# Operációs rendszerek

## *Memóriakezelés*

## *1.1*

Pere László (pipas@linux.pte.hu)

PÉCSI TUDOMÁNYEGYETEM TERMÉSZETTUDOMÁNYI KAR

INFORMATIKA ÉS ÁLTALÁNOS TECHNIKA TANSZÉK

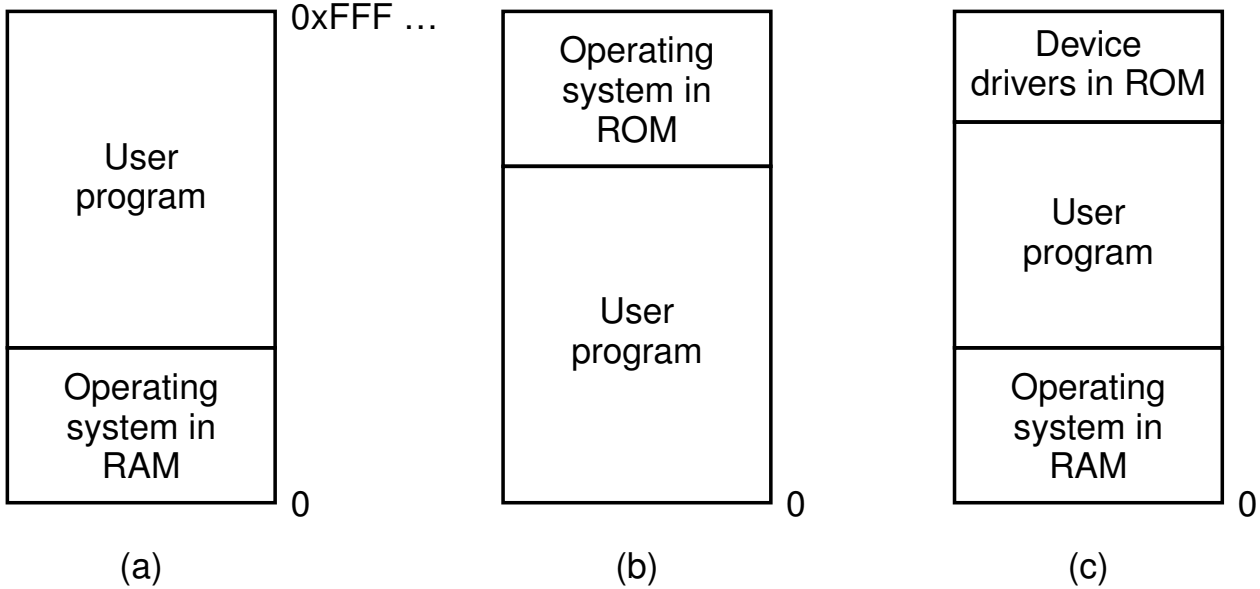
# A memóriakezelő

A memória fontos erőforrás, a kezelését a memóriakezelő (*memory manager*) végzi. Ennek feladatai:

1. Nyilvántartás: a memória mely részei vannak használatban és melyek szabadok?
2. Kiosztás: a folyamatok a memóriakezelőtől kérhetnek memóriát és a segítségével szabadíthatják fel.
3. Memória bővítése: kezeli a csereterületet, vagyis szükség esetén a háttértárra menti majd visszatölti a memória tartalmát.

# Monoprogramming

A monoprogramozott környezetben igen primitív memóriamodell használhatunk.




1. ábra. Monoprogramozott memóriamodellek

# Monoprogramming

A következő modelleket használják a swapping nélküli monoprogramozott rendszerek:

(a) Alul a RAM-ban az OS, felül a RAM-ban a futtatott program (mainframek, minicomputerek, ma már nem használjuk).

(b) Alul a RAM-ban a futtatott program, felül ROM-ban az OS (palmtopok, beágyazott rendszerek).

(c) Alul a RAM-ban az operációs rendszer, középen a RAM-ban a futtatott program, felül az eszközkezelők ROM-ban (DOS, PC rendszerek).

# Multiprogramming

A multiprogramozott rendszerekben egy időben több folyamat is létezik.

Ma kényelmi szempontokat tartunk szem előtt, régebben azonban a multiprogramozott modelltől azt vártuk, hogy a CPU kihasználtsága növekedjék:

