehajtani, a felhasználók elől a bővítés rejtve maradjon. A bővíthetőség egyaránt vonatkozik hardverelemekre (további perifériák, memória, kommunikációs interfész), illetve szoftverelemekre is (az operációs rendszer szolgáltatásai, kommunikációs protokoll, erőforrás-megosztó szolgáltatások). Így a nyitottság mértékét az határozza meg, hogy az új osztott erőforrásokat kezelő szolgáltatásokat mennyire lehet „folytonosan" hozzáadni a rendszerhez (működés megszakítása, illetve komponensek duplikálása nélkül).

Felmerül a kérdés, hogy hogyan lehet elérni a **nyitottságot**? Erre elég nehéz kimerítő választ adni, azonban egy elengedhetetlen feltételnek mindenféleképpen meg kell felelni: a főbb szoftver interfészeket nyilvánosságra kell hozni (publikálni kell). A korai rendszerek zártak voltak, mivel nem feleltek meg ennek az igen fontos feltételnek. Összefoglalva, a nyitott elosztott rendszerek főbb ismérvei:

- a főbb (operációs rendszer) interfészek publikáltak,
- egységes folyamatok közötti kommunikáció, publikált interfészek az osztott erőforrások eléréséhez,
- eltérő HW/SW, de a publikált szabványokhoz igazodni kell.

Nyitottság szempontjából a később ismertetett Unix az azt megelőző rendszerekhez képest „nyitottabbnak" számít:

- Az *alkalmazásfejlesztők* számára nyitottabb, mert hozzáférnek a rendszer által nyújtott összes szolgáltatáshoz.
- A *hardverforgalmazók és a rendszeradminisztrátorok* számára is nyitottabb, mert az operációs rendszert viszonylag egyszerűen lehet bővíteni, könnyen hozzá lehet adni új perifériákat.

- A *szoftverforgalmazók és a felhasználók* számára is nyitottabb, mert hardver-független. A szoftverfejlesztők olyan programokat írhatnak, amelyek módosítás nélkül futnak több hardverplatformon is. (Persze, mint azt később látni fogjuk, ez az állítás csak akkor igaz, ha adott UNIX változatra készülnek a programok, vagy azzal a feltételezéssel élnek, hogy az adott UNIX változat megfelel bizonyos később ismertetett szabványoknak.)

Konkurencia

A konkurencia és a párhuzamos végrehajtás természetesen vetődik fel az elosztott rendszerekben, mivel a felhasználók egymástól elkülönülő tevékenységeket végeznek, független erőforrások (is) megtalálhatók a rendszerben és a szerverfolyamatok külön számítógépeken is futnak. Ezen tevékenységek szétválasztása lehetővé teszi, hogy a feldolgozás párhuzamosan folyjon a különböző számítógépeken. Azonban nagy hangsúlyt kell fektetni az osztottan használt erőforrások **hozzáférés szabályozására** és **frissítésére**, a **szinkronizációs sémákat** gondosan meg kell tervezni, valamint biztosítani kell, hogy a konkurens végrehajtás biztosította előnyök nem tékozlódnak el egy rossz szinkronizációs séma miatt.

Ha a számítógép csak egyetlen processzort tartalmaz, és egyszerre több folyamatot szeretnénk futtatni, akkor a folyamatok végrehajtását össze kell fésülni, megfelelő támogatást biztosítva a folyamatok közötti váltáshoz, az adatszerkezetek mentéséhez és visszaállításához. Ebben az esetben a folyamatok koncepcionálisan úgy érzékelik, hogy párhuzamosan futnak. Ezt **virtuális párhuzamosságnak** nevezik. Ha egy gépben *N* processzor található, akkor *N* folyamat futhat valóban fizikailag is párhuzamosan. Ekkor, ha a folyamatok közel függetlenek, a gyorsulás az adminisztratív tevékenység idejét leszámítva majdnem *N*-szeres lehet. Azonban az esetek nagy részében együttműködő folyamatokkal van dolgunk, amelyek egymás eredményeit felhasználják, kommunikálnak egymással, így az *N*-szeres sebességnövekedés közel sem érhető el. Az elosztott rendszereknél sok számítógép kapcsolódik össze egy hálózaton keresztül. Amennyiben a folyamatok az egyes gépekre szétoszthatók, akkor hasonló a helyzet a többprocesszoros rendszerekéhez: a feldolgozás itt is jelentősen felgyorsítható. Ezen struktúrát tipikusan a gyakran használt szolgáltatások használják ki intenzíven. Például az állományszerverek nagy terhelésnek vannak kitéve egy elosztott rendszerben, ezért szokás több állományszervert is működtetni, méghozzá különböző gépeken. Így a rendszer szempontjából az állományszerver nyújtotta szolgáltatás a kliensei számára lényegesen felgyorsulhat. (Meg kell jegyezni, hogy a nagyteljesítményű szerverek ráadásul tipikusan többprocesszoros architektúrákon futnak, tovább javítva a szolgáltatás teljesítményét.)

Skálázhatóság

Az elosztott rendszerek eltérő méretűek lehetnek, kezdve a legkisebb, két gépből álló rendszertől a lokális hálózatra épített akár több száz gépet is magába foglaló rendszerig. A **skálázhatóság** fogalma azt jelenti, hogy amikor a rendszer méretét növeljük, akkor ne kelljen a rendszer architektúráját vagy a szoftver alkalmazásokat megváltoztatni. Egy rendszer tervezésénél figyelembe kell venni annak várható életciklusát, az esetleges

felmerülő bővítési igényeket, és úgy kell kialakítani, hogy ezek az ésszerű igények radikális változtatás nélkül megvalósíthatóak legyenek.

Az elmúlt években tanúi lehettünk egy skálázhatósági problémának: pár évvel ezelőtt a budapesti telefonszámok még hatjegyűek voltak. A telefonok, illetve faxberendezések rohamos terjedésével azonban a hat számjegy kevésnek bizonyult, így át kellett térni a hétjegyű telefonszámokra, ami miatt a régi telefonszámok megváltoztak, vagyis a régi számok tulajdonosai a rendszer bővítése során radikális változtatást érzékeltek. A tervezésnél persze nem szabad átesni a ló túlsó oldalára, nem szabad irreális követelményekre tervezni, a praktikusságot, kényelmes használatot is szem előtt kell tartani. Valószínűleg egyetlen olvasónk sem repesne a boldogságtól, ha például 20 számjegyes telefonszámokat kellene használnia, holott az a közeljövőben biztosan nem fog skálázhatósági problémákhoz vezetni.

A skálázható rendszerekben azt az egyszerű tervezési filozófiát alkalmazzák, hogy semmilyen erőforrás nem korlátozott, ha a rendszer bővülésével valamilyen erőforrásnak mégis szűkében lennénk, újabbat adunk a rendszerhez. Például, ha az elosztott állományszolgáltatás lelassul, mert a szerver nem képes már tolerálható időn belül kiszolgálni a megsűrűsödött kéréseket, újabb állományszervert állíthatunk szolgálatba.

Hibatűrés

A számítógépes rendszerek bonyolultságukból adódóan néha meghibásodnak. Mivel a hardver- és szoftverhibák miatt hibás eredmények születhetnek, illetve a programok a feladataik elvégzése előtt is befejezhetik futásukat, így feltétlenül foglalkozni kell a **hibatűrés** megvalósításával.

A hibatűrő viselkedés megvalósításához mind **hardverredundanciát (hardware redundancy)** (redundáns komponensek alkalmazása), mind pedig **szoftverfelépülést (software recovery)** (olyan szoftverek alkalmazása, amik hibákból felépülnek, folytatják a helyes működést) alkalmaznak.

A hardver redundanciát gyakran **meleg tartalék** formájában valósítják meg, vagyis egy adott kritikus funkciót megvalósító berendezés mellett párhuzamosan működtetnek egy másik, azonos, vagy azonos funkciót nyújtó berendezést. Ennek a megoldásnak hátránya, hogy nagyon költséges, azonban számos kritikus alkalmazásnál a biztonsági szempontok dominálnak és megkövetelik a meleg tartalékot. Nagyon sok alkalmazásnál azonban ennek költségei megengedhetetlenek, más megoldásokat kell alkalmazni.

Az elosztott rendszerekben a redundancia finomabb szinten is tervezhető, például kritikus szerverek **replikálhatók**. Másrészt az elosztott rendszerekben a redundáns hardvert ki lehet használni nemkritikus tevékenységek végrehajtására, amíg nincs rá szükség.

A szoftverfelépülés megvalósításához, mint azt már korábban láttuk, olyan szoftver komponenseket kell tervezni, amelyek segítségével az állandó adatok meghibásodás észlelésekor visszaállíthatók, a számítások **visszagörgethetők** (recovery, roll-back).

Az elosztott rendszerek egy másik előnyös tulajdonsága, hogy hardver- hibák jelenlétében is nagyfokú **rendelkezésre állást** biztosítanak. Egy többfelhasználós rendszerben egyetlen hiba hatására majdnem minden esetben a rendszer elérhetetlenné válik az összes felhasználója számára. Ezzel szemben, amikor egy elosztott rendszerben meghibásodik valamelyik komponens, a hiba csak a meghibásodott komponensen folyó munkát érinti.

Átlátszóság (fizikai széttagoltság elrejtése)

Az **átlátszóság (transparency)** nem más, mint egy elosztott rendszerben a komponensek elosztott természetének elrejtése a felhasználó és az alkalmazásfejlesztő elől, hogy azok a rendszert egy egységes egészként, nem pedig független komponensek laza összességeként lássák. Ennek eléréséhez hálózati és kommunikációs, explicit menedzsment és integrációs technikákra van szükség. Az alábbiakban röviden vázoljuk a nyolcféle átlátszóság definícióját.

- A *hozzáférés átlátszóság (access transparency)* lehetővé teszi a helyi és a távoli erőforrások azonos műveleteket használó, azonos módon történő kezelését.
- A *hely átlátszóság (location transparency)* lehetővé teszi információs objektumok elérését azok helyének ismerete nélkül. A hozzáférés átlátszóságot és a hely átlátszóságot együttesen szokás *hálózati átlátszóságnak (network transparency)* nevezni.
- A *konkurencia átlátszóság (concurrency transparency)* lehetővé teszi folyamatok konkurens együttműködését osztott információs objektumok használatán keresztül, anélkül, hogy azok zavarnák egymást.
- A *másolat átlátszóság (replication transparency)* lehetővé teszi, hogy az információs objektumokból (a nagyobb megbízhatóság, jobb teljesítmény érdekében) több másolat is létezzen a rendszerben anélkül, hogy a felhasználó és a programok tudomást szereznének a másolatokról.
- A *hiba átlátszóság (failure transparency)* lehetővé teszi a meghibásodások elrejtését, lehetővé téve a felhasználók és az alkalmazói programok számára, hogy a feladataikat hardver- vagy szoftverhibák jelenlétében is elvégezzék.
- A *vándorlási átlátszóság (migration transparency)* lehetővé teszi az információs objektumok szabad mozgatását a rendszerben anélkül, hogy befolyásolnák a felhasználó, illetve az alkalmazói programok működését.
- A *teljesítmény átlátszóság (performance transparency)* lehetővé teszi a rendszer átkonfigurálását, hogy a rendszer teljesítménye terhelésváltozáskor javítható legyen.
- A *skálázási átlátszóság (scaling transparency)* lehetővé teszi a rendszer bővíthetőségét, a rendszer struktúrájának, illetve az alkalmazások algoritmusainak megváltozása nélkül.

A fenti átlátszóságok közül a két legfontosabb a hozzáférés és a hely átlátszóság, ezek megléte vagy hiánya van a legnagyobb hatással az elosztott erőforrások használhatóságára.

Az alábbiakban egy példát mutatunk a hálózati átlátszóság hiányára. A Unix jól ismert *rlogin* parancsával egy felhasználható beléphet egy megnevezett gépre. Itt máris sérül a hely átlátszóság, mert a gépet mindenképp meg kell nevezni. Továbbá az *rlogin* használatakor követett eljárás eltér a lokális gépre történő bejelentkezéstől, mivel a lokális gépre történő bejelentkezésnél a login programot a rendszer automatikusan futtatja, nem kell azt explicite meghívni. Ebből adódóan a hozzáférés átlátszóság sem teljesül. Egy átlátszóságot megvalósító rendszerben elképzelhető, hogy a felhasználó nem egy adott gépre, hanem egy *domainbe* lép be. Ekkor a felhasználó jogosultságot kapna az adott domain összes szolgáltatásának a használatához.

Ezzel szemben az elektronikus levelezés a hálózati átlátszóság tulajdonságát mutatja. Egy elektronikus levelezési cím, mint például *roman@mit. bme.hu* egy felhasználói és egy domain névből áll. A domaineket szervezeti struktúrák alapján definiálják és osztják ki. A felhasználókhoz egy adott domainen belül rendelnek levelezési nevet. Ha egy ilyen felhasználónak küldünk levelet, ahhoz nem kell ismernünk a felhasználó fizikai vagy hálózati helyét, illetve magának a levélküldésnek a folyamata is független a címzett helyétől. Ebből adódóan az Interneten történő elektronikus levelezés rendelkezik a hálózati átlátszóság tulajdonsággal.

4.4.1.2. Elosztott rendszerek tervezési szempontjai

Az előző alfejezetben áttekintettük az elosztott rendszerek hasznosságát leíró főbb jellemzőket. Ebben az alfejezetben az elosztott rendszerek tervezésénél figyelembe veendő legfontosabb tervezési szempontokat vizsgáljuk. Bár egy elosztott rendszer vagy alkalmazás tervezésénél számos olyan szempontot is figyelembe kell venni, ami nem kapcsolódik a rendszer elosztottságához, mint például szoftver mérnöki technikák, ember–gép-kapcsolat és algoritmus tervezés, az alábbiakban csak azokkal a tervezési szempontokkal foglalkozunk, amelyek kifejezetten a rendszer elosztott természetéből fakadnak. A továbbiakban az alábbi témaköröket tárgyaljuk:

- megnevezés,
- kommunikáció,
- programstruktúra,
- terheléskiosztás,
- konzisztencia karbantartás.

-

Megnevezés

Ha egy folyamat olyan erőforráshoz akar hozzáférni, amit nem a folyamat maga felügyel, akkor meg kell neveznie az erőforrást. A továbbiakban **név** alatt egy olyan megnevezést értünk, amit az **emberi felhasználók** könnyen értelmezni tudnak, mindamellett programok is használhatnak, míg **azonosító** alatt csak **programok**

által értelmezhető megnevezést értünk. Ebben az esetben tipikusan valamilyen kompakt ábrázolásmódot, bitmintát alkalmaznak.

Azt a folyamatot, amely során egy nevet leképeznek egy olyan alakra, ami lehetővé teszi az erőforrásokon való műveletvégzést, **névfeloldásnak (name resolution)** nevezzük. Elosztott rendszerekben a névfeloldás talán az egyik legfontosabb szolgáltatás, melynek során általában egy **kommunikációs azonosítót** állítanak elő. Például az Internetes kommunikációban a kommunikációs azonosító két részből áll: egy **host azonosítóból** és egy **port számból**.

Mivel a névfeloldás erősen befolyásolja az elosztott rendszer hatékonyságát, használhatóságát, így megválasztásánál nagyon körültekintően kell eljárni. A választásnál az alábbi ortogonális tervezési szempontokat kell figyelembe venni: egyrészt választhatunk véges vagy potenciálisan végtelen névteret, másrészt kialakíthatunk strukturált vagy lapos (egyszintű) névteret.

A nevek feloldása mindig valamilyen környezetben történik és a név feloldásához mindig meg kell adni azt a környezetet is, amiben a feloldást el kell végezni. Például az állományrendszer esetében minden egyes könyvtár egy környezetet jelent.

A nevek és azonosítók mindig a rendeltetésüknek leginkább megfelelő alakot öltik. Bizonyos neveket az emberek számára olvasható alakúra terveznek, hogy az emberi felhasználó könnyen eligazodjon közöttük. Az állománynevek, mint például ~roman/dokumentumok/konyv és a magasszintű hálózati nevek, mint például az Internetes domain nevek – például mit.bme.hu – ebbe a kategóriába esnek. Más neveket ezzel szemben úgy alakítanak ki, hogy azok tömör ábrázolásmódot biztosítsanak, helytakarékosak legyenek, esetleg az azonosított erőforrás elhelyezkedéséről áruljanak el valamit. (Meg kell jegyezni, hogy az utóbbi esetben sérül a hely átlátszóság.)

Nagyon elterjedten alkalmaznak **hierarchikus névteret**. Az egyik legjelentősebb előnye, hogy a név minden egyes része eltérő környezetben kerül feloldásra, így ugyanaz a név komponens többször is felhasználható. Ebből adódóan egy hierarchikus névtér **potenciálisan végtelen névteret** jelöl ki.

A megnevezési sémák kialakításánál figyelembe lehet venni **védelmi szempontokat** is. Ki lehet alakítani olyan névteret, amely már a név megválasztásával véd a jogosulatlan használat ellen. Ennek leggyakoribb módja, hogy olyan azonosítót alkalmaznak, aminek az előállítása számításigényes olyan folyamatok számára, amelyek azt nem birtokolják. A számításigény annyira megnövelhető, hogy ezáltal **praktikusan lehetetlenné** válik jogosulatlan nevek megszerzése azok véletlen generálásával.

Kommunikáció

Egy elosztott rendszer komponensei mind logikailag, mind pedig fizikailag széttagoltak, együttműködésükhöz **kommunikációra** van szükség. Egy folyamatpár közötti kommunikáció a küldő és a fogadó folyamatok olyan műveleteit foglalja magába, aminek eredményeképp adatátvitel jön létre a küldő folyamat környezetéből a fogadó folyamat környezetébe és **szinkronizáció** valósul meg a két folyamat között. A kommunikációt két

alapvető programozástechnikai primitív, a *send* és a *receive* (küldés és fogadás) valósítja meg. A kommunikáció ebben a sémában **üzenetküldésen** alapszik. A kommunikációs mechanizmus, mint azt korábban láttuk, lehet **szinkron** vagy **blokkolódó**, vagyis a küldő folyamat bevárja az üzenet vételét, vagy **aszinkron** vagy **nem blokkolódó**, vagyis a küldő folyamat továbbhalad anélkül, hogy megvárná az üzenet fogadását.

A továbbiakban megvizsgálunk három kommunikációs sémát: a **kliens–szerver** sémát, a **csoportos multicastot** és a **függvényszállítást**.

Kliens–szerver séma

A kliens–szerver kommunikációs modellt szervizek nyújtására dolgozták ki. A kommunikáció három lépésből tevődik össze:

- A kliens elküldi a kérését a szervernek.
- A szerver elvégzi a kliens számára a kért műveletet (szolgáltatást).
- A szerver elküldi a választ a kliensnek.

A kliens–szerver kommunikációs sémát meg lehet valósítani a *send* és *receive* primitívekkel, azonban azt nagyon gyakran nyelvi szintre emelve **távoli eljáráshívásokkal** valósítják meg. Ennek részleteit a későbbiekben ismertetjük.

A kommunikáció során leggyakrabban az ún. **Request-Reply protokollt** használják, erre optimalizálják a kommunikációt. (A kérés tartalmaz egy kommunikációs azonosítót, ahová a választ a szervernek vissza kell küldeni.)

Mint azt korábban láttuk, az elosztott rendszereknél a rendszer nyíltsága nagyon fontos követelmény. A szerverek és az általuk nyújtott szolgáltatások könnyű bővíthetősége és integrálhatósága érdekében a szerverekhez *dinamikusan kell az azonosítókat* hozzárendelni. Ennek legelterjedtebb módja, hogy a rendszerben működik egy (vagy több) név szolgáltatás, ahová az új szervereknek be kell jelentkezni, regisztrálni kell. A szerver regisztrációkor kap egy azonosítót. A kliensek a szerver azonosítóját a név szolgáltatótól kapják meg.

Csoportos multicast

A **csoportos multicasting** esetén a folyamatok szintén üzenetküldéssel kommunikálnak, azonban ebben az esetben az üzenetet nem egyetlen folyamatnak, hanem egy folyamatcsoportnak küldik el. Egy csoportos üzenetküldéshez a csoport minden egyes folyamatának egy üzenet fogadása kapcsolódik. Szokás megkülönböztetni a **broadcastot (üzenetszórás)** a **csoportos multicasttól.** Az üzenetszórást elterjedten használják a lokális hálózatoknál. Ekkor az üzenetet mindenki megkapja, aki a hálózathoz kapcsolódik. A csoportos multicast ezzel szemben valamilyen logikai csoportosítás alapján szelektál ebből a sokaságból.

A csoportos multicast használatát az alábbi motivációk támasztják alá:

- *Objektumok megkeresése.* (Például ha több állományszerver van a rendszerben, és keresünk egy állományt, akkor csoportos multicastot küldhetünk a szervereknek, és csak az válaszol, aki tárolja a keresett állományt.)
- *Hibatűrés.* (Például a kliens a kérését nemcsak egyetlen szervernek küldi el, hanem szerverek egy csoportjának. Ha a szervercsoport valamelyik szervere meghibásodik, akkor egy másik szerver nyújtja a kért szolgáltatást.)
- *Többszörös frissítés.* (Esemény értesítésre szolgálhat. Például egy pontos idő szolgáltató adott időközönként „szétsugározhatja" a pontos időt másodlagos idő szolgáltatóknak „most 17:00 az idő" jellegű üzenetekkel.)

A hálózati hardver nem biztos, hogy támogatja a csoportos multicastot. Ebben az esetben szoftverből, az üzenetek szekvenciális elküldésével lehet csoportos multicastot megvalósítani.

Függvényszállítás

Tulajdonképpen a **függvényszállítás** tekinthető a kliens–szerver-modell egy speciális esetének is. Míg a kliens–szerver-modellben tisztán adatok áramolnak (bár az első üzenet azt határozza meg, hogy a szerver mit is hajtson végre a kliens számára), addig a függvényszállítás esetén utasításblokkok, eljárás definíciók is mehetnek az üzenetekben, a szerveren interpreter értelmezi azokat. Ezzel a technikával a szerver képességei dinamikusan bővíthetők, egy speciális feladat megoldására „kiképezhető". A függvényszállítás legismertebb példái a postscript nyomtatók és például a Sun NeWs ablakozó rendszer.

Programstruktúra

A centralizált számítógép-rendszerek operációs rendszerét gyakran nevezik **monolitikusnak**, mert az általuk nyújtott absztrakciókat egy mereven lezárt interfész biztosítja, mint például a Unix-rendszerhívás interfésze. Ezzel szemben az elosztott rendszerekben az alkalmazói programok számos szolgáltatást érhetnek el, amelyek mind a saját interfészüket biztosítják a bezárt erőforrások eléréséhez. Mivel az elosztott rendszereknél a nyitottság kulcs- szerepet tölt be, így azok számára egy merev, lezárt interfész alkalmatlan.

A centralizált és az elosztott rendszerek **réteg struktúrája** eltérő képet mutat. A 4.6. ábra mutatja a centralizált rendszerek rétegstruktúráját.

Alkalmazások

Programnyelvi támogatás

Operációs rendszer

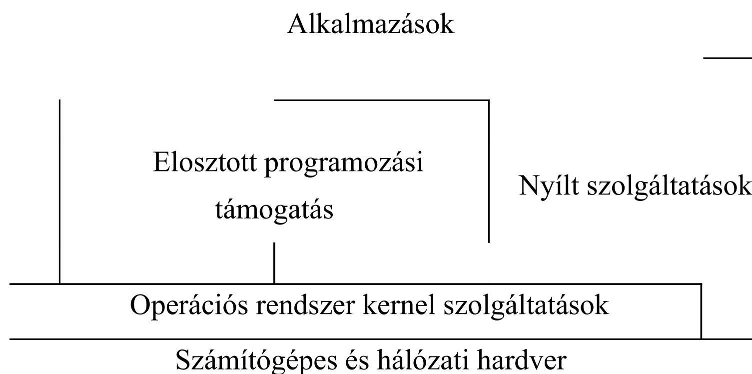
Hardver

4.7.ábra. táblázat - Elosztott rendszerek réteg struktúrája

Mint az jól látszik, a négy réteg – a hardver, operációs rendszer, a programnyelvi támogatás és az alkalmazások – egymásra épülnek. Minden két réteget egy határfelület választ el, a felsőbb rétegek csak az

alattuk lévő rétegek szolgáltatásain keresztül érhetik el az alsóbb rétegeket. Ebben az elrendezésben az operációs rendszer a fő szoftverréteg. Az operációs rendszer kezeli a legfontosabb erőforrásokat és fontos szolgáltatásokat biztosít az alkalmazások és a felhasználók számára. A főbb feladatok magukban foglalják a memóriaallokációt és -védelmet, a folyamatok létrehozását és a processzor ütemezést, a periferiális berendezések kezelését, a felhasználó hitelesítést és hozzáférés szabályozást, az állomány kezelést, az óra szolgáltatásokat stb.

Ezzel szemben az elosztott rendszerek eltérő rétegszerkezetet mutatnak (4.7. ábra).



4.8.ábra. táblázat - Az RPC-üzenetek felépítése

Itt már nem csak két szomszédos réteg található, illetve az operációs rendszer feladatai az alapvető erőforrások – a memóriaallokálás és -védelem, a folyamatok létrehozása és processzor ütemezés, a folyamatok közötti kommunikáció és a periferiális berendezések kezelése – kezelésére korlátozódnak.

A nyitottság itt azt jelenti, hogy az elosztott rendszereket egy adott felhasználói kör vagy alkalmazási kör igényeinek megfelelően lehet konfigurálni.

Terhelés szétosztás

Egy centralizált rendszerben a rendelkezésre álló számítási és memória erőforrásokkal az operációs rendszer a pillanatnyi terhelésnek leginkább megfelelő módon gazdálkodik. Az elosztott rendszerek legegyszerűbb architektúrájában munkaállomások kapcsolódnak össze egy lokális hálózattal, és ehhez a hálózathoz kapcsolódnak még különböző szerverek, mint például állomány szolgáltatók, nyomtató szolgáltatók stb. Ebben a struktúrában a munkaállomások képviselik a számítási kapacitást. Ezt a modellt szokás **munkaállomás-szerver modellnek** nevezni. Mint azt majd a későbbiekben látni fogjuk, a felhasználói kezelői felület konzisztencia szempontjából előnyös, ha az erősen interaktív alkalmazások feladatai magán a munkaállomáson futnak, amely képernyője előtt ül a felhasználó, mivel így a hálózati késleltetés kiiktatható. Ezt a munkaállomás-szerver modell hatékonyan tudja megvalósítani. Nyilvánvaló azonban, hogy az egyszerű munkaállomás-szerver modell nem optimális a rendszerben található számítási és memória erőforrások kihasználása tekintetében, továbbá nem teszi lehetővé, hogy egyetlen, nagy számítás- és memóriaigényű feladatokat futtató felhasználó további számítási és memória-erőforrásokra tegyen szert.

A rendelkezésre álló számítási kapacitás jobb kihasználását célozza a munkaállomás-szerver modell módosításán alapuló **processzor pool modell**. A modell elsődleges célja, hogy a rendelkezésre álló számítási kapacitást dinamikusan, az igényeknek megfelelően ossza szét a felhasználók között. Ebben a modellben akár egyetlen felhasználó is használhatja a processzor pool nyújtotta teljes számítási kapacitást, így nagy rugalmasságot biztosítva a számítási kapacitás rendszeren belüli felhasználásában. A processzor pool modellt támogató elosztott rendszerek egy munkaállomás rendszerből állnak, amihez hozzácsatoltak egy vagy több processzor poolt. A processzor pool elemei alacsony költségű számítógépek, gyakran nem tartalmaznak mást, mint egy processzort, memóriát és hálózati csatolót. A költségkímélés miatt gyakran ezek a számítógépek egyetlen alaplapra vannak integrálva és osztoznak egy közös tápegységen. A modell segítségével a felhasználók kis hardver erőforrásokkal rendelkező számítógépeken, sőt hálózati terminálokon – mint például az **X terminálok** – is végezhetnek hasznos munkát, hisz a processzor pool nyújtotta számítási kapacitás a rendelkezésükre áll. Ekkor a felhasználó munkaállomása vagy terminálja csak eszközt biztosít a pool számítási kapacitásának kihasználásához. Mint láttuk, ezen modell járulékos hardver elemeket igényel. Egy munkaállomás-szerver modellben a tétlen, kihasználatlan számítási kapacitással rendelkező munkaállomások fölös kapacitását a munkaállomás-szerver modell szoftver kiterjesztésével hasznosítani lehet. Ez nem igényel járulékos hardver elemeket, azonban a terhelés szétosztás gondos körültekintést igényel az azt megvalósító szoftver rendszertől.

Meg kell itt említenünk, hogy a manapság rendelkezésre álló ún. **osztott memóriás multiprocesszoros számítógép rendszerek** hatékonyan alkalmazhatók mindkét fent ismertetett modell megvalósítására, különösen nagy számításigényű, több processzorra szétosztható feladatok megoldására. Ilyen feladatok gyakran jelentkeznek leterhelt szerverek esetén, ott tág tér nyílik ezen architektúra alkalmazása számára.

Konzisztencia-karbantartás

Az elosztott rendszerekben a számítási erőforrások széttagoltsága miatt fokozottan kell foglalkozni a **konzisztencia** kérdésével. Az alábbiakban áttekintjük az elosztott rendszerekben felmerülő különféle konzisztenciaproblémákat.

A legfontosabb konzisztenciatípusok:

- frissítés konzisztencia,
- másolat konzisztencia,
- cache konzisztencia,
- hiba konzisztencia,
- óra konzisztencia,
- felhasználói interfész konzisztencia.

Frissítés konzisztencia

A frissítés konzisztencia nem csak az elosztott rendszereknél jelentkezik. A probléma számos olyan alkalmazásnál felmerül, ahol osztott adathasználatra van szükség. A legkézenfekvőbb ilyen alkalmazások az adatbázis kezelő alkalmazások. Az elosztott rendszereknél azonban kiemelten foglalkozni kell a frissítés konzisztenciával, mivel egy elosztott rendszerben nagy valószínűséggel nagyon sok felhasználó fér hozzá az osztottan használt adatokhoz (például egy elosztott állományrendszerhez), másrészt magának a rendszernek a működése függ bizonyos adatbázisok konzisztenciájától. A rendszerben tehát biztosítani kell, hogy összetartozó adatok változását a folyamatok atominak, azonnal végbemenőnek érzékeljék, annak ellenére, hogy technikailag a változások bizonyos időintervallum alatt következnek be.

Másolat konzisztencia

Egy fontos adatforrásról másolatok készítése és a másolatok, replikák használata elterjedt módszer az adatbázisok területén. A másolatok használatának hatékonysági előnyeit az elosztott rendszerekben is kiaknázzák. Azonban a másolatok készítése és használata (amennyiben a másolatokat az egyes felhasználók módosíthatják) magában hordozza az inkonzisztencia veszélyét. Mivel az adatok egy közös forrásból származnak, így a másolatoknak, replikáknak minden pillanatban azonosaknak kell lenniük. Ezt az azonosságot elég nehéz biztosítani, mivel az elosztott rendszerben az információ átadásra egy kommunikációs hálózatot kell használni, ami a fizikai széttagoltság miatt eltérő késleltetéseket jelent az egyes komponensek felé. Egy tipikus másolat inkonzisztencia típus, amikor az információt hordozó üzenetek eltérő sorrendben érkeznek be az egyes rendszer komponensekhez, akár a logikai ok-okozat sorrendet is felborítva. Az Internet *netnews* tipikus példája a másolat inkonzisztenciának. Itt előfordulhat, hogy egy hírcsoportnak küldött üzenet bizonyos csomópontokra inkonzisztens sorrendben érkezik meg – a kérdésekre adott válaszok gyakran hamarabb megérkeznek, mint maga a feltett kérdés.

Cache konzisztencia

A cache-elési technikák minden számítógépes rendszerben, így az elosztott rendszerekben is kulcsszerepet töltenek be. Nélkülük számos elosztott szolgáltatás hatékonysága elfogadhatatlan lenne. Például, ha egy folyamatnak minden egyes alkalommal igénybe kellene vennie a név feloldási szolgáltatást, ahelyett, hogy a helyi cache-ben keresné ki a kérdéses információt, akkor a rendszer működése szinte elfogadhatatlanná válna. Azonban ez a helyzet nagyon hasonlít a másolat konzisztenciánál vázolt helyzethez. Amenynyiben a cache-ben tárolt globális adatot egy másik folyamat, csomópont módosítja, akkor a helyi cache-ben tárolt információ érvénytelenné válik. Ezért biztosítani kell valamilyen mechanizmust, amivel a helyi másolatokat frissíteni lehet. Tipikus megoldás, hogy a szerver tárolja, hogy egy adott adatelemet mely csomópontok kérdeztek le, így mely csomópontok lokális cache-ében találhatók meg nagy valószínűséggel. Amikor az adott adatelem módosul, a szerver egy üzenetet, értesítést küld az összes érintett csomópontnak.

Hiba konzisztencia

Egy centralizált számítógép rendszer meghibásodása esetén az azon futó folyamatok egyformán érzékelik a hibát, egyszerre hibásodnak meg – ilyen rendszerekben egyetlen hibamód jelentkezik. Elosztott rendszerekben ha egy komponens meghibásodik, akkor a többi komponens folytatja futását, még azok is, amik a meghibásodott komponenssel együttműködtek. Ha azonban az együttműködő komponensek, folyamatok a későbbiekben függnek a meghibásodott komponens további működésétől, akkor valamilyen későbbi időpontban ezek is meghibásodhatnak. Ebből adódóan eltérő fázisban történhetnek újabb meghibásodások. Ez az elosztott rendszerek *többszörös hibamódja*. Az együttműködő programok eltérő pontot érhetnek el a meghibásodásig. Ahhoz, hogy biztosítani lehessen, hogy az összes program által tárolt állandó adatok konzisztensek maradjanak, a meghibásodás után felépülési eljárásokkal az állandó adatokat vissza kell görgetni valamilyen ismert, konzisztens állapotba.

Óra konzisztencia

Az alkalmazásokban és rendszer szoftverekben alkalmazott algoritmusok jelentős része időbélyegeket alkalmaz, konzisztens működésük függ az időbélyegekbe foglalt idő információ konzisztenciájától. Centralizált rendszerben ez nem jelent problémát, mert az egy gépen futó folyamatok ugyanazon idő szolgáltatótól – ebben az esetben a rendszer órájától – szerzik be az idő információt. Egy elosztott rendszerben azonban minden folyamat a saját gépnek az óráját használja. Sajnos ezek az órák nem szinkronban járnak, így elosztott rendszerekben a fizikai idő kezelése problémát okozhat. A problémát a számítógépek közötti információküldés véges sebessége és az egyes gépek közötti eltérő mértékű kommunikációs késleltetés okozza. A konzisztencia biztosítása érdekében az órákat időről-időre valamilyen módszerrel szinkronizálni kell. Ezek az algoritmusok figyelembe tudják venni a hálózati késleltetést és számos alkalmazás számára kellő pontosságú idő információt tudnak biztosítani.

Szerencsére egy elosztott rendszerben számos esetben nem az abszolút idő pontos ismeretére van szükség, hanem események, mint például állományok frissítésének ideje, relatív sorrendje a fontos. Ezen problémára kidolgozták a *logikai órákat*, amik segítségével az esemény sorrendezés problémája kezelhető az elosztott rendszerekben.

Felhasználói kezelői felület konzisztencia

Ez a probléma elsősorban elosztott környezetben futtatott interaktív feladatokra vonatkozik. Ergonómiai mérések azt támasztják alá, hogy az ember 0,1 másodperces késleltetést még nem érzékel megszakításnak. Így az interaktív rendszerekben a válaszidőt ez alatt a határ alatt kell tartani. Elosztott rendszerekben, ahol a kezelői felületen végrehajtott változások gyakran valamilyen távoli számítási csomóponton indítanak műveleteket, ez szigorú megkötést jelenthet. Az interaktív késleltetés a felhasználói beavatkozás szerverhez történő elküldésének, feldolgozásának, a válasz visszaküldésének és a képernyő állapotának megváltoztatásához szükséges idők összege. Amennyiben a hálózati késleltetés jelentős, akkor a felhasználó érzékeli az elosztottságot, a rendszer nem lesz átlátszó. Ezen probléma kezelésére is gyakran alkalmazzák a cache-elési technikákat.

4.4.2. Elosztott fájlrendszerek

Az elosztott fájlkezelés a helyi operációs rendszer fájlkezelési szolgáltatásainak kiterjesztése számítógép hálózaton keresztül kapcsolódó számítógépekre.

Az elosztott fájlrendszer felhasználói a rendszert alkotó gépeken található fájlokat használják a fájl pontos helyének ismerete nélkül. A fájl lehet **helyi** *(local)*, azaz a felhasználó számítógépéhez kapcsolódó valamelyik háttértáron elhelyezkedő, illetve **távoli** *(remote)*, azaz egy másik számítógéphez csatolt periférián található. Ideális esetben a fájl tényleges elhelyezkedése a felhasználó előtt rejtve van.

Az elosztott fájlrendszer az elosztott rendszerek *(distributed systems)* tipikus alkalmazási példája. Elosztott rendszerek esetén a szolgáltatások hálózatba kötött számítógépeken úgy működnek, hogy a felhasználónak (és alkalmazásainak) nincs tudomása azok tényleges elhelyezkedéséről, illetve nem szükséges azt tudniuk, hogy mi módon építhető ki a kapcsolat saját kliens gépeik és a szolgáltatásokat üzemeltető szerver gépek között. (Ezzel ellentétben a hálózati modellben a hasonlóképpen elosztott szolgáltatások eléréséhez szükséges az elérési mód ismerete.)

4.4.2.1. Az elosztott fájlrendszer szolgáltatás

Elosztott fájlrendszerek esetén a fájlt tároló számítógép a **szolgáltató**, amely a többi csomópont, az **ügyfelek** számára szolgáltatásként műveleteket biztosít a fájljain. Az előző fejezetben ismertetett *Telnet* és FTP protokollok is lehetővé teszik a távoli fájl hozzáférést, illetve fájlok mozgatását a hálózatba kapcsolt gépek között, de egyrészt ezek használata külön tudást igényel, másrészt nem használják ki igazán a kliens gép operációs rendszere nyújtotta lehetőségeket. Az elosztott fájlrendszer egy olyan megoldás, amely a felhasználó és alkalmazásai számára a helyi hozzáféréssel azonos módon teszi lehetővé műveletek végrehajtását távoli állományokon.

Az elosztott fájlkezelést kezdetben a hagyományos operációs rendszerre épülő olyan szoftver réteg valósította meg, amely több operációs rendszer fájlkezelése között teremtett kapcsolatot. A korszerű elosztott operációs rendszerek integráns része az elosztott fájlkezelés.

Legelterjedtebb a UNIX operációs rendszer NFS *(network file system)* elosztott fájlrendszere, amely PC-alapú operációs rendszerekből is elérhető. Manapság egyre inkább terjed a PC-alapú rendszerekből kialakult másik elosztott fájlrendszer, az CIFS *(Common Internet File System)*, amely az SMB *(Server Message Block)* protokollon alapszik.

A fájlokat egyedi nevük alapján azonosíthatjuk. A név a felhasználó elől elrejti a fájl elhelyezkedésének, tárolásának részleteit. Elosztott fájlkezelés esetén a névnek azt is el kell(ene) takarnia, hogy a fájl melyik számítógép háttértárján található.

4.4.2.2. Az állományok azonosítása

Az állományok azonosításánál két különböző szintet különböztetünk meg, a felhasználói szintű *neveket*, illetve a rendszerszintű *fájlazonosítókat*. A fájlkezelő feladata a kétszintű azonosító egymáshoz rendelése.

A fájlnevek és a fájlok a felhasználó számára átlátszó *(transparent)* egymáshoz rendelésénél két fogalmat különbözhetünk meg:

- **rejtett elhelyezkedés** *(location transparency):* a fájl *neve* nem utal arra, hogy az melyik gépen található,
- **elhelyezkedés-függetlenség** *(location independence):* a fájl *neve* nem változik meg akkor sem, ha a fájl átkerül egy másik gépre.

Szorosan ide tartozik a fájlok az elosztott rendszer felügyelete alatti, kezdeményezésre történő vándorlásának *(file migration)* fogalma.

Míg az első esetben a felhasználói nevek leképzése statikus táblázatok alapján történhet, addig az elhelyezkedés-független elnevezési rendszerben dinamikusan változó leképzési információt kell használni. A jelenleg elterjedt, kiforrott elosztott fájlkezelő rendszerek nem támogatják ezt a – lényegesen bonyolultabb – módszert.

Az elhelyezkedés-független nevek előnyei:

- Az elnevezés elrejt minden, a fizikai tárolással kapcsolatos információt, a fájl az információtárolás teljesen absztrakt fogalma marad.
- A fájlvándorlás lehetőséget nyújt arra, hogy az elosztott operációs rendszer a rendelkezésre álló teljes háttértár területet egységesen kezelje, a szabad területekkel rendszerszinten gazdálkodjon. Lehetőség van a háttértárak kihasználtságának dinamikus kiegyensúlyozására.
- Az elnevezési rendszer szerkezete teljesen független a számítógépek összekapcsolódásának konkrét szerkezetétől, nem szükséges speciális fájlokat – például egy egységes könyvtár-struktúrában a gyökér könyvtárat – előre kijelölt csomópontokon tárolni.

4.4.2.3. Elnevezési módszerek

Az elnevezési rendszer feladata, hogy a fájlneveket leképezze konkrét csomópontra és azon belüli fizikai elhelyezkedésre.

- *Csomópont explicit megnevezése*. A fájlokra hivatkozás két részre bontható: a csomópont megnevezése és a csomópont helyi fájlrendszerében a fájl neve; például a VMS operációs rendszerben <csomópont név>: :<fájlnév>. Bár ez a fajta megnevezés nem felel meg a rejtett elhelyezkedés követelményének, hiszen a felhasználónak a hivatkozásban meg kell jelölnie a fájl konkrét helyét,

ám a módszer a hálózati operációs rendszereknél többet nyújt azzal, hogy a távoli fájlokon a helyi fájlokkal teljesen azonos műveletek végezhetők.

- *A távoli fájlrendszer a helyi könyvtár-hierarchia része*. A távoli fájlrendszert – vagy annak egy részét – a helyi könyvtár-hierarchia egy pontjára csatlakoztatják *(mount)*. Ehhez a művelethez meg kell nevezni a távoli gépet is, de ezután az ott lévő fájlok a helyi fájlokkal azonos módon kezelhetők. Ez a megoldás található például a UNIX operációs rendszerekben elterjedt NFS *(Network File System)* rendszerben. A megoldás korlátozott érvényű – csak a távoli gépen felajánlott hierarchiarészek láthatók – és egy bizonyos fájlra a hálózat különböző gépein különböző elérési útvonalon kell(het) hivatkozni.
- A teljes elosztott rendszert lefedő egységes elnevezések: A fájlokra globális, a rendszer egészén érvényes nevekkel hivatkozhatunk.

A fentebb felsorolt leképzési módok táblázatok segítségével valósíthatók meg. Az implementáció lehetőségei és problémái:

- *Állománycsoportok szerinti leképzés*. Ha a leképzési táblákban minden fájl szerepel, a táblák kezelhetetlenül nagyokká válhatnak. Célszerűbb a fájlokat csoportokba *(component units)* szervezni és a leképzést ezekre a csoportokra együtt végezni. Például az NFS rendszerben ilyen csoportnak tekinthetők a (távoli) könyvtárak, könyvtár-hierarchiák.
- *Leképzési táblák többszörözése*. A leképzési táblákban tárolt információhoz a rendszer minden csomópontja hozzá kell férjen. Ez megoldható egy központi nyilvántartással is, de így ezt a nyilvántartást kezelő csomópont kiesése a rendszer szétesését vonja maga után. A táblázatok többszörözésének indokai:
  - decentralizált, hibatűrő rendszer,
  - adatbiztonság,
  - a táblázatok gyors elérése.
- Nem szükséges a teljes táblázatot a rendszer több csomópontján tárolni, elég lehet a táblázatok egy – a helyi rendszer számára szükséges – részének helyi átmeneti tárolása is *(caching)*. A fájlok vándorlásánál természetesen a leképzési táblákban tárolt információt minden másolatban aktualizálni kell.
- *Kétszintű leképzési táblák*. A több leképzési táblában történő módosítást egyszerűsíti, ha bevezetünk elhelyezkedés-független alacsony szintű fájl azonosítókat. A „felső” szintű leképzés a felhasználói neveket ezekre az azonosítókra képzi le, amelyek tartalmazzák azt, hogy a fájl melyik csoportba tartozik. Egy másik leképzési mechanizmus végzi el a csoportok csomópontokra történő leképzését.

Az elosztott fájlkezelés a fájloknak a rendszerben való megtalálásához szükséges idő, a hálózaton történő megbízható adatátvitelhez szükséges információtöbblet és a nagyobb adminisztrációs teher miatt óhatatlanul lassabb, mint a helyi fájlkezelés, de rejtett szolgáltatás esetén az elosztott fájlkezelés teljesítményének a hagyományos fájlrendszerrel összehasonlíthatónak kell lennie. Az elosztott fájlkezelés teljesítménye a következőképpen növelhető.

- Speciális hardverelemekkel:
  - gyors kommunikációs csatornákkal,
  - a szolgáltatók nagy sebességű háttértáraival,
  - a szolgáltatók speciális architektúrájával, nagy központi táraival;
- Szoftvermódszerekkel:
  - gyors, kevés kiegészítő információt tartalmazó kommunikációs protokollokkal,
  - a szolgáltató operációs rendszerének, ütemezési algoritmusainak a feladathoz hangolásával,
  - a szolgáltatónál, illetve az ügyfélnél végzett átmeneti tárolással.

4.4.2.4. Az ügyfelek kéréseinek kielégítése

Miután a név alapján a rendszer megtalálta a hivatkozott fájlt, a felhasználó a fájlokon a helyi fájlkezelésnek megfelelő műveleteket akarja végezni. Ezt két elvileg különböző, a gyakorlatban gyakran összefonódó módon lehet elvégezni.

(a) Távoli szolgáltatásokon keresztül. A felhasználói műveletek mint kérések eljutnak a szolgáltató csomóponthoz, amely ezen kéréseket végrehajtja, majd az eredményt – amely lehet például a fájlban tárolt információ, illetve a művelet végrehajtásának eredményessége – a hálózaton visszaküldi a kérőnek. A hálózaton a szolgáltatóhoz továbbított kérések pontosan megfeleltethetők a helyi fájlrendszer felé kiadott műveleteknek: írás, olvasás, létrehozás, törlés, átnevezés, attribútumok lekérdezése, módosítása.

A távoli szolgáltatást tipikusan a távoli eljáráshívás mechanizmusára építve implementálják:

– Minden, a fájlrendszernek szóló művelethez tartozik egy távoli eljárás. Az ügyfélfolyamatok a kívánt művelet paramétereit az eljárás paramétereibe írják, majd meghívják a megfelelő eljárást. A szolgáltató az eljárás visszaadott paramétereiben válaszol a kérésre.

– A szolgáltatóban minden távoli eljáráshíváshoz tartozik egy speciális „démon” *(daemon)* folyamat, amely feladata, hogy a különböző kliensek felől érkező kéréseket kiszolgálja. A folyamat egy hozzá rendelt kaput (portot) figyel, ha ott egy kérés érkezett, a szükséges műveletet végrehajtja, majd a választ a kérésből megállapítható feladónak küldi vissza. (Az ügyfél folyamat és a démon között aszimmetrikus megnevezésen alapuló

kommunikáció valósul meg. Az ügyfélnek meg kell neveznie egy kaput, és ezzel a hozzá tartozó démont, a démon azonban bárkitől elfogad kéréseket.)

(b) Helyi átmeneti tárak segítségével. Az elosztott fájlrendszer teljesítményének növelésére, a hálózati adatforgalom csökkentésére a helyi gépek a szükséges fájlokat, illetve azok részeit átmenetileg tárolják, a kívánt műveleteket azon végzik. A módszer elve azonos a virtuális tárkezelésnél megismert átmeneti tárkezeléssel *(caching)*:

– ha szükséges, az új információnak helyet csinálunk,

– a szükséges információt a hálózaton keresztül az átmeneti tárba töltjük,

– a műveleteket a helyi másolaton végezzük,

– változás esetén a módosult információt visszaírjuk a távoli gépre.

A távoli eljáráshívás speciális problémái

- A hálózati kommunikáció nem megbízható. A fájlkezelő rendszernek vigyáznia kell arra, hogy minden kérést pontosan (legfeljebb) egyszer hajtson végre. A megoldáshoz a kéréseket az ügyfelek sorszámozzák – időbélyeget *(time stamp)* fűznek hozzá –, a szolgáltató az egyszer már kiszolgált kéréssel azonos sorszámú következő kéréseket elhanyagolja.
- Az ügyfél programokban a távoli eljárásokat a szolgáltató megfelelő kapuira kell leképezni, ahol a kívánt démon várakozik. A leképzés lehet
    - statikus, minden művelethez előre kijelölt kapu tartozik,
    - dinamikus, műveletenként a megfelelő kapu számát egy speciális, mindig egy kötött kapuban ücsörgő démonnak, a házasságközvetítőnek *(matchmaker)* küldött kérésre adott válaszból kapjuk.

A helyi átmeneti tárak alkalmazásának problémái

- Az átviteli egység méretének meghatározása. A gyakorlatban a különböző rendszerek a háttértár egy blokkjától teljes fájlokig különböző méretű egységekben végzik az átvitelt.

Figyelembe kell venni

-
    - a rendelkezésre álló átmeneti tár méretét, (ne legyen túl kevés blokk benne),
    - az alapszintű hálózati protokoll, illetve az *RPC* mechanizmusában megengedett blokkméretet.
- Hol legyen az átmeneti tárolás?

    - Általában a helyi gép központi tárában; ez gyors megoldás és háttértár nélküli rendszerekben is alkalmazható.
    - A helyi gép háttértárán; ami megbízhatóbb, hiszen a helyi gép katasztrófája után is megmarad az átmeneti tárban lévő információ, vagy teljes fájlok egy időben történő másolásánál a központi tár ezeknek a tárolására már nem elegendő.
- A változások érvényre juttatása *(update).* A helyi másolaton végzett módosításokat a szolgáltatóval közölni kell. Ez történhet:
    - azonnal; lassú, de biztonságos(abb) módszer,
    - késleltetve *(delayed write)*; gyors, de a helyi rendszer esetleges katasztrófája miatt nem biztonságos megoldás.

Az írás történhet:

-
    - ha szükség van az átmeneti tárban helyre,
    - időközönként,
    - a fájl lezárásakor.
- Az átmeneti tár konzisztenciája. Ugyanazt a fájlt az elosztott rendszerben több ügyfél használhatja egyszerre, amennyiben valamelyik a fájlt megváltoztatja, a többi ügyfél másolata már nem aktuális. A másolat felújítása történhet
    - az ügyfél kezdeményezésére (az ügyfél, ha „gyanakszik”, kérheti a szolgáltatótól annak ellenőrzését, hogy a saját másolata helyes-e),
    - minden hozzáférésnél,
    - a fájl újra megnyitásánál,
    - időközönként,
    - a szolgáltató kezdeményezésére.

A szolgáltató nyilvántartja összes ügyfeléről, hogy azok mit tárolnak helyileg (speciális üzeneteket igényel az ügyfelektől), ha a fájljaiban olyan változás van, amely valahol inkonzisztenciát okoz, értesíti az ügyfeleket. Az, hogy a szolgáltató mikor értesíti az ügyfelet arról, hogy a másolata érvénytelen, a rendszerben megvalósítandó elosztott fájlkezelési stratégia *(consistency semantics)* függvénye: történhet azonnal vagy csak akkor, amikor a fájlt a módosító lezárja. Amennyiben a szolgáltató értesül arról, hogy az ügyfelek mikor és milyen műveletekre nyitnak meg egy fájlt, előrelátható problémák esetén már megnyitáskor üzenhet az ügyfélnek, hogy ne alkalmazzon helyi tárolást, hanem használja a távoli szolgáltatásokat.

A két módszer összehasonlítása

Az átmeneti tárolás előnyei:

- a műveletek jelentős része helyben végrehajtható, az ügyfél programjai lényegesen gyorsabban futhatnak, csökken a hálózat, illetve a szolgáltató terhelése,
- nagy blokkok egyszerre történő átvitele viszonylag gazdaságosabb, mert
    - a hálózati protokoll a blokk méretéhez képest kevesebb többletinformációt jelent,
    - a szolgáltató lemezműveletei is gyorsulhatnak, ha egyszerre nagy blokkok átvitelét kérjük.

A távoli szolgáltatás előnyei:

- nincs konzisztencia probléma, hiszen a fájl egyetlen példányát a szolgáltató kezeli. A konzisztencia biztosítása bonyolult algoritmusokat, hálózati többletforgalmat igényelne,
- ott is működik, ahol az ügyfél erőforrásai nem teszik lehetővé az átmeneti tárolást,
- a távoli szolgáltatások elérésének felülete megfelel a helyi fájlkezelő szolgáltatások igénybevételének módjával.

4.4.2.5. A szolgáltató implementációja

- **Állapotot tároló** *(stateful)* szolgáltató. A szolgáltató az ügyfelek kéréseiről, kiszolgálásuk folyamatáról, állapotáról a saját központi tárában információkat tárol. Az új fájlok megnyitásáról értesül, és ehhez egy kapcsolat-leírót készít, amellyel a lezárásig nyomon követi az átvitelt. Előnyei:

– nagyobb teljesítmény,

– a fájlokhoz való egymás utáni hozzáférések már egy előkészített fájlra hivatkoznak,

– szekvenciális olvasás esetén a szolgáltató előre olvashat,

– az ügyfelek átmeneti tárolása miatti konzisztencia problémákat figyelheti, kezelheti.

Hátrányai:

– a szolgáltató leállásakor az állapotinformáció elveszik újrainduláskor,

– ilyenkor az ügyfelek átviteli kéréseit nem tudja kiszolgálni, az ügyfeleknek is terminálinuk kell,

– az ügyfelekkel konzultálva újra fel kell építenie az elveszett állapotinformációt, ez bonyolult protokollt igényel,

– a szolgáltatónak fel kell ismernie, ha egy ügyfél váratlanul terminált *(orphan detection)*, hogy a hozzátartozó állapotinformációt érvénytelenítse.

- **Állapot nélküli** *(stateless)* szolgáltató. A szolgáltató az ügyfelekről nem tárol semmilyen információt, az ügyfelek minden kérése önállóan is kielégíthető. (A fájlok megnyitására és lezárására nincs szükség.) Előnye, hogy az állomások meghibásodását jól tolerálja, ha az ügyfél nem kap választ a kérésére, egyszerűen újra kísérletezik. Hátránya, hogy lassú, mert

– kérésenként több információt kell átvinni,

– a fájlt minden kérés kiszolgálásához a helyi fájlrendszerben újra meg kell találni.

4.4.2.6. A fájlok többszörözése

Az elosztott fájlrendszerekben érdemes lehet egyes fájlokat megsokszorozni *(file replication)*:

- növeli a rendszer hibatűrő képességét, ha a másolatokat olyan csomópontokon helyezzük el, amelyek meghibásodás szempontjából függetlenek egymástól,
- gyorsabb kiszolgálást biztosíthat, ha a rendszer megtalálja a „legközelebbi” vagy a legkevésbé terhelt szolgáltatón lévő másolatot.

Felmerülő problémák:

- a fájl megnevezését a felhasználó számára láthatatlanul valamelyik másolathoz kell kötni,
- a rendszernek automatikusan kell kezelni a fájlok többszörözését, a másolatok különböző csomópontokon történő elhelyezését, megszüntetését,
- a szükséges változtatásokat az összes másolaton el kell végezni.

4.4.3. Folyamatkezelés

Az elosztott rendszerek kialakítását leggyakrabban a kliens–szerver-modellel és egy protokollal, a távoli eljáráshívással (remote procedure call, RPC) kötik össze, melyet gyakran használnak a modell megvalósítására.

4.4.3.1. Kliens–szerver-folyamatok

A **kliens–szerver-modell** alapötlete egy szoftver rendszer olyan particionálása, amely meghatározza a rendszer által nyújtott **szolgáltatásokat (szervizeket)** a hozzájuk tartozó algoritmusokkal **(szerverekkel)**, valamint tartalmazza az adott alkalmazásnak megfelelő **kliens programokat**, melyek a szerverek szolgáltatásait igénybe véve oldják meg a feladatot a felhasználó számára. Ebben a modellben a kliens alkalmazás programok nem kommunikálnak egymással, hanem direkt módon érik el a szerverek szolgáltatásait.

A szervereken ún. **démon folyamatok (daemon processes)** várják a kliensektől beérkező kéréseket. Egy démon folyamat általában csak a kapcsolat kezdeti kiépítését végzi el egy, a kliens kiszolgálásához rendelt szerver folyamathoz. A démon folyamatok állandóan futó folyamatok, melyek a szolgáltatás állandó elérhetőségét biztosítják. TCP/IP vagy UDP/IP protokoll feletti kommunikáció esetén a démon egy TCP vagy UDP porthoz kötött folyamat, amely a porton beérkező kérésekhez rendel kiszolgáló rutint. UNIX-rendszer alatt

az önálló folyamatokon kívül létezik egy általános megoldás is a feladat elvégzésére: az Internet démon (internet daemon, röviden inetd). Ennek több TCP és UDP port, és a hozzájuk tartozó szerverprogram is megadható egy erre szolgáló konfigurációs fájlban (inetd.conf), így egyetlen folyamattal lehet megvalósítani a kérések fogadását. A démon folyamatok UNIX alatt a háttérben futnak, nem tartozik hozzájuk terminál viszony. Elindulásuk után elszakadnak a terminál kapcsolatuktól, ily módon garantálva azt, hogy a terminál kapcsolat lebomlása után is tovább fussanak.

A **hálózati számítási modell**ben (amelynek alappillére a kliens–szerver-modell) a szerverek eléréséhez a kliens alkalmazásoknak tisztában kell lenni a szerverek pontos címével (például IP-cím és portszám), a szolgáltatásuk elérésének módjával, valamint a kapott eredmények értelmezésével. Az **elosztott számítási modell** szintén alkalmazza a kliens–szerver-megközelítést, de ebben az esetben a kliens elsősorban a szerverek által nyújtott szolgáltatások alapján éri el a szervert, mintsem annak címének pontos ismerete alapján. Egy elosztott rendszerben a kliens–szerver-kapcsolat kiépítését egy külön **kommunikációs közbülső réteg (middleware)** biztosítja.

Rengeteg példát találunk kliens–szerver-rendszerekre, mint a fájlszerverek (hálózati fájlmegosztás), adatbázis-szerverek, hálózati autentikációs szerverek, webszerverek stb. Ezek túlnyomó többsége a hálózati számítási modellt követi, kisebb részük, mint például az NFS és az elosztott objektumorientált rendszerekre épülő alkalmazások az elosztott számítási modell felé közelítenek.

Egy elosztott környezetben a szerverek egyszerre több klienssel is kapcsolatban állnak, és a kliensek egy időben több szerver szolgáltatásait is igénybe vehetik. A kliens és szerver folyamatokat tervezésük során alkalmassá kell tenni az ilyen többirányú kapcsolatok kiépítésére. Egy kliens és egy szerver közötti kapcsolat kiépítése, a **kötés (binding),** az a folyamat, amelynek során a kliens megtalálja a számára szükséges szolgáltatást nyújtó szervert, és kettőjük között kialakul a kommunikációs kapcsolat. Többféle kötési rendszer létezik, azaz többféle módot választhatunk a szerver kiválasztására és a kapcsolat kiépítésére. Például egy webszerver esetén a kliens (webböngésző) a szerver címének (internetcím és portszám) ismeretében a HTTP protokollban meghatározottak szerint küld egy üzenetet a szervernek. A szerver általában külön kiszolgáló eljárást rendel a klienshez (hogy egy időben több kérést is kiszolgálhasson), mely értelmezi a klienstől kapott üzenetet, elkészíti a választ és elküldi a kliensnek. A kliens–szerver kapcsolat a webrendszerben ekkor le is bomlik (nem perzisztens). Ha a gépünk óráját akarjuk beállítani, akkor ezt olyan kliens program segítségével is megtehetjük, amely nem közvetlenül egy szervert szólít meg a feladat megoldására, hanem a hálózatra egy szórt (broadcast) üzenetet küld, és az összes szervertől beérkező válasz alapján állítja be gépünk óráját.

A kötés folyamatát tovább bonyolíthatja, ha alkalmazásuk megbízhatóságának növelése, nagyobb adatbiztonság, vagy egyszerűen csak a gyorsabb válaszidők érdekében több, konkurrens, ugyanazt a szolgáltatást nyújtó szervert is üzemeltetünk. Ez a **replikáció (replication)**. Egy szerver kiesése (például az NFS rendszerben egy fájlszerviz leállása) ilyenkor a kliens–szerver-kötés megváltozását eredményezheti (NFS esetében a fájl

szerviz tartalék szervere szolgálja ki a következő klienskérést), gyakran olyan módon, melyről a kliens nem is szerez tudomást.

Átmeneti gyorsítótárak alkalmazása (caching) szorosan összefügg a replikációval. Ez a fájlrendszereknél és adminisztrációs adatbázisoknál gyakran alkalmazott technika a kliens által gyakran kért információt egy helyi átmeneti tárolóban helyezi el a klienshez közel. Ily módon a következő megegyező kérés kiszolgálási sebessége megnő, ugyanakkor az adatok konzisztenciájának biztosítása nagyobb feladatokat ró a szerver kidolgozójára. Az elosztott gyorsítótárak egy speciális fajtája a **proxy**, mely elsősorban a web- szerverek területén elterjedt. A proxy egy olyan szervert takar, amelyik valamilyen más szerveren tárolt információt helyileg is megőriz a kliensek számára. A proxy általában nem közvetlenül a szervertől kérdezi le az adatokat, hanem egy másik, a szerverhez a hálózaton közelebb elhelyezkedő proxytól. Ily módon a proxy-szerverek egy hierarchikus rendszert alkotnak, amelyben egy kliens kérés addig továbbítódik az adatot eredetileg tároló szerver felé, míg egy proxy átmeneti tárában azt meg nem találja. Egy proxy-rendszer elsősorban a hálózati kapcsolat lassúsága miatti időkésést hivatott csökkenteni.

Kliensek és szerverek közötti kommunikáció kialakításának legelterjedtebb formája a **távoli eljáráshívás (remote procedure call, RPC)**. Ennek során a szolgáltatásokat eljárások valósítják meg, a kliensek pedig a szolgáltatásokat az eljárások meghívásával veszik igénybe. A távoli eljárások meghívása az alkalmazások szempontjából nagyon hasonlatos a helyi eljárások meghívására – a lényeges különbséget (a hálózati kapcsolat kialakítását, az adatok átjuttatását és az eredmények visszaküldését) a távoli eljárást megvalósító rendszer elfedi az alkalmazások elől.

4.4.3.2. Távoli eljáráshívás – RPC

Az RPC-rendszer egy **protokoll-leírást** és egy **programozói interfészt** tartalmaz. Alapvetően egy kliensgép számára lehetővé teszi, hogy egy kiválasztott szervergépen egy előre meghatározott eljárást lefuttasson. A kliensgépen futó folyamat eljárás hívása a szervergépen futó folyamat egy eljárásának meghívását eredményezi. A bemenő paraméterek és a visszatérési érték átviteléről, valamint a szervereljárás meghívásáról az RPC-alrendszer gondoskodik. Több magasabb szintű szolgáltatás épül az RPC-rendszerre, mint például az NFS (Network File System) és a NIS (Network Information System).

Az RPC-protokoll alapvetően a kliens és a szerver közötti kommunikáció adatformátumát határozza meg. Ezenkívül foglalkozik az üzenetek átvitelének mechanizmusával, illetve a kapcsolatban álló felek azonosításával is. A protokoll megbízható kapcsolatot épít ki a kommunikáló felek között, azaz garantálja a kérések továbbítását a célállomásig, illetve a válaszok visszajuttatását. Bár az RPC alapvetően szállítási protokoll független rendszer, általában UDP/IP (User Datagram Protocol/Internet Protocol) felett implementálják, mely alapvetően megbízhatatlan (ellentétben a TCP/IP protokollal). Az RPC-réteg feladata a biztonságos átvitel megvalósítása az üzenetek periodikus megismétlésével a nyugtázás vételéig.

A 4.8. ábra egy tipikus RPC-kérés–válasz párt szemléltet. Az *xid* az átvitel azonosító, mely egyértelműen megjelöli a kérést. Az azonosítót a kliens egyedileg készíti minden üzenete számára, a szerver pedig ugyanezt az azonosítót alkalmazza a válaszában. Ily módon a kliens egyértelműen azonosítani tudja azokat a kéréseket, amelyekre választ kapott a szervertől. A *direction* mező azt jelzi, hogy az aktuális üzenet egy kérés, vagy egy válasz. Az *rpc_vers* mező az RPC-protokoll verziószámát rögzíti, a *prog* és *vers* azonosítók a szerver folyamat (RPC-szolgáltatás) program és verzió azonosítói. Ezek után az azonosítással kapcsolatos információk következnek, majd a kérésben a meghívott eljárás paraméterei, a válaszban pedig az eljárástól érkező eredmények.

Az RPC egy egységes, ún. **kiterjesztett adatreprezentációt (Extended Data Representation – XDR)** használ a hálózati adatforgalomban. Az XDR szabvány (Sun Microsystems) egy gépfüggetlen adatábrázolási mód. Többféle egyszerű adattípust definiál, illetve szabályokat határoz meg bonyolultabb adatstruktúrák létrehozására.

RPC-hívás	RPC-válasz
xid	xid
direction (= hívás)	direction (= válasz)
rpc_vers (= 2)	reply_stat
prog	Autentikációs
vers	információ
proc	accept_stat
Autentikációs információ	Eljárásspecifikus eredmények
Eljárásspecifikus argumentumok	

5.7.ábra. táblázat - A UNIX prioritási tartományai

Alapvető adattípusai a következők:

- Egész szám. 32 bites entitás, ahol a 0. bájt reprezentálja a legnagyobb helyiértékű bájtot. Előjeles egészeket kettes komplemens alakban tárolnak.
- Változó hosszúságú adatfolyam. Egy négy bájtos *hossz* mezővel leírt adat. A hossz mező után az adatfolyam következik. Az adatot nullák egészítik ki a négy bájtos határig.
- Szöveg. Az adatfolyamhoz hasonlóan egy *hossz* mezővel kezdődő ASCII karaktersorozat, nullákkal kiegészítve a négy bájtos határra (amennyiben a szöveg hossza épp néggyel osztható, nincs lezáró nulla, ellentétben a UNIX-rendszereknél szokásos gyakorlattal).
- Tömb. Azonos típusú elemekből álló vektor, mely szintén egy négy bájtos *hossz* mezővel kezdődik, majd a vektor elemei következnek. Minden elemnek néggyel osztható hosszúságúnak

kell lennie. Bár az elemeknek azonos típusúnak kell lenniük, hosszuk különbözhet (például szövegekből álló tömb esetén).

- Struktúra. A struktúrák komponenseit sorrendben tárolja a szabvány azzal a megkötéssel, hogy minden komponens hosszát néggyel oszthatóra igazítja nullák beszúrásával.

Az adatstruktúrák meghatározásán kívül az XDR egy formális nyelvet is bevezet az adatok leírására. Az RPC-rendszer is ezen nyelv kiterjesztését használja a távoli eljáráshívás formális leírására.

Az XDR alapvető hátránya, hogy az adatreprezentációjától jelentősen eltérő reprezentációt használó architektúrákon erőforrásigényes lehet a szükséges konverziók megvalósítása. Ez elkerülhető lenne, ha az adatok átalakítása helyett a kommunikációban részt vevő felek egymás adatreprezentációs különbségeiket tisztáznák csak. Ilyen megoldást alkalmaz a DCE (Distributed Computing Environment, OSF 1992) RPC az XDR helyett.

Az RPC az adatcserén és a távoli eljárások meghívásán kívül programozási eszközöket is nyújt a kommunikációs szoftverek megvalósításához. A már említett RPC nyelv, mely az XDR formalizmusát követi, alkalmas a szerver interfészének formális leírására. A formális leírásból az **rpcgen** program képes a szerver és a kliens programok megfelelő részeit, valamint a szükséges XDR konverziós függvényeket elkészíteni C nyelven. Az így kapott C forráskódú modulokat a kliens és szerver alkalmazással kibővítve kapjuk a teljes kommunikáló rendszert. A következő programlista egy egyszerű távoli idő szerver interfészét mutatja.

```
/*
* date.x – Specification of remote date and time service.
*/
/*
* Define 2 procedures:
* bin_date_1() returns the binary time and date (no arguments).
* str_date_1() takes a binary time, returns a human-readable string.
*/
program DATE_PROG {
version DATE_VERS {
long BIN_DATE(void) = 1; /* procedure number = 1 */
string STR_DATE(long) = 2; /* procedure number = 2 */
} = 1; /* version number = 1 */
} = 0x31234568; /* program number = 0x31234568 */
```

4.4.3.3. Szálak alkalmazásának előnyei

A mai RPC-rendszerekhez általában szorosan kapcsolódik a **szálak (threads)** alkalmazása, mely lehetővé teszi egy program számára, hogy több feladatot is végrehajtson egyazon időben. Bár a szálaknak kevés köze van a hálózati kommunikációhoz, mégis gyakran alkalmazzák őket a kliens–szerver-rendszerekben. Használatukkal ugyanis lehetővé válik a kliens kérések párhuzamos kiszolgálása, illetve párhuzamos szerver kérések elküldése egyazon folyamaton belül.

A szálak alkalmazásának több előnye is van. Ezek egy része a szerverek teljesítményét növeli, más részük a tervezéskor és megvalósításkor könnyíti a programozók dolgát.

Képzeljük el a következő példát!

Egy RPC-szerver a klienstől érkező feladata megoldásához más szerverek szolgáltatását is igénybe veszi. Azok további szervereket kapcsolnak a feladat megoldásába, míg végül valamelyikük az eredeti kliens valamely szolgáltatását szeretné igénybe venni. Kialakult a holtpont. Helyi függvényhívások esetén ez rekurzióhoz vezetne, az RPC esetén azonban holtpont alakul ki. Ez a helyzet feloldható a szálak alkalmazásával. Ha az RPC-szerver a beérkező eljáráshívásokhoz különálló szálakat rendel (melyek a folyamatokkal ellentétben képesek ugyanazon adatstruktúrákon dolgozni), a holtpont feloldódik, átalakul rekurzióvá.

A szálak alkalmazásával mind a szerverek, mind a kliensek párhuzamosíthatják futásukat az RPC-kommunikáció alatt. A szerverek minden beérkező RPC-kérésre különálló szálat indítanak el (adott maximumig). A kliens a távoli eljárást egy külön szálban indítja el, így annak eredményére várakozva más feladatokat is megoldhat. Egy távoli eljárás meghívása során a kliens és szerver programokban létrejövő szálakat logikailag egy szállá foghatjuk össze. Ez az **RPC-szál (RPC thread)**. Az RPC-szál a kliens oldalon keletkezik (mint kliens szál), majd kiterjesztődik a hálózaton keresztül a szerverre, ahol szerver szállá válik, amin belül végrehajtódik a távoli eljárás. Az eljárás lefutása után a szál „visszatér” a klienshez és ismét kliens szálként szolgáltatja a függvény visszatérési értékeit. A szerver egyszerre több RPC-szálat is fogadhat a kliens kérések kiszolgálása érdekében. Praktikusan létezik a szerverben párhuzamosan futtatható szálak számának egy maximális értéke, mely után a következő beérkező RPC-szál várakozni kezd mindaddig, míg egy szerver szálnak létrehozása lehetővé nem válik. Ez a mechanizmus része a DCE (Distributed Computing Environment) 1.1 RPC specifikációjának.

A szálak természetesen nem csak a távoli eljáráshívás modelljében használatosak. A mai korszerű operációs rendszerek mindegyike kínál megoldást a szálak létrehozására. Szálak alkalmazásával általában egyszerűbben megvalósítható konkurens futási környezetet alakíthatunk ki. Lehetővé válik a párhuzamos feldolgozás a lehető legkisebb vízfejjel – mind kernel, mind alkalmazás szinten. A kernel a szálak közötti váltást lényegesen kevesebb adminisztrációval oldja meg, míg az alkalmazások mentesülnek a folyamatok közötti kommunikációs terhektől a szálak közötti adatcserében.

A szálak alkalmazásának ugyanakkor vannak hátrányai is. Szálakat alkalmazó programoknál sokkal körültekintőbben kell eljárni a tervezés során. A szálak okozta konkurens adat- és erőforrás hozzáférés az alkalmazás készítőjére hárítja a kölcsönös kizárás tervezésének és megvalósításának feladatát. E nélkül a többszálú rendszer adatai könnyen inkonzisztens állapotba kerülhetnek. Másrészt a szálak futása nem szakad el teljesen a folyamat futásától. Egy operációs rendszerben a szálak megvalósításától függhet a folyamatok és a szálak futási viszonya és viselkedése. Amennyiben a szálakat felhasználói szinten, a kerneltől függetlenül valósítják meg **(user-level threads)** és egy szál kiad egy olyan rendszerhívást, ami blokkolható, az egész folyamat futása blokkolódik. Kernel szintű szálak megvalósítása esetén **(kernel-level threads)** ez a probléma nem áll fent. Ez azonban újabb követelményeket jelent a rendszerhívások megvalósításával szemben. Az ilyen operációs rendszerek külön jelzik, ha egy rendszerhívás biztonságos a szálak szempontjából **(thread-safe)**. Ettől eltérő esetben (a gyakorlati esetek többségében) a szálakban a rendszerhívások alkalmazása gondosan tervezendő (például kölcsönös kizárás alkalmazásával) az újrahívások elkerülése végett.

4.4.4. Időkezelés és koordináció elosztott rendszerekben

Mint azt korábban láttuk, az elosztott rendszerek számos tekintetben eltérő követelményeket támasztanak a rendszer funkcióival és kialakításaival szemben, mint a centralizált rendszerek. Az időkezelés sem kivétel ez alól: elosztott rendszerekben az idő, és annak konzisztens kezelése központi jelentőségű. Egyes alkalmazásokban a pontos, valós időre van szükség (például naplózás, számlakezelés), míg az esetek jelentős részében elegendő olyan időkezelés, amely biztosítja a rendszerben bekövetkező események sorrendezését. Az alábbiakban áttekintjük az ezen feladatok ellátásával kapcsolatos fogalmakat és a szükséges eszközöket.

4.4.4.1. Időkezelés

Az időkezelés két alapvető funkciót hivatott támogatni: a **valós idő pontos ismeretét** (ami elengedhetetlen például naplózási, számlázási feladatok ellátásához), illetve a rendszerben bekövetkezett **események sorrendezését (event ordering)**. Ez a két feladat eltérő igényeket támaszt az időkezeléssel szemben, és mint azt majd látni fogjuk, az előbbit fizikai órák, míg az utóbbit logikai órák használatával lehet a leghatékonyabban megoldani.

Koordinált Univerzális Idő (KUI)

Az emberek életében az idő kitüntetett szerepet tölt be. Míg a hétköznapi életben nagyon gyakran megelégszünk az idő „hozzávetőleges” ismeretével, számos, elsősorban csillagászati alkalmazások ennél precízebb idő fogalmat igényelnek. Ma az idő mérésére a legpontosabb eszköz az atomóra. Ennek pontossága 10^{-13} sec/sec.

A **Nemzetközi Atomidő** az atomóra pontosságával méri az eltelt időt, mely szerint egy másodperc 9192631777 átmenet a Cs^{133} alapállapotának 2 hiperfinom szintje között. Ez pontos időmérést tesz lehetővé. Azonban az emberiség az időt a csillagászati időhöz igazítja, így bevezették a **Koordinált Univerzális Idő –**

KUI (Coordinated Universal Time) fogalmát, aminél meghatározott időközönként szökőmásodperceket kell beiktatni, vagy törölni.

Ezt a koordinált univerzális időt használják a valós fizikai idő etalonjaként. Ezt az időt rádióállomások (például az amerikai WWV), űrszondák (például GOES: *Geostationary Operational Environmental Satellites*), (GPS: *Global Positioning System*) sugározzák, így amennyiben külső szinkronizálásra van szükség, akkor az ezeket a jelet vevő berendezések segítségével ez megtehető. A szinkronizálás pontossága az alkalmazott módszertől függően 0,1–10 milliszekundumos nagyságrendbe esik.

Ezen berendezések alkalmazásánál azonban a jel vételének módját figyelembe kell venni, mert az meghatározza a szinkronizálás pontosságát. Ha például Magyarország keleti és nyugati határánál egy-egy időszolgáltató egy közeli földi rádióállomás alapján szinkronizál a KUI-hez, akkor ezen két időszolgáltató órája között nagyságrendileg milliszekundumos eltérés lesz. Ennek oka az eltérő terjedési idő, ami sajnos a légköri zavarok miatt nem becsülhető pontosan.

A valós, fizikai idő kezelése

A valós fizikai idő mérésére a rendszer órája szolgál. Egy centralizált rendszerben ez egyetlen órát jelent, míg elosztott rendszerben a rendszer minden egyes számítási csomópontja tartalmazhat egy saját fizikai órát. Ekkor nagyon fontos, hogy biztosítsuk ezen fizikai órák együtt járását. Mivel az órák referencia jelét egy analóg berendezés (tipikusan egy kvarc oszcillátor) szolgáltatja, így a referencia jelek adott pontosságúak. A kvarc oszcillátorok tipikusan 10^{-6}–10^{-7} sec/sec pontosságúak. Ebből adódóan az egyes órák a rendszerben csúsznak egymáshoz képest, így azokat időről időre szinkronizálni kell, hogy a rendszerben adott pontossággal tudjunk egy ún. globális időt mérni.

Elosztott rendszerekben nagyon gyakran az az elvárás, hogy az egyes órák adott pontossággal együtt járjanak. Ezt a követelményt a **belső szinkronizációval** tudjuk biztosítani. Ekkor az órák az időről időre elvégzett szinkronizálás miatt együtt járnak, egymáshoz képest nem csúsznak, pontosabban a csúszás jól kontrollált. Ezzel szemben bizonyos esetekben az elosztott rendszer együtt működik más rendszerekkel, és elvárás, hogy az órák a valódi fizikai időt mérjék. Ilyenkor nem elég, hogy a rendszer órái egymáshoz képest nem csúsznak, a valós fizikai időhöz, a Koordinált Univerzális Időhöz képest sem csúszhatnak. Ezt a **külső szinkronizálással** lehet elérni, amikor időről időre a Koordinált Univerzális Időhöz szinkronizál a rendszer.

A továbbiakban vázlatosan áttekintjük az elosztott rendszerekben alkalmazott legismertebb óra rendszereket.

Elosztott rendszerekben alkalmazott óra rendszerek

Elosztott rendszerekben három típusú óra rendszert szokás alkalmazni a fizikai idő nyilvántartására: a pontos központi órával rendelkező rendszereket, a központilag felügyelt órákkal rendelkező rendszereket és a teljesen elosztott órás rendszereket. Az alábbiakban ezen rendszerek főbb tulajdonságait adjuk meg.

Pontos központi órával rendelkező rendszerek

A pontos központi órával rendelkező rendszerek az alábbi jellemző tulajdonságokkal rendelkeznek:

- Egy pontos óra szolgáltatja az időt az egész rendszernek. (A többi óra jelenlétéről nem veszünk tudomást, amíg a központi óra meg nem hibásodik.)
- A hibatűrés meleg tartalékként jelenik meg. Ha a központi pontos óra meghibásodik, egy másik óra veszi át a szerepét.
- A módszer pontos (az alkalmazott órától függően ns – ms), de drága.
- Speciális, a processzorba integrált hardvert igényel. Az óra ezt állítja, a többi folyamat ezt olvassa.
- Kommunikációs költség alacsony: 1 üzenet/szinkronizáció/hely, vagy broadcast.
- Példa: GPS, 4 szatelit, körülbelül ns-os pontosság.

Központilag felügyelt órákkal rendelkező rendszerek (master-slave)

A központilag felügyelt órákkal rendelkező rendszerek az alábbi jellemző tulajdonságokkal rendelkeznek:

- A pontosnak elfogadott master óra lekérdezi a slave-eket.
- Csúszásokat mér, korrekciót küld a slave-eknek.
- Ha a master meghibásodik, akkor valamilyen választási algoritmus alapján egy új mastert kell választani.
- Az átviteli időt becsülik.

Elosztott órás rendszerek

Az elosztott órás rendszerek az alábbi jellemző tulajdonságokkal rendelkeznek:

- Minden hely homogén, azonos algoritmust futtat.
- Minden hely a többi óra üzenete alapján frissíti a saját óráját (becsüli saját pontosságát).
- A hibatűrés protokoll alapú. A csomópontok észreveszik, ha valamelyik óra meghibásodott, és figyelmen kívül hagyják annak értékét.
- Nagy kommunikációs költség.

Az órák szinkronizálásánál nem szabad azzal a naiv feltételezéssel élni, hogy elég ismerni az egyes órák közötti eltérést, és akkor megoldható a szinkronizálás. Ez a gondolatmenet azért hibás, mert az egyes órák **csúsznak (drift)** egymáshoz képest. A kvarc oszcillátor alapú órák is csúsznak, vagyis az időt más „frekvenciával” mérik, mint a pontos referencia óra, így az általuk mutatott idő egyre jobban eltér attól. A csúszás lehet nagyon pici, de ez az idő múltával akkumulálódik. Például kvarc oszcillátoros órák esetén 10^{-6} sec/sec pontosságot feltételezve 1000000 másodperc, vagyis 11,6 nap alatt csúsznak 1 másodpercet.

Óracsúszás kompenzálása

Egy adott gép órája kétféleképpen csúszhat a valós, pontos órához képest: késhet, illetve siethet. Az első esetet viszonylag egyszerűen le lehet kezelni. Ha a valós időt T_{val}, a pontatlan lokális időt T_{lok} jelöli, és $T_{val} > T_{lok}$, akkor $T_{lok} := T_{val}$ állítással a problémát meg lehet oldani. Ez a módosítás teljesíti a természetes elvárást, hogy az idő értéke növekedjen. Sajnos ennél nehezebb problémával találjuk szemben magunkat, ha a lokális óra siet. Nem tehetjük meg, hogy egyszerűen visszaállítjuk, mert ekkor az a feltételezés, hogy az idő előre halad nem teljesülne. (Például a Unix *make* csak azokat az állományokat fordítja újra, amelyeknek a forrása frissebb, mint a hozzájuk tartozó object állomány. Ha megengednénk az óra visszaállítását, akkor előállhatna az az eset, hogy módosítunk egy forrásállományt, és a *make* mégsem fordítja újra, mert közben a szinkronizálás miatt az órát a rendszer visszaállította, és így az object állomány későbbinek tűnik.) Így azt kell elérni, hogy az óra lassabban járjon. A hardverórát általában nem lehet lassítani, így a szoftverórát kell állítani. Az alábbiakban ismertetünk egy óracsúszás kompenzációs algoritmust. Ehhez az alábbi jelöléseket vezetjük be:

S(t): szoftveróra,

H(t): hardveróra,

d(t): kompenzáció,

$S(t) = H(t) + d(t)$.

Amennyiben a módosítást folytonosnak szeretnénk, akkor *d(t)* legyen egy lineáris függvény:

$d(t) = aH(t) + b$.

Tegyük fel, hogy $S(t) = Tcsúszás$ amikor $H(t) = h$, és ekkor a pontos referenciaóra $T_{valódi}$-t mutat.

Ha azt akarjuk elérni, hogy *S N* további tikk (óraütés) után mutassa a pontos időt, akkor

$T_{csúszás} = (1+a)h + b$,

$T_{valódi}+N = (1+a)(h+N) + b$.

innen

$a = (T_{valódi} – T_{csúszás})/N$

és

$b = T_{csúszás} – (1+a)h$.

Tehát ezt a módszert alkalmazva elérhető, hogy a szoftverórát úgy járassuk lassabban, hogy az *N* tikk, vagyis óraütés múlva a pontos időt mutassa.

Óraszinkronizációs módszerek

Az alábbiakban vázlatosan áttekintünk három óraszinkronizációs módszert.

Cristian algoritmus

Cristian algoritmusa átmenet a pontos központi órával rendelkező és a központilag felügyelt órás rendszerek között. Az algoritmus feltételez egy pontos időszolgáltatót, amihez a kliensek időkéréseket juttatnak el, ha a pontos időre van szükségük. Amikor az időszolgáltató kap egy idő kérést, a nagyobb pontosság érdekében a válaszul összeállított üzenetbe a lehető legutolsó pillanatban helyezi bele az időt. Ezt az üzenetet visszaküldi a kliensnek. A kliens a megkapott üzenetből kiolvassa az időt és kompenzálja az átviteli idővel.

A nagyobb pontosság elérése érdekében az oda-vissza üzenet úthoz tartozó idő átlagával kompenzál. Az átviteli időt úgy modellezi, hogy $T_{átvitel} = min + x$, vagyis az átvitelt felbontja két komponensre: a min azt az időt jelöli, amennyi időt az átvitel akkor igényelne, ha csak az időt kérő folyamat futna és nem lenne más hálózati forgalom, illetve egy járulékos tagra, ami a többi folyamat és a többi hálózati forgalom hatását modellezi. Ezzel a modellel a szinkronizálás pontosságára a $\pm(T_{oda\text{-}vissza}/2-min)$ eredmény adódik.

A Berkeley algoritmus

A **Berkeley algoritmus** egy tipikus master-slave algoritmus. Itt a Cristian algoritmussal ellentétben a master folyamatosan lekérdezi a slave órákat és kiszámítja az egyes órák csúszásait. Továbbá nem a master által mutatott időt küldi el a slave-eknek, hanem a csúszást, ezáltal csökkentve az átviteli késleltetés okozta pontatlanságot. A nagyobb pontosság érdekében a kompenzációra felhasznált értékeket átlagolja, ezáltal a kommunikációs csatorna változó késleltetését és egyéb járulékos hibáit is csökkenti. Az algoritmus már hibatűrő tulajdonságokkal rendelkezik. Egyrészt a kirívó értékeket figyelmen kívül hagyja, másrészt amennyiben a master csomópont meghibásodik, egy (a későbbiek folyamán tárgyalt) választási algoritmussal új master csomópont kerül megválasztásra. Az algoritmussal elért pontosság durván 20–25 ms.

A Network Time Protocol

A **Network Time Protocol** egy időszolgáltató architektúrát és protokollt ír le, amit eltérő hálózatokból összekötött elosztott rendszerben az idő információ terjesztésére lehet használni. Ez az algoritmus képezi az Internetes óraszinkronizáció alapját. A megtervezésénél az alábbi főbb szempontokat vették figyelembe:

- kliensek pontosan szinkronizálhassanak a KUI-hez,
- megbízható szolgáltatás legyen, éljen túl hosszú idejű szétcsatolást,
- kellően gyakori szinkronizálást tegyen lehetővé,
- védjen időszolgáltatás hamisítás ellen.

Ennek megvalósítása érdekében egy hierarchikus struktúrát építettek ki. Az algoritmus terminológiájában a **stratum** a szinteket jelöli a hierarchiában. Az 1-es stratumon az ún. **elsődleges időszolgáltatók** helyezkednek el, amelyek valamilyen célberendezés segítségével közvetlenül a KUI-hoz szinkronizálnak. A 2-es stratumon **másodlagos időszolgáltatók** helyezkednek el, amelyek elsődleges időszolgáltatókhoz szinkronizálnak. A sémából jól látszik, hogy minél magasabb stratumon helyezkedik el egy időszolgáltató, annál távolabb van a KUI-től, így annál pontatlanabb az általa szolgáltatott idő, viszont annál kisebb a költsége. Nagyon sok

alkalmazás nem igényli, hogy az általa „látott” idő KUI pontosságú legyen, számos alkalmazásnál elegendő olyan óra pontosság, amit a hétköznapi életünkben használunk. A struktúra erősen hibatűrő és a „graceful degradation” (elegáns letörés) tulajdonságot mutatja.

A rendszerben három típusú szinkronizálás valósul meg:

1. *Multicast* (LAN). Kis késleltetést feltételez, pontatlan, de sok alkalmazás számára ez a pontosság is elégséges.
2. *Eljáráshívás.* Kliens–szerver-modellt alkalmaz, nagyobb pontosságot ér el az előbbi típusnál. Akkor alkalmazzák, ha nagyobb pontosságot kell elérni, vagy ha a multicast nem támogatott.
3. *Szimmetrikus mód. <ofszet, konfidenciaintervallum, késleltetés>* alakú üzeneteket használ. A gépek az utolsó nyolcat tárolják, a legkisebb késleltetésűt választják.

Míg 1991-ben megközelítőleg 20–30 elsődleges szerver és 2000 másodlagos szerver kapcsolódott az Internethez, addig ezek a számok mára nagyságrendekkel növekedtek. A Network Time Protocollal megközelítőleg 30 milliszekundumos pontosság érhető el.

Itt meg kell jegyezni, hogy ez az időszolgáltatás ún. **best-effort** szolgáltatás, nincs garantált időkorlát.

Logikai idő és logikai órák

Az eddigiekben tárgyaltuk a fizikai órákat és kezelésüket, elsősorban az elosztott rendszerek óráinak szinkronizálási kérdéseire koncentráltunk. A bevezetőben említettük, hogy az egyik legfontosabb időkezeléssel kapcsolatos feladat az esemény sorrendezés. Míg tetszőleges folyamatban a folyamat eseményeit egyértelműen sorba lehet rendezni a folyamat fizikai órája alapján, addig elosztott rendszerekben sajnos fizikai órát nem lehet események sorrendezésére használni, mert a folyamatok fizikai óráit nem lehet tökéletesen szinkronizálni. Ezért szükség van valamilyen módszerre, amely biztosítja a folyamatok eseményeinek sorrendezését elosztott környezetben. Már korábban is láttuk, hogy erre a feladatra a fizikai órák nem igazán alkalmasak. Ezt támasztja alá az is, hogy például 10 ms-os pontosságot figyelembe véve, ez a pontosság nem elégséges számos alkalmazás számára azok eseményeinek sorrendezésére. Egy 10 MIPS-es gép ennyi idő alatt 100 000 utasítást végez el, ami alatt számos, például szinkronizációs esemény következhet be. Ebből jól látszik, hogy ha a rendszerben az események az óra felbontásánál gyakrabban (nagyobb frekvenciával) következnek be, akkor azokat fizikai órával nem lehet sorrendezni, szükség van valamilyen, kifejezetten az események sorrendezését támogató absztrakcióra.

Mielőtt rátérnénk a módszer tárgyalására, definiáljuk a **korábban-történt (happened before)** relációt. Ennek értelmezéséhez tekintsük először azonban az alábbi két triviálisnak tűnő kijelentést:

1. Ha két esemény ugyanabban a folyamatban történt, akkor azok sorrendje megegyezik azzal a sorrenddel, ahogyan azt a folyamat látja.

2. Amikor egy folyamat üzenetet küld egy másik folyamatnak, akkor az üzenet elküldése megelőzi az üzenet megkapását.

Ezek után már definiálható a **korábban-történt** (KT) reláció, az alábbi jelölésekkel:

: *x* és *y* két esemény, mindkettő egyazon *P* folyamatban történt, és *x* megelőzte *y*-t.

: *x* korábban-történt relációban van *y*-nal (*x* és *y* nem feltétlenül egy folyamat eseményei)

KT1: Ha $ *P* folyamat, melyre , akkor *x* -> *y* adódik

KT2: ˙m üzenetre send(*m*) -> rcv(*m*), ahol

– send(*m*) az üzenet elküldésének,

– rcv(*m*) pedig az üzenet megkapásának eseménye.

KT3: Ha *x*, *y* és *z* olyan események, amelyekre

x -> *y* és *y* -> *z*, akkor *x* -> *z*

Ezek után már bevezethetjük a logikai órákat.

Logikai órák

A korábban történt rendezéshez támogatást kell nyújtani, hogy azt számszerűen meg lehessen ragadni. Ezt a szerepet hivatott betölteni a logikai óra. A logikai óra monoton növekvő szoftver számláló, amelynek az értéke nem kell, hogy bármilyen fizikai órához kapcsolódjon. Minden *P* folyamatnak van egy saját logikai órája, C_P, amelyet a folyamat események időcímkével való ellátására használ. A következőkben a *P* folyamatban az *a* esemény időcímkéjét $C_P(a)$ jelöli, míg egy tetszőleges folyamat *b* eseményét *C(b)*. A folyamatok az alábbi szabályok szerint (LSZ) állítják a logikai órájukat, illetve az üzenetekben az alábbi időcímkéket küldik:

LSZ_1: C_P minden egyes esemény *P*-beli bekövetkezte előtt inkrementálódik: $C_P := C_P + 1$

LSZ_2:

(a) Amikor egy *P* folyamat egy *m* üzenetet küld, akkor a folyamat az *m* üzenetet $t = C_P$ időcímkével látja el.

(b) Amikor egy *Q* folyamat megkap egy (*m*, t) üzenetet, kiszámolja $C_Q := max(C_Q, t)$-ot és aztán alkalmazza LSZ_1-et az *rcv(m)* esemény időcímkével történő ellátása előtt.

A fenti logikai órára teljesül az alábbi összefüggés:

a ® b Ţ C(a) < C(b)

Vagyis ha egy *a* esemény korábban történt egy *b* eseménynél, akkor az *a* eseményhez kisebb logikai óraérték tartozik, mint a *b* eseményhez. Könnyen belátható azonban, hogy a fenti összefüggés fordítottja nem igaz, vagyis *C(a) < C(b)*-ből nem következik, hogy *a* ® *b*. Ez az eset tipikusan konkurens eseményeknél áll fenn.

Teljesen rendezett logikai órák

Az előbbiekben tárgyalt logikai óra konstrukció csak részleges rendezést ad az eseményeken, hisz léteznek olyan különböző folyamatokhoz tartozó események, amelyekhez azonos logikai óraérték tartozik. Azonban viszonylag egyszerűen ki lehet terjeszteni a fenti sémát, hogy az teljes rendezést adjon. Ehhez a folyamatok azonosítóját is fel kell használni az időcímkék megkonstruálásához. Ha *a* a P_a folyamat egy eseménye, amely T_a helyi időcímkével rendelkezik, és *b* a P_b folyamat egy eseménye, amely T_b helyi időcímkével rendelkezik, akkor az ezen eseményekhez tartozó globális időcímkék: (T_a, P_a), illetve (T_b, P_b). Ezek segítségével a teljes rendezés: $(T_a, P_a) < (T_b, P_b)$ akkor és csak akkor, ha $T_a < T_b$ vagy $T_a = T_b$ és $P_a < P_b$.

Itt feltételeztük, hogy a folyamatokra létezik egy teljes rendezés, ami a gyakorlatban mindig fennáll.

4.4.4.2. Elosztott koordináció

Elosztott folyamatoknak is koordinálni kell a tevékenységüket, különös tekintettel az osztottan kezelt erőforrásokra. A Sun NFS egy állapotmentes szerver, így ott állományok megosztott kezelésére nem lehet a szerveren zárakat használni, hisz ez ellentmondana az állapotmentességnek. Így ebben az esetben a Unix egy külön daemon – a *lockd* segítségével oldja meg a szinkronizálást. Azonban gyakran célszerű az erőforrást kezelő szerverrel szervesen egybeépíteni a szinkronizálási eszközöket.

A leggyakrabban előforduló szinkronizálás osztottan használt erőforrások esetén a kölcsönös kizárás. Elosztott rendszerekben az elosztottság a korábban megismert megvalósításokkal szemben újabb igényeket támaszt, amelyeket elosztott kölcsönös kizárási algoritmusokban figyelembe kell venni. A továbbiakban ezekre nézünk meg néhány példát.

