Operációs rendszerek

Németh Gábor
gnemeth@inf.u-szeged.hu

Szegedi Tudományegyetem

2011-2012-II
levelező tagozat

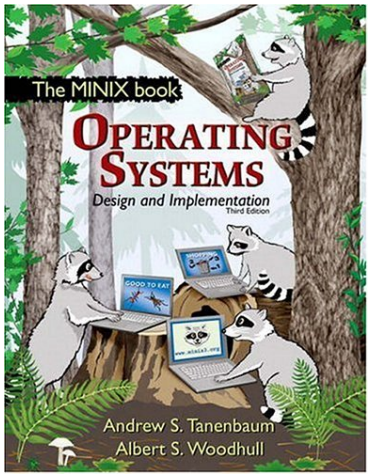
A kurzusról

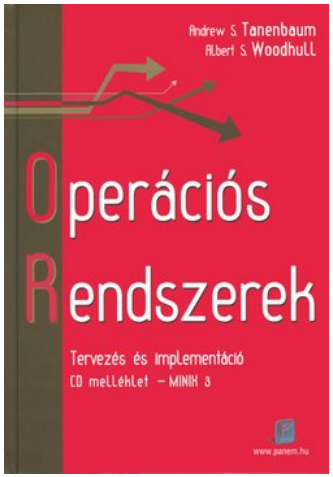
Követelmények ismertetése

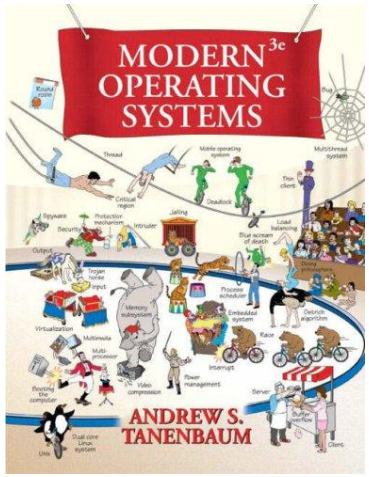
A fóliákat Erdőhelyi Balázs készítette Makay Árpád jegyzete alapján.

Ajánlott irodalom:

- Andrew S. Tanenbaum - Albert S. Woodhull: Operating Systems; Design and Implementation, Prentice Hall, 2006. Operációs rendszerek; tervezés és implementáció, Panem, 2007
- Andrew S. Tanenbaum - Modern Operating Systems 3. ed., Prentice Hall, 2007







Mi egy operációs rendszer?

Egy modern számítógép a következő dolgokból áll:

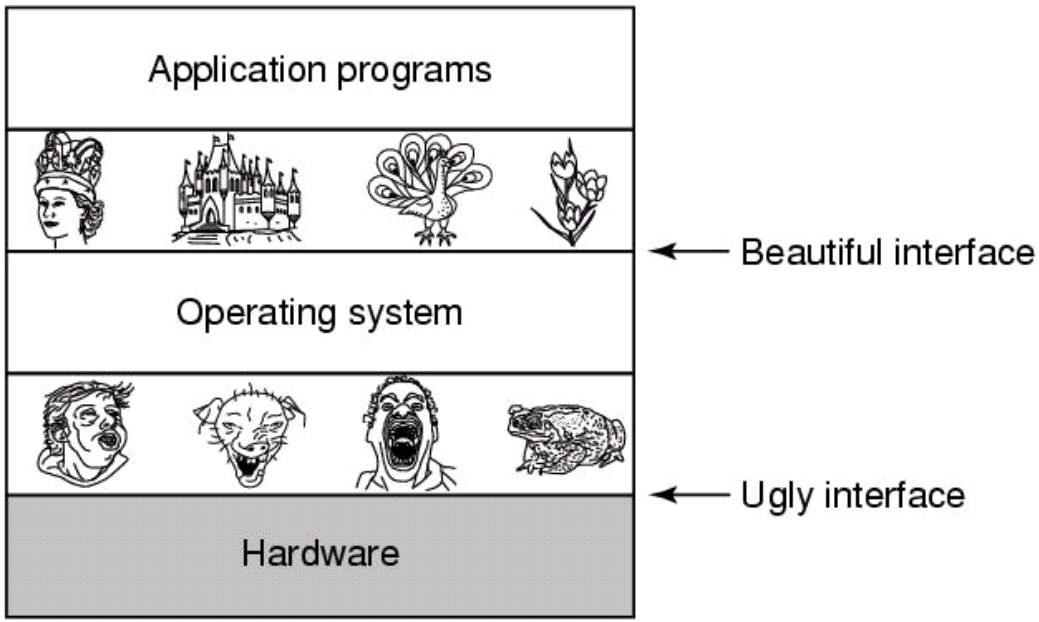
- Egy vagy több processzor
- Memória
- Lemezek
- Nyomtatók
- Különböző I/O eszközök
- ...

Ezen komponensek kezelése egy szoftver réteget igényel – Ez a réteg az **operációs rendszer**

Operációs Rendszerek helye

Számítógépes Rendszer + Felhasználók = Egységes Tervezési Egység.

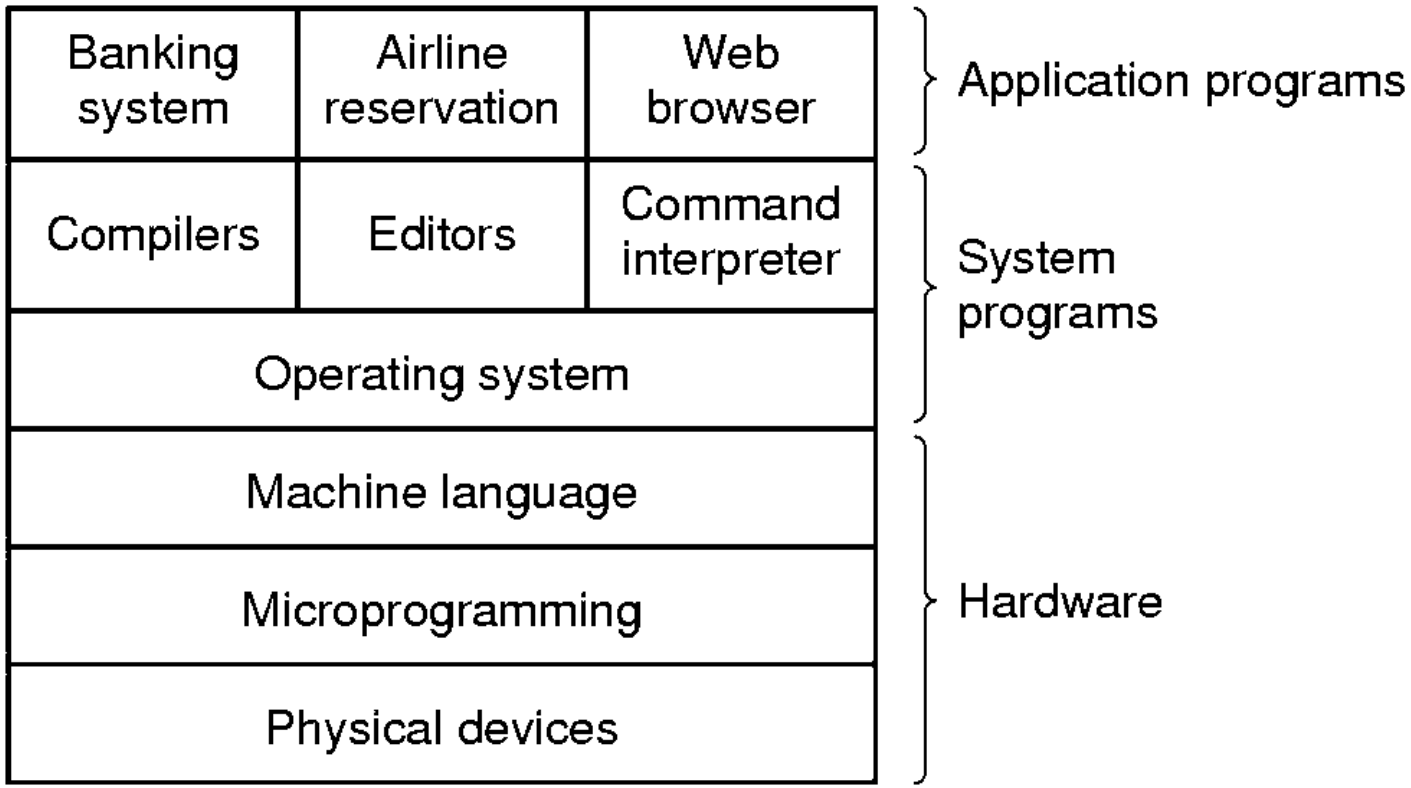
- Számítógépes Rendszer: hardver, szoftver, hálózat.
- Felhasználók: ember (programozó is), gép, program, másik számítógépes rendszer.



Operációs Rendszerek helye - Szoftver elemek

- Alapszoftverek: általános funkciókat biztosítanak
  - Hardver-közeli, pl. driver-ek
  - Hálózati, pl. kommunikációs
  - Operációs Rendszer (OS)
  - Segédprogramok: fordítók, egyszerű szöveg-editorok, adatbázis-kezelők
- Felhasználói v. applikációs programok: konkrét probléma megoldására született program.
  - Általános applikációk: táblázatkezelők, szövegszerkesztők, WEB böngésző, ... stb.
  - Célrendszerek: ETR, banki alkalmazások, ... stb.

Operációs Rendszerek helye



Operációs rendszerek alapvető feladatai

- A felhasználó kényelmének, védelmének biztosítása = Kiterjesztett (Virtuális) gép
  - Az emberi felhasználó számára kényelmes kezelési funkciók; jogosultságok kezelése
  - Programok számára futás közben használható eljárások, függvények; a futó program memóriaterületének védelme
- A rendszer hatékonyságának, teljesítményének maximalizálása = Erőforrás-kezelés
  - Hardver erőforrások: CPU (processzor), memória, I/O (Bemeneti/Kimeneti) eszközök, fájlkezelés, ... stb.
  - Szoftver erőforrások: eljárások, buffer-ek (átmeneti tárolók), üzenetek, szolgáltatások, ... stb.

Operációs rendszerek alapvető tulajdonságai

- Processzusok (folyamatok) az alapvető aktív alkotóelemek
  - Processzus (proc): végrehajtás alatt lévő program
  - Processzus létrejön (programbetöltés, végrehajtás indítása, ...), létezik ("fut"), majd megszűnik (exit)
- Egyidejűleg több processzus létezik: A processzor (CPU) idejét meg kell osztani az egyidejűleg létező procok között: időosztás (time sharing)
- Az erőforrások centralizált kezelése: Az erőforrásokat a processzusok (kérésükre) az OS-től kapják
- Esemény-vezérelt
  1. esemény
  2. megszakítás
  3. esemény feldolgozása
  4. esemény kiszolgálása
  5. visszatérés a megszakítás helyére

Operációs Rendszer Állatkert

- Nagyszámítógépes (mainframe) operációs rendszerek: Nagy I/O kapacitás, Sok job végrehajtása, Batch, tranzakciók feldolgozása. Pl: OS/390, UNIX, Linux
- Szerver operációs rendszerek: sok user, kozösen használt hw és sw erőforrások. Pl. Solaris, FreeBSD, Linux, Windows Server 200x
- Többprocesszoros operációs rendszerek, Pl. Windows, Linux
- PC-s operációs rendszerek, Pl. Linux, FreeBSD, Windows
- Handheld operációs rendszerek: PDA. Pl. Symbian OS, Palm OS
- Beágyazott operációs rendszerek: Tv, hűtő. Pl. QNX, VxWorks
- Érzékelő csomópontos operációs rendszerek: Tűz érzékelő. Pl: TinyOS
- Real-time operációs rendszerek: Hard, soft. Ipari robotok, digitális multimédia eszközök.
- Smart card operációs rendszerek: Java SmartCard

Generációk

- Az első generáció: Vákuumcsövek és kapcsolótáblák (1945-1955)
- A második generáció: Tranzisztorok és kötegelt rendszerek (1955-1965)
- A harmadik generáció: Integrált áramkörök és multiprogramozás (1965-1980)
- A negyedik generáció: Személyi számítógépek (1980-)

Az első generáció

Vákuumcsövek és kapcsolótáblák (1945-1955)

- Nincs operációs rendszer
- Howard Aiken, Neumann János, J. Presper Eckert, John William Machley, Konrad Zuse
- 1 felhasználó, 1 program, fix memóriacímek, megbízhatatlan hardver
- "Üresen indul a gép": bootstrap;
- Programozó = Gépkezelő = Programok végrehajtásának vezérlője
- Programozási nyelvek ismeretlenek (még assembly sem): bináris kódolás
- 1950-es évek eleje: lyukkártyák

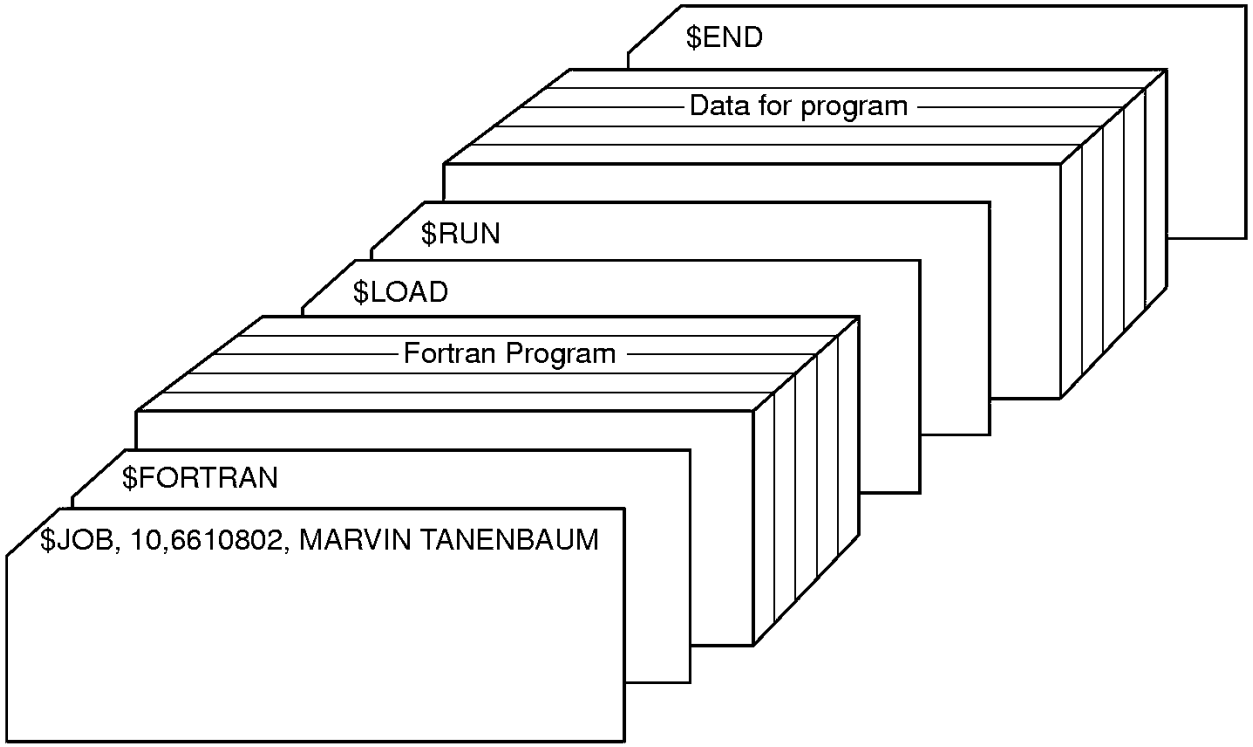
A második generáció

Tranzisztorok és kötegelt rendszerek (1955-1965)

- Operációs rendszer van, rezidens a memóriában
- Tranzisztorok – megbízható számítógépek
- Mainframe / nagyszámítógép
- Papíron írt programok – lyukkártya – kezelő - kávézás – eredmény – nyomtató – kiviteli terem
- Egyidejűleg csak 1 proc
- Feladat (job) sorozat (köteg) végrehajtását vezérli az OS $\rightarrow$ teljesítőképesség növekedett
- Programozók támogatására bevezetett megoldások:
  - verem-memória
  - eszközvezérlők
  - hívható OS eljárások
  - relatív címezhetőség
- FORTRAN, ALGOL programozás

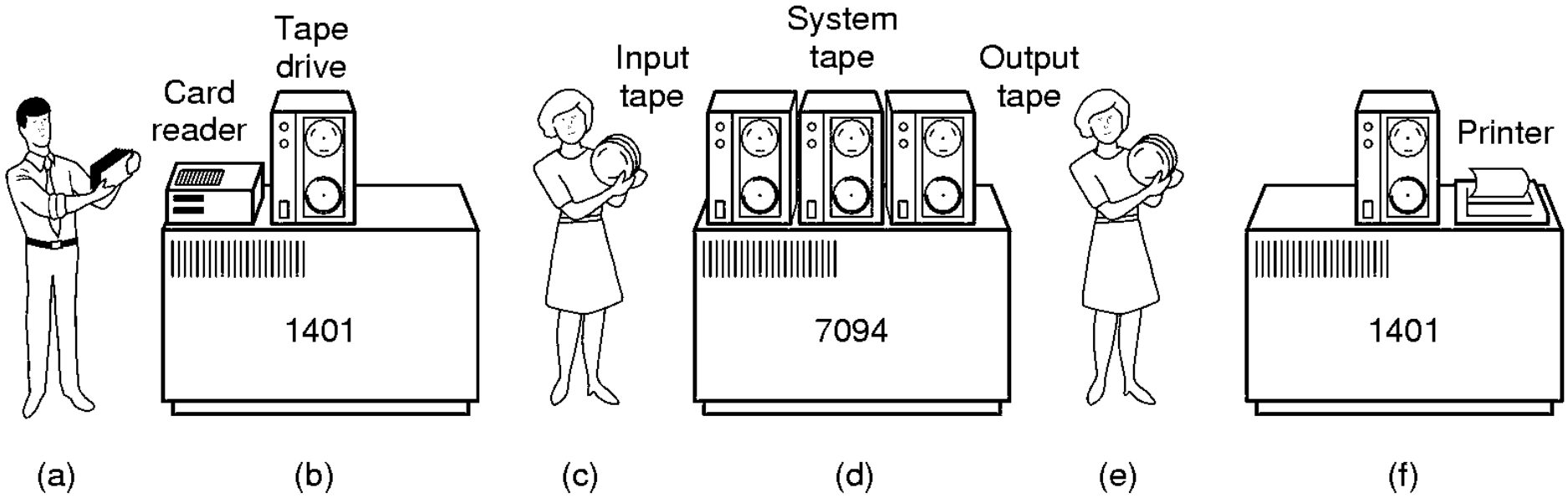
A második generáció

Egy szokásos FMS feladat



A második generáció

Egy korai kötegelt rendszer



(a) A programozó a kártyáit az 1401-eshez juttatja
(b) A feladatköteg szalagra olvasása
(c) A gépkezelő átviszi a bemeneti szalagot a 7094-eshez
(d) Számolás végrehajtása
(e) A gépkezelő átviszi a kimeneti szalagot a 1401-eshez
(f) Eredmény nyomtatása

A harmadik generáció

Integrált áramkörök és multiprogramozás (1965-1980)

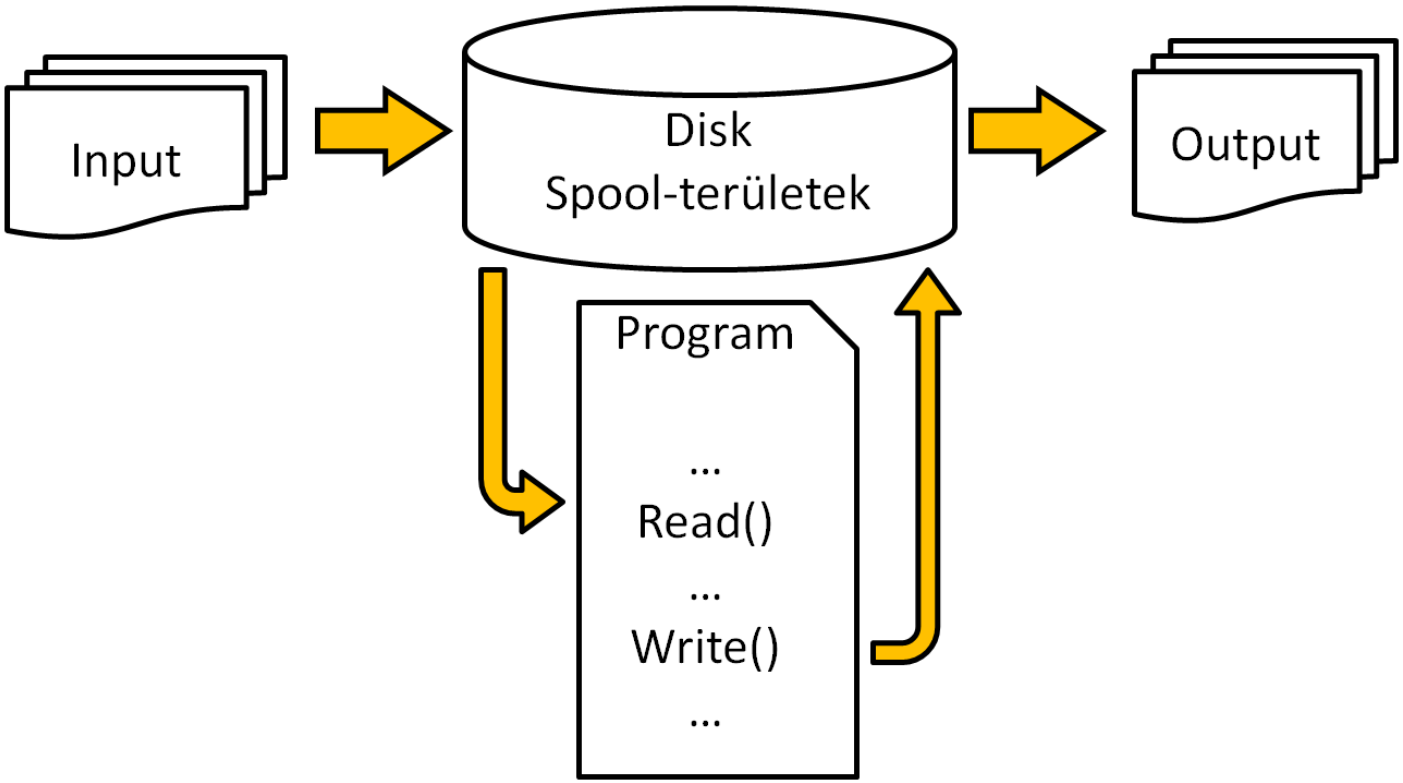
- Operációs rendszer van, rezidens/tranziens a memóriában
- CPU és I/O átfedően képes dolgozni - a CPU teljes idejének kihasználása csak több proc egyidejű létezésével lehetséges $\rightarrow$ multiprogramozás
- Több felhasználó: terminálok, on-line programvégrehajtás
- CPU időszeletelés (time slicing)
- Átmeneti tárolás (spooling)
- Memória-partíciók: minden partícióban egymástól függetlenül folynak job kötegek végrehajtásai
- COBOL, ADA, PASCAL, SIMULA, ... programozás

Időszeletelés - Time Slicing

- CPU időszeletelés: egy proc csak egy meghatározott maximális időintervallumon keresztül használhatja a CPU-t folyamatosan
- Legyen *tmax* az az időintervallum, amely alatt egy *p* proc folyamatosan birtokolhatja a CPU-t.
- Ha *tmax* letelik: az OS processzus-ütemező alrendszere átadja a CPU-t egy (másik) proc-nak; később *p* újra kap *tmax* intervallumot.
- *tmax* kb. 10-20 msec

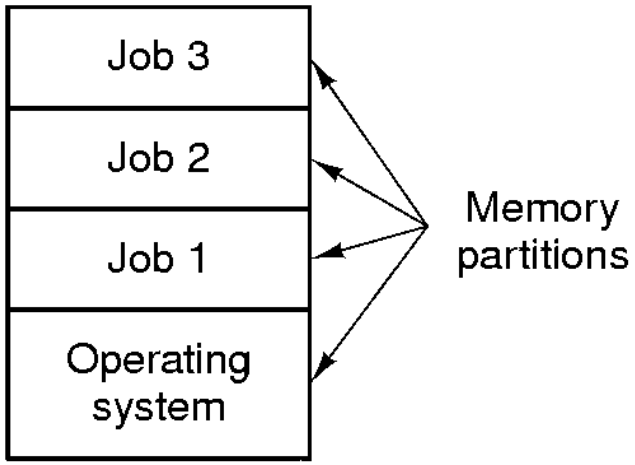
Átmeneti tárolás - Spooling

- az I/O adatok először gyors háttértárolóra (disk) kerülnek (spool terület), a proc innen kapja / ide írja az adatait
- jobb periféria (erőforrás)-kihasználás (olvasók, nyomtatók)



Memória-partíciók

- Statikus: OS indulásakor fix darabszámú (pl.16), fix méretű partíció képződik. Egy program végrehajtásához megkeres egy szabad és elegendő méretű partíciót, amelyben a program végrehajtását megkezdi.
- Dinamikus: OS egy program végrehajtásához a szabad memóriaterületből készít egy elegendő méretű partíciót, amelyben a program végrehajtását megkezdi – a partíciók száma és mérete változó



A negyedik generáció

Személyi számítógépek (1980-)

- Operációs rendszer van, több típus
- LSI áramkörök fejlődése
- Kategóriák
  - PC, Workstation: egyetlen felhasználó, egy időben több feladat (Pl. Windows, Mac OS, OS/2)
  - Hálózati OS: hálózaton keresztül több felhasználó kapcsolódik, minden felhasználó egy időben több feladatot futtathat (Pl. UNIX, LINUX, NT)
  - Osztott (hálózati) OS: egy feladatot egy időben több számítógépes rendszer végez. Erőforrások (CPU, Memória, HDD, ...) megosztása egy feladat elvégzéséhez. (pl. GRID, SETI)

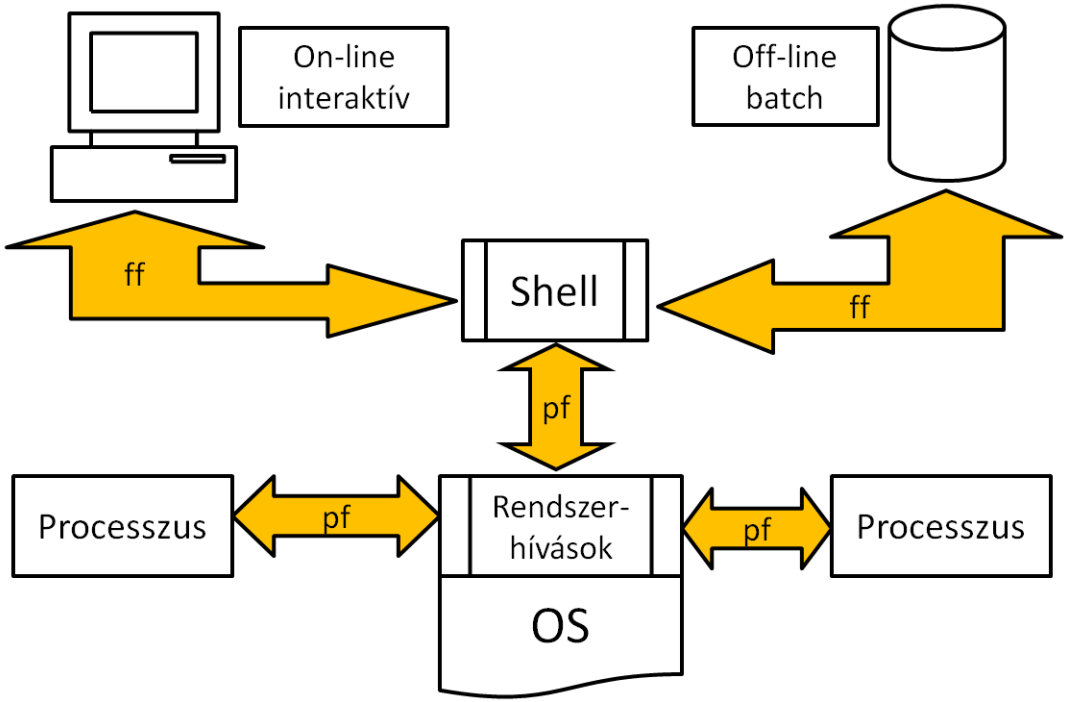
A negyedik generáció jellemzői

Személyi számítógépek (1980-)

- Alkalmazásfejlesztés segítése: Objektum-orientált és kliens-szerver módszerek.
- Hordozhatóság (hardver-függetlenség) segítése. Problémák: grafikus felületek alapjaiban különböznek.
- Egyes szolgáltatások leválasztása az OS magjáról, a szolgáltatások egymástól való függetlensége
- Spool-technika (cache) hardverrel segített és széles körben alkalmazott (RAM-ban is) adatokra, programokra egyaránt

OS felületei

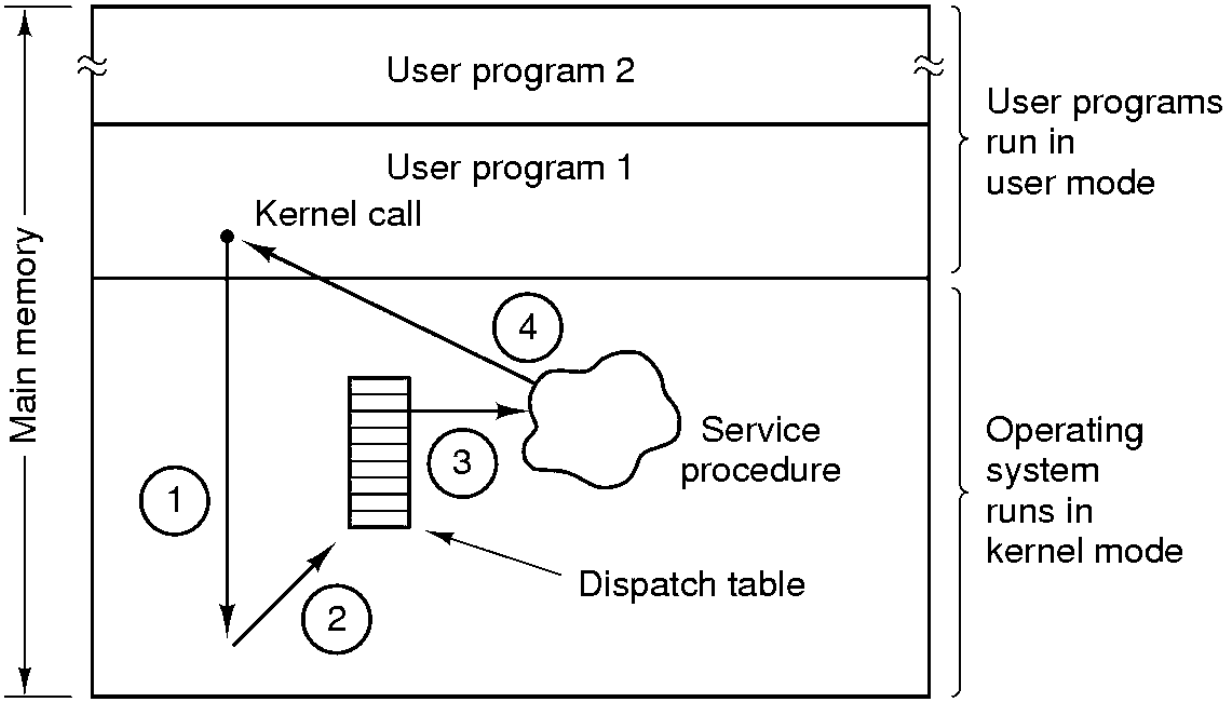
- **Felhasználói felület** (ff): a felhasználó (ember) és az OS kapcsolati alrendszer.
  Parancsértelmező (Command Interface, Shell); Grafikus felület
- **Program felület** (pf): proc és OS kapcsolati alrendszer
  Rendszerhívások (OS eljárások, függvények; system calls); OS szolgáltatások (services)



