Fazekas Gábor

# Operációs rendszerek

**mobiDIÁK könyvtár**

Fazekas Gábor

# Operációs rendszerek

**mobiDIÁK könyvtár**

**SOROZATSZERKESZTÔ**

**Fazekas István**

Fazekas Gábor

egyetemi docens
Debreceni Egyetem

# Operációs rendszerek

Oktatási segédanyag

Elsô kiadás

**mobiDIÁK könyvtár**
Debreceni Egyetem

Lektor


mobiDIÁK könyvtár
Debreceni Egyetem
Informatikai Intézet
4010 Debrecen, Pf. 12.
Hungary
http://mobidiak.inf.unideb.hu/

 A mû „A mobiDIÁK önszervezô mobil portál” (IKTA, OMFB-00373/2003) és a „GNU Iterátor, a legújabb generációs portál szoftver” (ITEM, 50/2003) projektek keretében készült.

# Bevezetés

- Mi az operációs rendszer?
- Korai rendszerek.
- A kötegelt  feldolgozás egyszerû rendszerei. (Simple Batch)
- A kötegelt  feldolgozás multiprogramozott rendszerei. (Multiprogramming Batched Systems)
- Idôosztásos (time-sharing) rendszerek.
- Személyi számítógépes rendszerek.
- Párhuzamos rendszerek.
- Elosztott rendszerek.
- Valós idejû rendszerek.

- Mi az operációs rendszer?

  *Operációs rendszer:* egy program(rendszer), amely közvetítô szerepet játszik a számítógép felhasználója és a számítógép hardver között.

  *Operációs rendszer célok:*

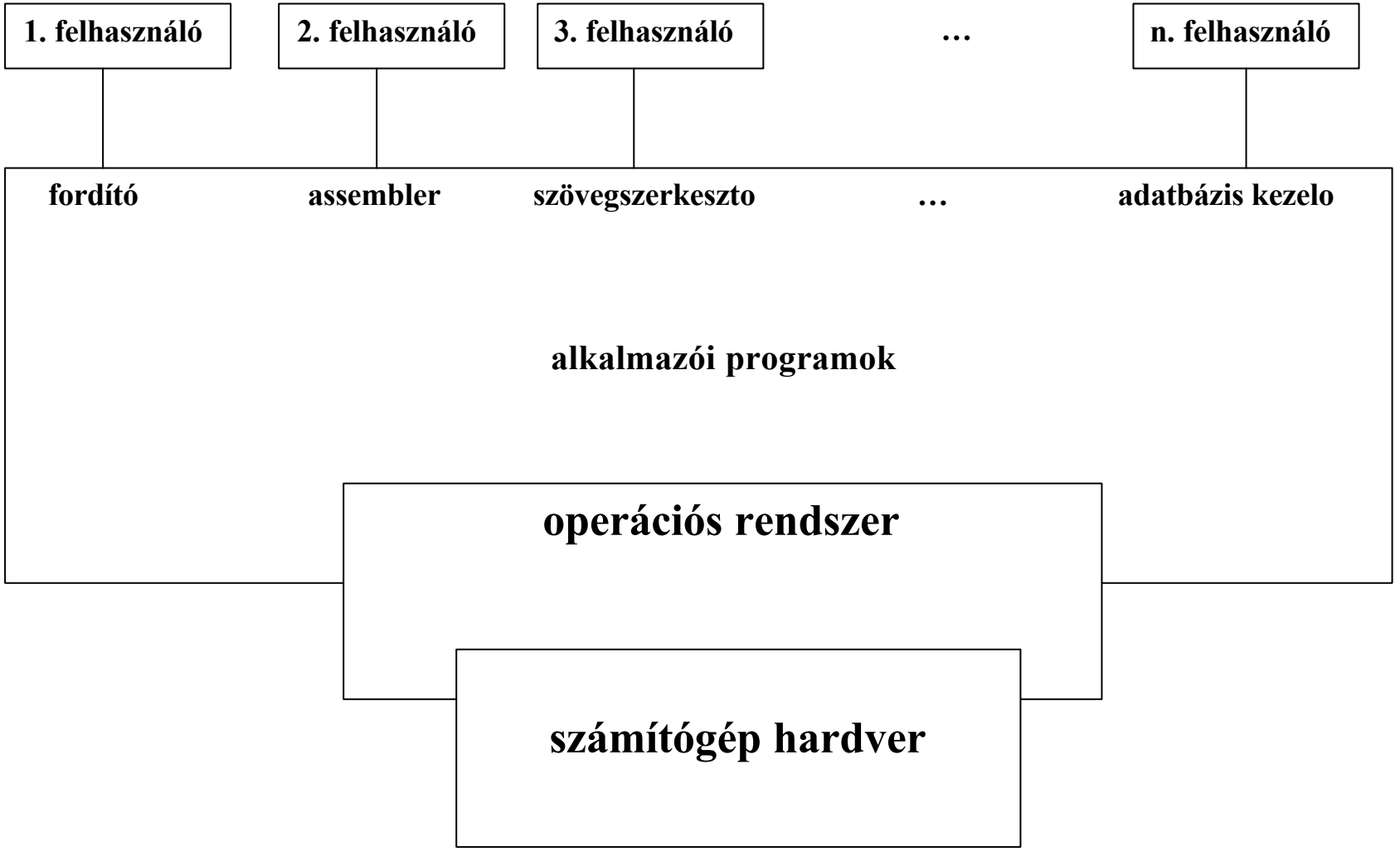
  - Felhasználói programok végrehajtása, a felhasználói feladat-megoldás *megkönnyítés*e.
  - A számítógép rendszer használatának *kényelmesebb*é tétele.
  - A számítógép hardver kihasználásának *hatékonyabb*á tétele.

Megjegyzés: az operációs rendszer a felhasználónak *"overhead".*

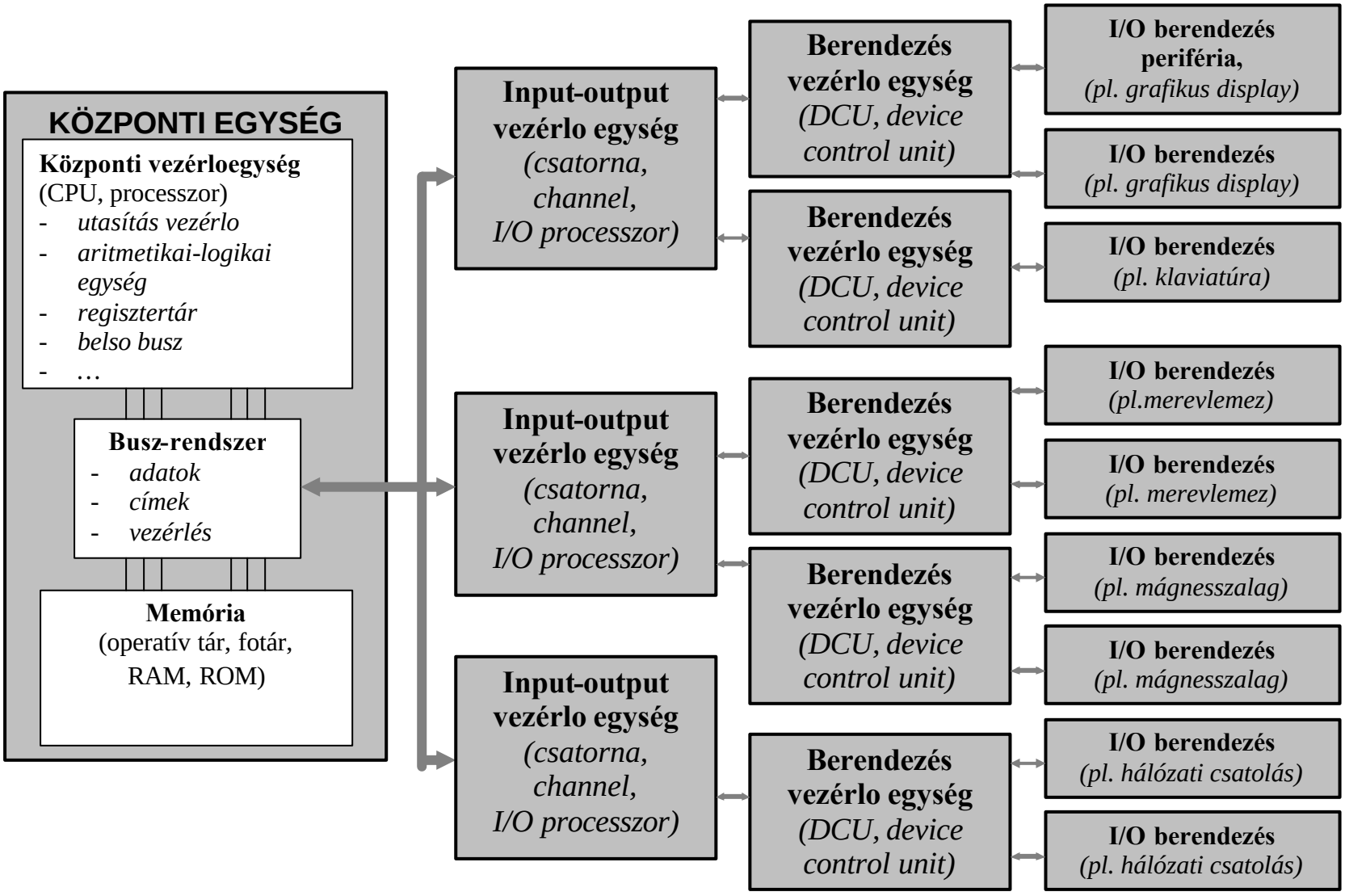
# Számítógép rendszerek komponensei (séma)

1. **Hardver** – az alapvetô számítási erôforrásokat nyújtja (CPU, operatív memória, I/O berendezések).
2. **Operációs rendszer** – koordinálja és vezérli a hardver erôforrások különbözô felhasználók különbözô alkalmazói programjai által történô használatát.
3. **Alkalmazói programok** – definiálják azt a módot, ahogyan az egyes rendszer-erôforrásokat a felhasználók számítási problémáinak megoldásához föl kell használni (fordítók, adatbázis kezelôk, videó játékok, ügyviteli programok).
4. **Felhasználók** (emberek, gépek, más számítógépek).

# Számítógép rendszerek komponensei (séma)



# A számítógép funkcionális felépítése



# Operációs rendszer definíciók

… nézetfüggô :

- ***Erôforrás allokáló***/kiosztó – menedzseli és kiosztja a hardver erôforrásokat
- ***Felügyelô program*** – felügyeli a felhasználói programok végrehajtását, az I/O berendezések mûködését.
- ***Kernel (mag)*** – az egyetlen program, amelyik "állandóan fut" (minden más program alkalmazói program).

- Korai rendszerek – "pucér" gép (1950-es évek eleje)
  - Szerkezeti jellemzôk
    - a nagyméretû gépet a *konzolról* irányítják,
    - egy felhasználós rendszer,
    - a programozó egyben operátor is,
    - lyukszalagos és/vagy lyukkártyás adatbevitel és kivitel.
  - Korai szoftver
    - assemblerek,
    - betöltôk (loaderek),
    - kapcsolat szerkesztôk (linkage editor),
    - közös szubrutin-könyvtárak,
    - fordítók (compiler-ek),
    - I/O berendezés kezelô rutinok (device driver-ek).
  - Biztonság
  - Drága erôforrások rossz hatékonyságú kihasználása
    - alacsony CPU kihasználtság,
    - jelentôs mennyiségû "beállítási idô" (setup time).

- A kötegelt feldolgozás rendszerei (Simple Batch) I.
  - Vegyünk fel egy (professzionális) operátort.
  - Felhasználó ? operátor.
  - Adjunk a rendszerhez egy kártyaolvasót.
  - Redukáljuk a *beállítási idôt* a (hasonló) munkák (job) kötegelésével.
  - Automatikus soros munka végrehajtás (job sequencing): a vezérlés egyik jobról (a job vége után) automatikusan kerül át a következôre. (Az elsô elemi oprációs rendszer megjelenése).
  - Rezidens monitor (felügyelôprogram) mûködési elve:
    - kezdetben a vezérlés a monitornál van,
    - a vezérlés átadódik a job-nak,
    - ha a job befejezôdött a job vissza kerül a monitorhoz.

# • A kötegelt feldolgozás rendszerei (Simple Batch) II.

Egy tipikus job szerkezete és a számítógépes problémamegoldás folyamata, job lépés (jobstep).

## Problémák:

1. Hogyan szerezhet a monitor tudomást az adott job természetérôl (pl. FORTRAN vagy ASSEMBLY), vagy melyik programot kell végrehajtani?
2. Hogyan tudja a monitor megkülönböztetni
   - egyik job-ot a másiktól?
   - az adatot a programtól?

**Megoldás:** Vezérlô kártyák, pozicionálás

- Speciális kártyák, amelyek megmondják a monitornak, mely programot kell futtatni ($JOB, $FTN, $RUN, $DATA, $END)
- Speciális karakterek különböztetik meg az adat és program kártyákat. (//, $, 7-2 lyukasztás)

- A kötegelt feldolgozás rendszerei (Simple Batch) III.
  - A rezidens monitor funkcionális részei
    - *Vezérlô kártya interpreter* – felelôs a vezérlôkártyák beolvasásáért és értelmezéséért.
    - *Betöltô (loader)* – háttértárból betölti az egyes rendszer és felhasználói programokat az operatív memóriába.
    - *Készülék meghajtó programok* (device drivers) – ismerik a rendszer az egyes I/O berendezéseinek tulajdonságait és mûködtetésük logikáját.
  - Elôny: csökken a beállítási idô (setup time)!

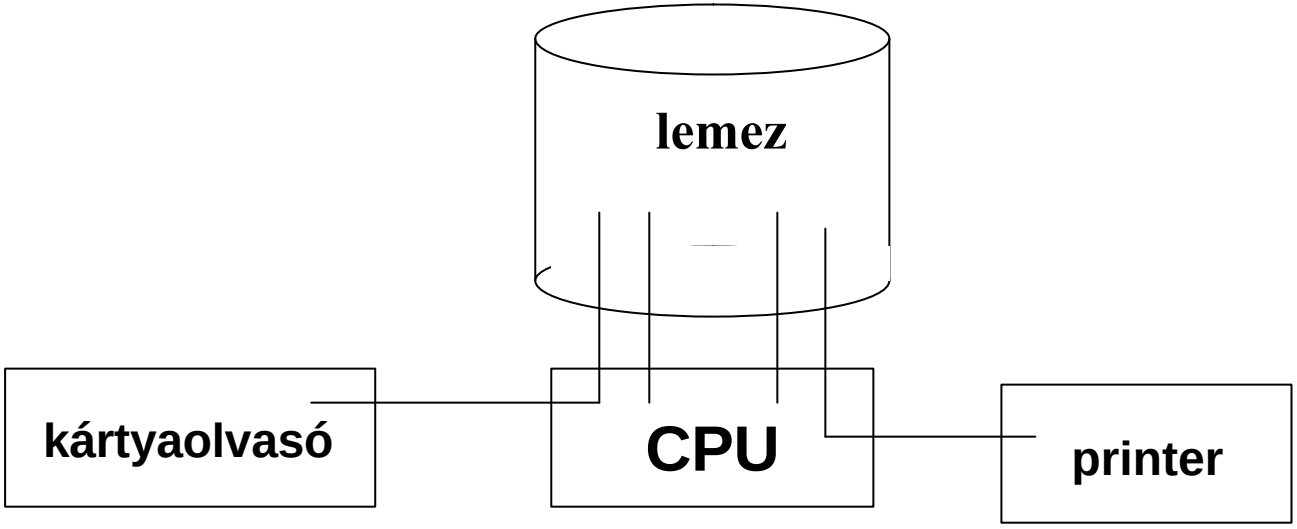
Probléma: Alacsony teljesítmény – mivel az I/O és a CPU mûveletek nem fedhetik át egymást (párhuzamosság!) és a kártyaolvasó nagyon lassú!

Megoldás: Off-line elô- és utófeldolgozás – a jobokat egy másik gép segítségével szalagra másoljuk, ill. az eredményeket szalagra írjuk, majd egy másik gép nyomtatja ki. (*Absztrakt periféria fogalom igénye megfogalmazódik!)*

Simultaneous Peripheral Operation On-Line (SPOOL): IBM704, 1960…

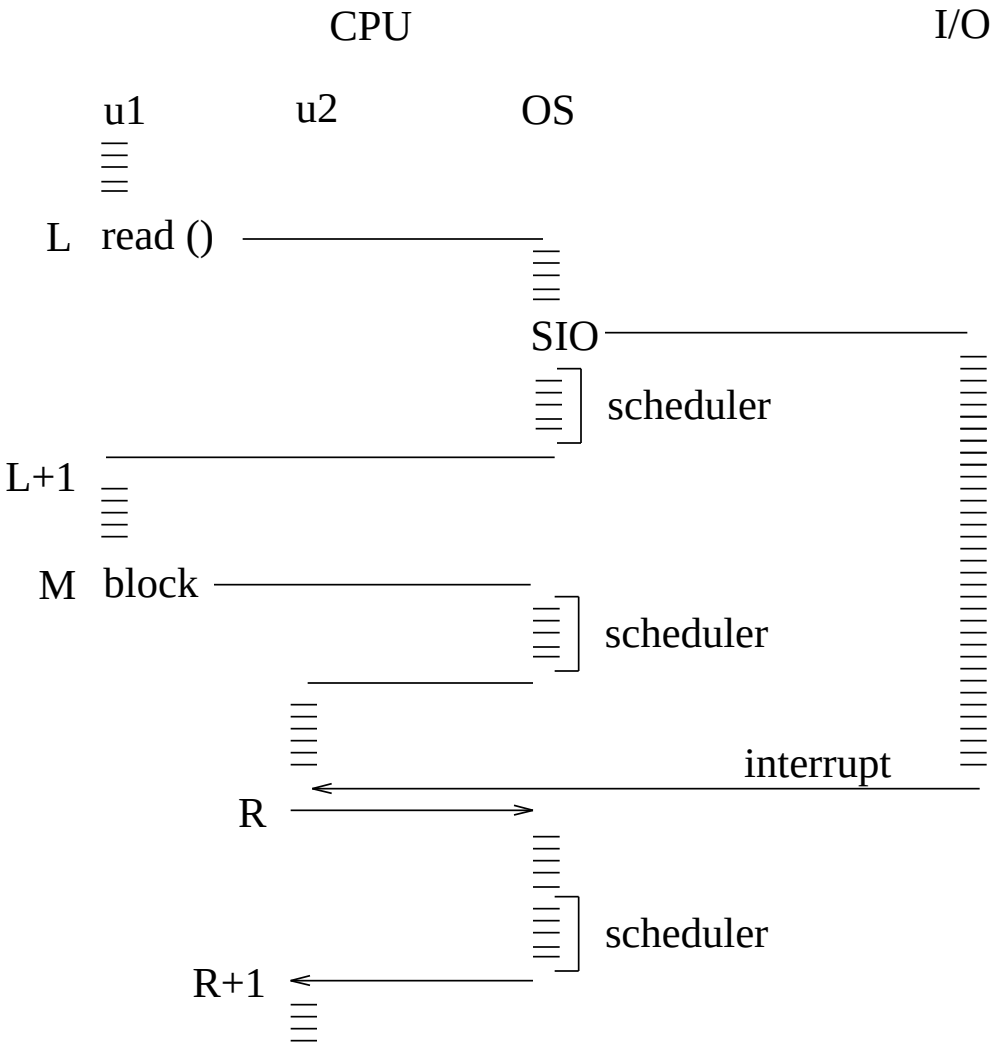
# A kötegelt feldolgozás rendszerei (Simple Batch) IV.

Még jobb megoldás: **Spooling**!




- Mialatt egy job végrehajtódik, az operációs rendszer:
  - beolvassa a következô jobot a kártyaolvasóról a lemezre (job queue)
  - egy elôzô job által nyomtatni szánt adatokat lemezrôl printerre továbbítja
- Job pool – olyan adatszerkezet, amelynek segítségével az operációs rendszer kiválaszthatja a következô job-ot, CPU kihasználtság növelése.

- A kötegelt feldolgozás multiprogramozott rendszerei. (Multiprogramming Batched Systems)
  - Alapelv: néhány job(step) futtatható kódja állandóan az operatív memóriában helyezkedik el és készen áll arra, hogy “utasításokkal lássa el” a CPU-t.
  - Vigyázat! Nem “párhuzamosan futó” programokról van szó!
  - Az operációs rendszer valamilyen *stratégia* szerint adja oda a CPU-t a futásra kész programoknak.


CPU
I/O
u1
u2
OS
L
read ()
SIO
scheduler
L+1
M
block
scheduler
interrupt
R
scheduler
R+1

- A multiprogramozás által az operációs rendszerekkel szemben támasztott követelmények
- Az I/O-nak az operációs rendszer részérôl történô teljes körû felügyelete. (adatvédelem!)
    - Az I/O-t az operációs rendszer nem egyszerûen támogatja, hanem végrehajtásához elkerülhetetlen.
    - Hardver feltételek! (kernel/supervisor mode, privileged operations)
- Memória gazdálkodás – a rendszernek fel kell osztania a memóriát a futó jobok között.
    - Hardver feltételek! (kernel/supervisor mode, privileged operations, segmentation)
- CPU ütemezés – a rendszernek választani kell tudni a futásra kész jobok között.
- Készülékhozzárendelés
    - Nem “jut” minden jobnak, printer, lemez, stb.
- Idôosztásos (time-sharing) rendszerek – interaktivitás

- A kötegelt rendszerek hátránya: nincs interaktivitás!
- TS esetén a CPU váltakozva áll olyan joboknak a rendelkezésére, amelyek a memóriában, vagy lemezen találhatók. (Természetesen a CPU-t csak olyan job kaphatja meg, amely éppen a memóriában van.)
- Egy job a lemezrôl a memóriába, ill. a memóriából a lemezre betölthetô/kimenthetô az ütemezési stratégiának (idôosztás!) megfelelôen. Új fogalom: folyam (process)!
- A rendszer és a felhasználó között on-line kommunikációt tételezünk fel; ha az operációs rendszer befejezi egy parancs végrehajtását, a következô "vezérlô utasítás"-t nem a kártyaolvasóról, hanem a felhasználó klaviatúrájáról várja.
- Egy – adatokat és utasításkódokat tároló – on-line fájl-rendszer kell, hogy a felhasználók rendelkezésére álljon.

# • Személyi számítógépes rendszerek.

- Személyi számítógépek – a teljes számítógép rendszer egy egyszerû felhasználónak kizárólagos rendelkezésére áll.
- Tipikus konfigurációjú I/O berendezések – klaviatúra, egér, képernyô kijelzô, kis teljesítményû nyomtató.
- Elôtérben a felhasználó (személy) kényelme és felelôssége.
- Sokszor adaptál – eredetileg nagygépes operációs rendszerekre kidolgozott információ technológiai megoldásokat. (migráció!)
  - példa: MULTICS (MIT,1965-70) $\Rightarrow$ UNIX (Bell Labs, 1970)
- A felhasználó személy sokszor a számítógép kizárólagos tulajdonosa, felhasználója, és így nincs szüksége fejlett CPU kiszolgáló és adatvédô szolgáltatásokra.

- Párhuzamos rendszerek – multiprocesszoros rendszerek több mint egy – szoros kommunikációs kapcsolatban levô – CPU-val

- Szorosan kapcsolt/csatolt rendszerek – a processzorok közösen használják a memóriát és a rendszer órát. A kommunikáció a közös memória segítségével történik.
- Párhuzamos rendszerek elônyei:
  - Megnövelt átbocsátó képesség,
  - Gazdaságosság,
  - Növekvô megbízhatóság,
    - Redundancia,
    - Graceful degradation,
    - Fail-soft rendszerek.

# Párhuzamos rendszerek

- Szimmetrikus multiprocesszálás
  - Minden egyes processzor az operációs rendszer azonos változatát (másolatát) futtatja. Ezek egymással szükség szerint kommunikálhatnak.
  - Sok processzus futhat egyszerre teljesítménycsökkenés nélkül.
    - I/O problémák, ütemezés
- Aszimmetrikus multiprocesszálás (master-slave modell)
  - Minden egyes processzor a hozzárendelt specifikus feladatot (task) oldja meg. A feladatot a mester határozza meg! Ezek a taskok egymással szükség szerint kommunikálhatnak.
  - Nagyon nagy rendszerekben elterjedtebb megoldás.
    - RJE (remote job entry), front-end processzorok.

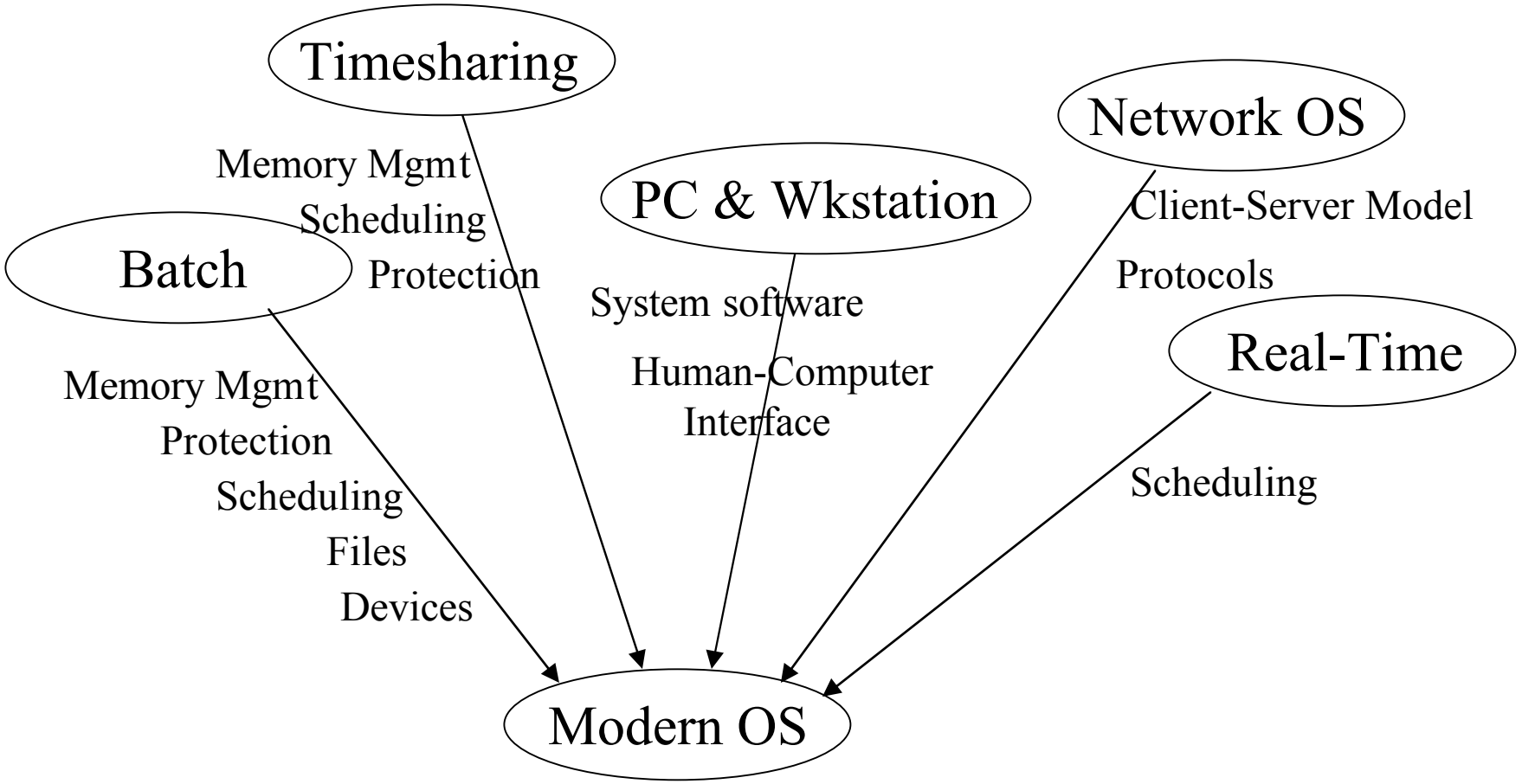
- Elosztott rendszerek – a számításokat több processzor között osztják meg.

- *Lazán kapcsolt/csatolt rendszerek* – a processzorok saját lokális memóriát és rendszer órát használnak. A kommunikáció nagy kapacitású adatvonalak, vagy telefon vonalak segítségével történik.
  - (Pl. speciális LAN; új fogalmak: site, node)
- Elosztott rendszerek elônyei:
  - Erôforrás megosztás (printer, stb.),
  - Számítási teljesítmény növelés,
    - túlterhelés védelem (load sharing),
  - Növekvô megbízhatóság,
  - Kommunikáció (e-Mail, stb).

# • Valós idejû (real-time) rendszerek

- Gyakran úgy jelenik meg, mint valamilyen dedikált alkalmazás (pl. tudományos kísérlet támogatása, orvosi képfeldolgozás, ipari kontroll, kijelzô rendszerek) irányító-felügyelô rendszere.
- A “kiszolgálás” azonnal megkezdôdik! Jól definiált, rögzített idejû korlátozások.
- “Hard” ("merev" valós idejû) rendszerek.
  - A másodlagos tár korlátozott, vagy teljesen hiányzik; az adatokat az operatív memóriában (RAM), vagy akár ROM-ban tárolják.
  - Konfliktus az idôosztásos rendszerekkel.
- “Szoft” ("puha" valós idejû) rendszerek.
  - Korlátozott szolgáltató programok az ipari kontroll, a robotika területén.
  - A fejlett operációs rendszer szolgáltatásokat igénylô alkalmazásoknál (Multimédia, VR) igen hasznosak.

# Evolution of Modern OS



# 2. A számítógép rendszer strukturális jellemzôi

- A számítógépes rendszer mûködése
- I/O struktúra
- Tár struktúra és hierarchia
- Hardver védelem
- Általános rendszer architektúra

- A számítógépes rendszer mûködése
  - Az I/O berendezések és a CPU szimultán képes mûködni.
  - Minden berendezés vezérlô egység egy meghatározott berendezés típus mûködtetéséért felelôs,
  - Minden berendezés vezérlô egységnek saját, lokális puffere van.
  - A CPU oldaláról az I/O a fôtár és e lokális pufferek közötti adatmozgatást jelenti.
  - A berendezés vezérlô egységek, I/O processzorok egy-egy megszakítás generálásával jelezhetik a CPU-nak az I/O mûvelet befejezését. (DMA)

# • A megszakításkezelés fogalmai

- A *megszakítás (interrupt)* átadja a vezérlést a *megszakítás feldolgozó rutinnak*. Ez általában a *megszakítási vektor* segítségével történik, amelynek megfelelô elemei tartalmazzák a *megszakítási osztályokhoz* tartozó feldolgozó rutin elsô végrehajtandó utasításának címét.
- A megszakítási rendszernek tárolnia kell a megszakított utasítás címét.
- Egy megszakítás feldolgozása idején jelentkezô további megszakítások letilthatók (mask out, disable), hogy el ne "vesszenek" (lost interrupt).
- A megszakítási jel forrását tekintve egy megszakítás lehet külsô (pl. I/O, Timer, Hardver, … ), vagy belsô (szoftver megszakítás, angolul: *trap*).
- Az operációs rendszer (megszakítás feldolgozó rutin) közvetlen feladatai:
    - a CPU állapotának megôrzése (a hardver ehhez minimális kezdeti támogatást nyújt);
    - a megszakítás okának, körülményeinek részletesebb elemzése;
    - "maszkolás";
    - a megszakított programhoz történô "visszatérés" megszervezése.

# • I/O struktúra

- Az I/O folyamat elindult, a felhasználói program a vezérlést csak az I/O befejezése után kapja vissza.
  - "wait" utasítások a következô megszakításig;
  - "wait loop" az adat megérkezéséig (hand-shaking, ready bit!);
  - csak egyetlen egy I/O lehet egyszerre folyamatban.
- Az I/O folyamat elindult, a (felhasználói) program a vezérlést azonnal vissza-kapja tekintetet nélkül az I/O befejezôdésére.
  - rendszer hívás (system call); az I/O-t a felügyelô program indítja;
  - készülék - státusz táblázat (device state);
  - a felügyelô program kezeli - gazdálkodás, ütemezés.
- DMA (Direct Memory Access)
  - egy megszakítás blokkonként (nem bájtonként!)

# • Tár struktúra és hierarchia

- Fôtár, operatív tár – az egyetlen, amit a CPU közvetlenül képes elérni.
- Másodlagos tár – a fôtár *kiterjesztése*, nagy, nem törlôdô tárolás lehetôsége.
  - Swap, paging.
- Mágnes lemezek
  - sáv, szektor, cilinder fogalma
- A tár rendszerek hierarchikus struktúrába rendezhetôk
  - sebesség,
  - ár (költség),
  - a tárolt adatok tartóssága

  alapján.
- Caching, cache szervezés:
  - belsô/külsô cache, virtuális lemezek.

# • Hardver védelem

- Duál - módú mûködés
  - felhasználó mód – rendszer (supervisor, monitor) mód,
  - módus bit,
  - privilegizált mûveletek.
- I/O védelem
  - minden I/O utasítás privilegizált
  - meg kell(ett) oldani, hogy felhasználói programok ne kerülhessenek monitor módba
- Memória védelem
  - szegmensek, tárvédelmi kulcs
  - monitor módban korlátlan a tárhozzáférés
- CPU védelem
  - Timer megszakítások szerepe

# • Általános rendszer architektúra

- Az I/O utasítások privilegizáltak. Hogyan hajthat végre egy felhasználói program I/O mûveletet?
- Rendszer hívás - speciális megszakítás az operációs rendszer szolgáltatásainak igénybe vételére.
  - INT gépi utasítás,
  - paraméter átadás, funkció kódok,
  - paraméter ellenôrzés ,
  - végrehajtás,
  - vezérlés visszaadása.

# 3. Operációs rendszer struktúrák

- Az operációs rendszer komponensei
- Operációs rendszer szolgáltatások
- Rendszer-hívások
- Rendszerprogramok
- Rendszer-szerkezet
- Virtuális gépek
- Rendszer tervezés és implementáció
- Rendszer-generálás

- Az operációs rendszer komponensei (idealizált)
  - Folyamat kezelés (Process management)
  - Memória kezelés (gazdálkodás)
  - Másodlagos tár kezelés
  - I/O rendszer kezelés
  - Fájl kezelés
  - Védelmi rendszer
  - Hálózat-elérés támogatása
  - Parancs interpreter rendszer

- Folyamat kezelés (Process management)
  - Folyamat (process - processzus) – egy végrehajtás alatt levô program. A folyamatnak bizonyos erôforrásokra (így pl. CPU idô, memória, állományok, I/O berendezések) van szüksége, hogy a feladatát megoldhassa.
  - Az operációs rendszer az alábbi tevékenységekért felel a folyamatok felügyeletével kapcsolatban:
    - Folyamat létrehozása és törlése.
    - Folyamat felfüggesztése és újraindítása.
    - Eszközök biztosítása a
      - folyamatok szinkronizációjához,
      - a folyamatok kommunikációjához.