Elosztott kölcsönös kizárás

Egy osztottan használt erőforrással kapcsolatosan a kölcsönös kizárással (KK) szemben az alábbi követelményeket támasztjuk:

KK1(biztonság): Egy időpillanatban egyszerre csak egyetlen folyamat tartóz-

kodhat a kritikus szakaszban.

KK2(haladás): Egy folyamat, amely be akar lépni a kritikus szakaszba, véges

időn belül beléphet. Ez a kritérium a rendszer holtpont

mentességét és kiéheztetés mentességét fogalmazza meg.

KK3(rendezés): A kritikus szakaszba a folyamatok a *korábban-történt* ren-

dezés alapján lépnek be, vagyis ha valaki korábban kérte

a belépés jogát, az korábban is kapja meg.

Központi szerver algoritmus

Az elosztott kölcsönös kizárás legegyszerűbb megvalósítása a **központi szerver algoritmus**. Itt minden egyes osztott erőforráshoz hozzárendelünk egy szervert, amelyre a kölcsönös kizárást meg kell valósítani. Ez a szerver engedélyezi a belépést a kritikus szakaszba. Minden alkalommal csak egy folyamatot enged be. Ha már tartózkodik egy folyamat a kritikus szakaszban, akkor a további kéréseket várakoztatja. Ezt a modellt úgy is elképzelhetjük, hogy a szerver birtokol egy tokent az általa felügyelt kritikus szakaszhoz. A szakaszba mindig csak az a folyamat léphet be, amelyik birtokolja a tokent. Mivel kezdetben egy token volt a rendszerben, így a kölcsönös kizárás biztosított. A szerver semmi mást nem tesz, mint amennyiben egy folyamat be akar lépni a kritikus szakaszba és a token a szervernél van (vagyis a kritikus szakaszban nem tartózkodik egyetlen folyamat sem), akkor a kérő folyamatnak odaadja a tokent, ellenkező esetben várakozásra kényszeríti. A megoldás problémája, hogy a szerver kritikus hibapont, annak meghibásodását megfelelően kezelni kell.

Logikai órákat alkalmazó elosztott algoritmus

Egy másik megközelítés a **logikai órákat alkalmazó elosztott algoritmus**. Logikailag ez az algoritmus is a belépést engedélyező token meglétéhez köti a kritikus szakaszba történő belépést, itt azonban a token a rendszerben lévő többi folyamattól kapott megfelelő üzenetek együtteséből áll elő. Az algoritmus lényege, hogy ha egy folyamat be szeretne lépni a kritikus szakaszba, akkor az összes többi folyamatnak elküld egy üzenetet, ami ezen szándékát jelöli, és az üzenetet ellátja a szándék időpontját jelző időbélyeggel. A folyamat ekkor bevárja, hogy a rendszer összes folyamata válaszoljon. Amikor az összes válasz megérkezett, a folyamat beléphet a kritikus szakaszba. Ameny-nyiben egyszerre több folyamat is be akar lépni a kritikus szakaszba, akkor több belépési szándékot jelölő üzenet is van a rendszerben. Ekkor, ha egy folyamat be akar lépni a kritikus szakaszba, és egy másik folyamattól egy hasonló jellegű üzenetet kap, akkor az üzenet időbélyegét összehasonlítja a saját üzenete időbélyegével. Amennyiben a saját időbélyege kisebb, akkor nem válaszol a kérésre, hanem egy várakozási sorban várakoztatja, és majd csak a kritikus szakaszt elhagyva válaszol neki. Amennyiben a saját időbélyege nagyobb, akkor azonnal válaszol a kérésre (ekkor majd a másik folyamat fogja őt várakoztatni). Jól látszik, hogy a sorrendezési kritérium is teljesül az algoritmusra.

A gyűrű alapú algoritmus

A **gyűrű alapú algoritmus** egy logikai gyűrűt épít a folyamatokból, amelyben egy token kering. Amelyik folyamat be akar lépni a kritikus szakaszba, az megvárja a token megérkezését, és belép, majd a kritikus szakasz elhagyása után küldi tovább a tokent.

Ezek az algoritmusok azt sugallják, hogy azok az osztottan használt erőforrás menedzserétől függetlenül futtathatók. Ez valóban így van, azonban egy praktikus implementációban célszerű a kölcsönös kizárást az osztott erőforráshoz a lehető legközelebb megvalósítani, vagyis célszerű ezt a feladatot is az erőforrás menedzserre ruházni.

A továbbiakban a választási algoritmusokat tárgyaljuk.

Választási algoritmusok

Elosztott rendszerekben a választási algoritmusok nagy szerepet kapnak. Feladatuk egy folyamatcsoportból egy kitüntetett folyamat kiválasztása. Ezen folyamat több szempontból is lehet kitüntetett: például kölcsönös kizárást biztosító szerver, koordinátor, token generátor, időszolgáltató stb.

A választási algoritmus eredményeként kiválasztott folyamattal szemben a fő követelmény, hogy a kiválasztott folyamat **egyedi** legyen, a csoport minden folyamata ugyanazt a folyamatot gondolja megválasztottnak, még akkor is, ha egyszerre több folyamat is elindítja a választást.

A továbbiakban két algoritmust vizsgálunk meg: a **bully algoritmust**, amely esetén a résztvevő folyamatok ismerik egymás azonosítóit, azok prioritását, illetve a **gyűrű algoritmust**, amelynél csak az egyik szomszéd kommunikációs azonosítójára van szükség.

A bully (erőszakos) algoritmus

Az algoritmus akkor alkalmazható, ha a folyamatcsoport tagjai ismerik egymás azonosítóit (illetve a hozzájuk rendelt prioritásokat – gyakran a prioritás megegyezik az azonosítóval), és hálózati címeit. Az algoritmus a folyamatcsoportból kiválasztja a legnagyobb prioritású még aktív folyamatot, és megválasztja koordinátornak. Az algoritmus működése során feltételezzük, hogy a kommunikáció megbízható, de a folyamatok maga a választási algoritmus alatt is meghibásodhatnak, továbbá a tárgyalás leegyszerűsítése érdekében a folyamat azonosítója egyben a prioritása is, és a nagyobb azonosító nagyobb prioritást jelöl. Az algoritmusban három eltérő típusú üzenet jelenhet meg:

- *választás:* – a választás megkezdését jelzi,
- *válasz:* – választási üzenetre adott válasz,
- *koordinátor:* – az új, megválasztott koordinátor szétküldi az azonosítóját.

Az algoritmus mindig azzal kezdődik, hogy egy folyamat észreveszi a koordinátor meghibásodását. Legyen ez P_i. Ekkor:

- P_i *választás*(P_i) üzenetet küld minden egyes P_j folyamatnak, amelyre $j > i$ (az indexek prioritási sorrendet jelölnek). Ezután P_i várakozik *válasz*(P_j)-re T_{vv} ideig. Ha nem érkezik válasz, akkor P_i magát tekinti a legnagyobb prioritású, még működő folyamatnak (hisz a magasabb prioritásúaktól nem kapott választ), és az alacsonyabb prioritású folyamatoknak kiküld egy *koordinátor*(P_i) üzenetet, vagyis, megválasztja magát koordinátornak.
- Ha érkezett valamilyen P_j folyamattól válasz, amelyre $j > i$, akkor P_i vár T_{vk} ideig, hogy a magasabb prioritású folyamat koordinátornak deklarálja magát. Amennyiben T_{vk} idő alatt nem jön ilyen üzenet, újabb választást kezdeményez.
- Ha egy folyamat egy *koordinátor*(P_j) üzenetet kap, akkor feljegyzi az üzenetben szereplő folyamat azonosítóját, és ettől kezdve ezt a folyamatot koordinátorként kezeli.

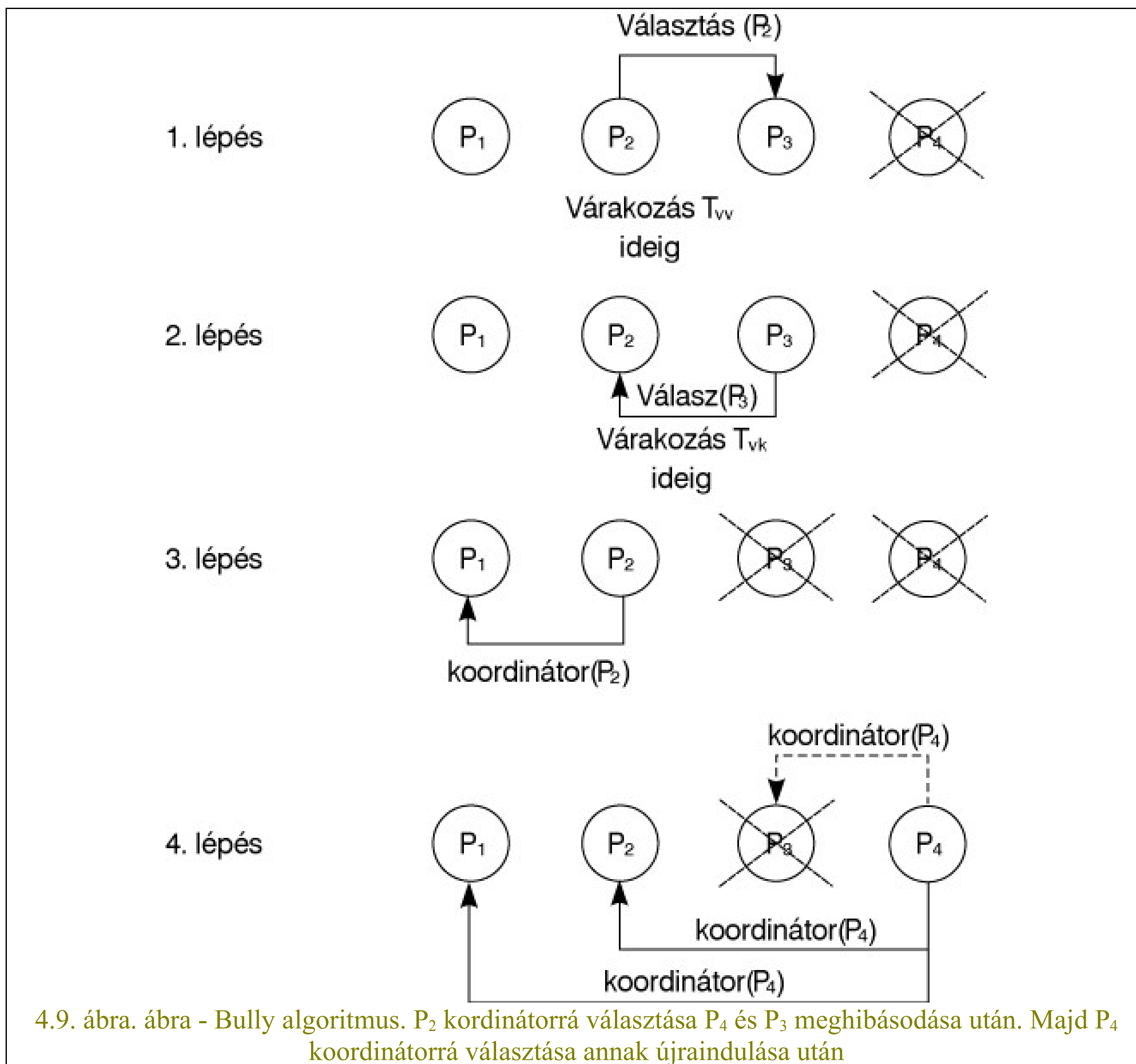
- Ha egy P_j folyamat *választás(P_i)* üzenetet kap, akkor visszaküld egy *válasz(P_j)* üzenetet, és P_j is kezdeményez egy választást, hacsak már nem kezdeményezett korábban.

Mint azt korábban említettük, az algoritmus kezeli a választási algoritmus futtatása során történő folyamat meghibásodásokat is. Erre szolgál a T_{vk} várakozási idő szerepeltetése az algoritmusban. T_{vk} a koordinátor megválasztási üzenet megérkezésére történő várakozás időkorlátját jelöli. Mint láttuk, ha egy folyamat észleli a koordinátor meghibásodását, választási üzenetet küld minden nála nagyobb prioritású folyamatnak (mivel ezek lehetnek a potenciális következő koordinátorok). Ha ezek közül valaki válaszol, az azt jelenti, hogy működőképes és ő akar lenni a koordinátor. Ehhez egy adott T_{vk} időkorláton belül egy üzenettel koordinátornak kell nyilvánítania magát. Ha az üzenet mégsem érkezik meg T_{vk} időn belül, ez azt jelenti, hogy a folyamat a választási algoritmus befejeződése előtt meghibásodott. Ekkor újabb választás indul.

Amikor egy meghibásodott folyamat újra indul, azonnal választást kezdeményez. Amennyiben ennek a folyamatnak a legnagyobb prioritású az azonosítója, akkor úgy dönt, hogy ő lesz a koordinátor, még akkor is, ha a korábbi koordinátor működőképes. Ezért kapta az algoritmus a nevét, vagyis, hogy erőszakos, hisz minden körülmények között a legmagasabb prioritású működő folyamat lesz a koordinátor.

Az algoritmus működésének bemutatására tekintsük az alábbi egyszerű példát (4.9. ábra):

Négy folyamat alkotja a folyamatcsoportot, ezek P_1, P_2, P_3 és P_4. P_2 észreveszi a P_4-es koordinátor folyamat meghibásodását, és egy választást kezdeményez. (a 4.9. ábra 1. lépése). A választás üzenet elküldése után P_2 T_{vv} ideig vár a válaszra, ami azt jelzi, hogy P_3 él, így ő lehet a koordinátor. A 2. lépésben P_3 visszaküldi a választ P_2-nek. Ekkor P_2 T_{vk} ideig ismét elkezd várakozni, hogy P_3 bejelentse, hogy ő lett a koordinátor. Ekkor P_3 meghibásodik (3. lépés). Mivel P_2 az időkorláton belül nem kapott koordinátori üzenetet, magát tekinti megválasztott koordinátornak, így egy *koordinátor(P_2)* üzenetet küld P_1-nek. Egy idő után a P_4-es folyamat újra indul, és azonnal értesít mindenkit, hogy ő a koordinátor.



4.9. ábra. ábra - Bully algoritmus. P_2 kordinátorrá választása P_4 és P_3 meghibásodása után. Majd P_4 koordinátorrá választása annak újraindulása után

Az algoritmus során küldött üzenetek száma. A legkedvezőbb esetben a második legnagyobb prioritású folyamat veszi észre a koordinátor meghibásodását és kezdeményez választást. Ekkor rögtön megválaszthatja magát, és szétküld *(n-2)* koordinátor üzenetet az alacsonyabb prioritású folyamatoknak. A legrosszabb esetben a Bully algoritmus $O(n^2)$ üzenetet igényel. Ebben az esetben a legkisebb prioritású folyamat észleli a koordinátor meghibásodását, és mind az *(n-1)* folyamat választást kezdeményez, választás üzeneteket küldve a magasabb prioritású folyamatoknak.

A gyűrű algoritmus

A folyamatcsoportban lévő folyamatokat **egyirányú logikai gyűrűbe** kell szervezni. *Ez nem jelent megkötést a fizikai topológiára*. Ebben az esetben feltételezzük, hogy az egyes folyamatok nem ismerik egymás azonosítóit, így prioritásukat sem, csak azt tudják, hogyan kell kommunikálni az egyik, mondjuk az óra járásával

megegyező szomszédjukkal. Az algoritmus célja a **legnagyobb prioritású egyedi koordinátor** megválasztása. *Az algoritmus feltételezi, hogy a folyamatok az algoritmus alatt nem hibásodnak meg, és elérhetők maradnak.*

Kezdetben minden folyamat a választásban **nem résztvevő**nek van jelölve. Bármelyik folyamat kezdeményezhet választást. Ekkor első lépésben választásban **résztvevő**nek jelöli magát, és az azonosítóját egy **választás** üzenetben elküldi a szomszédjának.

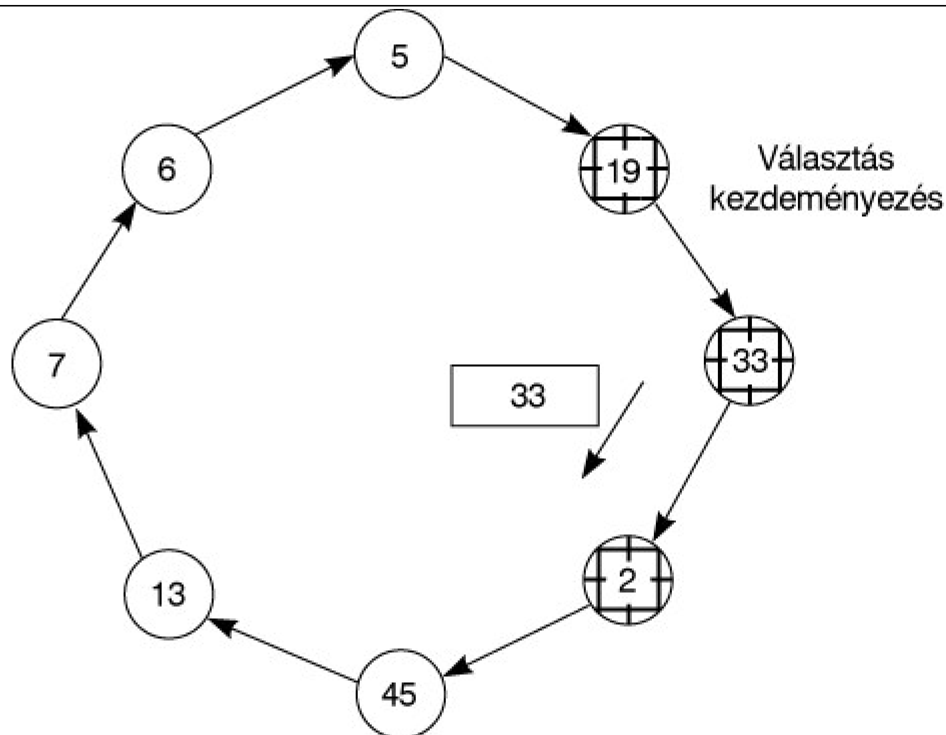
Amikor egy folyamat egy **választás** üzenetet kap, összehasonlítja az üzenetben foglalt azonosítót a sajátjával. Amennyiben a beérkezett azonosító nagyobb, a folyamat továbbítja az üzenetet a szomszédjának. Azonban, ha a beérkezett azonosító kisebb a saját azonosítójánál, akkor a folyamat először megvizsgálja, hogy **résztvevő**-e. Amennyiben nem, akkor az üzenetben lecseréli az azonosítót a sajátjára, és **résztvevő**nek jelöli magát, majd továbbítja a **választás** üzenetet a szomszédjának. A folyamat akkor is **résztvevő**-nek jelöli magát, ha nem cserélte le az azonosítót.

Amennyiben az azonosító megegyezik az üzenetet kapó folyamat saját azonosítójával, akkor a kör bezárult, a folyamat a legnagyobb prioritású folyamat a körben, ő lesz a koordinátor. A folyamat **nem résztvevő**nek jelöli magát, és egy **megválasztott** üzenetet küld a szomszédjának a saját azonosítójával, így tudatva a koordinátor kilétét.

Ha egy nem koordinátor folyamat **megválasztott** üzenetet kap, magát **nem résztvevő**nek jelöli, megjegyzi a koordinátor azonosítóját és továbbítja az üzenetet a szomszédjának.

Folyamatok **résztvevő**nek, illetve **nem résztvevő**nek jelölésének az értelme akkor domborodik ki, amikor egyszerre több folyamat kezdeményez választást. Ekkor a fenti jelöléseket ki lehet használni, és minimalizálni lehet az üzeneteket.

Az algoritmus során küldött üzenetek száma. Ha csak egy folyamat kezdeményezi a választást, akkor a legrosszabb eset olyankor áll fenn, amikor a kezdeményező óra járásával ellentétes szomszédja birtokolja a legmagasabb azonosítót.



4.10. ábra. ábra - A gyűrű választási algoritmus működése. A választást most a 19-es csomópont kezdeményezte

Ekkor ezen szomszéd eléréséhez *n*-1 üzenet szükséges, amely addig nem nyilvánítja magát koordinátornak, amíg a saját üzenetét vissza nem kapja újabb *n* üzenettel később. Ekkor a **megválasztott** üzenet is *n*-szer megjelenik, így összesen 3*n*-1 üzenetre van szükség. A 4.10. ábra egy gyűrűs választási algoritmust mutat. A választást a 19-es folyamat kezdeményezte. A **választás** üzenet jelenleg a 33-as azonosítót tartalmazza, de a 45-ös folyamat ezt majd lecseréli a saját azonosítójára, ha majd megkapja az üzenetet. A jelölt körök a **résztvevő** folyamatokat jelölik.

4.4.5. Elosztott rendszerek biztonsági kérdései

Az elosztott rendszerek fokozottan ki vannak téve a különféle biztonsági veszélyeztetéseknek. Bár a biztonság megteremtéséhez szükséges eszközök többsége adott, alkalmazásuk azonban alapos tervezést és széles körű vizsgálatokat igényel.

Elosztott rendszerek biztonságát különböző **biztonsági módszerek** alkalmazásával érthetjük el. Ezen módszerek azonban nem alkalmazhatóak önállóan, vagy folyamatosan bővíthetően – **a rendszer** egyidejű, a kívánalmaknak megfelelően **teljes körű védelmére** van szükség. Egy elosztott, nyílt rendszerben a teljes körű biztonság megteremtése egyetlen rendszerrel nem valósítható meg. Helyi, egyes gépeket, szolgáltatásokat védő rendszerek telepítése könnyebb feladat, ezek azonban egy elosztott rendszerben kezelhetetlenül bonyolulttá tehetik a rendszert. A felhasználók által megjegyzendő jelszavak számának minimális szinten tartása megköveteli, hogy egy elosztott rendszerben egyetlen azonosítás elegendő legyen a teljes rendszer használatára.

Nem elegendő a módszerek teljes körű alkalmazása sem: a biztonság megteremtésére és megőrzésére ún. **biztonságpolitikára** is szükség van. Nem elegendő a módszerek bevezetése, azok alkalmazását is pontosan szabályozni kell. Hiába alkalmazzuk a lehető legkifinomultabb, legbiztonságosabb azonosító rendszert, ha a felhasználók az íróasztalukon kitűzve tárolják jelszavaikat. A biztonságpolitikának a módszerek helyes alkalmazásán kívül hangsúlyt kell fektetnie a felhasználók oktatására és ellenőrzésére is.

4.4.5.1. Mi a biztonság?

Egy rendszer biztonsága alatt különböző emberek különböző dolgokat értenek. A számítógépeket használók többsége a számítógépes vírusokra és az Internet-kalózok támadásaira gondol, ha a számítógépek biztonságáról van szó. Mások a bizalmas adatok védelmét, titkosságát és a hozzáférés szabályozását értik e fogalom alatt. Egy rendszer biztonságossá tétele mindezen problémák kezelését megköveteli. Általában a következő területeket értjük alatta:

- Azonosítás (authentication). Felhasználók és számítógépek megbízható és egyértelmű azonosítása, üzenetek biztonságos aláírása és bizonyos események, akciók egyértelmű rögzítése.
- Felhatalmazás (authorization). A felhasználók, számítógépek és programok által végrehajtható akciók nyilvántartása.
- Titkosítás (encryption). Hozzáférés szabályozása bizalmas információkhoz.
- Rendszer védelem (system protection). A rendszerek megbízható és folyamatos működésének biztosítása (vírus védelem, szolgáltatások folyamatos fenntartása, véletlen rendszerhibák elkerülése).

4.4.5.2. Kik a támadók és mik a fenyegetések?

Elosztott rendszerekben tárolt információ megszerzésére, illetve a rendszerek működésének megzavarására emberek és programok (folyamatok) törekedhetnek. A továbbiakban a **támadó** kifejezést alkalmazzuk az elosztott rendszerben tárolt információt vagy erőforrást illetéktelenül megszerezni próbáló ügynökre. A támadó a rendszer valamilyen biztonsági hibáját használja ki a **támadás** véghezvitele során. A biztonsági hiba lehet a **biztonsági módszerek** vagy a **biztonságpolitika** hibája.

A következő főbb fenyegetés fajták léteznek:

- jogosulatlan megbízók információszerzése,
- információ (beleértve programokat is) jogosulatlan módosítása,
- az erőforrások jogosultság nélküli használata,
- zavarkeltés a rendszer megfelelő működésében anélkül, hogy a támadónak ebből haszna származna.

4.4.5.3. A támadás módszerei

A fenyegetések valóra váltásához hozzá kell férni a rendszerhez. Ennek legelterjedtebb módjai:

- Lehallgatás. Az elosztott rendszer komponensei közötti üzenetváltásokról jogosultság nélküli információ szerzése. Ez történhet úgy is, hogy a támadó közvetlenül a hálózatról szerzi be az üzeneteket, vagy nem megfelelően védett tárolt információt olvas el.
- Maszkírozás. Egy felhasználó vagy program rendszerbeli azonosítóját használva a támadó üzenetet küld, illetve fogad a megfelelő jogosultságok nélkül.
- Üzenetmódosítás. A támadó üzeneteket fog el és megváltoztatja a tartalmukat, majd a megváltoztatott üzenetet továbbküldi a rendszer megtévesztése céljából.
- Visszajátszás. A támadó a rendszer egy üzenetét eltárolja, és egy későbbi időpontban újra lejátssza. A visszajátszás ellen nem lehet egyszerű titkosítással védekezni, mert a visszajátszást használni lehet erőforrás lopásra és vandalizmusra is, még akkor is, ha a támadó nem érti az üzenetet.

A fenti támadások véghezviteléhez a támadónak hozzá kell férnie a rendszerhez, hogy futtathassa a programot, amely megvalósítja a támadást. A legtöbb támadásért a rendszer jogosult felhasználói a felelősek nem megfelelő jelszóhasználati gyakorlatukkal. A legegyszerűbb beszivárgási módszer a jelszó feltörés, vagyis a szisztematikus próbálgatás. Léteznek szótárak, amelyek a leggyakrabban használt jelszavakat tartalmazzák. Ez ellen megbízható jelszó választásával lehet védekezni.

4.4.5.4. Az elosztott biztonsági rendszer tervezése

Elosztott rendszerekben a fenyegetettség fő forrása a kommunikációs csatornák nyitottsága. Fel kell tételezni, hogy a rendszer minden szintjén lévő kommunikációs csatorna és program ki van téve a fenti fenyegetéseknek. A lehetséges betolakodókat nem könnyű azonosítani, így olyan világnézetet kell alkalmazni, amely nem feltételez bizalmat. De ahhoz, hogy használható rendszert építhessünk, néhány megbízható komponensből kell kiindulni. Egy hatékony tervezési megközelítés lehet annak feltételezése, hogy minden kommunikáció megbízhatatlan forrásból jön, amíg az ennek ellenkezőjét nem bizonyítja. A kommunikáló felek megbízhatóságát minden egyes kommunikációs csatornahasználatkor bizonyítani kell.

A biztonság megvalósítására használt mechanizmusokat nagy gonddal kell ellenőrizni, például egy biztonságos kommunikációs protokollnak és az azt megvalósító programnak bizonyíthatóan helyesnek kell lenni az összes lehetséges üzenet szekvenciára. A bizonyításnak ideális esetben formális alapokon kell nyugodnia.

A biztonsági rendszer tervezése során elsősorban nem a biztonsági módszerek „erőssége” befolyásolja a teljes rendszer biztonságát, hanem alkalmazásuk körülményei és teljessége. Biztonsági rendszerek véleményezése során a felhasználók gyakran esnek a technikai részletek elsődleges minősítésének hibájába. Sokkal jobban figyelnek a titkosításban alkalmazott kulcsok hosszára, mintsem alkalmazásuk helyességére és teljességére. A

biztonsági rendszerek tervezésének **első aranyszabálya** szerint **tökéletes biztonság nem létezik**. A biztonság mindig a védendő rendszer értékének és a biztonsági rendszer kiépítési költségének kompromisszumán alapszik. Drágább és jobb technológia alkalmazása javíthatja a rendszer biztonságát, azonban mindig létezni fog sikeres (és egyre költségesebb) támadási technika a rendszer ellen. A rendszer védelmi mechanizmusait úgy kell megválasztani, hogy a támadó által elérhető nyereség (vagy okozott kár) mindig lényegesen kevesebb legyen a támadás költségénél.

A rendszer tervezésének **második aranyszabálya** szerint **a legtöbb biztonsági problémát mindig az emberek viselkedése okozza**. Semmilyen biztonsági mechanizmus nem lehet sikeres, ha használói nem viselkednek biztonságos módon. Ezért a rendszer tervezése során olyan megoldásokat és módszereket kell alkalmazni, melyek ösztönzik a felhasználók biztonságos viselkedését.

Az összes biztonsági technika három alapvető mechanizmuson alapszik: titkosításon, azonosítási mechanizmusokon, és hozzáférés szabályozáson.

4.4.5.5. Titkosítás

Biztonságos rendszerek megvalósításában a titkosítási módszerek központi szerepet játszanak. Amikor valamilyen információt titkosítunk, akkor egy olyan transzformációt végzünk rajta, ami az inverz transzformáció ismerete nélkül lehetetlenné (pontosabban praktikusan lehetetlenné, vagy a titkosított adatok értékéhez mérten lényegesen költségesebbé) teszi az információ visszafejtését. A gyakorlatban használt titkosítási algoritmusok két nagy csoportba oszthatók: titkos kulcsú és nyilvános kulcsú titkosítási algoritmusok.

A titkos kulcsú titkosítás során a sima szöveget egy előre megegyezett függvénnyel titkosítják, egy titkos kulcsot használva. A megfejtés során az inverz függvényt kell alkalmazni ugyanazon titkos kulccsal. Így, ha biztosítjuk, hogy a titkos kulcsot csak a kommunikációban résztvevő két folyamat birtokolja, akkor az információ titkosságát biztosítottuk. Mivel a kulcsokat titokban, mások elől elrejtve tartjuk, így a függvényt, illetve az inverz függvényt nem kell titokban tartani.

Általánosságban a titkosító függvény és annak inverze lehet különböző, de választható olyan függvény is, amelynek az inverze megegyezik magával a függvénnyel. A gyakorlatban ilyen függvényeket alkalmaznak. A titkosító algoritmusnak biztonságosnak kell lennie szisztematikus feltörési kísérletekkel szemben is. A leggyakoribb ilyen próbálkozás, hogy a rendelkezésre álló sima szöveg, és a hozzá tartozó titkosított szöveg alapján megpróbálják kitalálni a kulcsot, illetve titkosított szövegekre alkalmazzák az inverz függvényt véletlenül megválasztott kulccsal, és ha az eredmény értelmesnek tűnik, akkor a kulcs helyes.

A nyilvános kulcsú titkosítás egy olyan titkosító módszer, amelyben a kommunikáló feleknek nem kell bízni egymásban. A kommunikációban minden egyes résztvevő létrehoz egy kulcspárt, egy kódoló és egy dekódoló kulcsot. A dekódoló kulcsot titokban tartja (titkos kulcs), a kódoló kulcsot nyilvánosságra hozza (nyilvános kulcs). Mindenki, aki titkos üzeneteket szeretne küldeni neki, a kódoló kulcs segítségével titkosítja az üzenetét, melyet csak a dekódoló kulccsal rendelkező fél tud visszafejteni. A módszer a két kulcs között kapcsolatot

teremtő olyan függvényen alapul, amelyre a kódoló kulcs ismeretében nagyon nehéz meghatározni a dekódoló kulcsot. A függvény olyan $Y = f(X)$ függvény, amelyre X kiszámítása Y ismeretében annyira számításigényes, hogy gyakorlatilag kivitelezhetetlen. Ez a nyilvános kulcsú titkosítás, amely során nincs szükség titkos kulcsok küldözgetésére a kommunikációs csatornán. A módszer két nagyon nagy prímszám szorzatának használatán alapul, kihasználva azt a tényt, hogy ilyen nagy számok prímfelbontásának meghatározása olyan számításigényes, hogy gyakorlatilag kivitelezhetetlen. (Megjegyzendő, hogy matematikailag nem bizonyított, hogy nem létezik gyors felbontási algoritmus, de praktikusan elfogadott tény.)

A titkosítás alkalmazható bizalmas információk elrejtésére, amikor csak az fejtheti meg az üzenetet, aki birtokolja a megfejtéséhez szükséges kulcsot. A módszer kommunikáció hitelesítésénél segédeszközként is használatos. Ha a vevő sikeresen dekódol egy üzenetet, feltételezheti, hogy az hiteles forrásból származik, mert a küldő ismerte a kulcsot. Az azonosítás finomabb formájának, a digitális aláírás létrehozásában is alkalmazható. Ennek feladata azt ellenőrizni, hogy az üzenet valóban sértetlenül és a feltételezetett feladótól érkezett-e meg, valamint annak biztosítása, hogy a feladó a későbbiekben ne tagadhassa le az üzenet elküldését.

4.4.5.6. Hozzáférés-szabályozás

A hozzáférés-szabályozás feladata az adatokhoz és erőforrásokhoz való hozzájutás szabályozása és korlátozása az azonosított felhasználók és programok körére. A hozzáférés-szabályozás az erőforrások használóinak azonosítása után az erőforrás használati szabályok (például időpont – erőforrás – felhasználó mátrix) alapján engedélyezi vagy tiltja meg a hozzáférést.

4.4.5.7. Azonosítás

Az azonosítás az elosztott rendszer résztvevőinek egyértelmű és megbízható hitelesítését jelenti. Centralizált többfelhasználós rendszerekben viszonylag egyszerűen meg lehet oldani az azonosítást: a munkamenet elején valamilyen jelszó ellenőrzéssel hitelesíteni lehet a felhasználót (vagy programot), és az azonosítás után a munkamenetet egy olyan tartománynak lehet tekinteni, amelyben az összes műveletet az azonosított felhasználó nevében lehet elvégezni. Ez a megközelítés feltételezi a rendszer erőforrásainak centralizált kezelését, ami nehezen oldható meg, vagy általában nem kívánatos elosztott rendszerekben.

Elosztott rendszerekben a hitelesítést általában egy kitüntetett szolgáltatás – az azonosító szolgáltatás – végzi, amely titkosítási eszközöket használ a feladat megoldására. A rendszer elosztott szolgáltatásai az azonosító szolgáltatást használhatják fel a felhasználók (és programjaik) azonosítására.

4.4.5.8. Azonosítás és kulcs szétosztás

Titkos kulcsú rendszerekben régebben a kulcsokat valamilyen más hordozón osztották szét. Például papíron oda lehet adni a kulcsot, majd később megsemmisíteni. Ez a módszer elosztott számítógépes hálózatokban nehezen járható megoldás, általában valamilyen más titkosítási mechanizmust alkalmaznak a titkos kulcsok átvitelére. A

titkos kulcsok tárolására és elosztására speciális kulcsszervereket lehet használni, de a hitelesítésre nagy gondot kell fordítani.

Nyilvános kulcsú titkosítás esetén a nyilvános kulcs megkapóinak biztosnak kell lenniük, hogy a kapott kulcs hiteles, vagyis annak a kulcspárnak a nyilvános része, amelynek titkos kulcsát a kommunikációs partner birtokolja. Ellenkező esetben egy ál partner bizalmas információkhoz juthat, ha hamis nyilvános kulcsot ad. A hitelesség biztosítására elektronikus aláírásokat lehet használni.

A hitelesítés és a kulcsok biztonságos szétosztását Needham és Schroeder **hitelesítő szerveren** alapuló hitelesítési és kulcs szétosztási algoritmusán keresztül mutatjuk be. A hitelesítő szerver feladata, hogy folyamat pároknak biztonságos módon kulcsot biztosítson a kommunikációjukhoz. Ehhez a klienseivel a szerver titkosított üzenetekkel kommunikál. A továbbiakban az algoritmus titkos kulcsokon alapuló változatát tekintjük át.

A következőkben a Needham–Schroeder titkos kulcsú hitelesítési protokoll egyes lépéseit foglaljuk össze. A példában két állomás (A és B) a hitelesítési szerver (S) segítségével vált üzenetet. Az üzenetváltásban felhasználják kettőjük K titkos és nyilvános kulcsait.

Fejléc	Üzenet	Megjegyzés
1. $A \bullet S$:	A, B, N_A	*A* *S*-től kér egy kulcsot a *B*-vel történő kommunikációhoz.
2. $S \bullet A$:	$\{N_{A,B}, K_{AB}, \{K_{AB}, A\}_{K_B}\}_{K_A}$	*S* visszaküld egy üzenetet, amelyet *A* titkos kulcsával titkosít. Az üzenet tartalmazza a frissen generált K_{AB} kulcsot, és egy *B* titkos kulcsával titkosított jegyet. Az N_A címke azt mutatja, hogy a jelen üzenet válasz a megelőzőre. *A* azt hiszi, hogy az üzenetet *S* küldte, mert csak *S* ismeri *A* titkos kulcsát.
3. $A \bullet B$:	$\{K_{AB}, A\}_{K_B}$	*A* elküldi a jegyet *B*-nek.
4. $B \bullet A$:	$\{N_B\}_{K_{AB}}$	*B* visszafejti a jegyet, és az új K_{AB} kulccsal titkosítja az N_B címkét.
5. $A \bullet B$:	$\{N_B - 1\}_{K_{AB}}$	*A* azzal mutatja meg *B*-nek, hogy ő küldte az előző üzenetet, hogy egy megegyezett transzformációt végez N_B-n.

Jelölések:

A	A kommunikációt kezdeményező megbízó neve.
B	A kommunikációs partner megbízójának neve.
K_A	*A* titkos kulcsa.
K_B	*B* titkos kulcsa.
K_{AB}	*A* és *B* közötti kommunikáció titkos kulcsa.
N_A	*A* által generált címke.
$\{M\}_K$	*K* kulccsal titkosított *M* üzenet.

A protokoll egyik gyenge pontja, hogy *B*-nek nincs információja arról, hogy a 3. üzenet friss. Egy behatoló megszerezheti a K_{AB} kulcsot és a $\{K_{AB}, A\}K_B$ hitelesítőt, és akkor *A*-nak adhatja ki magát. Ez a probléma időcímke

használatával küszöbölhető ki. Az algoritmusnak létezik nyilvános kulcson alapuló változata is, de helyszűke miatt itt azzal nem foglalkozunk.

4.4.5.9. Kerberos: hitelesítési protokoll nyílt hálózati rendszerekre

A Kerberos egy titkos kulcsú titkosításon (DES) alapuló hitelesítési rendszer, melyet az MIT (Massachusetts Institute of Technology) fejlesztett ki az Athena projekt keretében. A Kerberos rendszerben a szolgáltatások igénylőit (legyenek azok felhasználók vagy programok) **kliensnek**, a **szolgáltatásokat** nyújtó programokat pedig **szervernek** nevezzük. A rendszer ún. **megbízó leveleket (credentials)** alkalmaz a hozzáférés-szabályozás megoldására. A megbízó levelek beszerzésének mechanizmusa a Kerberos azonosító rendszere.

A Kerberos egy megbízható kívülálló hitelesítési szerviz, amely a korábban ismertetett Needham és Schröder-modellen alapul. A rendszer a módszert kiegészíti az időbélyegek alkalmazásával.

A Kerberos tartalmaz egy adatbázist a klienseiről és szerverekről, illetve ezek kulcsairól. A kliensek és szerverek privát kulcsát csak a Kerberos, illetve a kulcs tulajdonosa ismeri. Felhasználók esetén a privát kulcs egy titkosított jelszó.

A Kerberos a titkos kulcsok ismeretében képes egyrészt a kommunikáló felek azonosítására, másrészt titkosított üzenetváltásra a rendszer minden szereplőjével. A felek egymás közötti (a Kerberostól független) kommunikációjához a Kerberos ún. **viszonykulcsot (session key)** ad, mely egy adott kapcsolatban felhasználható az üzenetek titkosítására.

A Kerberos három különböző védelmi szintet határoz meg. Az alkalmazást készítő programozó választja ki ezek közül az alkalmazások igényeinek leginkább megfelelőt. Az alkalmazások egy részében elegendő a megbízható azonosítás a kapcsolat kezdetén (például az NFS esetében). Ennél egy szinttel magasabban minden egyes üzenetváltásnál szükséges a felek azonosítása. A harmadik szinten nemcsak az azonosítás, hanem az üzenetek titkosítása is megoldható.

Egy kliens csak akkor használhat egy szervizt, ha már korábban szerzett erre egy **jegyet**, azaz speciális megbízólevelet. Ezt kell bemutatnia a szervernek, annak bizonyítása céljából, hogy valóban jogosult a szerviz használatára. Ilyen jegy beszerzéséhez a kliensnek előbb azonosítania kell magát a Kerberos rendszer számára, majd egy megbízólevelet kell szerezni az ún. **jegyosztóhoz (ticket granting service – TGS)**.

A jegyet tehát egyetlen szerver-kliens pár között használjuk. A jegy tartalmazza a szerver nevét, a kliens nevét, a kliens Internet-címét, egy időbélyeget, egy lejárati időt, és egy viszonykulcsot. A kibocsátott jegyet a kliens mindaddig használhatja a szerver elérésére, míg az érvényes. Mivel ezen jegy titkosított a szerver kulcsával, ezért a szerveren kívül más nem tudja ezt értelmezni, így szabadon átadható a kliensnek, mely nem tudja módosítani.

A kerberos jegy: {s,c,addr,timestamp,life,Ks,c}Ks

Ettől eltérően a hitelesítő jegyet a kliens csak egyszer használhatja, minden olyan esetben újat kell generálnia, amikor egy új szervizt akar használni. A hitelesítő jegy a következőket tartalmazza: a kliens nevét, a munkaállomás IP címét, és a munkaállomás aktuális idejét. A hitelesítő jegyet a szerverre és kliensre kiadott kapcsolati kulcssal titkosítja.

A kerberos hitelesítő: {c,addr,timestamp}Ks,c

A rendszerbe lépéskor az azonosítás a **Kerberos szerver** segítségével történik a felhasználó jelszavát használva. Az azonosítás után a Kerberos szerver állít ki olyan megbízóleveket, melyek a kliens hitelességét bizonyítják. Ilyen megbízólevéllel lehet a jegyosztótól a különböző szolgáltatásokhoz jegyet kérni. Az azonosításhoz a felhasználó átadja a felhasználói nevét és a TGS nevét a Kerberos szervernek. A Kerberos szerver ellenőrzi ezt a klienséről tárolt információk alapján. Ha rendben találja, akkor generál egy véletlen viszonykulcsot (Kc,tgs), amelyet később a kliens és jegyosztó szerver fog használni egymás közötti kommunikációjukra. A Kerberos ezenkívül elkészíti a jegyet (Tc,tgs) a TGS számára, amely tartalmazza a kliens nevét, a TGS nevét, az aktuális időt, a jegy élettartamát, a kliens IP-címét és az elkészített random viszonykulcsot (Kc,tgs). Ezt a jegyet a TGS privát kulcsával titkosítja (Ktgs), ily módon biztosítva, hogy a kliens ezt ne tudja megváltoztatni.

A hitelesítő szerver ezután visszaküldi a kódolt jegyet és a Kc,tgs viszony- kulcsot a kliensnek, annak privát kulcsával titkosítva (Kc). Ez lesz a válaszüzenet a felhasználó jelszavára. Ezzel véget ért a bejelentkezési folyamat. Most már a kapott jegy (Tc,tgs) segítségével a kliens a TGS-től beszerezheti a kívánt szerverre a megbízásokat (amíg a jegy élettartama le nem jár).

A Kerberos előnyös tulajdonságai:

- titkosítja a rendszerben áramló adatokat,
- nagyon megbízható,
- TCP/IP protokollra implementálták,
- a felhasználói felületen semmit sem kell változtatni,
- sohasem kerül fel a hálózatra kódolatlan jelszó.

Hátrányai:

- a Kerberos szerver egy kitüntetett gép a rendszerben, ezért ennek biztonságossága az egész rendszerét meghatározza (fizikailag is védeni kell),
- a Kerberos szerver döntései mindenki számára elsődlegesek,
- az összes szoftvert újra kell fordítani, „Kerberizálni”, hogy azok Kerberos hívásokat használjanak.

5.UNIX

Tartalom

5.1. Bevezetés
5.2. A UNIX rövid története
5.3. Belső szerkezet és működés
- 5.3.1. Szerkezet
- 5.3.2. Folyamatkezelés
- 5.3.3. Ütemezés
- 5.3.4. Szinkronizáció
- 5.3.5. Folyamatok közötti kommunikáció (interprocess communication)
- 5.3.6. Állományrendszer implementációk
- 5.3.7. Teljes folyamatok háttértárra írása (swapping)
- 5.3.8. Igény szerinti lapozás

5.4. Hálózati és elosztott szolgáltatások a UNIX-ban
- 5.4.1. A TCP/IP protokoll család
- 5.4.2. A SUN Network File System (NFS)

5.5. POSIX
- 5.5.1. Alapfogalmak, felépítés
- 5.5.2. POSIX környezet
- 5.5.3. Hordozható alkalmazások
- 5.5.4. Folyamatkezelés
- 5.5.5. Állománykezelés
- 5.5.6. Jelzéskezelés
- 5.5.7. Terminálkezelés

5.6. A LINUX-RENDSZER
- 5.6.1. A Linux fejlődésének állomásai
- 5.6.2. A Linux felépítése és működése

5.1. Bevezetés

A UNIX operációs rendszer 1999-ben ünnepli a 30. születésnapját. Ezalatt a 30 év alatt hardverplatformok tucatjaira implementálták, cégek, egyetemek, kutatóintézetek számtalan változatát fejlesztették ki. Mivel a UNIX az egyik legrégebb óta használt operációs rendszer, a fejlesztői közösség annak fejlesztése, implementálása során rengeteg tapasztalattal gazdagodott. Másrészt a fejlesztés különböző lépései viszonylag jól dokumentáltak. Ez a két tényező kiváltképp alkalmassá teszi a UNIX-ot, hogy egy, az operációs rendszereket feldolgozó könyvben esettanulmányként szolgáljon az operációs rendszerek általános elveinek és algoritmusainak demonstrálására.

Ebben a fejezetben áttekintjük, hogy az operációs rendszerek építőköveit hogyan valósították meg a UNIX operációs rendszerben. Célunk, hogy egy konkrét implementáción keresztül megmutassuk az elvek megvalósulását, felhívva az olvasó figyelmét a fejlesztés során felmerülő problémákra és azok orvoslási lehetőségeire. Nem a UNIX pillanatnyilag ismert legfrissebb implementációinak pontos ismertetésére törekszünk (ehhez az érdeklődő olvasó bőséges irodalmat talál az irodalomjegyzékben), hanem, mint azt a könyv címe is mutatja, mérnöki szemlélettel elemezzük az egyes funkciók megvalósítását, annak nehézségeit, illetve azt a gondolatmenetet, ahogyan a fejlesztők eljutottak a mai rendszerekig. A jelen könyv terjedelme nem engedi meg

az összes funkció részletes ismertetését, így didaktikai okokból úgy döntöttünk, hogy egyrészt kiemeljük a legfontosabb funkciókat, másrészt elsősorban az elveket szépen és egyszerűen bemutató implementációt írunk le (nem feltétlenül a legfrissebb implementációt), rámutatunk az ezen implementációk által megvalósított elvekre, kiemelve erényeiket, azonban nem hallgatjuk el az adott implementáció hiányosságait sem. A hiányosságok leírásakor utalunk azok orvoslásának mai megközelítéseire.

Mielőtt rátérnénk a UNIX operációs rendszer belső szerkezetének az ismertetésére, hasznos lehet röviden áttekinteni a UNIX történetét és fejlődését.

5.2. A UNIX rövid története

Az 1960-as évek végén a Bell Telephone Laboratories, a General Electric és az MIT közös projektben vett részt, amelynek célja egy többfelhasználós operációs rendszer kifejlesztése volt. A projekt kudarcba fulladt, de a Bell laboratórium egyik munkatársa, Ken Thompson, 1969-ben egy „Űrutazás" játékot fejlesztett egy PDP-7-es gépre, amihez egy kényelmes futtatási környezetre volt szüksége. Ez motiválta a UNIX kezdetleges változatának a kifejlesztését. (Mint azt ebből is láthatjuk, a nagy dolgokat leginkább az emberi elme játékos hajlama lendíti előre.) A programozási környezet első elemét egy állományrendszer alkotta, ami a későbbiek folyamán a System V állományrendszerré nőtte ki magát. Ennek a felépítését az 5.3.6. alfejezet részletesen tárgyalja.

A programozási környezet egyre népszerűbbé vált a laboratórium munkatársainak körében, de a továbblépéshez a legnagyobb lökést a C nyelvre való áttérés jelentette, amit Dennis Ritchie, Thompson kollégája fejlesztett ki. Ennek óriási jelentősége volt a UNIX fejlesztésében és jelentős szerepet játszott a későbbi gyors terjedésében. A UNIX-ot ettől kezdve már magas szintű programozási nyelven írták, illetve arra törekedtek, hogy az implementáció amennyire csak lehet, hardverfüggetlen legyen. Ehhez modularizálták a rendszer felépítését, elválasztották a hardverfüggő részeket a hardverfüggetlen részektől, és az előbbiek kivételével mindent C-ben kódoltak. Az előbbit hatékonysági okokból továbbra is alacsony szinten írták meg.

A nagy népszerűségnek köszönhetően Thompson és Ritchie 1971-ben publikálta az első UNIX kézikönyvet. Mivel ekkor egy 1956-os trösztellenes törvény miatt az AT&T nem jelenhetett meg a számítógép piacon, a UNIX-ot (forráskód formájában is) ingyenesen az oktatási intézmények rendelkezésére bocsátotta, ami meggyorsította a UNIX elterjedését. Azonban minden éremnek két oldala van: a gyors terjedés mellett ettől kezdve a UNIX-nak rengeteg egymástól eltérő változata jelent meg, ami a későbbiekben jelentős problémát eredményezett.

A kaliforniai Berkeley Egyetem 1974-ben jutott hozzá egy UNIX-licenszhez. A használat során számos hasznos kis programmal egészítették ki, illetve módosításokat eszközöltek rajta. Ennek eredményeként létrejött a Berkeley Software Distribution (BSD), ami ma a UNIX egyik legelterjedtebb alapváltozatát fémjelzi. A másik legjelentősebb alapváltozat a System V a UNIX-ot útjára bocsátó AT&T nevéhez kötődik. Számos kereskedelmi változat is megjelent a piacon, amelyek általában valamelyik alap UNIX-változatra építve értéknövelt szolgáltatásokat nyújtottak, illetve a rendszer bizonyos részeit kereskedelmi alkalmazásokra alkalmassá tették. A

legismertebb változatok: a Sun cég SunOS változata, ami a 4.2BSD-n alapul, majd a későbbi Solaris változatok, amik már a System VR4 rendszert vették alapul, a Santa Cruz Operation kidolgozta a System VR3 alapjaira építve Intel processzorra az SCO UNIX-ot, az IBM az AIX-et (ami elsők között alkalmazott kereskedelmi napló alapú *(journaling)* állományrendszert), a Hewlett-Packard Corporation megjelent a HP UX rendszerével, és a Digital piacra dobta az Ultrix-ot (majd később a DEC OSF/1-et, amit Digital UNIX-ra kereszteltek). Az Ultrix volt az egyik első többprocesszoros UNIX-rendszer.

A UNIX elterjedéséhez, mint azt már korábban láttuk, nagyban hozzájárult, hogy eleinte ingyen hozzá lehetett jutni a forráskódhoz, viszont egyben ez jelenti ma az egyik legnagyobb problémát, ugyanis számos, egymástól többé-kevésbé eltérő változat jelent meg. Az utóbbi időben a probléma orvoslására több szabványosítási kísérlet is megindult. A legtöbb gyártó megegyezett pár szabványban. Ezek közé tartozik az AT&T System V Interface Definition (SVID) szabványa, az IEEE POSIX-szabvány (amit az 5.5. alfejezet részletesen tárgyal) és az X/Open konzorcium X/Open Portability Guide. Ezen szabványok mindegyike a programozó és az operációs rendszer közötti interfésszel foglalkozik, és nem törődik annak megvalósításával. Az 1990-ben megjelentetett POSIX1003.1 szabvány (elterjedt nevén POSIX.1) ötvözi az SVR3 és a 4.3BSD lényegi részeit. Széles körű elfogadásának egyik oka, hogy a szabvány egyik UNIX-változat mellett sem kötelezi el szorosan magát (lásd az 5.5. alfejezetet).

A rendszer fejlődésével egyre bővült, gazdagodott a rendszer nyújtotta szolgáltatások halmaza. Ezek közül talán a legfontosabbak a System V IPC-ként ismert folyamatok közötti kommunikációs eszközök, illetve a hálózati kommunikációt támogató eszközök, amik manapság már minden UNIX-változat kerneljében megtalálhatók.

A UNIX-rendszerek lépést tartanak a számítógépes hardvertechnológia fejlődésével. A leggyakrabban ez a rendszer átemelését *(portolását)* jelenti újabb processzorokra és architektúrákra. Ez általában nem okoz túl nagy gondot, hisz mint azt már láttuk, a rendszer számottevő része C-ben íródott. Néhány esetben azonban a rendszerfejlesztők lényegesen keményebb feladattal találják szemben magukat: jelentős módosításokat kell végrehajtani a kernelen. A hagyományos UNIX-ot egyprocesszoros rendszerekre tervezték, így annak idején nem tervezték bele az adatok védelmét több processzor konkurens adathozzáféréseivel szemben. Bizonyos gyártók ezt zárak bevezetésével orvosolták, míg mások radikálisan megváltoztatták a kernel szerkezetét.

A különféle hardvertechnológiák eltérő fejlődési sebessége jelentős hatást gyakorol az operációs rendszer tervezésére. Az első, PDP-7-esen futtatott UNIX-rendszer megjelenése óta a CPU sebessége körülbelül három nagyságrenddel gyorsult, a felhasználók rendelkezésére álló memória és diszk terület több mint egy nagyságrenddel növekedett, azonban a memória és a lemez sebessége alig duplázódott meg. A ’70-es években a UNIX teljesítményét a processzor sebessége és a memória mérete korlátozta, ezért a rendszer folyamatok háttértárra írásával (swapping) és lapozási technikákkal próbálta kezelni a memória gondokat. Az idő előre haladtával a memória és a CPU-sebesség okozta problémák egyre inkább elvesztették jelentőségüket, a rendszer egyre inkább B/K korlátozottá vált. Ez jelentős kutatásokat inspirált az állományrendszer, a virtuális memória

kezelés és a tárolás megszervezésének a területén, hogy megbirkózzanak a lemezen tárolt adatok elérési ideje, mint szűk keresztmetszet okozta problémákkal. Megjelentek az újszerű elveken alapuló állományrendszerek (például journaling), illetve kidolgozták a RAID-technikát.

A technológia gyors fejlődése lehetővé tette újszerű alkalmazások kialakulását. Megjelentek a multimédiás alkalmazások, amik korlátos válaszidőket és erőforrás rendelkezésre állási garanciákat igényeltek. A beágyazott alkalmazások is hasonló igényeket támasztottak az operációs rendszerrel szemben. Ennek hatására a modern UNIX-rendszerek kernelében megjelentek ún. „puha valósidejű” *(soft real-time)* elemek. (Például a Solaris új változatai támogatják a *soft real-time* ütemezési algoritmusok alkalmazását.)

Az eredeti UNIX-rendszer egyik nagy erénye a kis méretében rejlett. Az egész kezdeti fejlesztés hűen tükrözte az „*A kicsi szép*” filozófiát. A hatékony működést egyszerű eszközök, építőkövek alkalmazása garantálta, amiket rugalmasan lehetett összeépíteni (az egyik legszebb példa erre a szűrők alkalmazása). A hagyományos UNIX-kernel azonban monolitikus és nehezen bővíthető volt. Ahogy egyre több funkcióval bővült, egyre inkább elhajlott a kezdeti filozófia irányától, egyre nagyobbá vált.

A továbbiakban áttekintjük a UNIX belső felépítését és tárgyaljuk a legfontosabb alrendszerek funkcióit és megvalósításait.

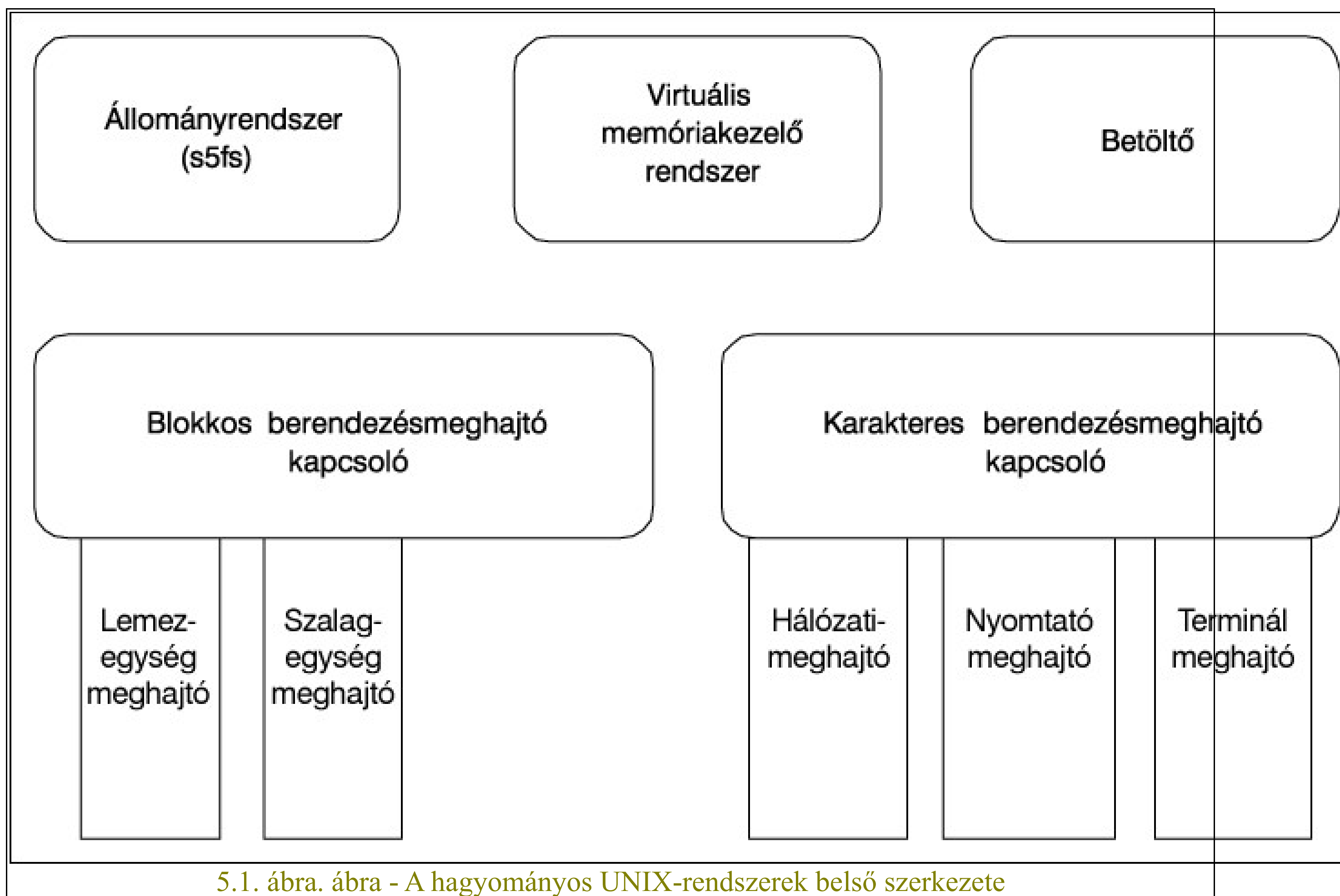
5.3. Belső szerkezet és működés

Az alábbi részek a UNIX-rendszer legfontosabb alrendszereit és funkcióit tárgyalják.

5.3.1. Szerkezet

Ebben a részben a UNIX-kernel felépítését tárgyaljuk. Röviden ismertetjük a hagyományos és a mai modern UNIX **kernel réteg** (illetve modul) **szerkezetét** és vázoljuk ezek fő feladatait.

A korai ’80-as évekig az eltérő UNIX-változatok ellenére a kernelek elég egységes képet mutattak. Egyetlen állományrendszer típust, egyetlen ütemezési politikát és egyetlen végrehajtható állomány formátumot támogattak (5.1. ábra).



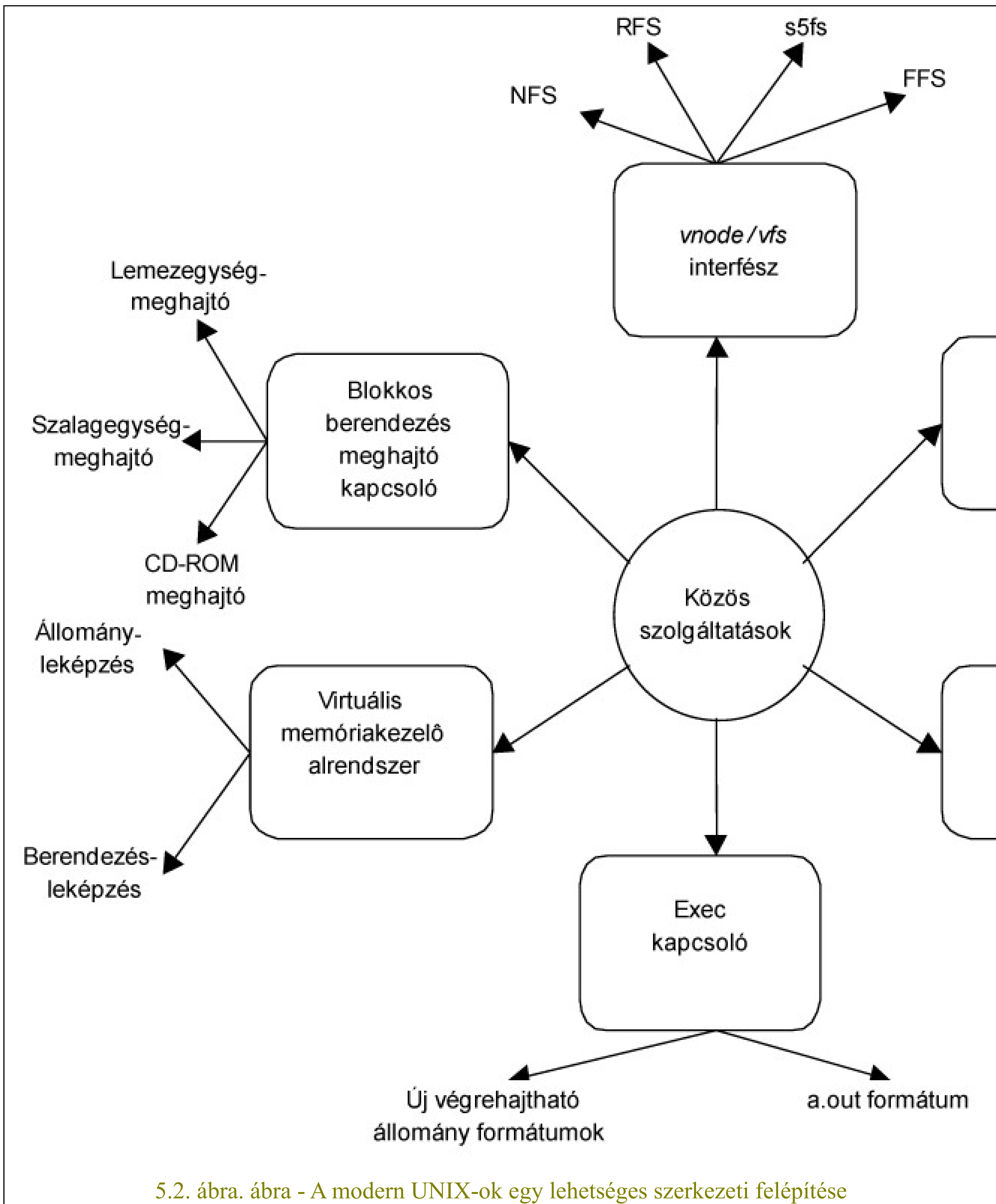
5.1. ábra. ábra - A hagyományos UNIX-rendszerek belső szerkezete

Az 5.1. ábra jól mutatja a moduláris felépítést. Ebben a szerkezetben a rugalmasságot a blokkos és a karakteres berendezés kapcsolók biztosították. Ezek lehetővé tették, hogy a rendszerben eltérő típusú berendezéseket egy egységes interfészen keresztül lehessen elérni. A **réteges szervezésben** rejlő előnyöket már ekkor is hatékonyan kihasználták. Az egymásra épülő rétegek a közvetlen felettük lévő réteg szolgáltatásait egy jól definiált **interfészen** keresztül vehették igénybe. Sajnos a korai implementációkban a megvalósítás leegyszerűsítése érdekében ez alól akadt néhány kivétel, ami a későbbiekben meg is nehezítette a fejlesztők dolgát.

Mint azt a későbbiekben látni fogjuk, a ’80-as évek közepén egyre nagyobb teret nyerő elosztott állományrendszerek megkövetelték, hogy a UNIX adjon támogatást mind a lokális, mind pedig a távoli állományrendszerek kezeléséhez. Másrészt az osztott könyvtárak megjelenésével szükségessé vált több futtatható fájlformátum támogatása is, illetve speciális alkalmazások (mint például a korábban említett multimédiás alkalmazások) az ütemezéssel szemben is újfajta igényeket támasztottak. Ebből adódóan rugalmasabb kernelszerkezet kialakítására volt szükség, ami több alrendszer, interfész laza, de szerves kapcsolatából épül fel. Az 5.2. ábra ezt az új szerkezetet mutatja. A külső lekerekített téglalapok interfészeket reprezentálnak, amiket számos eltérő módon lehet megvalósítani. Jól látható, hogy az egyes főbb funkciókhoz

tartozik egy-egy interfész (de akár több is), amelyeken keresztül az adott funkciók többféle megvalósítását el lehet érni.

Erre talán az egyik legjobb példa az állományrendszer. Míg a korai UNIX-rendszerekben csak egyetlen állományrendszer állt rendelkezésre, a gyakorlati igények megkövetelték, hogy egyszerre több, akár eltérő típusú állományrendszerrel is dolgozni lehessen. Ennek eredményeképp előállt a *vnode/vfs* interfész, ami már absztrakt szinten kezeli az állományrendszert, lehetővé téve ezáltal eltérő állományrendszerek egységes kezelését. Az interfészből kiinduló nyíl a különféle implementációkat mutatja. Ebben a példában a hagyományos System V *s5fs* mellett található egy Berkeley **FFS**, egy elosztott **NFS** és **RFS** állományrendszer implementáció is. Az állományrendszerek részletes leírását és fejlődésüket az 5.3.6. alfejezet részletesen tárgyalja. A többi alrendszer esetén is hasonló igények merültek fel. Az 5.2. ábrán például a futtatható állományok kezeléséért felelős *exec* kapcsolóhoz (alrendszer) is a hagyományos a.out végrehajtható állomány formátum mellett más formátumok kezelését is lehetővé kell tenni.



5.2. ábra. ábra - A modern UNIX-ok egy lehetséges szerkezeti felépítése

A továbbiakban a UNIX főbb funkcióit, alrendszereit, és a hozzájuk kapcsolódó interfészeket tárgyaljuk.

5.3.2. Folyamatkezelés

Mint azt korábban láttuk, az operációs rendszer elsődleges feladata, hogy a felhasználói programoknak, alkalmazásoknak **végrehajtási környezetet (execution environment)** biztosítson. A UNIX végrehajtási környezet a **folyamat (process)** absztrakción alapszik. A hagyományos UNIX-rendszerekben a folyamat egyetlen utasítássorozatot hajt végre egy **címtérben (address space)**. A modern UNIX-változatokban már egy folyamaton belül egyszerre több szálon futhat végrehajtás, több vezérlési pont található. A **szálak (threads)** vagy **könnyűsúlyú folyamatok (lightweight processes)** bevezetése bizonyos alkalmazásoknál jelentős hatékonyság növekedést eredményezett. A **kliens-szerver architektúra** szerver komponense hatékonyan kihasználja a szálak alkalmazásából adódó előnyöket.

Ebben a részben a UNIX hagyományos folyamatmodelljét ismertetjük. A UNIX modern folyamatmodelljének a részleteit az érdeklődő olvasó az irodalomjegyzékben felsorolt idevágó irodalmak tanulmányozásából részletesen megismerheti.

A UNIX-rendszer egy multiprogramozott környezet, a rendszerben egyszerre párhuzamosan több folyamat is aktív. Ezen folyamatok egy **virtuális gépen (virtual machine)** futnak, minden folyamat a B/K-műveletek és a készülékek vezérlésén kívül úgy érzékeli, mintha csak egyedül ő futna a gépen. A B/K-műveleteket és a készülék vezérlést az operációs rendszer végzi el a folyamat számára. A virtuális gép koncepciójának megfelelően minden folyamat rendelkezik egy regiszterkészlettel, ami a valós hardverregisztereknek felel meg. A rendszerben számos aktív folyamat található, viszont általában – az egyetlen CPU-nak megfelelően – csak egy hardverregiszter-készlet létezik. A kernel az aktív, futó folyamat regisztereit tartja a hardverregiszterekben, a többi folyamat regiszterkészletének tartalmát elmenti folyamatonkénti adatszerkezetekbe.

A kernel tehát az a speciális program, ami közvetlenül a hardveren fut, és a kernel valósítja meg több más rendszer szolgáltatással egyetemben a folyamat modellt.

5.3.2.1. Végrehajtási módok és környezetek

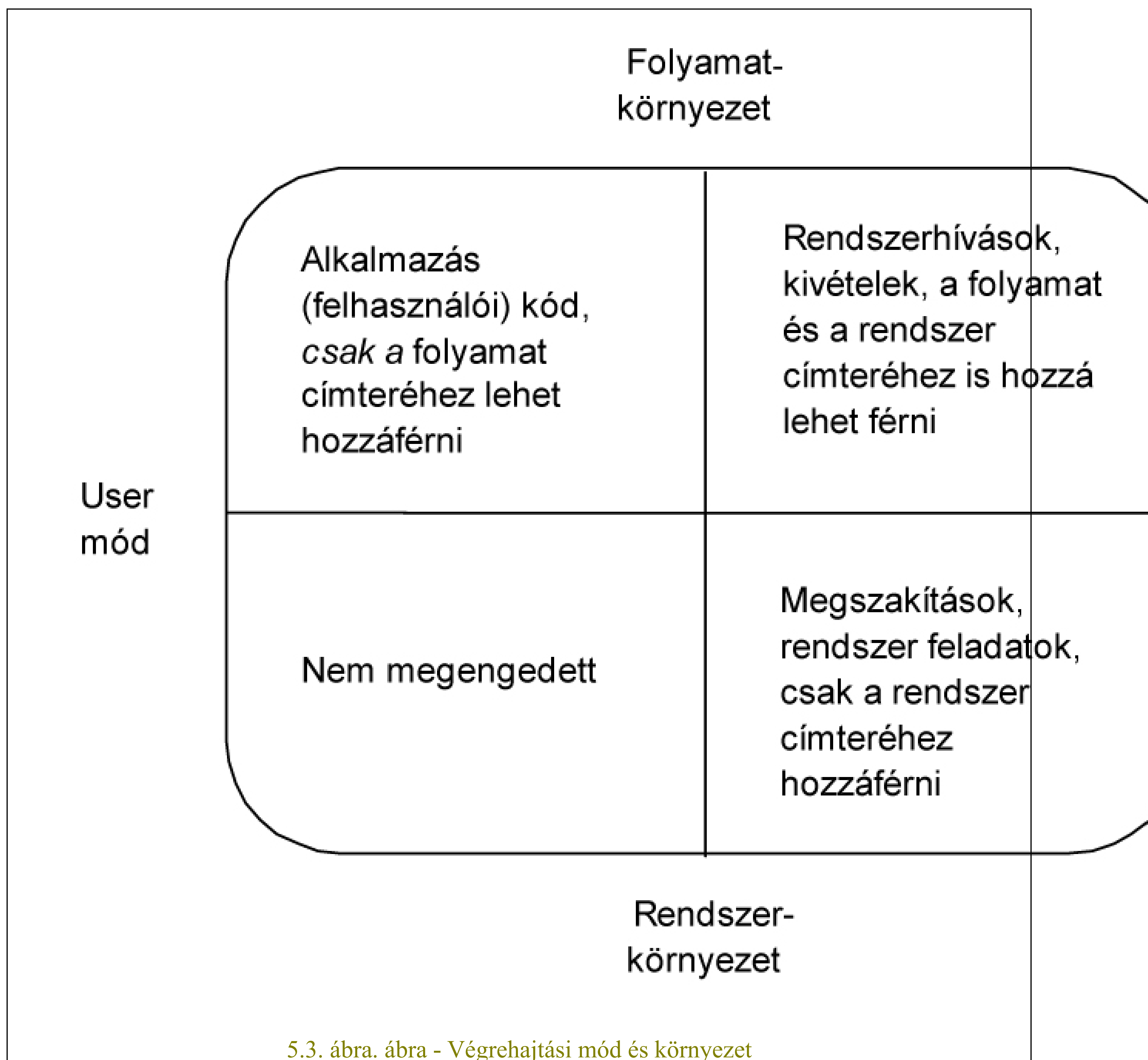
A UNIX futtatásához olyan hardverre van szükségünk, ami legalább két **végrehajtási módot (execution mode)** biztosít: egy privilegizált **kernel módot**, és egy **felhasználói (user) módot**. A kernel a címtér egy részét védi *felhasználói módú* hozzáféréssel szemben, valamint bizonyos *privilegizált utasítások csak kernel módban adhatók ki*. Ezek tipikusan memóriakezelő, illetve B/K-kezelőutasítások.

Ma már szinte minden általános célú UNIX-változat alkalmaz **virtuális memóriakezelést (virtual memory)**. Ekkor a folyamatok a virtuális címtartományban adnak ki címeket és a rendszer *címleképzési táblázatokkal* futási időben köti ezekhez a virtuális címekhez a valós fizikai címeket. Minden folyamat virtuális címtartományának egy rögzített része a kernel fizikai címtartományára képződik le, ahol a kernel kódja és adatszerkezetei találhatók. Ezt a **kernel területet (kernel area)** csak kernel módban lehet elérni. A folyamatok közvetlenül nem, csak rendszerhívásokkal tudják elérni a kernelt.

Meg kell említeni két fontos folyamatonkénti adatszerkezetet, amiket bár a kernel kezel, gyakran a folyamat címterében implementálnak. Ezek az **u-terület (u area)** és a **kernelverem (kernel stack)**. Az u-terület a kernel számára fontos információkat tárol a folyamatról, mint például a megnyitott állományok táblája, azonosítók, a regiszterek elmentett tartalma, ha a folyamat nem fut stb. Ezen információkat a későbbiekben részletesebben tárgyaljuk. Bár az u-terület a folyamat címterében található, ahhoz csak kernel módban lehet hozzáférni, annak tartalmát a folyamat maga önkényesen nem módosíthatja.

A **végrehajtási környezet (execution context)** a folyamat szempontjából nagy jelentőséggel bír. A kernel részek végrehajtódhatnak **folyamat** vagy **kernel környezetben (process, kernel context)**. Folyamat környezetben a kernel az adott folyamat számára hajt végre valamilyen feladatot (például egy rendszerhívás során). Ekkor a kernel hozzáfér a folyamat címteréhez, u-területéhez és a kernel verméhez. Sőt, a kernel folyamat környezetben blokkolhatja is az aktuális folyamatot, ha a végrehajtás során valamilyen erőforrásra vagy eseményre kell várakoznia.

A kernelnek bizonyos rendszer szintű feladatokat is végre kell hajtania, amiket nem kifejezetten egy adott folyamat számára hajt végre, ezeket rendszerkörnyezetben hajtja végre. Tipikus rendszerkörnyezetben végrehajtott feladatok a külső megszakítások kezelése, a prioritások újraszámítása stb. Rendszerkörnyezetben a kernel nem fér hozzá közvetlenül az aktuális folyamat címteréhez, u-területéhez vagy kernel verméhez. Meg kell jegyezni, hogy bizonyos speciális indirekt leképzések segítségével a kernel ekkor is elérheti az aktuális folyamat címterét. Rendszerkörnyezetben a kernel nem blokkolódhat, mert ekkor ezzel egy „ártatlan” folyamatot blokkolna. Az 5.3. ábrában a mód és a környezet tengelyek négy negyedre osztják az ábrát. Az egyes negyedekbe beírtuk az adott állapotban elvégezhető műveleteket.



5.3. ábra. ábra - Végrehajtási mód és környezet

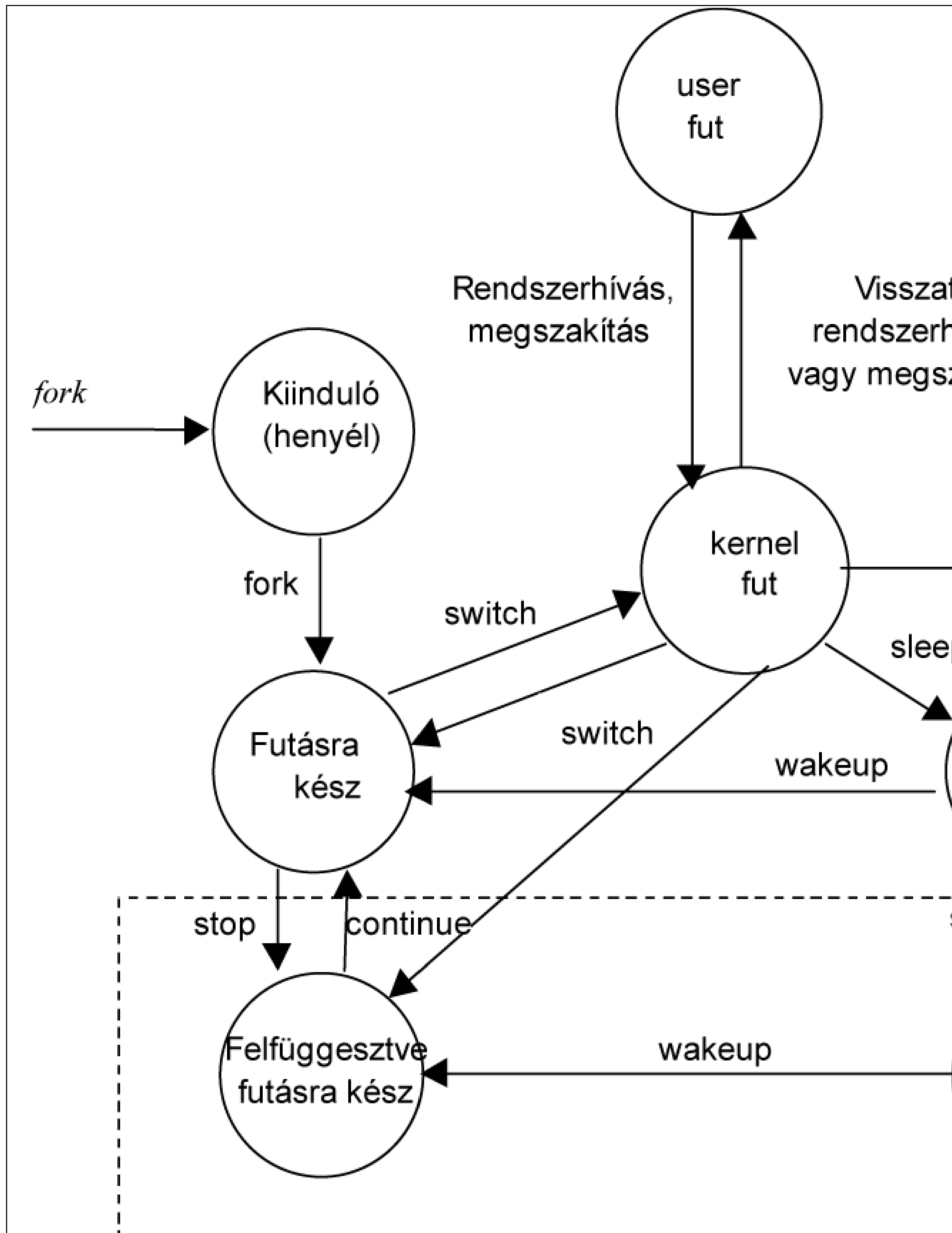
5.3.2.2. A folyamat absztrakció – a folyamatok állapotai és az állapotátmeneti gráf

A folyamatokat gyakran úgy definiálják, hogy „A folyamat egy végrehajtás alatt álló program.” A UNIX-rendszerrel kapcsolatban talán szerencsésebb, ha úgy fogalmazunk, hogy a folyamat egy olyan entitás, ami egyrészt futtat egy programot, másrészt biztosítja a futáshoz szükséges végrehajtási környezetet.

A UNIX folyamatai jól definiált hierarchiát alkotnak. Minden folyamatnak pontosan egy **szülője (parent)** van (aki mint azt majd a későbbiekben látjuk, *fork* rendszerhívással hozza létre gyermekét), és egy vagy több

gyermek folyamata (child process) lehet. A folyamat hierarchia tetején az *init* folyamat helyezkedik el. Az *init* folyamat az első létrehozott felhasználói folyamat, a rendszer indulásakor jön létre. Minden felhasználói folyamat az *init* folyamat leszármazottja. Néhány rendszer folyamat, mint például a *swapper* és a *pagedaemon* (a háttértár kezelésével kapcsolatos folyamatok), szintén a rendszer indulásakor jön létre és nem az *init* folyamat leszármazottja. Ha egy folyamat befejeződésekor még léteznek aktív gyermek folyamatai, akkor azok **árvákká (orphan)** válnak és azokat az *init* folyamat örökli.

A UNIX folyamatai minden időpillanatban egy jól definiált állapotban találhatók. Ezeket az állapotokat és a közöttük lehetséges állapotátmeneteket egy állapotátmeneti gráffal lehet szemléletesen ábrázolni. Az 5.4. ábra ezt a gráfot mutatja.


user
fut
Rendszerhívás,
megszakítás
Visszatérés
rendszerhívásból
fork
Kiinduló
(henyél)
kernel
fut
exit
fork
switch
sleep
Futásra
kész
switch
wakeup
stop
continue
stop
Felfüggesztve
futásra kész
wakeup

5.4. ábra. ábra - A folyamatok állapotátmeneti gráfja

Mint azt korábban láttuk, új folyamatot a *fork* rendszerhívással lehet létrehozni, amelynek hatására a folyamat **kezdeti** állapotba kerül. Itt végrehajtódnak a legfontosabb inicializálások, majd a *fork* lefutása után a folyamat készen áll a futásra, már csak a processzorra van szüksége. Ezt az állapotot jelöli a **futásra kész (ready)** állapot. Innen az ütemezés hatására egy **környezetváltással (context switch)** (*swtch* rendszerhívás) **kernel fut (kernel running)** állapotba kerül. Egy frissen létrejött folyamat a rendszerhívás befejeződése után átlép **user fut (user running)** állapotba. Egy *user módban* futó folyamat egy rendszerhívás vagy egy megszakítás hatására léphet át *kernel fut* állapotba, ahonnan a rendszerhívásból vagy a megszakításból való visszatérés után léphet vissza a *user fut* állapotba. Az 5.4. ábrán a további állapotátmenetek: amikor egy folyamat befejezi futását és végrehajtja az *exit* rendszerhívást, átlép **zombi (zombie)** állapotba. A *zombi* állapotban a folyamat már felszabadította a foglalt memóriát, lezárta az állományokat, minden erőforrását visszaadta a rendszernek, csak a ***proc*** struktúráját tartja fogva, amiben visszatérési és statisztikai információkat tárol a szülő számára. A szülő *wait* rendszerhívásának hatására a folyamat véglegesen kilép az állapot átmenet gráfból, felszabadul a *proc* struktúra is, a folyamat teljesen megszűnik létezni.

Amikor egy folyamatnak valamilyen erőforrásra kell várakoznia, akkor kiad egy *sleep* rendszerhívást és **alvó** állapotba megy. Az alvó állapotból a várt esemény bekövetkezése által kiváltott *wakeup* rendszerhívás hatására a folyamat átlép a *futásra kész* állapotba és várja, hogy az ütemező ismét futásra ütemezze. Az 5.4. ábra alján a bekeretezett két állapotot először a 4BSD vezette be, majd később a SVR4 is átvette. Egy *alvó* folyamat egy *stop* jelzés hatására átlép a **felfüggesztve alszik (stopped + asleep)** állapotba. (A *stop*[1] jelzés különlegessége, hogy azt a rendszer a többi jelzéssel ellentétben azonnal kezeli. A jelzések kezelésével részletesebben az 5.3.4. rész foglalkozik.) Ebből az állapotból a *continue* jelzés hatására a folyamat visszalép az *alvó* állapotba. Ha a *felfüggesztve alszik* állapotban következik be egy *wakeup* rendszerhívás, akkor a folyamat átlép a **felfüggesztve futásra kész** állapotba, ahonnan egy *continue* jelzés hatására kerül át a *futásra kész* állapotba.

5.3.2.3. Folyamatok környezete (kontextus)

Mint azt korábban láttuk, a folyamat absztrakció egy *végrehajtási környezetet* biztosít. Ennek a környezetnek tartalmaznia kell a folyamatok leírásához szükséges összes információt. Ezek az alábbi főbb részekből állnak:

- felhasználói (user) címtér,
- vezérlési információk:
  - *u*-terület,
  - *proc* struktúra (folyamattábla bejegyzés),
- hitelesítők (credentials), többek között a felhasználó és csoport azonosítók),
- környezetváltozók,

- hardverkontextus (program számláló, veremmutató, a processzor állapota, memóriakezelő, regiszterek, lebegőpontos egység regiszterei stb.).

A hardver regiszterei mindig a végrehajtás alatt álló aktuális folyamat környezetét tárolják. Környezetváltáskor ezen regiszterek tartalmát az operációs rendszer elmenti az aktuális folyamat *u-területére*. Az ütemező által kiválasztott új folyamat környezetét a kernel betölti a regiszterekbe, és az új folyamat folytathatja a futását.

A továbbiakban fontosságuk miatt részletesebben ismertetjük a hitelesítőket és megadjuk az *u-területen* és a *proc* struktúrában tárolt főbb információkat.

Hitelesítők (credentials)

A UNIX-rendszerben minden felhasználót egy egyedi azonosító, a **felhasználóazonosító** (UID) azonosít. Ezen felül minden felhasználó egy vagy több csoportba tartozik, amiket szintén egyedi azonosítók, a **csoportazonosítók** (GID) azonosítanak. Minden rendszerben van egy kitüntetett felhasználó, a **superuser**, aki kitüntetett jogosultságokkal rendelkezik a rendszerben. A superuser UID-je 0, GID-je 1. Meg kell jegyezni, hogy a modern UNIX-rendszerek fejlett biztonsági megoldásokat nyújtanak, ahol a privilégiumokat összefogva kezelő superuser absztrakció helyett műveletenkénti privilégiumkezelés valósul meg.

Az azonosítókból minden folyamat egy párral rendelkezik: a **valódi (real)** és az **effektív (effective)** azonosítókkal. Az effektív azonosítók (UID, GID) az állománykezelésben, míg a valódi azonosítók a jelzések kezelésében töltenek be meghatározó szerepet. Az operációs rendszer ki is használja ezt a kettősséget. Amikor egy folyamat az *exec* rendszerhívással futtat egy programot, aminek a *suid* bitje be van állítva, akkor a kernel a folyamat effektív UID-jét átállítja a végrehajtható állomány tulajdonosának az azonosítójára. Hasonlóan, ha a program *sgid* bitje be van állítva, akkor a kernel a folyamat effektív GID-jét átállítja a végrehajtható állomány tulajdonosának az azonosítójára. Ezen a mechanizmuson keresztül a UNIX különleges jogosultságokat tud biztosítani a felhasználóknak bizonyos feladatok végrehajtásához. A mechanizmus alkalmazásának egy gyakran idézett példája a *passwd* program, amivel a felhasználó megváltoztathatja a jelszavát. A jelszó egy, a rendszer tulajdonában lévő jelszó-adatbázisban található, amit a felhasználók közvetlenül nem módosíthatnak. A *passwd* program tulajdonosa a superuser és a *suid* bitje be van állítva. Így amikor a felhasználó a programot futtatja, hogy megváltoztassa a jelszavát, akkor állománykezelés szempontjából superuseri jogosultságokat kap, így elvégezheti a megfelelő módosítást.

Ezenfelül még a *setuid* és a *setgid* rendszerhívásokkal lehet az azonosítókat módosítani. A felhasználók ezekkel a rendszerhívásokkal az effektív azonosítóikat visszaállíthatják a valódi azonosítóikra, míg a superuser mind az effektív, mind pedig a valós azonosítókat megváltoztathatja.

Az u-terület és a proc struktúra

A folyamatokhoz kapcsolódó vezérlési információkat két folyamatonkénti adatszerkezet az *u-terület* és a *proc* struktúra tárolja. Korábbi implementációkban a kernel egy rögzített méretű *proc* struktúrákból álló tömböt, egy

úgynevezett **folyamattáblát (process table)** tartalmazott. A folyamattábla mérete meghatározta a rendszerben az egyidejűleg létező folyamatok maximális számát. A későbbi implementációk ezt a merev szerkezetet leváltották egy dinamikus allokációs sémával: a kernel szükség esetén újabb *proc* struktúrákat tud allokálni. Egy nagyon lényeges különbség van a *proc* struktúra és az *u-terület* között: míg a *proc* struktúra rendszerterületen található, addig az *u-terület* a folyamat címterének része (bár a kernel felügyelete alatt áll). Ebből adódóan a kernel az összes (nem csak a futó) folyamat *proc* struktúrájához hozzáfér, míg csak a futó folyamat *u-területét* éri el (indirekt mechanizmusokon keresztül elérheti a többi folyamat *u-területét* is, azonban ez lassabb hozzáférést biztosít). Ebből adódóan a UNIX-rendszerek evolúciója során hatékonysági okok miatt bizonyos korábban az *u-területen* tárolt (többek között az ütemezéssel és a jelzéskezeléssel kapcsolatos) információkat áthelyeztek a *proc* struktúrába. Az alábbiakban megadjuk az *u-területen* és a *proc* struktúrában tárolt leglényegesebb információkat.

Az u-területen tárolt főbb információk

- PCB (Process Control Block) ® hardverkörnyezet,
- mutató a folyamattábla bejegyzésre (*proc* struktúrára),
- valós és effektív UID, GID,
- argumentumok, visszaadott értékek, hibaérték az aktuális rendszerhívásból,
- jelzéskezelők,
- program-fejlécinformációk (kód-, adat- és veremméret stb.),
- folyamatonkénti állományleíró tábla (nyitott állományokról),
- mutató az aktuális könyvtár és a vezérlő terminál *v-node*-jára,
- CPU használati statisztikák, diszkkvóta, erőforrás-korlátok,
- folyamat kernelverme.

A proc struktúrában tárolt főbb információk

- PID, PGID, SID (munkamenet-azonosító),
- kernel címleképzési térkép az *u*-területhez,
- az aktuális folyamatállapot,
- mutatók az ütemezési (alvás esetén az alvó) sorokba,
- ütemezési prioritás és információk,
- jelzéssel kapcsolatos információk (például maszk),
- memóriakezelési információk,
- mutatók az aktív, szabad, illetve zombi folyamat sorokra,

- egyéb jelzőbitek,
- hash mutatók,
- hierarchiainformációk (folyamatok közötti kapcsolatról).

5.3.2.4. Folyamatok létrehozása

A UNIX-ban új folyamatok létrehozására a *fork* rendszerhívás szolgál. A létrejött folyamat majdnem teljes mása a szülő folyamatnak, csak a megkülönböztetésükhöz szükséges adatokban (folyamat azonosító) térnek el. A *fork* rendszerhívás az alábbi főbb feladatokat hajtja végre:

- háttértár *(swap)* területfoglalás a folyamat számára,
- PID generálás a gyermekfolyamat számára,
- *proc* struktúra foglalás és inicializálás,
- címleképzés táblák allokálása,
- *u*-terület frissítése, hogy az már az új címleképzési táblát tükrözze,
- osztottan használt kódtartomány megfelelő kezelése,
- szülő verem- és adattartományainak duplikálása,
- referenciák begyűjtése osztottan használt erőforrásokra (nyitott állományok stb.),
- hardver kontextus inicializálása szülőtől másolva,
- a folyamat futásra késszé tétele, illetve a megfelelő ütemezési sorba helyezése,
- a gyermeknek 0, a szülőnek a gyermek PID-jének, mint visszatérési értéknek a visszaadása.

5.3.2.5. Folyamatok befejezése (terminálás)

Egy folyamat kétféle okból fejeződhet be (terminálódhat). Ha elvégezte a feladatát, működését normál módon fejezte be. Ekkor a folyamat maga adja ki az *exit* rendszerhívást, ami az *exit()* függvényt meghíva terminálja a folyamatot. Egy folyamat egy jelzés hatására is befejezheti a futását. Az ilyen „rendellenes" terminálás során az operációs rendszer hívja meg az *exit()* függvényt a folyamat megszüntetése céljából. Az alábbiakban vázlatosan ismertetjük az *exit()* függvény által elvégzendő feladatokat.

- a jelzések „kikapcsolása",
- a nyitott állományok lezárása,
- a text állomány és más erőforrások, mint például az aktuális könyvtár felszabadítása,
- a megfelelő információk beírása a statisztika naplóba (log),
- az erőforrás használati statisztikák és a kilépési státus elmentése a *proc* struktúrába,

- átlépés zombi állapotba és a folyamat a zombi listára kerül,
- amennyiben a folyamatnak még léteznek élő gyermekei, azokat az *init* folyamat örökli,
- a címtér, a foglalt területek, táblák, címtérképek, a háttértár (swap) terület felszabadítása,
- SIGCHLD jelzés küldés a szülőnek (amely általában figyelmen kívül hagyja azt),
- a szülő felébresztése, amennyiben az alszik,
- *swtch()* rendszerhívással átütemezés kezdeményezése.

Ezek után a folyamat már csak egy folyamattábla-bejegyzést foglal. Ezt a folyamat állapotot nevezik zombi állapotnak. A szülő feladata, hogy a statisztikai információk begyűjtése után ezt a fennmaradó memóriadarabot felszabadítsa. Tipikusan a szülő *wait()* rendszerhívással megvárhatja a gyermek terminálását és felszabadíthatja ezt az utolsó erőforrást is.

5.3.3. Ütemezés

A CPU-ütemező, illetve az ütemezési algoritmus minden multiprogramozott operációs rendszer lelke. A rendszer teljesítménye, illetve a hardver kihasználtsága szempontjából az egyik legfontosabb, ha nem a legfontosabb tényező a megfelelően kiválasztott és a körültekintően paraméterezett ütemezési algoritmus.

Ebben a részben ismertetjük a tradicionálisnak tekintett UNIX ütemezési algoritmust, amit az SVR3, illetve 3.1BSD UNIX-rendszerek alkalmaztak. Az ismertetetésre kerülő ütemezési algoritmust a UNIX-rendszerekben használt ütemezés tipikus példájának tekinthetjük, annak ellenére, hogy a ma használt UNIX-rendszerek az algoritmus valamilyen továbbfejlesztett változatát használják.

Mint látni fogjuk, a UNIX-ban használt ütemezés meglehetősen összetett. Az ismertetésre kerülő ütemezési algoritmus kiválasztásával az volt a célunk, hogy a UNIX-rendszerek ütemezőiben használt minél több ötletet bemutathassunk.

5.3.3.1. Az ütemezési algoritmussal szemben támasztott követelmények

A UNIX-rendszert elsősorban többfelhasználós, interaktív és batch programokat egyaránt futtató felhasználói környezetre tervezték. Az ütemezési algoritmussal szemben támasztott követelményeket a következőkben foglalhatjuk össze:

- alacsony válaszidő biztosítása az interaktív folyamatok támogatásának érdekében,
- nagy átbocsátó képesség biztosítása,
- az alacsony prioritású, háttérben futó folyamatok éhezésének elkerülése.