Rendszerhívások – System calls

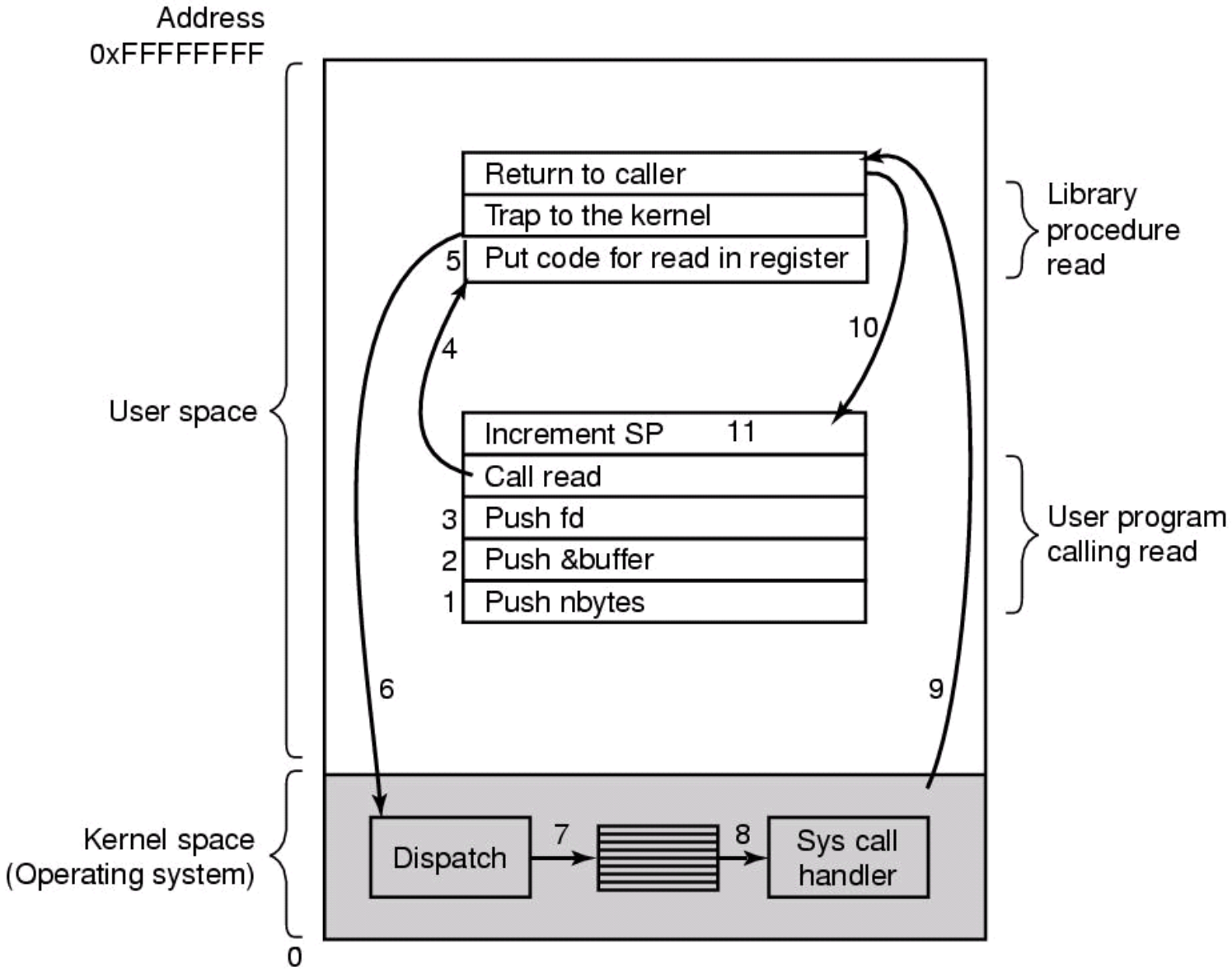
- Processzus kezelés (process management)
- Jelzések (signals), Események (events) kezelése
- Memóriakezelés (memory management)
- Fájl (file), Könyvtár (directory) és Fájlrendszer (file system) kezelés
- Védelem (protection) – adat, user, program, ...
- Üzenetkezelés (message handling)

Rendszerhívások – System calls



1. a hívó proc-nak joga volt-e
2. hívás paraméterek ellenőrzése
3. táblázat rendszer eljárások kezdő címét tárolja
4. vezérlés visszaadása a proc-nak

Példa: a read(fd, buffer, nbytes) rendszerhívása



Processzuskezelés

```
pid = fork();           // Create a child process identical to the parent
pid = waitpid(pid, &statloc, opts); // Wait for a child to terminate
s = wait(&status);      // Old version of waitpid
s = execve(name, argv, envp); // Replace a process core image
exit(status);           // Terminate process execution and return status
size = brk(addr);       // Set the size of the data segment
pid = getpid();         // Return the caller's process id
pid = getpgrp();        // Return the id of the caller's process group
pid = setsid();         // Create a new session and return its proc. group id
l = ptrace(req, pid, addr, data); // Used for debugging
```

Szignálkezelés

```
s = sigaction(sig, &act, &oldact); // Define action to take on signals
s = sigreturn(&context);           // Return from a signal
s = sigprocmask(how, &set, &old); // Examine or change the signal mask
s = sigpending(set);               // Get the set of blocked signals
s = sigsuspend(sigmask);           // Replace the signal mask and suspend the process
s = kill(pid, sig);                // Send a signal to a process
residual = alarm(seconds);         // Set the alarm clock
s = pause();                       // Suspend the caller until the next signal
```

Könyvtár- és fájlrendszerkezelés, védelem

```
s = mkdir(name, mode); // Create a new directory
s = rmdir(name);       // Remove an empty directory
s = link(name1, name2); // Create a new entry, name2, pointing to name1
s = unlink(name);      // Remove a directory entry
s = mount(special, name, flag); // Mount a file system
s = umount(special);   // Unmount a file system
s = sync();            // Flush all cached blocks to the disk
s = chdir(dirname);    // Change the working directory
s = chroot(dirname);   // Change the root directory

s = chmod(name, mode); // Change a file's protection bits
uid = getuid();        // Get the caller's uid
gid = getgid();        // Get the caller's gid
s = setuid(uid);       // Set the caller's uid
s = setgid(gid);       // Set the caller's gid
s = chown(name, owner, group); // Change a file's owner and group
oldmask = umask(complmode); // Change the mode mask
```

Fájlkezelés

```
fd = creat(name, mode);      // Obsolete way to create a new file
fd = mknod(name, mode, addr); // Create a regular, special, or directory i-node
fd = open(file, how, ...);   // Open a file for reading, writing or both
s = close(fd);               // Close an open file
n = read(fd, buffer, nbytes); // Read data from a file into a buffer
n = write(fd, buffer, nbytes); // Write data from a buffer into a file
pos = lseek(fd, offset, whence); // Move the file pointer
s = stat(name, &buf);        // Get a file's status information
s = fstat(fd, &buf);         // Get a file's status information
fd = dup(fd);                // Allocate a new file descriptor for an open file
s = pipe(&fd[0]);            // Create a pipe
s = ioctl(fd, request, argp); // Perform special operations on a file
s = access(name, amode);     // Check a file's accessibility
s = rename(old, new);        // Give a file a new name
s = fcntl(fd, cmd, ...);     // File locking and other operations
```

Rendszerhívások – System calls

A Win32 API hívások, amelyek nagyjából hasonlítanak a UNIX hívásokra

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

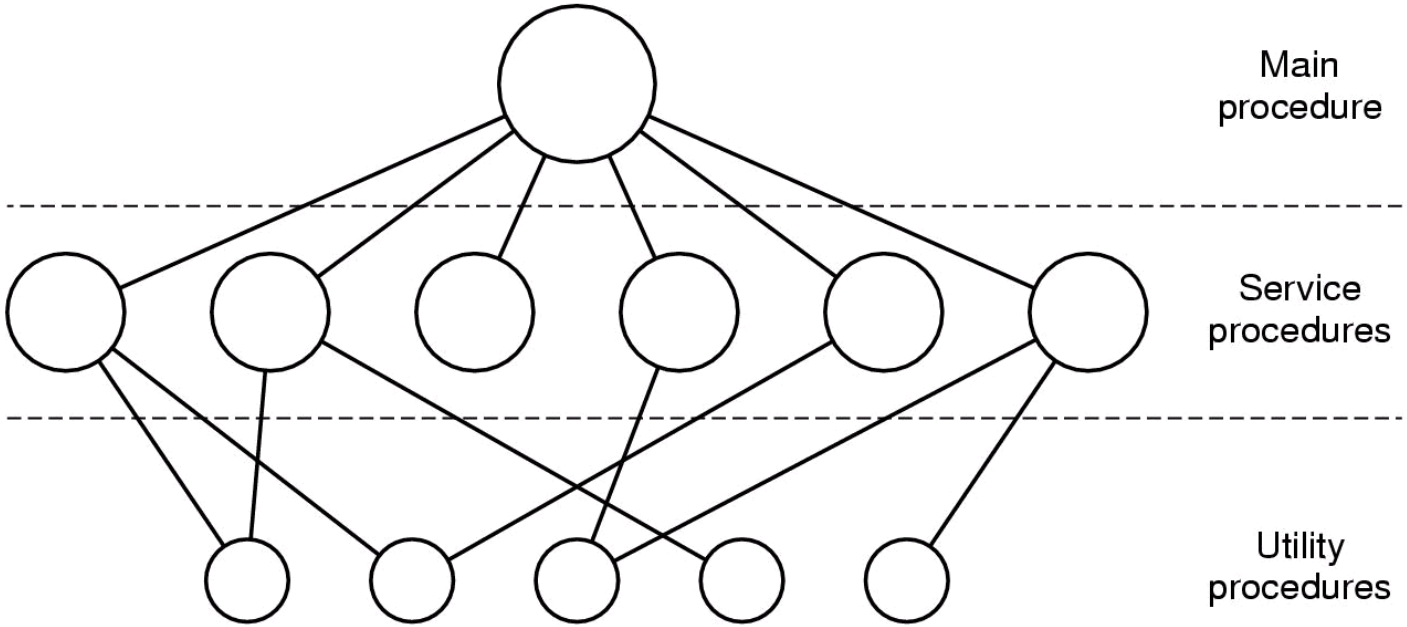
OS struktúrák

- Monolitikus rendszerek
- Rétegelt rendszerek
- Virtuális gépek
- Exokernelek
- Kliens szerver

Monolitikus rendszerek

A "Nagy összevisszaság"

- Legelterjedtebb szervezési mód. **Főprogram**: meghívja a kért szolgáltatás eljárását, **szolgáltató eljárások**: teljesítik a rendszerhívásokat, **segéd eljárások**: segítik a szolgáltatás eljárásokat.
- Struktúrája a strukturálatlanság
- Eljárások gyűjteménye, bármelyik hívhatja a másikat

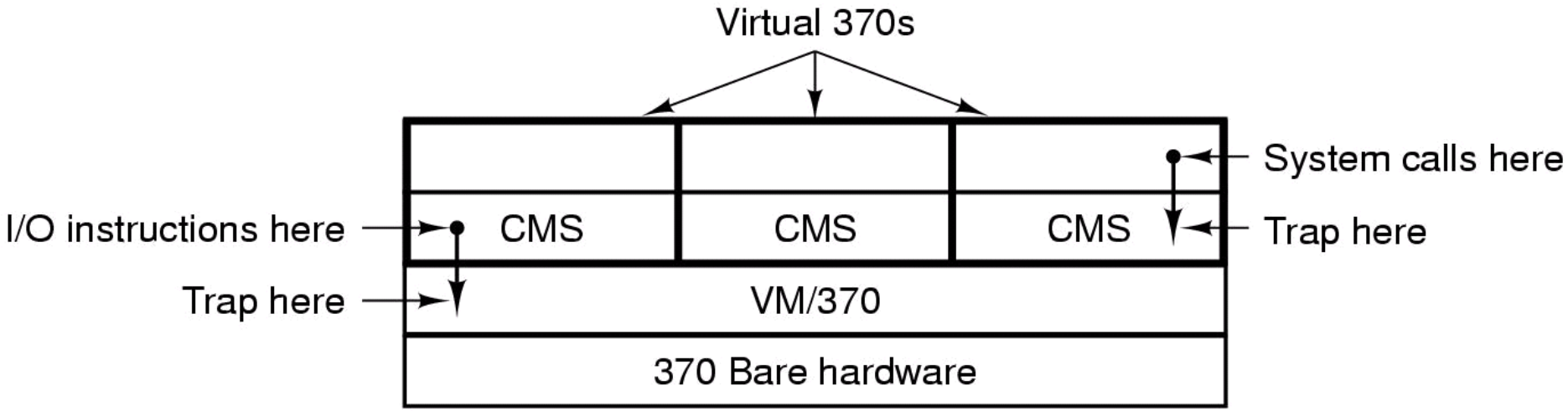


Rétegelt OS

- Eljárások, Függvények halmaza, amelyek rétegekbe csoportosulnak
- Felügyelt eljárás-hívások (supervisor calls)
- Programvégrehajtás-szintek: normál mód (az OS eljárások egy meghatározott részhalmaza hívható; felhasználói programok); kitüntetett (supervisor, kernel) mód(ok) (speciális OS eljárások is hívhatóak; rendszerprogramok)
- Szabály a rétegekre: bármely n-edik rétegbeli eljárás csak a közvetlen alatta lévő n-1-edik réteg eljárásait hívhatja - Áttekinthető eljárásrendszer alakítható ki
- A rétegek saját (globális) adatterülettel rendelkezhetnek - Áttekinthető adatáramlás
- Konfigurálható: pl. az eszközvezérlők egy rétegbe csoportosíthatóak
- Rezidens, Tranziens rétegek
- Egy réteg = Virtuális (absztrakt) gép: a rétegbeli függvények által nyújtott funkciók összessége (pl. IBM VM/370)
- 1-rétegű ("monolitikus"), 2-, 3-, ... rétegű

Virtuális gépek

- Virtuális gép monitor a nyers hardveren fut és akár több virtuális gépet is szolgáltat.
- A virtuális gépek különbözőek is lehetnek.
- Pentium processzorokba beépítettek egy virtuális 8086 módot.
- VMWare, Microsoft Virtual PC, Virtual Box
- Java Virtual Machine

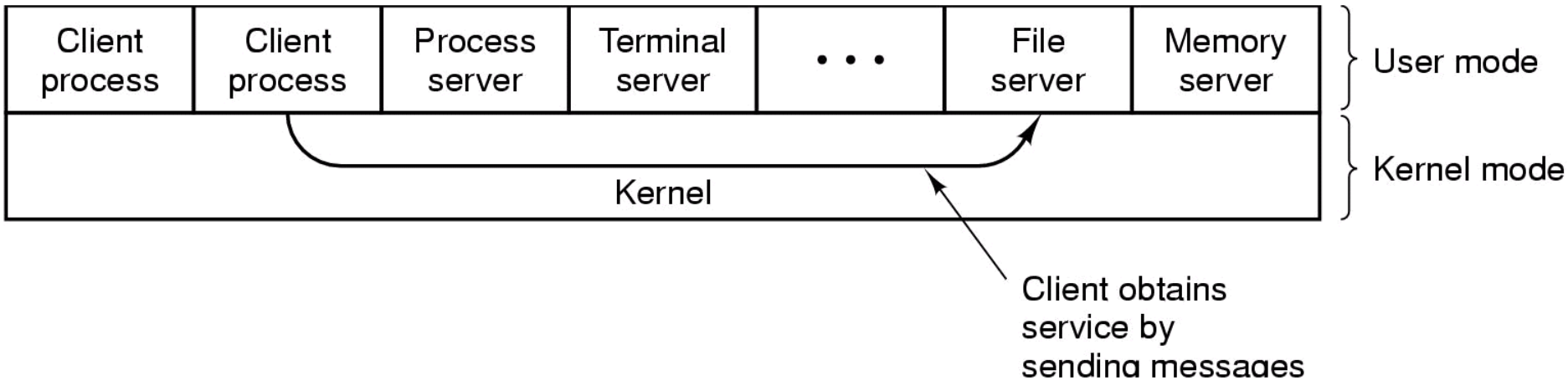


Kliens/Szerver

- Egymással "kommunikáló" processzusok rendszere
- Kommunikáció: üzenet adás/vétel
- Szerver proc: meghatározott szolgáltatásokat végez: terminál-, web-, file i/o-, szolgáltatás-csoportok
- Kliens proc: egy szerver proc szolgáltatásait veszi igénybe; kérés = üzenet (tartalma specifikálja a pontos kérést): terminál-kliens ; válasz = üzenet
- Üzenet = fejléc (header) + tartalom (body)
- Header: az üzenet továbbításához szükséges információk; címzett (proc), feladó, hossz, kódolás, idő-bélyegzők, ...
- Tartalom: feladó állítja össze, címzett értelmezi; szerkezete pontosan specifikált = kommunikációs protokoll OS feladata: az üzenetek közvetítése processzusok között; csak a fejléccel kell foglalkoznia

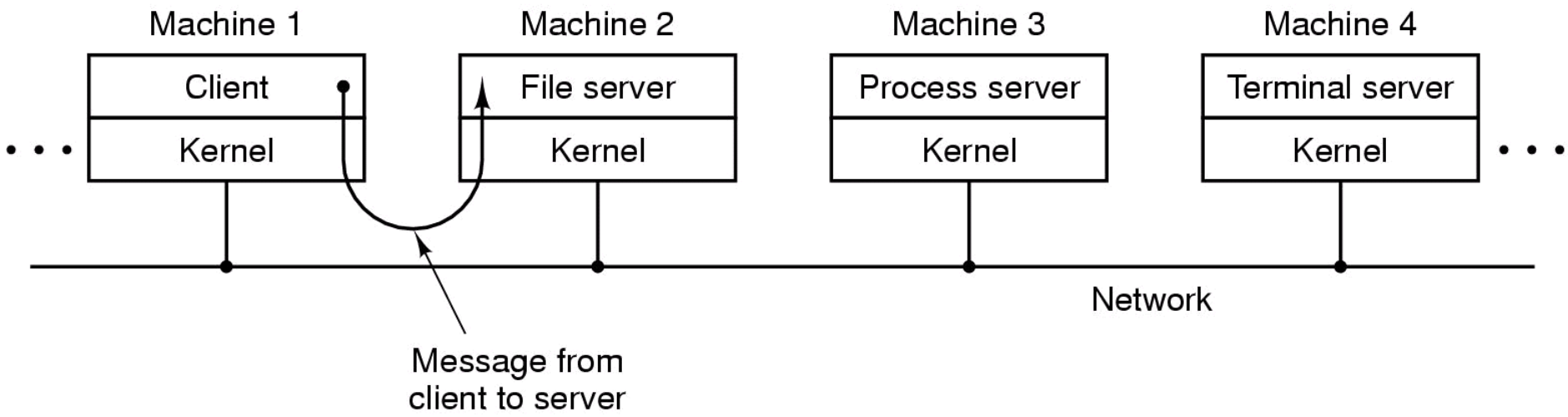
Kliens/szerver modell előnyei

- Szerver processzusokkal az OS szolgáltatásai bármikor bővíthetők – az OS eljárásokra épülő része (kernel) szűkebbre vehető
- Szerver programok egymástól teljesen függetlenül programozhatóak
- Hálózaton keresztül is alkalmazható: szerver és kliens proc külön OS fölött fut, ezek hálózaton keresztül cserélnek üzeneteket



Kliens/szerver modell osztott rendszerekben

Kliensnek nem kell tudnia, hogy a szerver lokálisan, vagy hálózaton keresztül lesz elérhető



Kliens/szerver programozás

- 2-, 3-rétegű alkalmazás-szerkezet: a kliens gépen kliens proc fut, a szükséges szolgáltatásokat (hálózaton keresztül) szerver gép(ek)től üzenetek formájában kéri
- Kliens gép feladata: képernyő kezelése, felhasználó akcióinak megfelelő kérések továbbítása, válaszok fogadása
- X-terminál, Adatbázis-szerver, WEB, Fájl-megosztás, ...
- ASP – Application Service Provider – Távoli Alkalmazás Kiszolgáló

Operációs rendszerek alrendszerei

- Processzuskezelő alrendszer; process handling
- Erőforráskezelő alrendszer; resource handling (Kölcsönös kizárás, Szinkronizáció, Üzenetközvetítés)
- Memóriakezelő alrendszer (spec erőforrás); memory management
- Bevitel/Kivitel; Input/Output
- Fájlkezelő alrendszer (spec erőforrás); file, directory and file system management
- Időkezelés; time management (spec esemény)
- Védelmi (protection), biztonsági (security) rendszer
- Jelkezelő alrendszer; signal handling
- Eseménykezelő alrendszer; event handling
- Felhasználói felület(ek); parancsértelmező (shell)

Processzusok

Processzus

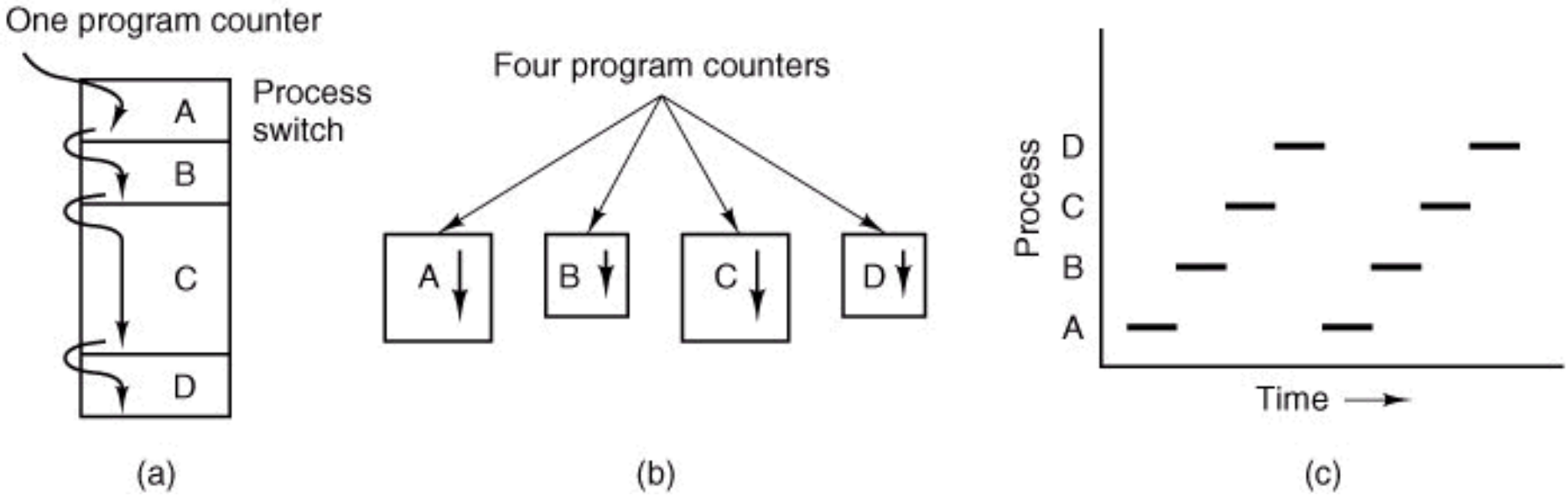
Szekvenciálisan végrehajtódó program.

- Erőforrásbirtokló (resources) egység; kizárólagosan és osztottan használt erőforrások
- Címtartománnyal (address space, memory map) rendelkezik; saját és (más procokkal) osztott memória (private/shared memory)
- Végrehajtási állapota (state) van; IP (USz), Regiszterek, SP, SR, ...
- Veremmemóriája (stack) van; saját
- Jogosultságokkal (privileges) rendelkezik; system/appl proc, adathozzáférési jogok

Kontextus csere

- Több egyidejűleg létező processzus - Egyetlen processzor (CPU): A CPU váltakozva hajtja végre a procok programjait (monoprocesszoros rendszer)
- Kontextus csere (Context Switching): A CPU átvált a P1 procról a P2 procra
- P1 állapotát a CPU (hardver) regisztereiből menteni kell az erre a célra fenntartott memóriaterületre; IP, SP, stb.
- P2 korábban memóriába mentett állapotát helyre kell állítani a CPU regisztereiben
- Többprocesszoros (multiprocesszoros) rendszerben is szükséges (n proc, m CPU). Mindegyik CPU-n végre kell hajtani a cserét.

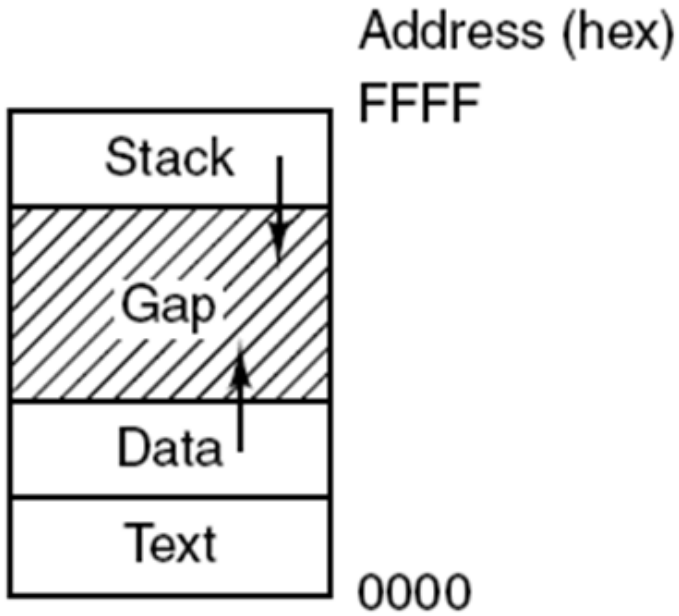
Processzusmodell



(a) Négy program multiprogramozása. (b) Négy független, szekvenciális processzus elméleti modellje. (c) Minden időpillanatban csak egy program aktív.

Memóriatérkép

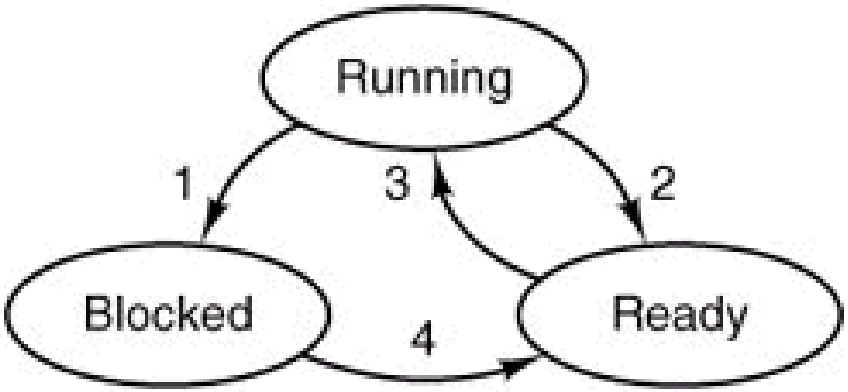
- Programszöveg: Konstans (text)
- Adat: Konstansok (text), Változók
- Verem (adat + vezérlési információk): Lokális (eljáráson belül érvényes) változók; Last-in-First-out (LIFO) kezelés; Push/Pop
- Dinamikus változók: Heap: new/delete (C++); new/dispose (Pascal)



Processzusok állapotai

- **Futó** (Running): A proc birtokolja a CPU-t.
- **Futáskész** (Ready): készen áll a futásra, de ideiglenesen leállították, hogy egy másik processzus futhasson.
- **Blokkolt** (Blocked): erőforrásra várakozik, pl. egy input művelet eredményére
- Segédállapotok: iniciális (megjelenéskor), terminális (befejezéskor), aktív, felfüggesztett (hosszabb időre felfüggesztés, pl. memória szűkössége miatt).

Processzusok állapotátmenetei



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

1. A processzus bemeneti adatra várva blokkol
2. Az ütemező másik processzust szemelt ki
3. Az ütemező ezt a processzust szemelte ki
4. A bemeneti adat elérhető

Processzus leírása

Processzustáblázat

A proc nyilvántartására, tulajdonságainak leírására szolgáló memóriaterület (Proc Table, Proc Control Block, PCB)

Tartalma

- Azonosító – procid: egyértelmű, hivatkozási sorszám
- Létrehozó proc azonosítója; procok fastruktúrát alkotnak
- Memóriatérkép; létrejöttekor, később változik
- Állapot/Alállapot; időben változik
- Jogosultságok, prioritás; létrehozó állítja be
- Birtokolt erőforrások; kizárólagosan, osztottan használható erőforrások; pl. a nyitott fájlok nyilvántartása
- Kért, de még meg nem kapott erőforrások
- CPU állapot kontextuscseréhez; Usz, Verempointerek, Regiszterek, ... tartalmának tárolására
- Számlázási, statisztikai információk...

Processzus leírása

Az egyidejűleg létező procok leírásai (PCB-k) a processzus leírások láncára vannak fűzve.
Proc létrejöttekor a PCB a láncra fűződik (pl. a végére). Proc megszűntekor a PCB láncszem törlődik a láncról.
Műveletek a láncon

- Egy proc gyermekeinek keresése pl. a szülő megszűnésekor a fastruktúra megőrzéséhez
- Futáskész állapotú procok keresése pl. a következő időintervallumra a CPU kiosztásához
- Erőforrást igényelt processzusok keresése pl. az erőforrás odaítélésekor

A processzus tábla (PCB) néhány tipikus mezője

Process management	**Memory management**	**File management**
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Processzus létrehozásának lépései

1. Memóriaterület foglalása a PCB számára
2. PCB kitöltése iniciális adatokkal; kezdeti memóriatérkép, állapot = iniciális, kizárólagosan használható erőforrások lefoglalása, CPU állapot USz = indítási cím, jogosultságok a létrehozó jogosultságaiból,...
3. Programszöveg, adatok, verem számára memóriafoglalás, betöltés (ha kell; ezt általában a létrehozó feladatának tekinti az OS)
4. A PCB procok láncára fűzése, állapot = futáskész. Ettől kezdve a proc osztozik a CPU-n.

Processzus létrehozása - C (UNIX)

int fork(void);

A létrehozó proc tökéletes másolatát készíti el. A fork() után a két proc ugyanazzal a memóriaképpel, környezeti sztringekkel, nyitott fájlokkal fog rendelkezni.

Mindkét proc programja közvetlenül a fork() után folytatódik, de elágazás programozható:

- pid=fork() $>$ 0, akkor a szülőről van szó, pid a gyermek azonosítója
- pid=fork() == 0, akkor a gyermekről van szó
- pid=fork() $<$ 0, akkor error, sikertelen a gyermek létrehozása

Processzus létrehozása - C (UNIX)

Programvázlat

```
int pid = fork();
if (pid < 0) { exit(1); } // hiba
if (pid == 0) { exec(); } // gyermek sikeres
if (pid > 0) {...}        // szülő folytatja munkáját
```

Alkalmazás

Szülő – Gyermek egy csövön (pipe) keresztül kommunikál; szülő egy szöveget ír a csőbe, gyermek a szöveget feldolgozza (transzformálja), pl. makro-processzorként

Processzusok befejezése

- Szabályos kilépés (exit(0))
- Kilépés hiba miatt hibakód jelzéssel a létrehozó felé (exit(hibakód))
- Kilépés végzetes hiba miatt (Pl. illegális utasítás, nullával való osztás, hivatkozás nem létező memóriacímre, stb.)
- Egy másik processzus megsemmisíti (kill() + jogosultságok)

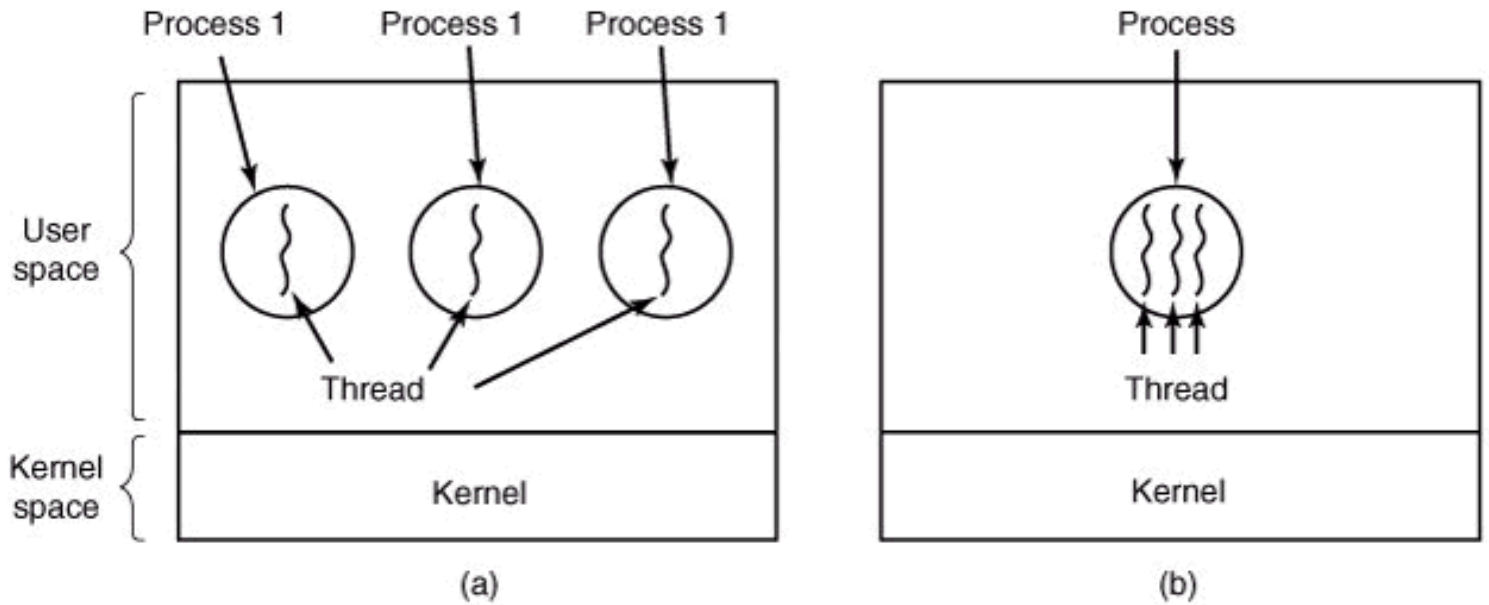
Processzus megszüntetése

A megszüntetés lépései (a létrehozás fordított sorrendjében):

1. A gyermek procok megszüntetése (rekurzív) vagy más szülőhöz csatolása
2. A PCB procok láncáról levétele, állapot = terminális. Ettől kezdve a proc nem osztozik a CPU-n.
3. A megszűnéskor a proc birtokában lévő erőforrások felszabadítása (a PCB-beli nyilvántartás szerint, pl. nyitott fájlok lezárása)
4. A memóriatérképnek megfelelő memóriaterületek felszabadítása
5. A PCB memóriaterületének felszabadítása

Szálak / Fonalak

- Szál (thread, lightweight proc) = szekvenciálisan végrehajtódó program, proc hozza létre
- De osztozik a létrehozó proc erőforrásain, címtartományán, jogosultságain
- Viszont van saját állapota, verme
- Kezelése az OS részéről a procnál egyszerűbb, jelentős részben a programozó felelőssége



Processzusok és szálak

Processzus elemei

Címtartomány
Globális változók
Megnyitott fájlok
Gyermekprocesszusok
Függőben lévő ébresztők
Szignálok és szignálkezelők
Elszámolási információ

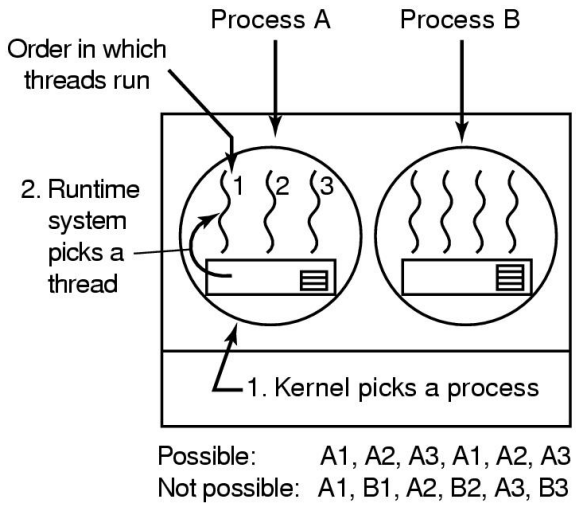
Szál elemei

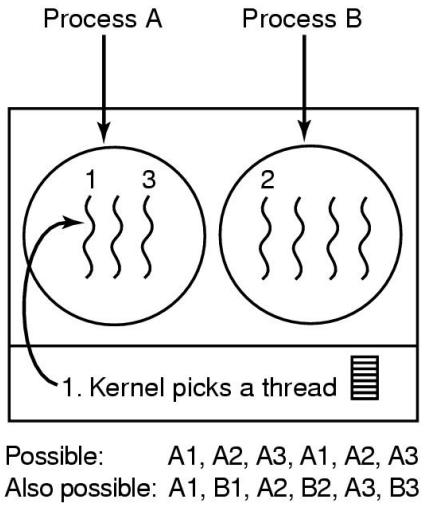
Utasításszámláló
Regiszterek
Verem
Állapot

Szálak megvalósítása

A **felhasználó** kezeli a szálakat egy szál-csomag (függvénykönyvtár) segítségével. A kernel nem tud semmit a szálakról. Megvalósítható olyan operációs rendszeren is, amely nem támogatja a szálakat.

