

Operációs rendszerek

1. BEVEZETÉS

Felhasznált irodalom:

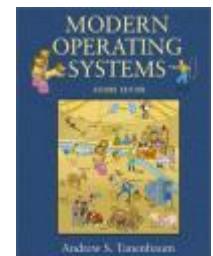
- Kóczy-Kondorosi (szerk.): Operációs rendszerek mérnöki megközelítésben
- Tanenbaum: Modern Operating Systems
- Silberschatz, Galvin, Gagne: Operating System Concepts

Irodalom

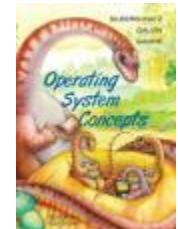
Kóczy-Kondorosi (szerk.): Operációs rendszerek mérnöki megközelítésben
Panem Könyvkiadó, 2000



Tanenbaum: Modern Operating Systems
Fordítása: Operációs rendszerek
Műszaki Könyvkiadó, 1999



Silberschatz, Galvin, Gagne: Operating System Concepts
Wiley, 2005



Tartalom

Az operációs rendszerek fogalma, célja

Leggyakoribb operációs rendszerek

Az operációs rendszerek feladatai

Az operációs rendszerek története

Az operációs rendszer fogalma

DEF 1.

A számítógépen állandóan futó program

- Az a program, amely közvetlenül vezérli a gép működését (operációs rendszer magja, *kernel*).
- *Minden egyéb alkalmazói program.*

DEF 2.

Az összes program, ami a szállítótól "operációs rendszer"-ként érkezik.

- A gyakorlati feladatokat ellátó programok.
- Minden ami a gép "általános" felhasználásához szükséges.
- Pl. grafikus felület, editor, számológép,...

Az operációs rendszerek célja

felhasználók kényelmét szolgálja

- egyszerű,
- kényelmes,
- biztonságos használat

hatékony gépkihasználás

- jó felhasználó élmény („gyors”)
- adott idő alatt minél több program végrehajtása
 - főleg nagy gépeken, szervereken

Statisztika 1

2020:

Asztali operációs rendszerek



~ 77%



~ 17%



~ 2%

Mobil operációs rendszerek



~ 74%



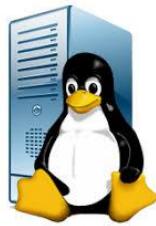
~ 25%



< 0.1%

Statisztika 2

Szerverek

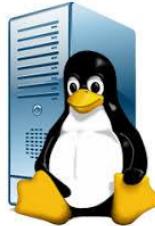


~ 72%



~ 13%

Mainframe-ek, szuperszámítógépek



~ 100%

Windows

A Microsoft Corporation által fejlesztett operációs rendszer

1985:

- Windows 1.0
- Az MS-DOS grafikus kiterjesztése, nem önálló OR
- Nagyon egyszerű ablakozás: csempés szerkezet

Korai Windows-ok (1.x, 2.x):

- Több párhuzamos alkalmazás futtatása
 - Kooperatív multitaskolás
- Virtuális tárkezelés
 - Az aktuális memóriánál nagyobb méretű alkalmazások is futottak
 - A szükséges kódrészletek beemelése, nem szükséges részek eltávolítása a memóriából

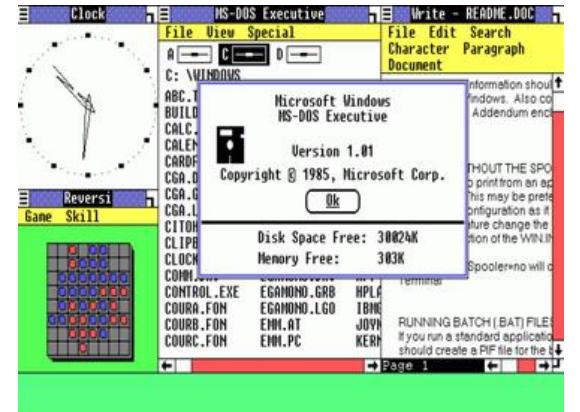
```
Starting MS-DOS...

HIMEM is testing extended memory...done.

C:\>C:\DOS\SMARTDRV.EXE /X
C:\>dir
Volume in drive C is MS-DOS 6
Volume Serial Number is 4B77-00EB
Directory of C:\

DOS           <DIR>          11-23-17  12:07a
COMMAND.COM    COM            54,645 05-31-94  6:22a
WIN320        386             9,349 05-31-94  6:22a
CONFIG.SYS     SYS            71 11-23-17  12:07a
AUTOEXEC.BAT   BAT            78 11-23-17  12:07a
              5 file(s)       64,143 bytes
              517,021,696 bytes free

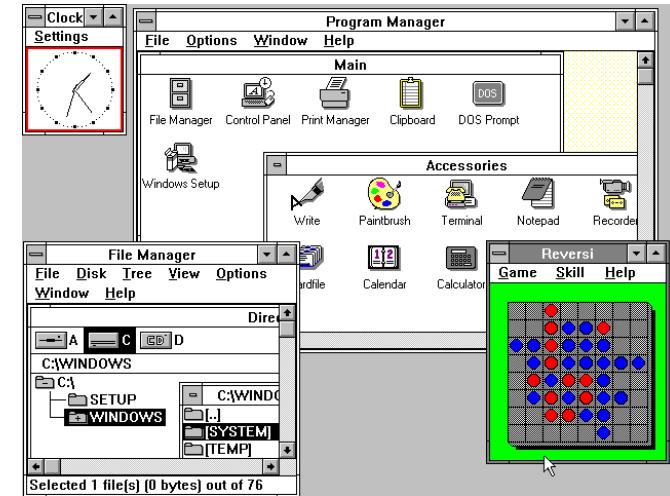
C:\>_
```



Windows

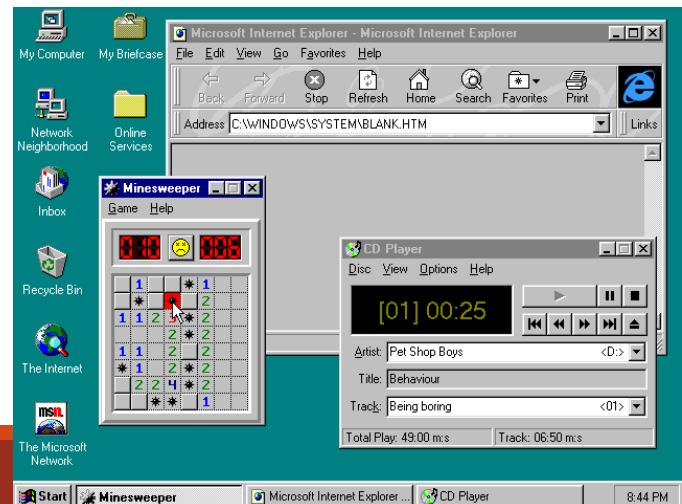
1990:

- Windows 3.0
- Fejlettebb ablakkezelés (átfedő, mozgatható)
- Virtuális eszközkezelők (device driver)
- Gyorsabb kód
- Még mindig kooperatív multitaskolás



1995:

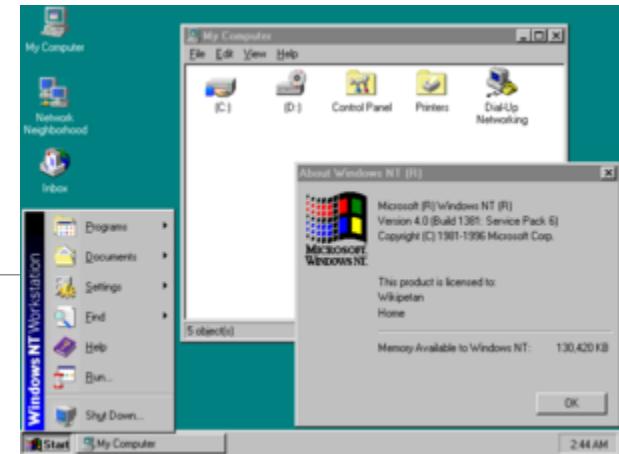
- Windows 95
- Még mindig MS-DOS alapú
- Hosszú fájlnevek (max. 255 karakter)
- 32 bites alkalmazásokat is támogat
- Preemptív ütemező!



Windows

1993:

- Windows NT
- Önálló OR lett
- A további MS OR-ek (XP, Vista, W7, W8, W10) alapja ez lesz



2001:

- Windows XP

2007:

- Windows Vista

2009:

- Windows 7

2012:

- Windows 8.x

2015:

- Windows 10



MacOS

Az Apple saját fejlesztésű OR-e

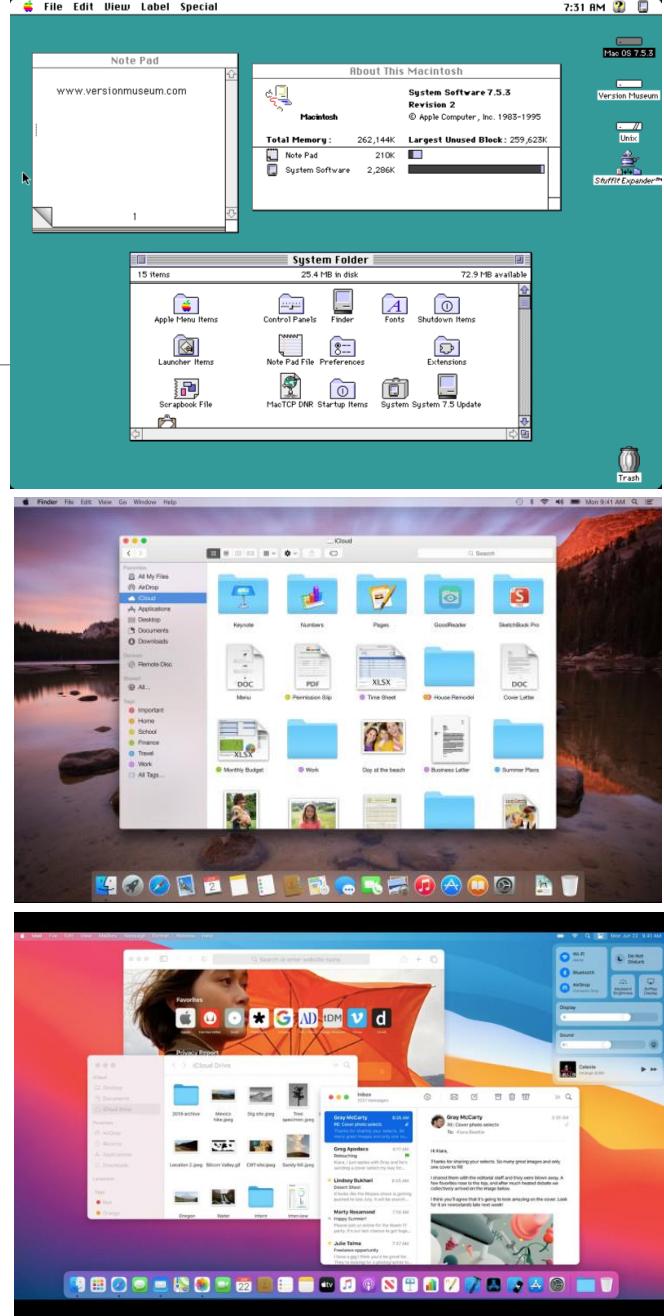
Magja Unix alapokon nyugszik

Elődei:

- 1984-1999:
 - „classic” Mac OS
 - Utolsó verzió: Mac OS 9
- 2000-2015
 - OS X
- 2016-
 - macOS



OSX

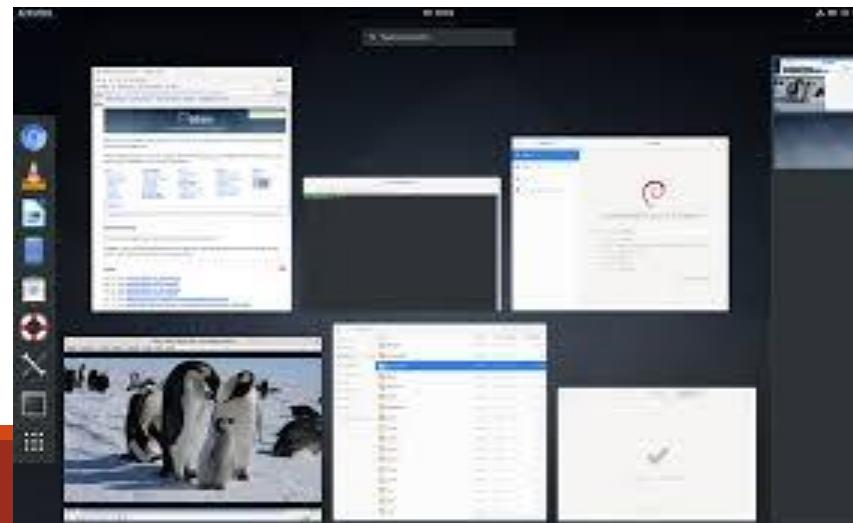
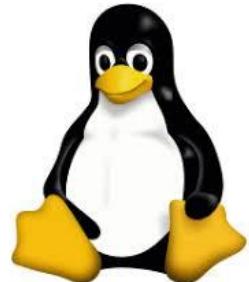


Linux

Unix-hoz hasonló

1991:

- Linus Thorvalds kezdeményezésére és irányításával született
- Nyílt forráskódú
- Rengeteg változata és továbbfejlesztése létezik
 - Asztali gépek
 - Szerverek
 - Mobil eszközök (Android)
 - Beágyazott eszközök (órák, e-könyv olvasók, stb.)





Android

Mobil eszközökre készült

2007: Google projekt

- nyílt forráskódú (csak a mag)
- Linux kernel
- Java-kompatibilis felhasználói környezet

Lehetővé teszi a módosítást, továbbfejlesztést

A legtöbb gyártó saját zárt komponenseivel egészíti ki:

- Telefon
- Tablet





iOS

Az Apple saját fejlesztésű OR-e

Az Apple mobil eszközeire készült

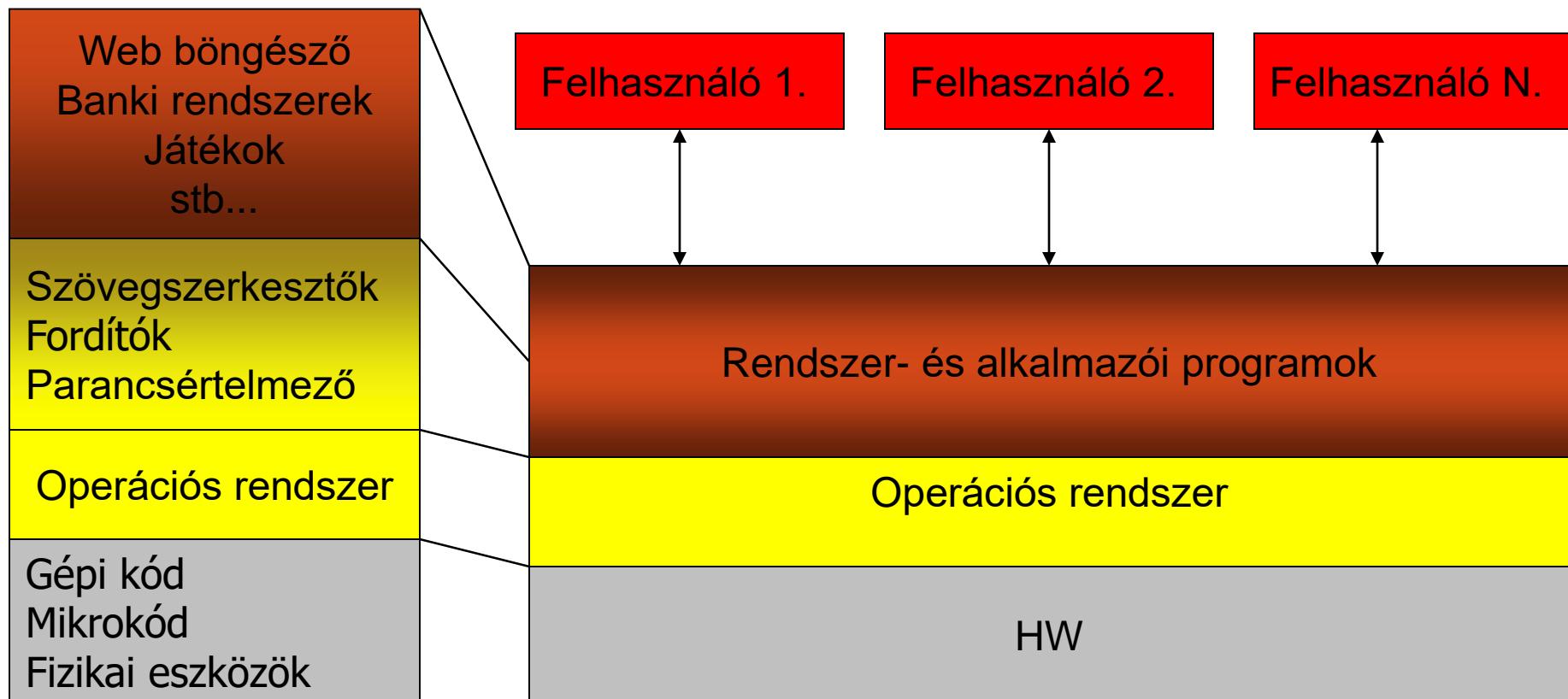
2007: iPhone és iOS

Azóta több platformon megjelent:

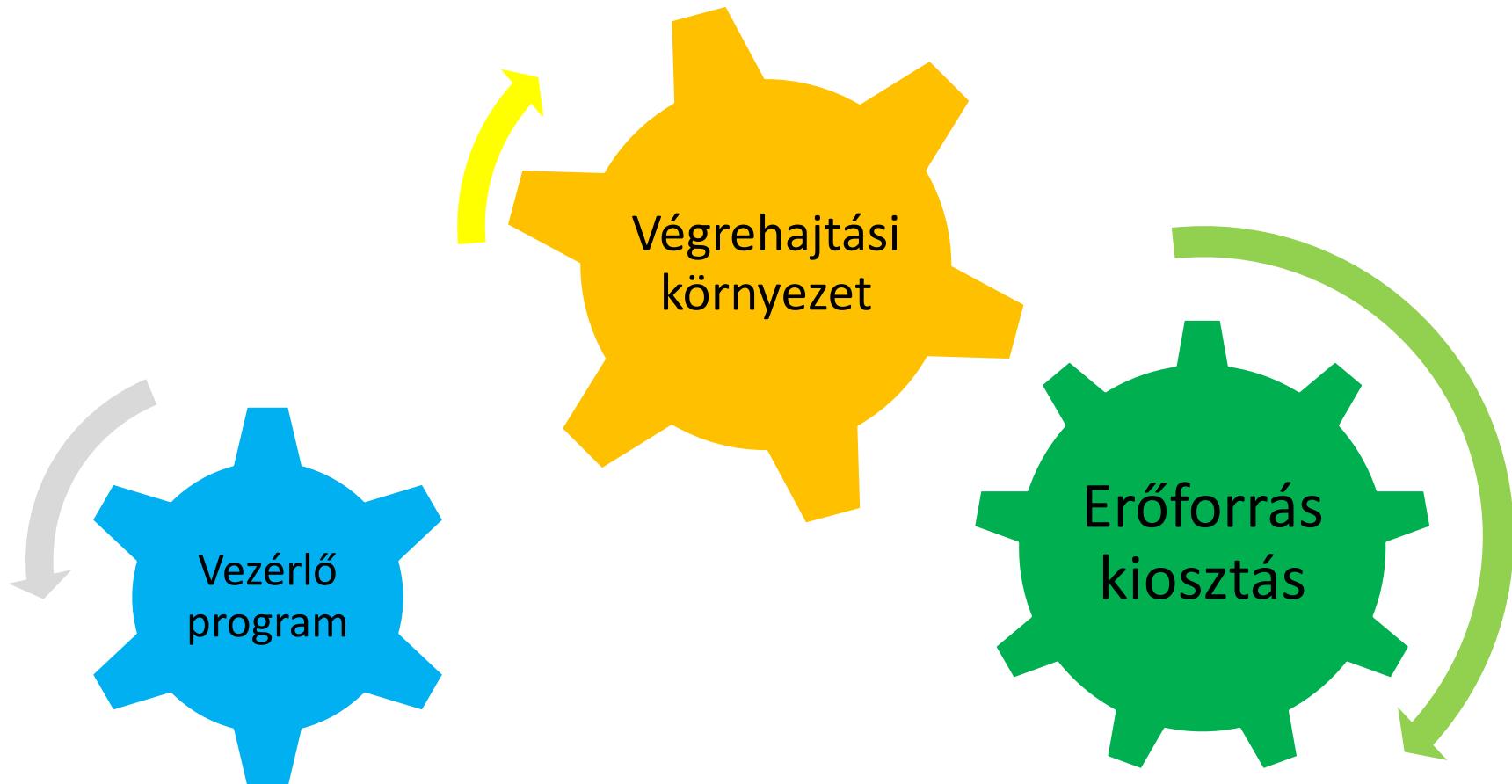
- iPhone
- iPod
- iPad



Számítógépes rendszerek szerkezete

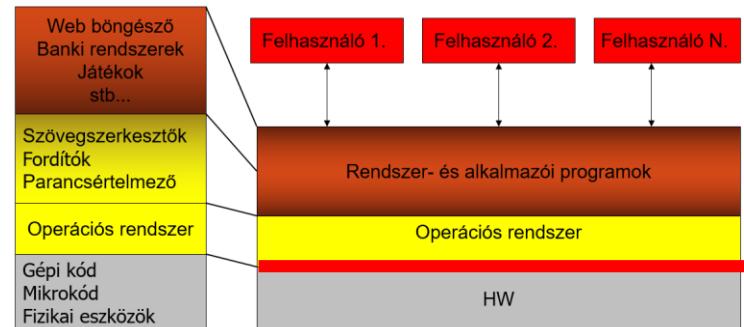


Az operációs rendszerek feladatai



Operációs rendszer feladatai 1

Végrehajtási környezet

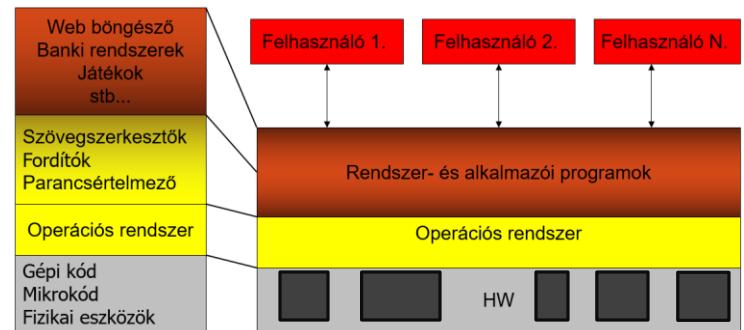


- Olyan környezet, ahol a felhasználók és programjaik hasznos munkát végezhetnek.
- A számítógép **hardver szolgáltatásainak bővítése**
- Elrejti a „piszkos” részleteket, könnyű felhasználhatóságot biztosít

Operációs rendszer feladatai 2

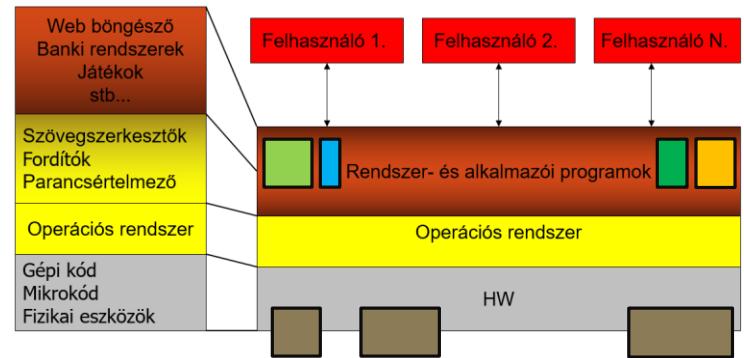
Erőforrás kiosztás

- Kezeli a rendszer erőforrásait
 - CPU,
 - központi tár,
 - merevlemez, stb.
- Tulajdonságai:
 - hatékony
 - biztonságos
 - igazságos felhasználás.



Operációs rendszer feladatai 3

Vezérlő program



- vezérli a **felhasználói programok** működését,
- a felhasználói programok helyett **vezérli a perifériák** működését.

2. Az operációs rendszerek története

Korai rendszerek

Rendszerek a perifériás műveletek gyorsítására

Multiprogramozott rendszerek

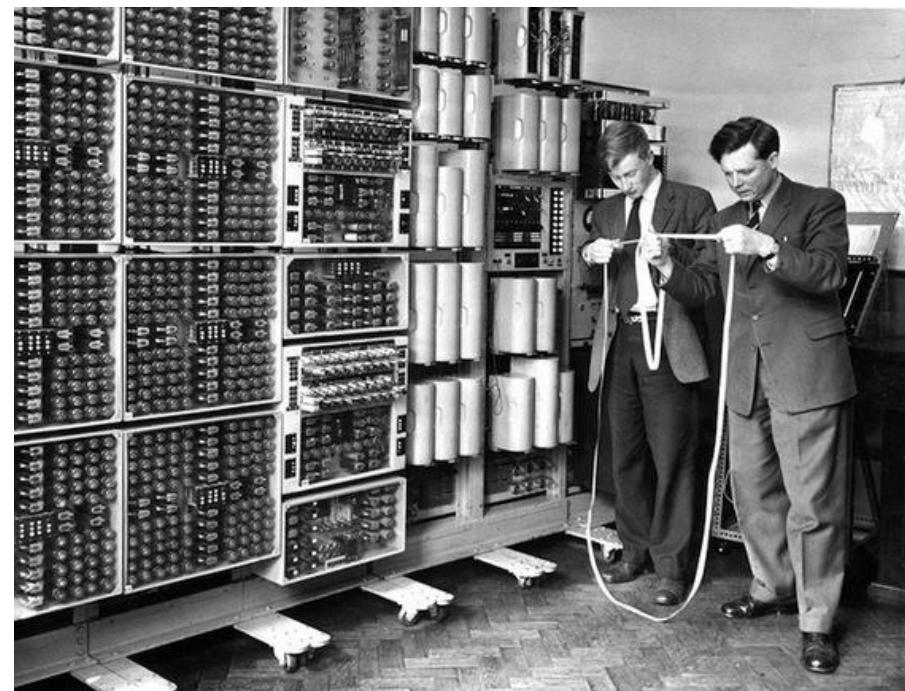
Napjaink rendszerei

Korai rendszerek 1



Nincs operációs rendszer (1945-1955)

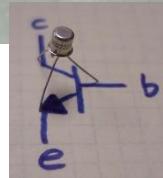
- Hardver: nagy drága
 - konzolról vezérelt CPU
 - egyszerű perifériák (lyukkártya, nyomtató)
- Használat: programozó = operátor
 - kézi vezérlés
 - gépidő foglalás
- Mindent mindenki maga csinál
 - sok hiba
 - sok hibakeresés
 - rossz kihasználtság.



Korai rendszerek 2

Kötegelt feldolgozás (1955-1965)

- Képzett operátor már van. Gyorsabb gépkezelés, de hibakeresés még nincs (core dump kiíratása).
- A programozó utasításokat mellékel a programhoz (munka, *job* leírása).
- Az operátor csoportosítja a munkákat, illetve egyes fázisait = kötegelt feldolgozás (*batch*-ek).



Egyzerű „monitor”

- A gép vezérlését egy *állandóan* a memóriában lévő program (a *monitor*) végzi, az operátor a perifériákat kezeli (ezek lassúak).
- a munkához rendelt vezérlő információkat a monitor értelmezi, végrehajtja.
- egy tevékenység befejezése után újra a monitor kapja meg a vezérlést, amely beolvassa a következő munkát.

Job Control language

\$JOB - munka kezdete

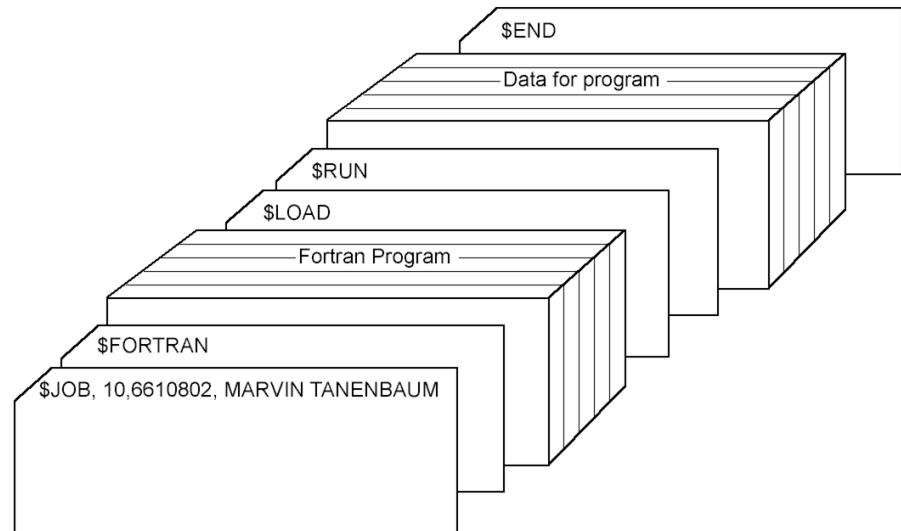
\$END - munka vége

\$FORTRAN - FORTRAN fordító

\$ASM - az assembler hívása

\$LOAD - program betöltése

\$RUN - program futtatása



A perifériás műveletek gyorsítása

- A munkák között automatikus az átkapcsolás (GYORS)
- DE: a perifériák sebessége a CPU-hoz képest LASSÚ.
- A teljes rendszer sebességét a lassú periféria határozza meg.

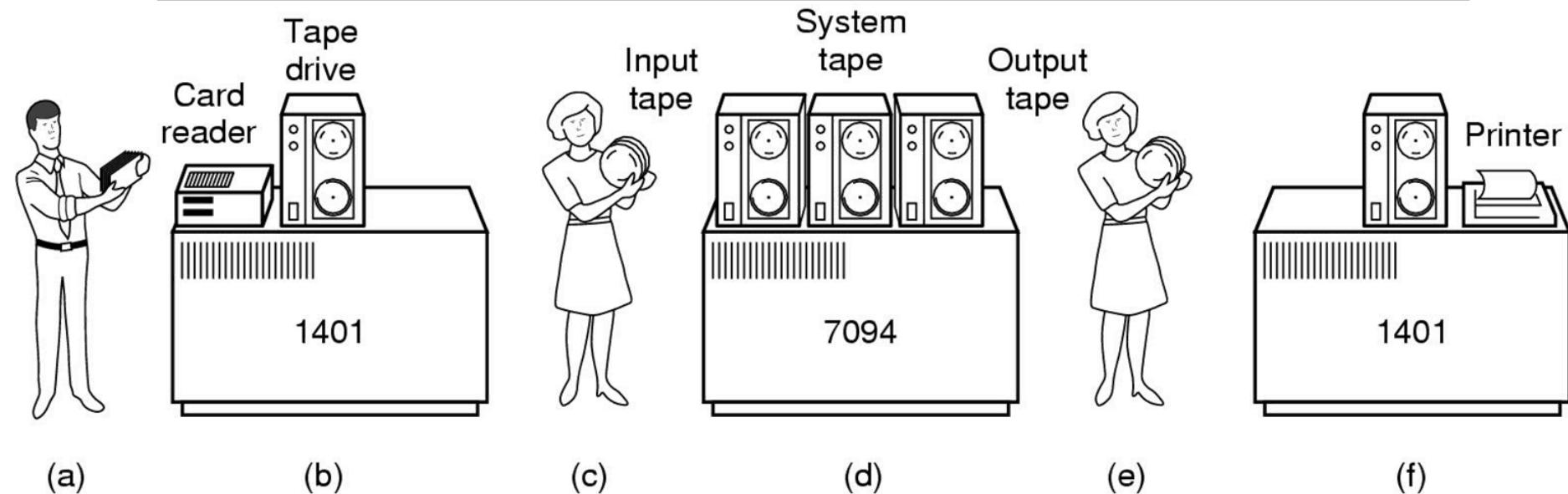
Megoldások:

- Off-line I/O műveletek
- Pufferelés

Off-line I/O műveletek

- Bemenet szalagról (lényegesen gyorsabb mint a kártya).
 - Egy másik gép végzi a másolást kártyáról a szalagra.
- Kimenet: szalagról
 - Egy másik gép végzi a nyomtatást a szalagról.
- +2 gépet igényel
 - A kisegítő gépek egyszerűbb/olcsóbb gépek
- Az I/O műveletek felgyorsulnak, a processzor kihasználtsága nő.
- A három számítógép párhuzamosan működik.
- Periféria-független programok, szabványos felületek.

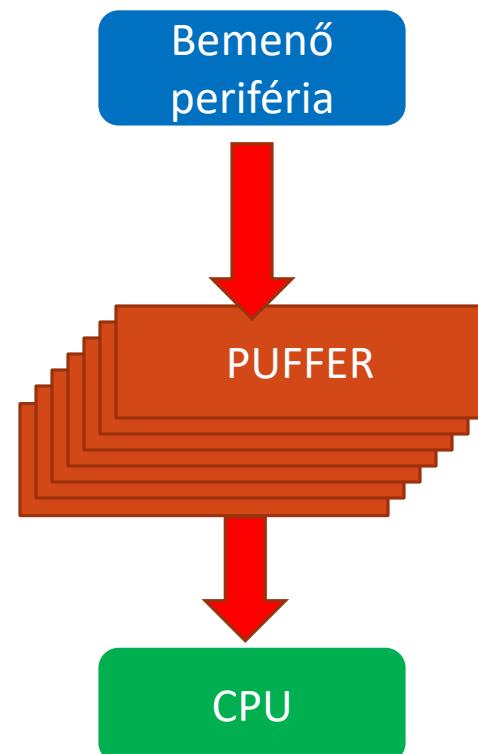
Off-line I/O műveletek



- a) Kártya a 1401-be
- b) Kártya másolása szalagra
- c) Szalag a 7094-hez
- d) 7094 elvégzi a számítást
- e) Szalag a 1401-be
- f) 1401 kinyomtatja az eredményt

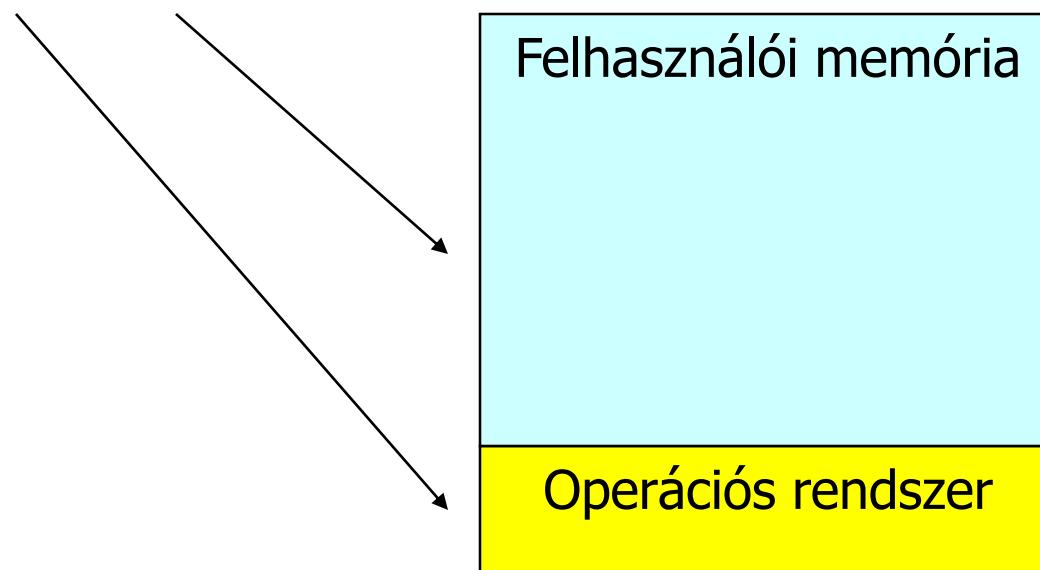
Pufferelés

- CPU és a periféria vezérlő egyidejűleg működnek.
 - A perifériáról történő beolvasás a pufferbe megy.
 - A CPU az adatokat a pufferből olvassa ki
- Az adatfeldolgozás és az I/O művelet átlapoltan történik.
- A kimenet hasonlóan működik
 - A sebességingadozásokat jól kompenzálja
 - Csak akkor jelent hosszú távon sebességnövekedést, ha
 - az I/O műveletek összes ideje és
 - a CPU műveletek összes ideje
 - kb. azonos (ami általában nem igaz)

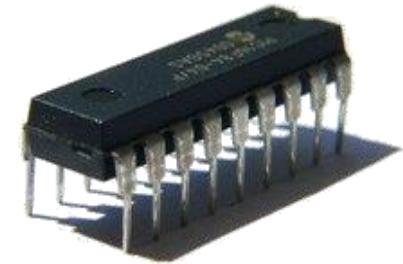


Kötegelt rendszerek memória-kiosztása

Memória partíciók



Multiprogramozás

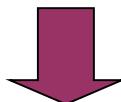


1960-1980

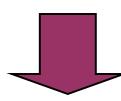
A CPU sebessége megnőtt (IC-k)



érdemessé kihasználni az időt, amíg egy munka a lemezről olvasás eredményére vár



erre az időre másik program működhet.



Az OS egyszerre több munkát futtat.

A multiprogramozás lépései

A rendszer nyilvántartja és tárolja a futtatandó munkákat.

A kiválasztott munka addig fut, amíg várakozni nem kényszerül.

Az OS feljegyzi a várakozás okát, majd kiválaszt egy másik futni képes munkát és azt elindítja.

Ha a félbehagyott munka várakozási feltételei teljesülnek, akkor azt alkalmassint elindítja.

Napjainkban a legtöbb rendszer a *multiprogramozás* elvét alkalmazza.

Multiprogramozás

Felvetődő problémák:

Az átkapcsoláshoz több program van a tárban:
tárgazdálkodás.

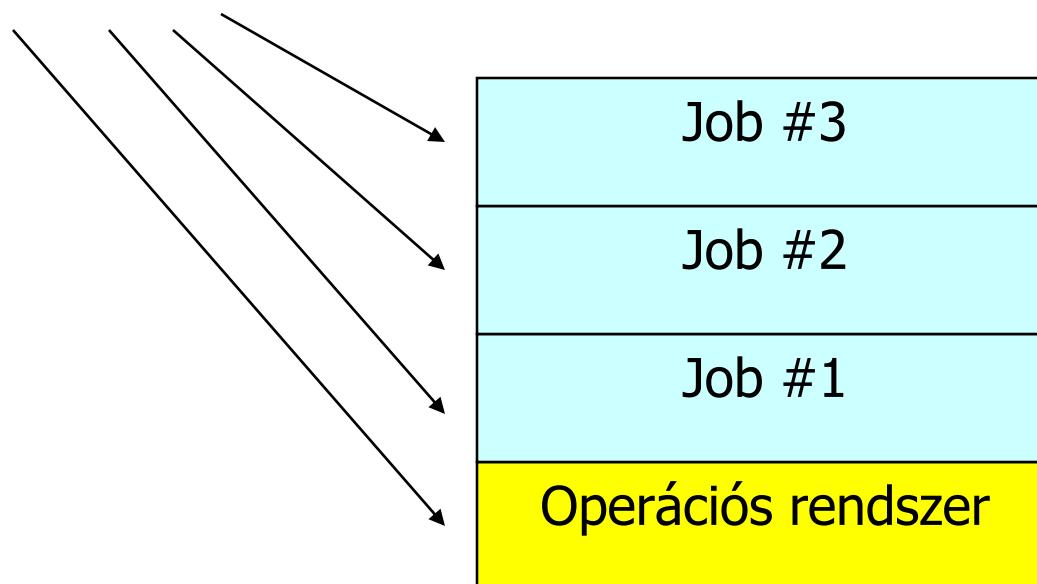
Egy időben több futásra kész program: *CPU ütemezés*

A gépi erőforrások felhasználásának koordinációja.
Allocáció, holtpont kezelése.

Védelmi mechanizmusok, hogy a programok ne zavarják egymást és az OS-t.

Multiprogramozott rendszer memória-kiosztása

Memória partíciók



Napjaink rendszerei



1980-: főleg személyi számítógépek

Időosztásos rendszerek

Valós idejű rendszerek

Beágyazott operációs rendszerek

Időosztásos rendszerek

A időosztásos rendszerek (timesharing, multitask systems) közvetlen interaktív kommunikációt biztosít a felhasználó és a programja, ill. az OS között.

Adatok tárolására közvetlen (on-line) állományrendszerben.

A felhasználó interakciója nagyon lassú, közben az OS más tevékenységet tud végrehajtani.

Gyors reakció a parancsokra, válaszidő (response time) kicsi. Sűrűn kell a programok között kapcsolatot tartani.

Több felhasználó is lehet. Felhasználók függetlenül használják a gépet, mintha mindenki egy saját gépen dolgozna.



Valósidejű rendszerek



Kemény valós idejű rendszerek (hard real-time)

- **Szigorúan definiált és betartott válaszidők**
- Kritikus rendszerek (pl. atomreaktor, járművek)
- Adattárolás: RAM, ROM, másodlagos tárolást (diszk) nem támogatják
- Általános célú operációs rendszerek nem támogatják

Puha valós idejű rendszerek (soft real-time)

- **A válaszidők betartására törekednek, csúszás megengedett**
- Nem kritikus folyamatirányítási feladatok
- Multimédia rendszerek, virtuális valóság

Beágyazott operációs rendszerek

Tipikus alkalmazási területek:

- Házterületi berendezések
- Járművek
- Játékok
- Robotok

Nem általános célú!

- nem független az alkalmazástól (összefordul)

Korlátos erőforrások

- Beágyazott környezetben kevés memória, gyenge CPU
- A szolgáltatások is korlátosak.
- Az OR-nek csak a szükséges része fordul be.



Operációs rendszerek

2. FOLYAMATOK

Felhasznált irodalom:

- Kóczy-Kondorosi (szerk.): Operációs rendszerek mérnöki megközelítésben
- Tanenbaum: Modern Operating Systems
- Silberschatz, Galvin, Gagne: Operating System Concepts

Tartalom

- ❑ Bevezetés
- ❑ Folyamatkezelés multiprogramozott rendszerben
- ❑ Környezet váltás
- ❑ Folyamatleírók, I/O leírók
- ❑ Szálak
- ❑ Megszakítások

A folyamat (process)

A folyamat (process) a multiprogramozott OS rendszerek alapfogalma:
végrehajtás alatt álló program.

Multitaszkos rendszerben több folyamat van a rendszerben egyidejűleg.

Elnevezések:

- munka (job) - batch rendszereknél
- feladat (taszk) - real time
- felhasználó - időosztásos rendszereknél

Általános a *folyamat* név.

Folyamatok

```
www-data 6725 0.0 7.6 69440 19392 ? S< Sep06 0:53 \_ /usr/sbin/apache2 -k start
www-data 6921 0.0 7.6 69696 19392 ? S< Sep06 0:50 \_ /usr/sbin/apache2 -k start
www-data 6995 0.0 7.5 69696 19136 ? S< Sep06 0:53 \_ /usr/sbin/apache2 -k start
www-data 7228 0.0 6.4 69824 16256 ? S< Sep06 0:51 \_ /usr/sbin/apache2 -k start
www-data 7237 0.0 7.4 68992 18944 ? S< Sep06 0:53 \_ /usr/sbin/apache2 -k start
root 2846 0.0 0.4 3712 1216 ? Ss Sep06 0:04 /usr/sbin/ifplugged -i eth0 -q -f -u0 -d5 -w -I
root 2872 0.0 0.3 3584 768 ? S Sep06 0:00 /usr/local/bin/resetbtnd /dev/input/event0 4 /usr/local
root 3411 0.0 0.5 4544 1472 ? Ss Sep06 0:00 /sbin/mdadm --monitor --pid-file /var/run/mdadm/monitor
root 3456 0.0 1.5 9920 4032 ? Ss Sep06 0:00 /usr/sbin/sshd
root 5269 0.0 3.3 13696 8576 ? Ss 12:05 0:00 \_ sshd: root@pts/0
root 5272 0.0 1.7 5760 4352 pts/0 Ss 12:05 0:00 \_ -bash
root 6867 14.6 1.6 6528 4160 pts/0 D 12:43 0:00 \_ find / -name alma
root 6868 0.0 1.1 4864 3008 pts/0 R+ 12:43 0:00 \_ ps -auxf
root 3472 0.0 1.0 5568 2752 ? S Sep06 0:00 /bin/sh /usr/local/sbin/monitorTemperature.sh
root 5973 0.0 0.7 3904 1856 ? S 12:22 0:00 \_ -bash 1000
root 3488 0.0 1.0 5632 2752 ? SN Se
root 6861 0.0 0.7 3904 1856 ? SN 12:22 0:00 \_ -bash 1000
root 3544 0.0 0.4 4032 1088 ? Ss Se
root 3564 0.0 0.3 8000 896 ? Ss Se
root 3596 0.0 0.1 4480 448 ? Ss Se
root 3607 0.0 2.4 14720 6208 ? Ss Se
root 3610 0.0 1.8 49664 4608 ? S<s Se
root 3619 0.0 1.0 49664 2560 ? S< Se
nobody 30569 0.0 4.8 52288 12352 ? S< 09
root 3632 0.0 0.6 5248 1728 ? S< Se
root 3634 0.0 0.4 8832 1216 ? S< Se
root 3635 0.0 0.4 5248 1152 ? S< Se
root 3678 0.0 0.6 3776 1536 ? Ss Se
root 4133 0.0 1.1 70848 2880 ? Sl Se
103 4188 0.0 0.4 5312 1152 ? Ss Se
```

Windows

Task Manager								
File Options View		Processes	Performance	App history	Startup	Users	Details Services	
Name	Status	19%	77%	0%	0%	Network	Power usage	Power usage trend
Apps (12)								
> Windows Terminal (7)		0%	10.8 MB	0 MB/s	0 Mbps	Very low	Very low	
> Windows PowerShell (2)		0%	24.5 MB	0 MB/s	0 Mbps	Very low	Very low	
> Windows Explorer (2)		0%	35.2 MB	0.1 MB/s	0 Mbps	Very low	Very low	
> Total Commander 32 bit (32 bit) (2)		0%	2.5 MB	0 MB/s	0 Mbps	Very low	Very low	
> Task Manager		1.9%	20.8 MB	0 MB/s	0 Mbps	Very low	Low	
> Skype (6)		0%	141.2 MB	0 MB/s	0 Mbps	Very low	Low	
> Microsoft Word (32 bit) (2)		0.2%	2.3 MB	0 MB/s	0 Mbps	Very low	Very low	
> Microsoft PowerPoint (32 bit)		0%	35.8 MB	0 MB/s	0 Mbps	Very low	Very low	
> Microsoft Outlook (32 bit)		0%	18.1 MB	0 MB/s	0 Mbps	Very low	Very low	
> Groove Music (2)		0%	14.8 MB	0 MB/s	0 Mbps	Very low	Very low	
> Google Chrome (37)		0.2%	818.3 MB	0 MB/s	0 Mbps	Very low	Low	

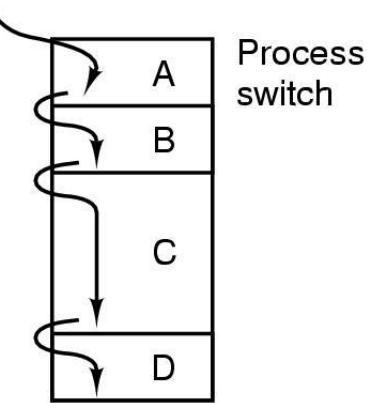
Linux

Folyamatkezelés multiprogramozott rendszerekben

- ❑ Folyamatok modellezése multiprogramozott környezetben
- ❑ Folyamatok állapotgráfja
- ❑ Az állapotok és állapotátmenetek
- ❑ Kibővített állapotgráf

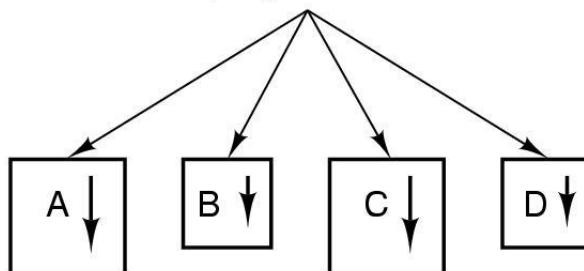
Folyamatok modellezése

One program counter

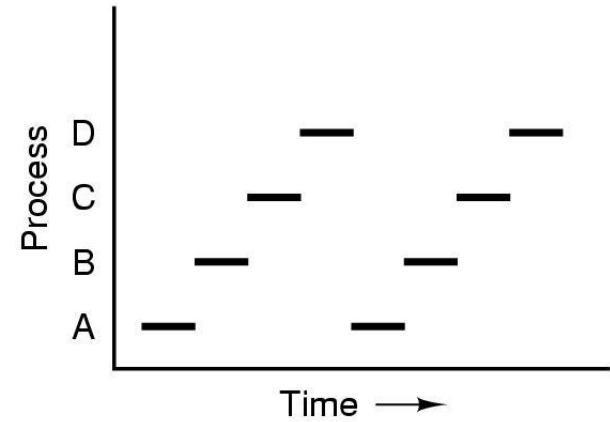


(a)

Four program counters



(b)



(c)

- a) Multiprogramozott rendszer: 1 processzor, 1 PC (program counter)
- b) Valódi párhuzamos rendszer: 4 processzor, 4 PC
- c) Folyamatok időbeli eloszlása multiprogramozott rendszerben

Folyamatok állapotai

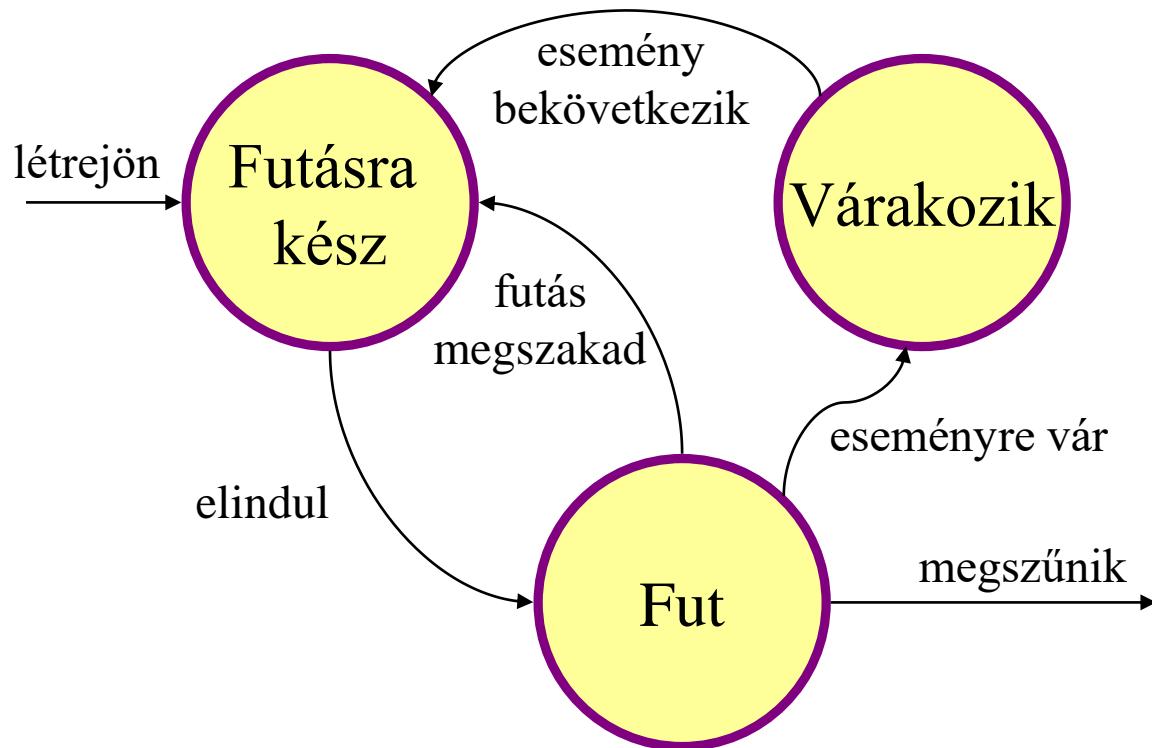
Több folyamat , 1 CPU → látszólagos párhuzamosság.

CPU kitüntetett erőforrás, egy adott pillanatban csak 1 program hajtódik végre.

Gráffal lehet ábrázolni. Állapotok:

- **Fut:** A CPU a folyamathoz tartozó utasításokat hajtja végre, CPU-nként egyetlen ilyen folyamat lehet.
- **Várakozik, blokkolt:** A folyamat várakozni kényszerül, működését csak valamilyen esemény bekövetkezésekor tudja folytatni. Több ilyen is lehet a rendszerben.
- **Futásra kész:** minden feltétel adott, a CPU éppen foglalt. Több ilyen is lehet a rendszerben.

Folyamatok állapotai



Folyamatok állapot-átmeneti gráfja

Folyamatok állapotátmenetei

Folyamat létrehozása

Folyamat befejeződése

Eseményre vár

Esemény bekövetkezik

Folyamat elindul

Futás megszakad

Folyamat létrehozása 1.

Mikor jön létre egy folyamat?

Rendszer elindításakor (boot)

- Pl. felhasználói interfések, démonok

Folyamatot létrehozó rendszerhívás hatására (pl. fork)

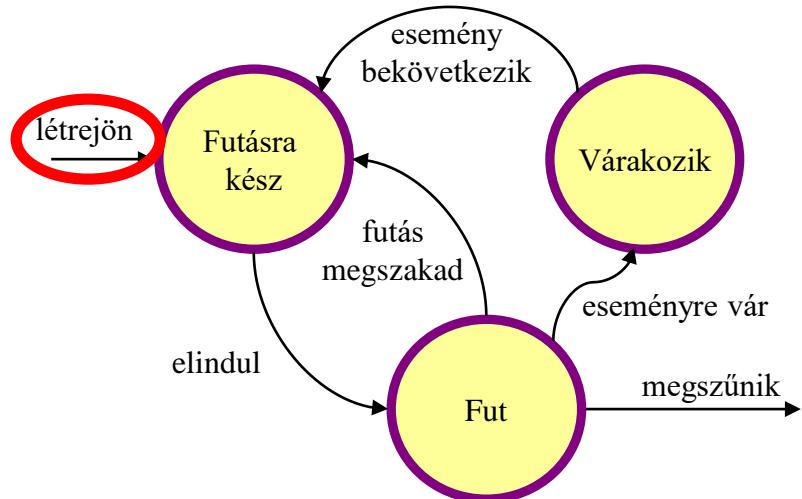
- Egy folyamat gyermek folyamatot hozhat létre
 - Hierarchia (UNIX): „process group”
 - minden folyamat egyenlő (Windows)

Felhasználó

- Program indítása

Új batch-job indítása

- Kötegelt rendszereknél OS dönti el, mikor kezdhet futni egy új munka



Folyamat létrehozása 2.

Létrehozó folyamat: szülő,
a létrejöttek a gyerekek.

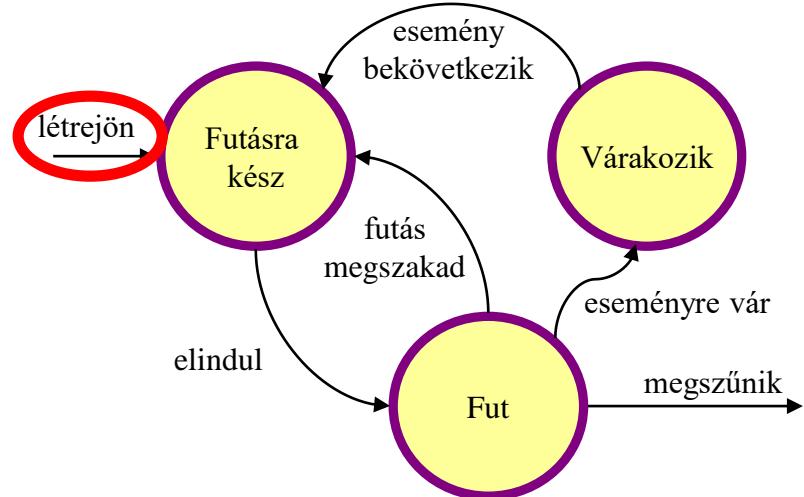
Leszármazási reláció, hierarchikus struktúra.

Erőforrás kell neki

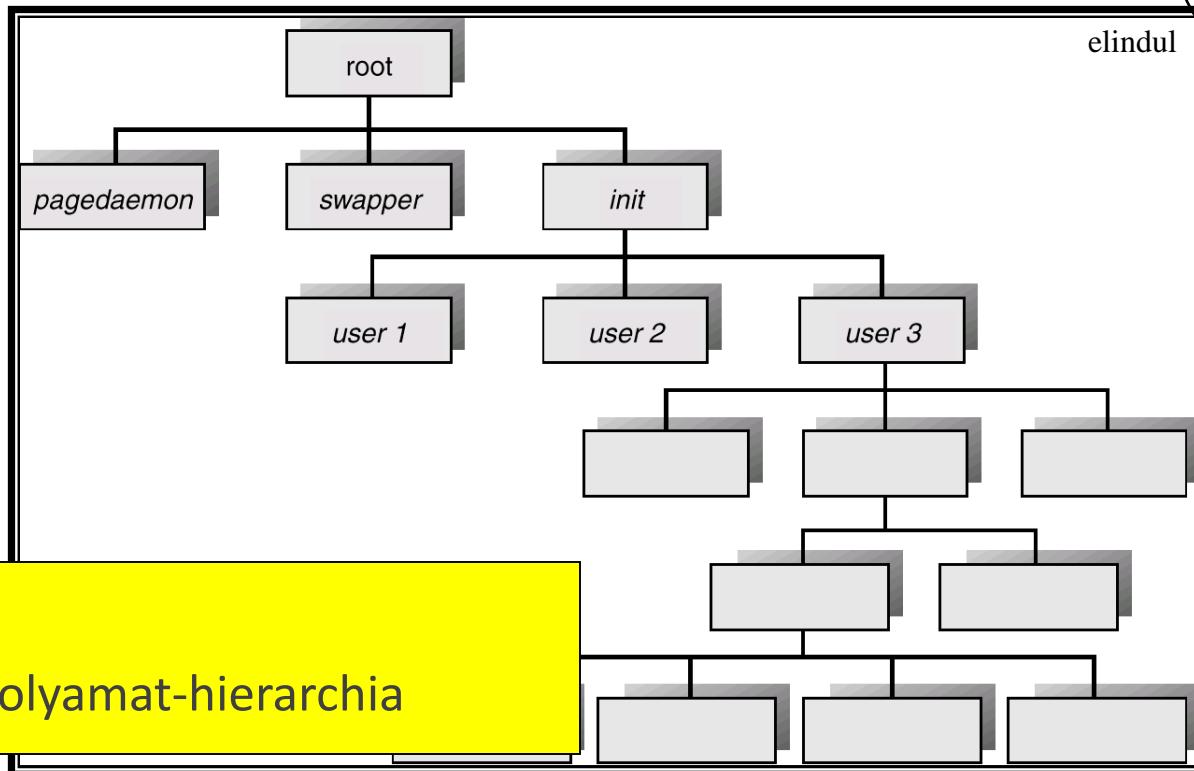
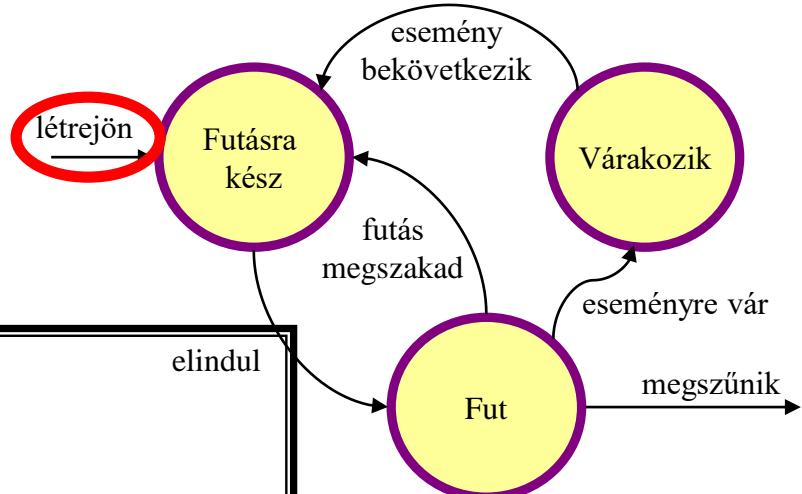
- OS-től kapja
- szülő erőforrásain osztozik

Szülőtől paramétereket kap (befolyásolja a futását)

Gyerek a szülővel párhuzamosan fut, vagy bevárja a gyerekének, gyerekeinek befejeződését.



Folyamat létrehozása 3.



USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.0	0.0	8892	312	?	Ssl	10:05	0:00	/init
root	82	0.0	0.0	8896	216	tty2	Ss	11:29	0:00	/init
simon	83	0.0	0.0	15168	2528	tty2	S	11:29	0:00	_ -bash
simon	129	84.8	0.0	16192	1692	tty2	S	12:52	0:10	_ find / -name alma
simon	131	0.0	0.0	16640	1772	tty2	R	12:52	0:00	_ ps -auxf

Folyamat befejeződése

Önszántából

- végrehajtotta utolsó utasítását
- hiba miatt leáll

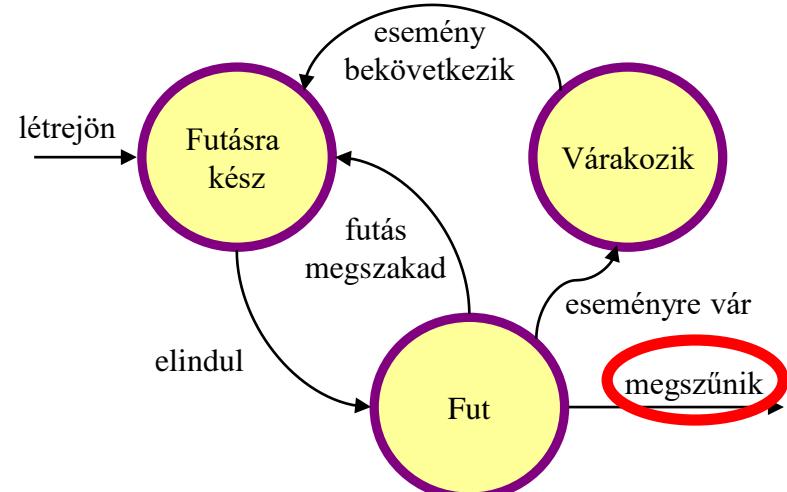
OS vagy rokon (általában a szülő) leállítja

- Hibás utasítás, erőforrás használat túllépése
- kill utasítás (másik folyamat)

Erőforrások felszabadulnak

- attól függően, hogy kitől kapta: felszabadul vagy szülőhöz kerül

Szülőnek információt adhat vissza.



Eseményre vár

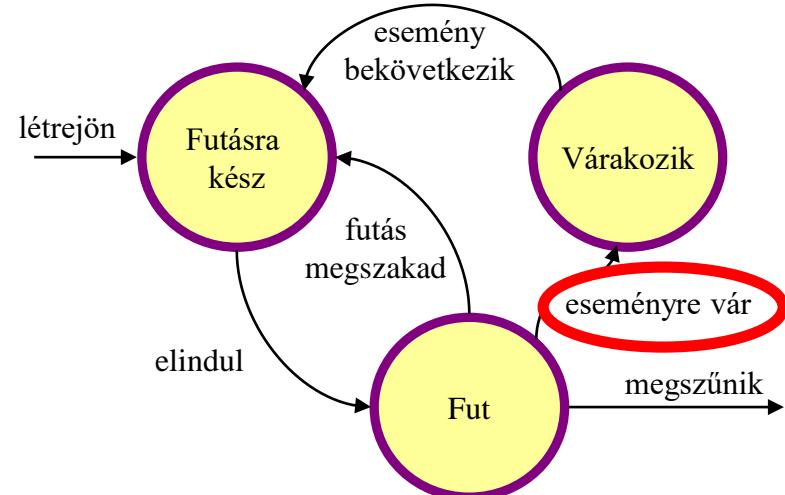
Fut → várakozik

Valamit kér, amire várnia kell

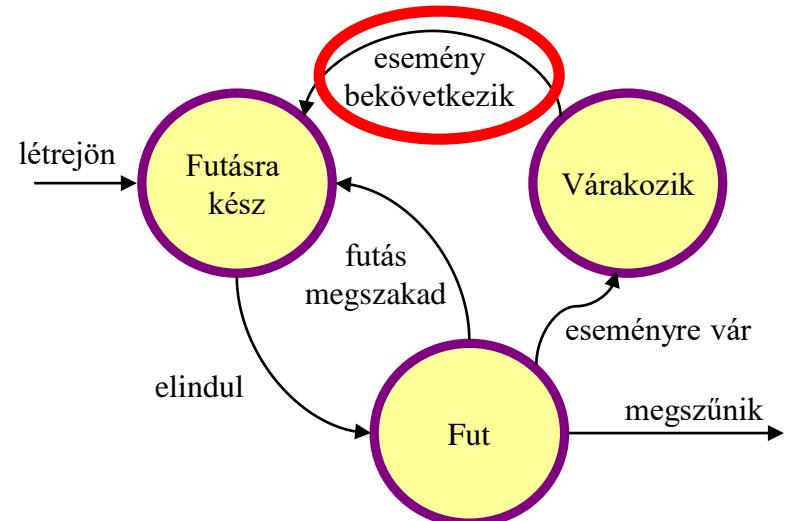
- jelzés,
- erőforrás, stb.

OS feljegyzi, hogy ki mire vár.

Több folyamat is várhat ugyanarra.



Esemény bekövetkezik



Várakozik → futásra kész

A várt esemény bekövetkezett

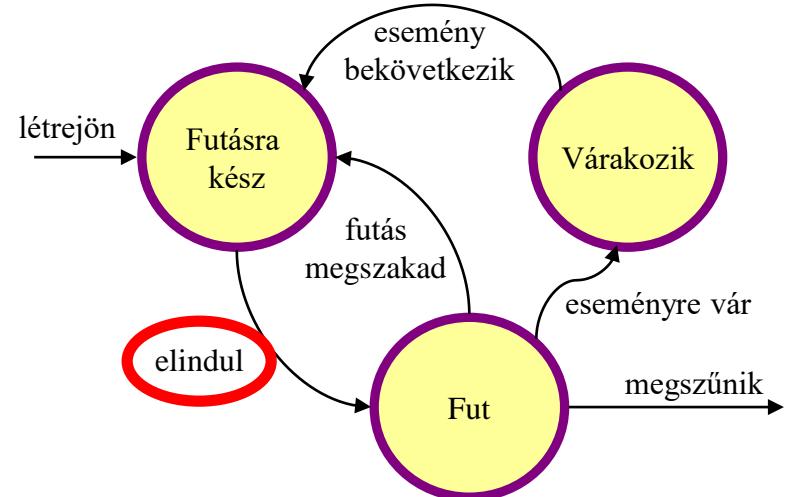
Még nem fut!

Folyamat elindul

Futásra kész → fut

ha a CPU felszabadul, *egy folyamat futhat*

kiválasztás kritériumok alapján (**CPU ütemezés**)



Futás megszakad

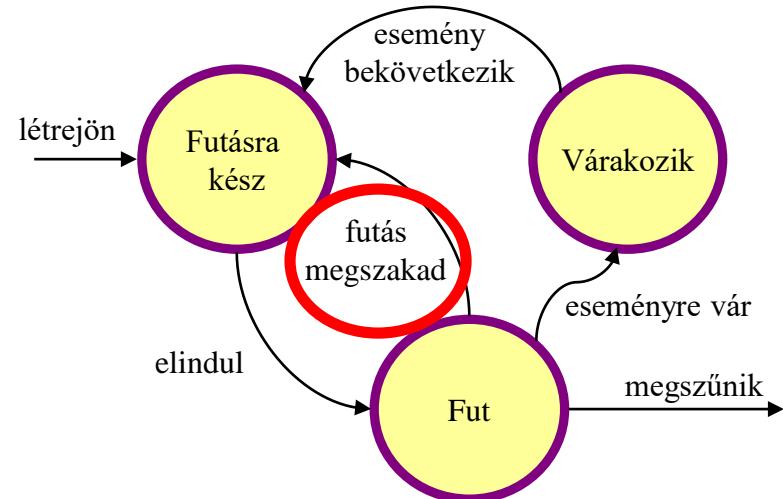
Fut → futásra kész

Önként lemond a CPU-ról.

- Pl. kooperatív viselkedés: hosszú feladatok esetén újraütemezést kér.

OS elveheti a CPU-t, még akkor is, ha a folyamat egyébként nem kényszerül várakozásra (*preemptív ütemezés*)

- Pl. időosztásos rendszerek. Túl sok ideje futott. Óra megszakítás jelzi az eseményt.



Folyamatok kibővített állapotai

Bővítés: az OS felfüggeszthet folyamatokat (középtávú CPU ütemezés)

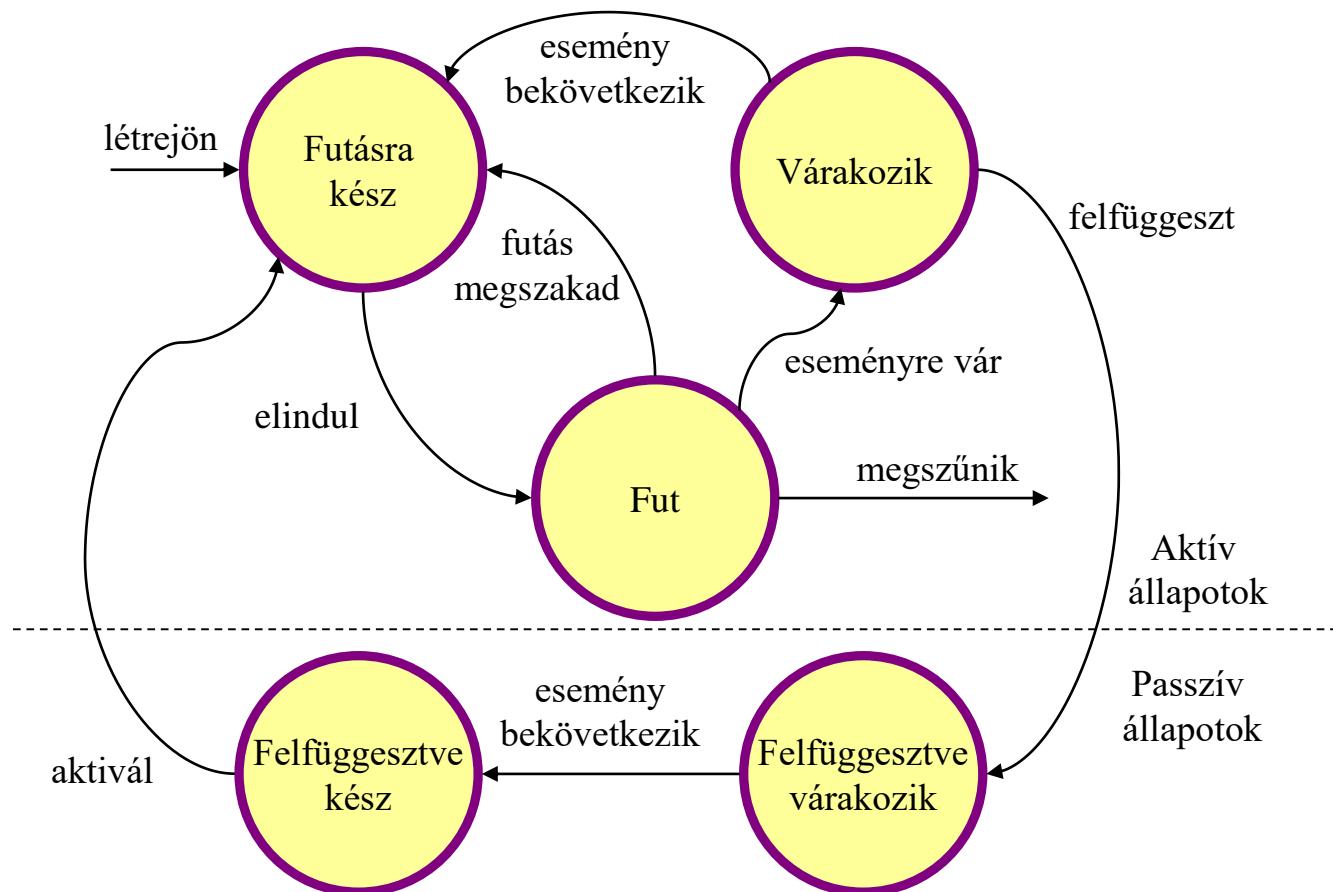
- a rendszer túl van terhelve (sok program vetélkedik a futás jogáért, vagy a tár túlzottan megtelt, stb.)
- vészhelyzet esetén
- felhasználó kezdeményezésére

A felfüggesztett folyamatok erőforrásait elvesztik (de a rendszer számon tartja őket, később folytatódhatnak)

Két új állapot:

- felfüggesztve vár
- felfüggesztve kész

Folyamatok kibővített állapot-átmeneti gráfja



Új állapotátmenetek

Felfüggeszt

- OS felfüggeszti (futásra kész vagy várakozik állapotból)
- erőforrásokat elveszi (pl. memória), néhányat megtarthat (pl. nyomtató)

Aktivál

- Erőforrásokat visszaad

Felfüggesztve várakozik → felfüggesztve futásra kész

- Esemény bekövetkezik, de CPU-t nem kaphat

A felfüggesztve vár → futásra kész átmenetnek nincs értelme
(tovább várakozna, de lekötné az erőforrásokat)

Környezetváltás

Környezet váltás (kontext switch):

- a futó folyamat elhagyja a futó állapotot,
- egy futásra kész pedig elindul.

Az átkapcsolás zökkenőmentes legyen:
állapotjelzőket meg kell őrizni

1. Folyamat állapota.
2. Végrehajtó gép állapota.

Multiprogramozásnál fontos a gyors átkapcsolás
(hatékonyság).

A folyamat állapota

Állapotváltozók:

programkód

változók aktuális értéke

verem tartalma

hol tart a program végrehajtása (programszámláló)

- ezt a végrehajtó gép tartalmazza!

Tárban több folyamat van, így a folyamatok állapotjelzői megmaradnak a tártartalom megőrzésével. Csak a *végrehajtó gép* állapotát kell menteni.

A végrehajtó gép állapota

A rendszer rétegszerkezete miatt több szintű. A legegyszerűbb eset, amikor a folyamat kódja csak gépi utasításokat tartalmaz: hardver-szoftver határfelület, legtöbbször az OS alapszolgáltatásait is tartalmazó virtuális gép határfelülete.

- CPU regiszterek
- OS változók
 - rendszertáblák,
 - memória kezelési információk,
 - periféria hozzárendelések, stb.

A végrehajtó gép (OS) a folyamat környezete, a gép állapotjelzőinek összességét kontextusnak nevezük.

Folyamatleírók, I/O leírók

Speciális adatszerkezetek

Az OS-nek a folyamatok és az I/O egységek kezeléséhez szükséges adatait tárolja

- Process Control Block, PCB
- Input Output Control Block, IOCB

Folyamatleíró blokk

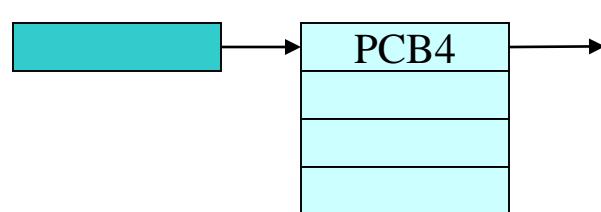
Process Control Block, PCB:

- folyamat azonosítója
- szülők, gyerekek azonosítója
- folyamat állapota
- folyamathoz tartozó összes tárterület leírása (mutatók, virtuális tárkezeléshez tartozó adatok, cím transzformáció)
- a folyamat által használt egyéb erőforrások leírása (pl. nyitott állományok)
- regiszterek tartalma
- várakozó folyamatoknál: várt esemény leírása
- ütemezéshez információk (prioritás, várakozási idő)
- statisztikák

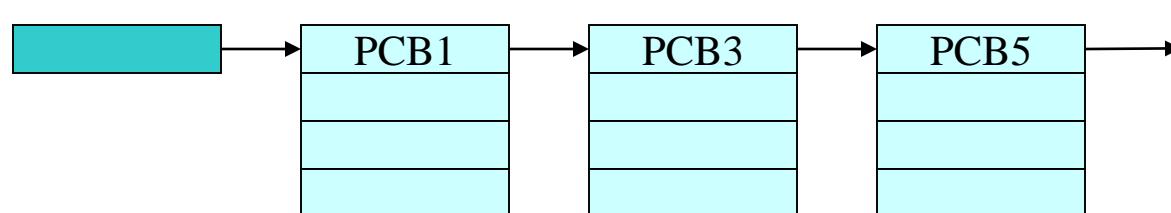
Folyamatleírók kezelése

Láncolt listák:

Futó folyamat



Futásra kész folyamatok



I/O műveletek leírása

Input Output Control Block, IOCB:

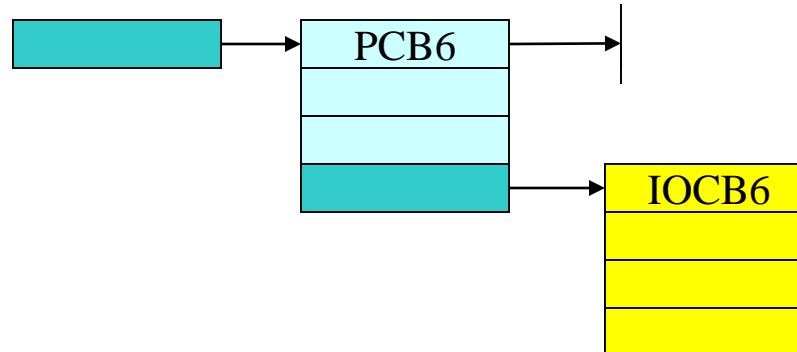
Az I/O művelet végrehajtásához tartozó minden adat:

- művelet kijelölése (írás, olvasás, ...)
- tárterület címe (ahonnan, ahova a művelet végrehajtandó)
- I/O készülék egyéb adatai (mágneslemez szektorcíme, stb.)
- átvendő adatmennyiség
- állapotjelző
- stb...

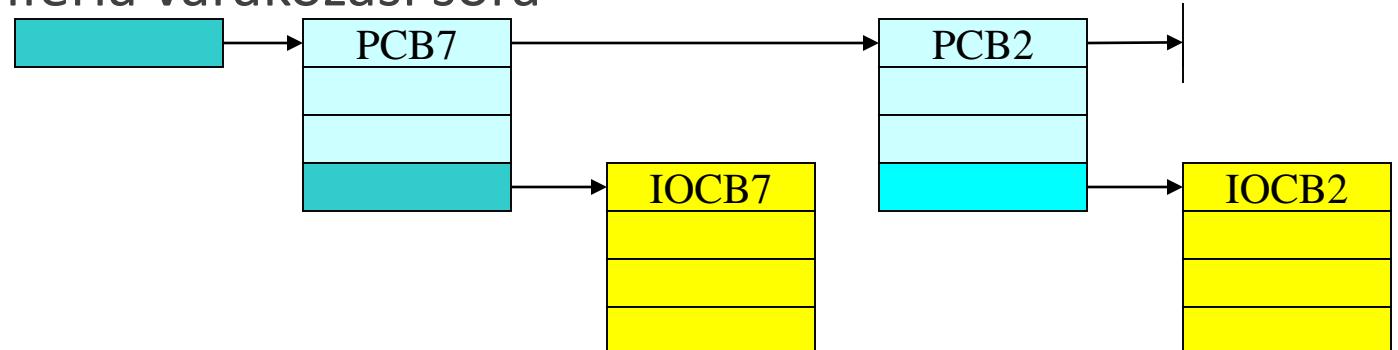
I/O leírók kezelése

Az IOCB a PCB-hez fűződik:

1. periféria várakozási sora



2. periféria várakozási sora



I/O műveletek végrehajtása

Folyamat:

- Kitölti az IOCB-t
- Rendszerhívás, paraméter az IOCB

OS

- Láncolja az IOCB-t a PCB-hez
- PCB-t befűzi a periféria várakozási sorába
- Ha a sor üres, indítja a perifériát az IOCB paramétereivel
- Folyamatot várakozó állapotba teszi
- Újraütemez és visszatér (másik folyamatra)

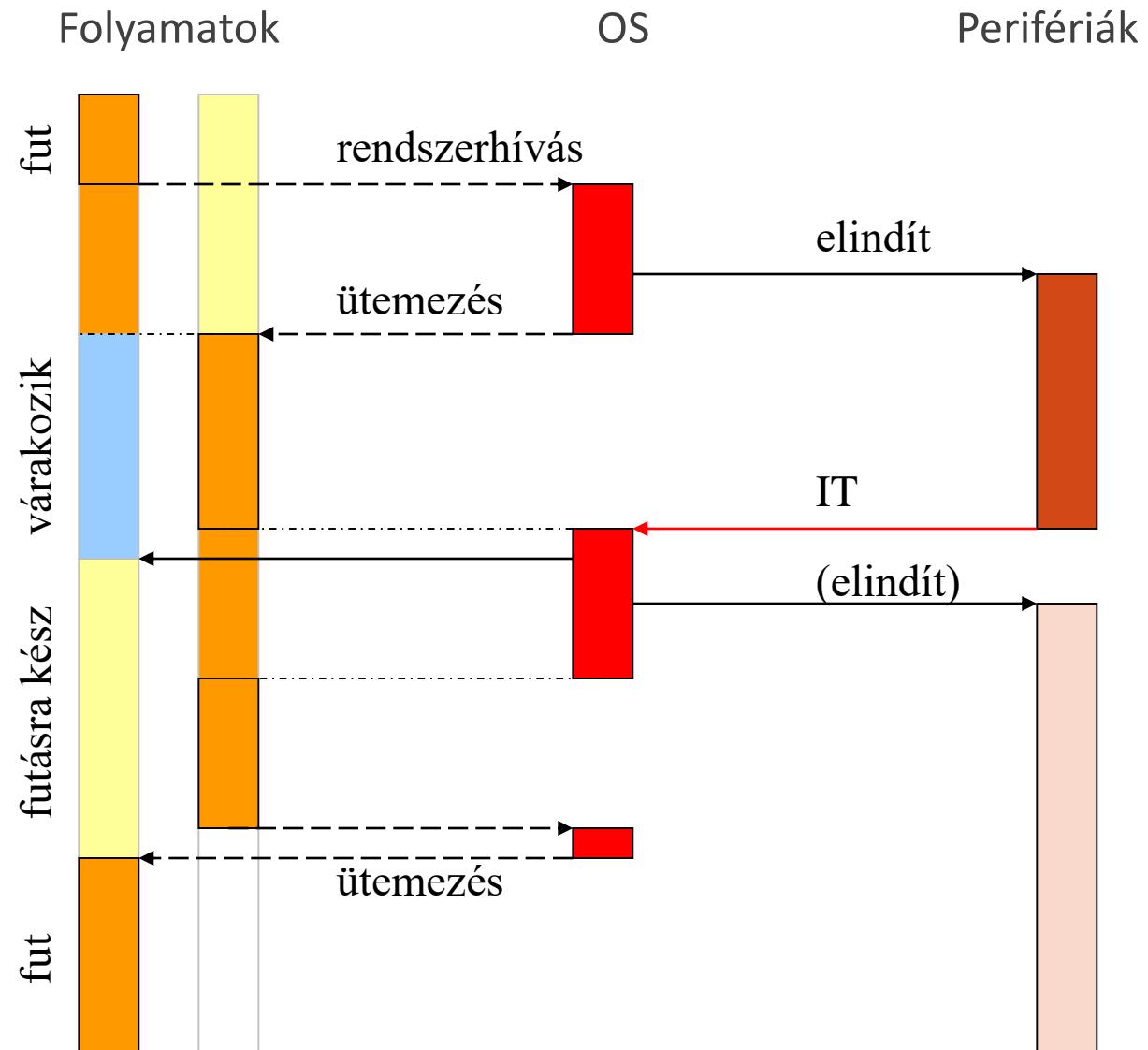
Periféria

- Feladat végeztével megszakítást okoz

OS

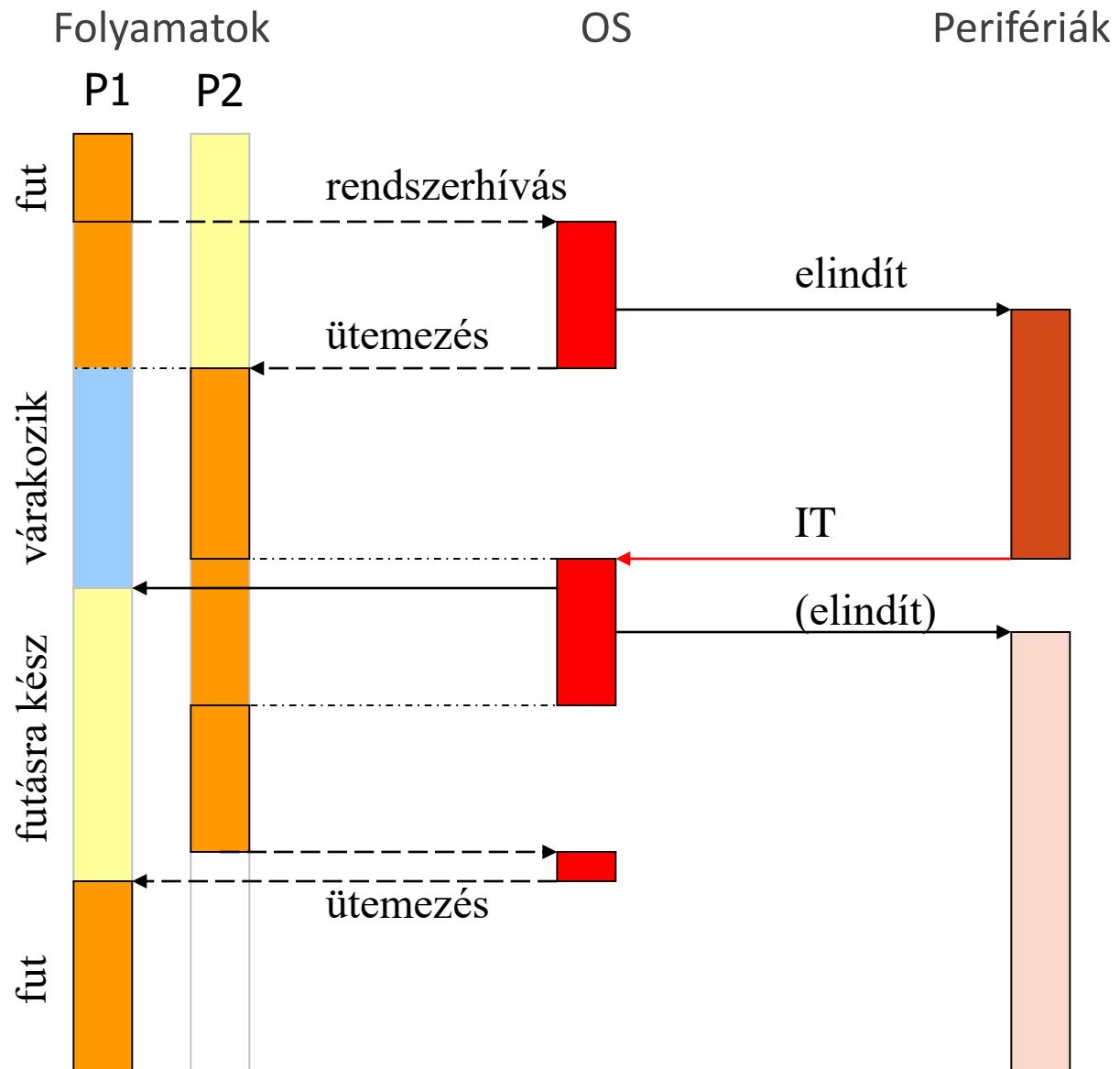
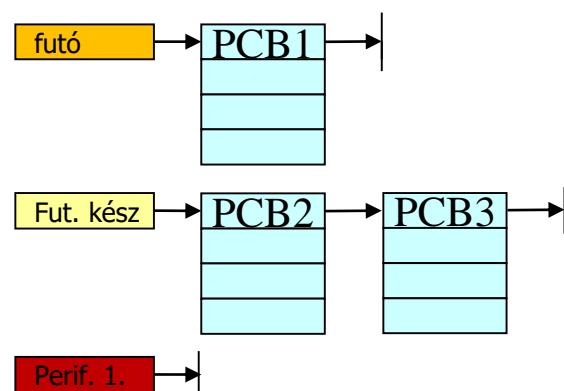
- IOCB-be a végrehajtás eredményére utaló jelzést ír (helyes/helytelen)
- PCB-t a futásra kész állapotba helyezi
- Ha van még várakozó a perifériára, akkor a perifériát indítja
- Újraütemez
- visszatér

I/O műveletek végrehajtása



I/O műveletek végrehajtása

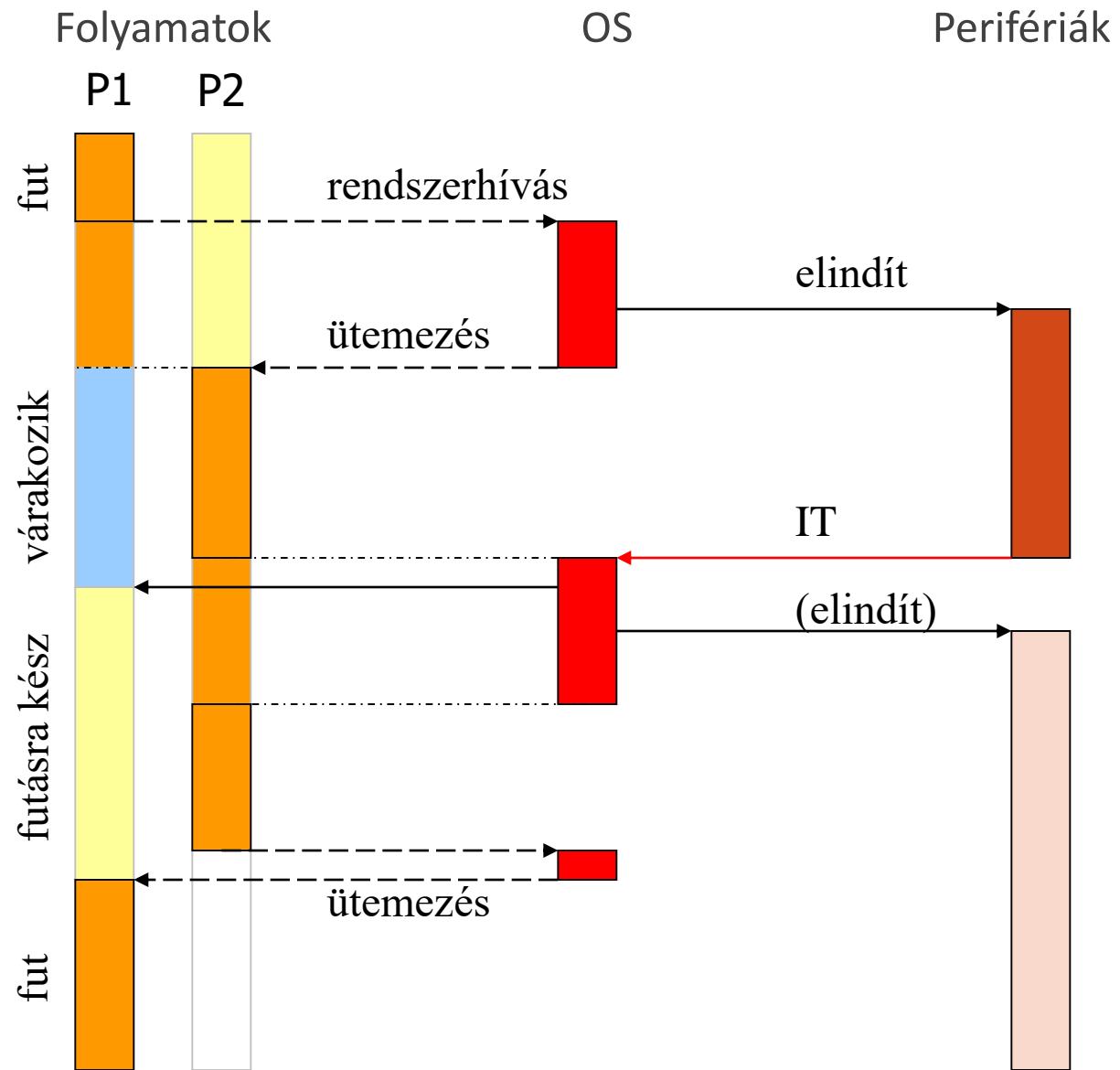
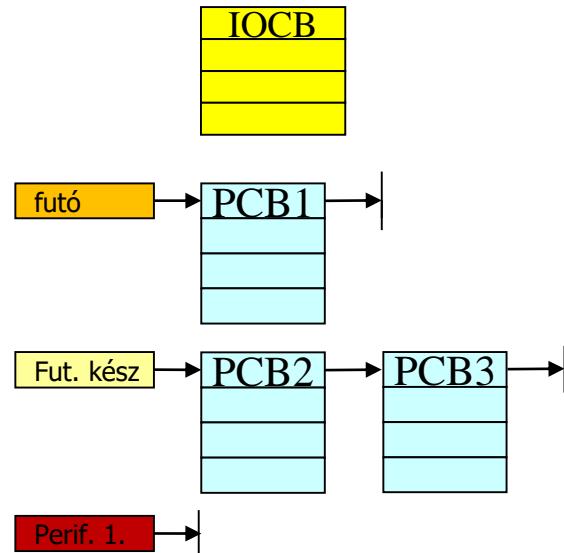
1. Futó folyamat: P1



I/O műveletek végrehajtása

1. Futó folyamat: P1

Perifériát akar használni
Kitölti az IOCB-t
Rendszerhívás:
paraméter az IOCB

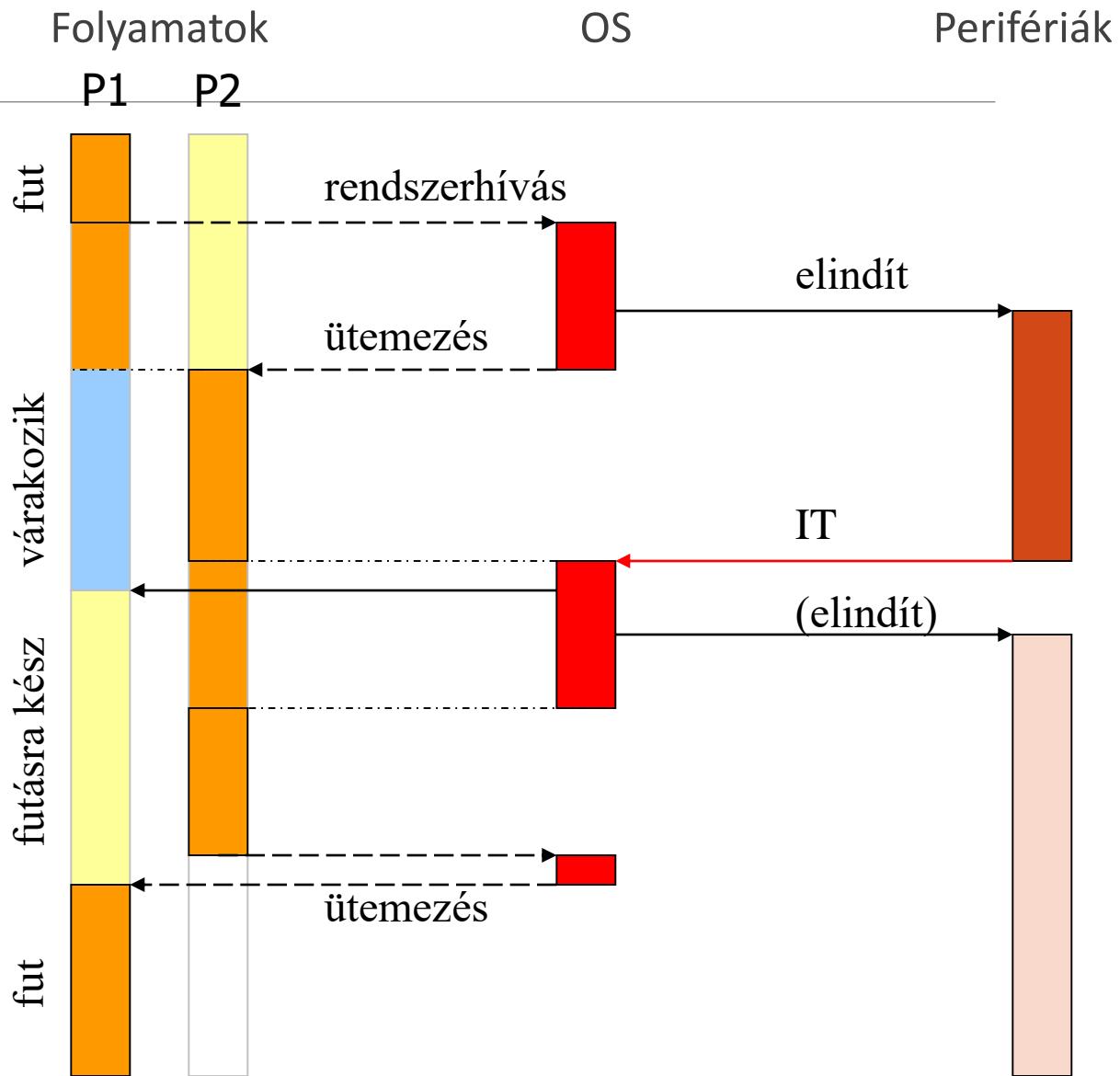


I/O műveletek végrehajtása

2. OS

Láncolja az IOCB-t a PCB-hez
PCB-t befűzi a periféria
várakozási sorába
(P1 folyamat várakozó)

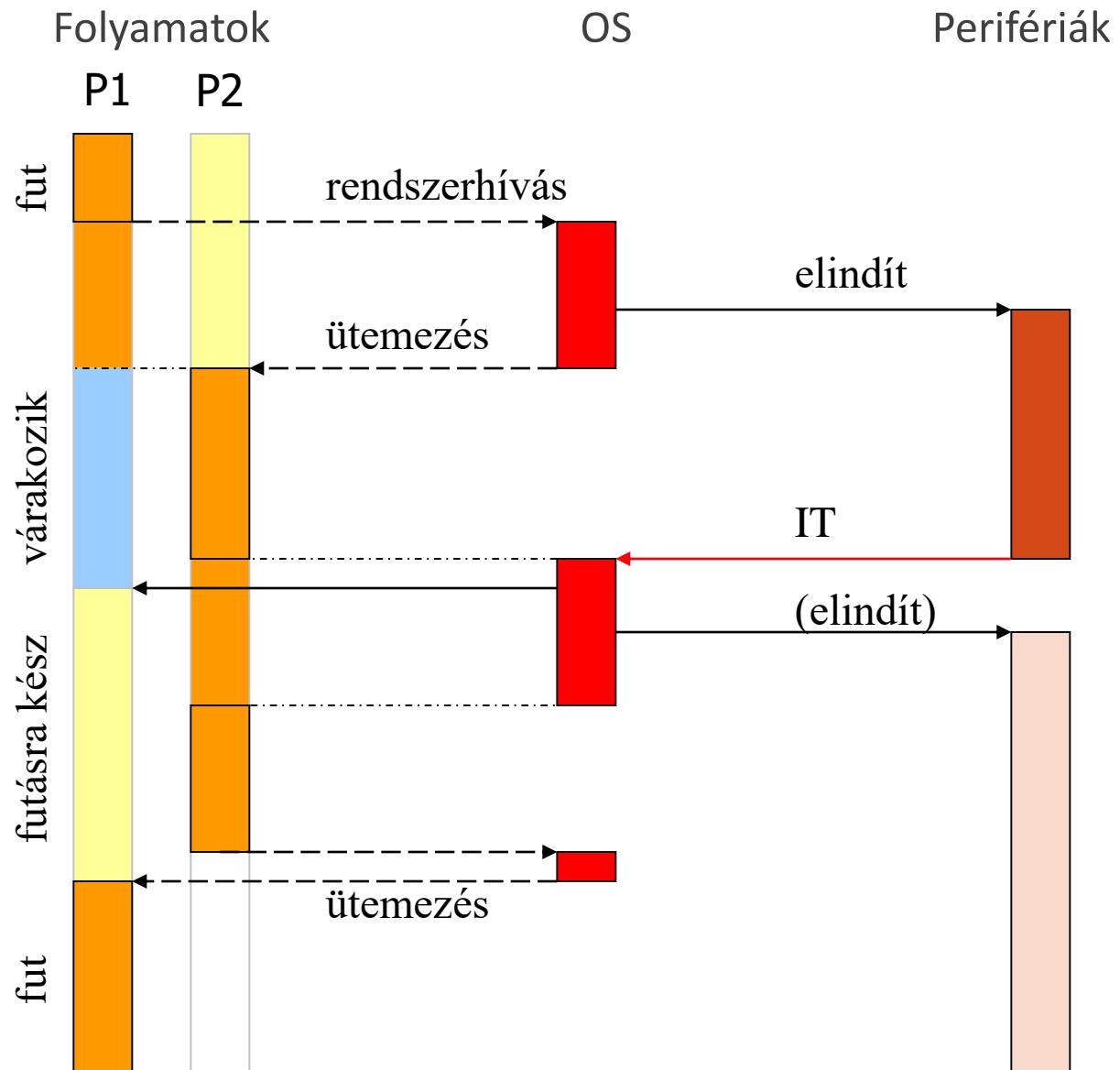
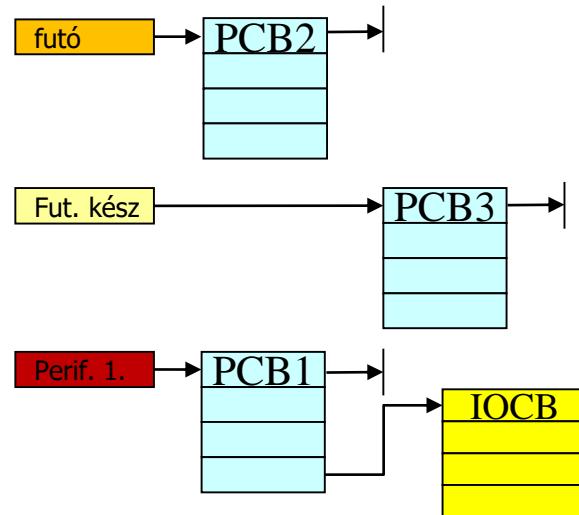
Ha a sor első eleme,
elindítja a perifériaműveletet



I/O műveletek végrehajtása

3. OS

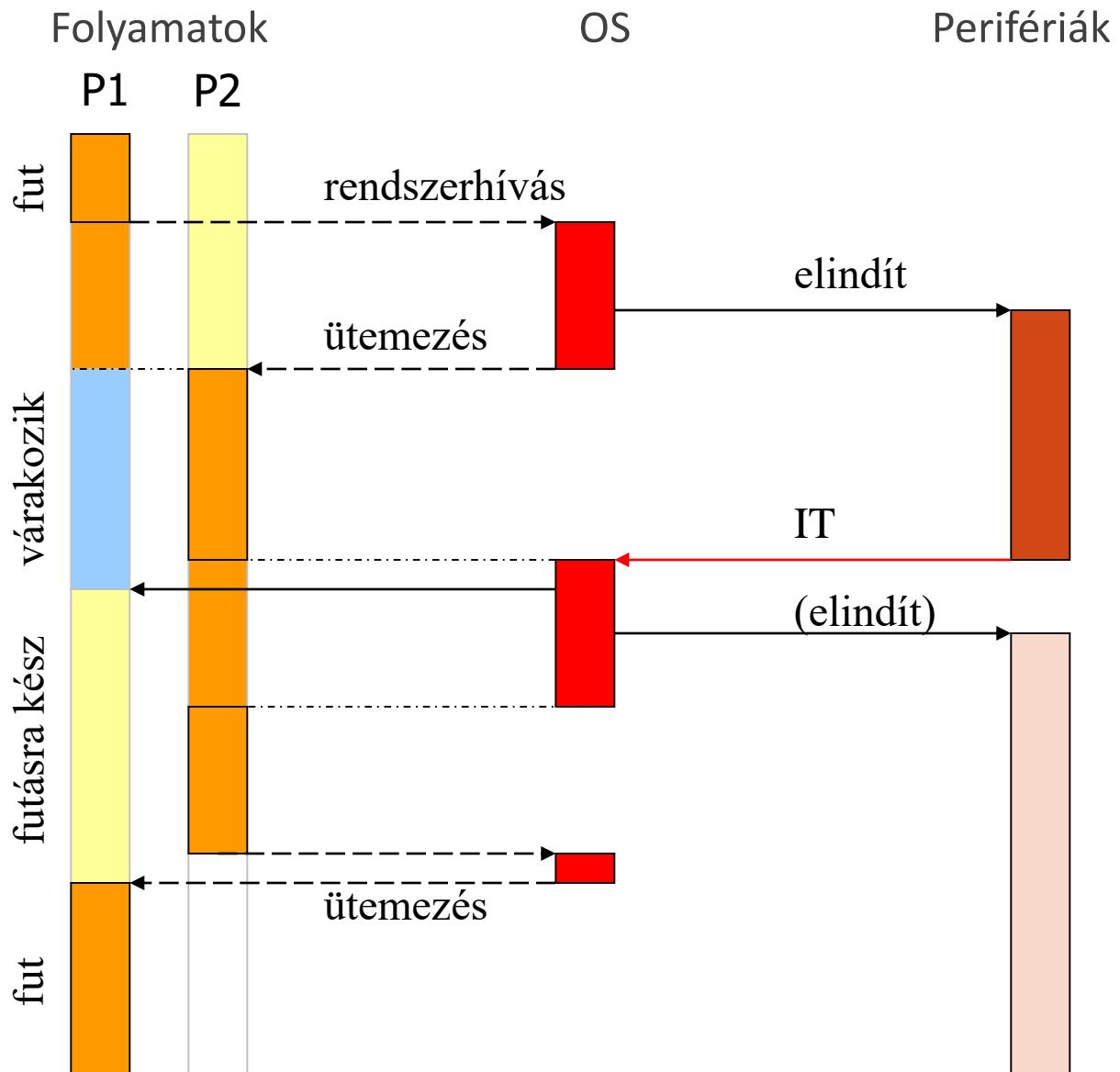
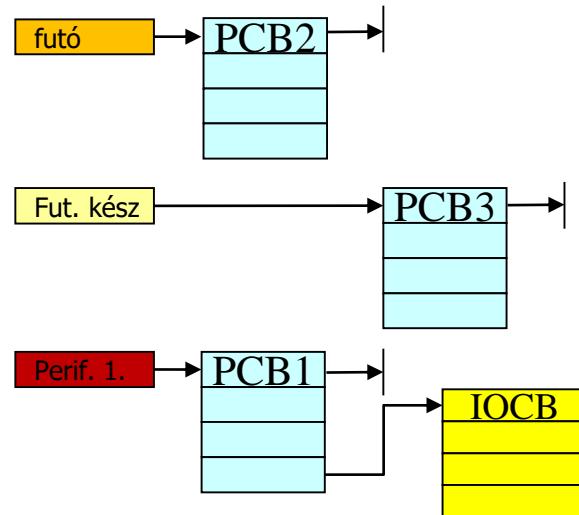
Újraütemez (P2 futó állapot)
Visszatér (P2 fut)



I/O műveletek végrehajtása

4. Periféria

A feladat végeztével megszakítást okoz



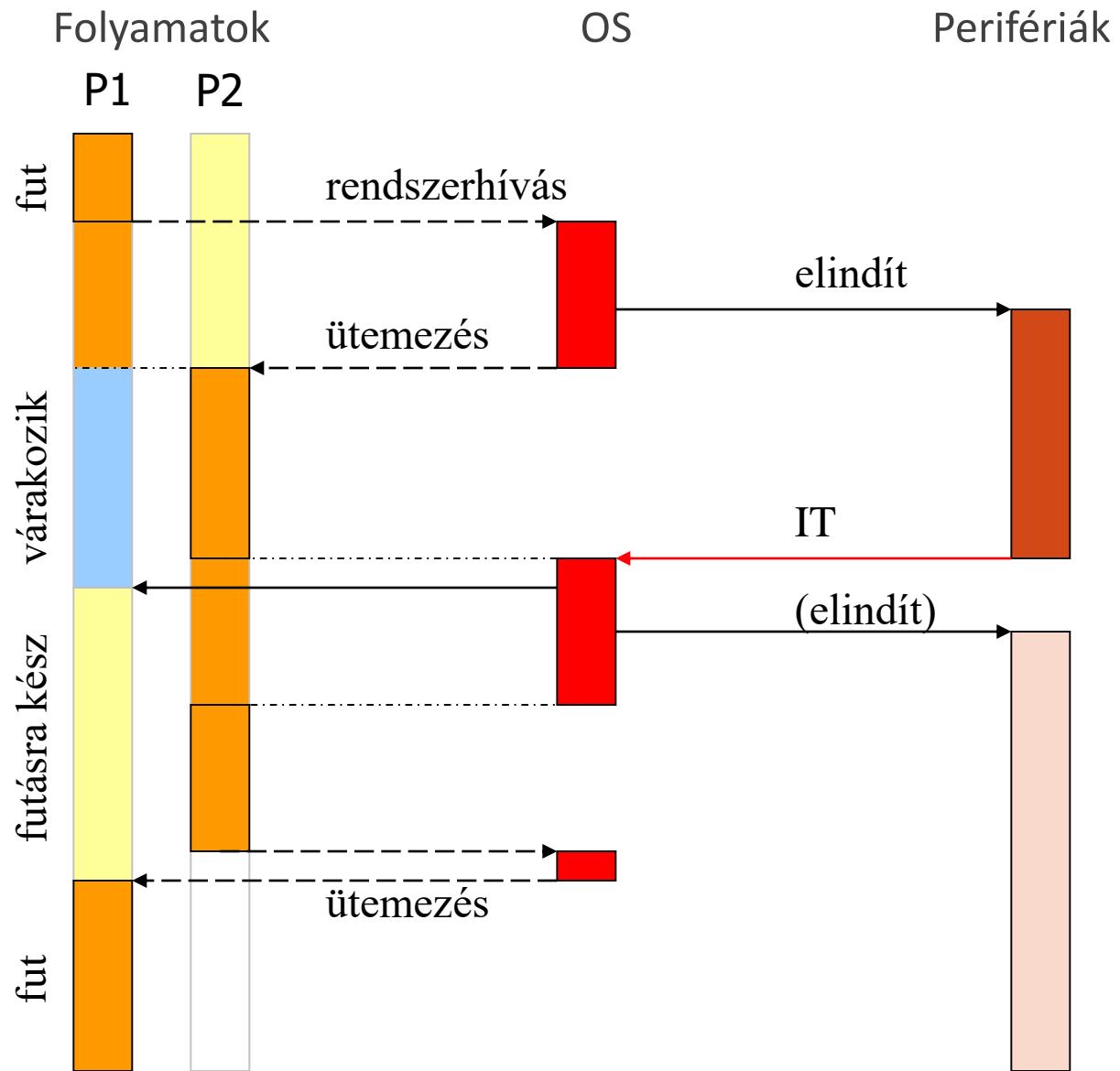
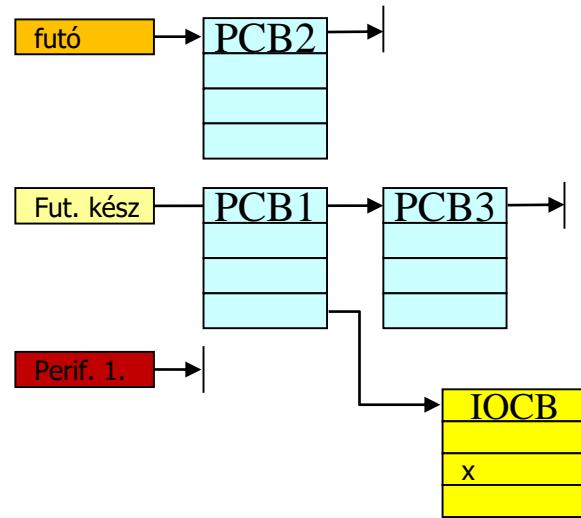
I/O műveletek végrehajtása

4. OS

IOCB-be a végrehajtás eredményére utaló jelzést ír (helyes/helytelen)

PCB: futásra kész állapotba

Ha van még várakozó a perifériára, akkor a perifériát indítja



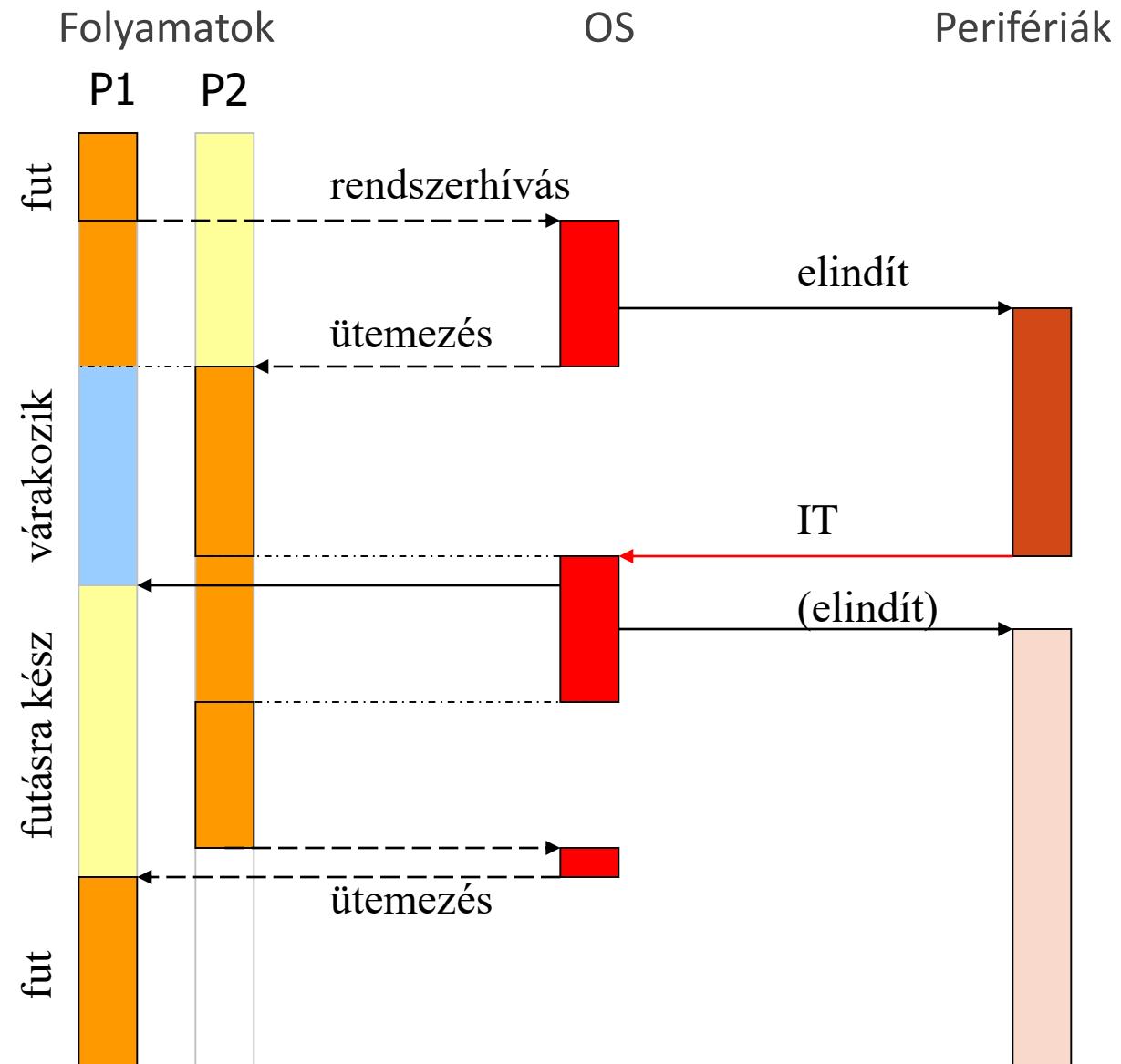
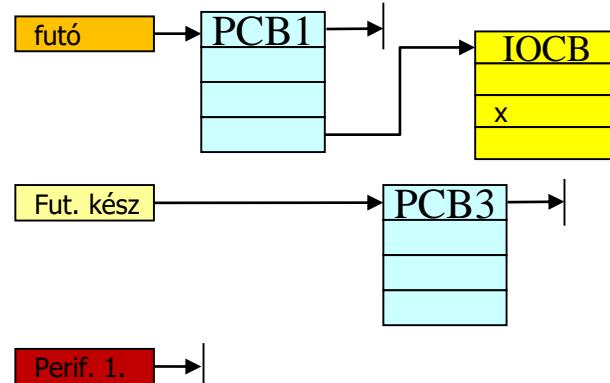
I/O műveletek végrehajtása

5. OS

Újraütemez:

P1: futó állapot
P2: kész

Visszatér (P1 fut)



Szál (thread)

Folyamatokhoz hasonló fogalom

Nincs saját memória, saját erőforrás

- csak a *regiszterek* és a *verem* sajátja (+állapot)
- minden más közös a folyamatával

Előny:

- Gyors váltást tesz lehetővé
- Osztott erőforrások (memória!)

Állapot-átmeneti gráfja megegyezik a folyamatoknál tárgyalattal.

Szálak és folyamatok 1.

Minden folyamatnak van:

Címtér

Globális változók

Megnyitott fájlok

Gyermekek folyamatok

Várt események listája

...

Minden szálnak van:

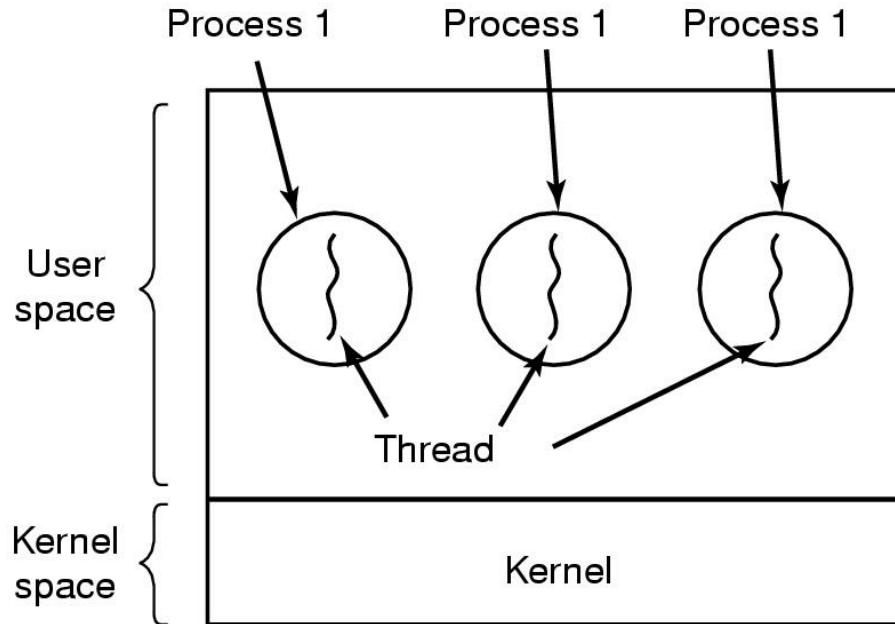
Programszámláló

Regiszterek

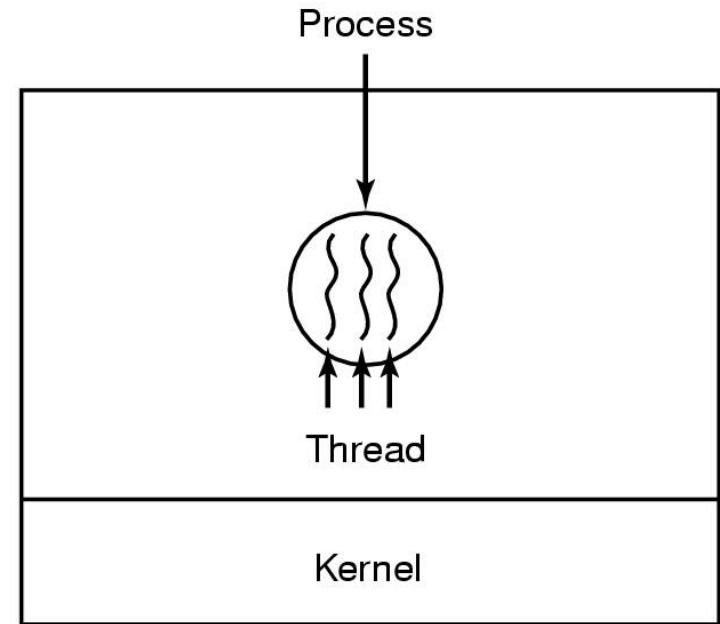
Stack

Állapot

Szálak és folyamatok 2.