A fenti követelményeket még kiegészíthetjük az ütemezőkkel szemben támasztott általános kritériumokkal:

- a rendszer terhelését figyelembe vevő ütemezés, mely a terhelés növekedésével nem hirtelen omlik össze, hanem lehetőséget ad beavatkozásra,

- a felhasználónak legyen lehetősége a folyamatok futási esélyeinek befolyásolására.

A UNIX ütemezési algoritmusa a fenti követelmények megvalósítása jegyében született, azonban néhány ponton a fenti követelmények egymásnak ellentmondó igényeket képviselnek, így – mint látni fogjuk – az algoritmus nem mindegyik követelményt elégíti ki maradéktalanul.

5.3.3.2. A UNIX-ütemezés rövid jellemzése

A UNIX ütemezése *prioritásos*. A rendszer minden egyes folyamathoz hozzárendel egy, az időben dinamikusan változó prioritást. Az ütemezés felhasználói, illetve kernel módban eltér egymástól:

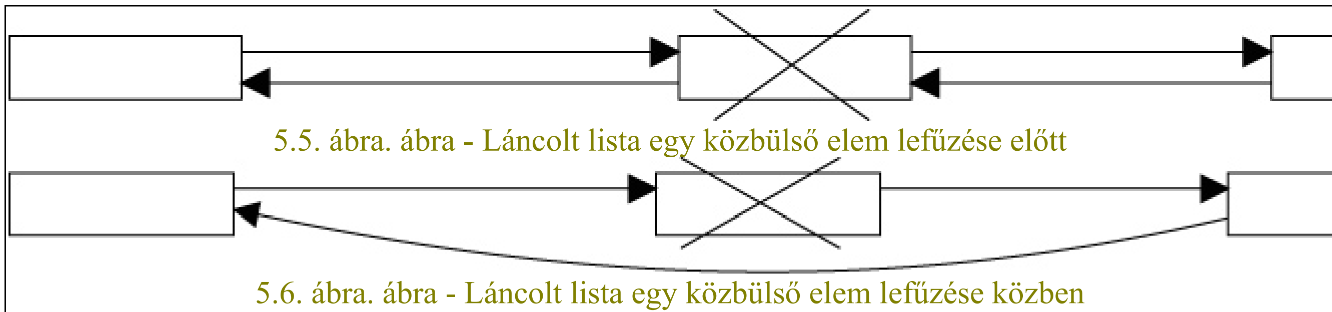
- *A felhasználói módban futó folyamatok ütemezése:* preemptív ütemezés, időosztásos, időben változó prioritású folyamatok, azonos prioritású folyamatok esetén *körforgásos (Round–Robin)*, FCFS *(First Come First Served:* érkezési sorrend szerinti kiszolgálás).
- *A kernel módban futó folyamatok ütemezése:* nem preemptív ütemezés, rögzített prioritású folyamatok.

A kernel módban az ütemezés szigorúan nem preemptív. Kernel kódot végrehajtó folyamatot (például rendszerhívás, megszakítás-kezelés) nem lehet kényszeríteni, hogy lemondjon a CPU használatról egy nagyobb prioritású folyamat javára. Újraütemezés akkor következik be, amikor a folyamat önként lemond a futás jogáról (*sleep* rendszerhívást hajt végre), illetve, amikor a folyamat visszatér kernel módból felhasználói módba.

Megjegyzendő, hogy a kernel kód újrahívható, vagyis egyszerre több példányban is végrehajtódhat. (Például, ha egy kernel módban futó folyamat lemond a futás jogáról, vagyis egy sleep rendszerhívást hajt végre, más folyamat rendszerhívása miatt ugyanaz a kernel eljárás újra elindulhat.)

Annak oka, hogy a kernel futása során az ütemezés nem preemptív, egyszerűen az, hogy az operációs rendszer számos olyan adatszerkezetet használ, amit nem lehet egy lépésben megváltoztatni. Ha egy ilyen adatszerkezet változtatása közben vennék el a futás jogát egy kernel módban futó folyamattól, az nem konzisztens állapotban hagyná az operációs rendszer adatszerkezeteit, vagy pedig tele kellene tűzdelni szinkronizációs műveletekkel a kölcsönös kizárás biztosítása érdekében.

Egy ilyen szituációt szemléltet az 5.5. ábra, ami egy kétirányban láncolt lista listaelemének lefűzését mutatja. Tegyük fel például, hogy a kernel a szabad memórialapok láncolt listájából szeretne egy elemet kivenni. A probléma abból adódik, hogy a listaelemet nem lehet egy lépésben lefűzni a láncból. A lefűzési lépések köztes állapotát (amikor a második elem lefűzése közben az egyik irány mutatói már megváltoztak) illusztrálja az 5.6. ábra. Ha a folyamattól el lehetne venni a vezérlést miközben az a listán dolgozik, a következő folyamat a lista végéről keresve eggyel kevesebb memórialapot látna, mint elölről indulva, vagyis a kernel adatai nem lennének konzisztens állapotban.



5.5. ábra. ábra - Láncolt lista egy közbülső elem lefűzése előtt

5.6. ábra. ábra - Láncolt lista egy közbülső elem lefűzése közben

5.3.3.3. Folyamatok ütemezési prioritása

A UNIX-ban a folyamatok prioritását 0-127 közötti egész számok jelölik. (Némelyik UNIX-változatban ettől eltérő prioritási tartományt használnak.) A nullás prioritás jelöli a legnagyobb prioritást, vagyis a kisebb prioritási érték nagyobb prioritáshoz tartozik.

A rendszer a felhasználói és a kernel módban futó folyamatok prioritását eltérő módon kezeli. A prioritási értékeket két prioritási tartományra osztja amint azt a következő, 5.7. ábra mutatja. Az 50 fölötti prioritási értékeket felhasználói módban futó folyamatokhoz rendeli a rendszer, míg az 50 alatti prioritások kernel módú folyamatokhoz tartoznak.

0	Legnagyobb prioritás	
.		} KERNEL-prioritások
.		
.		
49		
50		
.		
.		} FELHASZNÁLÓI prioritások
.		
127	Legkisebb prioritás	

5.8.ábra. táblázat - Az óramegszakításhoz kötődő ütemezési tevékenységek

Prioritás meghatározása kernel módban

A kernel módban futó folyamat prioritása statikus, nem függ attól, hogy a folyamat mennyit használta a CPU-t, vagyis mennyi ideig futott. A prioritás attól függ, hogy a folyamat milyen ok miatt hajtott végre *sleep* rendszerhívást, vagyis, hogy milyen eseményre várakozik. Emiatt a kernel prioritást szokták alvási prioritásnak is nevezni.

Jogos a kérdés, hogy mi alapján határozza meg a rendszer annak a kernel módban futó folyamatnak a prioritását, amelyik nem hajtott még végre *sleep* rendszerhívást mióta kernel módba került. A válasz nagyon egyszerű. A rendszer a folyamatok prioritását az ütemezés, vagyis az ütemező folyamat futása alkalmával vizsgálja. Mivel kernel módban a UNIX nem preemptív, ütemezésre csak akkor kerülhet sor, ha a futó folyamat végrehajt egy *sleep* rendszerhívást. A *sleep* rendszerhívás végrehajtása után viszont meghatározható a kernel módú prioritás.

Vizsgáljuk meg egy kicsit részletesebben is a kernel prioritás generálását. Hogyan kerülhet egy folyamat kernel módba? Két lehetőség van: vagy rendszerhívást hajt végre (vagyis implicit módon egy *trap* utasítást), vagy külső megszakítás kiszolgálása történik.

Ha rendszerhívás történik, a kernel módú futást felhasználói módú futás előzte meg, amikor is a folyamat prioritása adott. Ezt az értéket – mint majd látni fogjuk – a rendszer későbbi használatra elmenti. Ebben a pillanatban a folyamat kernel módú prioritása – mint láttuk – elvileg nem határozható meg, de ez nem okoz problémát.

Megszakítás esetén a rendszer nem ütemezi át a folyamatokat, a megszakítási rutin az éppen futó folyamat környezetében hajtódik végre. A megszakítási rutint az operációs rendszer úgy tekinti, mintha az éppen futó (vagyis a megszakított) folyamat futna tovább. (Ennek következtében minden megszakítási rutin által felhasznált idő a megszakított folyamat időkvótáját terheli.) Ütemezésre csak a megszakítás kiszolgálása után kerülhet sor, hiszen a megszakítási rutin kernel módban hajtódik végre. Ha a megszakított folyamat eredetileg is kernel módban futott, a rendszer nem tér vissza felhasználói módba, vagyis az ütemezőnek megint csak nincs lehetősége futni. Ha a megszakított folyamat felhasználói módban futott, akkor a visszatérés után a megszakított folyamat felhasználói módú prioritása él tovább. Láthatjuk tehát, hogy megszakítás kiszolgálása esetén, bár kernel módú futás történik, a rendszernek nincs szüksége kernel módú prioritás számolására. (Megjegyezzük, hogy az ütemezés kritikus szakaszaiban, amikor például éppen nincs futásra kijelölt folyamat és a környezet is indefinit, a megszakítások tiltva vannak.)

Folyamatok prioritásának meghatározása felhasználói módban

Felhasználói módban a prioritást egy adott pillanatban két dolog határozza meg:

- a felhasználó által adott külső prioritás *(nice szám, kedvezési szám)*,
- a folyamat korábbi CPU használata.

A *nice* szám a felhasználó által meghatározható érték, amivel kifejezheti, hogy mennyire fontos az általa indított folyamat. Minél kisebb egy folyamat *nice* száma, annál fontosabb a folyamat, vagyis annál nagyobb lesz a prioritása a futás során.

A prioritás számításához a UNIX a következő négy paramétert használja:

- *p_pri:* aktuális ütemezési prioritás,

- *p_usrpri:* felhasználói módban érvényes prioritás,
- *p_cpu:* a CPU használat mértékére vonatkozó szám,
- *p_nice:* a felhasználó által futás elején adott nice szám.

Az operációs rendszer a fenti paramétereket minden folyamat esetén külön-külön számon tartja.

Az ütemező a *p_pri* paraméterben tárolja a folyamat aktuális prioritását, tehát ez alapján választja ki, hogy melyik folyamatot ütemezze futásra. Amikor a folyamat felhasználói módban fut, akkor a *p_pri* megegyezik a *p_usrpri*vel. Kernel módba váltásnál a *p_pri* paraméter megkapja a kernel módban érvényes prioritás értékét, míg a *p_usrpri* értéke nem változik, sőt a rendszer továbbra is karbantartja a *p_usrpri* értékét az ütemezési algoritmus szerint. Amikor visszatér felhasználói módba, a rendszer *a p_pri* értékét a *p_usrpri*ből frissíti.

Amikor egy folyamat felébred egy *sleep* hívás után, akkor a kernel módú prioritása lesz érvényes, ami, mint láttuk, mindig magasabb, mint a felhasználói módú folyamatok prioritása. A rendszer így biztosítja a kernel módú folyamatok preferenciáját.

A *p_cpu* paraméter a folyamat CPU használatára jellemző érték. A folyamat *p_cpu* paramétere a folyamat indításakor nulla értékre áll. A folyamatok *p_cpu* értékének módosításai az óra-megszakításhoz kapcsolódnak a következők szerint (5.8. ábra):

- *p_cpu*t minden óramegszakítás alkalmával emeli a kiszolgáló rutin a futó folyamatnál:

p_cpu := p_cpu +1,

- ha több azonos felhasználói prioritású folyamat van a rendszerben az aktuálisan legmagasabb prioritási szinten, minden tizedik megszakításnál lefut a Round–Robin-algoritmus, vagyis az ütemező az azonos prioritású folyamatok között forgatja a végrehajtás jogát,
- minden századik megszakításnál az ütemező újraszámolja minden folyamat felhasználói prioritását a következő három lépésben:

– korrekciós faktor (KF) számítása:

KF: = 2*futásra kész folyamatok száma/(2* futásra kész folyamatok száma+1)

– minden folyamat CPU használatára jellemző változó kiszámítása:

p_cpu = p_cpu * KF

– felhasználói prioritás kiszámolása:

p_usrpri = P_USER+p_cpu/4+2*p_nice,

ahol P_USER egy alkalmas konstans.

A fenti képletben szereplő P_USER egy konstans. Arra szolgál, hogy a *p_usrpri* változó, vagyis a folyamat felhasználói prioritása, ne hagyja el a felhasználói prioritási tartományt. (A P_USER értéke a legkisebb felhasználói prioritás értékével egyezik meg, vagyis példánkban P_USER = 50.)

	Futó folyamat	Minden folyamat
Minden óramegszakítás	p_cpu := p_cpu +1	Megvizsgálja, van-e a futónál magasabb prioritású folyamat. Ha van, újraütemez.
Minden 10. óramegszakítás	Round-Robin algoritmus: ha több azonos prioritás-osztályú folyamat van a legmagasabb prioritású pozícióban, 10 óra-megszakításonként váltja a futó folyamatot	
Minden 100. óramegszakítás		korrekciós faktor számítása p_cpu = p_cpu * KF p_usrpri = P_USER+p_cpu/4+2*p_nice

5.10.ábra. táblázat - Ütemezési példa

A KF korrekciós faktor számításával a rendszer terhelését igyekszik figyelembe venni az ütemező. Ha megvizsgáljuk a korrekciós faktor viselkedését, láthatjuk, hogy a korrekciós faktor értéke annál inkább egyhez tart, minél több futásra kész folyamat volt a rendszerben az elmúlt ütemezési periódusban (elmúlt 100 óramegszakítás ideje alatt), vagyis minél nagyobb volt a rendszer terheltsége: 1 futásra kész folyamatnál ez a szám 2/3; 2 folyamatnál 4/5, 10 folyamatnál 10/11 stb. Megjegyezzük, hogy számos UNIX-rendszer a korrekciós faktor értékét konstansnak (1) veszi.

Nagyon fontos kiemelni, hogy az ütemezési algoritmus, illetve a hozzá kötődő tevékenységek nem közvetlenül a megszakítási rutinban hajtódnak végre, hiszen ekkor feleslegesen hosszú időt töltene a rendszer a megszakítás kiszolgálásával. Az ütemezéshez kötődő rutinokat – hasonlóan más ciklikusan végrehajtandó rendszerfeladatokhoz – a call-out mechanizmus segítségével hajtja végre a rendszer. Erről a későbbiekben még részletesen szó lesz.

Másik ide tartozó megjegyzés, hogy a fent említett, óra-megszakításokhoz kötődő három tevékenység gyakorisága változhat az egyes UNIX-rendszerekben. A fent szereplő 10-es, illetve 100-as számok tipikus értékek.

5.3.3.4. Környezetváltás ütemezéskor

Az ütemezési algoritmus ismeretében tanulságos végignézni, hogy milyen esetekben történik környezetváltás a UNIX-ban, vagyis milyen események hatására kerül át a CPU-használat joga – az ütemező közreműködésével – az egyik folyamattól a másik folyamathoz. Környezetváltás a következő esetekben történik:

- Nem preemptív ütemezés

– Egy folyamatnak várnia kell valamilyen eseményre (sleep rendszerhívást hajt végre).

– Egy folyamat befejeződik (exit rendszerhívást hajt végre).

- Preemptív ütemezés

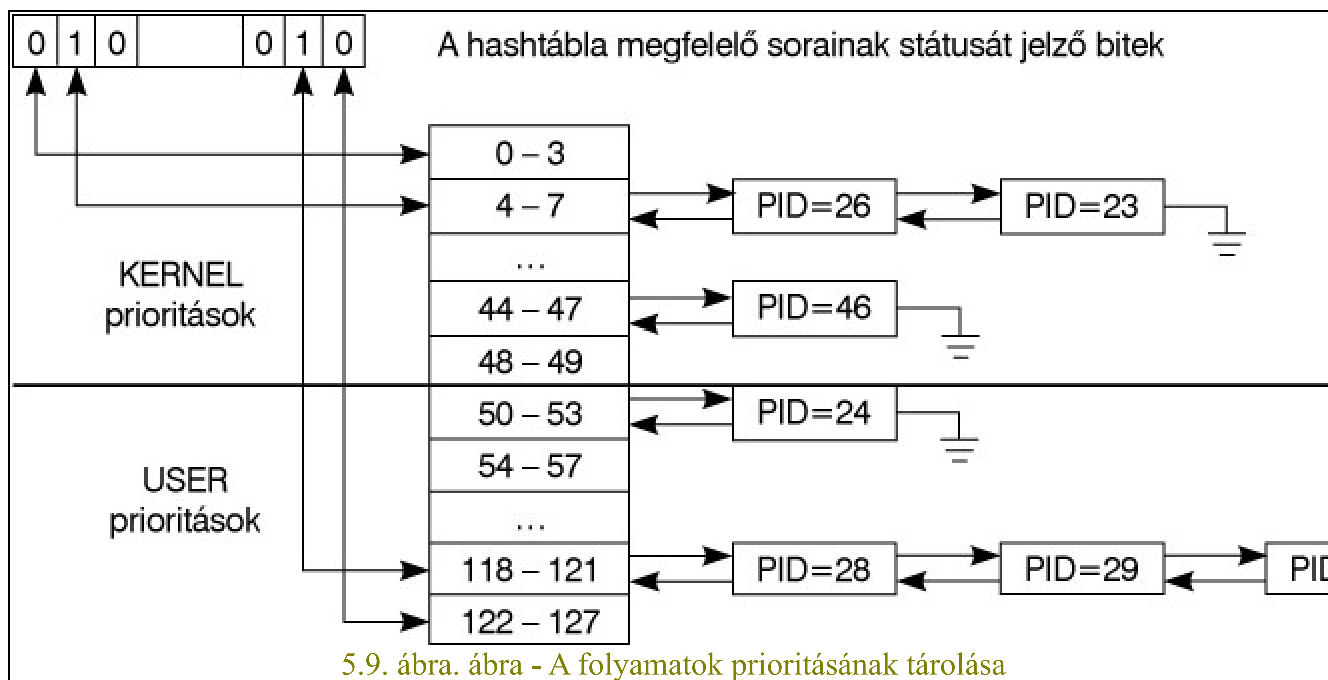
– A 100-adik óraciklusban a prioritások újraszámításakor az egyik folyamat prioritása nagyobb lesz, mint a futó folyamat prioritása. Ekkor prioritásvizsgálat után új folyamatot futtat a rendszer, tehát környezetváltás szükséges.

– A 10-edik óraciklus esetén a Round–Robin-algoritmus egy másik, azonos prioritású folyamatot választ futásra.

– Egy futó folyamat, vagy megszakítási rutin működésének eredményeképpen felébred (ready to run állapotba jut) egy, az aktuálisan futónál magasabb prioritású, eddig várakozó folyamat.

5.3.3.5. Adatszerkezetek a folyamatok prioritásának tárolására

Az időben változó prioritások tárolására a rendszer dinamikus adatszerkezetet használ. Az azonos prioritású, futásra kész folyamatok láncolt listán tárolódnak. Az adott prioritású folyamatok keresésének megkönnyítése érdekében a listafejeket egy *hashtáblázatban* tárolja a UNIX. Egy listafejhez négy egymás után következő prioritásértékkel rendelkező folyamat tartozik. A hashtáblás ábrázolást mutatja be az 5.9. ábra.



5.9. ábra. ábra - A folyamatok prioritásának tárolása

A fenti megoldás nemcsak azért előnyös, mert csökkenti a hashtábla bejegyzéseinek számát, hanem lehetővé teszi adott prioritású folyamatok létezésének ellenőrzését is. A rendszer minden sorhoz hozzárendel egy jelzőbitet. A jelzőbit értéke mutatja, hogy az adott sorban létezik-e futásra kész folyamat. Ezeket a biteket a listába történő beszúrás, illetve lefűzés alkalmával frissíti a rendszer. Az ellenőrzés gyorsítása azért fontos, mert nagy gyakorisággal (minden óramegszakítás esetén) lefut, tehát ennek a tevékenységnek az ideje a rendszer teljesítményét nagymértékben befolyásolja.

A négy prioritási szint egyetlen hashtábla-bejegyzésbe történő összevonása azért előnyös, mert így pontosan 32 jelzőbit lesz, amit egy memóriaszóba összevonva egyszerű kezelni a 32 bites rendszerekben. (A jelzőbiteket tartalmazó változó neve: *whichqs*.)

5.3.3.6. Példa az ütemezés számolására

A következőkben egy konkrét példán keresztül mutatjuk be a prioritás számolását. Legyen a korrekciós tényező értéke $^{1}/_{2}$, és ne változzon az ütemezés alatt. (Ez egyébként több UNIX-rendszerben is hasonlóan van.)

Tekintsünk el a Round–Robin-algoritmus használatától. Válasszuk a *nice számot* minden folyamatnál 0-nak.

Esetünkben tehát az ütemező algoritmus tevékenységei:

- minden óra-megszakításnál emeli a futó folyamat p_cpu értékét:

p_cpu := p_cpu +1

- minden 100-adik óra-megszakításnál újraszámolja minden folyamat felhasználói prioritását a következő két lépésben:

kiszámítja a minden folyamat CPU használatára jellemző változó, (p_cpu) értékét:

p_cpu = p_cpu * KF = p_cpu * 1

kiszámítja a felhasználói prioritás értékét:

p_usrpri = P_USER+p_cpu/4+2*p_nice = 50 + p_cpu/4

Minden folyamatnak négy, ütemezésnél használt paraméterét (*p_pri*, *p_usrpri*, *p_cpu*, *p_nice*) tartja számon a rendszer, azonban esetünkben csak kettő (*p_pri*, *p_cpu*) érdekes, mivel a *p_nice* szám zéró, a *p_pri* értéke pedig megegyezik a *p_usrpri* értékével, hiszen csak felhasználói módú futásról van szó. A számításokat az 5.10. ábrán látható táblázatban foglaltuk össze.

Láthatjuk, hogy mindegyik folyamat futási lehetőséghez jutott a vizsgált periódus alatt. A várakozó folyamatok prioritása a várakozási idő növekedésével folyamatosan növekedett, míg el nem érték a lehető legmagasabb prioritási szintet. A futó folyamat prioritása a futási idővel arányosan csökkent. Az ütemező így előzi meg a folyamatok éheztetését.

5.3.3.7. A UNIX-ütemezés értékelése

A UNIX ütemezése viszonylag jó rendszerkihasználást és jó áteresztőképességet biztosít a felhasználók számára átlagos, vagyis nem szélsőséges rendszerterhelés esetén. Az utóbbi években azonban a felhasználók igényei megemelkedtek az ütemezővel szemben, nem utolsósorban azért, mert megjelentek a UNIX-nál nagyobb hatékonyságot és több lehetőséget biztosító rendszerek.

A		B		C			
p_pri	p_cpu	p_pri	p_cpu	p_pri	p_cpu	Lépés	Futó folyamat

50	0	50	0	50	0	1	A
50	1	50	0	50	0	2	A
…	…	…	…	…	…	…	A
50	99	50	0	50	0	99	A
50+50/4 63	100/2 50	50	0	50	0	100	A
63	50	50	1	50	0	101	B
…	…	….	…	…	…	…	B
63	50	50	99	50	0	199	B
50+25/4 56	50/2 25	50+50/4 63	100/2 50	50	0	200	B
56	25	63	50	50	1	201	C
…	…	….	…	…	…	…	C
56	25	63	50	50	100	299	C
50+13/4 53	25/2 13	50+25/4 56	50/2 25	50+50/4 63	100/2 50	300	C

5.13.ábra. táblázat - Jelzések

Az ütemezés értékelésénél így elsősorban az algoritmus negatívumait emeljük ki:

- Nem méretezhető megfelelően. Az alkalmazott algoritmus a folyamatok számának emelkedése esetén nem tud rugalmasan alkalmazkodni, vagyis változni a terhelés növekedésével. A korrekciós faktor használata nem elég hatékony eszköz.
- Az algoritmussal nem lehet meghatározott CPU-időt allokálni egy adott folyamat, illetve a folyamatok egy csoportja számára, vagyis nem lehet a CPU-t adott esetben „kiosztani”.
- Nem lehet a folyamatok fix válaszidejét garantálni. Bár az algoritmus igyekszik minél alacsonyabb válaszidőt biztosítani, de nagy rendszerterhelés esetén ez limit nélkül megnőhet, vagyis nem lehet garantált válaszidőről beszélni. Emiatt a UNIX-ütemezést nem lehet valósidejű (real-time) rendszerben alkalmazni. Ezt a hiányosságot számos rendszerben (SVR4, Digital UNIX stb.) megpróbálták kiküszöbölni.
- Az előző hiányossággal rokon probléma, de érdemes külön is kiemelni, hogy ha egy kernel rutin sokáig fut, az feltartja az egész rendszert, mivel a kernel nem preemptív.

- A felhasználó a folyamatainak prioritását nem tudja megfelelő módon befolyásolni. A gyakorlatban a nice szám nem elég hatásos eszköz erre a célra.

5.3.3.8. Call-out

A *call-out* mechanizmus arra szolgál, hogy egy adott tevékenységet (függvényt) a kernel egy későbbi időpontban hajtson végre (hívjon meg). A *call-out* függvények adott időben történő meghívását a *timeout* rendszerhívással lehet megadni, illetve az *untimeout* rendszerhívással lehet törölni.

Fontos megjegyezni, hogy a *call-out* függvények rendszerkörnyezetben futnak, így nem aludhatnak, illetve nem érhetik el a folyamatok környezetét.

A *call-out* függvények periodikusan ismétlődő feladatok (tipikusan kernel-funkciók) végrehajtására használhatók, például:

- hálózati csomagok ismételt elküldésére,
- ütemezési és memóriakezelő függvények hívására,
- készülékek monitorozására (nehogy megszakítások maszkolása miatt elvesszünk bemeneti adatokat),
- olyan készülékek lekérdezésére, amelyek nem megszakításkéréssel működnek.

A *call-out*-ok meghívását az óra-megszakítás végzi, azonban mivel a *call-out* függvények kernel rutinok, futásuk alatt a megszakítások fogadása nem lehet letiltva, hiszen ekkor hosszú ideig nem fogadná a rendszer az óramegszakításnál alacsonyabb prioritású megszakításokat. Emiatt a megszakításkezelő nem közvetlenül hívja meg a *call-out* függvényeket, hanem csak ellenőrzi, hogy elérkezett-e az ideje valamelyik *call-out* függvény meghívásának. Ha igen, beállít egy erre a célra rendszeresített jelzőbitet. A kernel ezt a jelzőbitet minden esetben ellenőrzi, amikor a megszakítások befejezése után visszatér a megszakított tevékenységéhez, vagyis feloldaná a legkisebb prioritású megszakítás tiltását. Így biztosított, hogy a kernel a *call-out-okat* a lehető leghamarabb, de az összes megszakítás kiszolgálása után meghívja.

A *call-out* függvények meghívásának adminisztrálására a rendszer különböző adatszerkezeteket használhat. Mindegyik adatszerkezet dinamikus, hiszen a *call-out* függvények száma futás közben változik, és nem lehet tudni, hány *call-out* függvény lesz egy adott rendszerben. Az adatszerkezetek megválasztásakor ismét az óra-megszakításkor végrehajtandó műveletek gyorsítása a cél. Két *call-out* listakezelési megoldást ismertetünk:

- a láncolt listás és
- az időkerekes

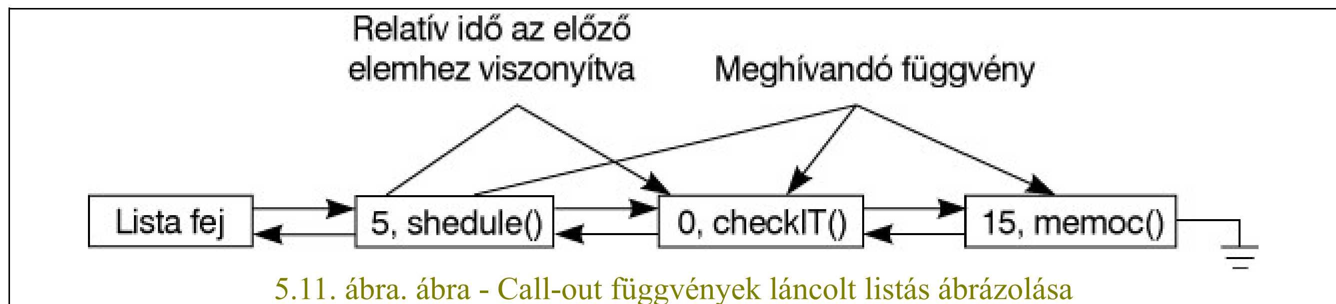
Call-out függvények láncolt listás ábrázolása

Ebben az esetben a *call-out* függvények listája egy rendezett lista, amely a *call-out* függvényeket „tüzelési sorrendben” (meghívási sorrendben) tartalmazza. Az időt a rendszer óra-megszakításokban számolja. Ezt az időegységet hívjuk tikknek.

A listaelemekben a láncolt listás tárolás esetén *relatív időpontok* vannak tárolva. A rendszer az egyes függvények meghívásának idejét az előző elem tüzelési idejéhez képest relatív számként tárolja, vagyis a rendszernek a lista előző eleméhez képest az adott listaelemben tárolt számú tikkel később kell aktiválni a listaelemben megnevezett függvényt.

A kernel minden egyes óramegszakítás esetén eggyel csökkenti az első listaelem időpont-számlálóját, és amikor az eléri a 0-át, akkor aktivizálja az első listaelemben tárolt függvényt, illetve mögötte az összes 0 időt tartalmazó elemet.

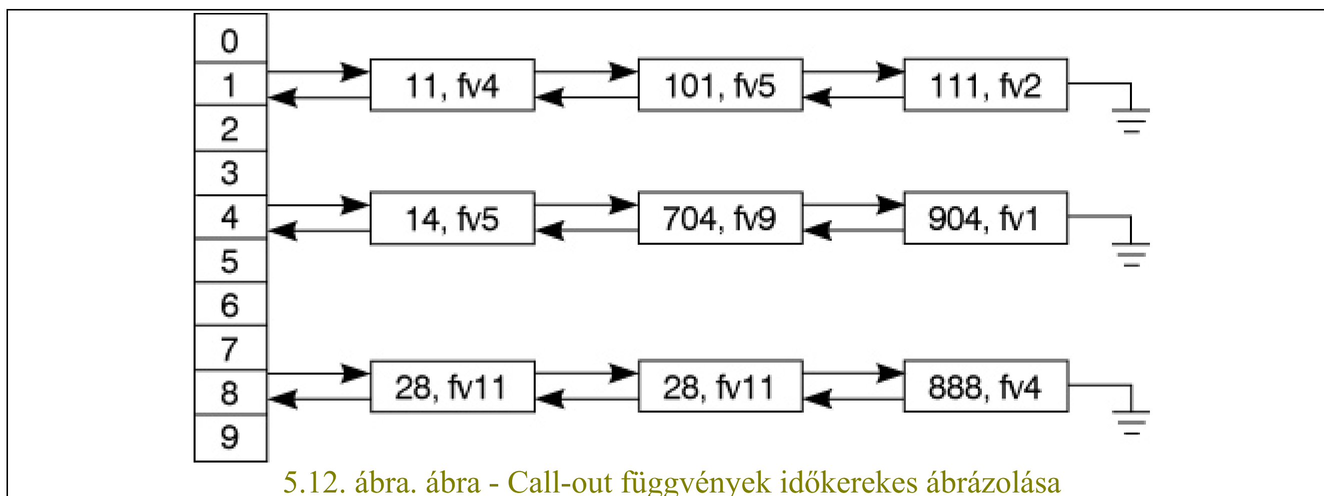
A megoldás előnye az egyszerű kezelhetőség, viszont hátránya, hogy a lista igen hosszú lehet, ami időigényessé teheti a lista kezelését, például a listába való beszúrást (5.11.ábra).



5.11. ábra. ábra - Call-out függvények láncolt listás ábrázolása

Call-out függvények tárolása időkerékkel

Az időkerekes ábrázolás esetén a rendszer lényegében felszabdalja a listát nagyjából egyenlő részlistákra. A listaelemekben az *abszolút* meghívási idő van tárolva tikkekben mérve. A függvények meghívási ideje alapján egy hashtáblát készít a rendszer. Ha mondjuk egy 10 elemű hashtábla van használatban, akkor az első listában azok a *call-out* függvények lesznek, amelyek meghívási ideje 10-zel osztva 1-et ad maradékul. Így átlagosan egy lista hossza az előző tárolási móddal összevetve a tizedére csökken. Az egyes listák rendezettek, az elsőként meghívandó függvény áll a lista első helyén. Az időkerekes ábrázolást láthatjuk az 5.12 ábrán.



5.12. ábra. ábra - Call-out függvények időkerekes ábrázolása

Az időkerék kezelése a következő módon történik: A rendszer megvizsgálja, hogy melyik részlistában tárolódnak az adott időpillanathoz tartozó *call-out* függvények. (Megnézi, 10-zel osztva mennyi maradékot ad az aktuális óra tikk-sorszáma.) Összehasonlítja az első listaelem tüzelési idejét az aktuális idővel. Ha a két érték megegyezik, meghívja a hozzá tartozó rutint, majd a lista esetleges további, azonos tüzelési idejű függvényeit. Ha a tüzelési idő nagyobb, nem tesz semmit. A nevét a módszer onnan kapta, hogy a hashtáblán a rendszer ellenőrzéskor körbe-körbe jár.

5.3.4. Szinkronizáció

A **szinkronizáció** *(syncronization)* egy folyamat végrehajtásának olyan időbeli korlátozása, ahol ez egy másik folyamat futásától, esetleg egy külső esemény bekövetkezésétől függ. Alapeseteit a korábbiakban tárgyaltuk.

A UNIX operációs rendszer alatt a szinkronizáció legegyszerűbb eszközei a **jelzések** *(signals)*.

5.3.4.1. UNIX-jelzések

A jelzések elsődleges célja a UNIX-rendszerekben az, hogy a folyamatok különböző (rendszer- vagy alkalmazás szintű) események bekövetkezéséről értesüljenek. A jelzések részletes működése és rendszer szintű megvalósítása az egyes UNIX-változatokban lényeges eltéréseket mutat. Az eredeti System V jelzés implementáció alapvetően megbízhatatlan és hibás volt. A BSD UNIX-variációk ezért egy robusztusabb, megbízhatóbb jelzésmechanizmust dolgoztak ki, amely azonban nem kompatíbilis a System V jelzésekkel. Ezt a problémát a POSIX.1-szabvány (IEEE 90) próbálta meg feloldani egy szabványos interfész definiálásával. A System V Release 4 (SVR4) UNIX-variáns már ennek megfelelően, a BSD-jelzések bizonyos tulajdonságait is ötvözve tartalmazza a jelzéskezelést. A mai modern UNIX-rendszerek (Solaris, AIX, HP-UX, Digital UNIX stb.) a POSIX.1 szabványnak megfelelő jelzéskezelő rendszert tartalmaznak.

A UNIX-jelzések arra kínálnak módot, hogy események egy meghatározott halmazának bekövetkezése esetén a futó folyamat egy eljárása meghívásra kerüljön. Az eseményeket egész számokkal reprezentálják a rendszerek, és szimbolikus konstansokkal hivatkoznak rájuk. Az eredeti System V megvalósítás 15 jelzést tartalmazott, a mai rendszerek általában 31 jelzéssel dolgoznak.

Az 5.13. ábra néhány tipikus jelzést ismertet.

A jelzésrendszerben két lépés különíthető el: a jelzések keletkezése és kezelése. A következőkben ezeket a lépéseket ismertetjük.

A jelzés	Leírása
SIGABRT	A folyamat megszakítása
SIGTERM	A folyamat leállítása
SIGSEGV	Szegmentációs hiba
SIGALRM	Valósidejű órariasztás
SIGBUS	Buszhiba
SIGPIPE	Olvasó nélküli csôvezetékbe írás

5.14.ábra. táblázat - Tipikus UNIX-jelzések és értelmezésük

5.3.4.2. Jelzések keltése

Sokféle esemény válthat ki jelzéseket a futó folyamat környezetében: a futó folyamat maga, más folyamatok, az operációs rendszer, külső események, megszakítások. A jelzések forrásai a következő főbb kategóriákba csoportosíthatók.

- **Kivételek (exception).** A kivételek (például illegális utasítás végrehajtásának kísérlete) bekövetkezése esetén a kernel értesíti a folyamatot egy jelzés elküldésével.
- **Más folyamatok.** Egy folyamat jelzést küldhet egy más folyamat, vagy folyamatok csoportja számára az erre szolgáló rendszerhívások (*kill()*, illetve *sigsend()*) segítségével. A folyamat önmaga számára is küldhet jelzést.
- **Terminálmegszakítások.** A terminál bizonyos karaktereinek (mint például a CTRL+C) leütése a terminálhoz csatlakozó, előtérben futó folyamat számára jelzést generál.
- **Munkamenet-kezelés (job control).** Az ilyen képességgel rendelkező parancsértelmezők (shellek) az előtérben, illetve háttérben futó folyamatokat jelzések segítségével manipulálják, illetve a kernel egy folyamat megszűnése esetén ilyen jelzéssel értesíti a szülő folyamatát.
- **Kvótajelzések.** Amikor egy folyamat eléri a számára rendelkezésre bocsátott CPU-használat vagy fájlméret határait, a kernel jelzést küld számára.
- **Értesítések.** Egy folyamat bizonyos események (például egy periféria készen áll az adatátvitelre) bekövetkezéséről értesítést kérhet a kerneltől.
- **Riasztások.** Egy folyamat beállíthat riasztást egy adott időhosszra, amely letelte után, a kernel jelzést küld a folyamat számára.

5.3.4.3. Jelzések kezelése

Egy folyamat számára a hozzá beérkező jelzések aszinkron események, azaz futása során bármikor beérkezhetnek. A jelzés létrejötte után a második fázis a folyamatok értesítése és a jelzések kezelése. A jelzés akkor számít kézbesítettnek, ha az a folyamat, amelynek a jelzés szól, tudomásul veszi a megérkezését, és normális futását megszakítva valamilyen meghatározott akciót hajt végre.

Minden jelzéshez tartozik egy alapértelmezett akció *(default action)*, amit a kernel hajt végre, amennyiben a folyamat nem határozott meg más akciót. Öt lehetséges alapértelmezett akció létezik.

- **Megszakítás (abortálás).** A folyamat megszakítását eredményezi egy futási lenyomat *(core dump)* generálása után. A lenyomat egy **core** nevű fájlban keletkezik a folyamat aktuális könyvtárában. Tartalmazza a folyamathoz tartozó tárterület tartalmát (memóriakép) és a processzor regisztereinek pillanatnyi értékét (regiszterkép). Ez a lenyomat felhasználható a későbbiekben a megszakítási helyzet analízisére.
- **Kilépés.** A program leállítása **core** fájl generálása nélkül.
- **Figyelmen kívül hagyás (ignore).** A jelzést figyelmen kívül hagyja a kernel (folyamat).
- **Felfüggesztés.** A folyamat felfüggesztődik.
- **Folytatás**. A folyamat folytatódik, ha fel volt függesztve, egyébként figyelmen kívül hagyja a jelzést.

Az 5.14. ábra a korábbiban felsorolt jelzésekre adja meg az alapértelmezett akciókat.

A jelzés	Leírása	Alapértelmezett akció
SIGABRT	A folyamat megszakítása	megszakítás
SIGTERM	A folyamat leállítása	kilépés
SIGSEGV	Szegmentációs hiba	megszakítás
SIGALRM	Valósidejű órariasztás	kilépés
SIGBUS	Buszhiba	megszakítás
SIGPIPE	Olvasó nélküli csővezetékbe írás	kilépés

A folyamatok futásuk során bármikor felülírhatják ezeket az alapértelmezett akciókat a jelzések többségére. Az így meghatározott akció lehet a jelzés figyelmen kívül hagyása, vagy egy, a folyamat által meghatározott, ún. **jelzéskezelő (handler)** eljárás meghívása. A folyamat bármikor visszaállíthatja a jelzéshez tartozó alapértelmezett akciót is. Bizonyos jelzések esetében, mint például a *SIGKILL* vagy a *SIGSTOP* az alapértelmezett akciók felülírása nem lehetséges.

Fontos megjegyezni, hogy a jelzésekhez tartozó akciók a címzett folyamat környezetében hajtódnak végre, azaz csak akkor, ha a hozzájuk tartozó folyamatok futó állapotba kerülnek. Bizonyos esetekben ez lényeges késleltetést okozhat a jelzések kezelésében.

A jelzést fogadó folyamat tudomására akkor jut egy jelzés érkezése, ha a kernel meghívja az erre a célra szolgáló *issig()* rendszerhívást. Ez a következő esetekben fordul elő:

- visszatéréskor felhasználói futási szintre egy rendszerhívásból vagy megszakításból,
- egy megszakítható eseményre történő várakozás előtt, és
- közvetlenül egy nem megszakítható eseményre történő várakozás után.

Látható, hogy a jelzések kezelése szempontjából a folyamat várakozó (alvó) állapotai közül a nem megszakítható eseményre várakozás különleges jelentőséggel bír. Az ilyen típusú várakozás esetén a kernel a jelzést várakozó állapotban tartja mindaddig, míg a folyamat vissza nem tér (fel nem ébred) ebből az állapotából. Megszakítható várakozások (mint például a terminál bevitelre történő várakozás) esetén a kernel a jelzése beérkezésének hatására a folyamatot felébreszti.

Ha az *issig()* hívás *IGAZ* értékkel tér vissza, a kernel a *psig()* hívás segítségével kezeli a jelzést. A *psig()* vagy végrehajtja a kernel által meghatározott alapértelmezett akciót, vagy a *sendsig()* hívás segítségével meghívja a folyamatkezelő függvényét. A kezelő függvény meghívása előtt a folyamat kilép a kernel futási módból. A *psig()* a verem és a folyamat kontextusának manipulálásával arról is gondoskodik, hogy a folyamat a jelzés kezelése után normálisan folytatható legyen.

5.3.4.4. Megbízhatatlan jelzések

Az eredeti Ssytem V jelzésrendszer több hibát is tartalmaz, ezért *megbízhatatlan* jelzővel illetik. Egyik tipikus problémája, hogy a kernel a jelzés kezelésekor minden alkalommal visszaállítja az alapértelmezett akciót, így a folyamat jelzéskezelő eljárásának gondoskodnia kell az egyedi kezelés ismételt beállításáról. Ez versenyhelyzetet teremt a jelzések keletkezése és az egyedi akciók beállítása között, ami egy-egy jelzés esetében könnyen az alapértelmezett akció végrehajtását eredményezheti.

Egy másik tipikus probléma a jelzéseket leíró tábla elhelyezéséből adódik. A táblát a kernel ugyanis a folyamathoz kötött *u*-területen tárolja, amihez csak az aktuális folyamatnál fér hozzá. Ily módon egy nem megszakítható várakozásban lévő folyamat esetén a kernel nem tudja eldönteni, hogy a folyamat maga kezeli-e a jelzést, vagy az alapértelmezett akciót kell végrehajtania. Ez csak a folyamat felébresztése után derülhet ki.

Végezetül hiányzott a jelzések kézbesítésének átmeneti blokkolása, illetve az olyan munkamenetek kezelése, ahol folyamatok csoportosan kezelhetőek a terminál-hozzáférés során.

5.3.4.5. Megbízható jelzések

Az előbbi problémákra a BSD 4.2-es verziója egy új jelzésrendszer bevezetésével hozott megoldást, illetve az AT&T SVR3 verziójában található *megbízható* jelzések is ebben az irányban változtak. A két megoldás ugyanakkor nem volt kompatíbilis, mindkettő más és más rendszerhívások bevezetésével orvosolta a problémákat.

A POSIX.1 szabvány teremtett rendet egy egységes interfész bevezetésével, melynek implementációs részleteit nem rögzítették, így mindkét UNIX-variáns beépíthette. Az SVR4 UNIX egy új, POSIX.1-nek megfelelő rendszert tartalmazott, amely kompatíbilis volt a korábbi BSD és SVR3 verziókkal is.

A megbízható jelzéskezelő rendszerek mindegyike a következő közös szolgáltatásokat nyújtja:

- **Perzisztens kezelők (persistent handlers).** A jelzéskezelő eljárás végrehajtásának beállítása a jelzés kezelése után is megmarad, így nincs szükség újbóli beállítására.
- **Maszkolás (masking).** Egy jelzés átmenetileg maszkolható, avagy blokkolható. Az ilyen jelzések keletkezését a kernel megjegyzi, de nem értesíti a hozzájuk rendelt folyamatokat. Ez lehetővé teszi a folyamatok kritikus részeinek védelmét a jelzések megszakításaival szemben.
- **Alvó folyamatok (sleeping processes).** A folyamatokhoz tartozó jelzéseket leíró információk egy része akkor is látható a kernel számára, ha a folyamat éppen nem fut (az *u*-terület helyett a *proc*-területen tárolódik), ily módon a kernel a folyamat felébresztése nélkül ellenőrizheti, hogy az megadott-e egyedi akciót, vagy az alapértelmezett jelzéskezelő akciót kell végrehajtani.
- **Felszabadítás és várakozás (unblock and wait).** A megbízható jelzések mechanizmusa egy speciális rendszerhívás (*sigpause()*) segítségével képes megoldani azt, hogy egy folyamat felszabadítsa a blokkolt jelzést, majd azonnal várakozó állapotba kerüljön a jelzés megérkezéséig.

5.3.4.6. Az SVR3 implementáció

Az AT&T által kidolgozott implementáció minden lényeges tulajdonsággal rendelkezett a megbízható jelzéskezelés kidolgozásához, azonban megvalósításának voltak sajátos hátrányai. Legfontosabb hiányossága, hogy egyszerre nem képes több jelzést blokkolni, így nehéz több jelzést is kezelő kritikus régiók megírása (ahol a folyamatot nem szakíthatják félbe jelzések). Az SVR3-ból hiányzott a munkafolyamat-kezelés is. Mindezeket a hiányosságokat a 4BSD rendszere pótolta.

5.3.4.7. BSD jelzésmenedzsment

A 4.2BSD volt az első megbízható jelzésrendszerrel ellátott UNIX-variáns. A rendszerhívások többsége tartalmaz egy jelzésmaszk paramétert, így egy hívás egyszerre több jelzésen is végrehajthat egy művelet. A jelzéseket blokkoló *sigsetmask()* függvényhívásnak egyszerre több jelzést is megadhatunk.

A jelzések kezelésére szolgáló eljárásokhoz megadásukkor jelzésmaszkot rendelhetünk, így végrehajtásukkor a kernel gondoskodik a maszkolt jelzések helyes és automatikus beállításáról. Ily módon egy jelzés második példányát a rendszer nem fogja kezelni addig, míg az első kezelését be nem fejezte.

A BSD implementáció további újdonsága, hogy a jelzéseket különálló vermet használva is kezelheti a folyamat. Így lehetővé válik például a verem túlcsordulását jelző jelzés kezelése is.

Ez az implementáció további jelzéseket is bevezetett, elsősorban a munkafolyamat-kezelés területén. A munkafolyamat összefüggő folyamatok olyan csoportja, amelyek általában egy csővezetéket formálva helyezkednek el (standard be-, illetve kimeneteiket összekötve). Egy terminálhoz tartozó több folyamat esetén csak egy rendelkezhet a terminál írásának és olvasásának jogával (előtérben futó folyamat), a többiek a háttérben futnak. A háttérben futó folyamatok jelzéseket kapnak, amikor megpróbálják elérni a terminált, és általában felfüggesztődnek. A felhasználó parancsértelmezője jelzések küldésével képes a folyamatokat előtérbe vagy háttérbe helyezni, felfüggeszteni és továbbindítani a futásukat.

Végezetül a 4BSD UNIX képes a jelzések által félbeszakított, hosszú ideig futó rendszerhívások automatikus újrahívására. Ilyen rendszerhívások lehetnek a karakteres eszközök írás és olvasás műveletei, a hálózati kapcsolatokat és csővezetékeket kezelő műveletek és a várakozás jellegűek (például *wait()*, *waitpid()*, és *ioctl()*). Amikor egy ilyen hívást félbeszakít egy jelzés, a jelzés lekezelését követően a rendszerhívás automatikusan újra meghívódik (enélkül *EINTR* hibával térne vissza).

A BSD jelzésinterfész hatékony és flexibilis. Legfontosabb hátránya, hogy nem kompatíbilis az eredeti AT&T interfésszel. Ez azt eredményezte, hogy a szoftvergyártók maguk kezdtek olyan közös interfészeket kifejleszteni, amelyek mindkét (BSD, SysV) UNIX-variánssal használhatóak. Ezt a problémát oldotta meg az egységes POSIX.1 jelzésinterfész, amelyet az SVR4 (System V Release 4) vezetett be.

5.3.4.8. Az SVR4 jelzések

A UNIX Systems Laboratories *UNIX SVR4.2 Operációs rendszer API Interfésze* által bevezetett jelzéskezelő rendszerhívások a BSD és korábbi SVR3 variánsok és a nem megbízható jelzések teljes halmazát lefedik. A régi *signal(), sigset(), sighold(), sigelse(), sigignore() és sigpause()* rendszerhívások mellett új, a BSD javított jelzéskezelését megvalósító rendszerhívásokat vezettek be. A jelzésekkel kapcsolatos állapotinformációk kezelését a BSD megoldásnál látható módon oldották meg, kisebb változó- és függvénynév módosításokkal.

A kernel a folyamatok felébresztése nélkül képes ellenőrizni, hogy azok foglalkoznak-e a jelzéssel. Ha igen, akkor egy bitsorozat *(p_cursig)* megfelelő elemét bebillentve a kernel jelzi a folyamat számára a jelzés megérkezését. (Ily módon egy jelzésből egyszerre maximum egy kezeletlen lehet.) Ha a folyamat megszakítható várakozásban van és a jelzés nem blokkolt, akkor a kernel felébreszti a folyamatot.

A folyamat a BSD-megoldásnál megismert *issig()* rendszerhívással ellenőrzi a jelzések meglétét (a *p_cursig* bitjeinek tesztjével). A kernel itt is a *psig()* rendszerhívással kezdeményezi a jelzések kezelését.

5.3.4.9. Kivételkezelés

Kivételen azt értjük, amikor egy program egy nem megszokott szituációval, tipikusan valamilyen hibával találja magát szemben. Ezek valamilyen hardver eszköz (többnyire a processzor) által generált megszakítások, mint például a hibás címzés, nullával való osztás stb. Ez általában hibamegszakítást okoz, amelynek kiszolgálásakor a kernel jelzések segítségével értesíti a programot a bekövetkezett kivételről.

A jelzés típusa a kivétel természetétől függ. Például egy hibás címzés *SIGSEGV* (szegmentációs hiba) jelzést eredményez. Amennyiben a folyamat meghatározott egy kezelő függvényt a jelzéshez, úgy a kernel azt hívja meg. (Ezt a mechanizmust használják a nyomkövetők is a program futásának befolyásolására).

A UNIX kivételkezelésnek – többek között – két nagy hátránya van:

- a kivételkezelés a kivétellel megegyező kontextusban fut; a kernel ugyan átadja a kivétel bekövetkezésekor aktuális kontextus bizonyos részét a kivételkezelőnek, de ennek pontos tartalma UNIX-implementációnként változhat,
- a jelzéseket alapvetően egyszálú folyamatok számára találták ki; többszálú végrehajtást is támogató UNIX-variánsok (például Solaris 7) csak körülményesen alkalmazhatják a jelzéseket.

5.3.4.10. Folyamatcsoportok és terminálkezelés

A UNIX a korábban említett folyamatcsoportokat használja a terminál hozzáférés és a felhasználói viszony kezelésére. (Felhasználói viszonyon azt értjük, amikor a felhasználó interaktív kapcsolatot létesít az operációs rendszerrel.) Minden folyamathoz tartozik egy **folyamatcsoport azonosító (process group ID)**, melyet a kernel a folyamatok teljes csoportjára vonatkozó akciók végrehajtásában használ fel. A csoportoknak lehet egy **csoportvezetője**, melynek processzazonosítója (PID) megegyezik a folyamatcsoport azonosítóval. Normális esetben a folyamatok szülőjüktől öröklik csoportazonosítójukat.

Az egy csoportba tartozó folyamatokhoz egy vezérlő terminál tartozik, általában a felhasználó belépésének terminálja, ahonnan a folyamatokat elindította. Ehhez a terminálhoz egy speciális eszköz, a */dev/tty* rendelődik (az eszköz neve kiegészülhet a terminált azonosító jellel, például */dev/tty01*).

A **munkamenet-kezelés** az a mechanizmus, mely egy folyamatcsoport futását felfüggesztheti, vagy továbbengedheti, és meghatározza a csoport kapcsolatát a terminálhoz. Munkamenet-kezelésre alkalmas *shellek*, mint például a *csh* vagy *ksh* speciális vezérlő karaktereket (például *CTRL+Z*) és parancsokat (például *fg* és *bg*) használnak ezen szolgáltatások elérésére.

A kezdeti System V implementációk a folyamatcsoportokat általában a felhasználókhoz kötött folyamatok reprezentálására használták, és nem tartalmaztak munkamenet-kezelést. A 4BSD minden parancssorhoz új folyamatcsoportot vezetett be, így megjelent a munkamenet fogalma. Az SVR4 vezette be az egységes, felhasználói belépési viszonyon és munkameneten alapuló folyamatcsoport-kezelést.

Az SVR4 (illetve a 4.4BSD) egy új **viszony objektum (session object)** bevezetésével írja le a felhasználói kapcsolatot a folyamatokhoz. A folyamatok egyaránt tartozhatnak viszonyhoz és folyamatcsoporthoz. A viszonyok leírására egy új azonosítót *(SID – session id)* használ a rendszer. A viszony objektum a felhasználóhoz és a vezérlő terminálhoz rögzített. A viszony kapcsolatban levő folyamatok vezetője *(session leader)* jogosult egyedül a vezérlő terminál lefoglalására, vagy felszabadítására.

Amikor a felhasználó belép a rendszerbe, a termináljához rendelt első folyamat a *setsid()* rendszerhívás segítségével létrehoz egy új viszony csoportot, mely egyben folyamatcsoport is lesz. A felhasználó által elindított további folyamatok ehhez a viszony csoporthoz fognak tartozni, viszont újabb, az elsőtől független folyamatcsoportot is alakíthatnak. Így egyetlen belépéshez több folyamatcsoport is tartozhat. Ezen csoportok egyike az **előtérben futó csoport (foreground group)**, amely kizárólagos hozzáféréssel rendelkezik a terminálhoz. Amennyiben a háttérben futó folyamatok hozzá akarnak férni a terminálhoz, egy speciális jelzés értesíti őket arról, hogy ezt nem tehetik meg, mivel a háttérben futnak.

A viszony csoporton belül a folyamatok megváltoztathatják aktuális folyamatcsoportjukat, de más viszonyhoz tartozó folyamatcsoportba nem léphetnek át. Erre csak egy mód van: a folyamat maga alapít egy új viszony csoportot, teljesen szakítva az előzővel.

5.3.5. Folyamatok közötti kommunikáció (interprocess communication)

Bizonyos programozási környezetekben gyakran együttműködő folyamatok csoportját használhatjuk egymással összefüggő feladatok megoldására. A folyamatok között információáramlásra van szükség. Az információcsere alapvetően két módon valósulhat meg:

- közösen használt tárterületen, vagy
- kommunikációs csatornán keresztül.

Ebben a részben azzal foglalkozunk részletesebben, hogy a UNIX-rendszer milyen eszközöket nyújt a második eset, azaz a folyamatok üzenetváltásos együttműködésének megszervezéséhez.

A UNIX operációs rendszer többféle IPC-mechanizmust kínál. Ezek a következők: jelzések *(signals),* csővezetékek *(pipes)* és folyamat-nyomkövetés *(process tracing)*. Bizonyos UNIX-variánsok ezen kívül rendelkeznek az osztott memória *(shared memory)*, szemaforok *(semaphores)* és üzenetsorok *(message queues)* lehetőségeivel is. A következőkben röviden áttekintjük ezen eszközöket.

5.3.5.1. Jelzések

A jelzések elsődleges célja a folyamatok értesítése aszinkron módon bekövetkező eseményekről. Bár eredetileg hibák jelzésére szolgáltak, egyszerű folyamatok közötti kommunikációt is lehetővé tesznek. A modern UNIX-rendszerek 31 jelzést alkalmaznak, melyek közül kettő, a *SIGUSR1* és *SIGUSR2* az alkalmazások számára fenntartott. Egy folyamat jelzést küldhet egy vagy több másik számára rendszerhívások (*kill()*) segítségével. A jelzésekkel részletesebben az 5.3.4. részben foglalkoztunk.

5.3.5.2. Csővezetékek

A **csővezeték (pipe)** tradicionálisan egy egyirányú, FIFO jellegű, nem strukturált, változó méretű adatokat közvetítő adatfolyam. A csővezeték írói adatokat helyeznek a csőbe, olvasói kiolvassák azokat. A kiolvasott adatok eltűnnek a csőből. Egy üres csőből olvasni kívánó folyamat megáll addig, míg adat nem érkezik a csőbe. Hasonlóképpen, egy tele csőbe írni akaró folyamat is megáll, amíg a csőből egy adat nem távozik.

A *pipe()* rendszerhívással hozható létre egy cső. A rendszerhívás két fájl-leíróval tér vissza, a csővezeték írására és olvasására való leírókkal. A leírókat felhasználva a csővezetéket a fájlkezelésben használható *read()* és *write()* rendszerhívásokkal olvashatjuk, illetve írhatjuk. Ezeket a leírókat a folyamatból létrejött gyermekek öröklik, megosztva ezzel a fájl használatát. Ily módon egy csővezetéknek több írója és olvasója lehet. Egy adott folyamat önmaga is lehet egyszerre író és olvasó is. A csővezetéket azonban általában két folyamat közötti egyirányú kommunikációra használják, amikor pontosan egy írója és egy olvasója van. Ilyen csővezetéket a felhasználók a parancsértelmezőben is létrehozhatnak a **csővezeték jel ('|')** segítségével. A jel bal oldalán szereplő folyamat a cső írója, a jobb oldali pedig az olvasója lesz. A parancsértelmező gondoskodik a csővezeték létrehozásáról a folyamatok indítása során.

A csővezetékek legfontosabb hátrányai a következők:

- mivel nincs rögzített adatméret és -struktúra, a vevőnek nincs tudomása arról, hogy hol vannak az adatok határai,
- amennyiben több olvasó is van, úgy az író nem tudja kiválasztani, hogy melyik olvasónak küldi az üzenetet, és
- mivel az olvasás kiveszi az adatot a csővezetékből, nincs lehetőség több címzetthez is eljuttatni az adatot.

A System V rendszerekben létezik egy speciális csővezeték, az **elnevezett csővezeték (named pipe)**. Ez létrehozási módjában és használatában is különbözik az eddig megismertektől, bár lényegében ugyanazt a célt szolgálja. A felhasználó a fájlrendszerben létrehoz egy FIFO elvű speciális fájlt egy speciális parancs segítségével *(mknod)*. A FIFO-t a folyamatok a fájlokhoz hasonlóan nyithatják meg és használhatják. Viselkedését tekintve az elnevezett csővezeték nem különbözik a korábban megismert, névtelen csővezetéktől. Lényeges különbség, hogy a FIFO a folyamatoktól függetlenül létezik, az írók és olvasók megszűnésével nem törlődik. Ez a megoldás több előnyt is jelent a korábbi csővezetékekkel szemben: a FIFO-kat egymástól független folyamatok is használhatják (csak a nevet kell ismerniük), az adat megmarad még akkor is, ha az összes hozzáférő folyamat megszűnik. Hátrány azonban, hogy kevésbé biztonságosak (csak a fájlrendszerben alkalmazott biztonsági mechanizmusok vonatkoznak rájuk), valamint könnyen előfordulhat, hogy senki sem törli őket, és feleslegesen foglalnak erőforrásokat. A névtelen csővezetékeket egyszerűbb létrehozni és kevesebb erőforrást kötnek le.

Az SVR4 **kétirányú csővezetéket** is implementál. A *pipe()* rendszerhívás itt is két leíróval tér vissza, azonban mindkettő használható írásra és olvasásra is. Valójában két független csővezeték keletkezik *(fd[2])*, melyeknél a kernel oldja meg a leírók egymáshoz rendelését. Mindkét csővezetékhez egy-egy leírót rendel a kernel, így megvalósítva a kétirányú adatforgalmat. Az *fd[0]*-ba írt adat az *fd[1]*-ből olvasható ki, míg az *fd[1]*-be írt adat az *fd[0]*-ból olvasható. A kétirányú csővezeték hasznos eszköz, mivel az alkalmazások többségénél kétirányú adatforgalmat szeretnénk kiépíteni.

5.3.5.3. Folyamat-nyomkövetés

Ezt a lehetőséget elsősorban a nyomkövető programok használják. A *ptrace()* rendszerhívás segítségével egy folyamat befolyásolhatja a gyermeke futását. A következő főbb feladatok oldhatóak meg a rendszerhívás segítségével:

- adat olvasása és írása a gyermek címtartományában,
- adat olvasása és írása a gyermek *u*-területén,
- a gyermek folyamat általános célú regisztereinek írása és olvasása,
- adott jelzések „elkapása”, aminek során a kernel a gyermeket felfüggeszti és értesíti a szülőt a jelzésről,
- megfigyelési pontok (watchpoints) beállítása és törlése a gyermek címtartományában,
- a leállított gyermek folyamat futásának folytatása,
- a gyermek futásának folytatása egy utasítás végrehajtására,
- a gyermek futásának leállítása,
- a gyermek engedélyezheti a szülő számára a nyomkövetést.

A nyomkövető tipikusan gyermek folyamataként hozza létre a követendő programot, amely a *ptrace()* rendszerhívás segítségével engedélyezi a nyomkövetést. A szülő ezek után a *wait()* rendszerhívással várakozik a gyermekkel kapcsolatos eseményekre. A hívás visszatérési értéke ad információt arra nézve, hogy aktuálisan milyen esemény következett be. Ezek után a szülő a *ptrace()* rendszerhívással befolyásolja a gyermeket, majd ismét várakozni kezd.

Bár a *ptrace()* alapvetően lehetővé tette nyomkövető programok létrehozását, rendelkezik fontos korlátokkal és hátrányokkal:

- A nyomkövető kizárólag a közvetlen gyermekeit tudja nyomon követni, az általuk létrehozott gyermekeket már nem.

- A rendszer nem hatékony, rengeteg környezetváltást igényel. A *ptrace()* hívással kiadott utasítások ugyanis nem közvetlenül érik el a gyermeket, hanem a kernel segítségével egy környezetváltás után.
- A nyomkövető már futó folyamatokat nem követhet, mivel a gyermeknek előbb engedélyeznie kell a nyomkövetést.
- *setuid* jelzéssel ellátott programok nyomkövetése általában tiltott, mivel súlyos biztonsági problémákat vetne fel (a címterület módosításával a program működése lényegesen befolyásolható, például ha a gyermek egy másik folyamatot indít, a nyomkövető megváltoztathatja a program nevét).

A mai modern UNIX-rendszerek új módszereket kínálnak a nyomkövetés megoldására a */proc* fájlrendszeren keresztül. Ez a rendszer a felsorolt hiányosságok nagy részét megoldja, így a korszerű nyomkövető programok a *ptrace()* interfész helyett ezt használják.

5.3.5.4. System V IPC

A következő kommunikációs eszközöket (szemaforok, üzenetsorok, osztott memória) összefoglaló néven csak System V IPC-ként szokás említeni. Eredetileg a SYSV UNIX-variánsokban tranzakciófeldolgozásra kifejlesztett eszközök mára részeivé váltak minden korszerű UNIX-rendszernek. Ezen eszközök (IPC-erőforrások) használata és programozói interfésze lényeges közös elemeket tartalmaz.

Minden IPC-erőforrás rendelkezik a következő azonosítókkal:

- **kulcs (key)** – a felhasználó által meghatározott egész szám, mely az erőforrás egy példányát azonosítja,
- **létrehozó (creator)** – az erőforrás létrehozójának felhasználói és csoportazonosítói (UID, GID),
- **tulajdonos (owner)** – az erőforrás tulajdonosának felhasználói és csoportazonosítói (UID, GID),
- **hozzáférési jogok (permissions)** – a fájlrendszerhez hasonlatos olvasási/írási/végrehajtási jogok a tulajdonos, csoportja és mások számára.

Minden erőforrásnak van létrehozó (*shmget(), semget(), msgget()*) és vezérlő rendszerhívása (*shmctl(), semctl(), msgctl()*). A létrehozáskor a kulcs megadása mellett a létrehozási opciókat, illetve egyéb, az adott erőforrásra jellemző paramétereket adhatunk meg. A létrehozó függvények az erőforrás elérésére alkalmas leírót adnak vissza *(shmid, semid és msgid)*. A vezérlő függvényeknek átadott parancsok segítségével befolyásolhatjuk az erőforrás működését. Ezek egyike a megszüntető parancs *(IPC_RMID)* is. A megszüntetés nélkül a kernel folyamatosan fenntartja az erőforrásokat, még akkor is, ha minden kezelő folyamatuk megszűnt. Ennek megvalósítása érdekében a kernel egy elkülönített, a folyamatoktól független területen tartja fent az erőforrásokat és a kezelésükhöz szükséges adminisztrációs táblákat. Ez a terület korlátos méretű, általában

kernel-paraméterként méretezhető. Bizonyos alkalmazások (például adatbázis kezelők) megkövetelhetik az operációs rendszer alapértelmezett területméretének növelését.

5.3.5.5. Szemaforok

A SYSV IPC szemaforok az általános szemafor mechanizmusát és operációit (*P() és V()*) valósítják meg. A megvalósítás némiképp bonyolultabb, körültekintőbb használatot igényel, de összetettebb megoldások is kialakíthatóak segítségével.

A szemaforokat a *semget()* rendszerhívással hozhatjuk létre:

```
semid = semget(key, count, flag);
```

A kernel a megadott kulccsal *(key) count* darab szemafort hoz létre. Létrehozásuk után a *semop()* rendszerhívás segítségével használhatjuk őket.

```
status = semop(semid, semops, nsemops);
```

Az *nsemops* méretű *semops* tömb *sembuf* típusú elemei tartalmazzák a szemaforon végrehajtandó utasításokat. A *sembuf* struktúra a következő:

```
struct sembuf {
unsigned short sem_num;
short sem_op;
short sem_flg;
};
```

A *sem_num* jelöl ki egy szemafort a létrehozottak tömbjéből, a *sem_op* által meghatározott művelet pedig a következő lehet:

- *sem_op > 0* esetén *sem_op* értéke hozzáadódik a szemafor értékéhez,
- *sem_op = 0* esetén a kernel blokkolja a folyamatot mindaddig, míg a szemafor értéke nulla nem lesz,
- *sem_op < 0* esetén a kernel blokkolja a folyamatot, míg a szemafor értéke nagyobb vagy egyenlő lesz, mint a *sem_op* abszolút értéke. Ha ez bekövetkezik, akkor ezzel az értékkel csökkenti a szemafor értékét és továbbengedi a folyamatot.

Látható, hogy egyetlen *semop()* hívás több szemaforon többféle művelet végrehajtását is jelentheti. A kernel garantálja azt, hogy a műveletek végrehajtásáig (vagy a folyamat blokkolásáig) más műveleteket ezen a halmazon nem kezd el végrehajtani (a szemafor műveletek kizárólagosak). Ha bármelyik művelet blokkolja a folyamatot, akkor a kernel visszaállítja az esetlegesen módosított szemaforok értékét a *semop()* hívás előtti értékekre. A *flag* paraméter beállításával *(IPC_NOWAIT)* blokkolás helyett hibaüzenettel való visszatérés is

választható. Másrészt a *SEM_UNDO* flag beállítása esetén a kernel megjegyzi a folyamat által végrehajtott műveleteket, és azokat automatikusan visszaállítja, ha a folyamat kilép. Ez a módszer használható fel olyan holtpont helyzetek kezelésére, amikor a szemafort tartó folyamat kilépése miatt a szemaforra várakozók örökre leállnának a *P()* műveletben.

A System V szemaforimplementáció legnagyobb problémája, hogy a szemaforok létrehozása és inicializálása nem atomi művelet. A két rendszerhívás (*semget() és semctl()*) között más folyamatok is hozzáláthatnak ugyanazon szemafor létrehozásához. A másik probléma a SYSV IPC általános vonása: a szemaforok törlésük nélkül a hozzájuk tartozó folyamatok megszűnése után is létezni fognak, feleslegesen foglalva a rendszer erőforrásait.

5.3.5.6. Üzenetsorok

Az üzenetsor egy olyan leíró, mely üzenetek láncolt listájára mutat. Minden egyes üzenet egy típusjelző után egy adatterületet tartalmaz. A felhasználói folyamat az *msgget()* rendszerhívással hozhat létre egy üzenetsort, melyet aztán az *msgsnd()* és *msgrcv()* rendszerhívásokkal írhat és olvashat a *read()* és *write()* rendszerhívásokhoz hasonlóan.

Az üzenetsorok a csővezetékekhez hasonlatosan FIFO működésűek, azaz a kernel nyilvántartja az üzenetek érkezési sorrendjét, és a legelőször érkezett üzenetet adja vissza az első olvasási kérésre. Két lényeges különbség van a csővezetékekhez képest. Az üzenetek típusjelzője felhasználható szűrőként az olvasáskor bizonyos típusú üzenetek kiválasztására, azaz ilyenkor az elsőként érkezett, megegyező típusú üzenettel tér vissza az *msgrcv()* hívás. Ez a mechanizmus felhasználható például üzenetprioritási rendszer megvalósítására.

A másik különbség az üzenetsorok azon tulajdonsága, hogy az adatokat nem formázatlan folyamként, hanem elkülönülő üzenetekben továbbítják, ami az adatok pontosabb feldolgozását teszi lehetővé, mivel az üzenetek belsejében a tartalmi feldolgozást segítő típusazonosítókat lehet elhelyezni.

Az üzenetsorok jól használhatóak kisméretű adatcsomagok küldésére, de kevéssé alkalmasak nagy mennyiségű adat átvitelére. A kernel belső adatbuffereket használ az üzenetek tárolására a kiolvasásig. Az üzenet beérkezésekor a kernel a küldő adatterületéről átmásolja az üzenetet a belső tárolóba, majd a kiolvasáskor tovább másolja a fogadó adatterületére. Ily módon egyazon adatcsomagot kétszer kell mozgatni, ami nagy mennyiségű adat esetén költséges megoldás. Az üzenetsorok másik hátránya, hogy nem nevezhető meg címzett az üzenetekhez. Bármilyen, megfelelő jogosultságokkal rendelkező folyamat kiolvashatja az üzeneteket. Ennek megfelelően a kooperációban részt vevő feleknek meg kell egyezniük valamilyen saját címzési protokollban, amit a kernel semmilyen eszközzel nem támogat.

5.3.5.7. Osztott memória

Az osztott memória a központi tár egy olyan régiója, melyet egyszerre több folyamat is használhat. A folyamatok ezen tartományt bármilyen virtuális címhez hozzárendelhetik saját címtartományukban. A hozzárendelés után a memória a folyamat saját memóriájával teljesen megegyező módon – rendszerhívások nélkül – használható.

Ezért az osztott memória a *leggyorsabb kommunikációs forma azonos gépen futó folyamatok között*. A memória létrehozója a kulcs megadása mellett meghatározhatja a memória méretét:

shmid = shmget(key, size, flag);

Már létező osztott memóriaterülethez az *shmat()* rendszerhívással lehet virtuális címet rendelni, illetve az *shmdt()* hívással lehet a hozzárendelést megszüntetni:

addr = shmat(shmid, shmaddr, shmflag);

shmdt(shmaddr);

A memória teljes megszüntetéséhez az *shmctl()* rendszerhívással ki kell adni az *IPC_RMID* parancsot. Ez azonban nem jár a memória azonnali megszüntetésével, csak megjelöli azt. A tényleges megszüntetés akkor következik be, amikor minden, a memóriához kapcsolódott folyamat lekapcsolódik róla. A törlésre megjelölt memóriához azonban új folyamatok már nem kapcsolódhatnak. Törlés nélkül a memória a folyamatok megszűnése után is megmarad, megőrizve minden adatot, amit a folyamatok ott elhelyeztek. Ily módon lehetővé válik például a folyamatok két futása közötti adatátmentés.

Az osztott memória rendkívül gyors és egyszerűen megvalósítható módszer az azonos gépen futó folyamatok közötti adatcserére. Legnagyobb hátránya, hogy semmilyen szinkronizációs mechanizmus nem kapcsolódik hozzá, így az adatok módosítása konkurens lehet. Még önálló adatstruktúrák módosításán belül is lehetséges a párhuzamosság, ami teljesen inkonzisztens adatokat eredményez. Ezért az osztott memóriát használó folyamatoknak egymás között meg kell oldaniuk a konzisztens módosítás rendszerét (például szemaforok segítségével).

5.3.5.8. Hálózati kommunikáció – socket programozás

A számítástechnikában a gépeket összekapcsoló hálózat megjelenése egy új fejezetet nyitott. A soros vonali terminál illesztés (dialup) továbbfejlődése gépek közötti pont-pont adatkapcsolat kialakítását tette lehetővé. Megjelent a kliens-szerver modell, mely szolgáltatásalapon különítette el a felhasználót (kliens) és a szolgáltatót (szerver).

A UNIX operációs rendszerben alapvetően a TCP/IP protokollcsaládra alapuló kommunikációs eszközöket fejlesztettek ki a különböző gépeken futó folyamatok közötti kommunikáció megoldására.

A különböző gépeken futó alkalmazások közötti adatátvitel programozói interfésze nagyon hasonlatos az eddigiekben megismertekhez. A hálózati kommunikációs csatorna leírója az ún. **socket**, egy absztrakt objektum, ami felhasználható üzenetek küldésére és fogadására. A socket programozói interfész nemcsak hálózati, hanem gépen belüli IPC megvalósítására is alkalmas.

A socket (és az adatforgalom) alapvetően két típusú lehet (zárójelben az IP fölötti réteg protokolljának rövidítése olvasható):

- **folyam (stream)** (TCP): megbízható, sorrendhelyes, kétirányú kapcsolat
- **üzenetszórás (datagram)** (UDP): kapcsolatmentes csomagküldés, ahol a sorrend és a megérkezés nem garantált.

Amikor egy folyamat egy socket hívást hajt végre, a kernel ezen alacsony szintű adatátviteli protokollokat hívja meg az üzenetek átvitelére. A protokollhoz tartozó átviteli függvényeket egy táblából választja ki a kernel, és elküldi hozzájuk az adatokat. A magasabb szintű kommunikációs rétegek az alacsonyabb szintűekkel közös adatstruktúrákon keresztül cserélik ki az adatokat.

A socket hívások a hívó folyamat környezetében hajtódnak végre. Ily módon minden hiba szinkron módon jelentkezhet a hívónál.

A socket hívások megvalósítására az ún. **socklib** felhasználói szintű könyvtárat lehet használni. Ez a legtöbb UNIX-variánsban elérhető.

A socket létrehozása

A hálózati programozás legkényesebb (legnehezebben megérthető) része a socket létrehozásával, illetve konfigurálásával kapcsolatos. Ellentétben a fájlműveletek egyszerű *open()* rendszerhívásával, itt több rendszerhívást is használni kell, valamint a kitöltendő adatstruktúrák is bonyolultabbak. A socket a következő rendszerhívással hozható létre:

int socket(int domain, int type, int protokol);

ahol a domain esetünkben az AF_INET (más domainek is léteznek, mivel a socket nem csak hálózati kommunikációra használható), a típus *stream* vagy *datagram* (SOCK_STREAM, vagy SOCK_DGRAM), a protokoll praktikusan 0 (akkor más, ha a típuson és domainen belül többféle protokoll is létezik). A visszatérési érték a socket-leíró, vagy hibajelzés. Ez azonban önmagában nem elegendő adatok küldéséhez.

A socket kötése

A socket létrehozása csak a névtérben történik meg a *socket()* rendszerhívással. A socket a következő rendszerhívással köthető a lokális gépen egy porthoz:

int bind(int sockfd, const struct sockaddr *addr, int addrlen);

Ezt tipikusan a szerver alkalmazásokban használjuk. Az így kötött portra kapcsolódhatnak a kliens alkalmazások a következő rendszerhívással:

int connect(int sockfd, const struct sockaddr *addr, int addrlen);

A *connect()* segítségével a socket egy tényleges adatúthoz köthető, azaz egy másik gépen futó alkalmazásig (a másik gép adott portjáig) vezető hálózati kapcsolatot hozunk létre. Ez a hívás egyrészt felépíti az adatutat, másrészt hozzárendel egy helyi portot, amihez köti a socketet. Első ránézésre nincs különbség az előző *bind()*-

hez képest, azonban a cím *(addr)* struktúra kitöltése különböző a két esetben. A cím kitöltéséhez egy struktúrát használunk:

struct sockaddr {

short int sin_family; /* Address family, esetünkben AF_INET */

char sa_data[14]; /* 14 bájtos protokoll cím */

}

Ez túl általános, ezért az AF_INET domainre készült egy jobb struktúra is:

struct sockaddr_in {

short int sin_family; /* AF_INET */

unsigned short int sin_port; /* portszám */

struct in_addr sin_addr; /* IP cím */

unsigned char sin_zero[8]; /* feltöltés sockaddr méretűre */

};

Ez a struktúra méretében megegyezik az előzővel (fontos a sin_zero feltöltése nullával), ezért használható helyette, csak a függvények hívásakor kell típuskonverziót végezni. Fontos megjegyezni, hogy a *sin_port* és a *sin_addr* hálózati bájtsorrendet követ (network byte order), tehát erre a formára kell hozni. Erre (illetve a visszaalakításra) a következö függvények valók: *htons(), htonl(), ntohs() és ntohl(),* ahol az első betű a forrás *(n: network, h: host)*, a negyedik betű a cél, míg az utolsó a típus *(l: long, s: short)*. A kommunikáció során az adatokat is célszerű ilyen bájtsorrendben küldeni, mivel így kikerülhetők a különböző architektúrák eltérő ábrázolási sorrendjéből fakadó problémák.

A fenti struktúrában szereplő *in_addr* kifejtése a következő:

struct in_addr {

unsigned long s_addr; /* 4 bájtos cím */

}

például:

struct in_addr server_addr =

inet_addr(„152.66.82.1”); /* hálózati bájt sorrenddel tér vissza */

A név szerinti címek leírására a következő struktúra, illetve kezelésükre a következő függvények használhatók:

```
/* FONTOS! Minden adat hálózati bájtsorrendben adott */

struct hostent {

char *h_name; /* elsődleges név */

char **aliases; /* alternatív nevek */

int h_addrtype; /* esetünkben AF_INET */

int h_length; /* a cím hossza bájtokban */

char **h_addr_list; /* a gép címei, nullával végződik */

}

#define h_addr h_addr_list[0]

/* a gép elsődleges címe */

/* A címek lekérdezése név alapján: */

struct hostent *gethostbyname(const char *hostname);
```

A socket használata

A szerverek a *bind()* hívás után bejövő adatokra várakoznak. Mielőtt ezt megtennék, egy rendszerhívással konfigurálják a várakozási sor hosszát:

```
int listen(int sockfd, int backlog); /* backlog: a sor hossza */

int accept(int sockfd, void *addr, int *addrlen);
```

Az operációs rendszer a portra beérkező kapcsolatokat egy sorban helyezi el, aminek hosszát határozza meg a *backlog* paraméter. A szerver az *accept()* rendszerhívással a sor elején álló kérést fogadja, az *addr* struktúrában megkapva annak paramétereit. A következő *accept()* a sorban következő kéréssel fog foglalkozni. Az *accept* visszatérési értéke egy új socketleíró, mely a klienshez létrejött kapcsolatban használható adatküldésre és fogadásra:

```
int send(int sockfd, const void *msg, int len, int flags);

int recv(int sockfd, void *buf, int len, unsigned int flags);

/* Mint minden fájlleíróra, itt is használható a read() és a write() */
```

Mindkét rendszerhívás az elküldött, illetve fogadott bájtok számával tér vissza, amely küldésnél kevesebb is lehet, mint az előírt.

A socket lezárása

A socket a *shutdown()* és a fájl műveleteknél megszokott *close()* rendszerhívásokkal zárható le. Az előbbinek paraméterként megadható, hogy a kétirányú adatforgalomból melyik irányt zárja le.

Tipikus szerver- és kliens-alkalmazás

Egy tipikus szerver-alkalmazás szerkezete UNIX alatt:

```
servsock = socket();
bind(servsock, ...);
listen(servsock, ...);
while (1) {
newsock=accept(servsock, ...);
if (fork()==0) { /* a gyerek kiszolgálja a kérést */
if (send(new_fd, „Hello, world!\n”, 14, 0) == -1)
perror(„send”);
close(new_fd);
exit(0);
} /* a gyerek vege */
close(new_fd); /* a szülőnek nem kell */
} /* while() */
```

A kliens programjának szerkezete:

```
#define SERVERPORT 1201
srvent = gethostbyname(server_hostname);
sockfd = socket();
srv_addr.sin_family = AF_INET;
srv_addr.sin_port = htons(SERVERPORT);
srv_addr.sin_addr = *((struct in_addr *)srvent->h_addr);
bzero(&(srv_addr.sin_zero), 8);
connect(sockfd, (struct sockaddr *)&srv_addr, sizeof(struct sockaddr))
numbytes = recv(sockfd, buf, MAXDATASIZE, 0)
buf[numbytes] = ‘\0’;
```

```
printf(„Received: %s”,buf);
...
close(sockfd);
```

Megjegyzések

Az itt ismertetett rendszerhívások nemcsak UNIX, hanem más operációs rendszer alatt is így használhatóak (könnyen írható olyan forráskód, mely UNIX, DOS, Windows, OS/2 alatt is lefordul). Legnagyobb különbség az egyéb rendszerhívásokban van, például új folyamatot másképp kell Windows és UNIX alatt indítani (de még a UNIX-verziók között is lehet választási lehetőség). Windows, OS/2, illetve újabb UNIX-verziók (például Solaris 2) esetén már használhatunk szálakat *(thread)* is (például a fent bemutatott tipikus szerverben a kérés kiszolgálására), míg régebbi UNIX-ok (például SunOS) esetén csak folyamatokban *(process)* gondolkozhatunk.

5.3.6. Állományrendszer implementációk

A mai modern UNIX-rendszerek számos különböző típusú állományrendszert támogatnak. Ezek két nagy csoportra oszthatók: lokális és távoli állományrendszerek. Az előbbi a rendszerhez közvetlenül csatlakoztatott berendezéseken tárolja az adatokat, míg az utóbbi lehetőséget nyújt a felhasználónak, hogy távoli gépeken elhelyezkedő állományokhoz férjen hozzá. A legtöbb modern UNIX-rendszerben megtalálható két általános célú állományrendszer a **System V állományrendszer** (*s5fs*) és a **Berkeley Fast File System** (FFS). Az *s5fs* az eredeti UNIX-állományrendszer. A System V rendszerek mindegyike és a kereskedelmi rendszerek nagy része is támogatja ezt az állományrendszert. Az FFS-t a 4.2 BSD vezette be, jobb hatékonyságú, robusztusabb és több funkcionalitást biztosít korábbi társánál. Kiváló tulajdonságai miatt széles körben elterjedt a kereskedelmi változatokban, sőt a SVR4-be is bekerült.

A korai UNIX-rendszerek egyetlenegy típusú állományrendszer kezelésére voltak képesek, így a rendszerek fejlesztői választásra kényszerültek. Mint azt majd a későbbiekben látni fogjuk, a Sun Microsystems által kidolgozott *vnode/vfs* interfész lehetővé tette több állományrendszer egyidejű alkalmazását is.

A következőkben először részletesen ismertetjük a *s5fs* felépítését, annak lemez szervezését és az alkalmazott adatszerkezeteket, majd rámutatunk azokra a hiányosságokra, amik elvezettek az FFS és a *vnode/vfs* interfész kidolgozásához. Ez a tárgyalásmód egyrészt segít megérteni az állományrendszerek általános felépítését és funkcióit, másrészt szépen mutatja a mérnöki fejlesztés, problémamegoldás logikus lépéseit.

5.3.6.1. A System V állományrendszer

Az állományrendszer felépítése

A Unix más operációs rendszerekhez hasonlóan **logikai szinten**, nem pedig diszk szinten kezeli az állományrendszert. A kernel logikai blokkokban szervezi az állományrendszert, egy blokk lehetséges mérete: $512*2^k$ byte, ahol a k kitevő tipikus értékei a 0–5 tartományba esnek, megválasztásánál az alábbi szempontokat szokás figyelembe venni:

- minél nagyobb a blokkméret, annál hatékonyabb az adathozzáférés, kisebb az adategységre jutó keresési idővesztéség, nagyobb az átvitel sávszélessége,
- minél kisebb a blokkméret, annál kisebb a tördelődés.

A kezdeti *s5fs* implementációban 512, majd a későbbiekben 1Kb-os blokkméretet alkalmaztak.

Az állományrendszer logikai szerkezete

A továbbiakban részletesen tárgyaljuk a *s5fs* lemezen található adatszerkezeteit. A lemezt az állományrendszer négy logikailag eltérő feladatot ellátó részre osztja:

boot blokk szuper blokk *inode* lista adat blokkok

Az egyes részek feladatai az alábbiak:

I. *boot blokk* – A rendszer elindulásához szükséges információkat tartalmazza. (Bár csak egyetlen állományrendszernél van szükség a boot blokkra, minden állományrendszer tartalmazza, legfeljebb üres).

II. *szuperblokk* – Az állományrendszerről tartalmaz a rendszer működéséhez szükséges **metaadatokat**, az állományrendszer állapotát írja le. Az itt tárolt információkat a későbbiekben részletezzük.

III. *inode lista* – Minden állományhoz pontosan egy *inode* tartozik, ezek alapján lehet az állományokra hivatkozni. A lista méretét a rendszergazda adja meg, meghatározva ezzel az állományrendszerben a maximális állományszámot. Az *inode*-ok és a lista szerkezetét a későbbiekben részletesen tárgyaljuk.

IV. *adat blokkok* – Ezek szolgálnak a fizikai adattárolásra.

A következőkben tekintsük át az egyes elemeket egy kicsit részletesebben.

A szuperblokk szerkezete

A szuperblokk az állományrendszer egészéről tartalmaz fontos információkat, ún. *metaadatokat*. A szuperblokkban tárolt legfontosabb információk:

- az állományrendszer mérete,
- a szabad blokkok száma,
- a szabad blokkok listája,
- a következő szabad blokk indexe,
- az *inode* lista mérete,
- a szabad *inode*-ok száma,
- inode lista (tömb),
- a következő szabad *inode* indexe,

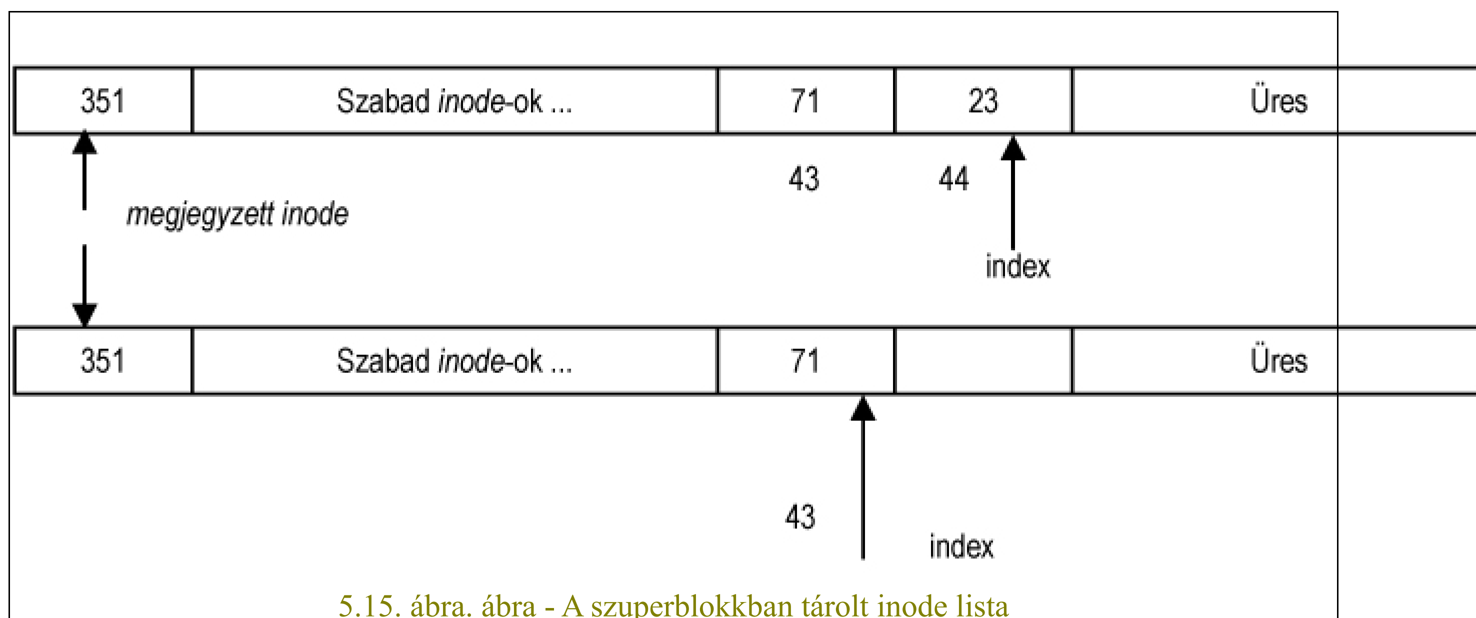
- lock mezők (*inode*, blokklista),
- módosítás jelzőbit.

A továbbiakban a szuperblokk két legfontosabb adatszerkezetének, a szuperblokkon belül tárolt *inode listának* és a *szabad blokkok listájának* a felépítését és kezelését ismertetjük.

Az inode lista

Mint azt már korábban láttuk, az állományrendszer a szuperblokk után egy lineáris listában tárolja az *inode*-okat. Az *inode* tartalmaz egy típus mezőt, amely értéke 0 ha az *inode* szabad, vagyis felhasználható. Így a kernel viszonylag könnyen kereshet szabad *inode*-okat. Azonban az állományrendszerben gyakori esemény, hogy új *inode*-ra van szükség, hisz minden megnyitott állományhoz hozzá kell rendelni egyet, így a fenti megoldás nem elég hatékony. Ezért a szuperblokk tartalmaz egy fix méretű tömböt (szuperblokk *inode* lista), amelyet gyorsítótárként (cache) használ a szabad *inode*-ok sorszámainak tárolására. Minden egyes elem egy szabad *inode* sorszámát tárolja. Ha a rendszernek szabad *inode*-ra van szüksége (például fájl létrehozása), keresés helyett kiveszi a tömbben tárolt utolsó elemet. A tömb kezelését egy index nyilvántartása segíti. Ez megmutatja, hogy hány felhasználható elem (szabad *inode* sorszám) van még a tömbben, illetve éppen az utolsó felhasználható elemre mutat. Az 5.15. ábra felső részén az index a 44. elemre mutat, vagyis a 44. elem tartalmazza a legközelebb felhasználandó *inode* indexét. Ez a példában a 23-as. Amennyiben a kernel felhasznál egy *inode*-ot, akkor az indexet eggyel csökkenti (lásd az 5.15. ábra alsó részét).

Amikor kiürül a szuperblokk *inode* listája, a kernel elkezdi vizsgálni az (igazi) *inode* listát, szabad *inode*-okat keres, és hátulról kezdve teljesen feltölti a szuperblokkon belüli *inode* listát. Az utolsó megtalált szabad *inode* sorszám kerül a tömb első helyére. Ezt **megjegyzett *inode***-nak is nevezik. Ennek nagy hatékonyság-növelő szerepe van, mert a kernel a legközelebbi feltöltéskor ettől a sorszámtól kezdődően kezdi el keresni a szabad *inode*-okat. Amikor a kernel felszabadít egy *inode*-ot, akkor megnézi, hogy van-e szabad hely a tömbben. Amennyiben igen, egyszerűen beteszi a sorszámot a tömb első szabad helyére, és eggyel növeli az indexet. Amennyiben a tömb tele van, megnézi, hogy a megjegyzett *inode* sorszáma kisebb-e, mint a felszabadítandó *inode*-é. Amennyiben kisebb, akkor a felszabadított *inode*-ot az algoritmus meg fogja találni a diszken, hiszen az a megjegyzett *inode* után következik, így nincs további teendő. Amennyiben a felszabadított *inode* sorszáma kisebb, mint a megjegyzetté, akkor annak helyére írja – ez lesz tehát a megjegyzett *inode* –, így biztosítva, hogy a kernel valóban a legkisebb sorszámú szabad *inode*-tól kezdje majd a következő keresést.



5.15. ábra. ábra - A szuperblokkban tárolt inode lista

A szabad lemezblokkok listája

Mint azt korábban láttuk, az *inode*-ról egyértelműen el lehet dönteni, hogy szabad-e: amennyiben a típus mező értéke 0, akkor szabad, ellenkező esetben foglalt. Sajnos a lemezblokkok (adatblokkok) esetében pusztán azok tartalmának vizsgálatával nem lehet eldönteni, hogy szabadok-e, vagy használatban vannak, ezért a szabad lemezblokkokat adminisztrálni kell, azokat külön kell kezelni. Ennek érdekében a szuperblokk tartalmaz egy szabad blokklistát, amelynek a szervezése az alábbi.

A lista rögzített méretű, és a szabad blokkok sorszámait tartalmazza. Az első (a felhasználás sorrendjében utolsó) elem pedig egy olyan blokk sorszáma, amelyik szintén szabad blokkok sorszámait tartalmazza. (Tegyük fel, hogy egy blokk mérete 1 K, és a sorszámok 4 byte-osak. Így egy blokkban 1024/4 = 256 szabad blokk sorszámát lehet tárolni.)

Például a szuperblokkban tárolt szabad blokkok listájának egy lehetséges állapota:

A szuperblokkban lévő szabad blokkok listája

105	108	115	211	…		

A 105-ös blokk tartalma

214	206	203	217	…		

A 214-es blokk tartalma

313	315	317	321	…		

Ezen lista kezelése hasonló a szabad szuperblokk *inode* lista kezeléséhez: egy index segítségével a kernel mindig nyilvántartja, hogy hányadik elem tartalmazza a következő szabad blokk sorszámát. Amennyiben szabad blokkra van szükség, a kernel visszaadja az első szabad blokk sorszámot. Ameny-nyiben felszabadul egy blokk, akkor a sorszámát a kernel beírja az első szabad helyre. Amennyiben nincs szabad hely már a listán, akkor a lista

tartalmát bemásolja a felszabadított blokkba, és a visszaadandó blokk sorszámát beírja a lista utolsó helyére. Így a szabad blokkok listája egy újabb blokknyi (256) index hellyel bővül. Amennyiben a szuperblokkban tárolt szabad blokkok listája már csak egy elemet tartalmaz, akkor ez olyan blokkra mutat, amelyik újabb szabad blokkok sorszámait tartalmazza. Ha most újabb szabad blokkra van szükség, először a kernel a maradék egyetlen sorszám által mutatott blokk tartalmát (újabb szabad blokkok sorszámai) bemásolja a szuperblokk szabad blokk listájába (ekkor a blokk valóban szabaddá vált), majd visszaadja a sorszámot.

Lássunk egy példát!

A szuperblokkban tárolt szabad blokkok listája már csak egyetlen sorszámot tartalmaz, a 105-öst (amely újabb 256 szabad blokk sorszámait tartalmazza).

A szuperblokkban lévő szabad blokkok listája

105				…		

A 105-ös blokk tartalma

214	206	203	217	…		

Tegyük fel, hogy ekkor a kernel felszabadítja a 934-es blokkot és visszateszi a szabad blokkok listájára.