A **kernel** tud a szálakról és ő kezeli azokat: szál-táblázat a kernelben; szálak létrehozása, megszűntetése kernelhívásokkal történik. Szál blokkolódása esetén, az OS egy másik futáskész szálat választ, de nem biztos, hogy ugyanabból a processzusból.

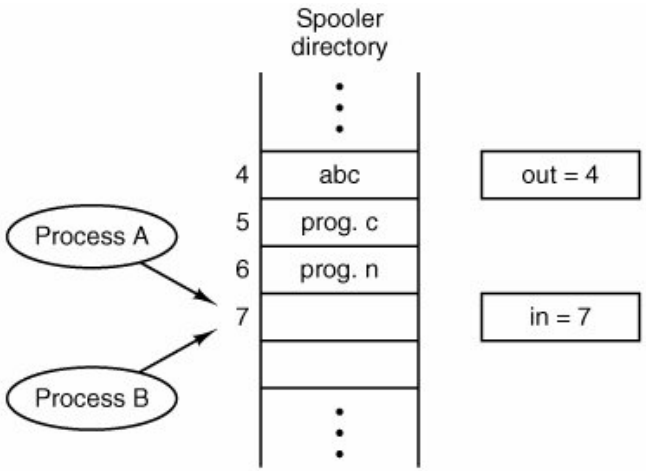




Versenyhelyzetek

Példa: háttérnyomtatás. Ha egy processzus nyomtatni akar, beteszi a fájl nevét egy háttérkatalógusba. A nyomtató démon ezt rendszeresen ellenőrzi, és elvégzi a nyomtatást, majd törli a bejegyzést.

1. *A* processzus kiolvassa az *in* közös változót és eltárolja lokálisan
2. Óramegszakítás következik be, és *B* processzus kap CPU-t
3. *B* processzus kiolvassa *in*-t, beleteszi a fájl nevét, növeli *in*-t, teszi a dolgát...
4. *A* ismét futni kezd, beleteszi a kiolvasott rekeszbe a fájl nevét, megnöveli *in*-t



Versenyhelyzet (race condition)

Processzusok közös adatokat olvasnak és a végeredmény attól függ, hogy ki és pontosan mikor fut.

Kölcsönös kizárás

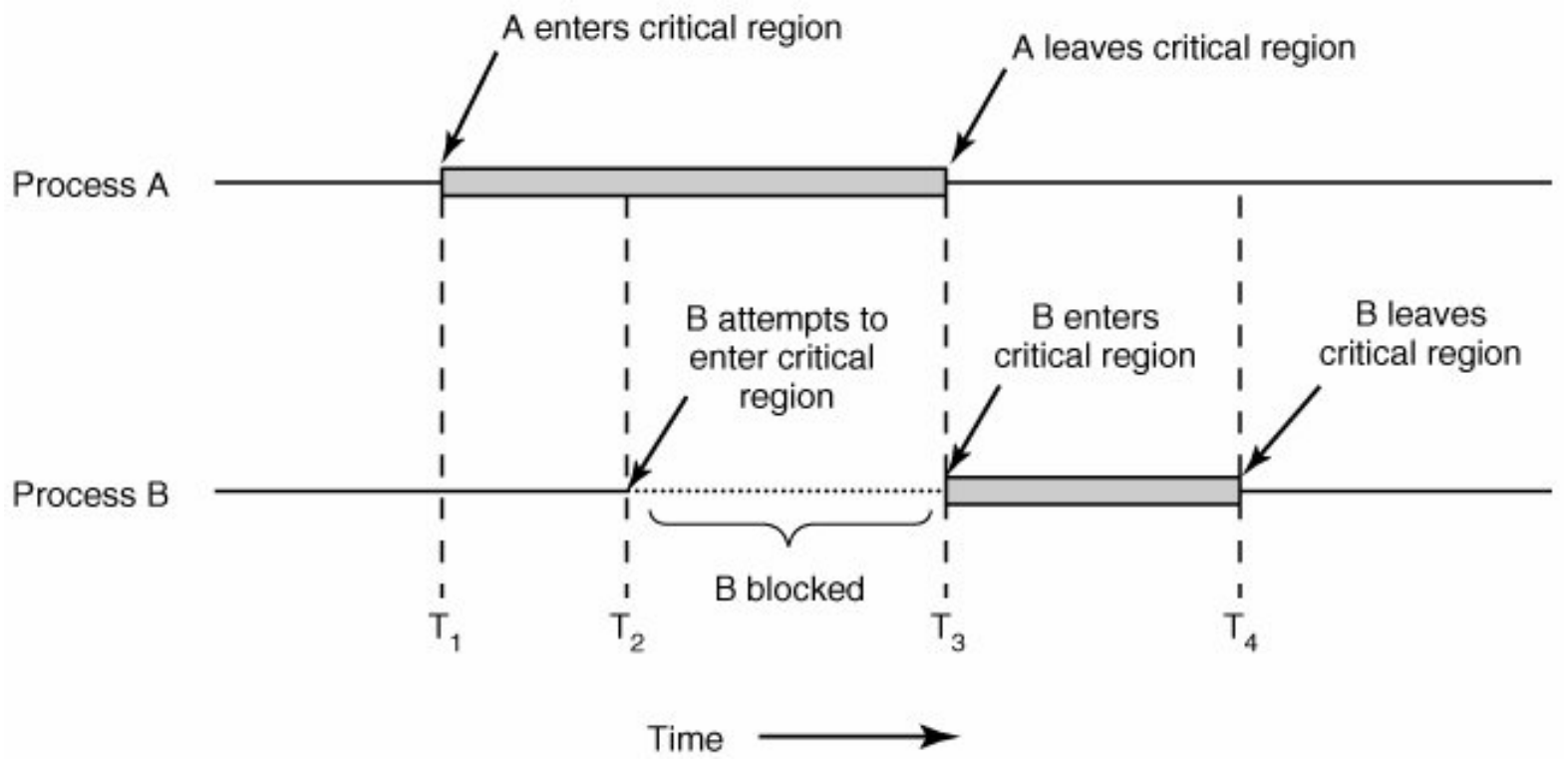
Kölcsönös kizárás (mutual exclusion): Ha egy processzus használ egy megosztott erőforrást, akkor a többi processzus tartózkodjon ettől.

Kettő vagy több processzus egy-egy szakasza nem lehet átfedő: p1 proc k1 szakasza és p2 proc k2 szakasza átfedően nem végrehajtható. k1 és k2 egymásra nézve **kritikus szekciók**.

Szabályok:

1. Legfeljebb egy proc lehet kritikus szekciójában
2. Kritikus szekción kívüli proc nem befolyásolhatja másik proc kritikus szekcióba lépését
3. Véges időn belül bármely kritikus szekcióba lépni kívánó proc beléphet
4. A processzusok "sebessége" közömbös

Kölcsönös kizárás kezelése kritikus szekciókkal

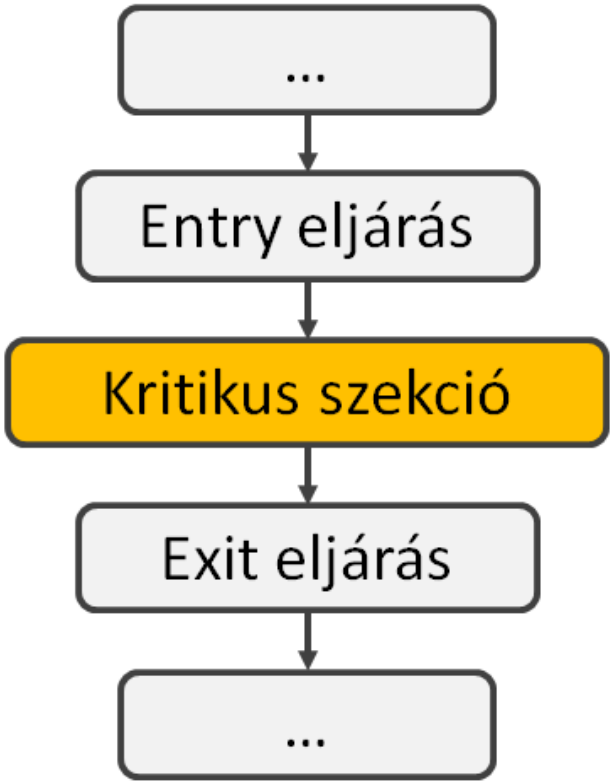


A kölcsönös kizárás megoldása egyértelműen az operációs rendszer feladata.

Entry-Exit módszerek

Entry eljárás: a kritikus szekció előtt kell meghívnia a programozónak. Igény az OS felé, hogy a proc be akar lépni. Az OS majd eldönti, hogy mikor enged be.

Exit eljárás: kilépés után kell meghívni. Jelzés az OS felé, hogy véget ért a kritikus szekció.



Entry-Exit módszerek - Hardver módszerek

Interruptok tiltása

```
Entry:  Disable (); // minden megszakítás tiltása
    k:  ...
 Exit:  Enable (); // megszakítás engedélyezése
```

A k kritikus szekcióban a processzus ütemező nem jut CPU-hoz, így p-től nem veheti el a CPU-t.
k csak rövid lehet, mert p nem sajátíthatja ki a CPU-t hosszú időre.
Nemcsak az ütemező, hanem pl. az esemény-kiszolgáló procok sem jutnak CPU-hoz – veszélyes.

Entry-Exit módszerek - Hardver módszerek

Test and Set Lock - TSL utasítás

TSL rx, lock

Beolvassa a lock memóriaszó tartalmát rx-be, és nem nulla értéket ír erre a memóriacímre úgy, hogy az utasítás befejezéséig más processzor nem érheti el a memóriát.

```
Entry:  TSL reg, lock  // reg:=lock; lock:=1
        CMP reg, #0
        JNE Entry
    k:  ...
Exit:   MOV lock,0     // lock:=0
```

Entry-Exit módszerek - Peterson "udvariassági" módszere

1981

```
i = 1,2; j = 2,1;
int Turn;                    // közös változó
int MyFlag_i = 0;            // saját változó

Entry_i : MyFlag_i = 1; // pi be akar lépni
          Turn = j;     // pj beléphet, ha akar
          wait while(Turn == j and MyFlag_j);
    k_i :  ...
 Exit_i : MyFlag_i = 0; // pi kilépett
          ...
```

- Tfh. egy sor végrehajtása közben a proc a CPU-t nem veszíti el.
- Elv: a proc jelzi, hogy ő szeretne belépni, de előre engedi a másikat
- Hátránya: csak 2 processzus eseten működik, közös változók

Entry-Exit módszerek - Peterson módszere

Megvalósítás C-ben

```
#define N     2                       // number of processes
int  turn;                            // whose turn is it?
int  interested [N];                  // all values initially 0 (FALSE)

void enter_region (int process) {// process is 0 or 1
  int other;                          // number of the other process
  other = 1 - process;                // the opposite of process
  interested [process] = TRUE; // show that you are interested
  turn = process;                     // set flag
  while (turn == process && interested[other] == TRUE) ;
}

void leave_region (int process) {// process: who is leaving
  interested [process] = FALSE; // indicate departure from crit. reg.
}
```

Entry-Exit módszerek - Lamport "sorszám" módszere

1974

```
i = 1,2, ...;
int N = 1;                  // sorszám-tömb
int MyNo_i = 0;             // saját sorszám-változó

Entry_i : MyNo_i = N++; //pi egyedi belépési sorszámot kap
          wait while(MyNo_i != min_j(MyNo_j, MyNo_j != 0));
      ki : ...

 Exit_i : MyNo_i = 0;   // pi kilépett
          ...
```

- Elv: Minden proc húz egy sorszámot. Belépéskor addig vár, amíg a sorszáma a legkisebb nem lesz a ki nem szolgáltak közül.
- Előnye: tetszőleges számú proc lehet, nem kell a többi proc változóit piszkálni
- Megvalósítás rendszerhívásokkal, és blokkolással

Entry-Exit módszerek - Dijkstra féle bináris szemafor

```
int S = 1;              // globális szemafor változó
Entry: P(S): wait while(S == 0); S = 0;

    k: ...

Exit: V(S): S = 1; // p kilepett
```

- Elv: Tevékeny várakozás amíg S értéke 0. Amint S nem nulla, azonnal nullára változtatja és utána lép be a kritikus szekcióba.
- Előnye: akárhány processzusra működik, egy globális változó, i-től független.
- Hátránya: várakozás CPU időt igényel
- Megvalósítás rendszerhívásokkal, és blokkolással

Entry-Exit módszerek - Szemafor

Általánosított eset

Legfeljebb n proc lehet egyidejűleg kritikus szekciójában

```
int S = n;          // globális szemafor változó

P(S): wait while(S == 0); S = S - 1;

V(S): S = S + 1;  // p kilépett
```

A **mutex** egy olyan változó, amely kétféle állapotban lehet: zárolt, vagy nem zárolt. Bináris szemafor.

Entry-Exit módszerek - Szemafor

Altatás - ébresztés (Sleep - Wakeup) implementáció

```
Entry: if (S == 0) {   // sleep
         insert (MySelf) into Queue_S;
         block(MySelf);
       }
       else S = 0;
  ki:  ...
Exit:  if (Queue_S is Empty) S = 1;
       else { // wakeup p from Queue_S
         p = remove(head(Queue_S));
         ready(p);
       }
```

- Az Entry és Exit alatt a proc nem veszítheti el a CPU-t.
- Elv: Ha a szemafor nulla, akkor a proc beilleszti magát a várakozó processzusok sorába, és blokkolja is magát.
- Előnye: nincs aktív várakozás
- Hátránya: a másik processzuson múlik, hogy felébred-e az alvó.

Hoare monitor

1974

Monitor

Eljárások, változó és adatszerkezetek együttese, egy speciális modulba összegyűjtve, hogy használható legyen a kölcsönös kizárás megvalósítására.

- Minden időpillanatban csak egyetlen processzus lehet aktív a monitorban.
- A processzusok hívhatják a monitor eljárásait, de nem érhetik el a belső adatszerkezetét.
- Programozási nyelvi konstrukció, azaz már a fordító tudja, hogy speciálisan kell kezelni.
- Megvalósítása a fordító programtól függ. Ált. mutex vagy szemafor.

Hoare monitor

1974

```
monitor example
   int i;          // globális minden eljárásra
   condition c;  // állapot- (feltétel-) változó
   producer(x) {
      ...    wait(c);   ...
   }
   consumer(x) {
     ...   signal(c);   ...
   }
end monitor;
```

- **wait(c)**: alvó (blokkolt) állapotba kerül a végrehajtó proc.
- **signal(c)**: a c miatt alvó procot (procokat) felébreszti. signal(c) csak eljárás utolsó utasítása lehet, végrehajtásával kilép az eljárásból.
- Az eljárástörzsek egymásra nézve kritikus szekciók

Processzusok kommunikációja

Processzusok együttműködésének módszerei

- Adatok cseréje (csővezeték – pipe, fájlok, adatbázis)
- Üzenetek adása – vétele
- Közös adatterületek (osztott memória – shared memory)
- Kliens – szerver modell

Csővezeték – pipe

```
pipeline(char *process1, char *process2) { // pointers to program names
  int fd[2];                    // pipe vector as two file descriptors
  pipe(&fd[0]);                 // create a pipe
  if (fork() != 0) {            // the parent process executes these statements
    close(fd[0]);               // process 1 does not need to read from pipe
    close(STD_OUT);             // prepare for new standard output
    dup(fd[1]);                 // set standard output to fd[1]
    close(fd[1]);               // this file descriptor not needed any more
    execl(process1, process1, 0); // calling process finished
  }
  else {                        // the child process executes these statements
    close(fd[1]);               // process 2 does not need to write to pipe
    close(STD_IN);              // prepare for new standard input
    dup(fd[0]);                 // set standard input to fd[0]
    close(fd[0]);               // this file descriptor not needed any more
    execl(process2, process2, 0); // calling process finished
  }
}
```

Üzenetportok

Az üzenetek egy láncra (FIFO) fűzve tárolódnak
Lánc feje: üzenetport – MsgPort

```
typedef struct {
  Message *head;          // első üzenetre mutató pointer
  int type;               // a port/üzenetek típusa
  char *name;             // üzenetport neve
  MsgPort *reply_port;    // esetleg a válasz-port címe
} MsgPort;
```

Láncszem: üzenet – Message

```
typedef struct {
  Message *next;          // a láncon következő üzenet címe
  int lenght;             // az üzenet (teljes) hossza
  char message_text[1];   // az üzenet (tényleges) szövege
} Message;
```

Üzenetportok

Műveletek:

```
MsgPort *CreatePort(char *name, int type = 0, MsgPort *reply_port = 0);
MsgPort *FindPort(char *name);

void SendMessage(MsgPort *p, Message *m);
Message * ReceiveMessage(MsgPort *p); // blokkol, ha p üres

int  TestPort(MsgPort *p);                    // = 0, ha p üres
```

Üzenetport használata

Adó proc

```
...
MsgPort *mp = CreatePort("portom"); // port létrejön
Message m = ...;                    // üzenet tartalmának kitöltése
SendMessage(mp, &m);                // üzenet elküldése
...
```

Vevő proc

```
...
MsgPort *mp = FindPort("portom"); // port megkeresése
Message *m = ReceiveMessage(mp);  // üzenet olvasása
if (m) {
    ...;                          // üzenet feldolgozása
}
...
```

Memóriafoglalási problémák lehetnek! Pl. Adó foglal, vevő felszabadít, vagy objektumorientált osztályok használatával.

Ciklikus üzenetbufferek

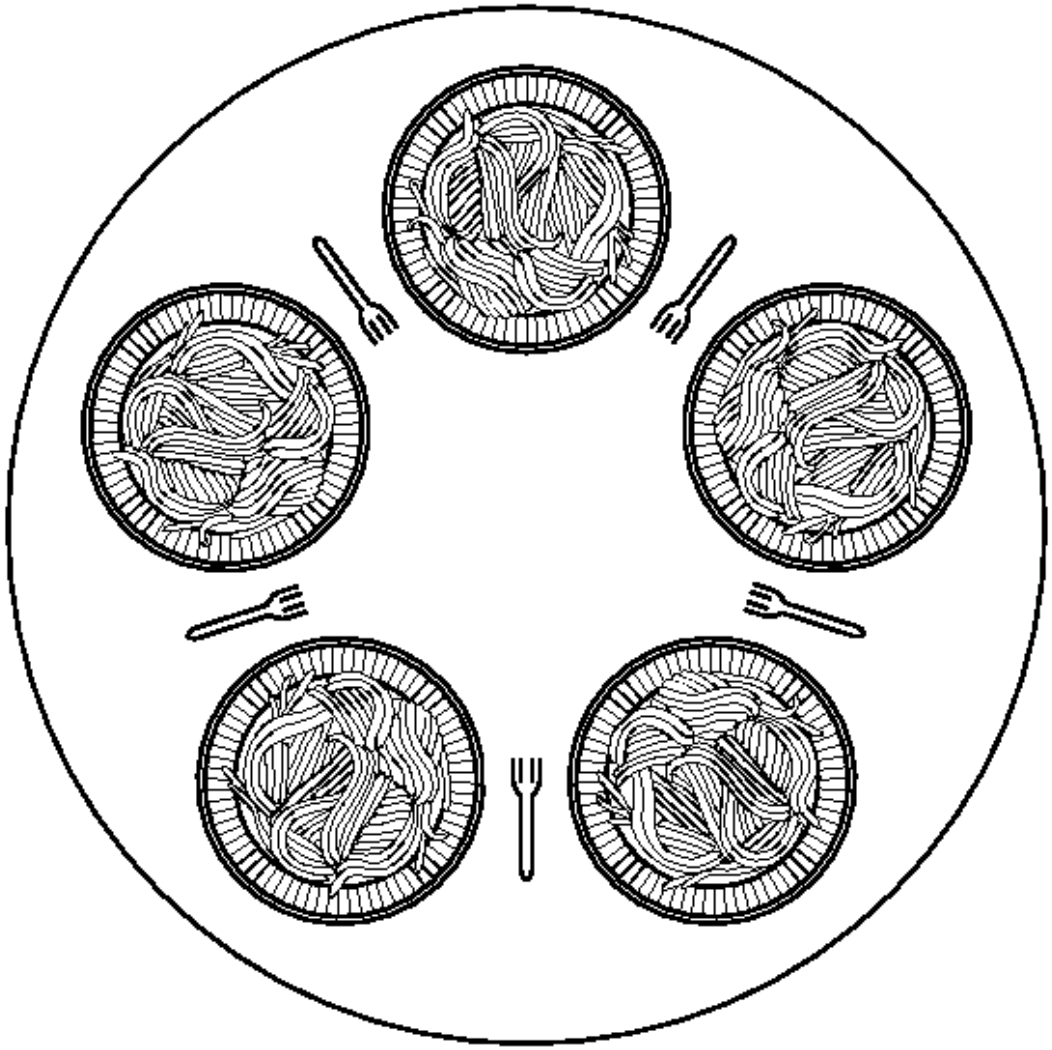
B tároló n db üzenet tárolására alkalmas
c0, c1, ..., cn-1 cellák, üresek vagy foglaltak

Műveletek:

- **send**(B, m): az m üzenetet a cj üres cellába helyezi, j = mink(k mod n; ck üres); blokkol, ha nincs üres cella; cj foglalt lesz.
- m = **receive**(B): a cj foglalt cellából az üzenetet kiadja, j = mink(k mod n; ck foglalt); blokkol, ha nincs foglalt cella; cj üres lesz.
- min: FIFO értelemben a "legkorábban feltöltött/kiürített" cella

PI. csővezeték (pipe) implementálható ciklikus üzenetbufferrel

Étkező filozófusok



Étkező filozófusok probléma egy hibás megoldása

```
#define N 5                     // number of philosophers

void philosopher(int i) { // i: philosopher number, from 0 to 4
  while (TRUE) {
    think();                    // philosopher is thinking
    take_fork(i);               // take left fork
    take_fork((i+1) % N); // take right fork; % is modulo operator
    eat();                      // yum-yum, spaghetti
    put_fork(i);                // put left fork back on the table
    put_fork((i+1) % N); // put right fork back on the table
  }
}
```

Holtpontot eredményezhet.

Étkező filozófusok probléma egyik megoldása

```
#define N          5          // number of philosophers
#define LEFT       (i+N-1)%N  // number of i's left neighbor
#define RIGHT      (i+1)%N    // number of i's right neighbor
#define THINKING 0            // philosopher is thinking
#define HUNGRY  1             // philosopher is trying to get forks
#define EATING   2            // philosopher is eating
typedef int semaphore;        // semaphores are a special kind of int
int state[N];                 // array to keep track of everyone's state
semaphore mutex = 1;          // mutual exclusion for critical regions
semaphore s[N];               // one semaphore per philosopher

void philosopher(int i) {     // i: philosopher number, from 0 to N-1
  while (TRUE) {              // repeat forever
    think();                  // philosopher is thinking
    take_forks(i);            // acquire two forks or block
    eat();                    // yum-yum, spaghetti
    put_forks(i);             // put both forks back on table
  }
}
```

Étkező filozófusok probléma egyik megoldása

```
void take_forks(int i) {        // i: philosopher number, from 0 to N−1
  down(&mutex);                 // enter critical region
  state[i] = HUNGRY;            // record fact that philosopher i is hungry
  test(i);                      // try to acquire 2 forks
  up(&mutex);                   // exit critical region
  down(&s[i]);                  // block if forks were not acquired
}

void put_forks(i) {             // i: philosopher number, from 0 to N−1
  down(&mutex);                 // enter critical region
  state[i] = THINKING;          // philosopher has finished eating
  test(LEFT);                   // see if left neighbor can now eat
  test(RIGHT);                  // see if right neighbor can now eat
  up(&mutex);                   // exit critical region
}

void test(i) {                  // i: philosopher number, from 0 to N−1
  if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
    state[i] = EATING;
    up(&s[i]);
  }
}
```

Gyártó-fogyasztó probléma

Korlátos tároló probléma (bounded buffer problem)

Gyártó-fogyasztó probléma (producer-consumer problem): Kép proc osztozik egy közös, rögzített méretű tárolón. A gyártó adatokat tesz bele, a másik, a fogyasztó, kiveszi azokat.

A naív megoldás versenyhelyzetet eredményezhet. Pl. tekintsük a következő eseménysorozatot.

1. A fogyasztó megállapítja, hogy a tároló üres, de mielőtt elmenne aludni elveszíti a CPU-t.
2. Gyártó gyárt egy elemet, majd jelzést küld a fogyasztónak, hogy ébredjen fel.
3. Mivel a fogyasztó nem alszik, az üzenet hatástalan.
4. A gyártó újabb elemeket gyárt, majd a tároló beteltével elmegy aludni.
5. A fogyasztó visszakapja a CPU-t, és elmegy aludni.

Gyártó-fogyasztó probléma hibás megoldása

```
#define N 100                       // number of slots in the buffer
int count = 0;                      // number of items in the buffer
void producer(void) {
  int item;
  while (TRUE){                     // repeat forever
    item = produce_item();          // generate next item
    if (count == N) sleep();        // if buffer is full, go to sleep
    insert_item(item);              // put item in buffer
    count = count + 1;              // increment count of items in buffer
    if (count == 1) wakeup(consumer); // was buffer empty?
  }
}
void consumer(void) {
  int item;
  while (TRUE) {                    // repeat forever
    if (count == 0) sleep();        // if buffer is empty, got to sleep
    item = remove_item();           // take item out of buffer
    count = count - 1;              // decrement count of items in buffer
    if (count == N - 1) wakeup(producer); // was buffer full?
    consume_item(item);             // print item
  }
}
```

Gyártó-fogyasztó probléma szemaforok használatával

```
#define N 100                    // number of slots in the buffer/pipe
typedef int semaphore;           // semaphores are a special kind of int
semaphore mutex = 1;             // controls access to critical region
semaphore empty = N;             // counts empty buffer slots
semaphore full = 0;              // counts full buffer slots

void producer(int item) {
  down(&empty);                  // decrement empty count
  down(&mutex);                  // enter critical region
  enter_item(item);              // put new item in buffer
  up(&mutex);                    // leave critical region
  up(&full);                     // increment count of full slots
}
void consumer(int &item) {
  down(&full);                   // decrement full count
  down(&mutex);                  // enter critical region
  remove_item(item);             // take item from buffer
  up(&mutex);                    // leave critical region
  up(&empty);                    // increment count of empty slots
}
```

Az olvasók és írók probléma

Olvasók és írók probléma (readers and writers problem): Több proc egymással versengve írja és olvassa ugyanazt az adatot. Megengedett az egyidejű olvasás, de ha egy proc írni akar, akkor más procok se nem írhatnak se nem olvashatnak.
Példa: adatbázisok, fájlok, hálózat, ... stb.

Ha folyamatos az olvasók utánpótlása, az írók éheznek.
Megoldás: érkezési sorrend betartása.
Következmény: hatékonyság csökken.

Az olvasók és írók probléma szemaforok használatával

```
typedef int semaphore;    // use your imagination
semaphore mutex = 1;      // controls access to 'rc'
semaphore db = 1;         // controls access to the data base
int rc = 0;               // # of processes reading or wanting to

void reader(record &rec) {
  down(&mutex);           // get exclusive access to 'rc'
  rc = rc + 1;            // one reader more now
  if (rc == 1) down(&db); // if this is the first reader ...
  up(&mutex);             // release exclusive access to 'rc'
  read_data_base(rec);    // access the data
  down(&mutex);           // get exclusive access to 'rc'
  rc = rc - 1;            // one reader fewer now
  if (rc == 0) up(&db);   // if this is the last reader ...
  up(&mutex);             // release exclusive access to 'rc'
}
void writer(record rec) {
  down(&db);              // get exclusive access
  write_data_base(rec);   // update the data
  up(&db);                // release exclusive access
}
```

Ütemezés

Ütemezés feladata (process scheduling)

Egy adott időpontban létező, futáskész állapotú procok közül egy kiválasztása, amely a következő időintervallumban a CPU-t birtokolja.

Mikor kell ütemezni?

- Amikor egy processzus befejeződik.
- Amikor egy processzus blokkolódik (Pl. I/O művelet miatt)

Mikor lehet ütemezni?

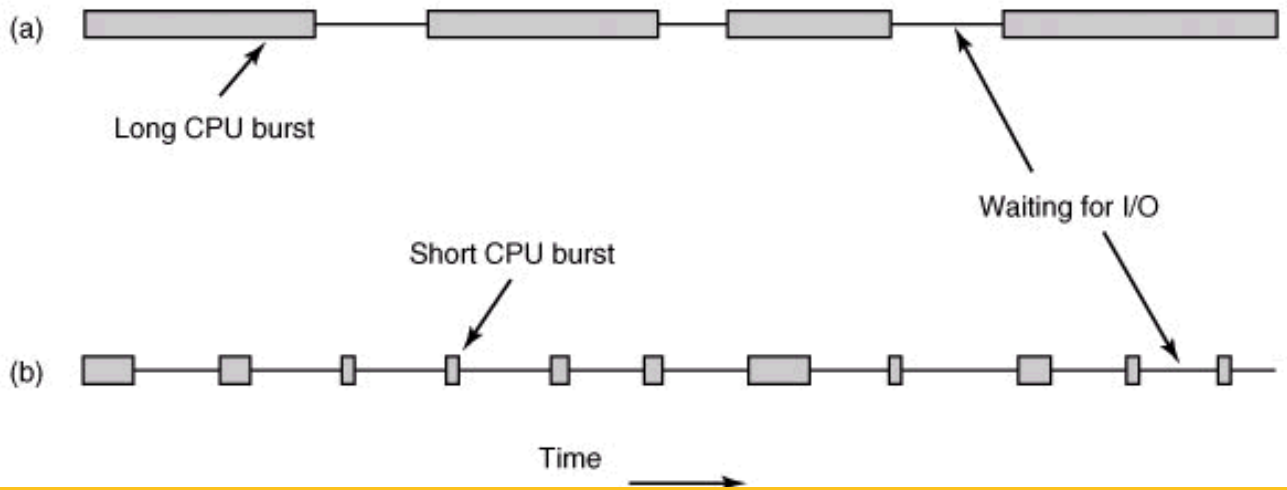
- Amikor új processzus jön létre.
- Amikor I/O megszakítás következik be.
- Amikor időzítőmegszakítás következik be.