- Naív: $5 \times 20\% = 100\%$

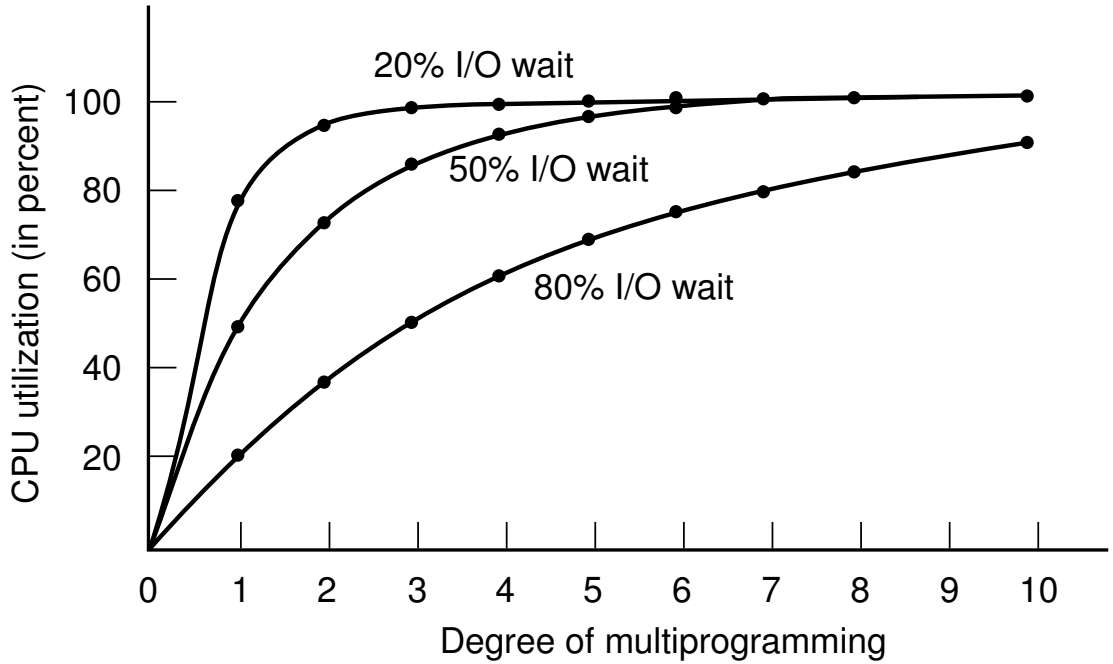
# Multiprogramming

A multiprogramozott rendszerekben egy időben több folyamat is létezik.

Ma kényelmi szempontokat tartunk szem előtt, régebben azonban a multiprogramozott modelltől azt vártuk, hogy a CPU kihasználtsága növekedjék:

- Naív: $5 \times 20\% = 100\%$
- Valóság: $1 - p^n$ ($p$ részt vár, $n$ process)

# Multiprogramming




2. ábra. A CPU kihasználtsága multiprogramozott környezetben

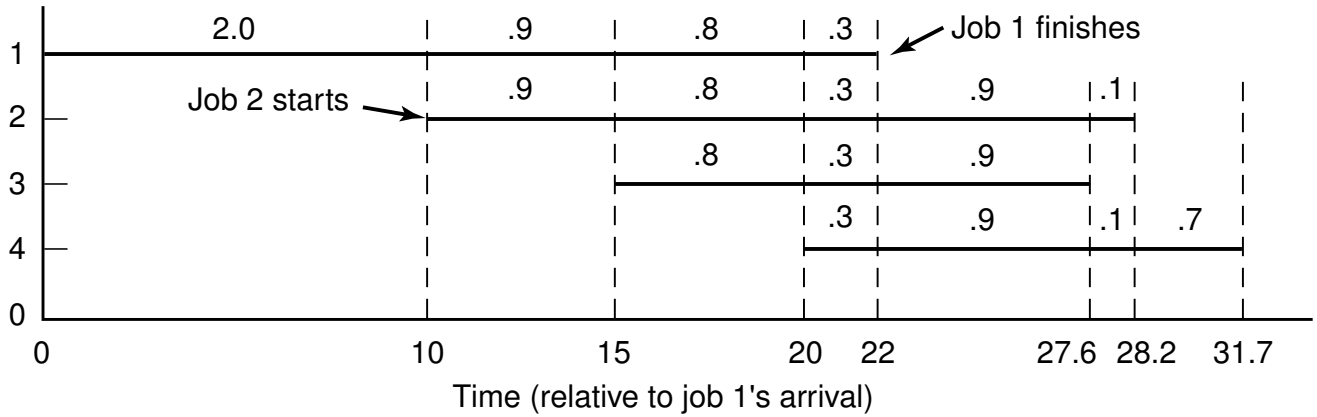
# Multiprogramming

| Job | Arrival time | CPU minutes needed |
|---|---|---|
| 1 | 10:00 | 4 |
| 2 | 10:10 | 3 |
| 3 | 10:15 | 2 |
| 4 | 10:20 | 2 |

(a)

| | # Processes 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| CPU idle | .80 | .64 | .51 | .41 |
| CPU busy | .20 | .36 | .49 | .59 |
| CPU/process | .20 | .18 | .16 | .15 |