A szuperblokkban lévő szabad blokkok listája

105	934			…		

A 105-ös blokk tartalma

214	206	203	217	…		

Tegyük fel, hogy ezek után szükség van egy szabad blokkra. A kernel visszaadja a 934-es blokkot. Ezután újabb blokkra van szükség. Ekkor a kernel a 105-ös blokk tartalmát bemásolja a szuperblokk szabad blokk listájába, majd visszaadja a 105-ös blokkot.

A szuperblokkban lévő szabad blokkok listája

214	206	203	217	…		

A 214-es blokk tartalma

	315	317	321	…		

5.16.ábra. táblázat - A példa állomány lemezblokkjainak elhelyezkedése

Az eddigiek során számos alkalommal hivatkoztunk már az *inode*-okra, vázlatosan ismertettük a rendeltetését, azonban még nem adtuk meg a belső szerkezetét, a benne tárolt információkat. A UNIX-állományrendszerben betöltött központi szerepe miatt az alábbiakban részletesen ismertetjük az *inode* felépítését.

Az inode

Az ***inode*** a fizikai állományhoz tartozó azonosító. Az állományrendszer minden egyes állományához pontosan egy *inode* tartozik. Az állományrendszerben az *inode*-ok diszken találhatók, azonban a gyorsabb működés érdekében a kernel cache-eli azokat a memóriában. A diszken tárolt *inode* számos, az állományhoz kapcsolódó információt tárol:

- tulajdonosazonosító (UID, GID),
- állomány típusa – reguláris, könyvtár, FIFO, karakteres vagy blokkos berendezés,
- állomány-hozzáférési jogosultságok (tulajdonos, csoport, illetve a világ számára, olvasási, írási, illetve végrehajtási jogok),
- időcímkék:
  - az utolsó állományhozzáférés ideje,
  - az utolsó állománymódosítás ideje,
  - az utolsó attribútum módosítás ideje (*inode* módosítás),
- linkek száma (hány könyvtári bejegyzés hivatkozik az adott fizikai állományra),
- címtábla (mutatók adatblokkokra – 10 db direkt, 1 db indirekt, 1 db kétszeres indirekt és 1 db háromszoros indirekt mutató),
- állományméret.

A kernel a memóriában cache-eli az *inode*-okat (ezeket **in-core *inode***-oknak is nevezzük). Ezek egy kicsit más felépítésűek, mint a diszken tárolt másolatuk. Tartalmazzák a diszken tárolt *inode* összes információját, azonban néhány további információval ki vannak egészítve:

- az in-core *inode* státusa
  - zárolt (locked),
  - folyamat vár rá,
  - eltér a diszken lévő változattól
  - attribútum írás következtében,
  - adat írás következtében,
- az állomány *mount pont*, vagyis az adott pontra egy másik állományrendszer csatlakozik,
- az állományt tartalmazó állományrendszer logikai berendezés azonosítója,
- az *inode* sorszáma (a diszken az inode-ok az *inode* listában sorszámuk által meghatározott, rögzített helyen találhatóak, nincs szükség a sorszám tárolására),

- mutatók más in-core *inode*-okra (hash sorok, szabad lista),
- hivatkozásszámláló (hány példányban van nyitva az állomány).

A hivatkozásszámláló fontos szerepet játszik a memóriabeli *inode*-ok kezelésében. Mivel a számláló megmutatja, hogy az adott állomány hány példányban van nyitva, így amíg az nagyobb nullánál, addig az *inode* aktív, használatban van. Amennyiben a számláló értéke nulla, akkor az *inode* inaktív, éppen senki sem használja, így az a szabad listára kerülhet, ahonnan majd a kernel újra allokálhatja, és esetleg már egy másik állományhoz rendelheti.

Reguláris állományok szerkezete

A UNIX-rendszer az indexelt allokáció egy speciális formáját alkalmazza a fájlokhoz tartozó adatblokkok lefoglalásakor. Az indexelt allokáció lehetővé teszi az adatterület blokkonkénti lefoglalását. Így a fájl egyes adatblokkjai a lemez tetszőleges blokkjában tárolhatók. A fájlhoz tartozó adatok elhelyezkedését az *inode*-ban található címtábla (13 mutató) írja le.

Tegyük fel, hogy a rendszer diszk blokkjai 1 K-sak, és 4 byte-on lehet a blokkokat címezni. Az *inode* címtáblájának első 10 mutatója egy-egy 1 K-s diszk blokkra mutat. Ezek a direkt elérésű blokkok. A következő mutató egy olyan diszk blokkra mutat, amely további mutatókat tárol, most már adatokat tartalmazó blokkokra. Mivel egy blokkba 256 mutató fér el, így a fenti egyszeres indirekt címzéssel további 256 K adatot lehet megcímezni. A következő mutató kétszeres indirekt mutató, vagyis egy olyan blokkra mutat, amely 256 mutatót tartalmaz, amelyek mindegyike egy-egy további indirekt blokkra mutat. Vagyis most hivatkozáskor az első két lépésben indirekt blokkokat kapunk, és csak a harmadik lépésben jutunk el az adatokhoz. Ezzel a módszerrel 64 M adatot lehet megcímezni. A következő, egyben utolsó mutató egy háromszoros indirekt mutató, vagyis még a harmadik hivatkozás is 256 mutatót tartalmazó indirekt blokkra mutat, és onnan egy újabb referenciával juthatunk el az adatokhoz. Ezzel a módszerrel 16 G adatot lehet címezni. Tehát a fent vázolt struktúrával összesen:

10 db direkt blokk 10 K

1 db egyszeres indirekt blokk 256 K

1 db kétszeres indirekt blokk 64 M

1 db háromszoros indirekt blokk 16 G

adatot lehet tárolni. A valóságban persze ennél kevesebbet, mert az *inode* állományhossz mezője is 4 byte, a méretet byte egységben tárolják, így ez 4 G-ra korlátozza a maximális állományméretet.

A jobb érthetőség érdekében tekintsünk két példát: (a blokkméret továbbra is 1 Kb és a blokkokat 4 byte-on címezzük, így egy blokkban 256 blokkcím tárolható. A példához használt állomány lemezblokkjait az 5.16. ábra mutatja.)

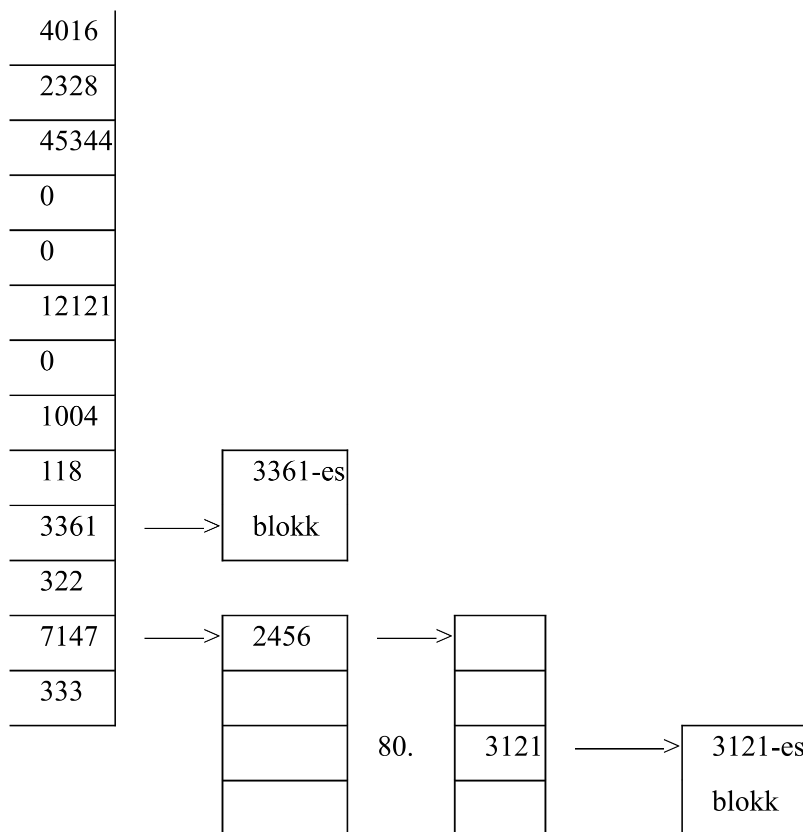
1. példa

Tegyük fel, hogy egy fájl 9500-adik byte-ját akarjuk olvasni, amihez a kernelnek meg kell találni az ehhez tartozó blokk címet.

A 9500-adik byte a 9. direkt blokkban van (9*1024 = 9216), ott pedig a 284-edik (a blokkokat 0-tól kezdődően sorszámozzuk). A példában az *inode* 9. direkt blokkra mutató pointer a 3361. lemezblokkra mutat, vagyis a 3361. blokk 284. byte-ja a keresett adat.

2. példa (5.16. ábra)

Tegyük fel, hogy a 355 000-es byte-ot akarjuk olvasni. Ehhez kétszeres indirekt blokkot kell használni, mert a 10 direkt és az egyszeres indirekt blokkal 10*1024 + 256*1024 = 272384 byte-ot lehet elérni. Innen számítva a 82616. byte-ot keressük. Az a kétszeres indirekt blokk első (0-ás indexű) mutatója által kijelölt egyszeres indirekt blokkban van, méghozzá annak 80. elemében (80*1024 = 81920). Az így kijelölt blokk 696. byte-ja a keresett adatelem. A fenti példa esetén a kétszeres indirekt blokk pointere a 7147. diszk blokkra mutat, annak az első pointere a 2456. blokkra, ami tartalmazza 80. elemként a keresett adatokat tartalmazó blokk címét, a 3121-et. Ezen a blokkon belül a 696. byte-ot kerestük.



5.19.ábra. táblázat - /etc könyvtár egy lehetséges tartalma

A fenti séma problémája, hogy az indirekciók időigényesek, nagy fájlok esetén lassul az adatelérés. A bevezetés időszakára jellemző statisztika szerint azonban az állományok kb. 85%-a kisebb 8 K-nál (így elférnek a direkt blokkokban), és ezen állományok kb. 48%-a kisebb 1 K-nál! Ebből következik, hogy indirekt elérésre

ritkán van szükség, így az effektív adathozzáférés gyors. Nagy fájlok esetén pedig egyrészt a felhasználók is türelmesebben viselik a hosszabb keresési időket, másrészt a rendszerek egyéb megoldásokat is alkalmaznak a gyorsítás érdekében (például buffer-cache).

Az állománykezelés logikai sémája és adatszerkezetei

A UNIX a fájlkezelés multiprogramozott rendszerekben felmerülő kérdéseire a következő választ adja:

- A folyamatoknak csak megnyitáskor kell névvel hivakozniuk egy fájlra. A megnyitás egy számot (logikai perifériaszám) ad vissza a folyamatnak, a további műveletekben (read, write) ezzel hivatkozhat a fájlra.
- Egy folyamat többször is megnyithat egy fájlt, minden megnyitáskor önálló logikai periféria jön létre, amelyek azonban külön-külön fájlmutatóval ugyan, de ugyanazt a fizikai fájlt jelentik. Így egy folyamat különböző logikai perifériaként ugyannak a fájlnak különböző részeit használhatja különböző fájlmutatókkal.
- Több folyamat megnyithatja és használhatja ugyanazt a fájlt párhuzamosan, a *read* és *write* rendszerhívások atomicitása teljesül, azonban hosszabb távú szinkronizáció megoldása külön műveleteket igényel.
- A gyermek folyamatok öröklik a szülők logikai perifériáit. Ha a gyermek és a szülő ugyanazt a logikai perifériát használja, a gyermek és a szülő fájlműveletei a CPU-ütemezéstől függő sorrendben írják, vagy olvassák a fájlt ugyanazon fájlmutató használatával.

A *s5fs* állományrendszerben a logikai állománykezelés három szintre tagozódik, amelyekhez egy-egy fontos adatszerkezet tartozik. Az alábbiakban az egyes lépcsőkhöz tartozó adatszerkezeteket (folyamatonkénti állományleíró tábla, globális állománytábla, és *inode* tábla) és a hozzáférésben betöltött funkciójukat ismertetjük.

Folyamatonkénti állományleíró tábla

Minden egyes folyamathoz tartozik egy állománytábla. Mint azt a neve is mutatja, ez egy folyamathoz tartozó adatszerkezet. A tábla egy bejegyzése egy logikai perifériának felel meg. A folyamatok állománytáblájuk bejegyzéseinek indexeit használják a logikai periféria azonosítására. Ebben a táblában minden, a folyamat által elérhető, megnyitott állományhoz legalább egy mutató tartozik. A mutató a globális állománytábla megfelelő elemére mutat.

Globális állománytábla

Minden egyes *open* rendszerhívás hatására létrejön egy bejegyzés (struktúra) a globális állománytáblában, amely az adott állománynévhez rendelődik, tartalmazza a megnyitás módját, jogosultságokat, egy fájlmutatót, hogy hol tartunk az állományban az olvasással/írással, illetve egy hivatkozásszámlálót, amely megmondja, hogy hány folyamatonkénti állományleíró tábla bejegyzés mutat rá. (A globális állománytáblában egy állománynévhez

több bejegyzés is tartozhat, itt a bejegyzések munkamenetet tárolnak.) A globális állománytábla használata az állományok osztott használatát teszi lehetővé.

Inode tábla (in-core inode tábla)

Minden egyes megnyitott állományhoz tartozik **egy** *inode* a memóriában, tartalmaz egy hivatkozásszámlálót, hogy az adott nevű fizikai állományt hány példányban használják, vagyis hány globális állománytábla bejegyzés mutat rá.

A folyamatonkénti állományleíró tábla első három eleme (0, 1, 2) a standard input-, a standard output-, illetve a standard error logikai perifériákat jelöli. Ezeket a folyamatok indulásukkor megnyitva kapják.

A UNIX-ban az állományok megnyitására az *open* rendszerhívás szolgál. Ennek alakja:

fd = *open*(path, flag, mode);

Kötelezően meg kell adni a megnyitandó állomány elérési útját (path) és a megnyitás módját. A rendszerhívás egy egész számot (fd) ad vissza, ami tulajdonképp nem más, mint egy index a folyamatonkénti állományleíró táblába.

Az alábbi példában megnyitunk három állományt, majd az egyik állományleírót megkettőzzük. Az 5.17. ábra mutatja, hogy a rendszerhívások hatására milyen bejegyzések jelennek meg a fent ismertetett három adatszerkezetben. Az ábrán a nyilak mutatókat jelölnek.

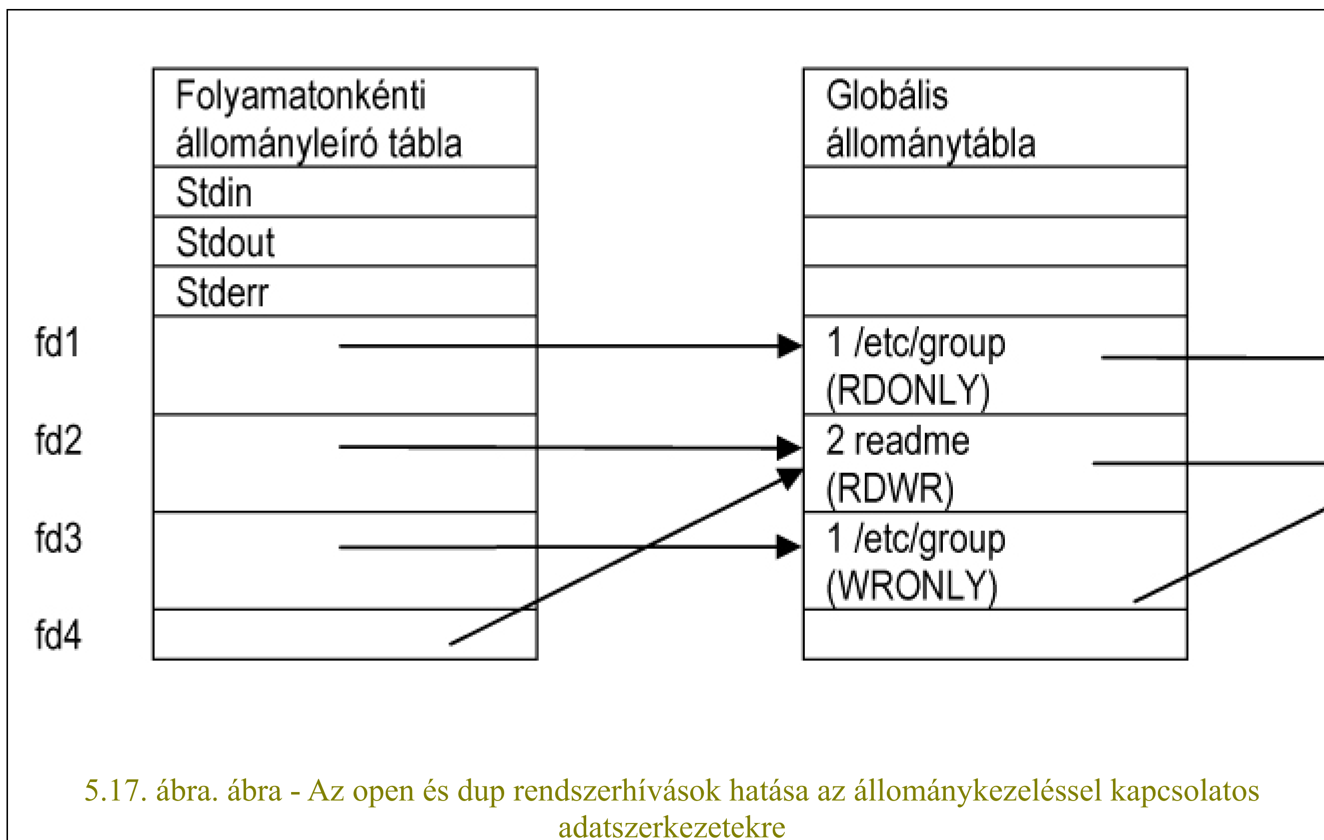
fd1 = *open*(„/etc/group”, O_RDONLY);

fd2 = *open*(„readme”, O_RDWR);

fd3 = *open*(„/etc/group”, O_WRONLY);

fd4 = *dup*(fd2);

Az 5.17. ábrán jól látszik, hogy minden egyes *open* rendszerhívás hatására a globális állománytáblában létrejön az adott állománymegnyitáshoz tartozó bejegyzés. A jelölt számok hivatkozásszámlálók, megadják, hogy a folyamatonkénti állományleíró táblák hány bejegyzése mutat az adott elemre. Bár a jobb áttekinthetőség érdekében az ábrán nem tüntettük fel, a globális állománytábla bejegyzései tárolják az állományból/ba történő olvasás/írás pillanatnyi pozícióját (fájlmutató), vagyis egy indexet, hogy hol tart az adott művelet az állományban.



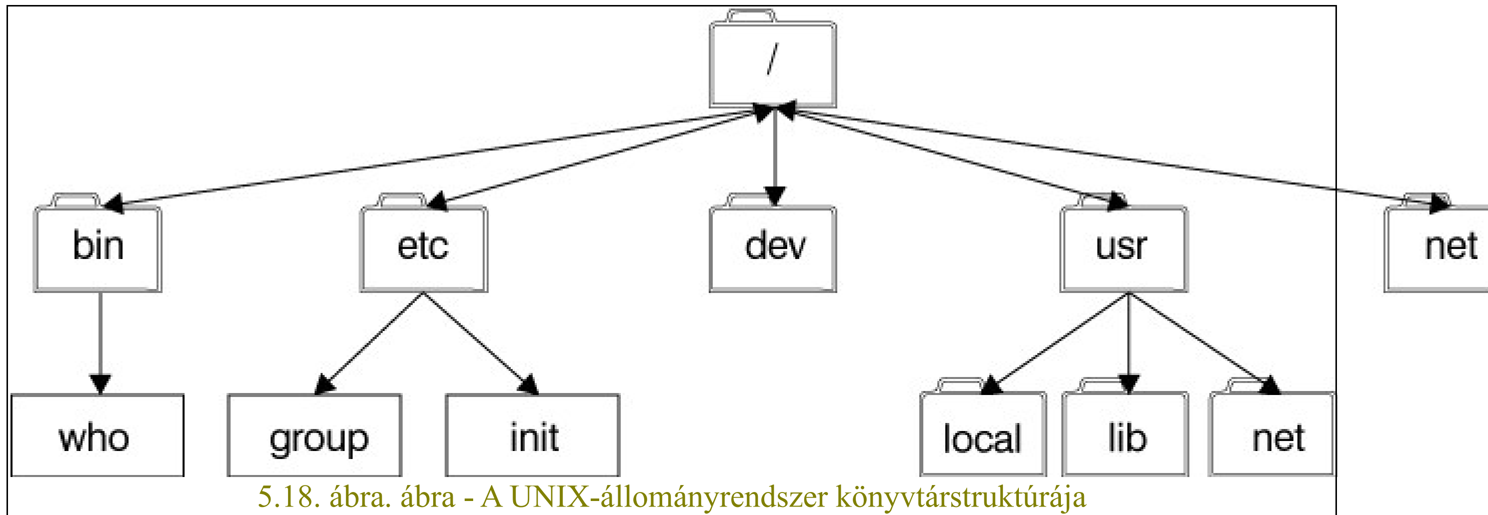
5.17. ábra. ábra - Az open és dup rendszerhívások hatása az állománykezeléssel kapcsolatos adatszerkezetekre

Az *open* rendszerhívás végrehajtásakor bejegyződik egy mutató a folyamatonkénti állományleíró táblába is, ami az adott állománymegnyitáshoz tartozó globális állománytábla bejegyzésre mutat. A *dup* rendszerhívás a paraméterül kapott állományleírót megduplázza és beírja a folyamatonkénti állományleíró tábla első szabad helyére. (Ezt a standard bemenet és kimenet átirányítására hatékonyan ki lehet használni.) Az ábrán jól látszik, hogy a *dup* rendszerhívás hatására egy újabb mutató mutat a globális állománytábla readme állományhoz tartozó bejegyzésére (ezt a hivatkozásszámláló is jelzi). Míg a globális állománytábla munkameneteket reprezentál, addig az *inode* tábla magukat az állományokat. Ebből adódóan minden egyes állományhoz és nem pedig az állomány megnyitásokhoz tartozik egy-egy bejegyzés. A fenti példa állománymegnyitásai így két új bejegyzést hoznak csak létre az *inode* táblában, hisz két megnyitás ugyanarra a fizikai állományra vonatkozik (ami persze két különböző munkamenetet jelent). A hivatkozásszámláló az egyes *inode* tábla bejegyzésekre mutató globális állománytábla bejegyzések számát mutatja.

Állományok és könyvtárak

Az állomány logikailag egy adattároló egység. A felhasználónak lehetősége van a fájl adatainak mind **szekvenciális (sequential)**, mind pedig **címzett elérésére (véletlen – random access)**, a kernel segítségével számos műveletet végezhet a fájlokon. Fontos megjegyezni, hogy a **kernel nem értelmezi az állományban tárolt adatokat**, nem feltételez semmilyen magasabb rendű struktúrát, egyszerűen byte-ok folyamának tekinti a fájlt. (Amennyiben szükség van valamilyen magasabb rendű szerkezetre – például rekordra – azt az alkalmazásoknak kell kezelni.)

A UNIX a felhasználó szemszögéből tekintve az állományokat egy **hierarchikus fa** struktúrájú névtérbe rendezi (5.18. ábra).



5.18. ábra. ábra - A UNIX-állományrendszer könyvtárstruktúrája

A nevek a / és a NULL karakter kivételével tetszőleges ASCII-karaktert tartalmazhatnak. A **gyökér** (root) **könyvtárat a** „/” név jelöli. A hierarchikus névtérből adódóan egy könyvtáron belül egyedi állományneveknek kell szerepelni, más könyvtárban azonban szerepelhet egy állomány azonos névvel (ekkor az egyediséget az eltérő elérési út biztosítja).

A UNIX támogatja az **aktuális könyvtár** fogalmát, amelyhez képest relatív elérési utat is lehet használni. Az aktuális könyvtárat a *chdir* rendszerhívással lehet megváltoztatni.

Egy állományhoz tartozó könyvtár bejegyzést az állományra mutató **hard linknek** vagy egyszerűen csak **linknek** nevezünk. Minden állományra több link is mutathat, akár más-más könyvtárakból is. Ebből adódóan a UNIX-ban egy állomány nem egyetlen adott névhez kötődik, nincs egyedi neve. A név nem az állomány attribútuma, van azonban egy ún. link számláló minden egyes állományhoz, amely megadja, hogy hány néven lehet hivatkozni az adott állományra. (Ez az *inode*-ban tárolt hivatkozásszámláló megmutatja, hogy hány könyvtári bejegyzés mutat a fájlra.) Egy állomány addig létezik, amíg a link számlálója nagyobb nullánál. Amikor a számláló nullává válik, az állomány már elérhetetlen, így az általa használt diszk terület felszabadítható. Az állomány bármelyik linken keresztül teljesen egyenrangúan érhető el.

A s5fs könyvtárszerkezete

A Unix alatt a könyvtárak is állományok, de speciális szerkezettel bírnak. Minden egyes könyvtár tartalmaz két különleges bejegyzést: a **.** és a **..** bejegyzéseket az aktuális könyvtárhoz, illetve a szülőkönyvtárhoz. A *s5fs* könyvtárak szerkezete a következő: minden bejegyzés tartalmaz egy 2 byte-os *inode* számot, amely az adott nevű állomány *inode*-ját adja meg, illetve egy 14 karakterből álló állománynevet. Így minden egyes bejegyzés 16 byte-ot foglal. Például a **/etc** könyvtár tartalmazhatja az 5.19. ábra szerinti adatokat.

Relatív byte-cím inode száma (2 Állománynév (14

	byte)	byte)
0	83	.
16	2	..
32	1798	group
48	1276	networks

Ezek után egy adott nevű állományhoz tartozó *inode* megkeresése elég egyszerű művelet: a kernel az adott könyvtárban végez egy lineáris keresést a név alapján, és a hozzá tartozó *inode*-ot a megfelelő könyvtár bejegyzésből ki tudja olvasni.

Az s5fs értékelése

Az s5fs-t az egyszerű felépítése emeli ki a többi állományrendszer közül. Ez az egyszerűség nagyon előnyös az érthetőség, áttekinthetőség és megvalósítás szempontjából, azonban a megbízhatóság, teljesítmény és funkcionalitás területén kívánnivalókat hagy maga után. A megbízhatósági problémák elsődleges forrása a szuperblokk kezelésével kapcsolatos. A szuperblokk az állományrendszer szempontjából életbevágóan fontos információkat, mint például a szabad blokkok listája, a szabad *inode*-ok száma, tartalmaz. A s5fs-ben minden egyes állományrendszer csak egyetlen példányban tartalmazza a szuperblokkot. Ebből adódóan ha a szuperblokk megsérül, a teljes állományrendszer használhatatlanná válhat.

A teljesítmény problémák is több okra vezethetők vissza. Az s5fs együtt kezeli az *inode*-okat, azokat az állományrendszer elején tárolja, majd utána következnek az adatblokkok. Ebből adódóan egy állomány hozzáférés esetén először be kell olvasni az *inode*-ot, majd utána a fizikailag távol elhelyezkedhető adatblokkot. Ez gyakran jelentős fejmozgásokat igényelhet, ami lassítja az adathozzáférést. Másrészt az *inode*-ok kiosztása is véletlenszerű, nem követi az esetlegesen meglévő logikai állománycsoportokat (például azonos könyvtárban lévő állományok). Nemcsak az *inode*-ok kiosztása véletlenszerű, hanem az adatblokkoké is. Bár kezdetben a szabad blokkok rendezetten tárolódnak, a folyamatos használat (foglalás, felszabadítás) hatására ez a rendezettség megszűnik. Egy állomány logikailag szomszédos blokkjai véletlenszerűen, akár távoli sávokon is elhelyezkedhetnek, aminek hatására a szekvenciális hozzáférés lelassulhat. A lemez blokkok mérete is problémát jelent. A korai, SVR2 rendszerekben 512 byte-os blokkokat használtak. Ennek hatására a belső tördelődést alacsony szinten lehetett tartani (átlagosan a blokkméret fele), ennek ára viszont, hogy egy lemezhozzáféréssel kevesebb adatot lehet beolvasni. A SVR3-tól kezdődően 1 K-s blokkokat használtak, ami javította az adatátvitelt, de növelte a tördelődést. Ez a probléma egy rugalmasabb allokációs módszer igényét vetette fel.

Végezetül a s5fs néhány funkcionalitásbeli problémával is küszködik. Ezek közül talán a legjelentősebb korlátozás az állománynevek 14 karakteres maximális hossza. A korai időkben ez minden bizonnyal elegendő volt, azonban manapság felettébb rontja a rendszer kereskedelmi értékét, hisz a mai felhasználók már egyre inkább megszokták, hogy hosszú, beszédes neveket adjanak az állományaiknak. Másrészt számos alkalmazás, mint például a fordítók, szövegfeldolgozók munkájuk során új kiterjesztéseket, tagokat fűznek a feldolgozandó

állomány nevéhez. Ebben az esetben a 14 karakteres állományhossz megkeseríti az ilyen alkalmazás íróinak az életét.

Mint azt korábban láttuk, a könyvtárszerkezetben minden állományhoz egy 16 byte-os bejegyzés tartozik. Ebből 14 byte az állomány neve (innen a 14 karakteres maximális állománynév korlátozás) és a maradék két byte azonosítja az állományhoz tartozó *inode*-ot. Ebből viszont következik, hogy egy állományrendszerben maximum 16 biten megkülönböztethető, vagyis 65 535 *inode* – vagyis állomány lehet. Ez a mai alkalmazások tükrében megengedhetetlen korlátozásnak tűnik.

Ezek a problémák motiválták a Berkeley Fast File System állományrendszer kidolgozását. Az alábbiakban vázlatosan kitérünk a fenti problémák megoldására.

5.3.6.2. A Berkeley FFS állományrendszer

Az FFS az s5fs számos hiányosságát próbálta meg orvosolni. A s5fs minden funkcionalitását megtartotta, a kernel adatszerkezetek nagy része, illetve a rendszerhívások kezelési algoritmusa is változatlan maradt. A legjelentősebb átalakítást a lemez használatában, a lemezen lévő adatszerkezeteken és a szabad blokk allokáción hajtották végre.

Az FFS-nél a korábban látott **partíció csoportosítás** mellett bevezették a **cilinder csoportokat (cylinder groups)**, ami valahány folytonos cilindert tartalmazott. Ezzel a kiegészítő szervezéssel lehetővé vált, hogy a Unix az egymáshoz tartozó adatokat egy cilinder csoporton tárolja, miáltal minimalizálni lehet a **fejmozgást**. (Az adatátvitel szempontjából a legjelentősebb időköltsége a fejmozgatásnak van.)

Ez a kiegészítés **újabb metaadatok** megjelenését vonta maga után, amiket cilinder csoportonként a csoport elején tárolnak. A s5fs megbízhatósági problémája elsősorban abból fakadt, hogy az állományrendszer egészéről életbevágóan fontos információkat tároló szuperblokk csak egy példányban, a **boot blokk** után került tárolásra. Az FFS-nél a **szuperblokkról** minden egyes cilinder csoport tartalmaz egy **másolatot**. Ezek a másolatok úgy vannak elhelyezve a cilinder csoportokban, hogy egyetlen sáv, cilinder vagy lemez sem tartalmazza az összes másolatot, így mechanikai sérülésekkel szemben is kellő védelmet biztosít.

Mint azt már láttuk, a blokkméret meghatározásakor ellentmondó szempontokat kell figyelembe venni. Minél nagyobb blokkokat használunk, annál jobb az adatátviteli sebesség, viszont annál nagyobb a belső tördelődés. Az FFS ezt a problémát **töredékek (fragment)** alkalmazásával próbálja orvosolni. Egy állomány minden blokkja azonos méretű (a minimális blokkméret 4 kB), azonban az utolsó blokk tartalmazhat egy vagy több, külön címezhető és allokálható folyamatos töredékblokkot. Meg kell jegyezni, hogy a töredékblokkok maximális száma az állományrendszer létrehozásakor kerül rögzítésre: 1, 2, 4 vagy 8 lehet. Ezzel a struktúrával 4k-s blokkokat és 8 töredékblokkot alkalmazva 512 byte-os alsó határt lehet elérni, ami megegyezik a szektormérettel. A nagyobb blokkméret alkalmazásának egy járulékos előnye, hogy 2^{32} byte (4 GB) méretű állományokat is két indirekciós szinten lehet címezni. Az FFS általában ebből kifolyólag nem is alkalmazza a háromszoros indirekt blokkokat.

A lemezblokk töredékek használatára érvényes néhány korlátozás. Egy állományblokknak egyetlen lemezblokkban kell lennie. Szomszédos lemezblokkok összefüggő töredékei nem vonhatók össze. Továbbá, ha egy állomány utolsó blokkja több töredéket is tartalmaz, ezen töredékeknek összefüggőeknek kell lenniük és azonos lemezblokkon kell elhelyezkedniük. Ezen adatszerkezet csökkenti a lemez tördelődését, használatához azonban módosítani kell a szabad lemezblokkok nyilvántartását: a listát le kell cserélni az összes töredékblokkot nyilvántartó bittérképre. A többlet adminisztráció mellett bizonyos esetekben adatok átmásolására is szükség van. Amikor például egy olyan állományt próbálunk meg növelni, amelynek utolsó blokkja egyetlen töredéket foglal el, és ugyanazon blokk többi töredéke más állományokhoz tartozik, akkor először allokálni kell egy új blokkot, ami több szabad töredéket tartalmaz, majd a töredéket át kell másolni. Lassan, kis lépésekben bővülő állomány esetén számos ilyen másolásra lehet szükség. Ebből adódóan a jobb hatékonyság érdekében az alkalmazások, amikor csak lehetséges, teljes blokkokat kell hogy az állományba írjanak.

Lemezblokk allokációs politika

Az FFS lemezblokk allokációs politikájának célja, hogy az egymáshoz kapcsolódó információkat egy helyen tárolja, és a szekvenciális hozzáférésre optimalizálja az allokációt. A s5fs-nél láttuk, hogy listában tárolja a szabad blokkokat és *inode*-okat. Bár kezdetben ezek lemezelfordulás szempontjából optimalizáltak voltak, a használat során teljesen véletlenszerűvé válnak, gyakorlatilag az operációs rendszernek semmilyen befolyása sincs az allokálandó adatblokk vagy *inode* elhelyezkedésére. Ezzel szemben az FFS itt is a cilindercsoportokat alkalmazza. Az alábbiakban a teljesség igénye nélkül felsorolunk néhány szempontot, amit az FFS az allokáció során figyelembe vesz.

- Az FFS az egy könyvtárban lévő állományok *inode*-ját megkísérli ugyanarra a cilindercsoportra helyezni. Ennek ésszerű magyarázata, hogy számos parancs, mint például az *ls* is, gyors egymásutánban olvassa a könyvtár *inode*-jait, így ha azok egy cilindercsoporton helyezkednek el, a fejmozgások csökkenthetők.
- Minden új könyvtárat más cilindercsoportra helyez, hogy az adatokat egyenletesen terítse szét a lemezen.
- Egy állományhoz tartozó adatblokkokat megpróbálja ugyanarra a cilindercsoportra helyezni, ahol az állomány *inode*-ja is található, hisz tipikusan az *inode*-ot és az adatokat együtt használja.
- A nagy állományokat „szétkeni” a cilindercsoportokon. Amikor az állományméret eléri a 48 kB-ot, új cilindercsoportra lép, majd 1 MB után újra stb. Ezzel meg tudja akadályozni, hogy egy óriási állomány feltöltse a teljes cilindercsoportot. A 48 Kb-os határ a direkt blokkokon tárolható méretet jelöli (az FFS-nél 12 db direkt blokk található az *inode*-ban), így a direkt blokkokban tárolt adatok azonos cilindercsoportra kerülnek az *inode*-dal.
- Az egymást követő blokkokat – amennyiben az lehetséges – lemezelfordulás szerint optimálisan allokálja.

-

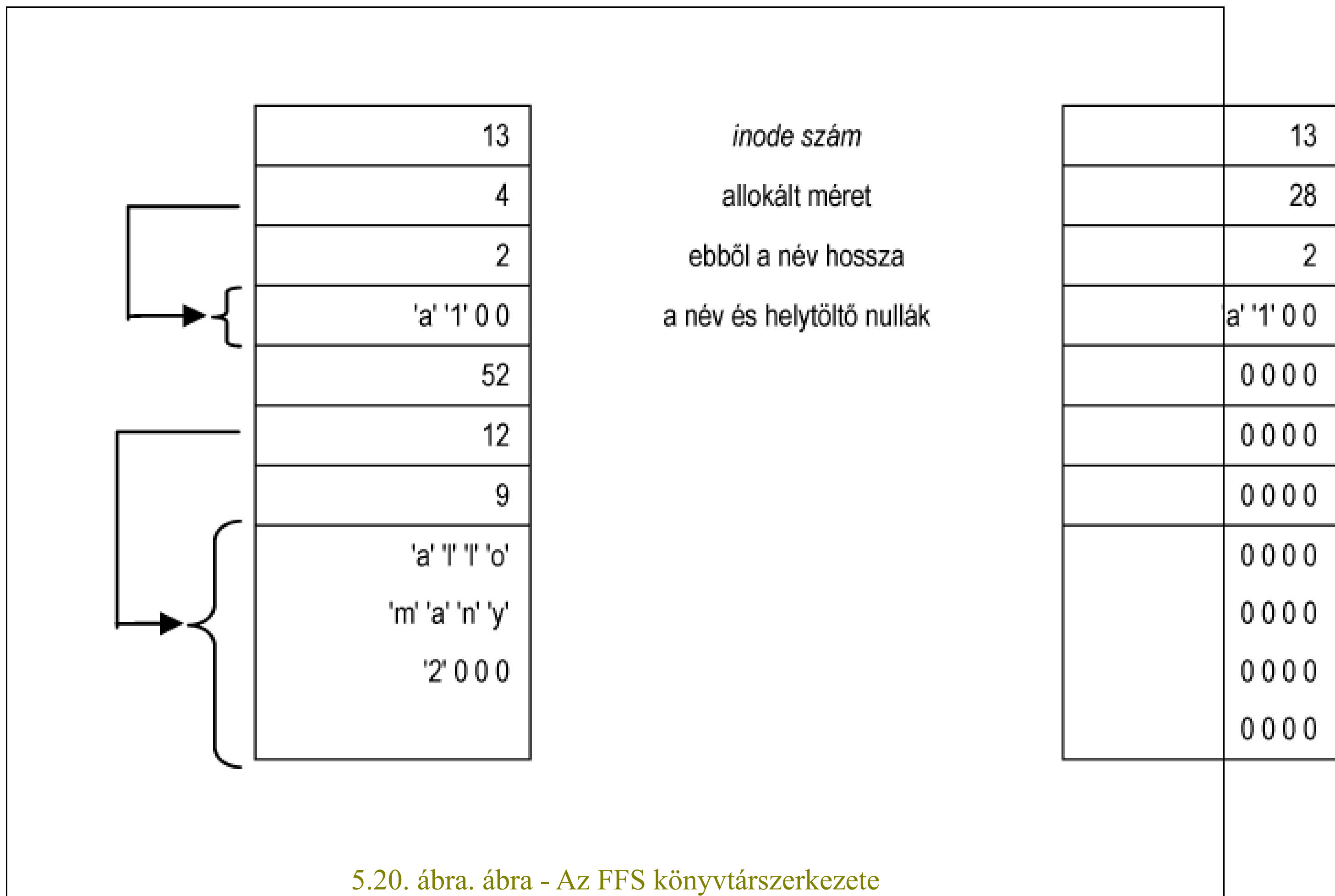
Funkcionális bővítések a s5fs-hez képest

Hosszú állománynevek

Mint azt láttuk, a 16 bájtos könyvtár bejegyzés miatt a *s5fs*-nél a maximális állománynév 14 karakter lehetett. Ez a mai alkalmazásoknál elég húsbavágó korlátozást jelent, így az FFS megalkotói egy eltérő könyvtárszerkezetet dolgoztak ki, ami feloldja ezt a korlátozást. A könyvtárbejegyzés egy rögzített és egy változó részből áll. A rögzített rész tartalmazza az *inode* számát, az allokált területet, illetve, hogy ebből a területből maga az állománynév mennyit foglal le. A rögzített részt követi a változó hosszúságú rész, ami a null karakterrel lezárt állománynevet tartalmazza, négy byte-os margóra igazítva. (5.20. ábra). Ebben az esetben a maximális állománynév hossz 255 karakter. Egy probléma merül fel ezzel a szerkezettel kapcsolatban: mit kell tenni, ha egy könyvtárbejegyzést törlünk? Ekkor a rendszer a törölt bejegyzés területét hozzáolvasztja az előző bejegyzés területéhez, a változást feltüntetve az allokált terület nagyságában (5.20. ábra jobb oldala).

Egyéb bővítések

Az FFS vezette be a **szimbolikus link** fogalmát. A szimbolikus linkek kiküszöbölik a **hard linkek** számos korlátját. A szimbolikus link tulajdonképpen egy olyan speciális állomány, ami egy másik állományra, az ún. célállományra mutat. A szimbolikus linket az *inode típus* mezője alapján lehet felismerni, és tulajdonképpen csak a célállomány elérési útját tartalmazza.



5.20. ábra. ábra - Az FFS könyvtárszerkezete

Továbbá az FFS bevezette a **kvóta mechanizmust**, amivel korlátozni lehet az egyes felhasználók rendelkezésére álló állományrendszer erőforrásokat.

Az FFS teljesítménye

Az FFS teljesítménye jelentősen túlszárnyalja a s5fs teljesítményét. Mérések szerint a s5fs olvasás esetén a megközelítőleg 30 Kb/s-os átbocsátóképességét (1 Kb-os blokkok használata mellett) az FFS megközelítőleg 220 Kb/s-ra javította (4 Kb-os blokkokat és 1 Kb-os töredékblokkokat alkalmazva). A CPU-kihasználtság is jelentősen növekedett: a s5fs körülbelüli 1%-os kihasználtságát sikerült majd megnégyszerezni. Írás esetén az eltérés nem ennyire domináns, de ott is jelentős. (A megadott adatok VAX/750 hardveren elvégzett mérésekből származnak.)

5.3.6.3. Az állományrendszerek megvalósításának újabb megközelítése

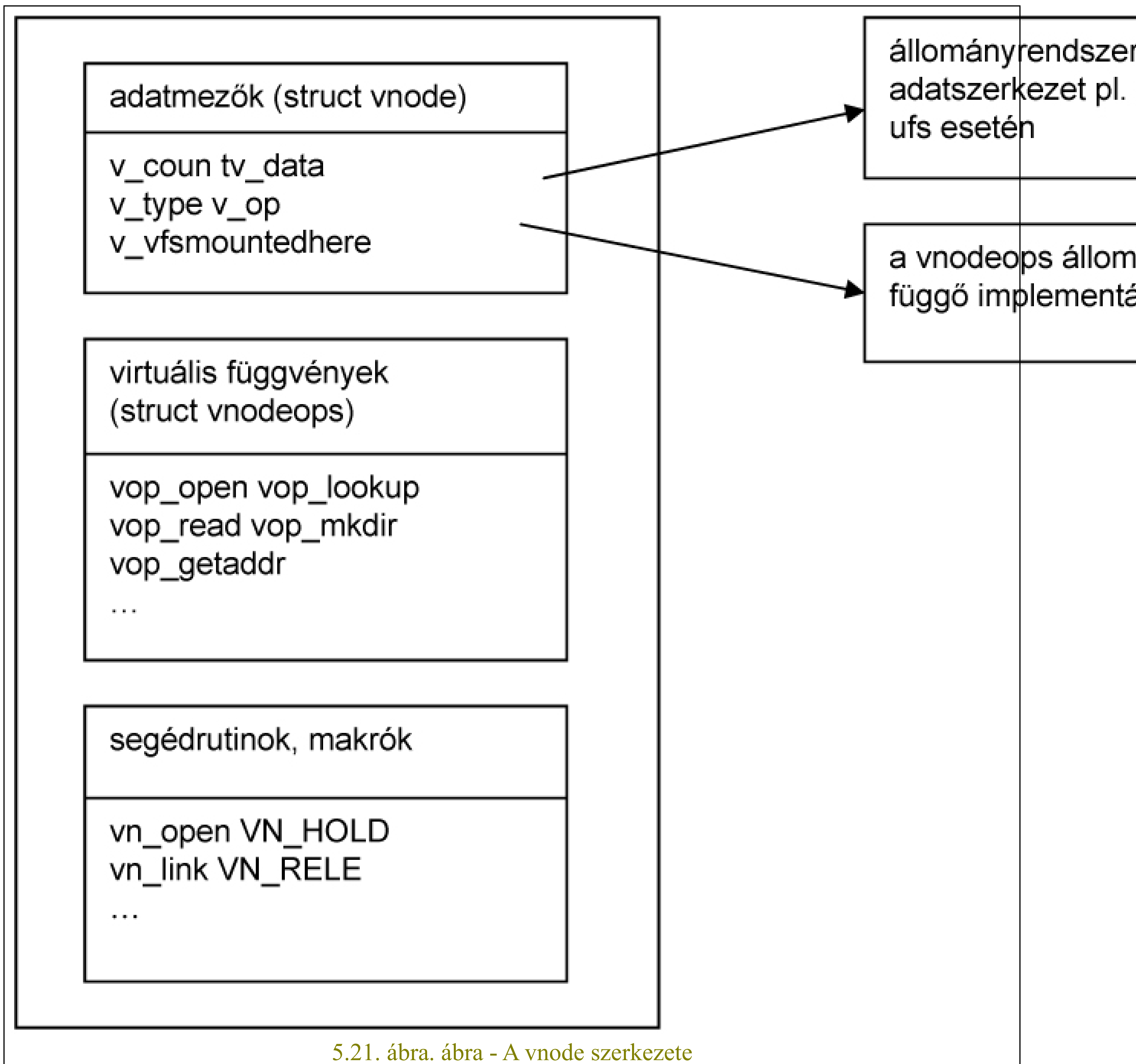
A korai UNIX-rendszerek azt a filozófiát követték, hogy egy rendszerben csak egyetlen állományrendszer létezik. Ebben a megközelítésben az állományok leírására elégséges volt az *inode* absztrakció. Hamar felismerték azonban, hogy ez a korlátozás indokolatlan, több állományrendszer (beleértve a több típust is)

számos előnnyel kecsegtet. A megvalósításához azonban már nem elegendők az *inode*-ok. Az új leíró adatszerkezet a **virtuális csomópont** *(vnode)* és a **virtuális állományrendszer** *(vfs)*.

Az új absztrakció bevezetésének célja, hogy

- egyszerre támogasson több állományrendszert (UNIX, nem UNIX),
- különböző diszk partíciók tartalmazhassanak különböző állományrendszert, viszont csatlakoztatás (mount) esetén egységes „képet” kell hogy mutassanak,
- támogassa a hálózaton történő állományok osztott használatát,
- modulárisan bővíthető legyen.

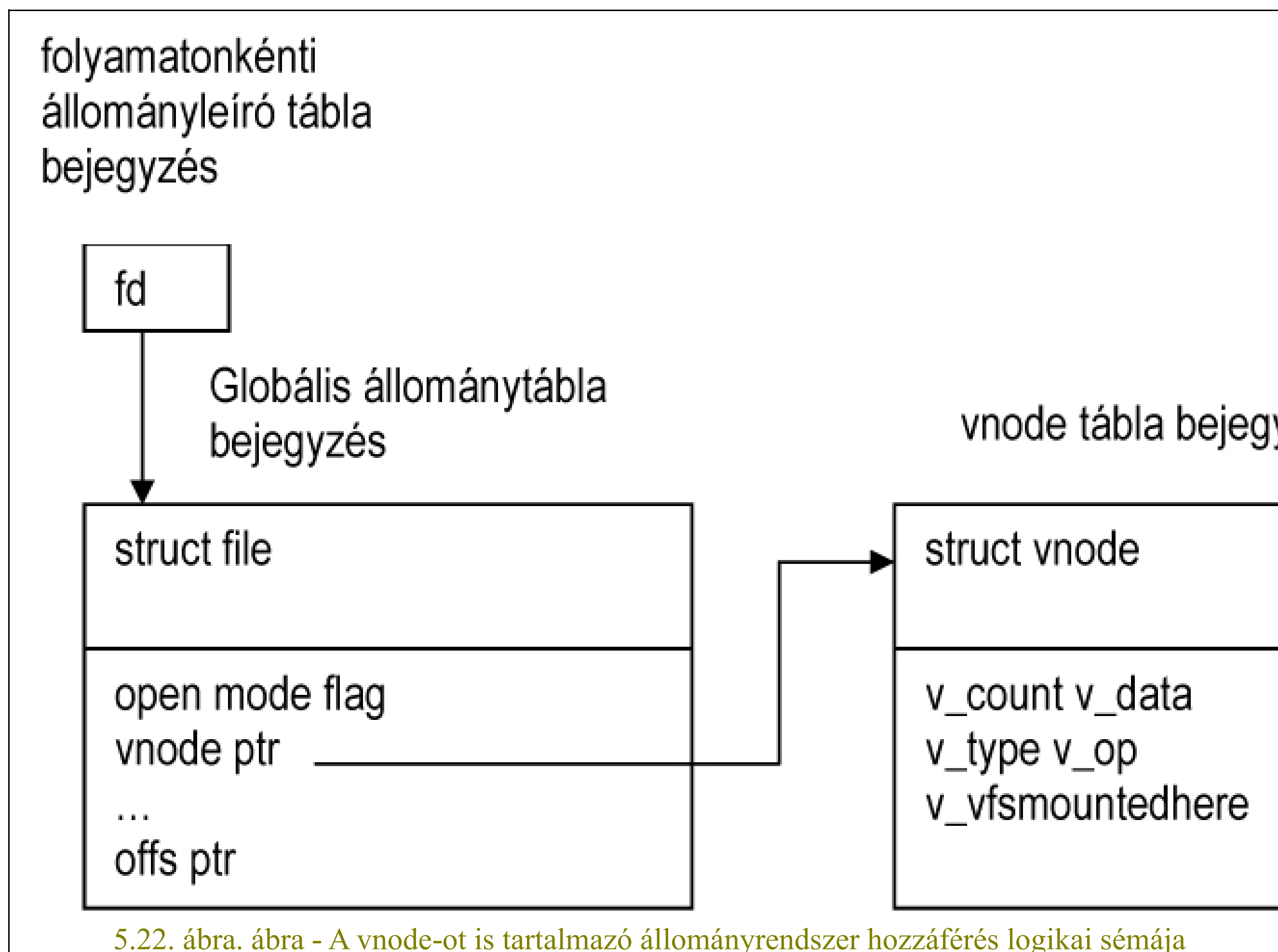
Az absztrakciót tulajdonképpen két adatszerkezet a *vnode* és a *vfs*, illetve azok kezelési módja valósítja meg. Először vizsgáljuk meg az *inode* szerepét átvevő vnode-ot. Az 5.21. ábra mutatja a *vnode* szerkezetét. Mint az az ábrán jól látható, az adatszerkezet három részre tagolódik: az adatmezőkre, a virtuális függvényekre valamint a segédrutinokra, és makrókra. Az adatmezők egyes elemei töltik be az *inode* megfelelő funkcióinak a szerepét: megjelenik a típusmező (v_type), ami a *vnode* által leírt állomány típusát adja meg, található itt hivatkozásszámláló (v_count), mount információ (v_vfsmountedhere) stb. Ezenfelül megtalálható itt két mutató (v_data, v_op), amelyek implementációfüggő valódi, a virtuális csomópont által elfedett adatszerkezetekre mutatnak, amelyekre a konkrét állományrendszerek kezeléséhez, karbantartásához van szükség. A v_data mező *s5fs* esetén pontosan egy *inode*-ra mutat. A v_op mező által mutatott táblában pedig a valódi adatszerkezeteken az absztrakt állományműveleteket konkrétan végrehajtó függvények címei találhatók.



5.21. ábra. ábra - A vnode szerkezete

A *vnode*-ban megtalálható virtuális függvények szerepe, hogy az állományműveleteket konkrét állományrendszer implementációtól függetlenül ki lehessen adni és majd a *vnode* kezelő mechanizmusa gondoskodik róla, hogy az adott absztrakt műveletet megvalósító konkrét, az adott állományrendszerhez tartozó állományművelet hajtódjon végre. Ehhez a *vnode* absztrakció definiál egy művelethalmazt (állomány megnyitás, olvasás, írás stb.) amit minden konkrét állományrendszer implementációban meg kell valósítani. Így egy indexelési és mutató indirekción keresztül el lehet jutni a konkrét műveletet megvalósító függvényhez.

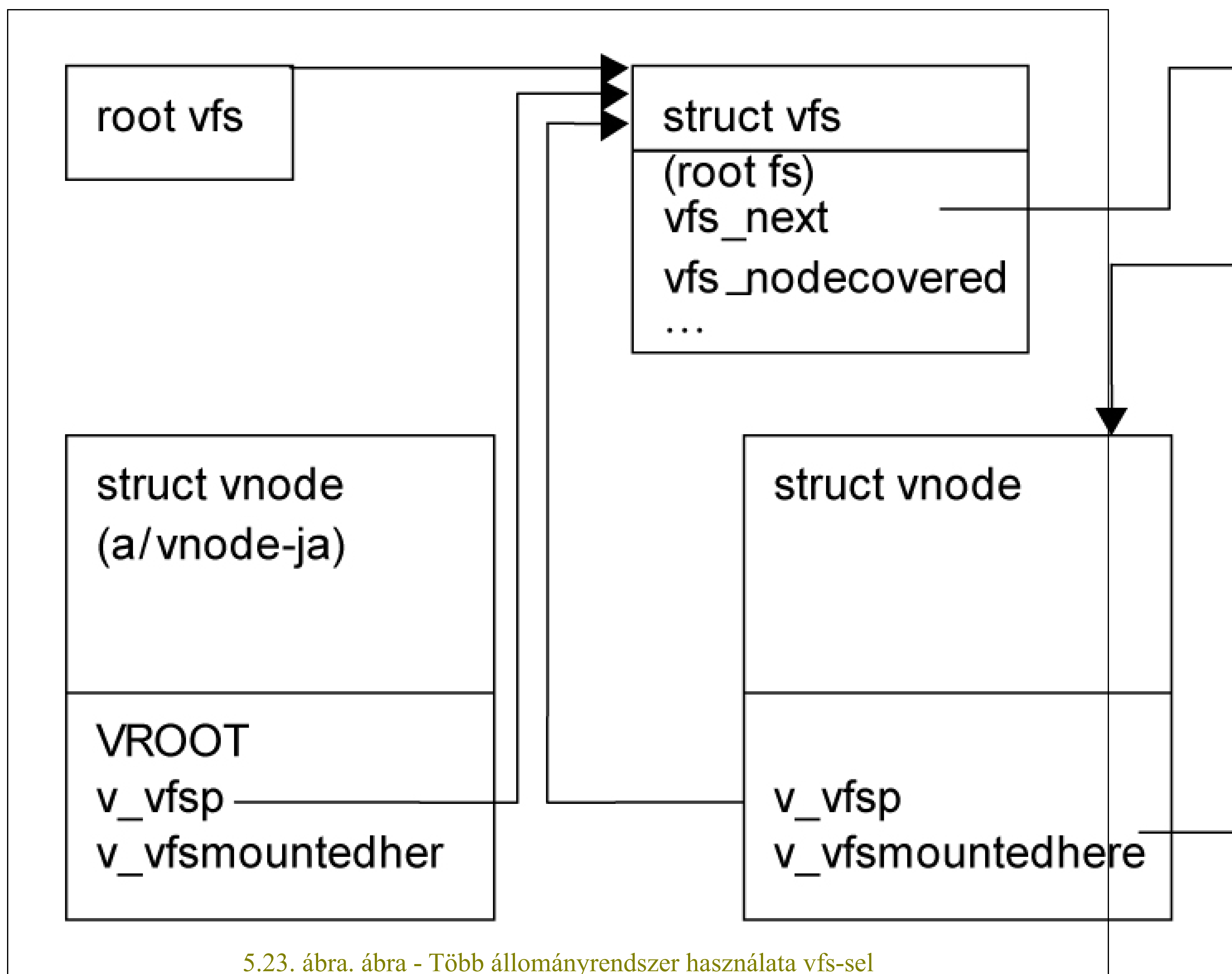
A *vnode* használatával a korábban látott kettős indirekció (folyamatonkénti állományleíró tábla, globális állománytábla bejegyzés, *inode* tábla bejegyzés) egy újabb elemmel bővül (5.22. ábra).



5.22. ábra. ábra - A vnode-ot is tartalmazó állományrendszer hozzáférés logikai sémája

A virtuális állományrendszer megvalósításához egy, a *vnode*-hoz hasonló adatszerkezetet, a *vfs*-t használják. Ennek felhasználásával több eltérő típusú állományrendszert egységesen lehet kezelni az alábbi módon (5.23. ábra).

Az 5.23. ábrán jól látszik, hogy van egy *root* állományrendszer, amire a többi állományrendszert csatlakoztatni lehet (fel lehet mount-olni). A rendszer minden egyes állományrendszer típushoz tartalmaz egy *vfs* struktúrát, ami tárolja az állományrendszerhez kapcsolódó, az állományrendszer kezeléséhez elengedhetetlen információkat. Továbbá minden egyes olyan virtuális csomópontban *(vnode)*, amely egyben egy állományrendszer gyökér csomópontja, a VROOT jelzőbit be van billentve.



5.23. ábra. ábra - Több állományrendszer használata vfs-sel

5.3.6.4. Speciális állományrendszerek

Az *s5fs* és az FFS állományrendszer a UNIX **általános célú lokális állományrendszerei**. Amennyiben az állományrendszernek távoli hozzáférést is támogatnia kell, akkor a kialakítását ezekhez az igényekhez kell alakítani. A **távoli állományrendszerekkel** a 4.4.2. rész foglalkozik.

A továbbiakban vázlatosan ismertetünk néhány speciális állományrendszert, amelyek egy-egy adott speciális igényt szolgálnak ki, a felépítésük ezekhez az igényekhez alkalmazkodott.

Ideiglenes állományrendszer

Számos program – főként a fordítóprogramok és az ablakkezelők – nagyon kiterjedten használ **ideiglenes állományokat (temporary files)** közbülső eredményeik, állapotuk tárolására. Ezekre az állományokra az alkalmazás befejeződése után már nincs szükség, nem kell túlélniük egy rendszer összeomlást. Ezen állományok

létrehozásának, hozzáférésének és törlésének a használatukból adódóan gyorsnak kell lennie. Már a korai rendszerekben is a kernel az ideiglenes állományok helyett a memóriában található **blokk buffer cache**-be írt, így késleltetve a valódi állományba írást. Ezzel a hozzáférés jelentősen gyorsult, azonban az állomány létrehozása és törlése a többszöri szinkron lemezhozzáférés miatt még mindig felettébb lassúnak számított.

Ezt a problémát a **RAM-diszkek** alkalmazásával küszöbölték ki. Mivel ekkor az adatok a fizikai memóriában kerültek tárolásra, így a hozzáférések jelentősen felgyorsultak. A RAM-diszk nagy sebessége azonban nem igazán tudja kompenzálni azt a kellemetlen tulajdonságát, hogy nagyon pazarlóan bánik a globális erőforrásokkal, vagyis elsősorban a memóriával. A rendszer működése során változóak az igények az ideiglenes állományrendszerrel szemben, sajnos azonban a RAM-diszk számára fenntartott memóriát a rendszer külön kezeli, így azt kisebb igények esetén sem lehet más célra felhasználni.

A **Memória Állományrendszer** (Memory File System – mfs) ezt a problémát próbálta orvosolni. Az állományrendszert egy folyamat, méghozzá az állományrendszert csatlakoztató (*mount*-oló) folyamat virtuális címtartományában építették fel. Mivel az állományrendszer egy folyamat virtuális címtartományában van, így a standard memóriakezelő mechanizmusokkal éppúgy ki lehet lapozni háttértárra, mint bármilyen más adatot. Mivel ebben a megvalósításban egy külön folyamat kezelt minden B/K-műveletet, így mindig két környezetváltásra van szükség.

Ezzel szemben a teljesen kernelben megvalósított, elterjedten használt ***tmpfs*** nem használ külön B/K-szervert, így elkerüli a felesleges környezetváltásokat. Mivel a metaadatokat nem kilapozható memóriában tárolja, így számos memória-memória másolást és néhány lemezműveletet is el lehet kerülni. Ezenfelül, mivel támogatja a memória leképzést, így gyorsan és közvetlenül is hozzá lehet férni az állomány adatokhoz.

A ***specfs* állományrendszer** a felhasználók számára láthatatlan módon egységes interfészt biztosít a készülék állományokhoz, ami jelentősen megkönnyíti a készülékek kezelését.

A /proc állományrendszer

A **/proc állományrendszer** elegáns és hatékony interfészt biztosít minden folyamat címteréhez. Eredetileg a hibakeresés támogatására dolgozták ki (hogy leváltsa a kényelmetlenül használható *ptrace* funkciót), azonban egy általános interfésszé nőtte ki magát a folyamat modellhez. A megközelítés nagy előnye, hogy a standard állományrendszer interfész segítségével a felhasználók folyamatok címteréből olvashatnak és módosíthatják azt.

A jogosultságokat a közönséges állományoknál megszokott jogosultságkezelési mechanizmussal lehet szabályozni. Az állományrendszer bevezetésekor minden egyes folyamathoz egy-egy bejegyzés tartozott a */proc* könyvtárban. A bejegyzések nevei megegyeztek a folyamatazonosítókkal. A későbbiek folyamán finomították a modellt és minden folyamatot egy könyvtár jelölt a */proc* könyvtárban, aminek a neve továbbra is a folyamat azonosítója maradt. A könyvtáron belül a logikailag eltérő funkciókhoz egy-egy külön állomány jelent meg, mint például a folyamat állapotát, a virtuális címtérképét, jelzés információkat stb. leíró állományok.

A Processzor állományrendszer

A **processzor állományrendszert** a többprocesszoros környezet és az ezekhez alkalmazkodó operációs rendszerek hívták életre. Az állományrendszer egy interfészt biztosít egy többprocesszoros számítógép egyes processzoraihoz. A */system/processor* könyvtárra mount-olódik fel és a rendszer minden processzorához tartalmaz egy állományt. Az állomány a processzorral kapcsolatos legfontosabb információkat tartalmazza, mint például a CPU típusa, a CPU sebessége, a cache mérete, a hozzá kapcsolódó speciális berendezések stb.

5.3.6.5. Modern állományrendszerek

Az operációs rendszerek tervezésekor figyelembe veszik a hardver adottságokat, maximálisan megpróbálják kiaknázni a bennük rejlő lehetőségeket, illetve a fejlesztés során hozott döntések magukon viselik az adottságok hatását. A UNIX kialakulásakor a tervezési döntéseket erősen befolyásolta, hogy a korai ’80-as években viszonylag kis memóriára, lassú processzorra és relatíve gyors lemezegységekre kellett alapozni a tervezést. Időközben a hardverfejlesztés eltérő léptékben haladt előre, a memória mérete jelentősen megnőtt, a CPU sebessége közel három, míg a memória mérete közel két nagyságrenddel megnövekedett, ezzel szemben bár mérete jelentősen nőtt, a lemezegység sebessége alig duplázódott meg. Ebből adódóan a UNIX belső szerkezete szempontjából megváltozott hardverfeltételek miatt számos tervezési döntés elvesztette létjogosultságát, azokat az új feltételeknek megfelelően újra át kellett gondolni.

A lemezegység sebességének lassú növekedése elsősorban az állományrendszert érinti hátrányosan. A hagyományos állományrendszereket a mai számítógépeken használva erősen B/K-korlátozott rendszerek jönnek létre, ami gátolja a jelentős CPU-sebességnövekedés kiaknázását.

A sebesség problémák kiküszöbölésére számos modern állományrendszer alkalmazza az ún. **naplózási (journaling** vagy **log) technikát**. A megközelítés lényege, hogy minden állományrendszer változást a rendszer egy csak hozzáfűzhető állományba naplóz, vagyis az egész állományrendszer tulajdonképpen egy hatalmas napló állomány. A naplót a rendszer szekvenciálisan írja, nagy darabokban, ami jelentősen javítja a lemezkihasználtságot és a teljesítményt. Továbbá egy rendszer összeomlás után a napló állománynak csak a végét kell vizsgálni, ami a felépülést jelentősen felgyorsíthatja.

Egy **naplózó állományrendszer** tervezésénél is számos döntést kell meghozni. Ezek közül az alábbiakban a teljesség igénye nélkül vázoljuk a legfontosabbakat:

- mit írjunk a napló állományba, műveletek vagy értékek,
- kiegészítő állományrendszerként alkalmazzuk, vagy cseréljük le a korábbi állományrendszert,
- a változtatásokat újra lejátszva (REDO) vagy a legutóbbi változtatásokat visszavonva (UNDO), állítsuk vissza a korábbi értékeket,
- milyen szemétgyűjtési stratégiát alkalmazzunk,
- hogyan alkalmazzuk a csoportvéglegesítést,
- hogyan keressük vissza az adatokat.

A hely szűke miatt itt nem tárgyaljuk részletesen az implementációt, csak felhívjuk a figyelmet a naplóalapú állományrendszer alkalmazásának néhány nyilvánvaló előnyös tulajdonságára, illetve a megoldandó problémákra.

Az előnyök eléggé szembeötlők. Mivel mindig a napló állomány végére írunk, így az írás mindig szekvenciális és nincs szükség fejmozgatásra. A naplóba írás során egyszerre mindig nagy mennyiségű adatot írunk, tipikusan egy egész sávot, ebből adódóan nem kell elfordulás alapján optimalizálni, illetve a lemezegység teljes sávszélességét ki lehet használni. Egy művelet minden adat összetevőjét (metaadatot és valódi adatot) össze lehet fogni egyetlen **atomikusan végrehajtott** írásba, ami jelentősen növelheti az állományrendszer megbízhatóságát. Az írás tehát nem okoz problémát.

Viszont a rendszer hatékonysága szempontjából nagy gondot kell fektetni az adat visszakeresés problémájára. Az adatokat a hagyományos módszerrel (állandó helyen tárolt szuperblokk, *inode* stb.) már nem lehet kezelni, a napló állományban keresést kell végrehajtani. Ezt a keresést jelentősen támogatja a memóriában történő *cache*-elés, ami a mai memóriaméretek mellett nem jelent problémát. Ettől függetlenül az állomány szerkezetének támogatni kell a napló állományban történő keresést, mert enélkül a rendszer használati értéke jelentősen csökken.

Számos napló alapú rendszert dolgoztak ki, mint például a 4.4BSD naplóalapú állományrendszere, metaadat naplózó rendszerek, az Episode állományrendszer. Ezeket részletesen az irodalomjegyzékben felsorolt források tárgyalják.

5.3.7. Teljes folyamatok háttértárra írása (swapping)

A korai UNIX-rendszerekben a virtuális memóriakezelést **teljes folyamatok háttértárra írásával** oldották meg (**swapping** rendszerek). Ennek történeti okai vannak: az első UNIX-rendszerek PDP-11-en futottak, ahol a folyamatokra 64 K-s memóriakorlát volt kiszabva. Ilyen kisméretű folyamatokat viszonylag gyorsan ki lehetett írni háttértárra. Nagy folyamatok esetén (manapság egy-egy folyamat virtuális memóriaméretére nincs elvi korlátozás, gyakoriak a több Mbyte-os folyamatok) a háttértárra írás időigényes, így önmagában, egyedüli módszerként nem használatos, azt egy igény szerinti lapozási technikával ötvözve alkalmazzák.

Bár a teljes folyamatok háttértárra írása **(tárcsere)** veszített a jelentőségéből, a modern rendszerek is alkalmazzák, hogy gyorsan orvosolják a leterhelt rendszereknél gyakran jelentkező erőforrás szűke okozta hatékonysági problémákat.

A továbbiakban áttekintjük, hogy a tárcsere rendszer megvalósítása milyen feladatokat ró az operációs rendszerre, és hogy ezeket a feladatokat hogyan oldják meg.

A swapping rendszerekben három, logikailag elkülönülő feladatot kell megoldani:

- a háttértár szervezését (swap device),
- folyamatok háttértárra írását,

- folyamatok háttértárról memóriába történő beolvasását.

A továbbiakban ezeket a feladatokat tárgyaljuk.

5.3.7.1. A háttértár szervezése

A **háttértár (swap device)** egy blokkos eszköz, általában merevlemez. Ellentétben a Unix állományrendszerrel (amely egy lépésben egy blokkot foglal le), a háttértárra íráskor a rendszer a folyamatoknak összefüggő blokkcsoportot foglal. Ez persze a rendelkezésre álló terület tördelődéséhez (fragmentation) vezet, de ebben az esetben ez nem jelent olyan nagy problémát, mert a háttértáron a folyamatok csak ideiglenesen tartózkodnak, onnan időről időre bekerülnek a memóriába, és ekkor a háttértáron használt terület felszabadul. A tördelődésnél fontosabb, kritikus probléma a háttértárra írás sebessége. Folyamatok háttértárra írása és onnan történő beolvasása nagyon gyakori esemény volt a korai rendszerek működése során, így a műveletek sebessége valóban létfontosságú volt. Az is maradt mindmáig, mert a gyakoriság ugyan csökkent, de a processzor és a háttértár közötti sebességkülönbség jelentősen növekedett.

A rendszer a háttértáron rendelkezésre álló szabad területeket egy **memóriában tárolt táblával** (ún. *map*-pal) tartja nyilván. A *map* szerkezete egyszerű: a bejegyzések (cím, szabad blokkok száma) alakú rendezett párok. Itt a cím a háttértár blokk relatív címe a swapping területen (a logikai swap device-on – a számozás 1-gyel kezdődik). A táblát a rendszer mindig a lehető legtömörebben tartja, felesleges bejegyzést nem tartalmaz. Például, feltételezve egy 100 000 blokkot tartalmazó háttértárat, azt a pillanatot, amikor a teljes háttértár üres, vagyis minden blokk szabad, az alább látható *map* írja le.

Szabad terület kezdőcíme	Az összefüggő szabad terület mérete (blokkokban)
1	100000

5.26.ábra. táblázat - A laptábla-bejegyzés által tárolt információk. A bejegyzést a folyamat virtuális címei címzik. Jelölések: Age – öregítés bit(ek), C/W – copy-on-write bit, Mod – módosítás bit, Ref – hivatkozás bit, Val – érvényességi bit,

A *map* egy dinamikusan bővülő, illetve szűkülő szerkezet, vagyis működés közben a táblába új sorok szúródnak be, illetve sorok törlődnek. Mivel a gyorsaság a legfontosabb tervezési szempont a háttértár kezelésében, ezért a foglalási stratégia is ezt tükrözi: a stratégia **first-fit**, vagyis az első megfelelő méretű területet használja fel a rendszer. A *map* struktúráját végiggondolva jól látható, hogy az a first-fit stratégiát támogatja.

5.3.7.2. A háttértár-foglalási és -felszabadítási algoritmus

A háttértár foglalása és felszabadítása során a kernel megpróbálja a map-ot a lehető **legtömörebben** tartani. Foglaláskor, ha egy teljes map bejegyzést foglalunk, akkor a kernel törli a bejegyzést (nem hagy meg 0 méretű bejegyzést). Felszabadításkor a kernel megvizsgálja, hogy a felszabadított terület pontosan beilleszkedik-e két map bejegyzés közé. Amennyiben igen, akkor a nagyobb indexű bejegyzést törli, és az alacsonyabb indexű

bejegyzés által megadott szabad blokkok számát úgy módosítja, hogy az lefedje a kisebb indexű bejegyzés, a felszabadított terület és a nagyobb indexű bejegyzés szabad blokkjait. Amennyiben a felszabadított terület nem tölt ki teljesen egy „lyukat” a *map* táblában, a kernel megvizsgálja, hogy a felszabadított terület illeszkedik-e valamelyik korábbi bejegyzéshez (egy korábbi *map* bejegyzéshez a felszabadított terület alulról vagy felülről illeszkedhet). Amennyiben igen, módosítja a már meglévő bejegyzést. Amennyiben a felszabadított terület egyetlen bejegyzéshez sem illeszkedik, akkor a kernel új bejegyzést hoz létre.

A korai UNIX-rendszerekben csak egy háttértár volt, manapság már a kernel több *swap* berendezést is használhat. Ezeket a rendszer konfigurálásakor a rendszergazda adja meg. Amennyiben több *swap* berendezés is van, a kernel azokat **round–robin-**stratégiával használja.

5.3.7.3. Folyamatok háttértárra írása

Az operációs rendszer működése során négy esetben lehet szükség folyamatok háttértárra írására:

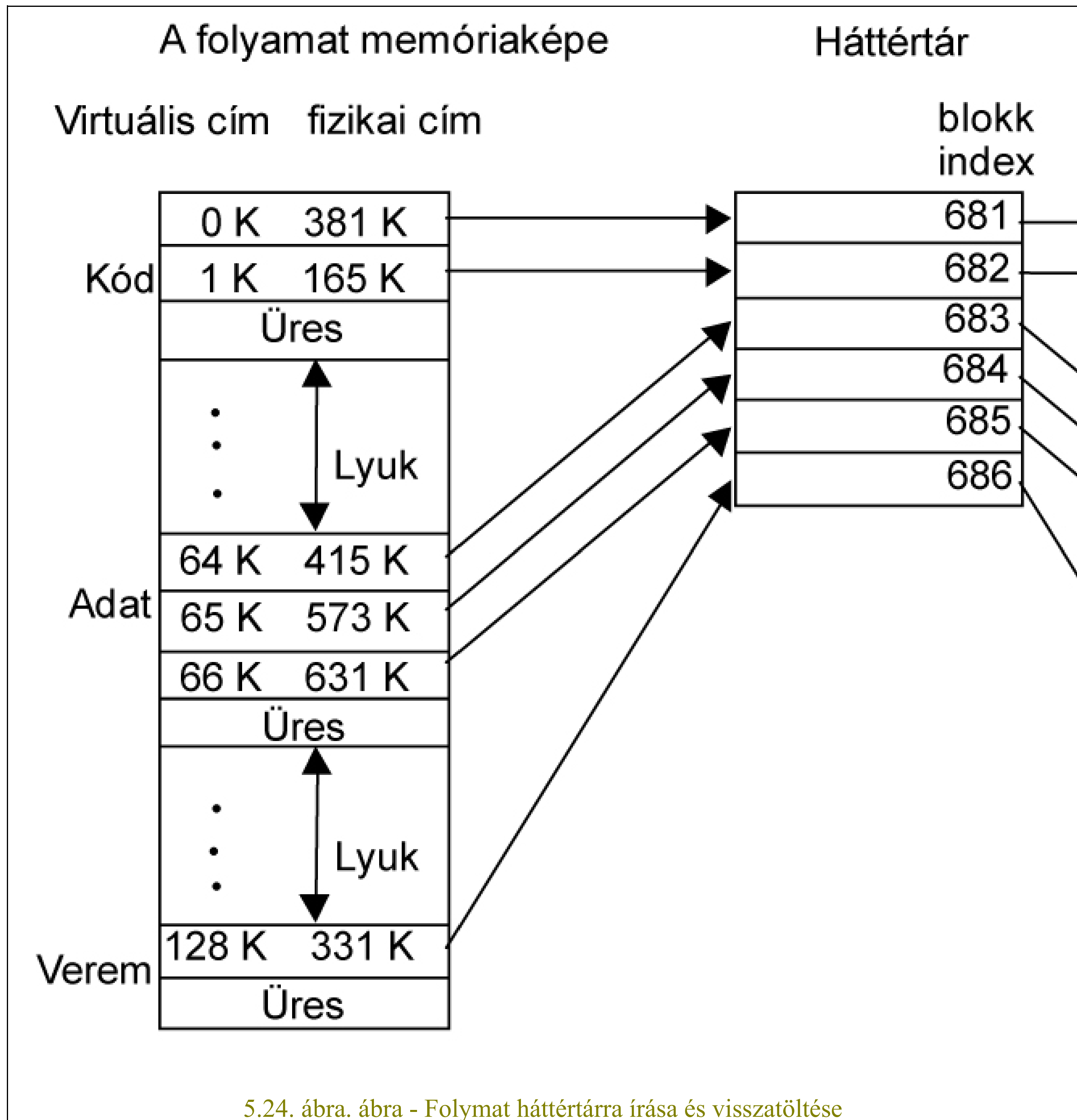
1. a kernel *fork* rendszerhívást hajt végre – új folyamatot hoz létre, és nincs elég memória,
2. a kernel *(s)brk* rendszerhívást hajt végre – vagyis memória tartományt bővít,
3. a verem dinamikusan bővül,
4. a kernel átütemez (korábban háttértárra írt folyamatnak kell a hely).

Tárcserénél (1–3) a kernel zárolja az érintett folyamatokat, hogy a *swapper* (0-ás folyamat) menet közben háttértárra ne írja. A memória objektumok éppúgy, mint az állománykezeléshez tartozó adatszerkezetek, hivatkozásszámlálóval rendelkeznek, ami megmondja, hogy egy adott időpontban az objektumot hányan használják. Amikor a *swapper* egy tartományt háttértárra akar írni, csökkenti eggyel a tartomány hivatkozás számlálóját, és ha az 0-ra csökken, ki is írja. Itt gondoljunk az **osztottan használt tartományokra (shared memory regions)**, ahol egy tartományt egyszerre több folyamat is használ. Ekkor a tartomány hivatkozásszámlálója a tartományt használó folyamatok számával egyezik meg. Amikor a *swapper* úgy dönt, hogy egy folyamatot háttértárra ír, a folyamat azon tartományait, amelyeket más folyamat is használ, nem szabadíthatja fel! Amikor a kernel egy tartományt kiír a háttértárra, akkor a tartomány háttértáron elfoglalt címét beírja a **tartománytáblába (region table)**.

A folyamatok **virtuális címtartománya „**hézagos”, ritka, vagyis a folyamatok általában nem használják a teljes virtuális címtartományukat. A **kód-**, **adat-** és **veremtartományok** általában nem összefüggők (már csak azért sem, hogy azok dinamikusan növekedhessenek). Ebből adódóan a *swapper* nem írja ki a folyamat teljes virtuális címtartományát a háttértárra, hisz az felesleges és pazarló lenne, hanem csak a valójában használt (fizikai címmel rendelkező) címtartományokat menti el (5.24. ábra).

Az 5.24. ábrán jól látszik, hogy a folyamatban csak hat laphoz tartozik fizikai memória, így a háttértárra csak ez a hat lap íródik ki. A tartománytáblában tárolódnak a virtuális címek az egyes tartományokhoz, így amikor a

folyamat visszatöltődik a memóriába, bár a fizikai címek megváltozhatnak, a virtuális címek (vagyis a folyamat memóriaképe) nem változnak (az 5.24. ábra jobb oldala).



5.24. ábra. ábra - Folymat háttértárra írása és visszatöltése

Háttértárra írás *fork* rendszerhívás esetén

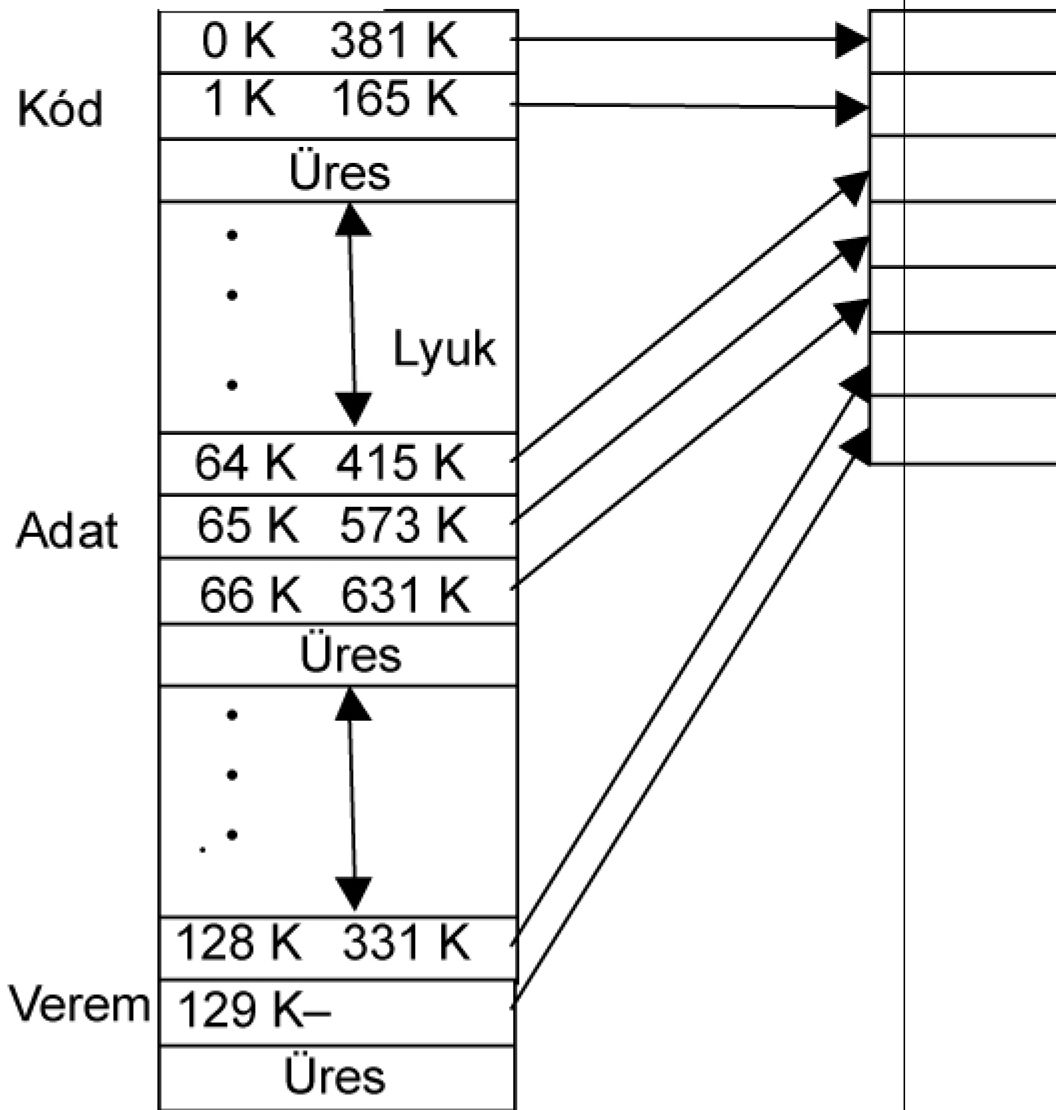
A *fork* rendszerhíváskor két helyzet állhat elő:

- van memória a gyermek folyamat létrehozásához (ekkor nem kell semmit sem háttértárra írni),
- nincs elég memória a gyermek folyamat létrehozásához.

A második esetben a *swapper* kiírja háttértárra a szülő memóriaképét (ez lesz majd a gyermek memóriaképe), de a hozzá tartozó fizikai memóriát nem szabadítja fel, a szülő a memóriában marad. Ezek után a *swapper* futásra késszé teszi a gyermeket a háttértáron. Mivel a háttértáron lévő futásra kész folyamatokat előbb-utóbb a *swapper* betölti, így a gyermek folyamat is bekerül majd a memóriába. Majd az ütemező valamikor futásra ütemezi, amikor majd befejeződik a gyermekben is a *fork* hívás, majd a gyermek folyamat *user* módba vált.

Tartománynövelés

Ez az eset akkor áll elő, amikor egy memóriában lévő folyamat valamelyik tartományának a méretét szeretné növelni, de nem áll rendelkezésre fizikai memória. Leggyakoribb eset, hogy a verem méretét szeretné növelni a kernel. A tartományok méretének növelését az *(s)brk* rendszerhívásokkal lehet megtenni. (A *brk* hívás adott memóriacímre állítja az adott tartományhoz tartozó legmagasabb memóriacímet, míg az *sbrk* adott mérettel bővíti a tartományt.) Amennyiben a rendszerhívás idején rendelkezésre áll a kívánt memória, a rendszerhívás lefut, lefoglalva a megfelelő méretű memóriát. Amennyiben nincs elég memória, a kernel a virtuális címtartományban elvégzi a címleképezést, de nem foglal az új memóriaterülethez fizikai memóriát, hanem a folyamatot kiírja a háttértárra és futásra késszé teszi. (A háttértáron megjelenik a memórianövelés előtti memóriakép, valamint a lefoglalt új memóriatartomány is, amelyet a kernel a háttértáron kinulláz – lásd az 5.25. ábrát.) Amikor legközelebb a *swapper* betölti a folyamatot, a kernel az új virtuális címtartományhoz is allokál fizikai memóriát, így a folyamat „megnövekedett” memóriával fut tovább. (A folyamat nem veszi észre, hogy futása megszakadt, és háttértáron töltött valamennyi időt – ez ütemezés miatt is előfordulhatott volna.)



5.25. ábra. ábra - Tartománybővítés háttértárra írással

A *swapper* folyamat mindig kernel módban fut, nem ad ki rendszerhívásokat, hanem közvetlenül meghívja a kernel rutinokat. Végtelen ciklusban dolgozik, folyamatokat próbál meg memóriába tölteni, illetve memóriából háttértárra írni. Amikor nem tud mit csinálni, alszik. A *swapper* nem kitüntetett folyamat, ugyanúgy fut, mint bármely más folyamat, csak magasabb a prioritása.

A *swapper* működési fázisai

- ha nincs a háttértáron futásra kész folyamat, a *swapper* alszik,
- ha van, de a rendszerben nincs memória, megpróbál folyamat(okat) háttértárra írni és kezdi elölről a futásra kész folyamatok keresését a háttértáron.

Itt fontos megjegyezni, hogy a *swapper* **zombi** folyamatot sohasem ír ki háttértárra, hiszen a *zombi* folyamatok csak folyamattábla bejegyzést foglalnak, fizikai memóriát nem.

5.3.7.4. A háttértárra írás, illetve a háttértárról való beolvasás szabályai

A *swapper* néhány egyszerű szabályt alkalmaz annak eldöntésére, hogy mely folyamatok alkalmasak háttértárra történő kiírásra, illetve háttértárból történő beolvasásra. Ezen szabályok a következők:

- a folyamat memóriába tölthető, ha már legalább 2 másodpercet[2] háttértáron töltött,
- a folyamat háttértárra írható, ha már legalább 2 másodpercet[3] memóriában töltött.

Az óra minden másodpercben felébreszti a *swappert*. Mivel a *swapper* nagy prioritású folyamat, minden másodpercben fut is. Fontos tulajdonsága a háttértárba író algoritmusnak, hogy amikor egy folyamat *sleep* rendszerhívást ad ki, akkor az algoritmus elölről indul, hiszen valószínűleg érdemesebb az épp „elalvó” folyamatot a háttértárra írni.

5.3.7.5. A háttértárra kiírandó folyamat kiválasztásával kapcsolatos problémák

A folyamatokat a *swapper* azért írja ki háttértárra, hogy helyet csináljon a betöltendő folyamatoknak. A folyamatok kiválasztásakor azonban az alábbiakat figyelembe kell venni:

- Lehet, hogy a kiírt folyamat kicsi, továbbra sem lesz elég hely a memóriában ahhoz, hogy egy másik folyamatot betöltsön a *swapper* (például ha kiír egy 2 K-s folyamatot, az nem fog elég helyet biztosítani egy 1 M-es folyamatnak). Erre a problémára megoldást jelenthet, ha a *swapper* folyamatcsoportokat ír ki háttértárra, amelyeknek összes memóriafoglalása már elég lesz a betöltendő folyamat számára.
- Ha a *swapper* azért alszik, mert nem volt elég memória, akkor ébredéskor újra kezdi az egész algoritmust. Ebből fakadóan egy korábban várakozó folyamat lehet hogy ismét nem jut be a memóriába, mert időközben egy másik folyamatot fontosabb lett betölteni.

- Amennyiben egy futásra kész folyamatot ír ki, előfordulhat, hogy az a folyamat még nem is futott (prioritás, *nice*).

Egy érdekes elvi problémára fel kell itt figyelni: szerencsétlen körülmények között holtpont is kialakulhat. Ennek feltételei:

- nincs hely a háttértáron,
- a memóriában minden folyamat alszik,
- minden futásra kész folyamat a háttértáron van,
- nincs elég hely a memóriában, hogy a *swapper* a háttértárból egy futásra kész folyamatot memóriába töltsön.

Szerencsére, ezen állapot kialakulásának kicsi az esélye.

5.3.8. Igény szerinti lapozás

Az operációs rendszer egyik elsődleges feladata, hogy a rendszer memória erőforrásait hatékonyan kezelje. Ezen feladat ellátására az operációs rendszerben a memóriakezelő rendszer egy alrendszert alkot. A **memóriakezelő rendszerrel** szemben természetes elvárás, hogy

- segítségével a fizikai memória méreténél nagyobb programokat lehessen futtatni,
- csak *részben* memóriában lévő programok is futtathatók legyenek,
- a **multiprogramozásból** adódóan támogatnia kell, hogy egyszerre több program is a memóriában lehessen,
- támogassa áthelyezhető programok kezelését,
- a memóriakezelés legyen gépfüggetlen,
- vegye le a programozó válláról a memória allokációs és menedzselési terheket,
- tegye lehetővé az osztott memória használatot.

Ezen feladatok megoldásához egy magas szintű absztrakcióra van szükség: a **virtuális memóriakezelésre (virtual memory)**. A virtuális memória használata feltételezi, hogy a **címtér (address space)** a fizikai memóriától független erőforrás. Mérések azt mutatják, hogy a memóriakezeléssel kapcsolatos tevékenységek jelentős CPU-időt emésztenek fel – terhelt rendszeren megközelíti a 10%-ot.

A UNIX, mint azt korábban láttuk, kezdetben csak a folyamatok háttértárra írását (swapping) biztosította. A modern operációs rendszereknek ennél hatékonyabb eszközökre is szükségük van. A ’80-as évek közepére már minden UNIX-változat az *igény szerinti lapozást* használta, mint elsődleges virtuális memóriakezelő technikát. A legújabb rendszerek már **előretekintő lapozási technikákat (anticipatory paging)** is alkalmaznak, amivel a

rendszer olyan lapokat is behoz a fizikai memóriába, amikről úgy hiszi, hogy majd a közeljövőben szüksége lesz rá.

Mivel a 3.4.2. rész már részletesen tárgyalta a virtuális memóriakezelés elvi alapjait, ismertetve a szükséges hardver és szoftver feltételeket, így ebben a fejezetben már nem térünk ki ennek részleteire, hanem a UNIX azon adatszerkezeteit és használatukat ismertetjük, amelyek az igény szerinti lapozás megvalósításához szükségesek. Fogalmi tisztasága miatt a következőkben a *SVR3* adatszerkezeteit ismertetjük. A legújabb UNIX-rendszerek már egy új, *memóriába ágyazott állományokon* alapuló virtuális memóriakezelő alrendszert alkalmaznak, azonban azok kialakítására nagy hatással volt a SVR3. A tárgyalás végén felhívjuk az olvasó figyelmét a hatékonysági problémákra, és rámutatunk, hogy a későbbi virtuális memóriakezelő alrendszerek hogyan orvosolták ezeket a problémákat.

5.3.8.1. A virtuális memóriakezelést támogató adatszerkezetek

A UNIX az alábbi fontosabb adatszerkezeteket használja memóriakezeléshez:

- *pfdata (page frame data table)*. Ez az adatszerkezet a fizikai lapok állapotát írja le. A tartalmazott információkat alább ismertetjük. A rendszer induláskor foglal helyet számára, statikus adatszerkezet.
- *laptábla-bejegyzés (page table entry)*. Egyrészt tartalmazza a virtuális-fizikai címleképzést, másrészt a memóriakezelő funkciók megvalósításához elengedhetetlen jelzőbiteket tárol.
- *diszkblokk leíró (disk block descriptor)*. A virtuális memória lapjaihoz tartozó háttértár címeket (a lap háttértáron tárolt másolatának a helyét) adja meg egyéb fontos információkkal együtt.
- *háttértár használat tábla (swap use table)*. A háttértár használatát adminisztrálja.

A továbbiakban a fenti adatszerkezeteket és használatukat ismertetjük.

A *pfdata* adatszerkezet

A *pfdata* a fizikai lapokat írja le, az alábbi információkat tartalmazza:

- a lap állapota (háttértáron, végrehajtható állományban található, DMA van folyamatban a lapra, a lap kiadható),
- hivatkozásszámláló: a lapra hivatkozó folyamatok száma (ez megegyezik az érvényes laptábla hivatkozások számával). Ezen mező támogatja a fizikai memória osztott használatát, illetve a kernel ez alapján tudja eldönteni, hogy egy adott lapot újra ki lehet-e adni,
- a lapra éppen melyik háttértár blokk van betöltve: a háttértár logikai címe, az azon belüli blokkcím (ez utóbbi jelentőségét a későbbiekben részletesen tárgyaljuk),
- mutatók *pfdata* bejegyzésekre (szabad lista, hashtábla).

A *pfdata* bejegyzések a háttértár blokk cím szerint **hashsorokba** vannak rendezve (amennyiben több háttértár is van a rendszerben, akkor a hashkulcsban szerepel a háttértár logikai címe is). Ezáltal gyorsan meg lehet találni a blokk cím alapján egy adott háttértár blokkhoz tartozó lapot a memóriában. Ez javítja a hatékonyságot, ugyanis, ha egy olyan lapra van szükség, ami már bent van a memóriában, akkor azt nem kell ismét betölteni, meg lehet takarítani egy lemezműveletet. Ezenfelül a szabad memórialapok egy szabad listára is fel vannak fűzve. Amennyiben az operációs rendszer lapot akar foglalni, ezen szabad lista elejéről foglal. Az lenne az ideális, ha a szabad lista elején olyan lapok lennének, amelyekre már soha sem lesz szükség a későbbiekben, vagyis ezeknek a lapoknak meg kellene előzni azokat a lapokat, amelyekre esetleg a későbbiekben szükség lehet. Ha nincsenek ilyen lapok, akkor a lokalitás elvéből adódóan a l**egrégebben használt (least recently used)** lapokat lenne célszerű felhasználni, azokat lenne célszerű a szabad lista elején tárolni.

Ezzel kapcsolatban felmerül egy gyakorlati probléma: a *legrégebben használt* sorrend fenntartásához minden egyes laphivatkozás után újra kellene rendezni a listát, aminek időköltsége nem megengedhető. Ezért egy ésszerű kompromisszummal a *legrégebben használt* stratégia helyett a **régen használt (not recently used)** stratégiát alkalmazzák. Ehhez egy hivatkozásbitet rendelnek minden egyes laphoz, ami az adott lapra történő hivatkozáskor beállítódik, illetve adott időközönként törlődik. Bár ez a stratégia nem optimális, garantálja, hogy csak olyan lapok kerüljenek újra kiadásra, amiket az elmúlt időszakban nem használtak. A későbbiekben ismertetjük, hogy ezt a technikát szimulált hivatkozásbit bevezetésével hogyan lehet megvalósítani olyan architektúrán, ami hardverből nem támogatja a hivatkozásbitet.

Laptábla bejegyzés

Minden egyes folyamathoz tartozik legalább egy **laptábla (page table)**, aminek a bejegyzései a folyamat címterét képezik le a fizikai memória címeire, illetve további memóriakezeléssel kapcsolatos információkat tárolnak. A laptábla elemei magától értetődően tartalmazzák a fizikai memóriacímeket és a hozzáférési jogosultság biteket. Az igény szerinti lapozás támogatására azonban még az alábbi biteket is nyilván kell tartani (5.26. ábra).