Ütemezés céljai

- Pártatlanság (fair): indokolatlanul a többi proc kárára ne részesüljön proc előnyben
- CPU kihasználtság: jó legyen
- Erőforráskihasználtság: (memória, I/O eszközök, ...) jó legyen
- Átfutási idő: (egy proc létrejöttétől megszűntéig az igényelt CPU idő arányában) minél rövidebb legyen
- Áteresztőképesség: egységnyi idő alatt minél több proc teljesüljön
- Válaszidő: (interaktív procok) jó legyen
- Megbízhatóság: a proc átfutási ideje ne nagyon függjön az időtől (a többi, egyidejűleg létező proctól)
- Előrejelezhetőség: pontosan előre jelezhető és szabályos ütemezés. Főleg multimédia rendszerekben fontos
- Rezsi (overhead) alacsony legyen: kiválasztó algoritmus nem lehet túl bonyolult

Proc ütemezés paraméterei

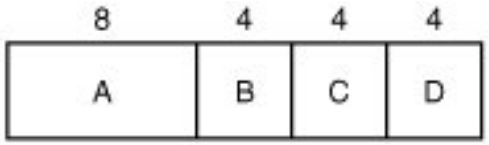
- Proc osztályok: Batch, Interaktív, Real-time
- Proc prioritás: Rögzített (létrejöttétől), Változó (ütemező változtatja)
- Proc típus: CPU-intenzív (számolós), Erőforrás-igényes (pl. sok I/O)
- Real-time feltételek: Egy eseményt kiszolgáló proc feladatát korlátozott időintervallum alatt teljesítheti
- Dinamikus információk (ütemező gyűjti) Pl: Eddig használt össz-CPU idő, max. memória, erőforrás-kérések gyakorisága vagy Valamely szempont szerint a proc "elhanyagoltsága", pl. interaktív proc rég nem kapott CPU-t.



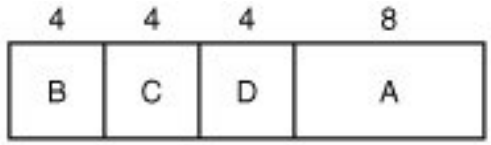
Ütemezés kötegelt rendszerekben

Sorrendi ütemezés (First-Come First-Served): olyan sorrendben kapják meg a procok a CPU-t amilyen sorrendben kérik. Nem megszakítható.

Legrövidebb feladat először (Shortest Job First): tfh. a futási idő előre ismert.



(a)

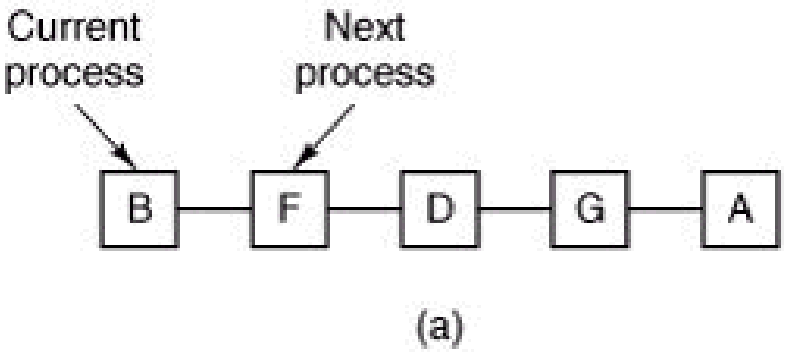


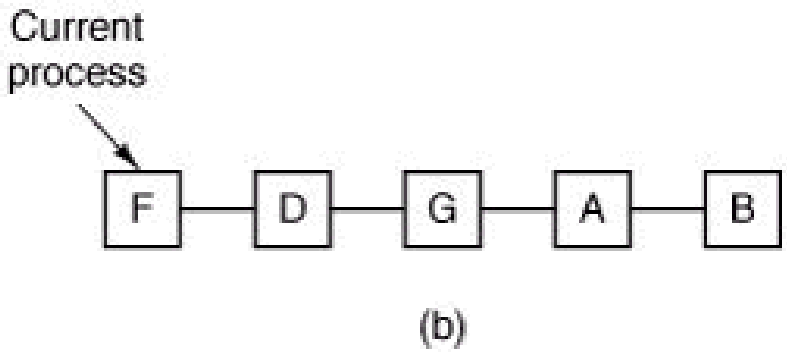
(b)

Legrövidebb maradék futási idejű először (Shortest Remaining Time Next)

Ütemező algoritmusok

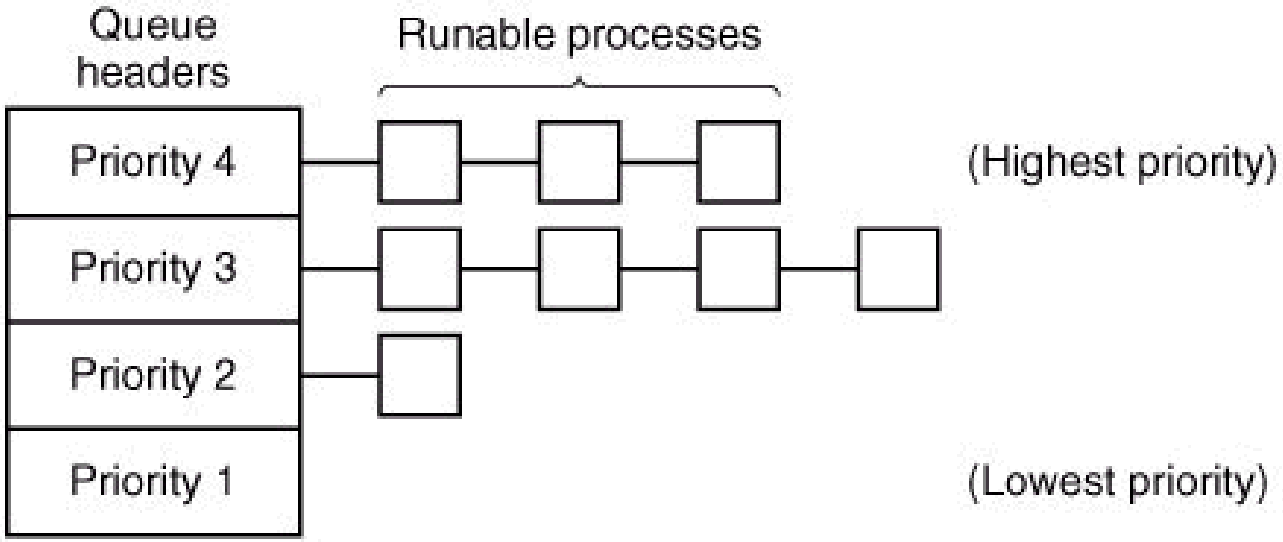
Round Robin: Procok sorba (FIFO) rendezettek; mindig a sorban első, futáskész kapja a CPU-t, ha elveszti, a sor végére kerül.
Osztályonként lehet egy-egy sor





Ütemező algoritmusok

Tiszta prioritásos: Mindig a legmagasabb prioritású, futáskész proc kapja a CPU-t.
Prioritási osztályokba sorolhatók a procok; a prioritásos algoritmus osztályon belül érvényesül, osztályok között más szempont dönt.



Ütemező algoritmusok

Aging (növekvő prioritásos): Prioritásos algoritmus + ha egy futáskész proc nem választódik ki, nő a prioritása

Feedback (visszacsatolásos): Prioritásos algoritmus + ha a CPU időszelet lejárta miatt veszti el a proc a CPU-t, csökken a prioritása, ha erőforrás-kérés miatt, marad a prioritása. A számolós és I/O igényes procok kiegyenlítődnek

Sorsjáték (Lottery): A processzusok sorsjegyet kapnak, ütemezés = sorshúzás. Prioritás szimulálható több sorsjeggyel.

Garantált: n proc esetén mindegyiknek jár a $\frac{letezesiido}{n}$ CPU idő.

Ütemező algoritmusok

Fair share (arányos): A procok osztályokba sorolódnak; minden osztály meghatározott arányban részesül az össz-CPU időből. Pl. 4 user mindegyike 25% CPU-t kap függetlenül az általuk futtatott procok számától.
Az ütemező méri az egyes osztályokbeli procokra fordított össz-CPU időt; ha egy osztály elmarad arányától, abból az osztályból választ (ha tud).
Osztályon belüli választás más szempont szerint

Real-time feltételek: Ha ismert minden e esemény maximális megengedett kiszolgálási ideje (t_{emax}), az e bekövetkezésének ideje (t_{e0}), az e kiszolgáló procának időigénye (t_e), akkor azt a procot választja ki, amelyiknek a legkevesebb ideje maradt a hozzá tartozó esemény kiszolgálására $min_e(t_{e0} + t_{emax} - t_e)$

Proc ütemezés elmélete

Szelekciós függvény: `int f(p, w, e, s);` ahol
p: proc prioritása
w: proc eddig a rendszerben eltöltött összideje
e: a proc által eddig elhasznált össz CPU idő
s: a proc által igényelt össz CPU idő

f értékét minden aktív procra kiszámolom, az érték szerint döntök
f $\sim$ max(w) $\sim$ FIFO
f $\sim$ const $\sim$ RoundRobin
f $\sim$ min(s) $\sim$ Shortest Process First

Petri séma

Procok szinkronizációjának grafikus ábrázolására használható, matematikai elmélete is van.

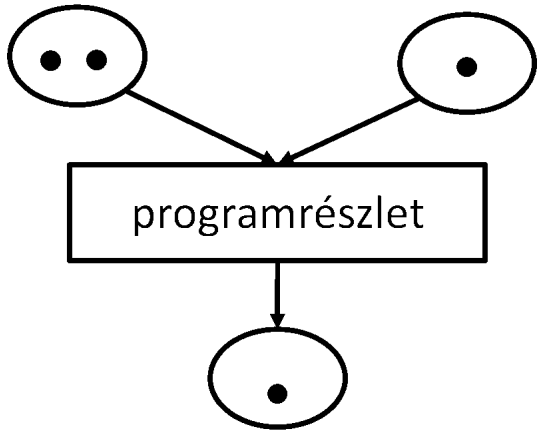
Akció: egy szekvenciálisan végrehajtandó programrészlet (pl. függvény).
Be- és kimenetek: az akciók bemenetei ill. kimenetei (karika)
Feltételek: az akciók végrehajtásának feltételei (nyilak)

Petri-féle szinkronizációs séma

Akciók bemeneteiken és kimeneteiken összekötött hálózata (gráfja)

Az *a* **akció végrehajtható**: ha minden bemenetén legalább 1 pont van.

Az *a* **akció végrehajtása**: minden bemenetéről 1 pont levonódik, minden kimenetéhez 1 pont hozzáadódik.

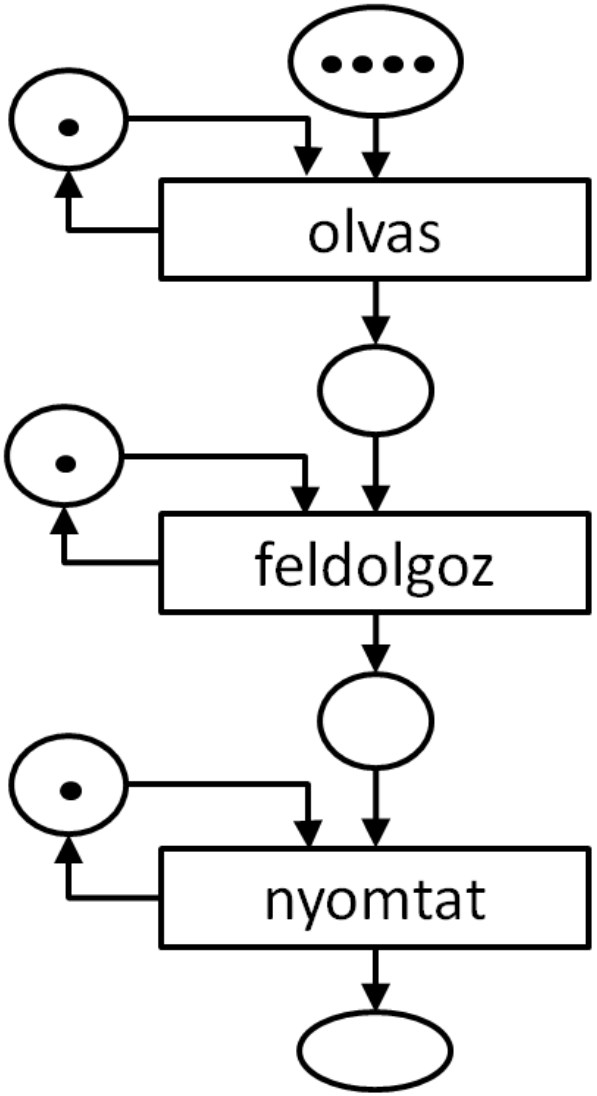


Petri séma

Kezdőállapot: a sémában a bemeneteken és kimeneteken kezdő-pontok elhelyezve.

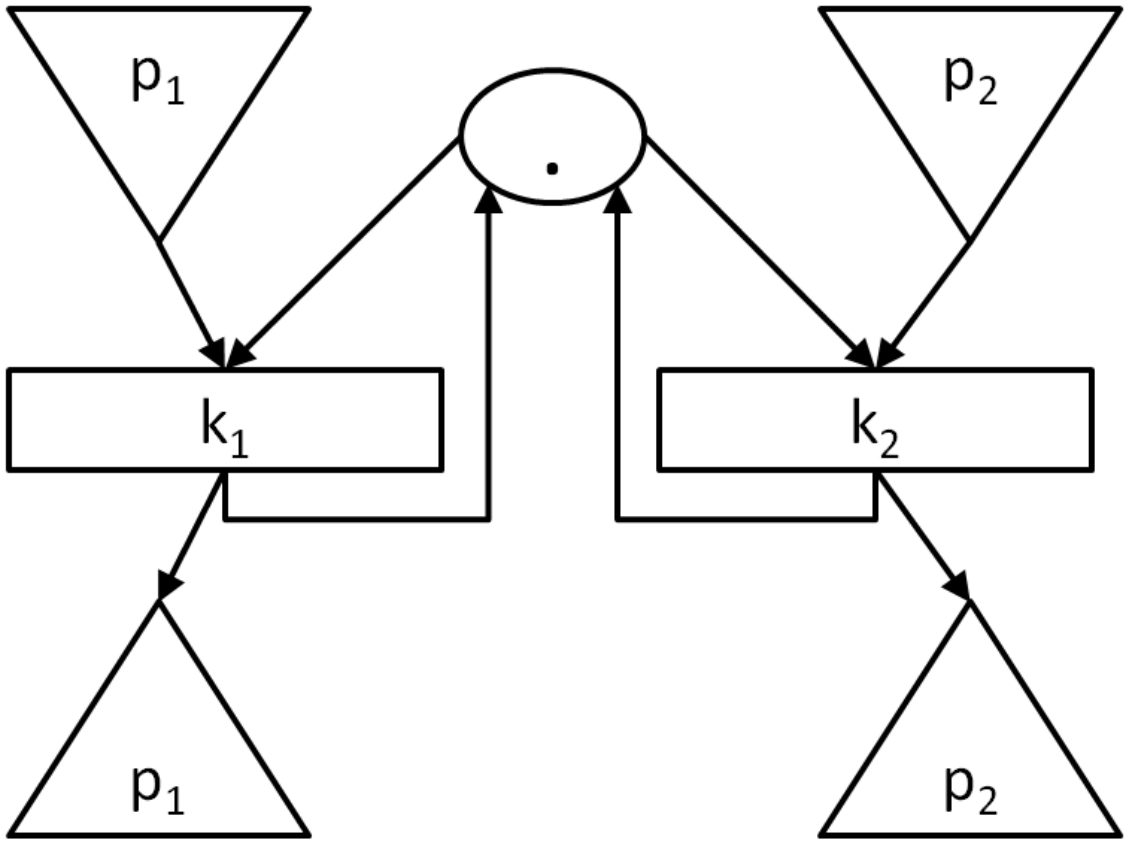
Végállapot: a sémában nincs végrehajtható akció.

Végrehajtás: az akciók lehetséges végrehajtásaival kapott séma- (állapot-) sorozat kezdőből végállapotig.



Petri séma

Példa: Kritikus szekciók



Erőforrások

Erőforrás (osztottan használt erőforrás): amit ugyanabban az időben csak egy processzus használhat. Egy proc kéri az erőforrást, használja, majd felszabadítja.

```
...
Request (...);  // kérés
...             // használat
Release (...);  // felszabadítás
...
```

Lehetnek:

1. **Hardver**/**Sw** erőforrások.
2. **Megszakítható** erőforrás: elvehető az azt birtokló proctól hiba bekövetkezte nélkül (pl. memória). **Megszakíthatatlan** erőforrás: elvétele a tulajdonosától hibát eredményez (pl. CD-író)

Erőforrás osztályok

Erőforrás-osztály (eo)

Hasonló tulajdonságú, hasonlóan kezelendő erőforrások együttese

Készlet: eo elemei, az elemek leírásai. **Várakozó sor**: blokkolt állapotú procok sora; erőforrást kértek, de még nem kapták meg. **Allokátor**: eo-beli erőforrás kérést/felszabadítást teljesítő OS eljárások

Erőforrás osztályok leírása

- Azonosító: res id
- Létrehozó proc: proc id
- Készlet listája (lánc): 1) Jellemzők leírása, pl. mem kezdőcím, hossz; 2) Szabad/foglalt, mely proc foglalja; 3) Új készlet-elem beléptető/törlő eljárás
- Várakozó lista: Proc id, Erőforrás-kérés paraméterei (pl. kért mem hossza), Belépési időpont, ...
- Allokátor eljárások: Request, Release eljárás-címek

Erőforrás kérés/felszabadítás

`Request(res_id, e-specifikáció,&e-id);`

1. A kérő proc blokkolt állapotba hozása
2. A kérő proc várakozó listába helyezése
3. A várakozó lista végignézése; ha egy proc (korábbi) kérése teljesíthető, akkor: 1) e-id kitöltése 2) e-id készletbeli erőforrás foglaltságának bejegyzése 3) proc futáskész állapotba hozása 4) proc törlése a várakozó sorból
4. Vezérlés átadása a proc ütemezőnek

`Release(res_id, e-id);`

1. e-id készletbeli erőforrás szabadságának bejegyzése
2. ugyanaz mint Request 3.
3. visszatérés (return) a hívó proc-hoz

Holtpont

Holtpont (Deadlock)

Procok egy halmaza egymás birtokában lévő erőforrásokra várakozik – örökre.

Példák:

- v1, v2 globális változók. p1 ír v1-be, közben olvasni akar v2-ből, p2 ír v2-be, közben olvasni akar v1-ből. Holtpont kialakulhat akkor is ha alkalmazzuk a kölcsönös kizárás módszereit v1-re és v2-re külön-külön. Megoldás: v1-et és v2-t összefogjuk egy rekordba, és azt védjük.
- Két proc: mindkettő szkennel és CD-re ír. Az egyik a szkennert, a másik a CD-írót foglalja le hamarabb, majd megpróbálja a másikat is.
- Egy adatbázisrendszerben: ha A proc zárolja R1 rekordot, B proc R2-t, majd mindkettő megpróbálja zárolni a másikét.
- 4 autó egyenrangú utak kereszteződésében

Holtpont kialakulásának feltételei

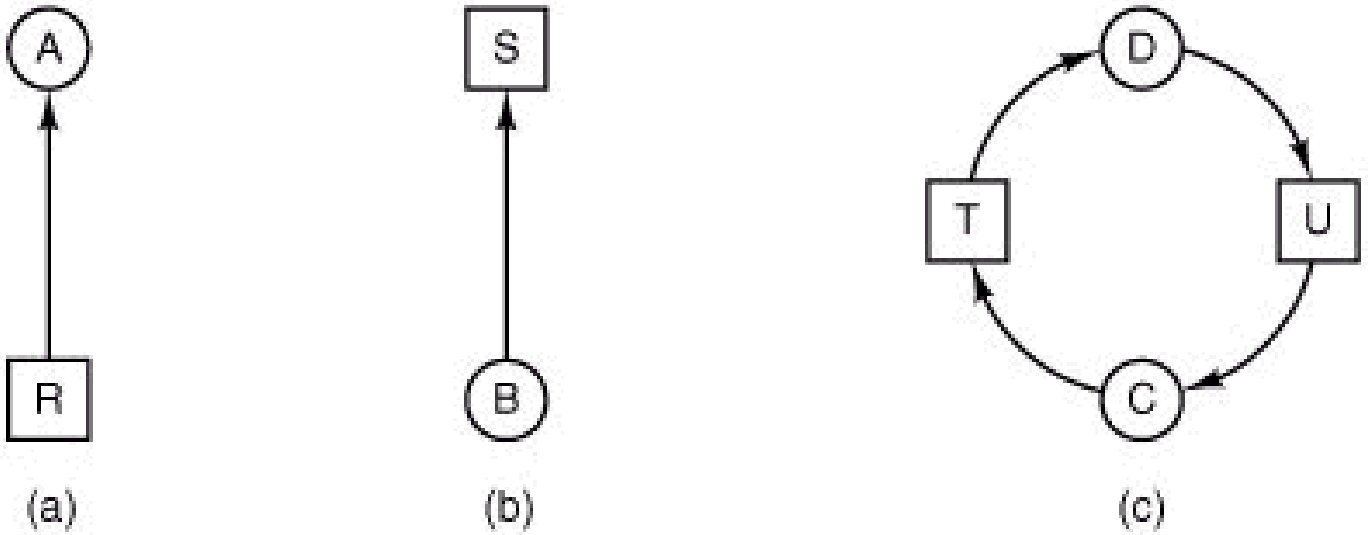
Az alábbi feltételek mindegyikének teljesülnie kell egy holtpontban.

1. **Kölcsönös kizárás** (Mutual Exclusion) feltétel: Egy erőforrást egy pillanatban csak max egy proc használhat.
2. **Birtoklás és várakozás** (Hold and Wait) feltétel: erőforrást birtokló proc kérhet újabb erőforrást.
3. **Megszakíthatatlanság** (No preemption) feltétel: Csak a birtokló proc szabadíthatja fel a birtokában lévő erőforrásokat.
4. **Ciklikus várakozás** (Circular wait) feltétel: Több procból álló lánc, amelynek mindegyik proca a láncban következő proc által birtokolt erőforrásra vár.

Az erőforrás-lefoglalás gráf

Resource allocation graph

- A gráf csúcsai: processzusok (kör) és erőforrások (négyzet)
- Erőforrásból processzusba mutató él: a proc birtokolja az erőforrást.
- Processzusból erőforrásba mutató él: a proc várakozik az erőforrásra, állapota blokkolt.
- Egy gráfbeli kör azt jelenti holtpont van.



Példa holtpont előfordulására

A
Request R
Request S
Release R
Release S

(a)

B
Request S
Request T
Release S
Release T

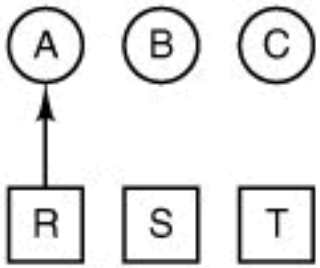
(b)

C
Request T
Request R
Release T
Release R

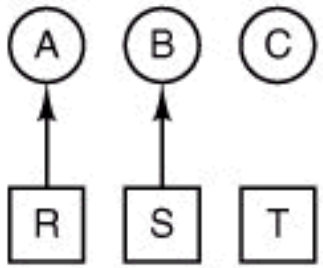
(c)

1. A requests R
2. B requests S
3. C requests T
4. A requests S
5. B requests T
6. C requests R
   deadlock

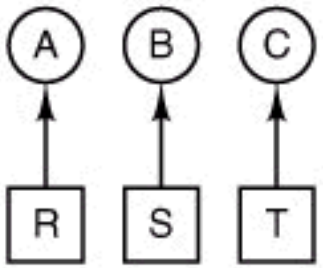
(d)



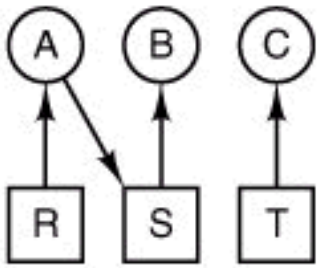
(e)



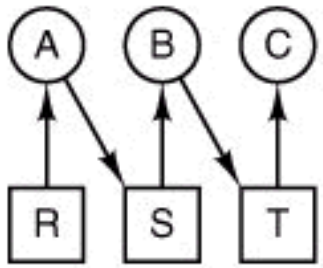
(f)



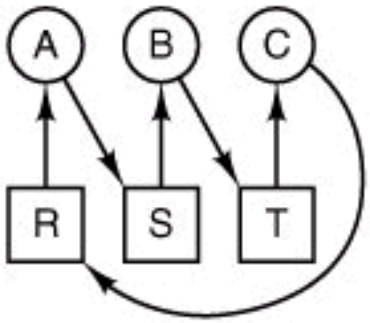
(g)



(h)



(i)



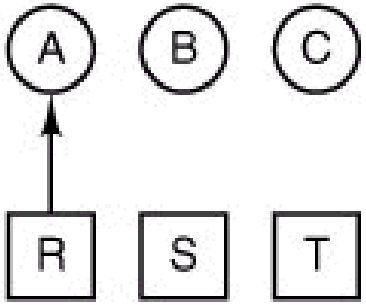
(j)

Példa holtpont elkerülésére

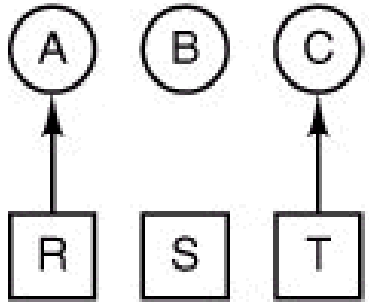
1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S

no deadlock

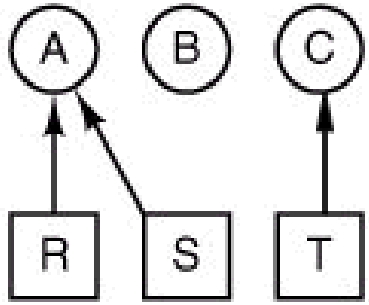
(k)



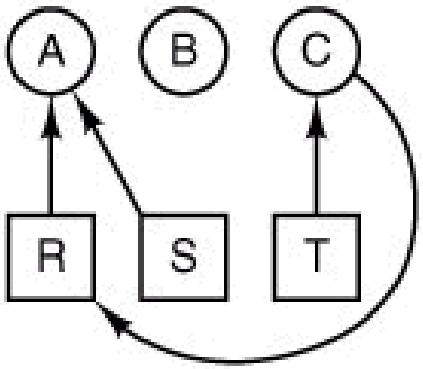
(l)



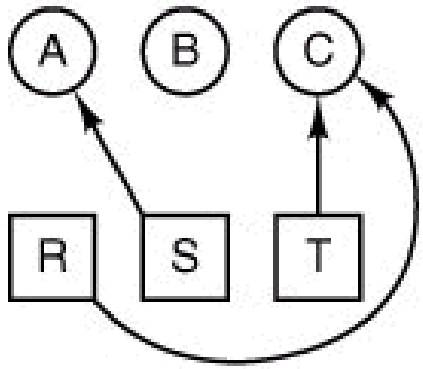
(m)



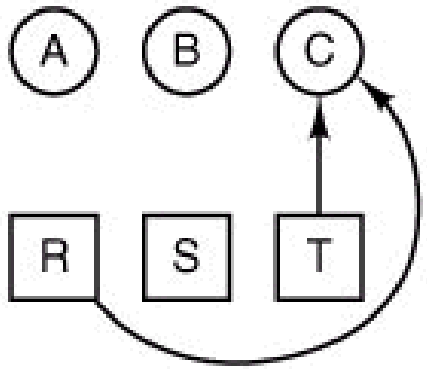
(n)



(o)



(p)



(q)

Holtpont kezelése

Lehetséges stratégiák:

- Feltételezzük, hogy nem alakul ki (strucc politika - Ostrich algoritmus). A rendszergazda feladata a "gyanúsan öreg" procok megszüntetése (Unix)
- Felismerés és helyreállítás (detection and recovery).
- Elkerülés (avoidance).
- Megelőzés (prevention). Megakadályozzuk a létrejöttét azzal, hogy korlátozzuk a procok erőforráshoz jutását – erőforrás-kihasználtság romlik

Felismerés és helyreállítás

Felismerés

- Típusonként egy erőforrás: kör detektálása az erőforrás-lefoglalási gráfban
- Típusonként több erőforrás

Helyreállítás

- Az erőforrás ideiglenes elvételével.
- Rollbackel. A proc állapotáról mentések készülnek, bmelyikhez vissza lehet térni.
- Valamely proc megszüntetésével.

A holtpont elkerülése

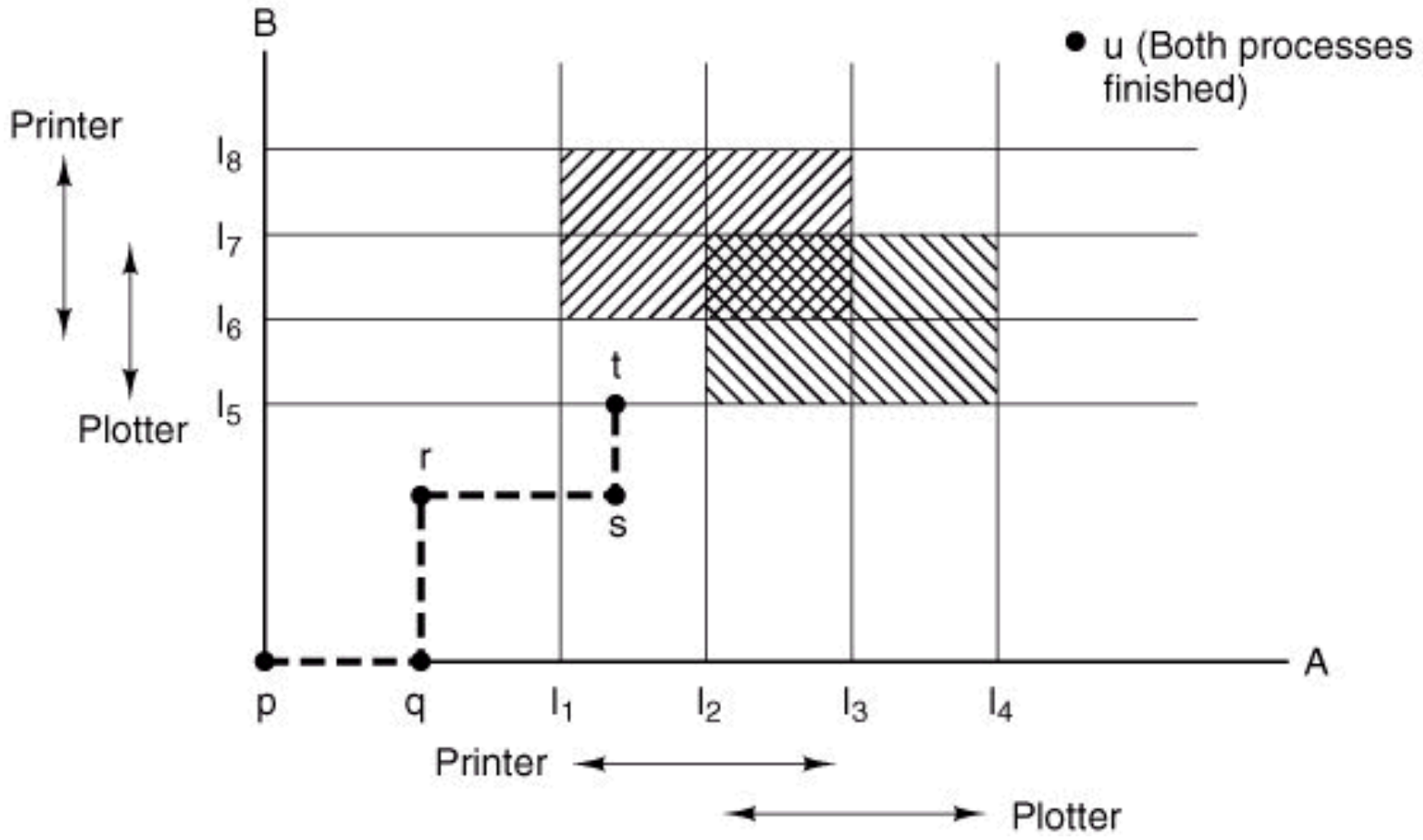
A procok az erőforrásokat nem egyszerre, hanem külön-külön kérik. Minden kérés esetén dönteni kell, hogy az erőforrás odaítélhető-e vagy sem. A holtpont elkerülhető, ha mindíg a helyes döntést hozzuk.

Módszerek:

- Erőforrás-pályagörbék
- Bankár algoritmus

Erőforrás-pályagörbék

Kétprocesszusú erőforrás-pályagörbék



A bankár algoritmus

A bankár algoritmus egyetlen erőforrásra

	Has	Max
A	0	6
B	0	5
C	0	4
D	0	7

Free: 10

(a)

	Has	Max
A	1	6
B	1	5
C	2	4
D	4	7

Free: 2

(b)

	Has	Max
A	1	6
B	2	5
C	2	4
D	4	7

Free: 1

(c)

A bankár algoritmus

A bankár algoritmus többpéldányos erőforrástípusok esetén

Process	Tape drives	Plotters	Printers	CD-ROMS
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Resources assigned

Process	Tape drives	Plotters	Printers	CD-ROMS
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Resources still needed

E = (6342)
P = (5322)
A = (1020)

Holtpont megelőzése

A kölcsönös kizárás megakadályozása

- Próbáljuk minimalizálni az erőforráshoz hozzáférő procok számát (pl. printer daemon)
- Csak akkor ítéljük oda az erőforrást, amikor szükséges (pl. spool-fájl lezárásakor es nem a megnyitásakor)

Hold and Wait megakadályozása

- Szabály: proc nem kérhet erőforrást, ha birtokol erőforrást.
- Enyhítés: egyszerre kérhet több erőforrást.
- Következmény: éheztetés, rossz erőforrás kihasználtság.
- Módosítás: megköveteljük, hogy kérés előtt engedje el a birtokolt erőforrásokat, és egyszerre kérje vissza őket

Holtpont megelőzése

Megszakíthatatlanság megakadályozása:

- Erőforrások erőszakos elvétele
- Virtualizáció
- Hátrány: nem mindig lehetséges

Cirkularitás megakadályozása: erőforrások sorba-rendezése

- $< e_0, e_1, ..., e_{N-1} >$ az eo készlet.
- Szabály: ha p_i az e_j erőforrást kéri, nem birtokolhat e_k-t, ha $k > j$.
- Szabály: ha p_i az e_j erőforrásról lemond, nem birtokolhat e_k-t, ha $k > j$

További problémakörök

Két-fázisú zárolás (Two-Phase Locking): 1. fázis: a rekordok zárolása egyenként. Ha valamelyik sikertelen, felszabadítja a korábbiakat, és újra kezdi. Ha minden zárolás sikeres, akkor a 2. fázisban elvégzi a műveletet és felszabadít.
Hátrány: nem mindig alkalmazható: pl. hálózatok esetében nem működik

Kommunikációs holtpont: A és B procok hálózaton kommunikálnak kérés-válasz formájában. Ha egy üzenet elvész, akkor holtpont alakul ki.
Megoldás: timeout + protokollok.