(a)



(b)

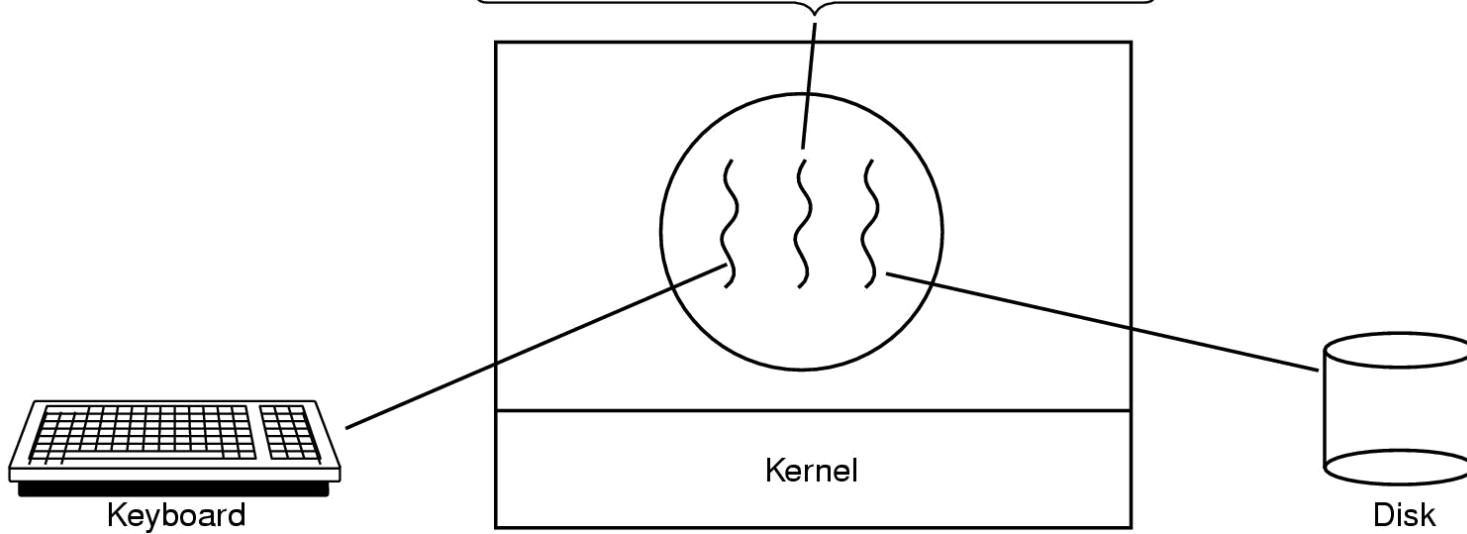
- a) Három folyamat egy-egy szállal
- b) Egy folyamat három szállal

Példa: szövegszerkesztő

Four score and seven years ago, our fathers brought forth upon this continent a new nation, conceived in liberty, and dedicated to the proposition that all men are created equal. Now we are engaged in a great civil war, testing whether that nation, or any nation so conceived and so dedicated, can long endure. We are met on a great battlefield of that war.

We have come to dedicate a portion of that field as a final resting place for those who here gave their lives that this nation might live. It is altogether fitting and proper that we should do this.

But, in a larger sense, we cannot dedicate, we cannot consecrate, we cannot hallow, this ground. The brave men, living and dead, who struggled here have consecrated it, far above our poor power to add or detract. The world will little note, here to the unfinished work which they who fought here have thus far so nobly advanced. It is rather for us to be here dedicated to the great task remaining before us, that we here highly resolve that these dead shall not have died in vain that this nation, under God, shall have a new birth of freedom, and that government of the people, by the people, for the people,

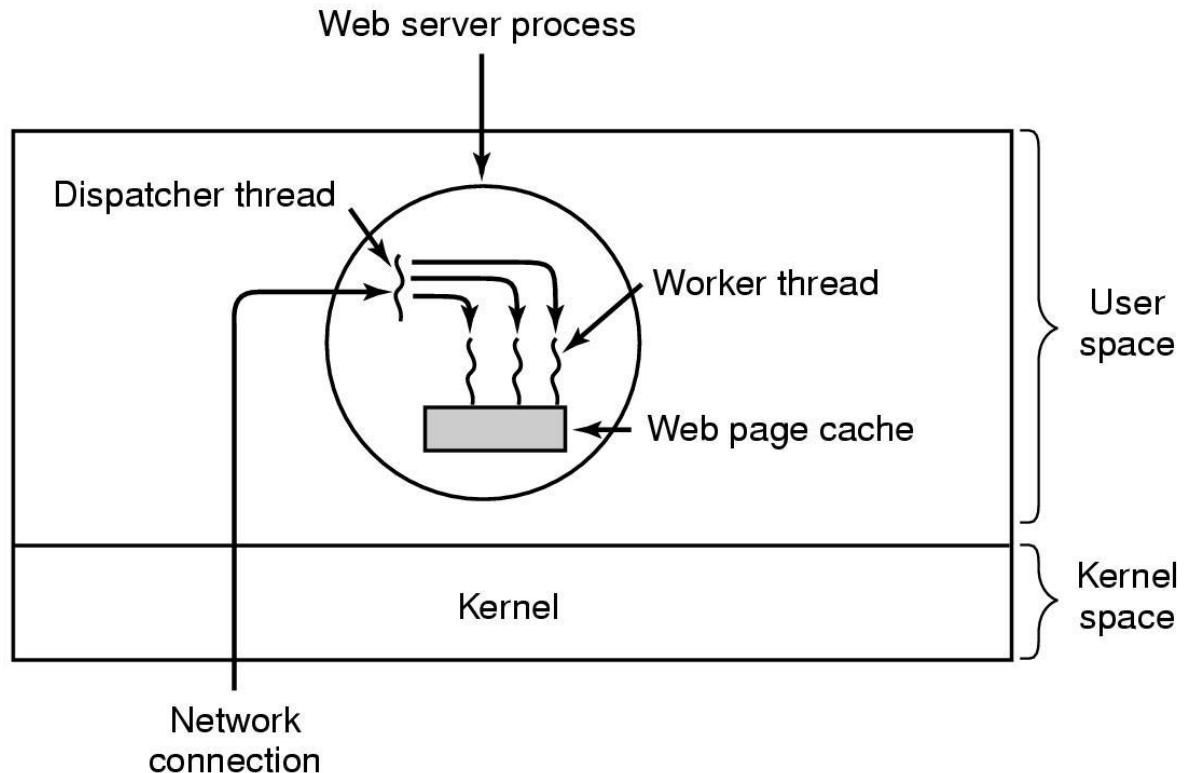


Billentyűzet-kezelő szál

Tördelő szál

Automatikus mentés szál

Példa: web-szerver 1.



Dispatcher szál: fogadja a kérést és kiosztja a munkát

Worker szál: teljesíti a kérést

Példa: web-szerver 2.

```
while (TRUE) {  
    get_next_request(&buf);  
    handoff_work(&buf);  
}
```

(a)

```
while (TRUE) {  
    wait_for_work(&buf)  
    look_for_page_in_cache(&buf, &page);  
    if (page_not_in_cache(&page)  
        read_page_from_disk(&buf, &page);  
    return_page(&page);  
}
```

(b)

- a) *Dispatcher* szál
- b) *Worker* szál

Megszakítások

Interrupt:

Az éppen futó folyamat félbeszakad, az interrupt kezelőre adódik át a vezérlés

Osztályai:

- perifériák (berendezések állapotának változásai, átvitel befejezése)
- belső hardver (timer)
- utasítás végrehajtási hiba (pl. 0-val való osztás, memória hiba, virtuális tárkezelésben laphiba, stb.)
- hardver hiba (pl. tápfeszültség)
- szoftver megszakítás (trap), speciális gépi utasítás pl. rendszerhívás

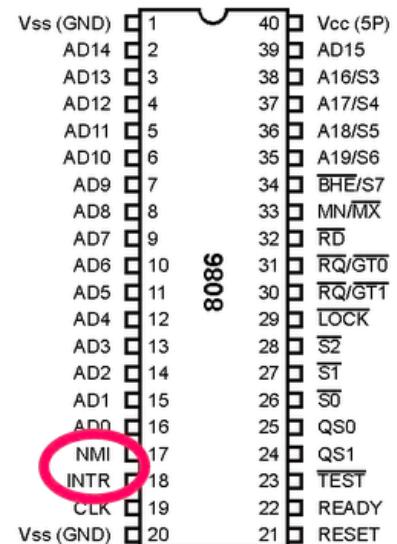
Prioritás

Nem minden megszakítás azonos súlyú

- pl. egy magasabb megszakíthat egy alacsonyabb prioritású kiszolgálást

Megszakítások letilthatók / engedélyezhetők

- csak korlátosztott ideig hiszen pl. a perifériáknál az átvitelkor adatok veszthetnek el).



Megszakítások kezelése

1. Nincs környezet váltás.

Megszakítás hatására OS rutin indul el, folyamat felfüggesztve, csak a kiszolgáló rutin által használt regisztereket mentjük el (ezek általában rövid kis programok).

2. Van környezet váltás (ritkábban, lassabb).

A megszakítás elindít egy arra váró folyamatot. A folyamat várakozóból rögtön futni fog (elkerüli a futásra kész állapotot).

Megszakításkezelés lépései

A futó folyamat megszakad, vezérlés az OS-nek

Megszakított folyamat állapotmentése
(1. vagy 2. módban)

- pl. regiszterek mentése (van HW támogatás)

Vezérlést a kiszolgáló rutin kapja meg

Befejezés után állapot visszaállítás
(1. vagy 2. mód)

Megszakított - vagy egy másik - folyamat folytatja a futását.

Operációs rendszerek

3. FOLYAMATOK KOMMUNIKÁCIÓJA

Felhasznált irodalom:

- **Kóczy-Kondorosi (szerk.): Operációs rendszerek mérnöki megközelítésben**
- **Tanenbaum: Modern Operating Systems**
- **Silberschatz, Galvin, Gagne: Operating System Concepts**

Tartalom

Bevezetés

Szinkronizáció

- Egyidejűség
- Előidejűség
- Kölcsönös kizárás

A kritikus szakasz és megvalósítási módozatai

Folyamatokból álló rendszerek

Egy rendszer sok folyamatból állhat. Ezek között csatolás lehet:

független folyamatok

- egymás működését nem befolyásolják,
- aszinkron működés
 - egymással párhuzamosan is végrehajtódhatnak,
 - Nincs időbeli függőség

függő folyamatok,

- logikailag független folyamatok, de megosztott erőforrás használat (pl. több user, azonos gépen dolgozik)
- logikailag függő folyamatok
 - közösen oldanak meg valamely feladatot
 - együttműködnek, kommunikálnak, közös változók, stb.

Együttműködő folyamatok használatának indokai

Erőforrások megosztása

- átlapolt működés, jobb kihasználtság

Számítások felgyorsulása (több processzor)

- Számítások párhuzamosítása, végrehajtási sebesség nő.

Felhasználók kényelme

- Egy időben több feladat megoldása.

Modularitás

- Egy adott folyamat kisebb részekre való bontása
- Jobb áttekinthetőség
- Bizonyos feladatoknál (párhuzamos részek) kézenfekvő modell.
- A függőséget valahogy biztosítani kell

Szinkronizáció

A folyamat végrehajtásának olyan időbeli korlátozása, ahol ez egy másik folyamat futásától illetve egy külső esemény bekövetkezésétől függ.

Gyakori feladatok:

- Precedencia (előidejűség)
- Egyidejűség
- Kölcsönös kizárás (versenyhelyzet, kritikus szakasz)

Egyéb kapcsolódó fogalmak:

- Holtpont (deadlock)

Precedencia

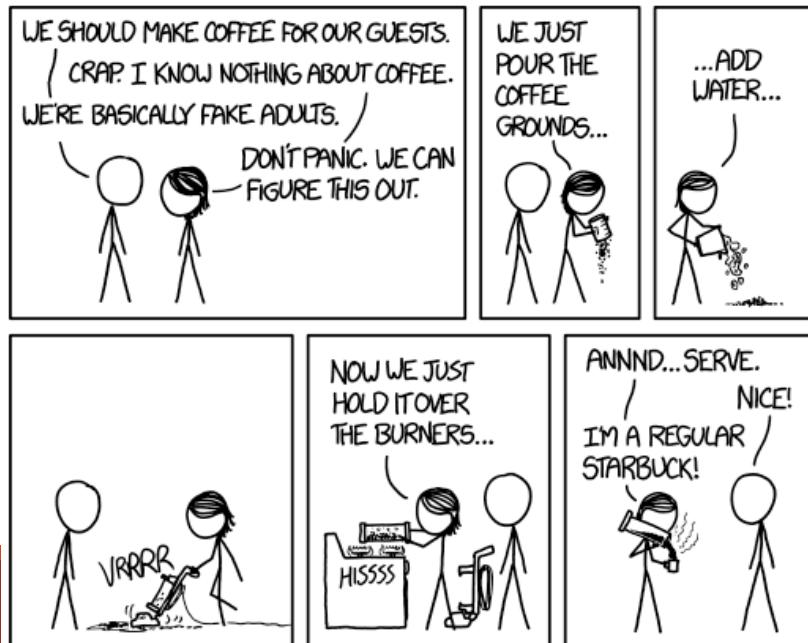


Meghatározott sorrend biztosítása.

Egy P_k folyamat S_k utasítása csak akkor mehet végbe ha a P_i folyamat S_i utasítása már befejeződött.

pl:

kukta-cukrot vesz → szakács-kávét főz
(különben kihűl a kávé)





Egyidejűség

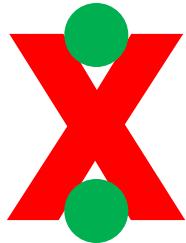
Két vagy több folyamat bizonyos utasításait ($S_k; S_j$) egyszerre kell elkezdeni.

Két folyamat találkozása (randevú).

Két folyamat bevárja egymást mielőtt további működését elkezdené.

pl.

szakács-kávét főz || kukta-habot ver
(különben kihűl a kávé vagy összeesik a hab)



Kölcsönös kizárás (mutual exclusion)

A résztvevő folyamatok utasításainak sorrendjére nincs korlátozás, de egy időben csak egyik futhat.

pl.

szakács-habot ver X kukta-habverőt mosogat



Versenyhelyzet

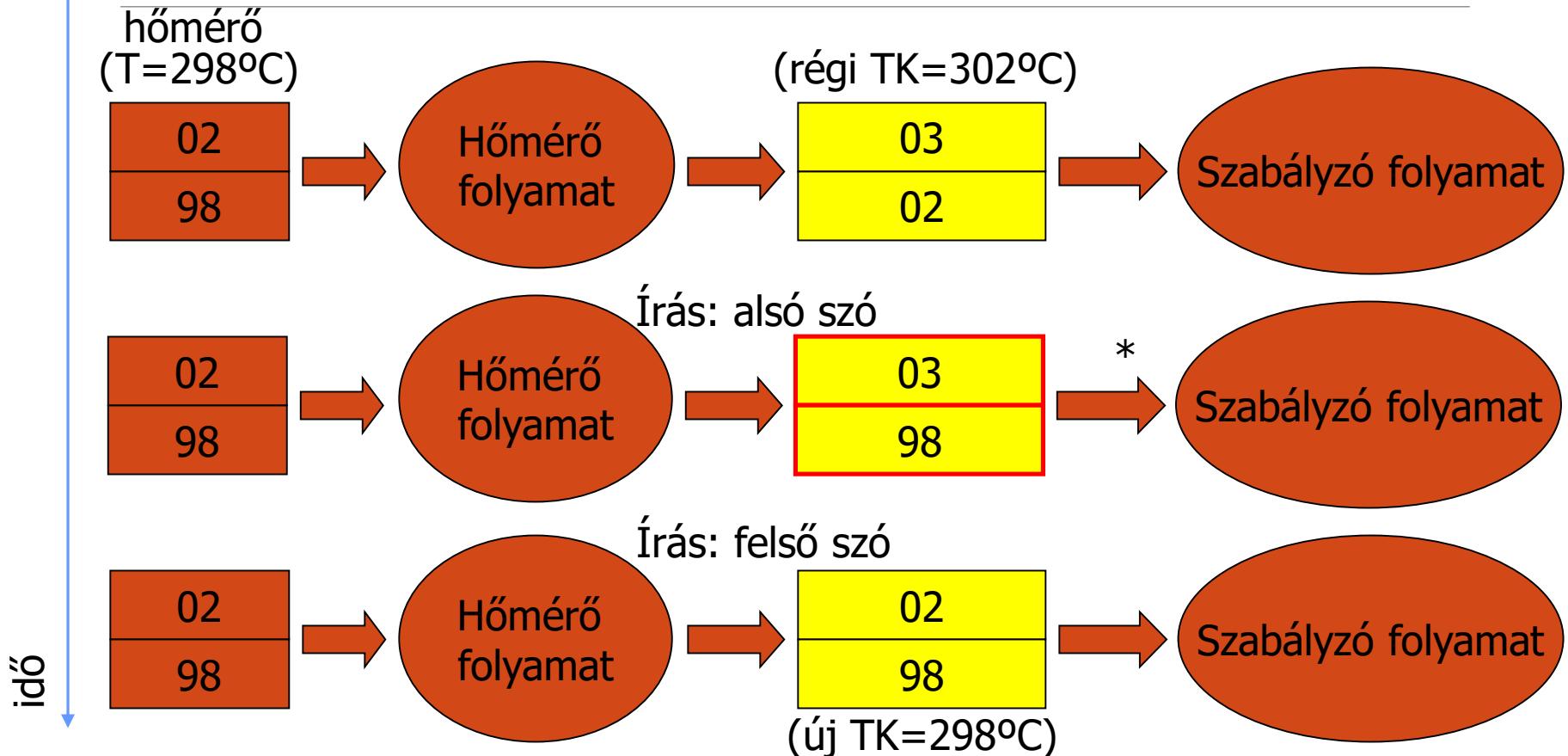
Több párhuzamosan futó folyamat közös erőforrást használ. A futás eredménye függ attól, hogy az egyes folyamatok mikor és hogyan futnak, ezáltal hogyan (milyen sorrendben) férnek az erőforráshoz.

Elkerülendő, nagyon nehéz debuggolni!

Példa:

A mért hőmérséklet (T) értékét egy két szó hosszúságú változóban (TK) tároljuk. A hőmérő folyamat a hőmérsékletet szavanként beírja a változóba, a szabályzó folyamat pedig kiolvassa a változót és annak értékét használja.

Példa versenyhelyzetre: hőmérséklet-szabályzó



*Ha ebben a pillanatban olvas, hibás, inkonzisztens értéket kap.

Kritikus szakasz

Kritikus szakaszoknak nevezzük a program olyan (általában osztott változókat használó) utasításszekvenciáit, amelyeknek egyidejű (párhuzamos) végrehajtása nem megengedett.

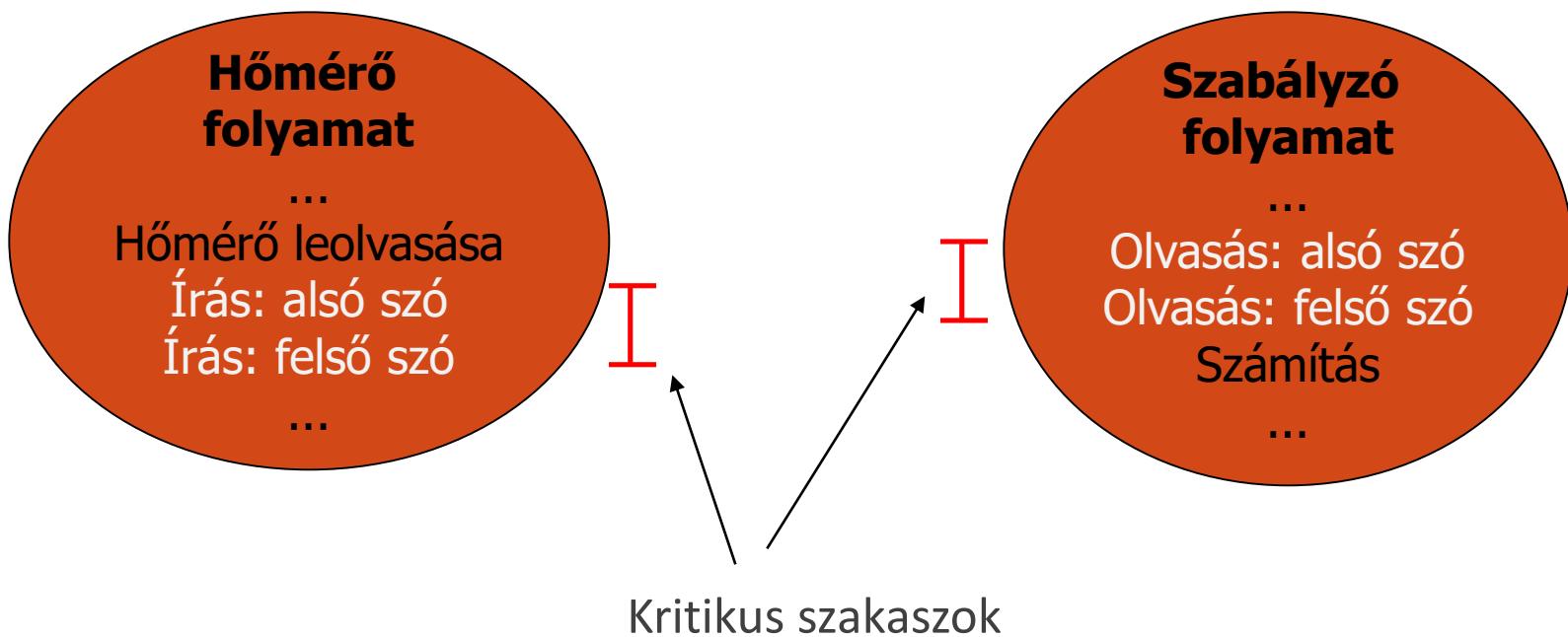
Versenyhelyzet elkerülésére a kritikus szakaszok kölcsönös kizárást biztosítani kell.

(Ha az egyik folyamat már a kritikus szakaszában van, akkor más folyamat nem léphet be a (természetesen saját) kritikus szakaszába.)

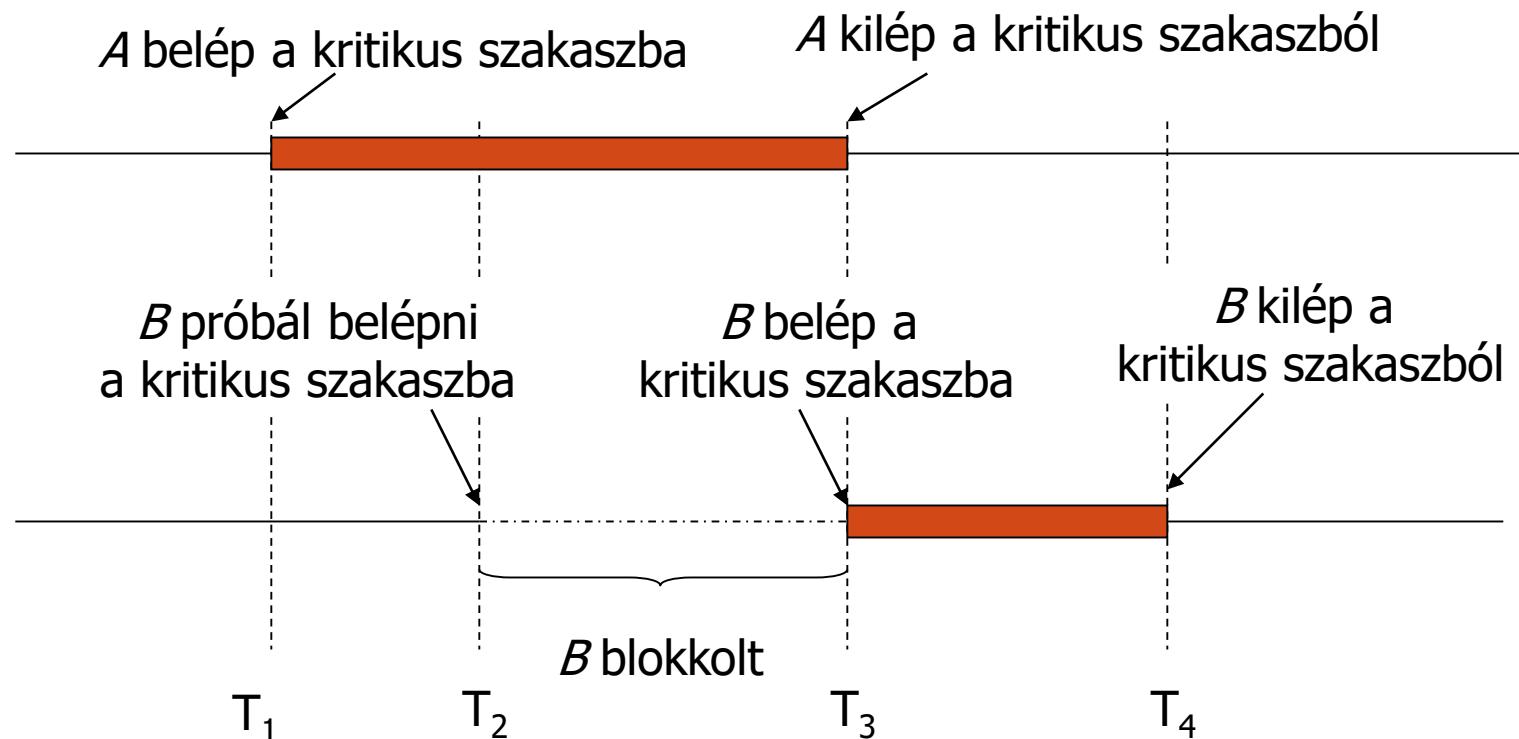
Pl.

Tipikus kritikus szakasz a közös memória használata.

Példa kritikus szakaszra: hőmérséklet-szabályzó



Kritikus szakaszok és a kölcsönös kizáráás



A kritikus szakaszban lévő A folyamat blokkolja B végrehajtását, amikor az is a kritikus szakaszba kíván lépni (T_2). Amikor A kilép a kritikus szakaszból (T_3), a blokkolás véget ér és B most már beléphet a kritikus szakaszba.

A kritikus szakasz megvalósítási kritériumai

A megvalósítás követelményei

- 1.** Biztosítsa a kölcsönös kizárást
 - Egy időben csak egyetlen folyamat hajthatja végre a kritikus szakaszban lévő utasításokat.
- 2.** Haladjon
 - Ha nincs folyamat a kritikus szakaszban, de van több belépő, akkor az algoritmus ezek közül véges idő alatt kiválaszt egyet és beengedi a kritikus szakaszba.
- 3.** Folyamat várakozása korlátozott legyen (ne éheztessen)
 - Csak véges számú esetben előzhetik meg.

A 3. kritériumot nem veszik annyira szigorúan (nem minden algoritmus teljesíti).

Kritikus szakasz megvalósítása

A folyamat általános struktúrája:

```
//Process i  
  
while (TRUE) {  
    non_critical_region1();  
    entry_section();  
    critical_region();  
    exit_section();  
    non_critical_region2();  
}
```

Kritikus szakasz megvalósítási módozatai

Interrupt tiltása

Busy waiting megoldások

- Nincs HW támogatás, tiszta SW megoldás.
 - Naiv megközelítés (ROSSZ)
 - Strict alternation (JOBB)
 - Peterson algoritmusa (HELYES)
- HW támogatással:
 - TestAndSet

Szemafor

Magas szintű módszerek

Interrupt tiltása

Megoldás:

```
disable_interrupt();  
critical_region();  
enable_interrupt();
```

Egyszerű megoldás:

- Nincs IT, tehát nem lehet a végrehajtást megszakítani. (Ütemező sem tud futni!)

Probléma:

- A folyamatnak joga van letiltani az IT-t. Az egész rendszert lefagyaszthatja.

Hasznos az OS szintjén

Nem célszerű a felhasználói szinten.

Naiv programozott megközelítés

Közösen használt változókon (flag) alapul.

Egy foglaltsági bitet használ, amit a belépni kívánó folyamat tesztel.

Gond: a változót többen is olvashatják, mielőtt az első foglaltra állítja!

```
//Process 0
while (TRUE) {
    while (flag!=0) /*loop*/;
    flag = 1;
    critical_region();
    flag = 0;
    non_critical_region();
}
```

```
//Process 1
while (TRUE) {
    while (flag!=0) /*loop*/;
    flag = 1;
    critical_region();
    flag = 0;
    non_critical_region();
}
```



Szigorú váltás (*strict alternation*) algoritmus

Közös változó: *turn*. Jelentése: ki van soron.

Csak felváltva lehet belépni.

Gond:

- Ha jelentős a sebessékgükönbség, akkor az egyik feleslegesen sokat vár.
- Nem halad!

```
//Process 0
while (TRUE) {
    while (turn!=0) /*loop*/;
    critical_region();
    turn = 1;
    non_critical_region();
}
```

```
//Process 1
while (TRUE) {
    while (turn!=1) /*loop*/;
    critical_region();
    turn = 0;
    non_critical_region();
}
```



Peterson algoritmusa 1.

```
define FALSE 0
define TRUE 1
define N 2
int turn;
int interested[N];
void enter_region(int process);
{
    int other;
    other = 1 - process;
    interested[process] = TRUE;
    turn = process;
    while (turn == process && interested[other] == TRUE) /* nop */;
}
void leave_region(int process);
{
    interested[process] = FALSE;
}
```

Peterson algoritmusa 2.

Ha csak egy folyamat akar belépni, az `enter_region()` visszatér és a folyamat a kritikus szakaszba léphet.

A másik folyamat addig nem léphet be, amíg a `leave_region()` az `interested` változót nem törli.

Ha két folyamat egyszerre akar belépni, a `turn` változót utoljára állító folyamat várakozik, míg a másik végre nem hajtja a saját `leave_region()` eljárását.

Megjegyzés: A `turn` változó írásának megszakíthatatlanak kell lenni (ez könnyen teljesíthető).

Hardver támogatás kritikus szakasz megvalósításához

Speciális megszakíthatatlan gépi utasítások

1. TestAndSet

- Kiolvassa és visszaadja a bit értéket, majd azonnal 1-be állítja.

```
int TestAndSet(int *flag) {
    tmp = *flag;
    *flag = 1;
    return tmp
}
```

oszthatatlan

2. Swap

- Két változó értékét cseréli fel

```
void swap(int *a, int *b) {
    tmp = *a;
    *a = *b;
    *b = tmp;
}
```

oszthatatlan

Kritikus szakasz TestAndSet utasítással

```
//init  
  
flag = 0;  
  
...  
  
//Process i  
  
while (TRUE) {  
    while (TestAndSet(&flag) !=0) /*empty loop*/;  
    critical_region();  
    flag = 0; /*exit critical section*/  
    non_critical_region();  
}
```

Nem garantálja az éhezés kiküszöbölését!

Kritikus szakasz swap utasítással

```
//init  
  
flag = 0;  
  
...  
  
//Process i  
  
...  
  
while (TRUE) {  
    myflag = 1;  
    while (myflag!=0) swap(&myflag, &flag);  
    critical_region();  
    flag = 0; /*exit critical section*/  
    non_critical_region();  
}
```

Nem garantálja az éhezés kiküszöbölését!

Szemafor

E. W. Dijkstra, 1965



Speciális adattípus (nemnegatív egészek),

Az s szemafor-változó két oszthatatlan utasítással érhető el:

P(s) - vizsgál/belép művelet

- Működési elv:
while $s < 1$ do üres utasítás;
 $s = s - 1$

V(s) – kilép művelet

- Működési elv:
 $s = s + 1$

Mutex

A mutex olyan speciális szemafor, amelynek csak két értéke lehet:

- nyitott (unlocked)
- zárt (locked)

Műveletei:

- mutex_lock
- mutex_unlock



Szinkronizáció szemaforral: előidejűség

Előidejűség: P1 folyamat U_1 utasítása előbb hajtódjon végre,
mint P2 folyamat U_2 utasítása

Incializálás:

$s = 0$

$P_1 : \dots U_1; V(s); \dots$

$P_2 : \dots P(s); U_2; \dots$

Szinkronizáció szemaforral: randevú

Randevú: U_1 és U_2 „egyszerre” hajtódjon végre

Incializálás:

$s_1 = 0; s_2 = 0$

$P_1 : \dots V(s_1); P(s_2); U_1; \dots$

$P_2 : \dots V(s_2); P(s_1); U_2; \dots$

Szinkronizáció szemaforral: kölcsönös kizárás

Kölcsönös kizárás: U_1 és U_2 kódrészletek ne tudjanak egyszerre végrehajtódni

Incializálás:

$s := 1$

$P_1 : \dots P(s); U_1; V(s); \dots$

$P_2 : \dots P(s); U_2; V(s); \dots$

Az s kezdő értékétől függ, hogy hány folyamat lehet egyszerre a kritikus szakaszban.

Legtöbbször 1, ezért általában bináris szemafort (mutex) használunk.

Szemaforok megvalósítása

Probléma a szemafor elvi megvalósításával: busy waiting

$P(s)$:

```
while s < 1 do üres utasítás;  
s := s - 1
```

- Ez blokkolt állapotban is állandóan futna, állandóan használná a CPU-t. Így nem használható.

Multiprogramozott megoldás:

- A $P(s)$ az üres utasítás helyén a folyamattól elveszi a CPU-t és várakozó állapotba teszi (feljegyzi a szemaforhoz tartozó valamilyen adatszerkezetekbe).
- A $V(s)$ -nél nem csak a számláló nő, hanem egy folyamatot (ha van) futásra kész állapotba tesz (többféle stratégia lehet).

Szemaforok megvalósítása

```
type semaphore = record  
    value: integer  
    list : list of process  
end;
```

```
P(s) :  
    s.value = s.value - 1  
    if s.value < 0 then  
        begin  
            a folyamat felfűzése a s.list-re  
            a folyamat felfüggesztése, újra ütemezés  
        end
```

V(s) :

```
    s.value = s.value + 1
```

```
    if s.value <= 0 then
```

```
    begin
```

leveszünk egy folyamatot s.list -ról

felébresztjük ezt a folyamatot

```
    end
```

Monitor

A szemafor hasznos szinkronizációs eszköz, de alacsony szintű.

- Sok hibalehetőség, ráadásul nehéz a hibakeresés.

Nyelvi elemekkel való támogatás:

- *monitor*, egy magas szintű szinkronizációs primitív.

A monitor eljárások, változók, adatszerkezetek speciális gyűjteménye.

A monitor eljárásai szabadon hívhatóak, de a változókhoz nem lehet kívülről közvetlenül hozzáférni.

Egyszerre csak egy eljárás lehet aktív a monitoron belül!

Monitor példa

Egyszerű monitor példa:

Csak a *producer* és a *consumer* eljárások láthatók

Az *i* és *c* belső változókat csak a belső eljárások kezelhetik

Legfeljebb egy eljárás lehet aktív: vagy a *producer*, vagy a *consumer*

```
monitor example
  integer i;
  condition c;

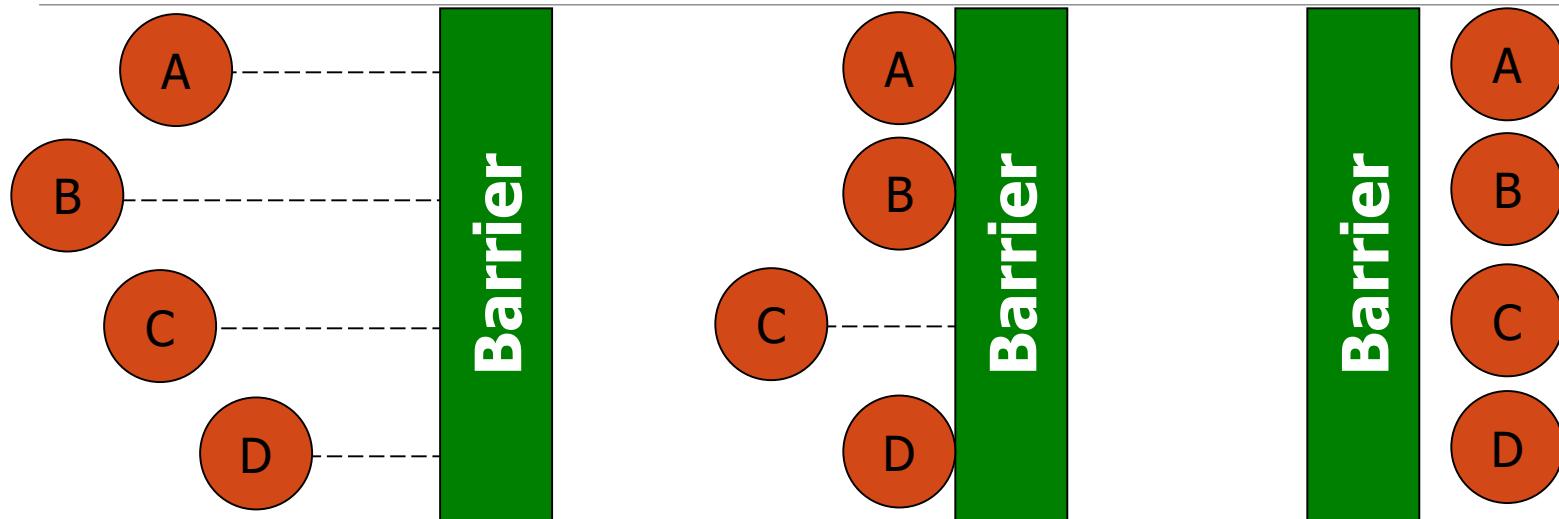
  procedure producer();

  end;

  procedure consumer();

  end
end monitor;
```

Az akadály (barrier)



- a) Több folyamat (A, B, C, D) szinkronizálására szolgál
- b) Az akadályt elérő folyamatok blokkolódnak
- c) Az összes folyamat megérkezésekor az akadály ledől, az összes folyamat egyszerre folytatja futását.

Operációs rendszerek

4-5. ÜTEMEZÉS

Felhasznált irodalom:

- Kóczy-Kondorosi (szerk.): Operációs rendszerek mérnöki megközelítésben
- Tanenbaum: Modern Operating Systems
- Silberschatz, Galvin, Gagne: Operating System Concepts

Tartalom

Bevezetés

CPU ütemezés

Ütemezési algoritmusok alapjai

Ütemezési algoritmusok

Bevezetés

OS egyik legfontosabb feladata

- a rendszer erőforrásainak kezelése,
- a folyamatoknak a futásukhoz szükséges erőforrásokkal való ellátása
- bizonyos szempontok alapján
 - gazdálkodási,
 - védelmi, stb.

Az ütemezés (scheduling) az a tevékenység, amely eredményeként eldől, hogy az adott erőforrást a következő pillanatban mely folyamat használhatja.

CPU ütemezés

Három különböző szint van:

Hosszútávú (long term) ütemezés vagy munka ütemezés:

Középtávú (medium term) ütemezés

Rövidtávú (short term) ütemezés

Nem minden OS-ben van meg mindegyik típusú ütemezés.

Hosszútávú ütemezés

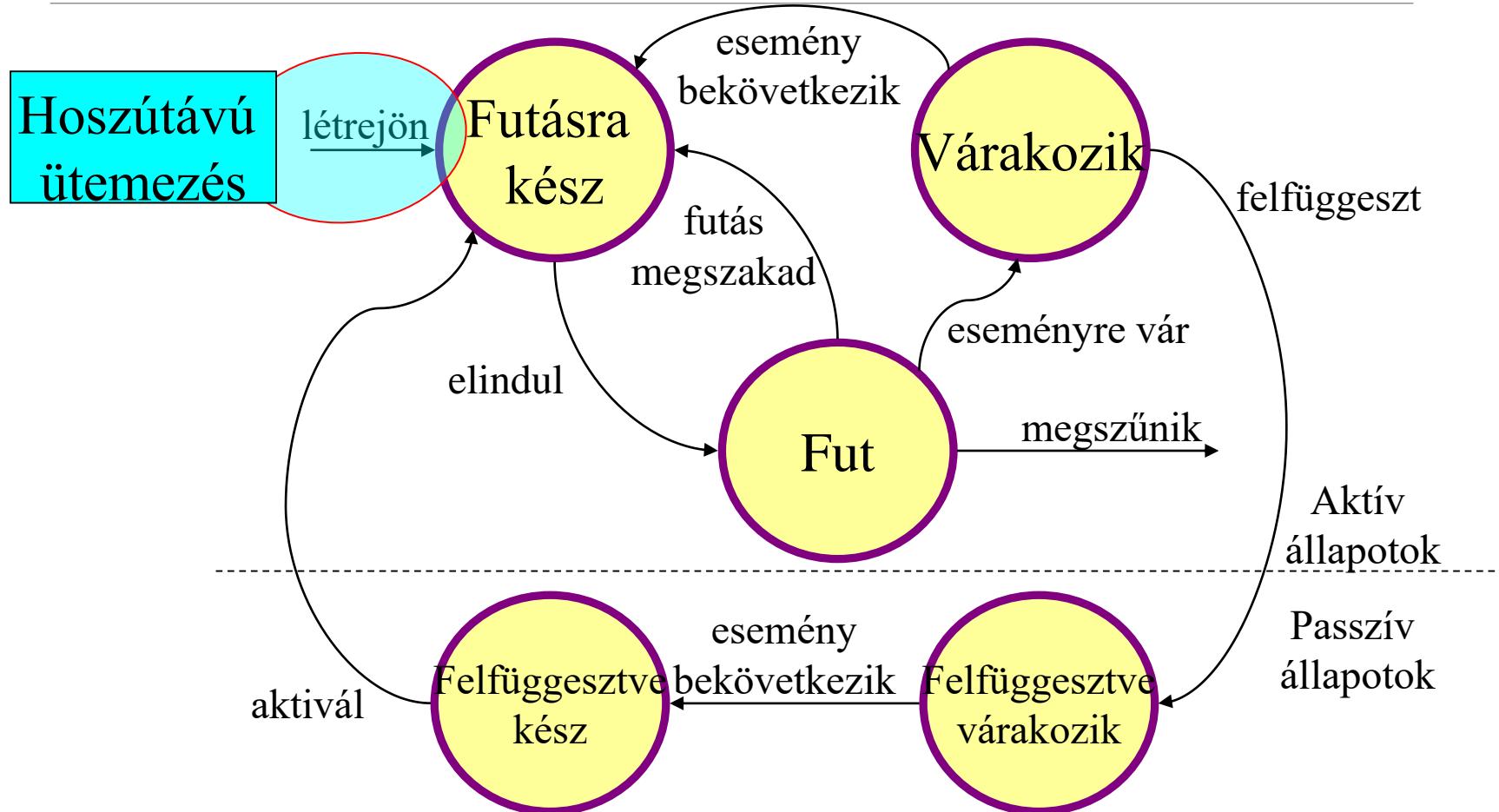
Hosszútávú (*long term*) ütemezés vagy munka ütemezés:

- A háttértáron várakozó, még el nem kezdett munkák közül melyek kezdjenek futni.
- Ritkán kell futnia; egy munka befejeződésekor választunk ki egy új elindítandót.
- Ritkán fut, így nem kell, hogy gyors legyen.

Követelmény:

- olyan munka-halmaz (*job-mix*) előállítása, amely a rendszer erőforrásait kiegyszűlyozottan használja.
- CPU-korlátozott és periféria-korlátozott munkák egyenletesen forduljanak elő.

Emlékeztető: folyamatok kibővített állapotai



Középtávú ütemezés

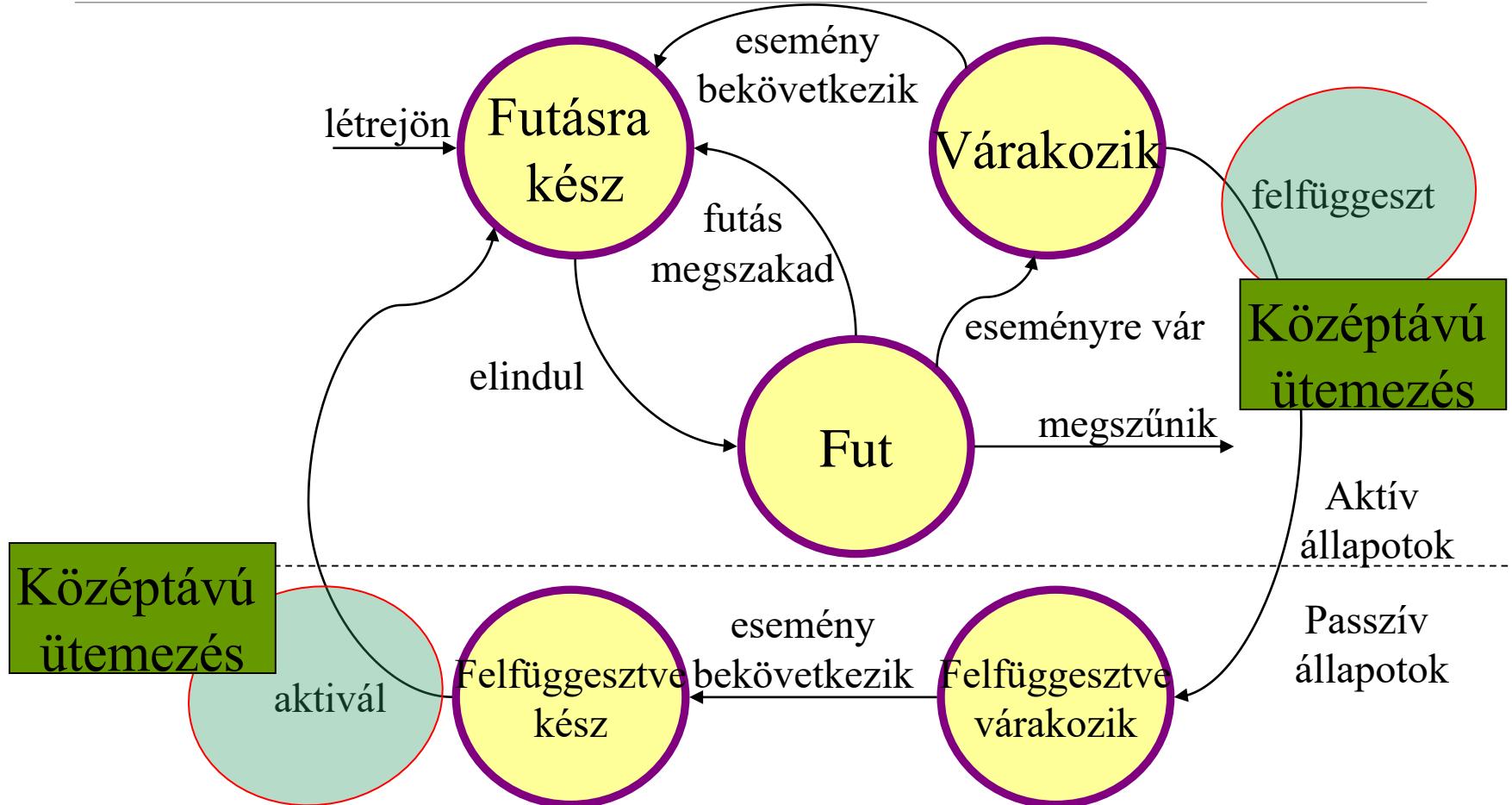
Középtávú (*medium term*) ütemezés

A rendszer időszakos terhelés-ingadozásait egyes folyamatok felfüggesztésével illetve újraaktiválásával próbálja kiegyenlíteni.

Felfüggesztés esetén folyamat a háttértáron tárolódik, megfosztják elvezető erőforrásaitól.

Az ilyen folyamatok nem versengenek tovább az erőforrásokért.

Emlékeztető: folyamatok kibővített állapotai

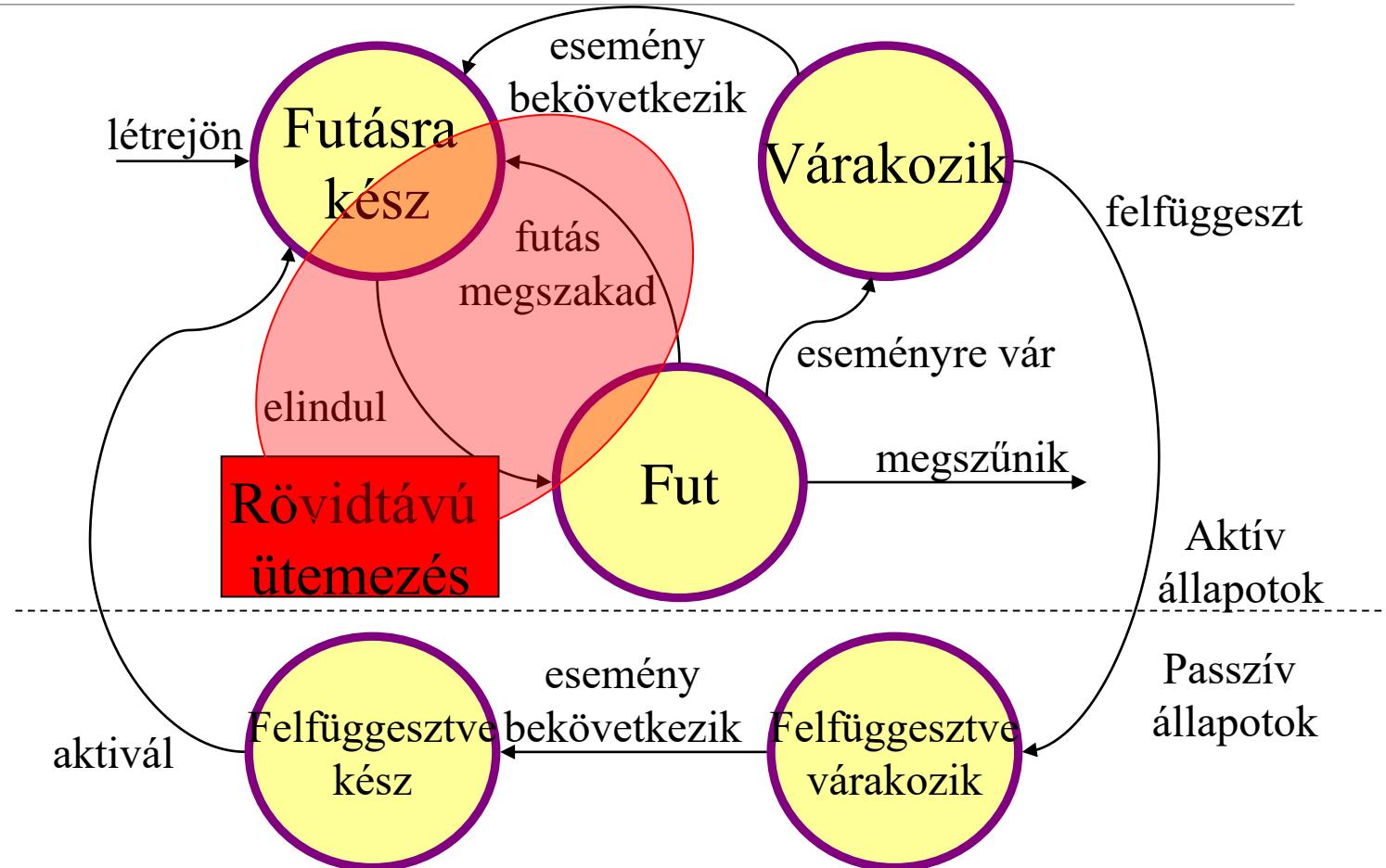


Rövidtávú ütemezés

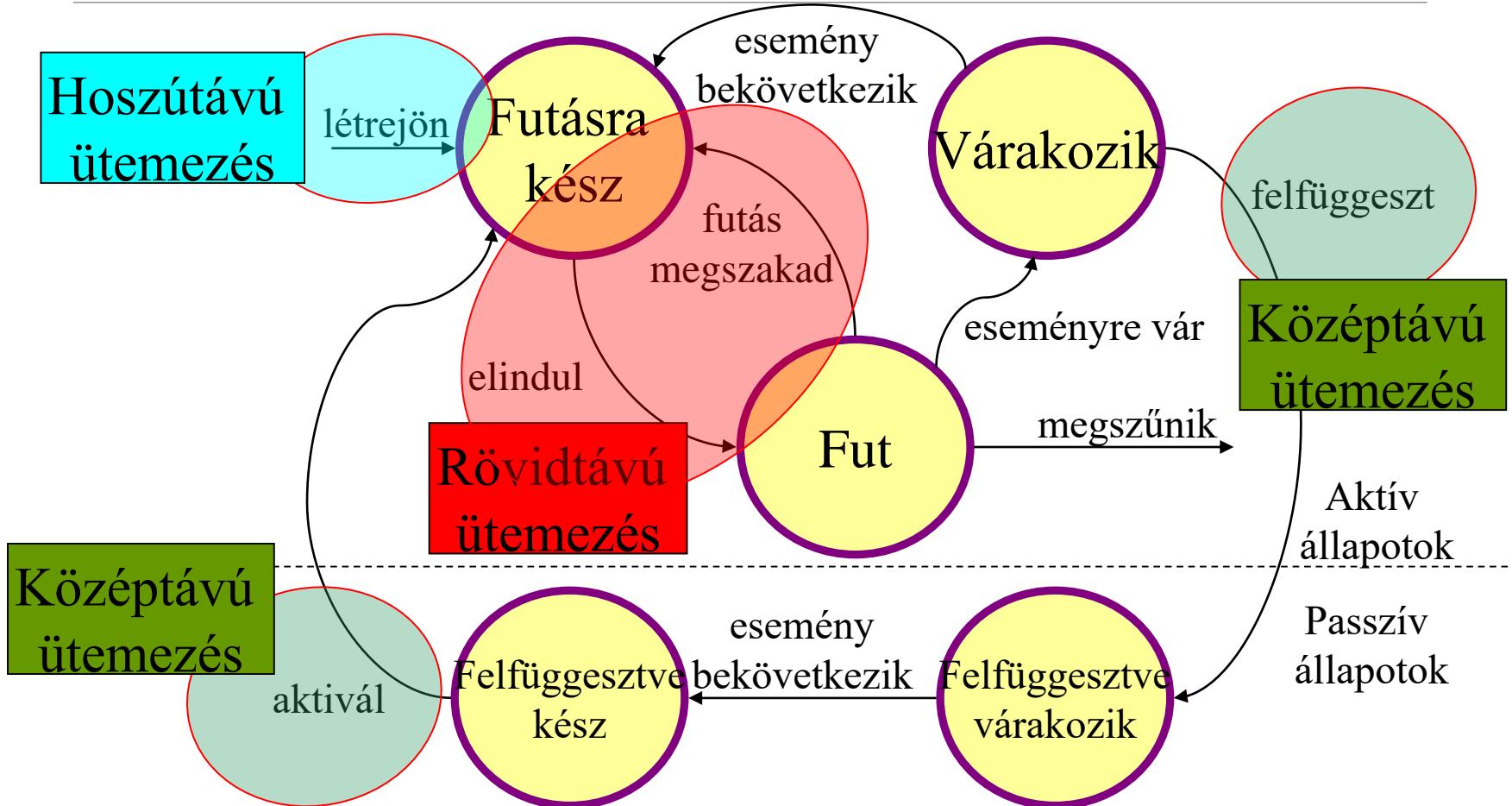
Rövidtávú (*short term*) ütemezés

- Mely futásra kész folyamat kapja meg a CPU-t (kerül futó állapotba)
- Gyakran fut, ezért gyorsnak kell lennie, különben a rendszer túl sok időt töltene az ütemezéssel, elvéve a CPU-t a folyamatoktól.
- Az ütemező mindig a tárban van, része a OS magjának.

Emlékeztető: folyamatok kibővített állapotai



Emlékeztető: folyamatok kibővített állapotai



Az ütemezési algoritmusok alapjai

Egy folyamat futása során két különböző jellegű tevékenységet végezhet:

- **CPU löket** (CPU burst) ideje alatt a folyamatnak csak CPU-ra és az operatív tárra van szüksége.
- **Periféria löket** (I/O burst) alatt a folyamat perifériás átvitelt hajt végre, annak lezajlására várakozik.

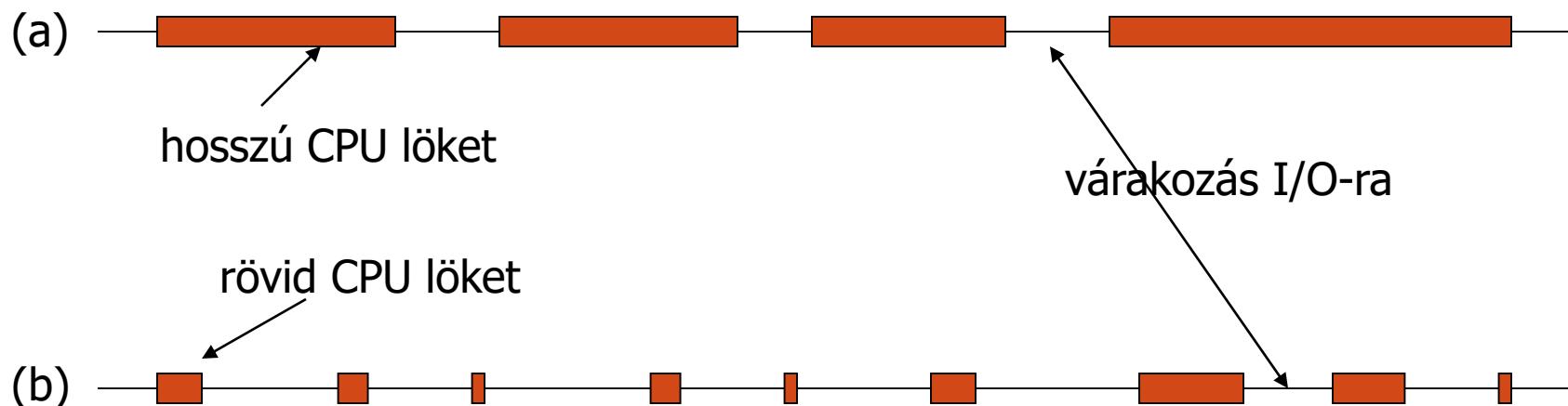
A folyamatnak periféria löket alatt nincs szüksége a CPU-ra.

CPU löket átlagos hossza folyamatonként változik

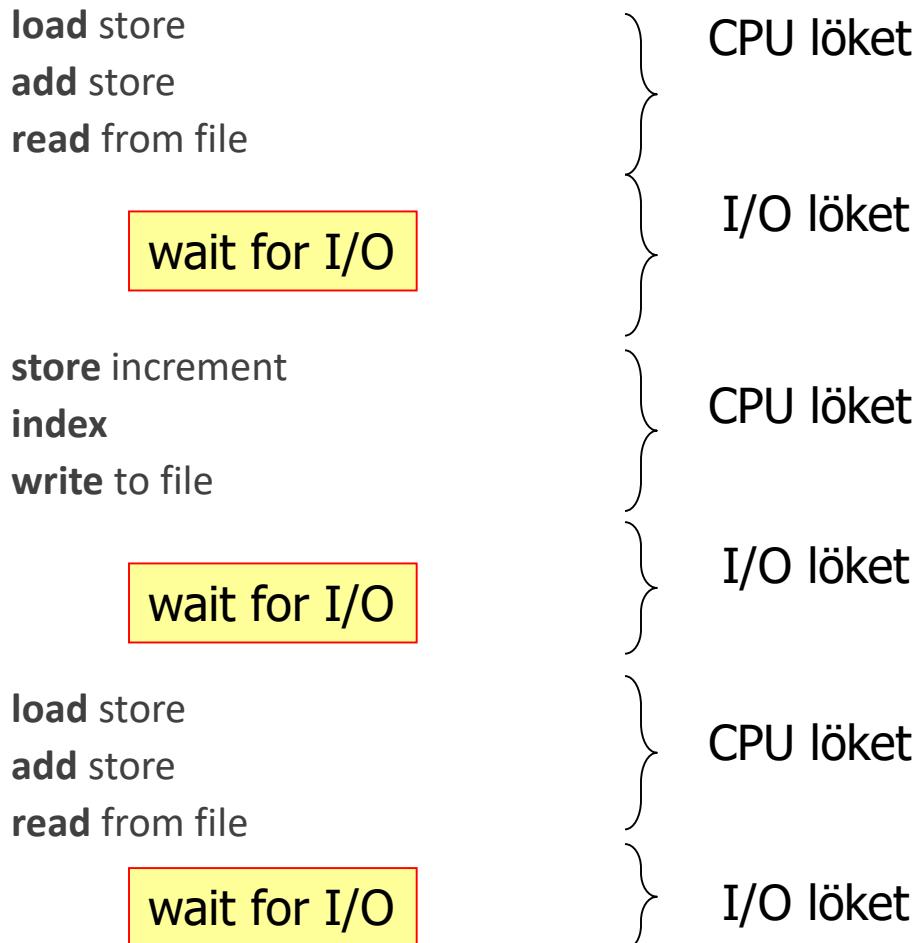
A CPU löket és az IO löket 1.