- érvényességi (valid),
- hivatkozás (reference),
- módosítás (modify, dirty),
- másolás írás esetén (copy on write – C/W),
- öregítés (age),
- védelem (protection).

Lap fizikai címe	Age	C/W	Nod	Ref	Val	Protection

Protection – védelmi bitek. táblázat - all

A hivatkozás- és módosításbiteket általában a hardver állítja. Léteznek olyan hardverek is, amelyek ezt nem teszik meg. A hivatkozásbit szoftver szimulációját majd a későbbiekben tárgyaljuk. Az érvényes, C/W és az öregítésbiteket a kernel kezeli.

Diszk blokk leíró és a háttértár használat tábla

A **diszk blokk leíró (disk block descriptor)** a virtuális memória lapjaihoz tartozó háttértár címeket adja meg. Ehhez tartalmazza a háttértár logikai címét, a lap háttértáron elfoglalt helyét, illetve megadja a lap típusát, vagyis, hogy milyen módon lehet a **tárolt másolatot (backing store)** elérni (5.27. ábra). A lap lehet a háttértáron (swap), lehet a memóriában, illetve lehet **fill-on-demand** típusú, vagyis ha szükség van a lapra, akkor azt egy adott tartalommal fel kell tölteni. Ennek két típusa ismeretes. **Fill-from-text** esetén a betöltendő tárolt másolat egy végrehajtható állományban található. Az ilyen típusú lapok általában kódot vagy inicializált adatot tartalmaznak. **Zero-fill** esetén nincs tárolt másolat, a kernel a lapot kinullázza, vagyis nullával tölti fel. Az ilyen típusú lapok általában nem inicializált adatot tartalmaznak. A kinullázással a kernel biztosítja, hogy a nem inicializált változók kezdeti értéke nulla legyen.

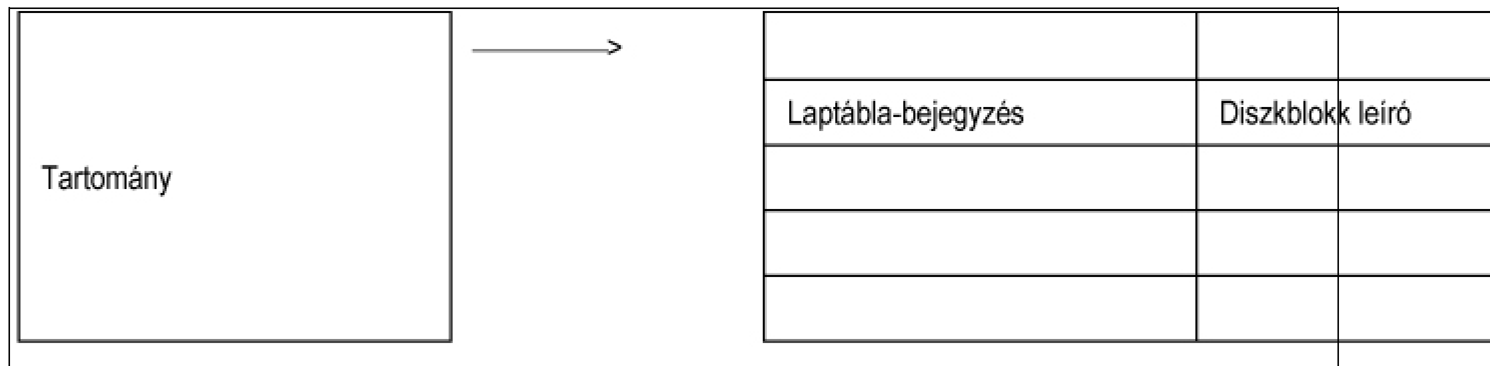
Háttértár logikai címe	Blokk sorszám	Típus (swap, file, ZF, FT)

5.27.ábra. táblázat - A diszk blokk leíró által tárolt információk. A bejegyzést a folyamat virtuális címei címzik

A háttértár használati tábla a háttértár használatát adminisztrálja. Ez az adatszerkezet is tartalmaz egy hivatkozásszámlálót, ami a *pfdata* hivatkozásszámlálójához hasonló szerepet tölt be. Amikor a kernel fizikai lapot allokál a háttértáron tárolt laphoz (vagyis amikor a lap betöltődik a fizikai memóriába), a laphoz tartozó *pfdata* hivatkozásszámláló értékét a háttértár használati tábla hivatkozásszámlálójának az értékére állítja, így a konzisztencia biztosítható.

5.3.8.2. A virtuális memóriakezelést támogató adatszerkezetek használata

A SVR3 a memóriakezeléshez a **tartomány (region)** modellt alkalmazta. Ennek lényege, hogy a lapszervezésű memóriát nagyobb logikai egységenként, tartományonként kezelte. Egy-egy tartományt rendeltek a folyamatok kód, adat és verem területéhez, illetve egyéb logikai egységeihez. A tartományok a memóriahasználat logikai szerkezetét tükrözik. Az 5.28. ábra a tartományok és a laptábla-bejegyzések, illetve diszkblokk leírók viszonyát mutatja.



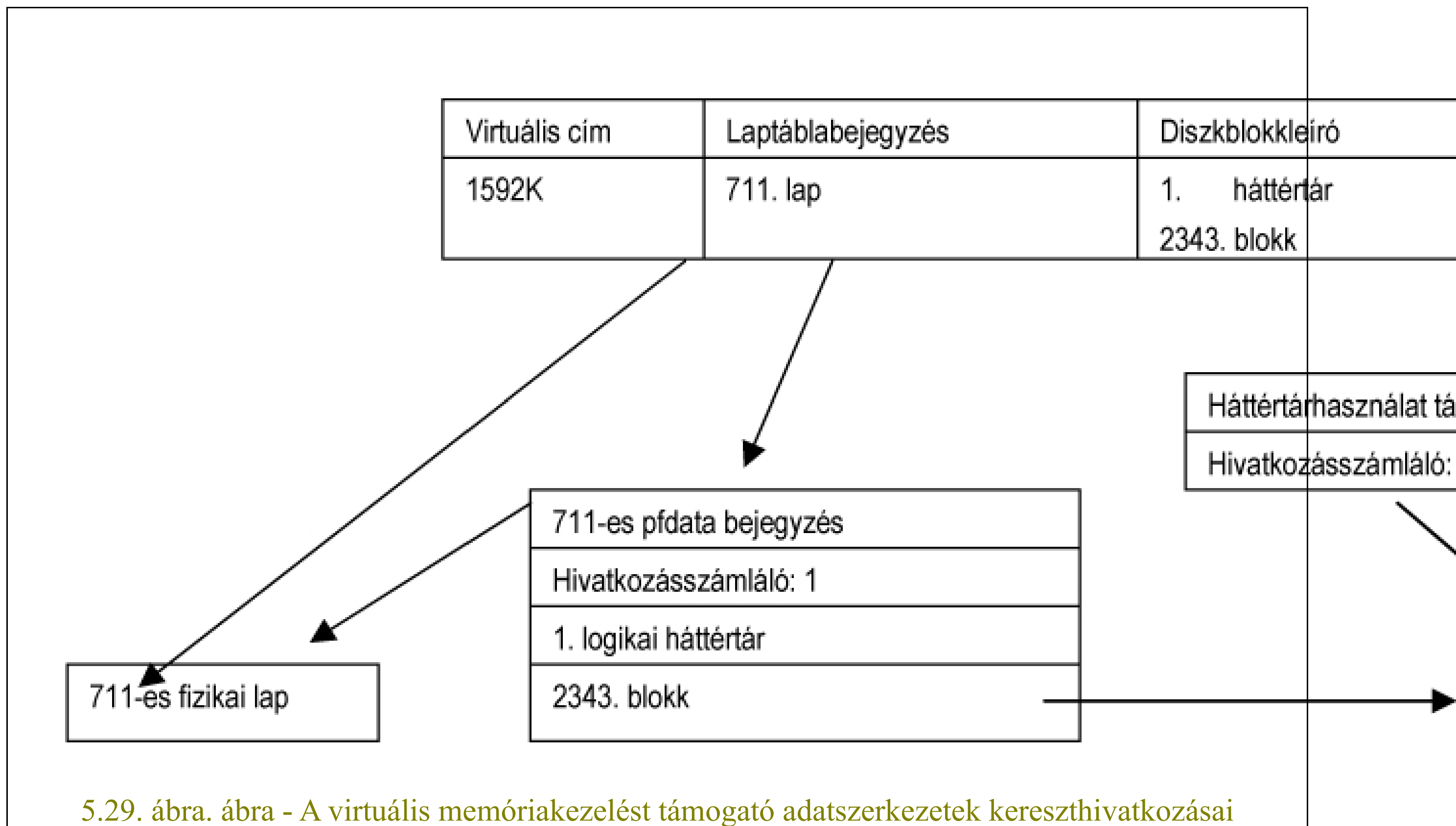
5.28. ábra. ábra - A tartománymodell és a virtuális memóriakezelést támogató adatszerkezetek kapcsolata

Az ábrán jól látszik, hogy minden tartományhoz tartozik egy laptábla, illetve, hogy a laptáblabejegyzéseket és a diszkblokk leírókat a kernel együtt kezeli (mindkettőt a folyamat virtuális címeivel címzi), azok akár egyetlen adatszerkezetet is alkothatnának, azonban eltérő funkcionalitásuk miatt és az áttekinthetőség kedvéért célszerű azokat külön adatszerkezetbe foglalni.

A virtuális memóriakezelést támogató adatszerkezetek közötti kapcsolat

A korábbi bekezdésekben ismertetett adatszerkezetek között számos kereszthivatkozás található, ami megkönnyíti az operációs rendszer hatékony működését. Az 5.29. ábra ezeket a kapcsolatokat mutatja. A jobb áttekinthetőség érdekében az ábra egyetlen laptáblabejegyzést és a hozzá kapcsolódó adatelemeket mutatja.

Mint az az 5.29. ábrán jól látszik, mind a laptáblabejegyzést, mind pedig a diszkblokk leírót a folyamat virtuális lapcímével címzik. A laptáblabejegyzésben tárolt fizikai lapcím egyrészt kapcsolatot teremt magával a fizikai lappal, másrészt az adott laphoz tartozó *pfdata* bejegyzéssel is (a példában ez a 711-es lapkeret). A 711-es lapkeretet (fizikai memórialapot) leíró *pfdata* bejegyzés természetesen szintén hivatkozik magára a 711-es lapkeretre. Továbbá ugyanezen bejegyzés hivatkozik arra a háttértár blokkra, amely tárolja a virtuális lap tartalmát (ebben az esetben az 1. háttértár 2343-as blokkjára). Ugyanerre a blokkra hivatkozik a diszkblokk leíró és a háttértár használati tábla megfelelő bejegyzése is. Első látásra ez a redundancia tékozlónak tűnhet, azonban egy példa hamarosan rávilágít, hogy ezen adatszerkezetek segítségével bizonyos esetekben jelentős hatékonyság növekedést lehet elérni.



5.29. ábra. ábra - A virtuális memóriakezelést támogató adatszerkezetek kereszthivatkozásai

A „laplopó” folyamat

A **„laplopó” folyamat** (vagy *pagedaemon*) kernel szintű folyamat, inicializáláskor jön létre. Akkor aktivizálódik, amikor a rendelkezésre álló szabad lapok száma egy alsó határ alá csökken, és addig fut, amíg a szabad lapok száma újra egy felső határ fölé nem ér. Erre a két határra azért van szükség, hogy elkerülje a rendszer a **vergődést (thrashing)**.

A lapoknak két állapota lehetséges:

- a lapot még nem lehet kiírni háttértárra, öregszik (ez tulajdonképpen több állapotot takar, és implementáció függő, hogy a laplopó egy lapot hányszor vizsgál meg kiírás előtt). A laplopó minden egyes vizsgálatnál törli a referenciabitet és öregít,
- a lap kiírható, újra kiadható.

A lapok nem feltétlenül arányosan vannak elosztva a folyamatok között. Egy fontos szabályt mindig betart a laplopó: használatban lévő lapot **nem lop el**.

Amikor a laplopó úgy dönt, hogy kiír egy lapot, három eset lehetséges:

- nincs másolat a háttértáron ® a lapot kiírásra ütemezi,
- van másolat, nincs módosítás ® laptábla bejegyzés valid bitjét törli, csökkenti eggyel a *pfdata* hivatkozásszámlálóját, és a lapot a szabad listára teszi,

- van másolat, és a memóriakép módosult ® kiírásra ütemezi a lapot, és az éppen használt háttértár helyet felszabadítja (blokkos kiírás).

A háttértár tördelődése jelentős lehet, mert bár a kiírás blokkos, de a felszabadítás, illetve a beolvasás laponkénti. Amikor a kernel kiírja a lapot, törli az érvényességi bitet, és a *pfdata* hivatkozásszámlálóját eggyel csökkenti. Amennyiben a hivatkozásszámláló 0 lett, akkor a lapot a szabad lista végére teszi.

5.3.8.3. Laphibák

Mint azt korábban láttuk, a címtranszformáció során a memóriakezelő egység a virtuális címet szétbontja egy virtuális lapcímre és egy lapon belüli eltolásra. A virtuális lapcím alapján megkeresi a laphoz tartozó laptábla bejegyzést, amiből kiolvassa a laphoz tartozó fizikai lapcímet. A fizikai lapcímhez hozzáilleszti a lapon belüli eltolást és így előáll a teljes fizikai cím. A címleképzés a következő okok miatt hiúsulhat meg.

- *Túlcímzés hiba (bounds error)*. A kiadott cím kívül esik az adott folyamat által kiadható érvényes címtartományon, ebből adódóan a címhez nem tartozik laptábla bejegyzés.
- *Érvényességi hiba (validity fault)*. A laphoz tartozik laptábla bejegyzés, azonban a hozzá tartozó érvényességi bit törölve van, ami általában azt jelenti, hogy a laphoz nem tartozik fizikai memórialap. Mint azt majd a későbbiekben látni fogjuk, a szoftverből szimulált referenciabit esetén az érvényességi bitet a rendszer felhasználja a szimulációhoz, és a lap akkor is érvényes lehet, ha az érvényességi bitje ki van törölve.
- *Védelmi hiba (protection fault)*. A lap nem engedi meg a kiadott művelet igényelte hozzáférést (például egy csak olvasható lapot nem lehet írni, vagy a felhasználó nem férhet hozzá a kernel lapokhoz). A *copy-on-write* technikánál is ezt a mechanizmust használja a kernel.

A fent ismertetett hibák minden esetben **kivételt (exception)** okoznak, amit egy kivételkezelő rutin kezel le. Ezeket a kivételeket általánosan **laphibának** nevezik. A kivételkezelő megkapja paraméterként a hibát kiváltó virtuális lapcímet, illetve a hiba típusát is (védelmi vagy érvényességi hiba – a határhiba is érvényességi hibát okoz). A kivételkezelő a hiba típusától függően megpróbálja a megfelelő lapot behozni a memóriába vagy egy jelzés elküldésével értesíti a folyamatot. A hibakezelők általában nem kerülnek alvó állapotba, kivéve ezen hibák kezelőit. Ezek aludhatnak, de ezek adott folyamathoz tartoznak. Továbbá a hiba kezelése alatt a tartományt zárolni kell, hogy a laplopó nehogy ellopjon lapokat a tartományból.

5.3.8.4. A laptábla-bejegyzés, a diszk blokk leíró és a pfdata együttes használata

A továbbiakban szemléletes példákon keresztül bemutatjuk a laptábla-bejegyzés, a diszkblokk leíró és a *pfdata* együttes használatát.

Mint azt korábban láttuk, az igény szerinti lapozás kihasználja, hogy az érvénytelen lapra történő hivatkozás laphibát okoz. A kivételkezelő, ha már tudomást szerzett róla, hogy a hiba azt jelzi, hogy egy lapot a memóriába kell tölteni, akkor a legfontosabb feladata, hogy a lapot megkeresse és betöltse.

A hibát okozó lap az alábbi állapotban lehet:

1. háttértáron, de nincs memóriában,
2. a szabad listán a memóriában,
3. végrehajtható állományban,
4. zero fill.

Az 5.30. ábra ezekre az esetekre mutat egy-egy példát.

	Virtuális cím	Fizikai lapcím	Érv.	Hely	Blokk
	0 K				
3. eset	1 K	1343	Inv.	File	3
	2 K				
	3 K	nincs	Inv.	TF	5
	4 K				
2. eset	64 K	1457	Inv.	Diszk	1336
4. eset	65 K	nincs	Inv.	ZF	
1. eset	66 K	1036	Inv.	Diszk	813

Lap	Diszk-blokk	Hivatkozás-számláló
1036	387	1
1343	1618	1
1611	1336	0

5.30. ábra. ábra - A laphibát okozó lap állapotai

1. eset: A lap a háttértáron van, de nincs a memóriában

Az 5.30. ábrán, az 1. esettel jelölt sorban látható, hogy a kérdéses lap nincs memóriában (az érvényességi bit törölve van, a lap *invalid*), és a háttértáron a 813-as blokkon található (a Diszk bejegyzés jelöli). A *pfdata* adatszerkezet diszkblokk oszlopát végigkeresve a 813-as blokkal, látjuk, hogy az nem szerepel, vagyis a szabad listán sincs sehol. A *pfdata* adatszerkezetből az is kiolvasható, hogy a 1036-os fizikai lap, ami korábban a 813-as háttértár blokk tartalmát tárolta, jelenleg a 387-es háttértár blokk tartalmát tárolja. A laphiba kezelő ekkor lapot allokál a 66K virtuális lapnak, megkapja az 1676-os fizikai lapot, és a 813-as blokk tartalma ide töltődik be. Az 5.31. ábra az ez utáni adatszerkezeteket mutatja.

66 K	1676	Val.	Diszk	813

1676	813	1

5.31. ábra. ábra - A módosult adatszerkezetek a lap allokálás után

2. eset: A lap a szabad listán, a memóriában található

Az 5.30. ábrán, a 2. esettel jelölt sorban látható, hogy a 64 K-s virtuális című lap még a szabad listán megtalálható a memóriában. Ezt a kernel a következőképpen találja meg: a lap a diszken a 1336-os blokkban található. Ez alapján a blokk alapján keres a *pfdata* adatszerkezet diszkblokk oszlopában, és megtalálja, hogy az ehhez a blokkhoz tartozó lap az 1611-es fizikai lapban található. A hivatkozásszámláló 0-ás értéke jelöli, hogy a lapot jelenleg senki sem használja, az valóban a szabad listán található. Ekkor a hivatkozásszámlálót eggyel növeli a kernel, és ezt a lapcímet beírja a laptábla bejegyzésbe. Ez után a 64 K-s virtuális lap bejegyzése az 5.32. ábra szerint alakul:

64 K	1611	Val.	Diszk	1336

1611	1336	1

5.32. ábra. ábra - A módosult adatszerkezetek a lap szabad listán történő megtalálása után

3. eset: A lap egy végrehajtható állományban található

Az 5.31. ábrán, a 3. esettel jelölt sorban látható, hogy a lap egy végrehajtható állomány adott blokkjában található (erre a *hely* oszlop *File* bejegyzése utal). A példában az 1 K virtuális címhez tartozó lap tartalma egy végrehajtható állomány 3. logikai blokkja. Az állomány megnyitásakor a kernel végez egy kis adat-előkészítést: az állományhoz tartozó *inode*-ba a végrehajtható állomány logikai blokkjaihoz tartozó fizikai blokkcímeket írja be sorfolytonosan, így a logikai blokk cím rögtön indexként használható.

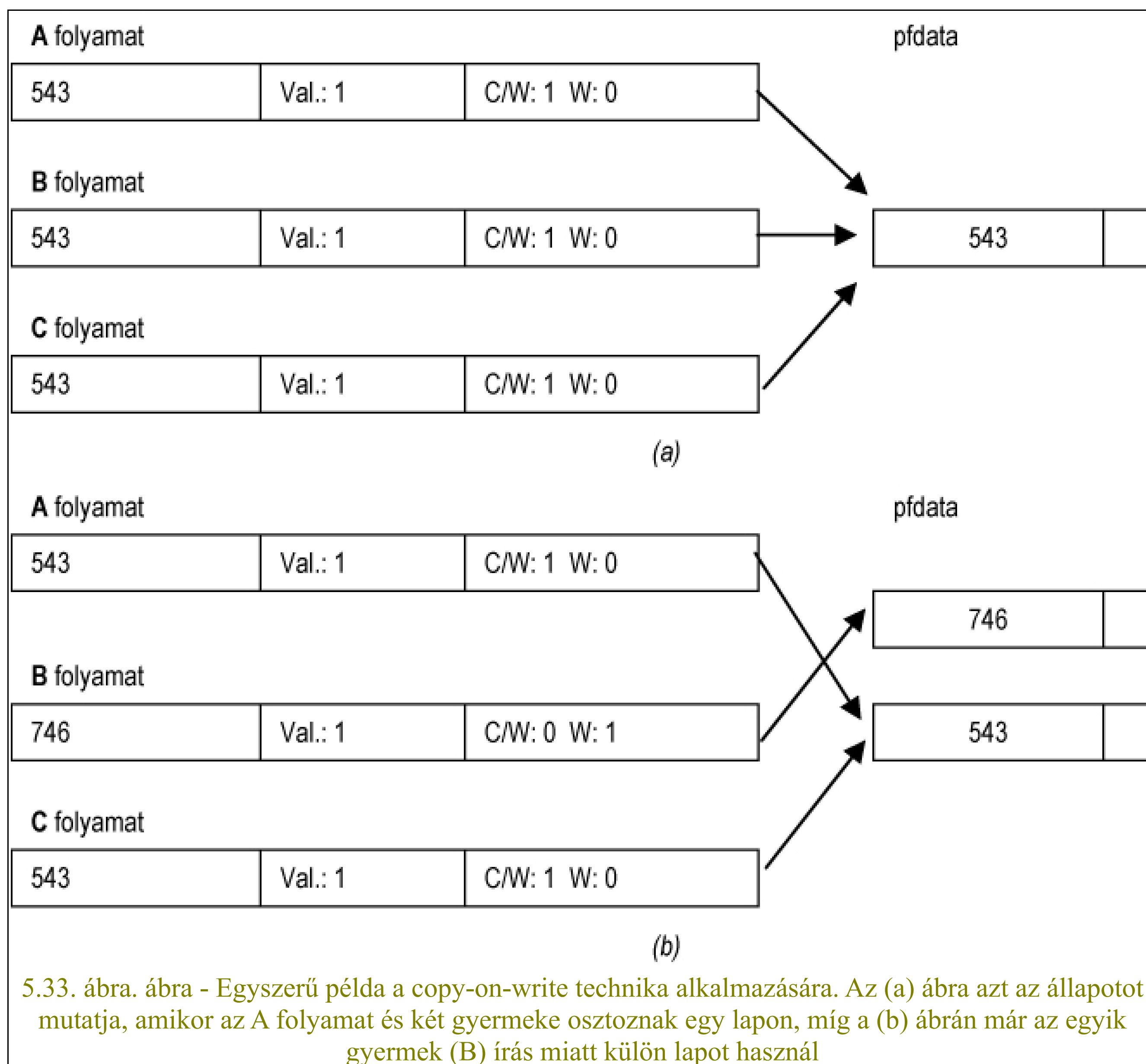
4. eset: A laphoz nem tartozik tárolt másolat, azt fizikai memória allokálásakor ki kell nullázni

Az 5.30. ábrán, a 4. esettel jelölt sorban látható, hogy a lap *zero-fill*, vagyis nem inicializált adatot tartalmaz (erre a hely oszlop ZF bejegyzése utal). Ekkor a kernel amikor fizikai memórialapot allokál, azt kinullázza, ezzel biztosítva, hogy a nem inicializált változók nulla kezdeti értékkel rendelkezzenek.

5.3.8.5. A copy-on-write technika és használata

Mint azt a folyamatkezeléssel kapcsolatban láttuk, a UNIX-ban új folyamatot a *fork* rendszerhívással lehet létrehozni. A rendszerhívás hatására a létrejövő gyermek folyamat memóriaképe megegyezik a szülő memóriaképével. A korai rendszerekben a folyamat létrehozásával automatikusan allokáltak memóriát a gyermek folyamat számára. Ez nagyon rossz memóriagazdálkodást eredményezett, hisz nagyon gyakran a gyermek folyamat ugyanazt a kódot futtatja, mint a szülője, és a gyermek folyamat gyakran memóriamódosítás nélkül fejeződik be. Ilyenkor felesleges külön memóriát allokálni a gyermek számára.

A probléma kezelésére kidolgozták a **copy-on-write** technikát. A módszer során, ha egy folyamat gyermek folyamatot hoz létre, a kernel először csak a mutatókat kezeli. Először csak az osztottan kezelt tartományok hivatkozás- számlálóját növeli. A privát tartományokra új tartománytábla bejegyzés készül, illetve a kernel új laptáblákat foglal. Ezek után a kernel végigmegy a szülő lapjain. Ha a lap érvényes, akkor a *pfdata* hivatkozásszámlálóját eggyel megnöveli (ezzel jelzi, hogy a fizikai lapot külön tartományok használják). Amennyiben a lap háttértáron van, akkor a háttértár használat tábla hivatkozásszámlálóját növeli eggyel. Továbbá a folyamatok laptábla bejegyzéseiben a hozzáférési jogosultságot csak olvasási jogosultságra állítja (W: 0) és bebillenti a C/W *(copy-on-write)* bitet, jelezve, hogy a *copy-on-write* technikát alkalmazza. Ha a szülő vagy a gyermeke megpróbálja írni a közösen használt lapok valamelyikét, akkor az írás **védelmi laphibát** (protection fault) okoz. A laphibakezelő megvizsgálja a C/W-bit állapotát. Ha a bit nincs beállítva, akkor valóban jogosulatlan műveletet próbáltak meg végrehajtani, ellenkező esetben viszont a kernel tudomást szerez róla, hogy most már nem lehet tovább elodázni a fizikai memóriafoglalást, a laphibát okozó folyamat számára lapot kell foglalni, és a foglalás tényét be kell jegyezni a laptáblába, illetve a korábbi leképzésben szereplő fizikai laphoz tartozó *pfdata* adatszerkezetben csökkenteni kell a **hivatkozásszámlálót**.



5.33. ábra. ábra - Egyszerű példa a copy-on-write technika alkalmazására. Az (a) ábra azt az állapotot mutatja, amikor az A folyamat és két gyermeke osztoznak egy lapon, míg a (b) ábrán már az egyik gyermek (B) írás miatt külön lapot használ

Az 5.33.*(a)* ábra egy ilyen helyzetet mutat be. Az **A** folyamat két gyermek folyamatot hozott létre, **B**-t és **C**-t. A három folyamat közösen használja az 543-as fizikai lapot. Az ábrán a jobb áttekinthetőség érdekében laptábla bejegyzés és a *pfdata* adatszerkezetnek csak a releváns komponenseit tüntettük fel, illetve csak egyetlen lapot, az éppen írt lapot ragadtuk ki. Az 5.33.(a) ábrán jól látszik, hogy mindhárom folyamat az 543-as lapot használja, amit a *pfdata* hivatkozásszámlálójának 3-as értéke is jól mutat.

Ekkor a **B** folyamat írni próbál a mutatott lapra. A folyamat védelmi laphibát okoz (W: 0 volt). A laphibakezelő látja, hogy az adott laptábla bejegyzés C/W-bitje be van állítva, így fizikai memória lapot kell allokálnia a **B** folyamat számára. Az 5.33.*(b)* ábra ezt az állapotot mutatja. Jól látszik, hogy a hiba kezelése során a **B** folyamathoz egy új lap (746) allokálódott, amelynek a hivatkozásszámlálója 1, hisz csak a **B** folyamat hivatkozik rá, míg az 543-as lap hivatkozásszámlálója eggyel csökkent, mert a **B** folyamat már nem rá

hivatkozik. Amennyiben a C/W-bit be van billentve, de a hivatkozásszámláló már csak 1, akkor a kernel nem allokál újabb lapot, hanem az eredetiben törli a C/W-bitet, és engedi a folyamatnak a lap írását.

5.3.8.6. Hivatkozás bit szimulálása szoftverből

Néhány számítógép architektúra, mint például a VAX-11 vagy a MIPS R3000, nem nyújt hardver támogatást a **hivatkozás (reference) bit** használatához. Ekkor annak funkcióját egy szimulált szoftver hivatkozás bit veheti át. Ezt egy egyszerű mechanizmussal lehet biztosítani: be lehet vezetni egy ún. **szoftver érvényességi** (valid) **bitet**, ami a memórialap valódi érvényességi információját hordozza. Nevezzük az eredeti érvényességi bitet megkülönböztetésül **hardver érvényességi bitnek**, ezt a hardver teszteli, és érvénytelen állapota okozza a *laphibát*. A *szoftver hivatkozás bitet* most a kernel kezeli, ez tölti be a szokásos hivatkozás bit szerepét, a szoftver érvényességi bitet ugyancsak a kernel állítja. Induljunk onnan, hogy a lap érvényes, a *hardver érvényességi bit* be van billentve. Amikor adott időközönként a kernel (vagy a laplopó folyamat) törli a (most szoftver) *hivatkozás bitet*, vele együtt a *hardver érvényességi bitet* is törli, és a *szoftver érvényességi bitet* bebillenti, jelezve, hogy a lap valójában érvényes. (5.34.*(a)* ábra.) Ezek után, amikor hivatkozunk a lapra, az laphibát okoz, hiszen a *hardver érvényességi bit* ki van törölve. A laphiba kezelő megnézi, hogy valóban érvénytelen-e a lap. Látja azonban, hogy a *szoftver érvényességi bit* be van állítva, vagyis a lap érvényes, így most hivatkozás történt egy érvényes lapra. Ekkor a kernel beállítja mind a *szoftver hivatkozás bitet*, mind a *hardver érvényességi bitet*. (5.34.*(b)* ábra.) Ezzel a rendszer információt szerzett arról, hogy a lapot nem „túl régen” használták, és az érvényesség is helyreállt, a közeli jövőben történő hivatkozás már nem okoz laphibát. Amikor a laplopó folyamat végigvizsgálja a lapokat a memóriában, nem választ hivatkozott lapot, de törli a *hivatkozás bitet* és most vele együtt a *hardver érvényességi bitet* is. Amennyiben a lapra történik újabb hivatkozás, a fentiek szerint a *szoftver hivatkozás bit* beáll, és a lap „friss”-nek fog tűnni, amennyiben nem, a laplopó legközelebb elveheti a lapot. Természetesen amikor a lap valóban érvénytelenné válik, mind a *hardver*, mind a *szoftver érvényességi bitet* törölni kell.

HW érvényességi bit	SW érvényességi bit	SW hivatkozás bit
0	1	0

(a)

HW érvényességi bit	SW érvényességi bit	SW hivatkozás bit
1	1	1

(b)

5.34. ábra. ábra - A hardver, szoftver érvényességi és a szoftverből szimulált hivatkozás bit állapota (a) memóriahivatkozás előtt, (b) memóriahivatkozás után

5.3.8.7. A 4.3 BSD virtuális memóriakezelése

A 4.3 BSD virtuális memóriakezelő sok szempontból hasonlít a SVR3 virtuális memóriakezelő alrendszerre, hasonló adatszerkezeteket használ, azonban a terminológia kissé eltérő. A 4.3 BSD-ben a fizikai memóriát egy ún. **memória térkép (core map)**, a virtuális memóriát **laptáblák**, míg a háttértárat a **diszk térképek (diszk map)** írják le. A fizikai memória a használat szempontjából három részre oszlik. Az alsó memóriaterületeken helyezkedik a **nem lapozott memória pool**, ami tipikusan kernel kódot és a kernel statikusan allokálható adatszerkezeteit tartalmazza, középen helyezkedik el a **lapozott memória pool**, amit általános célra, a felhasználók folyamatai és a kernel dinamikus adatszerkezetei használnak (ez teszi ki a fizikai memória jelentős részét) és a fizikai memória felső címtartományában található a **hiba buffer**, amit a rendszerhívások során generált hibaüzenetek tárolására tartanak fenn.

Mint azt láttuk, a SVR3 implementáció a nem memória rezidens lapokról egy külön adatszerkezetben, a diszkblokk leíróban tárol információkat. Ez a megoldás jelentős redundanciát tartalmaz, és többletmemóriát igényel. A 4.3 BSD ezt a problémát úgy oldotta meg, hogy kihasználta, hogy a védelmi és érvényességi biteken kívül a többi bitmezőt a hardver nem vizsgálja, ha az érvényességi bit nincs beállítva. Mivel a nem memória rezidens lapok esetén az érvényességi bit törölt állapotú, így ezek a mezők más olyan információ tárolására használhatók fel, amik nyilvántartják ezen lapok helyét. Erre egy tipikus példa, hogy egy **fill-on-demand** lap esetén, ha azt egy végrehajtható állományból kell betölteni, akkor mivel ilyenkor az érvényességi bit törölt állapotú, és így a memóriakezelő tudja, hogy a fizikai lapkeret címét leíró mezőben más információ jelenik meg. Ebben az esetben egy biten jelezni lehet, hogy a **fill-on-demand** lap végrehajtható állományból töltendő vagy ki kell nullázni, és ha végrehajtható állományból töltendő, akkor a lapkeret címe helyett a végrehajtható állomány állományrendszerben elfoglalt megfelelő blokkcímét lehet ezen a helyen tárolni. Ez jelentős memória megtakarítást eredményezett a SVR3 implementációhoz képest.

A legújabb UNIX-rendszerek már egy új, **memóriába ágyazott állományokon** alapuló virtuális memóriakezelő alrendszert alkalmaznak, ami kialakulásában magán viseli az elődök fejlesztési tapasztalatait, azonban egy jelentősen eltérő struktúrát vezet be. Ennek tárgyalása meghaladja ezen könyv kereteit. Az irodalomjegyzék bőséges forrást tartalmaz, ami alapján az érdeklődő olvasó elmélyülhet a témában.

5.4. Hálózati és elosztott szolgáltatások a UNIX-ban

A UNIX történetének egyik legfontosabb állomása volt, amikor az 1984-ben kiadott BSD UNIX-ba bekerült a TCP/IP protokoll támogatása. A TCP/IP támogatást ezután a többi UNIX-verzió is gyorsan átvette. A UNIX egyik nagy erőssége azóta, hogy igen könnyű hálózatba kötni az akár különböző hardveren futó UNIX-rendszereket.

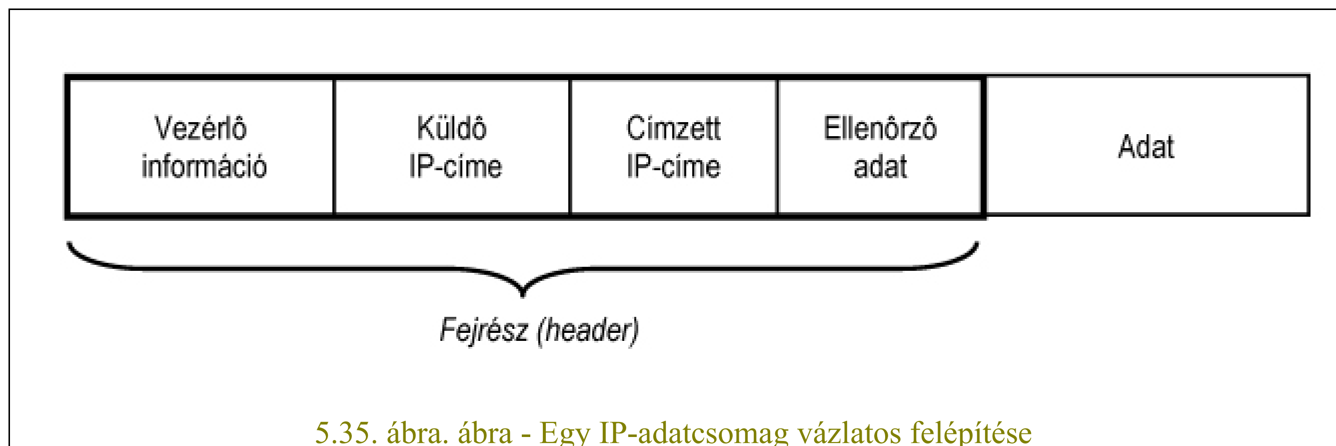
A UNIX-rendszerek fejlesztői azóta is nagy súlyt fektettek a hálózati szolgáltatások fejlesztésére. Az egyik legfontosabb hálózati szolgáltatás az elosztott fájlrendszerek biztosítása. A UNIX-rendszerek több alternatívát is kínálnak hálózaton keresztül történő fájlekérés támogatására (például HA-NFS, Remote File Sharing – RFS, Andrew File System, Unix to Unix Copy). Ezek közül a legsikeresebb a megjelenése óta az elosztott

fájlrendszerek szabványává vált SUN Network File System, amit részletesen is bemutatunk. Ennek a rendszernek a bemutatása alkalmat ad számos hálózati protokoll ismertetésére is.

5.4.1. A TCP/IP protokoll család

A TCP/IP családba tartozó protokollok nem a fizikai átviteli közeg elérését szabályozzák, mint például az Ethernet, hanem a fizikai átvitel fölé épülnek. Ezek a protokollok a csomópontok címzési módját, illetve a hálózaton továbbított adatcsomagok méretét és formátumát rögzítik.

A család legalacsonyabb szintű protokollja az IP (Internet Protokoll). Az IP protokoll nem biztosít megbízható átvitelt, az adatcsomagok késhetnek, sérülhetnek, duplikálódhatnak, elveszhetnek, de csak abban az esetben, ha az IP által használt hálózat hibázik. (Ez pontosan azt jelenti, hogy az IP használata nem jelent garanciát az ilyen hibák kiküszöbölésére.) Az IP adatcsomagok formátuma viszonylag bonyolult, méretük maximum 64 kbyte lehet. Mindig egy ún. *fejrésszel (header)* kezdődnek, ami tartalmazza a küldő és a címzett csomópont IP-címét. Ez az egyetlen része az üzenetnek, mely redundáns, vagyis védett a sérülések ellen, megteremtve a lehetőségét annak, hogy az IP protokollt megvalósító szoftver felismerje az esetleges hibákat (5.35. ábra).



5.35. ábra. ábra - Egy IP-adatcsomag vázlatos felépítése

Az IP protokollt megvalósító szoftver feladata ezek után az, hogy az IP-címet a helyi hálózatban használatos címmé alakítsa, és az adatcsomagot továbbküldje, elvégezve a használt alacsonyabb szintű protokollnak megfelelő változtatásokat az adatrészben. Az IP protokoll nagyon népszerű a hálózati rendszerekben. Több megvalósítása létezik különböző, alacsony szintű hálózati protokollok fölé, így Ethernet hálózat fölé is.

Az IP protokollra számos más protokoll épül. A TCP *(Transport Control Protocol)* az IP protokollra épülve megbízható hálózati átvitelt garantál, míg az UDP *(User Datagram Protocol)* ugyancsak az IP-re épülve csak az üzenetek továbbítását biztosítja. Mindkét protokoll lehetőséget ad a csomópontokon futó folyamatok közvetlen elérésére (címzésére) az ún. *process port*-okon keresztül.

Mind a TCP mind az UDP közvetlenül is elérhető különböző alkalmazások számára. Nagyon sok magasabb szintű szolgáltatást adó protokoll épül a TCP/IP család protokolljaira. A TCP-t használó legismertebb szolgáltatások: fájlok átvitele távoli gépekre *(FTP: file transfer)*, távoli gépekre történő belépés *(telnet)*, továbbá

elektronikus mail (e-mail) szolgáltatás alapját adó *SMTP* protokoll. Az *UDP*-t használó szolgáltatások: távoli gépeken dolgozó felhasználók azonosítójának lekérdezése *(rwho)*, fájlok hordozása *(TFTP)*.

5.4.2. A SUN Network File System (NFS)

A *SUN NFS* a SUN cég (USA) által kifejlesztett elosztott állományrendszer, első változata 1985-ben jelent meg. Megjelenésével egyidőben szabadon elérhetővé tették az általa használt protokollt, valamint referencia-implementációkat is közzétettek, melyek nagyban elősegítették a rendszer elterjedését. Ma már, széles körű használata miatt, mintegy szabványnak tekinthető az elosztott állományrendszerek körében.

5.4.2.1. A SUN NFS jellemző tulajdonságai

A SUN NFS a kliens-szerver modellre épül. A kliens és a szerver egymástól független, szétválasztható, azonos vagy távoli gépeken is futhatnak. A szolgáltatás az ún. *távoli eljáráshívásra (RPC: Remote Procedure Call)* épül, mely módszert a későbbiekben részletesen is ismertetjük. A SUN NFS az RPC *szinkron* megvalósítását használja, vagyis a kliens folyamatok mindig várakoznak a kérésük teljesítésére.

A SUN NFS felhasználói felületének lehetőségei a következők:

- Egy vagy több fájlrendszer teljes vagy részleges (részfa) exportálása (láthatóvá tétele a rendszer többi csomópontja számára) lehetséges egy adott csomópontról.
- Konfigurációs fájl készíthető az exportált fájlrendszert elérő kliensek definiálására, amelynek segítségével például a felhasználói azonosítók alapján szabályozhatjuk az egyes fájlrendszerek elérését.
- Távoli fájlrendszer csatlakoztatásának (*mount*-olásának) lehetősége a lokális fájlrendszerhez az elérési jogosultságok definiálásával.
- Szoros *(hard)*, illetve laza *(soft)* csatlakoztatás: Szoros csatlakoztatás esetén a rendszer addig próbálkozik a kívánt fájl elérésével, míg az sikerrel nem jár. Laza csatlakoztatás esetén néhány sikertelen próbálkozás után a rendszer hibaüzenetet küld.
- A SUN NFS a távoli fájloknak a lokális fájlokkal azonos módon történő elérését biztosítja.
- A szerver csak a saját, lokális fájlrendszerét exportálhatja a rekurzió elkerülése érdekében.

A SUN NFS tervezői a következő célokat tűzték a rendszer elé:

- A protokoll legyen megvalósítható minden operációs rendszer alatt.
- A protokoll hardverfüggetlen legyen.
- Létezzen egyszerű újraindítási lehetőség a kliens, illetve a szerver számára.
- A kliens kezelje az operációs rendszertől függő fájlelérési metodikát.
- Az NFS teljesítménye a helyi fájlrendszer teljesítményével összemérhető legyen.

- A hálózati összeköttetéstől független, illetve a forgalomnövekedéssel bővíthető kapacitású implementációt tegyen lehetővé.

Az NFS specifikáció következménye az állapotmentes megvalósítás. Ennek előnye, hogy a kliensek kérései függetlenek egymástól, így önállóan is értelmezhetők. A kliens állapota a szerverben nem tárolt (az csak statisztikai információt gyűjthet), ami biztosítja a szerver egyszerű újraindíthatóságát.

Hátránya az állapotmentes megvalósításnak, hogy a szerver csak stabil állapotában válaszolhat a kliens kéréseire, ami időkésleltetést okozhat a rendszer működésében. Egy egyszerű példával illusztrálható ez a szituáció. Tegyük fel, hogy a szerver, cache memóriát használ a fájlok elérésére. Az NFS szerver egy fájlírási kérés végrehajtása esetén csak a művelet teljes lezárását követően, azaz a cache lemezre történő kiírása után válaszolhat a kliens kérésére. Ellenkező esetben esetleges rendszerhiba (például áramszünet miatti rendszerleállás) esetén a cache tartalma elveszhet, ami a kliens fájlírás kérésének meghiúsulását jelentené. Ha a szerver a cache kiírása előtt nyugtázná a kliens kérését, akkor nem biztosítaná a kért szolgáltatás biztonságos megvalósítását, ami viszont az RPC protokollban előírás.

5.4.2.2. A SUN NFS részei

A SUN NFS működését nem egyetlen protokoll, hanem egymásra épülő protokollok halmaza biztosítja. A protokollok önmagukban is értelmesek, számos, az NFS-től független alkalmazás használja őket.

A SUN NFS által használt protokollok a következők:

- NFS *protokoll:* a fájlelérés magas szintű protokollja. Definiálja, hogy a kliens és a szerver hogyan tudnak együttműködni. A kliens a szervertől milyen szolgáltatásokat kérhet, milyen kérései lehetnek, illetve a szerver milyen válaszokat adhat. Például: *get/setattrib*(fájl), *lookup*(fájl_név), *write*(fájl), *read*(fájl).
- RPC *(Remote Procedure Call) protokoll*: a távoli eljáráshívás protokollja. Rögzíti, hogy két folyamat hogyan tud egymással kommunikálni, és ezáltal egyik a másik szolgáltatását elérni. Az NFS kliens és a szerver közötti kommunikáció RPC-csomagokkal (RPC-hívásokkal) történik.
- XDR *(EXtended Data Representation) protokoll:* a rendszerfüggetlen adatábrázolást rögzítő protokoll. Szabványos leírás a különböző típusú adatok egységes hardverfüggetlen ábrázolására és kommunikációs csatornán (hálózaton) történő továbbítására.
- *Mount protokoll:* távoli fájlrendszerek összekapcsolását leíró protokoll. A csatlakoztatás a távoli fájlrendszer helyi fájlrendszerben történő láthatóvá tétele, bekapcsolása. A csatlakoztatás után a távoli fájlrendszer a lokális fájlrendszer egyik alkönyvtárán keresztül lesz elérhető. A *mount* protokoll definiálja, hogy hogyan történjen a távoli fájlrendszer bekapcsolása, milyen csatlakoztatással kapcsolatos szolgáltatásokat érhet el a felhasználó.

A *mount* protokoll tipikus szolgáltatásai: *mount:* távoli fájlrendszer helyi fájlrendszerben történő láthatóvá tétele *unmount:* kapcsolat megszüntetése *dump:* a fájlrendszerbe „felmountolt" távoli fájlrendszerek kilistázása.

A protokollokat megvalósító, egy működő SUN NFS rendszerben megtalálható szoftver komponensek:

- NFS-szerver. Az a szoftver egység ami megvalósítja a protokollban definiált szolgáltatásokat, illetve a rendszer működéséhez szükséges egyéb funkciókat (üzenetküldés, fogadás stb.). Tartalmazza – általában függvények formájában – az egyes szolgáltatásokat megvalósító kódot. Például: a szerver oldalon lesz egy *lookup*() függvény, mely egy elérési út/fájl név paraméterrel meghíva kikeresi a megnevezett fájlt és visszaadja annak fájlleíróját.
- NFS kliens kód. A kliens oldali funkciókat (üzenetküldés, fogadás, adatkonverzió stb.) megvalósító szoftver. Biztosítja a kliens oldalon futó alkalmazásnak, hogy a helyi fájlok elérésével azonosan tudják kezelni a távoli fájlokat. Például: kliens az NFS-en keresztül távoli fájlt szeretne elérni, akkor a rendszer az NFS klienshez fordul, amely kapcsolódik a szerverhez, és megvalósítja a távoli fájl elérését.
- Démon (daemon) folyamatok. Az NFS-szerver állandóan elérhető szolgáltatásait megvalósító folyamatok. Ezek az NFS szolgáltatás indításakor elindulnak, és figyelik, majd lekezelik a bejövő kéréseket.

Tipikus démon processzek:

biod: blokkos adatátvitelt kezelő démon. A kliens felől érkező adattömeget kezeli és továbbítja a szerverhez, illetve a szerver felől érkező adattömeget kezeli és továbbítja a kliensnek.

mountd: csatlakoztatással kapcsolatos kéréseket elégíti ki.

nfsd: a fájlok elérésével kapcsolatos kéréseket intézi.

- NLM (Network Lock Manager). Önálló szoftverkomponense a rendszernek. Az NLM segítségével a kliensek jelezhetik, hogy kizárólagosan szeretnének egy fájlt megnyitni és használni, vagyis szeretnék a fájlt zárolni (*lock* művelet). Opcionális szoftver komponense az NFS rendszernek, nem minden NFS használja.
- NSM (Network Status Manager). A fájlok állapotának (lock-olt/nem lock-olt) lekérdezésére szolgáló komponens. Az NSM az NLM-hez hasonlóan opcionális része az NFS-nek.

5.4.2.3. XDR (EXtended Data Representation) protokoll

- Az XDR protokoll rögzíti különböző típusú adatok hardverfüggetlen ábrázolásának, illetve a hálózaton történő továbbításának módját. Definiálja az adatelemek méretét, azok sorrendjét átvitel esetén, valamint az adatelemek formátumát. Például az egész számokat (integer típus) négy bájton

ábrázolja, hálózati transzfer esetén elsőként a legfelső bájtot küldi át. A negatív egészeket kettes komplemens kódolásban ábrázolja. Tömbök átvitele esetén a tömb elé beszúrja a tömb hosszát (5.34. ábra).

Típus	Adat	XDR-ábrázolás			
Egész szám	0x123456	0x00	0x12	0x34	0x56
Háromelemű egészekből álló tömb	[tömb hossza]	0x00	0x00	0x00	0x03
	4	0x00	0x00	0x00	0x04
	2	0x00	0x00	0x00	0x02
	-1	0xFF	0xFF	0xFF	0xFF

5.36. ábra. ábra - Példa az XDR adatábrázolásra

Az XDR által definiált alap-adattípusok halmazát a felhasználó bővítheti. A protokoll az új adattípusok ábrázolására egy szabályrendszert (adattípus-leíró formális nyelvet) ad, melynek segítségével akár kombinált típusok is definiálhatók.

5.4.2.4. Az RPC protokoll

Számos RPC *(távoli eljáráshívás, remote procedure call)* protokoll létezik, több szoftverfejlesztő cég definiált egymással párhuzamosan RPC protokollokat. Az RPC protokollt megvalósító szoftverek nem csak az NFS rendszerekben működnek, az RPC szolgáltatást az operációs rendszer számos más komponense, illetve az alkalmazások is használják.

Az RPC protokoll legfontosabb tulajdonsága, hogy *megbízható* üzenettovábbítást valósít meg a kommunikáló partnerek között. (Meg kell jegyezni, hogy számos más protokoll létezik, ami nem biztosít megbízható üzenettovábbítást.) A megbízható üzenettovábbítás megvalósításának leggyakoribb megoldása, hogy a protokollt megvalósító szoftver addig ismétli a küldendő üzenetet, míg annak vételéről nyugtázás nem érkezik.

Az RPC protokollok előírják üzenetváltás esetére:

- az üzenetek formátumát (mit tartalmazhatnak az üzenetek),
- az üzenetközvetítés módját (mely üzenetek milyen sorrendben küldhetők),
- a partnerazonosítás (címzés) módját.

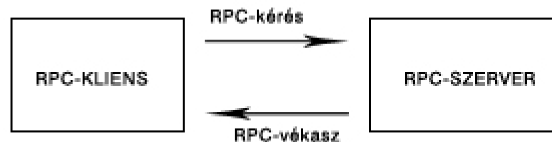
Fontos kérdés az RPC üzenetküldési módjának definiálásakor, hogy a kliens folyamat várakozik-e az általa kért szolgáltatás végrehajtására, vagyis a szerver válaszának megérkezéséig, avagy nem várakozik. Az első esetben *szinkron*, míg a második esetben *aszinkron* RPC protokollról beszélünk. Az RPC protokollok között léteznek mind szinkron, mind aszinkron megvalósítások.

A SUN NFS rendszerben az RPC protokoll a kliens és a szerver kommunikációjának formai definiálására szolgál. A kliens és a szerver között a kérés, illetve a válasz üzenetek RPC csomagok formájában vándorolnak. A SUN NFS a saját fejlesztésű SUN RPC protokollt használja, ami szinkron műveletvégzést és megbízható üzenettovábbítást biztosít. A SUN RPC protokoll az üzenetek formátumának definiálására az XDR protokollt használja.

5.4.2.5. Az RPC protokoll működése

Az RPC protokollt megvalósító rendszerek is a kliens-szerver modellre épülnek. A működés sémáját az 5.37. ábra szemlélteti.

Az RPC-kérés felépítése az 5.38. ábrán látható.



5.37. ábra. ábra - Az RPC működése

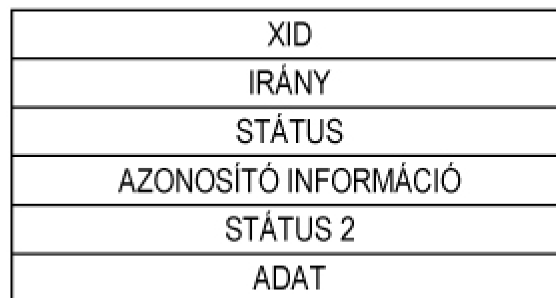
XID
IRÁNY
RPC-VERZIÓ
PRG-AZONOSÍTÓ
PRG-VERZIÓ
SZOLGÁLTATÁS
AZONOSÍTÓ INFORMÁCIÓ
ADAT

5.38. ábra. ábra - Az RPC-kérés felépítése

Az adatelemek magyarázata a következő:

- XID – Minden egyes üzenetnek generál az RPC egy egyedi *azonosítót*, aminek alapján az üzenetet számon tartja, és ismétli, ha nem érkezik rá válasz. Az XID ez az azonosító.
- IRÁNY – A kliens–szerver kommunikációban minden üzenet lehet *kérés* vagy *válasz*.
- RPC verzió – Az RPC protokollnak több verziója létezik. Az üzenet formája (vagyis hogy milyen részeket tartalmaz) függhet az RPC aktuális verziójától.
- PRG-azonosító – Az RPC-t, mint kommunikációs eszközt, több, párhuzamosan futó (applikáció) is használhatja. A PRG-azonosító tartalmazza a kért szolgáltatást nyújtó alkalmazás azonosítóját. Az NFS-szolgáltatások azonosítója például 1000003, vagyis minden NFS-csomagban ez az azonosító szerepel. Ebből tudja az RPC-szerver, illetve kliens, hogy az adott üzenetet az NFS-szervernek, illetve kliensnek kell továbbítania.
- PRG verzió – Az adott szolgáltatás verziója. Az NFS-nek például két verziója létezik: 2-es és 3-as.
- Azonosítási információ – A küldő folyamat azonosítója. UNIX-rendszer esetén ez például a *process_ID* (PID).
- Adat – Az üzenet adatrésze. Ennek tartalma az RPC-t használó alkalmazástól függ.

Az RPC-válasz felépítése az 5.39. ábrán látható.

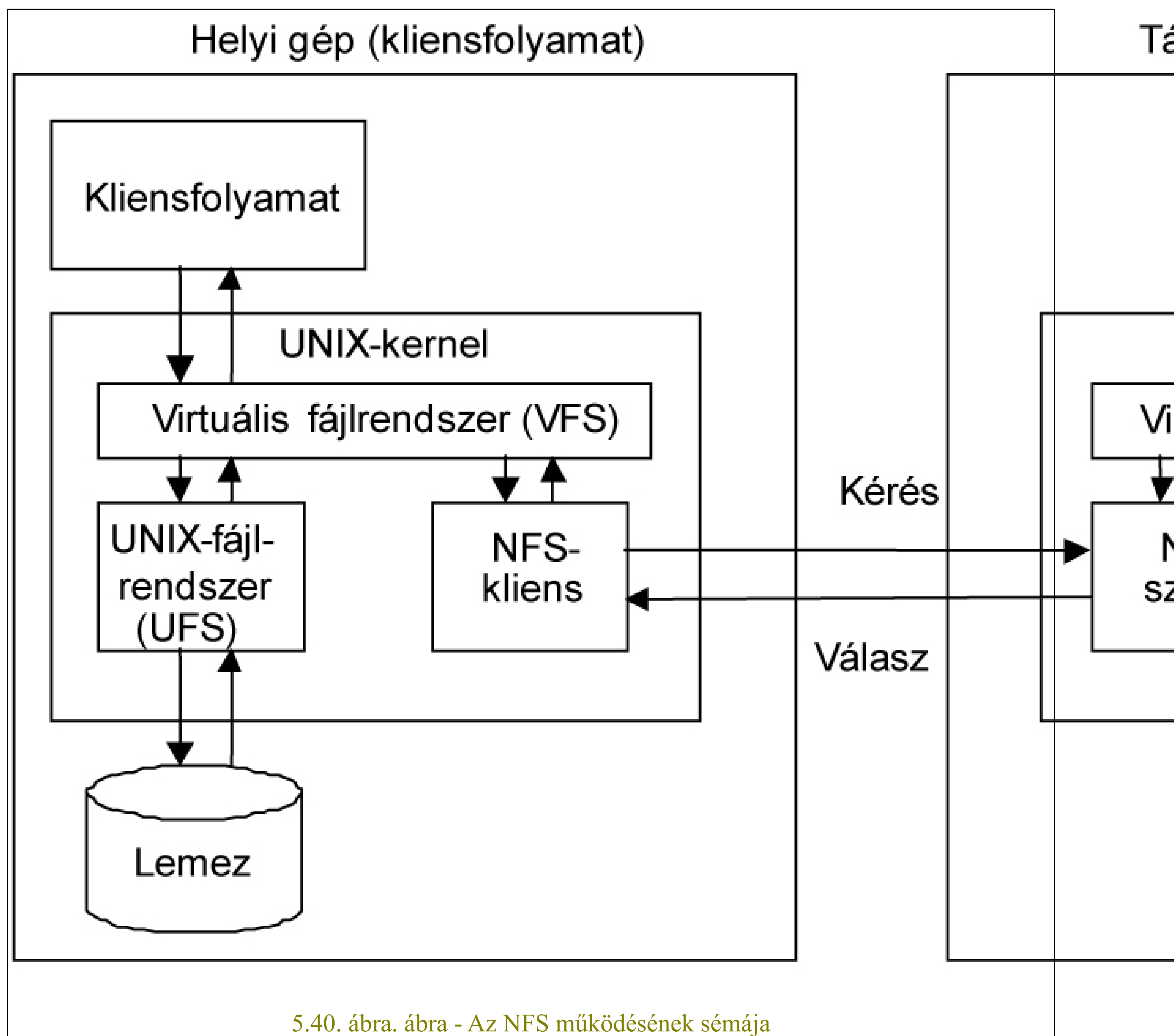


5.39. ábra. ábra - Az RPC-válasz felépítése

Az RPC-válasz adatelemei részben megegyeznek a kérés adat elemeivel. A válaszban szerepel a kért szolgáltatás státusa is, amely megadja, hogy sikeres volt-e a kért szolgáltatás végrehajtása, történt-e hiba a működés során stb.

5.4.2.6. A SUN NFS működése

Az 5.40. ábra az NFS-rendszer működését foglalja össze. Láthatjuk, hogy a fájlok elérését kezdeményező kéréseket, rendszerhívásokat az ún. virtuális fájlrendszer (VFS) kezeli. A virtuális fájlrendszer a hagyományos UNIX-fájlrendszert továbbfejlesztő fájlrendszer, ami lehetővé teszi különböző típusú fájlrendszerek kezelését a UNIX-rendszerben.



5.40. ábra. ábra - Az NFS működésének sémája

A virtuális fájlrendszer, ha lokális fájlt szeretne a kliens folyamat elérni, a kérést a helyi lemezt kezelő UNIX-fájlrendszer (UFS) felé továbbítja, amelyik a fájlt közvetlenül eléri és visszaadja a kért adatokat.

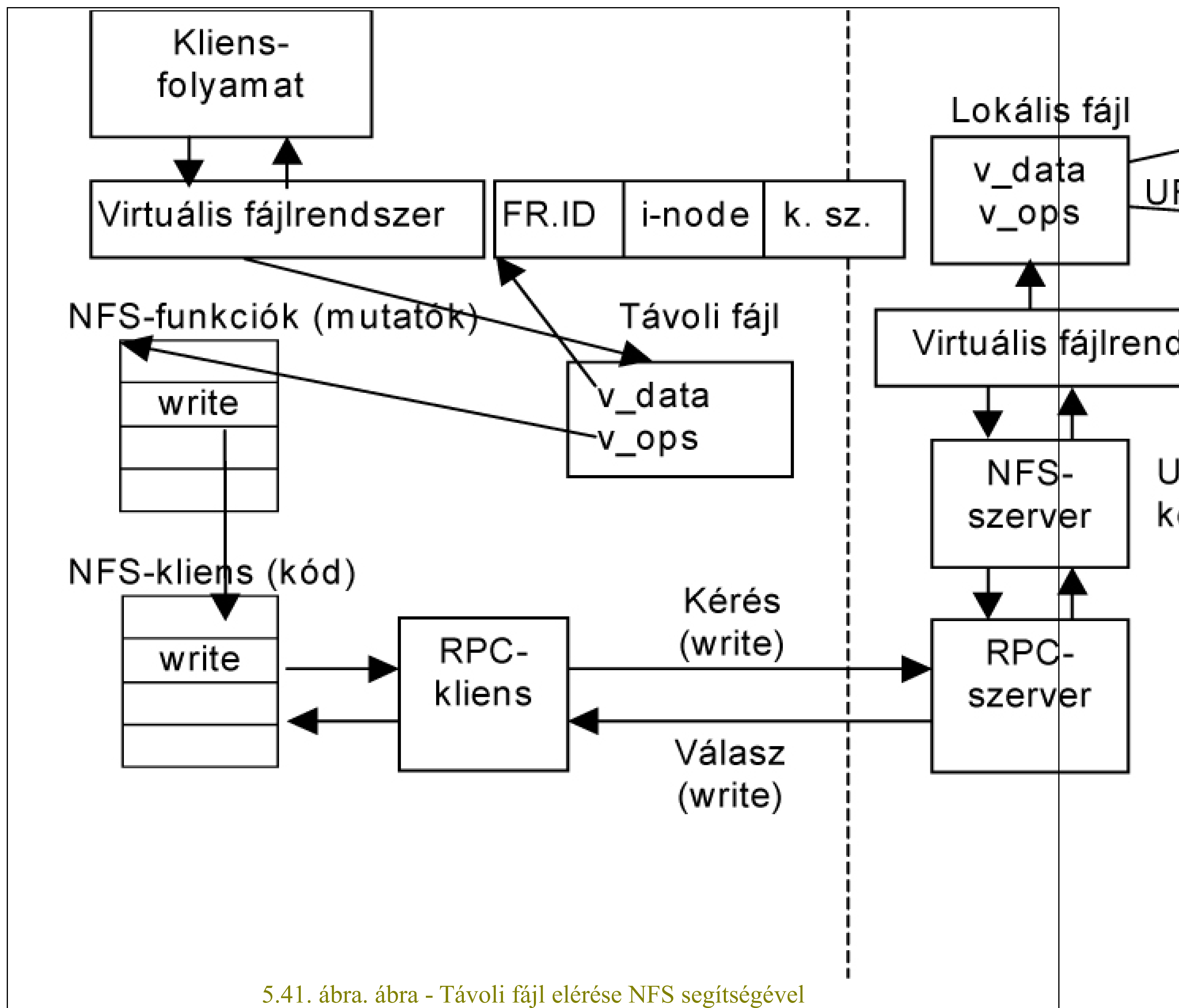
Ha a folyamat távoli fájlt szeretne elérni, a virtuális fájlrendszer a kérést az NFS-klienshez továbbítja. Az NFS-kliens a hálózaton keresztül eléri a kért fájlt tároló csomóponton működő NFS-szervert. A távoli NFS-szerver az ottani VFS-rendszeren keresztül kérést küld a helyi UNIX-fájlrendszernek (UFS). Az UFS visszaadja a kért adatokat az NFS-szervernek, aki továbbítja azokat a kérést kezdeményező kliens folyamat felé az NFS-kliens, illetve a kliens folyamat csomópontján működő virtuális fájlrendszeren (VFS) keresztül.

5.4.2.7. Távoli fájlok elérése NFS használatával

Az NFS-rendszer működését egy példán keresztül foglaljuk össze. Az 5.41. ábrán láthatóan egy kliens folyamat írást *(write)* hajt végre egy távoli fájlon.

A kliens folyamat a hagyományos interfészt használva kezdeményez egy fájlműveletet, vagyis végrehajt egy rendszerhívást [például *write*(fájlleíró_- sorszám, adat)].

A paraméterként megadott fájlleíró sorszám alapján a rendszer kikeresi a folyamatonkénti, illetve a globális fájltáblából a megfelelő bejegyzést. A *v_data* megmutatja, hogy milyen fájlrendszerbe tartozik az elérendő fájl, illetve az elérendő fájlrendszerben mik az azonosító adatai. A *v_ops* megmutatja, hol vannak az adott fájlrendszert kezelő rutin címei. A rendszer tudja, hogy az írás művelete, például a második rutin a fájlkezelő műveletek címeit tartalmazó táblázatban, így elugrik a második címre. A rutinok (távoli fájlról lévén szó) természetesen az NFS klienskódjában vannak implementálva.



5.41. ábra. ábra - Távoli fájl elérése NFS segítségével

Az NFS-kliens készít egy RPC-kérés csomagot, amit elküld a fájlszerver gépén működő RPC-szervernek. Az RPC-szerver észleli, hogy a csomag NFS-csomag, és továbbítja az NFS-szervernek. Az NFS-szerver a helyi VFS-rendszeren keresztül kezdeményezi az adott fájl írását. A távoli gépen (amelyiken a szerver fut) a VFS által használt globális fájl tábla-bejegyzésben természetesen már a lokális fájlok elérésére használt adatelemek lesznek, vagyis

a *v_data* az adott *i-node*-ra mutat, illetve a *v_ops* az UFS-kódjára.

Miután az UFS *write* művelete befejeződött, az NFS-szerver RPC-válasz formájában nyugtát küld vissza az NFS-kliensnek, amely felébreszti a várakozó kliens folyamatot.

5.5. POSIX

A számítógépes rendszerek fejlődésével a ’80-as évek végére egyre inkább igény mutatkozott egy hordozható rendszerinterfész megalkotására. Ezt erősítette, hogy a UNIX-rendszerek fejlődésének akkor közel 20 éve során a UNIX a legkülönfélébb hardverkörnyezetekben jelent meg a kis- és nagyképeken egyaránt, de sajnos némileg eltérő irányzatot képviselve. A változatok közötti kis eltérések azonban megkeserítették a szoftvergyártók életét, akik előtt a UNIX-rendszerek széles körű elterjedésével és elfogadásával felcsillant a lehetőség, hogy ne csak egy hardver platformhoz kötődő alkalmazásokat készítsenek. Ezen probléma megoldásaként született meg a **POSIX** (Portable Operating System Interface) szabvány, amely *az alkalmazói programok szempontjából egy általános felületet kívánt definiálni,* hordozhatóvá téve ezzel a szabványt betartó alkalmazásokat.

Amint a betűszó is utal rá, egy hordozható felületet kívántak a szabványban rögzíteni, de nem titkoltan a UNIX különböző változatainak figyelembevételével. Ez a magyarázata az X megjelenésének a POSIX betűszóban. Az 1990-ben megjelent szabványt röviden POSIX vagy POSIX.1 néven szokás emlegetni. A különböző szabványügyi szervezetek által használt nevek a következők:

- ISO/IEC IS 9945-1:1990
- ANSI/IEEE 1003.1-1990.

Ez a szabvány C nyelven definiál egy operációs rendszer interfészt, ami az alkalmazások szempontjából ír le egy szabványos rendszerfelületet. A POSIX.1 kidolgozásával *párhuzamosan több 1003.x jelű IEEE szabvány* kidolgozása is megkezdődött, melyeket gyakran POSIX.x szabványoknak neveznek. Ezek többnyire az 1003.1 előírásait tekintik alapnak és arra épülnek. Ezek közül a legfontosabbak az 5.42. ábra szerintiek.

A szabványosítási törekvések eredményeként számos más, az alkalmazások hordozhatósága szempontjából fontos szabvány is készült. Ezek nagy száma miatt azonban hamar tarthatatlanná vált az összes egymás melletti, és egymásra épülő szabvány minden kombinációjának megtartása, hiszen adott esetben még az is előfordulhatna, hogy két alkalmazás azért nem kompatíbilis, mert az alkalmazás számára előírt szabványok ellentétesek egymással. Ezt az elvi „rendetlenséget” próbálják feloldani az ún. **alkalmazási profilok** (AEP: Application Environment Profile). Ezek az alkalmazások különböző jellegéhez alakított szabványhalmazok. Így külön szabványhalmaz definiálja például a hálózati alkalmazások környezetét,a CAD alkalmazások környezetét stb.

A továbbiakban összefoglaljuk az egyik legfontosabb szabvány, a POSIX.1 célkitűzéseit és a szabvány környezetét, felépítését, fontosabb témáit. Az egyszerűség kedvéért az 1003.1 jelű szabványra egyszerűen csak POSIX néven hivatkozunk, ahol ez nem zavaró.

1003.0	Útmutató a POSIX nyílt rendszerkörnyezethez
1003.1	Rendszer interfész (C nyelven)
1003.2	Shell-ek és segédprogramok
1003.3	Tesztelési eljárások
1003.4	Real time kiterjesztés az 1003.1-hez
1003.5	Ada interfész az 1003.1-hez
1003.6	Biztonság
1003.7	Rendszeradminisztráció
1003.8	Transzparens állományelérés
1003.9	Fortran interfész az 1003.1-hez
1003.10	profil (AEP) a szuperszámítógépekhez
1003.11	profil (AEP) a tranzakciókezeléshez
1003.12	Protokoll-független hálózati elérés
1003.13	Real-time profil (AEP)

5.42. ábra. ábra - A legfontosabb POSIX.x szabványok

5.5.1. Alapfogalmak, felépítés

A POSIX.1 alapvető célkitűzése, hogy olyan rendszerinterfészt definiáljon, amely alkalmas hordozható alkalmazások készítésére. Ezt több mint 200 C függvény segítségével adja meg, melyek többsége a két nagy UNIX-irányzat, a BSD és a System V rendszerhívásával vagy rendszerfüggvényével azonos. Van azonban néhány terület, ahol a szabvány – szakítva a BSD és a System V megoldásaival – egy harmadik megoldást ír elő.

Fontos hangsúlyozni, hogy *a szabvány* csupán *rendszerfelületet definiál, és nem írja elő annak megvalósítási módját,* így nem írja elő azt sem, hogy egy adott függvény valójában egy rendszerhívás, vagy csupán egy könyvtári függvény. A megvalósítás oldaláról nézve a UNIX csupán egy lehetséges megvalósítás, de lehet más rendszer is POSIX-megfelelő.

Ha POSIX-megfelelőségről beszélünk, akkor értelemszerűen azt vagy az operációs rendszer oldaláról, mint egy POSIX-implementációról, vagy az alkalmazás oldaláról tesszük. Implementáció oldalról a megfelelőség azt jelenti, hogy egy adott operációs rendszer mennyire felel meg a szabvány előírásainak. Egy konkrét operációs rendszer megfelelőségét dokumentációval kell alátámasztani. A dokumentáció szerkezetének pontosan követni kell a POSIX-szabvány szerkezetét, amelyben nyilatkozni kell a megvalósítás megfelelőségéről. Maga a

szabvány *6 különböző kategóriába sorolja az előírásokat*, attól függően, hogy milyen jellegű az előírás. Ezek a kategóriák a következők:

- **Implementáció által definiált** (implementation-defined). Ezzel olyan működést, vagy értéket jelöl, amelyet a szabvány nem köt meg, de az implementáció megfelelőségi dokumentumában pontosan meg kell adni annak értékét, vagy le kell írni a pontos működést. Az ilyen jellegű dolgokat alkalmazás oldalról ki lehet használni, de lehetőleg kerülni kell.
- **Nem specifikált** (unspecified). Ezzel olyan működést vagy értéket jelöl a szabvány, mellyel szemben nincs semmilyen követelmény. Az implementáció szabadon rendelkezhet és a konkrét működést, vagy értéket nem kell dokumentálni.
- **Nem definiált** (undefined). Ez olyan működést, vagy értéket jelent, mellyel szemben egy hibátlan program nem támaszthat követelményeket. Természetesen ilyet az alkalmazás oldalról nem szabad kihasználni.
- **Kell** (shall). Az implementáció pontosan előírja a működést vagy a kérdéses értéket.
- **Kellene** (should). Ez egy javaslatot ír elő az adott értékre, vagy működésre, de az implementáció ettől eltérhet.
- **Lehet** (may). Ez az implementáció számára egy opciót jelöl, amit nem kötelező megvalósítani.

Az alkalmazás oldalról a megfelelőség tekintetében négy egymásra épülő szintet különböztetünk meg (5.43. ábra).

- **Szigorúan megfelelő** alkalmazás. Olyan alkalmazás, amely csak a POSIX-szabványra épül, és hibátlanul működik a szabványban nem specifikált és implementáció által definiált tulajdonságok és paraméterek tetszőleges értékével.
- **ISO/IEC-megfelelő** alkalmazás. Olyan alkalmazás, amely a POSIX-szabvány mellett felhasznál más ISO/IEC szabványokat is.
- **Nemzeti szabványokat is felhasználó** alkalmazás. Olyan POSIX-alkalmazás, amely nemzeti szabványokat is felhasznál.
- **Egyéb nem szabványos kiterjesztéseket** is felhasználó alkalmazás. Olyan POSIX-alkalmazás, amely felhasznál nem szabványos kiterjesztéseket is.

5.43. ábra. ábra - Alkalmazások megfelelősége

5.5.2. POSIX környezet

A szabvány értelmezési és felhasználási területét definiálja az a leírás, amely a környezetet adja meg. Ez a környezet *sokban hasonlít a UNIX-rendszerhez, de az implementációt tekintve nem feltétlenül UNIX*, hiszen a szabvány nem definiál implementációt, csak felületet. A környezet legfontosabb elemei a következők:

- Többfelhasználós, több folyamat konkurens futtatására alkalmas környezet, melyben a folyamatoknak és a felhasználóknak egyedi azonosítójuk van.
- Hierarchikus, nem tisztán fa szerkezetű állományrendszer. Fontos kiemelni, hogy a hierarchianevek szintjén, azaz egy névtérben jelenik meg. Az állományoknak van ugyan egy egyedi sorszám-azonosítója, de a megvalósítást a szabvány nem írja elő.
- A felhasználók adatainak elérését védelmi rendszer szabályozza, amely hasonlít a UNIX-rendszeréhez, de lehet attól szelektívebb, azaz szigorúbb is.

A szabvány a UNIX-implementációk felhasználásával keletkezett ugyan, de a fenti környezetet tekintve nem zár ki más rendszereket (például VMS, OS/2 MVS) sem.

5.5.3. Hordozható alkalmazások

Ahogyan azt a szabvány fogalmainak ismertetéskor láttuk, az implementáció tekintetében vannak választási lehetőségek (should, may). Ezek elsősorban a különböző UNIX-változatoknak a szabványhoz való igazítása miatt jöttek létre. A *legfontosabb* ilyen *implementáció által definiált területek* a következők:

- Job-kontroll megvalósítása.
- Elmentett *set-uid* megvalósítása.
- *chown()* kiterjesztett megvalósítása. Ez azt jelenti, hogy az implementáció lehetősége, hogy a System V filozófiának megfelelően minden tulajdonos használhatja a *chown()* rendszerhívást, vagy csak a *superuser* (POSIX-terminológiával: *megfelelő* user).
- Hosszú állománynevek kezelése.
- Speciális karakterek kezelésének tiltása.

A hordozhatóság érdekében a POSIX-szabvány által megengedett lehetőségekhez az alkalmazásokat úgy kell elkészíteni, hogy ezeket vagy ne használják ki, vagy rugalmasan alkalmazkodjanak az adott implementáció tulajdonságaihoz. Ez utóbbi megvalósítására a szabvány két lehetőséget kínál.

- **Fordítási időben** megfelelő feltételes fordítási opciók és konstansok használatával. Ezt a C nyelvi interfészen a *<unistd.h>* állomány konstansai biztosítják.
- **Futási idejű lekérdezéssel** a *sysconf(), pathconf()* illetve *fpathconf()* függvények használatával.

A hordozhatóság lehetősége azonnal felveti azt a kérdést, hogy hogyan, milyen formátumban lehetséges az adatok illetve programok átvitele az egyik környezetből a másikba. A POSIX.1 rendszerfelületet definiál ugyan, de mivel ezt a kérdést nem tudta megkerülni, egy külön fejezetben ajánlást tesz hordozható formátumokra és az ezeket előállító segédprogramokra is, amit később az 1003.2 (POSIX.2) jelű szabvánnyal foglalkozó bizottság dolgozott ki részletesen.

A POSIX.1 alapján egy alkalmazás új környezetbe való telepítéséhez a következő információkat kell egy hordozható csomagba becsomagolni:

- **Az alkalmazás programkódját**
-
    1. Forráskódban. Ez a kódkészlet problémáját veti fel, mivel a POSIX az ISO 646 kódkészletre épül. Sajnos az ISO 646 kódkészletben számos jel nem definiált, amit viszont a legtöbb programozási nyelv, így a C is használ. Ezek

helyettesítésére az ún. **trigraph karaktereket** használhatjuk. Ezek 3 karakteres karaktersorozatok, amelyek helyettesítik a megfelelő jelet.

2. Bináris változatban. Bináris formában természetesen csak bizonyos korlátozásokkal lehetséges programokat egyik környezetből a másikba átvinni. A POSIX.1 utal arra, hogy létezik ún. **ABI** *(Application Binary Interface)*, amivel részben lehetséges a probléma kezelése. A másik ígéretes próbálkozás az **ANDF** *(Architecture Neutral Distribution Format)*, amely egy közbülső kódot definiál. Ez a közbülső kód minden lényeges információt tartalmaz a végleges kód generálásához. A kódgenerálás fázisa így eltolódhat az adott architektúrán történő installálás pillanatáig.

- **Az alkalmazás adatait**, ami tartalmazza az on-line **dokumentációt is**. Ez az eltérő számábrázolási formátumok és pontosságok problémáját veti fel. Ezek elkerülhetők, ha az adatokat is szövegként visszük át, bár ahogyan a forráskód átvitelénél tárgyaltuk ez sem teljesen problémamentes.
- **Az installációs utasításokat** és scripteket. Annak ellenére, hogy a POSIX.1 nem elvileg alapszik más szabványokon, ezen a téren a POSIX.2-re utal, mint ajánlásra, hiszen ez foglalkozik a *shell*ekkel.

Egy hordozható alkalmazásnál különös figyelemmel kell lenni arra, hogy az alkalmazásban használt állományok nevei hordozhatók legyenek, és lehetőleg ne tartalmazzon abszolút útnévvel állományhivatkozást.

Az állományok csomagolására a *cpio* és a *tar* segédprogramokat javasolja a szabvány. Ezek kimeneti formátuma kellően egyszerű, így szinte minden operációs rendszerben lehetséges a kezelésük. Mindkét programnak vannak előnyös és hátrányos tulajdonságai a másikkal szemben, ami használatukat bizonyos körülmények között korlátozza. Így például a *tar* nem menti ki a *device* típusú állományokat. A *cpio* ezeket kimenti ugyan, de visszatöltéskor a *tar*ral ellentétben nem tartja meg a *linkeket*.

ASCII	Trigraph
[	??(
]	??)
{	??<
}	??>
\	??/
|	??!
^	??'
#	??=
~	??-

5.44. ábra. ábra - Trigraph karakterek

5.5.4. Folyamatkezelés

A POSIX-*környezetben a folyamatok konkurensen futnak*, melyeket a UNIX-rendszereknél megismert módon a *fork()* rendszerhívással lehet létrehozni, és az ott megszokott jellemzőkkel rendelkeznek. A szabványba a folyamatkezelés terén alapvetően a System V rendszerekben alkalmazott megoldások kerültek be. BSD-örökség viszont a *kiegészítő csoportazonosítók* és a *jelzés-maszk* alkalmazása. A terminálhoz rendelt szekciók (session) ötlete a két irányvonal ötvözéseként jött létre, ami lehetőséget ad a BSD-rendszerekben támogatott ún. job-kontroll koncepció, és a System V rendszerekben alkalmazott folyamatcsoport használatára is. Ennek lényege az, hogy minden folyamat tagja egy szekciónak és egy folyamatcsoportnak is. Egy szekcióhoz egyszerre több folyamatcsoport is tartozhat, ami lényegében a BSD job koncepciójának felel meg.

A folyamatok legfontosabb tulajdonságai a következők:

- Folyamatazonosító (PID). Egyedi azonosítószám.
- Szülő folyamat azonosítója (PPID).
- Folyamat csoportazonosító (PGID).
- Login név.
- Valós felhasználói azonosító (UID).

- Effektív felhasználói azonosító. Ez az ún. *set-uid* bittel rendelkező programoknál eltérhet a valós UID-től. Ez lehetőséget ad arra, hogy átmenetileg, vagy véglegesen a folyamatot indító felhasználó azonosítójától eltérő azonosítóval illetve jogokkal rendelkezzen a folyamat.
- Valós felhasználói csoportazonosító (GID).
- Effektív felhasználói csoportazonosító. Ez az ún. *set-uid* bittel rendelkező programoknál eltérhet a valós GID-től. Ez lehetőséget ad arra, hogy átmenetileg, vagy véglegesen a folyamatot indító felhasználó csoportazonosítójától eltérő azonosítóval, illetve jogokkal rendelkezzen a processz.
- Kiegészítő csoportazonosítók.
- Munkakatalógus.
- Állományok létrehozási maszkja (UMASK). Ez egy bitmaszk, mely minden új állomány létrehozásánál részt vesz a védelmi attribútumok kialakításában.
- Jelzés maszk.
- Terminál azonosító.
- Szekció (session) azonosító.
- Futási idők.

A szabvány a folyamatkezeléshez a következő, a UNIX-rendszerekből jól ismert függvényeket definiálja: *abort(), exit(), execl(), execle(), exelp(), execv(), execve(), execvp(), wait(), waitpid()*.

5.5.5. Állománykezelés

A szabvány állománykezeléssel foglalkozó része azt a programfelületet definiálja, ahogyan az állományokat az egyes alkalmazások elérhetik. *Az állományrendszer felépítésével a POSIX csak* az 5.4.2. pontban ismertetett *névtér absztrakció szintjén foglalkozik*. Ez azt jelenti, hogy az állományokat, pontosabban azok neveit *nem tisztán fa* struktúrájú *hierarchiában* képzeljük el. Ebben a hierarchiában öt állománytípust különböztetünk meg:

- normál állomány (regular file),
- katalógus (directory),
- FIFO,
- blokkos elérésű eszköz,
- karakteres elérésű eszköz.

A szabvány javasolja (should), hogy az *állományok nevei ún. hordozható állománynevek legyenek*. Ezek csak az angol abc kis- és nagybetűit, a pont (.), az aláhúzás (_) és a kötőjel (-) karaktereket tartalmazhatják. Ezen felül a nevek nem kezdődhetnek kötőjellel sem. Az útnevekben az egyes komponenseket per (/) karakterrel kell

elválasztani. Egymás után több ilyen elválasztójel is lehet, kivéve az útnév legelején. Az egymás utáni / jelek egy / jelnek értelmeződnek.

Az állományok legfontosabb tulajdonságai a következők:

- Állomány típusa.
- Hozzáférés védelmi kódja (permission).

A POSIX a UNIX 3x3-as védelmét követeli meg, de implementáció által definiált módon megenged ezen felüli védelmet is. Ha ilyen létezik, akkor:

- Annak állományonként engedélyezhetőnek illetve tilthatónak kell lennie.
- Az alkalmazói program szempontjából ugyanúgy kell megjelennie, mint az eredeti 3x3-as védelem, tehát létezni kell a „tulajdonos”, „csoport” és „bárki” fogalmaknak.
- Ha engedélyezve van, akkor az a korábbi védelmet felülírva jelenik meg a különböző rendszerhívásoknál (például *stat*, *fstat*).
- Egyedi azonosítószám. Ez gyakorlatban az eszköz azonosítóját és az *i-node* számot jelenti, de ez már implementáció kérdése.
- Hivatkozás (link) számláló.
- Tulajdonos azonosítója és csoportazonosítója (UID, GID).
- Állomány hossza byte-ban.
- Utolsó módosítási idő.
- Utolsó hozzáférési idő.
- Utolsó státusmódosítási idő. A fenti adatokat tároló adatstruktúra módosítási ideje. Gyakorlatban az *i-node* módosítási ideje.

Az állományok kezelése az ANSI C alacsony és magas szintű függvényeivel történhet a POSIX rendszerekben. Ezek leírására nem térünk ki részletesen, csupán néhány fontos elemet ragadunk ki, amelyek némileg eltérnek mind a BSD, mind a System V megoldásaitól.

- Nem használható az *mknod()* rendszerhívás. Helyette a speciális bejegyzések külön hívással állíthatók elő (például *mkdir(), mkfifo()*).
- A katalógusok kezelésére külön interfész van (*mkdir(), rmdir(), opendir(), readdir(), rwinddir(), closedir()*). A POSIX előírásai szerint a katalógusok nem olvashatók normál állományként, csak a fenti függvényeken keresztül, és csak szekvenciálisan érhetők el.

- A szabvány bevezeti a *rename()* rendszerhívást, bár normál állományokra a *link()* és *unlink()* páros is használható, de katalógusokra csak a *rename()*.
- Az állományok megnyitása más UNIX-változatokhoz hasonlóan lehetséges *O_APPEND* illetve *O_NONBLOCK* módon is, de a POSIX nem támogatja az *O_SYNC* flag használatát, és *sync()* rendszerhívás sincs a szabványban. Ez utóbbi helyett egy *fsync()* rendszerhívást definiál, ami állományokra szelektíven adható ki. A megnyitott állomány különböző tulajdonságai az *fcntl()* függvény használatával lekérdezhetők illetve beállíthatók. Ezzel oldhatók meg a kölcsönös kizárást biztosító zárolások *(lock)* is.

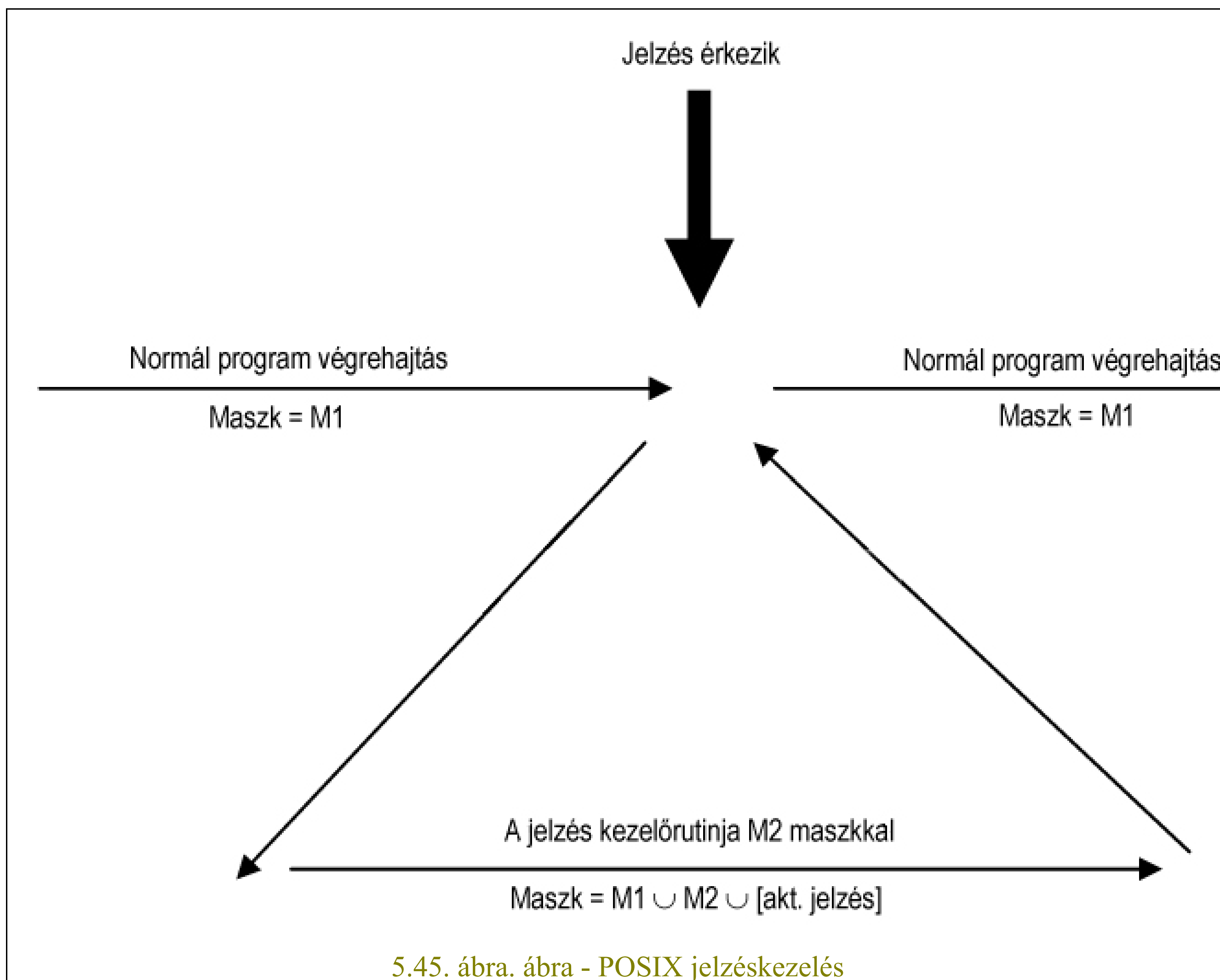
5.5.6. Jelzéskezelés

A POSIX jelzéskezelés az *egyik olyan terület, amely teljesen szakított a UNIX-hagyományokkal* és a rendszerfelületet tekintve eltér a UNIX-irányvonaltól. Ennek oka, hogy mind az eredeti System V, mind a BSD-jelzés kezelése megoldhatatlan problémákat vet fel. A fő gondot az a dilemma jelenti, hogy mi történjen abban az esetben, ha egy jelzés érvényre jut, majd hatására *elindul az alkalmazásban definiált kezelő rutin, és közben újból bejön ugyanaz a jelzés*.

A System V filozófia szerint a jelzés érvényre jutásakor az adott jelzéshez tartozó kezelési mód kijelölése visszaáll az alapértékre. Ha az alkalmazásnak ez nem felel meg, akkor a kezelőrutinban újra át kell állítani, különben az alapértelmezés szerinti tevékenységet hajtja végre a rendszer, ami a legtöbb jelzés esetén programmegállást jelent. Azonban hiába veszi magához a program újból az adott jelzést a kezelőrutin elején, lehet hogy ott már késő, ugyanis van egy rövid idő, amikor a jelzés már alaphelyzetben van, és már érvényre juthat a következő jelzés.

A BSD filozófia szerint a jelzéshez rendelt kijelölés nem áll vissza alaphelyzetbe. Ez viszont feltételezi, hogy a kezelő rutin újrabelépő. Ami nem minden esetben teljesíthető.

A POSIX jelzéskezelése némi *hardveres szemléletet tükröz.* Ahogyan egy hardver megszakítás is maszkolható, úgy egy POSIX folyamathoz és a jelzést lekezelő rutinhoz is hozzárendelhető egy maszk. Ez az ún. jelzésmaszk előírja, hogy az adott pillanatban mely jelzések nem juthatnak érvényre, azaz blokkolódnak (5.45. ábra). Ez a megoldás a jelzéskezelés megszokott interfészét is érintette, hiszen egy adott szignáljelzéshez nemcsak a kezelőrutint kell hozzárendelni, hanem egy maszkot is. Így a jelzéskezelés beállításához nem a UNIX *signal()* rutinját, hanem a *sigaction()* POSIX-rutint kell alkalmazni, melynek egy struktúrában kell megadni a megfelelő paramétereket. A folyamat jelzésmaszkja a folyamat keletkezése során öröklődik, amit maga a folyamat is megváltoztathat a *sigprocmask()* rendszerhívással. Ha egy jelzés érvényre jut, akkor a kezelőrutin futása alatt egy új maszk jut érvényre, ami az eredeti maszk, a kezelő rutinhoz rendelt maszk, és az aktuális jelzés sorszámának uniójaként keletkezik. Ezzel és a maszkot manipuláló függvényekkel (*sigemtyset(), sigfillset(), sigaddset(), sigdelset(), sigismember()*) korrektül kezelhetők a fenti problémák.



5.45. ábra. ábra - POSIX jelzéskezelés

A jelzés küldésére továbbra is megmaradt a UNIX-rendszerekben használatos *kill()* függvény. Hasonlóan használható a *pause()* függvény is, amely egy jelzést generáló eseményre való várakozásra használható. Ennek POSIX kiterjesztése a *sigsuspend()* függvény, amelynek paraméterként megadható a várakozás idejére érvényes jelzésmaszk is.

5.5.7. Terminálkezelés

A POSIX terminálkezelés új elemeket és régi UNIX-hagyományokat is tartalmaz. Alapjában a terminál kezelését befolyásoló paraméterek és működési módok lényegében változatlanul maradtak, de a kezelést végző függvények teljesen megváltoztak. Így megmaradtak az input és output kezelését befolyásoló speciális karakterek és az input két fő működési módja a kanonikus és nem kanonikus mód. Megszűnt viszont a korai UNIX-verziókban megjelenő ioctl() függvény, mivel nem illeszkedett sem a POSIX sem az ANSI C filozófiájához. Ezt teljesen kiváltották a tcgetattr() és a tcsetattr() POSIX függvények. Az ioctl() függvény leginkább azért nem felelt meg a szabvány követelményeinek, mert nagyon sok feladata volt, és ennek

megfelelően különböző módon kellett paraméterezni. Jellemző a függvény összetettségére, hogy a harmadik argumentumának típusa és jelentése függött a második argumentumtól.

A terminálkezelés tekintetében azért is különösen fontos a strukturált, átlátható módszerek használata, mert ha egy program megváltoztatja a terminál kezelésének módját és paramétereit, az hatással van az összes azonos terminálról futó programra is. Ezért külön hangsúlyozza a szabvány, hogy a programok indulásakor el kell menteni az eredeti beállításokat, mielőtt megváltoztatnánk azokat, hogy az alkalmazás megállásakor az eredeti állapot visszaállítható legyen. Fontos továbbá, hogy a visszaállítás a különböző hibaágakon, illetve megszakítás hatására történő megálláskor is megtörténjen.

5.6. A LINUX-RENDSZER

A *Linux operációs rendszer* tulajdonképpen a UNIX önálló változata, amely az elmúlt évek során egyre növekvő népszerűségre tett szert. A rendszer napjainkban is egyre inkább elterjedőben van, amit nagymértékben elősegít az a tény, hogy a különböző fejlesztési változatainak forráskódja bárkinek ingyenesen rendelkezésére áll az Interneten. Ez egyben azt is jelenti, hogy a fejlesztésekbe is szabadon kapcsolódhatnak be az abban érdekelt műhelyek, intézmények. A szabad hozzáférés lehetősége egyébként a Linux eddigi teljes történetére jellemző volt, aminek következtében az egyes változatok a világ különböző felhasználóinak és fejlesztőinek együttműködéséből jöttek létre, javarészt az Interneten keresztül történő kommunikáció révén.

Elvileg bárki installálhat egy Linux-rendszert azáltal, hogy az Internetről *ftp*-vel letölti a legújabb rendszerkomponenseket, majd lefordítja és összeszerkeszti őket. Ezt a munkát lényegesen megkönnyítették azok a fejlesztők, akik standard, előreszerkesztett csomagokat tettek közzé a könnyebb installálás érdekében. Az így előkészített összeállítások általában jóval többet rejtenek magukban, mint az alapkiépítésű Linux. Tipikusan olyan komponenseket is tartalmaznak, mint rendszerkezelési segédprogramok *(utilityk)*, webkeresők, szövegfeldolgozó és szerkesztő eszközök, sőt még játékprogramok is. A legújabb megoldások szerint már külön csomagkezelő eszközök állnak rendelkezésre, a keresést megkönnyítő adatbázissal. Mindez a csomagok installálását, követését, módosítását, *upgrade*-olását, illetve eltávolítását segíti elő Interneten keresztül.

A Linux felhasználási jogát illetően a következő kikötések, feltételek vannak érvényben.