Éheztetés (Starvation): Ha egy proc soha nem kapja meg a kért erőforrását, mert a kiosztáskor mások részesülnek előnyben.
Megoldás: FIFO sor használata

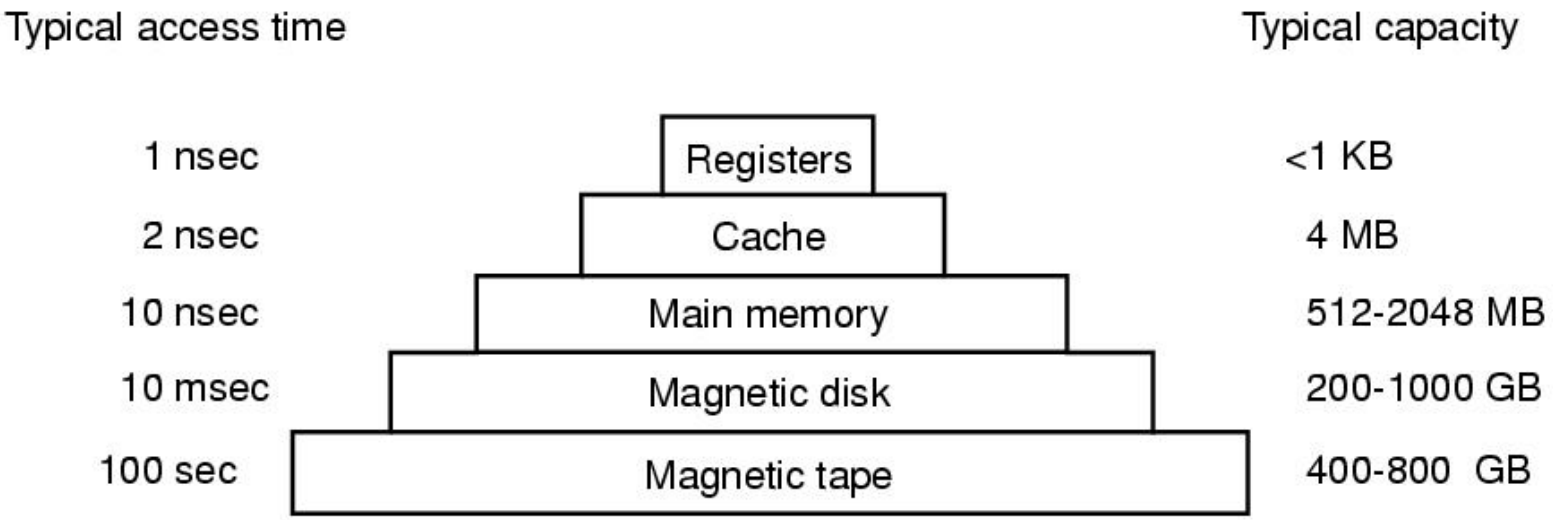
Memóriakezelés

Proc memóriája: **Szöveg** (változatlan): program + konstansok; **Adat** (változó): statikus + verem + heap

Feladatok:

- Szabad/foglalt területek nyilvántartása
- Memória-igények kielégítése (dinamikus memória)
- Csere (swap): memória – háttértár közötti csere
- Memória-védelem

Többszintű memória



Memóriakezelés

Feladatok proc létrehozásakor:

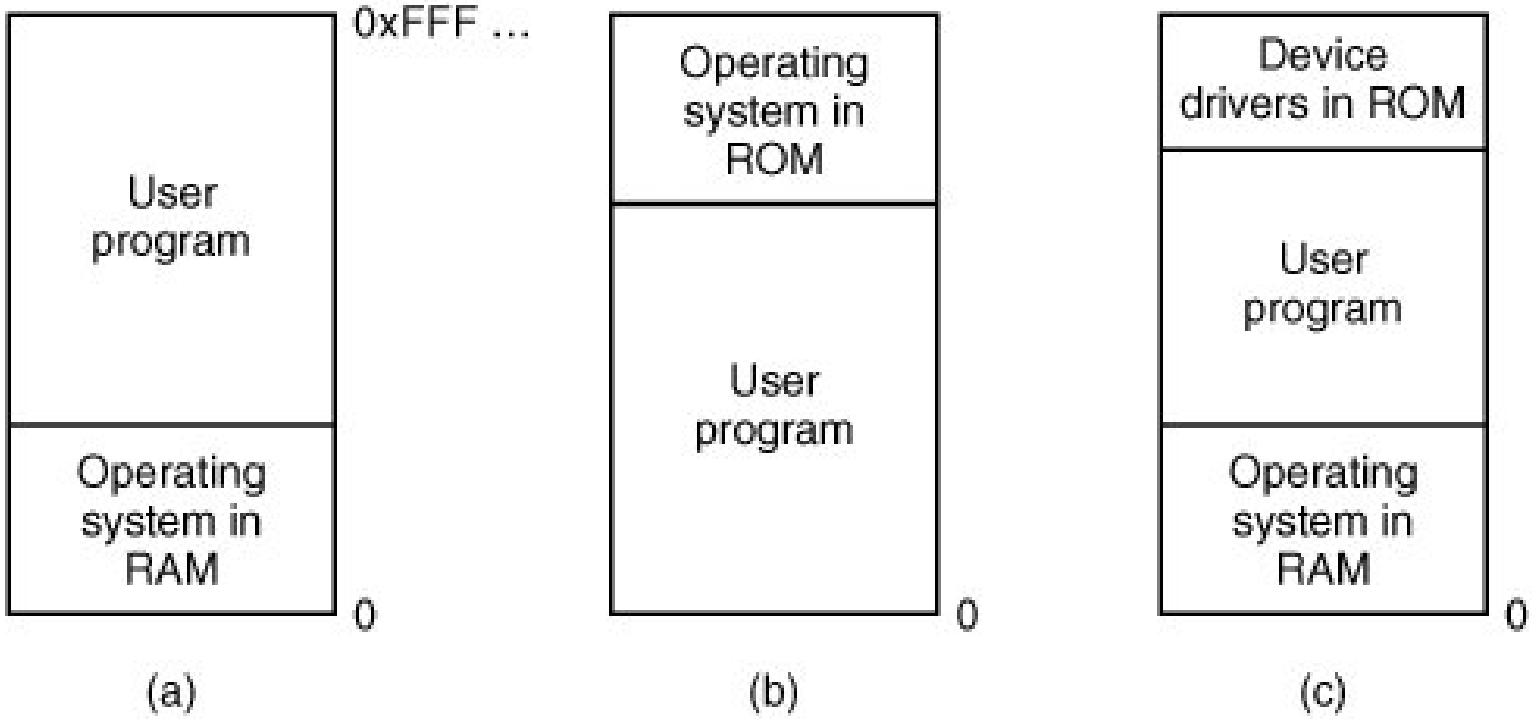
- Induló/max memória biztosítása
- Statikus/dinamikus partíciók
- Reallokáció: betöltési címnek megfelelően átcímzés
- Bázis-relatív címzés

Feladatok menet közben:

- Logikai $\rightarrow$ fizikai cím transzformáció
- Dinamikus memóriakérések/felszabadítások
- Csere
- Védelem (bázis határcím)

Monoprogramozás csere és lapozás nélkül

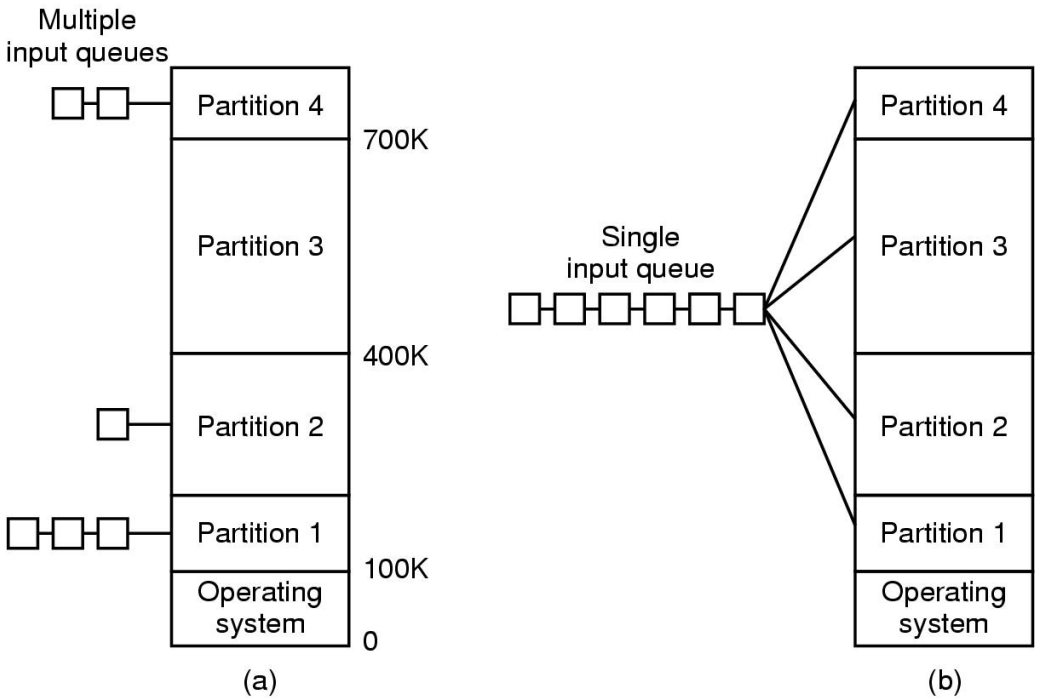
Egy időben csak egy program fut. A memória megoszlik az OS és a program között.



(a) nagyszámítógépeken volt jellemző, ma már ritka
(b) kézi számítógépek, beágyazott rendszerek
(c) korai személyi számítógépek (pl. MS-DOS)

Multiprogramozás rögzített méretű partíciókkal

Az új proc abba a várakozási sorba kerül, amelyik a legkisebb azok közöl, amelyikbe belefér.



Hátrány: sok kicsi, de kevés nagy proc esetén kihasználatlan memória.
Javítás: egy várakozási sor.

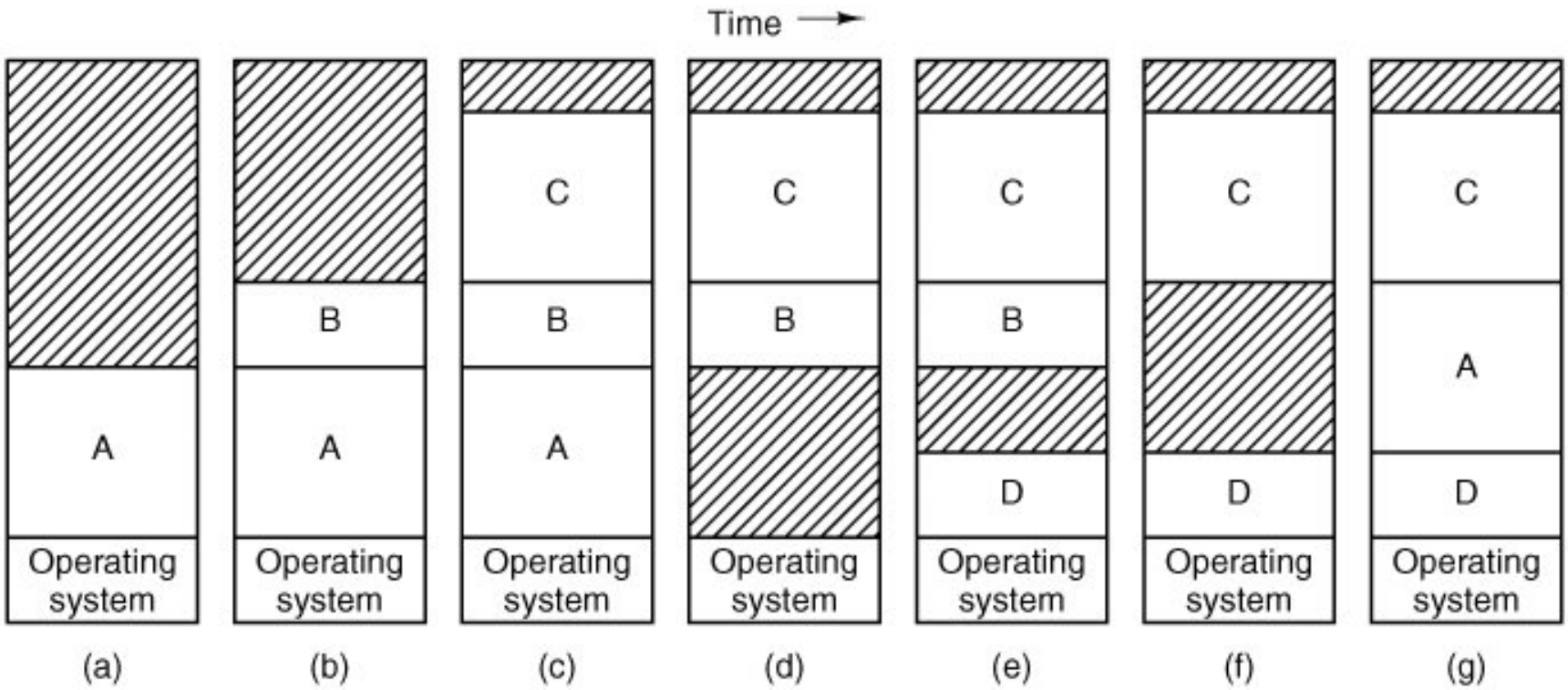
Memóriakezelés

Gyakran nincs elegendő hely a memóriában az összes proc számára, ezért némelyiket a lemezen kell tartani.

- **Csere** (swapping): a procokat teljes egészében mozgatja a memória és a lemez között.
- **Virtuális memória**: procok akkor is futhatnak, ha csak részeik vannak a memóriában.

Csere

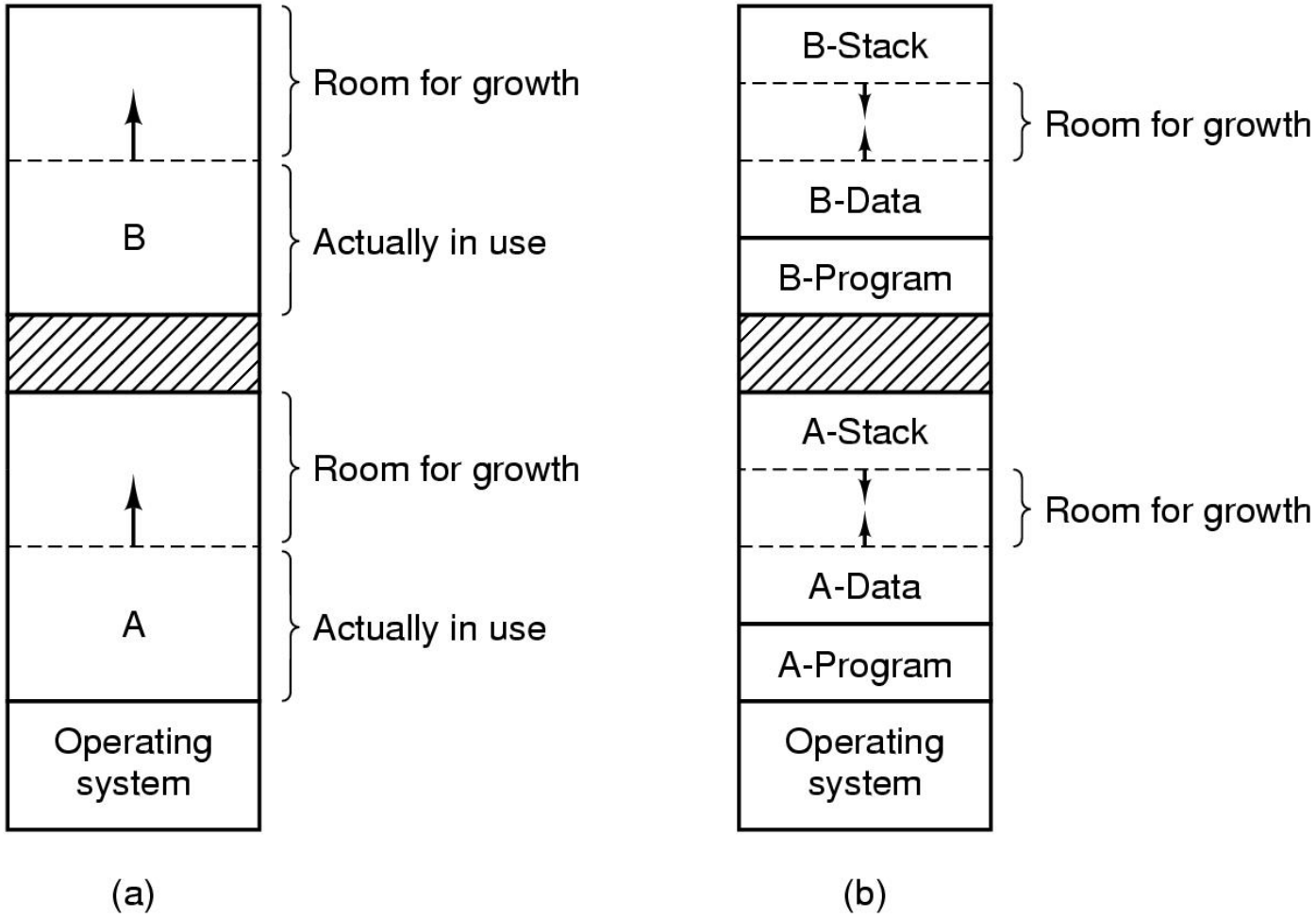
Változó partíciójú rendszerekben a partíciók száma, helye és mérete dinamikusan változik.



Memóriatömörítés (memory compaction): elaprózódott lyukak összeolvasztása. CPU-igényes művelet.

Csere

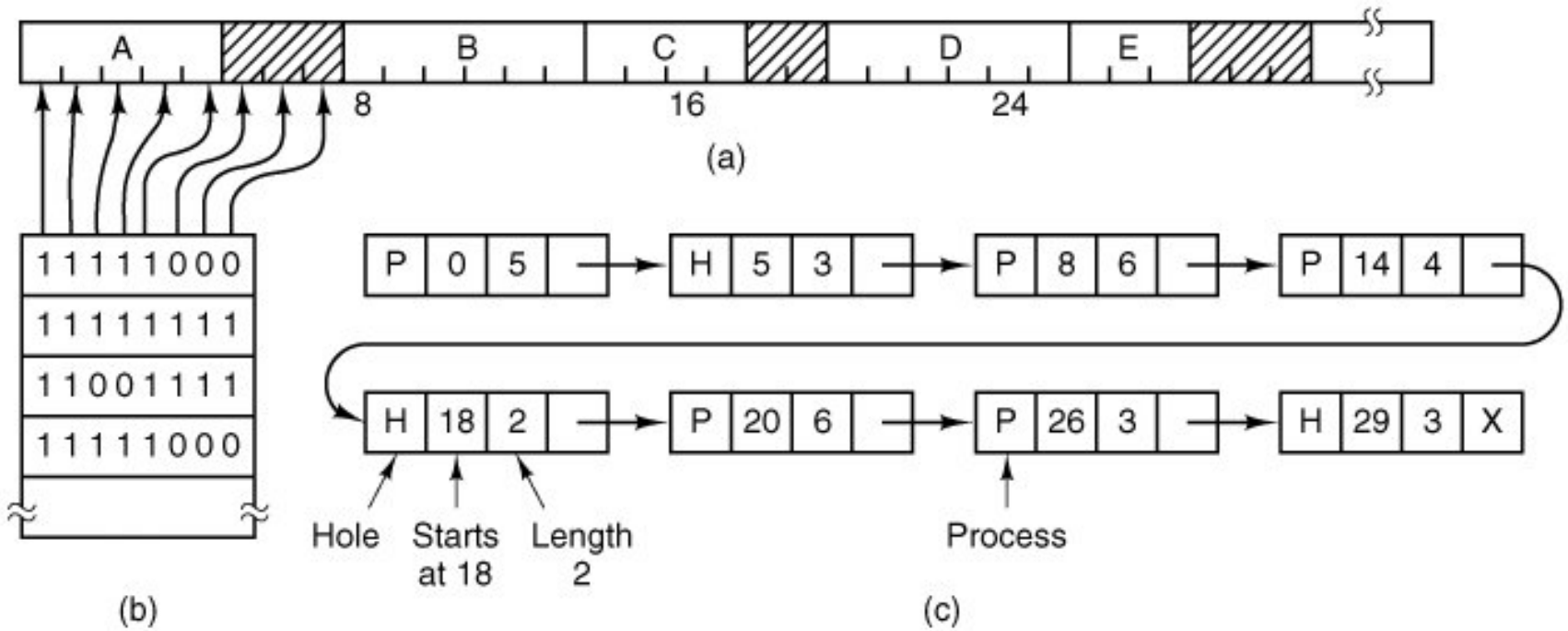
A processzusok mérete futás közben változhat:



Memóriakezelés bittérképpel

A memóriát **allokációs egységekre** osztjuk. Mindegyikhez tartozik egy bit a bittérképen: 0 = szabad, 1 = foglalt.

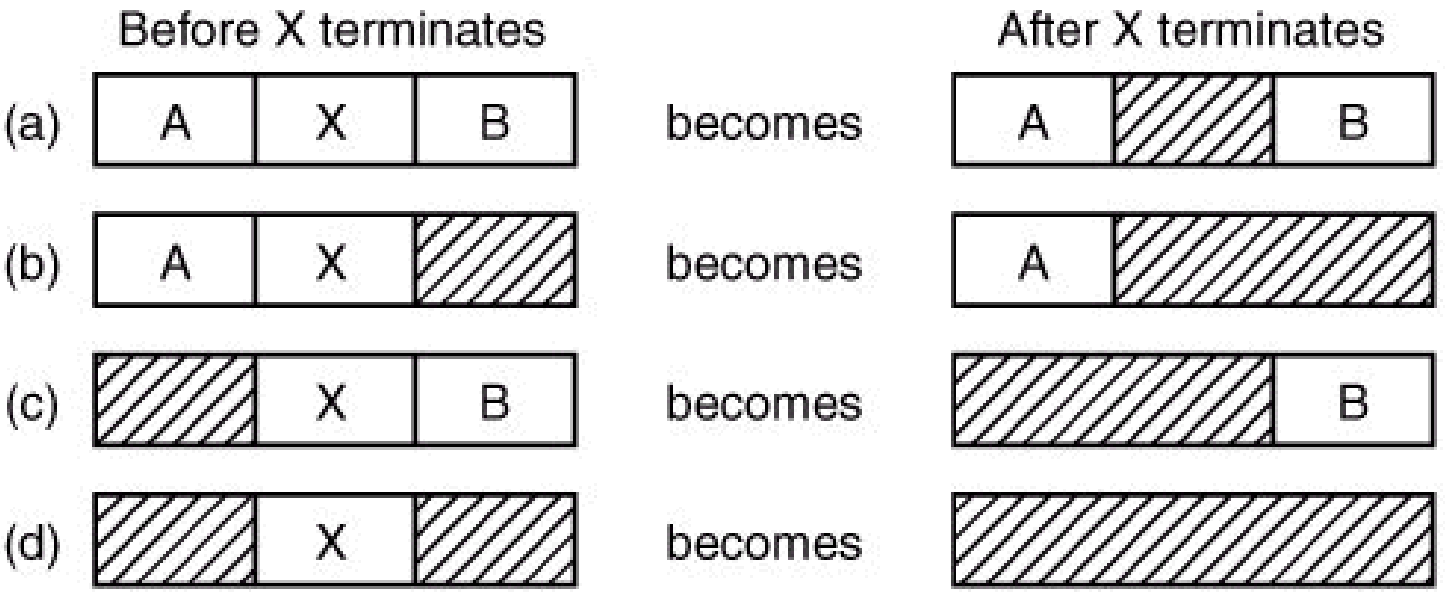
Fontos az allokációs egység mérete. Ha kicsi, akkor a bittérkép nagy. Ha nagy, akkor sok memória veszhet el a procok utolsó egységéből.



Memóriakezelés láncolt listákkal

A szabad és a foglalt elemeket láncolt listában tároljuk.

Processzus megszűnése esetén a listát is karban kell tartani:



Memóriakezelés láncolt listákkal

Algoritmusok a memória lefoglalására új procok számára:

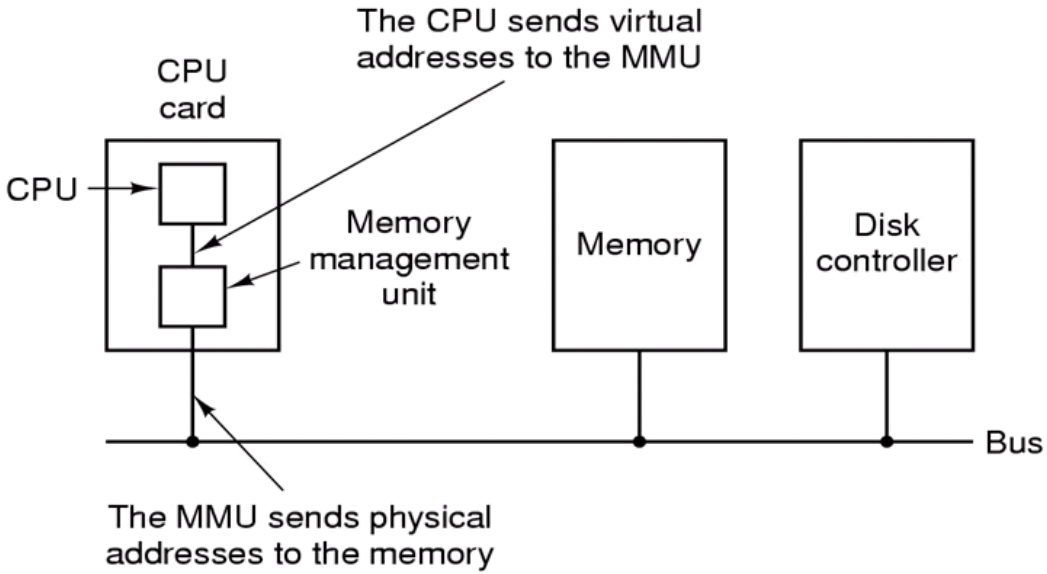
- **First fit**: első megfelelő méretű választása. Gyors, de apróz.
- **Next fit**: az utolsó találattól indítja a keresést. Rosszabb mint a first fit.
- **Best fit**: a legkisebb alkalmas lyukat keresi meg a teljes listában. Lassú, apróz.
- **Worst fit**: a legnagyobb lyukat választja. Nem olyan jó, mint a többi.

Javítási lehetőségek: Külön lista a lyukaknak, méret szerinti rendezés.
Külön lista a leggyakrabban kért méreteknek: **quick fit**.

Virtuális memória

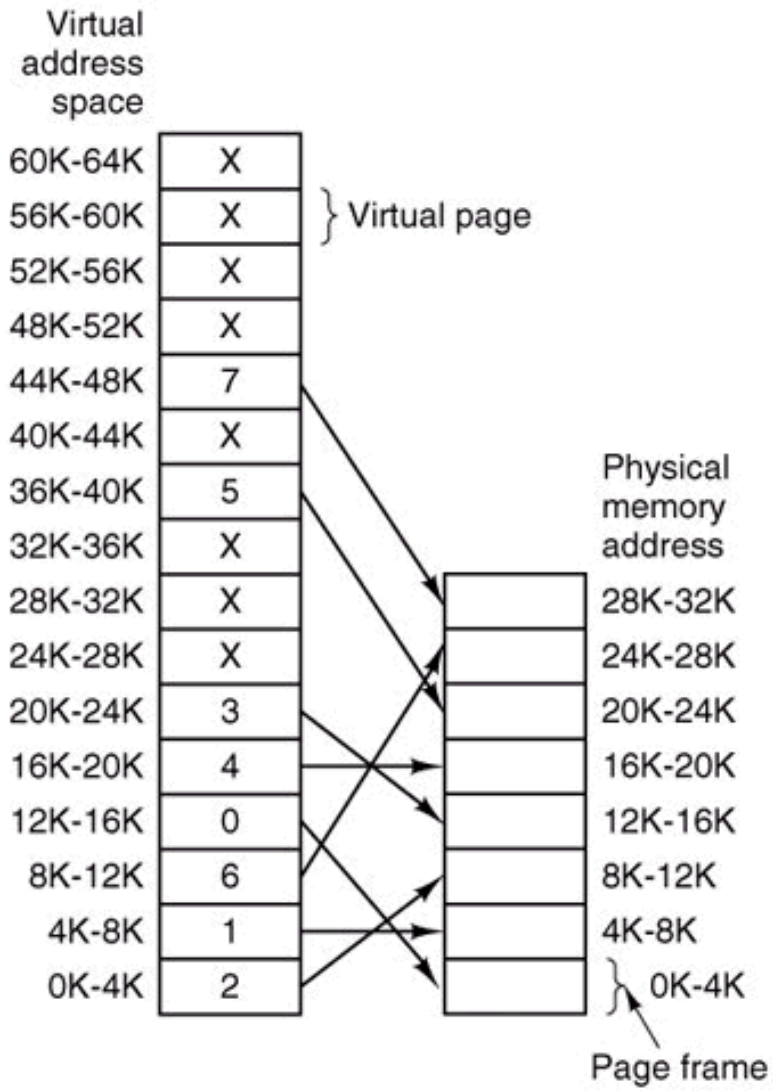
A program, az adat és a verem együttes mérete meghaladhatja a fizikai memória mennyiségét. Az OS csak a program éppen használt részeit tartja a memóriában.

A **virtuális címek** nem kerülnek közvetlenül a memóriasínre, hanem a **memóriakezelő egységbe** (Memory Management Unit - MMU) kerülnek, amely elvégzi a leképezést a **fizikai címekre**.



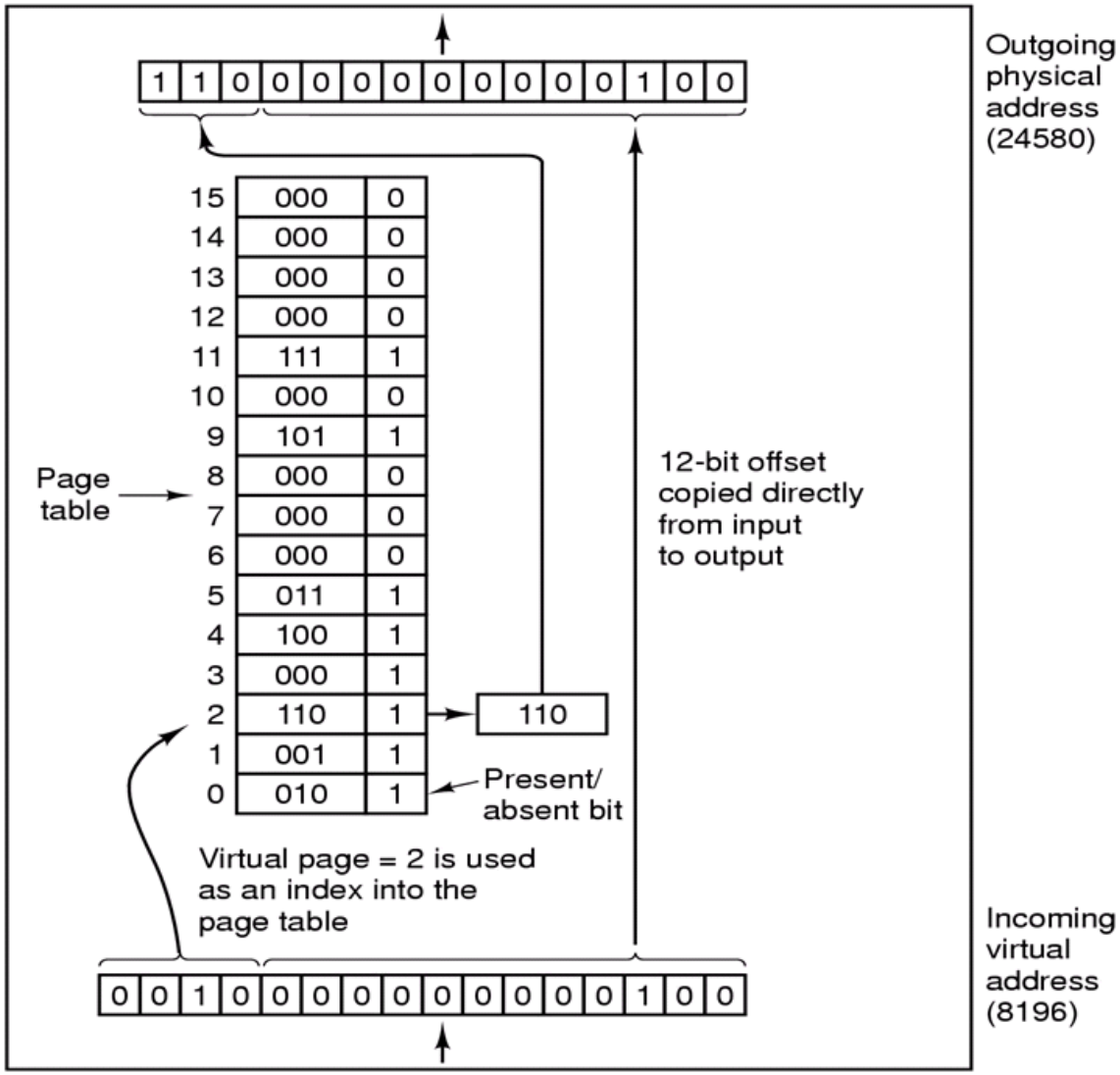
Laptábla

A virtuális címtér egységei a **lapok** (virtual page), ezeknek megfelelő egység a fizikai memóriában a **lapkeret** (page frame).