- Memória (fôtár) kezelés (gazdálkodás)
  - Az operatív memóriát bájtokból (szavakból) álló tömbnek fogjuk tekinteni, amelyet a CPU és az I/O vezérlô megosztva (közösen) használ.
  - Tatalma törlôdik a rendszer kikapcsolásakor és rendszer-hibáknál.
  - Az operációs rendszer a következô dolgokért felelôs a memória kezelést illetôen:
    - Nyilvántartani, hogy az operatív memória melyik részét ki (mi) használja.
    - Eldönteni, melyik folyamatot kell betölteni, ha memória felszabadul.
    - Szükség szerint allokálni és felszabadítani memória területeket a szükségleteknek megfelelôen

- Másodlagos tár kezelés
  - Mivel az operatív tár (elsôdleges tár) törlôdik (és egyébként sem alkalmas arra, hogy minden programot tároljon), a másodlagos tárra szükség van.
  - Merev lemezes tár, a másodlagos tár legelterjedtebb megjelenése.
  - Az operációs rendszer a következô dolgokért felelôs a másodlagos tár kezelést illetôen:
    - Szabad-hely kezelés
    - Tár-hozzárendelés
    - Lemez elosztás (scheduling)

- I/O rendszer kezelés
  - Az I/O rendszer az alábbi részekbôl áll:
    - Puffer (Buffer/Cache) rendszer
    - Általános készülék-meghajtó (device driver) interface
    - Speciális készülékek meghajtó programjai

# • Fájl (állomány) kezelés

- Egy fájl kapcsolódó információ (adatok) együttese, amelyet a létrehozója definiál. Általában program- (különbözô formák), vagy adatfájlokkal dolgozunk.
- Az operációs rendszer a következô dolgokért felelôs a fájl kezelést illetôen:
  - Fájl létrehozása és törlése
  - Könyvtár létrehozása és törlése
  - Fájlokkal és könyvtárakkal történô alap-manipulációhoz nyújtott támogatás.
  - Fájlok “leképezése” a másodlagos tárba.
  - Fájlok mentése valamilyen nemtörlôdô, stabil adathordozóra.

- Védelmi rendszer

  - Védelem általában valamilyen mechanizmusra utal, amelynek révén mind a rendszer-, mind a felhasználói erôforrásoknak a programok, folyamatok, vagy felhasználók által történô elérése felügyelhetô, irányítható.
  - A védelmi mechanimusnak tudnia kell:
    - különbséget tennie authorizált (jogos) és jogtalan használat között,
    - specifikálni az alkalmazandó kontrolt,
    - szolgáltatni a korlátozó eszközöket.

- Hálózat-elérés támogatása (elosztott rendszerek)

  - Egy elosztott rendszer processzorok adat és vezérlô vonallal összekapcsolt együttese, ahol a processzorokhoz nincs közös memóriájuk és órájuk. (lokális memória, óra).
  - Az adat- és vezérlô vonalak egy kommunikációs hálózat részei.
  - Az elosztott rendszer a felhasználóknak különbözô osztott erôforrások elérését teszi lehetôvé.
  - Az erôforrások osztott elérése lehetôvé teszi:
    - a számítások felgyorsítását,
    - a jobb adatelérhetôséget,
    - a nagyobb megbízhatóságot.

- Parancs interpreter (al)rendszer
  - Az operációs rendszernek sok parancsot ún. vezérlô utasítás formájában lehet megadni. Ezek a vezérlô utasítások az alábbi területekhez tartozhatnak:
    - folyamat létrehozás és kezelés
    - I/O kezelés
    - másodlagos tár kezelés
    - operatív tár kezelés
    - fájl rendszer elérés
    - védelem
    - hálózat kezelés
    - ...
- Az operációs rendszernek azt a programját, amelyik a vezérlô utasításokat (be)olvassa és interpretálja a rendszertôl függôen más és más módon nevezhetik:
  - vezérlô kártya interpreter (control-card Job Control, JC)
  - parancs-sor interpreter (command-line)
  - héj (burok, shell) (UNIX)

# • Operációs rendszer szolgáltatások

- Program végrehajtás (program betöltés és futtatás)
- I/O mûveletek (fizikai szint: blokkolás, pufferezés)
- Fájl-rendszer manipuláció (r, w, c, d)
- Kommunikáció – a folyamatok közötti információ csere (ugyanazon, vagy különbözô gépeken) Shared memory – Message passing.
- Hiba detektálás (CPU, memória, I/O készülékek, felhasználói programok, ...)

- Nem közvetlenül a felhasználó támogatását, hanem a hatékonyabb rendszermûködést segítik:
- Erôforrás kiosztás – multiprogramozás, többfelhasználós mûködés
- Accounting – rendszer és felhasználói statisztikák.
- Védelem – minden erôforrás csak az operációs rendszer felügyelete mellett érhetô el.

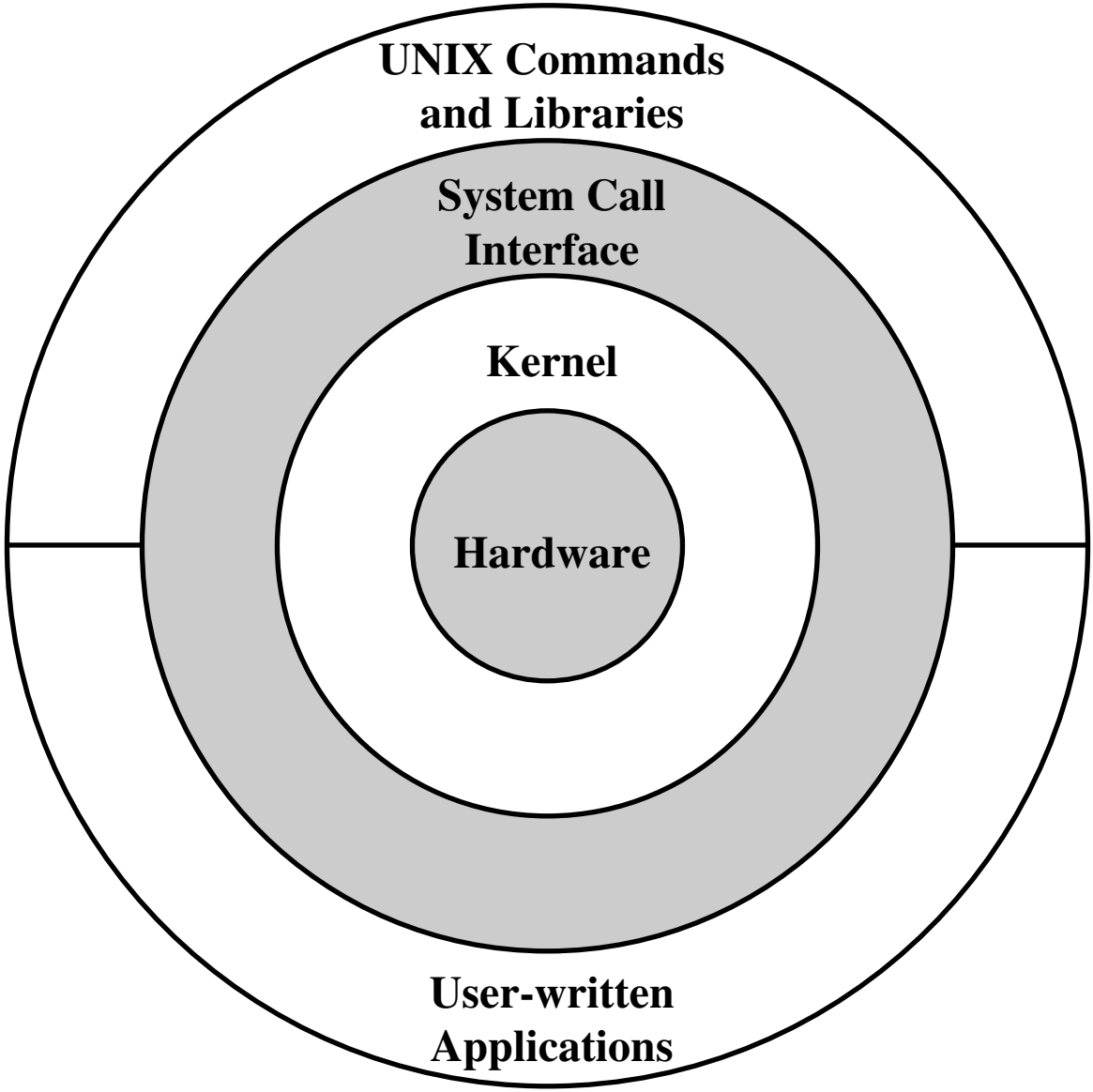
- Rendszer-hívások
  - Ha a futó (felhasználói) program valamilyen rendszerszolgáltatást igényel, ezt rendszerhívás formájában teheti meg.
    - Általában assembly szintû utasításként jelen van az architektúrában. (INT, Tcc, SC, stb.)
    - Magas szintû, rendszerprogramok írására szánt programozási nyelvekbe is beépítették. (C)
  - Paraméterátadás módjai a rendszernek:
    - regiszterben,
    - paraméter-táblázatban,
    - veremben (push- felh. program; pop- op. rendszer) ,
    - a módszerek kombinálása, statusword-ok

# • Rendszerprogramok

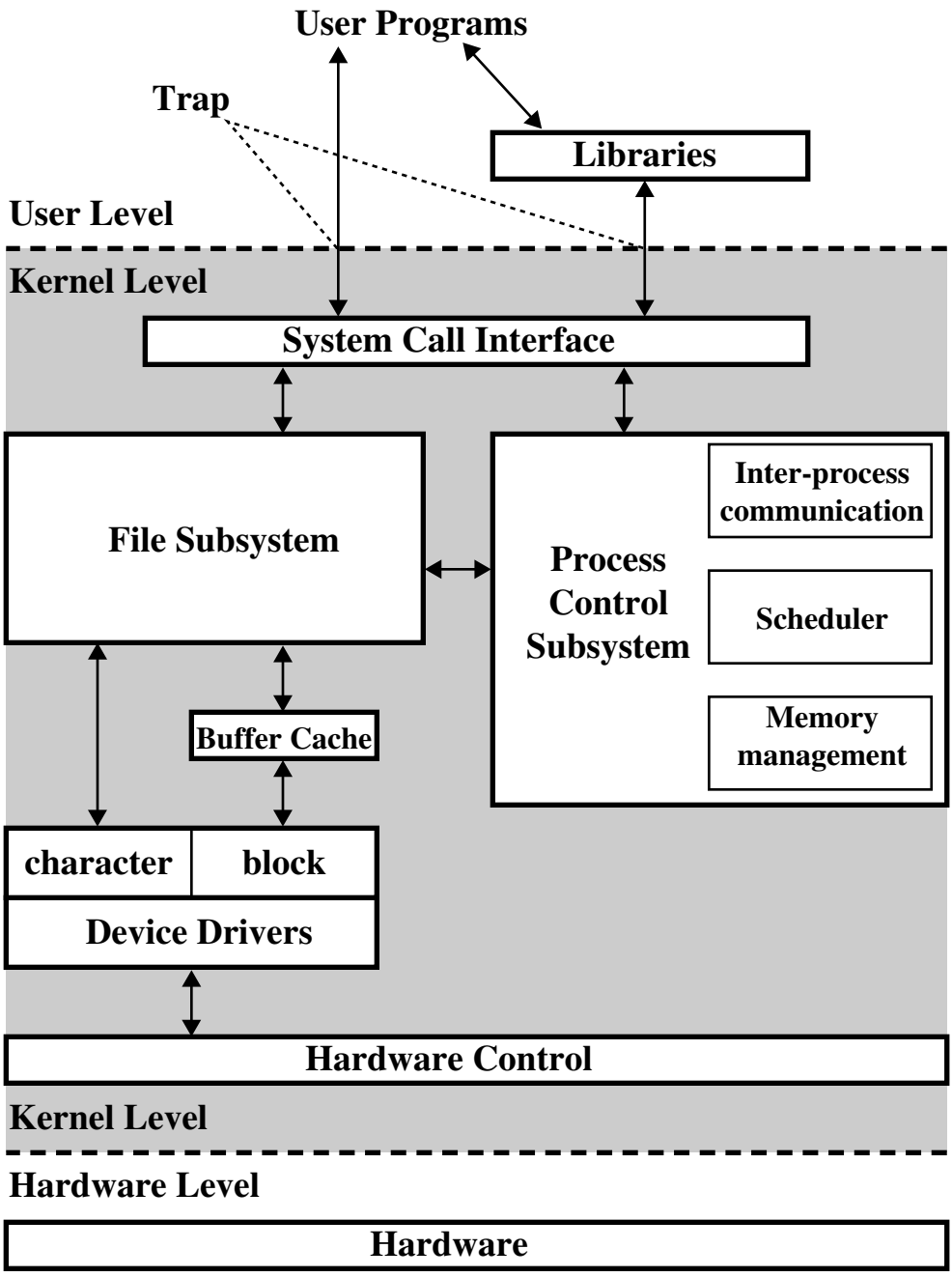
- A rendszerprogramok kényelmes környezetet teremtenek a program-fejlesztéshez és program végrehajtáshoz. Egy lehetséges osztályozásuk:
  - Fájl manipuláció
  - Státus információ
  - Fájl módosítás
  - Programozási nyelv támogatás
  - Program betöltés és végrehajtás
  - Kommunikáció
  - Alkalmazói programok ...
- Felhasználói szemszögbôl nézve (egyes nézetek szerint) az operációs rendszer sokszor a rendszerprogramok együttesével azonosítható.

# • Rendszer-szerkezet

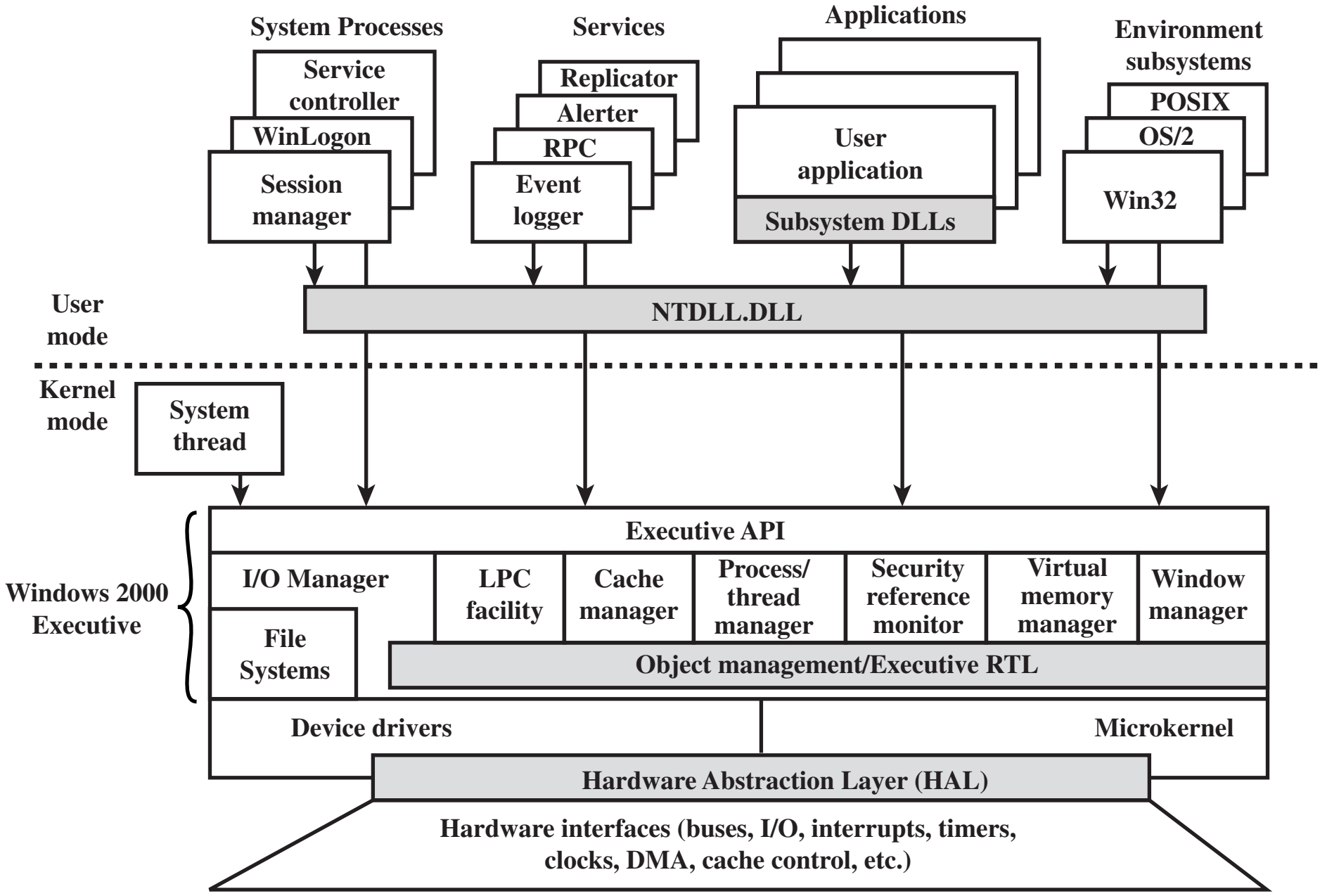
- **MS-DOS** – *a lehetô legtöbb funkcionalitást igyekszik besûríteni a lehetô legkisebb tárba:* (ezért) nincs modularitás. Az MS-DOS -nak van bizonyos szerkezete, de kapcsolatai és funkcionális szintjei nem különböztethetôek meg élesen. (⇒OS/2)
- **UNIX** – a hardver funkcionalitása a korlát; az eredeti UNIX operációs rendszer korlátozott struktúrával rendelkezett:
  - rendszerprogramok,
  - a kernel (mag), amely mindent magában foglal a rendszerhívás interfész alatt és a fizikai hardver fölött: (pl.) fájl rendszer, CPU ütemezés, memória gazdálkodás, más operációs rendszer függvé-nyek, azaz számos függvény egy szinten.




**General UNIX Architecture**




**Traditional UNIX Kernel]**




**Windows 2000 Architecture**

- Rétegelt (layered) megközelítés
  - Az operációs rendszer egymásra épülô szintekre bomlik. Legalsó szinten (layer 0) van a hardver, a legfelsôn (layer N) a felhasználói interfész.
  - Megfelelô modularitással minden szint (layer) kizárólag a nála alacsonyabb szint függvényeit és szolgáltatásait használja.
  - Ilyen absztrakt rétegelt megközelítést elôször *Dijkstra* javasolt a *T.H.E* operációs rendszerben:
    - 5. Szint: felhasználói program
    - 4. Szint: input/output pufferezés
    - 3. Szint: operátor konzol készülék meghajtó
    - 2. Szint: memória menedzsment
    - 1. Szint: CPU ütemezés
    - 0. Szint: hardver

# • Virtuális gépek

- A virtuális gép a rétegelt modell általánosítása: nemcsak a a fizikai hardvert, hanem az operációs rendszer magját is hardvernek tekinti.
- A virtuális gép a rendelkezésre álló hardverrel azonos interfészt nyújt.
- Az operációs rendszer azt az illúziót kelti, mintha minden processzus saját processzort és saját (virtuális) operatív memóriát használna.
- A fizikai gép erôforrásainak megosztásával (többszörös felhasználásával) implementálhatók a virtuális gépek:
  - CPU ütemezés ⇒ saját processzorok illúziója
  - SPOOLING és fájl rendszer ⇒ saját perifériák illúziója
  - Valódi idôosztásos terminál ⇒ a virtuális gép operátori konzolja
- A virtuális gépek elônyei és hátrányai
  - Izoláció ⇒ teljes védelmet nyújt, de kizárja az erôforrások célszerû közös használatát.
  - Jó közeg a rendszerprogramozónak: teljes szolgáltatás a környezet zavarása nélkül.
  - Bonyolult feladat az implementáció ( a valódi gép pontos multiplikálása).

# • Rendszer tervezés és implementáció

## • Rendszer tervezési célok

- **Felhasználói célok**: az operációs rendszer legyen kényelmesen használható, könnyen megtanulható, megbízható, biztonságos, gyors.
- **Rendszer célok**: az operációs rendszer legyen könnyen tervezhetô, implementálható, gondozható; továbbá rugalmas, megbízható, hiba-mentes, hatékony.

## • Mechanizmusok és politikák

- A mechanizmusok meghatározzák, hogy **hogyan** kell valamit csinálni, a politikák pedig, hogy **mit** kell csinálni. (példa: timer megszakítás.)
- A mechanizmus és a politika különválasztása nagyon fontos; maximális rugalmasságot teremt, ha a politika késôbb megváltozik.

## • Rendszer implementálás

- Régen assembly nyelven írták az operációs rendszereket, ma már ez nem igaz; magasszintû nyelven is megírhatók. (pl.: MCP/Borroughs ⇒ Algol; MULTICS ⇒PL/1; UNIX, OS/2, WinNT ⇒ C)
- A magasszintû nyelven írott kód gyorsabban elkészíthetô, kompaktabb, könnyebben áttekinthetô és nyomkövethetô (debug).
- A magasszintû nyelven írt rendszer portábilis, más hardverre könnyen átvihetô. (pl.: Unix, Linux) A szükkeresztmetszeteket meghatározva a gépi kód korrigálható (patch, service pack).

# Rendszer-generálás

- Az operációs rendszereket úgy tervezik, hogy azok mûködôképesek legyenek egy géposztály minden gépén. Ehhez azonban minden egyes számítógép-környezetre az operációs rendszert *konfigurálni* kell. Ezt a mûveletet *rendszergenerálás*nak (⇔installálás?) nevezzük. Program segíti (SYSGEN, setup, install).
- Egy SYSGEN program informálódik a hardver rendszer specifikus konfigurációjáról. (pl. Milyen processzor, aritmetika vehetô alapul; processzorok száma, típusa, stb. ; rendelkezésre álló memória mérete; perifériális berendezések típusai; megszakítási rendszer tulajdonságai).
- Tisztázni kell, hogy a rendszer mely szolgáltatásai tényleg szükségesek: multiprogramozási stratégia, hálózat elérés, stb.
- Booting, bootstrap loader – rendszer betöltés.

# •Rendszer-generálás

## A generált rendszerrôl szerzett adatok (UNIX példa)

```
   G E N E R A L   I N F O R M A T I O N

Host Name is              euklid
Host Address(es) is       131.234.xxx.xxx
Host ID is                80782854
Serial Number is          2155358292
Manufacturer is           Sun (Sun Microsystems)
System Model is           Ultra 1 Model 140
Main Memory is            128 MB
Virtual Memory is         210 MB
ROM Version is            OBP 3.0.4 1995/11/26 17:47
Number of CPUs is         1
CPU Type is               sparcv8plus+vis
App Architecture is       sparc
Kernel Architecture is    sun4u
OS Name is                SunOS
OS Version is             5.6
OS Distribution is        Solaris 2.6 5/98 s297s_hw3smccServer_09 SPARC
Kernel Version is         SunOS Release 5.6 Version Generic_105181-17
                          [UNIX(R) System V Release 4.0]
Boot Time is              Sat Feb 26 18:39:29 2000
```

K E R N E L  I N F O R M A T I O N

| | |
|---|---|
| Maximum number of processes for system is | 1034 |
| Maximum number of processes per user is | 1029 |
| Maximum number of users (for system tables) is | 64 |
| Maximum number of BSD (/dev/ptyXX) pty's is | 48 |
| Maximum number of System V (/dev/pts/*) pty's is | 48 |
| Size of the virtual address cache is | 16384 |
| Size of the callout table is | 112 |
| Size of the inode table is | 4712 |
| Size of the directory name lookup cache is | 4712 |
| Size of the quotas table is | 1674 |
| STREAMS: Maximum number of pushes allowed is | 9 |
| STREAMS: Maximum message size is | 65536 |
| STREAMS: Maximum size of ctl part of message is | 1024 |
| Maximum global priority in sys class is | 6488124 |
| Has UFS driver is | TRUE |
| Has NFS driver is | TRUE |
| Has SD driver is | TRUE |
| Has FD driver is | TRUE |
| Has IPCSHMEM is | TRUE |

```
     S Y S C O N F  I N F O R M A T I O N

Max combined size of argv[] and envp[] is                    1048320
Max processes allowed to any UID is                          1029
Clock ticks per second is                                    100
Max simultaneous groups per user is                          16
Max open files per process is                                64
System memory page size is                                   8192
Job control supported is                                     TRUE
Savid ids (seteuid()) supported is                           TRUE
Version of POSIX.1 standard supported is                     199506
Version of the X/Open standard supported is                  3
Max log name is                                              8
Max password length is                                       8
Number of processors (CPUs) configured is                    1
Number of processors (CPUs) online is                        1
Total number of pages of physical memory is                  16384
Number of pages of physical memory not currently in use is   7823
Max number of I/O operations in single list I/O call is      256
Max number of timer expiration overruns is                   2147483647
Max number of open message queue descriptors per process is  32
Max number of message priorities supported is                32
Max number of realtime signals is                            8
Max number of semaphores per process is                      2147483647
Max value a semaphore may have is                            2147483647
Max number of queued signals per process is                  32
Max number of timers per process is                          32
Supports asyncronous I/O is                                  TRUE
Supports File Synchronization is                             TRUE
Supports memory mapped files is                              TRUE
```

```
    D E V I C E   I N F O R M A T I O N

SUNW,Ultra-1 is a "Sun 501-2836"
openprom1 is a "Sun Open Boot PROM" device
options0 is a "PROM Settings"
aliases1 is a "PROM Device Aliases"
sbus0 is a "Sun SBus" system bus
audiocs0 is a "Crystal Semiconductor 4231" audio device
auxio is a "Auxiliary I/O"
flashprom1 is a "Sun Flash PROM"
eeprom1 is a "EEPROM" device
zs0 is a "Zilog 8530" serial device
zs1 is a "Zilog 8530" serial device
espdma is a "SCSI DMA" pseudo device
esp0 is a "Generic SCSI" SCSI controller
c0t0d0 (sd0) is a "SUN2.1G" 2.0 GB disk drive
ledma is a "LANCE Ethernet DMA" pseudo device
le0 is a "AMD Lance Am7990" 10-Mb Ethernet network interface
bpp is a "Sun Bidirectional Parallel Port" parallel device
cpu0 is a "Sun UltraSPARC" 143 MHz CPU
```

# 4. Folyamatok

- A folyamat (processzus) fogalma
- Folyamat ütemezés (scheduling)
- Folyamatokon végzett "mûveletek"
- Folyamatok együttmûködése, kooperációja
- Szálak (thread)
- Folyamatok közötti kommunikáció

# • A folyamat (processzus) fogalma

- A folyamat (processzus): végrehajtás alatt álló program.

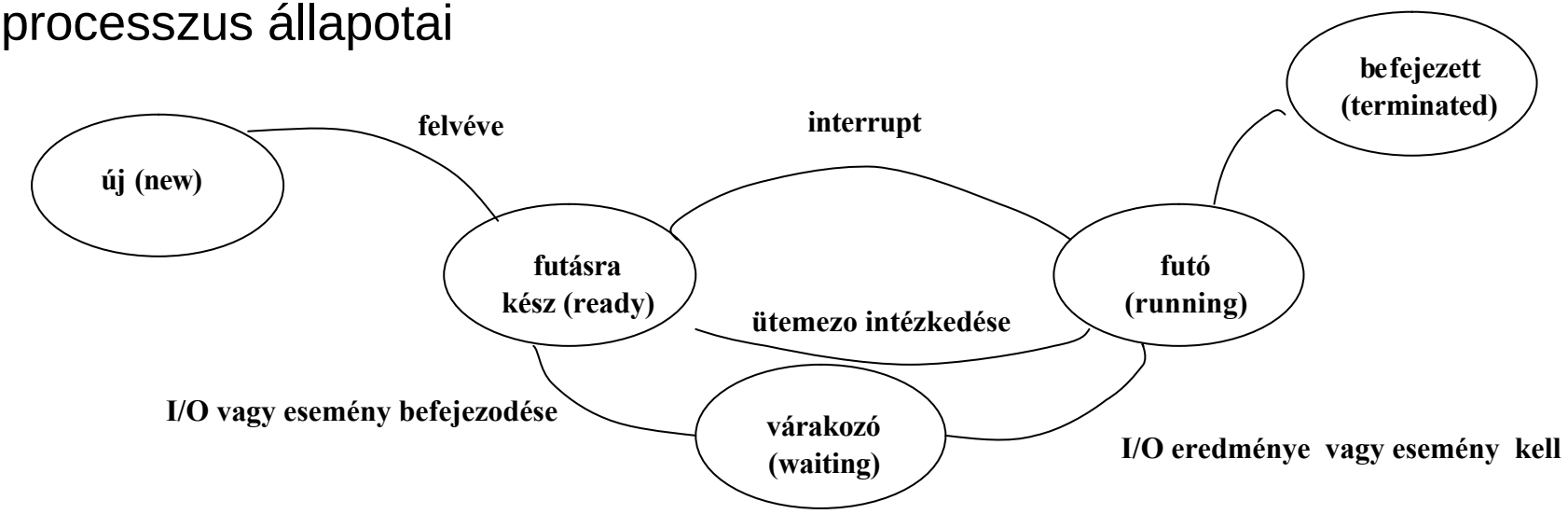
**Betöltheto program (jellemzoi)**

- passzív
- lemezen tárolt
- betöltési cím
- belépési pont
- …

**Processzus (jellemzoi)**

- aktív
- a memóriában van/volt
- program címszámláló értéke
- regiszterek értéke
- lokális/globális változók értéke
- verem állapota
- … core dump …

- A processzus állapotai



- Processzus - vezérlô blokk (Process Control Block – PCB)
  - processzus állapot (process state) : new, ready, running, waiting, halted, sleeping, …
  - Program címszámláló (PC) értéke
  - CPU regiszterek tartalma
  - memória foglalási adatok
  - account/user adatok
  - I/O státusz információ (a folyamathoz rendelt I/O erôforrások, állományok listája)
- Processzus állapot információk (UNIX: ps, top; NT: Task manager)

TOP:

```
last pid:  9099;  load averages:  0.00,  0.00,  0.01                 11:33:35
29 processes:  28 sleeping, 1 on cpu
CPU states: 99.6% idle,  0.2% user,  0.2% kernel,  0.0% iowait,  0.0% swap
Memory: 128M real, 58M free, 120M swap free

  PID USERNAME THR PRI NICE  SIZE   RES STATE   TIME    CPU COMMAND
 9099 fazekas    1  58    0 2608K 1952K cpu     0:00  0.42% top
  226 root       1  58    0  912K  672K sleep   0:00  0.02% utmpd
 4081 root       1  58    0 1904K 1648K sleep   0:00  0.02% sshd
 4084 fazekas    1  48    0 2600K 2160K sleep   0:00  0.00% tcsh
  583 root       1  45    0 1760K 1024K sleep   3:30  0.00% sshd
    1 root       1  58    0  680K  312K sleep   0:02  0.00% init
  166 root       5  58    0 2720K 2144K sleep   0:01  0.00% automountd
 8895 root       1  59    0 9208K   10M sleep   0:01  0.00% Xsun
  239 root       5  22    0 2296K 1928K sleep   0:00  0.00% vold
  212 root       3  22    0 1168K  848K sleep   0:00  0.00% powerd
  170 root       8  35    0 3608K 1936K sleep   0:00  0.00% syslogd
  592 root       1  38    0 1560K 1120K sleep   0:00  0.00% ttymon
```

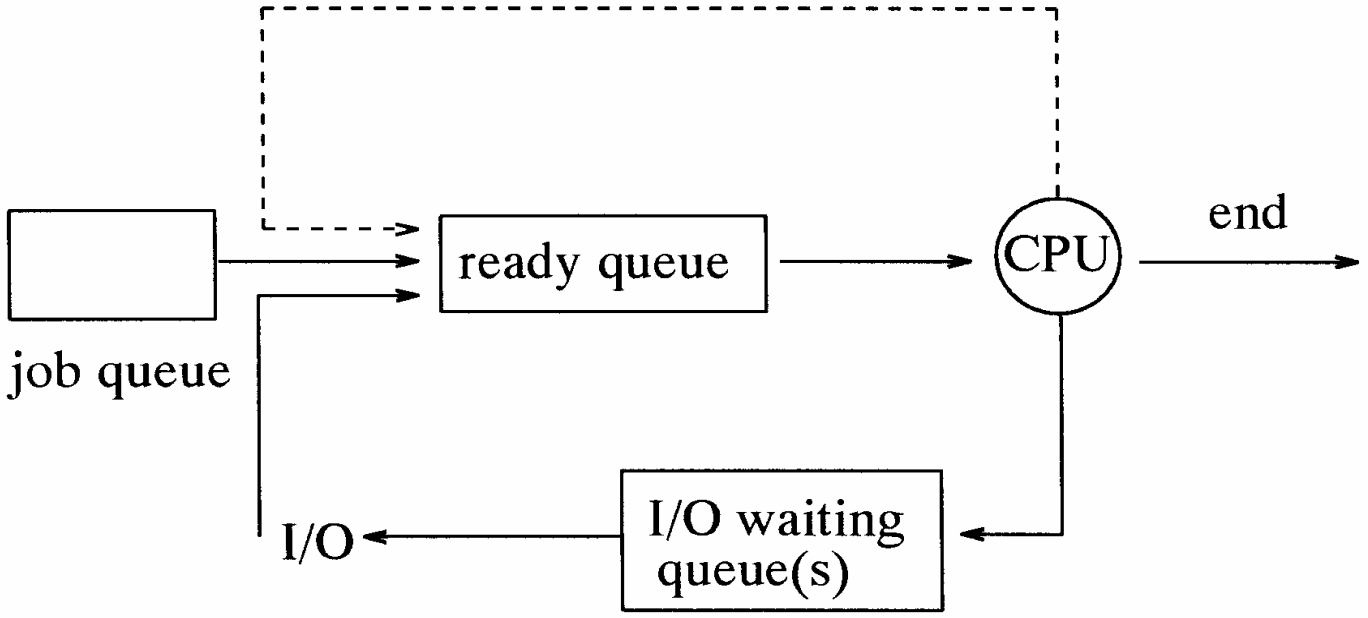
PS:

```
USER       PID %CPU %MEM   SZ  RSS TT       S    START  TIME COMMAND
```

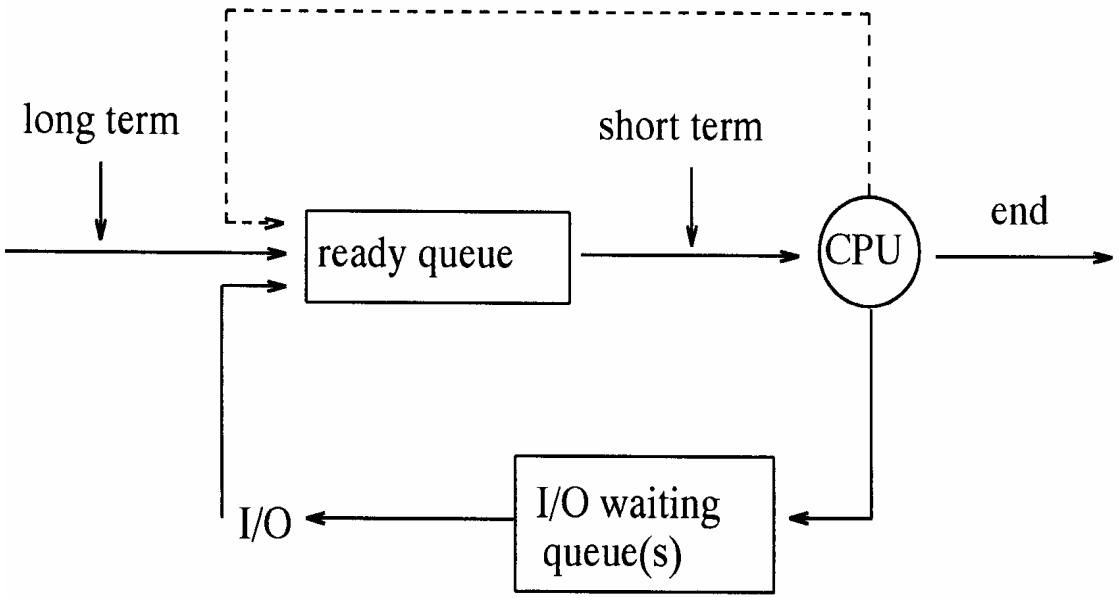
```
fazekasg   513  1.1  2.012192 9848 ?        S 15:00:12  0:38 /usr/openwin/bin/Xsun :0 -nobanner -auth /var/dt/A:0-
0s6gpT
root       989  0.2  0.2 1184 1064 pts/6    O 08:41:39  0:00 ps -augx
fazekasg   722  0.1  0.4 1992 1784 pts/6    R 15:39:28  0:00 /bin/tcsh
fazekasg   719  0.1  0.7 3840 3128 ??       S 15:39:28  0:00 /usr/openwin/bin/cmdtool
fazekasg   984  0.1  0.7 4528 3424 ?        S 08:40:53  0:00 /usr/openwin/bin/textedit
root         3  0.1  0.0    0    0 ?        S 14:52:17  0:37 fsflush
fazekasg   744  0.1  0.7 3864 3176 ??       S 15:57:25  0:00 /usr/openwin/bin/cmdtool
fazekasg   620  0.1  0.4 2472 2080 pts/2    S 15:01:22  0:00 olwm -syncpid 619
fazekasg   733  0.1  0.7 3848 3120 ??       S 15:43:04  0:00 /usr/openwin/bin/cmdtool
fazekasg   636  0.0  1.0 5496 4896 pts/2    S 15:01:38  0:03 /usr/openwin/bin/filemgr -Wp 0 291 -Ws 592 439 -WP 81 833
+Wi
root         0  0.0  0.0    0    0 ?        T 14:52:16  0:00 sched
root         1  0.0  0.1  664  312 ?        S 14:52:17  0:00 /etc/init -
root         2  0.0  0.0    0    0 ?        S 14:52:17  0:00 pageout
root       119  0.0  0.3 1872 1240 ?        S 14:53:04  0:00 /usr/sbin/rpcbind
root       121  0.0  0.3 1992 1296 ?        S 14:53:04  0:00 /usr/sbin/keyserv
root       127  0.0  0.3 1920 1488 ?        S 14:53:05  0:00 /usr/sbin/nis_cachemgr
root       149  0.0  0.4 1928 1712 ?        S 14:53:18  0:00 /usr/sbin/inetd -s
root       154  0.0  0.4 2264 1824 ?        S 14:53:18  0:00 /usr/lib/nfs/statd
root       156  0.0  0.3 1864 1528 ?        S 14:53:18  0:00 /usr/lib/nfs/lockd
root       174  0.0  0.5 2912 2488 ?        S 14:53:19  0:00 /usr/lib/autofs/automountd
root       178  0.0  0.4 3712 2032 ?        S 14:53:20  0:00 /usr/sbin/syslogd
root       189  0.0  0.3 1808 1456 ?        S 14:53:20  0:00 /usr/sbin/cron
root       201  0.0  0.5 2824 2512 ?        S 14:53:21  0:00 /usr/sbin/nscd
root       211  0.0  0.2 2640  952 ?        S 14:53:24  0:00 /usr/lib/lpsched
root       229  0.0  0.2 1152  840 ?        S 14:53:27  0:00 /usr/lib/power/powerd
root       232  0.0  0.4 2104 1696 ?        S 14:53:27  0:00 /usr/lib/sendmail -bd -q1h
root       244  0.0  0.2  888  712 ?        S 14:53:28  0:00 /usr/lib/utmpd
root       259  0.0  0.4 2248 1936 ?        S 14:53:29  0:01 /usr/sbin/vold
root       289  0.0  0.3 1840 1376 ?        S 14:53:35  0:00 /usr/lib/snmp/snmpdx -y -c /etc/snmp/conf
root       298  0.0  0.4 2496 1704 ?        S 14:53:37  0:00 /usr/lib/dmi/dmispd
root       299  0.0  0.5 3144 2304 ?        S 14:53:38  0:00 /usr/lib/dmi/snmpXdmid -s pader
```

# • Folyamat ütemezés (scheduling)

- Cél: mindig legyen legalább egy processzus, amelyik képes és kész a processzort lefoglalni.
- Processzus ütemezési sorok:
  - job queue (munka sor)
  - ready queue (készenléti sor)
  - device queue (berendezésre váró sor)
- Folyamat migráció az egyes sorok között:



- Folyamat ütemezôk:
  - Hosszútávú ütemezô *(long term scheduler, job scheduler)*
    - mi kerül a job queue-ba, lehet lassú, a multiprgramozás foka
  - Rövidtávú ütemezô *(short term scheduler, CPU scheduler)*
    - melyik folyamat kapja meg következô alkalommal a CPU-t, gyorsaság lényeges
  - Szempontok:
    - I/O igényes és CPU igényes folyamatok
    - Context switch (process context a processzus továbbindításához szükséges összes információ rendszerezve, struktúrálva – kapcsolódó adatszerkezetek.)