A Linux nem tekinthető ún. nyilvános *(public-domain)* szoftvernek. A *public-domain* megjelölés azt jelenti, hogy a szerzők nem tartanak igényt semmiféle szerzői jogra. A Linux-kód fölött azonban érvényesül a szerzőségi jog. Ugyanakkor a Linux *szabad* szoftver. Szabad abban az értelemben, hogy bárki lemásolhatja, módosíthatja, használhatja oly módon, ahogy neki tetszik, és továbbadhatja a saját példányait bármiféle megszorítás nélkül. A legfőbb kikötés itt az, hogy akár felhasználó, akár szerző lesz valaki, nem teheti saját tulajdonává az általa használt, vagy módosított terméket, így pénzért nem is forgalmazhatja azt. A másik kikötés az, hogy ha valaki kibocsát egy általa létrehozott példányt, akkor a bináris kódon kívül köteles a forráskódot is közreadni.

A fentiekhez kapcsolódóan érdemesnek tartjuk még megjegyezni, hogy ugyancsak széles körben terjedt el az A. S. Tanenbaum professzor és munkatársai által kifejlesztett *MINIX* nevű operációs rendszer (első változata

1987-ben jelent meg), amelynek terjesztése a Linuxéhoz hasonló elvek szerint történik. A MINIX elsősorban oktatási célokra használható fel jól, valójában a UNIX kicsinyített változata. Igen alkalmas arra, hogy az operációs rendszerekkel ismerkedő informatikusok ennek a rendszernek a tanulmányozása és használata alapján mélyítsék el tudásukat. A MINIX C nyelvű forráskódja megtalálható az 1999-ben magyarul is kiadott, az operációs rendszerekkel foglalkozó Tanenbaum-könyvben.*

5.6.1. A Linux fejlődésének állomásai

A Linux fejlesztése 1991-ben vette kezdetét, amikor egy finn egyetemista, Linus Torvalds önálló kernelt írt Intel 80386 processzorra. A Linux elnevezés tőle (a saját nevéből) származik. A kernel forráskódját szintén ő tette publikussá Interneten. Fontos még megemlíteni, hogy az Intel-386 volt az első 32 bites processzor a PC-kompatíbilis CPU-k körében.

A Linux-projektekben mindvégig alapvető tervezési szempont volt a UNIX-szal való kompatibilitás, azaz a UNIX-on futó standard alkalmazások többségének futniuk kellett Linuxon is.

Az 1991-ben kibocsátott első kernel még meglehetősen korlátozott felhasználást tett lehetővé. Nem rendelkezett hálózati támogatással, csak személyi számítógépen működött, ezenkívül igen kevés perifériát támogatott, kevés készülékkezelő (device driver) állt rendelkezésre.

A Linux történetében a következő fontos állomás az 1.0 verzió volt, amelyet 1994-ben bocsátottak ki. Ez már képes volt a hálózati működtetésre, a UNIX által is támogatott TCP/IP-protokollt használta, továbbá BSD-kompatíbilis *socket* interfészen keresztül lehetővé tette a hálózati programozást is.

Az 1.0-s kernel már fejlett fájlkezelő rendszerrel volt ellátva, amely nagyteljesítményű diszkelérést is lehetővé tett. A virtuális memória alrendszert kiegészítették a lapkezeléssel (paging), ami a tárcserét *(swapping)* nagyban meggyorsította. Jóllehet ez a verzió még mindig csak Inteles PC-ken üzemelt, a hardvertámogatás tovább bővült a floppy diszkek, CD-ROM-ok kezelésével, továbbá hangkártyák, nemzetközi billentyűzetek stb. használatával.

Három közbülső verzió megjelenését követően, a következő kiugró állomás a Linux 2.0 volt, 1996-ban. Ez két területen tartalmazott lényeges előrelépést: egyrészt többfajta architektúra használatát engedte meg, beleértve a tiszta 64 bites Alpha portot is, másrészt pedig alkalmassá vált a többprocesszoros működésre. A 2.0-s verzió már futott Motorola 68000 sorozatú processzorokon, továbbá a Sun Sparc rendszereken is.

További lényeges előrelépés történt a TCP/IP-protokollkezelés teljesítményében, és számos új hálózati protokollt fogadott be a rendszer, valamint belekerült az ISDN-támogatás is.

Ugyancsak fontos javítás volt még a belső kernelszálak lehetővé tétele, ami a betölthető modulok közötti függőségkezelésére alapozva a modulok automatikus betöltésének támogatására szolgált.

5.6.2. A Linux felépítése és működése

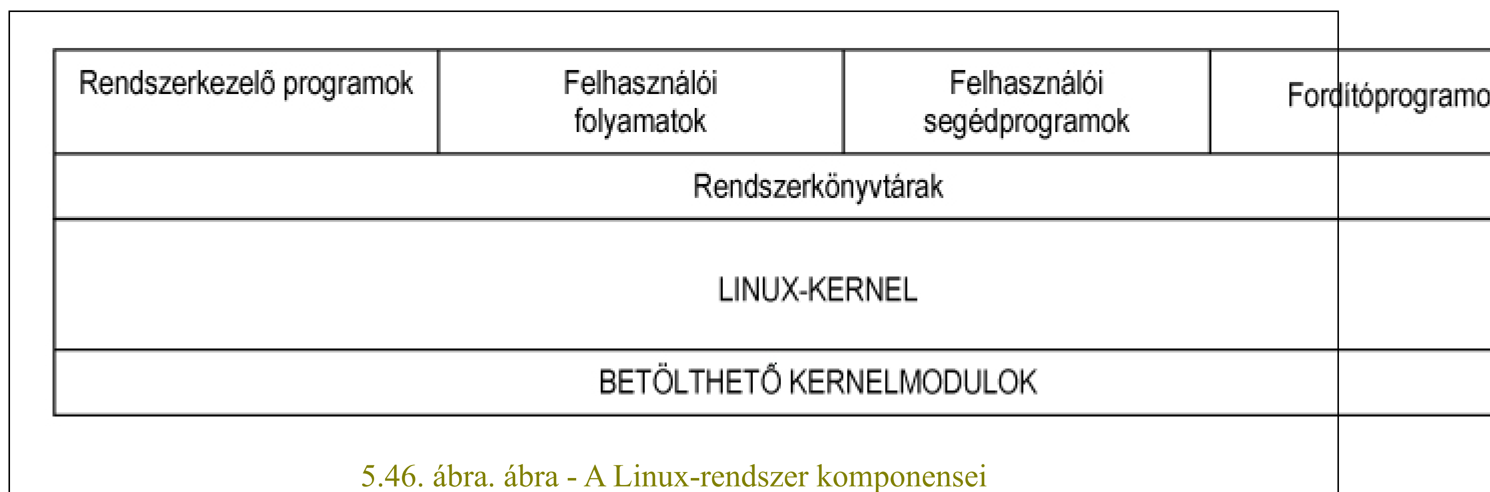
Általános tervezési szempontból a Linux hasonlít bármelyik hagyományos UNIX-implementációhoz. Többfelhasználós, multiprogramozott rendszer a UNIX-kompatíbilis eszközök teljes készletével. Fájlrendszere a

tradicionális UNIX-szervezést követi, ugyanazzal a hierarchikus könyvtári fa-struktúrával és ugyanazzal a nyelvi szemantikával. A fájlrendszer a standard UNIX-os hálózati modellt is teljes mértékben megvalósítja.

A Linux-rendszer kódja három fő részből tevődik össze, a legtöbb normál UNIX-implementációval teljes összhangban. A részek a következők:

- **Kernel:** az operációs rendszer összes fontos belső funkcióját látja el, mint például a virtuális memória kezelése, vagy a folyamatkezelés.
- **Rendszerkönyvtárak:** elemei azon standard funkciókat valósítják meg, amelyeken keresztül az alkalmazások együtt tudnak működni kernellel, másrészt pedig azokat a tevékenységeket látják el, amelyek nem igénylik a kernel felügyeletét.
- **Segédprogramok (utilityk):** olyan programok, amelyek egyedi, speciális feladatokat látnak el. A utilityk egy része inicializálásra, valamint konfigurálásra való. Mások folyamatosan használatban lehetnek: a hálózati kapcsolatokat intézik, logon-kérést fogadnak terminálról, vagy naplózási fájlokat újítanak fel.

Az 5.46. ábrán láthatjuk a Linux-rendszer felépítését. A legfontosabb különbség a kernel és a többi komponens között, hogy a kernel a processzor privilegizált (ún. kernel) végrehajtási módjában hajtódik végre, így a kernel szabadon eléri a számítógép összes fizikai erőforrását. A kernel a Linuxban egyáltalán nem tartalmaz felhasználói módú kódot. A processzor privilegizált módba történő átkapcsolása a rendszer programozói interfészét adó rendszerkönyvtárak rutinjaiban elhelyezett rendszerhívásokkal történik.



5.46. ábra. ábra - A Linux-rendszer komponensei

A Linux megtartotta a UNIX tradicionális szerkezetét. Ez azt jelenti, hogy a kernel egyetlen monolitikus bináris kód. Ennek fő oka a minél nagyobb teljesítmény elérése volt. Mivel a teljes kernelkód és az adatterületek egyetlen címtérben vannak, nincs szükség környezetváltásra *(context switch)*, amikor egy folyamat rendszerfunkciót hív meg, vagy egy megszakítás lép fel. Ezt a címteret nemcsak az ütemező és a virtuális memóriát kezelő kód használja, hanem a teljes kernelkód. Az összes készülékmeghajtó kódja, a fájlrendszer és a hálózatkezelő mind ugyanebben a címtérben találhatók.

A kernel azonban nem sűrít magába mindent, teret ad a modularitásnak is. Ugyanúgy, ahogyan a felhasználói alkalmazások futási időben be tudnak tölteni osztott elérésű könyvtárakat *(run time library)*, a kernel is be tud tölteni kernelmodulokat futási időben. Ezek a modulok önállóan betölthető komponensei a kernelnek.

A kernel a Linux operációs rendszer magját képezi. Mindazt a funkcionalitást biztosítja, ami a folyamatok futtatásához szükséges, továbbá olyan rendszerszolgáltatásokat nyújt, amelyek a hardver erőforrások szabályozott elérését biztosítják. Maga a Linux-kernel több lényeges ponton eltér a megszokott UNIX-kerneltől. A UNIX-kernelhez képest több extra funkció hiányzik belőle. Az általa megvalósított funkciókat pedig nem szükségszerűen ugyanabban a formában látja el, mint a tipikus UNIX-kernel.

A futó alkalmazások számára látható interfészt nem közvetlenül a kernel biztosítja. Az alkalmazások a rendszerkönyvtárakat hívják meg, amelyek szükség szerint meghívják az operációs rendszer szolgáltatásait.

A rendszerkönyvtárak legegyszerűbb funkciója, hogy lehetővé teszik az alkalmazások számára a kernel által biztosított rendszerszolgáltatások elérését. Egy rendszerhívás átkapcsolja a processzort felhasználói módból privilegizált (kernel) módba. Ennek az átváltásnak a részletei hardver architektúránként változnak.

A könyvtárak az alapvető rendszerhívásokon kívül összetett funkciókat is biztosítanak az alkalmazások számára. Például, a C nyelv fájlkezelő függvényei mind a rendszerkönyvtárakban vannak megvalósítva. Ez a fájlok elérésekor hatékonyságnövekedést eredményez. A könyvtárak olyan rutinokat is tartalmaznak, amelyek végrehajtása során nem hívódik meg a kernel. Ilyenek például a rendező algoritmusok, matematikai függvények, valamint a stringkezelő rutinok. Mindazok a funkciók, amelyek ahhoz szükségesek, hogy UNIX- vagy POSIX-alkalmazásokat lehessen futtatni Linux alatt, itt vannak megvalósítva.

A Linux-rendszer magában foglal még számos felhasználói módú programot, mind rendszer-segédprogramokat, mind pedig felhasználói segédprogramokat *(utility)*. A rendszer-segédprogramok közé tartoznak mindazok a programok, amelyek a rendszer inicializálásához szükségesek, mint például a hálózati eszközöket konfiguráló, illetve a kernel modulokat betöltő programok. A folyamatosan futó szerver programok ugyancsak rendszerprogramnak számítanak. Az ilyen programok kezelik a felhasználói beléptetéseket, a bejövő hálózati kapcsolatokat, valamint a nyomtatási sorokat.

Nem mindegyik standard segédprogram lát el rendszeradminisztrációs feladatot. A UNIX felhasználói környezete nagyszámú standard segédprogramot tartalmaz olyan mindennapos feladatok ellátására, mint a könyvtárak kilistázása, fájlok mozgatása és törlése, egy fájl tartalmának megjelenítése stb. Ennél bonyolultabb segédprogramok szövegfeldolgozó funkciót tudnak ellátni, például szöveges adatok rendezése, szövegrészekre történő keresés. Ezek a segédprogramok együttesen olyan standard eszközkészletet alkotnak, amit a felhasználók bármelyik UNIX-rendszeren megtalálnak, és el is várnak. Jóllehet ezek nem végeznek operációs rendszerre háruló tevékenységet, a Linux-rendszernek is alapvetően fontos részét képezik.

A Linux többfelhasználós rendszer. Biztosítja a folyamatok párhuzamos futtatásának lehetőségét, az ehhez szükséges védelmi funkciókkal együtt. Az ütemező időosztásos. Az újonnan létrehozott folyamatok osztozhatnak

a szülővel a környezetük egyes részein. Így lehetőség van többszálú programozásra *(multithreaded programming)*. A folyamatok közötti kommunikációt a UNIX-ban megszokott mechanizmusok támogatják, mint például az üzenetsorok (message queue), szemaforok, osztott elérésű memória, valamint a BSD *socket* interfésze. A különböző hálózati protokollok egyidejűleg érhetőek el a *socket* interfészen keresztül.

A felhasználói felületen a Linux fájlrendszere teljes mértékben a UNIX-szemantikát alkalmazza. Belülről a Linux egy absztrakciós réteget használ a különböző fájlrendszerek kezelésére. A készülékorientált, az NFS, és a virtuális (VFS) fájlrendszert egyaránt támogatja a Linux. A készülékorientált fájlrendszerek a diszket két cache-en keresztül képesek elérni: az ún. *lap cache-en (page cache)*, illetve a *puffer cache-en (buffer cache)* keresztül.

A Linux memóriakezelő rendszere támogatja az osztott memóriaelérést. Alkalmazza a *copy-on-write* módszert annak érdekében, hogy minimalizálja a különböző folyamatok által osztottan használt adatok duplikálását.

A memórialapokat a rendszer a rájuk történő első hivatkozáskor tölti be a fizikai memóriába. Háttértárra mentésük pedig akkor következik be, amikor a rendszer szabad memóriája egy adott szint alá kerül. Lapcserénél az átmenetileg a háttértárra mentendő memórialapot a Linux az ún. *LFU-algoritmussal* (LFU: Least-Frequently Used) választja ki, vagyis a többiekhez képest legritkábban használt lap kerül ki a memóriából.

6. A Windows NT operációs rendszer

Tartalom

6.1. A Windows NT kialakulása
- 6.1.1. Az NT-vel szemben támasztott követelmények
- 6.1.2. A Windows NT, a Windows 95 és a Windows 98 összehasonlítása
- 6.1.3. Az NT felépítésének fő jellemzői
- 6.1.4. Az NT objektumorientált szemlélete

6.2. A Windows NT felépítése
- 6.2.1. HAL
- 6.2.2. Kernel
- 6.2.3. Készülékkezelők (device driverek)
- 6.2.4. Executive
- 6.2.5. Rendszerfolyamatok
- 6.2.6. Szolgáltatások
- 6.2.7. NTDLL.DLL
- 6.2.8. Alrendszerek

6.3. A Windows NT belső mechanizmusai
- 6.3.1. Megszakítás- és kivételkezelés
- 6.3.2. Objektumkezelés
- 6.3.3. Szinkronizáció
- 6.3.4. Lokális eljáráshívás

6.4. Folyamatok kezelése és ütemezése
- 6.4.1. A Windows NT folyamatmodellje
- 6.4.2. Folyamatok kezelése a Windows NT-ben
- 6.4.3. Szálak kezelése az NT-ben
- 6.4.4. Szálak ütemezése

6.5. Memóriakezelés
- 6.5.1. Memória manager felhasználói interfésze
- 6.5.2. Memóriafoglalás
- 6.5.3. Osztott elérésű memória
- 6.5.4. Memóriavédelem
- 6.5.5. Copy-on-Write
- 6.5.6. Memória foglalása
- 6.5.7. A memória mérete
- 6.5.8. Címtranszformáció

6.6. A Windows NT fájlrendszere (NTFS)
- 6.6.1. Elvárások az NTFS-sel szemben
- 6.6.2. Az NTFS további előnyös tulajdonságai
- 6.6.3. Az NTFS által használt adattípusok, adatszerkezetek
- 6.6.4. Fájlok elérése NTFS alatt
- 6.6.5. File ReKord

6.7. Biztonsági alrendszer
- 6.7.1. A biztonsági alrendszer komponensei
- 6.7.2. Az objektumok védelme
- 6.7.3. A biztonsági auditálás
- 6.7.4. A logon

6.1. A Windows NT kialakulása

A Windows NT a Microsoft cég új generációs operációs rendszere. A Windows NT operációs rendszerrel a Microsoft a DOS-, illetve Windows-rendszereket kívánta felváltani. Eredetileg az OS/2-es rendszerek nyomdokán akarták az NT-s fejlesztéseiket folytatni, időközben azonban, alkalmazkodva a piaci igényekhez, a 32 bites Windows-rendszerekhez közelítettek.

A más Windows-rendszerektől struktúrájában is különböző operációs rendszer a *New Technology (NT)* szavak rövidítéséből kapta a nevét. Az elnevezés annyiban jogos, hogy az NT, elődeitől örökölt gyermekbetegségeitől eltekintve, ténylegesen új és korszerű megoldásokat alkalmaz.

Az NT rövid történetének fő állomásait a 6.1. ábra foglalja össze.

Megjelenés ideje	Verziószám (belső név)	A verzió tulajdonságai, illetve újdonságai
1989.	Az NT tervezésének kezdete	
1993. július	NT 3.1.	Kompatibilis a WIN 3.1.-gyel; 16-bites operációs rendszer
1994. szeptember	NT 3.5 (Daytona)	Optimalizálják a rendszer méretét, és teljesítményét; hatékonyságnövelés
1995. május	NT 3.51	Power PC architektúra támogatása
1996. július	NT 4.0 (Shell Update Release – SUR)	Azonos külső (felhasználói interfész) a Windows 95-ös rendszerekkel. Megnövekedett hatékony-ság: például a grafikus alrendszer, képernyő-kezelő funkciók (a Win32-es alrendszer egyes részei) átkerültek felhasználói módból kernel módba.

6.1. ábra. ábra - Az NT történetének fő állomásai

6.1.1. Az NT-vel szemben támasztott követelmények

A Windows NT tervezésekor a leendő rendszerrel szemben támasztott követelményeket két csoportra lehet osztani. Az első csoportban azok a tulajdonságok kerültek követelmények formájában megfogalmazásra, amely tulajdonságokkal a rendszernek feltétlenül rendelkeznie kell a rendszer piaci sikere érdekében. A követelmények második csoportját inkább tervezői célkitűzéseknek lehetne nevezni, mert olyan tervezési elveket és tulajdonságokat tartalmaznak, amik a rendszer átláthatóságát, karbantarthatóságát, valamint későbbi továbbfejleszthetőségét biztosítják.

6.1.1.1. Elvárások

A Windows NT-vel szemben a támasztott követelmények a következők:

- Valós 32 bites, preemptív (kiszorításos, vagyis bármikor megszakítható), újrahívható (reentrant), virtuális memóriakezelést megvalósító operációs rendszer legyen.
- Fusson különböző hardver platformokon.
- Fusson szimmetrikus multiprocesszoros környezetben, és skálázható tulajdonságával tegye lehetővé az adott környezetben rendelkezésre álló erőforrások hatékony kihasználását.
- Fusson elosztott hardver környezetben, és tegye lehetővé elosztott számítási környezet létrehozását.

- A „legtöbb” 16 bites MS-DOS és Windows 3.1-es alkalmazás (applikáció) futtatását tegye lehetővé.
- Teljesítse a POSIX 1003.1 szabványt (legyen POSIX-kompatíbilis).
- Teljesítse az amerikai biztonsági szabványokat.
- Használjon UNICODE-ot a karakterek és stringek ábrázolására.

A felsorolt kritériumok önmagukért beszélnek, csak az utolsóhoz szükséges némi magyarázat.

A UNICODE a karakterek gépi ábrázolásának szabványa. Előnye, hogy minden karaktert 16 biten ábrázol, így szinte minden nyelv ábécéjének karakterkészletét lehetséges azonos kódolást használva ábrázolni. A UNICODE-ot alkalmazva lehetőség nyílik az alkalmazások nyelvterülettől független változatának elkészítésére. Az első olyan, Microsoft által gyártott operációs rendszer, amely binárisan is egységes lesz az egész világon, a következő bejelentett verzió, a Windows 2000 (Windows NT 5.0) lesz.

Az NT 4.0 a belső karakterábrázolásában már UNICODE-ot használ. Mivel az NT-s alkalmazások nagy része még nem használ UNICODE-ot, így a string változókat paraméterként átadó programozói interfészben *(Win32 API)* definiált függvényeknek két változata létezik:

- „széles” UNICODE-os változat ami 16 bites karaktereket kezel
- „keskeny” ANSI változat ami 8 bites karaktereket kezel.

A keskeny változatú függvények megvalósítása egyszerű: először átkódolják a string paramétereket UNICODE-ba (a 8 bites karaktereket 16 bites karakterekre cserélik), majd meghívják az adott függvény UNICODE-os változatát.

6.1.1.2. Tervezői célkitűzések

Az NT tervezői az operációs rendszer minőségének, illetve korszerűségének biztosítása érdekében a fenti követelmények teljesítésén felül a következő célokat tűzték ki:

- Legyen az NT kódja „kiterjeszthető”, vagyis könnyen továbbfejleszthető. (A lehetőségeket figyelembe véve legyen nyílt rendszer.)
- Legyen hordozható a kód, vagyis legyen lehetőség a későbbiekben új hardver platformokra átvinni.
- A rendszer legyen megbízható és robusztus (teherbíró). Ennek három vonatkozását különböztették meg:
    - két applikáció futása ne befolyásolja egymást
    - egy applikáció ne dönthesse össze az operációs rendszert
    - az operációs rendszer belső komponensei „megférjenek” egymás mellett.

- A lehetőségekhez mérten maximálisan legyen kompatíbilis a meglévő rendszerekkel. Itt mind a felhasználói interfészre, mind pedig a programozói interfészre (API-ra) gondoltak. A kompatíbilis rendszereknek két csoportját különböztették meg:
  - a Microsoft korábbi operációs rendszerei: MS-DOS, Windows 3.2
  - nem a Microsoft által készített, azonban széles körben elterjedt rendszerek: UNIX, OS/2, NetWare.
- A rendszer a hardverkörnyezettől függetlenül legyen hatékony, vagyis a teljesítménye legyen maximális bármelyik hardver platformon.

6.1.2. A Windows NT, a Windows 95 és a Windows 98 összehasonlítása

A ’80-as évek végére a hardver eszközök fejlődése révén a Microsoft-termékek célpiacának tekintett személyi számítógépek teljesítményben annyira megerősödtek, hogy már felvehették a versenyt a tradicionálisan UNIX-rendszerek által uralt munkaállomásokkal (workstation).

A Microsoft cég első grafikus interfészt biztosító operációs rendszerei, a 16 bites Windows család tagjai, mind a DOS rendszerre alapulva, annak továbbfejlesztéséből alakultak ki. Ez erősen rányomta bélyegét a Windows-rendszerek felépítésére és működésére.

Az új, nagyteljesítményű PC-k felhasználói olyan igényeket támasztottak az operációs rendszerrel szemben, amit már nem lehetett a Windows-rendszerek továbbfejlesztésével kielégíteni. Ezért döntött a Microsoft egy új, struktúrájában is modern elveket követő operációs rendszer fejlesztése mellett. Ez az operációs rendszer a Windows NT.

Természetesen továbbra sem hagyhatta cserben régi vásárlóit a Microsoft, így folytatta a Windows-rendszerek fejlesztését is. Ennek a fejlesztőmunkának a terméke a Windows 95 és a Windows 98. Ezeknek az operációs rendszereknek a jellemzője, hogy bár szinte teljesen kompatíbilisek a korábbi Windows-rendszerekkel, funkcionalitásban fokozatosan közelítenek az NT felé. Ennek a folyamatnak a befejezése a rövidesen megjelenő Windows 2000, ami már publikáltan NT-alapú rendszer lesz, vagyis a két fejlesztési szál össze fog futni.

A közelítés egyik legfontosabb területe a felhasználói felület: a Windows NT, a Windows 95, és a Windows 98 felhasználói felületei szinte teljesen azonosak.

A másik fontos területe a rendszerek kompatibilitásának a *programozói felület (API: Application Programming Interface)*. A *Win32 API* közös programozói interfésze a fenti operációs rendszereknek. A felsorolt rendszerek API-ja persze nem teljesen azonos. A bennük szereplő hívások döntő többsége közös, azonban vannak olyan függvények, amelyek az egyik rendszer API-jában megtalálhatók, míg a másikban nem. Ilyen szempontból az NT 5.0 közeljövőben várható megjelenése nagy jelentőségű lesz, mivel az NT 5.0 API-ja tartalmazni fogja az összes többi rendszer API-jának hívásait. Bővebb információt az API-król a *http://msdn.microsoft.com/default.asp* Internet-címen olvashatunk.

Hasonlóan az NT 5.0-ban kerül alkalmazásra a Windows 98-ban kifejlesztett új *készülékkezelő (device driver)* modell, az ún. *WDM*.

A gyártó közös volta, illetve a fejlesztés párhuzamossága miatt nem meglepő, hogy a felsorolt rendszerek kódja *részben* közös.

A 6.2. ábrán látható táblázatban összehasonlítjuk az NT, illetve a Windows 95, és a Windows 98 tulajdonságait. (A szimbólumok jelentése: *: teljesen megvalósított, +: részben megvalósított, –: nem megvalósított.)

Látható, hogy az NT 5.0 az összes, az ábrában felsorolt kedvező tulajdonsággal rendelkezni fog, egyet kivéve: Az NT nem lesz képes az összes Windows 3.1-es és DOS alkalmazást futtatni. Ezt a tulajdonságot nem is szándékoztak megvalósítani a fejlesztők, hiszen az NT struktúrájában is különbözik a régi rendszerektől, ami lehetetlenné teszi számos alkalmazás futtatását.

Tulajdonságok	NT	95/98
Multiprocesszoros környezet támogatása	*	–
Változó HW platform	*	+ (Korlátozottan)
NTFS (biztonsági hozzáférés)	*	–
32 bites kód	*	+
Újrahívható (több példányban, párhuzamosan futtatható) kód	*	–
16-bites (Windows 3.1-es) alkalmazások futtatása	* (Biztonságos)	+ (Összeakadhatnak)
Megosztott memória elérésének korlátozása	Csak az ér el aki megnyitja	Mindenki elérheti
A rendszer adatstruktúráit nem érhetik el az applikációk	*	Sok adatszerkezet elérhető
User módban nem írhatóak az operációs rendszer adatai	*	–
Windows 2000-ben (NT 5.0-ban) várható újdonságok		
Plug and Play képesség	–	+
Power management	–	+
Infrared támogatás	–	+
FAT32 könyvtárszerkezet támogatás	–	+
Minden Windows 3.1-es és DOS alkalmazás futtatása	–(Nem is lesz)	*

6.2. ábra. ábra - Operációs rendszerek összehasonlítása

A Microsoft operációs rendszerekről szólva a felsorolást a teljesség kedvéért még ki kell egészíteni a Windows CE operációs rendszerrel. A Windows CE a korábban említett rendszerekhez nagyon hasonló felhasználói felülettel, és részben közös programozói felülettel rendelkezik. A legfontosabb különbség, hogy a Windows CE valósidejű *(on-line)* rendszerekhez készült, vagyis garantálja a limitált válaszidőt. Ez természetesen a többi rendszertől lényegesen eltérő belső felépítést eredményez. A Windows CE rendszerek felépítéséről, működéséről a továbbiakban csak utalások szintjén lesz szó.

6.1.3. Az NT felépítésének fő jellemzői

A Windows NT felépítése rétegszerkezetű, számos alrendszerének működése a kliens-szerver architektúrára alapul.

Az NT jól definiált rétegekre bomlik, mely rétegek szigorúan interfészeken keresztül érintkeznek. A szolgáltatások java részénél kliens-szerver architektúrát találunk.

Nem tartották viszont szigorúan be az NT fejlesztői azt az elvet, hogy amennyiben mód van rá, a szolgáltató folyamatok felhasználói *(user)* módban fussanak és a rendszer a *mikrokernel*en keresztül érje el a hardvert.

Ez azért van így, mert bizonyos szolgáltatásokat nem hatékony felhasználói módban megvalósítani. Így kernel módba azok a szolgáltatások kerültek, amelyek intenzíven használják a hardvert és futásuk gyorsaságára az egész rendszer teljesítménye érzékeny, vagyis a *user* módban történő megvalósításuk a rendszert nagyon lelassítaná a gyakori környezetváltás miatt. (Ilyen például a *cache manager*, a memóriakezelő, az objektumkezelő és biztonsági alrendszer, vagy a WIN32-es alrendszer grafikus szolgáltatásokat nyújtó részei stb.)

6.1.4. Az NT objektumorientált szemlélete

A Windows NT fejlesztésénél fontos szempont volt az objektumorientált szemlélet használata. A fejlesztők igyekeztek a rendszert jól meghatározható funkcionalitású objektumokból felépíteni, mely objektumok egymással az előre definiált interfészükön keresztül érintkeznek. A határozott törekvés ellenére, a hardverközeli programozás szüksége miatt azonban, csak az alábbi két objektumorientált tulajdonságot valósítja meg:

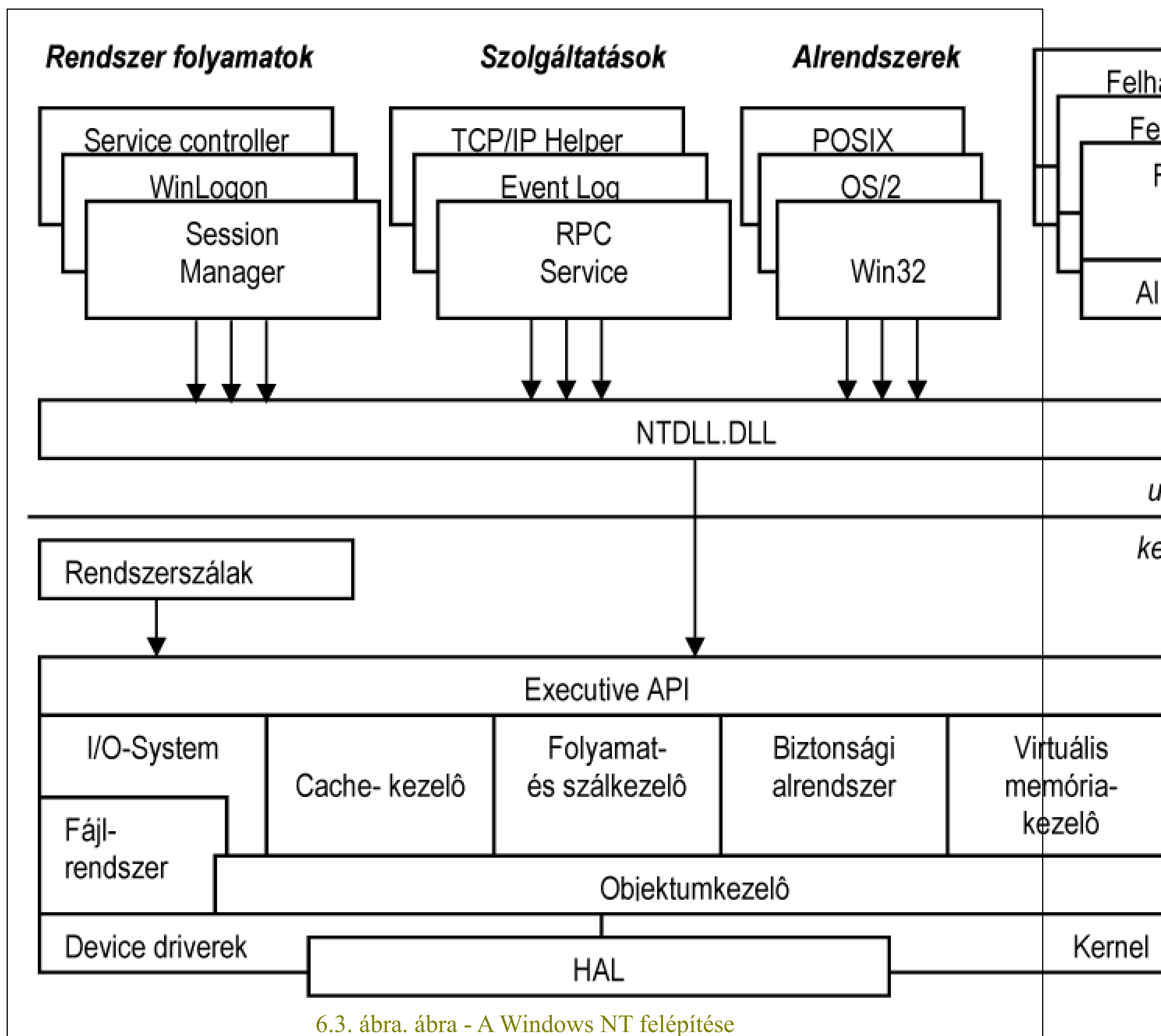
- adatrejtés, az operációs rendszer objektumai csak saját adataikat érhetik el.
- interfész-használat, az objektumok formális interfészt használnak egymás elérésére.

Ugyanakkor azonban a Windows NT nem valósítja meg a következő objektumorientált tulajdonságokat:

- Polimorfizmus. Azonos néven különböző objektumokat érhetünk el. Attól függ, hogy milyen objektumra hivatkozunk az adott azonosítóval, hogy mi az azonosító használatának környezete. Például polimorfizmus, amikor azonos névvel, de különböző paraméterlistával definiálunk függvényeket egy objektumon belül. Ekkor az aktuálisan használt paraméterek típusától függ, hogy melyik funkciót érjünk el.
- Öröklődés. Az objektumoknak hierarchikus származási rendszere létezik. A származtatott objektum örökli, és így használhatja a szülő megfelelően definiált adatelemeit illetve függvényeit.
- Dinamikus adattípus kötés. Definiálhatóak adattípussal paraméterezett objektumok.

6.2. A Windows NT felépítése

A Windows NT felépítését a 6.3. ábrán mutatjuk be. A téglalapok az egyes komponenseket szemléltetik, a köztük lévő kapcsolatot a nyilak mutatják. Jól láthatók az egymásra épülő rétegek, illetve az őket összekötő interfészek.



6.3. ábra. ábra - A Windows NT felépítése

A következő részekben részletesen bemutatjuk az NT komponenseit, ismertetve azok funkcióit.

6.2.1. HAL

A HAL *(Hardware Abstraction Layer)* az aktuálisan használt hardvert teljesen elfedő, legalacsonyabb szoftverréteg. Az operációs rendszer HAL fölötti rétegei (vagyis a device driverek, illetve a kernel) csak ezen keresztül érheti el a hardvert. Minden processzorhoz, amelyen az NT-t implementálták, készül külön HAL-réteg, vagyis a HAL a használt processzor típusától függ. Egy adott rendszerben az aktuálisan használt HAL-réteget a rendszer telepítésekor választja ki.

A HAL igyekszik az általa rejtett processzortól független szolgáltatásokat nyújtani a többi réteg számára. Azonos architektúrájú processzorok esetén ez teljesül is, vagyis például az x86 típusú processzorokon futó NT-rendszerek között csak a HAL-ban van különbség. Úgy is tekinthetjük, hogy a HAL az aktuális hardverre támaszkodva egy virtuális gépet valósít meg, amelynek funkcionalitása minden rendszerben azonos.

6.2.2. Kernel

A kernel a rendszer állandóan memóriában levő kernel (védett) módban futó része. A kernel a HAL-on kívül az egyetlen hardverfüggő része az NT-nek. Azonban míg a HAL a processzor típusától függ, addig a kernel csak a processzor architektúrájától. Ez azt jelenti, hogy azonos felépítésű processzorok esetén a kernel is azonos, csak eltérő felépítésű processzorok esetén lehet különbség. Például két x86-os processzoralapú rendszerben a kernelréteg is azonos attól függetlenül, hogy konkrétan milyen processzort használunk. Egy RISC-alapú rendszerben azonban a kernelréteg is különbözni fog. Az adott konfigurációban használt HAL-t az NT installálásakor választja ki. Az installáló CD-n így számos HAL-t találhatunk. Érdemes megnézni az NT CD \i386-os alkönyvtárát.

Fontos látni, hogy a kernel és a rá épülő további komponensek csatlakozási felülete már hardverfüggetlen. Ebből következik, hogy a rendszer további komponensei is hardverfüggetlenek, vagyis hordozhatók.

Az NT hordozhatósága tehát úgy valósul meg, hogy a hardvert minden processzoron hasonló interfésszel elérhető szoftverrétegek fedik el. Ez a két hardverréteg a HAL és a kernel. A kernelben kerültek megvalósításra a processzorok architektúrájától függő funkciók (például szálak környezetváltása), míg a HAL az azonos architektúrájú processzorok adott típusától függő funkcióit tartalmazza.

A hordozhatóságról szólva meg kell említeni a hordozhatóság másik eszközét, nevezetesen azt, hogy az NT – a UNIX-rendszerekhez hasonlóan – hordozható programnyelven íródott. A kód túlnyomó része C-ben készült. A grafikus alrendszernek és a felhasználói interfésznek egy kisebb részét C++-ban fejlesztették. Assembly nyelven csak a hardver közvetlen kezelését végző (például IT-kezelés), vagy a rendszer teljesítményét nagymértékben befolyásoló (például környezetváltás) részek íródtak.

A kernel feladata, a rendszer további részeinek a hardvertől történő teljes elhatárolásán túl, hogy az operációs rendszer többi komponense számára egyszerű, jól definiált működésű *alapmodulokat* (ún. *primitíveket*) biztosítson.

A primitívek a következő funkciókat valósítják meg:

- thread (szál) ütemezés,
- trap-kezelés és kivételkezelés,
- IT-kezelés,
- multiprocesszor-ütemezés,

- kernel objektumok kezelése.

A kernel objektumok egyszerűbbek, mint a többi réteg objektumai, mert a gyors kezelés érdekében a kernel nem végez ellenőrzést az egyes objektumok elérésekor, megbízik abban, hogy a rendszer tagjai helyesen használják az objektumokat.

6.2.3. Készülékkezelők (device driverek)

A *device driver*ek a B/K-alrendszer és a hardver között teremtenek kapcsolatot. Mint már említettük, nem közvetlenül érik el a hardvert, hanem a HAL- rétegen keresztül. Ez például lehetővé teszi, hogy a *device driver*ek forráskódban hordozhatók különböző hardverarchitektúrák között, és akár binárisan is hordozhatók azonos architektúrájú, de különböző típusú processzorok esetén.

A *device driver*ek négy típusát különböztetjük meg:

- Hardver *driverek*, melyek a hardver egységek elérését biztosítják, az eszközök közvetlen be- és kimeneteit kezelik.
- Fájlrendszer *driverek*, melyek a fájlrendszerek elérését biztosító kéréseket fogadnak el és B/K-kérésekké transzformálják azokat.
- Szűrő típusú *device driverek*, melyek az ún. réteg szerkezetű *device driver* struktúra lehetőségeit kihasználva speciális többletfunkciókat valósítanak meg. A szűrő típusú *device driverek* általában a magas szintű (például fájlrendszer *driverek*) és a B/K-kéréseket kezelő alacsony szintű *driverek* közé ékelődnek.
- Hálózat elérését biztosító *device driverek*, melyek a hálózati kéréseket szolgálják ki, illetve továbbítják azokat.

Az NT ún. *rétegszerkezetű device driver struktúrát* használ, ami lehetővé teszi a funkciók szétválasztását, illetve rugalmas *driver*szerkezet kialakítását. A réteg szerkezetű struktúrának köszönhetően lehetőség van a hagyományosakon túl olyan *driverek* használatára is, amelyek lehetőséget adnak többletfunkciók megvalósítására, mint például a redundáns adattárolás merevlemezek tükrözésével vagy több lemezen tárolt fájlrendszer kezelése. Ezekről a lehetőségekről a későbbiekben részletesen is lesz szó.

6.2.4. Executive

Az *executive* réteg az operációs rendszerek magas szintű szolgáltatásait nyújtó alrendszereket megvalósító rétege az NT-nek. Önálló részei a következők:

- folyamat- és szálkezelő,
- virtuálismemória-kezelő,
- biztonsági alrendszer (monitor),

- cache kezelő,
- B/K-rendszer kezelő.

Az *executive* réteg tartalmazza az NTDLL.DLL által definiált függvények hívásainak megvalósítását, valamint biztosítja az operációs rendszer belső objektumai közötti kommunikáció lehetőségét.

Az *executive* réteg legfontosabb szolgáltató funkciója az LPC (*Local Procedure Call*, lokális eljáráshívás) szolgáltatás megvalósítása. Az LPC az NT IPC (*Inter Process Communication*, folyamatok közötti kommunikáció) eszköze. Ennek segítségével egy felhasználói objektum egy másik felhasználói objektum függvényét hívhatja meg. Így érhetik el például az egyes alkalmazások a hozzájuk tartozó alrendszer szolgáltatásait. Ezen felül az *executive* réteg tartalmaz *run-time library* függvényeket és különböző támogató funkciókat megvalósító függvényeket a rendszerfolyamatok és szolgáltatások számára.

6.2.5. Rendszerfolyamatok

Rendszerfolyamatok közé az operációs rendszer azon funkcióit megvalósító önálló folyamatok tartoznak, amelyek független felhasználói folyamatként kerültek megvalósításra. Fontos kiemelni, hogy bár *user* módban futó folyamatokról van szó, a rendszerfolyamatok alapvető részei az operációs rendszernek, futásuk nélkül az NT nem képes működni.

Fontos rendszerfolyamatok:

- SMSS (Session Manager). A rendszer indításakor létrehozott folyamat, ami az alkalmazások elindításáért felelős. Az SMSS indítja el az egyes alrendszereket, ha futásukra szükség van, és még nem futnak; biztosítja a kapcsolatot a debugger és az általa futtatott applikáció között; valamint biztosítja a környezeti változók definiálásának és elérésének lehetőségét.
- Logon. A felhasználók ki- és beléptetését intéző folyamat. Minden ún. SAS *(Secure Attention Sequence)* billentyűkombináció (alapesetben: CTRL-ALT-DEL) aktivizálja. Beléptetéskor a felhasználói azonosító *(username)* és jelszó *(password)* kombinációt az önálló folyamatként futó *Local Security Authentication Server*hez (LSASS) továbbítva ellenőrzi. Ha az azonosítás sikeres, elindítja a USERINIT.EXE programot, ami beállítja a felhasználó által definiált környezetet, és elindítja az általa kért *shell*t (alapesetben: EXPLORER.EXE).
- Service Controller. A szolgáltatások indításáért és leállításáért felelős folyamat.

6.2.6. Szolgáltatások

Az NT-ben szolgáltatásnak hívják azokat a kliens-szerver modellben szerverként működő szolgáltató folyamatokat, amelyek a kliens programok (felhasználói vagy akár rendszer programok) számára az operációs rendszer lehetőségeire építve többletszolgáltatásokat nyújtanak. Létezik például RPC *(Remote Procedure Call)* szolgáltató, különböző protokollokat megvalósító hálózati kapcsolatot biztosító szolgáltatók, vagy az NT-

specifikus eseménynaplózó *(event logger)* szolgáltató. (Megjegyzendő, hogy egy-egy szolgáltatást gyakran nem egyetlen folyamat valósít meg.)

A szolgáltatások a rendszerfolyamatokhoz hasonlóan felhasználói módban futó folyamatok. Lényeges különbség azonban, hogy míg a rendszerfolyamatok szükséges részei a rendszernek, addig a szolgáltatásokat megvalósító folyamatok futása nélkül is képes működni az NT. A szolgáltatások a *Service Manager* segítségével elindíthatók és leállíthatók a rendszer működése közben.

A szolgáltatásokat megvalósító programok egyszerű Win32-es alkalmazások, azzal a különbséggel, hogy együttműködnek a *Service Conroller* (SERVICES.EXE) folyamattal. Az regisztrálja őket, és annak segítségével lehet őket elindítani, leállítani, szüneteltetni stb. A rendszerben elérhető szolgáltatásokat és azok aktuális státusát a felhasználói felületről is megnézhetjük és változtathatjuk a *Control Panel*en a *Services* ikonra kattintva. (Egy szolgáltatásnak így három elnevezése is lehetséges: az első az a név, ahogy a szolgáltatás a *Control Panel/Services* alpontján keresztül elérhető, a másik az a név, amin az a *Registry*-ben szerepel, a harmadik pedig az a név, amely a szolgáltatást megvalósító futtatható program neve.)

6.2.7. NTDLL.DLL

Az NTDLL.DLL az a dinamikusan kapcsolódó könyvtár *(Dinamically Linked Library – DLL)*, amin keresztül a felhasználói módú folyamatok elérhetik az NT-t. Mivel az egyes objektumok közötti kapcsolattartás az LPC mechanizmuson keresztül történik, így minden felhasználói objektum az NTDLL.DLL-en keresztül éri el a környezetét.

Az NTDLL által megvalósított működés egyszerű. Ha egy hívás érkezik, ellenőrzi a hívás paramétereit, és megvalósítja a *user–kernel* módváltást, majd meghívja az NT kért funkciót megvalósító függvényét.

6.2.8. Alrendszerek

Mint már említettük, az NT alkalmas különböző típusú applikációk futtatására. Ezt az alrendszerek segítségével valósítja meg. Három alrendszere van: Win32, POSIX, és az OS/2. A Win32 alrendszer kitüntetett abban a tekintetben, hogy a Win32 alrendszer nélkül az NT nem tud futni. A másik két alrendszer opcionális, csak abban az esetben kezdenek futni, amikor az adott alrendszerhez tartozó alkalmazást akar egy felhasználó elindítani.

Az alrendszerek elsődleges feladata, hogy a hozzájuk tartozó alkalmazások futásához szükséges szolgáltatásokat nyújtsák. Minden alrendszer a hozzá tartozó alkalmazásokat kontrollálja.

Minden alrendszerhez tartozik egy ún. alrendszer DLL, amin keresztül az alrendszerhez tartozó alkalmazás az NT szolgáltatásait elérheti. Ez tehát a programozói interfész *(API: Application Programming Interface)*, ami az egyes alrendszerekhez tartozó applikációk számára elérhető. Ezt azért fontos kiemelni, mert ez az a publikált, jól definiált felület, amit minden program használhat. Az összes többi interfész (például NTDLL.DLL) *nem publikus* interfész.

Az egyes alrendszerekhez tartozó API-k lényegesen különböznek egymástól. A legszélesebb lehetőségeket a Win32 API biztosítja (például ablakozás, szálak kezelése stb.). Fontos tudni, hogy egy applikáció csak egy alrendszerhez tartozhat, nem keveredhetnek egymással egy applikáción belül különböző alrendszerhez tartozó hívások.

6.2.8.1. POSIX alrendszer

A POSIX alrendszer szigorúan a POSIX 1003.1-es szabványban rögzített tulajdonságokat valósítja meg. Így van lehetőség új folyamatok létrehozására *(fork)*, fájlok több néven történő elérésére (*hard linkek* definiálására), folyamatok közötti kommunikációra (IPC), folyamatok kezelésére, valamint karakteres B/K kezelésére. Ezzel szemben nincs lehetőség szálak *(thread)* létrehozására, ablakkezelésre, távoli eljáráshívásra (RPC elérésére), *socketek* használatára. (A UNIX-alkalmazások hordozása érdekében készítettek már a szabványos POSIX-alrendszernél szélesebb lehetőségeket megvalósító POSIX-alrendszereket, illetve POSIX API-t, valamint vannak olyan programkönyvtárak, amelyek lehetővé teszik, hogy egy POSIX- (UNIX-) alkalmazást Win32-es futtatható állománnyá fordítsunk.)

6.2.8.2. Win32 alrendszer

A Win32 alrendszer futtatja nemcsak a 32 bites alkalmazásokat (vagyis azokat, amelyek Win32 API-t használnak), hanem a 16 bites és DOS-alkalmazásokat is.

A Win32 alrendszer nem csak abban különbözik a többitől, hogy nélküle nem futhat az NT, hanem abban is, hogy az alrendszer egy része kernel módban fut. (WIN32K.SYS) Ez a rész valósítja meg a grafikus képernyő-kezelési funkciókat. (A USER32.DLL, GDI.DLL, KERNEL32.DLL, ADVAPI.DLL könyvtárakban definiált funkciókat.) Ez a rendszer hatékonysága miatt került ide, mert így módváltás nélkül lehetséges az *executive*, illetve kernel szolgáltatások (függvények) elérése.

A Win32 API-hívások háromfélék lehetnek a megvalósításuk helye szerint:

- Az alrendszer DLL-ben van megvalósítva. Ezek egyszerű funkcionalitású függvények, a végrehajtásukhoz nincs szükség a rendszer más részeinek elérésére.

Az alrendszerben vannak megvalósítva. A hívást ebben az esetben az alrendszer DLL továbbítja az NTDLL.DLL felé, ami az executive réteg LPC szolgáltatását igénybe véve, eljut a Win32 alrendszerhez, ami a kérést teljesíti.

Az NT más, kernel módban futó rétege valósítja meg a hívást. A hívás ebben az esetben is az *executive* réteghez kerül, ami továbbítja a megfelelő kernel réteg felé.

A WIN32 API-ban számos grafikus funkció van definiálva. A grafikus eszközök és a nyomtató szabványos felületen történő kezelését a GDI *(Graphical Device Interface)*, illetve a hozzá tartozó GDI32.DLL (WIN32 API része) teszi lehetővé. Az ebben definiált funkciók a WIN32 alrendszer kernel módú részében (WIN32K.SYS)

vannak megvalósítva. Ezek a rutinok intézik az eszközökhöz tartozó *device driverek* meghívását. Egy-egy GDI32.DLL-ben definiált grafikus funkcióhoz az adott eszköztől függően akár több *device driver* is tartozhat.

6.3. A Windows NT belső mechanizmusai

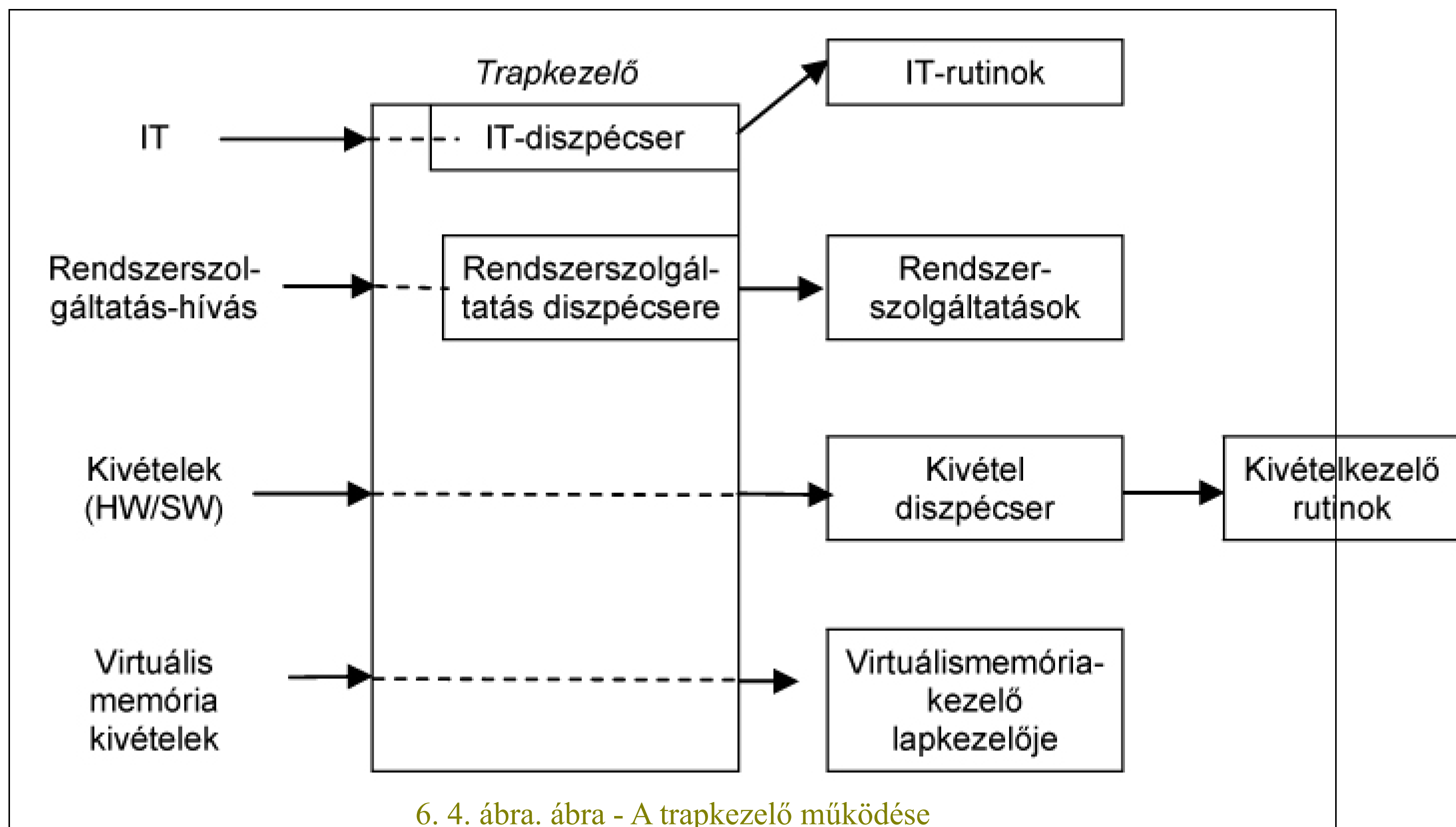
A Windows NT számos olyan mechanizmussal rendelkezik, amelyeket a kernel módú komponensek, mint az *executive*, valamint a *device driverek* használnak. Ebben a részben a következő rendszermechanizmusokat mutatjuk be:

- megszakítás- és kivételkezelés,
- *executive* objektumkezelés,
- szinkronizáció,
- lokális eljárás hívás *(Local Procedure Calls -LPC's)*.

6.3.1. Megszakítás- és kivételkezelés

A megszakítások (IT: interrupt) és a kivételek (exception) olyan események az operációs rendszerben, amelyek a CPU-t eltérítik az utasítások normál végrehajtási sorrendjétől. A megszakításokat és a kivételeket detektálhatják a rendszer hardver, és szoftver komponensei egyaránt.

Az NT-hez kapcsolódó szóhasználatban a *trap (csapda)* fogalom a processzor azon mechanizmusát jelöli, amelynek segítségével az egy végrehajtás alatt levő szálat állít meg egy megszakítás, vagy egy kivétel hatására. A *trap* mechanizmus a processzort felhasználói módról kernel módra kapcsolja át, és a vezérlést az operációs rendszer egy fix helyére, a *trapkezelő* rutinra adja át. A *trapkezelő* eldönti, melyik operációs rendszer komponens képes a megszakítást, vagy kivételt kiváltó eseményt lekezelni. A teljes képhez hozzátartozik, hogy a *trapkezelő* aktivizálódik a rendszer hívások végrehajtásakor is, amikor a megszakítások és kivételek kezeléséhez hasonlóan a rendszernek kernel módba kell váltania. A *trapkezelő* és az általa megvalósított funkciók összefoglalását a 6.4. ábrán láthatjuk.



6. 4. ábra. ábra - A trapkezelő működése

Bár általában igaz az, hogy a megszakítást hardveregység, a kivételt pedig szoftverkomponens idézi elő, ez nem feltétlenül igaz minden esetben. Hardver eredetű kivételre példa a buszhiba által kiváltott kivétel melynek forrása hardver meghibásodás. Szoftver által generált megszakításra is találunk bőséges példát az NT-ben. Az NT kernel szoftvermegszakításokkal kezeli néhány esetben a futó szál váltását, az időzített óra lejárását, illetve néhány szinkron B/K-folyamatot.

6.3.1.1. A megszakítások típusa és prioritása

A különböző processzorok különböző számú és típusú megszakítást képesek felismerni. A megszakításokhoz prioritási *szinteket* (ún. *IRQL: Interrupt Request Level*) rendel az operációs rendszer. A párhuzamosan érkező megszakítások kiszolgálása a prioritási szinteknek megfelelően történik. A magasabb prioritású IT megszakítja az alacsonyabb prioritású IT kiszolgálását. A 6.5. ábra a *DEC Alpha* processzorhoz tartozó megszakítási szintek tábláját (IRQL-táblát) mutatja be, a 6.6. ábra pedig az Intel x86-os processzorokhoz tartozót.

7	Felső szint
6	Processzor – IT
5	Órajel
4	B/K-eszköz/Felső szint
3	B/K-eszköz
2	Késleltetett eljáráshívás
1	Aszinkron eljáráshívás
0	Alsó szint

6.5. ábra. ábra - IRQL tábla az Alpha processzornál

31	Felső szint
30	Tápfeszültség-kimaradás
29	Processzor – IT
28	Órajel
	B/K-eszköz(n)
	. . .
	B/K-eszköz(2)
	B/K-eszköz(1)
2	Késleltetett eljáráshívás
1	Aszinkron eljáráshívás
0	Alsó szint

6.6. ábra. ábra - IRQL tábla az Intel x86 processzoroknál

A megszakítások felső szintjei a hardver-megszakításokra vannak fenntartva. Az 1-es és 2-es szintek mindkét táblázatban a szoftver-megszakítások. A késleltetett eljáráshívást eredményező IT-re egyes szálak biztonságos végrehajtása érdekében van szükség. Az aszinkron eljáráshívások a felhasználói programok, illetve

rendszerprogramok számára biztosítanak lehetőséget, hogy egy másik felhasználói szálon hajthassanak végre (hívjanak meg) egy eljárást.

A legalsó, nullás szint a normál szálvégrehajtáshoz tartozik. Ezen a szinten minden IT engedélyezett.

6.3.1.2. Kivételkezelés

A bármikor fellépő megszakításokkal ellentétben a kivételek olyan feltételek, amelyek egy futásban levő program végrehajtása következtében jönnek létre. A kivételeket egy speciális kernelmodul, a kivétel-felügyelő *(exception dispatcher)* dolgozza fel. A kivétel-felügyelő elsődleges feladata az aktuális kivétel azonosítása: memória-túlcímzés, nullával való osztás, egész-túlcsordulás, lebegőpontos kivételek, a nyomkövető töréspontjai stb. Ezek mindegyike architektúrafüggetlen eset.

Amikor egy kivétel lép fel, a kernelben egy eseménysláncolat indul el. A CPU a vezérlést a kernel *trapkezelőjének* adja át. Ez egy ún. *trapkeretet* állít elő, amely mindazt az információt tartalmazza, ami a kivételkezelés befejezése utáni visszatéréshez (újrakezdéshez) szükséges az operációs rendszer számára. A *trapkezelő* ezen kívül még előállít egy rekordot, ami a kivétel eredetét rögzíti, és más szükséges információt tartalmaz.

A nyomkövető programok töréspontjai rendszeresen kivételforrások. Emiatt a kivétel-felügyelő első teendője annak ellenőrzése, hogy a kivétel egy nyomkövető folyamattól származik-e.

6.3.2. Objektumkezelés

A Windows NT-ben levő *executive* egyik komponense az *objektumkezelő (object manager)*, amely az objektumok előállítására, törlésére, védelmére, és kezelésére szolgál. Az objektumkezelő összefogja az erőforrás-kezelő műveleteket, hogy azok ne legyenek szétszórva az operációs rendszer egyes komponenseibe.

Az objektumkezelőt a következő célok elérésére tervezték:

- Egységes mechanizmust biztosítson a rendszer erőforrásainak elérésére, illetve használatára.
- Az objektumok védelmét megvalósító kód egy helyre legyen összegyűjtve, a C2-es szintű biztonság elérése érdekében.
- Lehetőséget biztosítson a folyamatok objektumhasználatának figyelésére, amivel korlátozható a rendszer erőforrásainak használata.
- Olyan objektum-elnevezési módszert valósítson meg, ami lehetővé teszi a létező objektumcsoportok egyszerű megkülönböztetését.
- Elégítse ki a különböző alrendszerek által támasztott követelményeket: például egy folyamat legyen képes erőforrásokat örökölni szülőjétől (erre szükség van a Win32-es és a POSIX alrendszernél), vagy lehessen kis- és nagybetűt megkülönböztető *(case sensitive)* neveket használni (ami a POSIX alrendszer követelménye).

- Egységes szabályok szerint kezelje az objektumok „életben tartását”, vagyis egy objektum mindaddig álljon rendelkezésre, amíg az összes folyamat be nem fejezte használatát.

Belülről a Windows NT kétfajta objektummal rendelkezik: *executive objektummal és kernel objektummal*. Az *executive objektumokat* az *executive* réteg különböző komponensei hozzák létre, mint például a folyamatkezelő, memóriakezelő, B/K-alrendszer stb.

A *kernel objektumok* sokkal egyszerűbbek, mint az *executive objektumok*. Olyan alapvető funkciókat valósítanak meg, mint például a szinkronizáció, amely funkciókra az *executive objektumok* is épülnek. Az *executive objektumok* általában több *kernel objektumból* épülnek fel.

Amikor egy folyamat létrehoz és megnyit egy objektumot név szerint, akkor egy ún. *handle*-t kap vissza (a szó jelentése *fogantyú*, *nyél*, de a magyar elnevezést nem tartjuk megfelelőnek). A *handle* az objektumhoz való hozzáférési információt tartalmazza. Valójában nem más, mint egy összetett elérési cím, egy összetett pointer, aminek használatával jóval gyorsabban lehet elérni egy objektumot, mint a név szerinti kereséssel.

6.3.3. Szinkronizáció

A *kölcsönös kizárás* megvalósítása alapvető fontosságú az operációs rendszerekben elsősorban a nem osztott elérésű erőforrások használatának szabályozására (például akár a rendszer összetett adatszerkezetei esetén).

A kölcsönös kizárás különösen fontos a Windows NT esetében, amely támogatja a szorosan csatolt szimmetrikus-multiprocesszoros hardver architektúrát. Az egyes processzorok ugyanazt a rendszerkódot hajtják végre egy-idejűleg, bizonyos, közös memóriában tárolt adatstruktúrák megosztásával.

A Windows NT-ben a kernel feladata a több szál által is használt adatstruktúrák használatának összehangolása. A kernel ennek a feladatnak a megoldására a kölcsönös kizárást megvalósító alapmodulokat (ún. *primitíveket*) tartalmaz. Ezeket a primitíveket a kernelen kívül az *executive* réteg is használja a közös adatstruktúrák elérésének időbeli összehangolása, *szinkronizálása* céljából.

6.3.3.1. Kernel szinkronizáció

A kernel kód végrehajtásának különböző fázisaiban a kernelnek garantálnia kell, hogy egy kritikus szakaszt kizárólag csak egyetlen processzor hajt végre. A kernelben a kritikus szakaszokban lévő kódrészek módosítják a globális adatstruktúrákat (például a késleltetett eljáráshívások várakozási sora vagy a *kernel dispatcher* adatbázisa).

A szinkronizáció során a legnagyobb gondot a megszakítások jelentik. Előfordulhat például, hogy a kernel egy globális adatstruktúrát módosít, amikor egy olyan IT lép fel, aminek a kezelő rutinja éppen ugyanazt a struktúrát módosítaná. Egy processzor esetén a Windows NT a következő mechanizmust alkalmazza.

A kernel a globális erőforrások használata előtt ideiglenesen letiltja azokat az IT-ket, amelyeknek az IT-kezelő programja ugyanazt az erőforrást használja. Ehhez fel kell emelnie a processzor futási szintjét a szóban forgó megszakítások közül a legmagasabb szintjére.

Ez a megoldás azonban nem elégséges több processzor esetén. Ugyanis a futási szint megemelése egy processzornál még nem akadályozza meg egy megszakítás fellépését egy másik processzoron. A kernelnek azonban ilyenkor is garantálnia kell a kölcsönös kizárást az erőforrások használatakor. A kölcsönös kizárás megvalósítására a Windows NT ezért az egyedi *lezárás (locking)* módszerét alkalmazza. A kizárással használandó erőforrásokat – így a globális adatstruktúrákat – használatuk előtt az erőforrást használni kívánó szálnak le kell zárnia, meg kell szereznie az erőforráshoz tartozó ún. *spinlockot*. Amíg ezt nem tudja megtenni, várakoznia kell. A rendszer az egyes hozzáférési igényeket sorba állítja, és a lezárás megszűnésekor mindig csak egy várakozót enged tovább.

6.3.3.2. Executive szinkronizáció

A multiprocesszoros környezetben az executive réteg komponenseinek is szinkronizálnia kell a globális adatstruktúrákhoz való hozzáférést. A kernel funkciók hívásakor az executive is használhatja a lezárás módszerét. Ez azonban csak részben oldja meg az executive réteg szinkronizációs igényeit. Mivel egy lezárás feloldására történő várakozás ténylegesen is várakoztatja az érintett CPU-t, a megoldás csak az alábbi korlátozásokkal alkalmazható:

- a védett erőforrásnak gyorsan elérhetőnek kell lennie anélkül, hogy más kóddal komplikált kapcsolatban állna,
- a kritikus szakaszok nem lapozhatók ki a memóriából, nem hívhatnak külső eljárásokat (rendszerszolgáltatást sem), továbbá nem generálhatnak megszakítást vagy kivételt.

A fenti szigorú korlátozások sajnos nem tarthatók be minden esetben. Ráadásul az executive réteg különböző működéseinek szinkronizációja nemcsak a kölcsönös kizárást, hanem más típusú szinkronizációt (előidejűség, egyidejűség) is igényelnek. Ezen felül szinkronizációs lehetőséget kell biztosítani a felhasználói alkalmazások számára is.

Mindezeket a feladatokat speciális szinkronizációs objektumok kezelésével valósítja meg a Windows NT. Ezeket az objektumokat együttesen diszpécser objektumoknak (dispatcher object) nevezzük. A rendszer biztosít megfelelő függvényhívásokat (WaitForSingleObject, WaitForMultipleObject), melyek meghívásakor az ütemező az adott szálat várakozó (waiting) állapotba viszi és berakja a megnevezett objektum várakozási sorába. Az objektum felszabadulásakor a rendszer az objektum típusától függően egy vagy több várakozó folyamatot futásra kész (ready) állapotba tesz.

A dispatcher objectek között létezik például a kölcsönos kizárást megvalósító ún. mutex objektum, mely esetén mindig csak egy várakozó szál lesz futásra kész, de van olyan, mint például a semaphore, ahol – a semaphore inicializálási értékétől függően – akár több is lehet. Sőt létezik olyan dispatcher object is – a timer –, ami szabaddá válása (lejárása) esetén minden, az objektumra várakozó szál átkerül a ready állapotba. Így az executive rétegben lehetőség van bonyolult szinkronizációs feladatok megoldására is.

A dispatcher objectek és azokat kezelő hívások egy része a Win32 API-n keresztül is elérhető, lehetővé téve a szinkronizációt a felhasználói alkalmazások számára is.

6.3.4. Lokális eljáráshívás

A *lokális eljáráshívás (LPC: Local Procedure Call)* a folyamatok közötti kommunikációra ad lehetőséget, nagysebességű üzenettovábbítás formájában. Ez a belső mechanizmus csak a Windows NT operációs rendszer komponensei számára áll rendelkezésre, a Win32 API-n keresztül nem érhető el. Két példa az LPC-k használatára:

- Távoli eljáráshívások (Remote Procedure Call) LPC-t használnak akkor, ha a kommunikáló folyamatok ugyanazon a rendszeren belül vannak.
- A felhasználói bejelentkezéseket kezelő WinLogon processz LPC-hívásokat használ arra, hogy kommunikáljon helyi biztonsági jogosultság ellenőrző szerverrel (LSASS).

Az LPC-kommunikációt leggyakrabban az NT szolgáltatásait megvalósító szerver folyamatok használják a kliens folyamatokkal történő kapcsolattartásra. LPC-kapcsolat létrehozható két felhasználói módú folyamat között, vagy egy kernel módú komponens és egy felhasználói módú folyamat között.

Az LPC az üzenetváltás három különböző formáját teszi lehetővé:

- A 256 bájtnál rövidebb üzenet úgy küldhető el, hogy az LPC-üzenet küldésekor közvetlenül megadjuk egy pufferben az üzenetet. Az üzenetet a rendszer a híváskor átmásolja a saját címtartományába, majd onnan a hívott folyamat címtartományába.
- Ha egy kliens és egy szerver 256 bájtnál több adatot szeretne továbbítani, akkor mindketten meg kell hogy nyissanak egy osztott elérésű memóriaterületet. A küldő az üzenetet eltárolja az osztott elérésű memóriába, majd egy rövid LPC-üzenetet (lásd első eset) küld a hívott folyamatnak, melyben egy pointert ad az üzenet kezdetére.
- Ha egy szerver több adatot akar írni vagy olvasni, mint amennyi az osztott elérésű memórialapon elférne, akkor a szerver folyamat – megfelelő elérési tokenek megszerzése után – közvetlenül is elérheti a kliens címtartományát, ahonnan tud olvasni és oda írni. A üzenetváltás szinkronizálására megint csak rövid üzeneteket (lásd első eset) használhatnak a kommunikáló folyamatok.

6.4. Folyamatok kezelése és ütemezése

Ebben az alfejezetben azokat az adatstruktúrákat és algoritmusokat mutatjuk be, amelyek a folyamatokkal és szálakkal kapcsolatosak a Windows NT 4.0-s operációs rendszerben.

6.4.1. A Windows NT folyamatmodellje

A folyamatok az NT-ben egy adott program kódját végrehajtó (egy vagy több) szálból, valamint a szálak által lefoglalt erőforrásokból állnak. Az NT folyamat modellje a következő dolgokat foglalja magába:

- A végrehajtandó program kódját és adatait.
- Saját virtuális memória címteret, amit a folyamat használhat.
- Rendszererőforrásokat (szemaforokat, fájlokat stb.) amiket az operációs rendszer foglal a folyamat számára, amikor a folyamathoz tartozó szálak azokat megnyitják.
- A folyamat egyedi azonosítóját (process ID, PID).
- Legalább egy végrehajtható szálat.

A szál az az egység (entitás) az NT-ben, amit az ütemező kezel és végrehajtásra ütemez a CPU-hoz. A szálak a következő komponensekből állnak:

- A szálat végrehajtó processzor regiszterei, melyek a processzor állapotát írják le.
- Két veremtárat *(stack)*. Egyet a kernel módban történő program-végrehajtáshoz, egyet a felhasználói módban történő program-végrehajtáshoz.
- Kizárólagosan használható tárterületet a DLL-ek, run-time könyvtárak számára.
- A szál egyedi azonosítóját *(thread ID)*. (Megjegyzendő, hogy a thread ID és a process ID ugyanabból a névtérből kerül ki, így egy rendszerben sohasem lehet két egyező.)

A felsorolás első három elemét együttesen a szál környezeteként (thread context) emlegetjük.

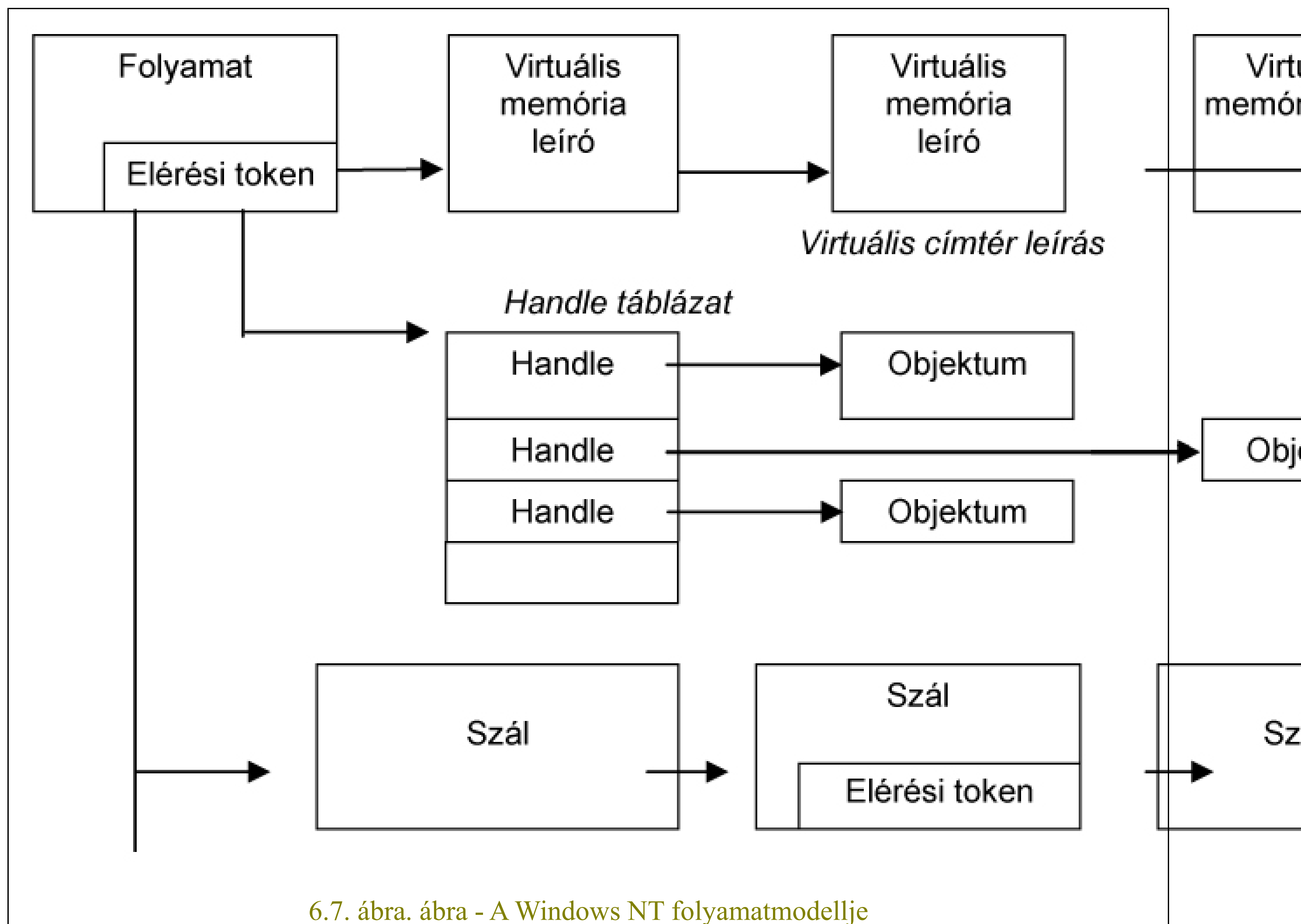
Láthatjuk, hogy az egy folyamathoz tartozó szálak közös virtuális címtartományt használnak. Az egyes folyamatok viszont külön címtartományban futnak, csak osztott elérésű memória használata esetén lehet átfedés két folyamat által használt memóriaterület között.

A folyamatok illetve egészen pontosan a folyamatokat alkotó szálak által használni kívánt memóriaterületeket természetesen a rendszertől kérni kell, le kell foglalni azokat.

A memóriához hasonlóan a folyamatnak le kell foglalni a folyamat által használni kívánt erőforrásokat, amelyeket az NT objektumokként reprezentál. Minden ilyen objektumhoz a megnyitása után a folyamat a gyorsabb elérés érdekében egy ún. handle-t kap.

A rendszer erőforrásainak védelme, illetve az erőforrás használat szabályozása érdekében minden folyamat rendelkezik egy ún. elérési tokennel, mely tartalmazza a folyamat biztonsági azonosítóját, illetve a folyamat jogosultságainak leírását.

Az NT folyamatmodelljének vázlatos képét a 6.7. ábrán láthatjuk.



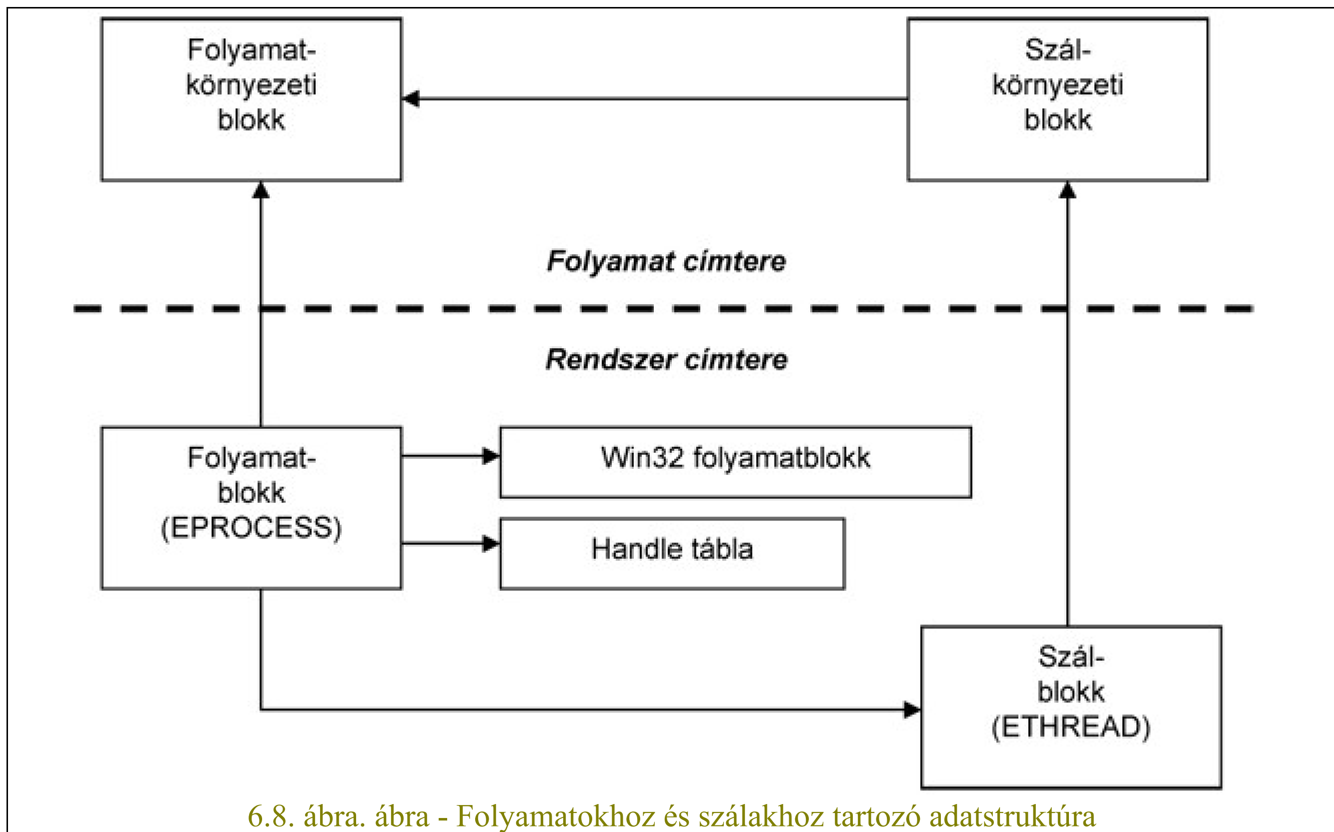
6.7. ábra. ábra - A Windows NT folyamatmodellje

6.4.2. Folyamatok kezelése a Windows NT-ben

A Windows NT executive rétege a folyamatokhoz tartozó adatokat egy ún. folyamatblokkban (EPROCESS) tárolja. Minden folyamathoz hozzárendel az NT egy ilyen adatstruktúrát a folyamat indulásakor. A blokk a folyamat kísérő adatait tartalmazza, valamint egy sereg mutatót a kapcsolódó adatstruktúrákra. Például minden folyamaton belül van egy vagy több szál, ahol a szálakat ún. executive szálblokk (ETHREAD) ír le. Az EPROCESS blokk és a hozzá tartozó adatstruktúrák a rendszer címterében helyezkednek el. Kivétel ez alól a folyamat futási környezetét leíró ún. folyamatkörnyezeti blokk (PEB: Process Environment Block), amely a folyamat címterében található. Ennek oka, hogy ez az adatstruktúra olyan információt tartalmaz, amit a felhasználói módban futó kód is megváltoztathat.

Az EPROCESS blokkon felül a Win32 alrendszer folyamata is kezel egy folyamatleíró adatstruktúrát (W32PROCESS). Minden Win32 kódot végrehajtó folyamathoz tartozik egy-egy ilyen adatstruktúra.

A folyamatok és szálak adatstruktúrájának egyszerűsített vázlatát a 6.8. ábrán mutatjuk be.



6.8. ábra. ábra - Folyamatokhoz és szálakhoz tartozó adatstruktúra

Folyamat létrehozása (CreateProcess)

Egy Win32 folyamat akkor jön létre, amikor egy alkalmazás meghívja a Win32 *CreateProcess* függvényét. A létrehozás több fázisból áll, amelyeket az operációs rendszer három részlete valósít meg: a Win32 kliensoldali könyvtárából a KERNEL32.DLL, a Windows NT executive, valamint a Win32 alrendszer folyamat (CSRSS). Mivel a Windows NT több környezeti alrendszert kezel, emiatt egy Windows NT executive réteg processz objektumának kezelése (amit más alrendszerek is tudnak használni) szét van választva attól a tevékenységtől, ami a Win32 folyamat létrehozásával jár.

A következő felsorolás összefoglalja a Win32 CreateProcess hívásának főbb lépéseit:

1. A processzen belül végrehajtandó image-fájl (.EXE) megnyitása.
2. A Windows NT executive processz objektumának létrehozása.
3. A kezdeti szál létrehozása.
4. A Win32 értesítése az új processzről, azzal a céllal, hogy az felkészüljön az új processzre és szálra.
5. A kezdeti szál végrehajtásának elindítása.
6. Az újonan létrehozott processz és szál környezetben a címtér inicializálása (például a szükséges DLL-ek betöltése), majd a program végrehajtásának elkezdése.

A fenti lépések mindegyikére vonatkozóan az alábbi megjegyzések érvényesek:

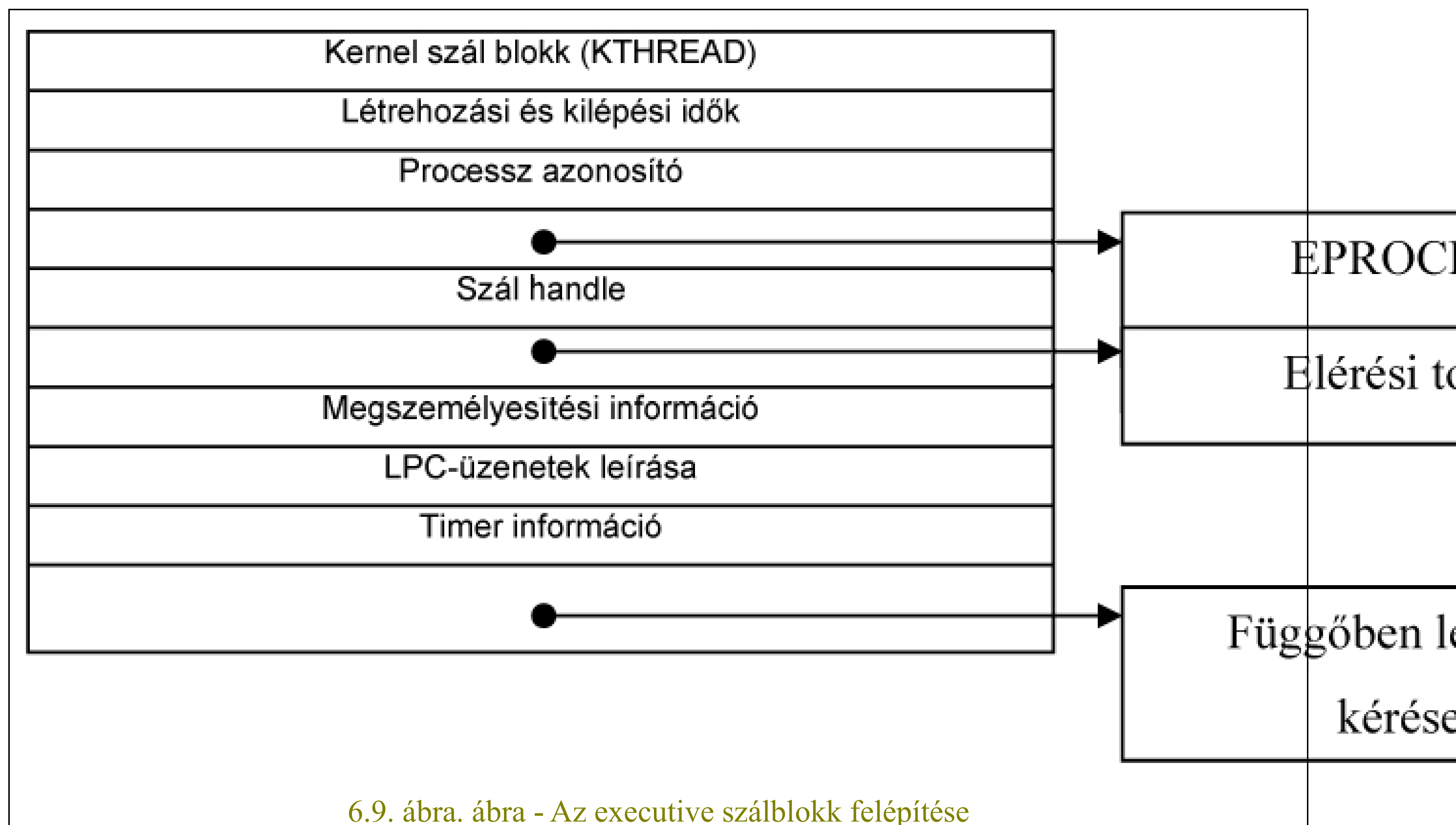
- Egyetlen CreateProcess híváshoz egynél több prioritási osztály specifikálható. A Windows NT a processzhez automatikusan a legalacsonyabb prioritási osztályt rendeli.
- Ha nincs prioritási osztály specifikálva az új folyamathoz, a prioritási osztály a Normal besorolást kapja.
- Minden ablakot egy desktophoz rendel a rendszer. (Desktopnak nevezi az NT a felhasználó által használható grafikus környezetet. Egy munkahelyen több desktopot is képes kezelni a rendszer.) Ha a CreateProcessben nincs megadva, melyik desktopot használja a folyamat, akkor a folyamat automatikusan a hívó aktuális desktopjához rendelődik.

6.4.3. Szálak kezelése az NT-ben

A következőkben először a szálak struktúrájával foglalkozunk. A továbbiakban, hacsak másként nem említjük, az itt leírtak egyaránt érvényesek mind a normál felhasználói módú szálakra, mind pedig a kernel módú rendszerszálakra.

Az operációs rendszer szintjén egy Windows NT szálat egy ún. executive szál blokk (ETHREAD) reprezentál. A blokkot a 6.9. ábra mutatja be. Az ETHREAD blokk és azok az adatstruktúrák, melyekre az ETHREAD blokkban tárolt pointerek mutatnak, a rendszer címtartományában találhatók, a szál környezeti blokkjának kivételével, ami a folyamat címtartományában van. Az ETHREAD blokkon kívül, a Win32 alrendszer folyamata is kezel egy szálleíró adatstruktúrát minden egyes szálhoz, ami egy Win32-es folyamatban jött létre.

Az ábrán látható első mező a kernel szálblokk (KTHREAD). Ezt követi a szálazonosítási információ, a processzazonosítási információ, a saját processzhez tartozó pointerrel, biztonsági információ az elérési tokenre mutató pointer formájában, a megszemélyesítési információ, és végül az LPC-üzenetekre, illetve a függőben levő B/K-kérésekre vonatkozó mezők. A KTHREAD blokk tartalmazza mindazokat az adatokat, amelyekre a kernelnek van szüksége a szálütemezés és a szinkronizáció végrehajtása érdekében.



6.9. ábra. ábra - Az executive szálblokk felépítése

Szál létrehozása (CreateThread)

Egy szál életciklusa akor veszi kezdetét, amikor egy program új szálat hoz létre. Az erre irányuló kérés eljut a Windows NT executive-hoz, ahol a *processzkezelő* (elnevezése: *process dispatcher*) helyet jelöl ki egy szálobjektum számára, majd a kernelt hívja, a kernel szálblokk kezdeti értékeinek beállítása végett. A következőkben végigvesszük azokat a lépéseket, amelyek a Win32 *CreateThread* függvénye szerint történnek a KERNEL32.DLL-ben, azzal a céllal, hogy egy Win32 szál jöjjön létre:

1. A CreateThread egy felhasználói módú stacket hoz létre a szál részére, a folyamat címterében.
2. A CreateThread beállítja a kezdeti értékeket a szál hardverkapcsolataihoz.
3. Az NtCreateThread függvény hívására kerül sor, ami az executive szál objektumát hozza létre. Az ide tartozó lépések sorozata kernel módban hajtódik végre, a Windows NT executive-ján és kernelén belül.
4. A CreateThread értesíti a Win32 alrendszert az új szálról, amire az alrendszer különböző beállításokat eszközöl az új szál részére.
5. A szál összetett elérési címe (handle) és azonosítója (amik a 3. lépésben lettek generálva) visszaadódik a hívónak.
6. A szál olyan állapotba került, hogy ütemezni lehet a végrehajtását.

6.4.4. Szálak ütemezése

Ebben a részben az ütemezési elvekkel foglalkozunk. A Windows NT *prioritáson alapuló, preemptív (kiszorító)* ütemezési elvet valósít meg. Egy NT rendszerben mindig a legmagasabb prioritású futtatható szál fut, azzal a megkötéssel, hogy a futást azok a processzorok korlátozhatják, amelyeken a szál futása meg van engedve. Ezt a jelenséget *processzor-affinitásnak* nevezzük. Általában egy szál bármelyik rendelkezésre álló processzoron futhat, de a processzor-affinitás megváltoztatható a Win32 ütemezési függvényeinek felhasználásával.

6.4.4.1. A kvantum

Amikor egy szál futásra választódik ki, egy előre megszabott ideig fog futni. Ezt az időszeletet *kvantumnak (quantum)* nevezzük. Egy kvantum az az időtartam, amennyit egy szál futhat, mielőtt a Windows NT megszakítja a szálat azért, hogy kiderítse, nem vár-e futásra egy másik szál ugyanakkora prioritással, vagy pedig nincs-e szükség a megszakított szál prioritásának csökkentésére. A kvantumok értéke szálanként változhat. Ugyanakkor azonban az is lehetséges, hogy egy szál nem jut el a kvantumja végéig. Ez a szituáció a preemptív ütemezés következtében léphet fel: ha egy másik szál, nagyobb prioritással, futásra kész állapotba kerül, a futásban levő szál ki lesz szorítva a saját időszelete lejárta előtt. Ezenfelül még az is előfordulhat, hogy egy szál futásra lesz kiválasztva, és még azelőtt leállítódik, hogy elkezdené a kvantumját.

A Windows NT ütemezési funkcióit a kernel valósítja meg. Nincs azonban külön ütemező modul a kernelen belül, az ütemező rutinok kódja a kernel különböző helyein van szétosztva. Az ütemezést megvalósító rutinok összességét közös néven a kernel *diszpécserének (dispatcher)* nevezik. Egy szál ütemezése az IRQL 2-es szintjén fordul elő, és a következő események bármelyike elindíthatja:

- Egy szál futásra kész állapotba kerül. Például egy újonnan kreált szál, vagy egy olyan, ami éppen felszabadult a várakozásból.
- Egy szál futása leáll, mivel a kvantumja véget ért, illetve befejeződött a futás, vagy várakozási állapotba került a szál.
- Egy szál pioritása megváltozik, vagy egy rendszerkiszolgálási hívás miatt, vagy mert maga a Windows NT változtatja meg a prioritási értéket.
- Egy futásban levő szál processzor-affinitása megváltozik.

Az operációs rendszernek a fenti esetek mindegyikében el kell döntenie, melyik szál fog futásra következni. A futó szál váltásakor a rendszer környezetváltást hajt végre: elmenti az éppen futó szál környezetét és betölti a futásra kijelölt szál környezetét.

Az ütemezési döntések szigorúan a *szálak alapján* történnek, ami azt jelenti, hogy egyáltalán nem számít, hogy egy szál melyik processzhez tartozik. Ha például egy A processznek 10 futtatható szála van, egy B-nek pedig 2, és mindegyik szálnak ugyanaz a prioritása, akkor mindegyik szál a CPU-idő 12-ed részét kapja. Vagyis a Windows NT nem ad 50 százalék CPU-időt az A processznek, és 50 százalékot a B-nek.

A szálütemező algoritmusok szorosan kötődnek a prioritási szintekhez. A Windows NT 32 prioritási szintet használ, 0-tól 31-ig. Ezek felosztása a következő:

- tizenhat valós idejű szint (16–31),
- tizenöt változószint (1–15),
- egy rendszerszint (0), ami le van foglalva minden NT-rendszerben egyedül létező ún. *zérus oldal (zero page)* szálának.

A szálakhoz rendelt kvantumok értékét a Windows NT a következő tényezők beszámításával határozza meg:

- Minden szálnak van egy kijelölt kvantumértéke, ami nem időtartam, hanem egy egész szám, amit nevezzünk kvantum egységnek.
- Határozatlan helyzetben a szálak a Windows NT Workstation-ön 6-tal indulnak, míg a Windows NT Serveren 36-tal. Az utóbbi érték azért nagyobb, hogy több idő jusson a kliensek beérkező kéréseinek biztonságos kiszolgálására.
- Amikor az órajel ad megszakítást, akkor a szál kvantumából 3 levonódik. Ha az 0 vagy kevesebb lesz, lejár a kvantum, és új szál lesz futásra kiválasztva. Eszerint az NT Workstation-ön 2 óraintervallum, az NT Serveren pedig 12 óraintervallum jut egy szál futására ilyenkor.
- Az óraintervallumok hossza természetesen függ a hardver platformtól. Másrészről a megszakítások gyakoriságát a HAL működése szabja meg, nem pedig a kernel. Például, az Intel rendszerek óraimpulzusa 10 msec (486-os), illetve 15 msec (Pentium Pro), míg a DEC Alpha AXP rendszereké 7,8 msec.