(a) CPU-korlátos folyamat

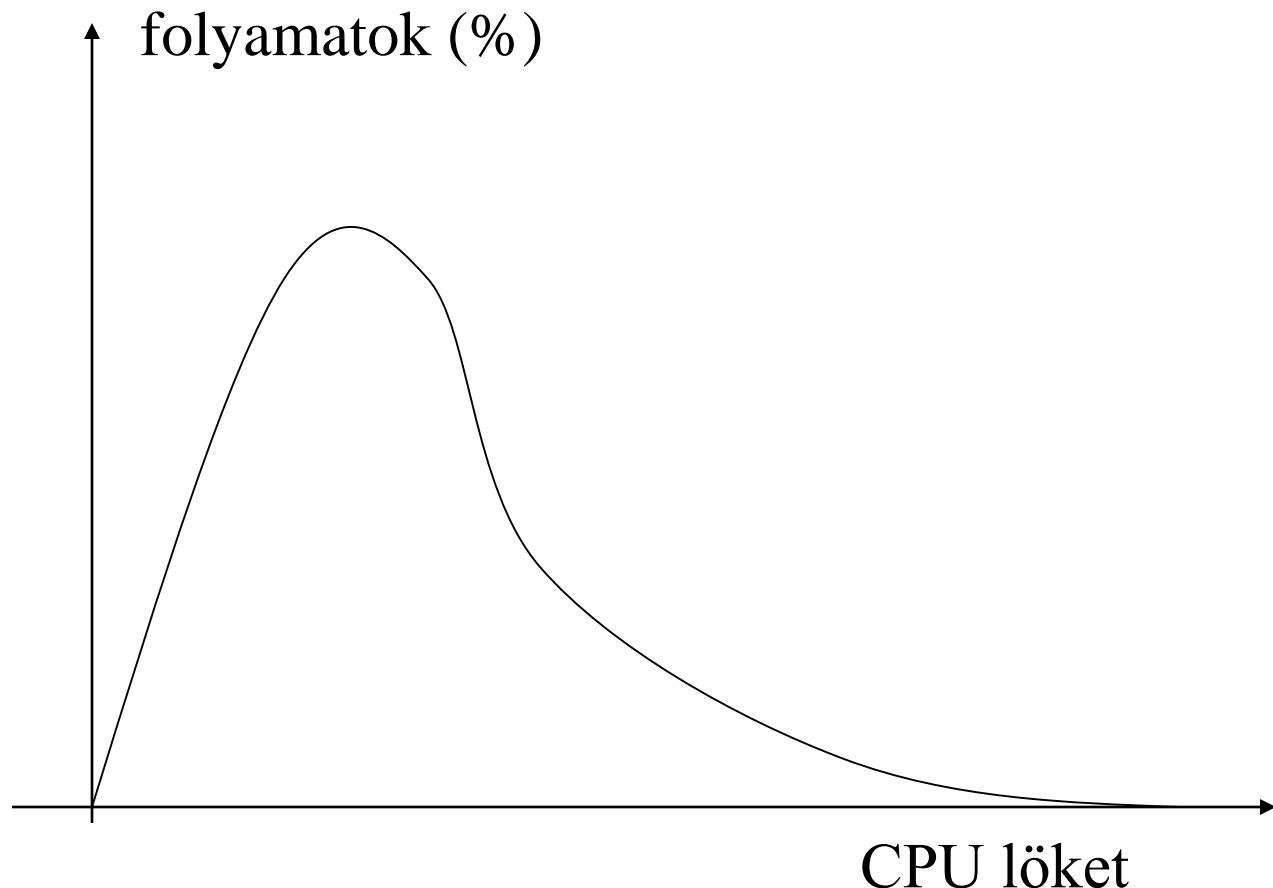
(b) I/O-korlátos folyamat



A CPU löket és az IO löket 2.



A CPU löket eloszlása

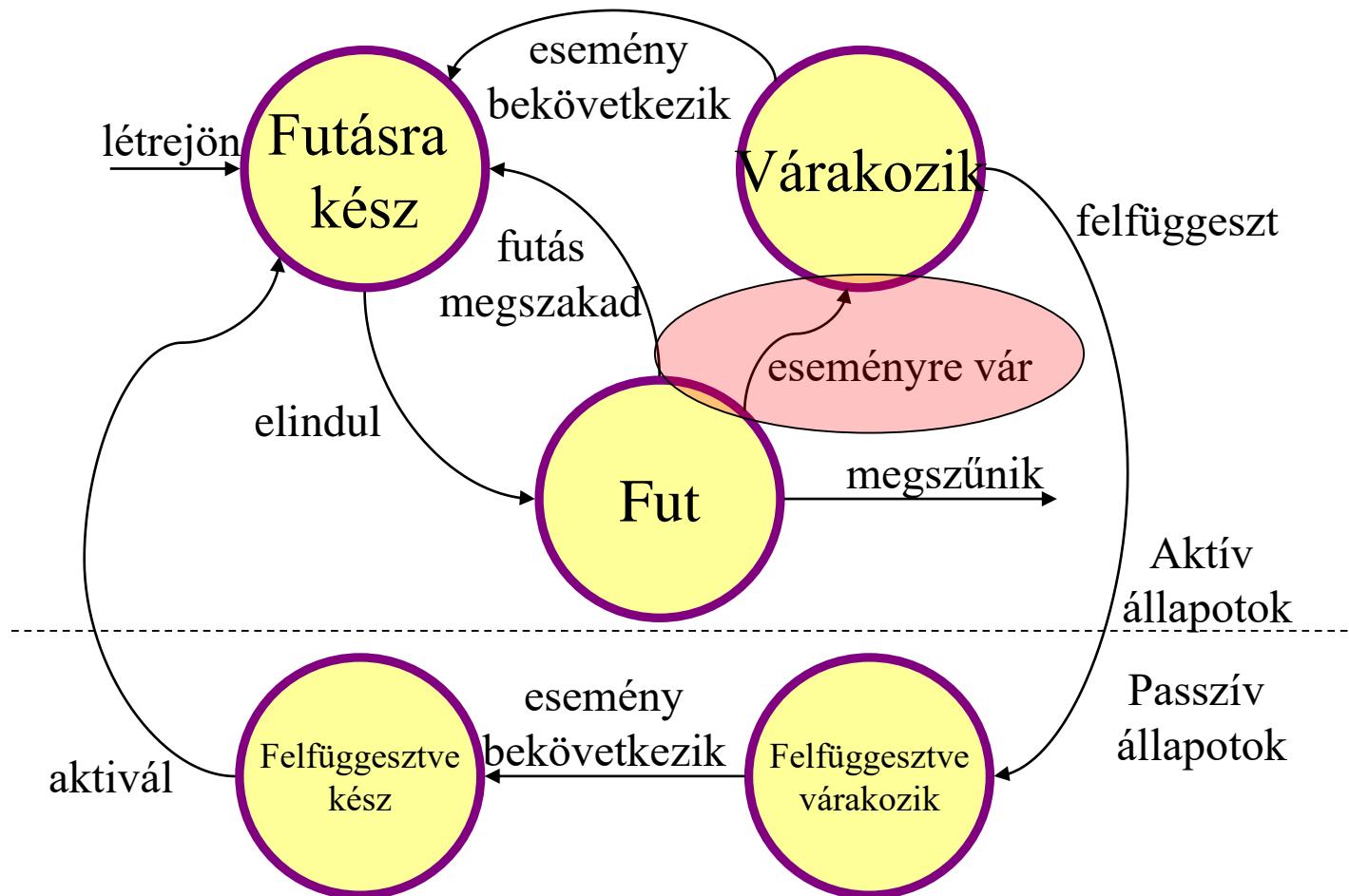


Hol történik ütemezés?

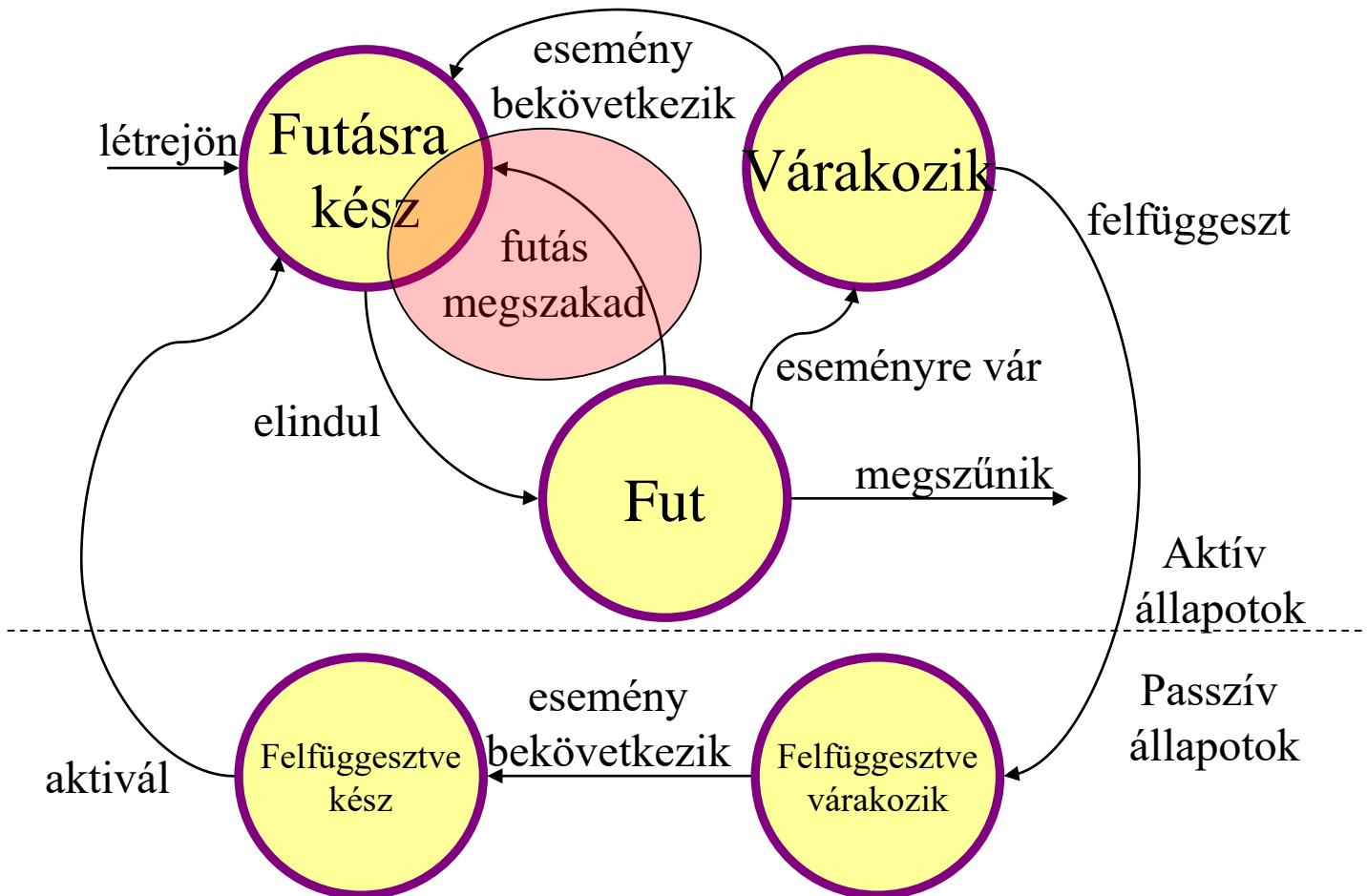
Ütemezés a következő állapotátmeneteknél következik be:

1. A futó folyamat várakozni kényszerül
2. A futó folyamat lemond a CPU-ról vagy elveszik tőle
3. A folyamat felébred, futásra késszé válik
4. A futó folyamat befejeződik

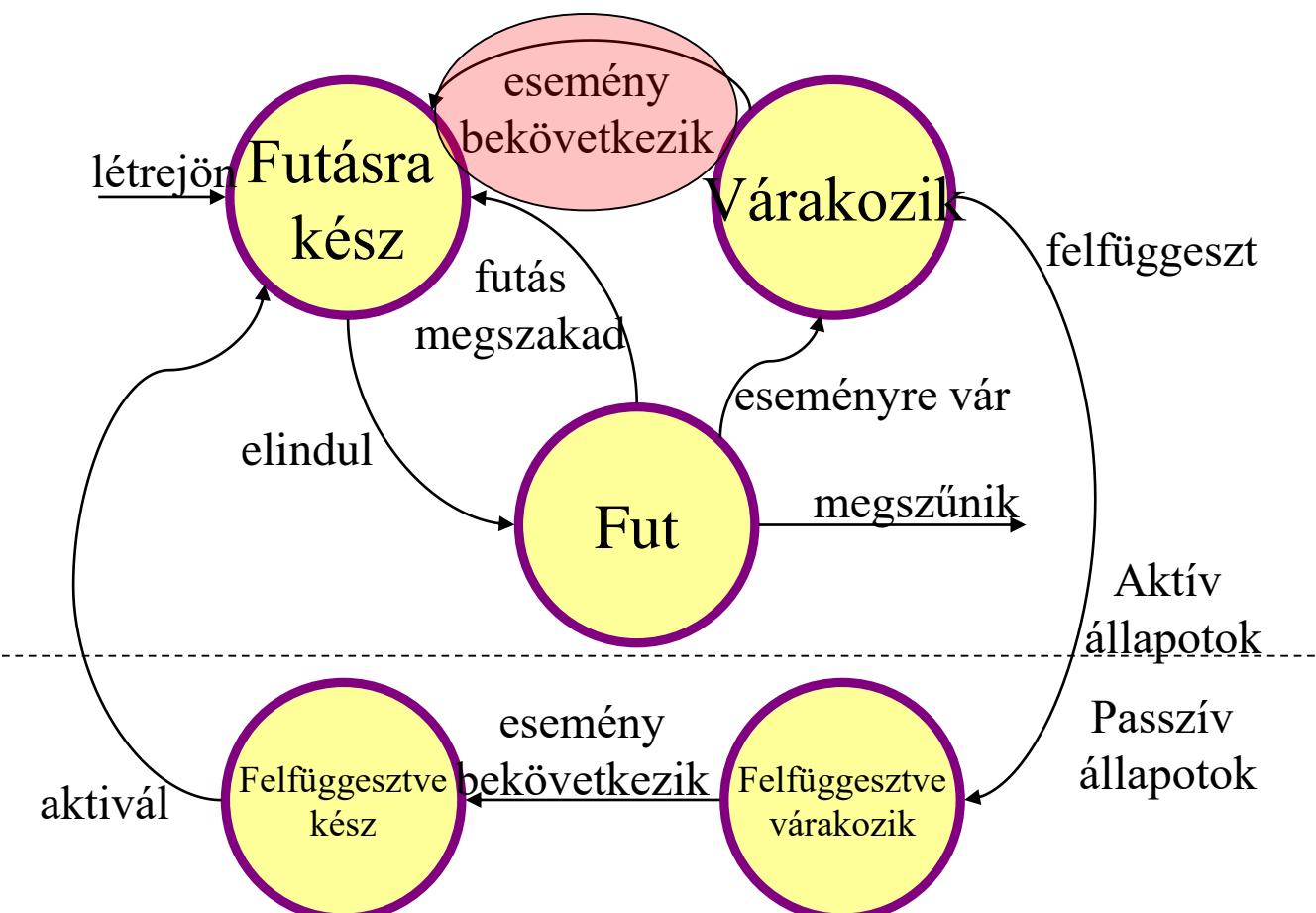
1. A futó folyamat várakozni kényszerül



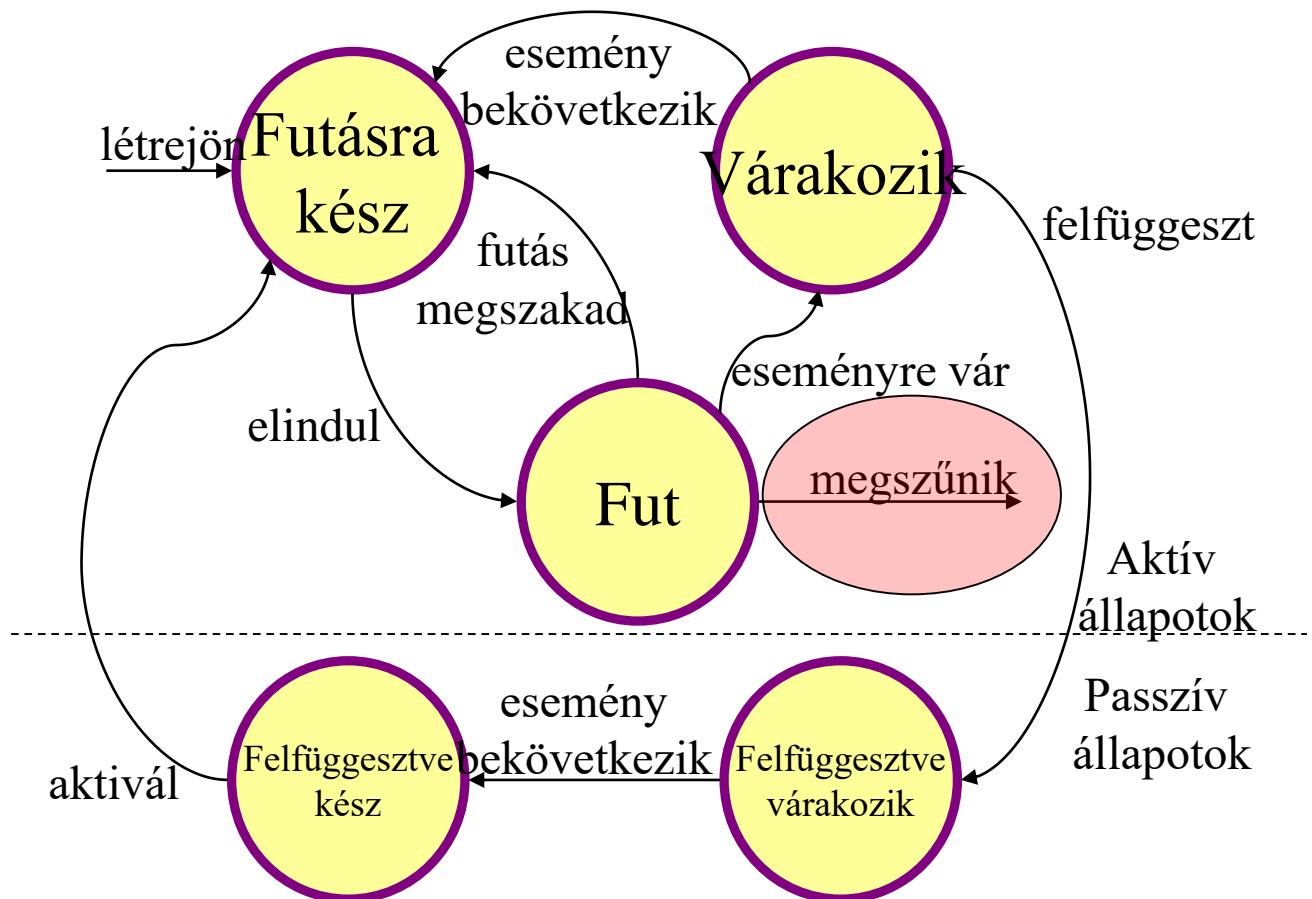
2. A futó folyamat lemond a CPU-ról, vagy elveszik tőle



3. A folyamat felébred, futásra késszé válik



4. A futó folyamat befejeződik



Ütemezés és környezetváltás

Ütemezés tehát a következő állapotátmeneteknél következik be:

1. a futó folyamat várakozni kényszerül
2. futó folyamat lemond a CPU-ról vagy elveszik tőle
3. folyamat felébred, futásra késszé válik
4. a futó folyamat befejeződik

Az 1. és 4. esetben *mindig* van környezetváltás (hiszen a futó folyamat nem folytatja a működését).

A 2. és 3. esetben *nem mindig* van környezetváltás.

Ütemezés típusai

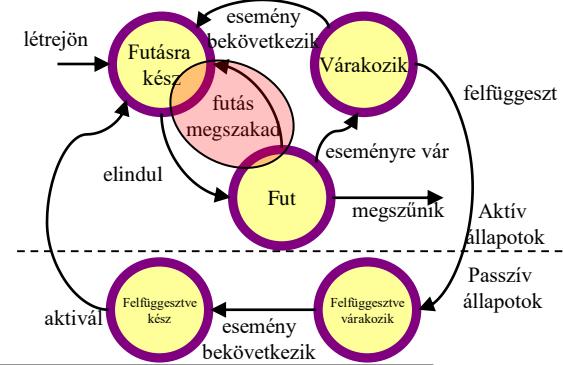
Az ütemezés típusa

Nem preemptív, ha egy folyamattól, miután megkapta a CPU-t, nem lehet azt elvenni.

- a folyamat csak az általa kiadott utasítások hatására vált állapotot.
 - erőforrásra, eseményre várakozás
 - befejeződés vagy
 - a CPU-ról önként lemondás

preemptív, ha az OS elveheti a futás jogát egy folyamattól

- futásra kész állapotba teszi a futó folyamatot és
- egy másik (futásra kész) folyamatot indít el.
- Pl. időosztásos, valósidejű OS-ek



Az ütemezési algoritmusok teljesítményének mérése I.

Az ütemezési algoritmusok teljesítményének mérésére a következő paramétereket szokták használni:

CPU kihasználtság (CPU utilization)

- A CPU az idejének hány százalékát használja a folyamatok utasításainak végrehajtására. Kihasználtságot csökkenti: CPU henyél (idle), rendszer-adminisztrációra fordított idő (rezsi) sok.

Átbocsátó képesség (throughput)

- Az OS időegységenként hány munkát futtat le.

Az ütemezési algoritmusok teljesítményének mérése II.

További paraméterek a teljesítmény mérésére:

Körülfordulási idő (turnaround time)

- Egy munka a rendszerbe helyezéstől számítva mennyi idő alatt fejeződik be.

Várakozási idő (waiting time)

- Egy munka (vagy folyamat) mennyi időt tölt várakozással (futásra kész állapot, várakozó állapot, felfüggesztett állapotok, (long-term) előzetes várakozás).

Válaszidő (response time)

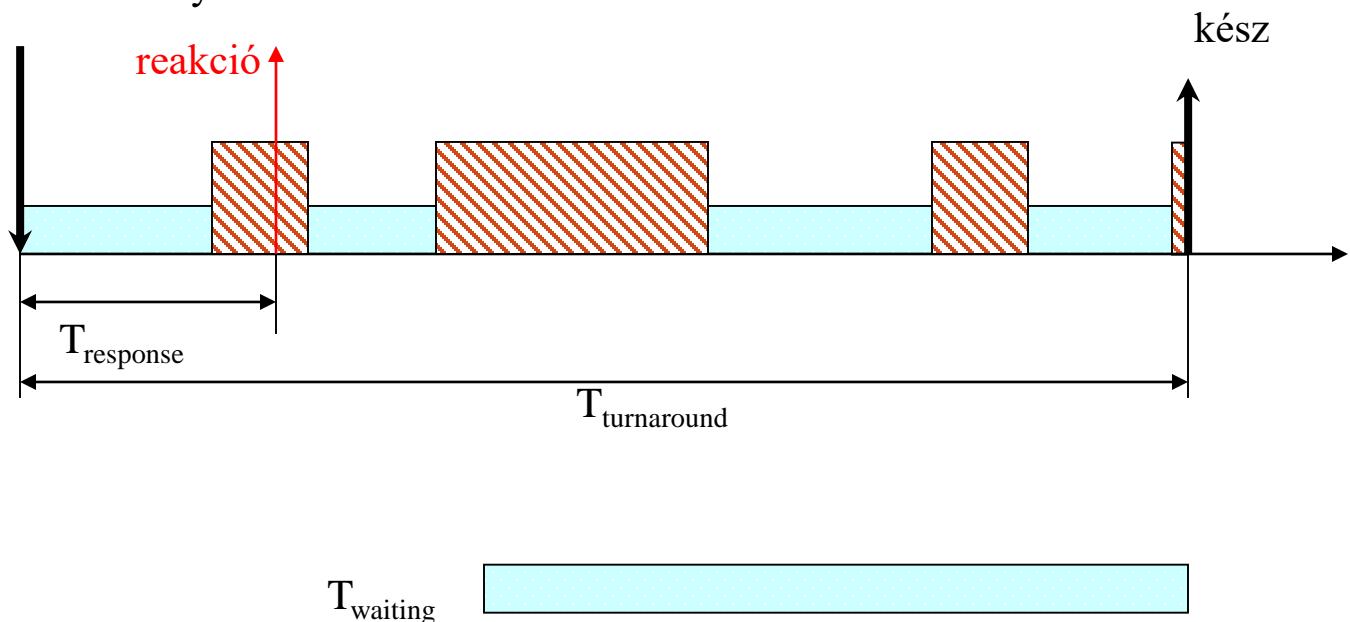
- Időosztásos (interaktív) rendszereknél fontos. Az OS reakció ideje, a kezelői parancs után a rendszer első látható reakciójáig eltelt idő.

Teljesítménymérés

fut 

vár 

rendszerbe helyezés



Az ütemezési algoritmusokkal szembeni követelmények 1.

valamely (előbbi paraméterekből képzett) célfüggvény szerint legyen **optimális**

legyen korrekt, kezeljen minden folyamatot (vagy bizonyos típusú folyamatokat) azonos módon (**igazságos**).

biztosítson egyes folyamatoknál **prioritást**

kerülje el a folyamatok **kiéheztetését**

legyen **megjósolható** viselkedésű (meg lehessen becsülni a várható maximális körülfordulási időt)

minimalizálja a **rezsi időt** (gyakran kis többlet adminisztrációval jobb általános rendszerteljesítmény érhető el)

Az ütemezési algoritmusokkal szembeni követelmények 2.

részeseítse előnyben

- a kihasználatlan erőforrásokat igénylő folyamatokat
- a fontos erőforrásokat foglaló folyamatokat

növekvő terhelés esetén a rendszer teljesítménye "elegánsan", **fokozatosan romoljon le** (graceful degradation), ne omoljon hirtelen össze.

Sok cél egymásnak ellentmond, így azok együttesen nem teljesíthetők.

Fontos megfogalmazni egy rendszer tervezése folyamán azokat a célokat, amelyek kiemelt fontosságúak.

Ütemezési algoritmusok

Egyszerű algoritmusok

Prioritásos algoritmusok

Többszintű algoritmusok

Többprocesszoros ütemezés

Egyszerű ütemezési algoritmusok

Legrégebben várakozó (First Come First Served, FCFS)

Körforgó (Round-Robin, RR)

Legrégebben várakozó

First Come First Served, FCFS

- A futásra kész folyamatok a várakozási sor végére kerülnek, az ütemező a sor elején álló folyamatot kezdi futtatni.
- Nem preemptív.
- Egyszerűen megvalósítható
- Konvoj hatás (egy hosszú CPU löketű folyamat feltartja a mögötte várakozókat).

Példa: FCFS

P_i	C_i (ms)
P_1	24
P_2	3
P_3	3

Végrehajtási sorrend: P_1, P_2, P_3

Átlagos várakozási idő:

Átlagos körülfordulási idő:

Átbocsátó képesség:

Végrehajtási sorrend: P_2, P_3, P_1

Átlagos várakozási idő:

Átlagos körülfordulási idő:

Átbocsátó képesség:



Megoldás: FCFS

Konvoj effektus!

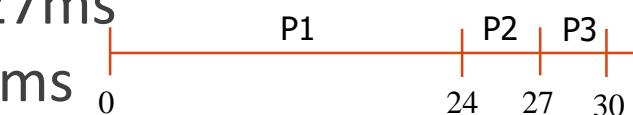
P_i	C_i (ms)
P_1	24
P_2	3
P_3	3

Végrehajtási sorrend: P_1, P_2, P_3

Átlagos várakozási idő: $(24+27)/3=17\text{ms}$

Átlagos körülfordulási idő: $(24+27+30)/3=27\text{ms}$

Átbocsátó képesség: $3\text{job}/30\text{ms}=1/10 \text{ job/ms}$



Végrehajtási sorrend: P_2, P_3, P_1

Átlagos várakozási idő: $(3+6)/3=3\text{ms}$

Átlagos körülfordulási idő: $(3+6+30)/3=13\text{ms}$

Átbocsátó képesség: $3\text{job}/30\text{ms}=1/10 \text{ job/ms}$



Körforgó

Round-Robin, RR

- Preemptív algoritmus
- Az időosztásos rendszerek valamennyi ütemezési algoritmusa ezen alapul
- Folyamatok időszeletet kapnak (time slice).
 - Ha a CPU löket nagyobb mint az időszelet, akkor az időszelet végén az ütemező elveszi a CPU-t, a folyamat futásra kész lesz és beáll a várakozó sor végére.
 - Ha a CPU löket rövidebb, akkor a löket végén a folyamatokat újraütemezzük.

Körforgó ütemezés időszelete

Időszelet meghatározása nehéz.

- Nagy időszelet: FCFS algoritmushoz hasonló lesz.
- Kis időszelet: folyamatok a CPU-t egyenlő mértékben használják, viszont a sok környezetváltás a teljesítményt rontja.

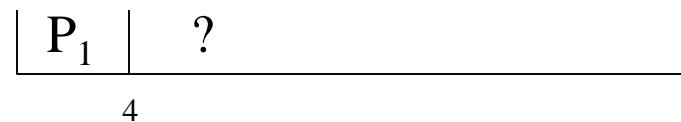
Ökölszabály: a CPU löketek kb. 80% legyen rövidebb az időszeletnél.

Általában 10-100ms

Példa: *Round Robin*

P_i	C_i (ms)
P_1	20
P_2	3
P_3	6

$$T_{\text{slice}} = 4 \text{ ms}$$

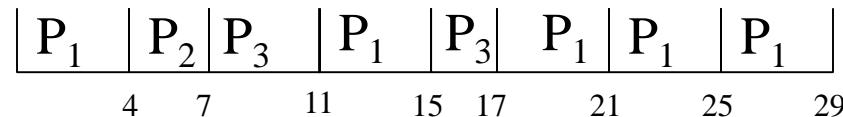


4

Megoldás: *Round Robin*

P_i	C_i (ms)
P_1	20
P_2	3
P_3	6

$$T_{\text{slice}} = 4 \text{ ms}$$



Példa: Round Robin

Az időszelet (T_{slice}) hatása a környezetváltás gyakoriságára

$$C_i = 10\text{ms}$$

T_{slice}	Környezetváltások száma
12	0
6	1
1	9

Prioritásos ütemező algoritmusok

A futásra kész folyamatokhoz egy prioritást (rendszerint egy egész számot) rendelünk.

A legnagyobb prioritású folyamat lesz a következő futtatandó folyamat.

Name	PriorityBoostEnabled	PriorityClass	F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
AcroRd32	True	Normal	4	S	0	1	0	0	80	0	-	68	poll_s	?	00:00:02	init
AcroRd32	True	Normal	1	S	0	2	0	0	80	0	-	0	kthrea	?	00:00:00	kthreadd
acrotray	True	Normal	1	S	0	3	2	0	80	0	-	0	ksofti	?	00:00:00	ksoftirqd/0
AGMService			5	S	0	4	2	0	-40	-	-	0	watchd	?	00:00:00	watchdog/0
AGSService			1	S	0	5	2	0	80	0	-	0	worker	?	00:00:00	events/0
ApplicationFrameHost	True	Normal	1	S	0	6	2	0	80	0	-	0	worker	?	00:00:00	khelper
armsvc			1	S	0	9	2	0	80	0	-	0	async_	?	00:00:00	async/mgr
bash			1	S	0	89	2	0	80	0	-	0	bdi_sy	?	00:00:00	sync_supers
browser_assistant	True	Normal	1	S	0	91	2	0	80	0	-	0	bdi_fo	?	00:00:00	bdi-default
browser_assistant	True	Normal	1	S	0	93	2	0	80	0	-	0	worker	?	00:00:30	kblockd/0
BtwRSupportService			1	S	0	98	2	0	80	0	-	0	worker	?	00:00:00	ata/0
chrome	True	Normal	1	S	0	99	2	0	80	0	-	0	worker	?	00:00:00	ata_aux
chrome	True	Idle	1	S	0	101	2	0	80	0	-	0	serio_	?	00:00:00	kseriod
chrome	True	Idle	5	S	0	121	2	0	80	0	-	0	worker	?	00:00:00	rpciod/0
chrome	True	Idle	1	S	0	152	2	0	80	0	-	0	watchd	?	00:00:00	khungtaskd
chrome	True	Idle	1	S	0	153	2	0	80	0	-	0	kswapd	?	00:01:16	kswapd0
chrome	True	Normal	1	S	0	154	2	0	80	0	-	0	worker	?	00:00:00	aio/0
chrom	True	Idle	1	S	0	155	2	0	80	0	-	0	worker	?	00:00:00	nfsiod
chrom	True	Normal	1	S	0	156	2	0	75	-5	-	0	slow_w	?	00:00:00	kslowd000
chrom	True	Idle	1	S	0	157	2	0	75	-5	-	0	slow_w	?	00:00:00	kslowd001
chrom	True	Idle	1	S	0	161	2	0	80	0	-	0	worker	?	00:00:00	xfs_mru_cache
chrom	True	Idle	1	S	0	162	2	0	80	0	-	0	worker	?	00:00:00	xfslogd/0
chrom	True	Idle	1	S	0	163	2	0	80	0	-	0	worker	?	00:00:00	xfslogad/0
chrom	True	Idle	1	S	0	164	2	0	80	0	-	0	worker	?	00:00:00	xfsconvertd/0
chrom	True	Idle	1	S	0	165	2	0	80	0	-	0	worker	?	00:00:00	crypto/0
chrom	True	Idle	1	S	0	264	2	0	80	0	-	0	worker	?	00:00:00	scsi_tgtd/0
chrom	True	Normal	1	S	0	276	2	0	80	0	-	0	scsi_e	?	00:00:00	scsi_eh_0
chrom	True	Idle	1	S	0	279	2	0	80	0	-	0	scsi_e	?	00:00:00	scsi_eh_1
chrom	True	Idle	5	D	0	308	2	0	58	-	-	0	msleep	?	00:00:00	btn_t

IDLE
BELOW_NORMAL
NORMAL
ABOVE_NORMAL
HIGH
REALTIME

Prioritás

Meghatározása

- belső
 - az OS határozza meg
- külső
 - az OS-en kívüli tényező (operátor, a folyamat saját kérése, stb.) határozza meg

Futás során

- statikus (végig azonos)
- dinamikus (az OS változtathatja)

Példa: Statikus prioritás

$|P_2|$?

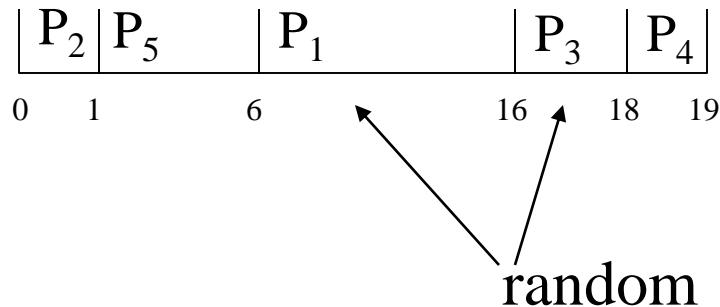
0 1

Folyamat	C_i (ms)	Prioritás
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

Átlagos várakozási idő: ?

Megoldás: Statikus prioritás

Folyamat	C _i (ms)	Prioritás
P ₁	10	3
P ₂	1	1
P ₃	2	3
P ₄	1	4
P ₅	5	2



$$\text{Átlagos várakozási idő: } (6+0+16+18+1)/5=8.2$$

Prioritás meghatározása

Prioritást sokszor a löketidő alapján határozzák meg.

A löketidő szükséglet meghatározása:

- a folyamat (felhasználó) bevallása alapján (a „hazugságot” az OS később bünteti)
- előző viselkedés alapján (a korábbi löketidők alapján becslés)

A kiéheztetés és elkerülése

Kiéheztetés:

- A folyamat sokáig (esetleg soha) nem jut processzorhoz

Prioritásos algoritmusoknál kiéheztetés felléphet

Ennek kivédése a folyamatok öregítése (*aging*):

- a régóta várakozó folyamatok prioritását növeljük

Egy éhes folyamat

Klasszikus példa: MIT, 1973

Folyamat: 1967-ből



IBM 7094

Dinamikus prioritásos algoritmusok

Legrövidebb (löket)idejű

- *Shortest Job First, SJF*

Legrövidebb hátralévő idejű

- *Shortest Remaining Time First, SRTF*

Legjobb válaszarány

- *Highest Response Ratio*

Legrövidebb (löket)idejű

Shortest Job First, SJF

Nem preemptív algoritmus, a futásra kész folyamatok közül a legrövidebb löketidejűt indítja.

Nincs konvoj hatás, optimális körülfordulási idő, optimális várakozási idő.

Alkalmazása:

- Hosszú távú ütemezés
- Rövid távú ütemezés (RT rendszerek)

Legrövidebb hátralévő idejű

Shortest Remaining Time First, SRTF

Az SJF *preemptív* változata

Ha egy új folyamat válik futásra késsé

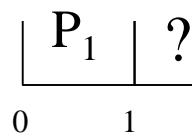
- akkor az ütemező újra megvizsgálja a futásra kész folyamatok, illetve az éppen futó folyamatot hátralévő löketidejét
- és a legrövidebbet indítja tovább.

A folyamat megszakítása és egy másik elindítása környezetváltozást igényel, ezt az időt is figyelembe kell vennünk.

Példa: SRTF

P_i	T_i (ms)	C_i (ms)
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

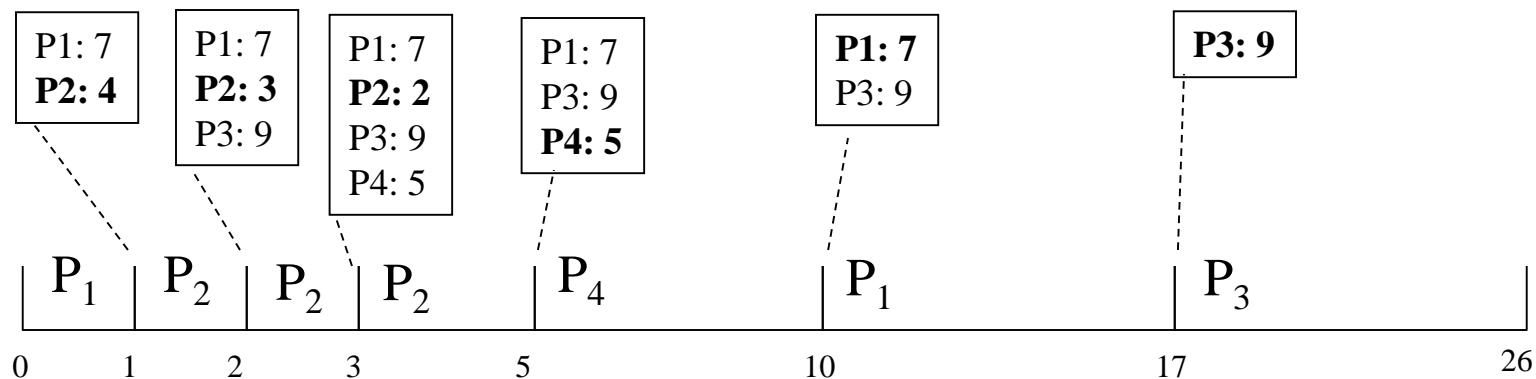
Ütemezés?



Átlagos várakozási idő: ?

Példa megoldás: SRTF

P_i	T_i (ms)	C_i (ms)
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5



Átlagos várakozási idő: $(9+0+15+2)/4=6.5\text{ms}$

Legjobb válaszarány

Highest Response Ratio

Az SJF algoritmus változata, ahol a várakozó folyamatok öregednek. A kiválasztás (a löketidő helyett) a

$$\frac{\text{löketidő} + k \cdot \text{várakozási_idő}}{\text{löketidő}}$$

válaszarány szerint megy végbe (a legnagyobbat választjuk ki), ahol k egy jól megválasztott konstans.

Többszintű algoritmusok

Futásra kész folyamatokat több külön sorban várakoztatják.

A sorokhoz prioritás van rendelve.

Egy kisebb prioritású sorból csak akkor indulhat el egy folyamat, ha a nagyobb prioritású sorok üresek.

Az egyes sorokon belül különböző kiválasztási algoritmusok működhetnek.

Példák:

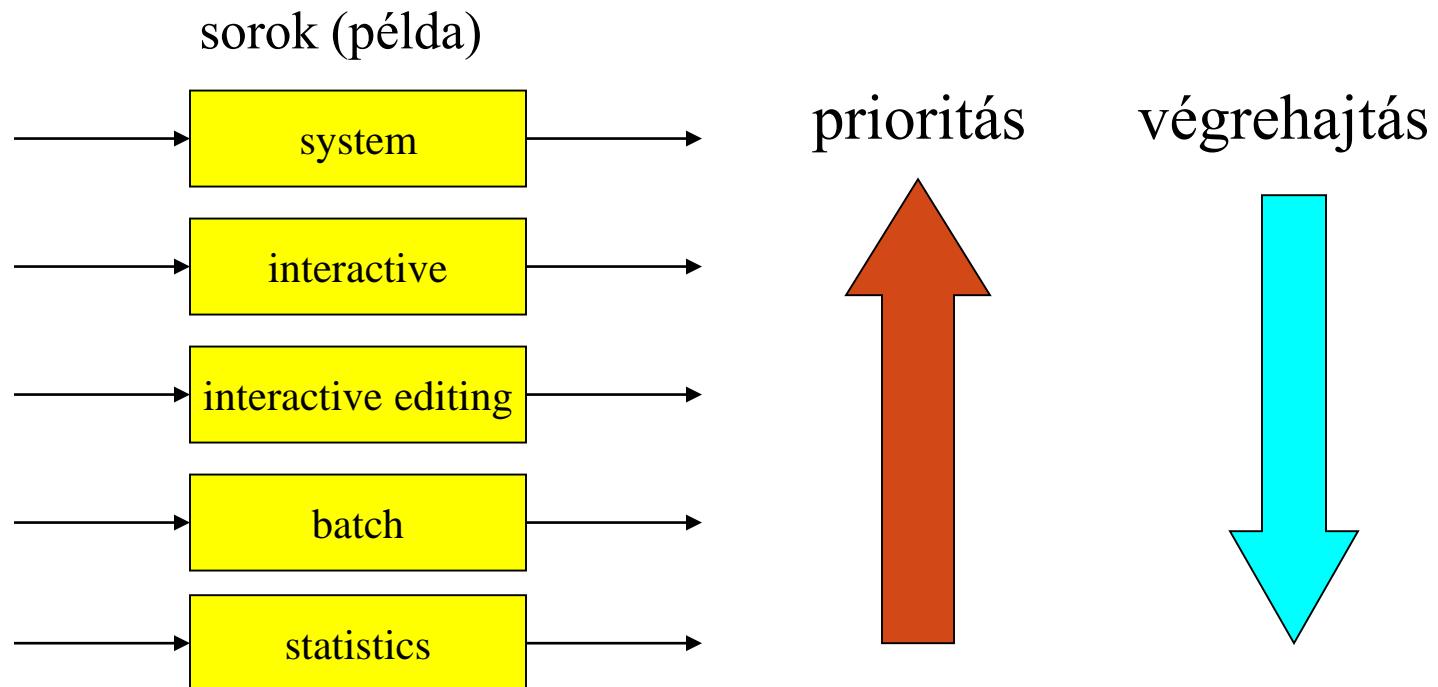
- Statikus többszintű sorok
- Visszacsatolt többszintű sorok

Statikus többszintű sorok

Static Multilevel Queue, SMQ

A folyamatot elindulásakor valamelyen kritérium alapján besorolunk egy várakozó sorba.

A folyamat élete során végig ugyanabban a sorban marad.



Statikus többszintű sorok

Egy lehetséges példa a prioritások besorolására:

- rendszer folyamatok (magas prioritás, közvetlen hatással vannak a rendszer működésére)
- interaktív folyamatok (biztosítani kell a felhasználó számára az elfogadható válaszidőt)
- interaktív szövegszerkesztők (kevésbé kritikusak)
- kötegelt feldolgozás (általában akkor futnak, ha "van idő")
- rendszerstatisztikákat gyűjtő folyamatok (alacsony prioritás, nincsenek közvetlen hatással a rendszer működésére)

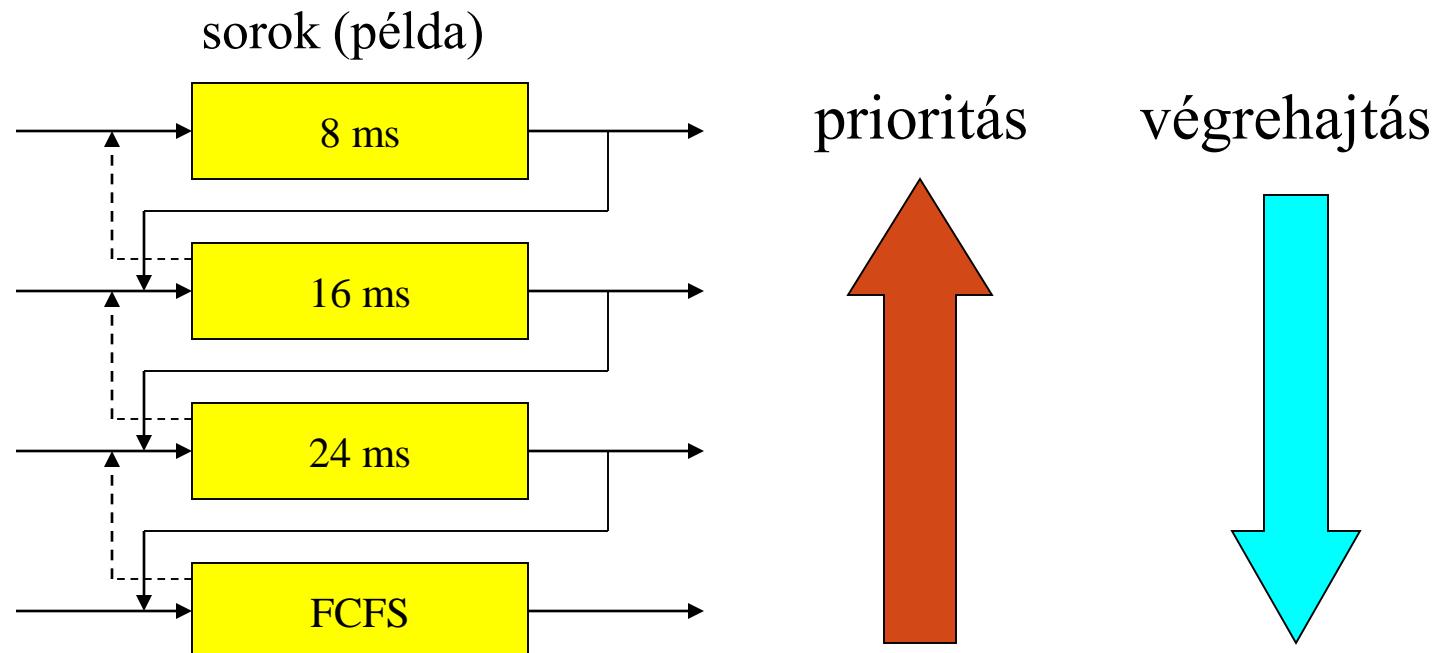
Visszacsatolt többszintű sorok

Multilevel Feedback Queues, MFQ

A sorokhoz egy időszelet tartozik:

- minél nagyobb a prioritás, annál kisebb az időszelet

A folyamatok futásuk során átkerülhetnek másik sorokba



Visszacsatolt többszintű sorok

A folyamatok sor- és prioritás-váltása

Nagyobb prioritásúból kisebb prioritású sorba:

- Amennyiben nem elég az adott időszak, a folyamat egy kisebb prioritású sorba kerül át.

Kisebb prioritásúból nagyobb prioritású sorba:

- A folyamat átlagos löketidejének változása (csökkenése) esetén
- A régóta várakozó folyamat öregedése miatt

Többprocesszoros ütemezés

Napjainkban egyre jobban terjednek a többprocesszoros - szorosan csatolt - rendszerek, ahol felmerül az igény, hogy a futásra kész folyamatok a rendszer bármely processzorán elindulhassanak.

Heterogén rendszerek esetében egy folyamat csak bizonyos processzorokon futhat.

Homogén rendszerekben a futásra kész folyamatokat közös sorokban tárolja.

Homogén többprocesszoros rendszerek

Homogén rendszerekben a futásra kész folyamatokat közös sorokban tárolja.

- Szimmetrikus multiprocesszoros rendszer

Minden CPU saját ütemezőt futtat, amely a közös sorokból választ. A sorok osztott használatához a kölcsönös kizárást biztosítani kell!

- Aszimmetrikus multiprocesszoros rendszer

Az ütemező egy dedikált CPU-n fut, ez osztja szét a folyamatokat a szabad CPU-k között.

Operációs rendszerek

6-7. TÁRKEZELÉS

Felhasznált irodalom:

- Kóczy-Kondorosi (szerk.): Operációs rendszerek mérnöki megközelítésben
- Tanenbaum: Modern Operating Systems
- Silberschatz, Galvin, Gagne: Operating System Concepts

Tárkezelés

Bevezetés

A program címeinek kötése

Társzervezési elvek

Többpartíciós rendszerek

Szegmens- és lapszervezés

Bevezetés

A központi tár (*main storage, memory*)

- Szervezése és
- kezelése

az OS tervezését, implementálását és teljesítményét befolyásoló egyik legfontosabb tényező.

A multiprogramozás igénye és a programméretek növekedése a *valós tár* kezelésén túl megkövetelte a *virtuális tár* kezelésének hardver és szoftver technikáit.

A program címeinek kötése

Logikai címtartomány:

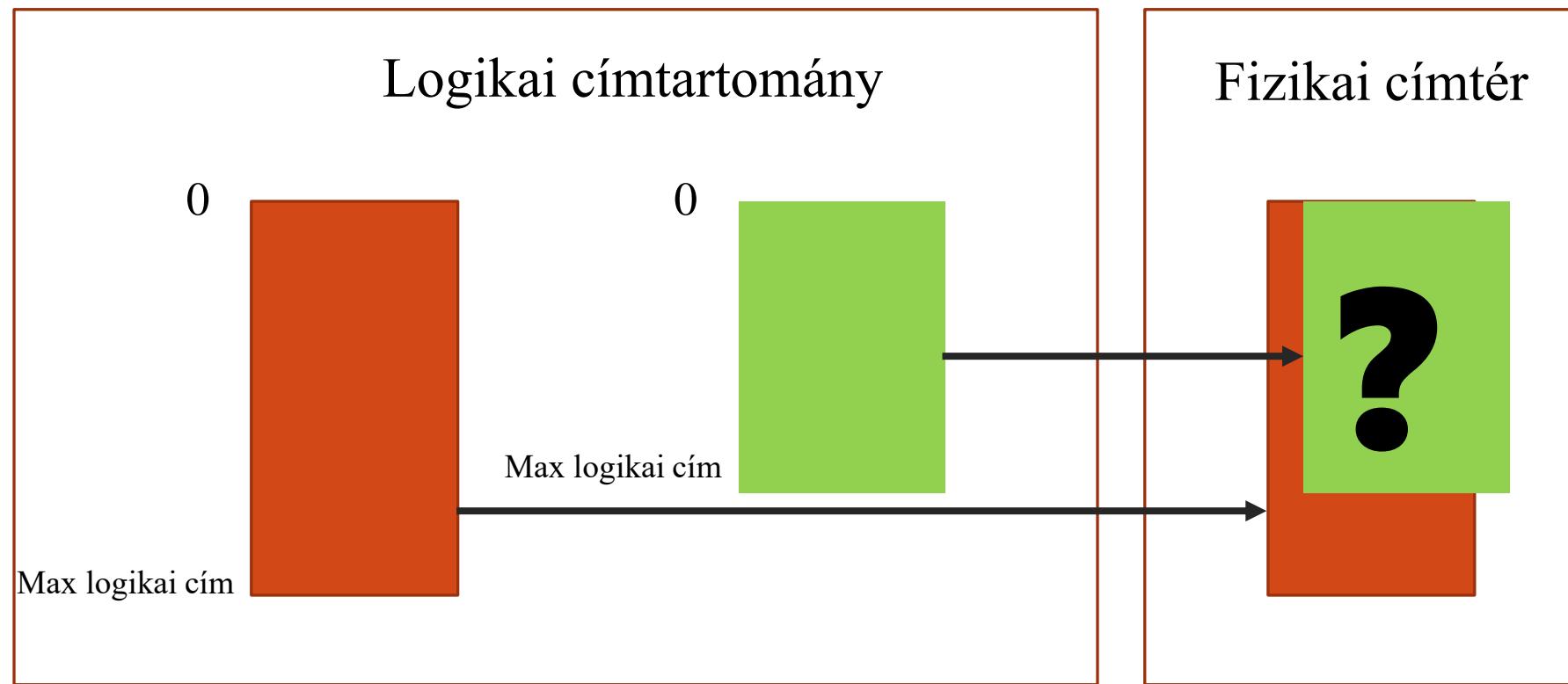
- Folytonos címtartomány
- 0-tól kezdődik
- Lineárisan nő
- a maximális értéig.

Fizikai címtartomány: a gyakorlatban

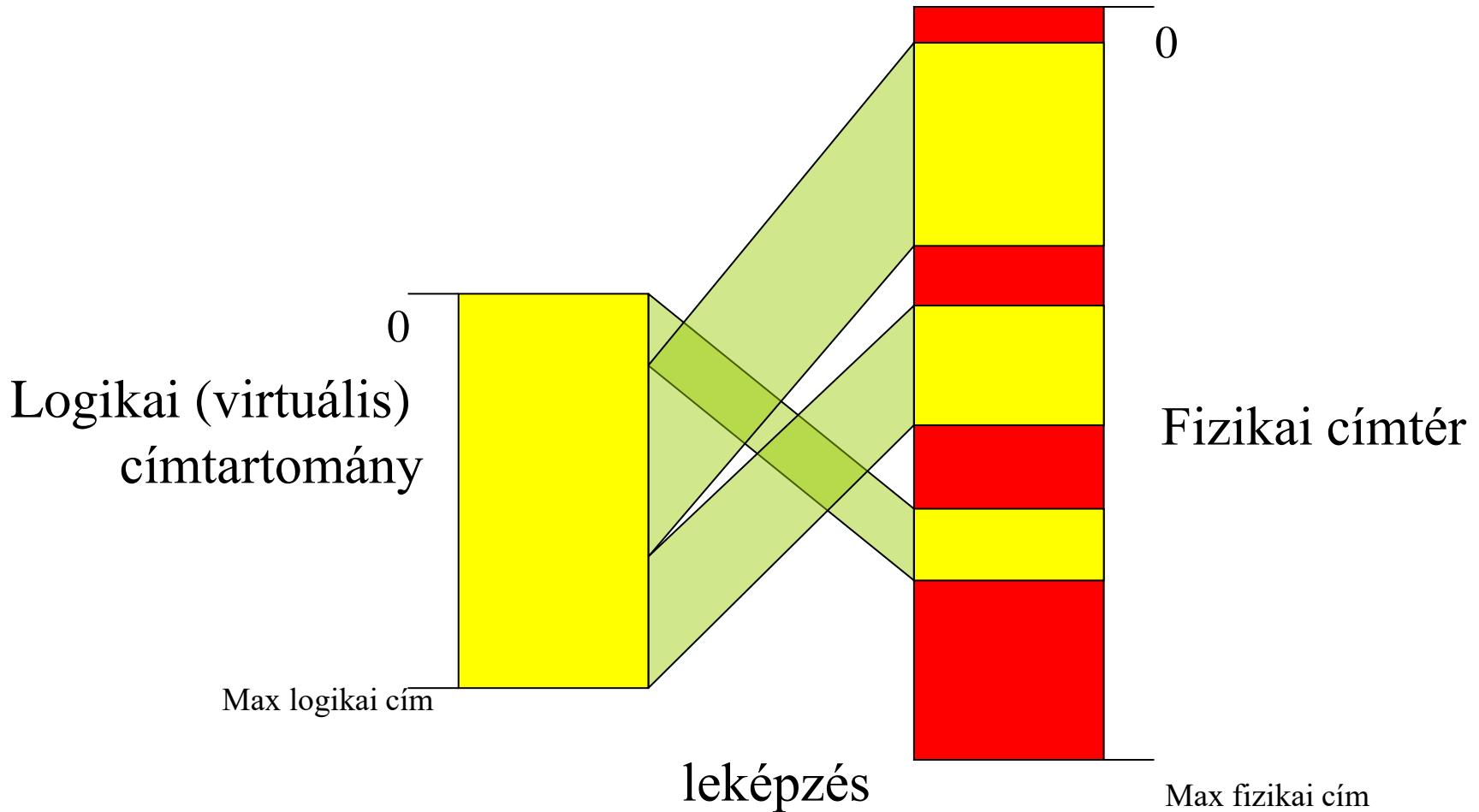
- nem 0 fizikai címtől kezdve történik a programok végrehajtása,
- sokszor nem folytonos memóriaterület áll rendelkezésre

A két tartomány között a megfeleltetés a leképzés (mapping).

Logikai – fizikai címek



A címleképzés



A címek kötésének lehetőségei

statikus

Fordítás közben (compile time)

- A fordítóprogram a program és az adatterület elemeihez abszolút címet rendel. Merev technika, elsősorban ROM-ban lévő programok esetében alkalmazzák.

Szerkesztés közben (link time)

- A függetlenül lefordított modulok saját logikai címtartományt használnak. A linker feladata, hogy az összes modult - egymás mögé - elhelyezze a fizikai tárba, valamint feloldja a modulok kereszthivatkozásait.

Betöltés közben (load time)

- A fordító áthelyezhető kódot generál, ennek a címhivatkozásait a betöltő program az aktuális címkiosztás szerint módosítja.

Futás közben (run time)

- A program csak logikai címeket tartalmaz, speciális hardver elemek határozzák meg a címet az utasítás végrehajtásakor.

dinamikus

A logikai és fizikai címek kapcsolata

Logikai (virtuális) cím:

- az a memóriacím, amit a CPU generál

Fizikai cím:

- a memória valós címe

A logikai és fizikai címek

- megegyeznek: statikus címkötés esetén
- különböznek: dinamikus címkötés esetén

A memória-menedzsment egység (MMU)

Memory-Management Unit, MMU:

- az a hardver egység, amely a virtuális címeket fizikai címmé alakítja

Megjegyzés: a felhasználói programok csakis virtuális címekkel operálnak, soha nem látják a valós fizikai címeket!

A dinamikus logikai-fizikai címleképzés alapvető módszerei

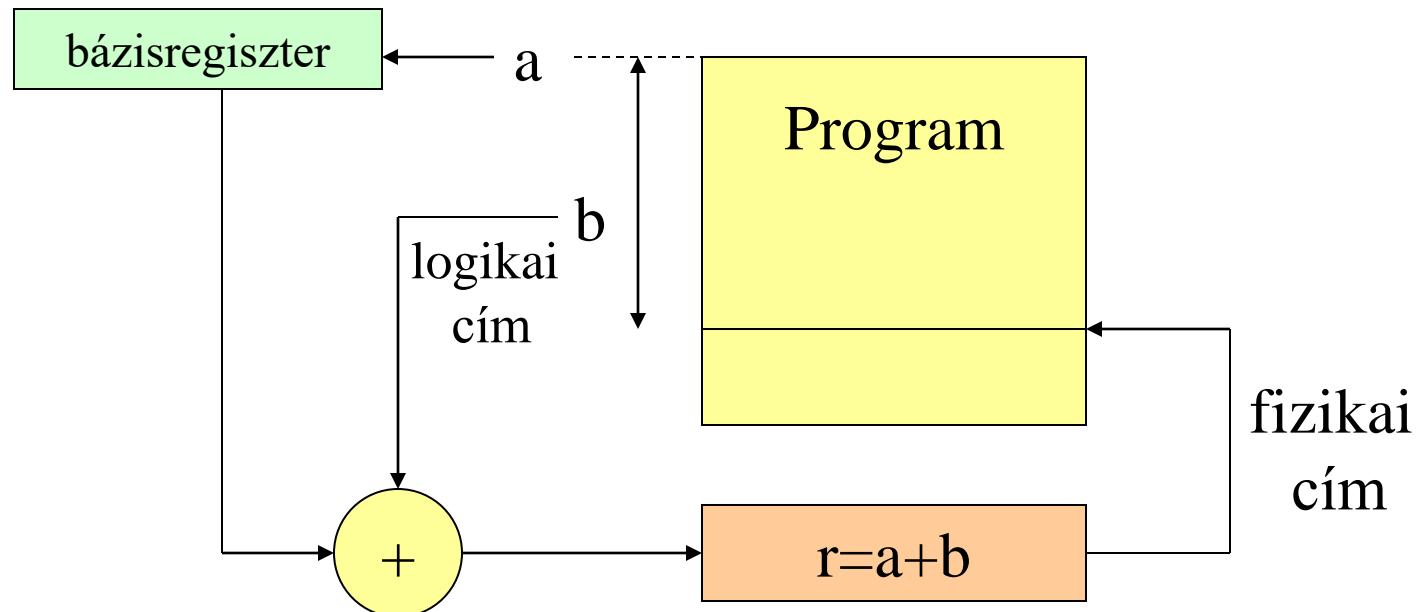
Bázis-relatív címzés

Utasításszámláló-relatív címzés

Bázis-relatív címzés

A program tetszőleges helyre betölthető

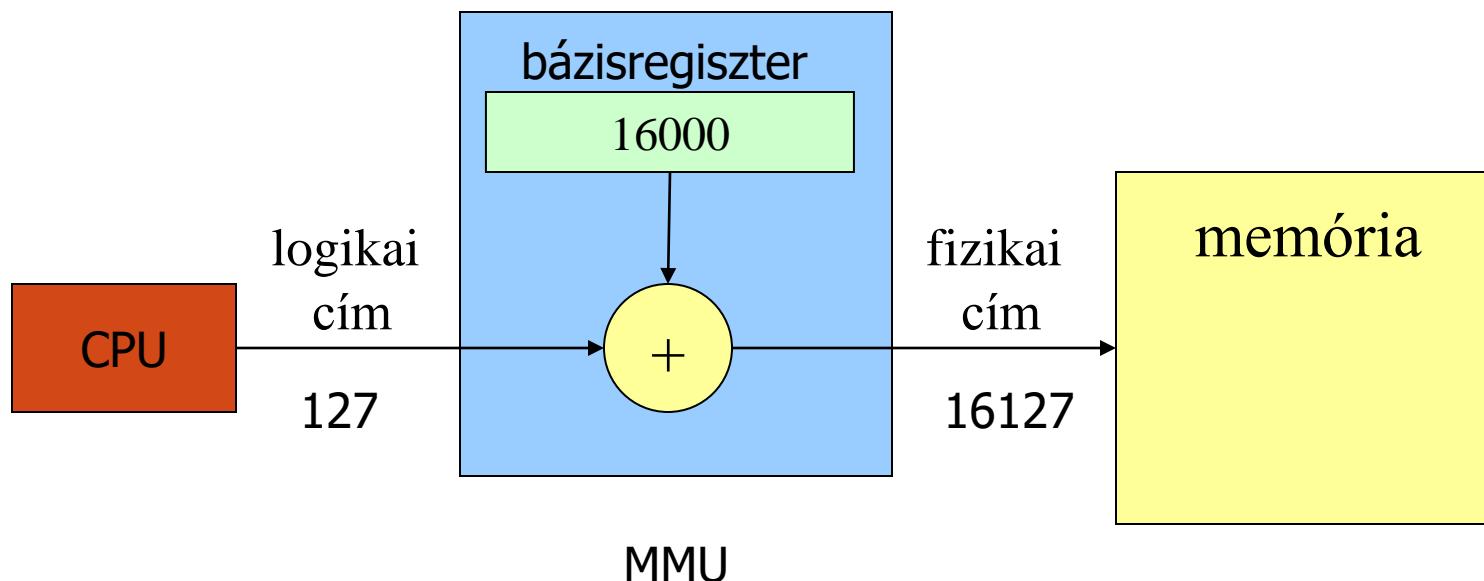
A bázisregisztert a betöltési kezdőcímre állítva a program végrehajtható.