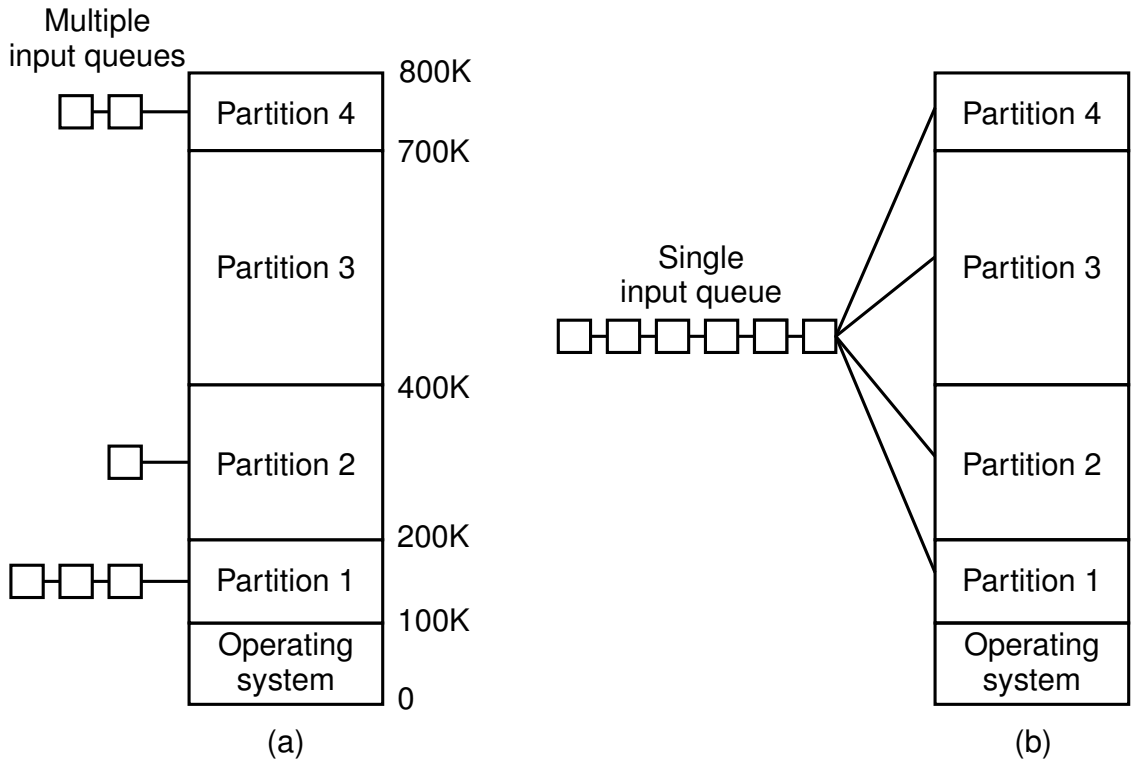
(b)




(c)

3. ábra. Konkrét példa (80% wait)

# Multiprogramming




4. ábra. Memóriapartíciók használata

# Multiprogramming

A legegyszerűbb modell fix méretű memória-partíciókkal dolgozik, amelyeket az operátor a rendszerindításkor alakít ki.

A beérkezett jobok a legkisebb memória-partícióba kerülnek, amelybe beleférnek. (Külön-külön vannak sorok vagy együtt.)

Több memóriaoptimalizálási problémát is felvet ez a modell, hiszen nehéz garantálni, hogy minden jobra sor kerüljön és ne vesztegessük a memóriát (egy partícióba csak egy job kerülhet).

# Relokáció

A memória partícionálása és a multiprogramozás felveti a relokáció problémáját, melynek során a programban található utasítások címét meg kell változtatni attól függően, hogy melyik partícióra töltjük be a programot.

Megoldást jelent, ha a linker készít egy táblázatot, amelyben felsorolja mely helyeken találhatók a címek, amelyeket az OS-nek módosítania kell, mikor a programot betölti.

# Memóriavédelem

A futó programok nem módosíthatják vagy olvashatják a többi job memóriaterületét. Erről az operációs rendszernek kell gondoskodnia.

Megoldást jelent, ha a memóriát felosztjuk lapokra, mindegyiket ellátjuk egy jellel, ahogyan a folyamatokat is (OS joga és feladata). A hardver megszakítást generál ha memóriasértés történik.

# Teljes megoldás

A következő megoldás mindkét problémát megoldja.

Készítsünk a processzorban két regisztert: BASE és LIMIT, egy áramkörrel mindig adjuk hozzá a címekhez a BASE értékét és figyeljünk rá, hogy ne kerülhessen nagyobb érték a buszra mint a LIMIT.

Amikor az ütemező másik folyamatra kapcsol át, írjuk a BASE és LIMIT regiszterekbe a folyamatra jellemző értékeket.

# Nincsen elegendő memória

Interaktív rendszereknél nem korlátozhatja az egyszerre futtatható programok számát a rendelkezésre álló memória (batch rendszereknél is hasznos ha több job közül válogathatunk). Két módszert használhatunk:

- Swapping, csere: a teljes partíciót kitesszük a háttértárra és beteszünk egy másikat.
- Virtuális memória: engedjük futni a partícióban található folyamatot akkor is, ha a partíció egy része nincs a fizikai memóriában.

# Csereterület

Tegyük lehetővé, hogy:

- a partíciók mérete dinamikusan változzon a futás közben, minden folyamat akkora partícióba kerüljön, amekkora szükséges a futtatásához,
- a folyamatok bármikor kikerülhessenek a swap területre,
- bármikor visszatölthetőek legyenek, akár egy másik területre is,
- a memóriapakolás során bármikor áthelyezhessük a folyamatokat,
- a folyamatok kérhessék a partíció növelését (dinamikus memóriaallokálás)

# Overlay technika

Az lefedéses memóriagazdálkodás (*overlay*) a virtuális memóriakezelés primitív módja, amelyet ma már általában nem használunk.

Az egyes modulok kivételét és betöltését az operációs rendszer végzi, a program feldarabolását azonban a programozó.

Nem túl szerencsés, ha a programozó nem programot ír, hanem darabolja a megírt programot.