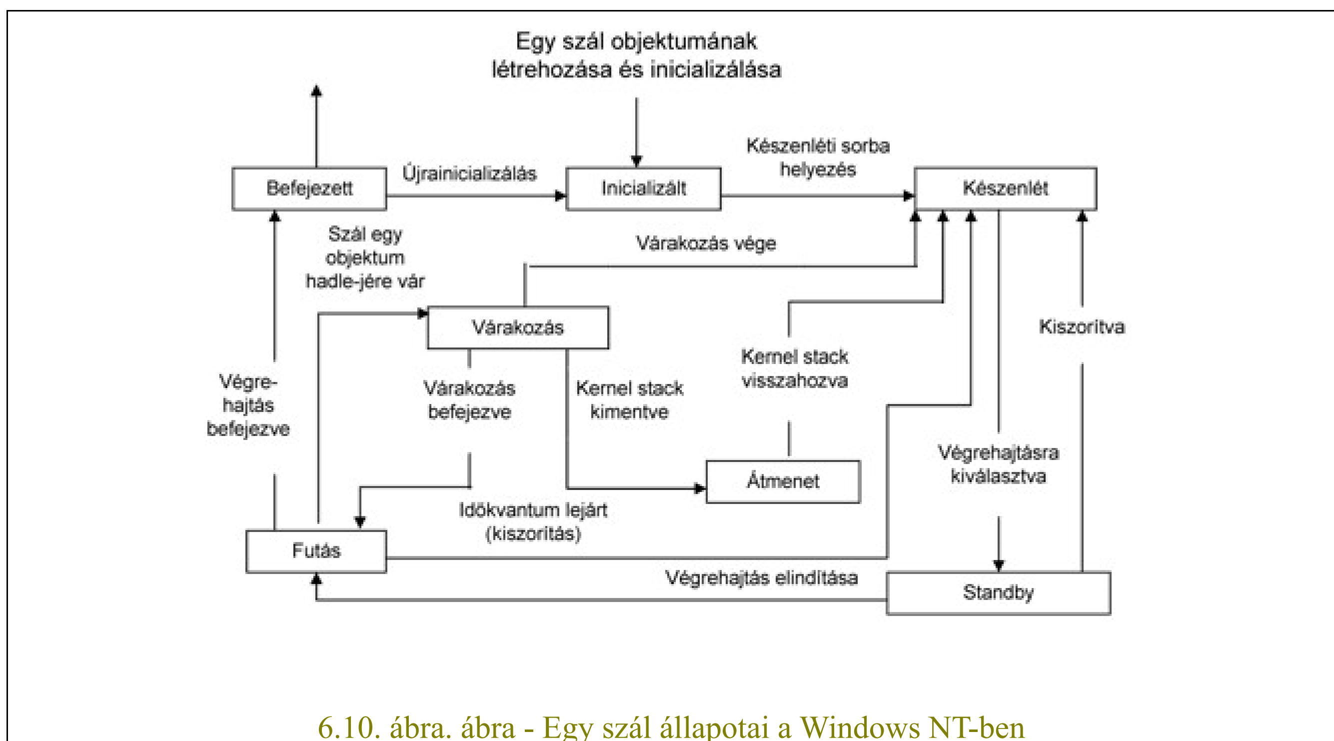
6.4.4.2. Egy szál állapotai

A Windows NT operációs rendszerben egy szál futása során több végrehajtási állapoton megy keresztül. Ezek a lehetséges állapotok a következők:

- *Készenlét.* A szál végrehajtásra kész állapotban van. A diszpécser csak az ebben az állapotban levő szálak halmazát veszi figyelembe a végrehajtás ütemezésekor.
- *Standby.* Egy szál akkor van standby állapotban, ha a dispatcher kiválasztotta egy adott CPU-n futásra az éppen futó szálat követően. Egy processzorhoz csak egyetlen szál létezhet standby állapotban az operációs rendszeren belül.
- *Futás*. Miután a diszpécser környezetváltást hajt végre, a futási jogot megkapó szál futási állapotba kerül és megindul a végrehajtása. A végrehajtás addig folytatódik, amíg a kernel ki nem szorítja a szálat egy nagyobb prioritású szál futása érdekében, vagy lejár a szál kvantuma, a szál befejeződik, vagy pedig saját magától a várakozási állapotba lép.

- *Várakozás*. Egy szál több módon léphet be a várakozási állapotba: önkényesen várhat egy objektumra, hogy az szinkronizálja a végrehajtását, az operációs rendszer (a B/K-rendszer például) várakozhat a szál érdekében, vagy egy környezeti alrendszer irányítja a szálat önmaga felfüggesztésére. Amikor a szál várakozása véget ér, akkor a prioritástól függően a szál vagy azonnal elkezd futni, vagy visszakerülhet készenléti állapotba.
- *Átmenet*. Egy szál az átmeneti állapotba lép át, ha készen áll a végrehajtásra, de az ő kernel stackje ki van mentve (page-elve) a memóriából. Miután a szál kernel stackje visszakerül a memóriába, a szál készenléti állapotba megy át.
- *Befejezett*. Amikor egy szál végrehajtása véget ér, a szál a befejezett állapotba kerül. Ebben az állapotban a szál objektumának a törlése az objektumkezelőtől függ. Ha az executive-nak van pointere az objektumhoz, az executive inicializálva ismét fel tudja használni a szál objektumát.

Az állapotok közötti átmenet gráfját a 6.10. ábra mutatja be.



6.10. ábra. ábra - Egy szál állapotai a Windows NT-ben

Multiprocesszoros környezetben a szálütemezés megoldása a CPU-k közötti munkamegosztás intenzív szervezését követeli meg. Mindent összevéve a Windows NT úgy jár el, hogy megkísérli a legmagasabb prioritású futtatható szálakat a szabadon lévő CPU-khoz ütemezni. Mindazonáltal több tényező befolyásolja annak megválasztását, hogy egy szál melyik CPU-n fusson. A kiválasztási algoritmus ismertetése előtt néhány fogalmat vezetünk be.

6.4.4.3. A processzoraffinitás

Minden egyes szál egy ún. *affinitási maszkkal (affinity mask)* rendelkezik, amely azokat a processzorokat adja meg, amelyeken a szál futása megengedett. A szál affinitási maszkja a processz affinitási maszkjából öröklődik. Alapesetben mindegyik processz, és ennek megfelelően mindegyik szál egy olyan affinitási maszkkal kezd, amely megegyezik a rendszer aktív procesz-szorainak a halmazával. Más szóval, mindegyik szál tud futni mindegyik processzoron. Ezt az állapotot két dolog képes megváltoztatni:

- Egy hívás az alkalmazás felől, amely át tudja állítani a processzhez, illetve a szálhoz tartozó maszkokat: ez a *SetProcessAffinityMask,* illetve a *SetThreadAffinityMask* függvények hívása.
- Az image-fejlécben specifikált affinitási maszk, amely a teljes image-re érvényes.

Mindegyik szálnak van két CPU-sorszáma, amik a kernel szálblokkban vannak tárolva:

- *ideális processzor,* vagyis a kitüntetett processzor, amin a szálnak futni kell,
- *következő processzor,* vagyis az a processzor, amelyik a szál következő futására lett kiválasztva (vagy amin az utoljára futott).

Az ideális processzor kiválasztása véletlenszerűen történik, amikor egy szál létrejön. Ehhez a processzblokkban egy számláló van fenntartva. Ez a számláló minden egyes szál létrehozásakor eggyel növekszik, ami által az új szálak ideális processzorai körben választódnak ki a rendelkezésre állókból. A Windows NT már nem változtatja meg az ideális processzort, miután a szál létrejött. A változtatásra csak az alkalmazásoknak van módjuk az olyan szálnál, ami a *SetThreadIdealProcessor* függvényt használja.

6.4.4.4. A processzor kiválasztása

Amikor egy szál futásra kész állapotba kerül, a Windows NT először egy tétlen processzorhoz próbálja a szálat ütemezni. Ha a tétlen processzorokból lehet választani, akkor előnyben részesül a szál ideális processzora, majd ezután jön a szál következő processzora. Ha ez sem áll rendelkezésre, akkor az utasításokat éppen végrehajtó processzor jön sorra. (Vagyis az, amelyiken az ütemezési kód fut.) Ha ezen CPU-k egyike sem tétlen, akkor az első rendelkezésre álló tétlen processzor lesz kiválasztva úgy, hogy a Windows NT végigellenőrzi a tétlen processzorok maszkját, növekvő CPU-sorszám szerint.

Ha pillanatnyilag mindegyik CPU foglalt, és egy szál futásra kész állapotba került, a Windows NT azt vizsgálja meg, hogy van-e olyan futásban vagy várakozásban levő szál, amit ki lehetne szorítani valamelyik CPU-n. A kiválasztás elve ilyenkor a következő.

Az első választás a szál ideális processzora, a második pedig a következő processzora. Ha egyik CPU sincs a szál affinitási maszkjában, akkor az első olyan aktív processzor lesz kiválasztva, amelyiken a szál futni tud.

Ha ez a processzor olyan, amin egy másik szál van legközelebb futásra jelölve (vagyis standby állapotban várakozik az ütemezésre), és annak a szálnak a prioritása kisebb, mint az indításra előkészített szálé, akkor az új szál kiszorítja a másikat a standby állapotból, és ő maga lesz a CPU következő szála. Ha ennél a CPU-nál

nincsen következő szál futásra választva, akkor az NT megvizsgálja, hogy az éppen futó szál prioritása kisebb-e, mint az új szálé. Ha kisebb, a futásban levő szálat kiszorításra jelöli ki, és elhelyez a sorban egy processzormegszakítást arra, hogy a futó szál félrelökődjék az új szál javára.

Megjegyezzük, hogy a Windows NT nem vizsgálja az összes CPU-n a jelenlegi és a következő szálak prioritását, hanem csak azon az egy CPU-n teszi ezt, amit a fentiek szerint választott ki. Ha ezen a CPU-n nincsen kiszorítható szál, akkor az új szál bekerül a prioritási szintje szerinti készenléti sorba, ahol várakozni fog, amíg ütemezve nem lesz.

Mint látható volt az előbbiekben, multiprocesszoros környezetben a Windows NT nem mindig a legnagyobb prioritású szálat választja ki egy CPU-n való futtatásra. Így egy olyan szál, ami egy adott CPU-n éppen futó szálnál nagyobb prioritású, készenlétbe kerülhet, de nem biztos, hogy azonnal kiszorítja a futó szálat.

6.5. Memóriakezelés

Az Windows NT memóriakezelőjének feladata, hogy a folyamatok virtuális címterének címeit megfeleltesse fizikai címeknek, vagyis elvégezze a logikai-fizikai címtranszformációt. A másik fontos feladata a virtuális memóriakezelés megvalósítása, vagyis a memória terheltsége esetén a régen nem használt memóriaterületeket háttértárra menti, valamint ha hivatkozás történik egy háttértárra mentett memórialapra, a szálat várakoztatja, míg bemozgatja a kérdéses lapot a fizikai memóriába.

Az NT 32 bites virtuális memóriakezelést valósít meg. A memóriát laponként (page) kezeli, foglalás, felszabadítás, háttértárra mentés esetén memórialapokat mozgat.

A fent említett alapszolgáltatáson felül az NT memóriakezelője olyan többletszolgáltatásokat is ad a felhasználóknak, melyek nagyban megnövelik a rendszer hatékonyságát. Ilyen szolgáltatás például a fájlok memóriaként történő elérése (memory mapped files), ami lehetővé teszi a fájlok osztott elérését, vagy a copy-on-write mechanizmus használata, mely lehetőségeket a fejezetben részletesen is ismertetjük.

6.5.1. Memória manager felhasználói interfésze

Az NT memóriakezelőjének szolgáltatásait az alkalmazások a Win32 API interfész függvényeinek meghívásával érhetik el. Az NT memóriakezelője a következő szolgáltatásokat nyújtja a felhasználóknak:

- virtuális memória allokáció illetve felszabadítás,
- osztott (shared) memória létrehozása,
- fájlok osztott memóriához hasonló elérése,
- virtuális memória kezelése (információkérés, memóriába rögzítése, kiírása háttértárra stb.),
- memória védelmi funkciók,
- kernel szintű funkciók (elsősorban a device driverek) támogatása (például fix fizikai memóriaterület használata).

6.5.2. Memóriafoglalás

A memóriafoglalás két lépésben történik a Windows NT-ben:

- *reserve:* virtuális címtartomány lefoglalása.
- *commit:* virtuális memória lefoglalása.

A felhasználó természetesen kérheti a két lépés egy függvényhívásban történő végrehajtását. A reserve művelet még nem jelent tényleges memóriafoglalást. A reserve végrehajtásával a folyamat csak deklarálja az operációs rendszer számára, hogy mennyi memóriára lesz, vagy lehet szüksége. A rendszer csak a commit művelet hívásakor foglal tényleges tárterületet a korábban a reserve művelettel már lefoglalt memória részére. Ennek megfelelően csak a commit művelet után tudja a folyamat a memóriát használni, a csak reserved memóriacímre történő hivatkozás hibát okoz.

A memóriafoglalás két lépcsőben történő megvalósításának célja a hatékonyabb működés biztosítása. A reserve hatására az operációs rendszer csak egy belső táblázatában jelzi, hogy a folyamat lefoglalta az adott címtartományt. A rendszer csak a commit hatására fog tényleges memórialapokat (page), illetve ún. backing store-t, a memóriaterület potenciális mentésére szolgáló háttértárat lefoglalni. A kétlépéses memóriafoglalással lehetősége nyílik egy folyamatnak előre lefoglalni egybefüggő címtartományokat úgy, hogy csak a memória tényleges használata esetén – vagyis a commit művelet végrehajtása után – terheli az a rendszer memóriáját.

Jól érzékelhetjük a memóriafoglalás két lépcsőben történő megvalósításának előnyeit például a szálak user stackjének foglalásakor. A stacknek természetesen egy folyamatos címtartománynak kell lennie. Alapértelmezés szerint, ha a szál indításakor másképpen nem kérjük, a rendszer 1 MB memóriát foglal reserve művelettel, de csak 2 lapnyi (x86-os rendszerekben kétszer 4 KB) memóriát foglal commit művelettel. (Az első lap a stack teteje lesz, a második lap pedig arra szolgál, hogy a rendszer érzékelje, ha a stack megtelt, és automatikusan foglaljon commit művelettel új oldalakat.) Ha nem lenne kétlépéses memóriafoglalás, minden szál indulásakor a rendszermemóriából ténylegesen le kellene foglalni 1 MB-nyi területet, melynek valószínűleg jelentős részét a szálak többsége nem is használná. A kétlépcsős memóriafoglalásnak köszönhetően azonban a stack csak akkor terheli a rendszer memóriáját, ha az ténylegesen is használatra kerül.

Meg kell említeni, hogy a memóriafoglalás műveletek mindig lapokat (page) foglalnak, akkor is, ha a folyamat ennél kisebb memóriát kér. (Az x86-os rendszerekben a memórialapok mérete 4 KB.) Ráadásul a reserved (és így természetesen a commited) memórialapok mindig 64 KB-os memóriaegységek határán kezdődnek, ezzel támogatva a memórialapok későbbiekben történő esetleges növelését.

6.5.3. Osztott elérésű memória

Az osztott elérésű memória használata a folyamatok közötti információcsere lehetőségén túl adott esetben jelentősen lecsökkentheti a rendszer memóriaterhelését. Vegyük például azt az esetet, hogy a rendszerben két folyamat is éppen C programot fordít. Az osztott memória használatával lehetséges, hogy a C fordító csak egy példányban kerüljön a memóriába.

A Windows NT az osztott elérésű memóriát (shared memory) illetve a fájlok osztott elérését hasonló módon kezeli. A kezelés alapeleme az ún. section object (szekció objektum). (A section object név helyett a Win32 API file mapping object elnevezést használ.)

A section object lényege, hogy létrehozásakor hozzátársíthatjuk egy fájlhoz vagy a fájl egy részéhez. Ha a section objecthez társított fájl a rendszer memóriáját a háttértáron reprezentáló ún. page file, a section objecttel osztott memóriát fogunk elérni. A section objectet létrehozása után akár több folyamat is megnyithatja, így biztosított a párhuzamos használata a section object által elért objektumnak, ami tehát lehet memória vagy egy létező fájl is.

Az NT-ben a fájlok mérete, így a section objecten keresztül elért fájlok mérete is lényegesen nagyobb lehet, mint a folyamat memória címtartománya. Ebben az esetben a folyamat nem tudná az egész fájlt elérni. Ennek a problémának a megoldására biztosítja az NT a fájlok egy részének, ún. nézetének (view) a mappelési lehetőségét. A section object létrehozásakor a folyamat definiálhatja, a fájl melyik nézetét, melyik részét szeretné az adott section objecttel elérni.

A fájlok section objecten keresztül történő elérése az NT-ben egy nagyon gyakran alkalmazott művelet. Az alkalmazások így hajtják végre azokat a B/K-műveleteiket, amelyek kimenetét egy fájlba szeretnék irányítani, vagy például így történik a végrehajtható (ún. image) fájlok és a DLL-ek memóriába mappelése is. A cache manager section objectek segítségével éri el az általuk kezelt fájlokat.

6.5.4. Memóriavédelem

A memóriavédelem biztosítja, hogy sem két folyamat, sem egy folyamat és az operációs rendszer ne módosíthassa az egymás címtartományába tartozó memóriaterületeket. (Az osztott elérésű memória természetesen kivétel ez alól.) A kiterjedt memóriavédelem az egyik legfontosabb funkciója az NT-nek, ami miatt megbízhatóan működő, robusztus operációs rendszerként tartják az NT-t számon.

A memóriavédelem négy szinten valósul meg:

- Az operációs rendszer kernel módban futó komponensei által használt adatstruktúrák a felhasználói módban futó folyamatok számára elérhetetlenek. Bármilyen ilyen elérési kísérlet hardver megszakítást eredményez, és a memóriakezelő egység jelzi a memória elérést kezdeményező szálnak az elérési jog megsértését.
- Minden folyamat külön virtuális címtartományt használ, melyeknek – eltekintve az osztott elérésű memóriától – nincs közös részük. A folyamatok a memóriát a memóriakezelő egységen keresztül érik el, ami amellett, hogy megfelelő hardver támogatással elvégzi a logikai–fizikai címtranszformációt, biztosítja, hogy egyik folyamat se érjen el egy másik folyamat által használt memórialapot.

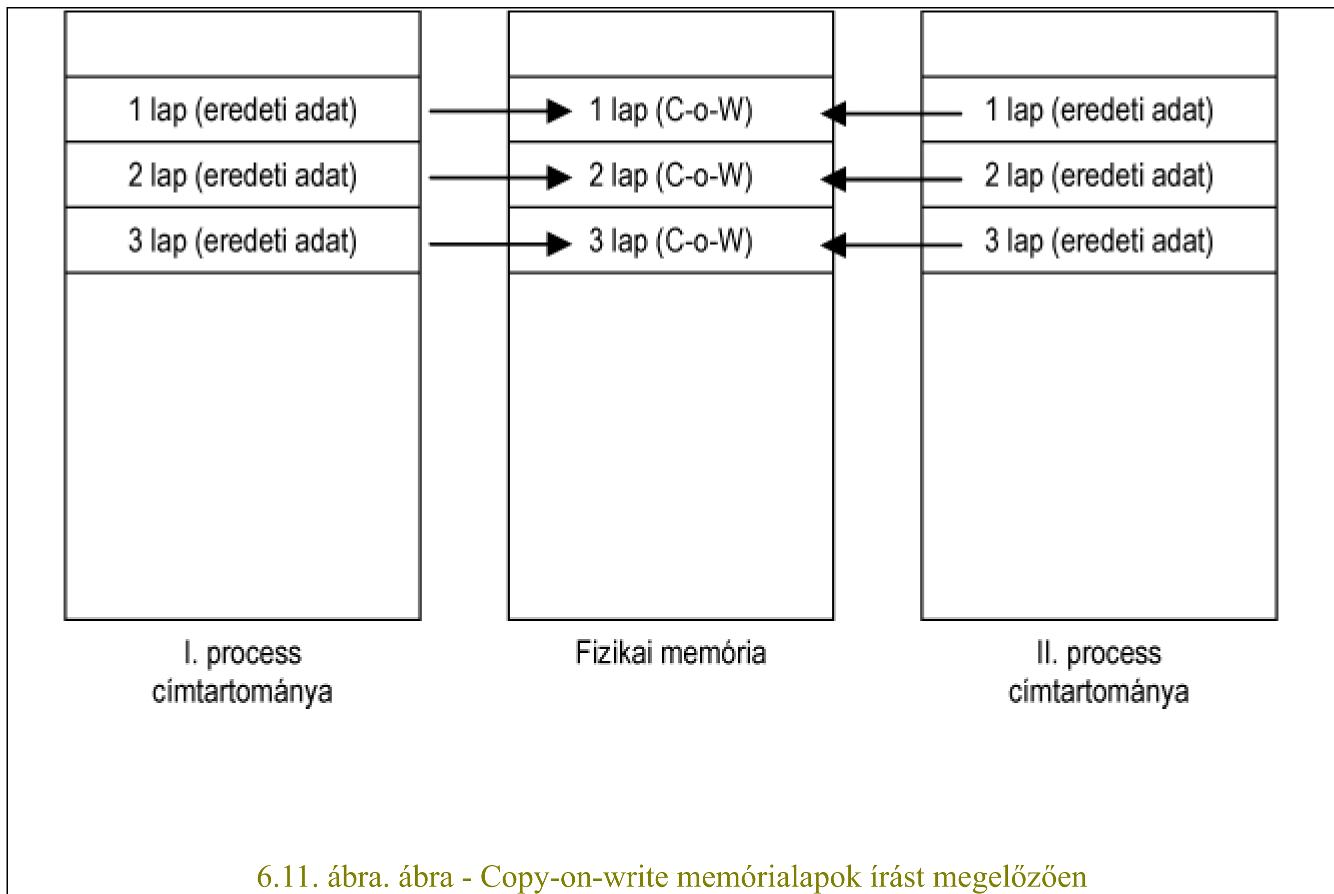
- A fenti, a memóriakezelő címtranszformációja által biztosított implicit memóriavédelmen túl az NT által támogatott processzorok lehetőséget adnak hardver memóriavédelemre. A processzortípustól függően az egyes memóriaterületek különböző védelmi attribútumúak lehetnek (például csak írhatóak, csak olvashatóak, írhatóak és olvashatóak stb.) Az egyes területek elérésekor a memóriakezelő hardver ellenőrzi az elérési jogosultságot. Például a kódszegmensek betöltéskor csak olvasható attribútumot kapnak, így megelőzve a kód átítását.
- A negyedik védelmi szint az osztott memória elérését biztósító section objectekhez kötődik. A section objectekhez definiálható egy ún. elérési lista (ACL: Access Control List), melyben megadható, hogy mely folyamatok jogosultak az adott section objectet elérni. Ez a módszer teljesen illeszkedik az NT-objektumok védelmi rendszerébe.

6.5.5. Copy-on-Write

A copy-on-write memóriahasználat – mely fordítása talán „írás esetén másolás” lehetne – egy széles körben alkalmazott módszer a használt memória illetve a rendszer memóriakezelő műveleteinek optimalizálására. A copy-on-write jó példája az ún. *lusta kiértékelés/végrehajtás (lazy evaluation)* elvnek, ami azt mondja, hogy a rendszerben a költséges, vagyis sok erőforrást igénylő műveleteket el kell halasztani mindaddig, amíg azok végrehajtása már ténylegesen kikerülhetetlen nem lesz.

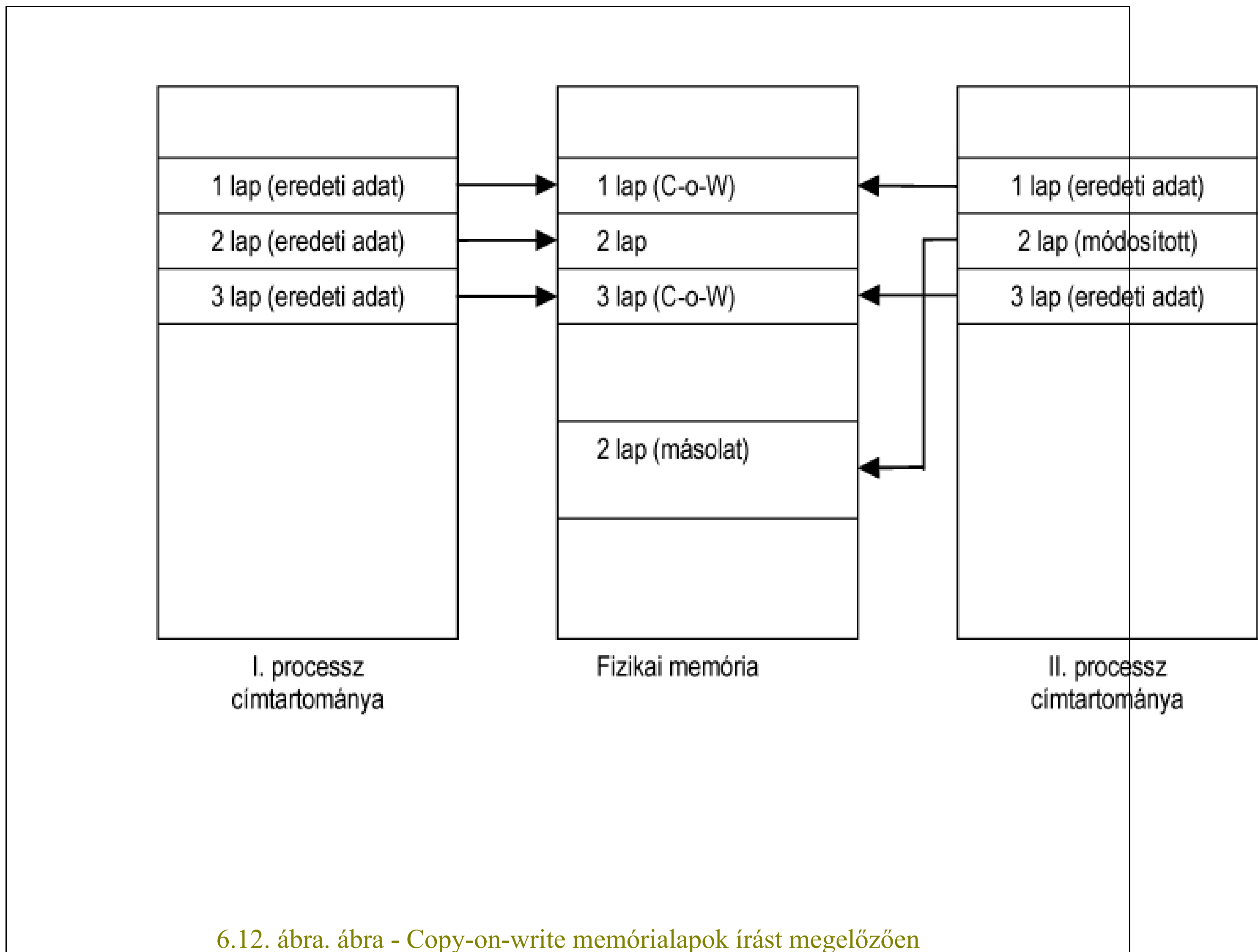
A copy-on-write memóriahasználat működése a következő.

Ha egy folyamat egy írható-olvasható, más folyamatok által is potenciálisan használt section objectet kezd használni, akkor a rendszer nem készít a folyamat számára egy lokális másolatot az egész elért memóriatartalomról annak érdekében, hogy a memória megváltoztatása csak a kérdéses folyamat számára legyen látható. Ehelyett az NT csak bebillenti a memórialapokhoz tartózó copy-on-write (C-o-W) jelzőbitet (flaget) (6.11. ábra).



6.11. ábra. ábra - Copy-on-write memórialapok írást megelőzően

Tényleges másolás a rendszerben csak akkor történik, amikor a section object bármelyik használója kezdeményez egy memória írási műveletet. Ekkor a rendszer mielőtt megváltoztatná a memóriatartalmat, a kérdéses memórialapról egy privát másolatot készít az írást kezdeményező folyamat számára (6.12. ábra). Így biztosítja a rendszer „takarékosan”, hogy a változtatás csak az írást kezdeményező folyamat számára legyen látható.



6.12. ábra. ábra - Copy-on-write memórialapok írást megelőzően

6.5.6. Memória foglalása

Az NT lapkezelést valósít meg, vagyis a memória allokálása memórialaponként (memory page) történik.

A memóriakezelő kevés szabad fizikai memória esetén háttértárra mentheti az egyes lapokat. Ezt a műveletet nevezzük *lapozásnak (paging*).

A folyamatok a memórialapokat foglalhatják a rendszer ún.

- lapozott memória tárából (paged memory pool) és a
- nem lapozott memória tárából (nonpaged memory pool).

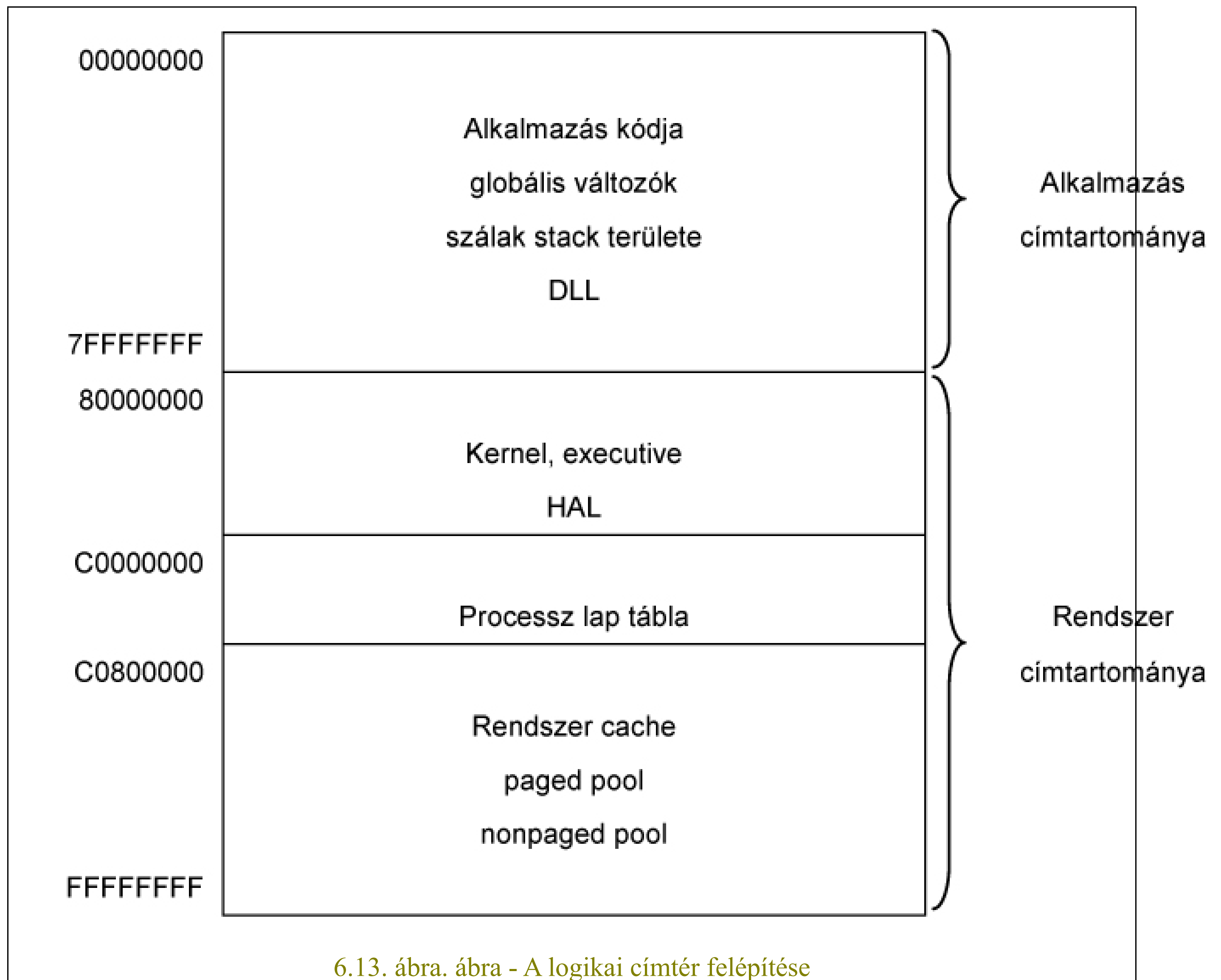
A két memóriatár közötti különbség a lapozásnál van. A nem lapozott memória tárából történt memóriafoglalás esetén a kért memórialapot a rendszer állandóan a fizikai memóriában tartja, sosem menti

háttértárra. Ezzel lehet biztosítani, hogy valamilyen szempontból fontos (például driverek által használt) memóriaterületek akkor is a fizikai memóriában maradjanak, ha ritkán használják őket.

6.5.7. A memória mérete

A Windows NT 32 bites memóriacímzést valósít meg, ami minden processz számára 4 GB virtuális címtartományt jelent. Alapesetben a 4 GB felső 2 GB-os részét használhatják az alkalmazások (00000000-7FFFFFFF címek), míg az alsó 2 GB (80000000-FFFFFFFF) az operációs rendszert tartalmazza. A Windows NT Server Enterprise Edition lehetőséget ad egy bootoláskor megadható paraméter használatával a user címtartomány 3 GB-ra történő növelésére.

A user, illetve a rendszer címtartomány felosztását, valamint a címtartományban a különböző memóriaterületek elosztását a 6.13. ábra mutatja.



6.13. ábra. ábra - A logikai címtér felépítése

Az NT memóriakezelőjének működése – például rendszer általa használt adatmezők méretének változtatásával, vagy a folyamatoknak biztosított memóriaterületek nagyságának módosításával – alkalmazkodik az adott számítógépében lévő fizikai memória méretéhez. Az NT három memória méretkategóriát különböztet meg:

- kicsi (small),
- közepes (medium),
- nagy (large).

A különböző kategóriákhoz tartozó memórianagyságot a 6.14. ábrán láthatjuk. (A memóriák árának csökkenése, így az általánosan használt memóriaméret növekedése ezeket a határokat viszonylag gyorsan megváltoztatja.)

Méretkategória	x86	Alpha
Kicsi	≤ 19 MB	≤ 31 MB
Közepes	20-32 MB	Nem használt
Nagy	≥ 32 MB (NT Workstation) ≥ 64 MB (NT Workstation)	≥ 32 MB (NT Workstation) ≥ 64 MB (NT Workstation)

6.14. ábra. ábra - A fizikai memória mérete különböző memóriamodellek esetén

6.5.8. Címtranszformáció

A Windows NT kétszintű indexelést használ a virtuális-fizikai címtranszformációhoz, amit mind az x86-os, mind az Alpha processzorok támogatnak.

A virtuális cím felépítése processzorfüggő. A 6.15. ábra mutatja, hogy x86-os processzor esetén hogyan áll össze a lap könyvtár indexből (PDI: Page Directory Index), laptábla indexből (PTI: Page Table Index), illetve a byte indexből (BI: Byte Index) a virtuális cím.

31		0 (LSB)
Laptábla könyvtár index (PDI)	Laptábla index (PTI)	Byte index (BI)
x86: 10 bit *Alpha:* 8 bit	*x86:* 10 bit *Alpha:* 11 bit	*x86:* 12 bit *Alpha:* 13 bit
Virtuális memórialap szám		eltolás

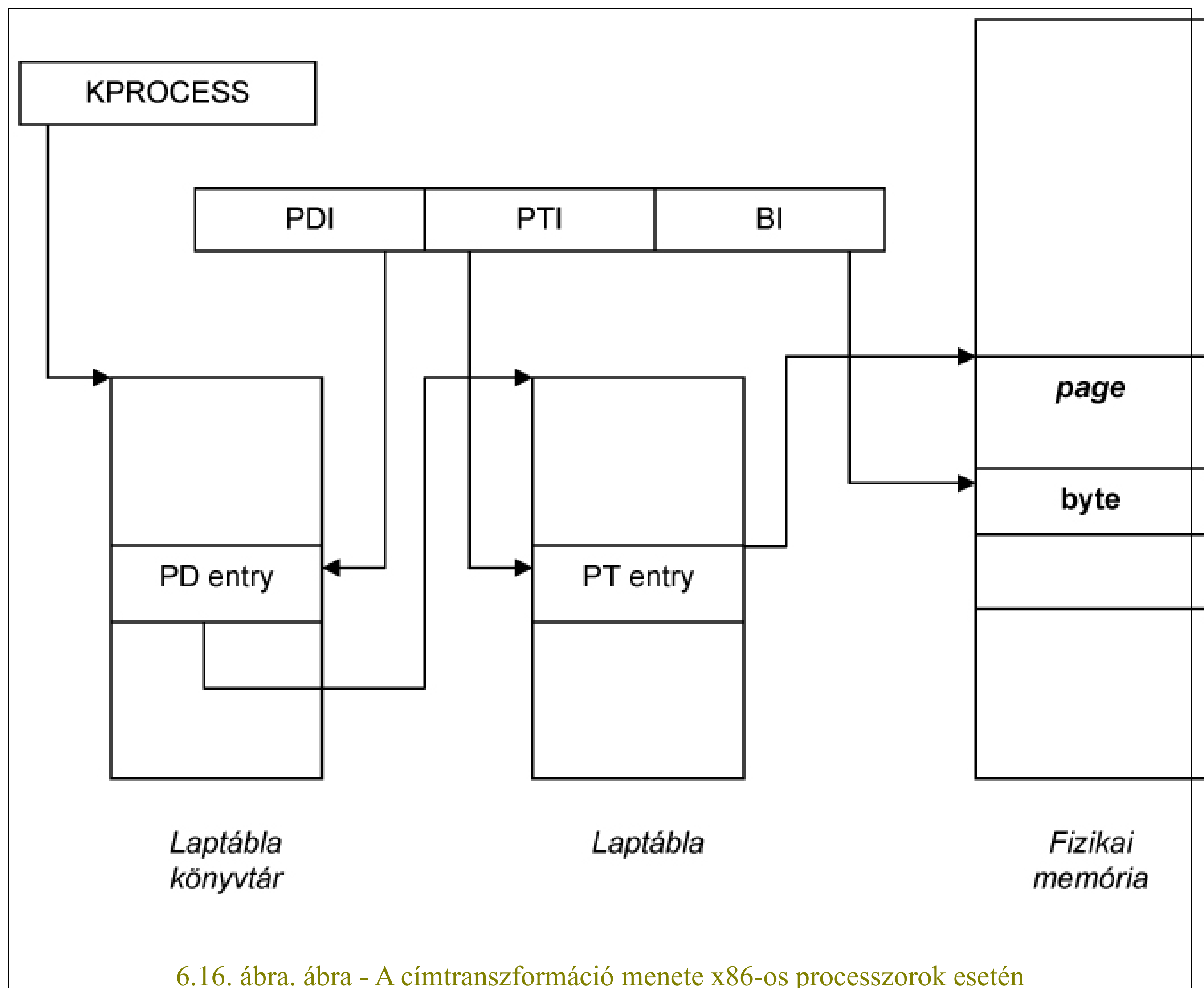
6.15. ábra. ábra - A virtuális cím felépítése

A PDI illetve a PTI kijelöli az elérendő memórialapot, míg a BI a lapon belüli eltolást adja meg.

A címtranszformáció menetét a 6.16. ábra mutatja be.

Minden folyamathoz tartozik egy lap könyvtár (page directory) maximum 1024 bejegyzéssel. Ennek kezdőcíme a folyamat leíróban (KPROCESS) van. A PDI kijelöli, hogy a táblázat melyik bejegyzésében találjuk a címtranszformációhoz használandó laptábla kezdőcímét. A laptábla egy maximum 1024 bejegyzést tartalmazó táblázat, melyből maximum 512 darab lehet egy folyamaton belül, ami egyben az egy rendszerben használható laptáblák maximális száma is.

Ebben a laptáblában a PTI kijelöli, hogy melyik bejegyzés tartalmazza a használandó memórialap kezdőcímét. A memórialapon belül pedig a BI pontosan kijelöli a keresett byte eltolását.



6.16. ábra. ábra - A címtranszformáció menete x86-os processzorok esetén

6.6. A Windows NT fájlrendszere (NTFS)

Ebben az alfejezetben a Windows NT fájlrendszerének belső felépítését és működését mutatjuk be. A Windows NT által használt fájlrendszer elterjedt rövid elnevezése: *NTFS (Windows NT File System)*.

6.6.1. Elvárások az NTFS-sel szemben

Tervezéskor az NTFS elé állított legfontosabb követelmények a következők voltak:

- Megbízható fájlrendszer, vagyis adatállományok sikertelen műveletek utáni visszaállíthatóságának, helyrehozhatóságának garantálása *(recoverability).*
- Állományok biztonságának, vagyis illetéktelen elérésektől történő védelmének garantálása *(security)*.
- Hibatűrés, redundáns tárolás lehetősége, vagyis az egyes fájlok elérhetőségének biztosítása *(fault tolerance)*.
- Nagy diszkek és nagy fájlok tárolásának lehetősége.

A megbízható fájlrendszer megvalósítását két alapvető eszközzel éri el a Windows NT:

- Tranzakciónkénti feldolgozás (transaction processing).
- Redundáns tárolása a fontos adatoknak.

A tranzakciónkénti feldolgozás a lemezműveletek biztonságos végrehajtásának elve, melyet a következő alfejezetben mutatunk be.

A redundáns tárolás azt jelenti, hogy a rendszer duplikálja a működés szempontjából fontos adatállományokat a lemezen. Igyekszik a lemez egymástól távoli régióiban tárolni a másolatokat, hogy csökkentse mindkét másolat egy időben történő sérülésének valószínűségét.

A fájlrendszer biztonságának garantálása érdekében a rendszer minden állomány-hozzáféréskor ellenőrzi az elérési jogosultságot. A fájlelérés ellenőrzése illeszkedik az NT általános objektumelérést ellenőrző biztonsági rendszerébe, amiről a későbbiekben még szólunk.

A nagy diszkek kezelését, illetve nagy fájlok tárolását az NT 64 bites *cluster-leíró* használatával támogatja. Egy fájlrendszerben összesen 2^{48} fájl lehet egyidőben. Egy fájl mérete maximum 2^{64} byte lehet. A fájlrendszer további korlátainak ismertetésére a tárolás részleteinek leírásakor még visszatérünk.

6.6.1.1. Tranzakciónkénti feldolgozás

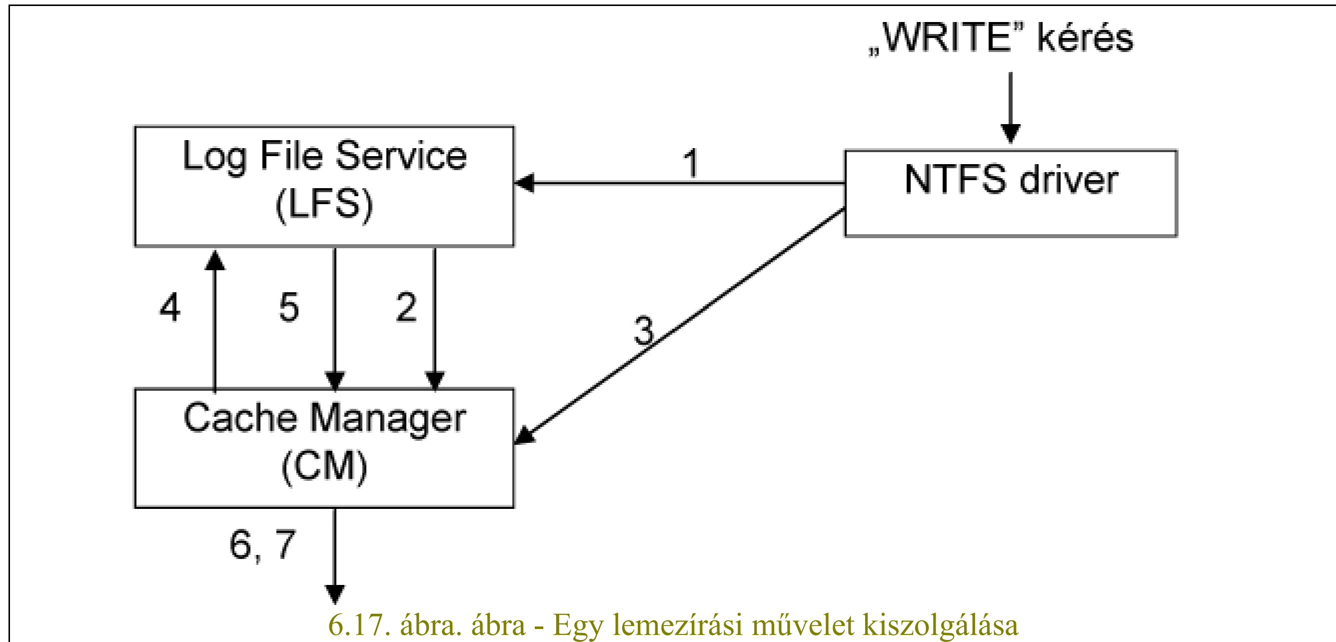
A tranzakciónkénti feldolgozás (műveletenkénti feldolgozás) a fájlhozzáférések megbízható végrehajtását teszi lehetővé.

A tranzakciónkénti feldolgozás a „Mindent vagy semmit” elvre épül. Ez esetünkben azt jelenti, hogy egy lemezművelet vagy teljesen végrehajtódik, vagy ha valamilyen oknál fogva megszakadna, a rendszer visszaállítja a fájlrendszer eredeti állapotát. Így mindig konzisztens fájlrendszer lesz a lemezen.

A rendszer az egyes fájlrendszeren végzett műveleteket lépésekre bontja, és kiegészíti azokat a megbízhatóságot növelő redundáns adminisztrációs lépésekkel. Ezek után meghatározza a lépések olyan

sorozatát, mely lehetővé teszi azt, hogy a lemezen minden pillanatban csak konzisztens adatok legyenek, illetve a lemezen lévő adatok alapján visszaállítható legyen egy-egy művelet megkezdése előtti állapot.

Egy fájlírási művelet során végrehajtott lépéseket a 6.17. ábra mutatja be:

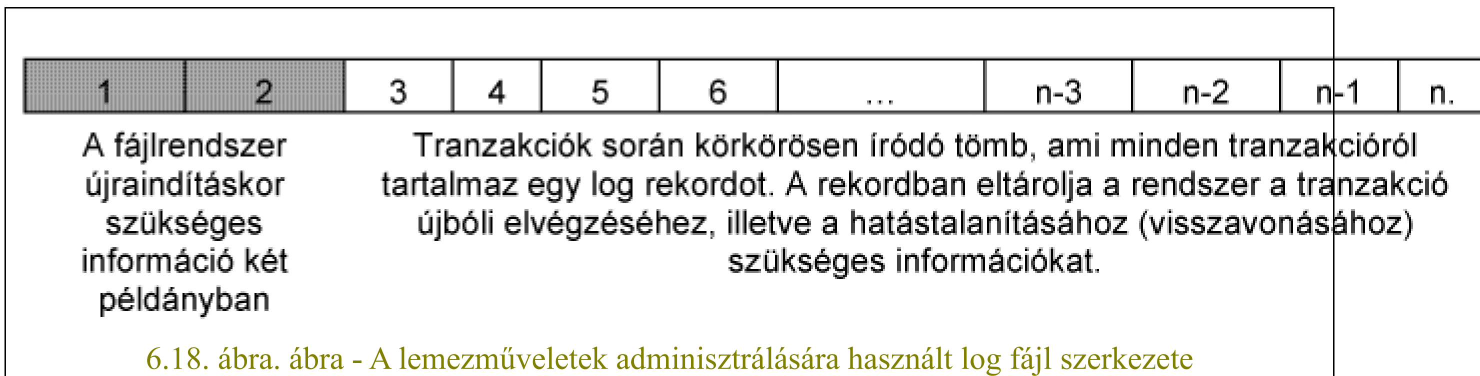


6.17. ábra. ábra - Egy lemezírási művelet kiszolgálása

A fájlírási művelet kiszolgálásának lépései:

- Az NTFS driver üzenetet küld az LFS-nek, hogy írás tranzakció következik, adminisztrálja azt (készítsen a tranzakcióról ún. log rekordot).
- Az LFS írja a cache-ben lévő log fájlt.
- Az NTFS végrehajtja a kért utasítást, írja a (cache-ben lévő) fájlt.
- A CM üzen, hogy az írás befejeződött, minden adat megvan.
- Az LFS megadja, hogy milyen adatokat kell a cache-ből üríteni. (A megváltoztatott fájlt és a log fájlt.)
- A CM kiírja a lemezre a log fájlt.
- A CM kiírja a lemezre az adatokat, vagyis a megváltoztatott fájlt.

A lemezen történt műveletek adminisztrálása úgy történik, hogy a rendszer a műveletek leírását, illetve a visszaállításukhoz szükséges adatokat egy ún. *log fájlba* írja. Ennek a Log File Service modul által használt fájlnak a szerkezetét a 6.18 ábra mutatja be.



6.18. ábra. ábra - A lemezműveletek adminisztrálására használt log fájl szerkezete

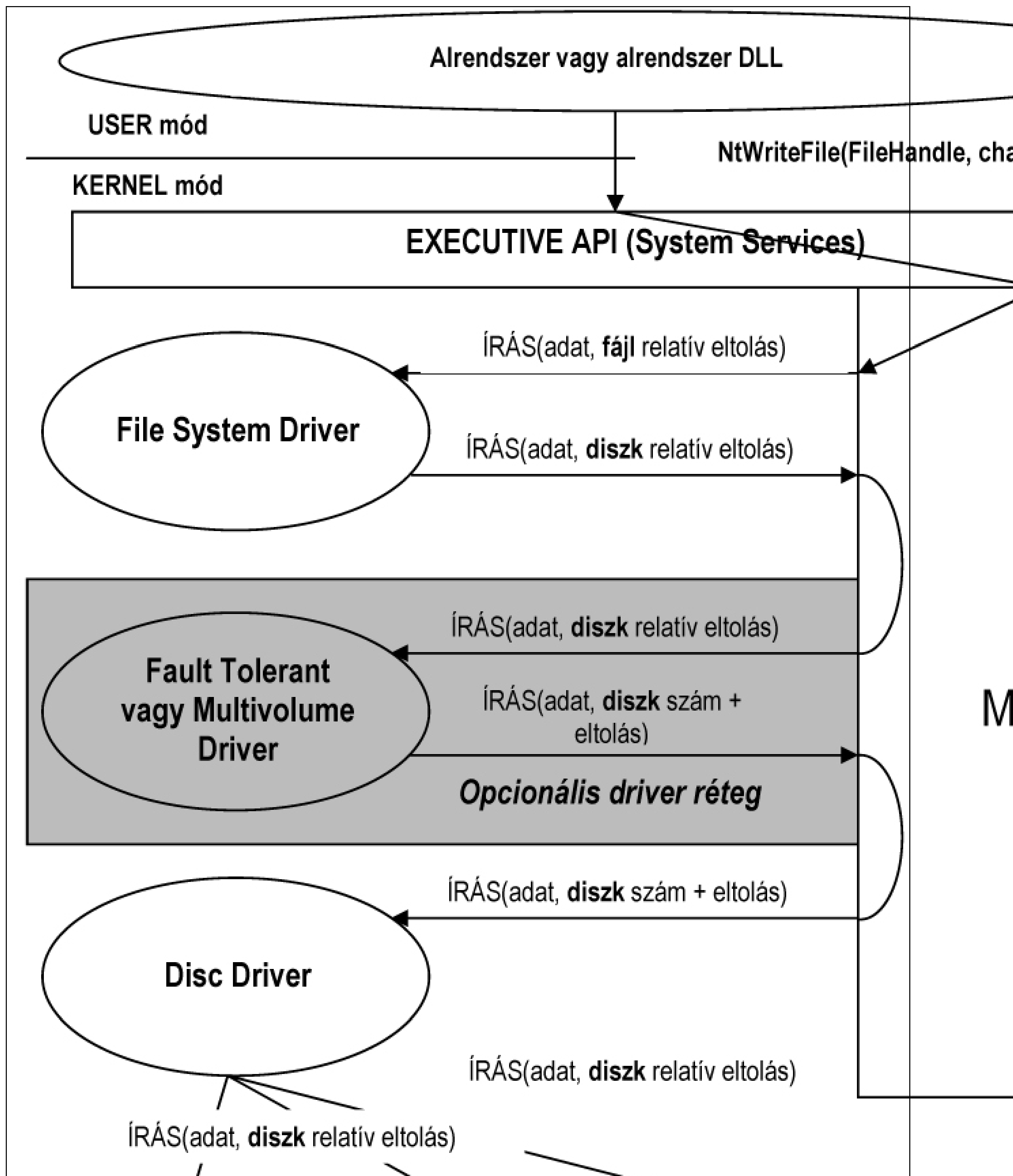
6.6.1.2. Réteg szerkezetű device driver struktúra

A réteg szerkezetű device driver struktúra lehetővé teszi a különböző funkcionalitású driverek rugalmas használatát, illetve a driverekből az igényeknek megfelelő rendszer építését. A módszer lényege, hogy a drivereknek különböző szintjeit különbözteti meg. A szintek funkcionalitása, illetve az adott funkcionalitáshoz tartozó interfész jól definiált. Emiatt a drivereket egyszerű egymáshoz illeszteni.

A réteg szerkezetű device driver struktúra lényegét legegyszerűbben egy példán keresztül érthetjük meg. A 6.19 ábrán egy lemez írási művelet végrehajtása látható. Az API-hívást az Executive réteg továbbítja az B/K Manager felé, amely eléri a Disc Drivereket. A File Manager Driver előállítja a fájl-handle és az adatbuffer alapján az írandó lemezen a megcímzendő cluster címét, illetve az írandó adatot.

Egyszerű esetben a File System Driver (az I/O Manageren keresztül) eléri a Disk Drivert, ami végrehajtja a lemez írását.

A réteg szerkezetű driverstruktúrának köszönhetően lehetőség van azonban további driverréteget a két szint közé illeszteni. Ebben az esetben a beillesztett driverréteg a File System Driver felé úgy viselkedik, mit egy egyszerű Disc Driver, de belül bonyolultabb többletszolgáltatásokat nyújtó funkciókat valósíthat meg. Megvalósíthat például hibatűrő *(redundáns)* tárolást (például lemez tükrözést) vagy megvalósíthat több lemezen tárolt *(multivolume file system)* fájlrendszert, ami lehetővé teszi különösen nagy fájlok tárolását.



Alrendszer vagy alrendszer DLL
USER mód
NtWriteFile(FileHandle, cha
KERNEL mód
EXECUTIVE API (System Services)
ÍRÁS(adat, fájl relatív eltolás)
File System Driver
ÍRÁS(adat, diszk relatív eltolás)
ÍRÁS(adat, diszk relatív eltolás)
Fault Tolerant vagy Multivolume Driver
ÍRÁS(adat, diszk szám + eltolás)
Opcionális driver réteg
M
ÍRÁS(adat, diszk szám + eltolás)
Disc Driver
ÍRÁS(adat, diszk relatív eltolás)
ÍRÁS(adat, diszk relatív eltolás)

6.19. ábra. ábra - Réteg szerkezetű device driver struktúra

Az NT által támogatott hibatűrő adattárolási technikák:

- RAID level 1: diszkek tükrözése,
- RAID level 5: diszkszeletek prioritásos védelme.

A RAID a *Redundant Array of Inexpensive Disks* (olcsó lemezegységek redundáns tára) rövidítése. Ezek a hibatűrő tárolást kommersz elemekkel lehetővé tevő módszerek szabványban definiáltak.

6.6.2. Az NTFS további előnyös tulajdonságai

A fenti tulajdonságokon kívül az NTFS további előnyös tulajdonságokkal rendelkezik. Ezek a következők:

- Streamek használata. Egy adott néven elérhető adatállománynak több függetlenül elérhető alszekciója (streamje) lehet. Ezeket külön-külön írhatjuk és olvashatjuk.
- UNICODE használható a fájlnevek megadásakor. A UNICODE használatán túl lehetőség van maximum 255 karakter hosszú fájlnevek választására, ami szóközöket, illetve pontokat is tartalmazhat.
- Indexelés lehetősége. Egy fájlrendszerben készíthetünk egy ún. index buffert, ami egy adott tulajdonság alapján tartalmaz közvetlen hivatkozásokat fájlokra.
- Dinamikus hibás *(bad)* szektorkezelés. A rendszer futás közben észleli, ha egy cluster meghibásodik, és utána többet nem használja. Hibatűrő fájlrendszer esetén akár el is fedheti a rendszer az adott lemezhibát.
- POSIX-szabvány támogatása:

– hard link

– kisbetűt és nagybetűt megkülönböztethető *(case sensitive)* fájlnevek

– időbélyegek *(time stampek)* használata.

6.6.3. Az NTFS által használt adattípusok, adatszerkezetek

Kötet (volume)

A lemez egy logikai partícióját nevezzük kötetnek. Minden kötethez egy logikai fájlrendszerhez tartozik, ami formázáskor alakul ki. Egy lemezen így lehet akár több, akár eltérő típusú (például FAT és NTFS) volume is.

Cluster

Az adattárolás alapegysége (lásd blokk). Néhány egymás után következő szektor, melyeket a rendszer együtt kezel. Az NT csak clustereket tart nyilván.

Logical Cluster Number (LCN)

Egy adatszerkezethez, például egy fájlhoz tartozó clusterek sorszáma. Egy fájlhoz tartozó clustereket például 0-tól n-ig folyamatosan számozzuk.

Virtual Cluster Number (VCN)

A lemezen elhelyezkedő clusterek azonosítására szolgáló sorszám. A rendszer egy lemezt a clusterek sorozatának lát. A lemezmeghajtó feladata, hogy a VCN alapján megtaláljon és elérjen egy adott a VCN-hez tartozó clustert.

A lemezen tárolt adatszerkezetek eléréséhez pontosan az LCN-ek VCN-ekhez történő hozzárendelését kell megadni, vagyis azt, hogy a fájl egy adott LCN-el jelölt clustere, melyik disk clusterben tárolódik, vagyis melyik VCN-el érhető el. Mindezt a 6.20. ábrán szemléltetjük.

LCN:	0	1	2	…	n.
VCN:	6324	857243	9454	…	542

6.20. ábra. ábra - LCN és VCN egymáshoz rendelése

NTFS metadata

Azoknak az adatoknak a gyűjtőneve, amelyek egy fájlrendszer kezeléséhez, illetve a benne tárolt fájlok eléréshez szükségesek. Az NTFS tartja magát ahhoz az elvhez, hogy minden, ami a lemezen van, az fájl. Így például a volume-leíró, a boot információ, a hibás szektorok leírása stb. mind-mind egy-egy fájlként van eltárolva.

Az NTFS metadata információkat tároló fájlokat a 6.21. ábra mutatja be.

$Boot fájl	A rendszer indulásakor használt információ.
$Bad sector	A hibás clusterek sorszáma (LCN).
$Bitmap	A clusterek foglaltsági térképe.
$Logfile	A Log File Service által a lemeztranzakciók adminisztrálására használt fájl.
$MFT	Master File Table.
$MFT mirror	A Master File Table (részleges) biztonsági másolata.
$Volume	A kötet leírását (típusát, fájlrendszer stb.) tartalmazó fájl.
$Attribute	A kötet tulajdonságait, attribútumait tartalmazó fájl.

6.21. ábra. ábra - NTFS metadata információk

MFT (Master File Table)

A Master File Table a fájlrendszerben tárolt fájlok leírását, elérésükhöz szükséges információt tartalmazza. Az NT szemlélete a következő.

Egy fájl nem más, mint egymással összerendelt adatok halmaza. A fájl neve, keletkezési időpontja, elérhetősége stb. mind-mind egy-egy adatokkal jellemzett „attribútuma" az adott fájlnak. Maguk a fájlban tárolt adatok is a fájl tartalom nevű attribútumának leírása.

A fájlok az ún. file rekordokban tárolódnak, amelyben a fájl attribútumok (file attribute) azonosítója (neve) és az attribútumhoz tartozó adatmezők van egymás után felsorolva.

A Master File Table nem más, mint file rekordok sorozata. A tárolás megkönnyítése érdekében a Mastar File Table nem az egész file rekordot, hanem csak annak első 1K-s darabját tartalmazza. A későbbiekben majd meglátjuk, milyen technikával tárolja az NT az 1K-nál nagyobb file rekordokat, illetve a rekordokban levő attribútumokat.

Ha végiggondoljuk az NT tárolási technikáját, akkor látjuk, hogy a Master File Table nem más, mint egy 1K-s bejegyzéseket tartalmazó táblázat. A táblázat minden bejegyzése egy-egy fájlt azonosít. A fájl egyedi azonosítója ezek után az a sorszám lesz, ami megmondja, hogy az MFT hányadik bejegyzése tartozik hozzá. A bejegyzések minden fájlra vonatkozó információt tartalmaznak, azonban szerkezetük nem kötött. A rekordok szerkezetének leírását, vagyis az egyes attribútumok tárolásának módját későbbi fejezetben ismertetjük.

A Master File Table első 16 bejegyzése mindig azonos fájlokhoz tartozik. Ezek az ún. rendszerfájlok a fájlrendszer számára fontos adatokat tartalmazzák. Nevük mindig $ jellel kezdődik. A Master File Table szerkezetét a 6.22. ábra tünteti fel:

Az ábrán látható, hogy az első 16 bejegyzést az operációs rendszer használja, utána pedig a felhasználói fájlok következnek.

$ MFT	0.
$ MFT mirror	1.
$ Logfile	2.
$ Volume	3.
$ Attrib	4.
…	…
$...	15.
USER FILE1	16.
USER FILE2	17.
…	…
USER FILEn	n.

6.22. ábra. ábra - A virtuális cím felépítése

Fontos megjegyezni, hogy a könyvtárak leírása is egyszerű felhasználói fájlokban tárolódik. Ebben az adott könyvtárban levő fájlok neveinek és a fájlokhoz tartozó MFT-bejegyzés sorszámának összerendelése található. Ennek megfelelően az egyik felhasználói fájl például a főkönyvtár leírását fogja tartalmazni. Ennek helyét azonban a fix helyen elérhető rendszerfájlok tartalmazzák.

6.6.4. Fájlok elérése NTFS alatt

Egy lemeztranzakciót az NT következő módon hajt végre. A lépések többségét csak a kötet rendszerindulás utáni első elérésekor kell végrehajtani.

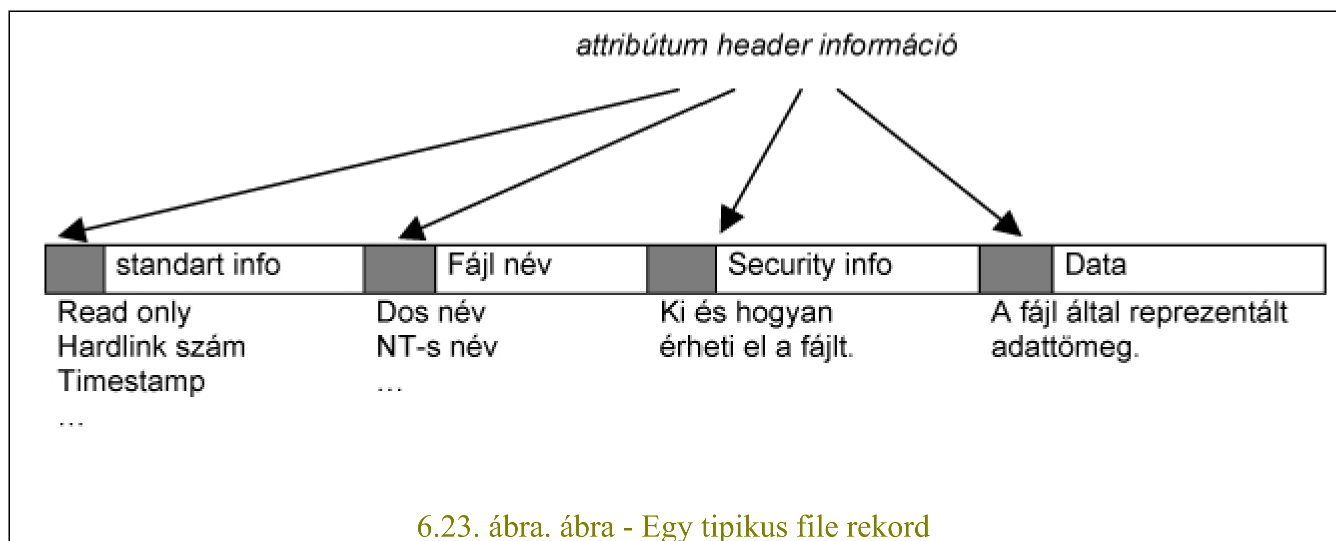
1. A $Boot fájl elérése, amely rögzített helyen van.
2. $MFT elérése. A $Boot tartalmazza az MFT kezdetének helyét. Az $MFT mindig az első MFT rekordhoz tartozik, $MFTMirr a másodikhoz. $LogFile a harmadikhoz stb.
3. $MFT ($MFTMirr) elérése, memóriába tárolása.
4. $LogFile és más meta-fájlok beolvasása.
5. A kötet (volume) rendszerindulás (boot) utáni első elérésekor végrehajtja az ún. recovery műveletet. A recovery nem más, mint a fájlrendszer konzisztens állapotának ellenőrzése, szükség esetén annak helyreállítása. A log fájl alapján ellenőrzi a rendszer, hogy kell-e tranzakciót „visszagörgetni” vagy újra végrehajtani. Recovery után a lemez konzisztens állapotba kerül.

6. A root „\” könyvtárhoz tartozó MFT-bejegyzés megkeresése, illetve a memóriában tárolása a későbbi elérések gyorsítása érdekében.
7. A tényleges fájltranzakció végrehajtása. Természetesen a log fájl írása minden tranzakciónál folyamatosan történik.

6.6.5. File ReKord

Mint már említettük, a file rekord tartalmazza a fájlhoz tartozó összes információt. Az adatok attribútumok formájában tárolódnak. Egy tipikus attribútum halmazt mutat be a 6.23. ábra.

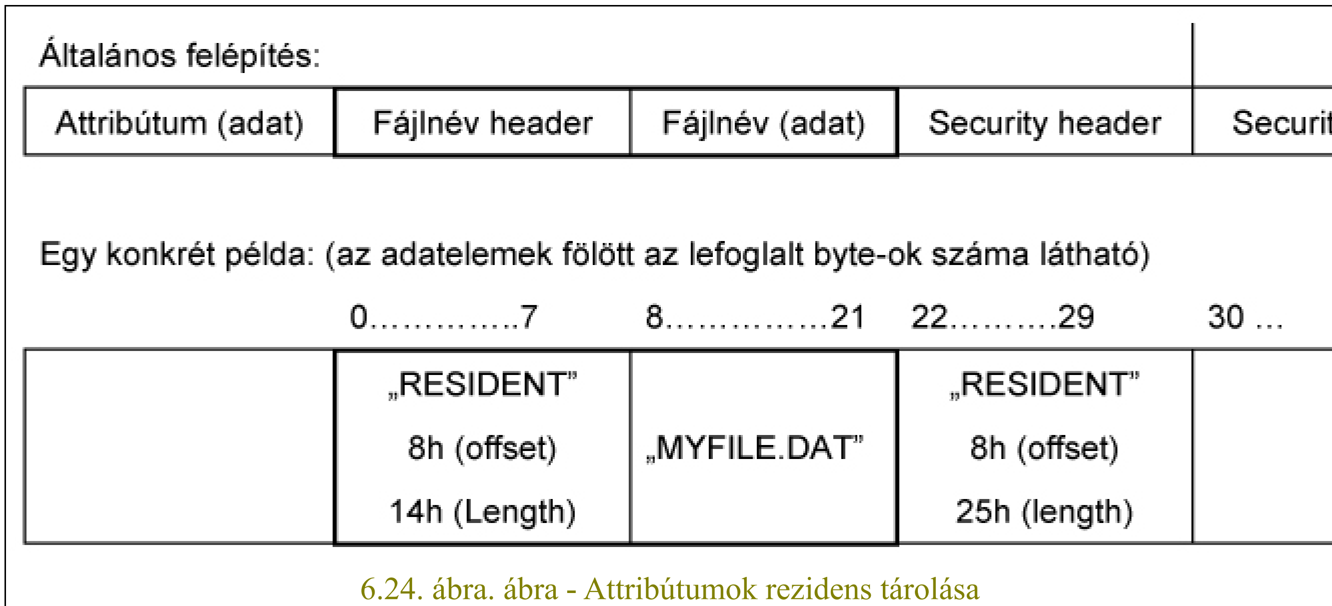
Az attribútumok értéke tárolódhat rezidens és nem rezidens módon attól függően, hogy fizikailag hol helyezkedik el az attribútumot jellemző adathalmaz. Ha közvetlenül az attribútum header után tárolódik, akkor rezidens tárolásról beszélünk. Nem rezidens tárolás esetén az attribútum után csak egy hivatkozást találunk, hogy a lemez mely clustere tartalmazza az attribútum adatait.



6.23. ábra. ábra - Egy tipikus file rekord

6.6.5.1. Rezidens tárolás

Az attribútum értékét reprezentáló bináris információ közvetlenül az attribútum header után van tárolva a rekordban. Erre vonatkozóan nézzünk egy példát a 6.24. ábrán:



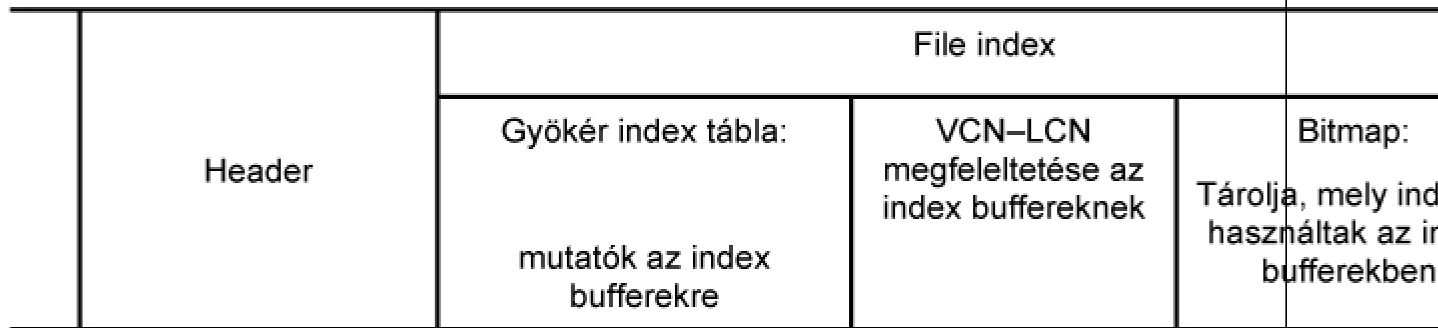
6.24. ábra. ábra - Attribútumok rezidens tárolása

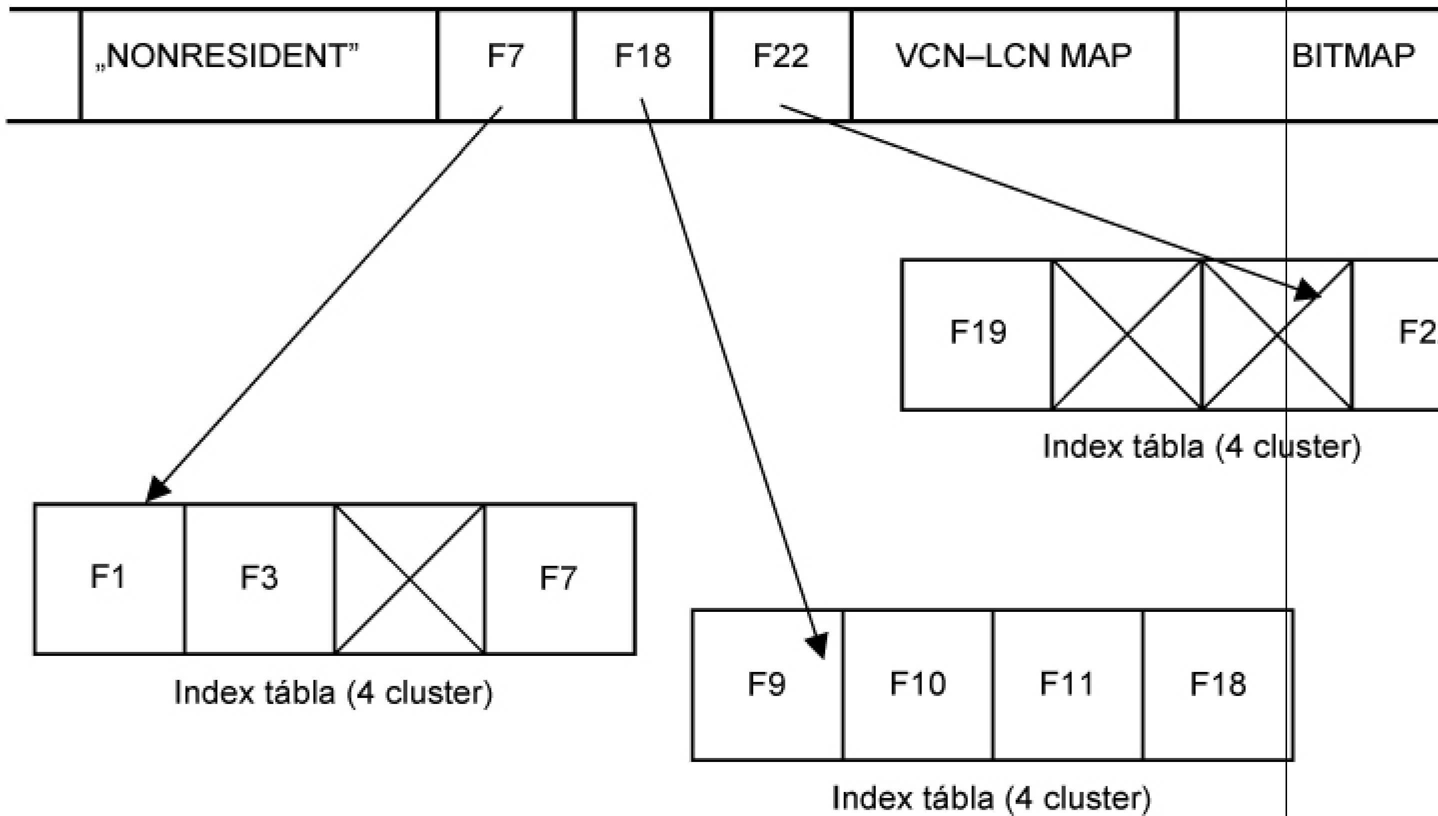
6.6.5.2. Nem rezidens tárolás

A header után csak az attribútum értékét reprezentáló adatok helye van rögzítve. Például az adatokat tartalmazó buffer címeit tárolja a rendszer a rekordban. Például egy könyvtár nem rezidens tárolása a 6.25. ábra szerint néz ki (kétszintű indexelés, dinamikus méretű indextáblával).

A nem rezidens tárolás esetén a header jelzi a nem rezidens tárolás tényét. A fájl rekordba ezután egy indextábla kerül, ami megmutatja, mely bufferek-ben tárolódik az adott attribútum értéke. Ezután egy VCN–LCN megfeleltetési táblázat következik, amelynek segítségével az attribútumok értékét tároló bufferek közvetlenül elérhetők. Az attribútum leírás végén egy táblázat következik, mely megmutatja, mely clusterek kihasználtak és mely clusterek nem használtak a bufferekben.

Az adatok fent leírt módon történő tárolását nevezzük dinamikusan növekvő indextáblát használó egyszintű indexelésnek.





6.25. ábra. ábra - Attribútumok nem rezidens tárolása

6.7. Biztonsági alrendszer

A Windows NT olyan átfogó és konfigurálható biztonsági szolgáltatásokat nyújt, amelyek együttesen kielégítik az USA Védelmi Minisztériuma által a megbízható operációs rendszerekre előírt C2-es szintű biztonsági követelményeket. A Windows NT Server és a Windows NT Workstation 3.51 1996. októberben, a Windows NT 4.0 1999. márciusban kapta meg a C2-es biztonsági jóváhagyást.

A biztonsági szolgáltatások és a szükséges alapvető tulajdonságaik a következők:

- *A biztonságos logon (bejelentkezés) lehetősége* megköveteli, hogy a felhasználó egyedi logon azonosítóval és jelszóval azonosítsa magát bejelentkezéskor.
- *A tetszőlegesen konfigurálható (discretionary) elérési ellenőrzés* lehetővé teszi, hogy egy erőforrás tulajdonosa meghatározza, hogy ki érheti el az erőforrást, és mit tehet vele. Így olyan jogosultságok adhatók, amelyek egy felhasználónak vagy egy felhasználói csoportnak engedélyeznek különböző fajta hozzáférést.
- *A biztonsági auditálás* azt a képességet jelenti, amivel a biztonságot érintő események felismerése és feljegyzése történik, beleértve a rendszer-erőforrások létrehozására, elérésére vagy törlésére irányuló lépéseket. Mindez megkönnyíti az illetéktelen akciót végrehajtó személy kinyomozását.
- *A memóriavédelem* megakadályozza, hogy egy jogosulatlan processz egy másik folyamat privát virtuális memóriáját elérhesse. Ezen túlmenően a Windows NT garantálja, hogy amikor egy memóriaoldal hozzárendelődik egy felhasználói folyamathoz, ebbe az oldalba soha nem kerülhetnek bele adatok másik folyamattól.

A fenti követelmények teljesítése a Windows NT biztonsági alrendszerén és az ahhoz kapcsolódó komponenseken keresztül valósul meg.

6.7.1. A biztonsági alrendszer komponensei

A Windows NT biztonságát megvalósító legfontosabb komponensek a következők:

- *Biztonsági referencia monitor (SRM: Security Reference Monitor)*. Ez a komponens az executive-ban van *(NTOSKRNL.EXE)*. Funkciói: az objektumok biztonsági elérésének ellenőrzése, privilégiumok (felhasználói jogok) kezelése, biztonsági auditálási üzenek előállítása.
- *Helyi biztonsági jogosultság ellenőrző (LSA: Local Security Authority) szolgáltatása.* Ez egy felhasználói módú folyamat, az *LSASS.EXE* image-et futtatja, ami a felhasználók bejelentkezési jogait és jelszavát ellenőrzi, a felhasználóknak és csoportoknak adott privilégiumokat, továbbá a biztonsággal kapcsolatos auditálás üzeneteit küldi az események naplójába.
- *LSA adatbázis.* Ez az adatbázis tartalmazza a rendszer biztonságos működésére vonatkozó beállításokat.
- *Biztonsági témaszám kezelő (SAM: Security Accounts Manager)* szerver. Ez egy szubrutinok halmazából álló szolgáltató. A szubrutinok a felhasználói neveket és a csoportokat tartalmazó adatbázist kezelik. Ezek a nevek és csoportok vagy a helyi gépre vannak definiálva, vagy – ha a gép egyben *tartomány (domain)* vezérlő – az adott domainre. Utóbbi esetben a rendszer tartományvezérlő. A SAM az LSASS folyamat környezetében fut.

- *SAM adatbázis.* Adatbázis, ami tartalmazza a definiált felhasználókat és csoportjaikat, jelszavukkal és egyéb attribútumaikkal együtt.
- *Logon processz.* Felhasználói módú folyamat, amely a *WINLOGON.EXE*-t futtatja. A folyamat a felhasználói nevet és jelszót veszi át, majd elküldi őket az LSA-hoz ellenőrzés céljából, ezután pedig a kezdeti folyamatot hozza létre.
- *Hálózati logon szolgáltatás.* Felhasználói módú szolgáltatás a *SERVICES.EXE* processzen belül, amely a hálózati logon kérésekre válaszol. A jogosultságot úgy kezeli, mint a helyi logonokat, azáltal, hogy ellenőrzés végett ezeket is az LSA processzhez küldi el.

Az SRM, ami kernel módban fut, és az LSA, ami felhasználói módban fut, egymás között a helyi eljáráshívás (LPC) révén kommunikálnak.

6.7.2. Az objektumok védelme

Az objektumvédelem a konfigurálható elérési ellenőrzés és az auditálás alapját képezi. A Windows NT-ben védhető objektumok a következők lehetnek: fájlok, hardver eszközök, postázási helyek, elnevezett és névtelen csővezetékek, folyamatok, szálak, események, szemaforok, időmérők, elérési tokenek, ablak-állomások, desktopok, hálózati megosztások, szolgáltatások, illetve nyomtatók.

Mivel a rendszer felhasználói szálba exportált erőforrásai objektumként vannak megvalósítva, a Windows NT objektumkezelője kulcsszerepet kap a biztonsági elérések szervezése során. Annak érdekében, hogy ellenőrizni lehessen, ki manipulálhat egy objektumot, a biztonsági rendszernek mindegyik felhasználó kilétéről tudnia kell. Ez az oka annak, hogy a Windows NT megköveteli a hitelesített logont, mielőtt a felhasználó bármelyik rendszer-erőforráshoz hozzáférne. Amikor egy szál megnyit egy objektumhoz tartozó handle-t, az objektumkezelő és a biztonsági rendszer a hívó biztonsági azonosítóját használja fel annak eldöntésére, hogy a hívó megkaphatja-e a kért handle-t.

A védelemben részesítendő objektumok ún. *biztonsági adatokkal (security descriptors)* rendelkeznek. Ezek szabják meg, hogy kinek milyen jellegű hozzáférési joga van az objektumhoz. A főbb biztonsági adatok a következők:

- a tulajdonos biztonsági azonosítója (security ID, SID),
- a felhasználó csoport biztonsági azonosítója (grup security ID; ezt csak a POSIX használja),
- konfigurálható elérési lista Azt specifikálja, hogy kinek milyen fajta elérése van az objektumhoz
- rendszerelérési lista

Azt specifikálja, hogy mely felhasználók mely műveleteit kell felsorolni a biztonsági auditálási naplóba.

Egy *elérési token (access token)* az az adatstruktúra, amely egy folyamat vagy egy szál biztonsági adatait tartalmazza: a biztonsági azonosítót, azon csoportok listáját, amelyeknek a felhasználó a tagja, valamint a

megengedett és letiltott privilégiumok listáját. Mivel az elérési tokenek felhasználói módba vannak exportálva, számos Win32 függvény hoz létre elérési tokeneket, illetve változtat rajtuk.

Mindegyik folyamatnak van egy elsődleges elérési tokenje, amelyet az őt előállító folyamattól örököl. Logonnál az LSA folyamat győződik meg arról, hogy a felhasználói név és a jelszó összhangban van-e az SAM adatbázisában levő információval. Ha igen, akkor az LSA logon folyamatnak visszaad egy elérési tokent a WinLogon, amely ekkor ezt a tokent hozzárendeli a kezdeti folyamathoz a felhasználói területen. A felhasználói területen létrejövő további folyamatok ezt az elérési tokent öröklik.

Az egyes szálaknak szintén lehet saját elérési tokenjük. Ez akkor van így, ha azok egy klienset *személyesítenek meg*. Ez a képesség lehetővé teszi a szálaknál, hogy olyan elérési tokenjük legyen, ami különbözik a folyamat elérési tokenjétől. Például, a szerver folyamatok tipikusan kliens folyamatokat személyesítenek meg úgy, hogy egy szerver folyamat egy kliensnek a nevében hajt végre műveleteket, a kliens biztonsági adatait használva a saját adatai helyett.

6.7.3. A biztonsági auditálás

Az objektumkezelő auditálási eseményeket tud generálni egy-egy elérési ellenőrzés eredményeként. A Win32-es alkalmazások számára elérhetők olyan API-függvények, melyek segítségével auditálási eseményeket közvetlenül is képesek generálni. A kernel módú kód mindig generálhat auditálási eseményt. Ugyanakkor pedig azok a folyamatok, amelyek auditálási rendszerszolgáltatást hívnak, a *SeAuditPrivilege* privilégiummal kell rendelkezzenek, hogy sikeresen generálhassanak egy auditálási rekordot. Ez a követelmény megakadályozza azt, hogy egy rosszindulatú felhasználói módú program elrontsa a biztonsági naplózást.

Az auditálási rekordok a megérkezésük után azonnal bekerülnek az LSA-nak küldendő rekordok sorába, a rendszer nem várakozik, hogy csomagokban küldje azokat tovább. Az auditálási rekordok az SRM-ről kétféle módon kerülnek át a biztonsági alrendszerhez. Ha a rekord kicsi (kisebb, mint a maximális LPC üzenetméret), akkor LPC üzenetként lesz elküldve. Az auditálási rekordok az SRM címteréből az LSA folyamat címterébe másolódnak át. Ha a rekord nagy, az SRM megosztott memóriát használ, hogy elérhetővé tegye az üzenetet az LSA számára. Ilyenkor egyszerűen csak egy pointert továbbít egy LPC üzenetben.

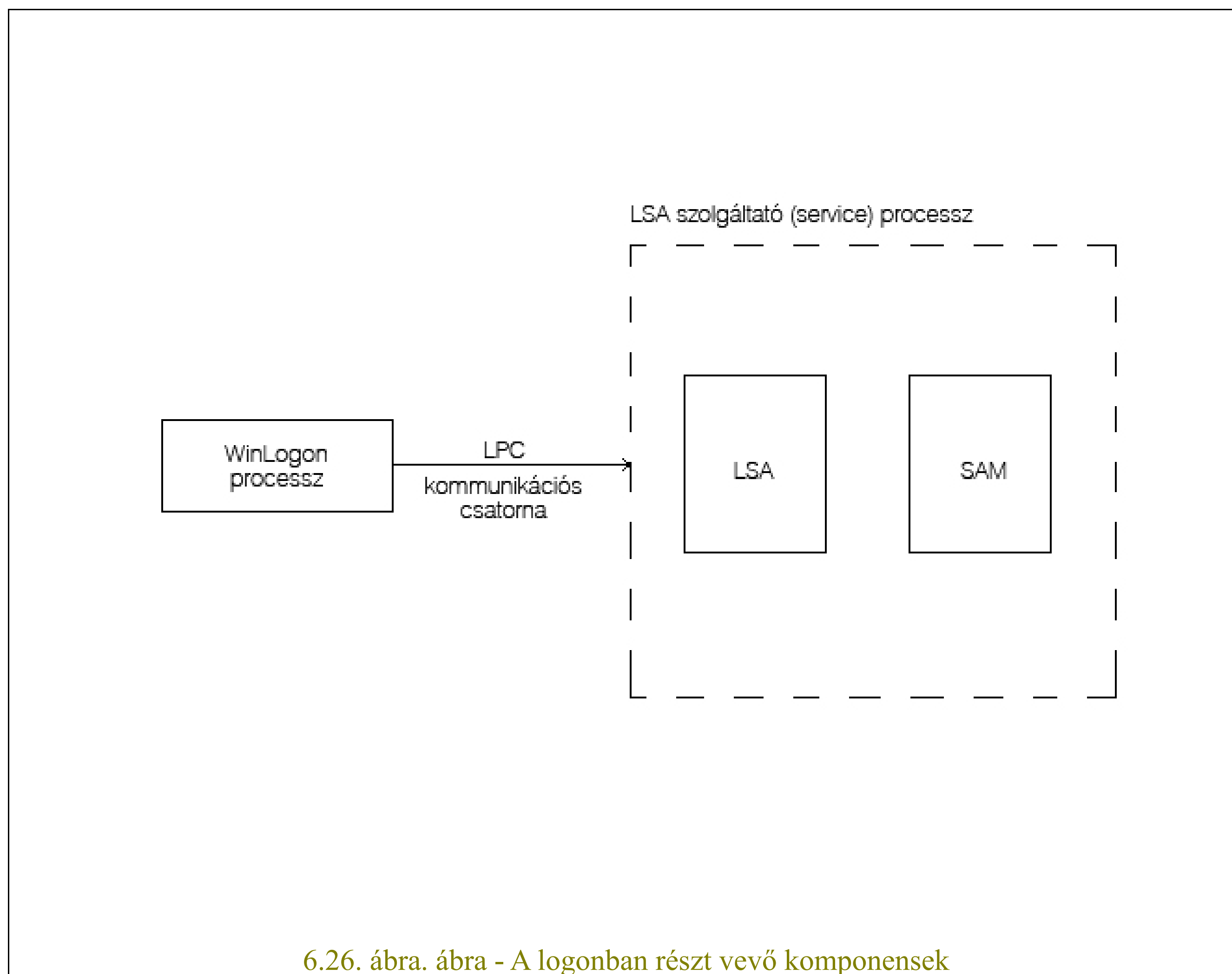
6.7.4. A logon

A logon (bejelentkezés) a *WinLogon* processz, az LSA, egy vagy több hitelesítő rutin, valamint az SAM közreműködésével megy végbe. A hitelesítő rutinok olyan DLL-ek, amelyek jogosultsági ellenőrzéseket hajtanak végre.

A WinLogon egy megbízható folyamat, amely a biztonsággal kapcsolatos felhasználói beavatkozásokat kezeli. Koordinálja a logont, kezeli a logoffot, illetve olyan műveleteket irányít, mint a jelszavak bevitele logonnál, jelszavak megváltoztatása, a munkaállomás lezárása és a lezárás megszüntetése. A WinLogon processznek kell biztosítania azt, hogy a biztonsággal kapcsolatos műveletek ne legyenek láthatóak semelyik más aktív folyamat

számára. Például, a WinLogon garantálja, hogy egy megbízhatatlan folyamat nem juthat hozzá egy desktop vezérléséhez ezen műveletek során, és ezáltal nem érheti el a jelszót.

A WinLogon az egyetlen folyamat, amely logon kérést fogad a billentyűzetről. Ekkor hívja az LSA-t, hogy az ellenőrizze a felhasználó jogosultságát a bejelentkezésre. Ha a jogosultság fennáll, a logon folyamat egy logon shellt aktivizál a felhasználó számára. A logonban részt vevő komponensek közötti kapcsolatot a 6.26. ábra mutatja be.



6.26. ábra. ábra - A logonban részt vevő komponensek

A rendszer inicializálásakor, amikor még egyik felhasználói alkalmazás sem aktivizálódott, a WinLogon olyan lépéseket hajt végre, amiknek eredményeként nála lesz a munkaállomás vezérlése, amikor a rendszer már kész kiszolgálni a felhasználót.

Ide tartozik például az, hogy létrehoz és megnyit három desktopot: egy alkalmazási desktopot, egy WinLogon desktopot és egy képernyő-mentési desktopot. A Winlogon desktop biztonságát az teremti meg, hogy ezt a desktopot egyedül csak a WinLogon képes elérni. A másik két desktop elérése a WinLogonon kívül a felhasználók számára is megengedett. Ez az elrendezés azzal jár, hogy ha a WinLogon desktop aktív, akkor

semmilyen más folyamat nem férhet hozzá olyan aktív kódhoz vagy adathoz, amely ezzel a desktoppal van összefüggésben. A Windows NT ezt a megoldást arra használja fel, hogy védje azoknak a műveleteknek a biztonságát, amelyek a jelszó kezelésére, valamint a desktop zárására és a zárás feloldására vonatkoznak.

Miután a WinLogon desktop létrejött az inicializálás során, ő lesz az aktív desktop. Amikor a WinLogon desktop aktív, akkor mindig lezárt állapotban van. A WinLogon csak akkor oldja fel a lezárást, amikor átkapcsol az alkalmazási desktopra, vagy a képernyő-mentési desktopra. Ehhez még megjegyzendő, hogy egyedül csak a WinLogon folyamat képes egy desktopot lezárni vagy a lezárást feloldani.

7.Kérdések, feladatok

Tartalom

7.1. Bevezetés
7.2. Az operációs rendszer mint absztrakt, virtuális gép
7.3. Multiprogramozott operációs rendszerek
7.4. Hálózati és elosztott rendszerek
7.5. UNIX
7.6. Windows NT operációs rendszer

7.1. Bevezetés

1. Melyek egy operációs rendszer alapfeladatai?

2. Mi a virtuális gép koncepció lényege?

3. Melyek igaz állítások?

Az operációs rendszerek szerkezeténél megismert virtuális gép *(virtual machine)* megközelítés előnye, hogy

- a virtuális gép a valódi hardvernél megbízhatóbb, védettebb működést biztosít.
- a virtuális gép segítségével egyidejűleg több, különböző operációs rendszer is futtatható anélkül, hogy ezeken változtatni kellene.
- a virtuális gép működését a rajta futtatott operációs rendszer menet közben is szabadon megváltoztathatja.

4. Sorolja fel, hogy a korai operációs rendszerek milyen módszereket használtak a perifériás műveletek lassúságának ellensúlyozására.

5. Ismertesse a korai operációs rendszerek történeténél megismert batch-rendszerek működését. Milyen elvi és gyakorlati HW-, illetve SW-fejlesztések kötődnek ehhez a korhoz?

6. Mit nevezünk spoolingnak? Miért vezet ez a módszer a számítógép jobb kihasználásához? Mondjon példát arra, hol használnak spoolingot a korszerű operációs rendszerekben.

7. Melyik állítás igaz a spooling rendszereknél?

- A spoolinghoz független számítógépek szükségesek, amelyek a lassú periféria és a központi számítógép közötti átvitelt intézik.
- A spooling működéséhez a lemez perifériának közvetlen táreléréssel (DMA) kell működnie.
- A spoolingnál különböző munkák feldolgozása és perifériás műveletei történhetnek egymással átlapoltan.

8. Melyek igaz állítások?

Minden időosztásos *(time-sharing)* operációs rendszer egyben

- valósidejű *(real-time)* is.
- elosztott *(distributed)* is.
- multiprogramozott is.

9. Mi a hasonlóság és mi a különbség az operációs rendszerek történeténél megismert pufferelt adatátvitel és az ún. spooling-módszer között? Mi a kapcsolat a jelenlegi operációs rendszerekben a multiprogramozás és a spooling között?

10. Mit jelent az operációs rendszereknél a multiprocesszálás fogalma?

11. Definiálja a valósidejű *(real-time)* operációs rendszer fogalmát. Ilyen-e az „általános” UNIX-rendszer?

12. Mit jelent a korszerű operációs rendszerekben a batch-feldolgozás?

13. Melyek az operációs rendszer fő környezeti kapcsolatai?

14. Milyen csatlakozási felületekkel rendelkezik egy operációs rendszer?

15. Hogyan látja az operációs rendszert az egyszerű felhasználó, az alkalmazásfejlesztő, illetve a rendszermenedzser?

16. Milyen előnyt nyújthat a szöveges felhasználói felület a grafikussal szemben?

17. Milyen előnyöket nyújthat a parancsnyelvű kezelői felület a menürendszerhez viszonyítva?

18. Mi a különbség a szinkron, illetve aszinkron működés között?

19. Mire szolgálnak a batch-fájlok, illetve shell-scriptek?

20. Miben tér el a rendszerhívás a közönséges szubrutinhívástól?

21. Mit jelent az, hogy az operációs rendszer „kiterjeszti” a számítógép utasításkészletét?

22. Hogyan közvetíti a programozási nyelv a rendszerhívásokat a programozó számára?

23. Milyen előnnyel jár a rendszerhívások valamilyen magas szintű programnyelvvel történő megadása?

24. Milyen módokon kapcsolódik az operációs rendszer a hardverhez?

25. Mi a készülékkezelő *(driver)* program feladata?

26. Rajzolja le egy egyszerű mikroszámítógép felépítését és adja meg, hogy milyen feladatokat látnak el az egyes részek!

27. Rajzolja le egy tipikus személyi számítógép felépítését és adja meg az egyes részek funkcióit!

28. Milyen architektúrabeli eltéréseket mutat egy szuperszámítógép egy személyi számítógéphez képest?

29. Melyek egy szuperszámítógép lehetséges alkalmazási stílusai?

30. Milyen kapcsolatban vannak egymással egy operációs rendszer rétegei?

31. Melyek a moduláris programozás alapelvei?

32. Miért ütközik nehézségekbe egy operációs rendszer tiszta rétegszerkezetű kialakítása?

33. Melyek egy operációs rendszer által megvalósítandó alapvető funkciócsoportok?

34. Hogyan valósul meg a rétegszerkezet a modern operációs rendszerekben?

35. Mire szolgál az operációs rendszer magja, a kernel?

36. Mi a virtuális hardver koncepció lényege?

37. Mi a kliens–szerver-modell lényege?

38. Mi a kernel szerepe kliens–szerver-modell esetén?

39. Milyen események aktiválhatják az operációs rendszert, ha az várakozó állapotban tartózkodik?

40. Milyen esetekben nem folytatódik az aktuális folyamat végrehajtása egy rendszerhívás végrehajtása után?

41. Mit nevezünk szinkron, illetve aszinkron rendszerhívásnak és hogyan valósíthatóak meg?

42. Milyen feltételnek kell eleget tennie a hardver architektúrának a multiprogramozás megvalósíthatóságához?

43. Hogyan valósul meg egy B/K-művelet szinkron rendszerhívás esetén?

44. Mi a specialitása a karakterenkénti, vagy bájtonkénti B/K-művelet megvalósításának?

45. Miért kell az operációs rendszernek várakozási sort szerveznie a B/K-készülékekre?

46. Mi a parancsértelmező feladata?

47. Milyen értelmezési prioritást követ a parancsértelmező?

48. Mire szolgálnak a külső megszakítások?

49. Miért kell a teljes megszakítási rendszert kizárólag az operációs rendszernek kezelnie?

50. Hogyan teszi lehetővé az operációs rendszer a megszakítások felhasználói programok által történő kezelését?

51. Milyen tipikus időkezelési funkciókat valósít meg az operációs rendszer?

52. Hogyan valósítható meg a korrekt időmérés?

53. Milyen hardver-, illetve szoftverhibákat kell az operációs rendszernek kezelnie?

54. Milyen lehetőségei vannak az operációs rendszernek valamely hiba bekövetkezése esetén?

55. Mi az alapvető különbség a külső megszakítás, illetve a hibakezelés között?

56. Miért van szükség a felhasználói programban kivételkezelés megvalósítására, és hogyan támogatja ezt az operációs rendszer?