Bázis-relatív címzés példa

A program tetszőleges helyre betölthető

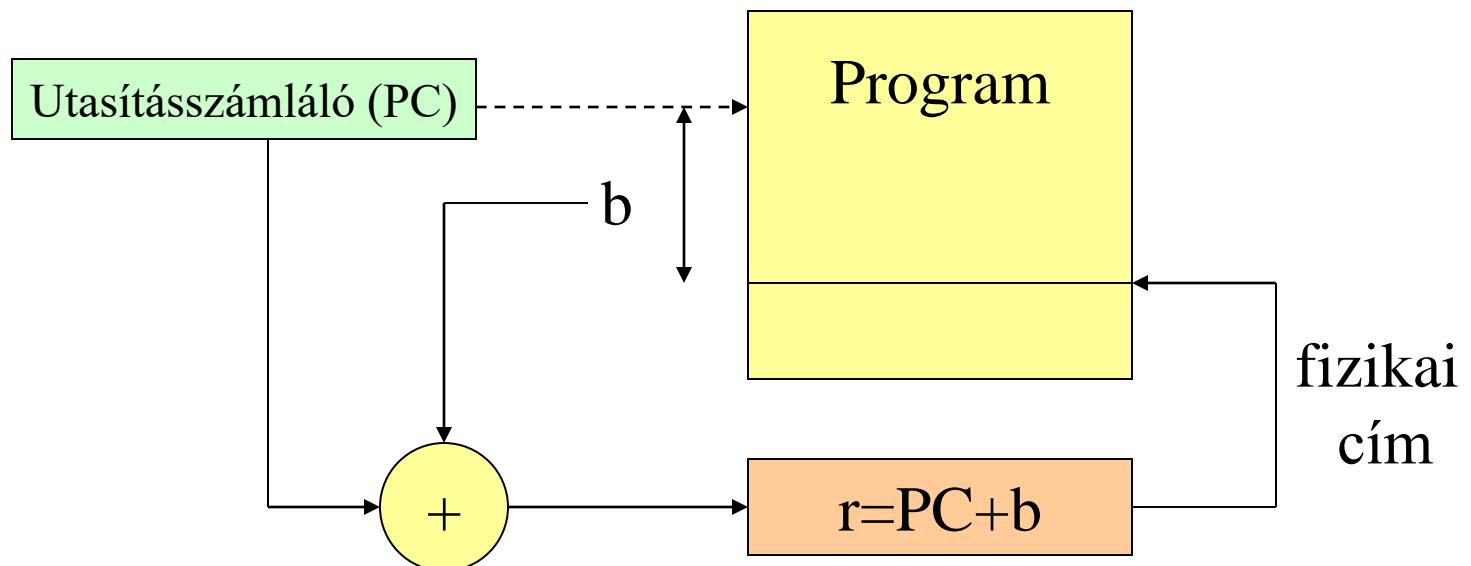
A bázisregisztert a betöltési kezdőcímre állítva a program végrehajtható.



Utasításszámláló relatív címzés

Pozíció-független kód:

- csak pozíció-független virtuális címeket tartalmaz



Programok végrehajtása kisebb tárterületen

Az eddigi módszereknél a fizikai memória mérete behatárolta a futtatható kód méretét (hiszen az egész programnak és a kapcsolódó adatoknak egyszerre a memóriában kellett lennie)

Ötlet: nem kell az egész programnak a memóriában lennie!

Késleltetett betöltés

Futás közben nincs az egész program a tárban. Szükség esetén töltődnek be egyes programrészek.

- dinamikus betöltés
 - *dynamic loading*
- dinamikus kapcsolatszerkesztés
 - *dynamic linking*
- átfedő programrészek
 - *overlay*

Dinamikus betöltés (dynamic loading)

A programhoz tartozó egyes eljárások a háttértáron vannak, ha valamelyikre szükség van, azt egy speciális programrészlet ezt betölti.

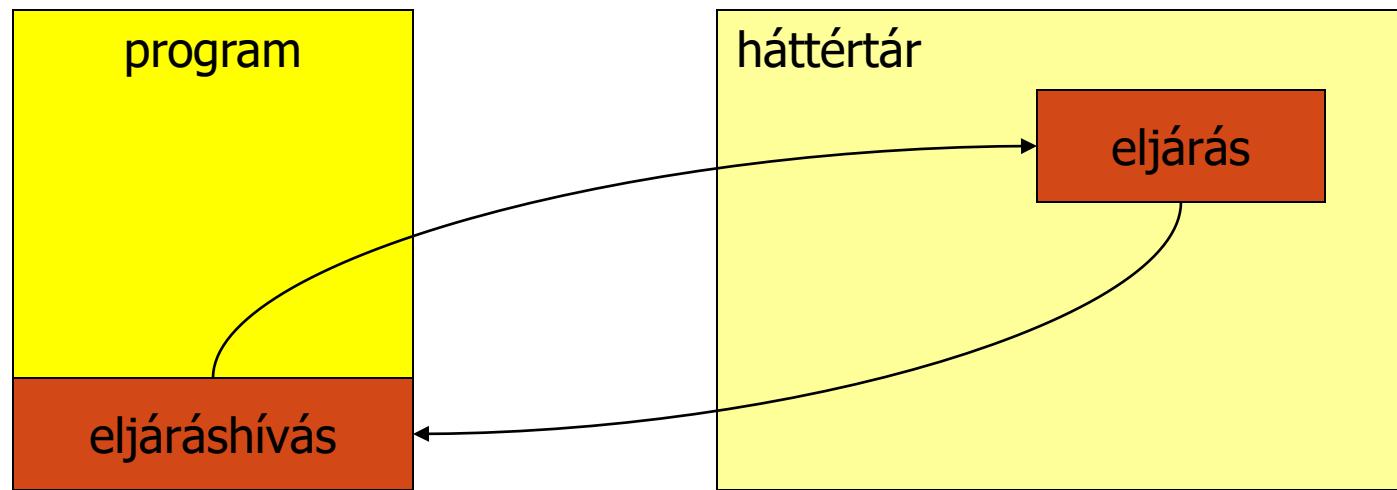
Programozó szervezi meg, az OS nem nyújt támogatást.

Pl. ritkán használt részek, hibakezelés, stb.

Lépései:

1. Hívás
2. Ellenőrzés: memóriában van?
3. Speciális programrész betölti a memóriába, vezérlést átadja

Dinamikus betöltés (dynamic loading)



Dinamikusan betöltött könyvtárak (dynamic linking)

A dinamikus betöltés változata, az OS támogatásával.

A programban használt rendszerkönyvtárak eljárásai helyett csak egy csonk (*stub*) kerül a programba

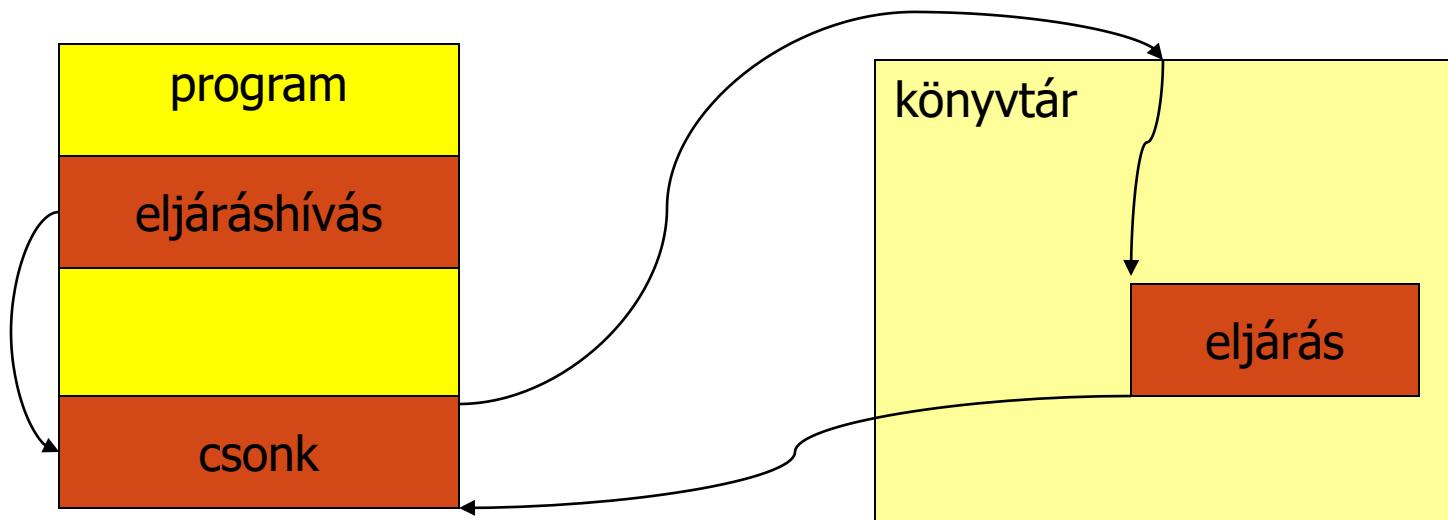
A csonk tartalmaz egy hivatkozást

- a könyvtárra
- és az azon belüli eljárásra.

Az csonk első meghívásakor az OS az eljárást betölti a tárba.

A következő hívások már az eljárást hívják meg időveszteség nélkül.

Dinamikusan betöltött könyvtárak (dynamic linking)



Dinamikusan betöltött könyvtárak tulajdonságai

Előny:

- csökken a tárfelhasználás
- hibás eljárás javításakor nem kell az összes programot újrafordítani

Hátrány:

- Verzió-kontrol nehéz: „dll pokol”

Pl.:

- Windows dll
- Unix shared library

Átfedő programrészek (overlay)

A program részekre bontása:

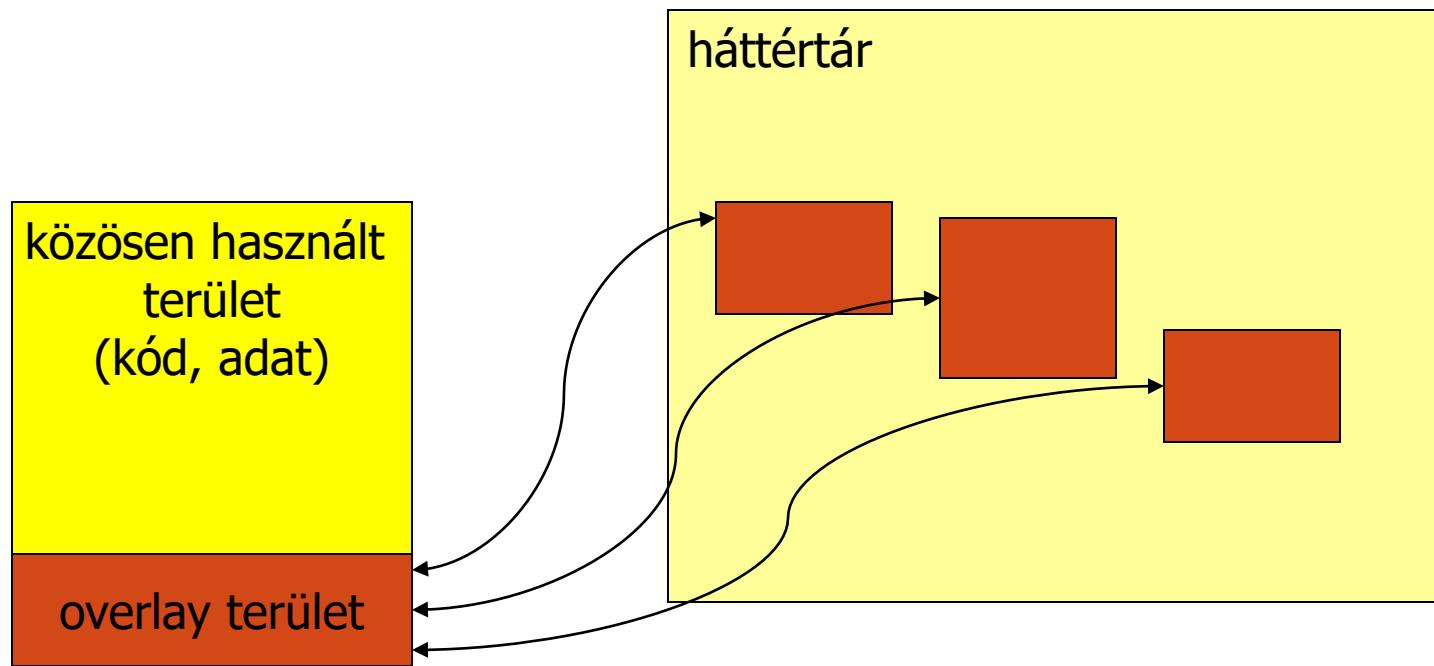
- közös adat- és programrészek (nem változik)
- olyan átfedő részek, amelyek közül egy időben csak egyre van szükség.

Az átfedő részeket egyesével töltjük be a memóriába.

Nincs szükség OS támogatásra.

Az átfedés számára fenntartott tárterület a legnagyobb átfedő programrész hosszával egyenlő.

Átfedő programrészek (overlay)



Memóriaallokációs elvek

Elvek:

- Egypartíciós rendszer
- Többpartíciós rendszer
- Tárcsere (swap)

Korszerű módszerek

- Szegmens szervezés
- Lap szervezés

Egypartíciós rendszerek

Memóriaszervezés:

Védett területek:

- OS
- speciális tárterületek
 - megszakítás vektorok
 - periféria címtartományok

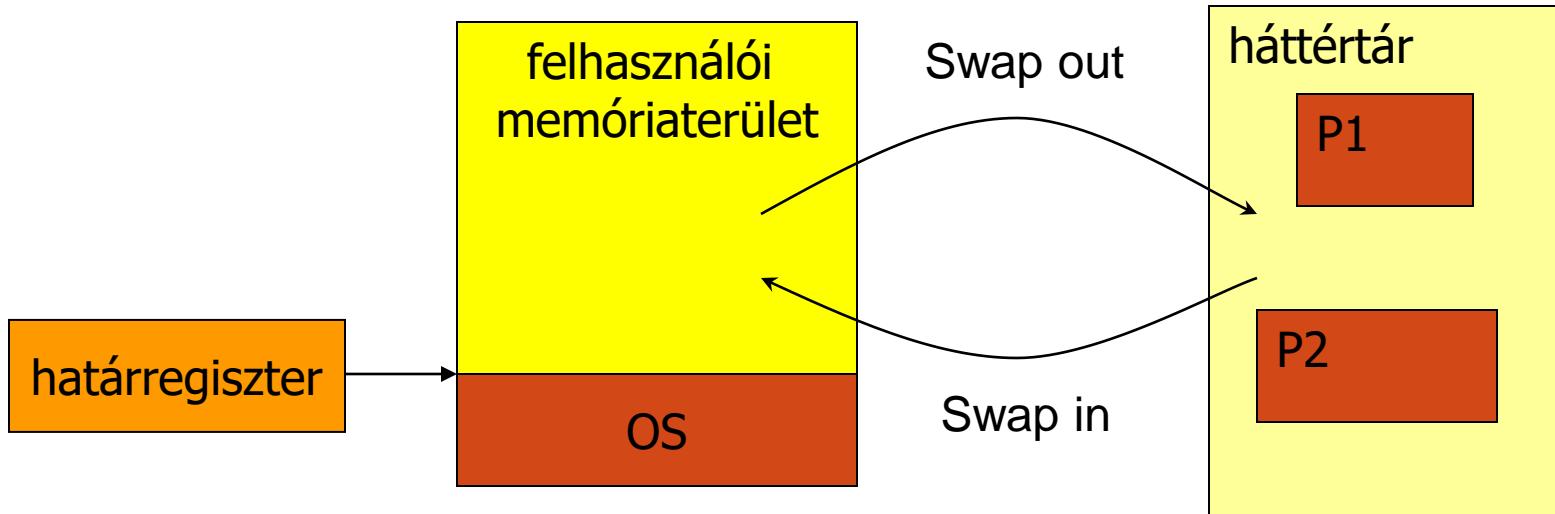
Felhasználói terület:

- A nem védett területen felüli cím-tartományt csak egy folyamat használja.
- A program az első szabad címre töltődik

Ha az OS-nek szüksége van memóriára, akkor

- vagy áthelyezi a programot,
- vagy a nem használt területről allokál.

Egypartíciós rendszer



Védelem: **felhasználói** és **rendszer** mód

Az OS területének védelmére elég egy regiszter, amely a program legkisebb címét tartalmazza

Felhasználói mód:

- Futás közben egy hardver figyeli, hogy minden hivatkozás a tárolt cím felett legyen.

Rendszer mód:

- Rendszerhíváskor a védelem kikapcsol, az OS az egész címtartományt eléri.

Többpartíciós rendszerek

A multiprogramozás elve megköveteli, hogy több folyamat legyen egy időben a tárban.

Lehetséges megoldások:

- Fix partíciós rendszerek:
 - az OS feletti tárterületet partíciókra bontják
 - a határok nem változnak.
- Változó partíció méretű partíció
 - a program igényeinek megfelelő partícióméret

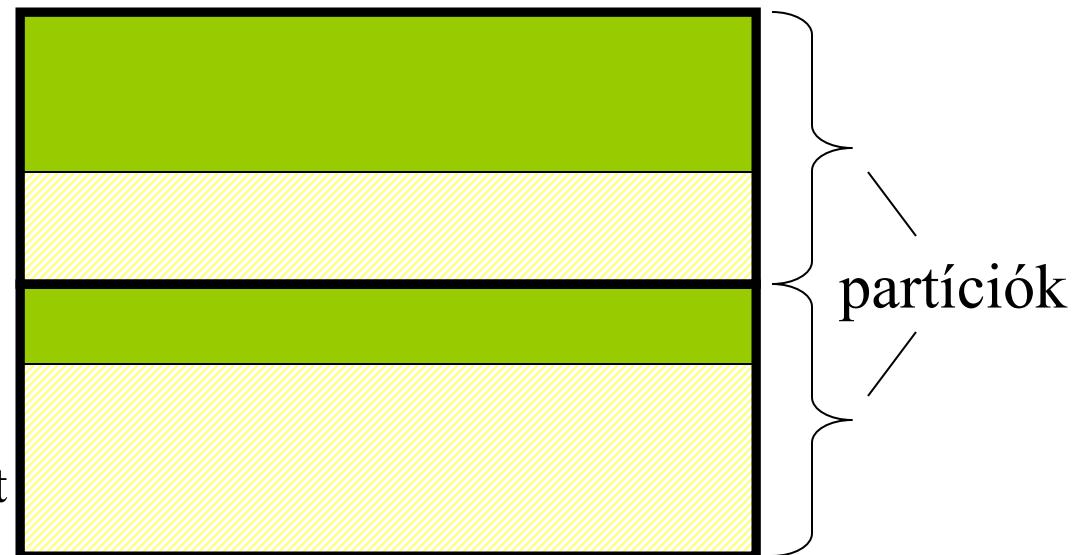
Belső tördelődés

Belső tördelődés (internal fragmentation):

- A folyamatok nem használják ki a rendelkezésére bocsátott partíciót.

Szabad tárterület

Felhasznált tárterület



Külső tördelődés

Külső tördelődés (external fragmentation):

- A szabad memória kis, egymással nem szomszédos részekre oszlik



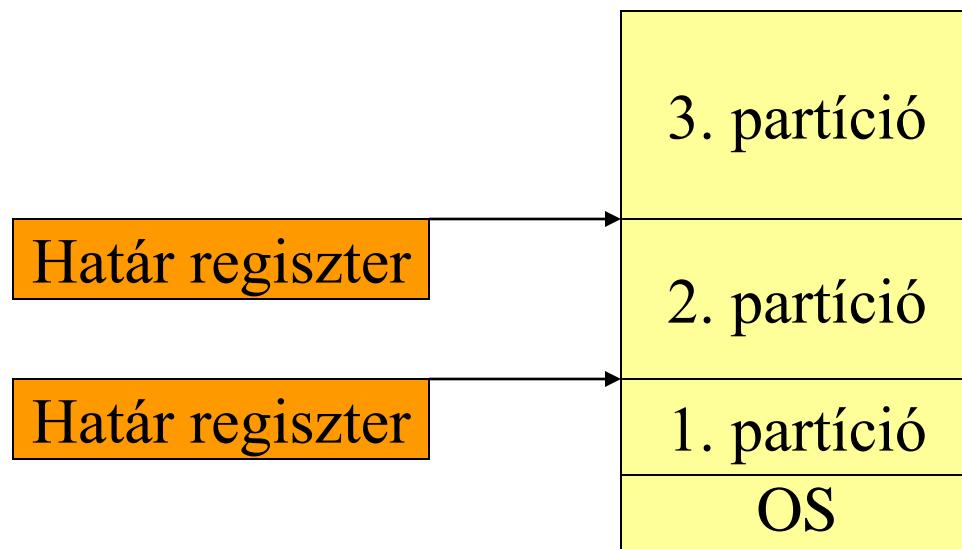
Fix partíciós rendszerek

Az OS feletti tárterületet fix partíciókra bontják.

A határok nem változnak.

Rossz hatékonyság: *belső tördelődés*

Védelem: határ-regiszterek



Változó partíciós rendszerek

A partíció a program igényeinek megfelelő méretű

Problémák: Szabad terület tördelődése

- Egy folyamat lefutásakor a használt memória felszabadul.
- Az OS nyilvántartja ezeket a területeket, az egymás melletti szabad területeket automatikusan összevonja.
- Sokszor ezen területek nem szomszédosak, így a szabad memória kis részekre oszlik (*külső tördelődés*).

Belső tördelődés nincs, hiszen csak a szükséges memóriát kapják meg a folyamatok.

Szabad területek tömörítése

compaction, garbage collection

A külső tördelődés bizonyos fok után lehetetlenné teszi újabb folyamat elindítását (elég a szabad terület, de a leghosszabb összefüggő szabad terület nem elég a folyamatnak).

Megoldás: a szabad helyek tömörítése.

Tömörítő algoritmusok:

- időigényes (nem biztos hogy megéri futtatni, esetleg jobban járunk, ha megvárjuk, míg néhány folyamat befejeződik).
- HW támogatást igényel.
- váratlanul lehet szükség rá (ezért pl. egy interaktív rendszer válaszideje hirtelen megnőhet)
- a tárterületek mozgatását körültekintően kell végrehajtani (az áthelyezési információk megőrzése, stb.)

Memóriaterületek lefoglalása

Stratégiák a külső tördelődés csökkentésére.

Az OS a szabad területek közül a következőképpen választhat:

legjobban megfelelő (*best fit*)

- legkisebb még elegendő méretű

első megfelelő (*first fit*)

- a kereséssel a tár elejétől indulunk, az első megfelelő méretűt lefoglaljuk.

következő megfelelő (*next fit*)

- a kereséssel az utoljára lefoglalt tartomány végéről indulunk, az első megfelelő méretűt lefoglaljuk.
- Igen gyors algoritmus, a memória 30%-a marad kihasználatlan
- (50%-os szabály, mivel a folyamatok által foglalt memória fele nincs kihasználva).

legrosszabban illeszkedő (*worst fit*)

- a legnagyobb szabad területből foglaljuk le, abban bízva, hogy a nagy darabból fennmaradó terület más folyamat számára még elegendő lesz.

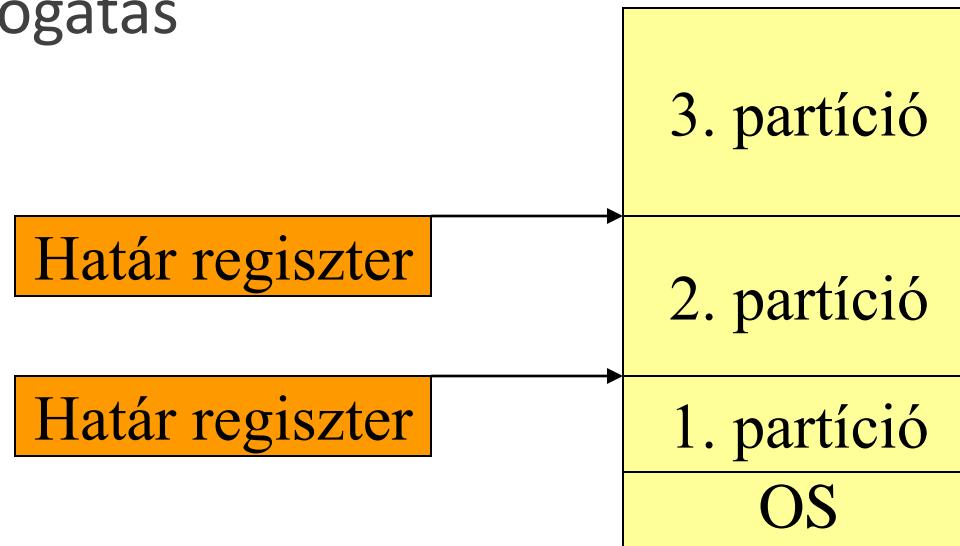
Hatókonyság szempontjából a legrosszabban illeszkedő a leggyengébb, a többi nagyjából azonos.

Védelem

Multiprogramozott rendszerekben nemcsak az OS, hanem más folyamatok területeit is védeni kell.

2 regiszter kell, amelyek a címtartomány határait tartalmazzák

hardver támogatás



Tárcsere (swap)

A tárcsere során az OS egy folyamat [teljes] tárterületét a háttértárra másolja, így szabadítva fel területet más folyamatok számára.

A perifériás átvitel miatt időigényes (sokkal több idő, mint egy környezetváltás, CPU ütemezéskor ezt figyelembe kell venni).

Problémák:

- Melyik folyamatot tegyük ki háttértárra?
 - Figyelembe kell venni a folyamat állapotát, prioritását, futási, várakozási idejét, vagy a lefoglalt partíció nagyságát.
- Melyik folyamatot mikor hozzunk be a háttértárról?
 - Az előző szempontok itt is érvényesek. Ügyelni kell a kiéheztetés veszélyére.
- Kerülendő a folyamatok felesleges pakolgatása.

A tárcsere korszerű módszerei

Hardver támogatás

Mesterséges folytonosság (*artificial continuity*)

- Virtuális címtartományban folytonos program a valóságban nem az.

A futó folyamatoknak nem az egész címtartománya van a központi tárban.

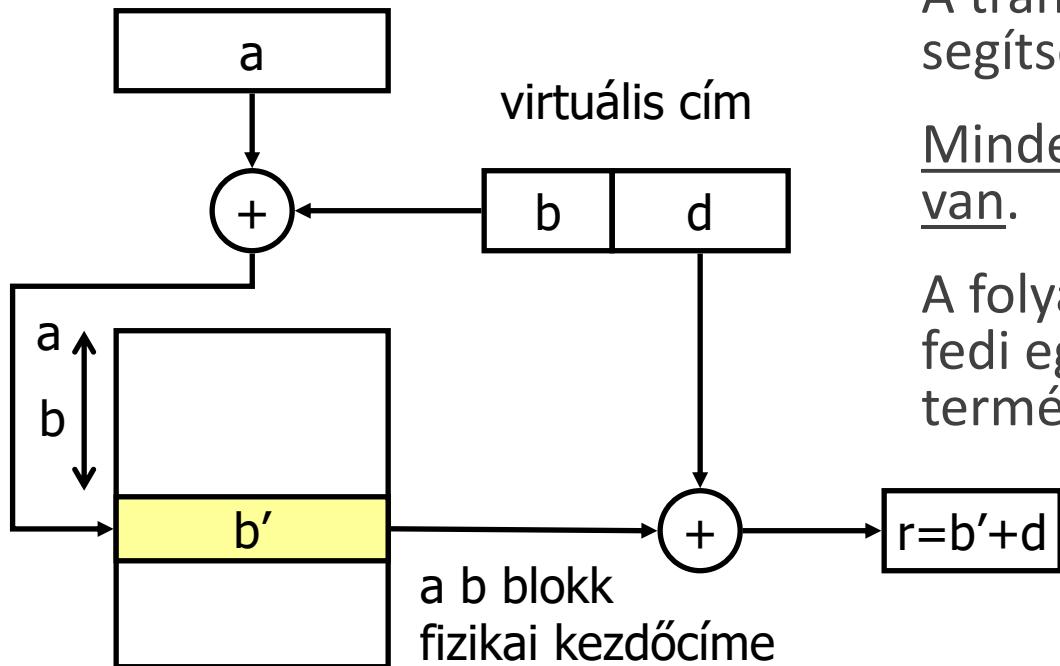
- A nem érvényes hivatkozás (nincs a központi tárban) esetén automatikus programrészlet betöltés (OS – hardver együttműködés).

Közös vonás a *futás közbeni címleképzés*.

- Csak hardver támogatással viselkedik megfelelő sebességgel.

Futás közbeni címleképzés

blokktábla kezdőcíme



Virtuális cím: $\langle b, d \rangle$

- b : blokkcím,
- d : eltolás, *displacement*

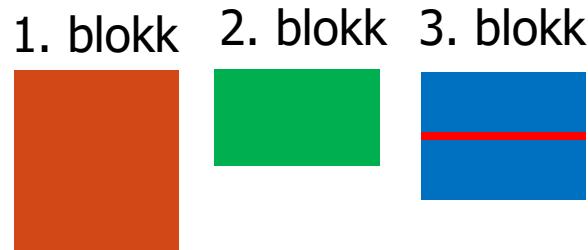
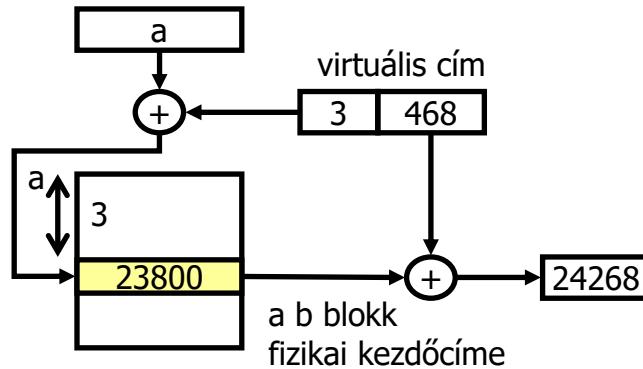
A transzformáció a blokktábla segítségével megy végbe.

Minden folyamatnak saját blokktáblája van.

A folyamatok virtuális címtartománya fedi egymást, de a fizikai címtartomány természetesen nem.

Futás közbeni címleképzés

blokktábla kezdőcíme



Logikai címtartomány

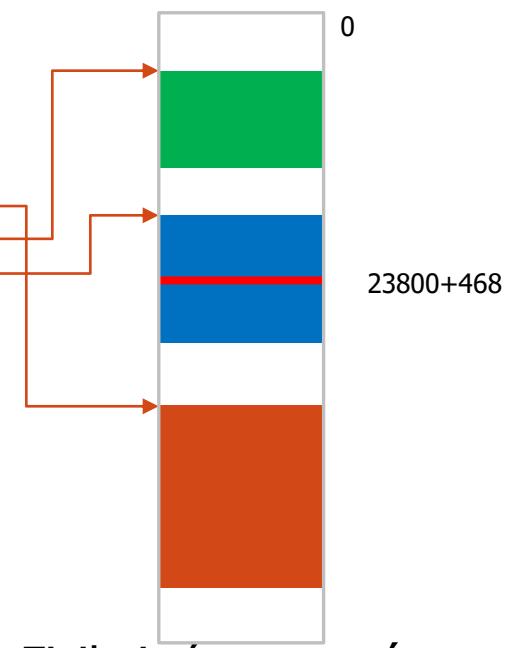
Virtuális cím: **<b, d>**

- **b**: blokkcím,
- **d**: eltolás, *displacement*

blokktábla

1.	55600
2.	12400
3.	23800

0
468



Szegmens szervezés

A logikai címtartományban a program memóriája nem egybefüggő terület, hanem önmagukban folytonos blokkok (**szegmensek**) halmaza.

A szegmens *logikai egység*. Pl.:

- főprogram,
- eljárások, függvények, módszerek, objektumok,
- lokális változók, globális változók,
- stack, szimbólumtábla, stb.

A címtranszformáció az előző általános modellnek megfelelő (lásd előző ábra!).

- Blokktábla → szegmenstábla
- Cím: <szegmenscím, eltolás>

Példa: szegmensek Linux alatt

Kevés szegmens, hogy hordozhatóbb legyen.

Fő szegmensek:

- *Kernel kód*
- *Kernel adat*
- *User kód* (az összes user folyamat osztozik rajta)
- *User adat* (ugyancsak osztott használat)

Szegmens szervezés védelme

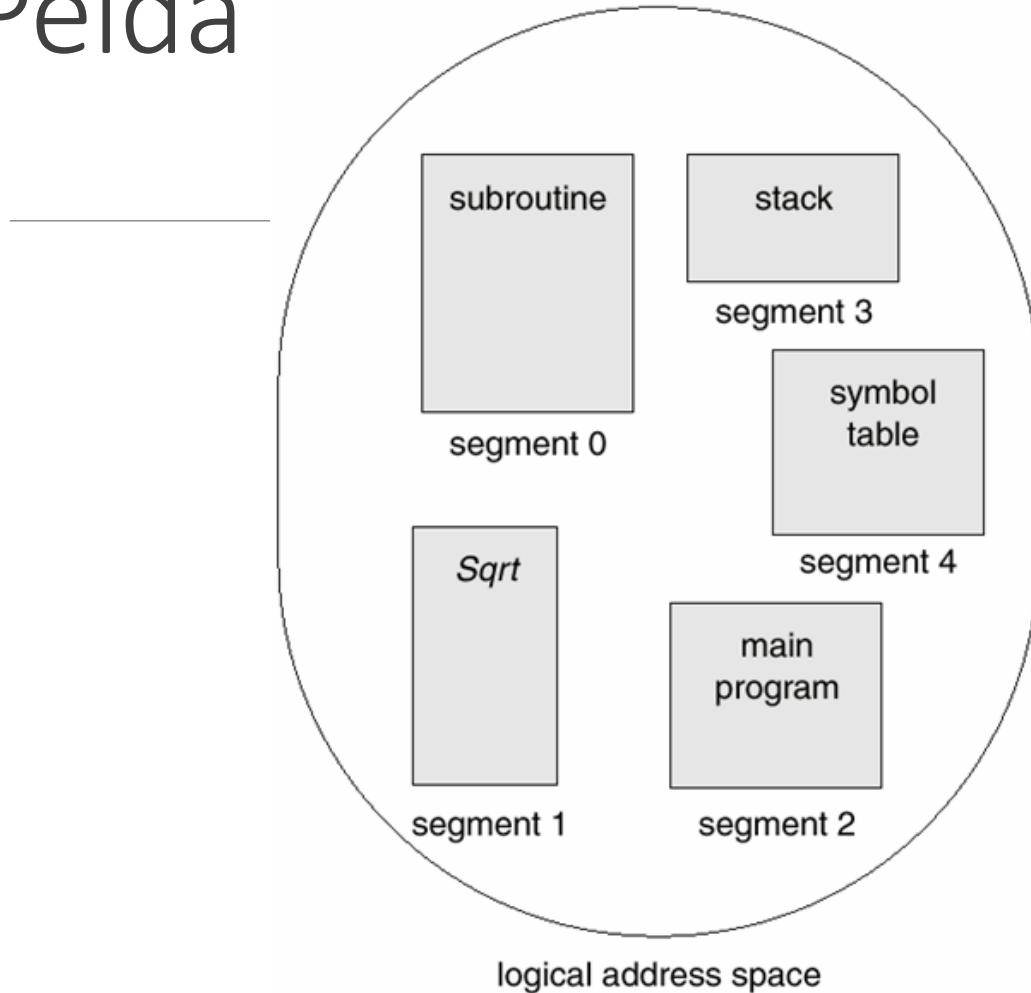
A folyamat nem címezhet ki a saját szegmenséből

- szegmensen belüli cím \leq szegmens mérete,
- különben megszakítás (segment overflow fault).
- A szegmenstábla tárolja a szegmens méretét (limit).

Hozzáférés ellenőrzés

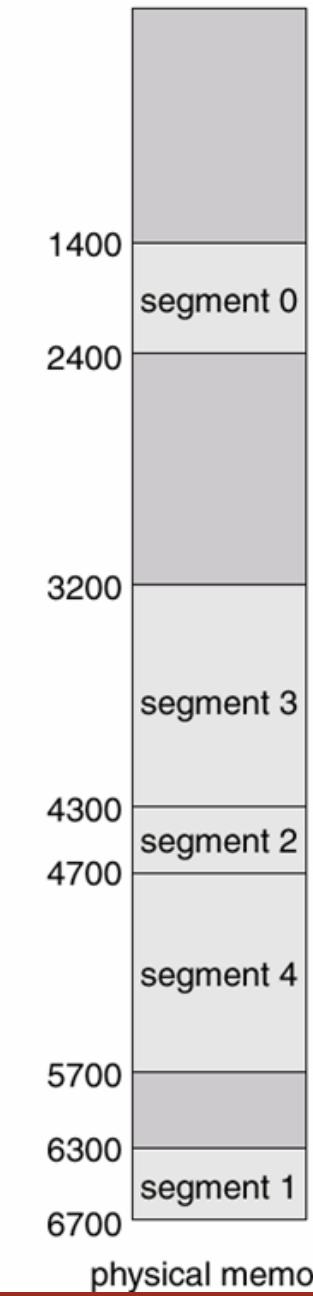
- olvasási jog (szegmens területét olvashatja)
- írási jog (szegmens területére írhat, az ott lévő értékeket módosíthatja)
- végrehajtási jog (a szegmensben gépi utasítások vannak, azokat a folyamat végrehajthatja)
- Jogosultság megsértése, megszakítást generál (segment protection fault).

Példa



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table



physical memory

Osztott szegmenshasználat

Közös utasítások használata:

- több folyamat azonos programot futtat
- kevesebb memóriahasználat
- lehet teljes program is, de rendszerkönyvtár is.

Közös adatterület

Folyamatok közötti kommunikáció.

Megvalósítás:

- A folyamatok szegmenstáblájában valamelyik szegmensnél azonos fizikai cím van.
- A jogosultságok természetesen különbözők, biztosítani kell a kölcsönös kizárást.

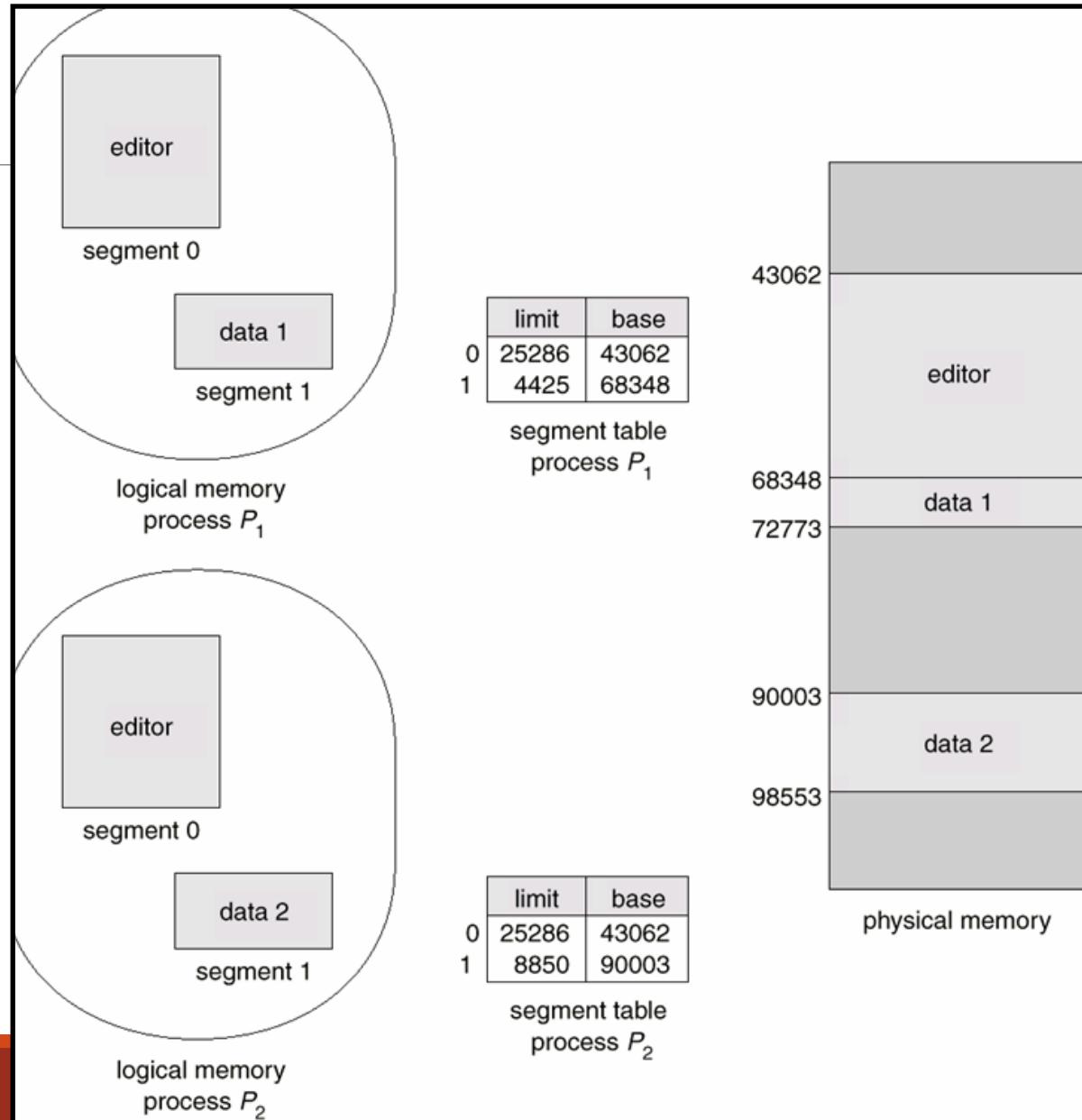
Példa

Szövegszerkesztő

- kódszegmens
- adatszegmens

Két példány:

- kódszegmens
osztott használata



Címtranszformáció háttértáron lévő szegmens esetén

Szegmenstabla tartalmaz még:

- benntartózkodási bitet (residency bit).
 - Jelentése: a szegmens a memóriában van-e.
- Információt, hogy hol van a háttértáron

Ha a benntartózkodási bit hamis:

- hiányzó szegmens hiba (*missing segment fault*) lép fel,
- amit az OS kezel le.

Szegmenshasználat hátránya:

- nagy külső tördelődés, mivel nem azonos méretűek a blokkok.

Szegmenstábla felépítése

Blokktábla/szegmenstábla

szegmensek		szegmens cím	szegmens hossz	RB	helye a háttértáron			R	W	X

Residency bit Hozzáférési jogosultságok
(Read/Write/Execute)

Lapszervezés

Azonos méretű blokkok (lap, page) használata

Megszünteti a külső tördelődést.

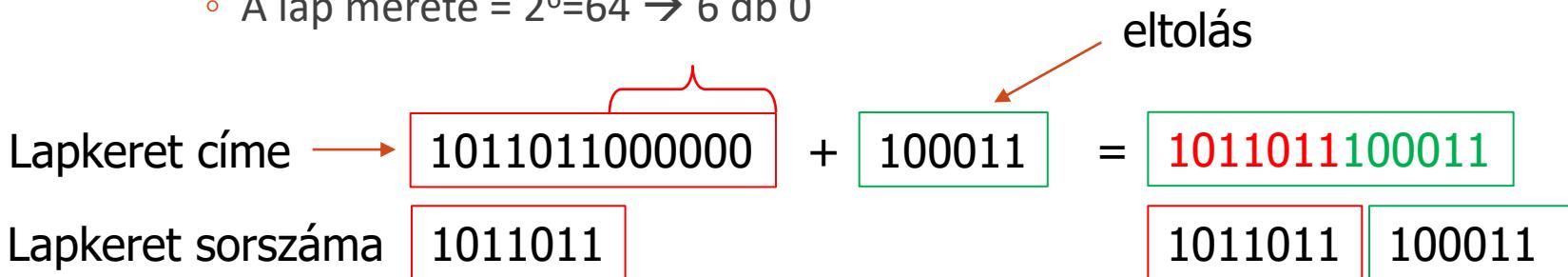
A belső tördelődés elkerülése miatt kis lapokat érdemes választani.

A lapok mérete mindig 2 hatvány.

Címtranszformáció

A lap mérete 2 hatványa!

- Pl.:
- A lapkeret címe: 1011011000000, az eltolás pedig 100011
- A lap mérete = $2^6=64 \rightarrow 6$ db 0

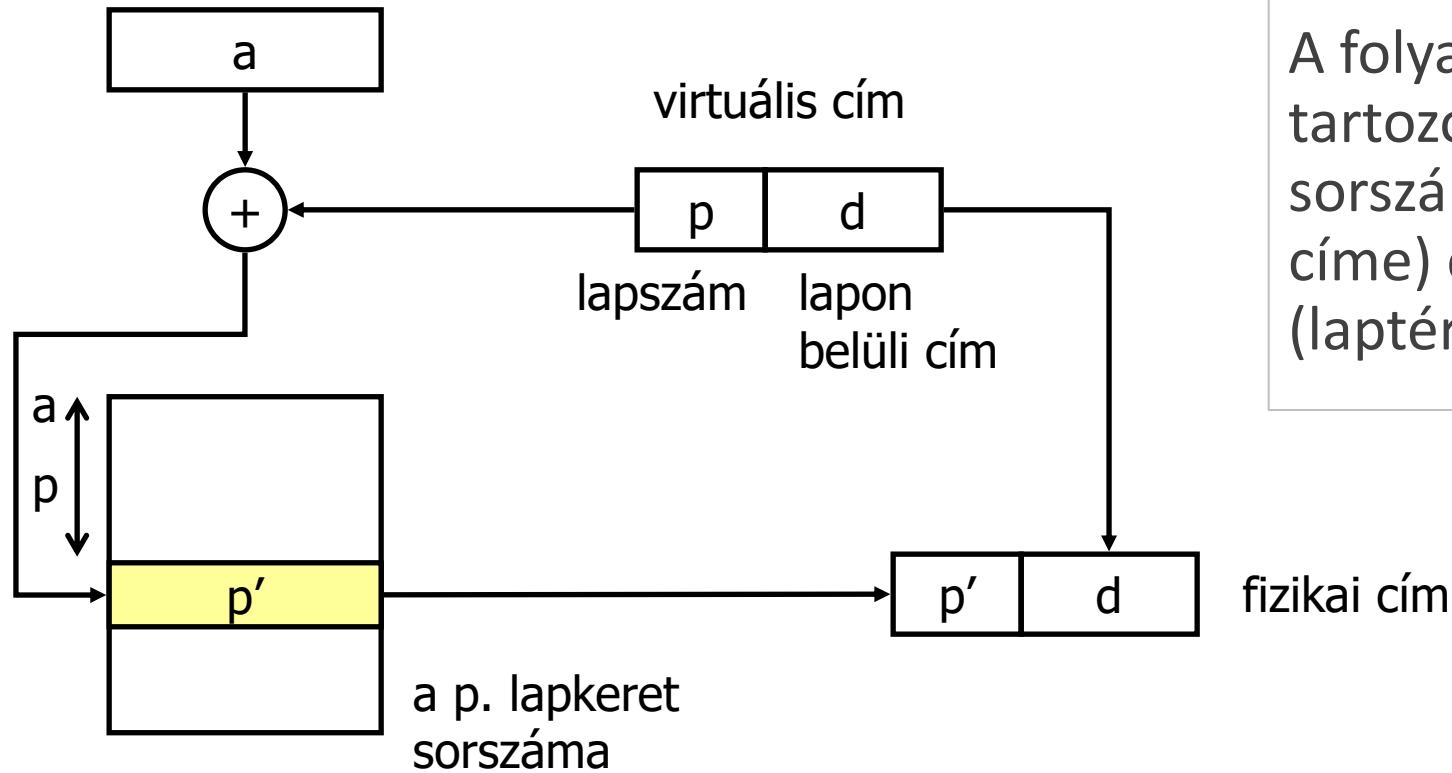


Címtranszformáció módszerei:

- Közvetlen leképzés
- Asszociatív leképzés
- Kombinált technikák

Közvetlen leképzés

laptábla kezdőcíme



Egszintű laptábla:

A folyamathoz tartozó minden lap sorszáma (tk. fizikai címe) egy táblában (laptérkép) van.

Példa

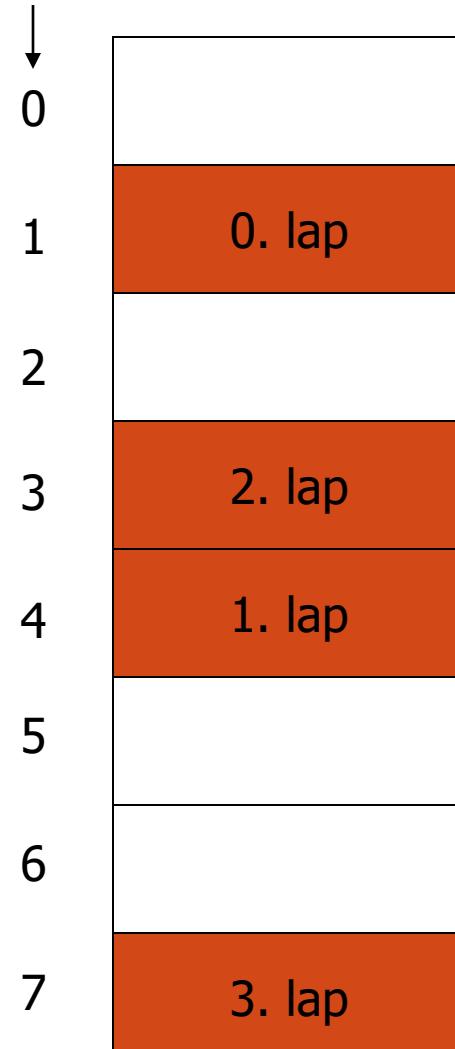
0. lap
1. lap
2. lap
3. lap

Logikai címtér

p	p'
0	1
1	4
2	3
3	7

laptábla

p' : lap-keretek (*nem kezdőcímek!*)



fizikai memória

Megjegyzés: a laptáblának ilyen kialakítás esetén természetesen nem kell tárolnia p értékeit, hiszen p maga a táblán belüli cím.

Példa

Lapok fizikai kezdőcímei → 0
(p^*N)

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

} lapméret $N = 4$

p	p'
0	1
1	4
2	3
3	7

laptábla



fizikai memória

Egyszintű laptábla problémái

Laptábla mérete nagy lehet, mivel a lapok száma sok.

Nehéz gyors elérésű tárban tartani.

Pl.:

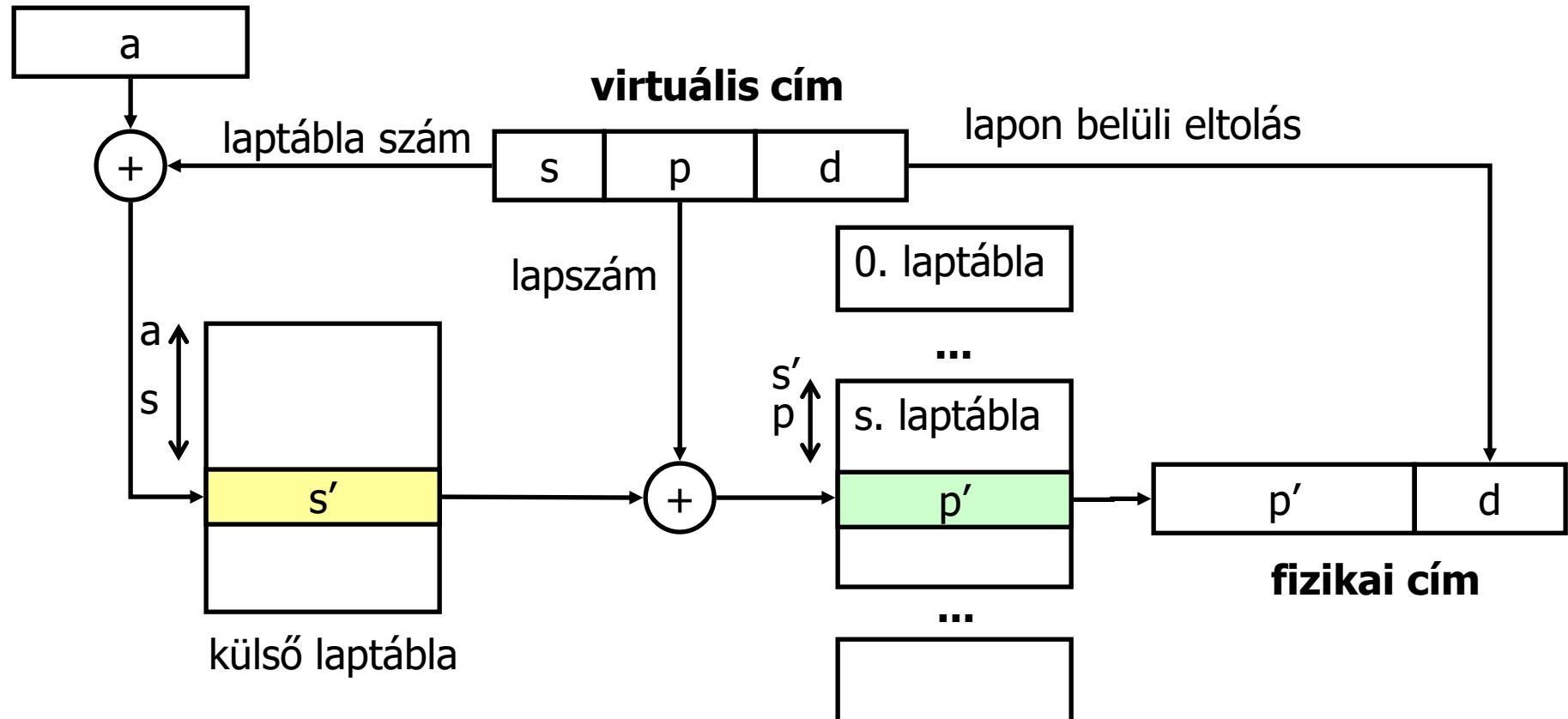
- 32 bites címtér (2^{32})
- 4 Kbites lapok (2^{12})
- 2^{20} ($\approx 10^6$) db bejegyzés

Megoldás: laptábla tördelése → többszintű laptábla.

Felfogható a laptábla lapozásának is.

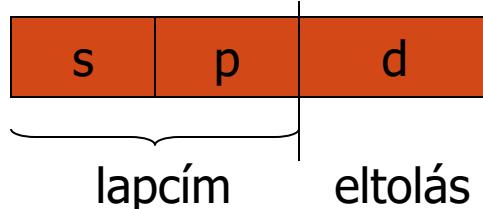
Többszintű laptábla

külső laptábla kezdőcíme

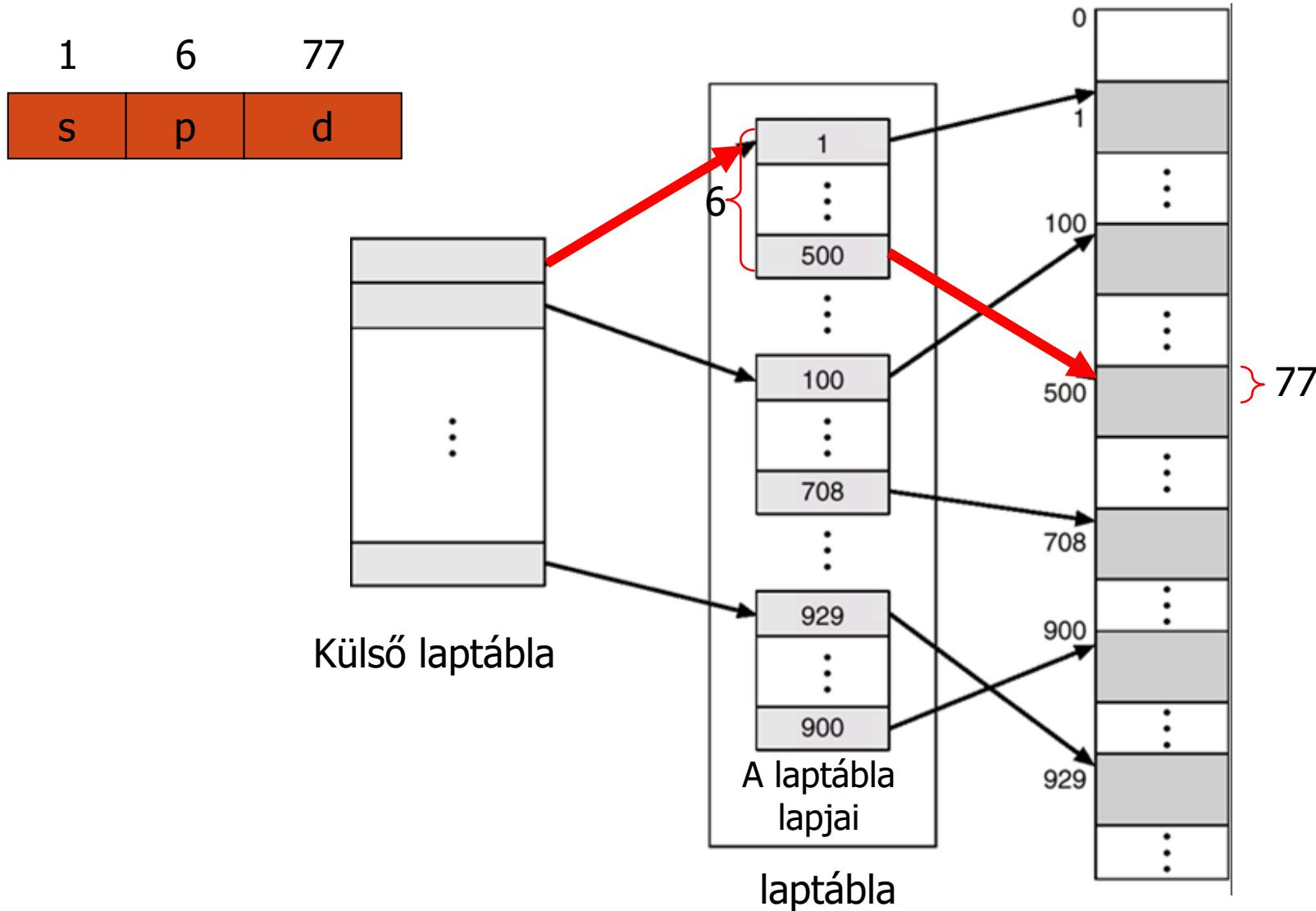


Tipikus példa (32 bites architektúrán):

- 10 bites laptábla-szám (s)
- 10 bites lapszám (p)
- 12 bites eltolás (d)



Többszintű laptábla példa



Asszociatív leképezés

Speciális gyors elérésű tár (asszociatív tár) segíti a címzést

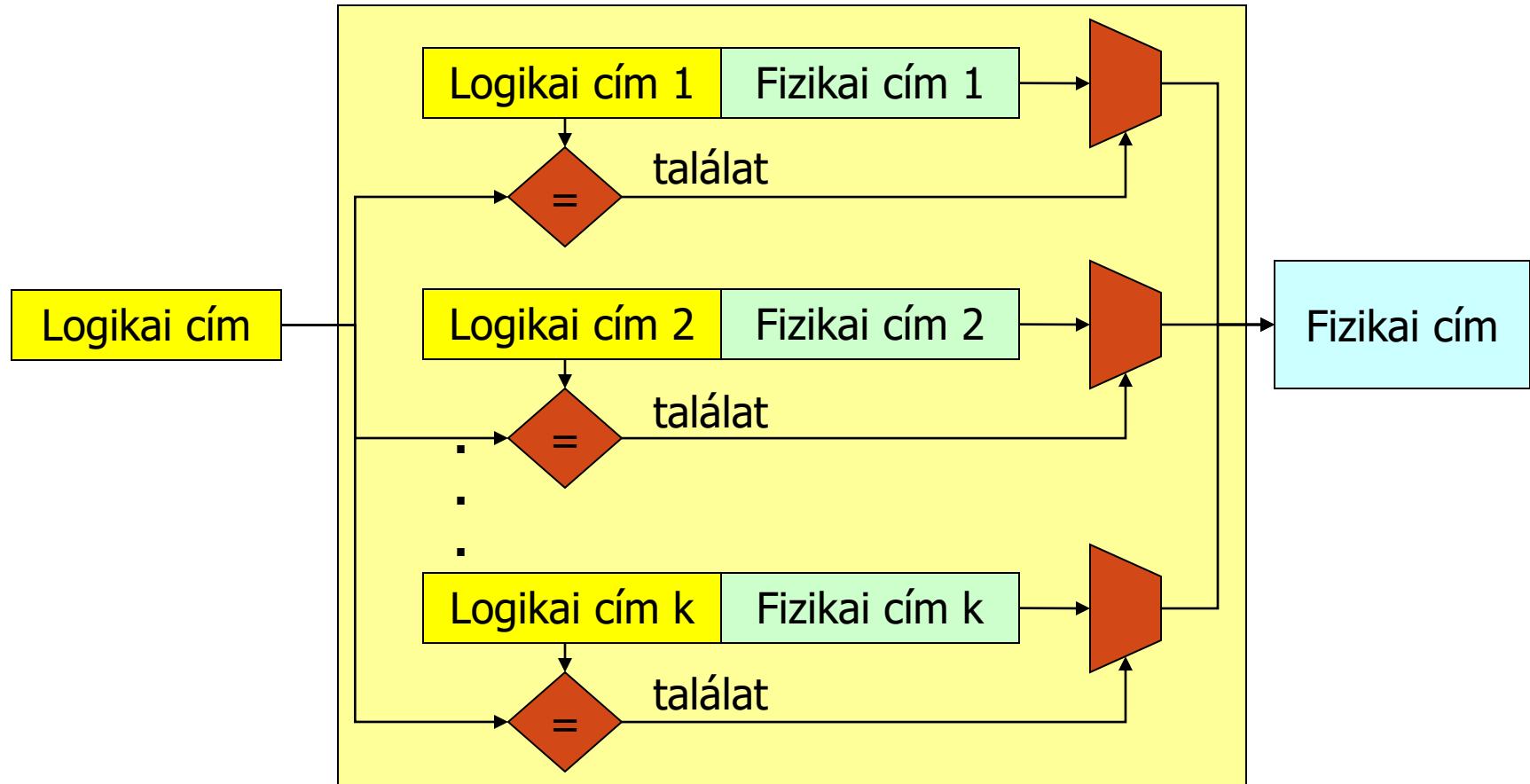
- translation look-aside buffer (TLB)

A laptábla gyorsító tárban a várhatóan gyakran használt lapok címét tároljuk.