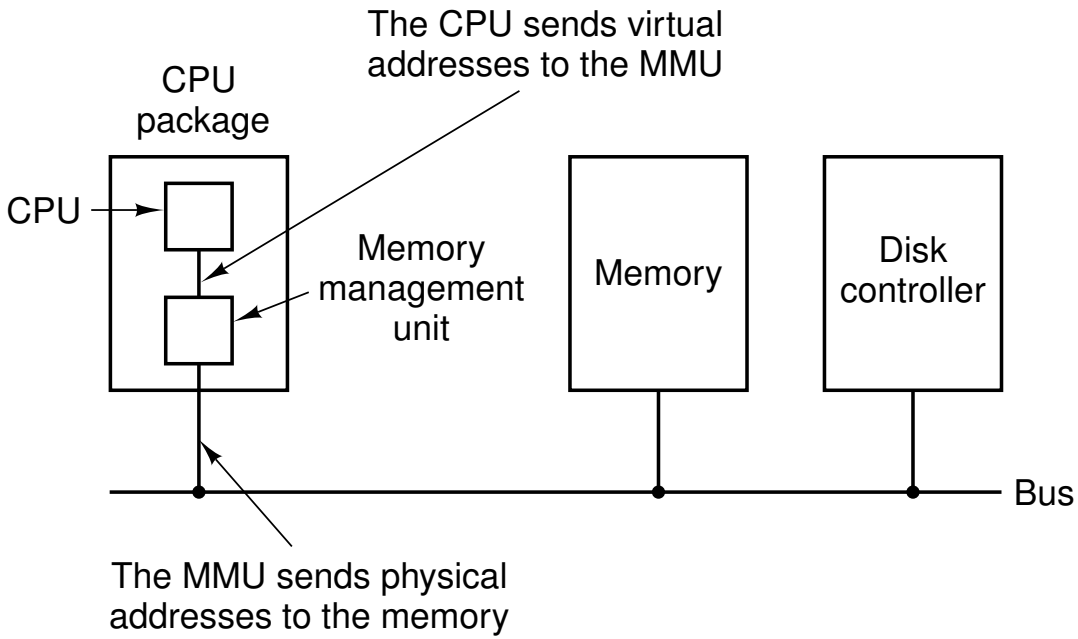
# Virtuális memória

A virtuális memória alapgondolata az, hogy a programot darabokra vágjuk és mindig csak azokat a darabokat tartjuk a memóriában, amelyek beleférnek, a többit a háttértárra mentjük.

A program és adatterületeinek szabályos darabolása, a szükséges darabok betöltése és a nélkülözhető darabok háttértárra írása ma már automatikusan történik.

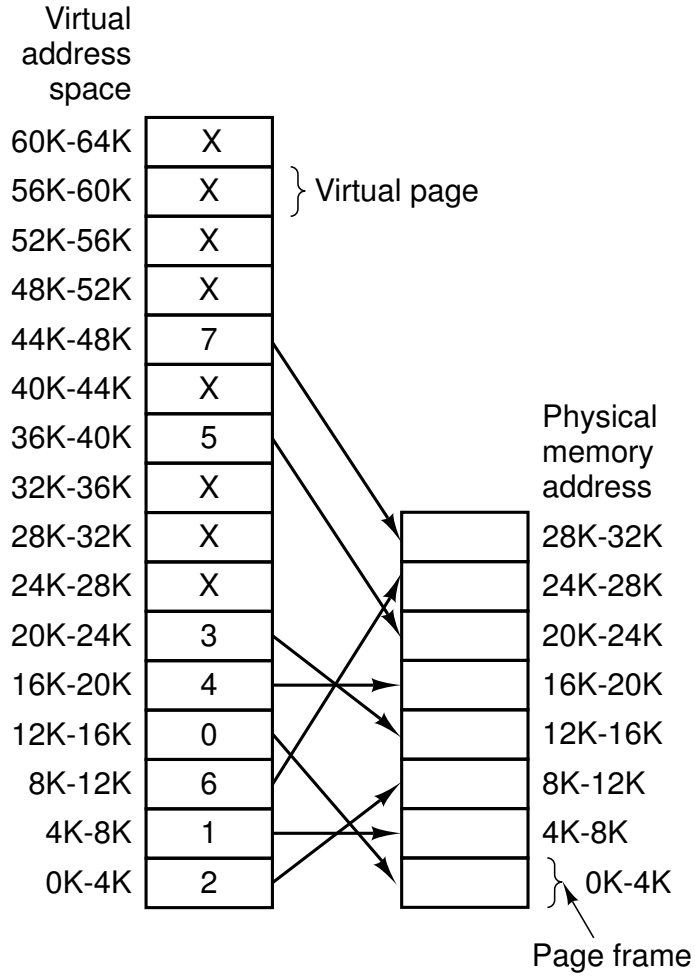
# Az MMU

A *memory management unit* a CPU és a memória közt található.