# • Folyamatokon végzett "mûveletek" (operációk)

- Processzus létrehozása
  - **(kezdeti betöltést kivéve processzust csak processzus hozhat létre!)**
- Mechanizmusa: egy szülô (parent) folyamat létrehozhat gyermek (child) folyamatokat, majd a gyermekek további gyermekeket $\rightarrow$ fastruktúra
- Erôforrás megosztás:
  - **Szülô és gyermek közösen használ minden erôforrást.**
  - **A gyermek a szülô erôforrásainak egy részét használhatja.**
  - **Nincs közös erôforráshasználat.**
- Végrehajtás:
  - **Szülô és gyermek konkurens módon fut. (UNIX: *parancs&*)**
  - **Szülô a gyermekre vár. (UNIX: *parancs* )**
- Címtér (address space)
  - **A gyermek a szülô duplikáltja.**
  - **A gyermek betölt egy programot önmaga helyett .**
- UNIX példák: **fork, execve**

- Processzus megszün(tet)ése (termination)
- A folyamat végrehajtja az utolsó utasítását, majd megkéri az operációs rendszert, hogy törölje (exit). Ennek során
  - output adatok kerülnek át a gyermektôl a szülôhöz (cf. **fork**),
  - a folyamat által használt erôforrásokat az operációs rendszer felszabadítja.
- A szülô folyamat felfüggesztheti a gyermek folyamatok végrehajtását (**abort**). Ennek lehetséges okai pl.:
  - A gyermek kimerítette a hozzárendelt erôforrásokat.
  - A gyermekhez rendelt taskra nincs többé szükség.
  - Maga a szülô is befejezi mûködését.
- Az operációs rendszer (legtöbb esetben) nem engedi meg, hogy egy gyermek tovább éljen, mint a szülô!
- Kaszkád termináció (A szülôvel együtt elhal az összes gyermeke).
- Példa: UNIX –
  - gyermek: **exit** rendszerhívás segítségével jelzi, hogy nem kíván tovább mûködni.
  - szülô: **wait** rendszerhívással várakozhat egy gyerek befejezôdésére.
  - az operációs rendszer a szülô befejezésével a gyerekeket is megszünteti!
- Folyamatok együttmûködése, kooperációja

- Független folyamatok nem befolyásolnak más folyamatokat és nem befolyásolhatók más folyamatok által.
- Együttmûködô folyamatok befolyásolhatnak más folyamatokat és befolyásolhatók lehetnek más folyamatok által.
- A kooperáció (kooperációt lehetôvé tevô operációs környezet) több vonatkozásban elônyös lehet:
  - Információ megosztás
  - Számítási folyamatok felgyorsítása
  - Modularitás
  - Kényelem, kényelmi szempontok.
- Termelô-fogyasztó folyamatok (a kooperáló folyamatok egy paradigmája)
  - Termelô (producer) folyamat valamilyen adatot hoz létre.
  - Fogyasztó (consumer) folyamat az adatot fel(el)- használja.
  - A kooperációhoz valamilyen puffer (“raktár”) szükséges.
  - Korlátlan (“végtelen”) puffer $\Leftrightarrow$ korlátos (“véges”) puffer megoldás.

- Közös memória véges pufferrel megoldás

| Megosztott adat | Termelô folyamat | Fogyasztó folyamat |
|---|---|---|
| **var** n;<br>**type** *item* = ... ;<br>**var** *buffer*: **array**[0..n-1] **of** *item*;<br>*in,out*: 0..n-1;<br>*in*:=0;<br>*out*:=0; | **repeat**<br>...<br>egy item létrehozása a *nextp* változóba<br>...<br>**while** (*in*+1) **mod** n = *out* **do** *nop*;<br>*buffer*[*in*]:= *nextp;*<br>*in*:=(*in*+1) **mod** n;<br>**until** *false*; | **repeat**<br>**while** *in=out* **do** *nop*;<br>*nextc*:=*buffer*[*out*];<br>*out*:=(*out*+1) **mod** n;<br>...<br>a *nextc* változóban levô adat “felhasználása”<br>...<br>**until** *false*; |

- A megoldás korrekt, de csak n-1 puffert tud feltölteni!

- Szálak (fonalak, könnyûsúlyú folyamatok, *thread, lightweight process*)
- A szál a CPU kiszolgálás egy alapegysége; jellemzôi:
  - program címszámláló
  - regiszter készlet (tartalma)
  - verem tartalma
- Egy szál megosztva használja a vele egyenrangú (társ)- szálakkal a
  - kód szekcióját,
  - adat szekcióját,
  - az operációs rendszer által biztosított erôforrásait;
  - ezek együttesét task-nak nevezzük.
  - Egy hagyományos, vagy könnyûsúlyú processzus nem más, mint egy task egyetlen szállal.
- Elôny: csökken a "context-switch" végrehajtására fordított idô!
  - Egy több szálat tartalmazó task esetén míg az egyik szál várakozik (blokkolt), a másik futhat.
    - Példák: fájl szerver, web - böngészô.

- A szálak olyan mechanizmust szolgáltatnak, amely lehetôvé teszi a szekvenciális processzusoknak a rendszerhívások blokkolását, s közben a "párhuzamosság elérését".
- Megvalósítása:
  - Kernel által támogatott szálak (Mach és OS/2)
  - Felhasználói szintû szálak
    - a kernel szint fölött helyezkednek el;
    - nincs *system call*, csak *library call* (a fejlesztô rendszer/fordító oldja meg);
    - Andrew project (CMU)
  - vegyes (hibrid) megközelítés: kernel- és felhasználói szintû szálak
    - Solaris 2 (1992)
- A szálak állapotaik, kezelésük, egyéb tulajdonságaik (kommunikáció, kooperáció) alapján a processzushoz állnak közel.

# • Processzusok közötti kommunikáció (IPC)

Az IPC olyan mechanizmust jelent, amely lehetôvé teszi, hogy folyamatok egymással kommunikáljanak, és akcióikat összehangolják, szinkronizálják.

- Üzenô rendszer: a folyamatok úgy kommunikálnak, hogy nem rendelkeznek közösen használható memóriával.
- IPC két mûveletet nyújt:
  - **send**(message)
    - az üzenetek lehetnek fix, vagy változó hosszúak
  - **receive**(message)
  - példák (UNIX rendszerhívások: send, sendmsg, socket, recv, recvfrom, … )
- Ha a P és Q folyamatok kommunikálni szeretnének, akkor szükségük van egy kommunikációs vonalra (communication link)
- A kommunikációs vonal implementációs kérdései
  - Fizikai implementáció: megosztott memória, hardver busz, hálózat, ...
  - Logikai implementáció:
    - Hogyan épül fel a link?
    - Tartozhat-e egynél több folyamathoz?
    - Két folyamat között hány élô link lehet? ... üzenet mérete, mozgás iránya, stb

- Direkt kommunikáció
  - **send**(P, message)
    - küldj egy üzenetet P-nek (utasítás Q-ban)
  - **receive**(Q, message)
    - fogadj egy üzenetet Q-tól (utasítás P-ben)
  - A kommunikációs vonal ebben az esetben automatikusan épül fel a két folyamat között (PID ismerete szükséges!)
  - A vonal pontosan két folyamat között létezik.
  - Minden egyes folyamatpár között pontosan egy link létezik
  - Egy, vagy többirányú is lehet.
  - Termelô-fogyasztó példa
  - Szimmetrikus/asszimetrikus címzés

- Indirekt kommunikáció
  - **send**(A, message)
    - küldj egy üzenetet az A Mail-boxba (utasítás Q-ban)
  - **receive**(A, message)
    - olvass ki egy üzenetet az A Mail-boxból (utasítás P-ben)
  - A kommunikációs vonal abben az esetben épül fel a két folyamat között ha közösen használhatják az A Mail-boxot (PID ismerete nem szükséges!)
  - A vonal több folyamat között létezik (mindenki, aki A-hoz hozzáférhet!).
  - Minden egyes folyamatpár között több link is létezik, létezhet (Mail-box-függô)
  - Egy, vagy többirányú is lehet.
  - “Ki kapja az üzenetet?” - probléma.

- Pufferelés (a link által egyidôben befogadott üzenetek száma)
  - Zéró kapacitás (0 üzenetet tárol a link; szinkronizáció kell, “rendezvous”)
  - Korlátozott kapacitás ( n üzenet lehet a linken)
  - Korlátlan kapacitás
  - Más: delay, reply, RPC (remote procedure call)

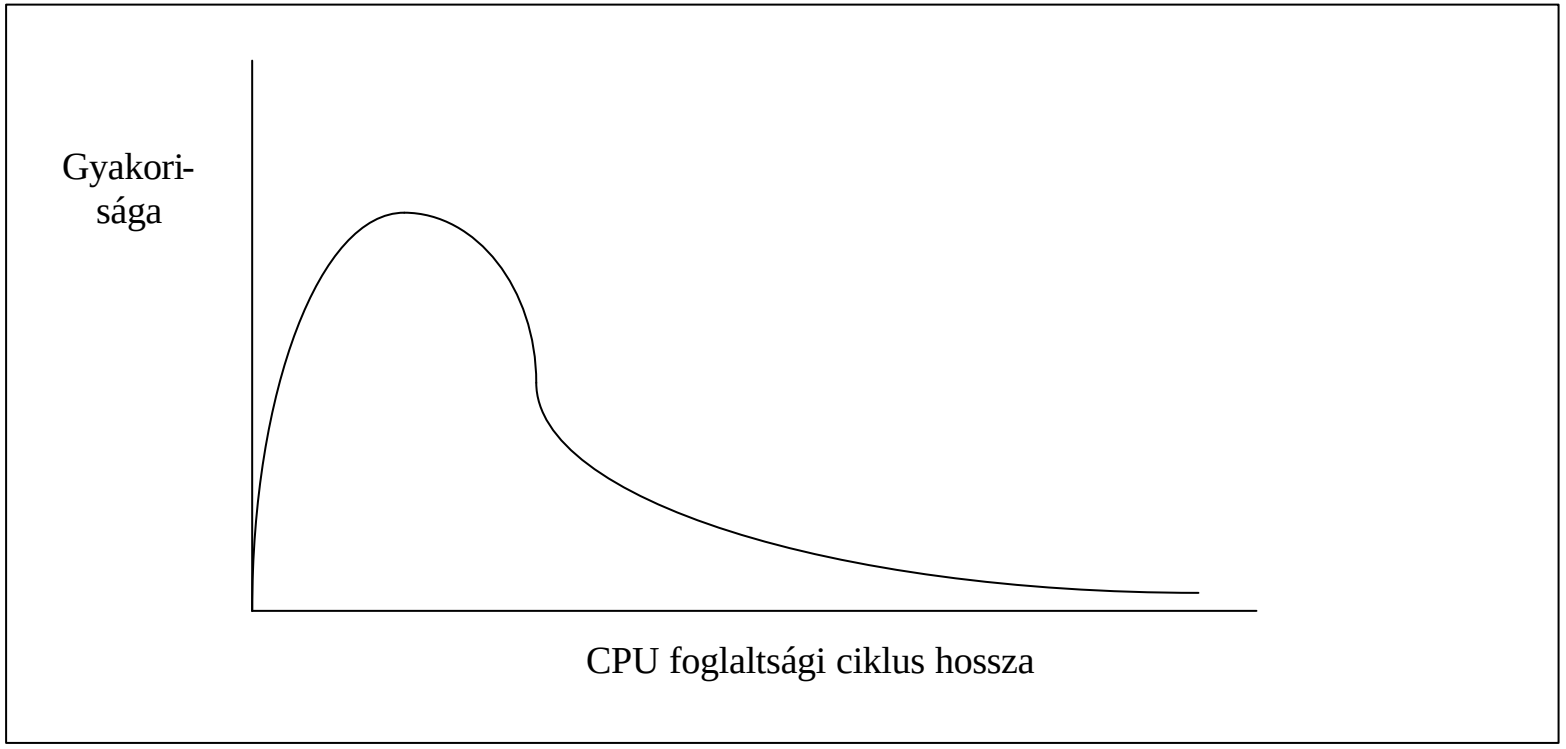
- Kivétel-kezelés, kivételes helyzetek
  - processzus felfüggesztés
  - elveszett  üzenetek
  - hibás (scrambled) üzenetek

# 5. CPU ütemezés

- Alapfogalmak
- Ütemezési feltételek (kritériumok)
- Ütemezési algoritmusok
- Több-processzoros eset
- Valós idejû (real time) ütemezés
- Algoritmus kiértékelés

# •Alapfogalmak

- A multiprogramozás célja: a CPU foglaltság (kihasználás) hatásfokának növelése
- A multiprogramozás lényege: egyidôben több folyamat is az operatív tárban helyezkedik el, készen állva a CPU kiszolgálására. Ha egy folyamatnak várnia kell (Pl. I/O-ra), a CPU-t egy másik folyamat kapja meg.
- CPU Burst cycle – I/O Burst Cycle (CPU foglaltsági ciklus – I/O ciklus)
- A CPU ciklusok gyakorisága:



- Rövid távú (*short term*) ütemezôrôl van szó
- Rövid távú ütemezési döntéshelyzet áll elô, ha egy folyamat
    1. futó állapotból várakozó állapotba kerül
    2. futó állapotból készenléti állapotba kerül
    3. várakozó állapotból készenléti állapotba kerül
    4. megáll
- Az 1. és 4. esetben az ütemezés *nem preemptív* (nem beavatkozó: a processzus maga mond le a CPU-ról) – (MS Win)
- Az 2. és 3. esetben az ütemezés *preemptív* (beavatkozó: elveszik tôle a CPU-t)
- A Diszpécser (dispacher)
    - A diszpécser adja át a vezérlést az ütemezô által kiválasztott folyamatnak. Ez magában foglalja:
        - a kontextus módosítását
        - user módba kapcsolást
        - a megfelelô címû utasításra ugrást (PC/IP beállitást)
- Diszpécser latencia : egy processzus megállításához és egy másik elindításához szükséges idô

# • Ütemezési feltételek (kritériumok)

- **CPU kiszolgálás** (CPU utilization) – a CPU az idô minél nagyobb részében legyen foglalt.
- **Átbocsátó képesség** (throughput) – egységnyi idô alatt befejezôdô processzusok száma.
- **Végrehajtási** (turnarund, fordulási) **idô** – = ∑ várakozás a memóriába kerülésre + készenléti sorban töltött idô + CPU-n töltött idô + I/O-val töltött idô.
- **Várakozási idô** (waiting time) – készenléti sorban (ready queue) töltött idô
- **Válaszidô** (response time) – a “kérés benyújtásától” az elsô válasz megjelenéséig (nem output!) eltelt idô. (Interaktív rendszerekben a turnaround nem jó jellemzô.)
- **Feladatok**: “tisztességes” kiszolgálás, minimum, maximum, átlag optimizálása, szórás optimizálása, jósolhatóság.

# • Ütemezési algoritmusok

- **Igénybejelentési sorrend szerinti kiszolgálás** (first-come, first-served = FCFS)
  - az átlagos várakozási idô nagy szórása, konvoj effektus
- **A rövidebb igény elôször** (shortest job first, shortest job next, SJF, SJN) kiszolgálás elve.
  - Elméletileg minimalizálja az átlagos várakozási idôt.
  - Probléma: nem tudjuk, hogy a sorbanállók közül melyik a “rövidebb”.
  - Printer spooling.
  - Predikció: a korábbi foglaltsági periódusok hosszának felhasználásával pl. exponenciális átlagolással (aging - öregedési algoritmus)

Jelölje $t_n$ az n-edik CPU foglaltsági ciklus hosszát, $t_{n+1}$ a következonek jósolt foglaltsági periódus hosszát.

Definiáljuk a következo foglaltság várható hosszát: Legyen $1 \geq \alpha \geq 0$ és

$$t_{n+1} = \alpha\, t_n + (1-\alpha)\, t_n$$

- Lehet preemptív, vagy non-preemptív.

- **Prioritási sorrend szerinti kiszolgálás**
- Prioritás: a processzushoz rendelt egész érték. A kisebb érték nagyobb prioritást jelent.
- Prioritási függvény
  - Az SJF, ill. FCFS is prioritásos kiszolgálásnak tekinthetô!
- Belsô és külsô prioritás.
- Preemptív, non-preemptív prioritási ütemezôk.
- Problémák: végtelen indefinit blokkolódás = éhezés, éhhalál.
  - Példa: MIT 1973 IBM 7094: 1967 óta várt egy processzus a CPU-ra!!
- Megoldás: aging (unix példa)

- **Round robin (RR, körleosztásos, körbejáró ütemezés)**
- Minden processzus sorban q ideig (q=10-100 millisec.) használhatja a CPU-t
- n processzus esetén a maximális várakozási idô (n-1)/q
- q nagy: $\Rightarrow$FCFS !
- q kicsi: $\Rightarrow$ kontextus váltási költség relatíve megnô!

# 6. Folyamat szinkronizáció

- Háttér
- A kritikus szakasz probléma
- Szinkronizációs hardver
- Szemaforok
- Klasszikus szinkronizációs problémák
- Kritikus régiók
- Monitorok

# •Háttér

- A termelô-fogyasztó folyamat egy módosított változata (counter!)

| Megosztott adat | Termelô folyamat | Fogyasztó folyamat |
|---|---|---|
| **var** n;<br>**type** *item* = ... ;<br>**var** *buffer*: **array**[0..n-1] **of** *item*;<br>*in,out,counter*: 0..n-1;<br>*in*:=0;<br>*out*:=0;<br>counter=0; | **repeat**<br>...<br>egy item létrehozása a *nextp* változóba<br>...<br>**while** *counter=n* **do** *nop*;<br>*buffer*[*in*]:= *nextp;*<br>*in*:=(*in*+1) **mod** n;<br>counter=counter+1;<br>**until** *false*; | **repeat**<br>**while** *counter = 0* **do** *nop*;<br>*nextc*:=*buffer*[*out*];<br>*out*:=(*out*+1) **mod** n;<br>counter=counter-1;<br>...<br>a *nextc* változóban levô adat “felhasználása”<br>...<br>**until** *false*; |

- A kód külön-külön korrekt, de konkurens módon végrehajtva nem az!

- A *counter:=counter* ± *1* utasítás assembly szintû implementálása gépi utasításokkal:

| | |
|---|---|
| *reg1:=counter* | *reg2:=counter* |
| *reg1:=reg1+1* | *reg2:=reg2 -1* |
| *counter:=reg1* | *counter:=reg2* |

- Legyen a *counter* értéke pl. 5! Akkor egy konkurens hozzáférés-pár után (Megszakitás minden gépi utasítás után bekövetkezhet!):

| | | | |
|---|---|---|---|
| L1 | Termelô | *reg1:=counter* | *reg1:=5* |
| L2 | Termelô | *reg1:=reg1+1* | *reg1:=6* |
| L3 | Fogyasztó | *reg2:=counter* | *reg2:=5* |
| L4 | Fogyasztó | *reg2:=reg2-1* | *reg2:=4* |
| L5 | Termelô | *counter:=reg1* | *counter:=6* |
| L6 | Fogyasztó | *counter:=reg2* | *counter:=4* |

Azaz a *counter* értéke 4 (és nem 5!).

# • A kritikus szakasz probléma

- n folyamat verseng egy bizonyos (közös) adat használatáért
- Mindegyik folyamatnak van egy kódszegmense, ahol ezt a bizonyos adatot eléri (olvassa/írja): ez az ún. *kritikus szakasz*.
- Probléma: hogyan biztosítható, hogy amíg egy processzus a kritikus szakaszát hajtja végre, addig egyetlen más processzus se léphessen be a saját kritikus szakaszába.
- Az i-edik processzus ($P_i$) szerkezete:

```
repeat
        entry szekció
    kritikus szekció
        exit szekció
    maradék szekció
until false;
```

- Követelmények:
  - Kölcsönös kizárás
    - nem lehet egynél több processzus a kritikus szakaszában
  - Progresszió
    - a kiválasztás (futásra) nem késleltethetô határozatlan ideig
  - Korlátozott várakozás
    - minden igény kiszolgálása megkezdôdik korlátozott számú kritikus szakasz végrehajtása után

- Algoritmusok két processzus esetére

| I. algoritmus | II. algoritmus | III. algoritmus |
|---|---|---|
| • közös változók<br>**var** *turn*: (0..1);<br>{kezdeti érték: *turn*=0;<br>ha *turn=i* $\Rightarrow$ $P_i$ beléphet a kritikus szekciójába} | • közös változók<br>**var** *flag*: **array**[0..1] **of** *boolean;*<br>{kezdeti érték: *flag[i]=false;*<br>ha *flag[i]=true* $\Rightarrow$ $P_i$ kész belépni a kritikus szekciójába} | • közös változók<br>• I. és II. kombinációja |
| • **Processzus $P_i$**<br>**repeat**<br>**while** *turn* ¹ *i* **do** *no-op;*<br>kritikus szekció<br>*turn =j;*<br>maradék szekció<br>**until false;** | • **Processzus $P_i$**<br>**repeat**<br>*flag[i]:= true;*<br>**while** *flag[j]* **do** *no-op;*<br>kritikus szekció<br>*flag[i]:= false;*<br>maradék szekció<br>**until false;** | • **Processzus $P_i$**<br>**repeat**<br>*flag[i]:= true;*<br>*turn:=j;*<br>**while** *flag[j]* **and** *turn=j* **do** *no-op;*<br>kritikus szekció<br>*flag[i]:= false;*<br>maradék szekció<br>**until false;** |
| Kölcsönös kizárás van, de progresszió nincs! | Progresszió van, de kölcsönös kizárás nincs! | Mindhárom feltételt kielégíti! |

# • Bakery algoritmus (kritikus szakasz n processzusra)

- Minden processzus kap egy sorszámot, mielôtt belép a kritikus szekciójába (ticket). A legkisebb sorszámú processzus hajthatja végre a kritikus szekcióját.
- Ha $P_i$ és $P_j$ sorszáma megegyezeik, akkor a kisebb indexû jogosult a kritikus szekciójába belépni.
- A sorszámozó rendszer mindig monoton növekvô sorszámokat generál.
  - Pl.: 1,2,2,3,3,3,4,4, ...
- Jelölés: “<” jelentse a (ticket #, process ID) párra definiált lexikografikus rendezést.

# Bakery algoritmus

- közös változók

**var** *choosing*: **array**[0..n-1] **of** *boolean;*
*number:* **array**[0..n-1] **of** *integer;*
{kezdeti érték: *i= 0,1, ... , n-1 -re choosing[i]:=false, number[i]:=0.*}

- **Processzus $P_i$**

**repeat**

```
choosing[i]:= true;
number[i]:=max(number[0], ... , number[n-1])+ 1;
choosing[i]:=false;
for j:=0 to n-1 do
  begin
    while choosing[j] do no-op;
    while number[j] ¹ 0 and (number[j],i)<(number[i],i) do no-op;
end;
```

kritikus szekció

```
number[i]:= 0;
```

maradék szekció

**until false;**

# • Szinkronizációs hardver

- A **test-and-set** (**TS**) gépi utasítás (bizonyos) architektúrákban egyszerre képes egy memória szó tartalmát lekérdezni (+/0/- ?) és ugyanakkor a szóba egy új értéket beírni.
- Felhasználása: A közös  adat elérésének ténye más processzusok számára érzékelhetôvé tehetô!

# • Szemaforok (Dijkstra, 1965)

- Aktív várakozás (busy waiting) problémája
- Szemafor (S)
  - integer változó
  - csupán az alábbi két elemi (atomikus) mûvelet hajtható rajta végre

*wait(S):* *S:=S-1;*
**if** *S<0* **then** *block(S)*

*signal(S):* *S:=S+1;*
**if** *S£ 0* **then** *wakeup(S)*

- block(S) – felfüggeszti a hívó processzus végrehajtását
  - a processzust hozzáadja az S szemaforon alvó processzusok sorához
- wakeup(S) – reaktivál pontosan egy, korábban block(S) hívást kezdeményezô folyamatot
  - az S szemaforon alvó processzusok sorából egyet felébreszt

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

  Let $S$ and $Q$ be two semaphores initialized to 1

| $P_0$ | $P_1$ |
|---|---|
| $wait(S)$ | $wait(Q)$ |
| $wait(Q)$ | $wait(S)$ |
| . | . |
| . | . |
| . | . |
| $signal(S)$ | $signal(Q)$ |
| $signal(Q)$ | $signal(S)$ |

- Starvation – indefinite blocking

  A process may never be removed from the semaphore queue in which it is suspended.

# • Klasszikus szinkronizációs problémák

## • A korlátos puffer probléma

- a modellezéshez használt szemaforok: *mutex, empty, full*
- *empty* kezdôértéke: n
- *full* kezdôértéke: 0
- számláló (*counting*) szemaforok.

## • Az olvasók és írók (readers and writers) probléma

- Egy adatot, állományt több processzus megosztva, párhuzamosan használ, egyesek csak olvassák, mások csak írják. Hogyan biztosítható az adatok konzisztenciája?
- Stratégiák:

  1. Minden olvasó azonnal hozzáférhet az adatokhoz, hacsak egy író nem kapott már engedélyt az írásra.

  2. Egy író azonnal írhat, ha erre kész és más írók éppen nem írnak.

- Mindkét stratégia éhezéshez (starvation) vezethet, de van megoldás.

## Bounded-Buffer Problem

- Shared data

  **type** *item* = ...

  **var** *buffer* = ...

  *full*, *empty*, *mutex*: *semaphore*;

  *nextp*, *nextc*: *item*;

  *full* := 0; *empty* := *n* ; *mutex* := 1;

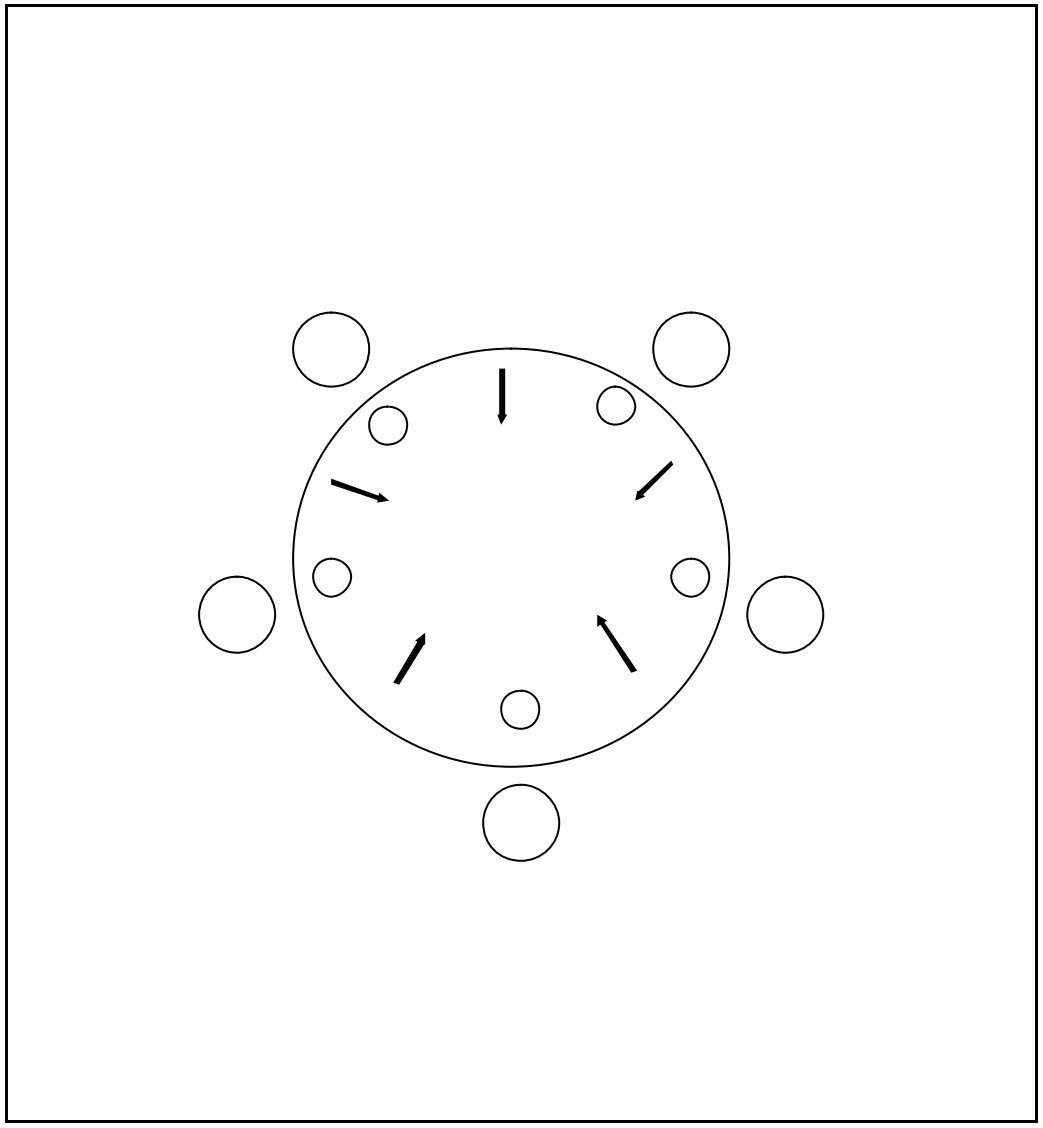
- Producer process

```
repeat
    ...
    produce an item in nextp
    ...
    wait(empty);
    wait(mutex);
    ...
    add nextp to buffer
    ...
    signal(mutex);
    signal(full);
until false;
```

- Consumer process

```
repeat
    wait(full);
    wait(mutex);
        ...
    remove an item from buffer to nextc
        ...
    signal(mutex);
    signal(empty);
        ...
    consume the item in nextc
        ...
until false;
```

# • A vacsorázó filozófusok (dining philosophers) problémája



- Egy köralakú asztal mellett öt filozófus ül, mindegyik elôtt van egy tányér rizs és a szomszédos tányérok között egy-egy evô-pálcika.
- Evéshez a filozófus a saját tányérja melletti két evôeszközt használ-hatja úgy, hogy ezeket egymás után kézbe veszi.
- Ha befejezte az étkezést, vissza-teszi az eszközöket, és gondolkodni kezd.
- Majd újra megéhezik, stb.
- Példa számos konkurencia kontroll problémára.