A tár mérete itt sem elég nagy.

A gyakorlatban a kombinált technikák (laptábla + gyorsítótár) használhatók.

Az asszociatív tár működése



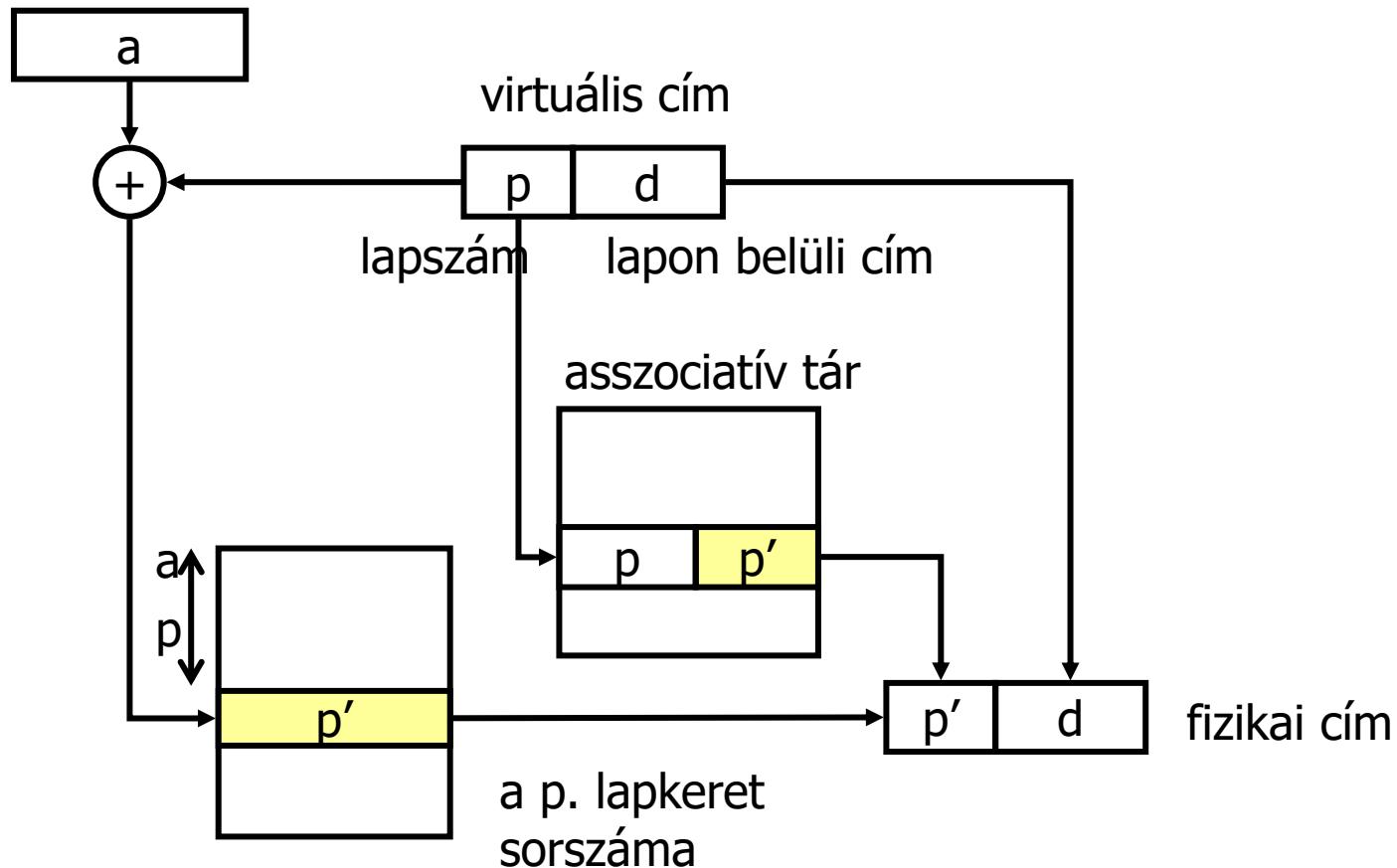
Keresés: párhuzamosan az összes tárolt logikai cím alapján.

Találat esetén a találatnak megfelelő fizikai cím kerül a kimenetre.

Drága → kis kapacitású

Kombinált technika

laptábla kezdőcíme



Kombinált technikák

A fizikai laptípus keresése egyszerre kezdődik mind az asszociatív tárban, mind a direkt laptáblában.

Ha az asszociatív tárban van találat, akkor a direkt keresés leáll.

Az asszociatív tárban lévő lapokat frissíteni kell, környezetváltás esetén az asszociatív laptáblát is cserélni kell.

Egy adott időszak alatt csak a teljes címtartomány kis része van kihasználva, így a találati arány elég magas lehet (80-99%).

Túlcímzés elleni védelem

Lapon belüli túlcímzés ellen nem kell védeni, hiszen minden lapon belül kiadható cím megfelelő

- kivétel: utolsó lap...

A lapok érvényességét egy bit jelzi (lásd a következő példát).

Címtranszformáció a háttértáron lévő lap esetén:

- Az érvényesség bit jelzi, hogy a memóriában van-e a lap. Ha nincs (invalid bit), akkor lehet, hogy a háttértáron van. Utóbbi esetben a tábla a háttértáron való elhelyezkedés információját tárolja.

Példa

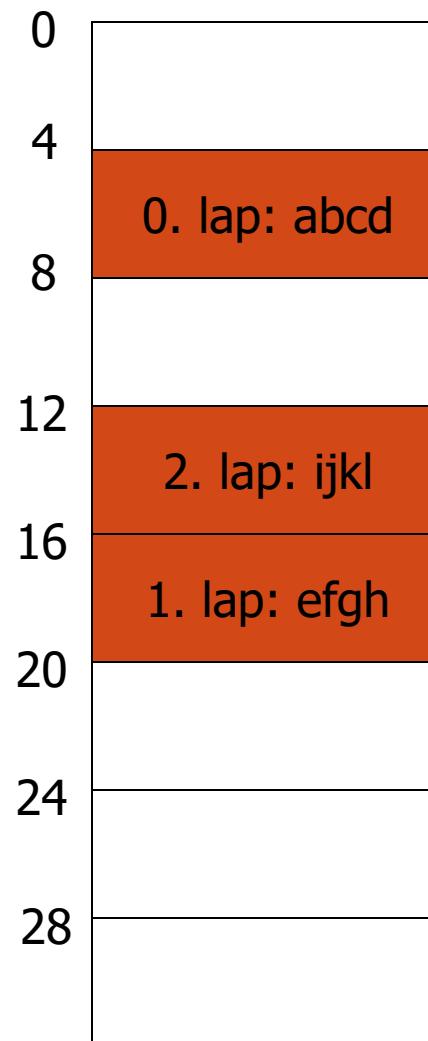
0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

Logikai címtér

valid/invalid bit

p	p'	v
0	1	v
1	4	v
2	3	v
3	0	i

laptábla



fizikai memória

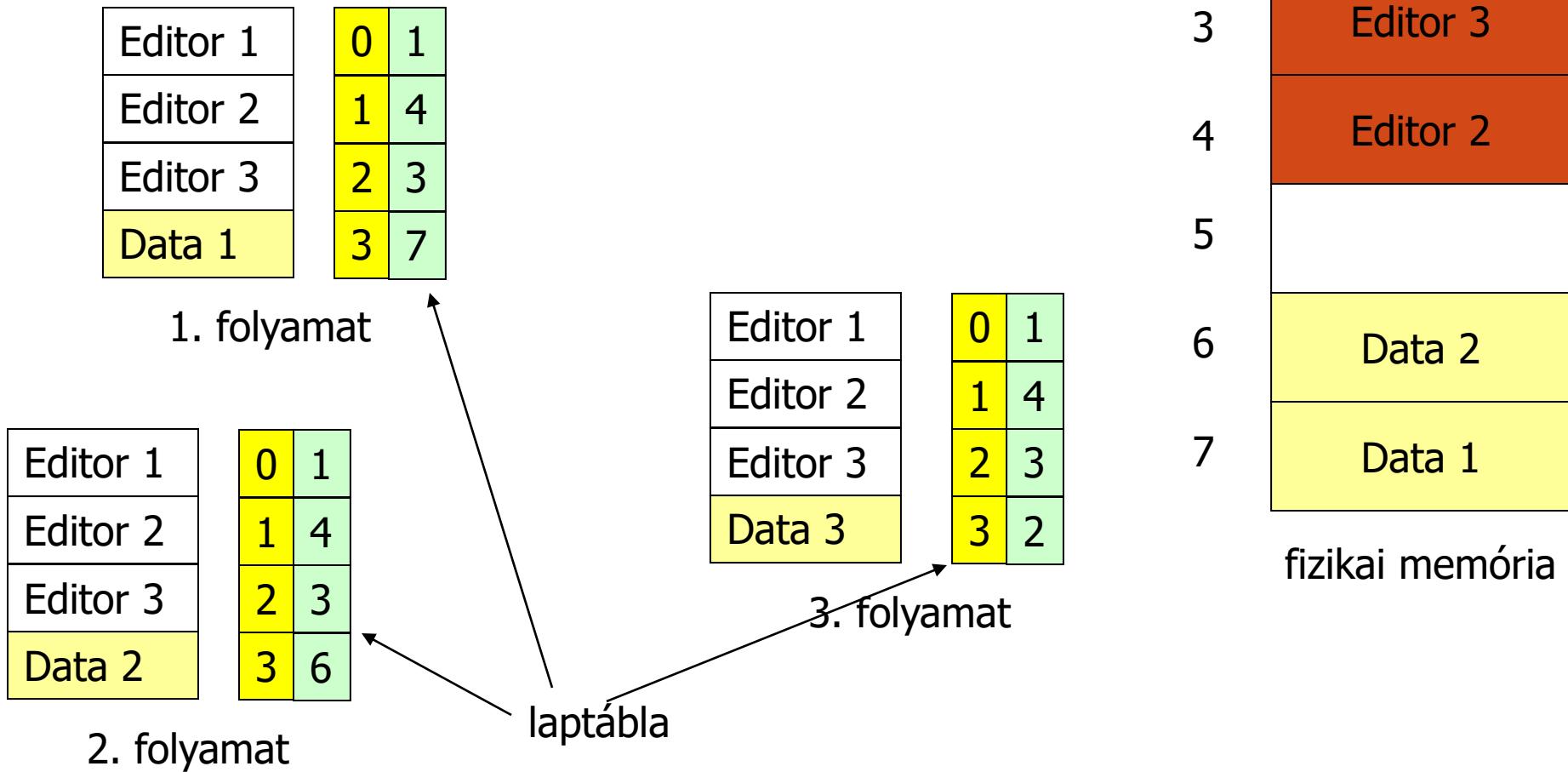
Osztott laphasználat

Hasonló az osztott szegmenshasználathoz, több folyamat laptérkép táblája azonos fizikai címekre hivatkozhat.

Pl.:

- közösen használt kód. Ilyenkor az OS gondoskodik a védelemről:
- Szövegszerkesztő
 - 3 lap kód
 - 1 lap adat
- Ha 20 példány fut, akkor a memóriaigény
 - $4 \times 20 = 80$ lap (nem osztott laphasználat)
 - $1 \times 20 + 3 = 23$ lap (osztott laphasználat)

Példa



6.5.3. Kombinált szegmens- és lapszervezés

Egyesíti a két technika előnyeit.

- Lap szervezés: nincs külső tördelődés, nem kell a teljes szegmensnek a tárban lennie.
- Szegmens szervezés: tükrözi a folyamat logikai társzerkezetét, hozzáférési jogosultság megoldható.

Címtranszformáció: lényegében egy kétszintű táblakezelés.

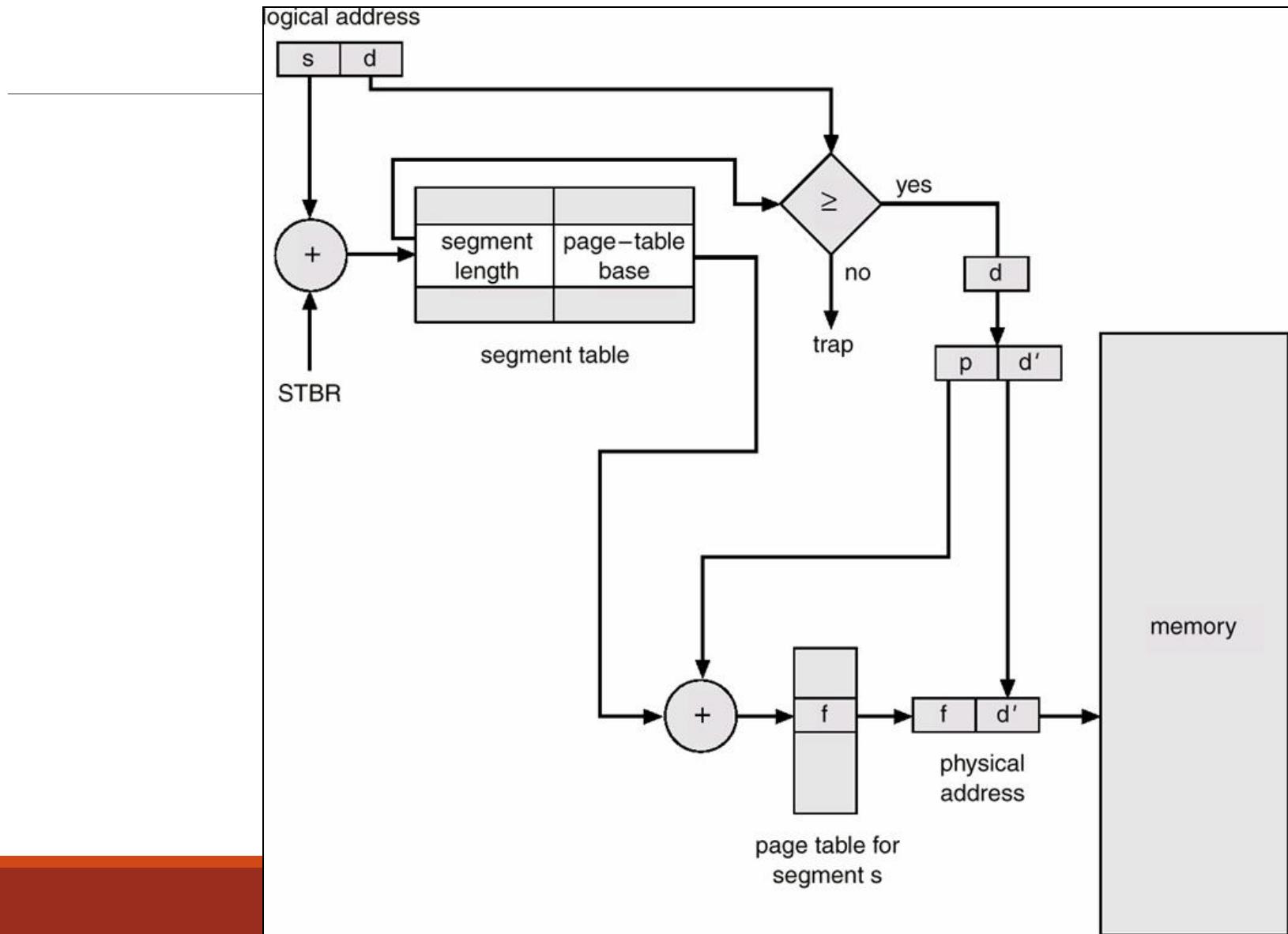
- Első szint: laptábla címeket tartalmazó szegmenstábla
- Második szint: fizikai lapcímeket tartalmazó laptábla (szegmensenként egy).

A cím három részre tagozódik (szegmenscím, lapcím, lapon belüli eltolás).

Hozzáférési jogok ellenőrzése a szegmens szervezésének megfelelően történik.

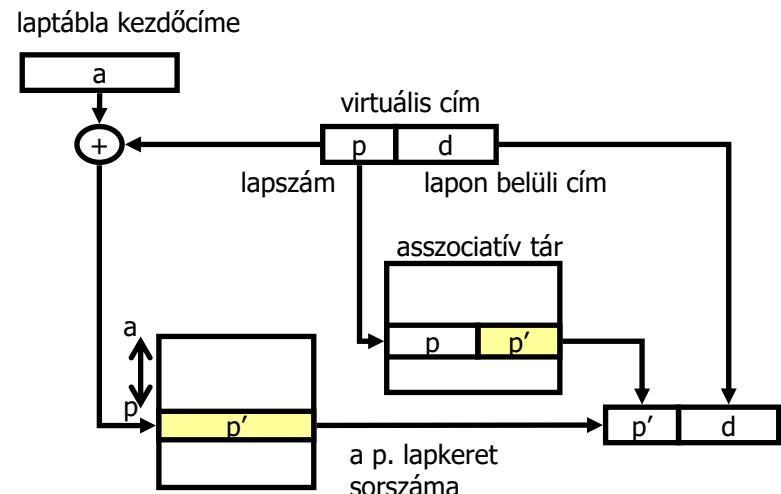
Osztott tárhelyszín által a szegmens szervezésének megfelelően történik.

Példa: MULTIX operációs rendszer címtranszformációja



Példa: Átlagos hozzáférési idő

Kombinált (laptáblát és asszociációs memóriát is tartalmazó) rendszerben a memória-hozzáférés ideje 100ns. Az asszociációs memória 20ns alatt érhető el. A találati arány 90%. Mekkora az átlagos elérési idő?



Válasz:

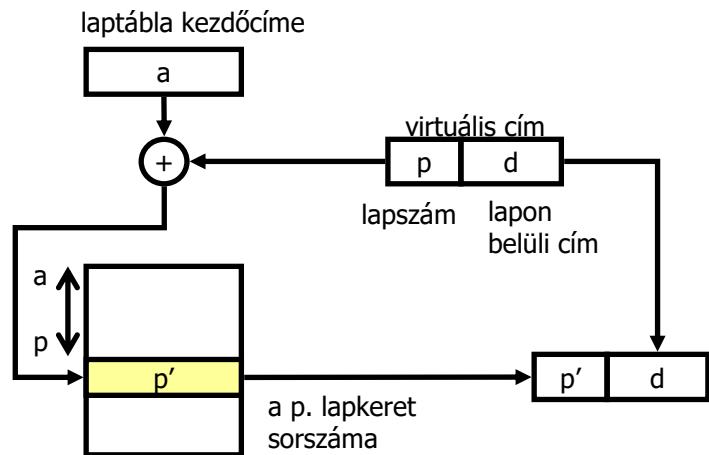
Fizikai cím elérése asszoc. memóriával: $20\text{ns} + 100\text{ns} = 120\text{ns}$

Fizikai cím elérése asszoc. memória nélkül: $100\text{ns} + 100\text{ns} = 200\text{ns}$

Átlagos elérési idő: $0.9 * 120\text{ns} + 0.1 * 200\text{ns} = 128\text{ns}$

Példa: Optimális lapméret

Egy rendszerben az átlagos folyamat mérete $s = 1\text{MB}$. Egy laptábla bejegyzés mérete $e = 8\text{ byte}$. Mekkora a lapok ideális méret (x)?



Válasz:

Az „elpazarolt” memória mérete (K):

- Laptábla bejegyzések: s/x^*e
- Az utolsó lap átlag fele üres: $x/2$

$$K = s/x^*e + x/2$$

$$\text{Optimum: } dK/dp = 0 = -se/x^2 + 1/2 \rightarrow p = \sqrt{2se} = 4\text{kB}$$

Operációs rendszerek

8. VIRTUÁLIS TÁRKEZELÉS

Felhasznált irodalom:

- Kóczy-Kondorosi (szerk.): Operációs rendszerek mérnöki megközelítésben
- Tanenbaum: Modern Operating Systems
- Silberschatz, Galvin, Gagne: Operating System Concepts

Tartalom

Bevezetés

A virtuális tárkezelés általános elvei

Lapcsere stratégiák

Folyamatok lapigénye, lapok allokációja

Egyéb tervezési szempontok

Bevezetés

Virtuális tárkezelés:

- Olyan szervezési elvek, algoritmusok összessége, amelyek biztosítják, hogy a rendszer folyamatai logikai címtartományainak **csak egy - a folyamat futásához szükséges - része legyen a központi tárban.**

Korábbi „helytakarékos módszerek”:

- késleltetett betöltés,
- tárcserék, stb.

Ezek nem voltak általános megoldások.

Motiváció 1

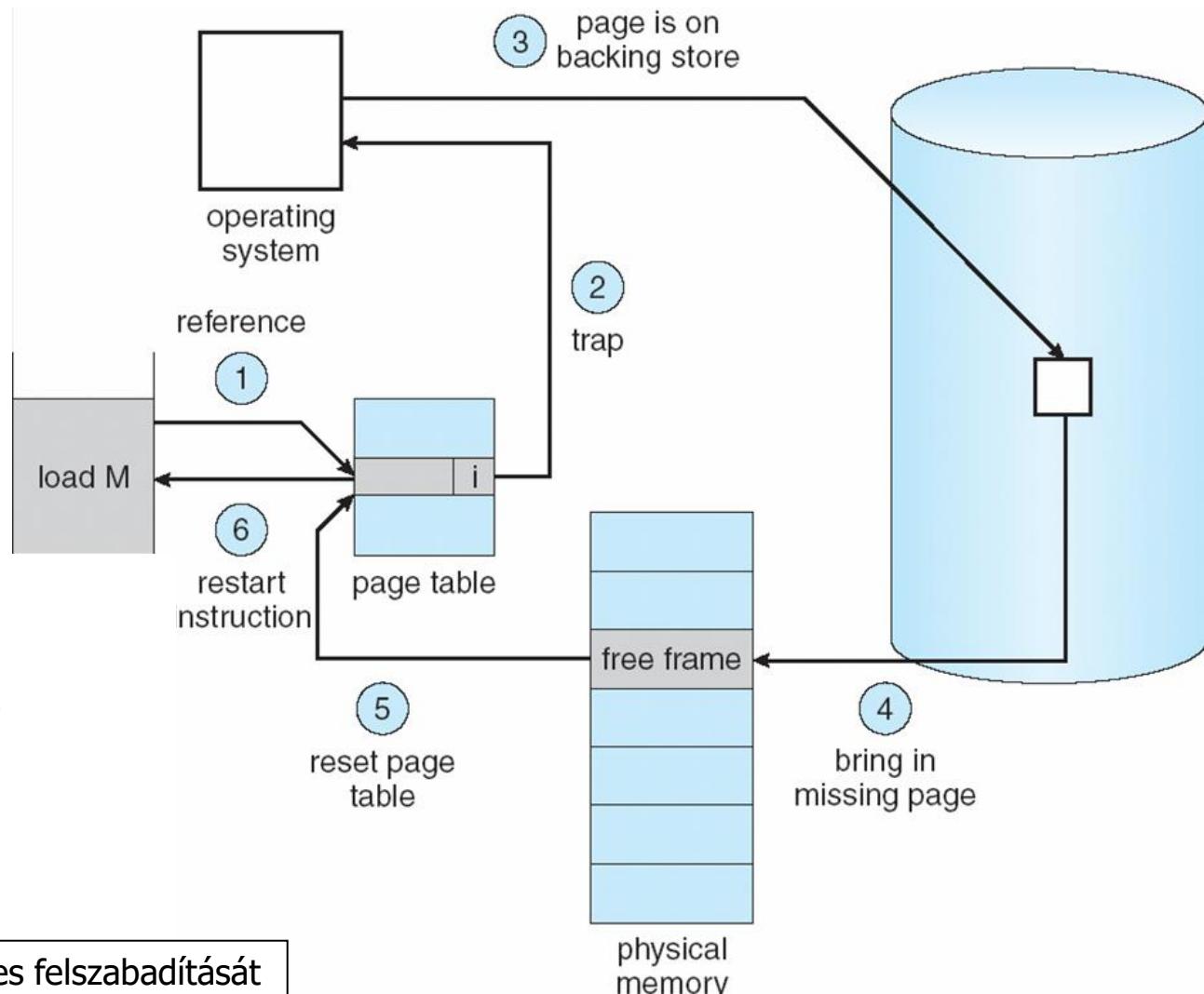
A programok nem használják ki a teljes címtartományukat

- tartalmaznak ritkán használt kódrészleteket (pl. hibakezelő rutinok)
- a statikus adatszerkezetek általában túlméretezettek (statikus vektorok táblák stb.).
- a program futásához egy időben nem kell minden részlet (overlay).
- Időben egymáshoz közeli utasítások és adatok általában a térben is egymáshoz közel helyezkednek el (lokalitás).

Nem célszerű a egész programot a tárban tartani:

- programok méretét ne korlátozza a tár nagysága, a program nagyobb lehet mint a ténylegesen meglévő memória mérete.
- a memóriában tartott folyamatok száma növelhető (nő a multiprogramozás foka, javítható a CPU kihasználtság és az átbocsátó képesség)

A megvalósítás elve



1. Érvénytelen laphivatkozás
2. Megszakítás
3. Lap háttértárról előkeresése
4. Lap üres keretbe betöltése*
5. Laptábla aktualizálása
6. Utasítás újraindítása

* Az ábra nem mutatja keret esetleges felszabadítását

A megvalósítás elve 1.

Amikor a folyamat érvénytelen, a memóriában nem található címre hivatkozik, hardver megszakítást okoz, ezt az OS kezeli, és behozza a kívánt blokkot.

Lépései:

Az OS megszakítást kezelő programrésze kapja meg a vezérlést (1,2)

- elmenti a folyamat környezetét
- elágazik a megfelelő kiszolgáló rutinra
- eldönti, hogy a megszakítás nem programhiba-e (pl. kicímzés)

A megvalósítás elve 2.

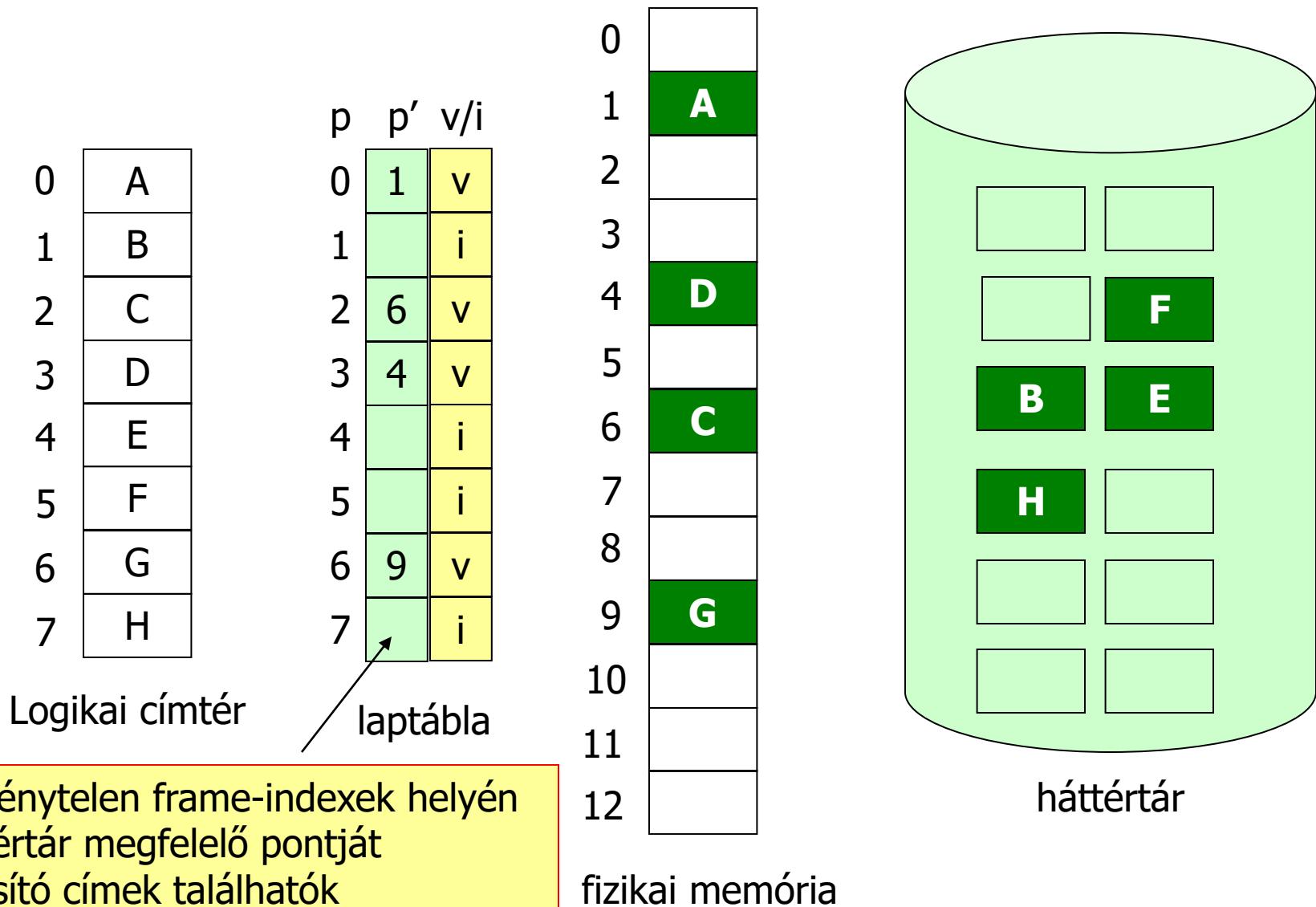
A kívánt blokk beolvasása a tárba (4,5)

- Az OS a blokknak helyet keres a tárban; ha nincs szabad terület, akkor fel kell szabadítani egy megfelelő méretű címtartományt, ennek tartalmát esetleg a háttértárra mentve
- Beolvassa a kívánt blokkot

Megjegyzés: A perifériás műveletek sok időt vesznek igénybe, ki kell várni az eszköz felszabadulását, az előkészítő műveletet (fejmozgás) és az átvitelt. A rendszer jobb kihasználtsága érdekében a megszakított folyamatot az OS várakozó állapotba helyezi, és egyéb folyamatot indít el. Amikor a kívánt blokk bekerül a tárba, a folyamat futás kész lesz.

A folyamat újra végrehajtja a megszakított utasítást (6)

Példa: A rendszer állapota



Laphiba hatása a folyamatok futásának sebességére

Effektív hozzáférési idő =

$$(1 - p) * \text{memória hozzáférési idő} + p * \text{laphiba idő}$$

p a laphiba gyakorisága

Mivel a laphiba kiszolgálása 5 nagyságrenddel nagyobb lehet, így p -nek kicsinek kell lennie.

Példa

Effektív hozzáférési idő (EAT) =

$$(1 - p) * \text{memória hozzáférési idő} + p * \text{laphiba idő}$$

memória hozzáférési idő = 1 μs

Egy lap kiírási/beolvasási ideje (swap in/out): 10ms

A helyettesítendő lapok 50%-a módosult, tehát ezeket a háttértárra ki is kell írni (swap out).

→ átlagos swap idő: 15ms

$$\text{EAT} = (1 - p) \times 1 \mu\text{s} + p \times 15000 \mu\text{s} \sim 15000p \ (\mu\text{s})$$

Alapvető kérdések

A betöltendő blokk (lap) kiválasztása (*fetch*)

A behozott blokk a valós tárba hova kerüljön (*placement*)

Ha nincs hely, akkor melyik blokkot tegyük ki (*replacement strategy*)

Hogyan gazdálkodjunk a fizikai tárral, azaz egy folyamat számára hány lapot biztosítsunk.

A betöltendő lap kiválasztása

Igény szerinti lapozás (demand pages)

- Előnye:
egyszerű a lapot kiválasztani, a tárba csak a biztosan szükséges lapok kerülnek be
- Hátránya:
Új lapokra való hivatkozás mindig laphibát okoz.

Előretekintő lapozás (anticipatory paging)

- Az OS megpróbálja kitalálni, hogy a folyamatnak a jövőben melyik lapokra lesz szüksége és azokat "szabad idejében" betölti.
- Ha a jóslás gyors és pontos, akkor a futási sebesség jelentősen felgyorsul
- Ha a döntés hibás, a felesleges lapok foglalják a tárat.
- A memória ára jelentősen csökken, így a mérete nő, a hibás döntés ára (a felesleges tárfoglalás) egyre kisebb.
- Az előretekintő lapozás egyre népszerűbb.

Lapcsere stratégiák

Akkor lenne **optimális**, ha azt a lapot választaná ki, amelyre legtovább nem lesz szükség.

A lapcserét nagyban gyorsítja, ha a mentést csak akkor végezzük el, ha az a lap a betöltés óta módosult. A hardver minden lap mellett nyilvántartja, hogy a lapra írtak-e a betöltés óta (*modified* vagy *dirty* bit).

Algoritmusok:

- Véletlen kiválasztás
- Legrégebbi lap (FIFO)
- Újabb esély
- Óra algoritmus
- Legrégebben nem használt lap
- Legkevésbé használt lap
- Mostanában nem használt lap

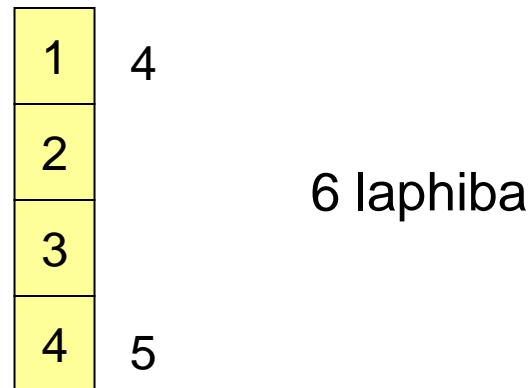
Példa: optimális algoritmus

Algoritmus: azt a lapot helyettesítjük, amelyre még a leghosszabb ideig nem lesz szükség.

Laphivatkozások (referencia-string):

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

4 keret (a folyamatnak ennyi lapja lehet egyszerre a memóriában)



Ezt használjuk referenciának az algoritmusok teljesítmény-mérésekor

Legrégebbi lap (FIFO)

A tárban lévő legrégebbi lapot cseréli le.
Megvalósítása egyszerű FIFO listával történik.

Hibája:

- Olyan lapot is kitesz, amelyet gyakran használnak
- Felléphet egy érdekes jelenség. Ha növeljük a folyamatokhoz tartozó lapok számát, a laphibák száma esetenként nem csökken, hanem nő! (Bélády-anomália)

Bélády László magyar származású kutató, az IBM virtuális tárkezelésének kidolgozója.

Példa: FIFO

Laphivatkozások (referencia-string): 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

3 keret (folyamatonként ennyi lap lehet egyszerre a memóriában)

1	1	4	5
2	2	1	3
3	3	2	4

9 laphiba

4 keret

1	1	5	4
2	2	1	5
3	3	2	
4	4	3	

10 laphiba

Béládi-anomália

A Béládi-anomália

Laphivatkozások

FIFO

3 lap kerettel

Laphibák

FIFO

4 lap kerettel

Laphibák

Újabb esély (Second Chance)

A FIFO egy változata

Minden lap tartalmaz egy R hivatkozás bitet, ami kezdetben törölve van.

A lapra hivatkozáskor $R=1$.

Lapcsere esetén:

- Ha a sor elején levő lapon $R=0$, akkor csere.
- Ha a sor elején levő lapon $R=1$, akkor az R bitet töröljük és a lapot a FIFO végére rakjuk. A következő (most már első) lappal próbálkozunk tovább.

Kiküszöböli a FIFO fő hibáját.

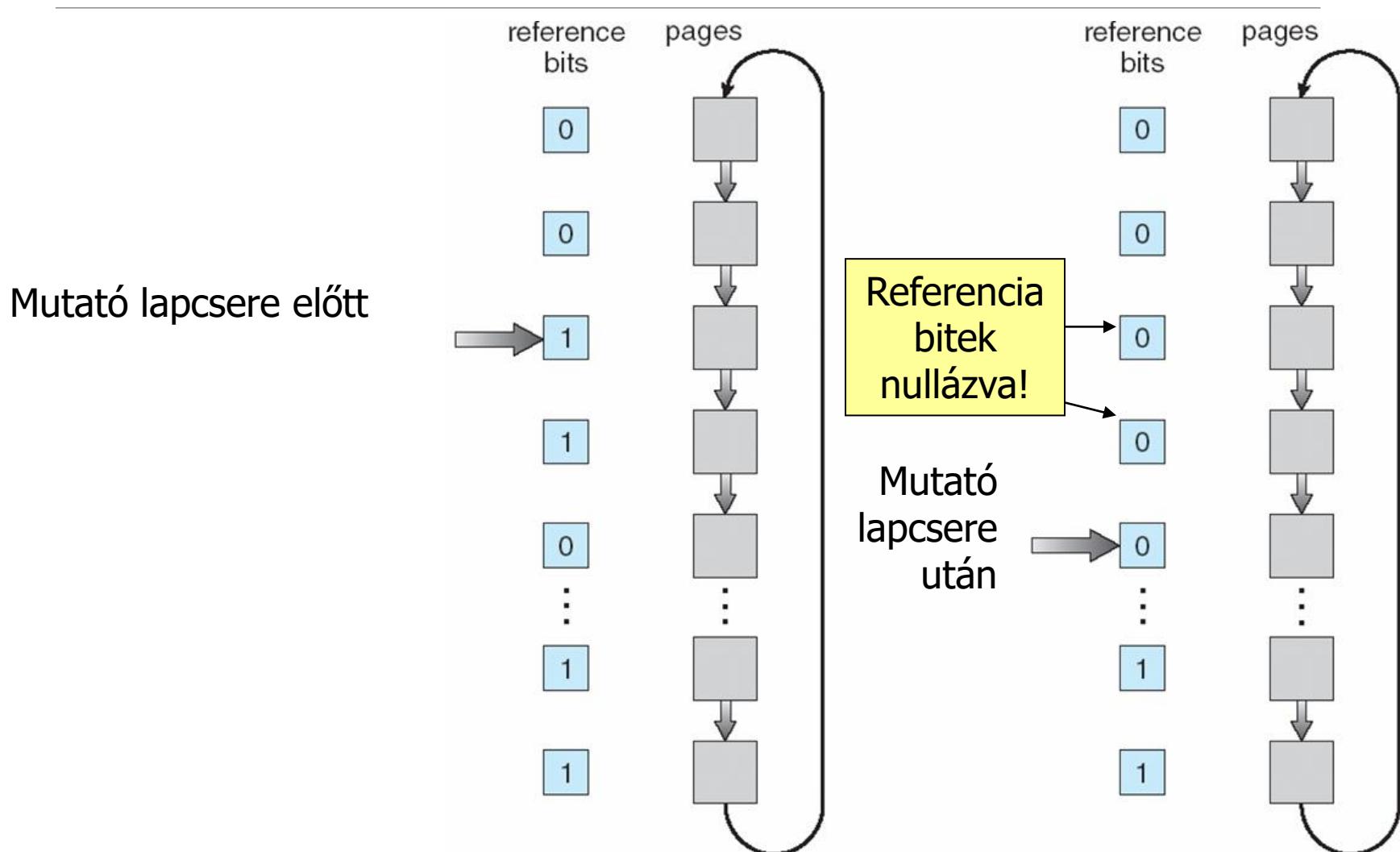
Óra algoritmus (Clock)

Az újabb esély algoritmus másik implementációja

A lapok körkörös láncban vannak felfűzve a betöltés sorrendjében.

Lapcsere előtt az algoritmus megvizsgálja az R bitet: ha egynek találja, akkor nem veszi ki a lapot, törli az R -t és a mutatót továbblépteti.

Példa: Újabb esély/óra algoritmus



Legrégebben nem használt lap (Least Recently Used, LRU)

Azt a lapot választjuk, amelyre a leghosszabb ideje nem hivatkoztak.

A múltbeli információk alapján próbál előreláttni, az optimális algoritmust közelíteni.

Szimulációs eredmények alapján jó teljesítmény, de nehéz implementálni.

LRU implementáció

Számlálóval

- A lapra történő hivatkozáskor feljegyezzük annak idejét
- A helyettesítendő lap kiválasztáskor a tárolt időpontok közül keressük a legrégebbit.

Láncolt listával

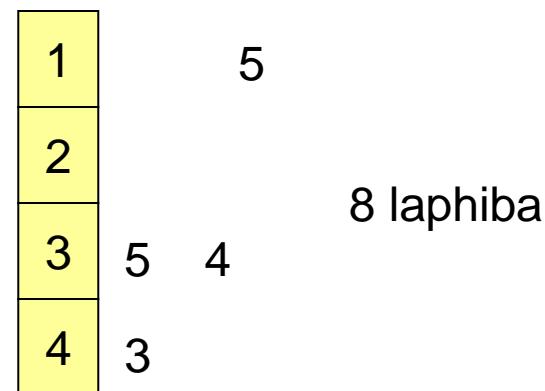
- A lapok egy láncolt listában vannak.
- A frissen behozott lap a lista elejére kerül.
- Hivatkozáskor a lista elejére kerül a lap. A lista végén van a legrégebben nem használt lap.
- Nem kell hosszadalmas keresés.

Példa: LRU

Laphivatkozások (referencia-string):

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

4 keret (folyamatonként ennyi lap lehet egyszerre a memóriában)



Legkevésbé használt lap (Least Frequently Used, LFU)

Leggyakrabban használt lapok a memóriában.

Implementálás:

- Lap hivatkozásakor R bebillentése
- Periodikusan minden lapnál egy számlálót növel, ha $R=1$, majd törli az R -t.
- Lapcsere esetén a legkisebb számláló-értékű lapot dobja ki.

Hátrány:

- A valamikor sokat használt lapok a memóriában maradnak (öregítéssel lehet segíteni ezen)
- A frissen betöltött lapok könnyen kiesnek (frissen behozott lapok befagyasztása, *page locking*)

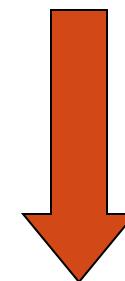
Mostanában nem használt lapok (Not Recently Used , NRU)

Hivatkozott (R) és módosított (M) bitek használata.

OS időközönként törli az R bitet, M bitet viszont őrizni kell (lap törlése információ veszteséghez vezetne).

Négy prioritás

1. nem hivatkozott, nem módosult
2. nem hivatkozott, módosított
3. hivatkozott, nem módosult
4. hivatkozott, módosult



prioritás

A módszer a legkisebb prioritásból választ véletlenszerűen.

Egyéb változatok

Trükkök a hatékonyság növelésére:

- ❑ Szabad lapok fenntartása
 - ❑ Az OS biztosítja, hogy mindenkor legyen szabad lap, amit azonnal odaadhat igény esetén
 - ❑ Az OS a régen nem használt lapokat „ellopja” és a szabad lapok közé helyezi
- ❑ Az OS a módosult lapokat időnként kiírja a háttértárra
 - ❑ így azok nem módosulttá válnak
 - ❑ szükség esetén ezeket gyorsabban ki lehet dobni
- ❑ A módosult lapokat henyélés közben írjuk ki.

Folyamatok lapigénye, lapok allokációja

Folyamat szempontjából az a jó, ha minél több lapja van bent a tárban.

Rendszer szempontjából az a jó, ha minden folyamatnak van elég lapja

Optimális lapszám nehezen határozható meg.

Vergődés (thrashing)

Ha egy folyamat vagy rendszer több időt tölt lapozással, mint amennyit hasznosan dolgozik.

Oka:

- Folyamat: kevés lap van a tárban, gyakran hivatkozik a háttértáron lévőkre
- Rendszer: Túl sok a folyamat, ezek egymás elől lopkodják el a lapokat, mindegyik folyamat vergődni kezd.

Elkerülése: a tárban lévő folyamatoknak biztosítani kell a futáshoz szükséges optimális számú lapot.

Lokalitás

Statisztikai tulajdonság: a folyamatok egy időintervallumban csak címtartományuk szűk részét használják.

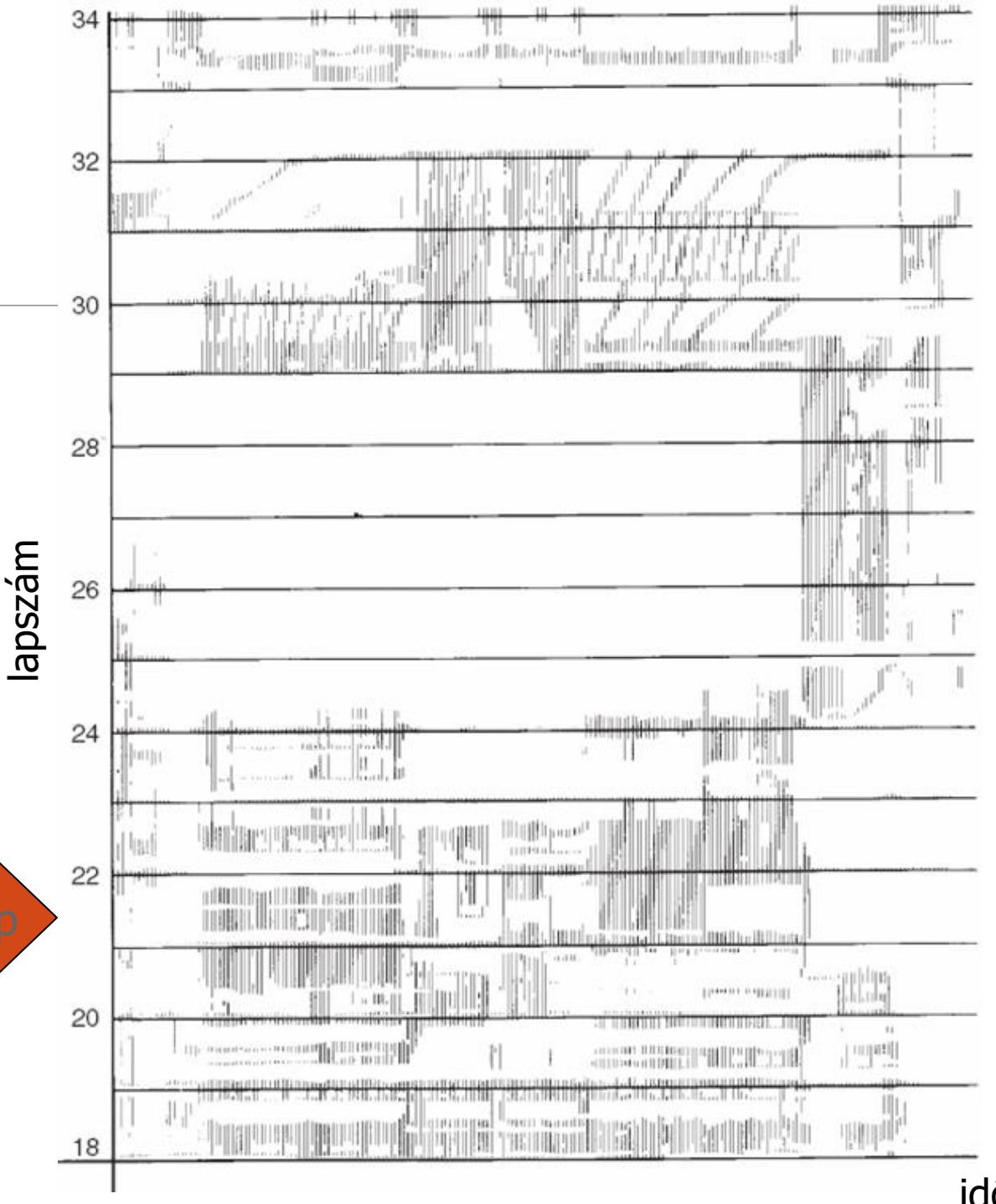
időbeli:

- a hivatkozott címre a közeljövőben nagy valószínűsséggel újra hivatkozni fog (ciklusok, eljárások, verem változók, stb.)

térbeli:

- hivatkozott címek egymás melletti címre történnek (soros kód, tömbkezelés).

Lokalitás



A munkahalmaz modell

A munkahalmaz (*working set*, WS) az elmúlt Δ időben (munkahalmaz ablak) hivatkozott lapok halmaza.

A lokalitás miatt a folyamatnak nagy valószínűsséggel ezekre a lapokra lesz szüksége a közeljövőben.

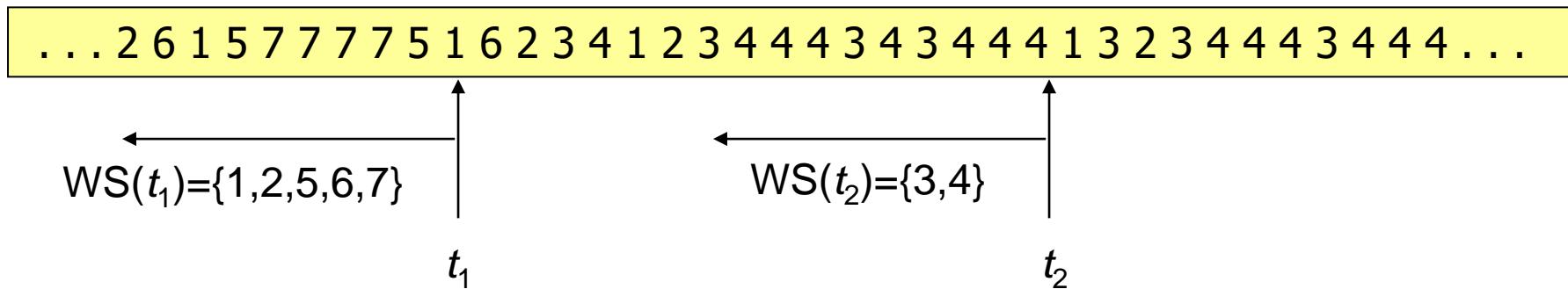
Munkahalmaz mérete folyamatonként és időben is változik (lásd a következő példát).

OS célja minden aktív folyamat számára biztosítani a munkahalmazt.

Pontos méréséhez bonyolult hardver kellene, ezért fix időintervallumonkénti mintavételezéssel oldják meg.

Munkahalmaz mérése

Példa ($\Delta = 10$):



Mérése (példa):

A WS mérete: $\Delta=10000$.

Újabb referencia bitek használata (pl. laponként 2 bit)

Időközönként (pl. 5000 laphivatkozásoknál) a referenciabitek vizsgálata/törlése.

- Ha valamely bit 1, akkor a lapra történt hivatkozás az utóbbi 5000, vagy az előtte lévő 5000 laphivatkozás alatt → lap része a WS-nek.

A munkahalmaz mérete

Fontos kérdés: mekkora legyen Δ ?

Ha túl kicsi:

- nem tartalmazza az egész lokalitást

Ha túl nagy:

- több lokalitást is tartalmaz (feleslegesen)

A munkahalmaz és a vergődés kapcsolata

WSS_i : az i-ik folyamat munkahalmazának mérete

- A példában: $WSS(t_1)=5$, $WSS(t_2)=2$

A rendszer teljes lapigénye: $D=\sum WSS_i$

A rendszerbeli lapkeretek száma: m

Vergődés akkor lép fel, ha $D>m$

- (vagyis a lokalitások összes mérete nagyobb, mint a rendelkezésre álló memória mérete)

Laphiba gyakoriság figyelése

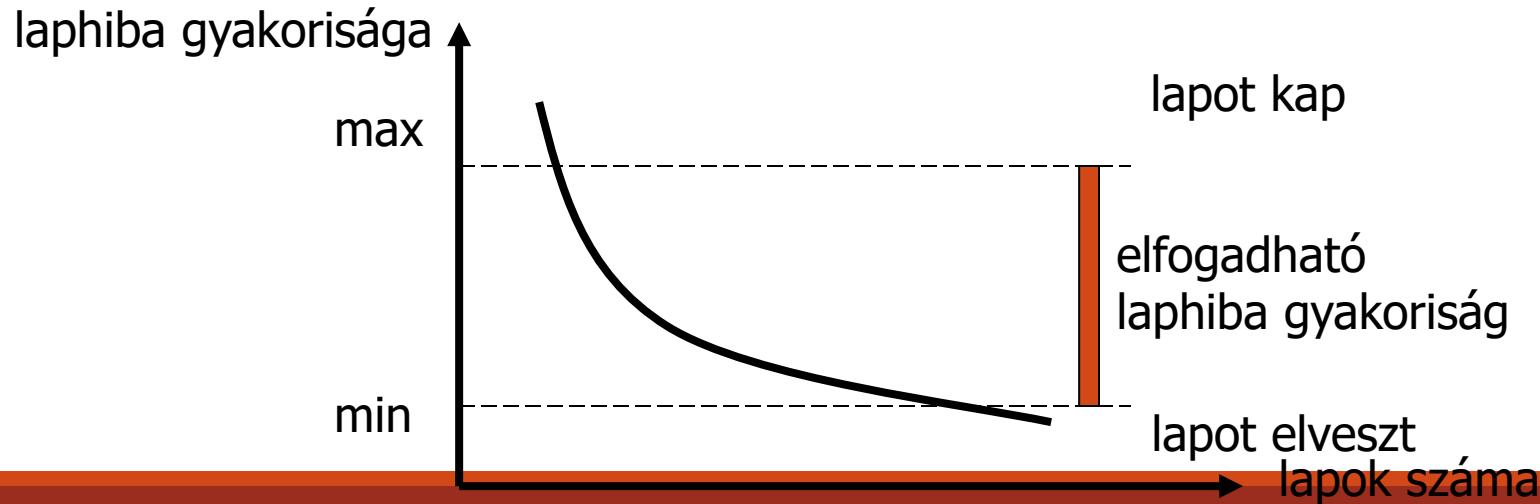
A munkahalmaz modellnél egyszerűbb módszer a vergődés elkerülésére

Stratégia:

- ha folyamatnak sok a laphibája, akkor újabb lapot kell adni neki,
- ha kevés, akkor más folyamatok rovására túl sok tartozik hozzá.

Az OS a laphiba gyakoriságát vizsgálja.

- Ha a laphibák gyakorisága a maximum felett van, akkor lapot kap
- Ha a laphibák gyakorisága a minimum alatt van, akkor lapot veszünk tőle
- Csak akkor indítunk el újabb folyamatot, ha van elég szabad lap.
- Szükség esetén folyamatokat fel lehet függeszteni, lapjait ki lehet osztani.



Egyéb tervezési szempontok

A legfontosabb tervezési szempontok:

- Lapcsere stratégia kiválasztása ✓
- Lapok allokációja ✓

Egyéb fontos szempontok:

- Előre lapozás
- Lapok mérete
- Asszociációs memória fedése
- Programozási trükkök
- Lapok befagyasztása
- COW

Előre lapozás

Folyamat indításakor, illetve aktiválásakor az összes lapja a háttértáron van.

Érdemes a várható munkahalmazt behozni a tárba.

Akkor hasznos, ha az előre lapozás találati aránya nagy, különben pazarlás.

Példa:

- munkahalmaz modell használata esetén a felfüggesztett folyamat teljes munkahalmazát aktiválás esetén előre be lehet lapozni.

A lapméret hatása

Lapméret (tipikusan: 4kByte – 4Mbyte)

Lap méretének növelése:

- laptábla mérete csökken
- perifériás átviteli idő csökken (nagyobb blokkot relatíve gyorsabb átvinni, mert kevesebb az keresési idő/adminisztráció)

Lap méretének csökkenése:

- a lokalitás jobban érvényre jut (kisebb a munkahalmaz)
- kisebb belső tördelődés

Az asszociációs memória fedése

Az asszociációs memória (TLB) találati aránya fontos jellemző (lsd. múlt óra)

Másik fontos jellemző a TLB *fedése*. Ez azt mutatja meg, hogy a TLB-n keresztül mekkora memóriát lehet elérni.

A TLB fedés értéke nyilvánvalóan:

TLB bejegyzések száma x lapméret

Programozási trükkök

Programírás

- több dimenziós tömbök tárbeli elhelyezésének megfelelő bejárása
- egyszerre használt változó egymás mellé
- egymást hívó eljárások egymás mellé
- célszerű kerülni a nagy ugrásokat (lokalitást mutató adatszerkezetek preferálása)

Fordítás

- eljárások egymás mellé helyezése
- kód és az adatterület szétválasztása (kód nem változik, a lapcserénél nem kell kiírni)

Példa: a programstruktúra hatása

A laptárat legyen 128 bájt.

```
int[128,128] data;
```

- A fordító soronként tárol: data[0][0], data[0][1], ..., data[0][127], data[1][0], data[1][1], ..., data[127][127]
→ minden sor egy lapon tárolódik

1. program

```
for(j = 0; j < 128; j++)
  for(i = 0; i < 128; i++)
    data[i,j] = 0;
```

128 x 128 =
16,384 laphiba

2. program

```
for(i = 0; i < 128; i++)
  for(j = 0; j < 128; j++)
    data[i,j] = 0;
```

128 laphiba

Lapok tárba fagyasztása

Az elindított perifériás műveleteknél a kijelölt címtartományt az átvitel idejére érdemes befagyasztani.

A beemelt lapokat az első hivatkozásig érdemes befagyasztani.