5. ábra. Az MMU

# Lapok és lapkeretek

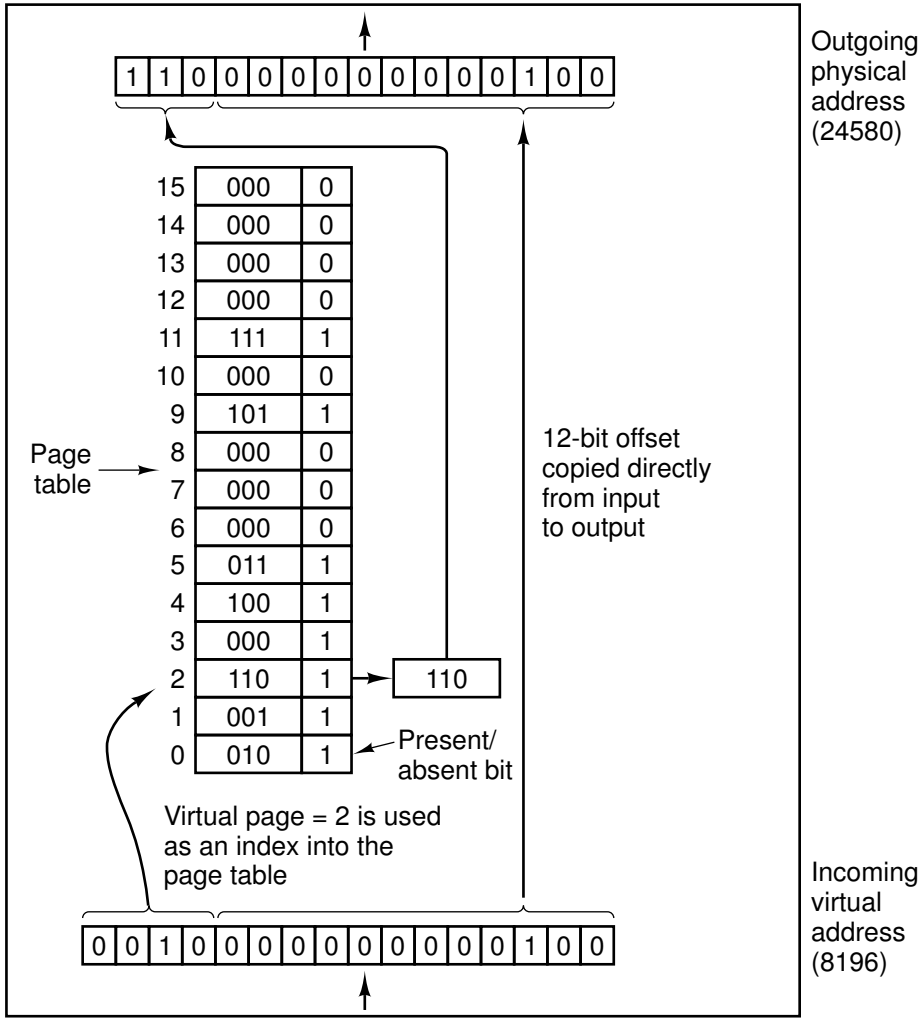


# Lapozás

A lapozás (*paging*) lényege a következő:

- A CPU, a programozó virtuális címeket kezel.
- A virtuális címeket az MMU (*memory management unit*) áramkör képezi le fizikai címekké.
- A virtuális címtartományt lapokra (*pages*) osztjuk, amelyek a memóriában lapkeretekbe (*page frame*) kerülnek.
- Unmapped page elérésekor az MMU page fault trap-et hív.

# A laptábla


Outgoing physical address (24580)
1 1 0 0 0 0 0 0 0 0 0 0 1 0 0
15 000 0
14 000 0
13 000 0
12 000 0
11 111 1
10 000 0
9 101 1
8 000 0
7 000 0
6 000 0
5 011 1
4 100 1
3 000 1
2 110 1
1 001 1
0 010 1
Page table
110
Present/ absent bit
12-bit offset copied directly from input to output
Virtual page = 2 is used as an index into the page table
0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0
Incoming virtual address (8196)

# A laptábla

A laptábla (*page table*) az MMU egyik legegyszerűbb megvalósítása:

- A virtuális címet két részre osztjuk: a felső rész a lap száma, az alsó az offszet.
- A lap számát a laptábla megcímzésére használjuk, így a laptábla megmutatja, hogy az adott lap melyik lapkeretben van.
- A lapkeret száma + offszet a fizikai cím.

# Problémák

A laptábla használatával gondok merülnek fel:

- A leképezésnek nagyon gyorsnak kell lennie (két, három konverzió egy gépi kódú utasítás alatt)
- A laptábla nagyon nagy lehet (32 bites címtartományban hány 4 KByte-os blokk van?)

A módszert más eszközökkel együtt használjuk ma is (memóriába mentett page table, multi-level page table stb.).

# A méret a lényeg

Virtuális laponként egy bejegyzés a laptáblában nagyon sok lehet:

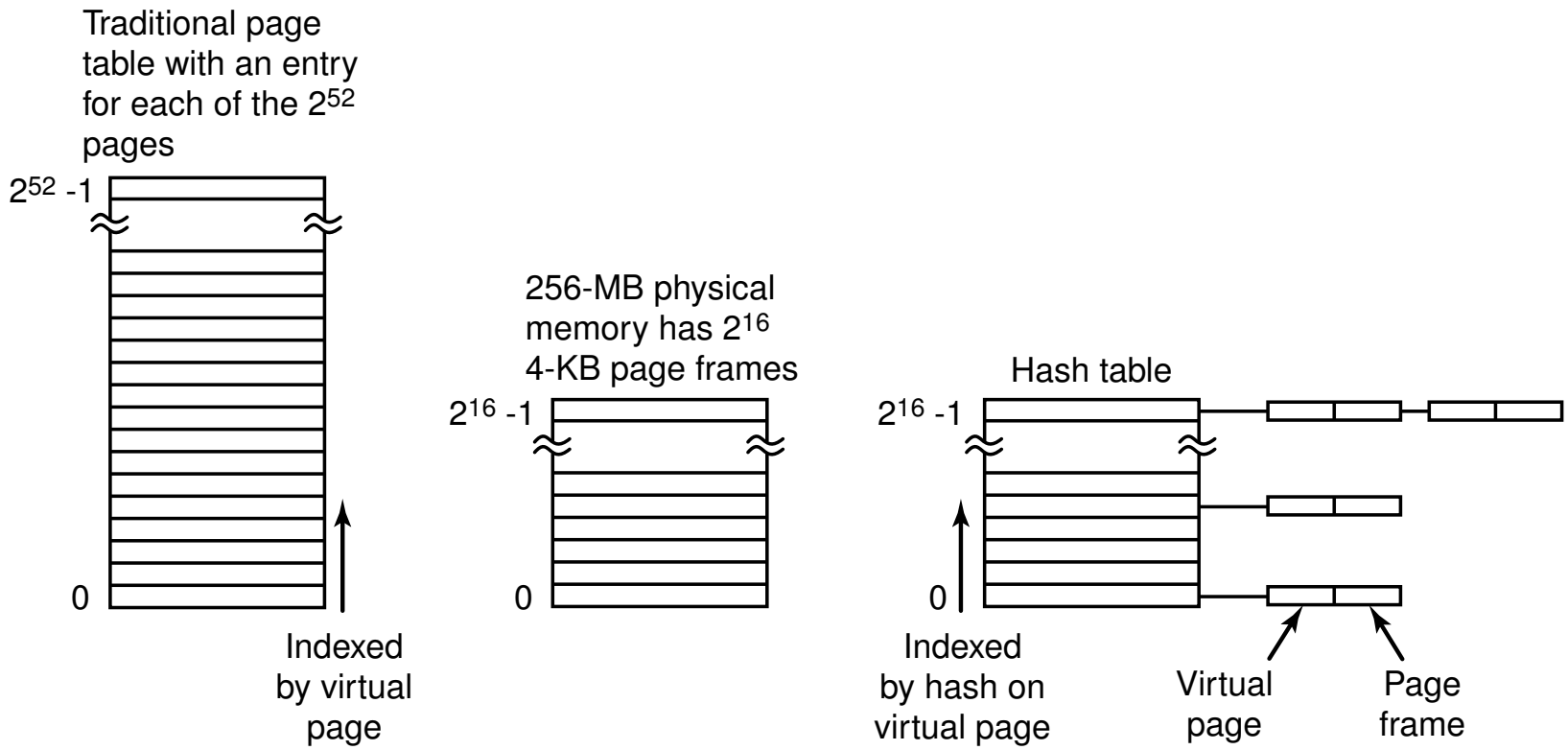
1. 32 bites címtartomány, 4096 byte-os virtuális lapokkal mintegy 1 millió bejegyzés. Minimálisan kb. 4 megabyte.
2. 64 bites címtartomány, 4096 byteos lapokkal, $2^{52}$ bejegyzés. Mintegy 30 000 000 gigabyte-os page table.

# Invertált laptábla

A lapkeretek számára készítünk egy-egy bejegyzést, nem lapok számára. Ez segít, feltéve, hogy a fizikai memória jóval kisebb mint a processzor által kezelt címtartomány.

A probléma itt az, hogy fordítva kell keresni és ez igen lassúvá teszi a műveletet. TLB-t (translation lookaside buffer) kell használni.