Az MMU működése

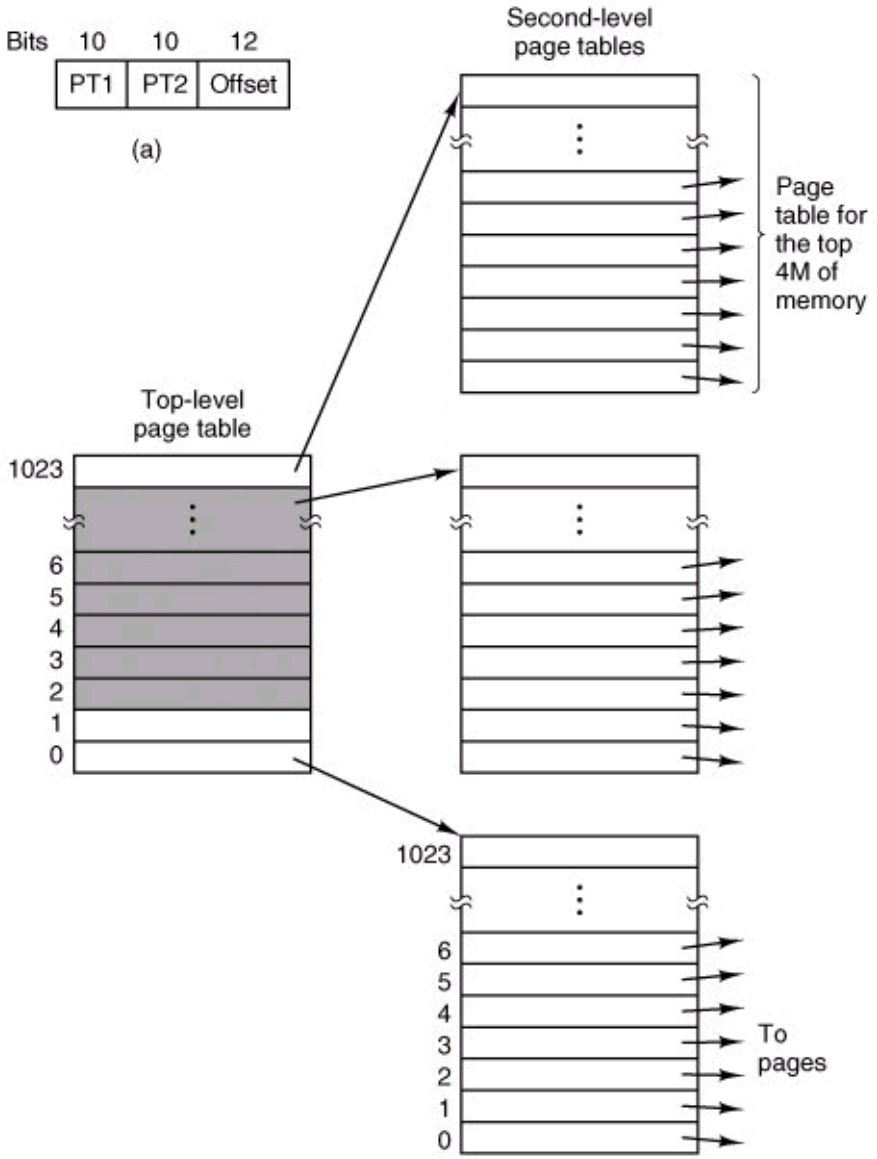
16 darab 4 KB-os lap esetén:



Többszintű laptáblák

32 bites cím két laptáblamezővel,

Egy kétszintű laptábla:



Lapcserélési algoritmusok

Laphiba (page fault): a lap nincs a memóriában, tehát be kell tölteni.

Teendők lapcsere esetén:

1. Ki kell választani egy lapot, amelyik helyére az újat fogjuk betölteni.
2. Kiválasztott lap tartalmának mentése (ha szükséges)
3. Új laptartalom betöltése
4. Laptábla cím kitöltése
5. A laphibát okozó utasítás újrakezdése

Feladat: a felszabadítandó lap kiválasztása.

Sorozatos rossz választás esetén: állandó lapcsere (vergődés).

Az optimális lapcserélési algoritmus

Minden lapot megjelölhetünk azzal a számmal, hogy hány utasítás múlva hivatkozunk rá legközelebb.

Válasszuk a legnagyobb számmal jelölt lapot!

Probléma: nem lehet megvalósítani.

Az NRU lapcserélési algoritmus

Minden laphoz 2 állapotbit tartozik: az M minden íráskor, az R minden hivatkozáskor 1-re állítódik. Az R bitet időnként nullázzuk (pl. óramegszakítás). 4 osztály lehetséges:

1. nem írt, nem olvasott
2. írt, nem olvasott
3. nem írt, olvasott
4. írt, olvasott

Az **NRU (Not Recently Used) algoritmus** véletlenszerűen választ egy lapot a fenti sorrend szerint a nem üres osztályok közül.

Tulajdonságai: egyszerű, közepesen hatékony implementálhatóság, jó teljesítmény.

A FIFO lapcserélési algoritmus

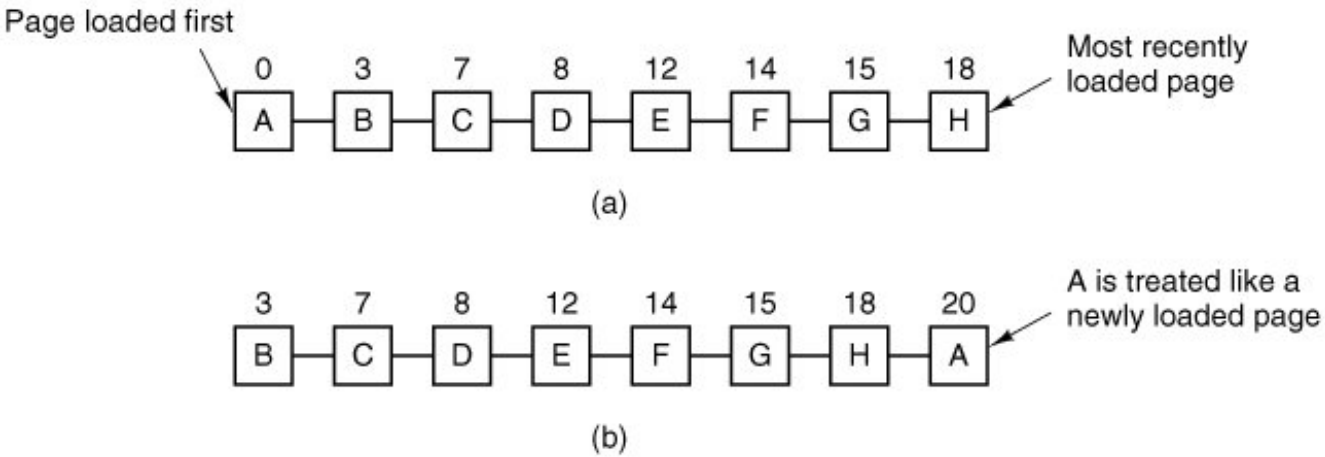
Az OS egy FIFO (First-In, First-Out) listában tárolja a memóriában lévő lapokat. Lista elején van a legrégebbi lap.

Laphiba esetén: az első lapot kidobja, az új lapot a lista végére fűzi.

Hátrány: gyakran használt lapok ugyanúgy kikerülnek, mint a ritkán használtak.

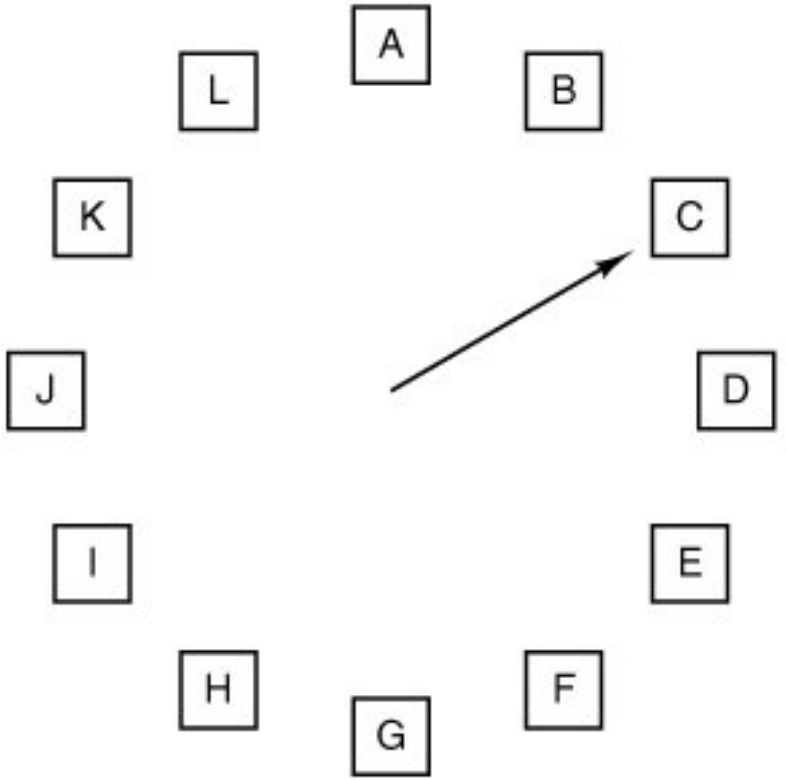
A második lehetőség lapcserélési algoritmus

Második lehetőség (second chance): A FIFO egyszerű módosítása, ha a sor elején levő lap R bitje 1, akkor kinullázzuk és visszarakjuk a sor végére.



Az óra lapcserélési algoritmus

Az **óra** lapcserélési algoritmus lényegében csak implementációban különbözik a második lehetőség algoritmustól.



When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:

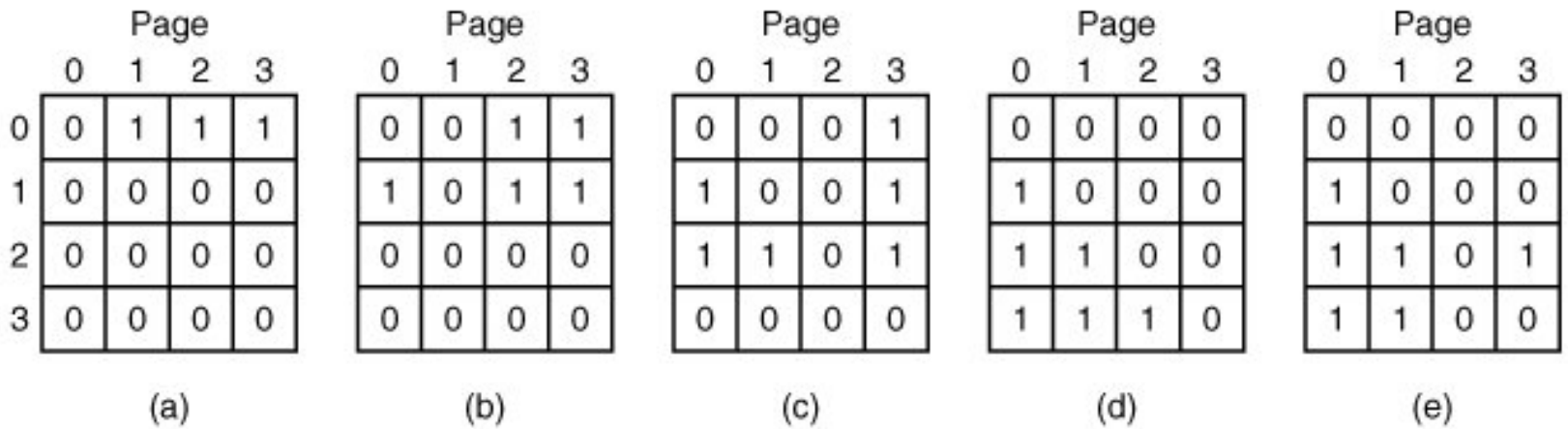
R = 0: Evict the page

R = 1: Clear R and advance hand

Az LRU lapcserélési algoritmus

LRU (Least Recently Used): Laphiba esetén a legrégebben nem használt lapot dobjuk ki. Megvalósítási lehetőségek:

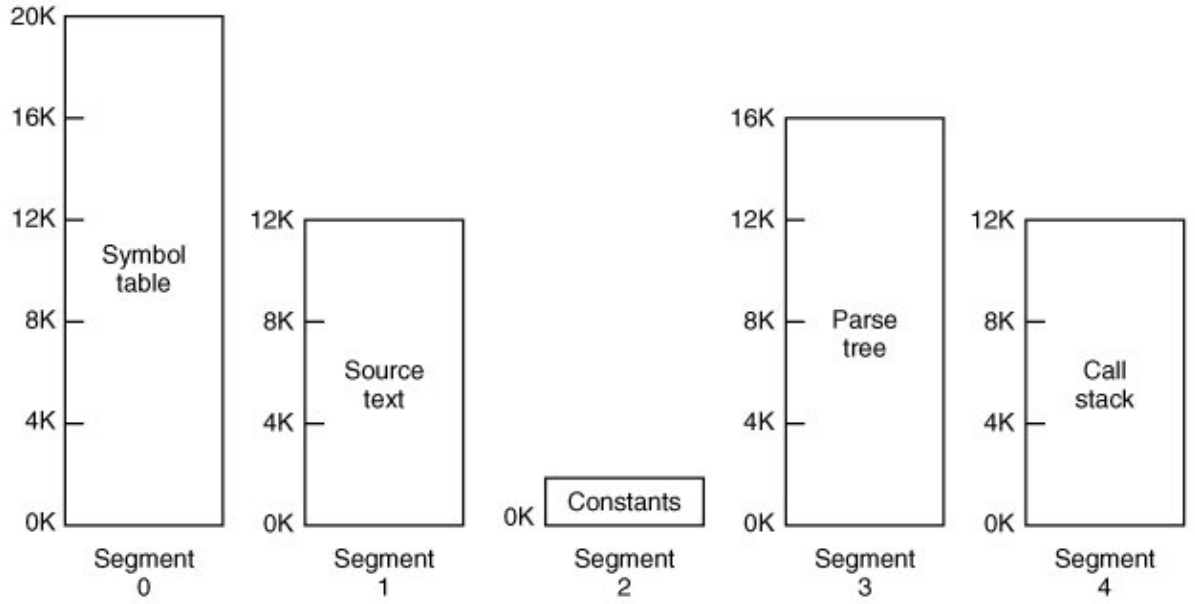
- FIFO sor. Minden hivatkozáskor a lapot áttesszük a sor végére.
- Spec harvder: 1 közös számláló, amit minden hivatkozáskor növelünk, majd letároljuk a hivatkozott laphoz. Laphiba esetén a legkisebb számlálóval rendelkező lapot kell eldobni.
- n lapkeret esetén egy $n \times n$ bitmátrixot kezelünk. k lapkeretre hivatkozás esetén a k-adik sort 1-esre, a k-adik oszlopot 0-ra állítjuk. LRU lap: a legkisebb bináris értékű sor. Példa: 0, 1, 2, 3, 2, ...



Szegmentálás

Szegmensek: egymástól független címterek. 2D memória.

- 0-tól valamilyen max.-ig címezhető.
- Különböző szegmensek eltérő hosszúak, méretük változhat.
- A címek 2 részből állnak: szegmens száma, szegmensen belüli cím.



Szegmentálás tulajdonságai

Előnyök:

- Változó méretű adatszerkezetek könnyen kezelhetőek.
- Programok linkelése egyszerű, ha minden eljárás külön szegmensbe kerül: újrafordítás esetén csak a megváltozott szegmenseket kell kicserélni.
- Függvény-könyvtárak könnyen megoszthatók proc-ok között (pl. grafikus könyvtárak csak egyszer vannak a memóriában).
- Különböző védelmi szintek alakíthatóak ki.

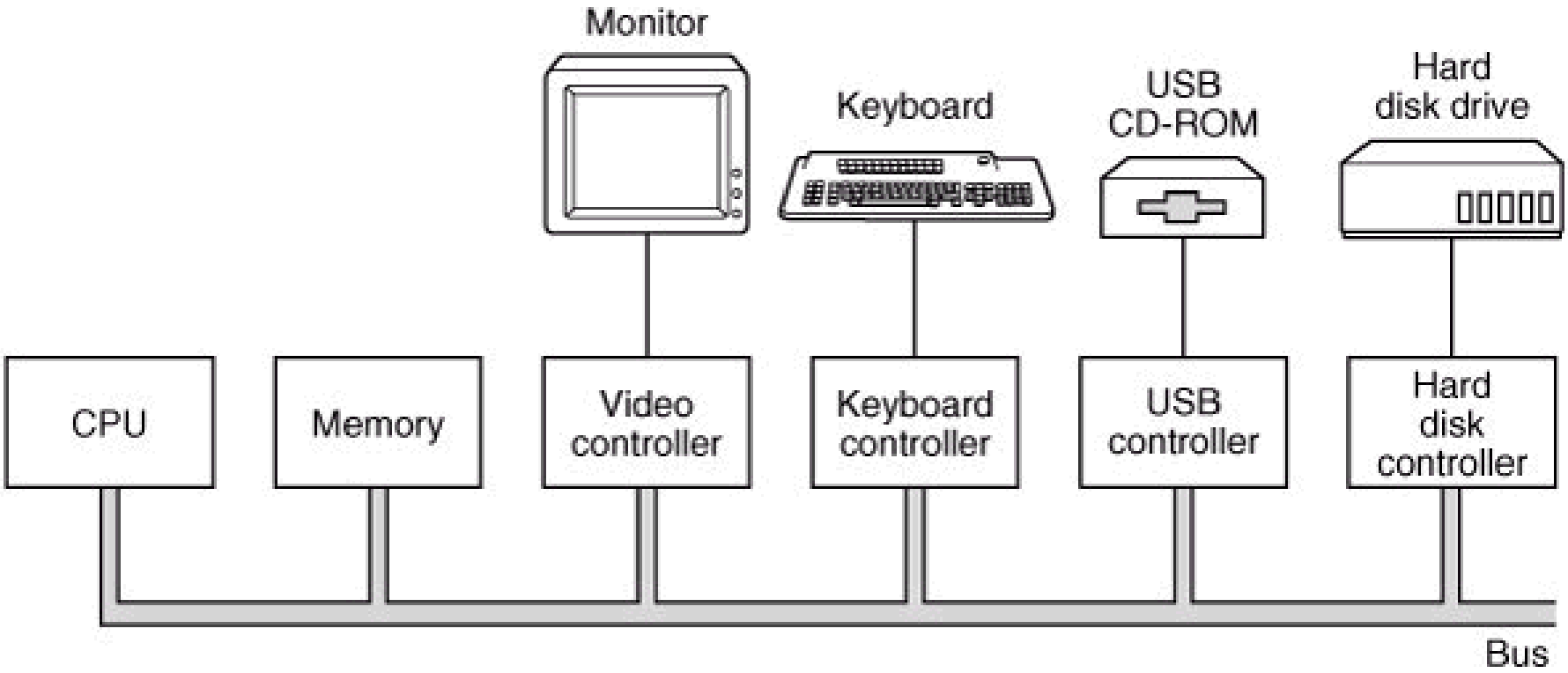
Hátrány: a programozónak tudnia kell, hogy ezt a technikát használja.

I/O eszközök

Blokkos eszköz: az információt adott méretű blokkokban tárolja és a blokkok egymástól függetlenül írhatók és olvashatók (seek művelet). Szokásos blokkméret: 512 bájt és 32768 bájt között. Pl: lemez.

Karakteres eszköz: strukturálatlan karakterfolyam, nem címezhető és nincsen seek sem. Pl: nyomtató, hálózati interfész, egér.

Eszközvezérlők



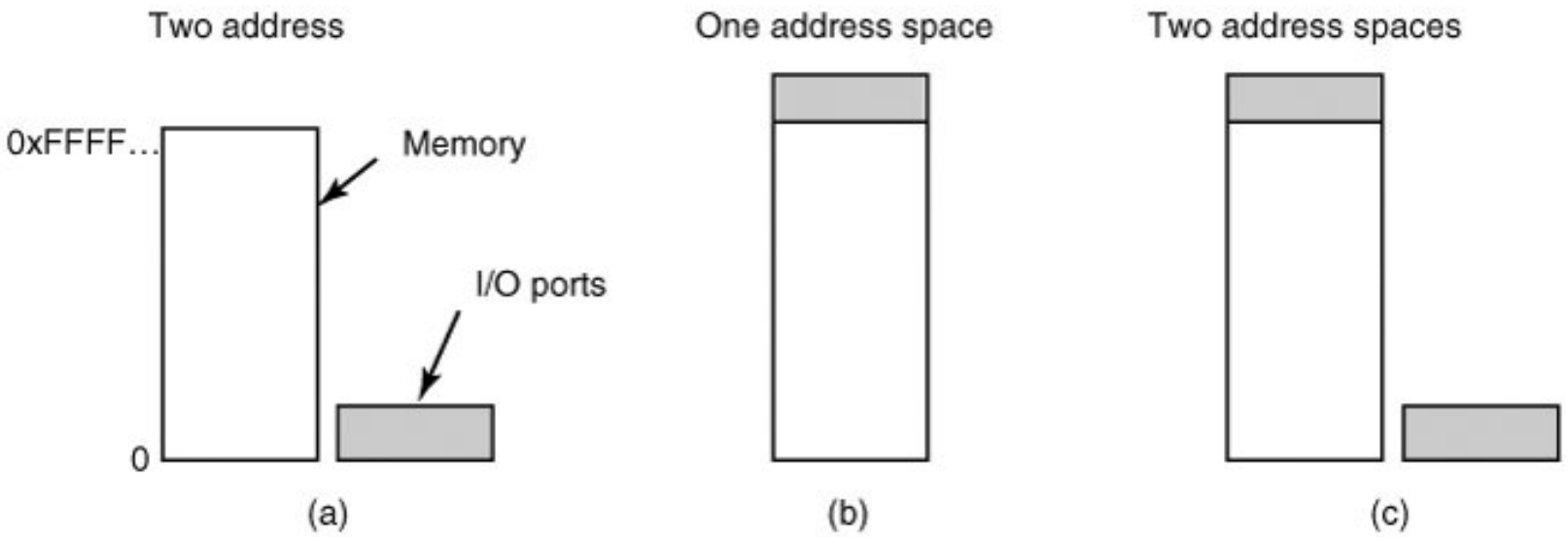
Memórialeképezésű I/O

Kommunikáció az eszközvezérlővel: regisztereken keresztül.

(a) **I/O kapu**n keresztül: `IN reg, port` utasítás a regiszterbe olvas, `OUT port, reg` a portba ír.

(b) **Memórialeképezésű I/O**: a vezérlőregiszterek a memória adott helyén találhatóak, másra nem használhatóak

(c) **Hibrid** megoldás. Pl. a Pentium: 640K-1M adatpufferek + I/O kapuk.



Megszakítások

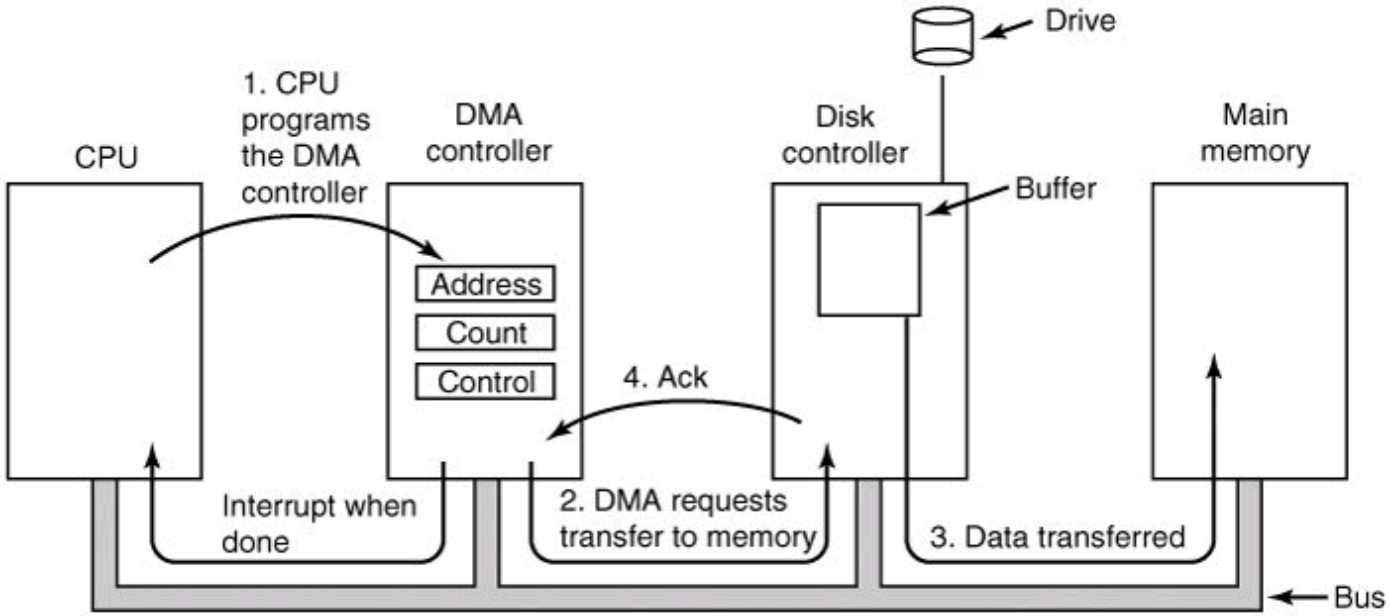
A vezérlő egy megszakításon (IRQ - Interrupt ReQuest) keresztül jelzi egy esemény bekövetkeztét ("művelet elvégezve" vagy "adat készen áll")

Pentiumos PC-n max 15 input lehetséges, néhány előre behuzalozott (pl. billentyűzet)

Plug'n Play: a BIOS indítási időben rendeli hozzá az IRQ megszakításkérést az eszközökhöz.

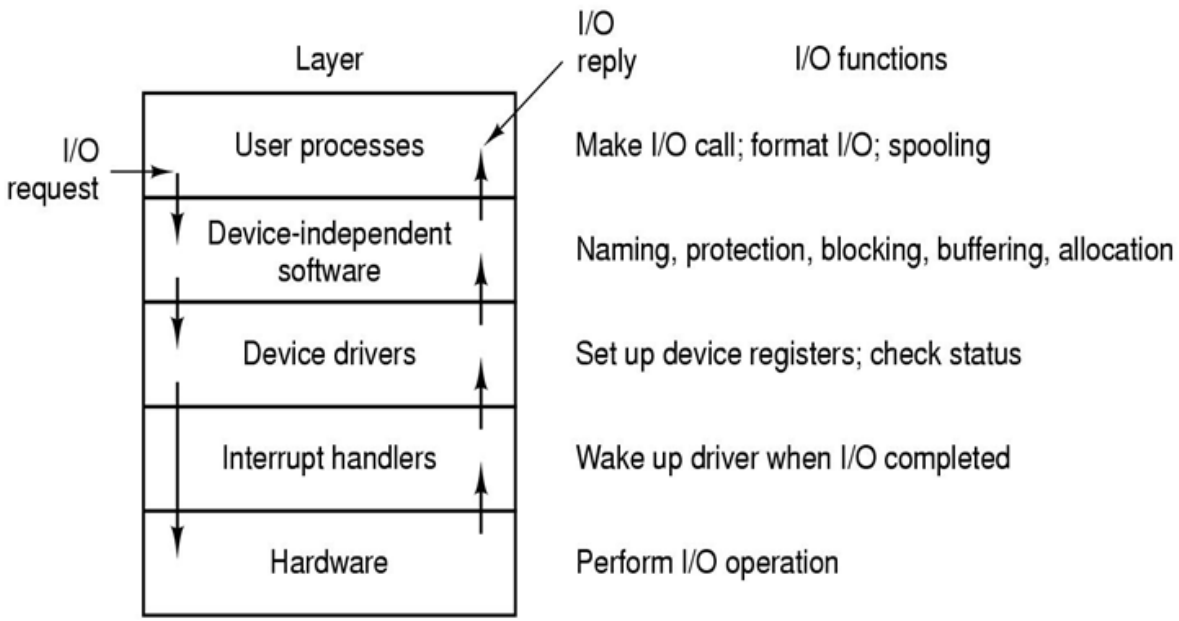
Közvetlen memóriaelérés (DMA)

CPU feladata: adatátvitel kezdeményezése. Paraméterek: Adatátvitel iránya (Be/Ki), Adatok memóriabeli kezdőcíme, Átviteli adatmennyiség (bájtszám).
Eszközvezérlő feladata: CPU-tól függetlenül az adatátvitel megvalósítása. Az adatátvitel során/végén az átviteli állapot lekérdezhető, Hiba/Befejezés esetén megszakítás, állapotinformációk közlése a kezdeményező proccal.



I/O végrehajtási szintjei

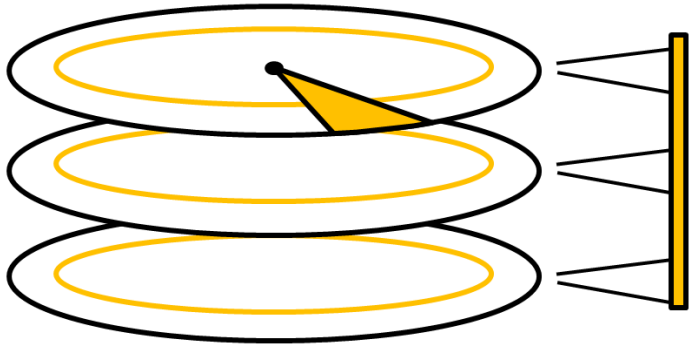
Célok: Egységes (felhasználói) programozási felület, Eszközökre való hivatkozás egységesítése, Blokkmérettől, fizikai felépítéstől, pufferezéstől való függetlenség, Egységes hibakezelés, Osztott használat adminisztrációjától való függetlenség.



Lemezes egységek

Címzés:

- CHS (Cylinder, Head, Sector), Sugár, fej és szög megadásával. Pl. IDE.
- LBA (Logical Block Address), minden szektornak külön azonosító. Pl. SCSI, EIDE.