# • Kritikus régiók

- Magas szintû szinkronizációs eszköz.
- Egy megosztott változó *v*, amelynek típusa *T*, deklarációja

**var** *v:* **shared** *T*;

- A **v** változó csak az alábbi alakú utasításokon keresztül érhetô el:

**region** *v* **when** *B* **do** *S*;

ahol *B* egy logikai kifejezés.

- Amíg az *S* utasítás végrehajtás alatt áll, másik processzus nem érheti el a *v* változót.
- Ha a processzus megpróbálja végrehajtani a region utasítást, a *B* logikai kifejezés kiértékelôdik. Ha *B* igaz, az *S* utasítás végrehajtódik. Ha hamis, akkor a processzus végrehajtása késleltetôdik addig, amíg a kifejezés igaz nem lesz és egyetlen más processzus sincsen a *v*-hez kapcsolt régióban.
- Példa: korlátos puffer problémája
- Implementáció: pl. szemaforokkal

- Példa: A korlátos puffer probléma egy lehetséges megoldása.

- **Megosztott puffer:**

```
var buffer: shared record
                pool: array [0..n–1] of item;
                count,in,out: integer;
            end;
```

- *nextc* **az osztott pufferbe:**

```
region buffer when count < n
   do begin
         pool[in] := nextp;
         in := in+1 mod n;
         count := count + 1;
      end;
```

- **Fogyasztó kiveszi** *nextc-t*

```
region buffer when count > 0
   do begin
         nextc := pool[out];
         out := out+1 mod n;
         count := count – 1;
      end;
```

# Monitorok (Hoare, 1974; Hansen, 1975)

- magas szintû szinkronizációs eszközök (szikronizációs primitív), amelyek lehetôvé teszik egy absztrakt adattípus biztonságos megosztását konkurens processzusok között.
- formálisan: a monitor eljárások, változók és adatszerkezetek együttese, amelyek egy speciális csomagba vannak integrálva. A processzusok hívhatják a monitorban levô eljárásokat, de nem érhetik el közvetlenül annak belsô adatszerkezetét.
- Speciális típus: **condition**
- Speciális mûveletek: **x.wait, x.signal**
- A monitor azt kívánja biztosítani, hogy egy idôben csak egy processzus legyen aktív (rajta).
- Megvalósítás: pl. szemaforokkal.

- Példa: A dining philosophers probléma egy lehetséges megoldása.

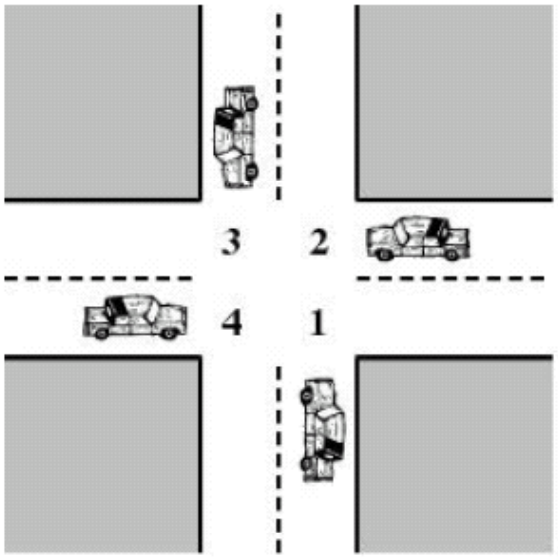
```
type dining-philosophers = monitor
    var state : array [0..4] of (thinking, hungry, eating);
    var self : array [0..4] of condition;

    procedure entry pickup (i: 0..4);
        begin
            state[i] := hungry;
            test (i);
            if state[i] ≠ eating then self[i].wait;
        end;

    procedure entry putdown (i: 0..4);
        begin
            state[i] := thinking;
            test (i+4 mod 5);
            test (i+1 mod 5);
        end;

    procedure test (k: 0..4);
        begin
            if state[k+4 mod 5] ≠ eating
              and state[k] = hungry
              and state[k+1 mod 5] ≠ eating
              then begin
                        state[k] := eating;
                        self[k].signal;
                   end;
        end;

    begin
        for i := 0 to 4
            do state[i] := thinking;
    end.
```

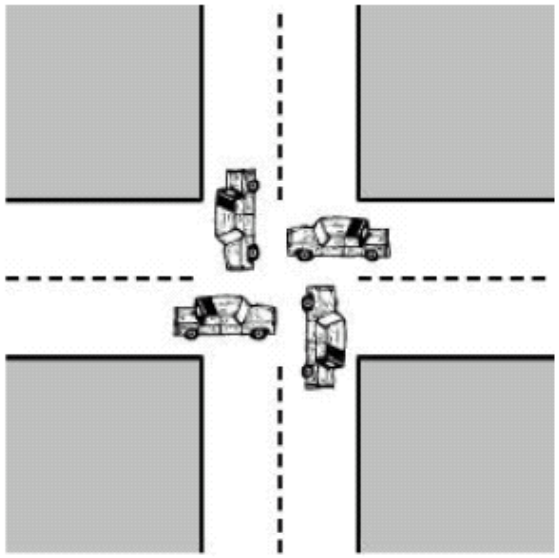
# 7. Holtpont és éhezés

1. A holtpont fogalma
2. Holtpont megelőzés
3. Holtpont elkerülés
4. Holtpont detektálás
5. A UNIX konkurenciakezelése

# A HOLTPONT

- holtpont fogalma: a rendszererőforrásokért versengő vagy egymással kommunikáló processzusok állandósult blokkoltsága
- Nincs általános megoldás!!
- Két vagy több processzus erőforrásszükségletek miatt állnak egymással konfliktusban



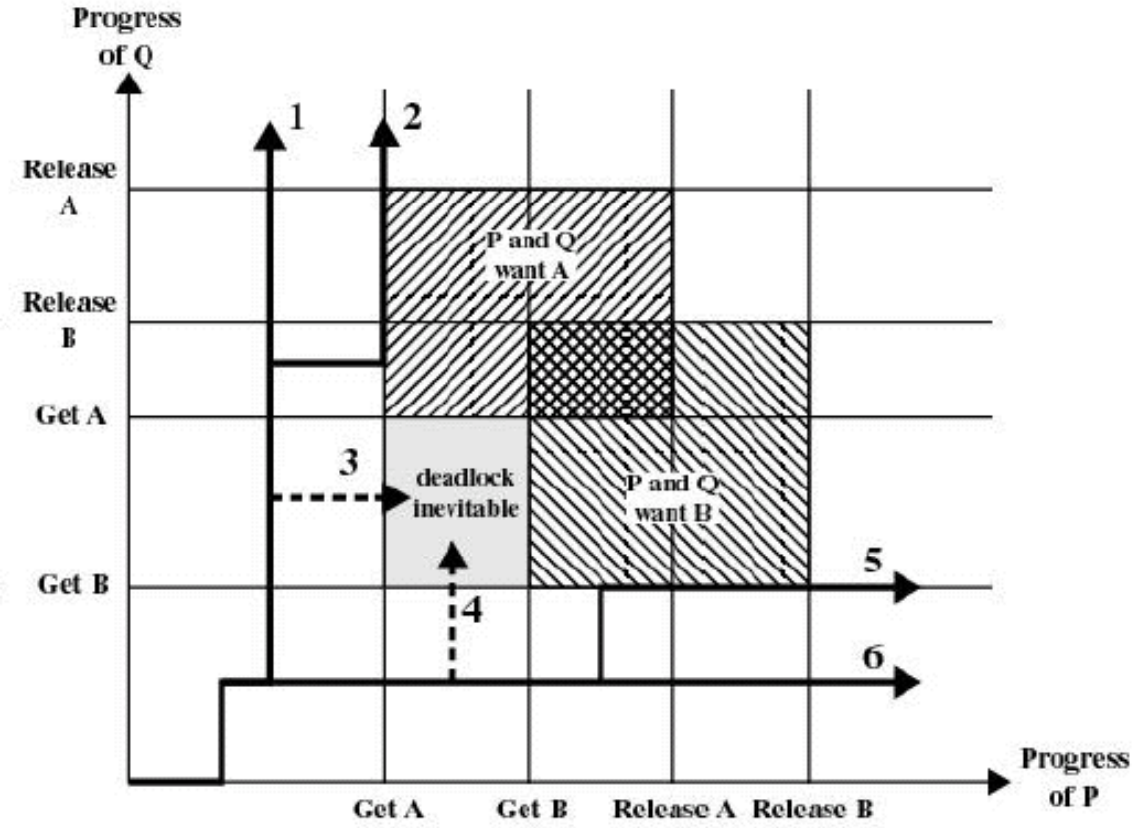
***a) Holtpont lehetséges*** ***a) Holtpont***

***1. ábra Holtpont (illusztráció)***

# A HOLTPONT

- Példa: két processzus (*P, Q*), két erőforrás (*A, B*), mindkét processzus igényt tart mindkét erőforrásra. A 2. ábra a hat lehetséges végrehajtási útvonalat mutatja (egyprocesszoros rendszerben egyszerre egy processzus végrehajtása lehetséges!)

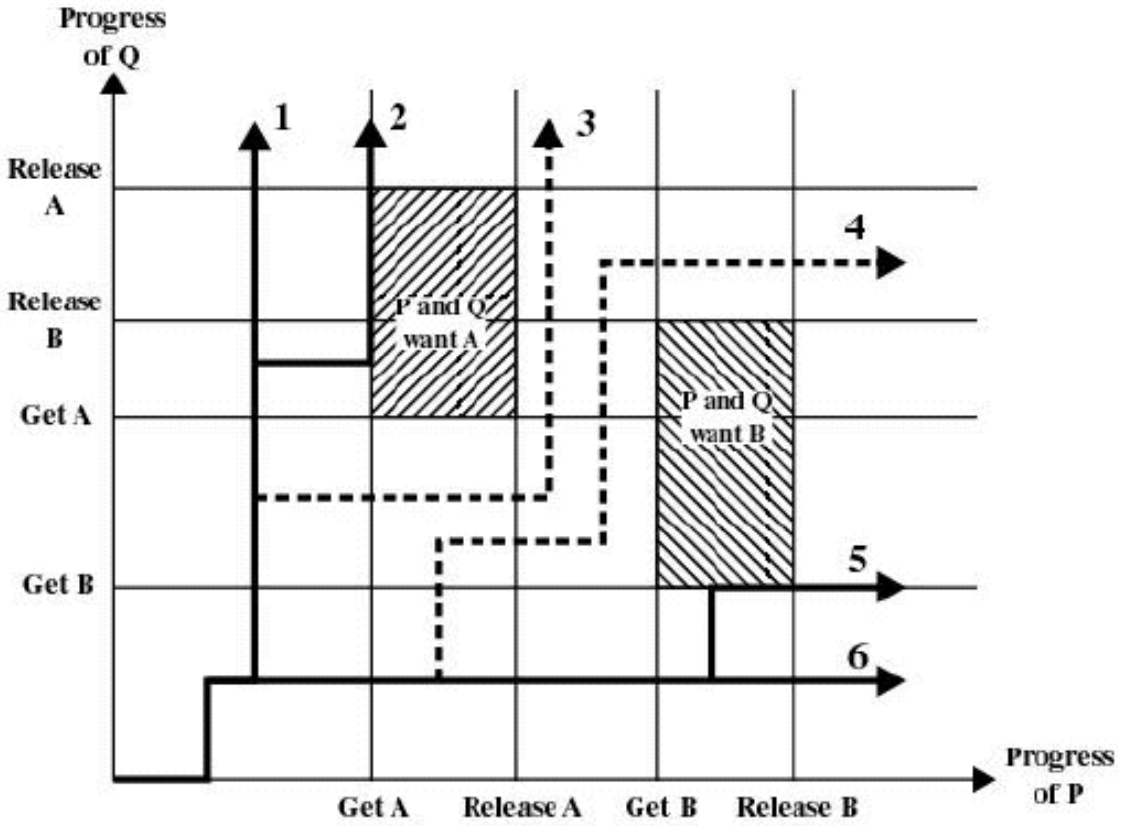
- a 3. és 4. útvonalnál a holtpont elkerülhetetlen!




***2. ábra Példa holtpontra***

# A HOLTPONT

- Példa: két processzus (*P, Q*), két erőforrás (*A, B*), csak az egyik processzus (*Q*) tart igényt egyszerre mindkét erőforrásra. A *P* processzus az erőforrásokat egymás után használja.




***3. ábra Példa holtpont elkerülésére***

# A HOLTPONT

**Újrahasználható erőforrások:**

- egyszerre egy processzus használja de a használat során nem „merül” ki
- processzusok elnyerik az erőforrást, melyet később felszabadítanak, hogy egy másik processzus használni tudja
- pl: processzorok, I/O csatornák, fő és másodlagos memóriák, fájlok, adatbázisok és szemaforok
- holtpont történik, ha mindkét processzus fenntart egy-egy erőforrást és a másikért folyamodik (*4. ábra* – a végrehajtási sorrend: p0p1q0q1p2q2..... holtpont)

**Process P**

| Step | Action |
|---|---|
| $p_0$ | Request (D) |
| $p_1$ | Lock (D) |
| $p_2$ | Request (T) |
| $p_3$ | Lock (T) |
| $p_4$ | Perform function |
| $p_5$ | Unlock (D) |
| $p_6$ | Unlock (T) |

**Process Q**

| Step | Action |
|---|---|
| $q_0$ | Request (T) |
| $q_1$ | Lock (T) |
| $q_2$ | Request (D) |
| $q_3$ | Lock (D) |
| $q_4$ | Perform function |
| $q_5$ | Unlock (T) |
| $q_6$ | Unlock (D) |

***4. ábra Két processzus újrahasználható erőforrásokért versenyez***

# A HOLTPONT

**Fel/el-használható erőforrások:**

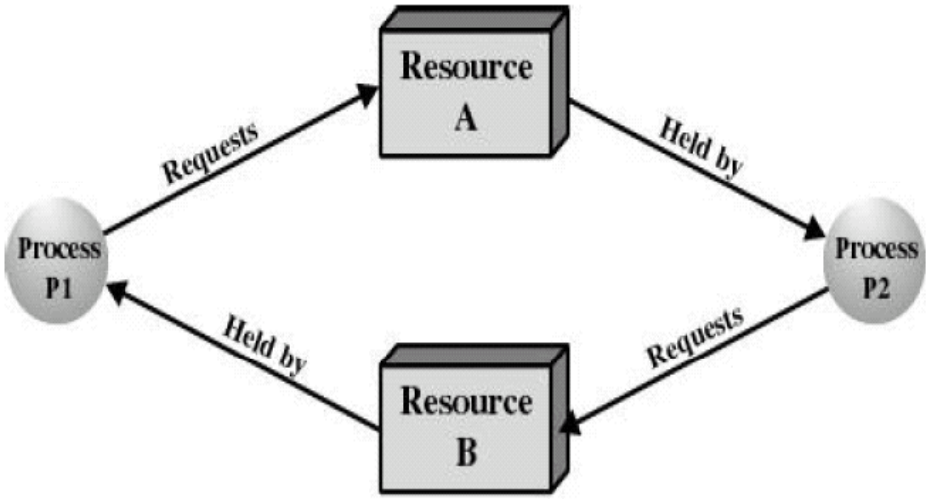
- processzus által létrehozott és megsemmisített erőforrások
- pl: megszakítások, szignálok, üzenetek és I/O pufferekben lévő információk
- két processzus (*P1, P2*) egymástól vár üzenetet, majd annak megkapása után üzenetet küld a másiknak. Így holtpont állhat elő, hiszen a *Receive* blokkolttá válik (*5. ábra*)


***5. ábra Két processzus felhasználható erőforrások által okozott holtponthelyzete***

# A HOLTPONT

**Holtpont kialakulásához vezető (de egyébként szükséges) stratégiák:**

- Kölcsönös kizárás: egyszerre csak egy processzus használhat egy erőforrást
- Tartani és várni (*Hold-and-wait)*
  - egy processzus lefoglalva tart erőforrásokat, míg más erőforrások megszerzésére vár
- Nincs beavatkozás:
  - erőforrást nem lehet erőszakosan elvenni egy processzustól, mely éppen használja
- Körkörös várakozás
  - processzusok zárt lánca keletkezik, ahol minden processzus lefoglalva tart egy erőforrást, melyre a következő processzusnak szüksége van (*3.ábra*)




***5. ábra Körkörös várakozás***

# Holtpont megelőzés

**Stratégiák szerinti prevenció:**

- Kölcsönös kizárás: nincs lehetőség megelőzésre
- Hold and wait:
  - blokkolni a processzust, amíg az összes számára szükséges erőforrás fel nem szabadul
  - egy processzushoz rendelt erőforrás sokáig üresjáratban lehet; ezalatt kiosztható más processzus számára
- Nincs beavatkozás:
  - ha egy processzus számára nem lehetséges további igényelt erőforrás elnyerése, akkor a korábban lefoglalt erőforrásokat fel kell szabadítania
  - az op. rendszer beavatkozhat és felszabadíthat egy erőforrást
- Körkörös várakozás:
  - erőforrások lineáris elrendezése
  - amíg egy erőforrás elfoglalt, addig csak a listán magasabban levő erőforrás elérhető

# Holtpont elkerülése

**Holtpont elkerülésének két megközelítése:**

- ne indítsunk el egy processzust, ha igényei holtponthoz vezetnek!
- ne elégítsünk ki erőforráskérelmet, ha az allokáció holtponthoz vezethet!

**Processzus indításának megtagadása:**

- *n* processzus, *m* erőforrás esetén bevezetésre kerül: erőforrás (*Resource*) vektor ($R_1$,...,$R_m$), rendelkezésre álló erőforrások (*Available*) vektora ($V_1$,...,$V_m$), allokációs (*Allocation*) mátrix ($A_{11}$,....$A_{nm}$), illetve az összes processzus összes erőforrásra vonatkozó igényeinek (*Claim*) mátrixa ($C_{11}$,...,$C_{nm}$)
- Így: egy új processzus akkor indítható el, ha $R_i \geq C_{(n+1)i} + \sum^{n}_{k=1} C_{ki}$ az összes *i*-re
- ez nem egy optimális stratégia, ugyanis a legrosszabbat feltételezi: az összes processzus egyszerre akarja megszerezni az összes szügségletet

# Holtpont elkerülése

**Erőforrás lefoglalásának megtagadása:**

- úgy is nevezik, hogy bankár algoritmus
- a rendszer állapota: az erőforrások aktuális kiosztása processzusokhoz
- biztonságos állapot az, amiből legalább egy végrehajtási sorrend lehetséges, mely nem holtponttal végződik (nem biztonságos állapot az, amire ez nem igaz)
- nincs visszaszorítás és beavatkozás!
- bankár algoritmusra vonatkozó korlátok:
  - a maximum erőforrásszükségletet előre meg kell állapítani
  - fix számú erőforrás foglalható csak le
  - processzus nem léphet ki, amíg erőforrást foglal éppen le

# Holtpont észlelése

**Holtpont detektálási algoritmus:**

- allokációs mátrix ($A$), erőforrás vektor, elérhetőségi vektor
- kérelem mátrix $Q$ bevezetése, ahol $q_{Ij}$ jelenti az $I$ processzus által igényelt $j$ típusú erőforrások mennyiségét
- kezdetben minden processzus jelöletlen
- Az algoritmus:
  1. jelöljünk meg minden processzust, melynek allokációs mátrixbeli sora csupa 0
  2. legyen $W$ egy vektor, mely megegyezik az elérhetőségi vektorral
  3. keressünk olyan processzust ($i$), mely jelöletlen, és $Q_{ik} \leq W_k$, ahol $1 \leq k \leq m$. Ha ilyen nincs, szakítsuk meg az algoritmust!
  4. ha van, jelöljük meg a processzust és állítsuk be az új $W$-t: $W_k = W_k + A_{ik}$, ahol $1 \leq k \leq m$, majd lépjünk vissza a 3. lépésre
- holpont létezik, ha az algoritmus végén jelöletlen processzusok maradnak

| | R1 | R2 | R3 | R4 | R5 |
|---|---|---|---|---|---|
| P1 | 0 | 1 | 0 | 0 | 1 |
| P2 | 0 | 0 | 1 | 0 | 1 |
| P3 | 0 | 0 | 0 | 0 | 1 |
| P4 | 1 | 0 | 1 | 0 | 1 |

Request matrix

| | R1 | R2 | R3 | R4 | R5 |
|---|---|---|---|---|---|
| P1 | 1 | 0 | 1 | 1 | 0 |
| P2 | 1 | 1 | 0 | 0 | 0 |
| P3 | 0 | 0 | 0 | 1 | 0 |
| P4 | 0 | 0 | 0 | 0 | 0 |

Allocation matrix

| R1 | R2 | R3 | R4 | R5 |
|---|---|---|---|---|
| 2 | 1 | 1 | 2 | 1 |

Resource vektor

| R1 | R2 | R3 | R4 | R5 |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |

Available vektor

# Holtpont észlelése

**Helyreállítási stratégia:**

- az összes holtpontot okozó processzus felfüggesztése (ez a leggyakoribb)
- az összes holtpontban levő processzus visszaállítása egy előzetesen definiált ellenőrzési pontra és az összes processzus újraindítása
  - az eredeti holtpont újból bekövetkezhet....
- a processzusok egymás után való leállítása, amíg a holtpont megszűnik, minden egyes processzus leállítása után a holtpontdetektáló algoritmus újraindítása szükséges
- az erőforrások egymás után való felszabadítása, amíg a holtpontszituáció meg nem szűnik (detektálóalgoritmus újraindítása minden erőforrásfelszabadítás után)
- a processzusok kiválasztása bizonyos megfontolások alapján történik (leghosszabb hátralevő futási idő, legkevesebb lefoglalt erőforrással rendelkező, kisebb priorítású processzusok, stb.)

# A UNIX KONKURENCIAKEZELÉSE

**A konkurenciakezeléshez használatos objektumok:**

- Csatornák (*Pipes*)
  - körkörös puffer, mely két processzus termelő-fogyasztó modellen alapuló kommunikációját tesz lehetővé (first-in-first-out). Kölcsönös kizárás szükséges!
- Üzenetek (*Messages*)
- Osztott memória (*Shared memory*)
  - leggyorsabb formája a processzusok közötti kommunikációnak
- Szemaforok
  - a következő elemekből áll: 1. a szemafor aktuális értéke, 2. a legutóbb szemaforon működő processzus azonosítója, 3. azon processzusok száma, mely arra vár, hogy a szemafor értéke nagyobb legyen, mint jelenlegi értéke, 4. azon processzusok száma, mely arra vár, hogy a szemafor értéke zérus legyen
- Szignálok
  - hasonlatosak a hardver megszakításhoz, de prioritás nélküliek

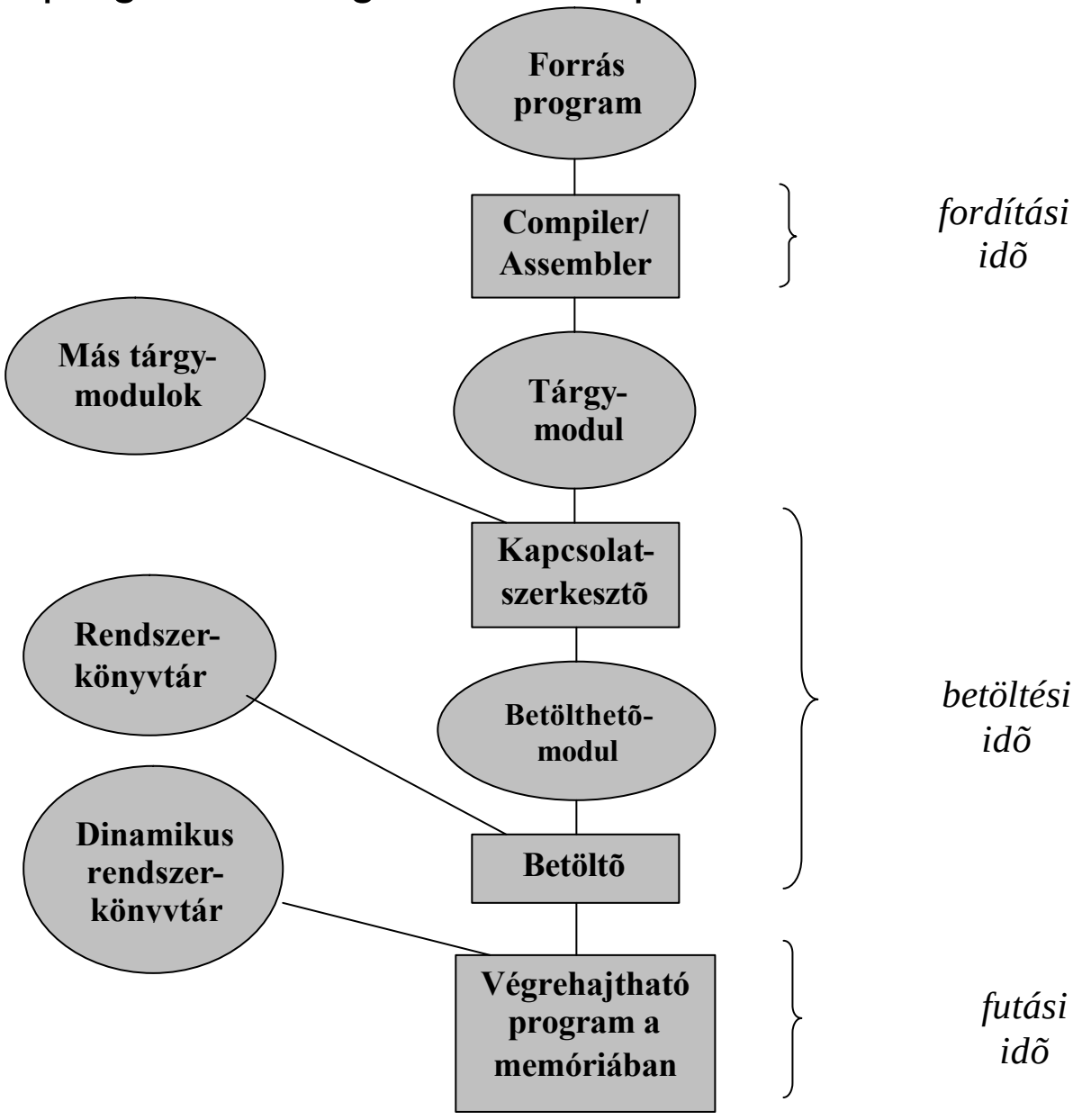
# 8. Memória management

- Háttér
- Logikai és fizikai címtér
- Swapping
- Folytonos allokálás
- Lapozás
- Szegmentáció
- Szegmentáció lapozással

# •Háttér

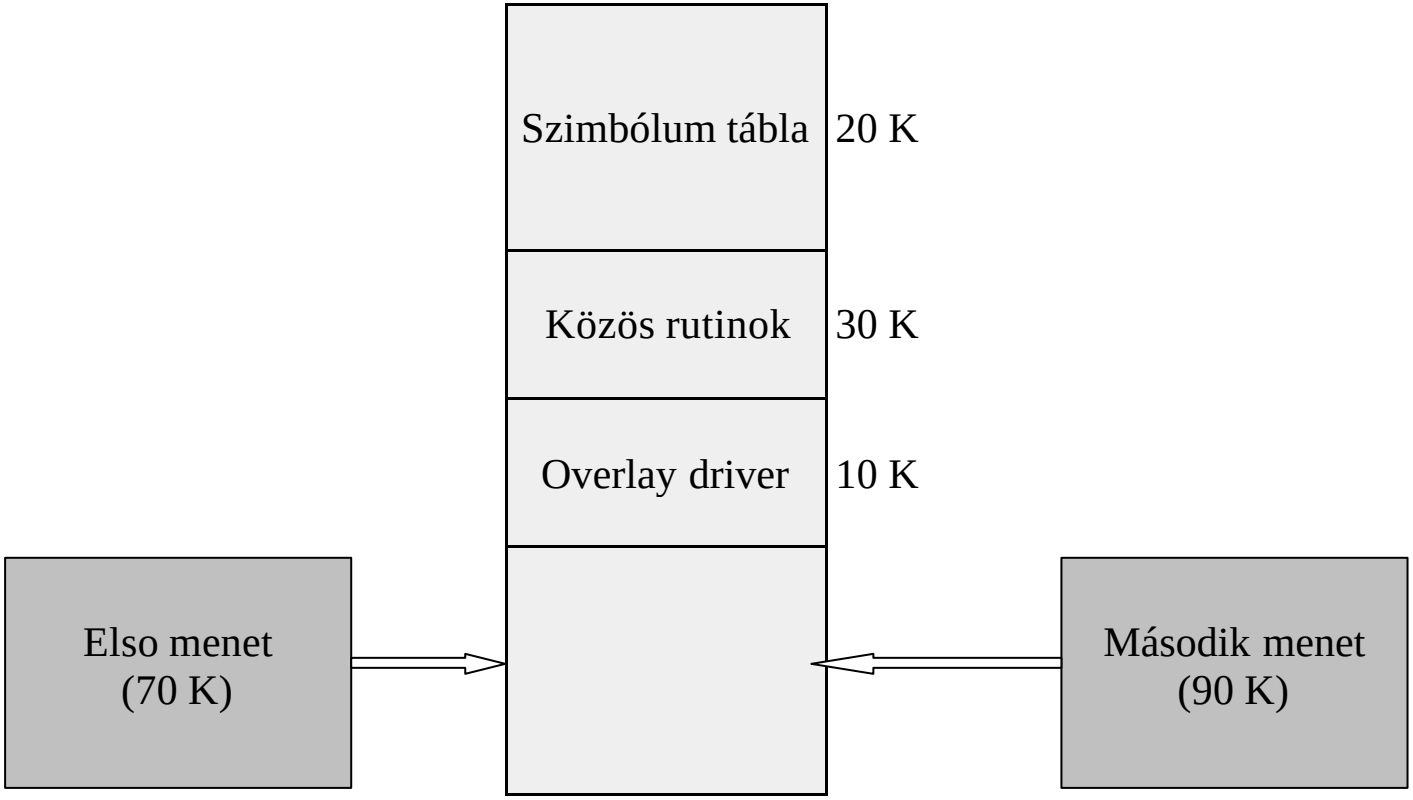
- Az számítógép (processzor) kapacitásának jobb kihasználása megköveteli, hogy egyszerre több processzus osztozzon a memórián (shared memory).
- Egy program alapvetôen valamilyen (bináris végrehajtható) fájl formában helyezkedik el a háttértárban. Végrehajtáshoz be kell tölteni a memóriába.
  - Végrehajtás közben – a memória kezelési stratégiától függôen – többször mozoghat a memória és háttértár között.
- *Input queue* – a végrehajtásra kijelölt és evégett sorban álló programok együttese.
- A programkódhoz és a program változókhoz valamikor memória címeket kell rendelni (*address binding*). (Ez történhet a betöltés elôtt, közben, vagy akár utána is.)
  - Fordítási idôben történô tárfoglalás (címkapcsolás).
  - Betöltési (szerkesztési) idôben történô tárfoglalás (címkapcsolás).
  - Futási idôben történô tárfoglalás (címkapcsolás).

- Egy felhasználói program feldolgozásának lépései



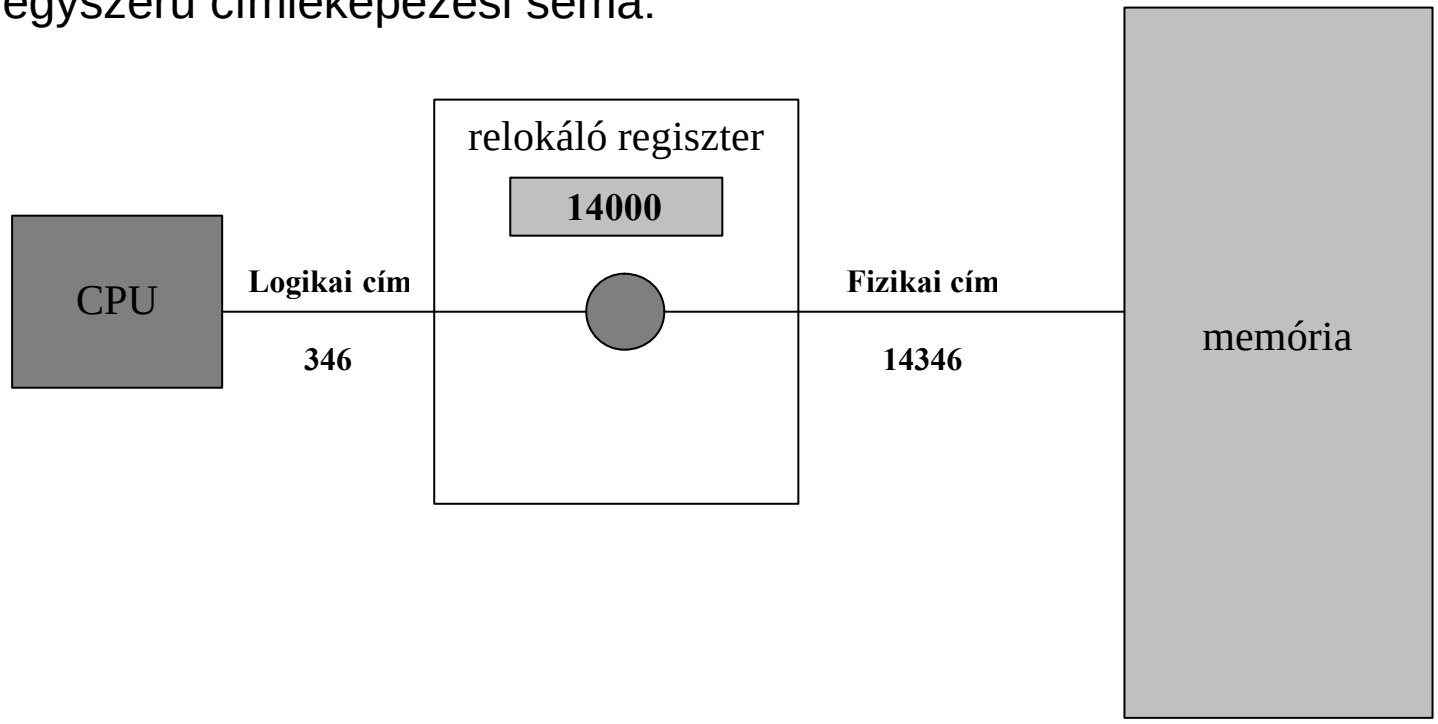
- Dinamikus betöltés
  - Egy szubrutin nem töltôdik be, amíg meg nem hívódik. Minden szubrutin a lemezen található áthelyezhetô bináris formában.
  - A hívó rutin elôször tisztázza, hogy a hívott benn van-e a memóriában. Ha nincs, akkor aktivizálódik az áthelyezô betöltô, és a betöltés után a program címhivatkozásai (címtáblázat, címkonstansok) módosulnak.
  - A nem szükséges rutinok soha nem töltôdnek be!
  - Nincs szükség speciális operációs rendszer támogatásra (a futtató rendszer - *run time system* saját hatáskörén belül megoldja).
- Dinamikus szerkesztés
  - Statikus szerkesztés: a nyelvi könyvtárak úgy kezelôdnek, mint bármely más (felhasználói) object modul. Probléma: a gyakran használt rutinok sok-sok végrehajtható program kódjával együtt letárolódnak. (lemez-pazarlás!)
  - Dinamikus szerkesztésnél nemcsak az (áthelyezô) betöltés, hanem a (szimbolikus) kapcsolat szerkesztés is kitolódik a futási idôre. Egy kisméretû kód (*stub*=csonk) helyettesíti a szükséges rutint a végrehajtható program kódjában, amely segítségével a szükséges rutin a memóriában (ha az memória rezidens), vagy a lemezes könyvtárban lokalizálható. A lokalizálás után (következô alkalommal) a rutin már direkt módon hajtódik végre (nincs újra töltés!).
  - További elônyök: könyvtár módosítások, verziók, *bug fixes, patches, service pack, shared library.*
  - Operációs rendszer támogatás: védett területre betöltött rutinok elérése.