Tipikus példák:

- OS magjának befagyasztása
- I/O lapok befagyasztása

Megoldás módja:

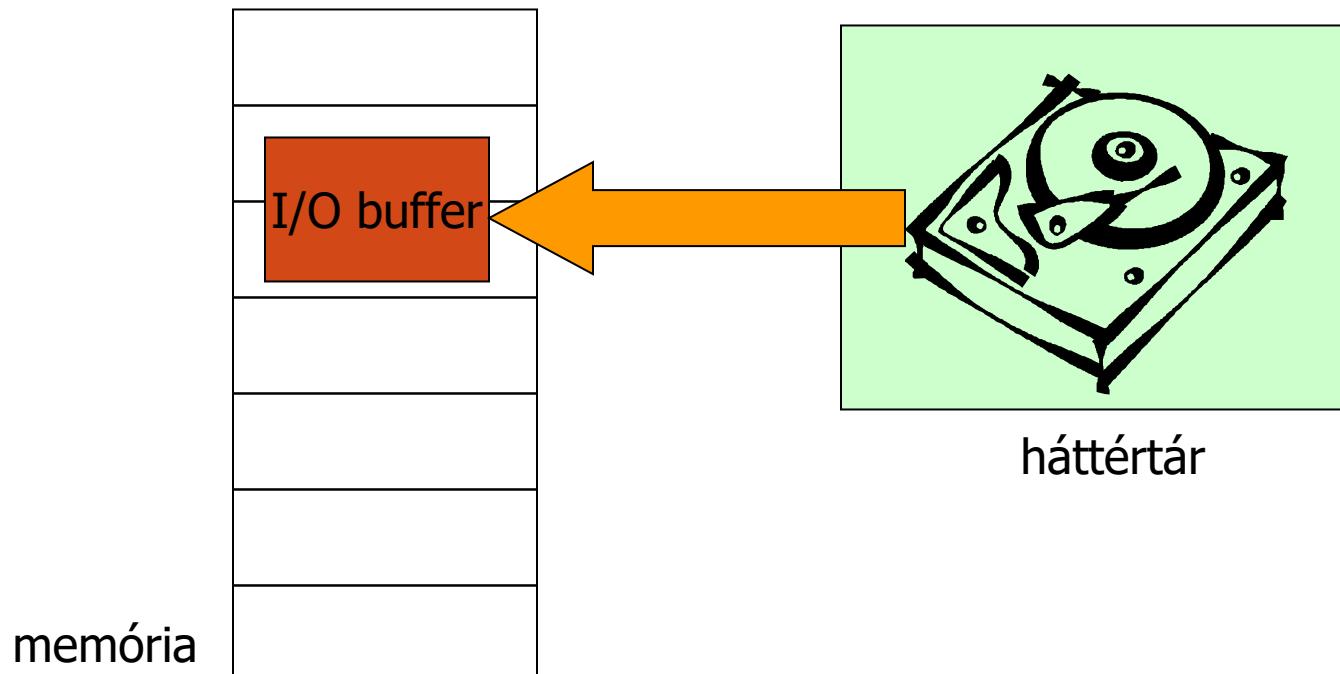
- minden laphoz „lock” bit

Példa: tárba fagyasztás I/O alatt

Háttértárról beolvasás pufferbe (segédprocesszor)

Ha kész, a CPU megszakítást kap

Művelet közben a lapnak a memóriában kell lenni!



Copy on write (COW)

Folyamatok indításának hatékony módszere:

A gyermek és szülő folyamatok kezdetben azonos memóriaterületet használnak

Ha a közös lapot valamely folyamat módosítja, akkor történik meg a lap másolása

Áttekintés

Mikor foglalkozik az OS lapkezeléssel?

1. Folyamat létrehozása
 - Programméret meghatározása
 - Laptábla létrehozása
2. Folyamat létrehozása esetén
 - MMU reset
 - TLB kiürítése
3. Laphiba
 - A hibát okozó virtuális cím meghatározása
 - Szükséges lap behozatala
 - Lap kivitele (felszabadítás),
4. Folyamat terminálása
 - A laptábla és a lapok felszabadítása

Esettanulmányok

Windows

Linux

Windows

Igény szerinti lapozás *klaszterezéssel* módosítva. A klaszterezés során a laphibát okozó lapot és annak környezetét olvassuk be.

Minden folyamatnak van két paramétere:

- *working set minimum*
- *working set maximum*

A *working set minimum* a memóriában tartott lapok minimális, garantált számát tartalmazza (pl. 50).

Egy folyamat kaphat a *working set maximum* által meghatározott számú lapot (pl. 345).

Ha a rendszer szabad memóriája egy küszöbérték alá csökken, akkor egy automatikus munkahalmaz vágás (automatic working set trimming) hajtódik végre.

Az automatikus munkahalmaz vágás során a folyamatuktól elvesszük a *working set minimum* feletti lapokat.

Lapcsere algoritmus:

- Egyprocesszoros x86 típusú rendszereken: óra algoritmus.
- Alpha, többprocesszoros x86: módosított FIFO

Linux

Szabad lapok listája, ebből ad laphiba esetén.

kswapd: page démon. Ellenőrzi, hogy van-e elég szabad lap (1/sec).

- Kevés szabad lap esetén felszabadításba kezd, egyre fokozódó intenzitással (először a paging cache-ből, majd a kevéssé használt osztott memóriából, végül a felhasználóktól).
- Óra-algoritmus variációja, de nem FIFO szerint keres, hanem virtuális cím szerint (lokálitás elve!).
- Kidobandó lap kezelése:
 - „tiszta”: azonnal kidob
 - „piszkos” és van már háttértáron korábbi másolata: kiírásra ütemez
 - „piszkos” és nincs a háttértáron korábbi másolata: paging cache-be kerül

bdflush: periodikusan ellenőrzi, hogy a lapok mekkora része piszkos. Ha túl sok, elkezdi kiírni őket.

Operációs rendszerek

9. HÁTTÉRTÁRAK

Felhasznált irodalom:

- Kóczy-Kondorosi (szerk.): Operációs rendszerek mérnöki megközelítésben
- Tanenbaum: Modern Operating Systems
- Silberschatz, Galvin, Gagne: Operating System Concepts

A háttértár kezelése

Bevezetés

Háttértárak típusai

A lemezegység fizikai szervezése

Lemezműveletek ütemezése

- Fejmozgás optimalizálása
- Elfordulás optimalizálása

SSD működése

Az adattárolás megbízhatósága

Miért van szükség háttértárra?

központi tár drága és kicsi a tárolókapacitása

a kikapcsolással az információk elvesznek a központi tárban

Háttértárak típusai

Mágneslemez (merev, hajlékony)

Mágnesszalag

Magneto-optikai lemez

Optikai lemez

Flash (SSD)

A jövő (?):

- Holografikus tárolás
- MEMS tárolók

Legelterjedtebbek a mágneslemez és az SSD

Mágneslemez

Vékony mágnes-réteggel bevont lemezek (merev- vagy hajlékony lemez)

Fej vékony levegőrétegen fut a lemez felett

Hajlékony: ~1MB

Merev: >1TB



Mágnesszalag

Nagy adatmennyiségek tárolására.

Nincs véletlen hozzáférés, lassú a keresés, de az átvitel a lemezekéhez hasonló sebességű.

Főleg mentési, archiválási célokra.

Gazdaságos.





Magneto-optikai lemez

Merev, mágneses réteggel bevont lemez. Üveg védőréteg.

Fej távolabb van, mint a mágneses lemeznél.

Rögzítés: Lézerrel felmelegítikenek egy pontot (1 bit), amelyre a gyenge mágneses tér most már rögzíti az információt.

Olvasás: Kerr-effektussal. Lézerrel megvilágítjuk a mágneses pontot (1 bit), amelyről visszaverődő lézersugár polarizációja tartalmazza az információt.

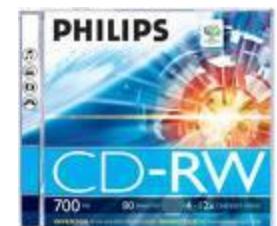
Írható/olvasható optikai lemez

CD-RW, DVD-RW

A lemez anyaga vagy kristályos (áttetsző), vagy amorf (opálos) lehet. Alatta fényvisszaverő réteg.

Írás/olvasás: lézersugárral

- Alacsony energia: olvasás visszaverődött fény mennyiségét érzékeli
- Közepes energia: törlés olvaszt és kristályos állapotba szilárdít
- Nagy energia: írás amorf állapotba olvaszt



Egyszer írható optikai lemez

CD-R, DVD-R

WORM: write-once, read-many-times

Vékony alumíniumfólia üveg védőréteg alatt

Írás: lézerrel „lyuk” égetése a fóliába

Olvasás: visszaverődés mérése

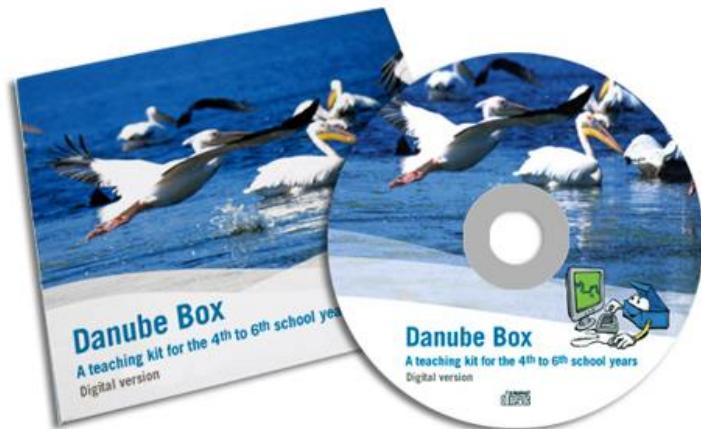


Csak olvasható optikai lemez

CD-ROM, DVD-ROM

Működés hasonló a WORM lemezekéhez

Az írást nem lézerrel, hanem nyomtatással állítják elő



Flash memória



Egy speciális EEPROM

Információ egy MOSFET szigetelt elektródáján (GATE1) tárolt töltés formájában tárolódik

Kiolvasás: GATE2-re adott vezérlő (olvasó) feszültség hatását módosítja a GATE1-en tárolt töltés → a DS áram értékéből → 0 vagy 1 bit. (Léteznek több bites eszközök is!)

Írás-törlés:

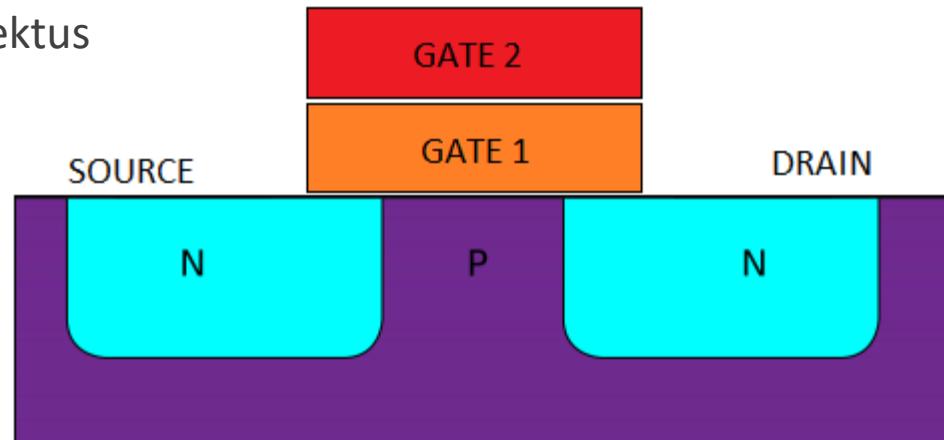
- nagy feszültség (10-15V) → alagút effektus

Törlés: blokkosan (pl. 2MB)

Írás/olvasás: lap (pl. 8kB)

Korlátozott élettartam:

- néhány millió írás/törlés



Holografikus rögzítés

Kísérleti stádiumban van

Hologram tekinthető egy 3D-s mátrixnak

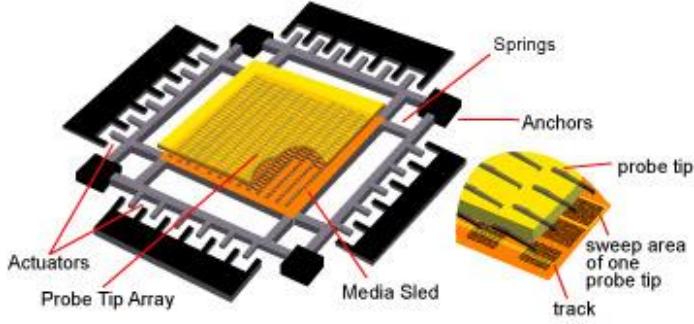
Minden pixel egy bit (fekete-fehér)

Nagyon gyors lehet: egy lézervillanással egy egész kép (sok bit) rögzíthető

Jelenleg max. 5TB/lemez



MEMS tárolás



MEMS: micro-electronic mechanical systems

Több ezer kis író/olvasó fej, felettük 1cm² mágneses tároló felület

A felület mozgatható a fejek felett, így a fejek

- elérhetik az információt,
- a mozgatás alatt pedig írni és olvasni tudnak.

A lemezegység fizikai szervezése

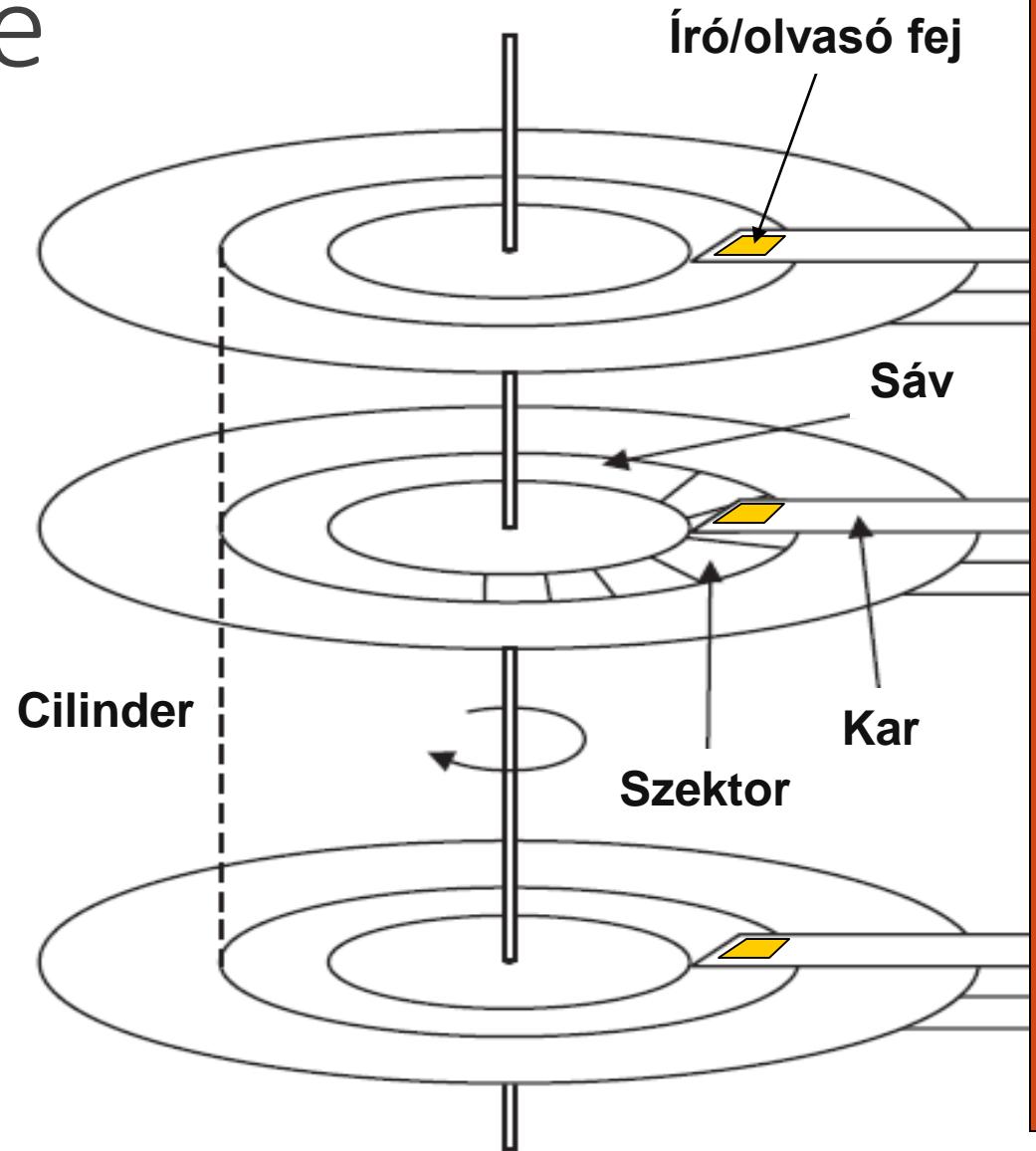
Mágneses bevonatú forgó korongok, a felület felett író-olvasó fej.

Sáv (track) a lemezterület (gyűrű) azon része, amelyet a fej elmozdulás nélkül egy fordulat alatt elér.

Cilinder (cylinder) az összes fej alatti sáv

Szektor (sector), a sáv azonos méretű blokkokra osztva. Az információátvitel legkisebb egysége: a lemezvezérlő egy teljes szektort olvas vagy ír.

A lemezegység fizikai szervezése



Szektorok címzése

0. szektor: a legkülső cilinder első sávjának első szektora.

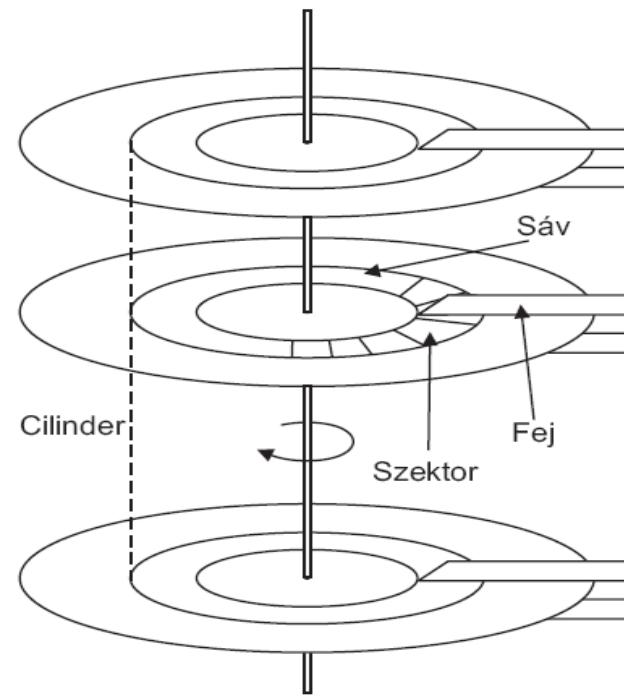
1., 2., stb. szektorok: ugyanezen a sávon egymás után következő szektorok, míg el nem fogy a sáv.

Ezután ugyanezen a cilinderen belül a következő sáv következik (a következő fej alatt), amíg el nem fogy az összes sáv a cilinderen belül.

Ezután a következő cilinder következik.

OS: lineárisan címez

Lemezillesztő: fordít → cilinder, fej, szektor



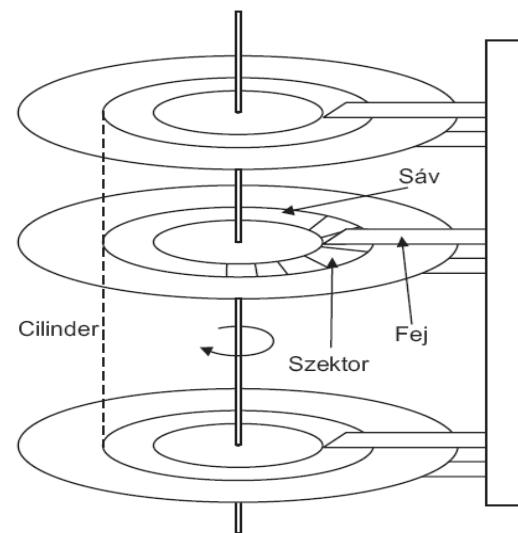
További fogalmak

Az átvitel kiszolgálásának ideje

- fejmozgási idő (**seek time**): az az idő, amely alatt a fej a kívánt sávra (cilinderre) áll
- elfordulási idő (**latency time**): az az idő amely alatt a kívánt szektor a fej alá fordul
- az információ átviteli ideje (**transfer time**)

Az idők között nagyságrendi különbségek vannak:

a fejmozgási idő a leghosszabb.



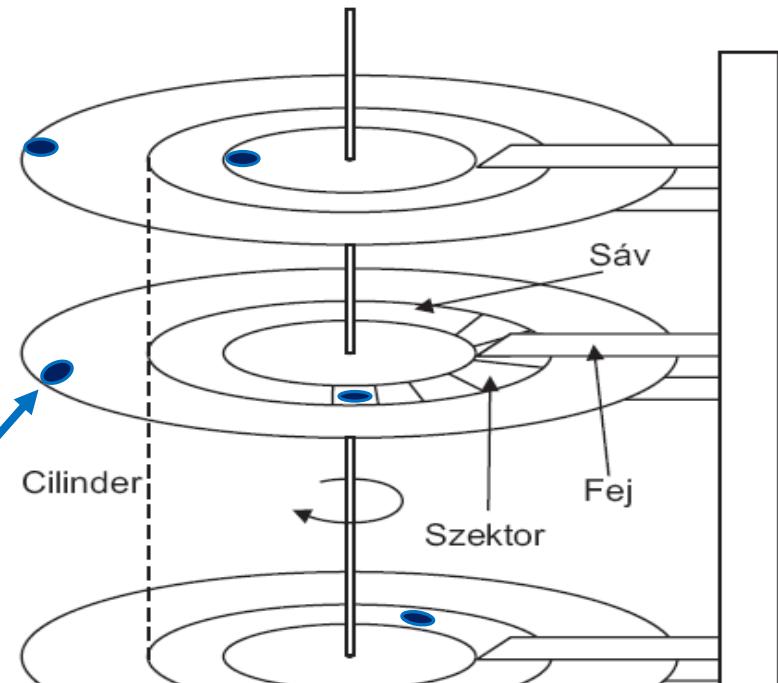
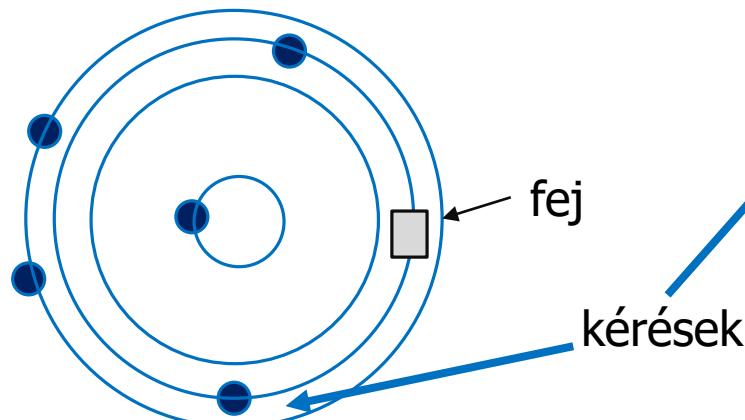
A lemezműveletek ütemezése

Egyszerre több folyamat verseng a háttértár perifériáért. Egyszerre több kérés várakozhat kiszolgálásra. Cél az átlagos seek és a latency idő csökkentése.

Természetesen így egyes folyamatok rosszabbul járnak, de a cél a *globális* teljesítmény növelése.

A kérések sorban jönnek

De mi legyen a kiszolgálás sorrendje?



A lemezműveletek ütemezése

Legrövidebb fejmozgási idő:

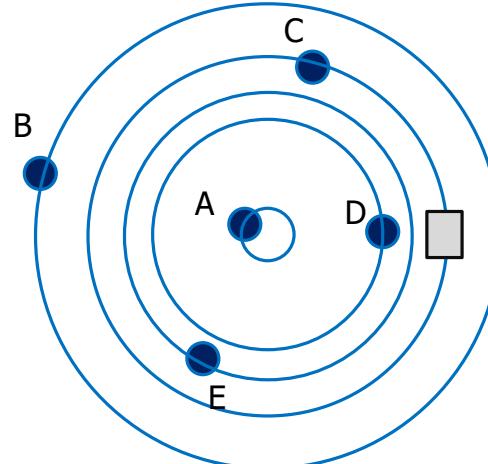
- a következő kiszolgált kérés az lesz, amelyiknek a legközelebb van a cilindere

SCAN:

- A fej egy irányban mozog, ezen sorban szolgálja ki a kéréseket, majd irányváltás

N-SCAN:

- Mint SCAN, de egyszerre csak N kérést szolgál ki



Sorrendi kiszolgálás: A,B,C,D,E

Legrövidebb fejm.: C, E, D, B, A

SCAN: C,B (fordul), E,D,A

N-SCAN (N=3): C,B (for.), A, (ford.) D, E

A lemezműveletek ütemezése

Legrövidebb fejmozgási idő:

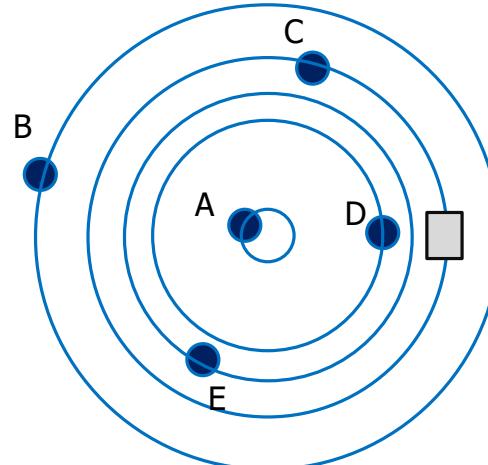
- a következő kiszolgált kérés az lesz, amelyiknek a legközelebb van a cilindere

SCAN:

- A fej egy irányban mozog, ezen sorban szolgálja ki a kéréseket, majd irányváltás

N-SCAN:

- Mint SCAN, de egyszerre csak N kérést szolgál ki



Sorrendi kiszolgálás: A,B,C,D,E

Legrövidebb fejm.: C, E, D, B, A

SCAN: C,B (fordul), E,D,A

N-SCAN (N=3): C,B (for.), A, (ford.) D, E

A lemezműveletek ütemezése

Legrövidebb fejmozgási idő:

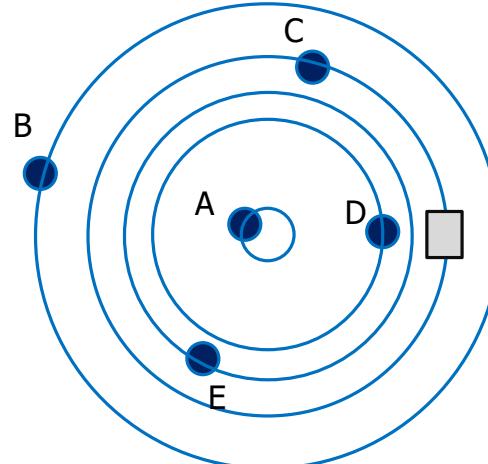
- a következő kiszolgált kérés az lesz, amelyiknek a legközelebb van a cilindere

SCAN:

- A fej egy irányban mozog, ezen sorban szolgálja ki a kéréseket, majd irányváltás

N-SCAN:

- Mint SCAN, de egyszerre csak N kérést szolgál ki



Sorrendi kiszolgálás: A,B,C,D,E

Legrövidebb fejm.: C, E, D, B, A

SCAN: C,B (fordul), E,D,A

N-SCAN (N=3): C,B (for.), A, (ford.) D, E

A lemezműveletek ütemezése

Legrövidebb fejmozgási idő:

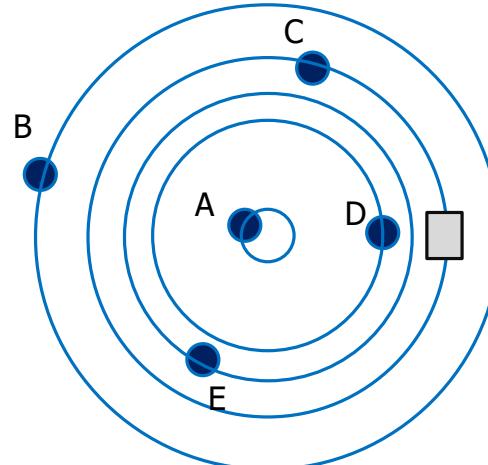
- a következő kiszolgált kérés az lesz, amelyiknek a legközelebb van a cilindere

SCAN:

- A fej egy irányban mozog, ezen sorban szolgálja ki a kéréseket, majd irányváltás

N-SCAN:

- Mint SCAN, de egyszerre csak N kérést szolgál ki



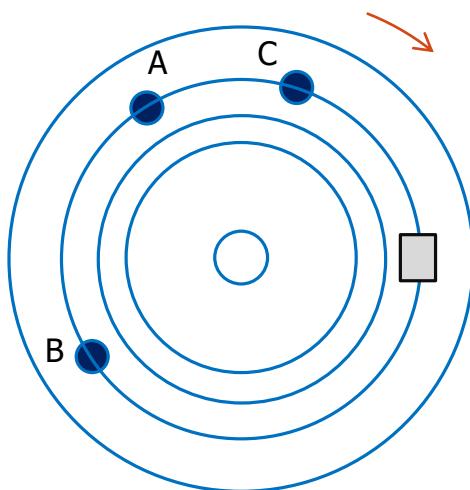
Sorrendi kiszolgálás: A,B,C,D,E
Legrövidebb fejm.: C, E, D, B, A
SCAN: C,B (fordul), E,D,A
N-SCAN (N=3): C,B (for.), A, (ford.) D, E

Az elfordulás optimalizálása

Cél: a seek time csökkentése

Megoldás:

- Az egy cilinderen belüli kérések a lemez aktuális pozíójának, valamint a szektorok sorrendjének ismeretében a kiszolgálás előtt sorba rendezhetők.



Sorrendi kiszolgálás: A,B,C
Optimális: C, A, B

Egyéb szervezési elvek a teljesítmény növelésére 1.

Lemezterület tömörítése (Disc Compaction)

- lokalitáson alapul
- Az egymáshoz tartozó blokkokat a lemezen is egymás mellé tesszük. Időnként egy rendezőprogrammal tömöríteni kell a háttértárat.

A gyakran szükséges adatok a lemez közepén

Gyakran használt adatok több példányban

- több cilinderen is tároljuk
- így minden fejálláshoz elég közel van
- Csak nagyon ritkán változó adatokkal érdemes csinálni:
 - adatok konzisztenciája
 - kölcsönös kizárási

Egyéb szervezési elvek a teljesítmény növelésére 2.

Több blokk átvitele egyszerre

- Az idő nagy része a fejmozgással telik → ha már ott vagyunk, vigyünk át minél több blokkot

Blokkok átmeneti tárolása:

Disc Cache (periférián lévő, központi tár)

- write through, egyidejűleg kiírjuk a lemezre
- copy back, csak akkor írjuk ki ha az átmeneti tárra szükségünk van (teljesítmény növelő, de meghibásodás esetén a módosítások nem kerülnek a lemezre)

Adattömörítési eljárások használata (Data Compression)

- Az lemezen az információ tömörített formában van. A ki-be tömörítést a perifériakezelő vagy célhardver végzi.
- Átvitel sebessége nő, adatvesztés valószínűsége nagyobb

SSD (Solid State Drive)



Nincs benne mozgó alkatrész

Sokkal gyorsabb, mint a HDD

Összehasonlítás:

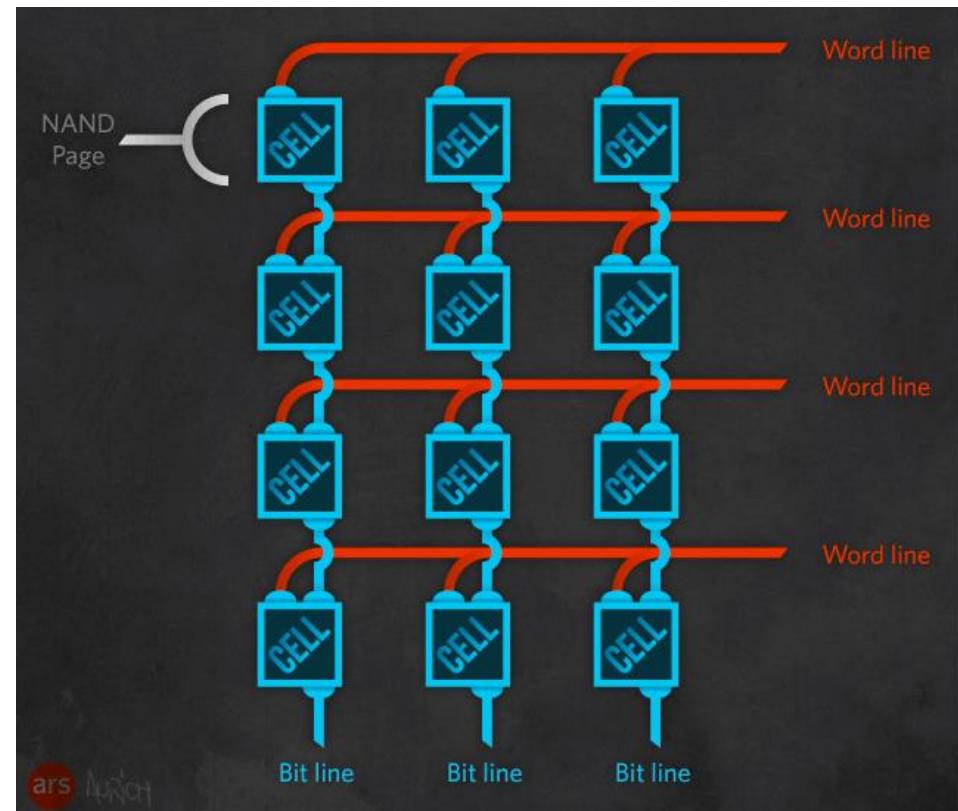
	SSD	HDD
Seek time	0	10 ms
Írási idő	250 μ s	5 ms
Olvasási idő	25 μ s	5 ms
Törlési idő:	1.5 ms	-
Írási ciklusok száma	néhány millió	-
Archiválás	✗	✓

SSD (Solid State Drive)

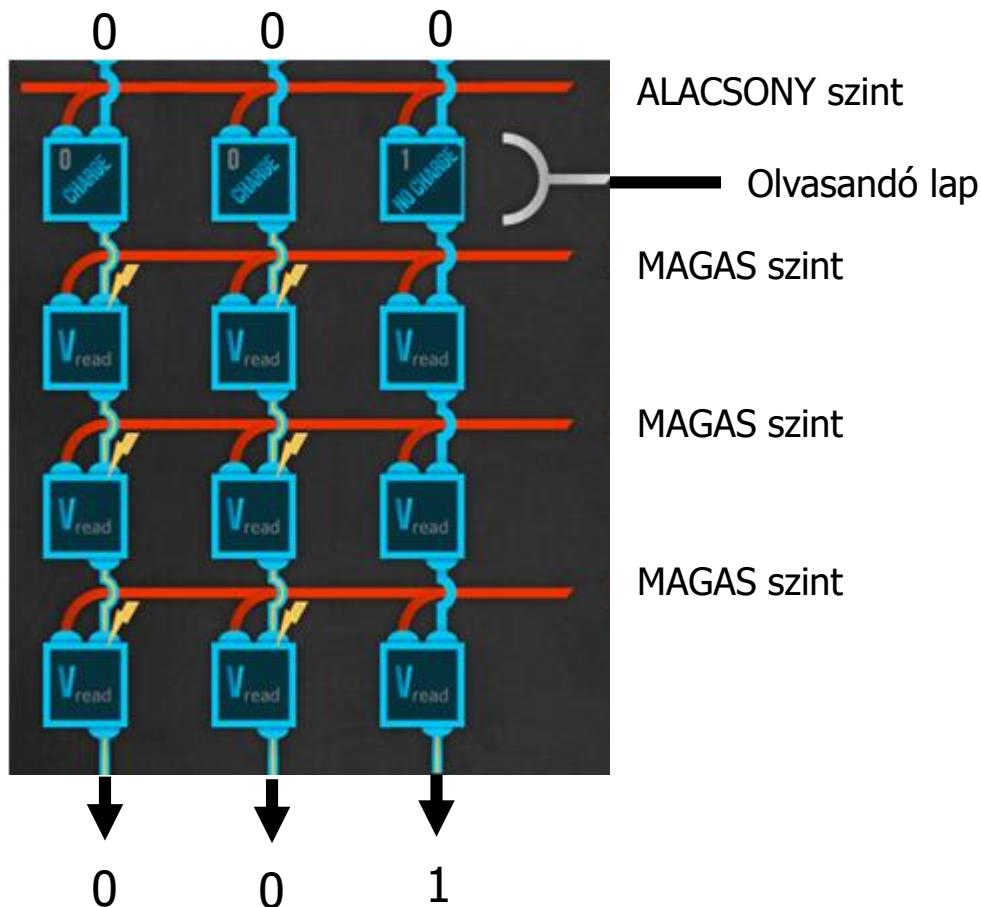


A memóriacellákat rácsos szerkezetbe rendezik

- Egy sor: lap
- A teljes rács: blokk (pl. 256 lap)
- Törlés: a teljes blokk
- olvasás: 1 lap
- írás: 1 lap (csak törölt lap írható!)
- → SSD kontroller
- 1 memóriacella tárolhat:
 - 1 bitet (SLC - *Single Level Cell*)
 - 2 bitet (MLC - *Multi Level Cell*)
 - 3 bitet (TLC – *Triple Level Cell*)
 - 4 bitet (QLC – *Quad Level Cell*)



SSD olvasása



CHARGE: FET vezet (0)

NO CHARGE: FET nem vezet (1)

Olvasás:

Kiválasztott lap *word line* → 0

Többi lap *word line* → magas

- ezek a FET-ek vezetni fognak

SSD írása

Írni csak teljes lapot lehet, üres helyre

Ha egy lapon belül írunk:

- Lap beolvasása memóriába
- Lap módosítása (memóriában)
- Lap visszaírása *üres* helyre
- Régi lap törlésre kijelölése (STALE)

Törölni csak teljes BLOKKOT lehet

Lap törlés helyett:

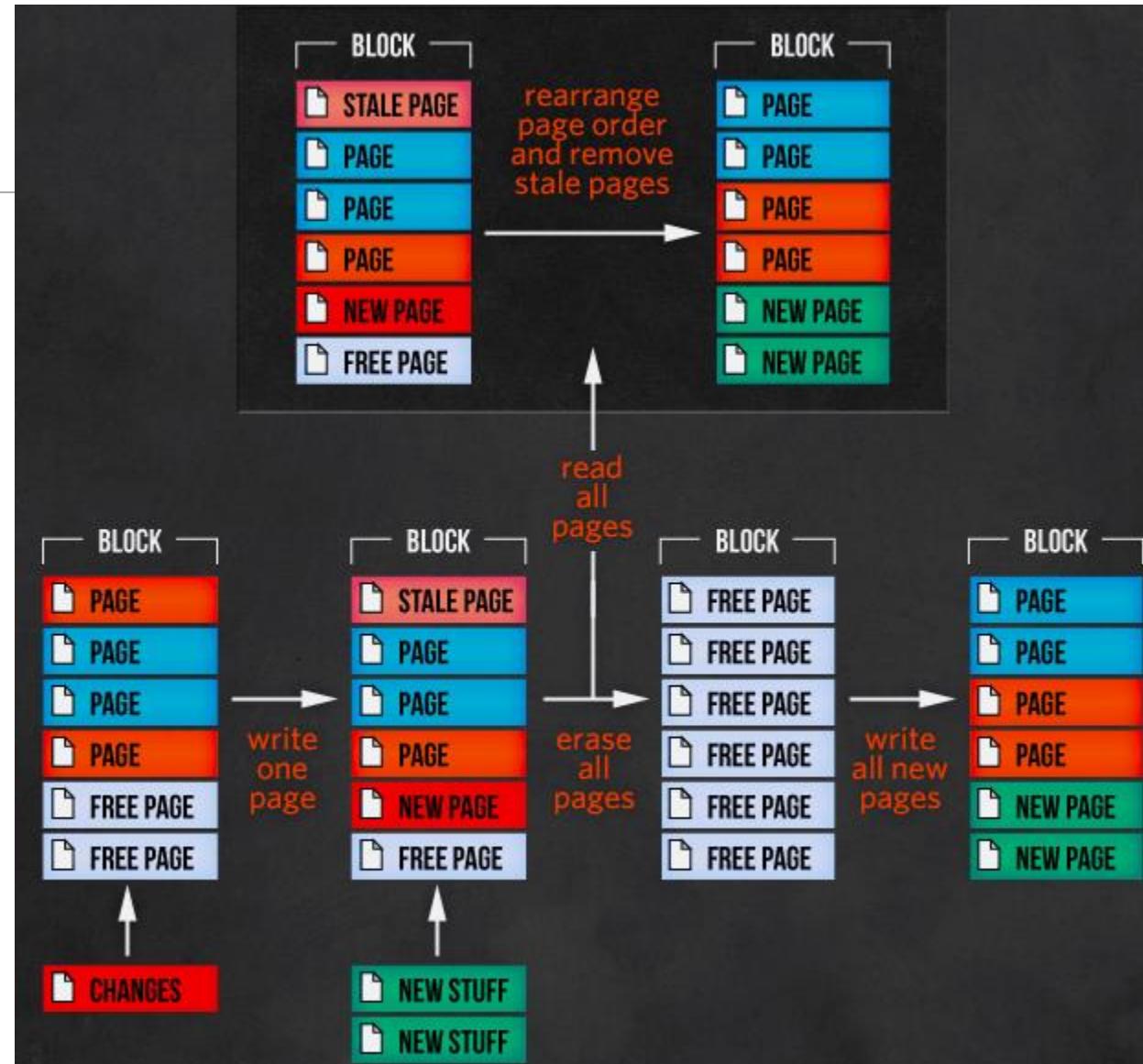
- Megjelölés: érvénytelen (STALE)
- Fizikai törlés később, a teljes blokkal együtt

	A	B	C
	D	free	free
Block X	free	free	free
	free	free	free
Block Y	free	free	free
	free	free	free
	free	free	free
	free	free	free

	A	B	C
	D	E	F
Block X	G	H	A'
	B'	C'	D'
Block Y	free	free	free
	free	free	free
	free	free	free
	free	free	free

SSD írása

Példa:



SSD karbantartása

Minden törlés/írás elhasználódáshoz vezet

A használatot igyekszünk egyenletesen elosztani a teljes eszközre

Terheléselosztás: wear leveling

- Élettartamot növeli
- Az SSD kontroller automatikusan végzi

- SSD kontroller:
 - Nagyon komplex eszköz
 - A gyártók védik algoritmusaikat

RAID

Redundant Array of Inexpensive/Independent Disks

Cél:

- Adatátvitel sebességét növelni
- Adattárolás biztonságát növelni

Alapötletek:

- Lemezegységek kétszerezés (disc shadowing, mirroring)
Az írásokat minden két egységen elvégezzük, hiba esetén a másik példány használható.
→ megbízhatóság nő, sebesség nem változik
- Tároljuk egy adatbájt bitjeit külön tárolókon. Párhuzamos hozzáférés.
→ megbízhatóság kissé csökken, de a sebesség kb. 8x

Megvalósítás: RAID 0-6

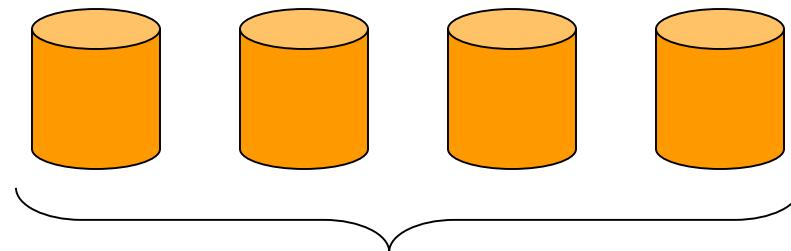
RAID 0

Non-redundant stripping

Az egymás utáni blokkok külön háttértárakon helyezkednek el, nincs redundancia (tükör, vagy paritás)

Adatátvitel sebessége megnő.

Biztonság némileg csökken (több kisebb tároló közül gyakrabban hibásodik meg egy, mint egyetlen nagyobb kapacitású tároló)



Adat (blokkos szervezés)

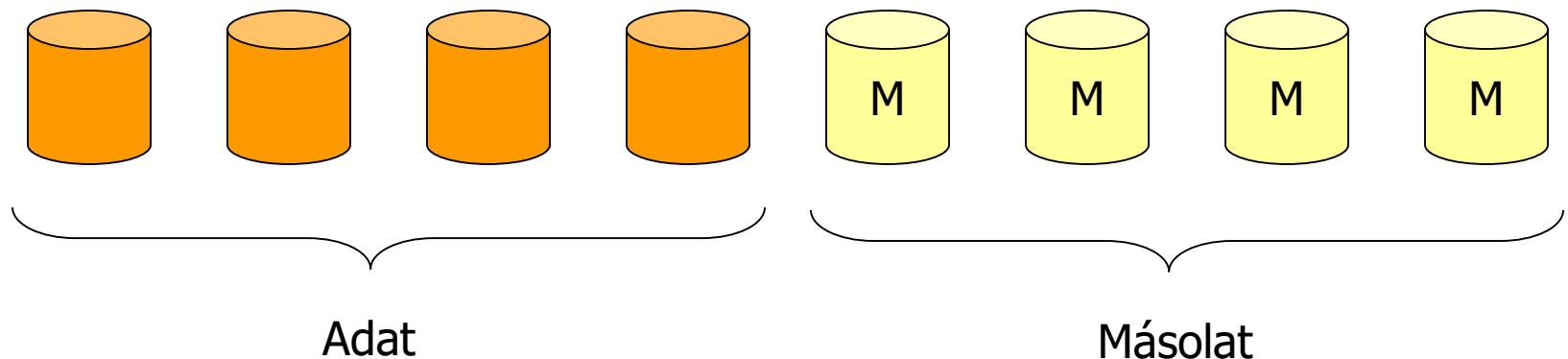
RAID 1

Disk mirroring

Minden háttértárnak van egy tükre.

Hiba esetén a tükrön lévő adat használható, a hibás egység erről helyreállítható

Sokkal nagyobb biztonság, de a sebesség nem nő.



RAID 2

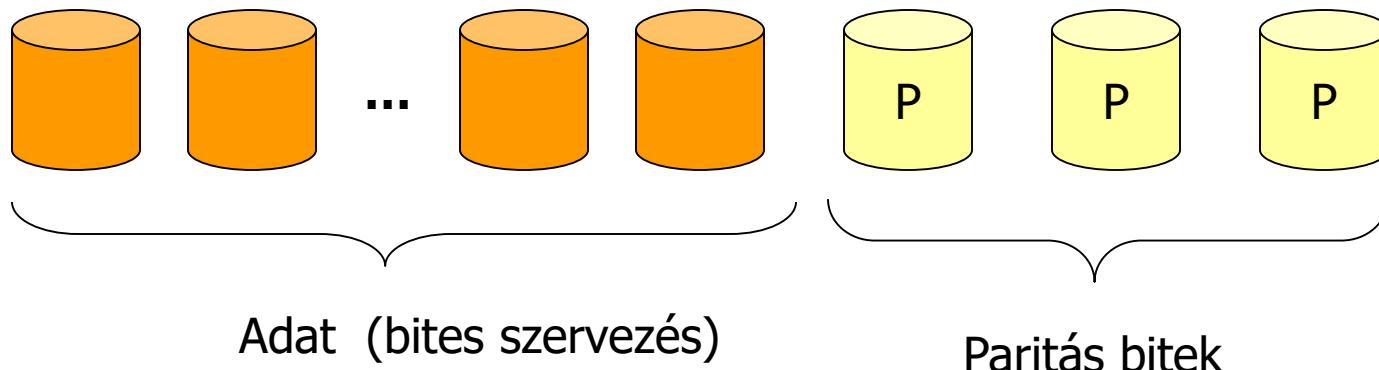
Memory-style error correction

Minden bitet más tároló tárol

A memóriákhoz hasonlóan paritás biteket használunk.

1 bit hiba a paritás bitekből javítható

Nagy sávszélesség és nagy biztonság kisebb redundanciával, mint RAID 1 esetén.



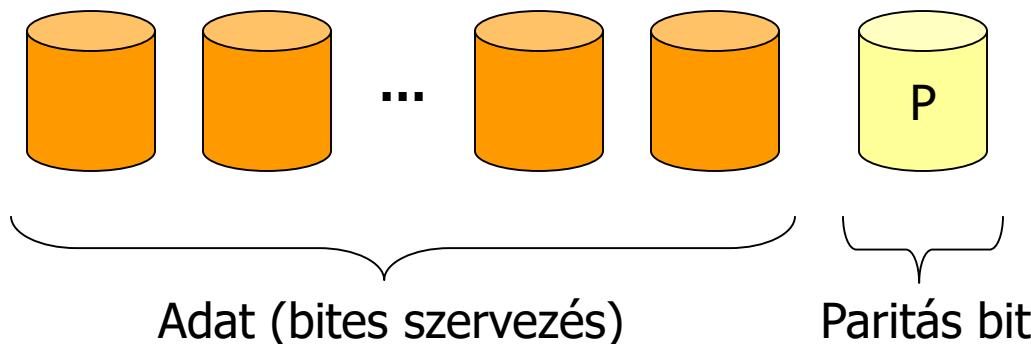
RAID 3

Bit-interleaved parity

RAID-2 továbbfejlesztése

Ötlet: ez nem memória! Hiba detektálása minden háttértáron önállóan működik → elég 1 paritás bit a javításhoz

Nagy sávszélesség és nagy biztonság nagyon kis redundanciával (egyetlen extra háttértárral).



RAID 4

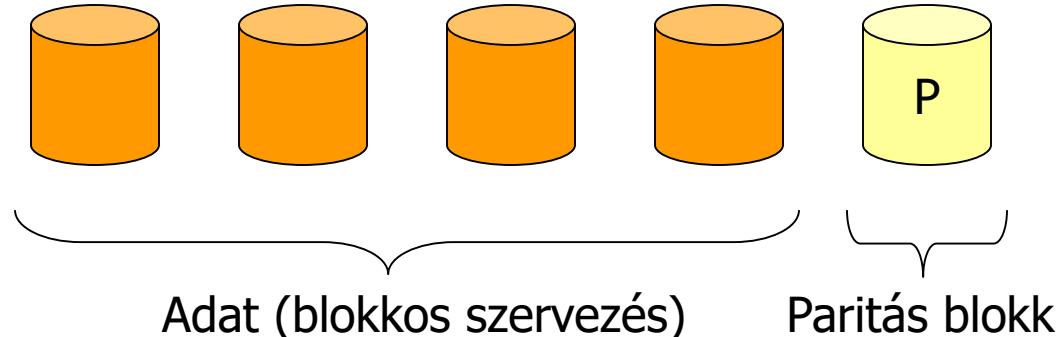
Block-interleaved parity

Mint RAID 3, de blokkos szervezés

Az egyik háttértár paritás-blokkokat tartalmaz

Biztonságos, nagy adathalmaz kezelésénél nő az átviteli sávszélesség is.

Kis adatmérétek esetén (pl. 1 blokk) nem nő a sávszélesség, (Sőt: 1 blokk írása → P olvasása, adatblokk írása, P írása)

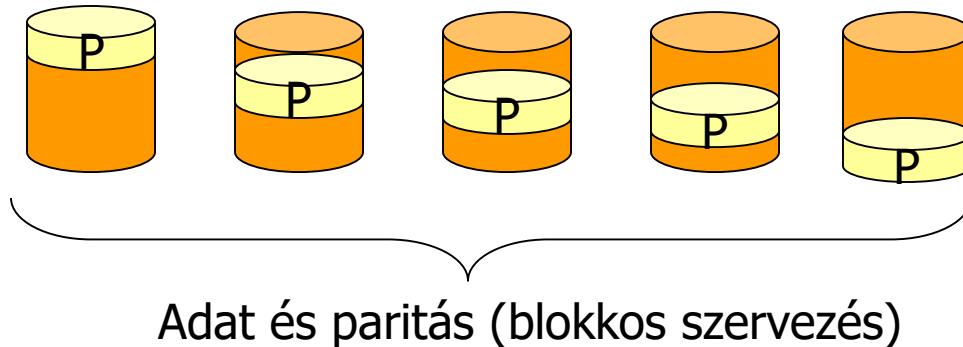


RAID 5

Block-interleaved distributed parity

Mint RAID 4, de a paritás blokkok elosztva

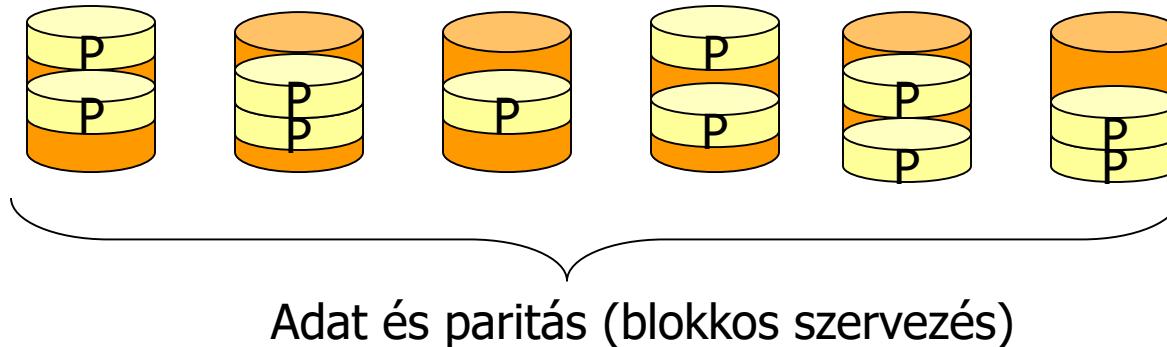
Ok: a RAID 4 esetén a paritás tárolót aránytalanul sokat használjuk → hamarabb meghibásodik. Itt a terhelés kiegyenlítődik



RAID 6

P+Q redundancy scheme

Mint RAID 5, de több (a példában 2) paritás blokkot tartalmaz
Véd a többszörös diszkhibák ellen is.



Operációs rendszerek

10. ÁLLOMÁNYOK

Felhasznált irodalom:

- Kóczy-Kondorosi (szerk.): **Operációs rendszerek mérnöki megközelítésben**
- Tanenbaum: **Modern Operating Systems**
- Silberschatz, Galvin, Gagne: **Operating System Concepts**

Tartalom

Alapfogalmak

Szabad blokkok nyilvántartása

A lemez blokkjainak allokációja

Állományok szerkezete

Könyvtárak szerkezete

Műveletek

Osztott állománykezelés

Hozzáférés szabályozása

Állomány (file)

A létrehozó által összetartozónak ítélt információk gyűjteménye.

Több százezer állomány egyidejűleg, egyedi azonosító (név) különbözteti meg őket.

Az állomány elrejti a tárolásának, kezelésének fizikai részleteit:

- Melyik fizikai eszközön található (logikai eszköznevek használata);
- a perifériás illesztő tulajdonságai;
- az állományhoz tartozó információk elhelyezkedése a lemezen;
- az állományban lévő információk hogyan helyezkednek el a fizikai egységen (szektor, blokk);
- az információ átvitelénél alkalmazott blokkosítást, pufferelést.

Könyvtár

Az állományok csoportosítása, OS és a felhasználó szerint.

A könyvtár tartalmát katalógus írja le.

Az állománykezelő feladatai

információátvitel (állomány és a folyamatok között)

műveletek (állományokon, könyvtárakon)

osztott állománykezelés

hozzáférés szabályozása

- más felhasználók által végezhető műveletek korlátozása (access control)
- tárolt információk védelme az illetéktelen olvasások ellen (rejtjelezés, encryption)
- információ védelme a sérülések ellen, mentés

Az állományrendszer réteges implementációja

Egymásra épülő programrétegek

1. a perifériát közvetlenül kezelő **periféria meghajtó**

Feladata a tár és a periféria közötti átvitel megvalósítása. (device driver, átvitelt kezdeményező, megszakítást kezelő eljárások)

2. **elemi átviteli műveletek** rétege

A lineáris címzésű blokkok átvitele, átmeneti tárolása.

3. **állományszervezés** rétege

Háttértár szabad blokkjainak, illetve az állományhoz tartozó blokkoknak szervezése.

4. **logikai állományszervezés**

Kezeli a nyilvántartások szerkezetét, azonosító alapján megtalálja az állományt, szabályozza az állományszintű átvitelt.

Állományok tárolása a lemezen

Lemezterületet blokkonként (néhány szektor) kezeli. Mérete a lapmérethez hasonló meggondolások alapján.

Az állomány tárolásához blokkok kellenek, ezeket nyomon kell követni.

- Szabad blokkok nyilvántartása
- Blokkok allokálása

Szabad blokkok nyilvántartása

Bittérkép

Láncolt lista

Szabad helyek csoportjainak listája

Egybefüggő szabad terület tárolása

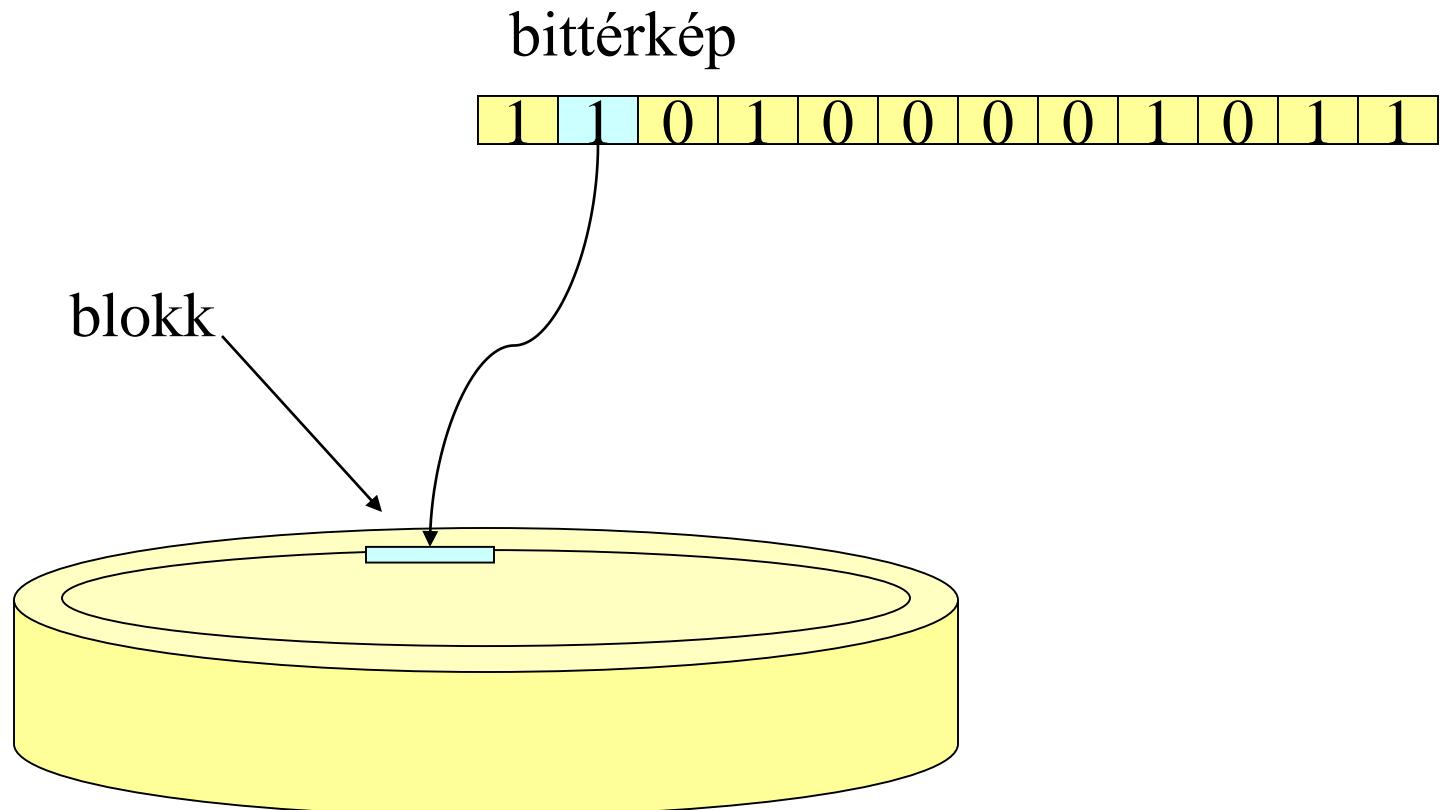
Bittérkép

Minden blokkra egy biten jelzi, hogy szabad-e

A bittérkép a lemez kijelölt helyén van

A bitvektort a memóriában kell tárolni. Sokk blokk van, így sok memóriát igényel.