Cilinder, Pálya, Szektor

Átviteli idők:

- Keresési (seek): cilinderek közötti ugrás
- Fordulási (rotation): szektorok közötti fordulás
- Átviteli (transfer): adatírási/olvasási idő

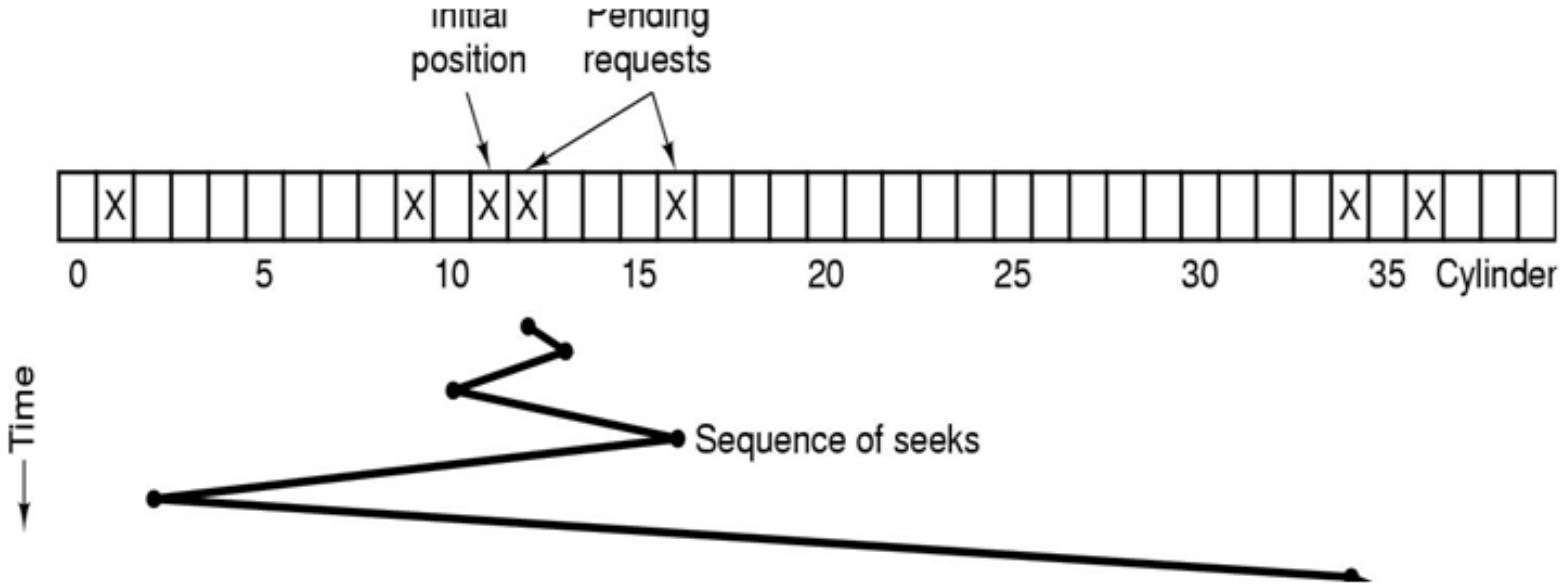
Adatátvitel optimalizálása

- Fordulási, átviteli idők adottak
- Keresési idő befolyásolható az átviteli kérések kiszolgálási sorrendjének meghatározásával

Lemezfej-ütemező algoritmusok

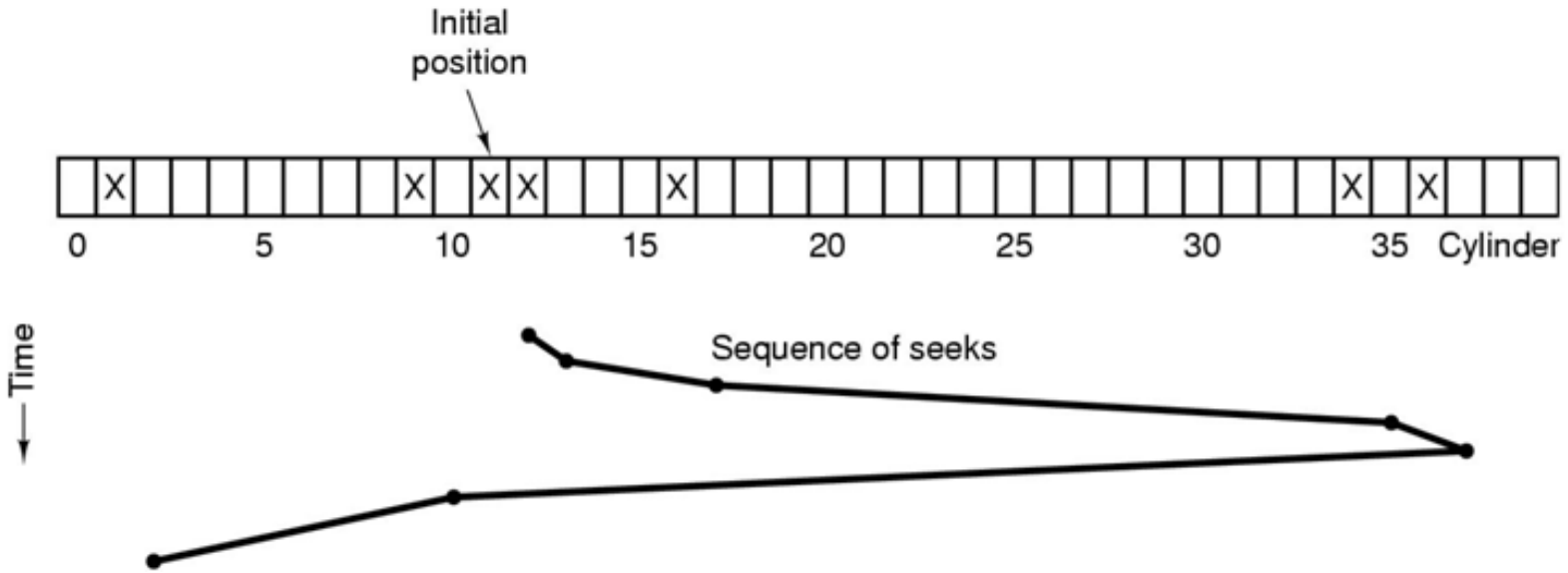
Első kérés - első kiszolgálás (First-Come, First-Served)

Legközelebbit keres elsőként (Shortest Seek First)



Lemezfej-ütemező algoritmusok

Liftes algoritmus (elevator algorithm)



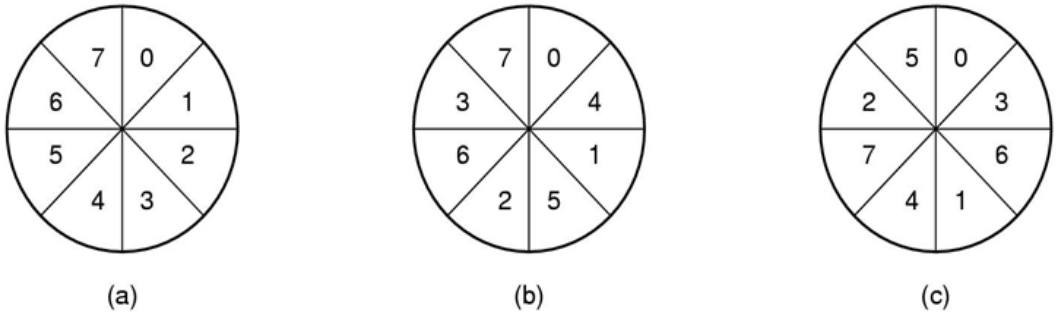
Pályánkénti pufferolás (Track-at-a-Time Caching)

Lemezek kezelése

Szabad/Foglalt szektorok nyilvántartása: Bittérkép vagy Láncolás
Hibakezelés:

- Adathiba: Hibajelző/Hibajavító kódolás
- Keresési hiba: szektorbejegyzések (henger/pálya/szektor)
- Kopás-kímélés: motor ki/be
- Cserélhető adathordozó: csere figyelése, írás befejezettsége

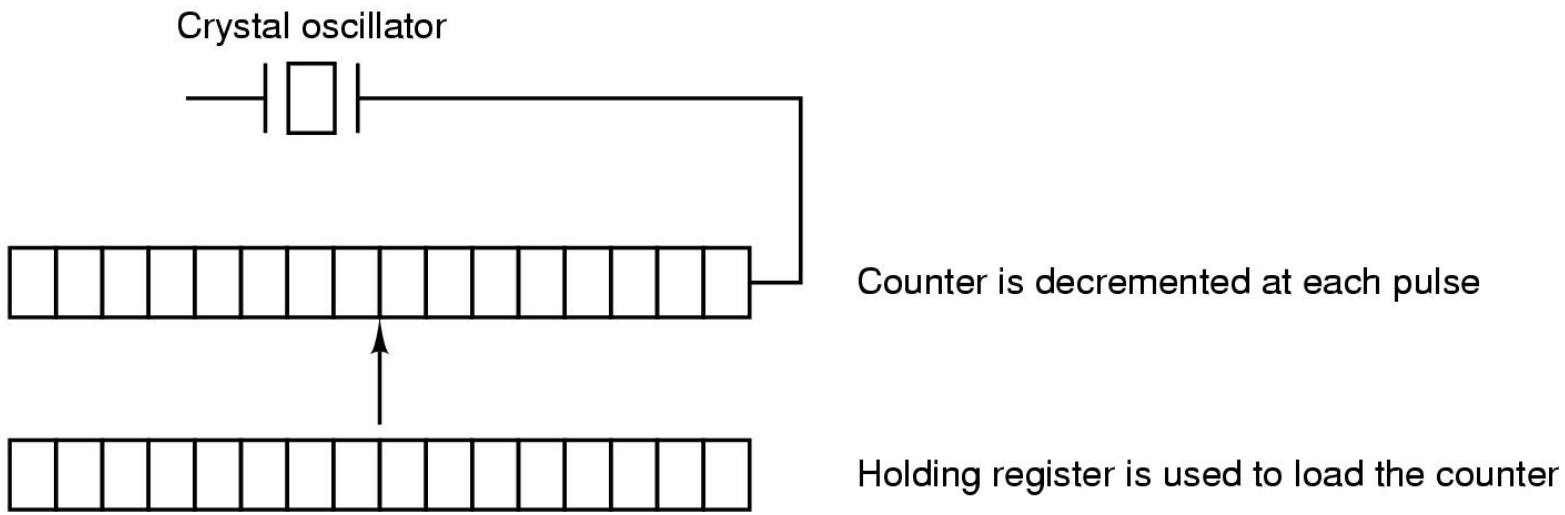
Fájlrendszerek - formázás



Óra (timer)

A hardver

Hw: oszcillátor számlálóval, órajellel (tick)



Pontos idő számítása

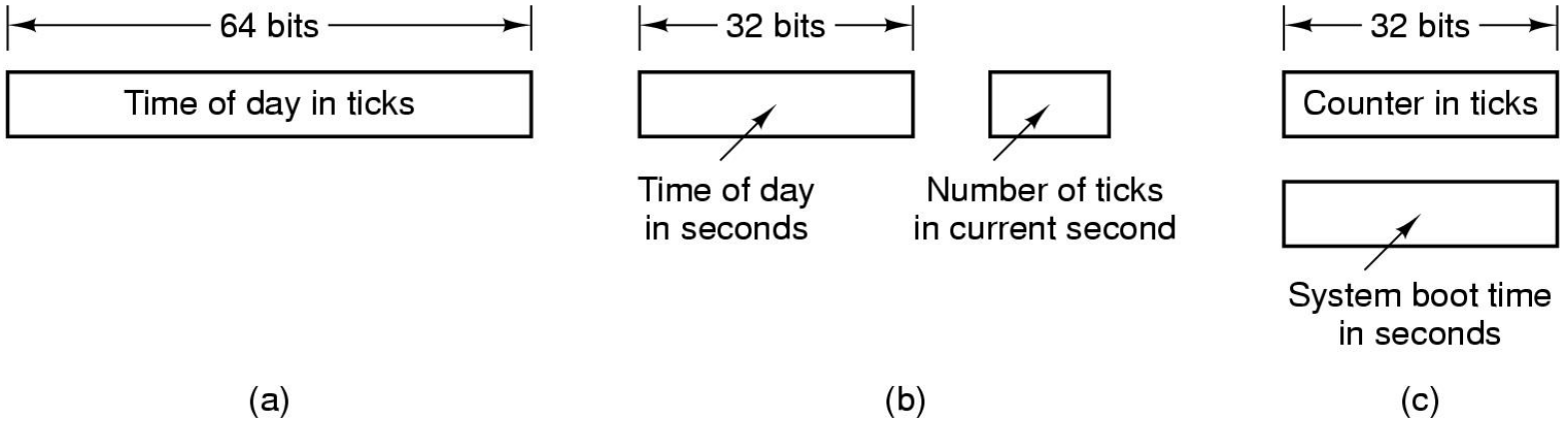
A szoftver

Probléma: 60Hz-es órajellel és 32 bites tárolóval kb. 2 évig mérhetjük az időt. Lehetőségek:

(a) 64 bites tároló

(b) másodperceket mérünk

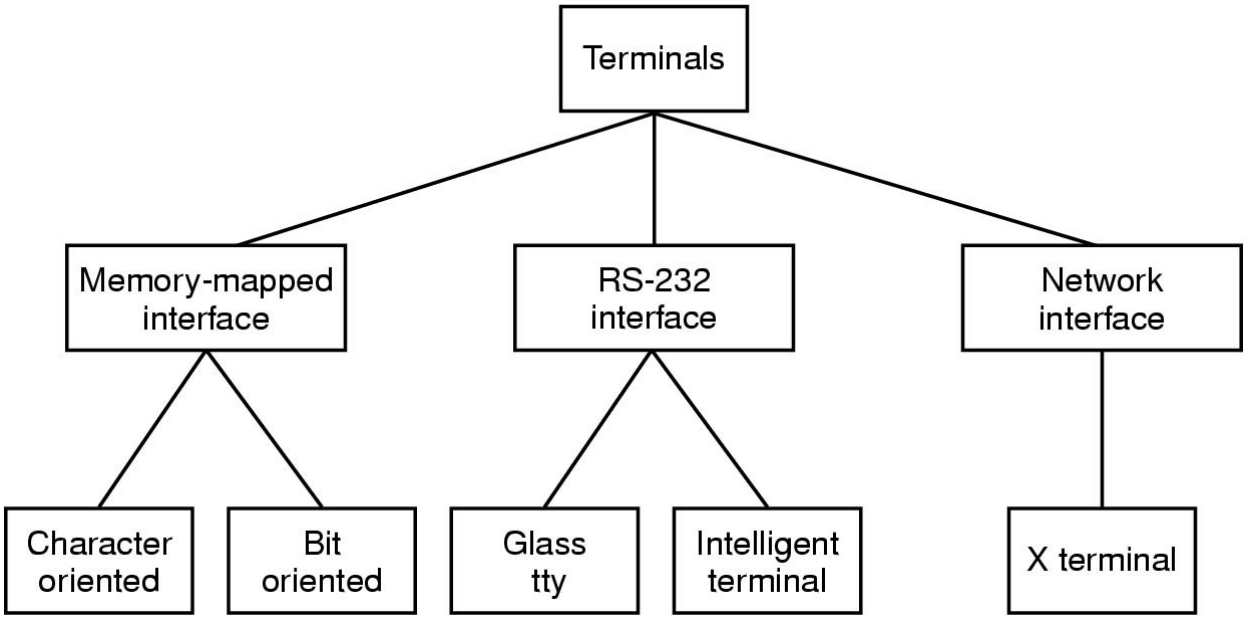
(c) rendszerindítástól számoljuk az órajeleket



Terminálhardver

Terminálok csoportjai:

1. Tárcímleképezéses csatoló
2. RS-232-es interface
3. hálózati csatoló



Fájlrendszerek

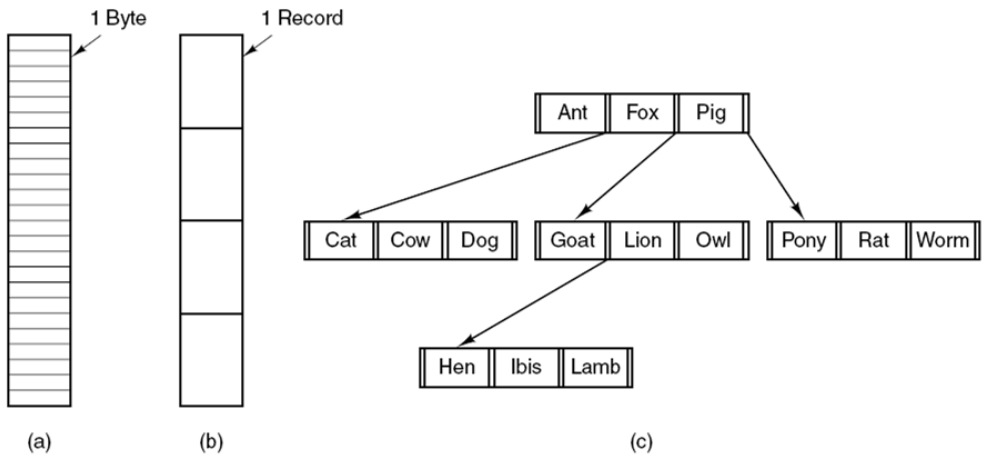
Cél: Nagyméretű adathalmazok tárolása, Processzusokat "túlélő" adatok, Több proc "egyidejű" hozzáférhetősége az adatokhoz

Szerkezet szerint:

(a) Bájtsorozat: Aktuális pozíció, bájtonként/soronként írás/olvasás

(b) Rekordsorozat: Mezők (field), kulcsmező, rekordonkénti írás/olvasás

(c) Fa-struktúra



Fájltípusok

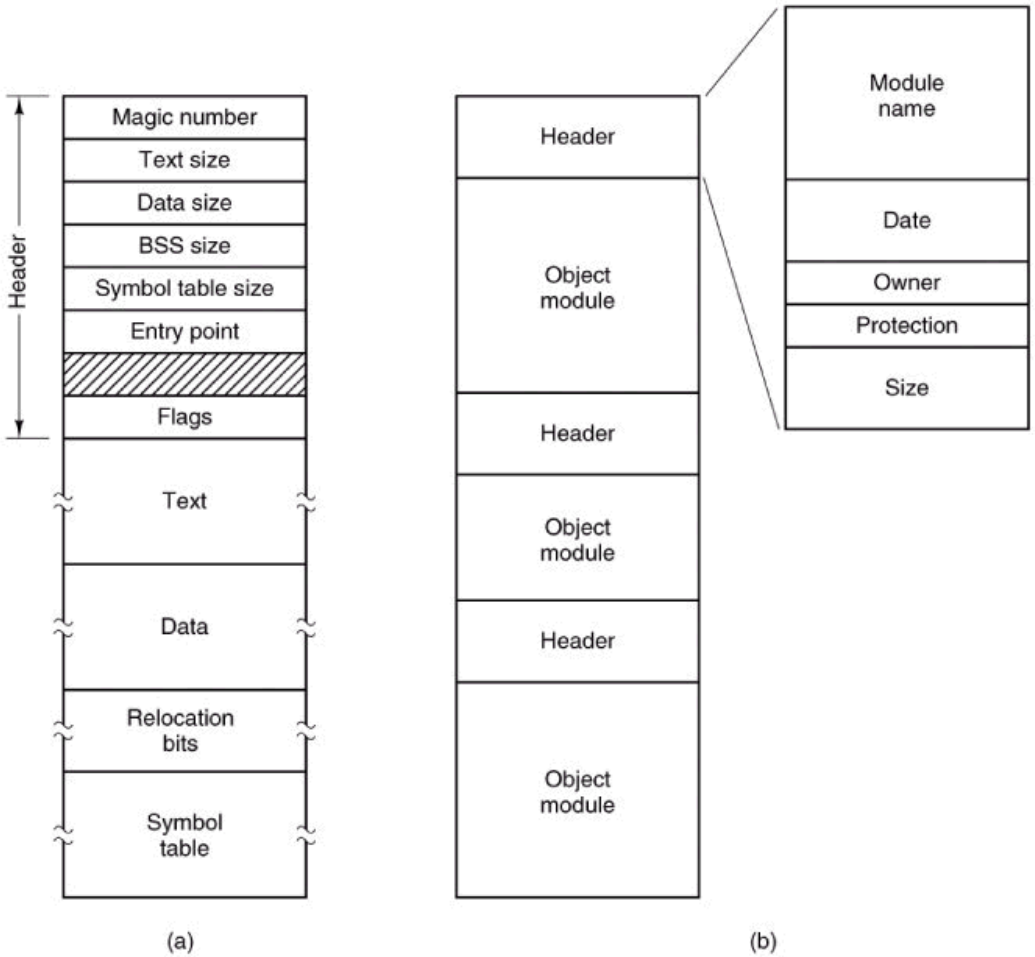
Közönséges fájlok: ASCII vagy bináris

Könyvtárak: rendszerfájlok a fájlrendszer szerkezetének megvalósításához.

Karakterspecifikus fájlok: a soros I/O eszközök modellezésére (Pl. terminál, nyomtató, hálózat)

Blokkspecifikus fájlok: mágneslemezegységek

Fájltípusok



Fájlelérés

Szekvenciális elérés (sequential access): A processzusok megadott sorrendben olvassák a file-okat, a sorrendtől eltérni nem lehet. Kezdés az első bájtnál/rekordnál. Pl. Mágnesszalagok.

Közvetlen elérés (random access): A bájtok/rekordok tetszőleges sorrendben olvashatóak. `read` és `seek` műveletek.

Fájlattribútumok

Minden fájlnak van neve és adattartalma. Ezen felül attribútumok vagy metaadatok lehetnek:

- Létrehozó, tulajdonos
- Tulajdonos/Csoport/Bárki jogok
- Írási/olvasási/láthatósági/végrehajthatási jog
- Létesítési/Hozzáfordulási/Módosítási időpontok
- Bináris/Szöveg/Program jelzés
- Méret/Rekordszám/Rekordméret
- Rendszer/Rejtett/Archív jelzők
- Zárolt (lock)

Fájlműveletek

Létesítés: File létrejöttének bejegyzése, attribútumok beállítása

Törlés: Lemezterület felszabadítása

Megnyitás: Attribútumok beolvasása, néhány lemezcím betöltése

Lezárás: Belső memória terület felszabadítása, utolsó blokk kiírása

Olvasás: Adatok betöltése a mem-ba akt. poz-tól.

Írás: Aktuális pozíciótól az adatok felülírása.

Hozzátoldás: Csak a file végéhez lehet

Pozicionálás: Aktuális pozíció beállítása

Attribútum lekérdezés: Feladatok elvégzéséhez szükséges információk

Attribútum beállítás: Pl. védelmi mód

Átnevezés: Név megváltoztatása vagy másolás és törlés

Zárolás: Egyszerre csak egy proc. férhet a fáljhoz vagy annak egy részéhez

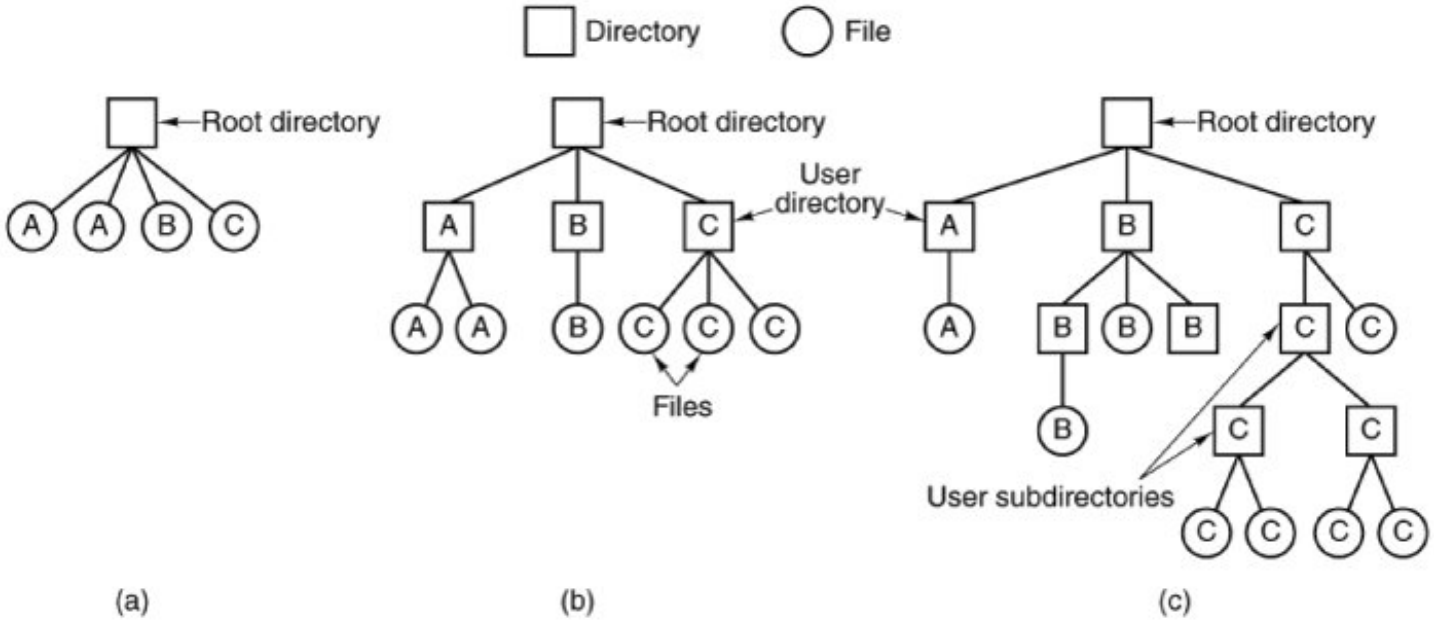
Fájlműveletek

```
FILE* fopen (char* filename, char* mode ); // létrehozás, megnyitás
int   fclose ( FILE* stream );             // lezárás
int   fflush ( FILE * stream );            // buffer fájlba írása

size_t fread ( void* ptr, size_t size, size_t count, FILE * stream );
size_t fwrite ( void* ptr, size_t size, size_t count, FILE * stream );

int   fprintf ( FILE* stream, char* format, ... ); // formázott írás
int   fputs ( char* str, FILE* stream );           // sztring írás
int   fputc ( int character, FILE* stream );       // karakter írás
int   fscanf ( FILE* stream, char * format, ... ); // formázott olvasás
char* fgets ( char* str, int num, FILE* stream );  // sztring olvasás
int   fgetc ( FILE* stream );                      // karakter olvasás

int   fseek ( FILE* stream, long int offset, int origin ); //pozícionálás
long int ftell ( FILE* stream );           // pozíció lekérdezése
int   feof ( FILE* stream );               // vége?
```

Könyvtárszerkezetek

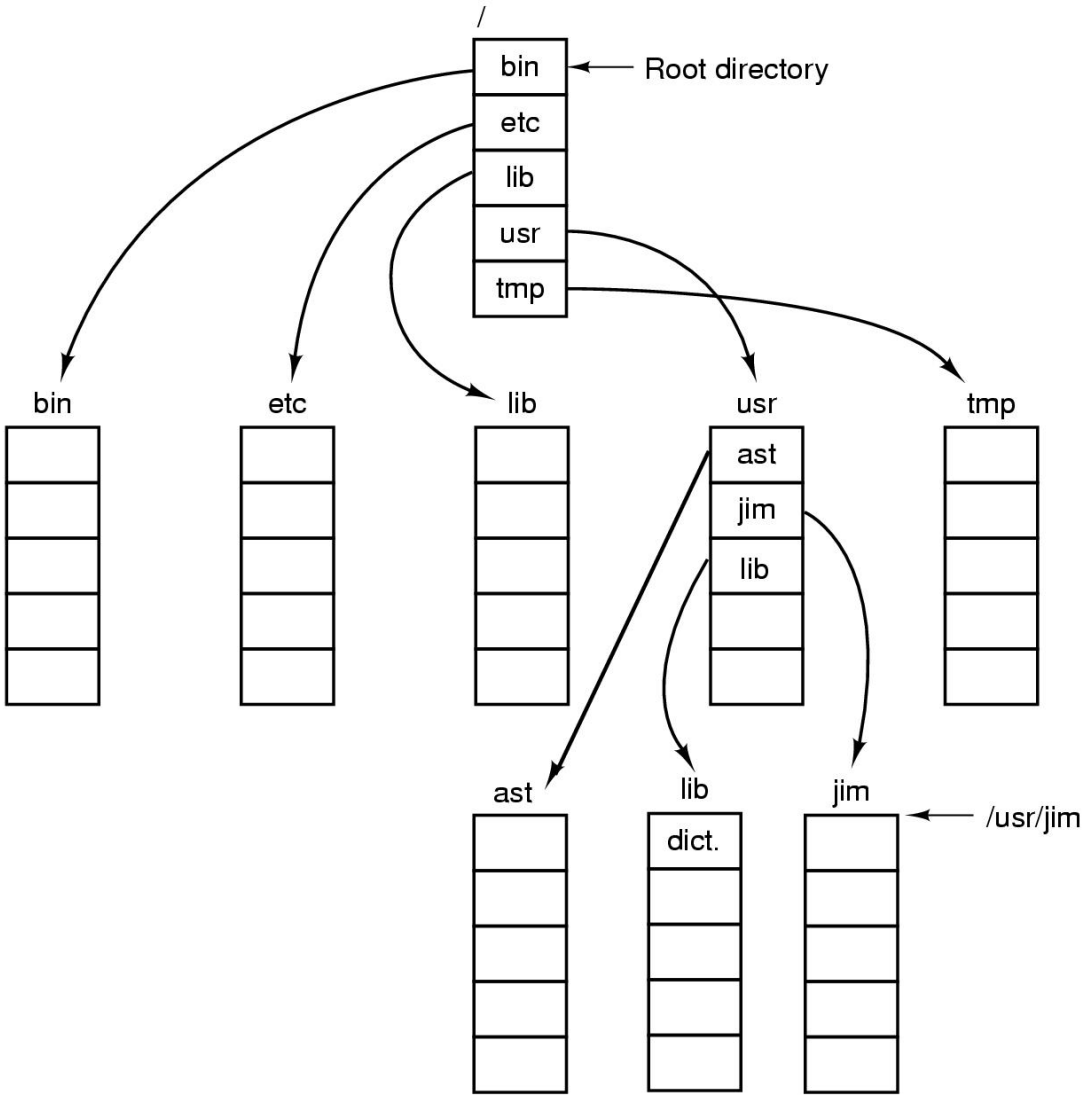
Egyszerű könyvtárszerkezet: Egyetlen könyvtár (a), vagy felhasználónként külön könyvtár (b). Pl. beágyazott rendszerek, PDA-k, digitális fényképezőgépek.

Hierarchikus könyvtárszerkezet: tetszőleges számú alkönyvtár (c). Pl. modern PC-k, fájlszerverek.



Útvonal megadása

Abszolút vs. relatív útvonal



Könyvtári műveletek

Létrehozás (create). Új, üres könyvtár

Törlés (delete). Csak üres könyvtár

Megnyitás (opendir). Olvasható lesz a könyvtár, Pl. listázás

Lezárás (closedir). Olvasás után a memória felszabadítása

Olvasás: (readdir). könyvtári bejegyzés olvasása

Átnevezés (rename). Hasonlóan a file-okhoz

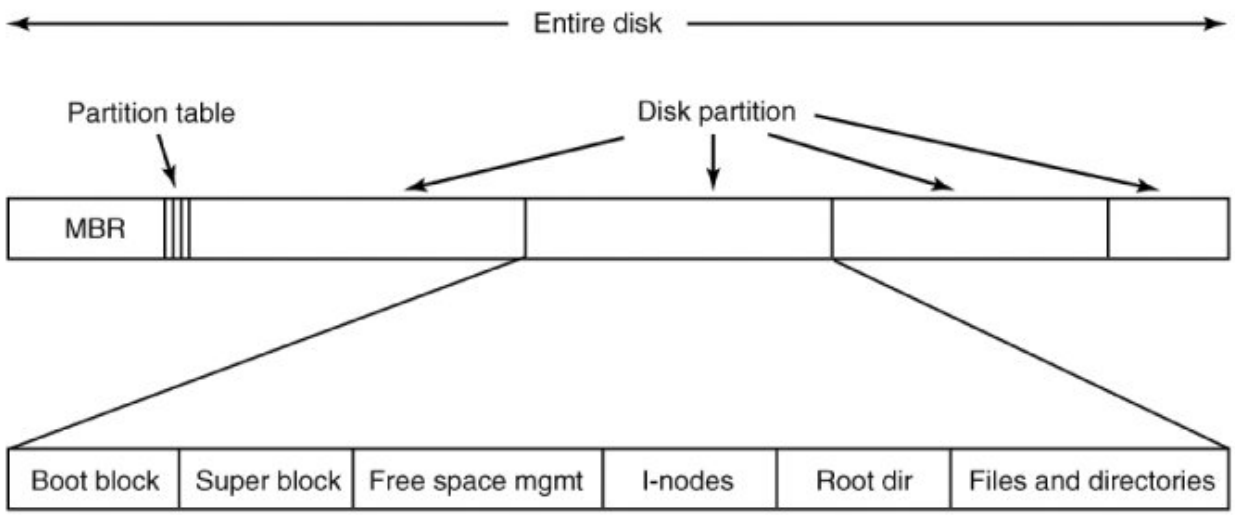
Kapcsolás (link). Merev kapcsolatok létrehozása (hard link).

Lekapcsolás (unlink). Egy könyvtári bejegyzés törlése

Fájlrendszerszerkezet

MBR (Master Boot Record) A lemez 0. szektora. A BIOS betölti és végrehajtja az MBR-ben lévő kódot. Az MBR program keresi meg az aktív partíciót, majd beolvassa és elindítja a partíció első blokkját.

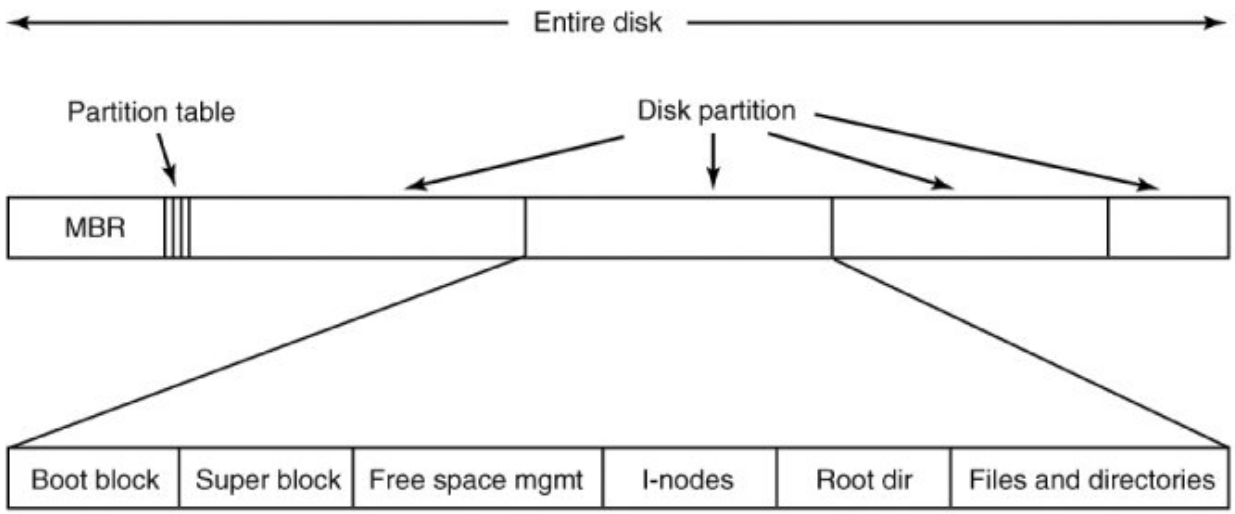
Partíciós tábla: Az MBR végén, tartalmazza a partíciók kezdetének és végének címeit. **Elsődleges partíció** max 4 db (PC-n). Egy **kiterjesztett partíció** tartalmazhat tetszőleges számú **logikai partíciót**.