# Invertált laptábla



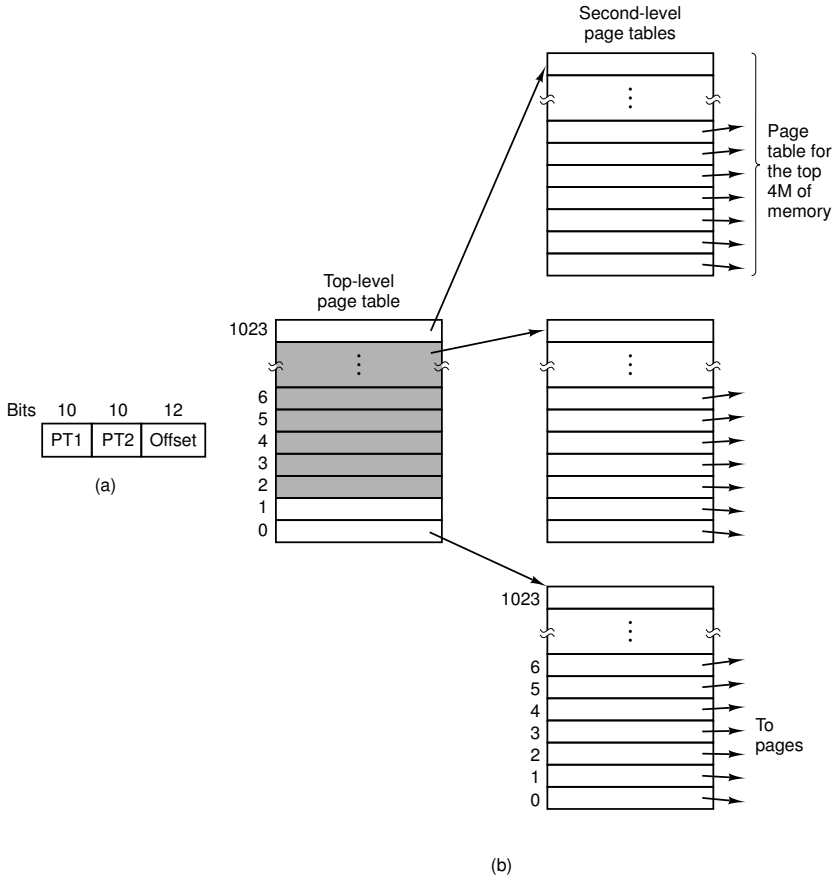
# Többszintű laptábla

A két vagy többszintű laptábla használatának a lényege az, hogy nem kell a nem használt memóriaterületekhez tartozó laptábla bejegyzéseknek létezniük.

Azoknak a másodszintű laptábláknak például, amelyekhez az elsőszintű laptábla bejegyzése nem létezőként jelez, nem kell másodszintű laptábla.

A módszer jól használható, mert általában nem használunk olyan programokat, amelyek a teljes címtartományt használnák.

# Kétszintű laptábla




6. ábra. 32 bites cím kétszintű laptáblával

# Laptábla bejegyzés

A laptábla bejegyzés (*page table entry*) erősen hw függő, de elterjedtek a következő elemek

- Page frame number
- Present/absent (page fault)
- Protection (read, write, execute)
- Modified (dirty)
- Referenced
- Disable caching (memory mapped I/O esetén)

A page fault kezeléséhez szükséges információnak nem kell itt lennie, azt az OS kezeli, nem a hw.

# A lapcserélés

A lapcserélő (*page replacement*) algoritmusok célja, hogy a page fault esetén olyan lapkeretet szabadítsunk fel, amelynek tartalmára előreláthatólag hosszú időn keresztül nem lesz szükség.

Az optimális lapcserélő algoritmus éppen az, amelyik a legkésőbb szükséges lapot távolítja el, de sajnos lehetetlen megvalósítani, mert a jövő ismeretlen.

# Not recently used

Minden lapkeret rendelkezik egy R és egy M bittel, az operációs rendszer időről időre törli az R bitet. Minden page fault esetén a következő kategóriákat jelöljük ki:

1. not referenced, not modified
2. not referenced, modified
3. referenced, not modified
4. referenced, modified

Mindig a legalsó kategóriából választunk véletlenszerűen.

# First-in, first-out

A legrégebbi lapkeretet (azt amelyik a legrégebb óta a memóriában van) ürítjük ki.

Nem túl kifinomult módszer, mert vannak olyan lapkeretek, amelyekre hosszú időn keresztül szükség van.

A boltos példánál maradva lehet, hogy eltávolítjuk a sót és a cukrot, ezért nem szokás használni.

# Második esély algoritmus

A FIFO módosított változata.

Vizsgáljuk meg a legrégebbi lapot. Ha R bit be van állítva, akkor töröljük és helyezzük a sor végére, mintha most töltöttük volna be, ha nem, ezt a lapkeretet örítjük ki.

Ha minden lapkeret volt használva, ez az algoritmus megegyezik a FIFO algoritmussal.

A boltos példa: ha az elmúlt héten vettek az árúból, akkor adunk neki egy második esélyt és a sor végére tesszük.

# Least recently used

Az LRU algoritmus esetén a legrégebb óta nem használt lapkeretet szabadítjuk fel.

A korrekt megvalósítás igen költséges volna, mivel minden memóriahozzáférésnél frissíteni kellene a listát, rögzíteni kellene, hogy mikor használtuk a lapot.

Megoldható volna, ha a laptábla bejegyzése tartalmazna egy számlálót, amely minden memóriahozzáféréskor inkrementálódna.

# Simulált LRU

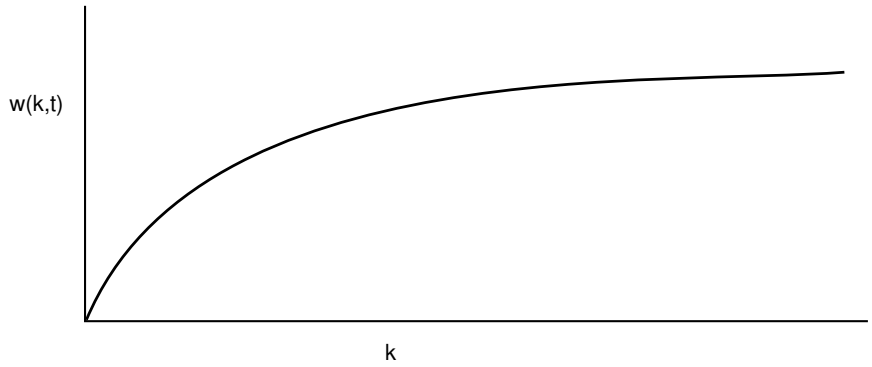
NFU (not frequently used): időről időre növeljük a szoftveresen kezelt lapkeret használtsági számlálót, növeljük, ha az R bit 1. Az R bitet ilyenkor mindig töröljük.

