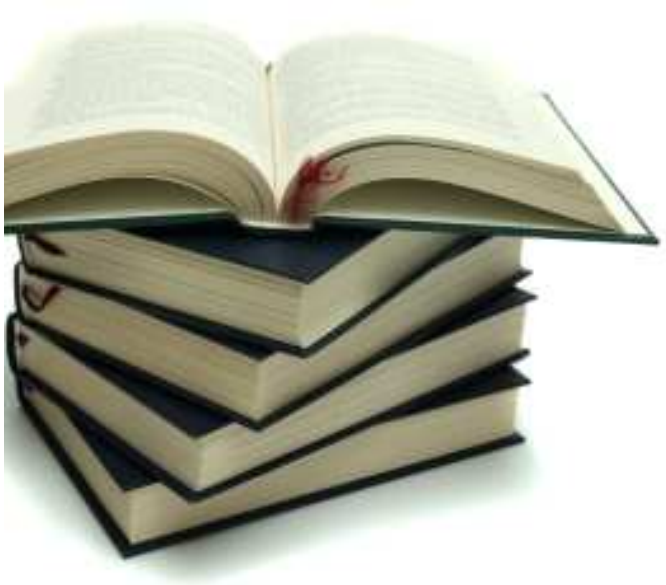
page crunch

Cursul 7



# 7

# Memoria virtuală

8 aprilie 2009

# Suport curs 7

- OSC
  - Capitolul 9 – Virtual Memory
- MOS
  - Capitolul 4 – Memory Management
    - Secțiunile 4.4, 4.5
- Linkers and Loaders
  - Capitolul 8 – Loading and Overlays
    - Secțiunile 8.1 – 8.4
  - Capitolul 10 – Dynamic Linking and Loading
    - Secțiunile 10.1 - 10.4

# Cuprins

- De ce memorie virtuală?
- Demand paging
- Page fault
- Copy-on-write
- Maparea fișierelor
- Algoritmi de înlocuire de pagină
- Execuția programelor

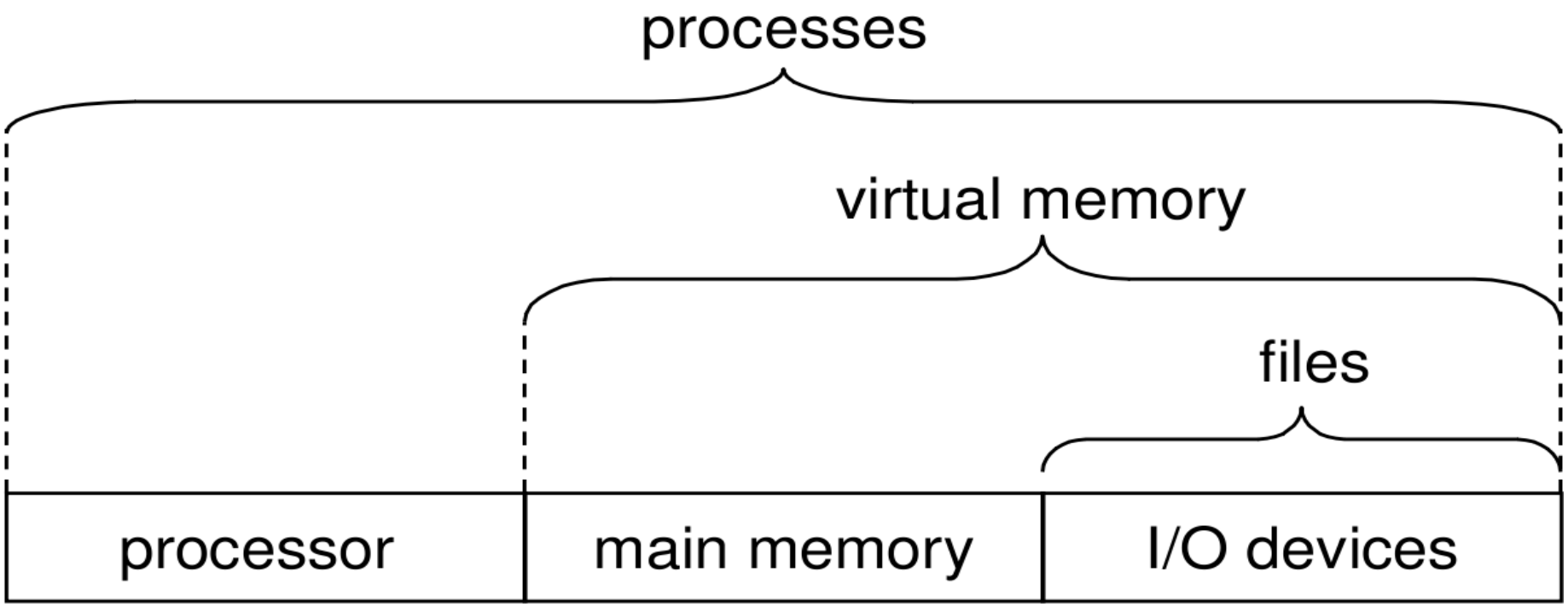
# De de memorie virtuală?