- Overlay
  - A felhasználói program logikájába (szubrutin struktúrájába) beépített dinamika.
  - Alapötlet: a teljes programnak (kód és adat) csak az a része legyen benn az operatív memóriában, amelyre ténylegesen szükség van.
  - Példa: többmenetes fordítók, assemblerek.



# • Logikai és fizikai címtér

- Logikai cím = a CPU által generált cím (virtuális cím).
- Fizikai cím = a *Memory Management Unit* (MMU) által generált cím (reális cím).
- A fordítási és betöltési idôben csak a logikai címtér (címhozzárendelés) elérhetô!
- A logikai címet a MMU képezi le fizikai címmé.
- Egy egyszerû címleképezési séma:



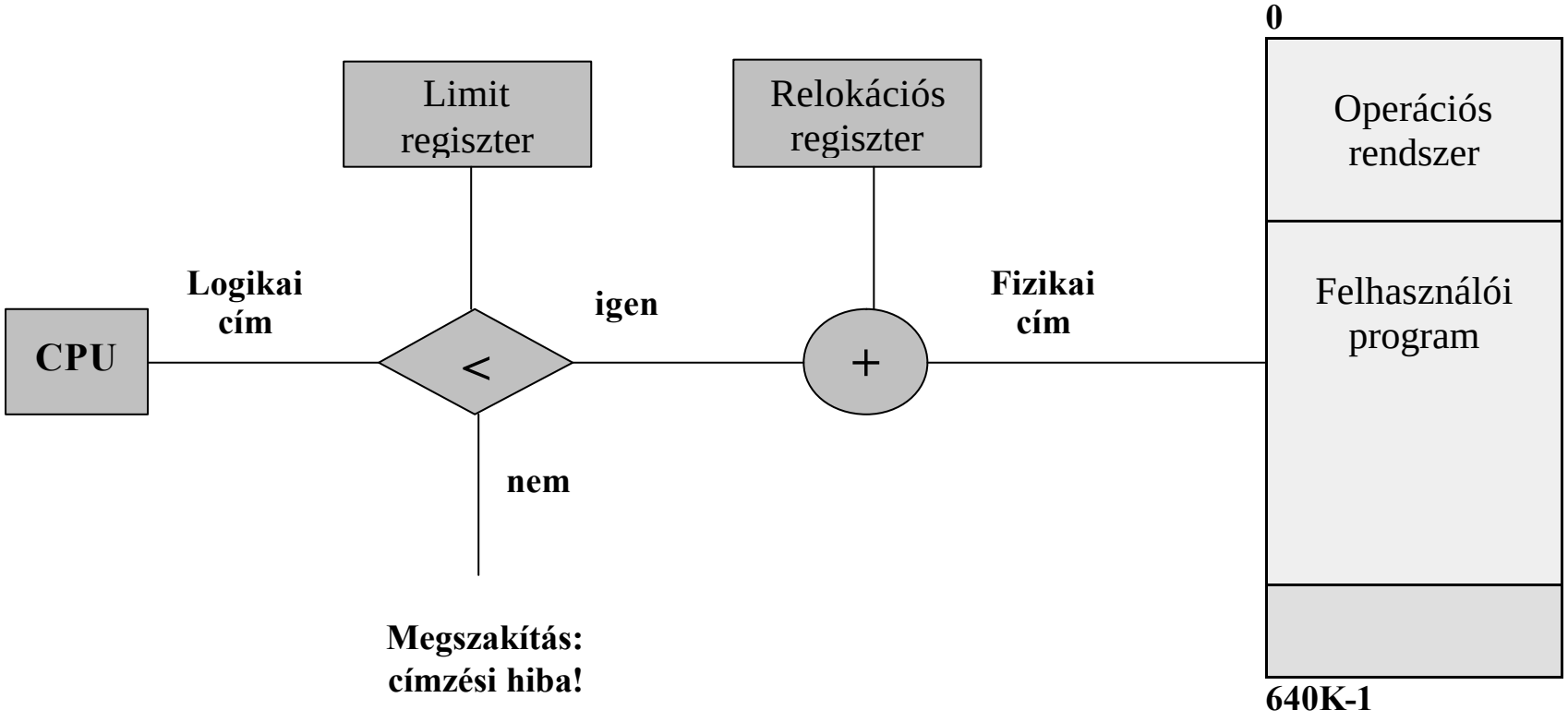
# • Swapping

- swap = csere (egy futó processzus kódjának és adatainak "lecserélése" egy háttértárban tároltra).
- Háttértár: nagy, összefüggô, gyorsan elérhetô lemezterület, amely elég nagy ahhoz, hogy minden futó program memóriabeli képét (core image) tárolja.
- Roll out, roll in: swapping stratégia változat: az alacsonyabb prioritású folyamatok kicserélôdnek a magasabb prioritásúakra.
- A swap idô nagyrészt *adatátviteli idô(!)*, arányos a processzus által lefoglalt operatív memória méretével.
- A swapping egyes verziói megtalálhatók a UNIX-ban (automatikus) és az MS Windows-ban is (kézi vezérelt?).
- Round Robin példa: idôosztás és átviteli idô.
- Más problémák: függô I/O.

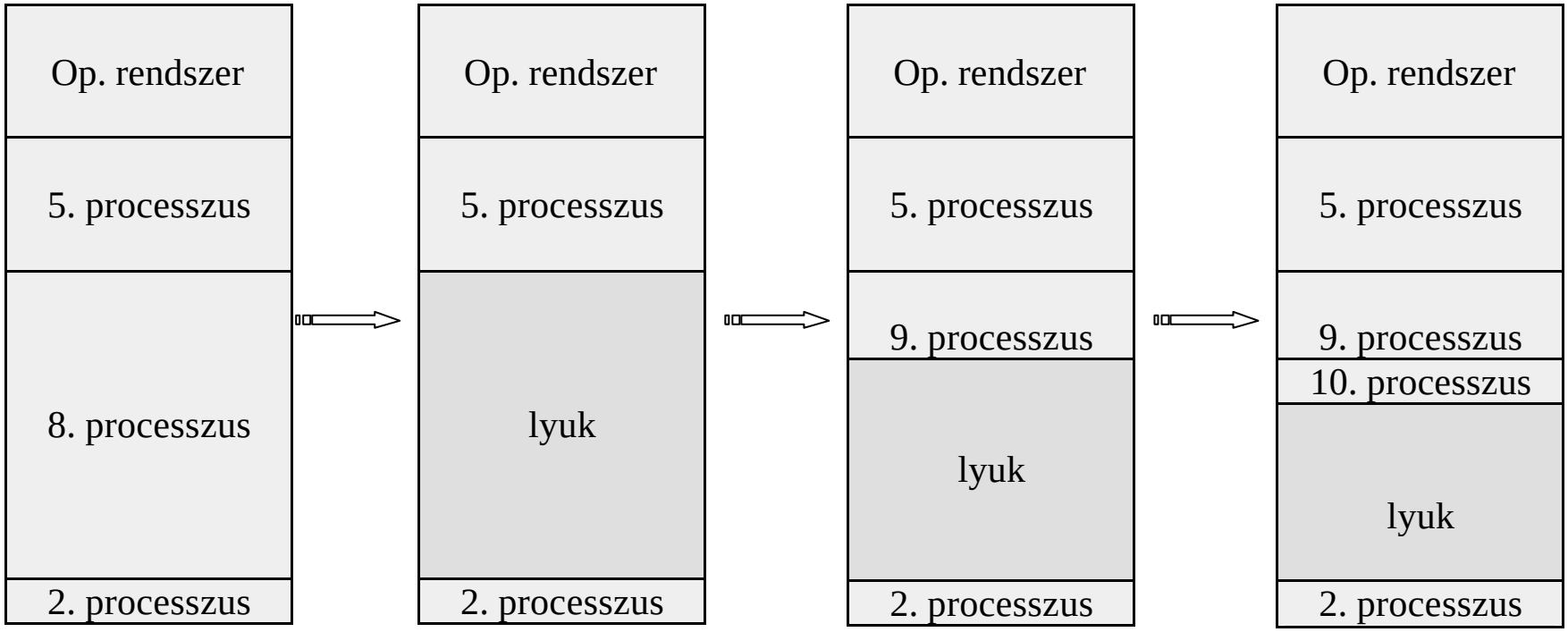
# • Folytonos tár-allokálás

- Az operatív memóriát egyetlen folytonos tömbnek tekinthetjük.
- A memória "rendszer"- és "felhasználói program" részekre bomlik.
- A rendszer résznek tartalmaznia kell a memóriába beágyazott megszakítási vektort, az I/O kapukat (portok).

• Egyszerû partíciós allokálás



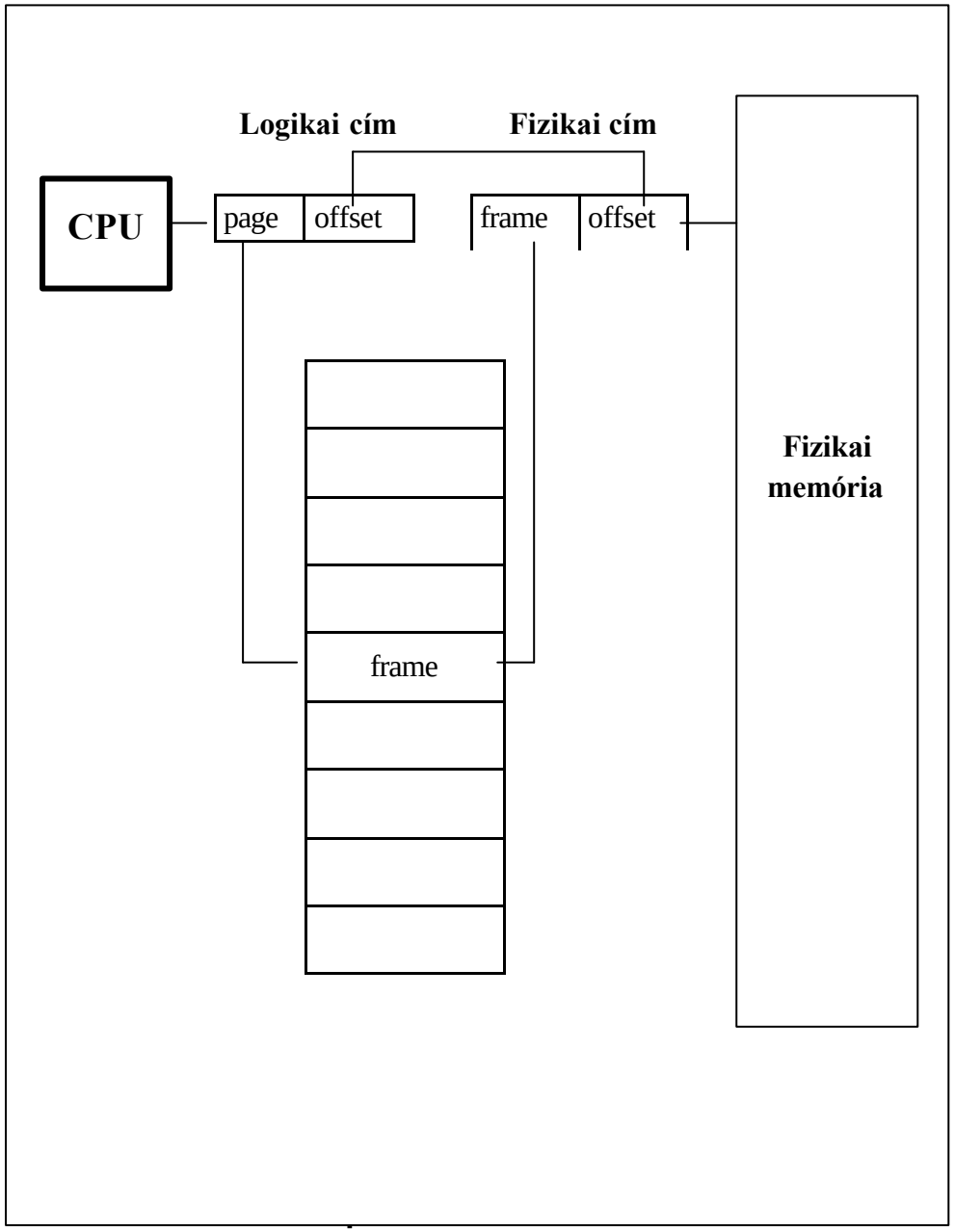
- Multipartíciós allokálás
  - A felhasználói programok számára elérhetô terület részekre (partíciókra) bomlik. Egy partíció egy felhasználói program (processzus) befogadására alkalmas.
  - Fix, változó számú és hosszúságú partíciók, prioritás.
  - Lyuk (*hole*): két foglalt partíció közötti szabad memória terület (blokk). A lyukak mérete változó lehet.
  - Ha egy processzust létre kell hozni, akkor ehhez az operációs rendszernek egy megfelelôen nagy méretû lyukat kell kiválasztania.
  - Példa:

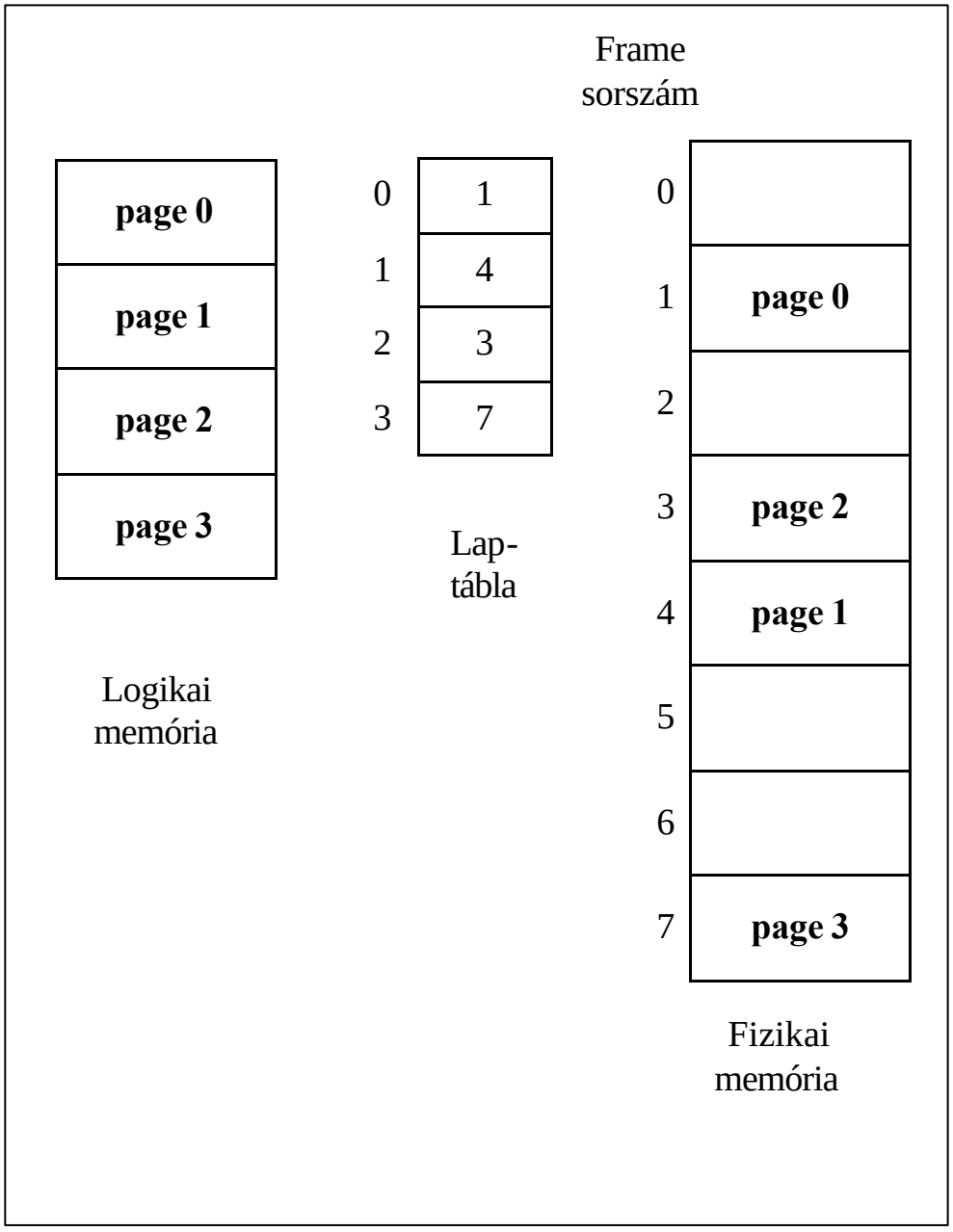


- Az operációs rendszer nyomon követi az allokált és szabad partíciókat (hely és méret alapján).
- Dinamikus tár-hozzárendelési probléma: hogyan lehet kielégíteni egy adott méretû allokációs (tárfoglalási) igényt?
  - **First -fit**: foglaljuk le az elsô lyukat, amely elég hosszú!
  - **Best-fit**: foglaljuk le az elég hosszú lyukak közül azt, amelynek hossza legközelebb esik a szükséges hosszhoz!
  - **Worst-fit**: foglaljuk le a leghosszabb lyukat (ha az elég hosszú)!
- Értékelési szempontok / értékelés:
  - Sebesség, tárkihasználás: a "first" és "best" jobb, mint a "worst".
  - A lyukak adminisztrálása is idôt és memóriát igényel, deadlockhoz is vezethet! (többe kerülhet mint a szabad hely!)
  - 50% szabály: (D. Knuth) a *first fit* stratégia statisztikai vizsgálata azt eredményezte, hogy n blokk elhelyezése során n/2 blokk (helye) elvész(!) (ez a kezdeti szabad terület egy harmada!).
- Külsô és belsô fragmentáció.
- A külsô fragmentáció csökkentése tömörítéssel.
  - cél: a lyukak egyesítése egy szabad területté.
  - dinamikus relokáció szükséges
  - függôben levô I/O problémája.

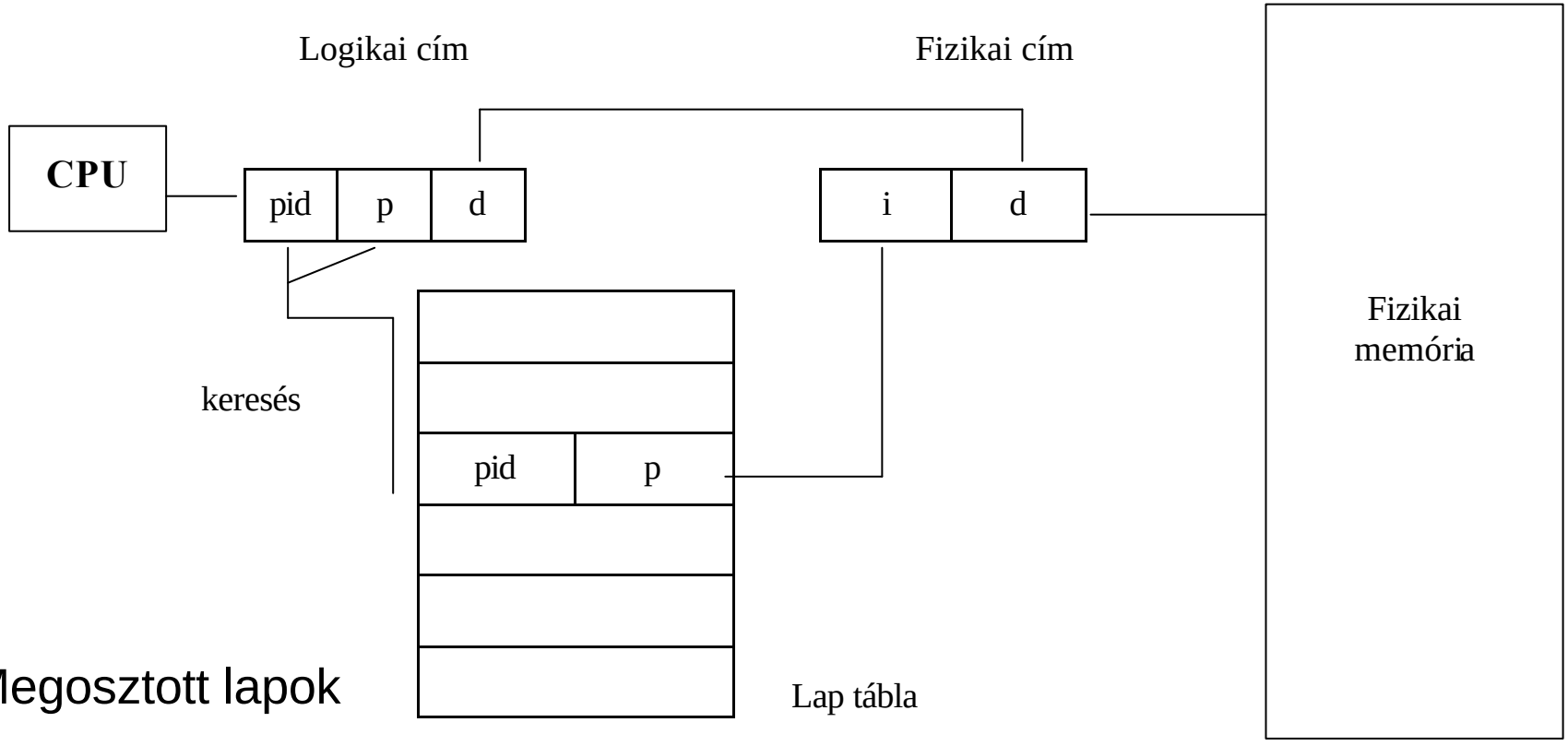
# • Lapozás (paging) – nem folytonos tár-allokálás

- A külsô fragmentáció problémájának egy lehetséges megoldását kapjuk, ha megengedjük, hogy a fizikai címtér ne legyen folytonos, megengedve egy processzushoz fizikailag össze nem függô memória blokkok allokálását.
- A logikai- és fizikai címtér független blokkokra (lapokra, keretekre) bomlik.
- A logikai blokk (lap/page) és a fizikai blokk (keret/frame) mérete megegyezik.
  - A méret 2 hatvány, jellemzôen 512-8192.
- Bármelyik lap elfoglalhatja bármelyik keretet.
- Nyilván kell tartani a szabad és foglalt kereteket.
- Egy *n* lapból álló program futtatásához elôbb *n* szabad keretet kell találni!
- Belsô fragmentáció.
  - (Külsô fragmentáció nincs, mert egy keret mindig teljesen lefoglalódik)
- A címképzés mechanizmusa:
  - **logikai cím**: (lap *sorszáma*, lapon belüli *relatív cím*)
  - **fizikai cím**: (keret *memóriabeli kezdôcíme*, kereten belüli *relatív cím*)
  - **laptábla**: minden logikai laphoz tartalmaz egy bejegyzést, amely a logikai lapot tartalmazó keret fizikai címe + más információk.


Logikai cím
Fizikai cím
CPU
page
offset
frame
offset
frame
Fizikai memória

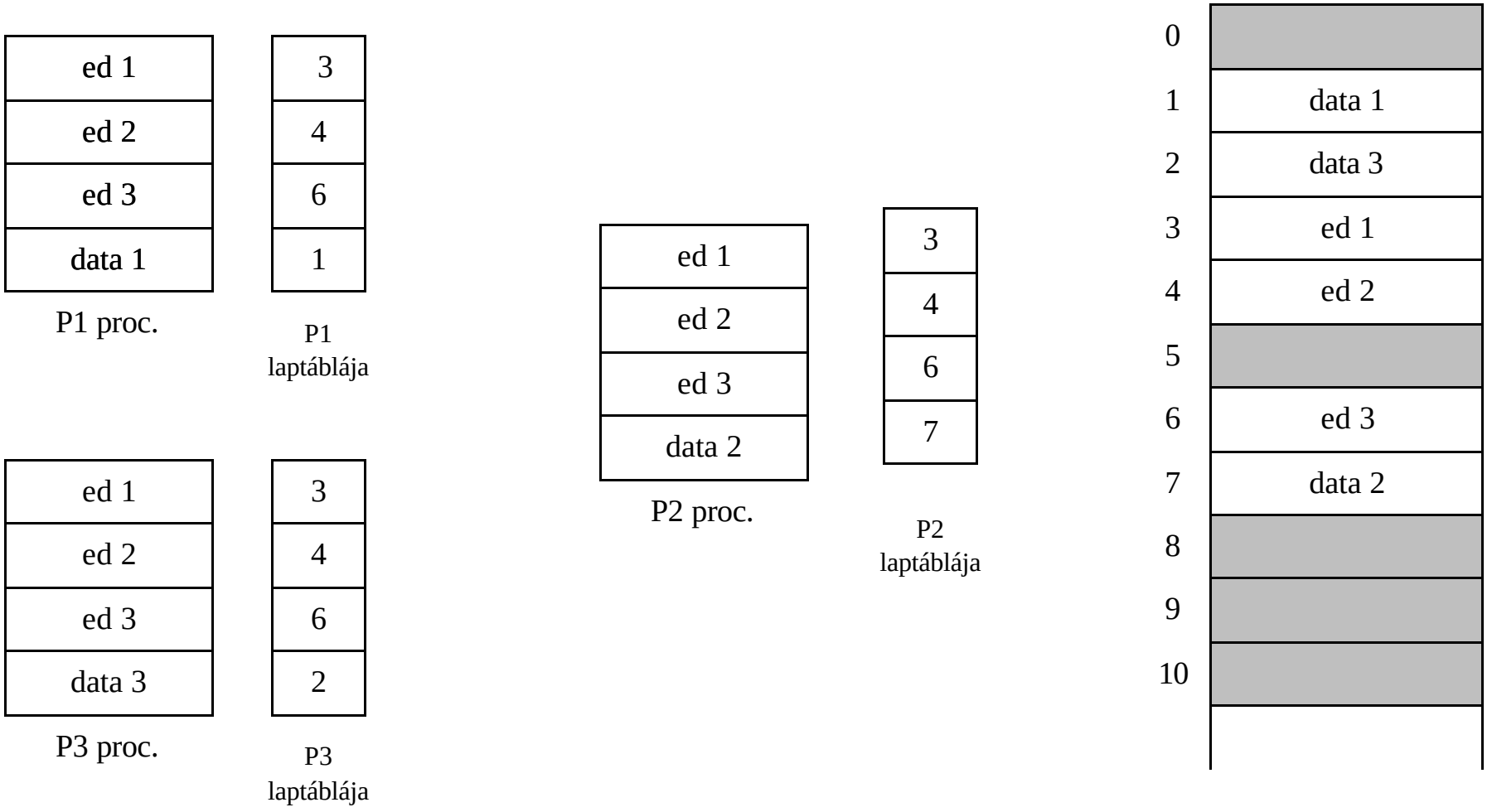

Frame sorszám
page 0
page 1
page 2
page 3
Logikai memória
0 1
1 4
2 3
3 7
Lap-tábla
0
1 page 0
2
3 page 2
4 page 1
5
6
7 page 3
Fizikai memória

- Minden egyes fizikai laphoz (frame-hez) tartalmaz egy bejegyzést (entry). Egy bejegyzés az adott frame-ben tárolt logikai lap virtuális címét és annak a processzusnak az azonosítóját tartalmazza, amelyikhez a frame tartozik.
- Csökken a laptáblák tárolásához szükséges memória mérete, de nô a tábla átnézéséhez keresési idô a lapreferencia felmerülése esetén.
- Hash - megoldásokkal a keresési idô csökkenthetô.

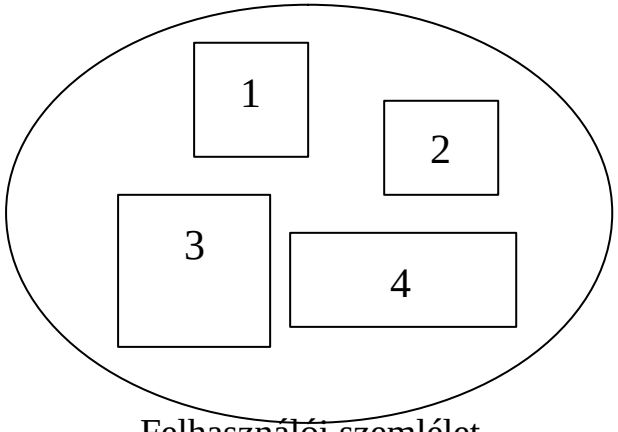



- Megosztott lapok

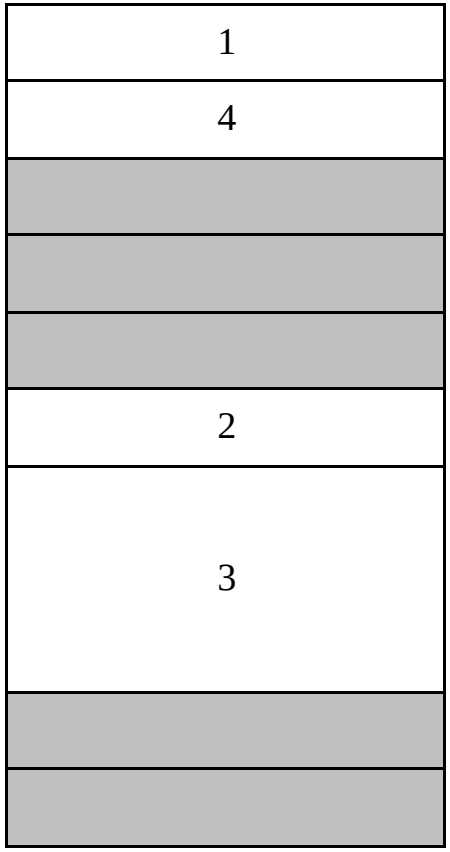
- A közös (re-entrant) csak olvasható kód (lap) megosztva használható több processzus által (pl. szövegszerkesztôk, kompájlerek, ablak rendszerek).



- Szegmentáció – szemantikus memória felosztás
  - A szegmentáció egy *felhasználói szemléletet tükrözô* memória kezelési sémát jelent.
- A program szegmensek együttese. A szegmens logikai egység, mint pl.
  - fôprogram
  - eljárás
  - függvény
  - lokális változók
  - globális változók
  - közös változók (common block)
  - verem
  - szimbólum táblázatok, tömbök
- Pl.




Felhasználói szemlélet




fizikai memória

- A logikai cím két részbôl áll:

**< szegmens szám, offset >**

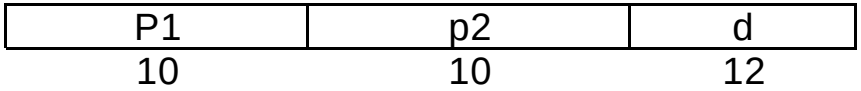
- Szegmens tábla – két dimenziós felhasználó által definiált címeket egy dimenziós fizikai címekké alakít; a táblában minden bejegyzés tartalmaz egy
  - bázist – amely a szegmens fizikai kezdôcímét adja meg,
  - mérethatárt (limit), amely a szegmens hosszát mondja meg.
- Szegmens táblázat bázis regiszter (STBR): a szegmens tábla memóriabeli helyére (kezdôcím) mutat (pointer).
- Szegmens táblázat hossz regiszter (STLR): a szegmens tábla maximális bejegyzéseinek számát adja meg.
  - az ***s*** szegmens szám akkor legális, ha ***s*** < [STLR].
  - Relokáció: – dinamikus
    – szegmens táblázat segítségével
  - Megosztás: – megosztott szegmensek
    – azonos szegmens (sor)szám
  - (Tár)védelem: minden egyes bejegyzéshez a szegmens táblában kapcsolódik egy:
    – érvényesítô (validation) bit (=0 $\Rightarrow$ illegális szegmens)
    – read/write/execute privilégiumok
  - Allokáció: – a hosszútávu ütemezônek el kell helyeznie a memóriában egy processzus összes szegmensét (hasonló megoldások és problémák lépnek fel, mint a változó hosszúságú multiparticiós rendszereknél)
    – first fit /best fit
    – külsô fragmentáció

# • Szegmentáció lapozással

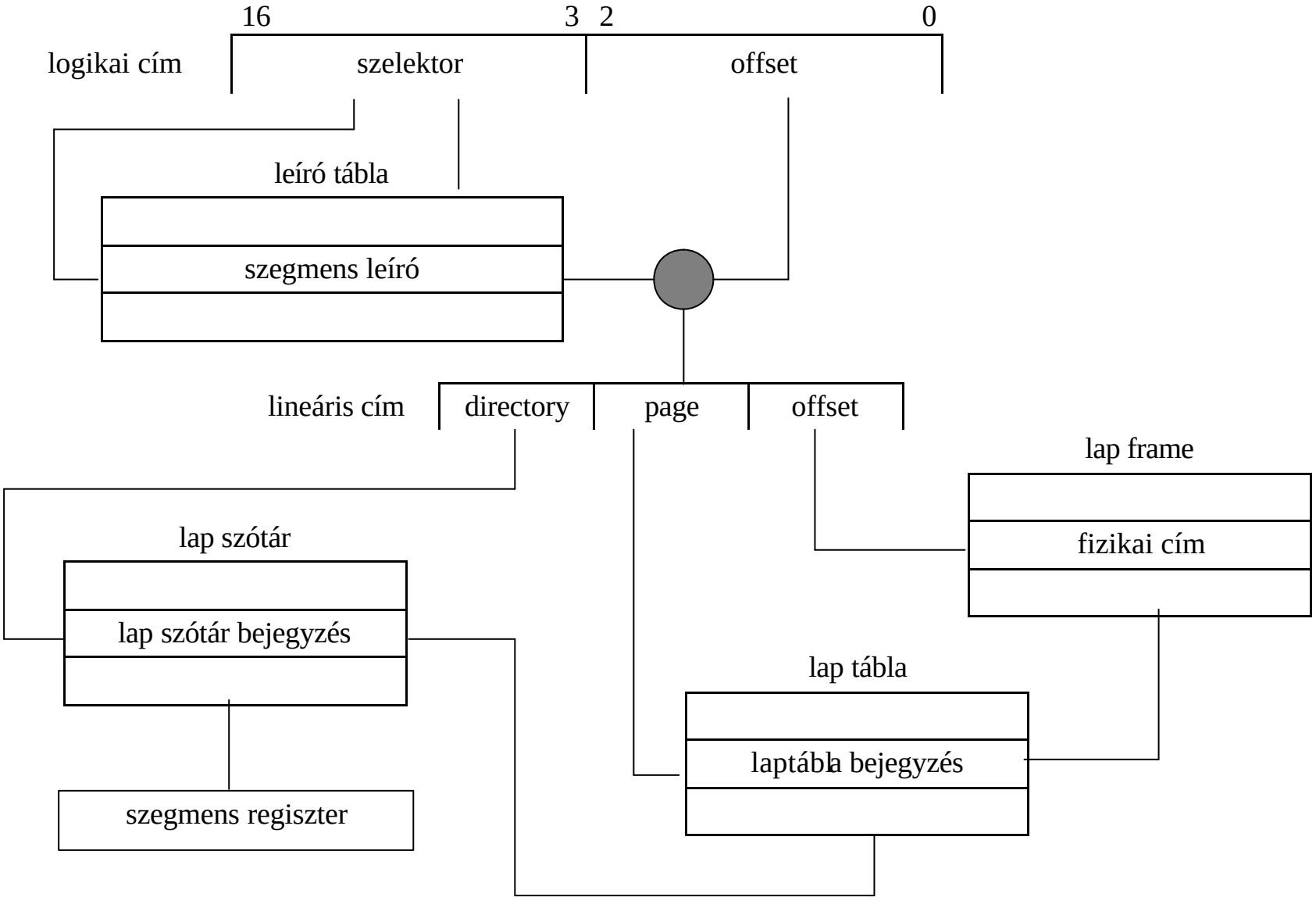
- **ötlet**: a lapozás a külsô fragmentációt, a szegmentálás a belsô fragmentációt csökkentheti!
- Pl. INTEL (>3)86 (Az OS/2 operációs rendszer már ki is használta)
  - Egy processzus által használható szegmensek maximális száma: 16K (!)
  - Egy szegmens mérete: 4 GB.
  - Lapméret: 4K=4096 bájt.
  - A szegmensek egyik fele privát, ezek címét (adatait) az LDT (Local Descriptor Table) tartalmazza
  - A többi (az összes processzusok által) közösen használt szegmens, ezek címét a GDT (Global Descriptor Table) tartalmazza.
  - Mindkét táblában egy-egy bejegyzés 8 byte, az adott szegmens leírója (kezdôcím és hossz).
  - Logikai cím: <szelektor, offset>, ahol az offset egy 32 bites érték, a szelektor


alakú, ahol s: szegmens szám, g: GDT, vagy LDT, p: protection (védelem) jelzése.