Öregedés: léptessük jobbra a számlálót, a legfelső bithez adjuk hozzá az R bitet. E módszerrel elérhetjük, hogy a régmúlt időben begyűjtött pontok ne rontsák a jóslást.

# Igény szerinti lapozás

Az igény szerinti lapozás esetében a folyamat egyetlen lapkeret nélkül indul, az operációs rendszer csak a page fault esetén tölti be a lapokat.

A módszert használva viszonylag kevés page fault után megjelenik a memóriában a szükséges memóriaterület (lokalitás).




7. ábra. A használt lapkészlet

# A használt lapkészlet

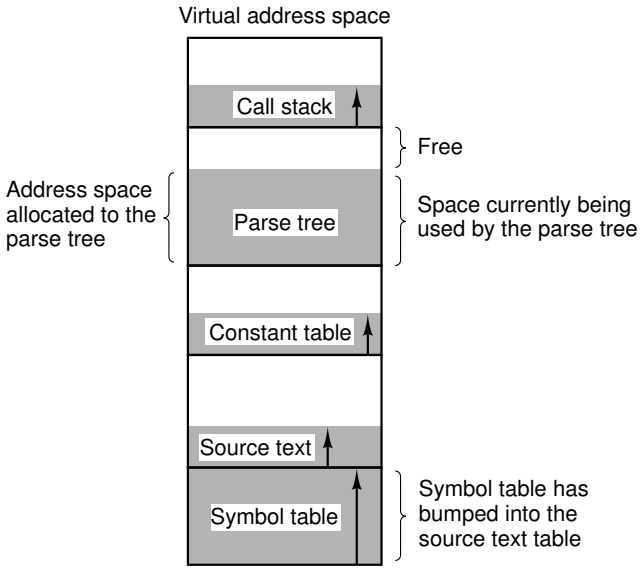
A working set az a laphalmaz (lapkészlet) amely az operatív memóriában az adott pillanatban megtalálható.

A használt lapkészlet és a teljes lapkészlet arányától függően kevés vagy sok (*trashing*, lapcséplés) page fault jelentkezik.

A lapkészlet modellt használó rendszerek megpróbálják elkerülni a lapcséplést például azzal, hogy a processz használt lapkészlete a fizikai memóriában kerül mielőtt az futna (*prepaging*).

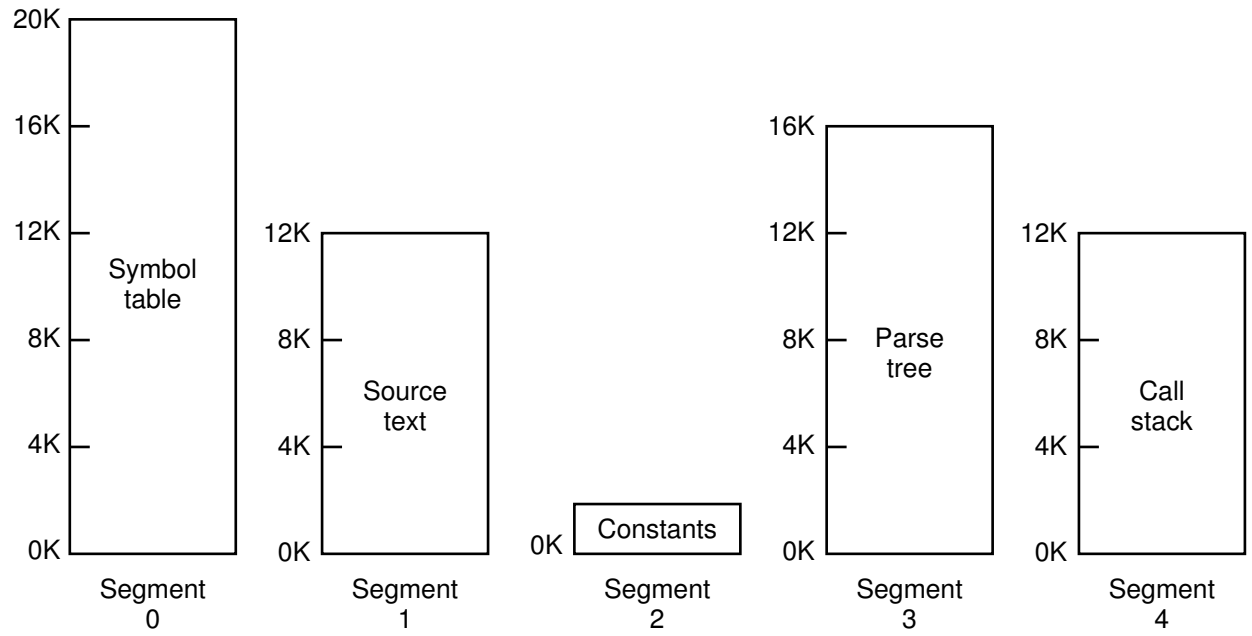
# A szegmentáció

Az eddig tárgyalt virtuális memória egydimenziós, a virtuális címtartomány 0-tól egy bizonyos értékig tart. Hasznosnak tűnik több virtuális címtartományt készíteni.




8. ábra. Fordítóprogram címtartománya

# A szegmentáció




9. ábra. Fordítóprogram szegmentációval