Bittérkép



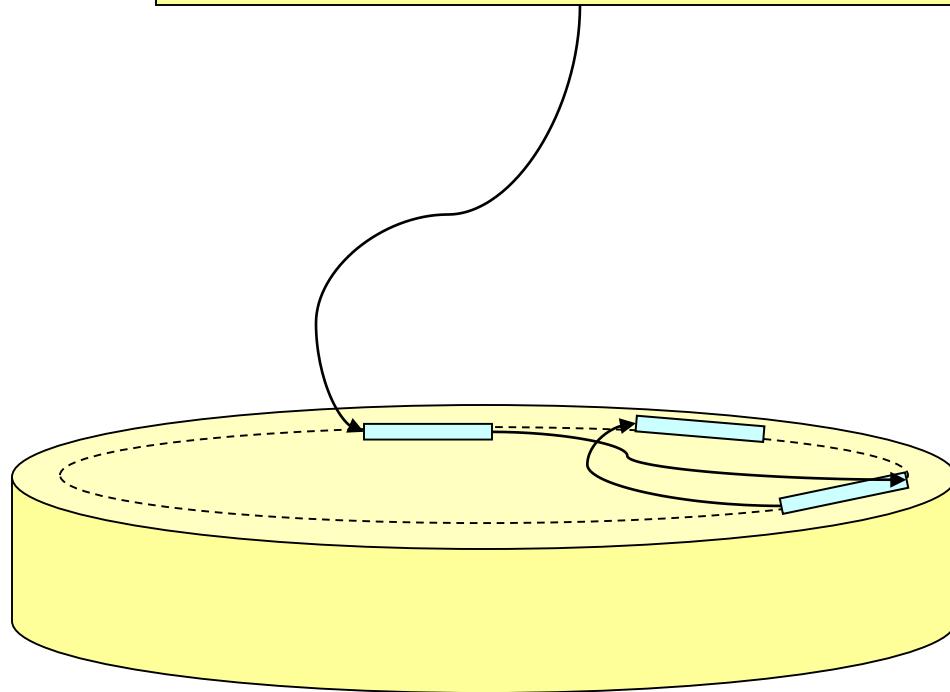
Láncolt lista

Az első szabad blokk címének tárolása, arra felfűzve a többi.

A blokk területéből vesszük le a cím területét.

Nem hatékony, lassú (sok lemezművelet).

Láncolt lista



blokk

Mutató a
következő
blokkra

Szabad helyek csoportjainak listája

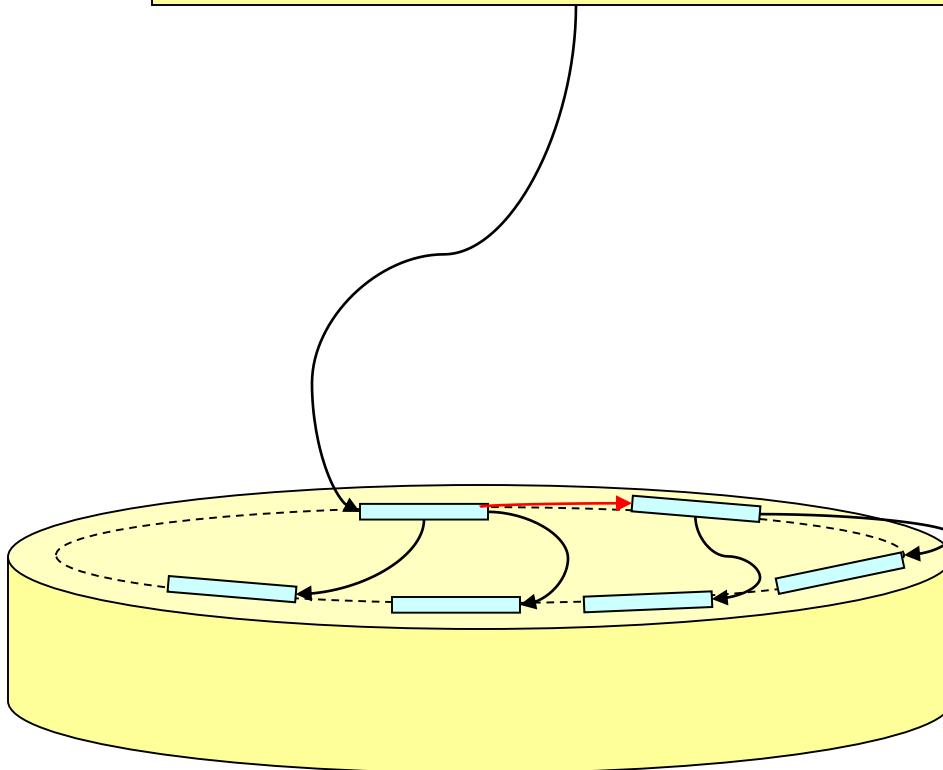
Láncolt lista javítása.

Minden blokk n db (ennyi cím fér el a blokkban) szabad blokkra hivatkozik

$n - 1$ ténylegesen szabad, az $n.$ a lista új elemére mutat.

Szabad helyek csoportjainak listája

Első szabad blokk kezdőcíme



blokk

Mutató egy
szabad blokkra

Mutató egy
szabad blokkra

Mutató egy
szabad blokkra

Mutató a
lista következő
blokkjára

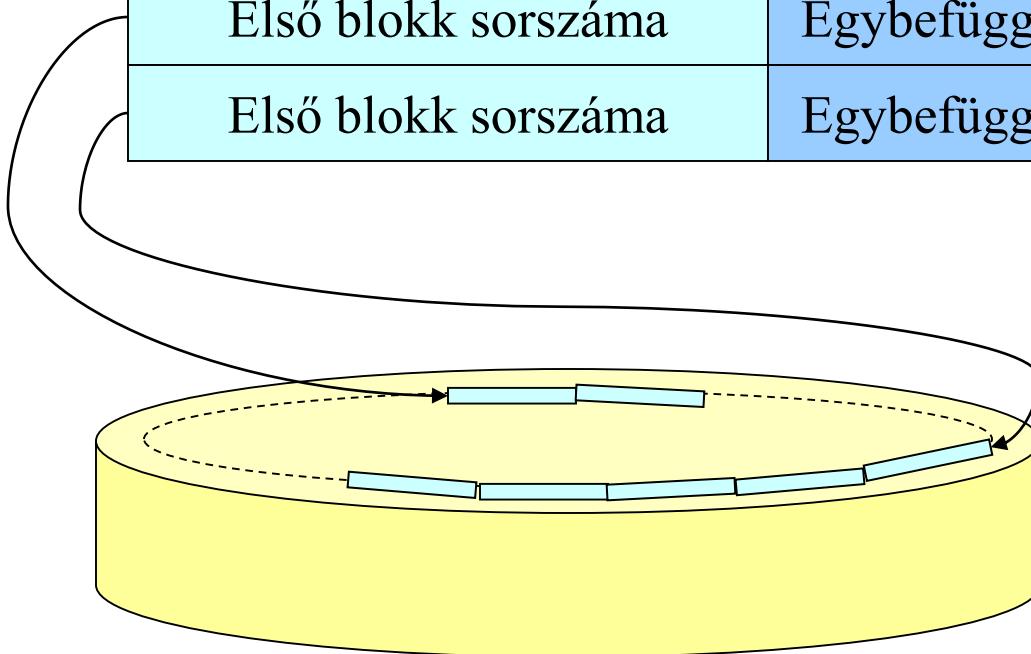
Egybefüggő szabad terület tárolása

Egy táblázatban tároljuk az összefüggő szabad blokkokat (első blokk sorszáma, blokkok száma)

Egybefüggő szabad terület tárolása

Táblázat

Első blokk sorszáma	Egybefüggő blokkok száma (2)
Első blokk sorszáma	Egybefüggő blokkok száma (5)



A lemez blokkjainak allokációja

Folytonos terület allokációja

Láncolt tárolás

Indexelt tárolás

Kombinált módszerek

Folytonos terület allokációja

Az összetartozó információkat egymás melletti blokkokban tároljuk. Első blokk sorszámát és a blokkok számát kell tárolni.

Hátrányai:

- Külső tördelődés itt is fennáll
A megoldáshoz itt is használhatók a már megismert algoritmusok (első illeszkedő, legjobban illeszkedő, legrosszabbul illeszkedő).
Nagyméretű tördelődés esetén tömöríteni kell.
- Sokszor nem tudjuk előre, hogy hány blokkra lesz szükségünk.
Lefoglaláskor becsülni kell, a rossz becslés gondot okozhat (hiba és leállás, átmásolni az eddig lefoglalt területet egy másik helyre)

Előnyei:

- A tárolt információ soros és közvetlen elérése is lehetséges.
- Jól használható tárcsere által kirakott szegmensekre (tudjuk előre a méretet)

Láncolt tárolás

Blokkokat egyenként allokáljuk, minden blokkban fenntartva egy helyet a következő blokk sorszáma számára (a rendszer az első és az utolsó blokkot tárolja).

Előnyei:

- Nincs külső tördelődés
- Rugalmas, a lefoglalt terület növekedhet.

Hátrányai:

- Csak soros elérés lehet.
- A blokkok sorszámaival nő az állomány mérete. A blokkos másoláskor a sorszámokkal külön kell foglalkozni.
- Sérülékeny, egyetlen láncszem hibája a tárolt információnak jelentős részének elvesztését jelenti.

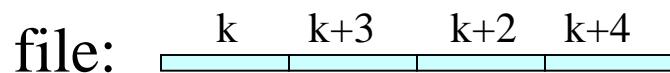
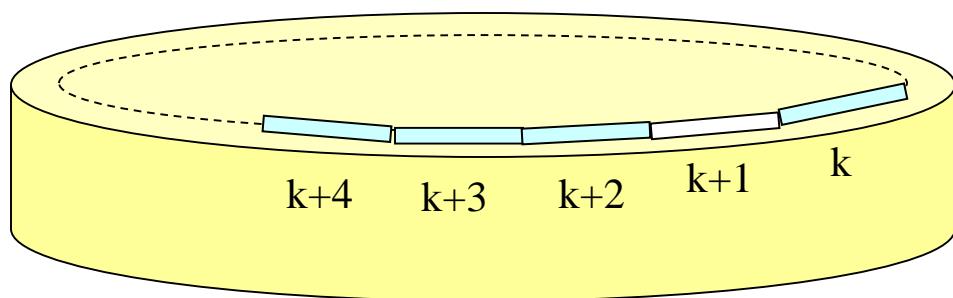
Módosított változata állomány allokációs tábla (file allocation table, FAT)

- A láncelemeket az állományuktól elkülönítve tároljuk.
- A szabad helyek tárolására is alkalmas a FAT.

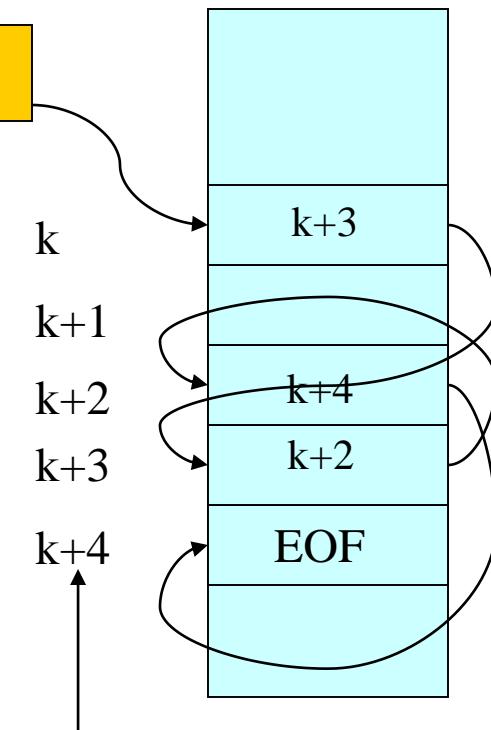
FAT

Könyvtári bejegyzés

név	attribútumok	...	start (k)
-----	--------------	-----	-----------



FAT



Blokkok sorszámai

Indexelt tárolás

Az állományhoz tartozó blokkok címei egy indextáblában vannak.

Előnyei:

- Közvetlen hozzáférés
- "Lyukas" állományok tárolása (nem minden blokk tartalmaz valós információt)

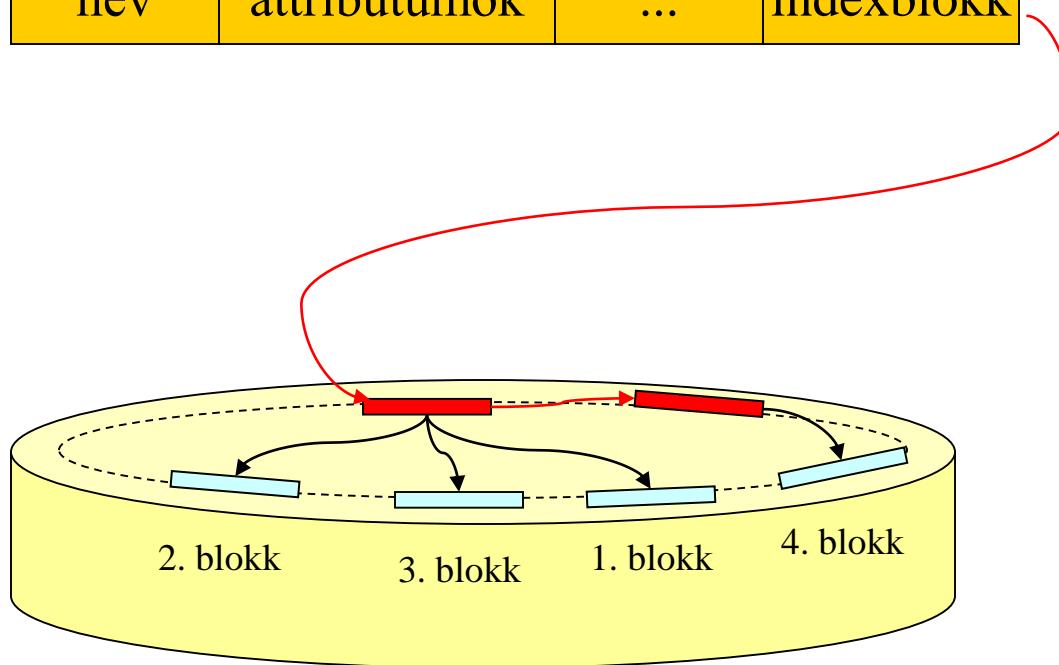
Hátrányai:

- Az indextábla tárolása legalább egy blokkot elfoglal (kis állományok esetében pazarló)
- Az indextábla mérete nem ismert, lehetővé kell tenni, hogy az növekedhessen.
 - láncolt indexblokkok
 - többszintű indextábla
 - kombinált mód, kis állományok esetében egy, majd többszintű indextábla

Indexelt tárolás

Könyvtári bejegyzés

név	attribútumok	...	indexblokk
-----	--------------	-----	------------



indexblokk

1. blokk száma
2. blokk száma
3. blokk száma
N. blokk száma
Következő indexblokk száma

Kombinált módszerek

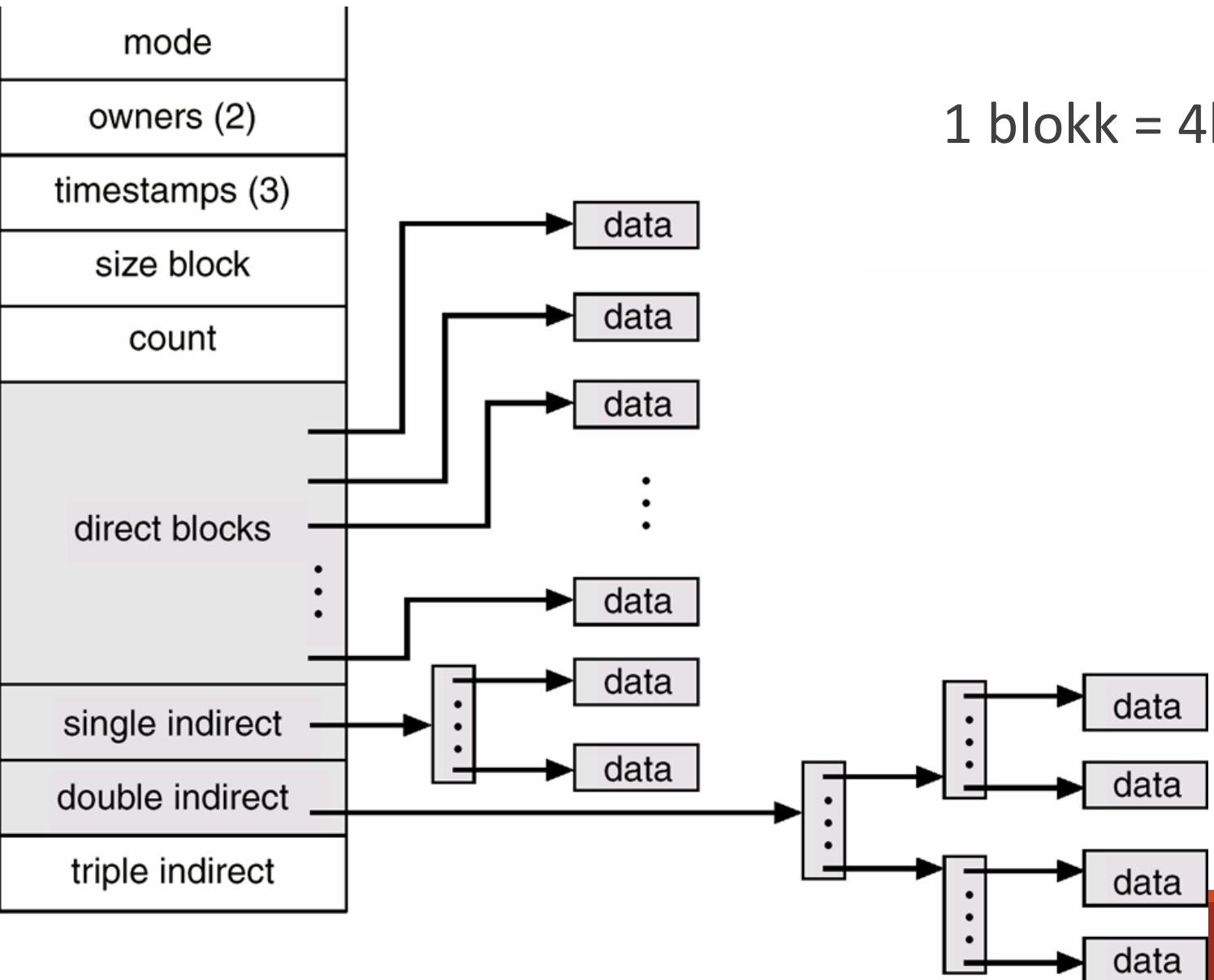
hozzáférési módja szerint:

- Soros hozzáférés esetén láncolt, közvetlen hozzáférés esetén indexelt

méret szerint:

- Kis állomány esetén folytonos, nagyok indexelve

Példa: UNIX



Az állományok belső szerkezete

Az állomány egy bitsorozat, amit a felhasználó a saját szempontjai szerint egységekbe csoportosíthat.

- Mező (field)
több bit, valamilyen típusú adatot ír le (byte-os szervezés általában)
- Rekord (record)
Mezők csoportja. Egy állomány azonos szerkezetű rekordok gyűjteménye vagy különböző, de azonosítható típusú rekordokból áll.

Mező és rekord lehet változó hosszúságú (a hossz egyértelműen meghatározható, végjel vagy hossz tárolása).

Az OS viszonya a belső szerkezethez

Nem foglalkozik vele

Csak az állományt kezelő programok ismerik az állomány szerkezetét. Egyszerű és rugalmas megoldás.

Az OS eljárásokkal támogatja

Mezőnkénti vagy rekordonkénti hozzáférési lehetőség az OS által. Nehéz megoldani, bonyolult, általában lehetetlen.

Hozzáférési módok

Soros (sequential)

- A tárolt információt csak a byte-ok sorrendjében lehet olvasni. Sok feldolgozási feladathoz elegendő a soros hozzáférés.

Közvetlen (direct)

- A tárolt elemek bármelyikét el lehet érni, ehhez meg kell adni az információ elem állományon belüli sorszámát.

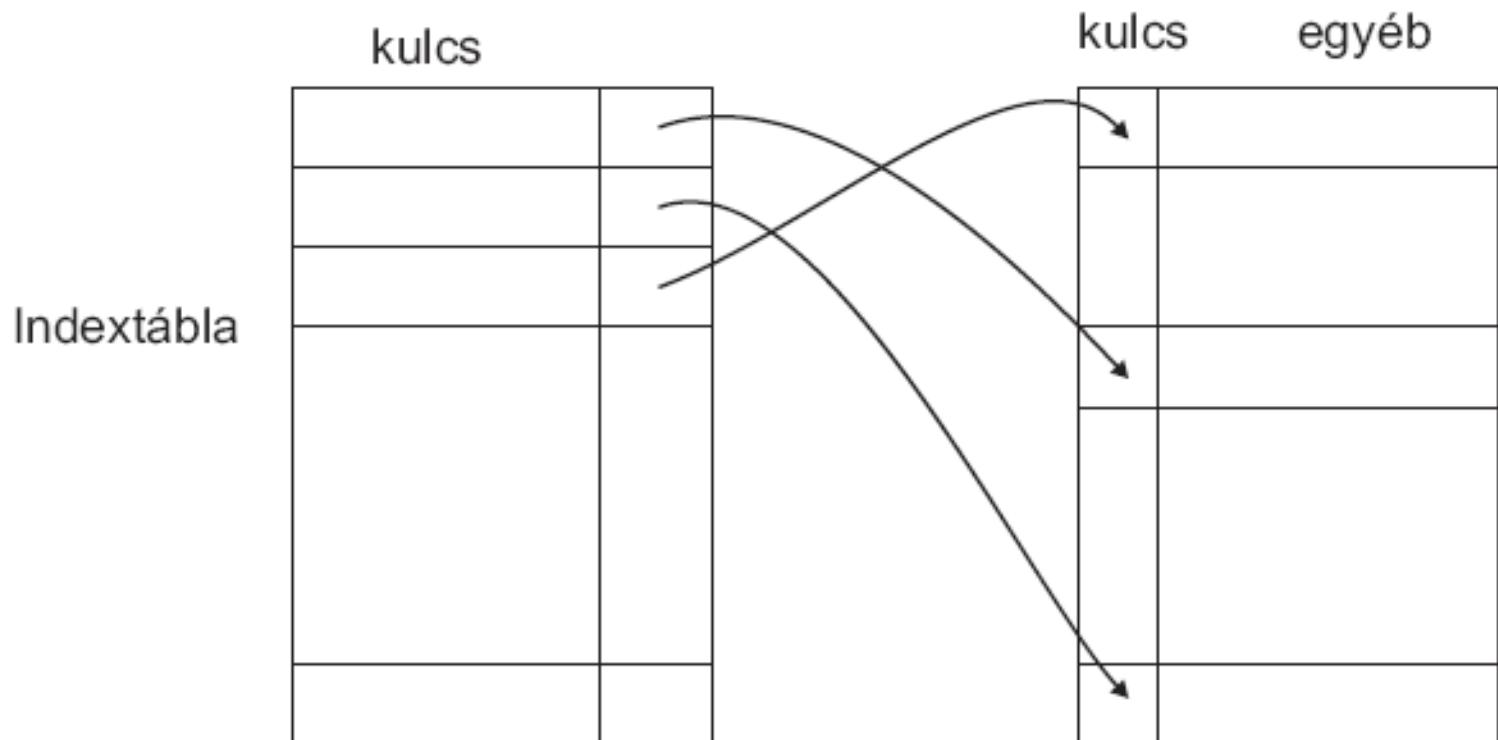
Indexelt, index-szekvenciális (index sequential access method, ISAM)

- Tartalom szerint akarunk hozzáférni.
- Kulcs szerint rendezzük az állomány rekordjait. A kulcsok egy index fájlban vannak rendezetten tárolva.

Particionált

- Az állományt soros rész-állományok alkotják, az állomány tartalmaz egy nyilvántartást arról, hogy a partíciók hol helyezkednek el a fájlban.

Indexelt, index-szekvenciális



Pl.: árunyilvántartás

Könyvtárak

Állományok, vagy más könyvtárak gyűjteménye.

Tartalmát egy nyilvántartás írja le.

Nyilvántartási bejegyzések (directory entry)

A könyvtárban tárolt állományok leírása.

Az állomány neve:

- Könyvtárként egyedi, homogén (lehet bármilyen karakterSORozat), részekre bontott (pl, név, típus - extension, verziószám).

Az állomány fizikai elhelyezkedését leíró információk:

- hossz, a hozzá tartozó háttértár blokkjainak leírása, hozzáférés módja.

Az állomány kezeléséhez kapcsolódó információk

- típusa
- tulajdonosának, létrehozójának (owner) azonosítója
- időpontok (létrehozás, utolsó módosítás, utolsó hozzáférés, argumentumának utolsó módosítása, érvényessége)
- hozzáférési jogosultságok
- hivatkozás számláló, amennyiben az állományra több különböző néven és/vagy helyen hivatkoznak

Nyilvántartási bejegyzések (folyt.)

Egyes OS-ek a tárolt információkat kettéválasztják
(UNIX)

Kötet nyilvántartás (volume directory)

A kötetben lévő összes állományt leíró fizikai
információk.

Az állomány nyilvántartások (file directory)

Tartalmazza az állományok könyvtárba
szervezésének leírását, valamint az állományokat
leíró logikai információkat.

Nyilvántartási bejegyzések (folyt.)

Egyéb információk (az OS központi tárban tárolja):

az átvitel állapota (folyamatonként)

soros hozzáférés pozíciója

megengedett műveletek (a folyamat az állományon
milyen műveleteket végezhet)

osztott kezeléssel kapcsolatos információk

- hány folyamat használja egyidejűleg
- kölcsönös kizárást milyen módon kell biztosítani
- várakozó folyamatok listája

A könyvtárak hierarchiája

Kétszintű könyvtárszerkezet

- Egyes felhasználóknak saját könyvtáruk van.

Faszerkezet

- Könyvtár tartalmazhat könyvtárat is.
- Fogalmak:
 - aktuális könyvtár (folyamatonként);
 - gyökér könyvtár,
 - elérési út,
 - keresési utak.

```
simon@ELENDIR:/$ tree / -axd
/
├── bin
├── boot
├── dev
├── etc
│   ├── alternatives
│   ├── apparmor.d
│   │   └── local
│   └── apt
│       └── ant.conf.d
```

```
simon@ELENDIR:~$ pwd
/home/simon
```

```
simon@ELENDIR:~$ cd /
simon@ELENDIR:/$ pwd
/
```

```
simon@ELENDIR:/$ cd /home/simon/tmp/
simon@ELENDIR:~/tmp$ |
```

```
simon@ELENDIR:~/tmp$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
```

A könyvtárak hierarchiája (folyt.)

Körmentes irányított gráf

- Lehetnek linkek is. Linkek mentén viszont nem lehet kör.
- Link lehet:
 - fizikai: az állomány leíró információkat megismételjük, pl UNIX-nál ugyanarra a kötetnyilvántartás bejegyzésre hivatkozunk
 - logikai
- Problémák:
 - egy állományhoz többféle elérés tartozik (mentésnél, statisztikáknál, osztott elérésnél csak egy útvonalat kell figyelembe venni)
 - annak meghatározása, hogy mikor kell egy állományt törölni (egy állományra hivatkozások számát nyilván kell tartani)

Általános gráf

- Kör lehetséges
 - Keresésnél végtelen ciklus.
 - Egy törlésével a teljes hurok a "levegőben lógva" maradhat (nem törölt könyvtárak, amik a gyökérből kiindulva nem érhetők el).

```
simon@ELENDIR:~$ ls -ali
total 7188
    3377699720535956 drwxr-xr-x 1 simon simon      512 Oct 21 17:32 .
106116066219925693 drwxr-xr-x 1 root  root      512 Sep  2 18:24 ..
    89790517570735912 -rwxr--r-- 3 simon simon 2306038 Oct 21 17:33 alma
    89790517570735912 -rwxr--r-- 3 simon simon 2306038 Oct 21 17:33 almalink
    89790517570735912 -rwxr--r-- 3 simon simon 2306038 Oct 21 17:33 almalnk
23080948090280888 lrwxrwxrwx 1 simon simon        4 Oct 21 17:30 almasimlink -> alma
```

Műveletek állományokon

átvitel, írás, olvasás

- Közvetlen átvitel esetén, információ címe szükséges
- Soros hozzáférésnél az aktuális pozíciót a rendszer növeli és tárolja.
- Szimultán írás-olvasás esetén soros hozzáférésnél közös vagy több pozíció használata.

hozzáadás (append)

- Az állomány végéhez új információt írunk. Az állomány mérete növekszik, esetlegesen új blokk lefoglalása.

pozicionálás

- Soros hozzáférés esetén megadhatjuk az aktuális pozíciót.

Műveletek állományokon (folyt.)

állomány megnyitása

- az állomány megkeresése
- jogosultságok ellenőrzése
- az elvégezendő műveletek megadása
- osztott állománykezelés szabályozása
- soros hozzáférés pozíciójának beállítása

A műveletek hatékonyabbak, ha nem kell mindegyiknél az állományt a nyilvántartásból kikeresni.

A sikeres megnyitás után OS felépít egy adatszerkezetet, a további műveletek erre az adatszerkezetre hivatkoznak.

Műveletek állományokon (folyt.)

állomány lezárása

- puffer esetén a ki nem írt információ kiírása, osztott állománykezelésnél az állomány felszabadítása.

állomány végrehajtása

- Az OS létrehoz egy új folyamatot, a programállományt betölti a folyamat tárterületére és elindítja.

A könyvtárra is hatással levő műveletek:

- állomány létrehozása (új bejegyzés, blokkok lefoglalása)
- állomány törlése (bejegyzés megszüntetése, blokkok felszabadítása)

Műveletek könyvtárakon

Állomány attribútumának módosítása (az állományhoz tartozó logikai információk megváltoztatása)

Új könyvtár létrehozása

Könyvtár törlése

- csak akkor törölhető, ha üres
- törli a benne lévő állományokat
- rekurzívan törli a benne lévő könyvtárakat is

```
simon@ELENDIR:~$ ls mappa
alma
simon@ELENDIR:~$ rmdir mappa
rmdir: failed to remove 'mappa': Directory not empty
simon@ELENDIR:~$ rm -r mappa
simon@ELENDIR:~$ ls mappa
ls: cannot access 'mappa': No such file or directory
```

Műveletek könyvtárakon (folyt.)

Keresés

- Egy névhez meg kell találni a hozzá tartozó állományt. A keresés típusa függ a nyilvántartás szerkezetétől: rendezetlen keresés, rendezett felező keresés, hash keresés.

Új bejegyzés létrehozása

- Nyilvántartás méretének dinamikusan növekednie kell. Rendezett nyilvántartás esetén a rendezettséget meg kell tartani.

Bejegyzés törlése

Osztott állománykezelés

Ez is egy erőforrás, amelyet egyidejűleg több folyamat is használni akar.

Csak olvasás esetén osztottan gond nélkül használható.

Írás esetén kölcsönös kizárással kell védeni a folyamatot.

- Egész állományra vonatkozó szabályozás (file lock)
 - Az állományt először megnyitó folyamat definiálja, hogy a későbbi megnyitási kérelmekből mit engedélyezhetünk.
 - kizárolagos használat
 - többi folyamat csak olvashatja
 - több folyamat is megnyithatja írási joggal
 - Írás esetén az olvasó folyamatok mikor veszik észre a változást:
 - azonnal,
 - ha a folyamat lezárta az állományt.
- Állomány részeire vonatkozó kizáráskor
 - Rekordonként lehet kizárási lehetőségeket definiálni.

A hozzáférés szabályozása

Cél: jogosulatlanok ne férjenek hozzá állományokhoz, ne végezhessék rajtuk műveleteket. A jogokat a létrehozója, vagy speciális jogokkal rendelkező felhasználó definiálja.

Jogosultságok:

- állományokra: írás, olvasás, végrehajtás, hozzáírás, törlés
- könyvtárakra: könyvtár módosítása, listázás, keresés, új állomány létrehozása, könyvtár törlése.

Jogosultság tartozhat:

- állományokhoz
- elérési útvonalhoz (különböző elérésekhez más-más jogosultságok)

Jogosultság definiálható:

- felhasználónként
- felhasználó csoportonként

```
-rwxr-xr-- 3 simon mygroup 151 Oct 21 17:47 alma
```

Operációs rendszerek

11. AZ OPERÁCIÓS RENDSZEREK BIZTONSÁGI KÉRDÉSEI

Felhasznált irodalom:

- Kóczy-Kondorosi (szerk.): Operációs rendszerek mérnöki megközelítésben
- Tanenbaum: Modern Operating Systems
- Silberschatz, Galvin, Gagne: Operating System Concepts

Tartalom

A védelem célja

A fenyegetés forrásai

Belső biztonság

Külső biztonság

A védelem célja

Az operációs rendszer hardver és szoftver elemekből áll

Minden elemnek egyéni azonosítója van és előre definiált műveletekkel érhető el

A védelmi probléma: minden elemet csak az arra jogosult eljárások érhessenek el és azt helyesen használják

A fenyegetés forrásai

Belső:

- rendszerben lévő folyamatok felől,

Külső:

- rendszertől független tényezőktől.

Belső biztonság

Belső biztonsági rendszer (internal security):

- programhibák elleni védelem,
- erőforrások jogosulatlan felhasználásának védelme.

Védelmi tartomány (protection domain): megadja, hogy a folyamat az adott állapotában mely erőforrásokhoz férhet hozzá és ezeken milyen műveleteket végezhet.

- Minél szűkebbre kell definiálni, csak a valóban szükséges jogokat megadni (*need-to-know* elv).
- Védelmi tartomány lehet statikus vagy dinamikus (állapottal változó).

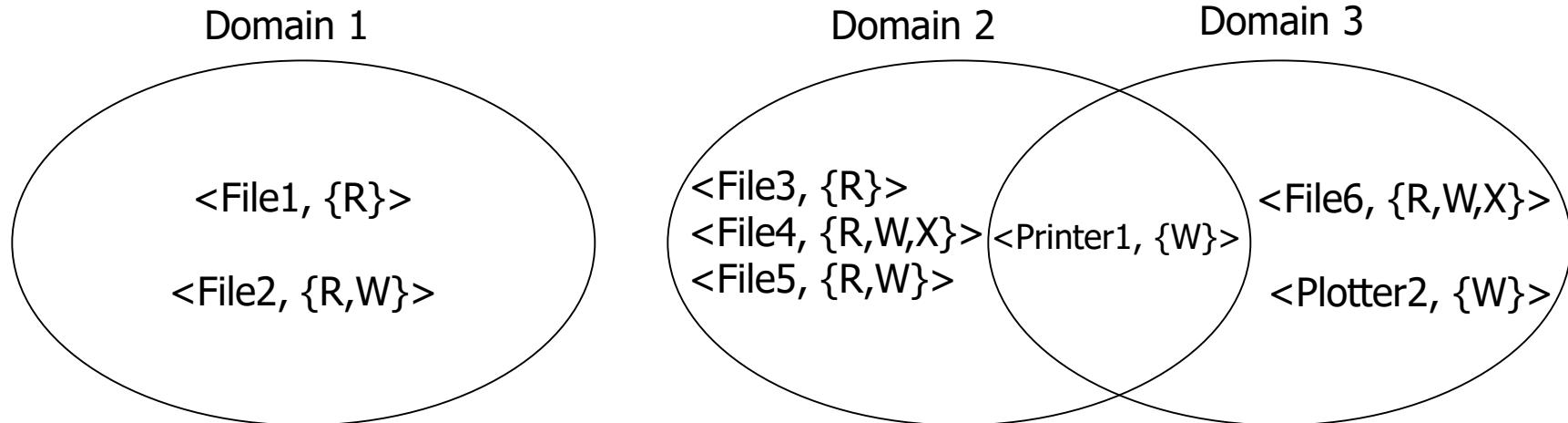
Védelmi tartományok

Hozzáférési jogok:

<Objektum, {jogosultságok listája}>,

ahol a jogosultságok listája az adott objektumon végrehajtható műveletek egy részhalmaza

Védelmi tartomány (domain): hozzáférési jogok halmaza



Pi: Egy domain 2-ben futó folyamat a File5 objektumon Read és Write műveleteket hajthat végre.

Statikus védelmi tartomány

A "jogok" a folyamat élete során nem változnak.

Hozzáférési mátrix:

- sorai a tartományok,
- oszlopai az erőforrások (ritka mátrix).

		Object							
		File1	File2	File3	File4	File5	File6	Printer1	Plotter2
Domain	1	Read	Read Write						
				Read	Read Write Execute	Read Write		Write	
3							Read Write Execute	Write	Write

Statikus védelmi tartomány ábrázolása

Globális tábla (global table)

Hozzáférési lista (access control list)

Jogosítványok listája (capability list)

Globális tábla

(global table)

A <tartomány, erőforrás, művelet> hármasokat tartalmazza.

Minden művelet esetén a táblát végig kell nézni.

A tábla túl nagy, így a keresés lassú.

Nem lehet csoportokat képezni (pl. mindenki számára elérhető erőforrásokat minden tartományhoz fel kell venni).

Domain	Object							
	File1	File2	File3	File4	File5	File6	Printer1	Plotter2
1	Read	Read Write						
2			Read	Read Write Execute	Read Write		Write	
3						Read Write Execute	Write	Write

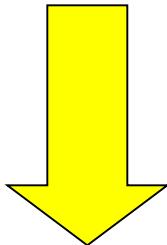


<Domain1, File1, R>,
<Domain1, File2, RW>,
<Domain2, File3, R>,
...
<Domain2, Printer1, W>,
<Domain3, Printer1, W>,
...

Hozzáférési lista (access control list)

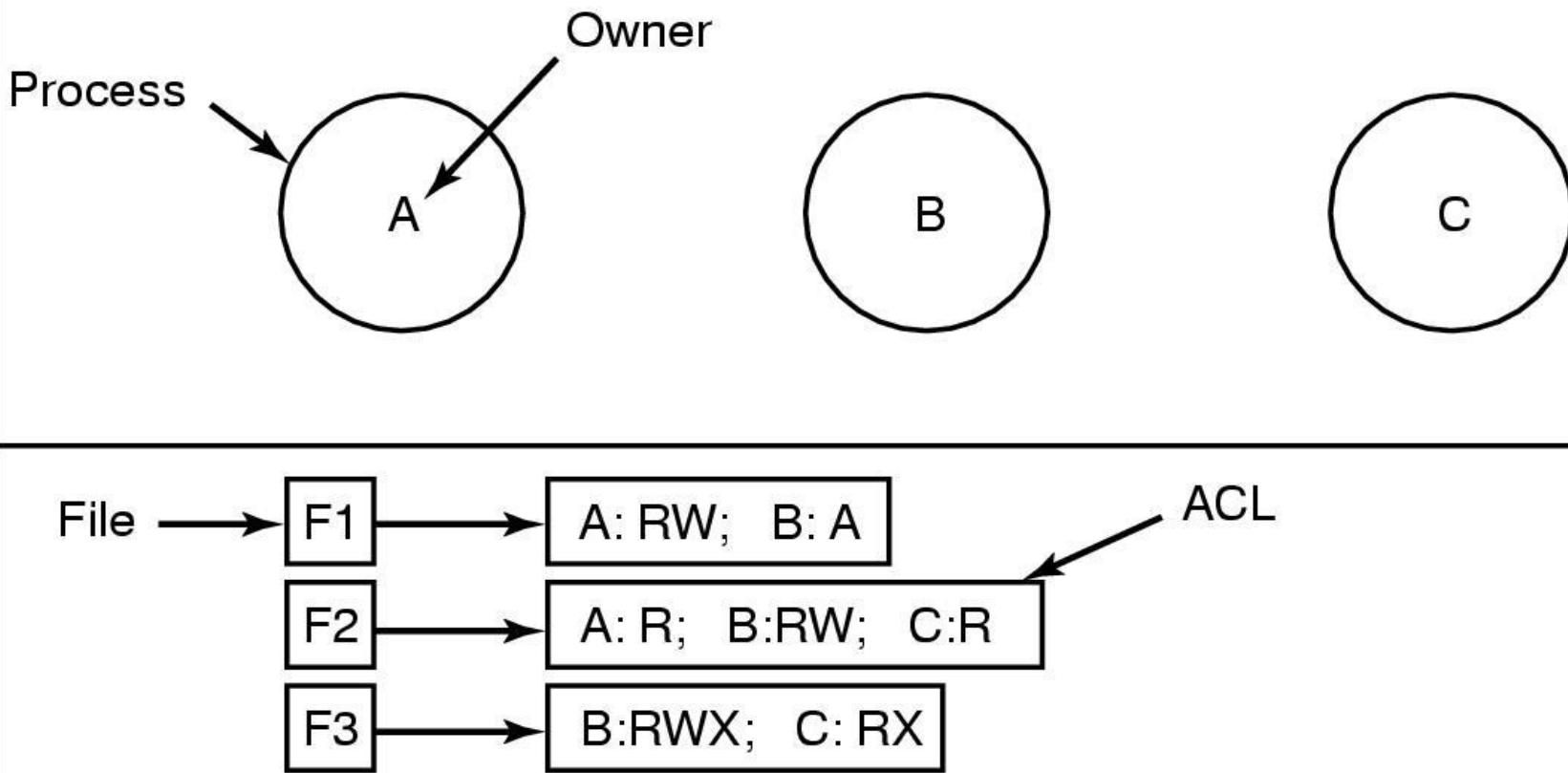
Minden erőforráshoz <tartomány, művelet> párok sorozata + alapértelmezett műveletek (minden tartomány számára engedélyezett művelet).

Felhasználói igényekhez igazodnak, új erőforrás létrehozásakor lehet definiálni a védelmi tartományokban kiadható műveleteket.



		Object							
		File1	File2	File3	File4	File5	File6	Printer1	Plotter2
Domain	1	Read	Read Write						
	2		Read	Read Write Execute	Read Write			Write	
3						Read Write Execute	Write	Write	

Hozzáférési lista 2.

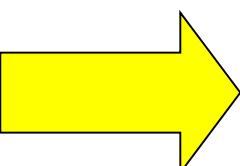


Jogosítványok listája (capability list)

Minden **tartományhoz** <erőforrás, művelet> párok sorozata. Jogosítvány egy <erőforrás, művelet> párra vonatkozó referencia. A tartományhoz tartozó folyamatok számára közvetlenül nem érhetők el (OS kezeli).

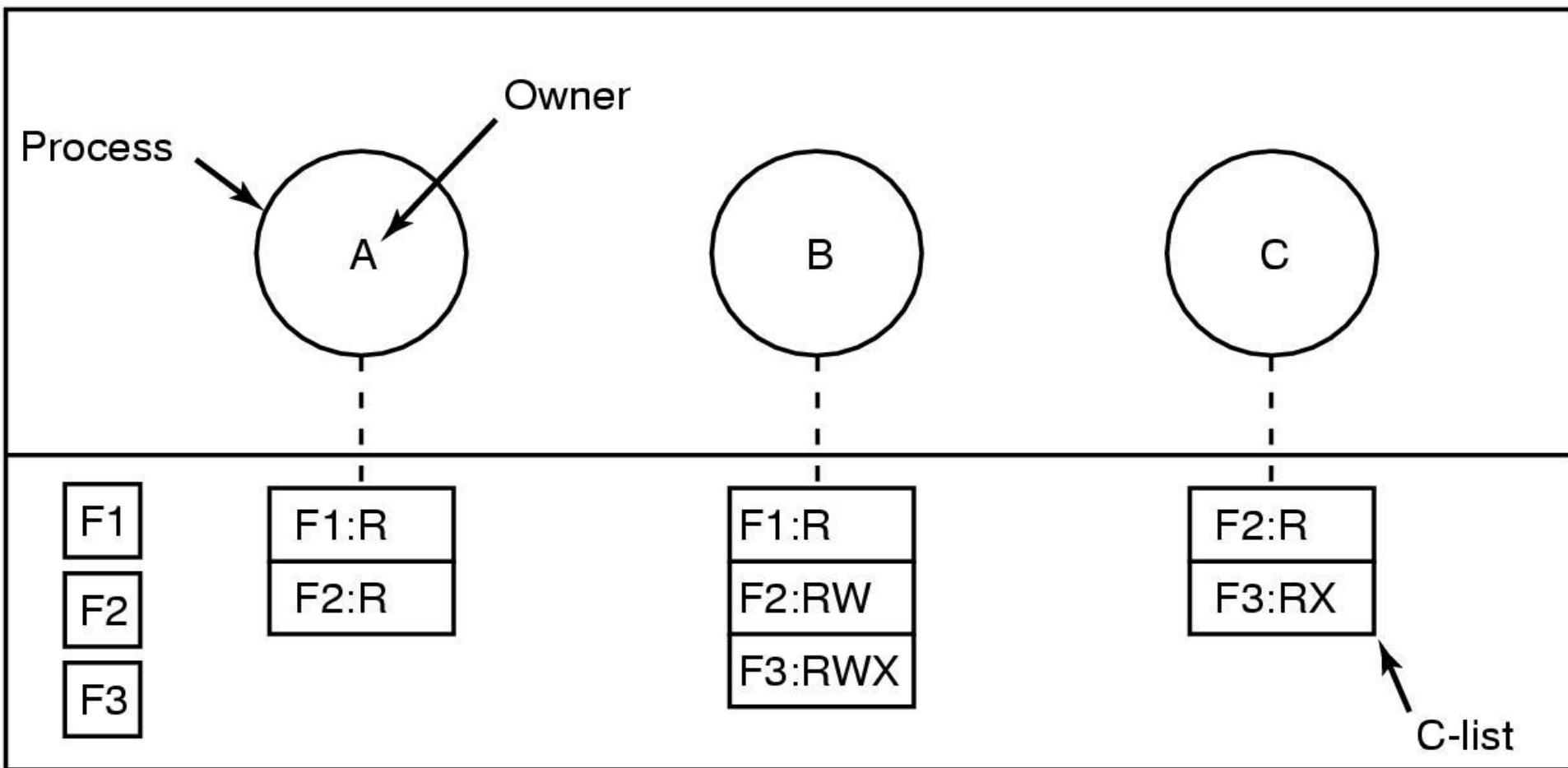
Nem az objektumokhoz igazodik, nem szemléletes.

Az erőforrás használatakor a folyamat csak bemutatja a jogosítványát, az OS-nek csak ellenőrizni kell az érvényességét (nincs szükség keresésekre).



		Object							
		File1	File2	File3	File4	File5	File6	Printer1	Plotter2
Domain		Read	Read Write						
1									
2				Read	Read Write Execute	Read Write		Write	
3							Read Write Execute	Write	Write

Jogosítványok listája



Statikus védelmi tartományok megvalósítása

Gyakori a hozzáférési és jogosítvány listák együttes használata.

Pi. UNIX rendszerekben egy állomány megnyitásánál a folyamatnak meg kell adnia a műveleteket. Ezt a hozzáférési lista alapján ellenőrzi az OS, majd a folyamat kap egy jogosítványt, a későbbi műveleteknél ezt használja.

```
-rw-r-xr-- 3 simon mygroup 151 Oct 21 17:47 alma
```

```
int filedesc = open("alma", O_APPEND);  
  
write(filedesc, "Egy uj sor\n", 11);
```

Jogosítvány: <filedesc, APPEND>

Jogosítvány ellenőrzés: OK

Dinamikus védelmi tartomány

A statikus védelmi tartományokban túl sok erőforrás van (mindet fel kell venni), a dinamikus védelmi tartomány csak a szükségeseket tartalmazza.

Hozzáférési mátrix itt is használható.

A védelmi tartomány is egy erőforrás, ezt is fel kell venni a mátrix oszlopai közé.

Hozzáférési mátrix dinamikus védelmi tartományokban

		Védelmi tartomány, mint erőforrás									Művelet	
		Object										
		File1	File2	File3	File4	File5	File6	Printer1	Plotter2	Domain1	Domain2	Domain3
1	Read	Read Write									Switch	
2			Read	Read Write Execute	Read Write			Write				
3						Read Write Execute		Write	Write			

Pi.: Egy Domain1-ben futó folyamat átkapcsolhat Domain2-re.

Dinamikus védelmi tartomány

Új bejegyzések hozzáférési mátrixban:

kapcsolás (switch)

- Az adott tartományból egy folyamat átkapcsolhat az oszlopan kijelölt tartományra.

másolás (copy)

- Az adott tartományban tartózkodó folyamat hozzáférési jogait átmásolhatja egy másik tartományba. A másoláson kívül lehet: *átadás* (transfer), amikor a jogok az eredeti tartományból törlődnek; *korlátozott másolás*, az új tartományból a jogot nem lehet továbbadni.

tulajdonos (owner)

- Az erőforrás tulajdonosa más tartományokban törölhet vagy adhat az erőforráshoz tartozó műveleteket (oszlopok mentén történő módosítások).

vezérlő (control)

- Sorok mentén történő módosításokra ad lehetőséget. Csak a tartományokhoz tartozó oszlopokban szerepelhet. Ha egy tartomány vezérli a másikat, akkor annak sorában elhelyezhet illetve törölhet jogosultságokat.

Külső biztonság

A felhasználók azonosítása

Naplázás

Rejtjelzés





A felhasználók azonosítása

Jogtalan hozzáférések megakadályozására a felhasználókat egyértelműen és megbízhatóan azonosítani kell:

- a felhasználó attribútumai (ujjlenyomat, retina, stb.)
- a felhasználó birtokában lévő tárgyak (kulcs, azonosító kártya, stb.)
- felhasználó által tudott ismeretek (pl. név, jelszó)

Jelenlegi rendszerek főleg jelszóval azonosítanak.

- biztonságos jelszó (jó jelszó választás, jelszó gyakori cseréje)
- rendszer szigorúan védett állományban tárolja (rejtjelezett)

Kérdés



Jogtalan hozzáférések megakadályozására a felhasználókat egyértelműen és megbízhatóan azonosítani kell:

- a felhasználó attribútumai (ujjlenyomat, retina, stb.)
- a felhasználó birtokában lévő tárgyak (kulcs, azonosító kártya, stb.)
- felhasználó által tudott ismeretek (pl. név, jelszó)

Kérdés:

Egy banki hozzáféréshez meg kell adnom a felhasználói azonosítómat, jelszavamat. Ezután kapok egy SMS-t, aminek tartalmát szintén meg kell adnom a belépéshoz.

Milyen azonosítási módszereket látunk itt?



**_

A jelszó kezelése

A rendszer a jelszót biztonsági okokból nem tárolja (ellopható)

A p jelszó helyett $f(p)$ -t tároljuk, ahol f „egyirányú” függvény

$f(p)$ sok esetben nyilvános

Támadási módok:

- Gyakori jelszavakkal való próbálkozás (pl. szótár). Nagy valószínűséggel néhány jelszót talál a jelszófájlban.
- Figyelem: ez nem egy adott felhasználó ellen irányul; cél bárkinek a jelszavát megszerezni.

Módosítás: „sózás”

Sózott jelszó

A jelszófájl nem $f(p)$ -t tárolja (hiszen p esetleg gyakori szó!)

Tárolunk

- egy véletlen N egész számot, ez a „só”, és
- $f(pN)$ -t.

Pl. (jelszó: alma)

User2, 67832, f(alma67832),

ahol

- User2 a felhasználó azonosítója,
- 67832 a só,
- f(alma67832) az eltárolt kódolt sózott jelszó.

Nehezebb a „bárki ellen” való véletlen próbálkozás

Példa

Linux alatt a kódolt jelszavakat a /etc/shadow file tárolja

Minden sora egy felhasználóhoz tartozik

Példa

```
simon:$6$s7iDYfRxmTKKcpS9$PhKMys.m [...] qEIas.dHJgAI0:18526:0:99999:7:::
```

↓
user

Kódolt jelszó:

\$f\$N\$f(pN)
f: 6 → SHA-512
N=s7iDYfRxmTKKcpS9
f(pN)=PhK...

↓
Érvényességi idő
(nap)

↓
Legalább ennyi napot
kell két jelszóváltoztatás
között várni

↓
Utolsó változtatás:
1970.jan.1. óta eltelt napok száma
(2020.szept.21.)



Naplózás

A rendszer feljegyzi

- az egyes erőforrásokhoz a hozzáférések időpontját,
- a műveletet és
- a felhasználót.

Ez nem véd, de az elkövető utólagos azonosítását megkönnyíti.

Rejtjelzés



A nem védett információs csatornán átadott adatokat rejtjelezéssel (encryption) védhetjük.

Kétkulcsos rejtjelezés (csapóajtó kódok):

Alkalmazás:

- Üzenet titkosítása
- Üzenethitelesítés (az üzenetet senki nem módosította)
- Partner hitelesítés (a küldő hitelesítése), elektronikus aláírás.

Aszimmetrikus titkosítás

Aszimmetrikus (kétkulcsos, v. nyilvános kulcsú) titkosítás

Üzenet:

- M

Két kulcsot (függvényt) használ, amelyekre igaz:

- S: *titkos*, csak a tulajdonos ismeri
- P: *nyilvános*, mindenki ismeri
- $S=P^{-1}$, tehát $S(P(M))=P(S(M))=M$
- P-ből nagyon nehéz (gyakorlatilag lehetetlen) S-et kitalálni

Üzenet titkosítása

- A bizalmas M üzenetet küld B -nek
 - A csatorna nem titkos (az üzenetet bárki láthatja)
 - Cél: csak B tudja elolvasni
-
- A:
 - Megírja M -et
 - Kódol B nyilvános kulcsával: $P_B(M)$
 - Elküldi $P_B(M)$ -et egy üzenetben
 - B
 - Veszi $P_B(M)$ -et
 - Dekódol: $S_B(P_B(M))=M$
 - C (támadó)
 - Olvassa $P_B(M)$ -et
 - Ismeri P_B -t
 - De nem tudja S_B -t, nem tudja dekódolni $P_B(M)$ -t

Üzenet aláírása

- A az M üzenetet küldi B -nek
- Cél: B biztos legyen benne, hogy a feladó A
- A:
 - Megírja M -et
 - Kódolja a saját titkos kulcsával: $S_A(M)$
 - Elküldi $S_A(M)$ -et egy üzenetben
- B
 - Veszi $S_A(M)$ -et
 - Dekódolja A nyilvános kulcsával: $P_A(S_A(M))=M$
- C (támadó)
 - Olvassa $S_A(M)$ -et
 - Ismeri P_A -t
 - Tudja dekódolni (olvasni) M -et
 - De nem tudja A nevében küldeni, mert nem ismeri S_A -t

Kérdés



Hogyan tudunk kétkulcsos titkosítással olyan üzenetet küldeni, ami titkos és a küldője is azonosítható?

A rosszindulatú programok és jellemzőik

Trójai faló (Trojan horse)

Rejtekajtó (trap door / back door)

Féreg (Worm)

Vírus

Stack és Buffer Overflow támadás

Túlterheléses (Denial of Service) támadás



Trójai faló

(Trojan horse)

- Terjedés: megtévesztés által
 - Emailban küldött kód
 - Reklám egy weblapon
 - ...
- Hatás:
 - Adatlopás
 - Számítógép feletti kontrol átvétele (pl. botnet)
- Egyszerű példák:
 - A keresési utakat felhasználva a kívánt rendszerprogram előtt (helyett) indul el
 - gyakran elgépelt parancsok használata (pl. ls → la)

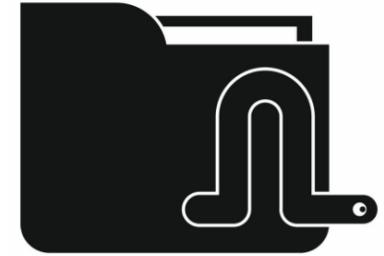


Rejtekajtó

(trap door / back door)

- Egy programban az írója a programban olyan részeket tesz amelyek nem dokumentált, jogosulatlan tevékenységeket végezhetnek.
- Léteznek fordítóprogramokba, vagy programkönyvtárakba épített rejtekajtók, amelyek a lefordított kódba is belekerülnek.
- Létezik jogos felhasználás is:
 - Gyártók tesztelésre vagy hibaelhárításra beépítik (vagy felejtik)

Féreg



(Worm)

- Szaporodik, terjed (általában a számítógépes hálózaton keresztül)
- Fertőzött gép újabb áldozatokat keres, ezeket tovább fertőzi
- A fertőzésen kívül vihet magával „csomagot”, ami kárt okoz:
 - Fájlok törlése
 - Bizalmas adatok (jelszavak) lopása
 - Adatok „túszul ejtése” (titkosítása), váltságdíjért (ransomware)
 - Botnet-eket férgekkel állítják fel



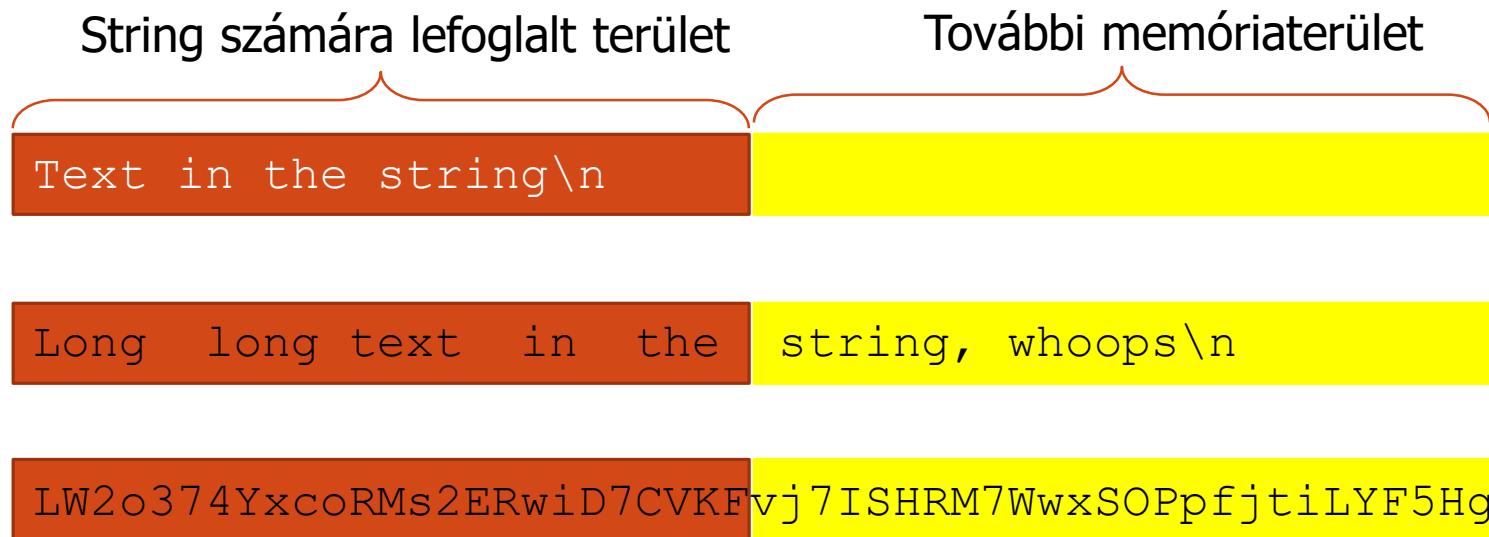
Vírus

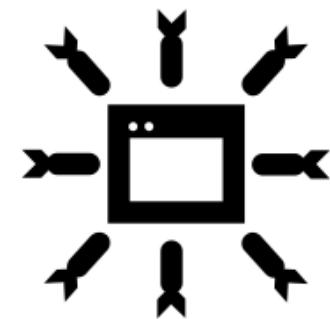
- Más programokba saját utasításait csempészi.
- Fertőzhet:
 - Végrehajtható állományokat (pl. .exe)
 - Dokumentumokat (pl. .doc)
 - Boot szektor
 - Háttértár
- Jól rejtőzik, nincs saját állomány
 - A jobb rejtőzkodés érdekében képes lehet saját magát változtatni
- A fertőzött program indításakor a vírus aktiválódik
 - tovább terjed
 - kártétel



Stack és Buffer Overflow támadás

- A programban jelen levő, nem ellenőrzött korlátok kihasználásával a vezérlés saját kódnak átadása (pl. hosszú inputba csempészett kód).
- Pl.:





Denial of Service támadás

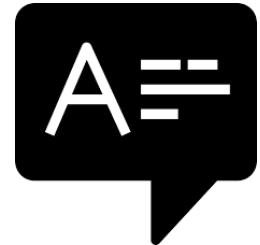
(DoS)

Cél:

- A célzott gép túlterhelése
- ezáltal a hasznos tevékenység megakadályozása
- Pl: Google, Amazon, GitHub, bankok

Fejlettebb formája: elosztott DoS (DDoS)

- Több gépről indul a támadás
- Védekezés nehezebb
- Általában botnet



Válaszok a kérdésekre:

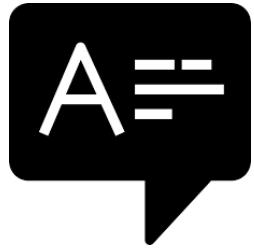
Kérdés:

Egy banki hozzáféréshez meg kell adnom a felhasználói azonosítómat, jelszavamat. Ezután kapok egy SMS-t, aminek tartalmát szintén meg kell adnom a belépéshez.

Milyen azonosítási módszereket látunk itt?

Válasz:

1. a jelszavas azonosítás ismeretet vizsgál
2. az SMS egy tárgy birtoklását ellenőrzi: telefon (pontosabban a telefonszámomat azonosító SIM-kártya)



Válaszok a kérdésekre:

- Hogyan tudunk kétkulcsos titkosítással olyan üzenetet küldeni, ami titkos és a küldője is azonosítható?
- A:
 - Megírja M-et
 - Kódolja a saját titkos kulcsával: $S_A(M)$
 - Tovább kódolja B nyilvános kulcsával: $P_B(S_A(M))$
 - Elküldi $P_B(S_A(M))$ -et egy üzenetben
- B
 - Veszi $P_B(S_A(M))$ -et
 - Dekódolja saját titkos kulcsával: $S_B(P_B(S_A(M))) = S_A(M)$
 - Tovább dekódolja A nyilvános kulcsával: $P_A(S_A(M)) = M$