- A processzor 6 szegmens regisztere egy-egy szegmens egyidejû gyors megcímzését teszi lehetôvé.
- 8 db 8-bájtos mikroprogram regiszter (cache) a megfelelô LDT/GDT bejegyzések tárolására.
- lineáris cím: (ellenôrzések - limit - után) 32 bit, amit fizikai címmé lehet konvertálni.
- lapméret=4K $\Rightarrow$ 1M elemû laptábla (4MB !) $\Rightarrow$ jobb megoldás a 2 szintû laptábla!



- INTEL (>3)86 címképzési séma



- Szempontok a memória management stratégiák összehasonlításához
  - A legerôsebben meghatározó tényezô a hardver támogatás. (A CPU által generált minden egyes logikai címet ellenôrizni kell (legalitás) és le kell képezni valamilyen fizikai címmé. Az ellenôrzés nem implementálható (efficiens) módon a szoftverben.
  - A tárgyalt memória management algoritmusok (folytonos allokálás, lapozás, szegmentáció, szegmentáció és lapozás) sok vonatkozásban különbözô értékeket mutat. Néhány ilyen vonatkozás:
  - Hardver támogatás: Az egyszerû és multipartíciós sémákhoz elég egy bázis regiszter, vagy egy bázis/limit regiszter pár. A lapozás és szegmentáció leképezô táblázatokat is igényel.
  - Teljesítmény: Az algoritmus bonyolódásával a végrehajtási idô is megnô.
  - Fragmentáció: A multiprogramozás szintjének emelésével processzor kihasználtság nôhet, de közben memóriát veszítünk el. (fix partíciók: belsô-, változó partíciók: külsô fragmentáció).
  - Relokáció: a program (logikai címtartományának) eltolása. Tömörítés és dinamikus áthelyezés.
  - Csere (Swapping): kiegészítô intézkedés arra az esetre, ha a processzus által elfoglalt fizikai memóriát fel kell adni.
  - Megosztás (Sharing): a hatékonyságot növelhetik a több processzus által közösen használt kód és adatok.
  - Védelem (Protection): a processzusok alaphelyzetben csak a hozzájuk rendelt címtartományt használhatják. (A megosztás természetes velejárója.)

# 9. Virtuális memória kezelés

- Háttér
- Igény szerinti (kényszer) lapozás
- A kényszer lapozás teljesítménye
- Laphelyettesítési algoritmusok
- Frame-k allokálása
- Vergôdés (csapkodás, thrashing)
- Kényszer szegmentálás

# • Háttér

- Az elôzô fejezetben tárgyalt memória kezelési technikák csak érintôlegesen tartalmazzák azt az esetet, amikor a processzus logikai címtartománya ténylegesen nagyobb, mint a fizikai címtartomány. (Az owerlay és a dinamikus betöltés segíthetik ennek a problémának a megválaszolását, de általában elôzetes átgondolást és külön kódolási erôfeszítéseket igényelnek a programozótól.) Ha nem akarjuk a programozót terhelni, akkor korlátozhatjuk a végrehajtható program méretét a fizikai memória méretére, de ez elég szerencsétlen megoldás. Ugyanis:
  - A programok gyakran tartalmaznak olyan (kód)részeket amelyek rendkívüli hibákat/eseteket kezelnek. Statisztikailag tekintve ezek olyan ritkák, hogy ez a kódrész szinte sohasem hajtódik végre.
  - Tömböknek, táblázatoknak, listáknak sokszor olyan sok memóriát allokálnak, amennyire a konkrét esetek nagyrészében nincs szükség.
  - A program bizonyos ágai, szolgáltatásai rendkívül ritkán aktivizálódnak. (Pl. USA költségvetés egyenlege.)
- Az a lehetôség, hogy egy olyan program végrehajtása is megkezdhetô/folytatható, amelynek kódja nincs teljes terjedelmében az operatív memóriában több elônnyel jár:
  - Nem korlát többé a fizikai memória mérete. A programozó nem veszôdik az overlay megtervezésével.
  - Egyszerre több program aktív kódrésze lehet benn a memóriában, ami jobb CPU kiszolgálást eredményezhet. (Közben nem növekszik a válaszadási és végrehajtási idô!)
  - Kevesebb I/O tevékenység szükséges a Swapping-hez. (A futási sebesség nô!)
- **A virtuális memória koncepciója a felhasználó/programozó memória szemléletének teljes szeparálását jelenti a fizikai memóriától.**
- **Elsô közelítésben azt az esetet vizsgáljuk, amikor a futó processzuskoz fizikai lapoknak (frame-knek) egy rögzített készlete tartozik. ( a processzusok nem igényelhetik egy másik processzus fizikai lapjait).**

# • Igény szerinti (kényszer) lapozás

- Csak akkor töltsünk be egy lapot a memóriába, ha szükséges
  - Kevesebb I/O szükséges
  - Kevesebb memória szükséges
  - Gyorsabb a válaszidô
  - Több felhasználó jut lehetôséghez
- Szükséges egy lap, ha egy aktuális (logikai) címhivatkozás rá utal.
  - Érvénytelen címhivatkozás ⇒ abortálás
  - A lap nincs a memóriában ⇒ be kell tölteni a memóriába
- Validációs (*valid/invalid*) bit (*v*) szerepe a laptáblában: [*v*=0] ⇒ [page fault]
- Page Fault (laphiba)
  - Az elsô hivatkozás a lapra biztosan laphibát okoz.
  - Az operációs rendszer eldönti, hogy a hivatkozás
  - maga is hibás-e (abortálás), vagy
  - a lap nincs a memóriába betöltve.
  - Üres keret (frame) keresés.
  - Lap betöltése a keretbe.
  - A megfelelô táblaelemek beállítása (*v*=1, stb...)
  - A hivatkozott (gépi) utasítás végrehajtásának folytatása (restart).
- Mi történik, ha egy hiányzó lap betöltéséhez nincs szabad (üres) frame a memóriában?

- Algoritmusok az “áldozat” lap kiválasztására és hatásuk az átbocsátó képességre (cél: a page faultok számának minimalizálása)
- A kényszer-lapozás teljesítô képessége
  - laphiba (page fault) arány: $0 \le p \le 1$ ($p=0$: nincs laphiba)
- A “módosított” (*dirty*) bit szerepe a lapcserére fordátott idô redukálásában
- Effektív elérési idô (EAT: Effective Access Time)

  EAT= (1-p) ∗ memória elérési idô +
  + p( a laphibával kapcsolatos póttevékenység•
  + [az áldozat-lap kimentési ideje]
  + a hivatkozott lap betöltési ideje
  + újraindítási idô)
- Példa:
  - memória elérési idô (laphiba kizárásával): 1 μsec
  - 50%-ban lesz egy áldozat lap módosított, ezért kimentendô
  - Lapmentési/betöltési idô: 10 msec = 10000 μsec
  - **EAT= (1-p)* 1 + p* (15000) » 1+15000 p** (μsec-ben kifejezve)
  - Az EAT-t a lapmentési/ betöltési idô domináns módon meghatározza!

• overhead

- **Laphelyettesítô (áthelyezô - replacement) algoritmusok (stratégia, politika)**
  - **Fô cél: a laphibák számának minimalizálása.**
- **FIFO (First In First Out) algoritmus**
- Az áldozatot (azaz a memóriát elhagyni kényszerülô lapot) a lapok fizikai sorrendjének megfelelôen jelöljük ki!
- Elôny: csekély hardver támogatás szükséges!
- **Referencia sztring: 1,2,3,4,1,2,5,1,2,3,4,5**
- 3 frame esete:

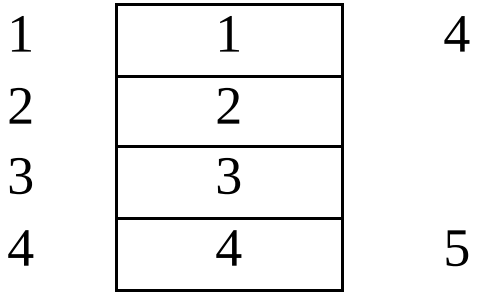
| | | | | |
|---|---|---|---|---|
| 1 | 1 | 4 | 5 | |
| 2 | 2 | 1 | 3 | 9 page fault |
| 3 | 3 | 2 | 4 | |

- 4 frame esete:

| | | | | |
|---|---|---|---|---|
| 1 | 1 | 5 | 5 | |
| 2 | 2 | 1 | 4 | 10 page fault(!) |
| 3 | 3 | 2 | | |
| 4 | 4 | 3 | | |

- Béládi - anomália: több frame $\neq\Longrightarrow$ kevesebb laphiba!

- **Optimális algoritmus (OPT/MIN)**
  - **Létezése:** véges probléma, létezik minimum.
  - Azt a lapot helyettesítsük, amelyiket a leghosszabb ideig nélkülözni lehet (mert nem lesz rá szükség!).
  - 4 keretes példa:
  - **Referencia sztring: 1,2,3,4,1,2,5,1,2,3,4,5**




6 page faults

  - Hogyan lehet implementálni? (Tisztán nem lehet!)
    - Esetleg predikcióval megkísérelhetô az implementáció.
  - Hasonlóság az SJF processzus ütemezéshez.
  - Elméleti jelentôsége: segítségével az egyes algoritmusok összevethetôk, értékelhetôk.

- **Legrégebben használt (LRU Least Recently Used) algoritmus**
  - Azt a lapot helyettesítsük, amelyiket a leghosszabb ideje nem használunk.
  - 4 keretes példa:
  - **Referencia sztring: 1,2,3,4,1,2,5,1,2,3,4,5**

| | | | |
|---|---|---|---|
| 1 | 1 | | 5 |
| 2 | 2 | | |
| 3 | 3 | 5 | 4 |
| 4 | 4 | 3 | |

8 page faults

  - Counter (számlálós) implementáció:
    - Globális számláló (***S***): minden memória hivatkozáskor 1-gyel nô.
    - Minden ***f*** fizikai laphoz tartozik egy $S_f$ lokális számláló, amely minden ***f***-re történô hivatkozáskor felveszi ***S*** értékét: $S_f=S$.
    - A lokális számlálók értékének összehasonlításával eldönthetô, melyik lap legyen az áldozat?
  - Verem implementáció:
    - A fizikai lapsorszámokat egy verembe helyezzük úgy, hogy a legutoljára használt legyen a tetején (ekkor a legrégebben használt lap a verem alján van).
    - Elônyei, és hátrányai.

- **AzLRU algoritmust approximáló algoritmusok**
  - **Miért lehet jó egy (rossz) approximáló algoritmus?**
  - **Referencia bit**
    - Minden fizikai laphoz egy bitet rendelünk, amelynek kezdeti értéke: 0.
    - Ha a lapra hivatkozás történik, akkor a bit értéke 1 lesz.
    - Lapcserénél az áldozatok a 0 referenciájú lapok közül kerülnek ki (mindegy milyen sorrendben)
    - Hardver támogatás jelentôsége!
  - **Kiterjesztett referencia bit: referncia bájt.**
    - Minden ***f*** fizikai laphoz egy bájtot $R_f$ rendelünk, amelynek kezdeti értéke: **X`00`**.
    - Szabályos idôközönként (pl. 100 msec: timer megszakítás! ) minden $R_f$ egy bittel (logikailag) jobbra tolódik.
    - Ha a lapra hivatkozás történik, akkor a bájt értéke $R_f = R_f$ **OR X`80`** lesz.
    - Lapcserénél az áldozatok a legrégebbi referenciát tükrözô lapok közül kerülnek ki mindegy milyen sorrendben (véges, diszkrét emlékezet!).

- **Második esély algoritmus**
  - A referencia bitet is tartalmazó lapokat ciklikusan vizsgáljuk.
  - Ha a referencia bit 0, akkor a lap áldozat lesz.
  - Ha a referencia bit 1, akkor lenullázzuk, de a lapot benn hagyjuk (second chance = második esély).
  - Helyette következô lapot vizsgáljuk meg az elôbbi elveknek megfelelôen.
  - Speciális eset adódik, ha minden bit = 1. (vö.: FIFO)

- **Kiterjesztett második esély algoritmus**
  - A referencia és a módosítás biteket is használja
  - Négy lehetôség: (0,0), (0,1), (1,0), (1,1) jelentése. Osztályok
  - Az elsô nem üres osztály elsô elemét távolítjuk el.
  - Alkalmazás: Apple Mcintosh.

- **Számláló (counter) algoritmusok**
  - Minden fizikai laphoz tartozik egy számláló (harver implementáció?), amely a lapra történt hivatkozások számát adja meg.
  - **LFU (Least Frequently Used) algoritmus**
    - Az lesz az áldozat amelyikre a legkevesebb referencia történt.
    - Probléma: ha pl. egy inicializáló lapra kezdetben sok hivatkozás történt, mindig benn marad (,mert nagy a referencia száma), pedig többé nincs szükség rá!
  - **MFU (Most Frequently Used) algoritmus**
    - Az lesz az áldozat amelyikre a legtöbb referencia történt.
  - Mindkét esetben az implementáció nagyon költséges!

- **Lap pufferelés**
  - pool = frame puffer
  - A beolvasás / kiírás frame pufferbe történik (így page fault esetén a processzus továbbindulhat anélkül, hogy várna az áldozat kiírására).
  - Más: ha a lapozó eszköz szabad, kiír egy módosított lapot és a dirty bitet nullázza.
  - (VAX/VMS – nél alkalmazzák FIFO helyettesítési algoritmussal.)

# • Frame-k allokálása

- Alapkérdés: hány frame tartozzon egy-egy processzushoz?
- Minden processzusnak szüksége van minimális számú lapra, hogy futni tudjon. (Ez a szám architektúra függô!)
  - Adott esetben egyetlen gépi utasítás végrehajtásához is több lapra lehet szükség.
  - Pl. IBM370 MVC utasítás (move) végrehajtása 6 lapot is igényelhet!
- Allokációs sémák:
  - Fix allokáció:
    - Egyelô leosztás, minden processzus ugyanannyi frame-t kap.
    - A processzus hosszától függô, arányos leosztás.
  - Prioritásos allokáció:
    - A kapott lapok száma a processzus prioritásától is függ.
    - A kettô keveréke jó megoldás lehet.
    - Egy processzus – page fault esetén – kaphat frame-t egy alacsonyabb prioritású processzus készletébôl.
- Globális – lokális allokáció:
  - Globális: új laphoz egy közös készletbôl lehet.
  - Lokális: minden processzus csak saját, kezdetben hozzárendelt frame készletét használhatja.

# • Vergôdés (csapkodás, thrashing)

- Ha egy processzus számára nem áll elég lap rendelkezésre, akkor a **page fault arány** nagyon nagy lesz.
- Következmények:
  - Alacsony CPU kihasználás.
  - Az operációs rendszer úgy gondolja, hogy növelnie kell a multiprogramozás fokát.
  - Új processzust indít.
- Vergôdés – ami ezután jön: a rendszer csak lapok kimentésével és betöltésével van elfoglalva.
- Lokalitás modell:
  - Lokalitás: azon lapok halmaza, amelyekre a processzusnak nagyjából egyszerre van szüksége.
  - A processzus vándorol az egyik lokalitásból a másikba.
  - A lokalitások egymást átfedhetik.
  - Vergôdés akkor lép fel, ha

$$S \text{ lokalitások mérete} > \text{teljes memória méret}$$

- Working-set (munkahalmaz, munkakészlet) modell

- Kényszer szegmentálás

# 10. Fájl rendszer interfész

- Fájl koncepció
- Elérési módok
- Könyvtár szerkezet
- Védelem
- Konzisztencia szemantika

# • Fájl koncepció

- A számítógépek az adatokat különbözô fizikai háttértárakon tárolhatják (pl. mágnes lemez, szalag kártya, optikai lemez, stb.). A számítógép kényelmes használhatósága érdekében az operációs rendszerek egy egységes logikai szemléletet vezetnek be az adattárolásra és adattárakra: az operációs rendszer elvonatkoztatva a tároló eszköz és a tárolt adat fizikai tulajdonságaitól, egy logikai tároló egységet (adatállomány – **fájl** – *file*) használ.
- A fájlokat az operációs rendszer képezi le a konkrét fizikai tároló berendezésre.
- A fájlokat tartalmazó fizikai tároló berendezések általában nem törlôdôek (de ez nem kritérium).
  - Pl. "képernyô" fájlok egyes rendszerekben.
- Felhasználói szemszögbôl: a fájl összetartozó adatok egy kollekciója, amelyeket egy másodlagos tárban tárolunk. A fájl a felhasználó számára az adattárolás legkisebb allokációs egysége: felhasználói adatot a háttértáron csak valamilyen fájlban tárolhatunk.
- Az operációs rendszer támogatást nyújthat a fájl *tartalmának kezelésében*, a fájl *szerkezetének* (adatszerkezet) *létrehozásában*:
  - Fájltípusok: adat – program, folyam (stream) – rekord.

# • Fájl attribútumok

- Az operációs rendszer **a fájl kezeléséhez szükséges információk**at (attribútumokat) is tárolja a háttértárban.
  - (Pl. egy elkülönített helyen, a szótárban, fájl jegyzékben – **directory**-ban .)
- **Név** – a fájl azonosítására szolgál (szimbolikus fájlnév, név hossza, osztott nevek, kisbetûk/nagybetûk szerepe)
- **Típus** – az egyes típusokhoz (text, script, bináris program, kép, stb) különbözô operációs rendszer szolgáltatásokat lehet rendelni.
- **Lokáció** – a fájl fizikai elhelyezkedésével kapcsolatos adatok (háttértár címek)
- **Méret** – aktuális / maximális méret
- **Védelem** – a fájlhoz történô hozzáférés vezérlése.
- **Idô, dátum, felhasználó azonosítás**.

# • Fájl mûveletek

- A fájl mûveleteket, mint egy **absztrakt adattípuson végezhetô mûveletek**et kell tekintenünk.
- Az operációs rendszer ezeket **rendszer-hívások** segítségével teszi elérhetôvé.
- Alapmûveletek:
  1. **Létrehozás** (create) – terület allokálás + új bejegyzés a directoryba.
  2. **Írás** (write) – write pointer szerepe
  3. **Olvasás** (read) – kurrens pozíció szerepe
  4. **Újrapozicionálás** (repositioning)
  5. **Törlés** (deleting)
  6. **Csonkítás** (truncate)
- További mûveletek: bôvítés (append), átnevezés (rename), másolás (copy), az attribútumok megváltoztatása.
- **Open/close** operációk szükségessége és szerepe:
  - A directory-hoz fordulások számának csökkentése *(open–file table).*
  - Multiuseres környezet támogatása, fájlvédelem *(open–count).*
  - A fájl elhelyezkedésével kapcsolatos információk memóriában tartása *(file pointer, memory mapping).*

- **Fájl típusok**

| fájl típus | kiterjesztés | funkció |
|---|---|---|
| végrehajtható (executable) | .exe .com .bin vagy "üres" | futtatható, gépi kódú program |
| tárgy (object) | .obj .o | lefordított, áthelyezhetô bináris kód |
| forrás (source) | .c .p. .pas .asm | program forráskódja |
| parancs (batch) | .bat .sh .cmd vagy "üres" | parancs fájl |
| szöveg (text) | .txt .doc | szöveges adat, dokumentum |
| szövegszerkesztô (WP) | .doc .rtf .tex .ltn .wp | változatos szövegszerkesztô formátumok |
| könyvtár (library) | .lib .dll | szubrutin könyvtárak |
| nyomtatás/megjelenítés (print/view) | .ps .pdf .gif .prn | megjelenítô berendezések saját fájlformátumai |
| archívum (archieve) | .arc .zip .tar | kapcsolt fájlok egy fájlba szervezve, esetleg tömörítve |
| alkalmazói rendszerek (appl. syst.) | ??? | |

# • Fájl szerkezetek

- A fájl típus indikál(hat)ja a fájl (belsô) szerkezetét. (A kreátornak és a fájlt feldolgozó programnak ezt természetesen ismernie kell.)
- Pl. forrásnyelvi-, tárgynyelvi- (object), ill. végrehajtható (bináris) programokat tartalmazó állományok szerkezete kötött.
- Kérdés: Mennyire ismerje az operációs rendszer az egyes fájlok belsô szerkezetét?
- Egy szerkezetet (végrehajtható program) biztosan ismernie kell!
- Egyébként olyan mértékben ismeri, amennyire a fájlkezeléshez "központi támogatást" akar nyújtani.
- Pl.:
  - Néhány régi nagy rendszer (IBM DOS 26.2; IBM OS/VS, stb.) támogatta a változatos blokkolási és fájl szervezési módokat (FIXUNB, FIXBLK, VARUNB, VARBLK, UNDEF; VSAM, VDAM, ISAM).
  - A UNIX minden fájlt bájtok egyszerû sorozatának tekint, ez a feldolgozó programoknak nagy flexibilitást, de minimális támogatást jelent. Szolgáltat ugyanakkor egy mechanizmust (rendszer hívás) a fájl típus meghatározására. (Vö.: *file* parancs, *magic number*)
  - VAX/VMS három szerkezetet is támogat!
- Az operációs rendszer által nyújtott szerény támogatást kiegészíthetik a programozási környezetekben / futtató rendszerekben implementált szerkezet-kezelési lehetôségek.
  - (Pl. adatbázis kezelôk.)

- Elérési módok
  - Szekvenciális elérési mód
    - read next
    - write next
    - reset
    - no *read* after last *write*
    - (rewrite)
  - Direkt elérési mód
    - read n
    - write n
    - position to n (pozicionálás az n-edik rekordra)
    - read next
    - write next
    - rewrite n

n= a relatív blokk–sorszám

# • Könyvtár szerkezet

- Bejegyzések (node) együttese, amely információt tárol a fájlokról
- A könyvtárszerkezet és a fájlok is a lemezen a háttértáron tárolódnak.
  - Kapcsolódó fogalmak: kötet (volume), partíció (partition), VTOC
- Egy directory-ban (fájl-jegyzékben) tárolt információ:
- (pl.) név, típus, cím, aktuális hossz, maximális hossz, legutóbbi elérés (access) idôpontja, legutóbbi módosítás (update) idôpontja, tulajdonos azonosító (owner ID), védelemmel kapcsolatos adatok.
- Egy directory-val kapcsolatban végrehajtható (absztrakt alap-) mûveletek:
- Fájl keresés (file search), fájl létrehozás (create), fájl törlés (delete), directory listázás (dir, list, ls), fájl átnevezés (rename), a fájl rendszer pásztázása (traverse), bejárása.
- A könyvtárszervezéssel szemben támasztott elvárások:
- Hatékonyság: minden fájl könnyen visszakereshetô legyen.
- Névadás (naming):
  - Két user használhassa ugyanazt a nevet más fájlokhoz.
  - Ugyanannak a fájlnak lehessenek különbözô azonosítói.
- Csoportosítás (grouping)
  - Fájlok csoportosítása tulajdonságaik alapján. (com, bat, pss, nfs ntfs, ... dtv)

- Egyszintû directory
  - – Az összes felhasználó állományai egyetlen *jegyzéket* (directory) alkotnak.
  - – Névadási- (névütközési-) és csoportosítási problémák
- Kétszintû directory
  - – Egy-egy felhasználó állományai elkülönített jegyzéket (directory) alkotnak.
  - – Szintek: MFD - UFD (master/user file directory)
  - – Megoldja a névütközés problémáját, de nincs csoportosítás.
  - – A felhasználók nem tudnak kooperálni (nehéz egymás állományait elérni).
  - – Új fogalom jelenik meg: *elérési út* (path)
  - – A rendszer fájlok használatának problémája
    - dedikált kópia?
    - egy (több) speciális (kitüntetett) UFD-ben lehet (pl.) elhelyezni. (Primitív csoportosítás.)
  - *Keresési út* (search path) fogalma.
- Fa-szerkezetû directory
  - Egy fájljegyzék bizonyos elemei lehetnek újabb fájljegyzékek, így fájljegyzékeknek egy hierarchikus rendszere jön létre.
  - Egy fájljegyzék által (fájlként) tartalmazott fájljegyzék = **al**fájljegyzék (subdirectory)
    - A jegyzékben minden esetben egy speciális bit jelezheti, hogy fájlról, vagy aljegyzékrôl van-e szó.
  - Fájljegyzéket létrehozni és törölni speciális rendszer-hívásokkal lehet.
    - Pl. *make:* mkdir(); *remove:* rmdir().
  - Kurrens (current) directory, kurrens directory váltás: cd().
  - Abszolút- és relatív elérési út, keresési utak, pásztázás (*traverse*).
  - Megoldja a névütközés problémáját, és lehet csoportosítani.
    - Az elérési utak megadása sokszor körülményes. Egy megoldás: *DeskTop file* (Mac).

- Aciklikus gráf - szerkezetû directory
  - Egy aljegyzék (fájl) osztott használatának problémája: több felhasználó/alkalmazás szeretné a saját directory rendszerében látni.
  - Megoldás: egy típusú directory bejegyzésnek, valamely fájlra, vagy aljegyzékre mutató kapcsolónak (*symbolic **link***) a bevezetése. Logikailag: alias-képzés!
    - Vö: link, ln rendszerhívás (UNIX), create link (NT).
    - Technikailag lehetséges lenne a mutató (link) helyett a teljes directory bejegyzést **duplikálni:**
      - (DE: konzisztencia! )
  - *Hard* link és *soft* link.
  - Ha a kapcsolókat követve nem juthatunk vissza egy olyan bejegyzéshez, amelyet már korábban elértünk, akkor a directory szerkezete aciklikus gráf.
  - Nehéz detektálni a ciklust.
  - Problémák: pásztázás, törlés.
  - Törlési politikák: Mi történjék a fájlra mutató linkekkel, ha töröljük a fájlt?
    - …, illetve mi legyen a fájllal, ha töröljük a linket?
  - "Függô" link megengedett: megmarad a link, ha töröljük a fájlt (vagy a fájl nem érhetô el).
    - Pl. UNIX soft link, lokális lemezek mountolása egy fájl rendszerbe. (Ez jól is jöhet!)
  - "Függô" link nem megengedett.
    - Megoldások:
      - fájl-referencia listák
      - referencia számlálók

- Általános gráf - szerkezetû directory
  - Hogyan lehet a ciklusmentességet biztosítani?
  - Vannak algoritmusok, de ezek "költségesek".
  - Ha a szerkezet ciklust (önreferencia) tartalmaz, akkor a keresés / pásztázás körülményes lesz.
  - Pl. egy fájl a körbe mutató linkek miatt nem lesz simán törölhetô. Adott esetben "hulladékgyûjtés" (garbage collection) is szükséges lehet.

# • Védelem

- A számítógépes rendszerben tárolt adatokat védeni kell
  - a fizikai sérülésektôl (*reliability, integrity*), és
  - az illetéktelen hozzáféréstôl (*protection, security*).
- A fizikai sérülések ellen a gondos kezelés mellett meghatározott stratégia szerint készülô biztonsági másolatokkal védekeznek.
- Hozzáférési operáció (alap-) típusok:
  - **Read** (olvasás a fájlból)
  - **Write** (írás a fájlba)
  - **Execute** (a fájl betöltése a memóriába és végrehajtása, futtatása)
  - **Append** (új adat hozzáírása a fájl végéhez)
  - **Delete** (a fájl törlése és az általa elfoglalt hely felszabadítása)
  - **List** (a fájl nevének és attribútumainak listázása)
  - Más operációkat (**rename, copy, edit**) is lehetne még tekinteni, ezek visszavezethetôk alap- operációkra.

A hozzáférési operációk hátterében az operációs rendszer, vagy egyes felhasználók által kezdeményezett processzusok állnak. Ha tehát egy-egy adott fájlhoz minden egyes processzusra (userre) megmondjuk, hogy melyik alapoperációt alkalmazhatja a fájlra, akkor a hozzáférés teljesen szabályozott lesz. Ez a

- Hozzáférési lista (Access Control List, ACL)

Az ACL megnöveli a fájlbejegyzés méretét, változó hosszúságú lesz, nehezen kezelhetô (de alkalmazzák!).

- Megoldás csoportosítással
  - a védelem szempontjából az összes felhasználó három kategóriába tartozhat:
    - ô maga a fájl tulajdonosa (**owner**),
    - tagja valamilyen jól definiált csoportnak (**group**, több csoport is lehet),
    - beletartozik az összes felhasználók csoportjába (**world**).
- a hozzáférési alapoperációk közül is hármat emelünk ki (a többit ezek mögé illeszthetjük)
  - read (r), write (w), execute (x)
- A hozzáférési lista ekkor leegyszerûsödik: egy-egy fájlra azt kell megmondani, hogy az egyes kategóriák mely operációkat hajthatnak végre a fájlon.
- UNIX példa: (az egyes oszlopok jelentése a `man ls` paranccsal lekérdezhetô!)

```
fazekas@paris [~] >>ls -l
total 354
drwxr-xr-x     8 fazekas       pcpc        512    Sep  7 14:50      ./
drwxr-xr-x 159 root      root       4096     Nov 23 14:03      ../
-rw-------       1     fazekas     pcpc        496    Sep  7 14:50      .Xauthority
-rwxr-xr-x     1     fazekas     pcpc       2463    Nov 20  1998      .alias*
drwx------    37     fazekas     pcpc       1024 May  3  1998 .fm/
-rw-------       1     root          other         0      Sep 17  1996 .homedir_angelegt_am_96.09.17_12:24:47
drwx------      3     fazekas     pcpc        512    Jul 21  1997       .netscape/
drwxr-xr-x     2     fazekas     pcpc        512 Sep 17  1996        .wastebasket/
```

- Más védelmi megoldások:
  - **jelszó** (password) **a fájlhoz** (ki tudja megjegyezni?)
    - archiváló (tömörítô), workflow (office) rendszerekben alkalmazzák
  - **jelszó a könyvtárakhoz** (mindent, vagy semmit!)
    - Pl.: TOPS-20, IBM VM/CMS (minidiszk védelem!)
  - **speciális attribútumok**: "read only", "system" (MS DOS)
- A fájljegyzékek, (directory) védelme
  - Külön probléma:
    - Pl. attól még hogy egy, a jegyzékben levô fájl olvasható, nem biztos, hogy a jegyzék tartalmát lehet listázni, vagy
    - nem biztos hogy egy könyvtár kurrens könyvtárrá tehetô.
- Unix példa:
- directory esetén (ls által adott listán az elsô betû d)
    - r jelentése: a directory tartalma listázható
    - x jelentése: cd (change dir) az adott jegyzékre vonatkoztatva megengedett

# • Konzisztencia szemantika

- Multiprogramozásnál elôfordul(hat), hogy több processzus közösen használ egy-egy állományt. (konkurens hozzáférés). Eközben biztosítani kell az állományban tárolt adatok konzisztenciáját (elkerülni az összeférhetetlenséget). Ehhez egy pontos szemantika (értelmezés) szükséges, amely világossá teszi, hogy az adat milyen állapotot tükröz.
- Röviden: hogyan és mikor válik láthatóvá, elérhetôvé egy processzus számára az állományon másik processzus által okozott változás?
- Fogalom: *fájl szeansz*/ülés (*file session*)= a processzus életének az a szakasza amely a fájl megnyitásától a fájl lezárásáig terjed. (vö. *terminal session*: logintôl logout-ig terjedô idôszak!)
- UNIX fájl rendszer szemantika:
  - Egy megnyitott fájlba történô írás azonnal látszik a többi felhasználóknak is, akik a fájlt nyitva tartják.
  - Bizonyos feldolgozó programok ezt lokálisan módosíthatják (lock, dedikált kópia használata.)
  - Még olyan megosztás is van, ahol a kurrencia pointer is közös és mindkét fél által mozgatható.
- Szekció szemantika (Andrew - fájl rendszer)
  - Egy megnyitott fájlba történô írás nem látszik a többi felhasználóknak, akik a fájlt nyitva tartják.
  - Csak azok látják majd akik azután nyitották meg, amikor az író processzus lezárta.
  - E szemantika szerint a fájlnak több másolata lehet egy idôben.
- Osztott megváltoztathatatlan (*immutable*) fájl szemantika
  - Egy ilyen fájl neve védett, tartalmát nem lehet módosítani. Osztott rendszerben egyszerûen implementálható (read only)

# 11. Fájl rendszer implementáció

- Fájl rendszer struktúra
- Allokációs módszerek
- Szabad hely kezelés
- Directory implementáció
- Hatékonyság és teljesítmény
- Helyreállítás

# • Fájl rendszer struktúra

- A fájl rendszer rétegekbe (layer) szervezhetô, ezek
  - *Alkalmazói program*
    - A fájl létrehozásával, olvasásával, módosításával kapcsolatos igények forrása.
  - *Logikai fájl rendszer*
    - A fájljegyzék (directory) felhasználásával, a fájl nevébôl kiindulva ez határozza meg a szükséges információkat a szervezési modul számára. Felel a fájl védelméért.
  - *Fájl szervezési modul*
    - Ez ismeri a és implementálja a kapcsolatot a logikai és fizikai rekordok között. Ismerve a fájl szervezési módját és allokációját a logikai rekord címeket fizikai blokkokra történô hivatkozássá alakítja. Adminisztrálja a szabad helyeket.
  - *Alap fájl rendszer*
    - Ez alakítja ki és adja ki a megfelelô device drivernek a megfelelô magas szintû parancsot egy blokk írására és olvasására. (cilinder, sáv, szektor logikában "gondolkodik".)
  - *I/O vezérlés*
    - Device driverek, interrupt handlerek. ezek inputja magas szintû utasításokból (pl. "olvasd a 12. blokkot") áll, outputját alacsony szintû, hardver specifikus bitképletekbôl alkotják (, amelyeket különbözô "portokra" ír).
  - *Fizikai berendezés*
- Fájl megnyitás és lezárás:
  - rendszer és processzus szintû Open File Table, Fájl vezérlô blokk (FCB)
  - UNIX: aktív INODE tábla
- Fájl rendszer mountolás

- Allokációs módszerek

folytonos

indexelt

FAT

UNIX I-NODE

# • Szabadhely kezelés

bit térkép

láncolás

csoportosítás (grouping)

számlálás (counting)

# Directory implementáció

## szekvenciális szerkezettel
## hashing

- Hatékonyság és teljesítmény

ható tényezôk: paraméterek, allokáció, stb.