Fájlrendszerszerkezet

Partíciók

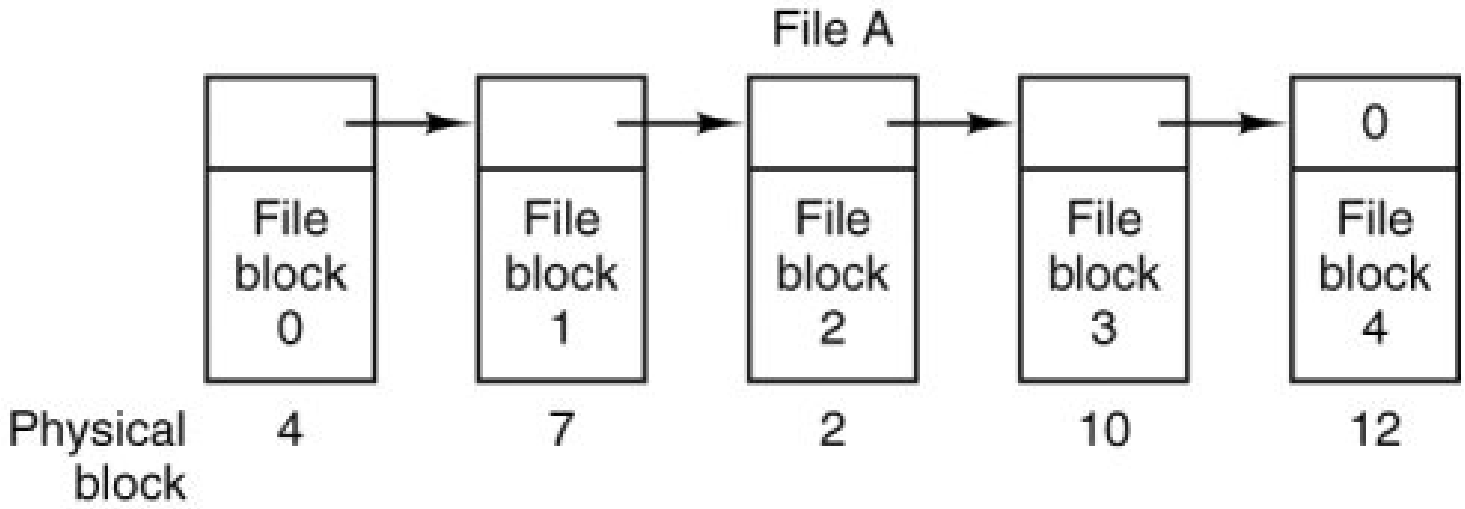
Indítóblokk (Boot block): a partíció első blokkja, feladata, hogy betöltse a partíción lévő os-t.
Szuperblokk: tartalmazza a fájlrendszer adatait.
Szabadterületkezelő: infó a szabad blokkokról.
I-node tábla: minden fájlhoz egy bejegyzés: attribútumok és az adatblokkok.



Fájlok megvalósítása

Egymást követő blokkok sorozata. Előny: egyszerű megvalósítás, elegendő az első blokkot és a hosszt tárolni, gyors olvasás. Hátrány: szabad területek elaprózódnak, méretet létrehozáskor ismerni kell. Pl. mágnesszalagok, CD, DVD

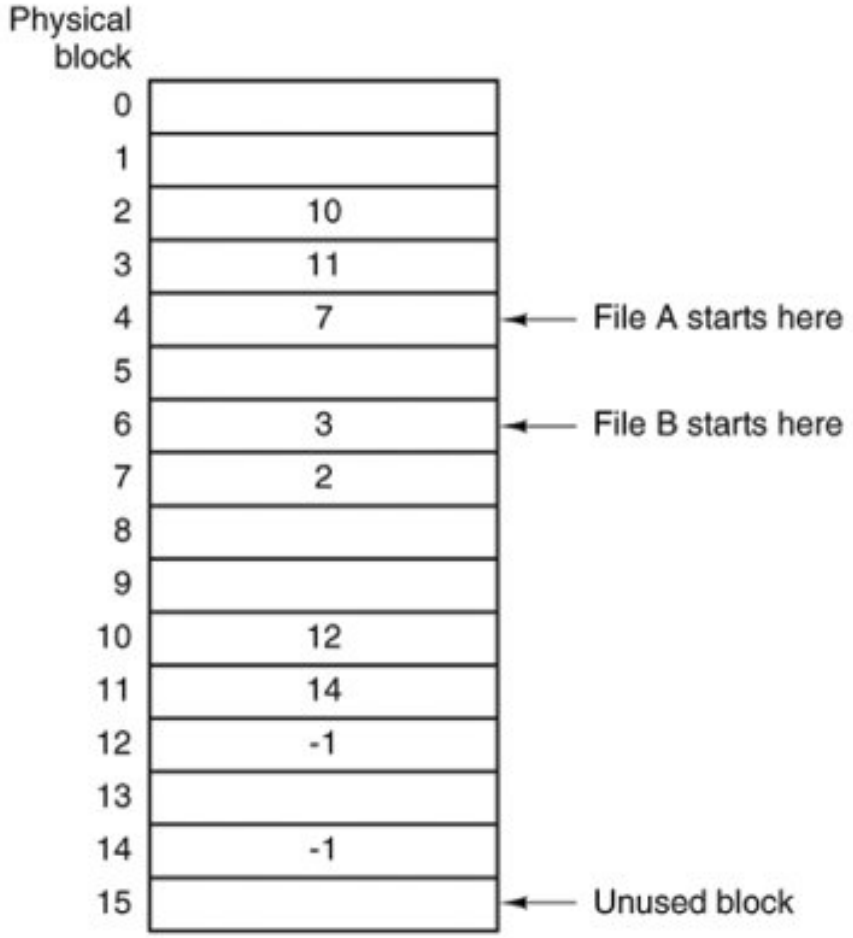
Láncolt listák. Lemezblokkokat láncra fűzzük, az első szó a következő blokkra mutat. utolsó blokkban belső töredezettség. Előny: nincs elaprózódás, szekvenciális olvasás egyszerű. Hátrány: közvetlen elérés lassú a sok fej-pozícionálás miatt, adatok mérete nem kettőhatvány.



Fájlok megvalósítása

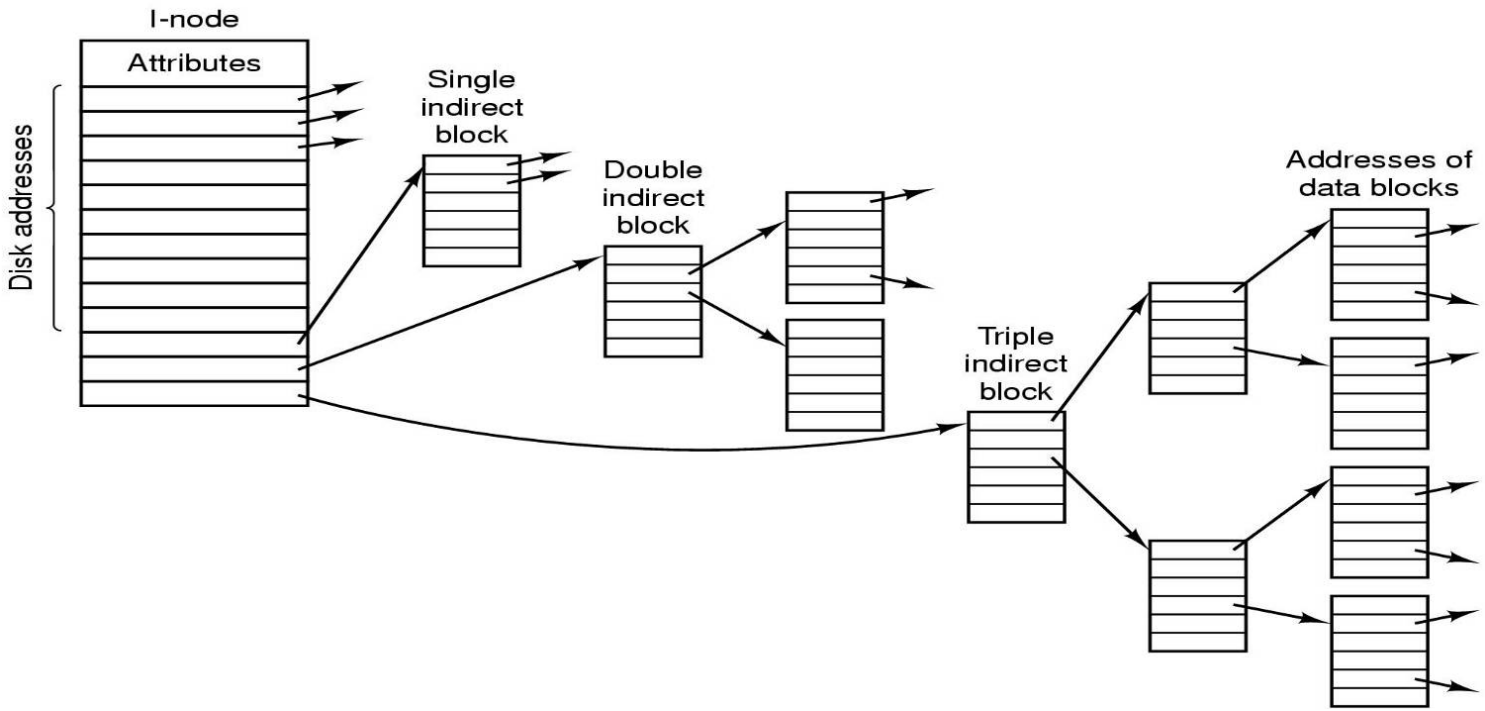
Láncolt listák a memóriában. FAT (File Allocation Table).
Előny: nincs elaprózódás, olvasás gyors, teljes blokk használható. Hátrány: a lemez méretével együtt nő a tábla mérete. Ha a lemez 20GB, és a blokkméret 1KB, akkor a tábla mérete 80MB (32 bites bejegyzések esetén).

PI. MS-DOS: FAT16,
Windows 98: FAT32 (28-bites)



Fájlok megvalósítása

I-nodeok. Előny: Csak a nyitott fájlok i-nodejai vannak a memóriában, a memóriaköltség nem függ a lemez méretétől. Hátrány: Hosszú fájlok esetén indirekt blokkokat kell alkalmazni.



Könyvtárak megvalósítása

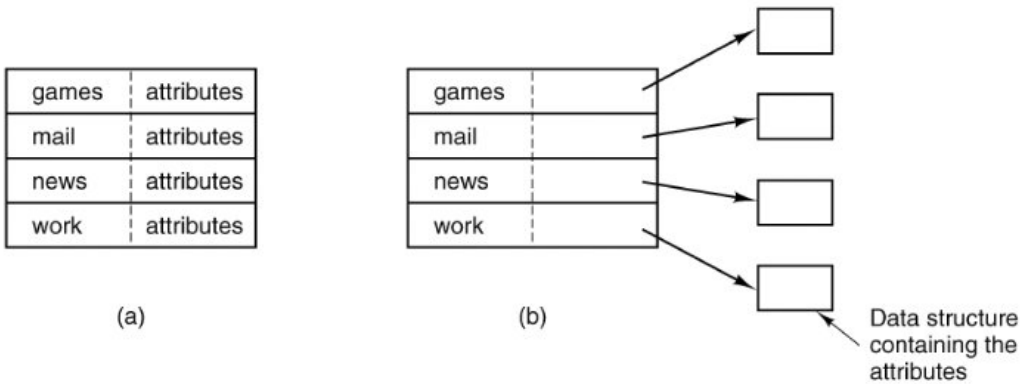
Gyökérkönyvtár helye:

- Rögzített helyen a partíción belül
- Unix: szuperblokk tartalmazza az I-nodetábla helyét, aminek első bejegyzése a gyökérkönyvtár
- Windows XP: bootszektor tartalmazza az MFT-t (Master File Table)

Hol tároljuk az attribútumokat:

(a) A könyvtári bejegyzésben

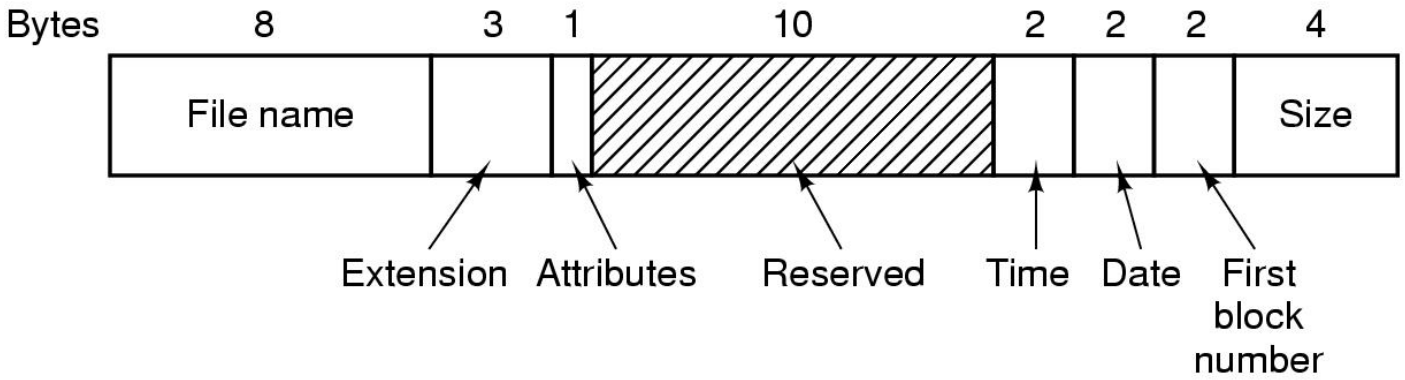
(b) Külön adatszerkezetben (pl. i-node)



Könyvtárak megvalósítása

MS-DOS könyvtárak

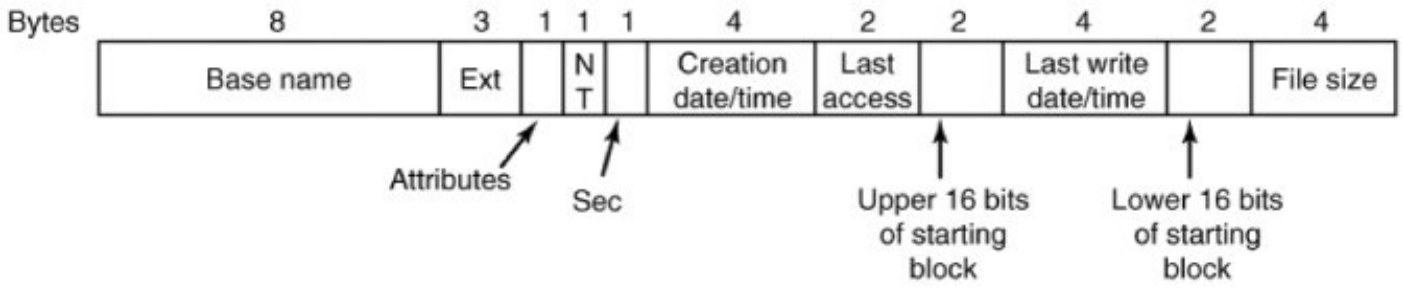
MS-DOS könyvtárbejegyzés:



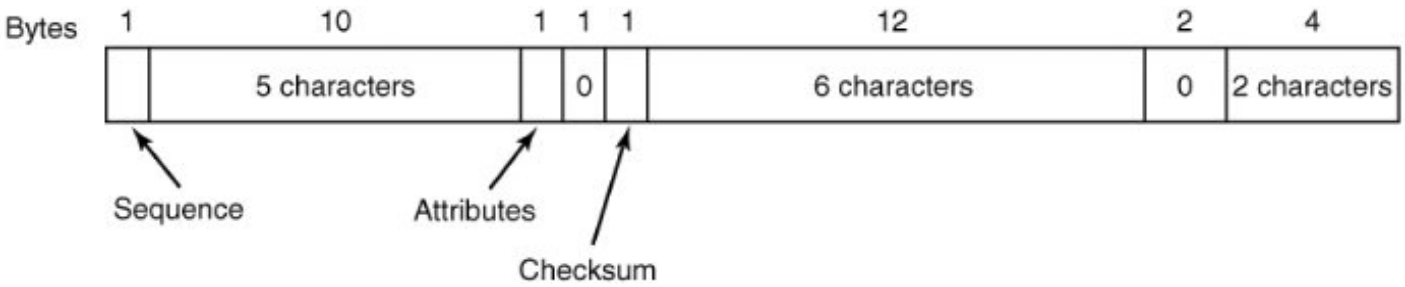
Könyvtárak megvalósítása

Windows 98 könyvtárak

Windows 98 alap könyvtárbejegyzés:



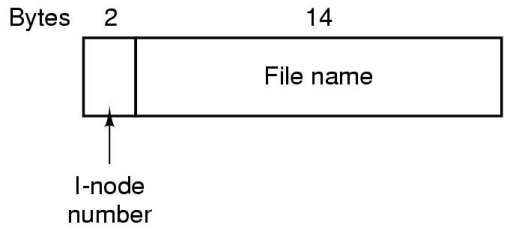
Hosszú fájlnév a Windows 98-ban:



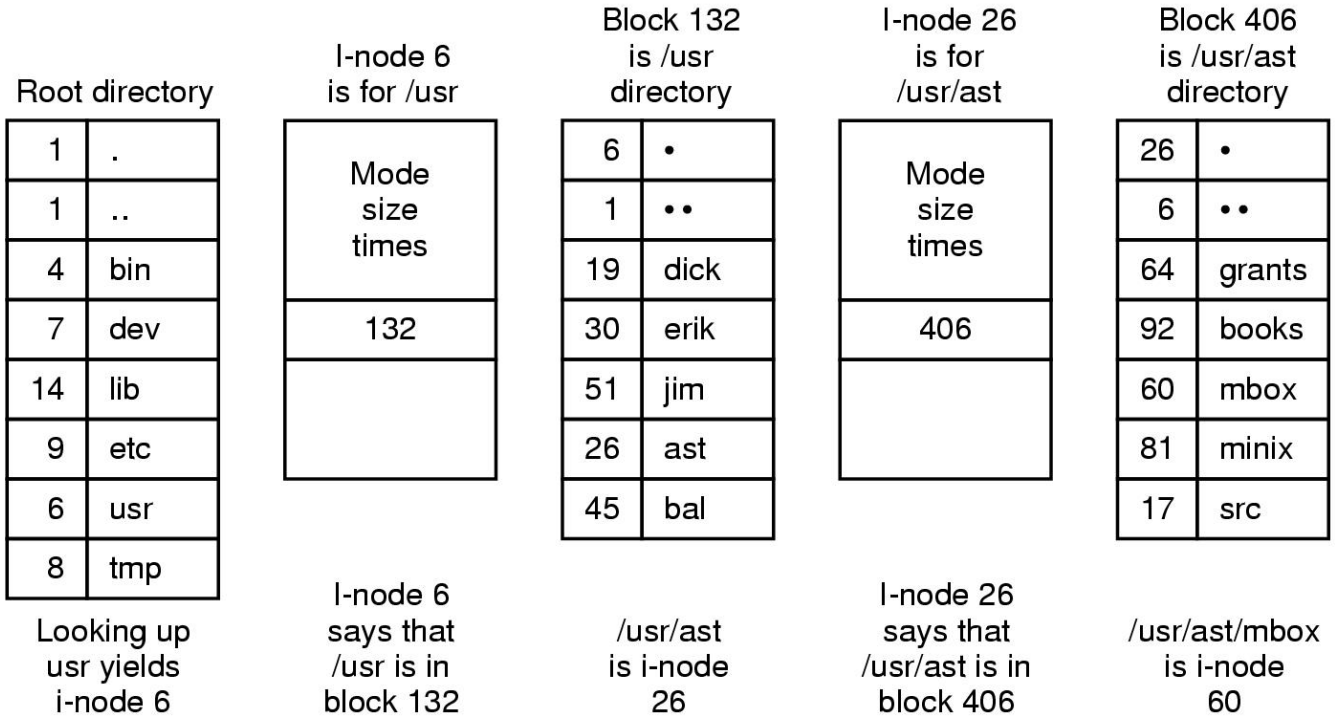
Könyvtárak megvalósítása

UNIX V7 könyvtárak

Unix-könyvtárbejegyzés:



A `/usr/ast/mbox` keresésének lépései:



Könyvtárak megvalósítása

NTFS könyvtárak

Tulajdonságok:

- 255 karakteres file nevek és 32 767 karakteres útvonalak
- Unicode a file nevek képzéséhez, de: problémás a rendezés
- Kvóta
- Védelem, titkosítás
- Adattömörítés

Problémák megoldása attribútumok bevezetése: A file nem más mint attribútumok gyűjteménye. Az adat is egyfajta attribútum. Több adatsor is lehetséges.

Master File Table (MFT): minden fájl számára tartalmaz egy bejegyzést. Egy bejegyzés 16 attribútumot tartalmazhat, mindegyik max 1KB. Egyes attribútumok lehetnek mutatók, újabb attribútumokra.

Blokkméret

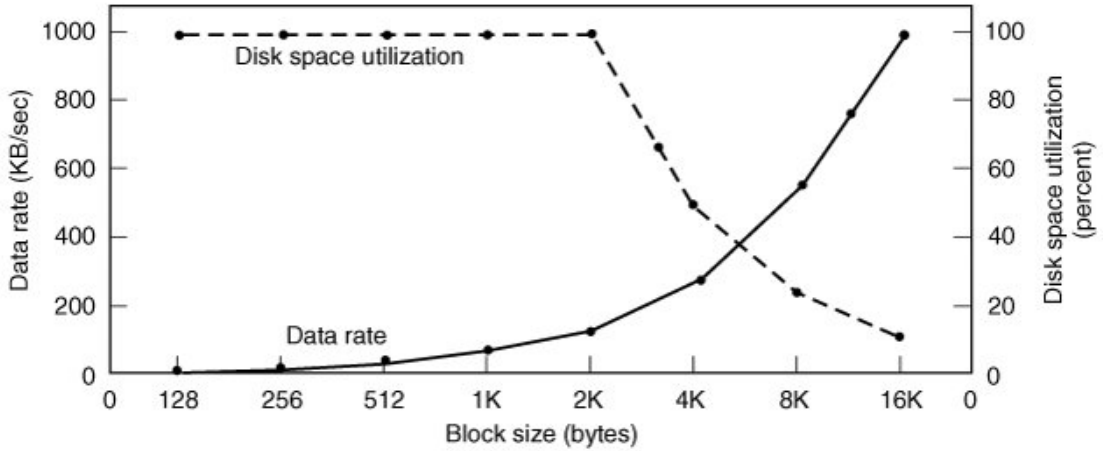
Hogyan válasszuk meg a lemez-blokkok méretét?

- Ha túl nagy, akkor a kicsi fájlok sok területet foglalnak.
- Ha túl kicsi, akkor lassú lesz az olvasás (sok pozícionálás)

Mekkora egy átlagos fájl?

- 1984-ben kb. 1KB
- 2005-ben 2475 bájt (medián)

Feltételezzük, hogy minden fájl mérete 2KB:



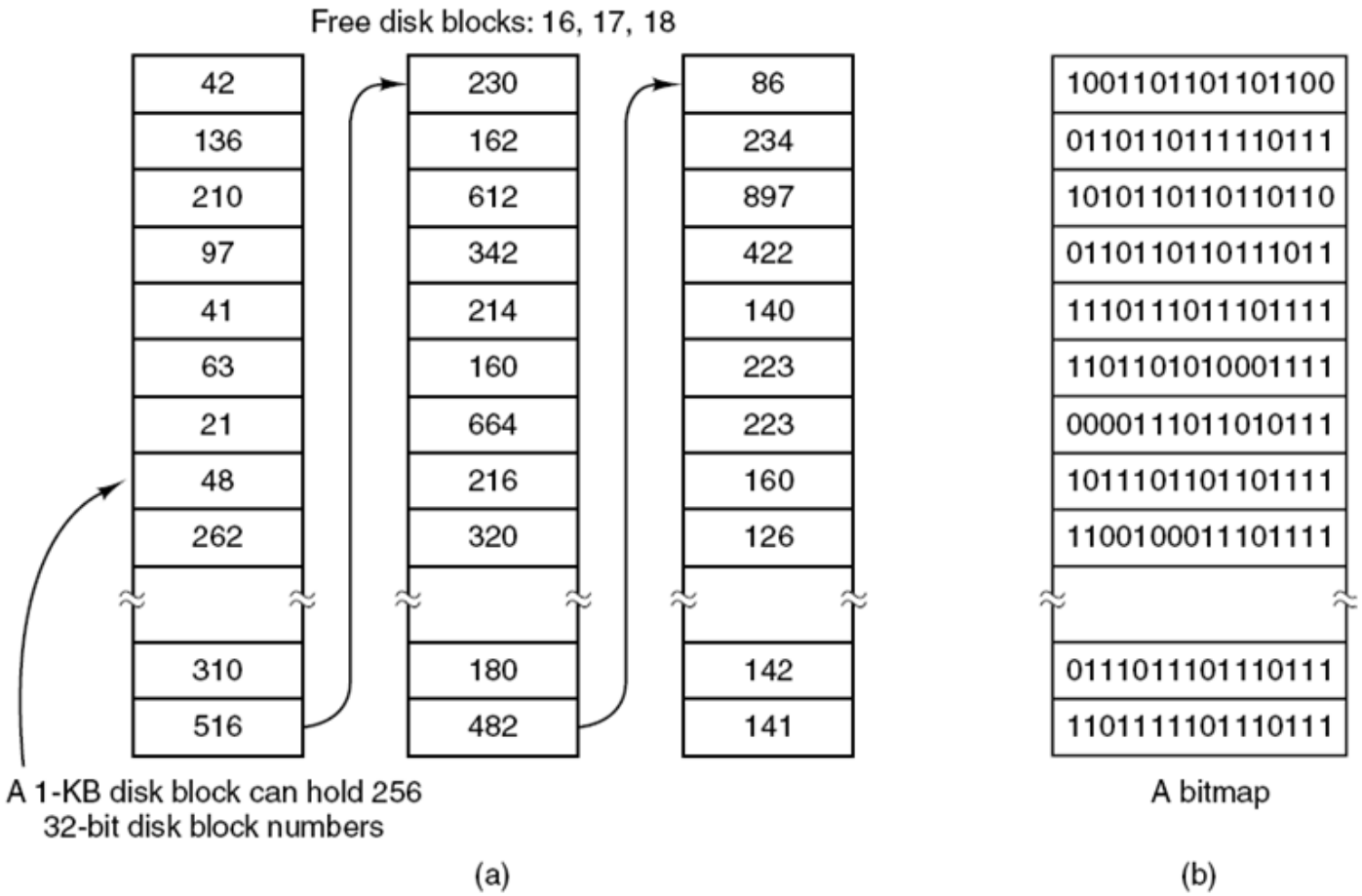
Tehát a 4KB jó választás, de Pl. UNIX-ban 1KB az általános.

Szabad blokkok nyilvántartása

Két módszer használnak széleskörben:

(a) Szabad blokkok láncolt listában

(b) Bittérkép



Mentések

Mentés célja: Helyreállítás katasztrófa esetén vagy hiba esetén. Hosszú idő, nagy hely igény

Inkrementális mentés: Teljes mentés havonta vagy hetente. Naponta csak azokat a file-okat kell menteni, amelyek megváltoztak a teljes mentés óta. Csak az utolsó mentés óta megváltozott file-okat mentjük.

Bonyolult a helyreállítás: Először a teljes mentést kell visszaállítani Aztán az inkrementális mentéseket fordított sorrendben

Tömörítés

Mentések

Fizikai mentés: 0. blokk-tól minden blokk kiírása egy szalagra. Nem használt blokkok? Ki kell írni a sorszámát is a blokknak. Hibás blokkok mentése vagy blokkok átrendezése? Előnye: Egyszerű és nagyon gyors. Hátránya: nem lehet fájlokat/könyvtárakat kihagyni, nem inkrementális, nem lehet egyedi fájlokat helyreállítani.

Logikai mentés: Egy vagy több kijelölt könyvtárban lévő file-t és könyvtárat ment rekurzívan. Menteni kell a file útvonalát, a könyvtárszerkezetet, az attribútumokat. Előnye: Egyszerűem helyre lehet állítani egyedi file-okat és könyvtárakat. Hátránya: szabad blokkok listáját külön kell kezelni, merev láncokat csak egyszer szabad helyreállítani.

File-rendszerek konzisztenciája

Konzisztencia sérülhet ha pl. nem minden módosult blokk került kiírásra a rendszer összeomlásakor. A hiba kritikus lehet, ha a blokk egy i-csomópont, vagy könyvtári bejegyzés, vagy szabadlista-elem.

Unix – fsck
Windows – checkdisk/scandisk

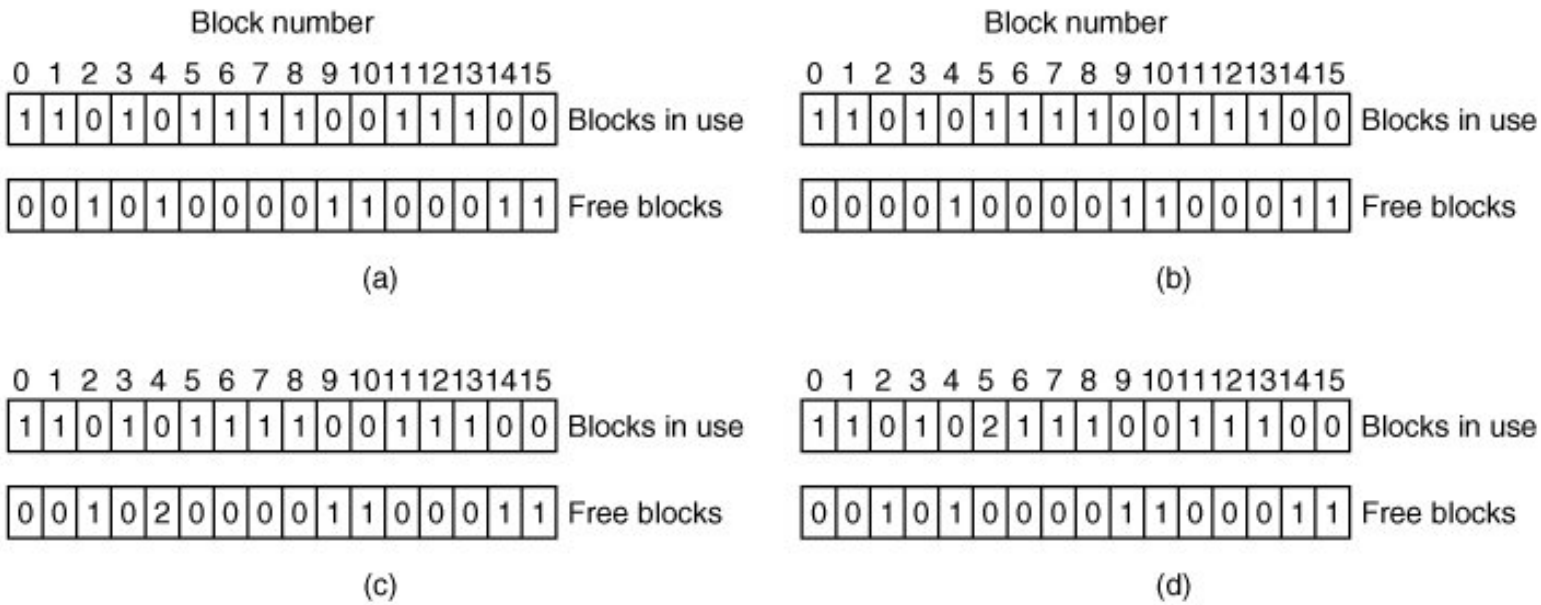
Konzisztencia-ellenőrzések lehetnek

- Heurisztikán alapuló: túl sok fájl egy könyvtárban, gyanús jogosultságok, illegális i-node számok, stb...
- Blokk
- Fájl

Blokk-konzisztencia ellenőrzés

Két táblázatot épít: 1) Hány file-ban fordul elő a blokk az i-csomópontok alapján 2) Hányszor fordul elő a blokk a szabad listában.

(a) Minden blokk vagy az egyik vagy a másik táblázatban fordul elő
(b) Hiányzó blokk
(c) Duplikált szabad blokk
(d) Duplikált adatblokk



Fájl-konzisztencia ellenőrzés

Gyökérkönyvtártól indulva rekurzívan bejárjuk a fájlrendszert, minden file esetén növelünk egy fájlhoz tartozó számlálót.
Összevetjük az i-csomópontban tárolt láncszámmal.

- Ha az i-csomópontban tárolt érték nagyobb: Az i-csomópont nem törlődne az utolsó fájl törlésekor sem. Tárhely kapacitása csökken. Javítás: Az i-csomópont számlálóját a helyes értékre kell állítani.
- Ha az i-csomópontban tárolt érték kisebb: Két könyvtári bejegyzés ugyanahhoz a file-hoz van kapcsolva. Akármelyik bejegyzést törölve az i-csomópont számlálója 0-vá válna, és törölné az adatokat. Javítás: i-node beli értéket javítjuk.

Vége