- Procesele pot folosi mai multă memorie decât este disponibilă
  - Demand paging
  - nu este necesar ca paginile unui proces să fie prezente în RAM
- Fiecare proces deține un spațiu de adresare propriu (4GB)
  - programatorul nu este preocupat de limitările memoriei fizice
- Fără probleme de fragmentare externă

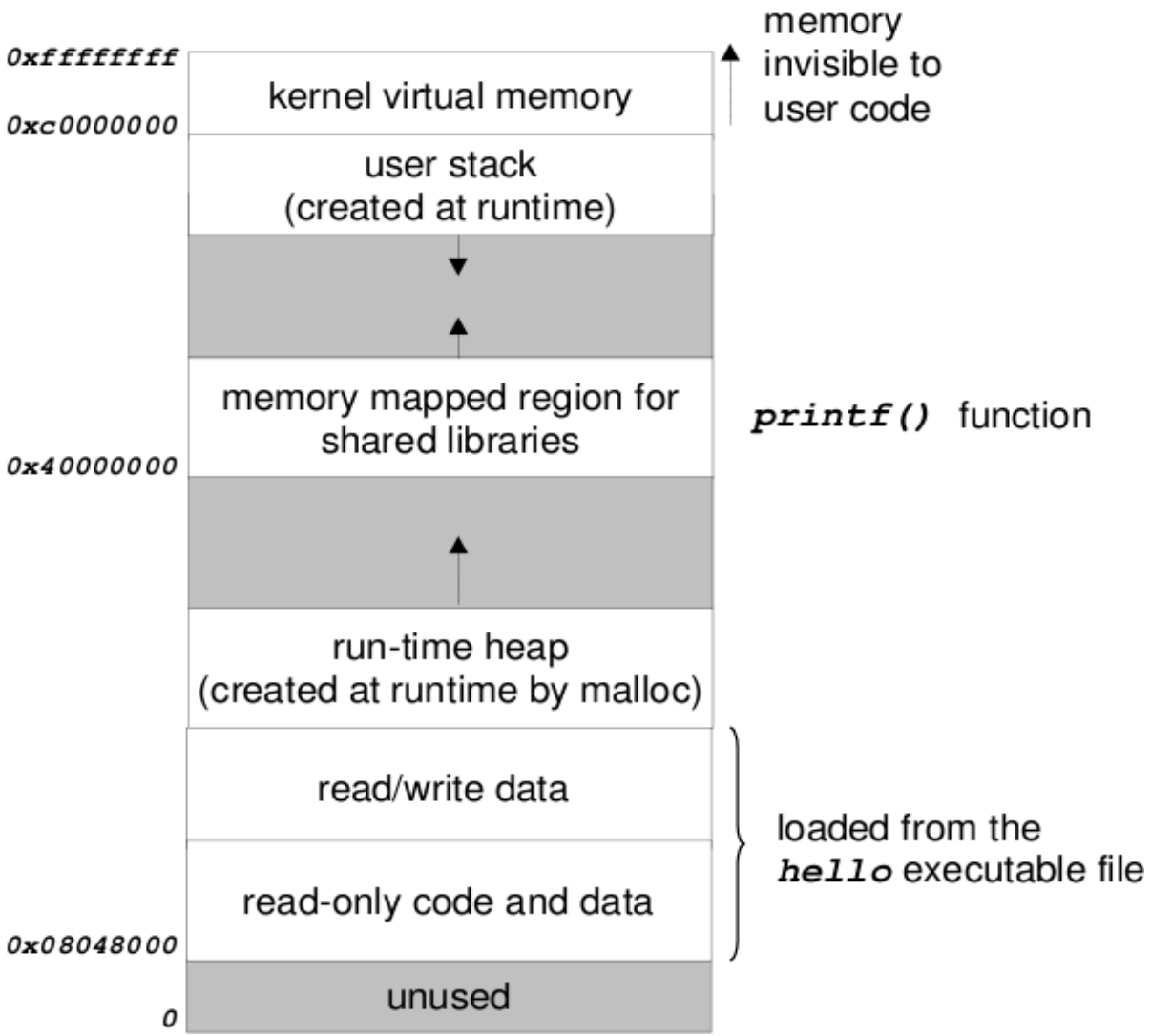
# De ce memorie virtuală? (2)