57. Milyen feladatokat kell végrehajtani egy operációs rendszer be-, illetve kikapcsolásakor?

7.2. Az operációs rendszer mint absztrakt, virtuális gép

1. Mi a különbség a multiprogramozás és a multiprocesszálás között?

2. Mi a különbség a vezérlési gráf és a vezérlési szál között?

3. Ismertesse a folyamatok logikai modelljét!

4. Mit jelent az, hogy a memória RAM-modell szerint működik?

5. Mi jellemzi egy folyamat adott pillanatbeli állapotát?

6. Miben tér el egymástól a folyamatokhoz tartozó logikai, illetve a számítógép fizikai processzorának utasításkészlete?

7. Mi a különbség a logikai és a fizikai memória között?

8. Mi az eltérés a folyamatok, illetve a szálak között, és milyen előnnyel jár a szálak alkalmazása?

9. Mit nevezünk multitaszkos operációs rendszernek?

10. Milyen célokat szolgálhat egyszerre több folyamat futtatása egy rendszerben?

11. Milyen járulékos nehézség merül fel egy folyamatokból álló rendszer fejlesztése során a hagyományos szekvenciális programfejlesztési módszerhez képest?

12. Definiálja a független, versengő, illetve együttműködő folyamatok fogalmát! Milyen támogatást kell nyújtson az operációs rendszer ezek megvalósításához?

13. Milyen módon jönnek létre egy rendszer folyamatai?

14. Mikor nevezünk statikusnak, illetve dinamikusnak egy operációs rendszert?

15. Mi a különbség a hierarchikus és a globális erőforrás-gazdálkodás között?

16. Miben tér el a PRAM-modell a RAM-modellhez képest?

17. Hasonlítsa össze a közös memórián, illetve az üzenetváltáson alapuló folyamatok közötti együttműködést!

18. Miért van szükség a folyamatok szinkronizálására?

19. Mit nevezünk kritikus szakasznak?

20. Mi a folyamatok közötti kölcsönös kizárás?

21. Definiálja két folyamat egyidejűségét (randevúját), illetve precedenciáját!

22. Milyen elvárásoknak kell megfeleljen a kölcsönös kizárás megvalósítása?

23. Milyen megoldást ajánlott Peterson két folyamat közti kölcsönös kizárás szoftveres megvalósítására?

24. Miért van szükség hardvertámogatásra a kölcsönös kizárás megvalósításához, illetve milyen módszerek segítségével oldható ez meg?

25. Hogyan garantálható, hogy a kritikus szakaszba belépni kívánó folyamatok véges idő alatt be is jussanak?

26. Adja meg a szemafor primitív műveleteinek definícióját!

27. Hogyan alkalmazható a szemafor a precedencia, illetve a kölcsönös kizárás megvalósítására?

28. Szemaforok felhasználásával írjon olyan programrészletet (például eljárást), amely lehetővé teszi *N* (előre adott konstans) folyamat randevúját, azaz az összes folyamat bevárja egymást.

29. Egy laktanyát őrség őriz, az őrség minden tagja (folyamat) tudja, hogy őrhelyét csak akkor hagyhatja el, ha a váltás, a következő őr megérkezett. Szemaforokat használva írja meg az őrségváltás lebonyolításának algoritmusát.

30. Van *N* (előre adott konstans) folyamatunk, mindegyikük tudja a saját sorszámát. Szemaforok felhasználásával írjon olyan

WaitForMyTurn(i: INTEGER)

eljárást, amelyet ha az egyes folyamatok a saját sorszámukkal meghívják, akkor onnan a sorszámuk szerinti sorrendben lépnek ki, azaz egy folyamat az eljáráson belül várakozik mindaddig, amíg az összes nála kisebb sorszámú folyamat ki nem lépett ebből az eljárásból!

31. Hasonlítsa össze a szemafor műveleteit az erőforrás-, illetve az esemény-szinkronizációs eszközökkel!

32. Miért kell az erőforrásokhoz és a szemaforokhoz várakozási sorokat létrehozni?

33. Lazán csatolt rendszerekben egymással kommunikáló folyamatok milyen különböző megnevezési módszereket használhatnak. Milyen paramétereket használnak az üzenetküldés *(send)* és üzenetfogadás *(receive)* parancs hívásakor a különböző módszerek alkalmazása esetén?

34. Üzenetváltásos modell esetén milyen módszereket használnak a kommunikációs műveletek helyességének visszajelzésére?

35. Milyen konzisztencia feltételt kell betartani a küld *(send)* parancs megvalósításakor?

36. Milyen megoldások alkalmazhatóak a küld *(send)*, illetve fogad *(receive)* parancsoknál fellépő várakozás kezelésére?

37. Milyen járulékos mellékhatásokat okoznak a kommunikációs műveletek a rendszer átmeneti tárolójának (pufferének) függvényében?

38. Definiálja a holtpont fogalmát!

39. Milyen problémákat okoz egy rendszerben a holtpont kialakulása?

40. Ismertesse az erőforrásokért versengő rendszerekre vonatkozó rendszermodellt!

41. Ismertesse a holtpont kialakulásának feltételeit! Mikor elégségesek ezek a feltételek?

42. Hogyan követhető nyomon a holtpont kialakulása az erőforrás-foglaltsági gráf segítségével?

43. Az operációs rendszer milyen általános eljárásokat használhat a holtpont kezelésére?

44. Milyen tényezőktől függ, hogy egy rendszerben milyen gyakorisággal következhet be holtpont?

45. Mit jelent a holtpont elkerülése *(deadlock avoidance)*?

46. Mondjon egy-egy példát a holtpont megelőzésénél *(deadlock prevention)* használt, a kialakulás különböző feltételeit figyelembe vevő algoritmusokra!

47. Milyen algoritmust javasolt Coffman a holtpont észlelésére?

48. Mit nevezünk a holtpont szempontjából biztonságos állapotnak?

49. Mi a bankár algoritmus szerepe a holtpont probléma megoldásában?

50. Hogyan kombinálhatóak a holtpont kezelésére alkalmazott technikák?

51. Mikor alakulhatnak ki kommunikációs holtpontok?

52. Egy rendszerben 4 erőforrásosztály van (A, B, C és D), az egyes osztályokba rendre 8, 11, 7 és 10 erőforrás tartozik. A rendszerben 4 folyamat verseng az erőforrásokért, a következő aktuális foglalással és maximális igénnyel:

	Maximális				Aktuális			
	A	B	C	D	A	B	C	D
P1	2	2	5	4	0	2	3	3
P2	7	6	3	4	3	1	2	2

P3	5	6	3	4	2	2	0	2
P4	4	1	2	3	2	1	2	2

A rendszer a bankáralgoritmust alkalmazza a holtpont elkerülésére. Biztonságos állapotban van-e jelenleg a rendszer? Ha igen, mutassa meg, a folyamatok hogyan tudják befejezni működésüket, ha nem, hogyan alakulhat ki holtpont.

53. Ismertesse a kiéheztetés *(starvation)* fogalmát. Az operációs rendszer milyen eljárásokat használhat a kiéheztetés elkerülésére?

54. Mutasson példát arra, mikor fordulhat elő kiéheztetés!

55. Ismertesse a termelő–fogyasztó problémát!

56. Milyen kérdést kell megoldani az írók–olvasók probléma esetén?

57. Mi az étkező filozófusok problémája?

58. Milyen probléma merül fel adatfolyamok illesztése esetén?

59. Mi a belső biztonság?

60. Mi a különbség a statikus és a dinamikus védelmi tartomány között?

61. Mi a hozzáférési mátrix?

62. Mi a globális tábla?

63. Mi a hozzáférési lista?

64. Mi a jogosítvány lista?

65. Mit takar a külső biztonság fogalma?

66. Mi az információ szivárgás?

67. Milyen fenyegetések léphetnek fel elosztott rendszerek külső biztonságával kapcsolatban?

68. Milyen támadási módszerek léteznek elosztott rendszerek külső biztonságával kapcsolatban?

69. Mi a „féreg”?

70. Mi a „trójai faló”?

7.3. Multiprogramozott operációs rendszerek

1. Mi a futásra kész állapotú folyamatok közös jellemzője?

2. Miért lehet szükség egy operációs rendszer felügyelete alatt futó folyamatoknál felfüggesztett állapotra? Mondjon konkrét példát a felfüggesztés okára!

3. Sorolja fel, hogy egy operációs rendszerben a rendszerhívások végrehajtása milyen főbb lépésekben zajlik! Mi a jelentősége annak, hogy a felhasználói programok és az operációs rendszer magja a CPU különböző működési módjában fut?

4. Rajzolja fel egy „tipikus”, felfüggesztett állapotokat is használó operációs rendszerben a folyamatok állapotátmeneti diagramját az állapotok és az átmenetek elnevezésével! Mikor, milyen okok hatására következhet be a futó állapotból futásra kész állapotba átmenet?

5. Az operációs rendszerek a megszakítások kiszolgálására milyen alapvető módszereket alkalmazhatnak?

6. Ismertesse a hosszú, közép- és rövid távú ütemezés feladatát! Egy folyamat állapotátmeneti diagramján jelölje be azokat az átmeneteket, amelyek a fenti ütemezéseket jelentik!

7. Milyen szempontok alapján történhet a folyamatok hosszú távú ütemezése?

8. Ismertesse, hogy az operációs rendszer megszakítás kiszolgálása során milyen tevékenységeket hajt végre! Hogyan jelentkezik a megszakítások kiszolgálásánál a preemtív és a nem preemptív ütemezés különbözősége?

9. Mit jelent a CPU-ütemezési algoritmusok paraméterei között a CPU-kihasználtság?

10. Mi a kapcsolat a CPU-ütemezési algoritmusok jellemzésére használt körülfordulási idő *(turnaround time)* és várakozási idő *(waiting time)* között?

11. Mikor nevezünk egy ütemezőt preemptívnek?

12. Milyen paraméterek alapján lehet a különböző CPU-ütemezési algoritmusokat értékelni? Definiálja a különböző paraméterek jelentését! Hasonlítsa össze valamely fenti paraméter alapján a legrégebben várakozó (FCFS) és a legrövidebb löketidejű (SJF) algoritmusokat!

13. Ismertesse a legrövidebb löketidejű *(SJF, Shortest Job First)* és a legrövidebb hátralévő löketidejű *(SRTF, Shortest Remaining Time First)* CPU-ütemezési algoritmusokat, kiemelve a közöttük lévő lényeges különbségeket!

14. Definiálja a preemptív és a nem preemptív ütemező közötti különbséget! Rajzoljon fel egy felfüggesztett állapotokat nem tartalmazó folyamat állapotátmeneti diagramot és illusztrálja ezen is a preemptív és nem preemptív ütemezés közötti különbséget!

15. Milyen szempontok alapján lehet osztályozni a folyamatok ütemezésénél megismert visszacsatolt többszintű sorokat *(Multilevel Feedback Queues)*?

16. Hogyan lehet a CPU-löketidőn *(burst time)* alapuló ütemezési algoritmusoknál a futásra kész folyamatok következő löketidejét meghatározni?

8. Egy operációs rendszerben a következő folyamatok találhatók futásra kész állapotban (az érkezési idő azt az időpillanatot jelenti, amikor a folyamat futásra késszé vált):

Folyamat	Érkezési idő	Löketidő
P1	0	4
P2	1	5
P3	5	3
P4	7	1
P5	9	3

Adja meg, hogy az egyes folyamatok milyen sorrendben futnak le, számolja ki a folyamatok átlagos várakozási idejét

- sorrendi (First Come First Serve, FCFS),
- legrövidebb löketidejű (Shortest Job First, SJF),
- legrövidebb hátralévő löketidejű (Shortest Remaining Time First, SRTF),
- 2 időegységnyi időszeletű körforgó (Round Robin, RR),
- nem preemptív prioritásos ütemezés esetén.

18. Melyik ütemezési algoritmus esetén a legkisebb a válaszidő *(response time)* szórása?

- Legrégebben várakozó *(First Come, First Served, FCFS),*
- Körbeforgó *(Round Robin, RR).*
- Legrövidebb löketidejű *(Shortest Job First, SJF).*

19. Mely állítások igazak a CPU-ütemezésnél használt körbeforgó *(Round Robin)* algoritmusra?

- Biztosítja a legkisebb átlagos várakozási időt.
- Használata esetén nem lép fel a folyamatok kiéheztetése.
- Használatával lehet valósidejű operációs rendszert készíteni.

20. Mely állítások igazak a preemtív ütemező algoritmusra?

- Mindig prioritásos ütemezési algoritmusokat használ annak eldöntésére, hogy mikor és melyik folyamatot kezdje el futtatni.
- Alkalmazása esetén egy éppen futó folyamatot bármelyik időpillanatban megszakíthat és futásra kész állapotúvá tehet az operációs rendszer.
- Használatával az operációs rendszer átlagos válaszideje *(response time)* csökkenthető.

21. Mely állítások igazak a prioritásos ütemező algoritmusokra?

- Jobban kihasználják a CPU-t.

- Használata holtpont kialakulásához vezethet.
- Használata kiéheztetéshez vezethet.

22. Milyen ütemezési módszereket alkalmaznak szimmetrikus, illetve aszimmetrikus multiprocesszoros rendszerekben?

23. Definiálja a külső *(external)* és belső tördelődés *(internal fragmentation)* fogalmát! Mondjon konkrét példát mindkét jelenségre!

24. Mit jelent a pozíció-független *(position independent, PIC)* programrészlet?

25. Mi a kapcsolatszerkesztő *(linker)* program feladata?

26. Mit jelent a változó méretű partíciós rendszerekben használt algoritmusoknál a tárkihasználtság 1/3-os szabálya?

27. Mit jelent az átfedő programrészekkel *(overlay)* történő tárkezelés?

28. Mi a tömörítés *(garbage collection)* szerepe a többpartíciós társzervezésben?

29. A programfejlesztés, illetve futtatás mely fázisaiban történhet a program memóriacímeinek kötése?

30. Magyarázza el a változó méretű partíciók lefoglalásánál használt

- legjobban megfelelő (best fit),
- első megfelelő (first fit),
- legrosszabban illeszkedő (worst fit)

algoritmusokat. Egy rendszerben az adott pillanatban 100 K, 600 K, 200 K, 300 K és 500 K méretű szabad területek vannak. Hogyan foglal helyet a fenti 3 algoritmus sorrendben 417 K, 212 K, 112 K és 426 K méretű partícióknak?

31. Ismertesse kombinált szegmens- és lapszervezést tartalmazó tárkezelő hardver esetén a logikai-fizikai címtranszformáció módját! Melyek a kombinált társzervezés előnyei?

32. Ismertesse az igény szerinti lapozáson *(demand paging)* alapuló virtuális tárkezelésnél a legrégebben nem használt *(LRU, Least Recently Used)* és az újabb esély *(Second Chance)* lapcsere algoritmusokat! Hogyan lehet(ne) az LRU-algoritmust implementálni? Milyen hardver támogatás szükséges az újabb esély algoritmushoz?

33. Definiálja a következő fogalmakat: vergődés *(trashing),* munkahalmaz *(working set),* térbeli és időbeli lokalitás! Hogyan lehet a folyamatok munkahalmazának méretét figyelembe venni a vergődés elkerüléséhez?

34. Milyen összefüggés van a virtuális tárkezelésnél munkahalmaz nagysága és a folyamat által okozott laphibák gyakorisága között? Hogyan lehet a laphiba gyakoriságot felhasználni a folyamatok számára allokált lapok számának meghatározásánál?

35. Ismertesse és hasonlítsa össze a virtuális tárkezelésnél használt FIFO és újabb esély *(second chance)* algoritmusokat! Milyen hardvertámogatást igényelnek ezek?

36. Mit nevezünk Belady-féle anomáliának?

37. Milyen részidőkből áll össze a háttértáron lévő lapokhoz való hozzáférési idő? Kis vagy nagy lapok használata esetén kapunk jobb transzferidőt?

38. Egy igény szerinti lapozást alkalmazó rendszerben 4 fizikai memórialap található és sorban a következő virtuális lapokra történik hivatkozás:

1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3

Számítsa ki a rendszerben bekövetkező laphibák számát a következő algoritmusok alkalmazása esetén:

- legrégebbi lap *(FIFO),*
- újabb esély *(Second Chance),*
- legrégebben nem használt *(Least Recently Used, LRU),*
- optimális.

A négy fizikai lap kezdetben „üres”, nem tartalmazza egyik virtuális lapot sem.

39. Mi a translation lookaside buffer és mi a szerepe a fizikai cím kiszámításánál?

40. Melyek igaz állítások?

A lapokat alkalmazó virtuális tárkezelésnél előforduló belső tördelődést csökkenteni lehet

- az egyes lapok méretének növelésével.
- az egyes lapok méretének csökkentésével.
- megfelelően kiválasztott lap elhelyezési *(placement)* stratégia alkalmazásával.

41. Melyek igaz állítások?

A virtuális tárkezelésnél egyes lapokat ideiglenesen a tárba kell „fagyasztani” *(page locking),* azaz megakadályozni, hogy a lapcsere algoritmus cserére kijelölje, ha

- azokat több folyamat is használja.
- az adott lapokon valamelyik folyamat verme található.
- az adott lapokra folyamatban van perifériás művelet.

42. Melyek igaz állítások?

Igény szerinti lapozást *(demand paging)* használó rendszerekben egy folyamat vergődik, ha

- a folyamatnak túl kicsi a prioritása.
- a folyamat munkahalmaza *(working set)* több lapból áll, mint ahány lapot a rendszer a folyamathoz rendel.
- a folyamat munkahalmaza nagyobb, mint a rendszer generálásakor megadott konstans.
- a folyamat futását gyakran szakítják meg nála nagyobb prioritású folyamatok.

43. Hogyan lehet egy szegmensszervezésű tárkezelő rendszerben a túlcímzés ellen védekezni?

44. Melyik állomány tárolási módszer esetén jelent gondot az állomány blokkjaihoz direkt hozzáférés megvalósítása?

- láncolt listás,
- indexelt,
- FAT-ot használó.

45. Ismertesse az állományokhoz tartozó blokkok tárolásához használt láncolt listás, az ún. FAT-os *(File Allocation Table)* és az indextáblás módszereket! Milyen előnyös tulajdonságokkal rendelkezik a FAT-os módszer?

46. Sorolja fel, milyen módszereket ismer a lemezegységen a szabad helyek, illetve az egyes állományokhoz tartozó blokkok nyilvántartására!

47. Mit jelent az adattárolás biztonságának növelésére használt inkrementális mentés *(incremental backup)?*

48. Rajzolja fel egy mágneslemezes háttértár tipikus felépítését, valamint adja meg az egydimenziós blokkcímek meghatározási módját!

49. Mi a RAID *(Redundant Array of Inexpensive Disks)* technika lényege?

50. Egy 200 sávos (0 .. 199) mágneslemez-egységen a fej jelenleg a 143-as sáv felett áll, ezt megelőzően a 125-ös sávon szolgált ki egy átviteli kérelmet. Jelenleg a következő sávokra várakozik – a megadott érkezési sorrendben – egy-egy átviteli kérelem: 86, 147, 91, 177, 94, 150, 102, 175, 130. Adja meg, hogy a kéréseket az

- időrendi kiszolgálás *(First Come, First Served, FCFS),*
- legkisebb fejmozgás *(Shortest Seek Time First, SSTF),*
- pásztázó *(SCAN),*
- körkörös *(Circular LOOK)*

algoritmus milyen sorrendben szolgálja ki, illetve közben a fej mekkora utat (hány sávnyit) tett meg!

51. Melyik, a lemezműveletek ütemezésénél használt algoritmusnál nem lép fel kiéheztetés veszélye:

- pásztázó *(SCAN),*
- N lépéses pásztázó *(N-SCAN),*
- legrövidebb fejmozgási idejű *(Shortest Seek Time First)*.

52. Melyek igaz állítások?

A lemezműveletek ütemezésénél használt algoritmusok közül a következőnek a legkisebb az átlagos kiszolgálási idő szórása:

- legrövidebb fejmozgási idő *(Shortest Seek Time First),*
- pásztázó *(SCAN),*
- egyirányú pásztázó *(Circular SCAN)*.

53. Melyek az ún. belső, illetve külső parancsok?

54. A felhasználóknak milyen igényei lehetnek egy felhasználói felülettel szemben?

7.4. Hálózati és elosztott rendszerek

1. Mit nevezünk elosztott rendszernek?

2. Soroljon fel legalább hármat az elosztott rendszerek alkalmazásával járó előnyök közül!

3. Elosztott rendszerekben mit jelent a skálázhatóság?

4. Mi a nyitottság elérésének legfontosabb feltétele?

5. Mi a különbség a szinkron, illetve az aszinkron üzenetküldés között?

6. Mi a különbség a csoportos multicast és a broadcast között?

7. Mi a függvényszállítás és mire szolgál?

8. Mit nevezünk protokollnak?

9. Milyen előnyökkel, illetve hátrányokkal jár a részlegesen összekapcsolt topológia a teljesen összekapcsolttal szemben?

10. Milyen részlegesen összekapcsolt hálózati topológiákat ismer? Hasonlítsa össze őket!

11. Adja meg az ISO OSI-modell rétegeit és azok funkcióját! Mire szolgál a modell?

12. Milyen módszereket alkalmazhatnak forgalomirányításra *(routing)*?

13. Mi az alapvető különbség az áramkör-, üzenet-, illetve csomagkapcsolás között?

14. Mi a távoli terminálszolgáltatás feladata?

15. Hogyan hozza létre a telnet program a különféle típusú terminálok közötti kapcsolatot?

16. Hogyan valósul meg a kliens–szerver-modell egy telnet kapcsolatban?

17. Mire szolgál a telnet parancsértelmező?

18. Mi a különbség az ftp bináris és szöveges átviteli módja között?

19. Hogyan valósul meg a kliens–szerver-modell egy ftp kapcsolat során Windows-, illetve UNIX-alapú rendszerekben?

20. Mire szolgál az anonim belépés egy ftp szerverre, hogyan valósították meg ezt UNIX alatt?

21. Milyen követelmények merülnek fel egy elosztott rendszerrel szemben?

22. Mit takar az erőforrás megosztás? Milyen módszereket alkalmaznak erőforrás megosztásra?

23. Hogyan biztosítható egy rendszer nyitottsága?

24. Mit jelent a virtuális párhuzamosság?

25. Hogyan biztosítható a skálázottság?

26. Hogyan tehető hibatűrővé egy rendszer?

27. Mit jelent az átlátszóság fogalma?

28. Egy elosztott rendszerben mire szolgál a név feloldás *(name resolution)?*

29. Mi az előnye a hierarchikus névtér alkalmazásának?

30. Mi a különbség a szinkron (azaz blokkolódó), illetve aszinkron (azaz nem blokkolódó) üzenetküldés között?

31. Hasonlítsa össze a kliens-szerver alapú, a csoportos multicast, illetve a függvényszállításon alapuló kommunikációs sémákat!

32. Egy elosztott operációs rendszer esetén mik az ún. nyílt szolgáltatások?

33. Hasonlítsa össze a munkaállomás-szerver és a processzor pool modellt!

34. Elosztott rendszerekben milyen konzisztenciákat kell biztosítani és miért?

35. Mi az előnye az elosztott fájlrendszer alkalmazásának a telnet vagy ftp alapú fájltranszferrel szemben?

36. Miben különbözik az elosztott állománykezelésnél az állományok elhelyezkedését eltakaró *(location transparent)*, illetve az elhelyezkedéstől független *(location independent)* elnevezés?

37. Milyen elnevezési módszereket alkalmazhatnak elosztott fájlrendszerekben?

38. Mire szolgál a leképezési táblák többszörözése?

39. Ismertesse az elosztott állományrendszerek implementációjánál használt távoli szolgáltatások *(remote services)* és helyi átmeneti tárolás *(local cache)* módszerek egymáshoz viszonyított előnyeit.

40. Átmeneti tárak használata esetén hogyan biztosítható azok konzisztenciája?

41. Definiálja és hasonlítsa össze az elosztott állománykezelésnél alkalmazott állapotot tároló *(stateful)* és állapot nélküli *(stateless)* szolgáltató implementációkat!

42. Mire szolgál elosztott rendszerekben a fájlok megsokszorozása?

43. Milyen modulokból épül fel egy elosztott állományrendszer szolgáltató?

44. Milyen problémát jelent, ha az NFS elosztott állományrendszer teljesítményének növelésére a kliens oldalon helyi gyorsító tárat *(cache)* is használ?

- A gyorsító tár betelése esetén a kliens oldal nem képes addig új állományt megnyitni, amíg valamelyik nyitott állományt le nem zárták.
- A szerver leállása és újraindulása után újra meg kell nyitni a korábban megnyitott állományokat.
- Az egyik kliens által a közösen használt állományszerkezetben létrehozott állományokat a többi kliens esetleg csak később veszi észre.

45. Melyek igaz állítások?

Az elosztott állománykezelésnél az állapotmentes *(stateless)* szolgáltató előnyösebb az állapotot tárolónál *(statefull)*, mert

- az állományok kezelése sokkal gyorsabb.
- egyszerűbb a szolgáltató újraindulása után a műveleteket folytatni.
- könnyebb a kérések jogosságának ellenőrzése.

46. Mi az ún. démon folyamatok feladata?

47. Mi a különbség a hálózati, illetve az elosztott számítási modell között?

48. Mi a middleware?

49. Kliens–szerver-rendszereknél mi a kötés *(binding)* feladata és milyen fajtái vannak?

50. Mi a szerver replikáció, illetve az átmeneti tárolás *(caching)* közti különbség?

51. Mire szolgálnak a proxy szerverek?

52. Mire szolgál az RPC-protokoll?

53. Mit definiál az XDR-szabvány, milyen esetben jelenthet hátrányt az alkalmazása?

54. Mi az RPC-szál, milyen előnnyel jár az alkalmazása?

55. Milyen problémákhoz vezethet egy operációs rendszerben a szálak alkalmazása?

56. Mit neveznek külső, illetve belső szinkronizációnak?

57. Mi a Koordinált Univerzális Idő (KUI)?

58. Milyen óra rendszereket ismer?

59. Mit jelent a stratum a Network Time Protocolban?

60. Hogyan működik a Network Time Protocol?

61. Mire szolgál a logikai óra?

62. Hogyan lehet definiálni a korábban-történt relációt?

63. Mit jelent a teljesen rendezett logikai óra fogalom, és minek a segítségével lehet elérni a teljes rendezést?

64. Sorolja fel az elosztott kölcsönös kizárás módszereit!

65. Ismertesse az elosztott rendszerekben a kölcsönös kizárás megvalósításra használható teljesen elosztott algoritmus működését! Melyek az algoritmus problémái a központosított algoritmussal összevetve?

66. Ismertesse az időcímkés elosztott kölcsönös kizárási algoritmust!

67. Mire szolgálnak a választási algoritmusok?

68. Mit feltételez a gyűrű választási algoritmus?

69. Mit feltételez a bully-algoritmus?

70. Ismertesse az elosztott rendszerekben alkalmazott ún. erőszakos *(bully)* választási *(election)* algoritmust!

71. Mi a feladata a biztonságpolitikának elosztott rendszerek védelménél?

72. Milyen területekre terjed ki egy rendszer védelme?

73. Milyen céljai lehetnek, illetve ezek elérésére milyen módszereket alkalmazhat egy lehetséges támadó?

74. Miért kellene formális módszerekkel igazolni a biztonsági protokoll helyességét?

75. Mi a biztonsági rendszerek két aranyszabálya?

76. Mi a különbség a nyilvános és a titkos kulcsú titkosítási rendszerek között?

77. Milyen problémát kel megoldani egy titkosító rendszerben a kulcsok szétosztásánál?

78. Mire szolgál a hitelesítő szerver elosztott rendszerekben?

79. Ismertesse a Needham–Schroeder titkos kulcsú titkosítást (mutassa be az egyes lépéseket két folyamat titkos kommunikációjával)!

80. Milyen védelmi szinteket definiál a Kerberos hitelesítő rendszer?

81. Mi a DES?

82. Mit tartalmaz egy Kerberos jegy?

83. Mit tartalmaz egy Kerberos hitelesítő (jegy)?

84. Kliens kérésére hogyan hitelesíti magát egy szerver Kerberos alatt?

85. Hogyan védekeznek a Kerberos-rendszerben a felhasználónak kiadott, valamely szolgáltató felhasználására feljogosító jegy *(ticket)* mások általi felhasználása ellen?

86. Milyen előnyökkel, illetve hátránnyal jár a Kerberos hitelesítő rendszer alkalmazása?

87. Mi a szerepe a Kerberos-rendszerben a hitelesítőnek *(authenticator)?*

- Azonosítja a felhasználót a Kerberos-rendszer számára.
- Igazolja, hogy a jegyet *(ticket)* a jogos tulajdonosa nyújtotta be a szervernek.
- Tartalmazza a felhasználó saját titkos kulcsát.

88. Mely állítások igazak?

A Kerberos-rendszerben egy szolgáltatás igénybevételére jogosító jegyet *(ticket)*

- az ügyfél és a szerver is dekódolni tudja a kapcsolatuk idejére kiadott titkos kulcs segítségével.
- csak a szerver tudja dekódolni a saját titkos kulcsa segítségével.
- egyik fél sem dekódolja, a szerver is kapott egy a felhasználóéval azonos jegyet, csak a benyújtott és ennek a tárolt jegynek az azonosságát ellenőrzi.

7.5. UNIX

1. Milyen előnyöket biztosít a UNIX-rendszerek réteges belső szervezése?

2. Milyen alapvető fejlesztéseket hajtottak végre a modern UNIX-rendszerek felépítésében?

3. Miért növeli a hatékonyságot UNIX-rendszerekben a szálak használata?

4. Rajzolja fel a UNIX operációs rendszer folyamatainak teljes állapotátmeneti diagramját! Mi a zombie állapot szerepe, mikor kerül egy folyamat ebbe az állapotba és mikor hagyja azt el?

5. Ismertesse a UNIX-rendszerben a folyamatok létrehozásának mechanizmusát! Térjen ki a kernel által elvégzendő feladatokra, és a folyamatok közötti leszármazási hierarchiára!

6. Ismertesse a UNIX-rendszerben a folyamatok leállításának mechanizmusát, a kernel által elvégzett feladatokat, és a folyamat szülőjének küldött jelzés jelentőségét!

7. Mikor tér vissza hibával a UNIX operációs rendszerben a fork rendszerhívás?

- Ha nincs a központi tárban elegendő szabad tárterület.
- Ha az operációs rendszer folyamat táblájában (process table) nincs üres hely.
- Ha a felfüggesztett folyamatok száma átlépett egy a rendszer generálásakor megadott értéket.

8. Mely állítások igazak?

A UNIX operációs rendszerben egy folyamat megszüntekor zombie állapotú marad addig, amíg

- vannak még élő gyerekei;
- az általa írt pipe-okat olvasó folyamatok még nem szűntek meg;
- a szülő – vagy annak halála esetén az init – el nem vette a folyamat visszatérési értékét.

9. Sorolja fel, hogy a UNIX-folyamatok esetén mi tartozik egy folyamat környezetéhez *(context)*!

10. A következő információk közül jelölje be azokat, amelyeket a UNIX-kernel a folyamatok ún. u-area területén tart.!

- folyamat prioritása,
- folyamat futása alatt érvényes aktuális könyvtár,
- folyamat adatterületének nagysága és címe,
- folyamat által eddig felhasznált összes CPU-idő,
- a nyitott állományok aktuális pozíciója,
- a folyamat zombie állapotú gyermekeinek visszatérési értéke,
- az aktuálisan végrehajtódó rendszerhívások paraméterei,
- a folyamat állapota,
- a folyamat várakozásának *(sleep)* oka,
- a folyamat által lekezelt jelzések (signal) kezelésére szolgáló függvények címe.

11. Mit jelent a UNIX operációs rendszerben, ha egy program setuid jogosultsággal rendelkezik?

12. Mire szolgálnak a hitelesítők (kredenciálisok) a UNIX-rendszerben?

13. Mire szolgál a UNIX operációs rendszerben a wait rendszerhívás?

14. Milyen ütemezést valósít meg a UNIX SVR3? (Gondoljon a kernel, user módra.)

15. Mi a különbség a megszakítható és a nem megszakítható alvás közöttt?

16. Mit jelent, hogy a kernel kód reentrens?

17. Miért nem preemtív kernel módban a UNIX ütemezése?

18. A UNIX operációs rendszer futásra kész folyamatai prioritása annál nagyobb,

- minél kevesebb folyamatot futtat az adott felhasználó;
- minél kisebb tárterületet használ;

- minél régebben nem használta a CPU-t.

19. Miért van a UNIX kerneljében a várakozó állapotba átmenetet kiváltó sleep hívás mindig egy while ciklus belsejében?

20. Jelölje be valamennyi olyan eseményt, amikor a UNIX operációs rendszerben biztosan környezetváltás (context switch) történik!

    - Egy futásra kész folyamat olyan jelzést kap, amit nem kezel le és nem is hanyagol el.
    - A futónál nagyobb prioritású folyamat futásra késszé válik.
    - A read rendszerhívás végrehajtásához szükséges diszk blokk nincs bent a gyorsító tárban (buffer cache).
    - A futó folyamat exit rendszerhívást hajt végre.
    - Minden óramegszakítás bekövetkeztekor.
    - A futó folyamat laphiba megszakítást okozott és a szükséges lap nincs a fizikai tárban.
    - Egy várakozó állapotban lévő folyamat a folyamat által nem lekezelt jelzést (signal) kap.

21. UNIX-kernelben hol preemptiv a folyamatok ütemezése?

22. Mire szolgálnak a callout-ok, hogyan biztosítható minél hamarabbi végrehajtásuk?

23. A UNIX operációs rendszer a folyamatainak ütemezésénél megkülönböztet rendszer- és felhasználói prioritásokat. Mikor kap egy folyamat rendszer prioritást és mitől függ ennek az értéke? Milyen paraméterek befolyásolják a felhasználói prioritás értékét?

24. Mire szolgál a *whichqs*?

25. Mutassa be a hagyományos UNIX-ütemező működését a következő példán: nice nincs, prioritás = CPU/2 + 60, csillapítás = CPU/2, az óra 60-at üt másodpercenként.

26. Melyek a hagyományos UNIX-rendszerekben megvalósított ütemezési algoritmus hiányosságai?

27. Ismertesse a UNIX operációs rendszer jelzéseinek *(signal)* fogalmát, az ezeket kezelő rendszerhívásokat, fontosabb paramétereit! Hogyan reagálhat egy folyamat a neki küldött jelzésekre?

28. Hol tárolja a UNIX operációs rendszer az egyes folyamatok által lekezelt jelzéseket *(signal)* kezelő eljárások kezdőcímeit?

29. Milyen célokat szolgálnak, és milyen hatásuk lehet a jelzéseknek (signals) a UNIX-rendszerekben?

30. Mutassa meg, hogy az SVR2-beli signalkezelés miért nem volt megbízható!

31. Ismertesse az SVR4 és a BSD verziókban alkalmazott jelzésmenedzsmentet!

32. Ismertesse a UNIX operációs rendszer jelzéseinek *(signal)* kezelésére szolgáló legfontosabb rendszerhívások feladatát! Mikor, milyen állapotátmeneteknél juthatnak érvényre a jelzések? Mondjon konkrét példát, eseményt, mikor az operációs rendszer küld jelzést egy folyamatnak.

33. Milyen kapcsolat van a folyamatcsoportok és a terminálkezelés között?

34. Mi a viszony-objektum *(session object)* feladata?

35. Milyen célokat szolgálhat a folyamatok közötti kommunikáció?

36. Milyen fizikai tulajdonságai vannak a kommunikációs csatornának?

37. Milyen módszereket használnak a folyamatok a megnevezésre *(naming)*?

38. Milyen típusú szinkronizáció történik tárolás nélküli átvitel, véges kapacitású tároló, illetve végtelen kapacitású tároló típusú átviteli csatornák esetén?

39. Mi történik ha a kommunikáció folyamán az egyik folyamat leáll?

40. Hogyan kezelhetőek a hibás, iletve elveszett üzenetek?

21. Sorolja fel a UNIX rendszerekben folyamatok közötti kommunikációra szolgáló módszereket. Melyek alkalmazhatok közülük csak szülő-leszármazottai viszonylatban?

42. Ismertesse hogyan történik az adatok átvitele a UNIX-csővezetékek *(pipe)* használatával! Térjen ki a szélsőséges esetekre (túltöltés, olvasás üres csővezetékből) is!

43. Ismertesse a UNIX System V IPC mechanizmusainak közös(!) vonásait!

44. Mi a különbség az elnevezett (named) és az anonim (unnamed) UNIX-csővezetékek *(pipe)* között? (Létrehozás, használat, megszűnés.)

45. Milyen problémák merülnek fel a System V által megvalósítot szemaforokkal kapcsolatban?

46. Mikor érdemes kommunikációra üzenet sorokat, illetve osztott memóriaterületet használni?

47. Melyek igaz állítások?

A UNIX-rendszer csővezetékét *(pipe)* használva

- az író *(write)* folyamatnak soha nem kell várakoznia.
- az olvasó *(read)* eljárás csak akkor tér vissza, ha a csővezetékből beolvasta a megkívánt számú Byte-ot.
- minden csővezetékhez külön tárbeli inode tartozik.

48. Jelölje be a felsoroltak közül az összes olyan információt, amelyek a UNIX operációs rendszer a memóriában lévő inode adatszerkezetben tárol!

- az átvitel (írás vagy olvasás) aktuális poziciója
- az inode azonosítója (kötet és a köteten belüli index)
- az állományt tartalmazó könyvtár inode-jának száma
- az inode-hoz tartozó állomány tulajdonosának azonosítói (UID és GID)
- mutatók a tárban lévő szabad inode-ok listájának megelőző és következő elemére

49. Jelölje be a felsoroltak közül az összes olyan információt, amelyek egy UNIX-kötet szuperblokkjában találhatók!

- A kötet mérete blokkokban.
- Annak a könyvtárnak a helye *(path)*, ahova a kötet fel van mount-olva.
- A szabad inode-ok egy részének sorszámai.
- A kötet blokkjainak mérete.
- A kötet tulajdonosa.

50. Hol és hogyan tárolja a UNIX operációs rendszer a lemezen lévő állományokhoz tartozó blokkokat? Írja le, hogyan található meg egy állomány állomány 274452! byte-ja (268x1024 + 20), feltételezve, hogy az egyes blokkok 1024 byte méretűek, illetve egy köteten legfeljebb 232 blokk lehet.

51. Háromszoros indirekt blokkal mennyi tárolóegységet lehet megcímezni (1 blokk 1K, egy cím 4 byte – a számolás menetét kell megadni)?

52. Hogyan néz ki UNIX alatt egy könyvtár bejegyzés?

53. Egy UNIX folyamat az open(„/path/filename”, O_RDWR, O_CREAT) rendszerhívással sikeresen létrehoz egy új állományt. Írja le, hogy a ennek során milyen változások történtek a köteten tárolt adatszerkezetekben!

54. A UNIX operációs rendszer hierarchikus könyvtárszerkezetében milyen adatszerkezettel ábrázolják az egyes állomány-nyilvántartásokat? Mi a különbség a szimbólikus és az ún. hard link között, hogyan ábrázolja a UNIX ezeket? Mikor nem lehet hard link-et létrehozni? Hogyan változik egy állományt leíró i-node tartalma, ha létrehozunk az állományra egy hard, illetve szimbolikus link-et?

55. A UNIX operációs rendszerben használt diszk gyorsítótár (buffer cache) megvalósításánál mikor szerepel egy blokk egyidejűleg a szabad blokkok listáján és a diszk blokkok hash listáján? Mikor kerül egy módosult blokk kiírásra? Sorolja fel a gyorsítótár alkalmazásának hátrányait!

56. Mi a szerepe a szabad helyek nyilvántartásánál a kötet szuperblokkjának?

57. Milyen problémák adódnak az S5FS állományrendszer esetén, és hogyan oldották meg ezeket a Berkeley FFS esetén?

58. Mi a virtuális állományrendszer?

59. Milyen speciális állományrendszerek találhatóak UNIX-rendszerekben?

60. Milyen szempontokat kell figyelembe venni egy naplózó állományrendszer tervezése esetén?

61. Mit nevezünk tartománynak?

62. Milyen algoritmus szerint allokál helyet a swapper a háttértáron?

63. Milyen esetben csökken a map bejegyzések száma?

64. Miért nem ír ki háttértárra zombi folyamatot a swapper?

65. Milyen feltételeknek kell teljesülni, hogy a swapper egy folyamatot háttértárra írjon, illetve onnan beolvasson?

66. Ismertesse a háttértár kezelését fork, illetve memória növelés esetén!

67. Mi a legfontosabb különbség a tárcsere *(swap)* és az igény szerinti lapozás *(demand paging)* között?

68. Mit jelent a virtuális tárkezelésnél alkalmazott *copy-on-write* módszer?

69. Ismertesse az igény szerinti lapozáshoz használt adatszerkezeteket és azok szerepét UNIX alatt!

70. Ismertesse a „laplopó” folyamatot, annak működését lapok háttértárra írásakor, illetve onnan történő olvasásakor!

71. Milyen hardver támogatásra van szükség igény szerinti lapozás megvalósításához?

72. Milyen állapotokban lehet egy laphibát okozó lap?

73. Mi teszi lehetővé az igény szerinti lapozás elvi alkalmazását?

74. Mit jelent a munkahalmaz ablakszélessége?

75. Mire szolgál a UNIX-ban a pfdata?

76. Milyen főbb adatszerkezeteket használnak a UNIX-ban memóriakezeléshez?

77. Milyen eltérés tapasztalható a BSD virtuális memóriakezelésében a System V implementációhoz képest?

78. Jellemezze az IP protokollt!

79. Mi a különbség az UDP és a TCP protokollok között?

80. Milyen lehetőségeket biztosít a SUN NFS a felhasználó számára?

81. Az NFS *(Network File System)* rendszerben a szerver oldalon az exportált könyvtáraknál a könyvtár nevén (elérési útján) kívül milyen paramétereket lehet megadni?

82. SUN NFS esetén mi a különbség a soft, illetve hard mount között?

83. Milyen célokat tüzek ki a SUN NFS tervezői, és milyen típusú megvalósítást választottak?

84. Milyen protokollokat használ a SUN NFS? Adja meg ezek funkcióját!

85. Adja meg a SUN NFS szoftver komponenseit és azok feladatát!

86. Milyen információkat tartalmaz egy RPC-kérés-, illetve válasz-üzenet?

87. Mi a szerepe a virtuális fájlrendszernek az NFS esetében?

88. Hogyan hajtódik végre egy *write* művelet távoli állomány esetén?

89. Mit ls hogyan definiál a POSIX-szabvány?

90. Milyen kategóriákat definiál a POSIX-szabvány az egyes előírások megvalósításának besorolására?

91. Az alkalmazás oldaláról milyen megfelelőségi kategóriákat különböztet meg a POSIX?

92. Hogy definiálja a felhasználási környezetet a POSIX?

93. Hogyan támogatja a POSIX a hordozhatóságot?

94. Milyen eltéréseket mutat a POSIX a könyvtárak kezelésében a hagyományos UNIX-megvalósításokhoz képest?

95. Mi a POSIX által a folyamatkezelésnél definiált session funkciója?

96. Mi a hasonlóság és különbség a System V, a BSD, illetve a POSIX-szignál kezelésében?

97. Milyen változást hozott a POSIX-terminál kezelése a hagyományos UNIX-rendszerekhez képest?

98. Milyen komponensekből tevődik össze a Linux?

99. Hogyan jelentkezik a modularitás a Linux kernelben?

100. Melyek a Linux rendszerkönyvtárainak funkciói?

101. Hogyan támogatja a Linux a többszálú (multithreaded) programozást?

102. Mire szolgál a Linuxban a socket interfész?

103. Milyen virtuális memória és fájlkezelést támogat a Linux?

7.6. Windows NT operációs rendszer

1. Melyik az az alrendszere az NT-nek, ami nélkül nem tud futni?

2. Ismertesse az NT objektumainak általános jellemzőit?

3. Sorolja fel az NT hardverfüggő rétegeit!

4. Mi volt a fő oka annak, hogy a 4.0-s NT-ben a képernyőkezelő és grafikus funkciókat megvalósító rendszerkomponensek kernel módba kerültek?

5. Mi az NTDLL.DLL fő funkciója és milyen műveleteket hajt végre?

6. Nevezzen meg egy kliens–szerver-modell alapján működő komponenst az NT-ben.

7. Mi a fő különbség egy Win32-es alkalmazás és egy szolgáltatást megvelósító folyamat futása között az NT-ben?

8. Milyen operációs rendszer megvalósítási ideológiát tükröz az NT felépítése?

9. Sorolja fel az NT Executive rétegének fő funkcióit?

10. Mondjon legalább három példát arra, hogy milyen többletszolgáltatást érhet el egy Win32-es programozói interface-t (felületet) használó alkalmazás egy POSIX-es alkalmazáshoz képest?

11. Miért van két változata a string paramétert is használó a WIN32-es API hívásoknak?

12. Miért kerültek a Win32-es alrendszer egyes részei (például ablakkezelés) kernel módban megvalósításra?

13. Melyik az a tulajdonsága a Windows 95/98-nak, amit az NT-s rendszerek sohasem fognak megvalósítani?

14. Milyen előnyei vannak a UNICODE használatának?

15. Ismertesse a NT alrendszereinek szerepét! Hogyan működnek együtt az alrendszerek a hozzájuk tartozó alkalmazásokkal? Példaként ismertesse, hogy egy Win32 API-ban definiált hívás az NT mely komponensében lehet megvalósítva?

16. Mutassa be részletesen az NT felépítését! Milyen szerepe és funkciója van a rendszerben a Kernel és a HAL (Hardware Abstraction Layer) rétegeknek?

17. Sorolja fel, milyen interface-eket ismer az NT-ben! Részletesen ismertesse az egyes interface-ek szerepét és a köztük lévő különbségeket!

18. Ismertesse, hogy milyen szolgáltatásokat és rendszer folyamatokat ismer az NT-ben és mi az egyes részek funkciója! Milyen operációs rendszer felépítési elvet tükröz az NT ezen részeinek megvalósítása? Rajzolja fel, hogy hogyan épül fel az NT szolgáltatásokat és rendszer folyamatokat megvalósító része!

19. Mit jelent az NT file rendszerének (NTFS) megvalósításakor használt „Mindent vagy semmit" elv?

20. Mi a különbség egy file attribútum rezidens és nem rezidens tárolása között az NT-ben?

21. Ismertesse az NT file rendszere elé állított legfontosabb tervezési követelményeket! Részletesen ismertesse, hogyan kerültek megvalósításra az egyes követelmények!

22. Mutassa be, hogy az NT hogyan tárolja az file-okat, illetve a könyvtárakat a lemezen! Definiálja az NTFS metadata és a Master File Table fogalmát! Rajzolja fel, hogy hogyan tárolja az NT az adatokat a file record-ban!

23. Sorolja fel az NT memória kezelésének legfontosabb jelemzőit!

24. Hogyan történik a memória foglalása az NT-ben?

25. Hogyan valósítja meg az NT az memóriaterületek osztott elérését?

26. Milyen szinteken történik a memóriavédelem implementálása az NT-ben?

27. Mi a különbség egy file attribútum rezidens és nem rezidens tárolása között az NT-ben?

28. Hogyan optimalizálja a copy-on-write módszer a memóriahasználatot?

29. Hogyan kezeli a memórialapokat az NT?

30. Milyen a folyamatok logikai címterének a felépítése az NT-ben?

31. Adja meg az NT által használt logikai cím felépítését!

32. Milyen adatszerkezeteket használ a logikai-fizikai címtranszformáció során az NT x86-os processzorok esetén?

33. Milyen biztonsági szolgáltatásokat nyújt az NT?

34. Sorolja fel az NT biztonsági alrendszerének részeit!

35. Ismertesse, hogyan történik az objektumok védelme az NT-ben?

36. Ismertesse a logon menetét az NT-ben!

37. Hogyan kezeli az NT az interruptokat? Adjon példát, milyen interruptokat kezel a rendszer x86-os processzor esetén!

38. Ismertesse a trap kezelő működését!

39. Mutassa be az NT objektumkezelésének alapjait!

40. Milyen szinkronizációs lehetőséget kínál az NT a folyamatok számára?

41. Hogyan történik a kernel és az executive réteg folyamatainak szinkronizálása?

42. Mi a lokális eljáráshívás *(Local Procedure Call)* szolgáltatás működésének lényege, és az NT mely részei használnak LPC-t?

43. Milyen típusai léteznek az LPC-hívásoknak?

44. Ismertesse az NT folyamatmodelljét!

45. Milyen adatstruktúrákat használ az NT a folyamatok leírásásra?

46. Mutassa be, hogyan történik a folyamatok létrehozása az NT-ben!

47. Milyen adatstruktúrákat használ az NT a szálak leírásásra?

48. Ismertesse egy szál létrehozásának lépéseit az NT-ben!

49. Milyen elvek alapján történik a szálak ütemezése az NT-ben?

50. Mit nevezünk kvantum-nak, és mi alapján számolja ki az NT a kvantum hosszát?

51. Milyen prioritási szintek vannak az NT-ben?

52. Mutassa be a szálak állapotátmenet diagrammját!

53. Mit nevezünk processzor affinitásnak, és mi a szerepe ütyemezéskor?

54. Hogyan választja ki az NT egy szál ütemezésekor, hogy melyik processzoron fog futni?

Tárgymutató

Tartalom

A
B
C, CS
D
E, É
F
G, GY
H
I
J
K
L
M
N
O,Ö
P
R
S, SZ
T
U
V
W
X
Z
/proc állományrendszer 343

2000. év probléma 55

50%-os szabály 165

A

ablakkezelés 30, 52, 209

acces time *lásd:* adatelérési idő

access control list, ACL (elérési lista) 420, 436

access token 437

Ada nyelv 106–107

adatátviteli protokoll 119

adatelérési idő (acces time) 112

adatfolyamok illesztése 103

adatkapcsolati réteg 218–219

adatmentés 205

adattárolás megbízhatósága 205

adattartomány 348

adatterület 110

adattömörítés (data compression) 205

aging *lásd:* öregítés

aktivitás naplózás (audit logging) 127

aktuális könyvtár 334

alkalmazásfejlesztő 119, 231

alkalmazási

– profilok 374

– réteg 218

alkalmazói

– felület (application interface) 197

– programok 17, 18, 19, 31, 44, 52, 53

állapot nélküli (stateless) szolgáltató 250

állapotátmeneti gráf 141, 143

állapotmodell 138, 141

állapotot tároló (statelfull) szolgáltató 249

állományok tárolása 188–189

alrendszer 399

– OS/2 399

– POSIX 399

– Win32 394, 399

általános célú

– állományrendszer 342

– számítógép 130

alternatíva adatszerkezet 107

alvó állapot 287

Amoeba 213

Application Programming Interface, API *lásd:* programozói interfész

áramkörkapcsolás 220

árva folyamat 287

ASCII 254

asszociatív tár (associative registers, translation look-aside buffer) 171

aszimmetrikus rendszer 135

aszinkron be/kivitel 132

átbocsátó képesség (throughput) 148, 203

átfedő programrészek (overlay) 161

áthelyezhető kód (relocatable code) 158

átlapolt tárcsere (overlapped swaping) 166

átlátszóság (transparency) 28, 229, 233–234

átmeneti tár *lásd:* gyorsítótár

átmenthető és visszaállítható (preemtable) erőforrások 85

átviteli idő (transfer time) 202

audit logging *lásd:* aktivitás naplózás

auditálás 435, 437

azonosítás (authentication) 271, 275

B

B/K (bemeneti/kimeneti)

– készülékek 37, 38, 52, 54, 55

– műveletek 48, 49, 50, 61

B/K

– leíró (IOCB, input-output control block) 144

– löket (I/O-burst) 138–139

– művelet végrehajtásának menete 145

– -intenzív folyamat 138, 140

– -rendszer kezelő 397

bakery (pékség) algoritmus 72

bankár algoritmus 95

batch *lásd:* kötegelt

bázisrelatív címzés 159

beágyazott rendszerek 56

Bélády-anomália 179

belső

– prioritás 151

– szinkronizáció 258

– tördelődés (internal fragmentation) 164

Berkeley algortimus 261

bittérképes ábrázolás 190

biztonság 270, 271, 273

biztonsági

– alrendszer (monitor) 397

– referencia monitor *lásd:* Security Reference Monitor, SRM

– témaszám kezelő *lásd:* Security Accounts Manager, SAM

biztonságos állapot 94–98

biztonságpolitika 213, 271–272

blokk buffer cache 342

blokkok átmeneti tárolása 205

blokktábla (block map table) 168

boot *lásd:* rendszerindítás

boot blokk 324

buffer *lásd:* puffer

buffer-cache *lásd:* gyorsítótár

bully algoritmus 267–268

bus *lásd:* sín

C, CS

cache *lásd:* gyorsítótár

call-out 302

Chorus 213

chroot rendszerhívás 227

CIFS (COmmon Internet File System) 244

ciklus lopás (cycle stealing) 132

cilinder csoport 336

címtér 284

címzett elérés 333

Clouds 213

cluster 430

cluster, Logical Cluster Number (LCN) 430–434

cluster, Virtual Cluster Number (VCN) 430–434

context switching: *lásd* környezetváltás

continue művelet 106–107

copy-on-write 361, 421–422

CPascal 103, 106

CPU

– kihasználás 138

– tétlenség 182

– ütemezés 129, 140, 147

– -burst: *lásd* processzor löket

– -intenzív folyamat 138, 140

– -tétlenség 142

create utasítás 65

CreateProcess 410

CreateThread 412

Cristian algoritmus 261

csatorna 78, 79

csavart érpár (UTP) 218

CSCW (Computer Supported Cooperative Working) 229

csere (swap) utasítás 74

csomagkapcsolás 221

csonk (stub) 160

csoportazonosító 289

csoportkommunikáció 78, 80

csoportos multicast 237

csoportvezető 311

csővezeték (pipe) 118, 312

CSP (Communicating Sequential Processes) modell 107

D

DCE (Distributed Computing Environment) 254, 256

deadlock *lásd:* holtpont

delay művelet 106

delayed write 248

démon folyamatok 213, 226, 247–248, 251, 368

DES 277

determinisztikus modellezés 155

device driver *lásd:* készülékkezelő

dialup terminal connection *lásd:* soros vonali terminálhozzáférés

Dinamically Linked Library, DLL 399

dinamikus betöltés 160

dinamikus prioritás 151

directory *lásd:* katalógusfájl

direkt blokk 329

diszktérkép 363

diszkblokk leíró 353

DMA, direct memory access *lásd:* közvetlen memória hozzáférés

domain 234–235

DP (Distributed Processes) modell 107

dup 329

E, É

effektív azonosító 289

egyidejűség (randevú) 69, 82

egyirányú pásztázó algoritmus (c-scan) 204

egypartíciós rendszer 163

egyszeres indirekt blokk 329

egyszerű ütemezési algoritmusok

– legrégebben várakozó (FCFS, first come first served) 150

– körbeforgó (RR, round robin) 150

egyszintű megszakítási rendszer 131

éhezés 99, 100

electronic signature *lásd:* elektronikus aláírás

elektronikus aláírás (electronic signature) 127

elektronikus diszk 109

elérési lista *lásd:* access control list, ACL

elérési mátrix 120, 122

elérési token *lásd:* access token

elfordulási idő (latency time) 202

elhelyezkedés-függetlenség (location independence) 244

eljáráshívás 262

elnevezés 245

elnevezett csővezeték 313

előrelapozás (prepaging) 185

előretekintő lapozás (anticipatory paging) 178, 352

eloszott

– számítási modell 251

– fájlkezelési stratégia (consistency semantics) 249

– fájlrendszer 243–244, 246

– kölcsönös kizárás 265

– koordináció 265

– óra 259

– rendszer 27, 28, 213, 228–230, 235–236, 238, 258

– szoftver 228

előtérben futó csoport 311

első megfelelő algoritmus (first fit) 165

elsődleges időszolgáltató 262

e-mail 127, 227

encryption *lásd:* kódolás

EPROCESS *lásd:* folyamatblokk

érkezési sorrend szerinti (FIFO) ütemezés 77

erőforrás

– kihasználás 138

– menedzser 230

– -foglalási gráf 86, 98

– -gazdálkodás 65, 137

erőforráskiosztás 137

– -megosztás 229

érvényességi bit 354

esemény 75, 76

események sorrendezése 257

eseménynaplózó *lásd:* event logger

esményvezérelt rendszer 131

ETHREAD *lásd:* szálblokk

étkező filozófusok problémája 102

event logger 399

exception *lásd:* kivétel

executive 397

executive objektum 405

exit utasítás 65

exokernel 46

external fragmentation *lásd:* külső tördelődés

F

fájl

– allokációs tábla 192
– -műveletek 111–112, 117
– -ok többszörözése (replication) 250
– rekord 431–434
– rekord, nem rezidends tárolás 434
– rekord, rezidends tárolás 433–434
– -rendszer, fájlkezelés 32, 47, 56
– -szerkezet 193–194
– vándorlás (file migration) 244–245
farmer-worker modell 80
FCFS, first come first served ütemezési algoritmus 150
fejmozgási idő (seek time) 202
felfüggesztett állapot 288
felhasználóazonosító 289
felhasználóbarát virtuális gép 129
felhasználói (user) mód 36, 55
felhasználói felület 243
felhatalmazás (authorization) 271
FIFO 78
file *lásd:* fájl
fill-from-text 355
fill-on-demand 355
fizikai
– címtartomány (physical address space) 157
– idő 242
– memória keret (frame) 170
– processzor 137

– réteg 218
fogad (receive) művelet 67, 69, 77, 78, 79, 81, 82, 237
foglalva várakozás (busy waiting) 93, 146
folyam 319
folyamat
– -ok állapotátmenete 141
– állapotok 141, 287
– azonosító (process ID) 408
– -ok befejezése 292
– -ok betöltése a háttértárról 350
– blokk 409–411
– csoport azonosító 311
– -ok felfüggesztése 140, 142
– folyamattábla 280
– -ok háttértárra írása 347
– kénti állományleíró tábla 331
– környezet 286
– -ok közötti kommunikáció 312
– -ok lapigénye 182
– -leíró (PCB, process control block) 143
– -leíró tartalma 143
– -leírók láncolása 144
– -ok létrehozása 291
– -modell 136
– -ok normál befejeződése 139
– szálkezelő 397
forgalomirányítás (routing) 218–220

fork utasítás 65

fork-join operátor 105

ftp (file transfer protocol) 225–227, 244

függvényszállítás 237–238

futási környezet 18

G, GY

globális

– állománytábla 332

– lapcsere stratégia 182

– tábla 122–123

GOES 257

GPS 257, 259

graceful degradation 150

grafikus

– eszközök 117

– munkahely 208

groupware *lásd:* CSCW

guarded command *lásd:* őrzött parancs

gyermek folyamat 287

gyökér könyvtár 334

gyorsítótár (cache) 108–109, 112, 133, 200, 205, 242, 247–249, 252, 397

gyűrű alapú algoritmus 266

H

HAL (Hardware Abstraction Layer) 395, 396

hálózati

– réteg 218–219

– számítási modell 251

– szolgáltatások 364
– topológia 214–217
handle 405, 409
hard link 334
hardver driver 397
hardver érvényességi bit 363
hardverredundancia 233
háromszoros indirekt blokk 329
háttértár
– felszabadítás 347
– foglalás 347
– használat tábla 353
– szervezés 346
– kezelés 200–201
helyi
– biztonsági jogosultság ellenőrző *lásd:* Local Security Authority, LSA
– hálózat (local area network, LAN) 218, 262
hiba
– buffer 364
– -megszakítások *lásd:* kivételkezelés
– -tűrés 213, 229, 233, 238, 259
hierarchikus fa 334
hierarchikus névtér 236
hitelesítés 127
hitelesítő 289
hivatkozás bit 354, 362
– szimulálása 362

hivatkozásszámláló 362
hivatkozott bit 178, 181
holtpont (deadlock) 71, 82, 83, 84, 86–94,
98–99
– elkerülés 88–89, 94
– megelőzés 88, 92
homogén rendszer 135
hosszú állománynevek 338
hosszú távú ütemezés 140
hozzáférés
– koordinálás 129
– -ek szabályozása (access control) 169, 197
– -i jogok 315

I

I/O burst: *lásd* be/kiviteli löket
ideiglenes állomány 342
időbélyeg (time stamp) 248
időbeni lokalitás 184
időkezelés 54, 55, 257
időszelet (time slice) 150
igény szerinti lapozás (demand paging) 178, 352
in-core inode 328
indexelt elérésű (ISAM) fájl 111
indítóprogram (bootsrap program) 131
inhomogén rendszer 135
inode 328
inode lista 324

inode tábla 332

interaktív

– működés 34, 64

– rendszerek 52, 243

interfész (interface) 42, 253, 282

internal fragmentation *lásd:* belső tördelődés

Internet 127, 213, 219, 252

interrupt, IT *lásd:* megszakítás

IOCB (Input Output Control Block) 116

IP-cím 223

IP-protokol 219

írók–olvasók problémája 101

ISAM *lásd:* indexelt elérésű

J

Java nyelv 107

jegyosztó - TGS (Ticket Grantig Service) 277–278

jelez (Signal) művelet *lásd:* V művelet

jelszó 125–127, 226–227

jelzés 304, 312

– kezelés 306

job *lásd:* munka

job pool 25

job-mix 140

jogosítvány 120–123

jogosultságok szabályozása 169–170, 197

journaling 344

K

kapcsolt hálózat 217

kapu (port) 79, 81, 223

katalógusfájl (directory) 110

Kerberos hitelesítési protokoll 277–278

kernel

– be/kiviteli alrendszere 199

– környezet 286

– (mag) 18–19, 43–46, 53

– mód 285

– (NT) 396

– objektum 405

– primitív 396

– prioritás 295

késleltetett betöltés 160

készülékfüggetlen programozás (device independence) 23, 116

készülékkezelő (device driver) 38, 43, 44, 57, 115, 397, 427–429

– fájlrendszer driver 397

– réteg szerkezetű struktúra 397, 427

– szűrő típusú 397

kétirányú csővezeték 313

kétszeres indirekt blokk 329

kétszerezés (disk shadowing, mirroring) 206

kezelő (operátor) 31

kezelői felület 52–53, 207

kill utasítás 65

kivételkezelés (trap, exception) 47, 55–56, 80, 310, 358, 396, 401–402, 404

kizárolagosan használható (non-preemtable) erőforrások 85, 129

kliens–szerver

– -architektúra 393

– -modell 46, 79, 99, 213, 222, 224, 226, 228, 230, 237–238, 247, 251–252, 255–256, 262, 277

kódolás (encryption) 127

kódtartomány 348

kódterület 110

kölcsönös kizárás (mutual exclusion) 68, 69, 70, 75–76, 82, 84–85, 92, 106, 405

kombinált szegmens- és lapszervezés 173

kommunikáció 104, 236

kommunikációs

– interfész 230

– protokoll 230

konkurencia 229, 231

konkurens Pascal *lásd:* Cpascal

könnyűsúlyú folyamat 285

könyvtár (directory) 194, 227

könyvtárszerkezet 110

konzisztencia 241–242

Koordinált Univerzális Idő (KUI) 257–258, 262

korábban történt reláció 263, 265

körbeforgó ütemezési algoritmus, RR, Round–Robin 150

körkörös várakozás 93

környezetváltás (context switching) 145, 147, 287

körülfordulási idő (turnaround time) 148

kötegelt (batch) feldolgozás 21, 26, 43, 63

kötés (binding) 252

kötet (volume) 110, 429

következő megfelelő algoritmus (next fit) 165

középtávú ütemezés 140

közös memória 65, 66, 68, 70–71

központi

– egység kihasználtság (CPU utilization) 148

– szerver algoritmus 265

– tár 133, 157, 161, 200

közvetlen

– (direkt) kommunikáció 78

– elérésű (direkt) fájl 111

– memória hozzáférés (DMA) 50, 115, 132

kritikus

– régió 106

– szakasz 68–76, 106, 265

KUI *lásd:* Koordinált Univerzáli Idő

kulcs 315

– szétosztás 275

küld (send) művelet 67, 69, 77–79, 81–82, 237

külső

– prioritás 151

– szinkronizáció 258

– tördelődés (external fragmentation) 165

kvantum 413

kvóta mechanizmus 339

LAN *lásd:* helyi hálózat

L

lap (page) 170

lapcsere stratégiák 178, 179–182

laphiba 175–176, 178–179, 358

– gyakoriság (PFF, page fault frequency) 183, 185–186

laplopó folyamat 357

lapméret 186

lapok tárba fagyasztása (page locking) 181, 187

lapozott memória pool 363

lapszervezés (paging) 170–171

laptábla 353–354

layer *lásd:* réteg

lazán csatolt rendszer 28, 135

legjobban megfelelő algoritmus (best fit) 166

legkevésbé használt algortimus (LFU, least frequently used) 181

legrégebben

– használt 354

– nem használt algoritmus (LRU, least recently used) 180

legrégebbi lap algortimus (FIFO) 179

legrosszabban illeszkedő algoritmus (worst fit) 166

legrövidebb fejmozgási idő algoritmus (SSTF, shortest seek time first) 203

lehallgatás 272

leképezési táblák 246

lemezegységek többszörözése 206

lemezműveletek ütemezése 203

lemezterület tömörítése 204

létrehozó 315

link 334

Linux 384

local area network *lásd:* helyi hálózat
Local Procedure Calls, LPC 401, 437
Local Security Authentication Server (LSASS) 398, 407
Local Security Authority, LSA 435, 438, 438
lockd 265
locking 406
lock-key *lásd:* zár-kulcs
log fájl 427
logikai
– címtartomány (logical address space) 157
– csatlakozó (socket) 118, 318
– memória 137
– óra 243, 263–265
– periféria 113, 116
– processzor 137
logikai-fizikai címleképezés 158–159, 169–174
login 223
logon 398, 434, 436, 438
lokális
– állományrendszer 342
– eljáráshívás *lásd:* LPC (Local Procedure Call)
– lapcsere stratégia 182
lokalitás 109, 175, 184–185
LPC (Local Procedure Call) 398, 399, 407, 438

M

Mach 213
mag *lásd:* kernel

mágneses diszk 133

mágneslemez 56

– egység fizikai felépítése 201–202

– szektorok címzése 202

map 346

másodlagos

– időszolgáltató 262

– tár 133

másolás írás esetén bit 354

maszkírozás 272

megbízhatatlan jelzések 307

megbízható jelzések 308

megbízó levél (credential) 277

megjegyzett inode 325

megjelenítési réteg 218

megnevezés 77–79, 235

megszakítás 47–48, 50, 54–55, 57, 131, 145

megszakítás (interrupt, IT) 396, 401–403

meleg tartalék 233

memória 137

– állományrendszer 343

– -ba ágyazott állomány 353

– címtranszformáció 424

– foglalás 418, 422

– foglalás, commit 418

– foglalás, reserve 418

– méret 422

– osztott elérésű 419

– térkép 363

– védelem 420, 435

– logikai címtaromány 423

– kezelő 352, 417

Message Passing Interface *lásd:* MPI

mesterséges folytonosság (artificial continuity) 167

MFT (Master File Table) 431–432

middleware 251

mikrokernel 213, 393

modem 119

módosított bit 178, 181, 354

Modula nyelv 106

modulok 37, 42–43

monitor 22, 106

mount 245

mount protokoll 367

MPI (Message Passing Interface) 107

multicast 262

MULTICS 26

multimédia 115

multiprocesszálás 46, 59–61, 74, 135, 137, 240

multiprocesszoros rendszerek *lásd:* multiprocesszálás

multiprocesszor-ütemezés 396

multiprogramozás 24–25, 35, 46, 50, 53, 58, 59, 60–64, 70–71, 82, 84, 104, 129, 137, 352

multiprogramozás foka 137–138, 140

Multitaszkos (time-sharing, multitasking) rendszerek 26, 34, 52–53, 58, 62, 104, 229

munka (task, job) 59, 62–64, 140

munkaállomás-szerver modell 239

munkahalmaz (working set) 184

munkamenet-kezelés 311

mutual exclusion *lásd:* kölcsönös kizárás

műveletek könyvtárakon 195–196

N

N lépéses pásztázó algoritmus (n-scan) 204

nagy területű hálózat (wide area network – WAN) 218

naplózás 344

naplózó állományrendszer 344

Needham Schroeder hitelesítő szerver 276–277

need-to-know szabályozás 120

nem lapozott memória pool 363

nem preemptív ütemezés 147

Nemzetközi Atomidő 257

Network Lock Manager 368

Network Status Monitor 368

Network Time Protocol 262

Neumann struktúra 20, 64, 130

névfeloldás (name resolution) 235–236

New Technology (NT) 389

NFS (Network File System) 244–245, 251–253, 265

NFS kliens kód 368

NFS protokoll 367

NFS-szerver 368

nice 296

NIS (Network Information System) 253

non-preemtable *lásd:* kizárolagosan használható

NTDLL.DLL 397, 399, 401

NTFS (Windows NT File System) 425

NTFS metadata 430

nyílt rendszerek 29

nyitottság 229–231

nyomkövetés 314

nyomtatás 68

nyomtató 120, 214

O,Ö

objektumelérési lista 122–123

objektumkezelő 404

objektumorientált szemlélet 394

off-line perifériás műveletek 21–22, 24

OlvasÉsIr (TestAndSet) utasítás 73

on-line kapcsolat 25

open 329

operátor *lásd:* kezelő

optikai diszk 133

optimális algoritmus 179

óra 54–55, 242, 258–261, 263–264

– algoritmus (clock) 180

– -csúszás 260

– -szinkronizáció 261

öregítés (aging) 152

öregítés bit 354

őrzött parancs 107

OS/2 alrendszer 399

OSI Referencia Modell 218–219

osztott

– -an használt tartomány 348

– fájlkezelés 196

– memória 240, 317

– szegmenshasználat (segment sharing) 169

overlay *lásd:* átfedő programrészek

P

P művelet 75–76

paging *lásd:* lapszervezés

Parallel Virtual Machine *lásd:* PVM

parancsértelmező (shell) 44, 52–53, 222–224, 227

parbegin-parend utasítások 106

párhuzamos (parallel) rendszerek 28, 46

párhuzamos végrehajtás 104–106

partíció (partition) 162

pásztázó algoritmus (scan) 203

PCB, process control block *lásd:* folyamat leíró

perifériafüggetlen programozás *lásd:* készülékfüggetlen programozás

perifériás művelet 132

pfdata 353

pipe *lásd:* csővezeték

Plug and Play 57, 393

polling *lásd:* programozott lekérdezés

port 235, 252

POSIX szabvány 36, 304, 374, 390, 399, 404, 429, 436

postaláda 77–79, 81, 118

pozíciófüggetlen kód (PIC, position independent code) 159

PRAM-modell 66–67, 69–70, 72–75, 92, 109

precedencia 69, 75–76, 82

precedenciagráf 105–106

preemptív ütemezés 142, 147

prioritás (priority) 151, 412, 414

prioritásos ütemezés 77, 293

prioritásos ütemezési algoritmusok

– legrövidebb löketidejű (SJF, shortest job first) 151–152

– legrövidebb hátralevő idejű (SRTF, shortest remaining time first) 151–152

– legjobb válaszarány (HRR, highest response ratio) 151–152

privilegizált utasítás 45, 55

proc struktúra 290

processzgráf 65

processzor

– -affinitás 412, 416–417

– állományrendszer 344

– -löket (CPU-burst) 138–139

– pool modell 240

program címeinek kötése 157

programozói

– felület *lásd:* programozói interfész

– interfész (API) 392, 400

programozott lekérdezés (polling) 115, 116

protokoll 70, 103, 214, 253

proxy 213, 252

puffer (buffer) 23, 56, 82, 100–101

PVM (Parallel Virtual Machine) 107

R

RAID (Redundant Array of Inexpensive Disks) technika 206, 429

RAM modell 60, 65, 109

RAM-diszk 109, 342

randevú *lásd:* egyidejűség

real-time *lásd:* valós idejű

régen használt 354

reguláris állomány 329

rejtett elhelyezkedés (location transparency) 244

rejtjelezés *lásd:* kódolás

remote procedure call *lásd:* távoli eljáráshívás

Remote Procedure Call (RPC) 398, 400

rendszer (system) mód 36, 54

rendszer védelem (system protection) 271

rendszeradminisztráció 120

rendszeradminisztrátor 32, 62, 119, 127, 231

rendszerfolyamat 398

rendszergazda *lásd:* rendszer-adminisztrátor

rendszerhívás (system call) 35, 44–45, 47–50, 52–53, 61, 69, 75, 131

rendszerindítás (boot) 56, 64

rendszerleállítás (shut down) 56

rendszermenedzser *lásd:* rendszer- adminisztrátor

rendszerprogramok 17–19, 44, 46, 53

rendszertervező 68

replikáció 252

request-reply protokoll 237

réteg (layer) 18, 42–44, 238–239

– szerkezet 282

rlogin parancs 234

rögzített partíciók (fixed partition) 164

rollback *lásd:* visszaállítás

root 334

routing *lásd:* forgalomirányítás

rövid távú ütemezés 140, 147

RPC (Remote Procedure Call) *lásd:* távoli eljáráshívás

rpcgen 255

RPC-szál 256

RR, Round–Robin ütemezési algoritmus 150

S, SZ

s5fs 323

Secure Attention Sequence (SAS) 398

Security Accounts Manager, SAM 435, 437, 438

Security Reference Monitor, SRM 435, 436, 437

select művelet 118

send *lásd:* küld

sequential *lásd:* soros elérésű

Service Controller 398, 399

Session Manager 398

shell *lásd:* parancsértelmező

shut down *lásd:* rendszerleállítás

signal *lásd:* jelez

sín (bus) 217

sínvezérlő (arbiter) 136

skálázhatóság 217, 229, 231

SMB (Server Message Block) 244

socket *lásd:* logikai csatlakozó

sorállási modell 138–139, 143

soros

– elérésű (sequential) fájl 110

– vonali terminálhozzáférés (dialup terminal connection) 221

sorrendi kiszolgálás 150, 203

spinlock 406

spooling 24

stack *lásd:* verem

start művelet 116

statikus prioritás 151

stop művelet 116

stratum 262

stream 118

strucc algoritmus 87

Sun NeWs abalakozó rendszer 238

SUN NFS 366

superuser 289

SV IPC 315

swap device 346

swapping *lásd:* tárcsere

szabad lemezblokkok listája 326

szál (thread) 58–59, 61–62, 69, 255–256, 285, 408

– azonosító (thread ID) 408
– ütemezés 396, 412
szálblokk 409–411
szállítási réteg 218
szegmensszervezés 168–169
szekvenciális elérés 133, 333
szemafor 75–76, 99, 106, 120, 146, 316
szerver folyamat 107
szimbolikus link 338
szimmetrikus rendszer 135
szinkron be/kivitel 132
szinkronizáció 67–68, 70, 75–77, 82–84, 99, 104, 106, 231, 236, 243, 304, 401, 405–407
szisztematikus támadás 125
szoftver érvényességi bit 362
szoftverfelépülés 233
szorosan csatolt rendszer 28, 135
szülő folyamat 287
szülő-gyermek viszony 65
szuperblokk 324

T

találati arány (hit ratio) 172
támadó 272
tápkimaradás 113
tárcsere (swapping) 56, 163–164, 166–168, 345–346
tárhierarchia 134
tárkezelés 157
tárolt másolat 355

tartománynövelés 349

tartománytábla 348

task *lásd:* munka

távoli állományrendszer 342

távoli eljáráshívás (Remote Procedure Call) 247–248, 251–256

távoli fájlrendszer 245

TCP protokoll 219, 251

TCP/IP protokoll család 218–219, 221–223, 226, 251, 253, 278, 365

teljes folyamatok háttértárra írása 345

telnet 221–225, 244

térbeli lokalitás 184

terhelés

– elosztás 213

– ingadozás 140

– kiegyenlítés 140

– kiosztás 235

– szétosztás 239

termelő-fogyasztó probléma 100–101

terminálkezelés 53, 118

TestAndSet *lásd:* OlvasÉsIr

thread *lásd:* szál

threat monitoring *lásd:* veszélyeztetett pontok figyelése

time sharing *lásd:* multitaszkos

titkosítás (encryption) 271, 274

tmpfs 343

többfelhasználós rendszerek 229–230

többpartíciós rendszer 163

többprocesszoros rendszerek *lásd:* multiprocesszálás

többszintű megszakítási rendszer 131

többszintű ütemezési algoritmusok

– statikus többszintű sorok (SMQ, static multilevel queues) 152–154

– visszacsatolt többszintű sorok (MFQ, multilevel feedback queues) 152–154

token ring *lásd:* vezérjeles gyűrű

tömegkiszolgálási modell 138

töredék 337

transparency *lásd:* átlátszóság

tranzakciónkénti feldolgozás 426

trap *lásd:* kivétel

trashing *lásd:* vergődés

trönk 214

tulajdonos 315

U

UDP protokoll 219, 251

UDP/IP 251, 253

új folyamat indítása 138, 140

újabb esély algoritmus (second chance) 180

újrahívható kód (reentrant code) 173

újraütemezés 142

UNICODE 390, 429

uróbbi időben nem használt algortimus 181

user *lásd:* felhasználó

user mód 285

user prioritás 295

ütemezés 43, 44, 77, 99, 292, 408, 412

ütemezés többprocesszoros rendszerben 156

ütemezés, preemptív 412

ütemezési algoritmusok 150–154

– jóságának értékelése 154–155

– jósolható viselkedése 149

– összehasonlítása 148–149

– -kal szemben támasztott követel-mények 149

ütemezéssel szembeni elvárások 293

u-terület 290

UTP *lásd:* csavart érpár

útvonalválasztás *lásd:* forgalomirányítás

üzenetek 64, 65

üzenetkapcsolás 220

üzenetküldés 236

üzenetmódosítás 272

üzenetsor 118, 317

üzenetszórás (broadcasting) 78, 80, 237

üzenetszórás (datagram) 319

üzenetváltás 67

V

V művelet 75, 76

V rendszer 213

válaszidő 149, 203

választási algoritmusok 266–270

valódi azonosító 289

valósidejű (real-time) rendszerek 29, 54, 56, 58, 63, 64

változó méretű partíció (variable partition) 165

vandalizmus 272

vár (Wait) művelet *lásd:* P művelet

várakozási idő 149

várakozási sor 137

várt esemény bekövetkezése 142

védelem 134, 239

védelem bit 354

védelmi rendszer 54

védelmi tartományok 120–123

védett közös (shared) változók 106

végrehajtási környezet 284, 286

véletlen hozzáférésű memória (random access memory) 133

verem (stack) terület 110

veremtartomány 348

vergődés (trashing) 183, 357

veszélyeztetett pontok figyelése (threat monitoring) 127

vezérjeles gyűrű (token ring) 216

vezérlési szál 60

virtuális

– állományrendszer 339

– áramkör 220

– csomópont 339

– gép (virtual machine) 19, 31, 42, 45–46, 58, 107, 109, 112–113, 285

– hardver 45–46

– memória *lásd:* virtuális tár

– memória kezelő 397

– párhuzamosság 232

– tár, virtuális memória 32, 43, 55, 56, 109, 112, 117, 174–177, 352, 418

– terminál 221–225

vírus 271

visszaállítás (rollback) 56, 91–92, 233, 272

viszony objektum 311

viszony réteg 218

viszonykulcs (session key) 277

volume *lásd:* kötet

W

wait művelet 116

WAN *lásd:* nagy területű hálózat

watchdog processzor 134

webböngésző 252

webszerver 251

wide area network *lásd:* nagy területű hálózat

Win32 alrendszer 394, 399

Win32 API 390, 392, 400, 401, 404, 437

Windows 2000 390

Windows 5.0 390

Windows 95 391

Windows 98 391

Windows CE 392

WWV 257

X

X protokoll 208

X terminál 240

X Window rendszer 207, 209

XDR (Extended Data Representation)protokoll 253–255, 367

Z

zár-kulcs (lock-key) módszer 123

zero-fill 355

zombi folyamat 350

Irodalom

Mach: A New Kernel Foundation for UNIX Development M., Baron, R., Bolosky, W., Golub, D.B., Rashid, R., Tevanian, A., Young, M.Accetta -Proceedings of the Summer 1986 USENIX Conference June 1986 pp 93-112

A File System for Continouos Media, ACM Trans. D., Osawa, Y., Govindan, R.Anderson -on Computer Systems, vol. 10, no. 4 1992 pp. 311-337

Concurrent Programming Principles and Practice G.R.Andrews -Redwood City CA: Benjamin Cummings 1991

Converting a Swap-Based System to a Do Paging in an Architecture Lacking Page-Rreferenced Bits, O., Joy, W.N.Babaoglu -Proceedings of the 8th ACM Symposium on Operating System Principles December 1981 pp. 78-86

The Design of the Unix Operating system, M.J.Bach -Prentice-Hall, Englewood Cliffs, NJ 1986

Concurrent Systems J.Bacon -Wokingham, Addison-Wesley 1993

A Multi-Service Storage Architecture J.M., Moody, K., Thompson, S.E., Wilson, T.D.Bacon -ACM Operating Systems Review, vol. 25, no. 4 1991

Operációs rendszerek I – II T., Zsadányi, P.Bakos -SZÁMALK Kelenföldi Kiadó 1992

Linux Kernel Internals M., Boehme, H., Dziadzka, M., Kunitz, U., Magnus, R., Verworner,D.Beck -Addison Wesley Longman 1996

An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine L.A., Nelson, R.A., Shedler, G.S.Belady -Communications of the ACM, Vol. 12, No. 6 June 1966 pp. 349-353

A Study of Replacement Algorithms for Virtual Storage Systems L.A.Belady -IBM Systems Journal, Vol. 5, No. 2, 1966 pp. 78-101

Principles of Concurrent and Distributed Programming M.Ben-Ari -Prentice-Hall, Englewood Cliffs, NJ 1990

Operációs rendszerek alapjai Kondorosi, K., Sziray, J.Benyó B. -SZIF-UNIVERSITAS Kft, Győr 1998

A UNIX és Windows NT operációs rendszer B. Sziray, J.Benyó -SZIF-UNIVERSITAS Kft, Győr 1999

Konkurens programozás Modula-2 nyelven BME Műszer- és Méréstechnika Tanszék munkaközösségeBME Folyamatszabályozási Tanszék -BME-FOT, BME-MMT 1993

The UNIX Shell S.R.Bourne -Bell System Technical Journal, Vol.57 No 6 July - August 1978 pp 1971-1990

Operating Systems Principles P.Brinch Hansen -Prentice-Hall, Englewood Cliffs NJ 1973

Structured Multiprogramming P.Brinch Hansen -Communications of the ACM, Vol. 15 No 7 July 1972 pp 574-578

The Nucleus of a Multiprogramming System P.Brinch Hansen -Communications of the ACM, Vol. 13 No 4 April 1970 pp 238-241, 250

N-Phylosophers: An Exercise in Distributed Control E.Chang -Computer Networks, Vol. 4, No. 2 Apr. 1980 pp. 71-76

System Deadlocks E.G., Elphick, M.J., Shoshani, A.Coffman -Computing Surveys, Vol. 3 No 2 June 1971 pp 67-78

Virtual Memory P.Collinson -SunExpert Magazine April 1991 pp. 28-34

Introduction and Overview of the MULTICS System F.J, Vyssotsky, V.A.Corbato -Precedings of the AFIPS Fall Joint Computer Conference 1965 pp. 185-196

Distributed Systems – Concept and Design G., Dollimore, J., Kindberg, T.Coulouris -Addison-Wesley 1994

Concurrent Control with 'Readers' and 'Writers' P.J., Heymans, F., Parnas, D.L.Courtois -Communications of the ACM, Vol. 14, No. 10 Oct. 1971 pp. 667-668

Probabilistic Clock Synchronization F.Cristian -Distributed computing, vol. 3 1989 pp. 146-158

Inside Windows/NT H.Custer -Microsoft Press, Redmond, WA 1993

An Introduction to Operating Systems H.M.Deitel -Addison-Wesley, Reading 1990

Timestamps in key distribution protocols D.E., Sacco, G.M.Denning -Communications of the ACM, 24(8) August 1981 533—536

The Working Set Model for Program Behavior P.J.Denning -Communications of the ACM, Vol. 11 No 5 May 1968 pp 323-333

Cooperating Sequential Processes E.W.Dijkstra -Technical Report EWD-123, Technological University, Eindhoven, the Netherlands 1965

Solution of a Problem in Concurrent Programming Control E.W.Dijkstra -Communications of the ACM, Vol. 8 No 9 September 1965 p 569

The Structure of the THE Multiprogramming System E.W.Dijkstra -Communications of the ACM, Vol. 11 No 5 May 1968 pp 341-346

A Continuum of Disk Scheduling Algorithms R., Daniel, S.Geist -ACM Transactions on Computer Systems, Vol. 5 No 1 February 1987 pp 77-92

Working Set and Page Fault Frequency Replacement Algorithms: A Performance Comparison, IEEE Transactions on Computers, Vol. C-27, No 8 R.K., Franklin, M.A.Gupta -pp 706-712 August 1978

The Accuracy of Clock Synchronization Achieved by TEMPO in Berkeley UNIX 4.3BSD, IEEE Trans. R., Zatti, S.Gusella -on Software Engineering, vol. 15 1989 pp. 847-853

Prevention of System Deadlocks A.N.Habermann -Communications of the ACM, Vol. 12 No 7 July 1969 pp 373-377, 385

Towards a Theory of Parallel Porgramming C.A.R.Hoare -In Hoare, C.A.R., Perrott, R.H. Eds.: 1972 Operating Systems Techniques, Academic Press, London

Monitors: An Operating System Structuring Concept C.,A.,R.Hoare -Communications of the ACM, Vol. 17 No 10 October 1974 pp 549-557

Communicating Sequential Processes C.A.R.Hoare -Communications of the ACM, Vol. 21 No 8 August 1978 pp 666-677

Communicating Sequential Processes C.A.R.Hoare -Prentice-Hall International 1985

Some Deadlock Properties of Computer Systems R.C.Holt -Computing Surveys, Vol. 4 No 3 September 1972 pp 179-196

Mixed Solution for the Deadlock Problem J.H.Howard -Communications of the ACM, Vol. 16 No 7 July 1973 pp 427-430

special issue on advanced I/O hardware and software for Electrical and Electronic EngineersInstitute -IEEE Computers, Vol. 27 No 3 March 1994

Information Technology–Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) [CLlanguage] for Electrical and Electronic EngineersInstitute -1003.1-1990, IEEE Dec.1990

Basic Reference Model of Open Distributed Processing, Part I: Overview and Guide to use, ISO/IECJTC1/SC212/WG7 CD 10746-1 Standards OrganizationInternational -International Standards Organization 1992

The Deadlock Problem: An Overview S.S., Marsland, T.A.Isloor -Computer, Vol. 13, No. 9 Sep. 1980 pp. 58-78

The Unix Environment B.W., Pike R.Kernighan -Prentice-Hall, Englewood Cliffs, NJ 1984
The Manchester Unuversity Atlas operating System, Part I: Internal Organization T., Hiwarth, D.J., Payne, R.B., Sumner, F.H.Kilburn -Computer Journal, Vol. 4, No 3 October 1961 pp 222-225
Operációs rendszerek I., Kondorosi, K.Kiss -Műegyetemi Kiadó 1992
Vnodes: An Architecture for Multiple File SystemTypes in Sun UNIX S.R.Kleiman -Proceedings of the Summer 1986 USENIX Technical Conference June, 1986
Queueing Sytems, Volume II: Computer Applications L.Kleinrock -Wiley-Interscience, New York 1975
The Art of Computer Programming (Second Edition) D.E.Knuth -Addison-Wesley, Reading, MA 1973
The Kerberos Network Authentication Service (V5) Neuman, C.Kohl J. -RFC 1510, Digital Equipment Corporation,USC/Information Sciences Institute September 1993
A New Solution of Dijkstra's Concurrent Programming Problem l.Lamport -Communications of the ACM, Vol. 17 No 8 August 1974 pp 453-455
Concurrent Reading and Writing L.Lamport -Communications of the ACM, Vol. 20 No 11 November 1977 pp 806-811
Time, Clocs and the Ordering of Events in a Distributed System L.Lamport -Communications of the ACM, vol. 21, no. 7 1978 pp. 558-565
Dynamic Protection Structures B.W.Lampson -Proceedings of the AFIPS Fall Joint Computer Conference 1969 pp 27-38
The Design and Implementation of the 4.3 BSD UNIX Operating System S.J., McKusick, M.K., Karels, M.J., Quarterman, J.S.Leffler -Addison-Wesley, Reading, MA 1989
Real-Time System Design S.T., Agrawala, A.K.Levi -McGraw-Hill International 1990
The Design of the Venus Operating System B.H.Liskov -Communications of the ACM, Vol. 15 No 3 March 1972 pp 144-149
A Fast File System for UNIX M.K., Joy, W.N., Leffler, S.J., Fabry, R.S.McKusick -ACM Transactions on Computer Systems, vol. 2 August 1984 pp. 181-197
Performance Improvements and Functional Enhancements in 4.3BSD M.K., Karels, M., Leffler, S.J.McKusick -Proceedings of the Summer 1985 USENIXConference, June 1985 pp. 519-531
Java Virtual Machine J., Downing, T.Meyer -O'Reilly and Associates, Sebastopol, CA 1997
A Virtual Machine Time-Sharing System R.A., Seawright, L.H.Meyer -IBM Systems Journal, Vol. 9 No 3 1970 pp 199-218
M.I.T. Project Athena, Cambridge, Massachusetts Neuman C., Schiller J., Saltzer, J.Miller S. E.2.1Section Authentication and Authorization SystemKerberos -December 21,1987
Internet Time Synchronization: the Network Time Protocol D.L.Mills -IEEE Trans. on Communications, vol. 39, no 1991 10, pp. 1482-1493
Object-Oriented Software Construction B.Myer -New York: Prentice Hall 1988
The Processor File System in UNIX SVR4.2 A.V.Nadkarni -Proceedings of the 1992 USENIX Workshop on File Systems May 1992 pp. 131-132
The Cambridge Distributed Computing System R.M., Herbert, A.J.Needham -Workingham: Addison-Wesley 1982
Using Encryption for Authentication in large Networks of Computers R.M., Schroeder, M.D.Needham -Communications of the ACM, 21(12) December 1978 pp.993-999
Names, In Distributed Systems, an Advanced Course, (Ed. Mullender, S.), Workingham: R.M.Needham -ACM Press/Addison-Wesley 1993 pp. 315-326
Operating System – A Modern Perspective G.J.Nutt -Addison-Wesley Publ. Comp 1997
Comment on Deadlock Prevention Method D.L., Habermann, A.N.Parnas -Communications of the ACM, Vol. 15 No 9 September 1972 pp 840-841

A case for Redundant Arrays of Inexpensive Disks D.A., Gibson, G., Katz, R.H.Patterson -Proceedings of the ACM SIGMOD International Conference of the Management of Data 1988

Plan 9 from Bell Labs R., Presotto, K., Thompson, K., Trickey, H.Pike -Proc. UK UNIX Users Group Summer 1990 Conference, London 1990

UNIX, POSIX, and Open Systems, the Open Standard Puzzle J.S., Wilhelm, S.Quarterman -Addison-Wesley Publishing Company 1993 416 oldal

An Optimal Algorithm for Mutual Exclusion in Computer Networks G., Agrawala, A.K.Ricart -Communications of the ACM, vol. 24, no. 1 1981 pp. 9-17

The Evolution of the UNIX Time-Sharing System D.M., Thompson, K.Ritchie -http://cm.bell-labs.com/cm/cs/who/dmr/hist.html

The UNIX Time-Sharing System D.M., Thompson, K.Ritchie -http://cm.bell-labs.com/cm/cs/who/dmr/cacm.html

A UNIX belső szerkezete. Segédlet az operációs rendszerek c. tárgyhoz – 1 Gy.Román -BME-MIT jegyzetek, Budapest 1997

Elosztott rendszerek. Segédlet az operációs rendszerek c. tárgyhoz - 2 Gy.Román -BME-MIT jegyzetek, Budapest 1997

An Introduction to Disk Drive Modeling C., Wilkes, J.Ruemmler -IEEE Computer, Vol. 27 No 3 March 1994 pp 17-29

A Quarter Century of UNIX P.H.Salus -Addison-Wesley, Reading, MA 1994

VM/370 - A Study of Multiplicity and Usefulness L.H., MacKinnon, R.A.Seawright -IBM Systems Journal, Vol. 18 No 1 1979 pp 4-17

Operating System Concepts, (Fifth Edition) A., Galvin, P.Silberschatz -Reading, MA, Addison-Wesley 1998

Operating System Concepts A., Peterson, J., Galvin, P.Silberschatz -Reading, MA, Addison-Wesley 1991

Advanced Computer Architectures (a Design Space Approach) D., Fountain, T., Kacsuk, P.Sima -Addison Wesley Longman 1997

Operating Systems W.Stallings -Macmillan, New York 1992

The Spring Kernel: A New Paradigm for Real-Time Operating Systems J.S., Ramamrithan, K.Stankovic -Operating Systems Review July 1989

Software Communication Mechanisms: Procedure Calls Versus Messages J.S.Stankovic -Computer, Vol. 15 No 4 April 1982

Message Passing Communication versus Procedure Call Communication J.Staunstrup -Software - Practice and Experience, Vol. 12 No 3 March 1982 pp 223-234

tmpfs: A Virtual Memory File System P.Synder -Proceedings of the Autumn 1990 European UNIX Users' Group Conference, October 1990, pp. 241-248

Bevezetés a UNIX operációs rendszerbe, (oktatási segédlet, 4.bővített kiadás) I.Szeberényi -BME-IITT 1998

Parallel Virtual Machine programozása, (oktatási segédlet) Sz., Szeberényi, I.Szigeti -BME-IIT 1998

Distributed Operating Systems A.S., Van Renesse, R.Tanenbaum -ACM Computing Surveys, Vol. 17 No. 4 December, 1985 pp 419-470

Operációs rendszerek A.S., Woodhull, A.S.Tanenbaum -Panem - Prentice-Hall, Budapest 1999

Operating System Design and Implementation (Second Edition) A.S., Woodhull, A.S.Tanenbaum -Prentice-Hall, Englewood Cliffs, NJ [100] 1997

Modern Operating Systems A.S.Tanenbaum -Prentice-Hall Int., Inc 1992

UNIX Internals: The New Frontier, 1st edition U.Vahalia -Prentice Hall 1995

Scheduling Algorithms for Modern Disk Drives B.L., Ganger, G.R., Patt, Y.N.Worthington -Proceedings of the 1994 Sigmetrics Conference on Measurement and Modeling of Computer Systems May 1994 pp 241-251

Special Issue on Real-Time Operating Systems W., Ed.Zhao -Operating System Review July 1989

The POSIX.1. Standard: A Programmer's Guide F.Zlotnick -The Benjamin/Cummings Publishing Company, Inc 1991