- Helyreállítás

Biztonsági másolatok készítése, stratégiák

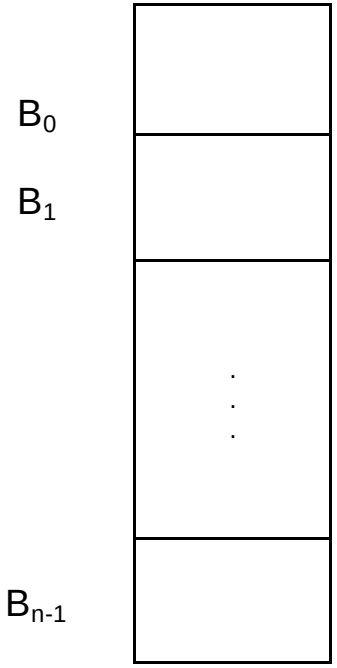
Markov megállítási probléma

# 12. Másodlagos tár szerkezet

- Diszk felépítés
- Diszk  ütemezés
- Diszk kezelés
- Swap (csere) terület kezelés
- Diszk megbízhatóság
- Stabil-tár implementáció

# • Diszk felépítés

- Logikailag a diszk blokkokból képezett lineáris tömbnek fogható fel.




- Létezik egy leképezô séma, amely a $B_i$ logikai címhez fizikai lemezcímet (cilinder, sáv, szektor) rendel.
- Legkisebb allokációs egység a blokk.
- A blokk egyben a belsô fragmentáció viszonyító mértéke.

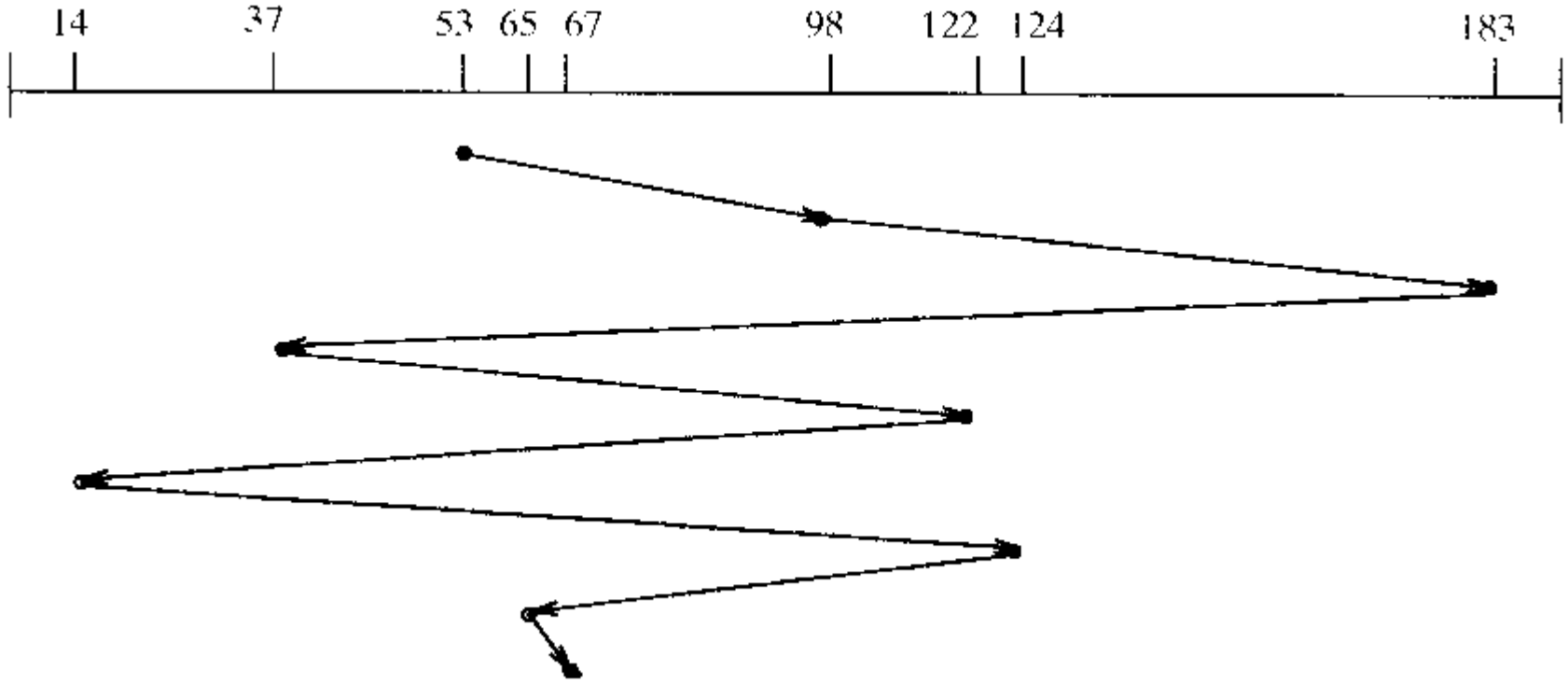
# • Diszk ütemezés

- A lemezmûvelet mindig valamelyik **sáv** valamelyik **szektor**ára történô **pozicionálás**sal kezdôdik:
  - keresési idô (**seek** time): az író/olvasó fej pozicionálása a sávra
  - várakozási idô (**latency** time): várakozás, amíg a kérdéses szektor az író/olvasó fej elé fordul
  - átviteli idô (**transfer** time): az adat írása / kiolvasása elektronikusan
- A keresési idô nagyságrendekkel lehet nagyobb, mint a másik kettô, ezért a diszk ütemezés célja a keresési idô minimalizálása.
  - A keresési idô arányos a keresési távolsággal (jó közelítéssel).
- Ütemezési algoritmusok (példákon illusztrálva).
  - Tegyük fel, hogy egy lemezegység 200 cilindert (0–199) tartalmaz, az író/olvasó fej az 53-ik cilinderen áll és hogy az I/O sorban levô **igények** rendre a

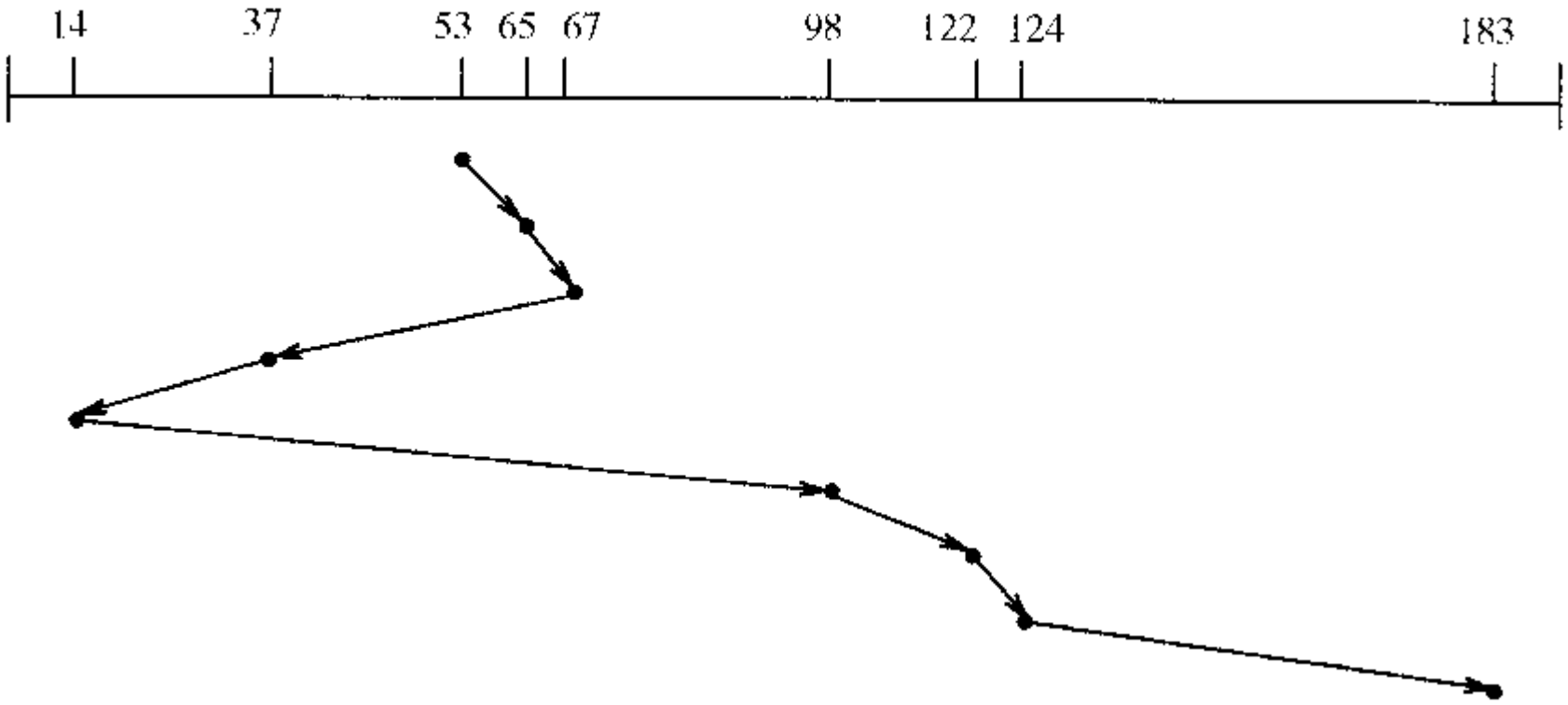
    **98, 183, 37, 122, 14, 124, 65, 67**

    sorszámú cilinderekkel kapcsolatosak.

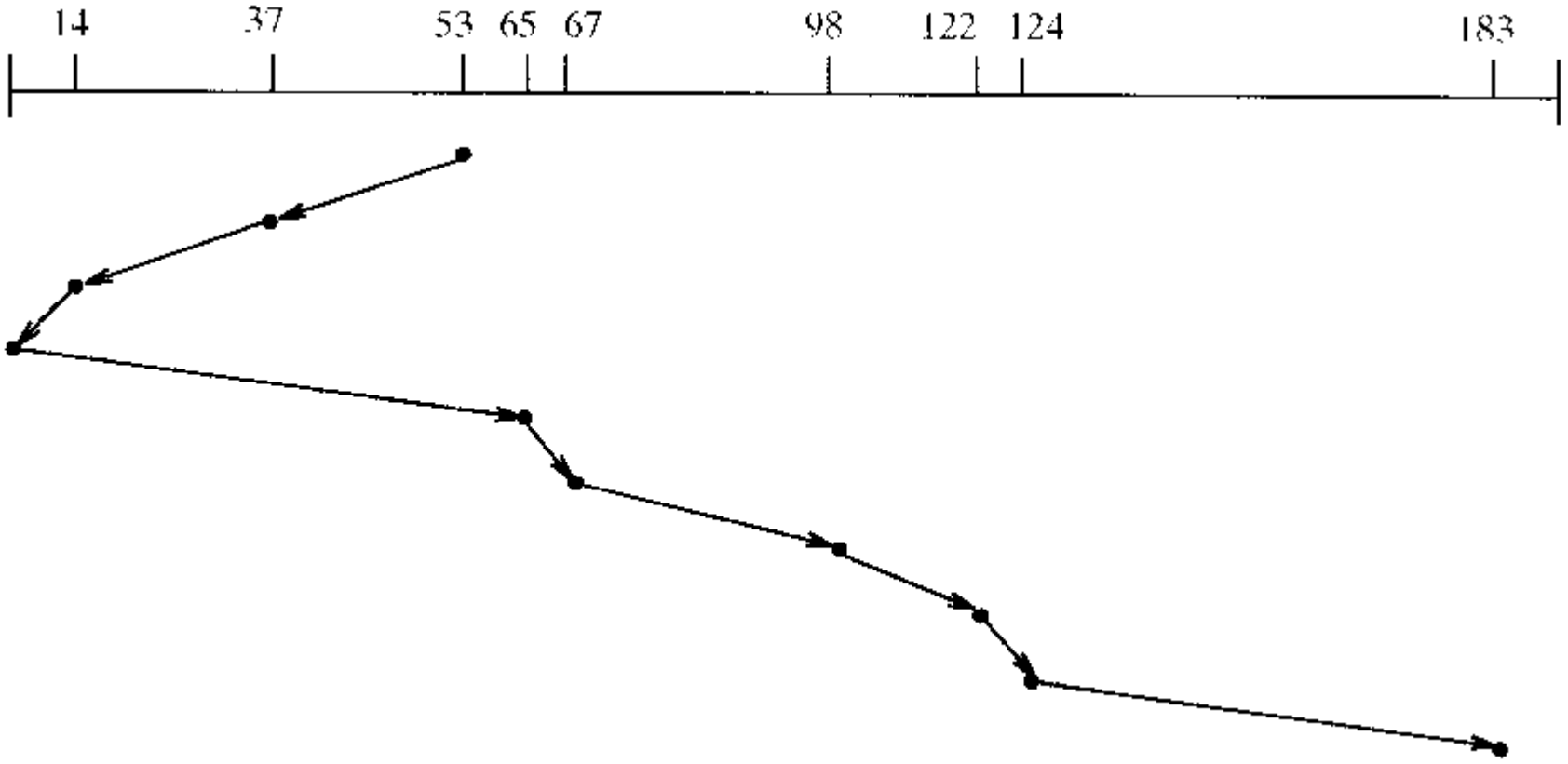
- FCFS (First Come First Served)
  - Egyszerûen implementálható, de nyilván nem szolgáltat jó átlagos kiszolgálási idôt.
  - Az író/olvasó fej ide-oda "ugrál".
  - A fej által összességében megtett "út" hosszú.
  - Minden igény elôbb-utóbb kielégül.



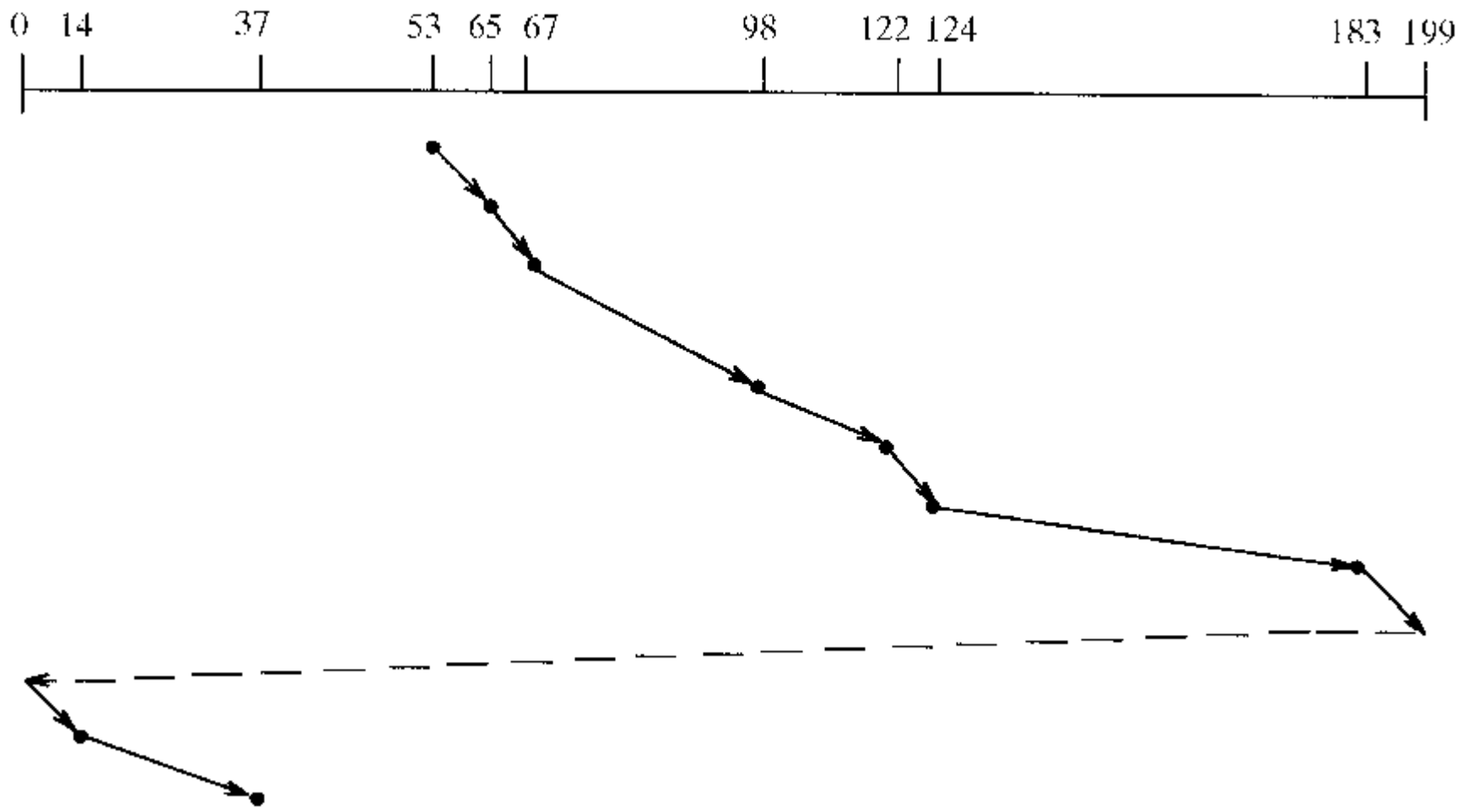
- SSTF (Shortest Seek Time First )
  - Hasonlít az SJF processzus ütemezési stratégiára (az ott optimális volt, ellenben ez itt)
  - Nem optimális: az **53, 37, 14, 65, 67, 98, 122, 124, 183** sorrend rövidebb összkiszolgálási idôt eredményez.
  - Az FCFS-nél azért kedvezôbb.
  - Éhezéshez (starvation) vezethet!



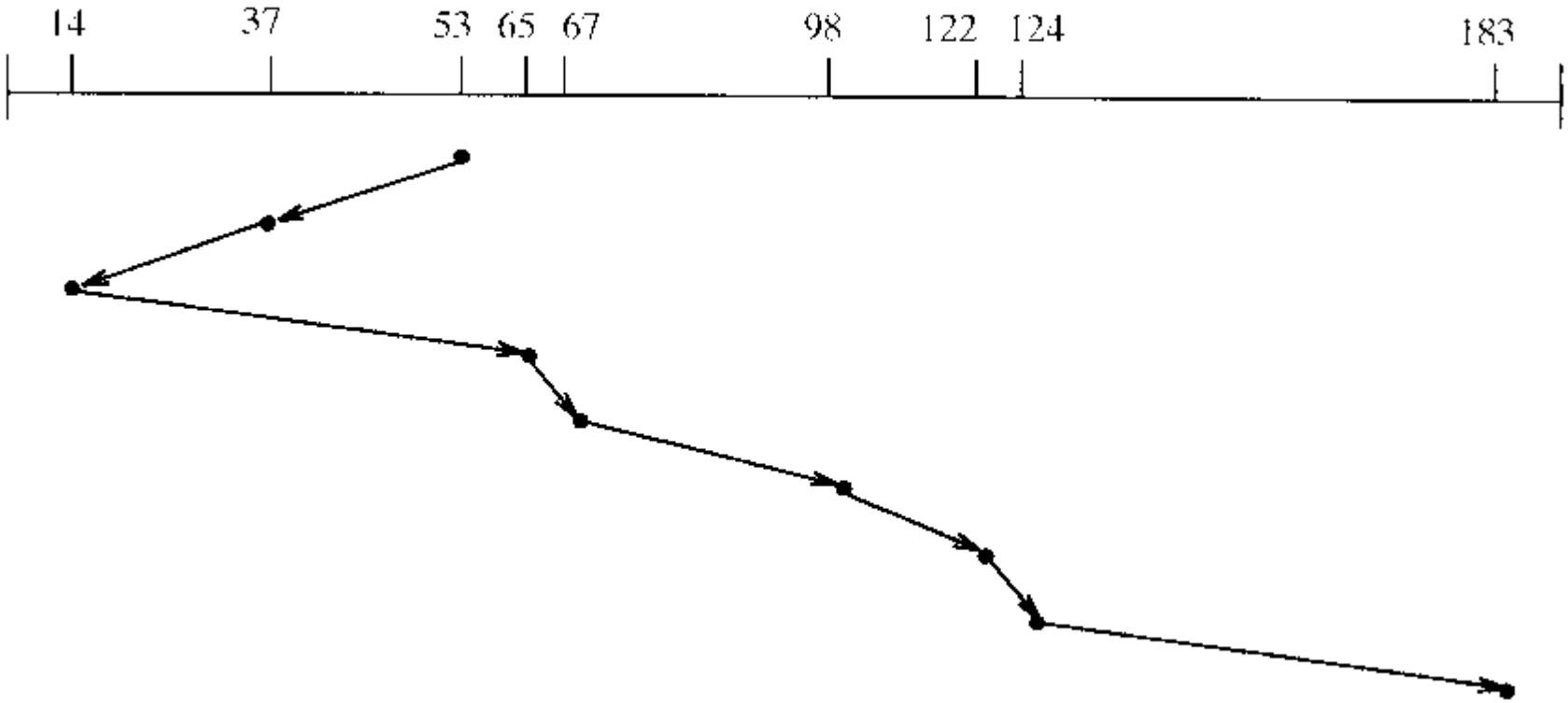
- SCAN (pásztázás)
  - Elevátor algoritmusnak is nevezik.
  - Ha a fejpozicionálási igények egyenletesen oszlanak el, akkor mindig amikor a fejmozgás irány a legszélsô cilindernél megfordul, a másik végén van a "tumultus". ? egyenetlen várakozási



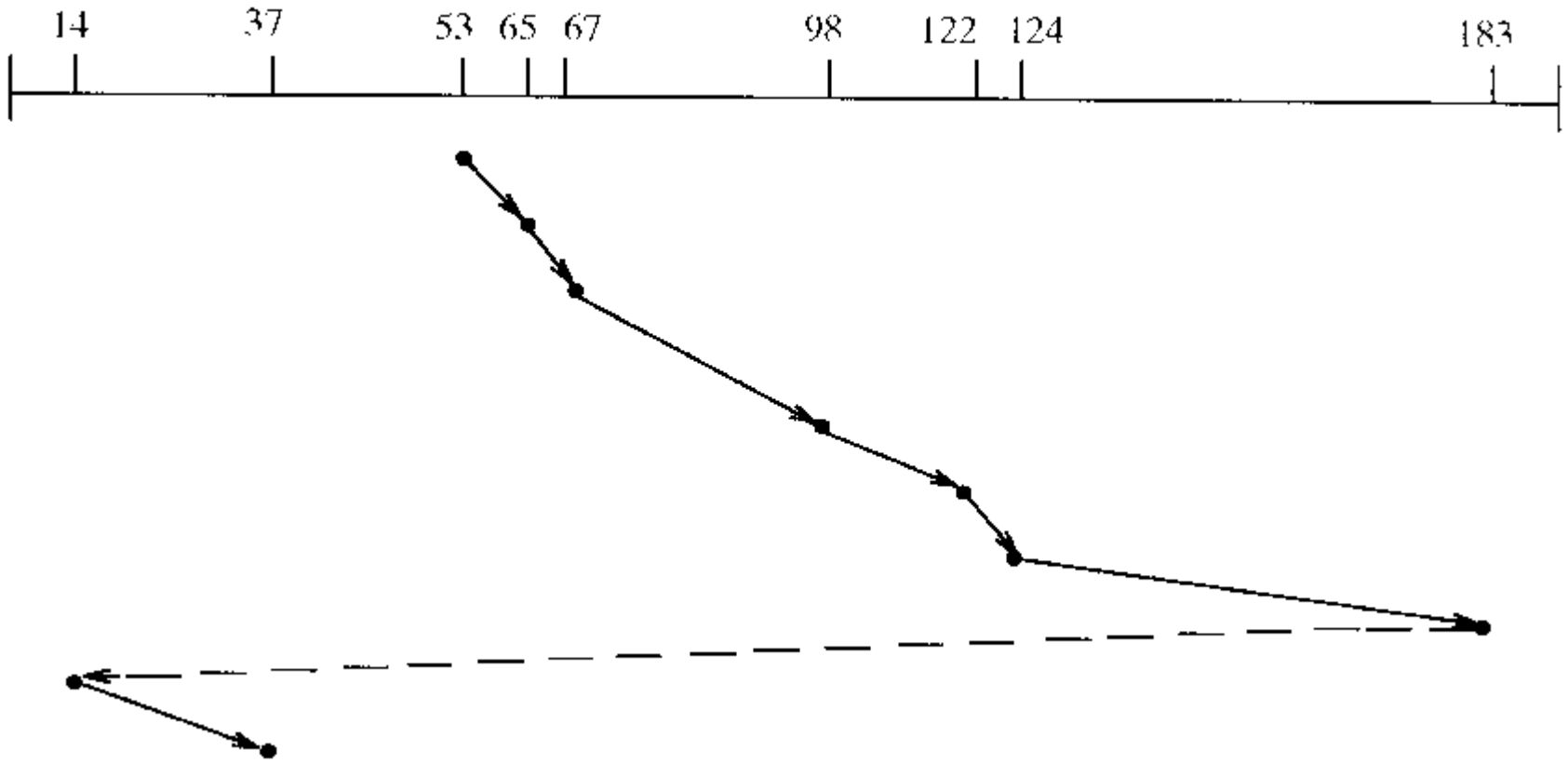
- C-SCAN (Circular SCAN)
  - Az egyenetlen várakozási idô egyenletesebbé tehetô vele.



- LOOK (for a request before moving)
  - Csak akkor megy valamelyik irányba tovább, ha tényleg van rá igény.



- C-LOOK
  - A LOOK cirkuláris változata



- A lemez - ütemezô algoritmus megválasztása

- Az SSTF algoritmus egyszerû és "csábító".
- A SCAN és C-SCAN alkalmasabbak olyan rendszerekben, ahol a lemezes adatforgalom nagyon intenzív.
- Lehetne definiálni az optimális algoritmust, de nem biztos, hogy a nagy számítási munka megtérül.
- Minden alkalmazott algoritmus végsô teljesítményét alapvetôen befolyásolja az *igények típusa és száma.*
- Pl. ha a sor általában egyelemû, akkor minden algoritmus ekvivalens, így elég az FCFS.
- Fontos befolyásoló tényezô a fájlok allokációs módja.

  lemez terület kihasználása $\Leftrightarrow$ hozzáférési idô

- Fájljegyzékek (directory), továbbá indexek helye .
- Következmény: szükség van az operációs rendszerben egy elkülönített *diszk ütemezô modulra.*

  Napjainkban ezt a tevékenységet egyre inkább a lemez vezérlô egységébe integrálják, igy az operációs rendszerben elég ha van egy FCFS ütemezô.

# • Diszk kezelés

A diszk kezelésnek több olyan aspektusa van, ami az operációs rendszer hatáskörébe tartozik.

- **Diszk formázás.** (NEM "formattálás"!!!)
  - Fizikai formázás
    - sávok (track), szektorok helyének kialakítása a homogén (mágnesezhetô) lemezfelületen.
    - header (sávcímmel) + adat helye + ECC (Error Correcting Code) helye
    - Ma: gyári fizikai formázás
  - Logikai formázás
    - Lemezcímke (Volume Label), kezdeti üres fájljegyzék, FAT, boot rekord, stb felírása az operációs rendszer által meghatározott formában.
  - Adatbázis kezelôk sokszor maguk alakítják ki a logikai szerkezetet.
- **Boot rekord, boot blokk. Bootstrap rendszer**
- **Hibás blokkok kezelése.**
  - IBM-PC MS DOS
    - IDE=Integrated Drive Electronics (manuálisan) A **format** parancs megjegyzi a FAT-ban ha hibás szektort talált. Késôbb csak egy manuálisan elindított program (**scandisk**) tud újakat hozzávenni.
    - SCSI= A kontroller egy listában nyilvántartja a hibás blokkokat és ezt naprakészen tartja. A hibás blokkok helyett saját (az op rendszer által el nem érhetô) készletébôl jó blokkot ajánl fel . (slipping, szektor átirányítás, ha lehet cilinderen belül.) Baj lehet az ütemezôvel!

# • Swap (csere) terület kezelés

- A swap terület használata
  - teljes processzus területének kitárolása
  - egyes lapok kitárolása
- A swap terület mérete
- A swap terület allokálása
  - Fájl rendszeren belül (Windows)
  - Külön partícióban (UNIX), és nem a fájl rendszer elérési mechanizmusait használva.
  - Külön partícióban a fájl rendszerbôl kiegészítve (Solaris)
- Swap terület kezelés, kezelési stratégiák
  - kód és adatszegmensek különválasztása
  - swap map-ek

# • Diszk megbízhatóság

- A diszk a számítógépes rendszer legsérülékenyebb részének tekinthetô.
- Disk striping: a blokk egyes részei külön lemezekre (szinkronizáció?)
- RAID (Redundant Array of Inexpensive Disks).
  - Paritás diszkek

# • Stabil-tár implementáció

- stabil tár = soha nem vész el róla az adat
  - pl. a *szinkronizációnál* a *write ahead log* kezelésére ilyen kellene
- A tökéletes háttértár szimulációja két lemezzel.

# 13. Védelem

- A védelem célja
- Védelmi tartományok
- Hozzáférési mátrixok (access matrix, AM)
- A hozzáférési mátrixok implementációja
- A hozzáférési jogok visszavonása
- Képesség-alapú rendszerek
- Nyelvbe ágyazott védelem

# • A védelem célja

- A számítógépes rendszer objektumok (hardver és szoftver) együttesének tekinthető.
- Minden objektumnak egyedi neve van és operációknak (műveleteknek) egy jól definiált halmazával érhető el.
  - Absztrakt adattípusok szerepe a modellezésnél!
- Védelmi probléma – biztosítani, hogy minden objektum korrekt módon legyen elérhető, és kizárólag azon processzusok által, amelyeknek az elérés megengedett.
  - need-to-know elv
  - mechanizmusok és politikák szétválasztása

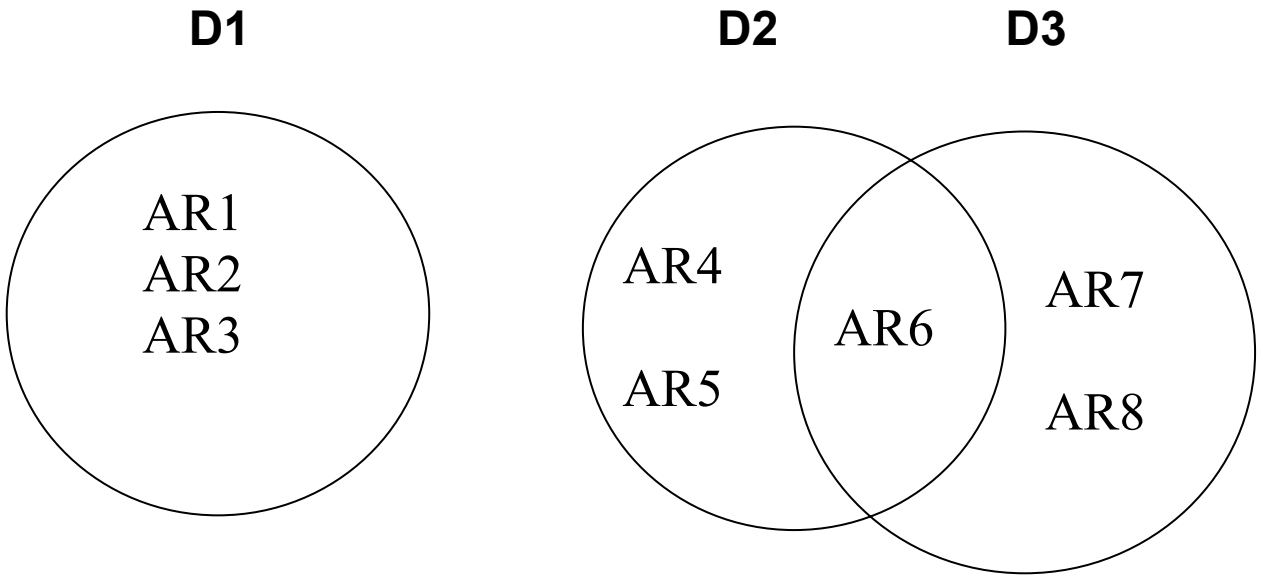
# • Védelmi tartományok (protection domain)

## • Tartomány szerkezet

- ***Hozzáférési jog* = <objektum név, joghalmaz>**

  A joghalmaz a rendszerben implementált (érvényes) operációknak az a részhalmaza, amelynek elemei az objektumra végrehajthatók.

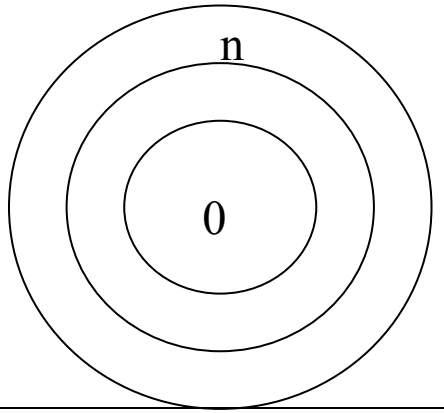
- ***Tartomány* = hozzáférési jogok halmaza**




AR1 = <file_A, {read, write}>

AR8= <file_A, {read}>

- Tartomány implementáció
  - **A rendszerben két *tartomány* van** (primitív)
    - user
    - supervisor
  - **UNIX**
    - *Tartomány* = USER-ID
    - *Tartomány váltást* a fájl rendszeren keresztül lehet végrehajtani.
      - Minden állományhoz tartozik egy tartomány-bit (setuid bit).
      - Ha az állomány végrehajtásra kerül és a setuid bit 1, akkor a USER-ID beállítódik a végrehajtás alatt levő fájl tulajdonosának megfelelően. Ha a végrehajtás befejeződött, a USER-ID visszaáll eredeti állapotába.
  - **Multics gyűrűk**
    - Legyen $D_i$ és $D_j$ két tartomány gyűrű.
    - Ha j<i, akkor $D_i \subseteq D_j$ .



# • Hozzáférési mátrixok

- Sorok: tartományok
- Oszlopok: tartományok + objektumok
- Bejegyzések: hozzáférési jogok, operátor nevek

| objektum / tartomány | FILE1 | FILE2 | FILE3 | PRINTER |
|---|---|---|---|---|
| **D1** | read | | read | |
| **D2** | | | | print |
| **D3** | | read | execute | |
| **D4** | read, write | | read, write | |

# A hozzáférési mátrix használata

- Ha egy processzus a **Di** tartományban az **Oj** objektumon az "**op**" műveletet kívánja végrehajtani, akkor "**op**" elő kell forduljon a hozzáférési mátrix megfelelő pozíció-jában
- Kiterjeszthető dinamikus védelemmé
  - Új operációk: Hozzáférési jogok hozzáadása, törlése
  - Speciális hozzáférési jogok:
    - *owner* of **Oi**,
    - *copy* "op" **Oi**-ból **Oj**-be,
    - *control*: átkapcsolás a **Di** és **Dj** tartományok között
- A hozzáférési mátrix világosan elválasztja a *mechanizmust* a *politikától*
  - **Mechanizmus**: az operációs rendszer szolgáltatja a hozzáférési mátrixot + megfelelő szabályokat.
    - Biztosítja, hogy a mátrix csak arra jogosult agensek által manipulálható és a szabályokat szigorúan betartatja.
  - **Politika**: a felhasználó diktálja.
    - Ki érhet el egyes objektumokat és milyen módban teheti ezt.

# • A hozzáférési mátrixok implementációja

- **Minden oszlop**: = egy **ACL** (Access Control List, hozzáférési lista) *egyetlen* objektumhoz.

  Definiálja, hogy **ki** és **mit** "operálhat" az objektumon**.**

- **Minden sor**: = egy **CL** (Capability List, jogosultsági lista) *egyetlen* tartományhoz.

  Definiálja, hogy **a különböző objektumokon mit** "operálhat" a tartományban levő processzus**.**

- **Lock-key rendszer** = egy objektumhoz zárak tartozhatnak, a tartományhoz kulcsok (bit pattern). Az operációs rendszer menedzseli.

- A hozzáférési jogok visszavonása
  - azonnal – késleltetve
  - szelektíven – általában
  - részben – teljesen
  - ideiglenesen – permanensen

- Megoldások:
  - újrakérés
  - back-pointerek
  - indirekció
  - kulcsok

- Képesség-alapú rendszerek
  - HYDRA
  - Cambridge CAP

- Nyelvbe ágyazott védelem
- JAVA Sandbox

# 14. Biztonság[1]

- A biztonsági probléma
- Program fenyegetettségek
- Rendszer és hálózati fenyegetettségek
- Kriptográfia mint biztonsági eszköz
- Felhasználó azonosítás (autentikáció)
- Biztonsági védelem implementációi
- Tűzfal védelem a rendszer és hálózat számára
- Számítógép biztonsági osztályok
- Windows XP példa

[1] Silbershatz, Galvin & Gagne, Operating system concepts with Java, 7th edition (2007), John Wiley & Sons, Inc, alapján

# • A biztonsági probléma

- A biztonsági probléma a rendszer erőforrásainak védelmét jelenti a rendszer külső környezetével szemben.
- Behatolók (támadók, „cracker”-ek, megkísérelhetik megsérteni a biztonságot.
- „**Fenyegetettség**” (Threat) a biztonság egy potenciális megsértése.
- „**Támadás**” (Attack) a biztonság megsértésének konkrét kísérlete.
  - A „**támadás**” lehet *véletlen*, vagy *rosszindulatú*.
  - A véletlen támadással szemben egyszerűbb védekezni, mint a rosszindulatúval szemben!

# • Biztonsági sérelmek (violations)

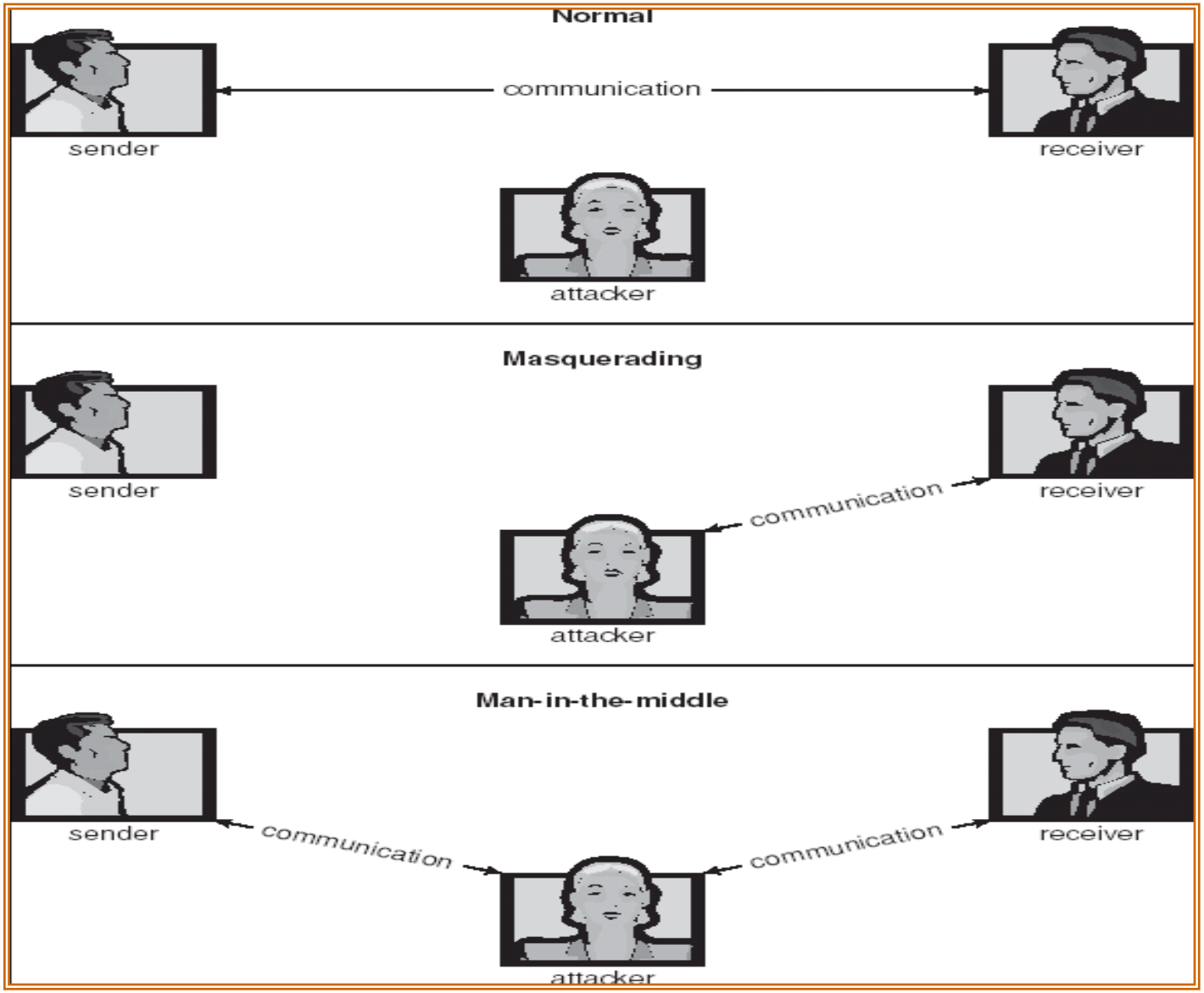
## • Kategóriák

- A titkosság (cofidentality) megsértése.
- Az épség (integrity) megsértése.
- A  hozzáférhetőség (availability) megsértése.
- A szolgáltatás eltulajdonítása (lopás, theft).
- A szolgáltatás megtagadása (denial of service).

## • Módszerek

- Álcázás (masquerading), az  autentikáció megsértése.
- Újrajátszási (replay) támadás.
- Üzenet módosítás.
- „Ügynök közbeiktatása” (Man-in-the-middle) támadás.
- Szekció rablás (Session hijacking)

# • Standard biztonsági támadások



# • Biztonsági intézkedések szintjei

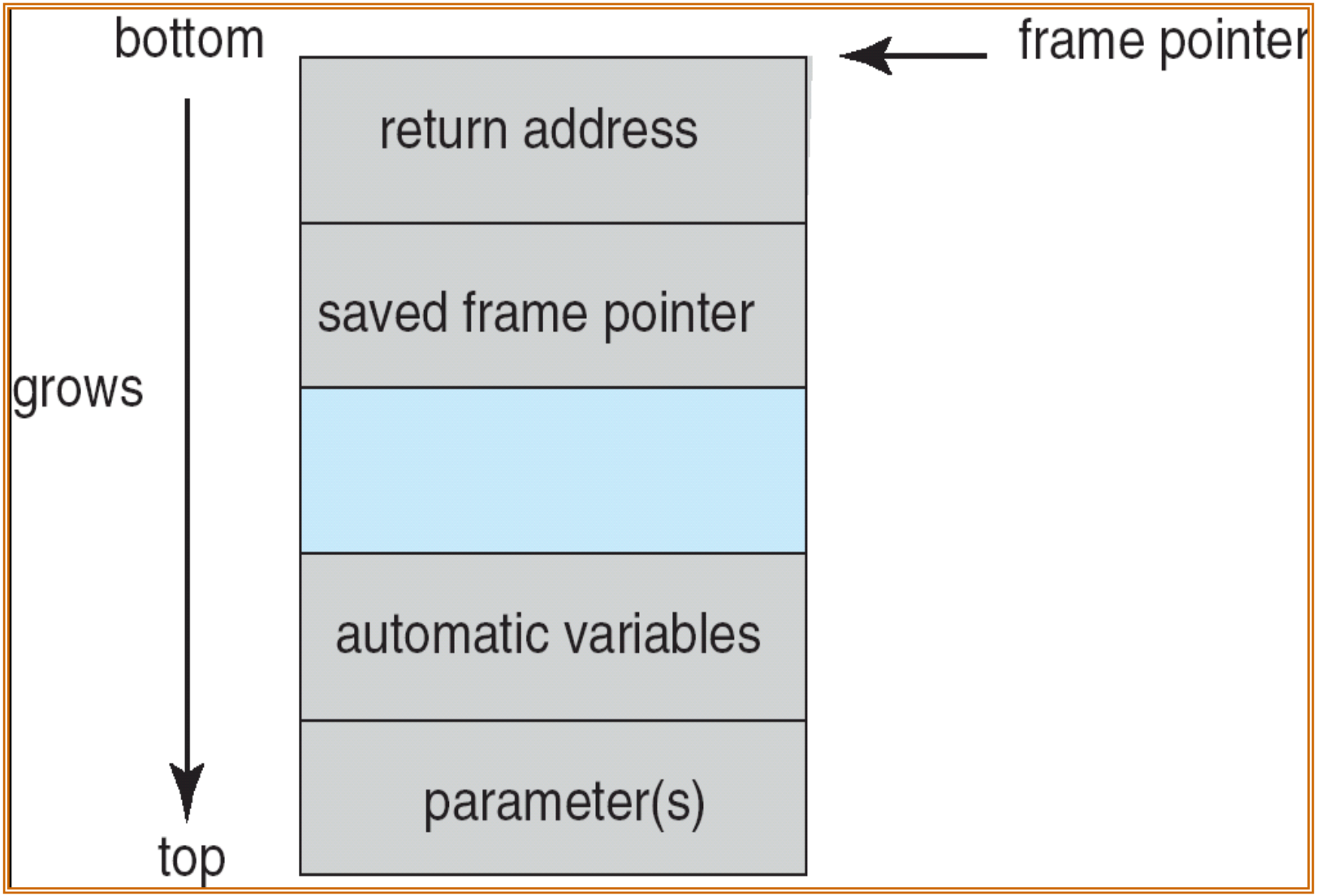
- A hatékonysághoz az intézkedésekneknégy szinten kell megjelenniük:
  - o Fizikai szint
  - o Humán szint
    - Pl. elkerülni: **social engineering, phishing (whaling!), dumpster diving**
  - o Operációs rendszer
  - o Hálózat
- „A hálózat olyan gyenge, mint a leggyengébb láncszeme”

- Program fenyegetettségek
  - **Trójai faló (Trojan Horse)**
    - Kód szegmens (program), amelyik visszaél futtatási környezetével.
    - Más nevében hajt végre olyan akciókat, amelyekre az igénybe vett felhasználó jogosult.
    - Példák: Spyware, pop-up browser windows, rejtett (covert) csatornák (channels).
  - **Csapda (Trap Door)**
    - Speciális azonosító, vagy jelszó, amelyek lehetővé teszik bizonyos biztonsági ellenőrzések megkerülését.
    - Beépítkető a compilerbe is!
  - **Logikai bomba**
    - Egy program, amelyik bizonyos körülmények között biztonsági problémát (security incident) vált ki.
  - **Verem és puffer túlcsordulás**
    - A program egy hibáját kihasználva a verem, illetve a pufferek túlcsordulására alapoz (, amivel a program kódja/adatai módosíthatók!).

- C-program puffer túlcsordulás lehetőséggel

```
#include <stdio.h>
#define BUFFER_SIZE 256

int main(int argc, char *argv[])
{
   char buffer[BUFFER_SIZE];

   if (argc < 2)
      return -1;
   else {
      strcpy(buffer,argv[1]);
      return 0;
   }
}
```

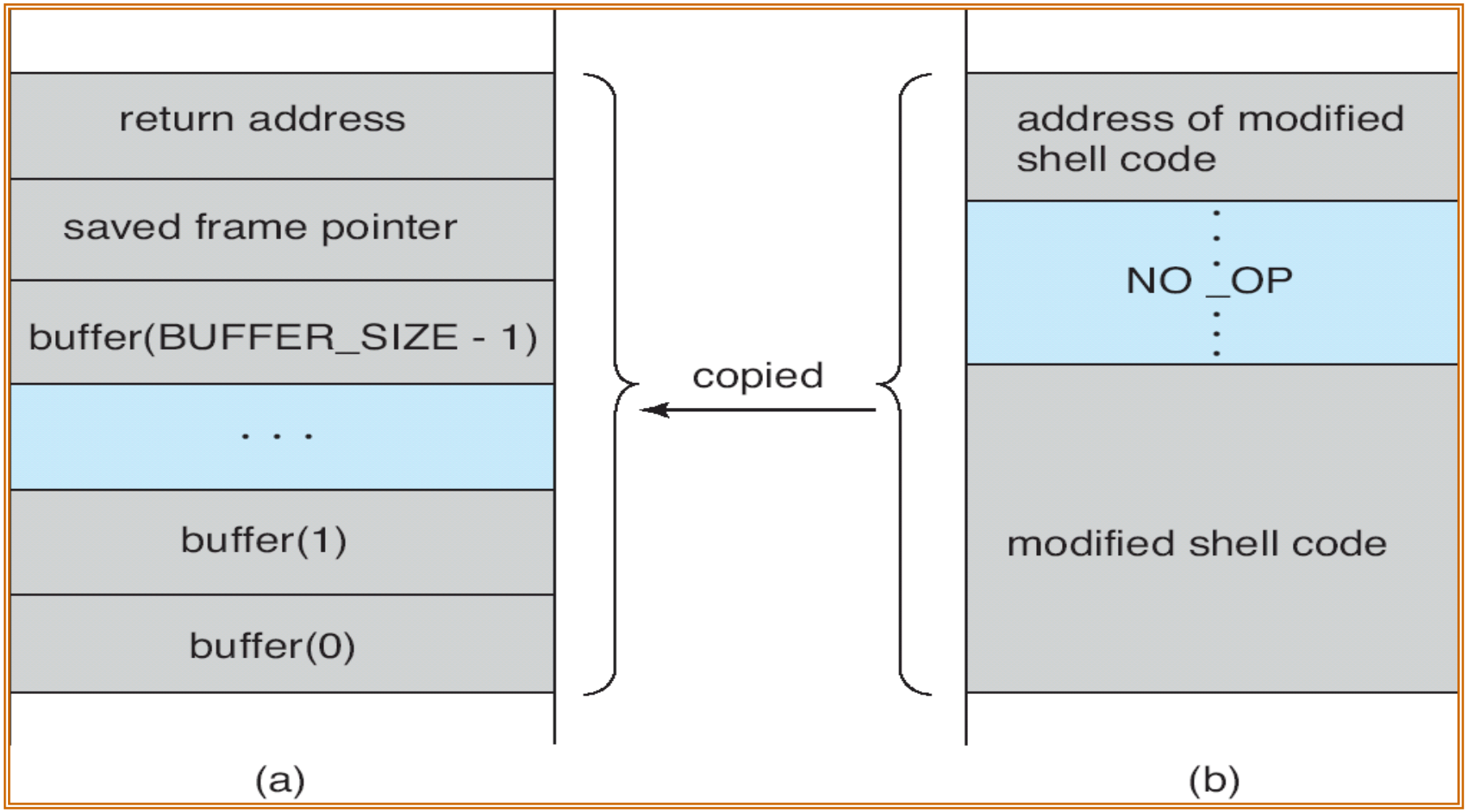
# • Tipikus verem elrendezés (layout)



- **Módosított shell kód**

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    execvp(''\bin\sh'',''\bin \sh'', NULL);
    return 0;
}
```

# • Hipotetikus stack keret (frame)



# • Program fenyegetettségek (folyt)

## • Vírusok

- Kódszegmens egy legitimált programba ágyazva.
- Specializált a CPU architektúrára, operációs rendszerre, alkalmazói programra.
- Rendszerint e-mail, vagy makró útján (részeként) terjed.

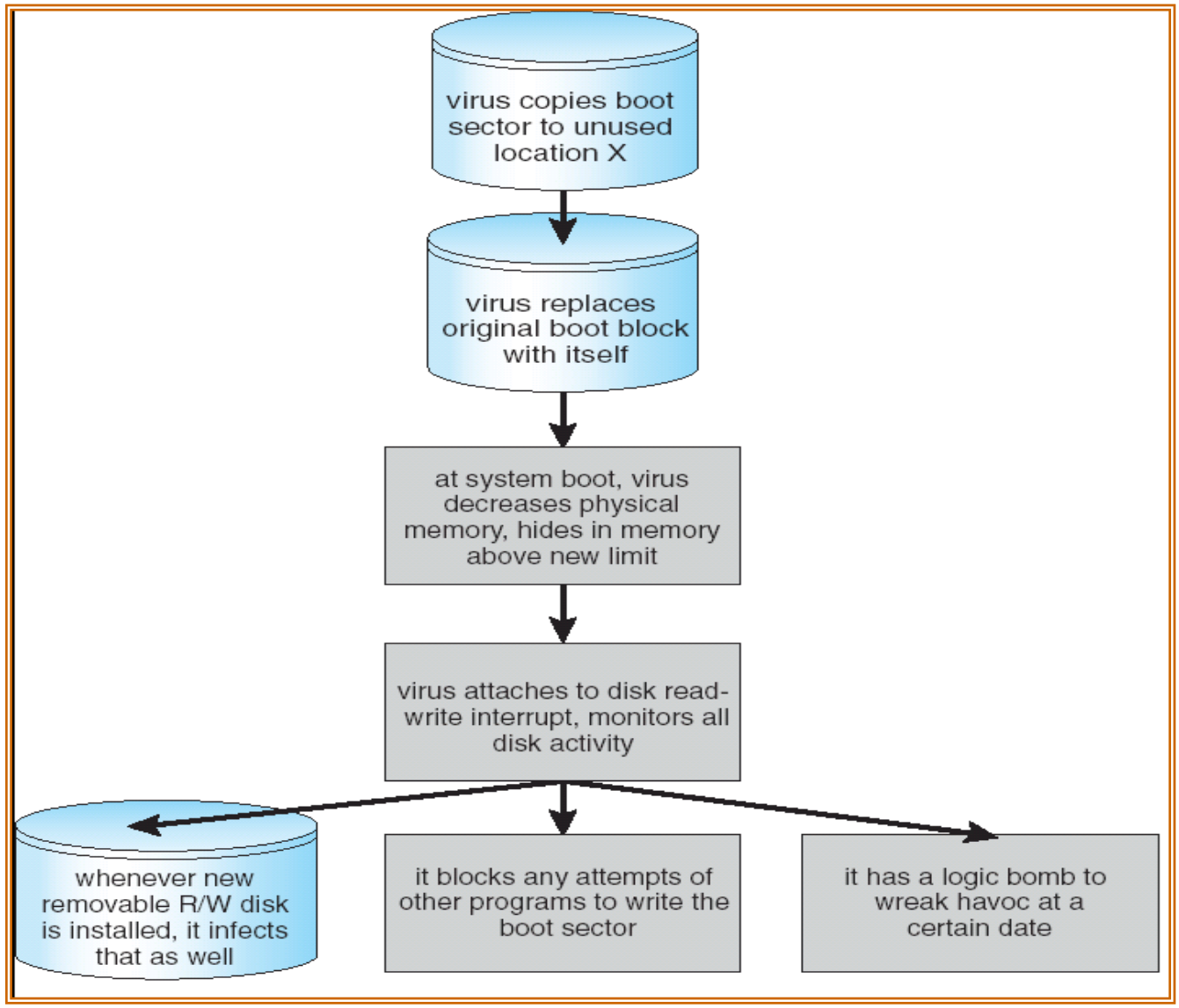
    - Pl.: egy Visual Basic makró, amelyik újraformázza a merevlemezt:

```
Sub AutoOpen()
Dim oFS
   Set oFS = CreateObject(''Scripting.FileSystemObject'')
   vs = Shell(''c:command.com /k format   c:'',vbHide)
End Sub
```

# • Program fenyegetettségek (folyt)

- Vírus „csepegtető” (virus dropper) illeszti be a vírust a rendszerbe.
- Számos kategória, összességében több ezer (konkrét) vírus létezik!
  - File
  - Boot
  - Macro
  - Source code (forráskód)
  - Polymorphic (polimorf, több alakú)
  - Encrypted (rejtjelezett)
  - Stealth (lopakodó)
  - Tunneling (alagút)
  - Multipartite (több részre osztott)
  - Armored (páncélozott)

# • Boot szektor vírus

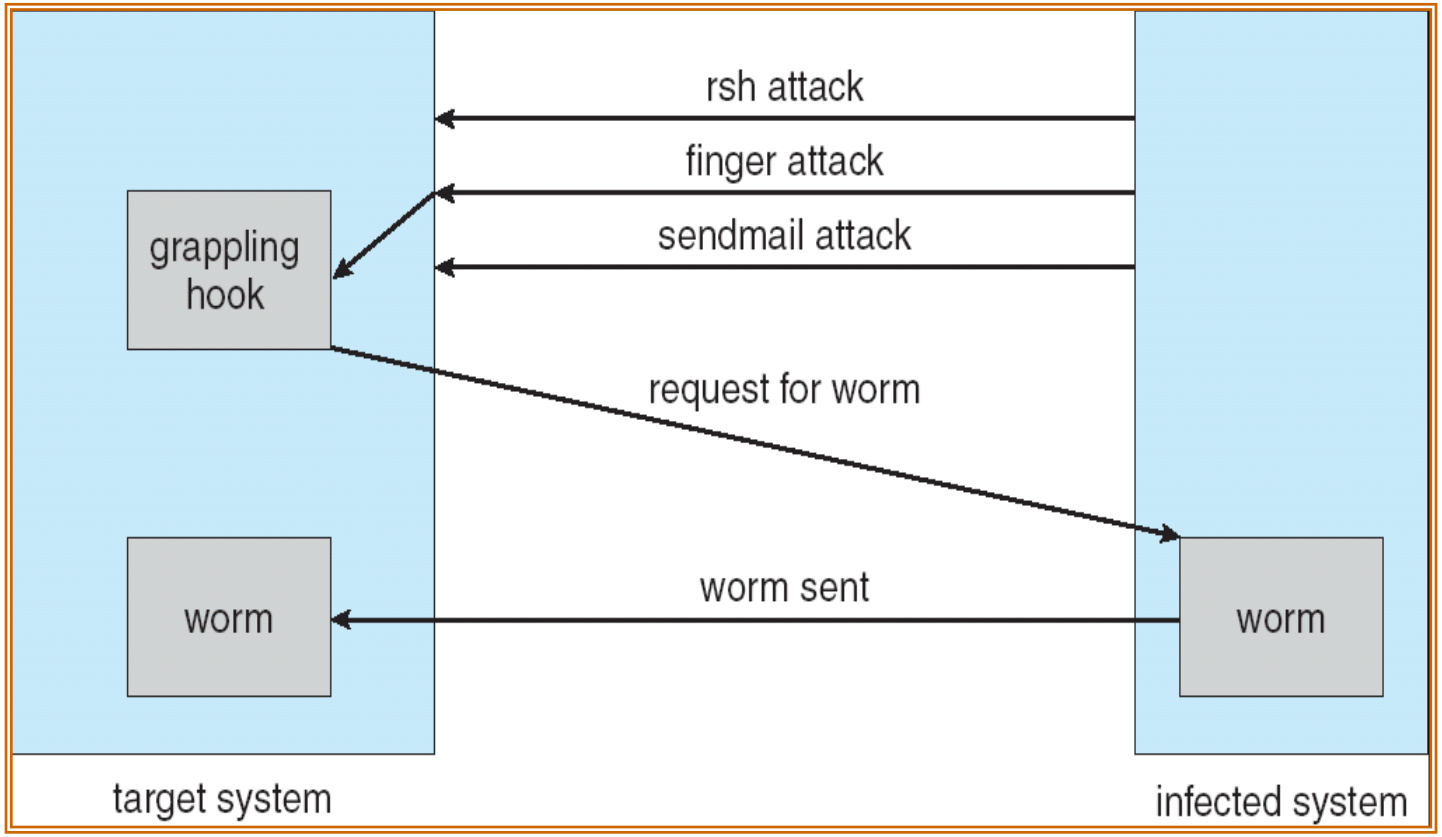


# • Rendszer és hálózati fenyegetettségek

- **Férgek** (worms) önálló programok, amelyek valamilyen szaporodási (spawn!) mechanizmust használnak a terjedéshez
- **Internet féreg**
  - A UNIX hálózati szolgáltatásait (remote access, rpc), illetve a *finger* és *sendmail* programokban levő hibákat (bug) használják ki.
  - A férget egy “horgony program” tölti le a hálózaton keresztül.
- **Port pásztázás (scanning)**
  - Automatizált kísérletek meghatározott tartományba eső IP címekhez tartozó meghatározott tartományba eső portokra történő csatlakozásra.
- **A szolgáltatás megtagadása (Denial of Service)**
  - A megcélzott számítógép (szerver) túlterhelése, hogy ne tudjon semmi „hasznos dolgot” csinálni.
  - Még veszélyesebb: Distributed denial-of-service (DDOS), ahol sokan támadnak egyszerre.

# • A Morris – féle Internet féreg

- 1988. november 2.: Robert Tappan Morris (első éves Cornell egyetemista)
- SUN3, VAX BSD Unix.
- 3 év próba, 400 óra közmunka, 10000$ pénzbüntetés




- Más: 2003. augusztus: Sobig.F

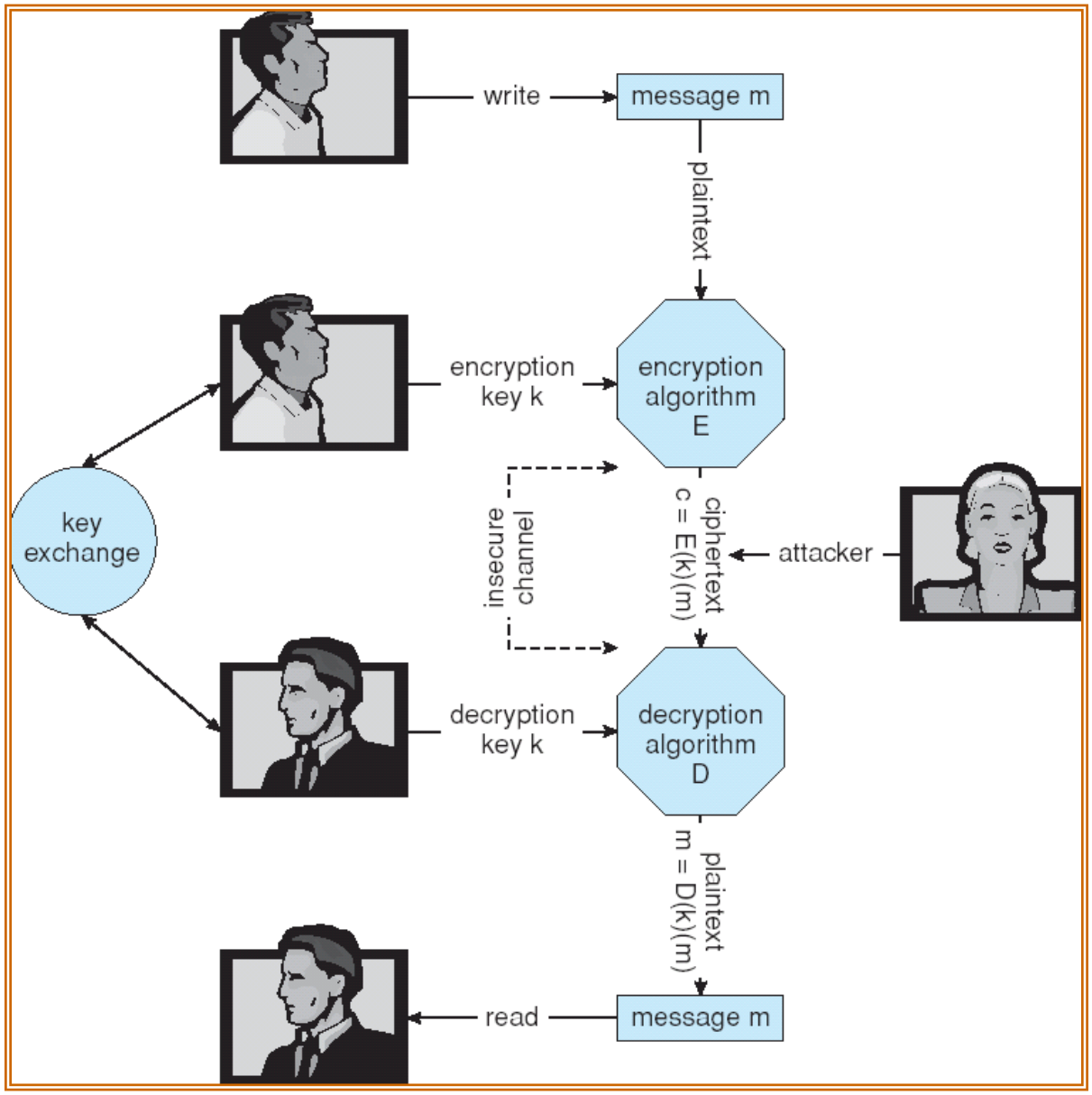
# • Port scanning Java-ban

```
public class PortScanner
{
   public static final int PORT_MAX = 255;

   public static final int TIMEOUT_VALUE = 1000; // 1 second

   public static void main(String[] args) {
     InetAddress host = InetAddress.getByName(args[0]);

     for (int port = 0; port <= PORT_MAX; port++) {
       try {
          SocketAddress addr = new InetSocketAddress(host,port);
          Socket sock = new Socket();

          // attempt to make a connection to (host + port)
          sock.connect(addr,1000);
          System.out.println("Listening at port: " + port);
          // we could now try to exploit the service
          // listening at this port
       }
       catch (java.io.IOException ioe) {
          // not listening to this port
       }
     }
   }
}
```

# • Kriptográfia mint biztonsági eszköz

- **Az elérhető legáltalánosabb biztonsági eszköz**
  - Az üzenet forrása és célállomása nem lehet bizalomra méltó kriptográfia nélkül.
  - Eszköz a potenciális adók (források) és/vagy a vevők (célállomások) korlátozására.
- **Valamilyen titkok (kulcsok) képezik a módszerek alapját**

# • Biztonságos kommunikáció közönséges eszközökkel



# • Rejtjelezés (kódolás, encryption)

# • A rejtjelező algoritmus elemei

- ***K*** **a kulcsok halmaza (keys)**
- ***M*** **az üzenetek halmaza (messages)**
- ***C*** **a rejtjelezett szövegek halmaza (cipher texts)**
- **Egy függvény** $E : K \rightarrow (M \rightarrow C)$**.**
  - Azaz, minden $k \in K$, $E(k)$ egy olyan függvény amely üzenetekhez rejtjelezett szöveget rendel.
  - Mind $E$ és $E(k)$ (minden lehetséges $k$- ra) hatékonyan kiszámítható függvény kell legyen.
- **Egy függvény** $D : K \rightarrow (C \rightarrow M)$**.**
  - Azaz, minden $k \in K$, $D(k)$ egy olyan függvény amely rejtjelezett szövegekhez üzeneteket rendel.
  - Mind $D$ és $D(k)$ (minden lehetséges $k$- ra) hatékonyan kiszámítható függvény kell legyen.

# • Aszimmetrikus rejtjelezés

- A nyilvános kulcsú rejtjelezés esetén minden partner két kulccsal rendelkezik:
  - **Nyilvános kulcs** (public key), amely publikus és a rejtjelezéshez használatos.
  - **Titkos kulcs** (private key), amelyet minden partner titokban tart és a rejtjelezett szöveg visszaalakításához használ.
- Szükség van egy rejtjelező sémára, amelyet nyilvánosságra lehet hozni anélkül, hogy belőle a visszafejtő sémát könnyen ki lehetne találni.
- Pl. egy ilyen rendszer az RSA
- 1978. R. Rivest, A. Shamir, L. Adleman (Tel Aviv University)

- gyors prímtesztek,
- lassú faktorizáció!

# • Aszimmetrikus rejtjelezés (folytatás)

- Összefoglalás:
- Az RSA esetében reálisan lehetetlen meghatározni a $D(k_d , N)$-t az $E(k_e , N)$-ból, $E(k_e , N)$-t nem szükséges titokban tartani és ezért széles körben terjeszthető
  - $E(k_e , N)$ (vagy $k_e$) a nyilvános kulcs **(public key)**
  - $D(k_d , N)$ (vagy $k_d$) a titkos (magán) kulcs **(private key)**
  - Emlékeztetőül: $N$ két nagy véletlen módon választott prím szám $p$ és $q$ szorzata (pl. $p$ és $q$ 512 bit méretűek )
- **Titkosítási algoritmus:**

$$E(k_e , N)(m) = m^{k_e} \bmod N,$$

ahol $k_e$ eleget tesz a $k_e k_d \bmod (p-1)(q-1) = 1$ kongruenciának.

- A **visszafejtő algoritmus** ekkor

$$D(k_d , N)(c) = c^{k_d} \bmod N.$$

# • Aszimmetrikus rejtjelezés (egyszerű! példa)

Legyen $p = 7$ és $q = 13$.
Akkor kiszámítható, hogy $N = 7* 13 = 91$ és $(p-1)(q-1) = 6*12 = 72$.
Választunk egy $k_e$ 72-höz relatív prím és < 72 számot, mondjuk legyen ez 5 .
Végül meghatározzuk $k_d$ –*t* úgy, hogy

$$k_e k_d \equiv 1 \pmod{72}$$

teljesüljön, ami , $k_d$ =29-hez vezet.

Kulcsaink:

Publikus kulcs: $k_e$, $N = 5, 91$
Privát kulcs: $k_d$ , $N = 29, 91$

Kódolás–dekódolás:

A “69” üzenet kódja a nyilvános kulcs segítségével: $69^5 \equiv 62 \pmod{91}$.
A titkos üzenet (“62”) dekódolható a titkos kulcs segítségével: $62^{29} \equiv 69 \pmod{91}$.

# • Autentikáció

- Constraining set of potential senders of a message
  - Complementary and sometimes redundant to encryption
  - Also can prove message unmodified
- Algorithm components
  - A set $K$ of keys
  - A set $M$ of messages
  - A set $A$ of authenticators
  - A function $S : K \rightarrow (M \rightarrow A)$
    - That is, for each $k \in K$, $S(k)$ is a function for generating authenticators from messages
    - Both $S$ and $S(k)$ for any $k$ should be efficiently computable functions
  - A function $V : K \rightarrow (M \times A \rightarrow \{\text{true, false}\})$. That is, for each $k \in K$, $V(k)$ is a function for verifying authenticators on messages
    - Both $V$ and $V(k)$ for any $k$ should be efficiently computable functions

# • Autentikáció

- For a message $m$, a computer can generate an authenticator $a \in A$ such that $V(k)(m, a) = \texttt{true}$ only if it possesses $S(k)$
- Thus, computer holding $S(k)$ can generate authenticators on messages so that any other computer possessing $V(k)$ can verify them
- Computer not holding $S(k)$ cannot generate authenticators on messages that can be verified using $V(k)$
- Since authenticators are generally exposed (for example, they are sent on the network with the messages themselves), it must not be feasible to derive $S(k)$ from the authenticators