- Mecanisme eficiente de creare a proceselor
  - Copy-on-write
- Partajarea memoriei
- Partajarea fișierelor
  - memory-mapped files
- Pot exista probleme de performanță

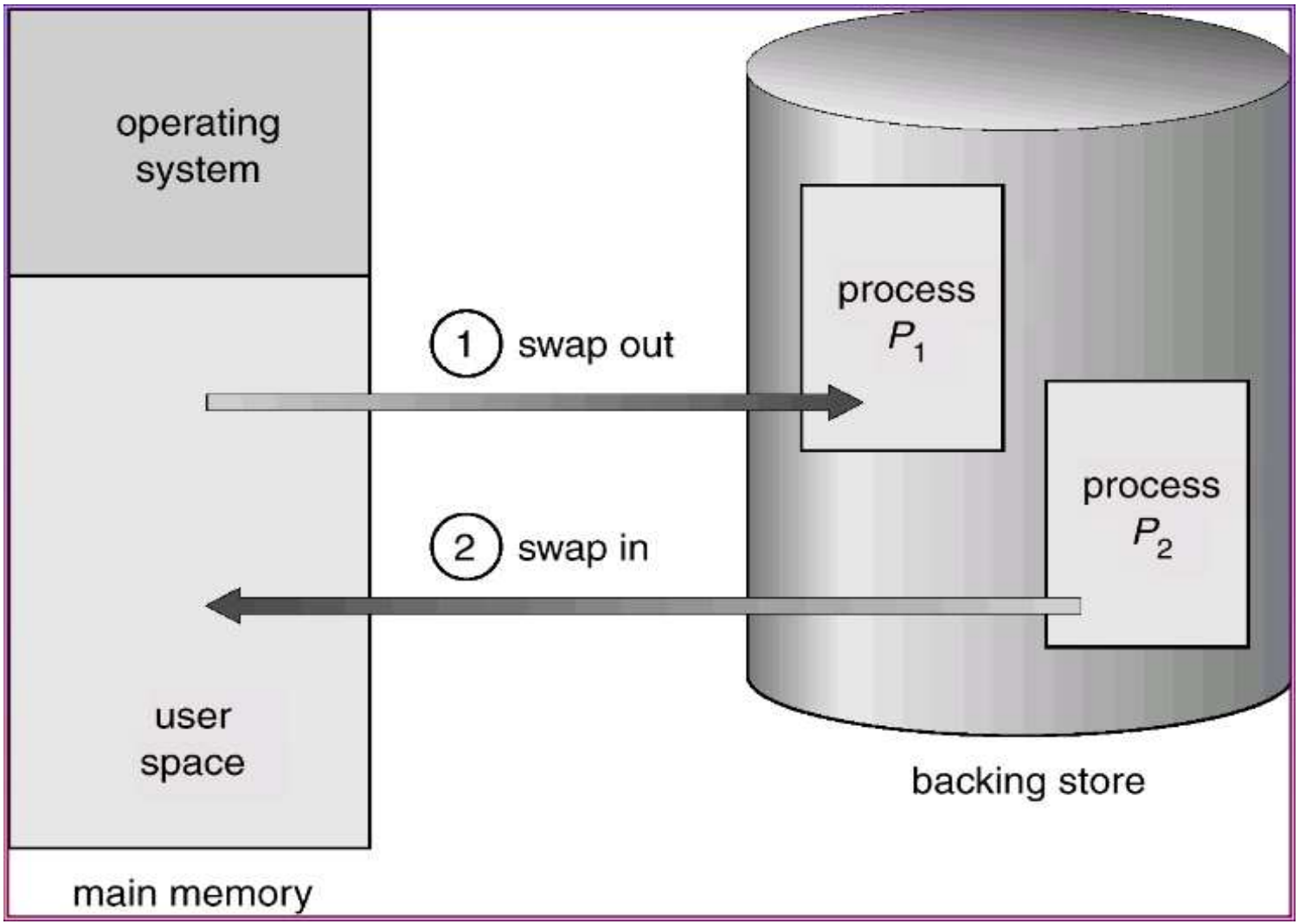
# Abstractizări ale SO



# Spațiul virtual de adrese (Linux)



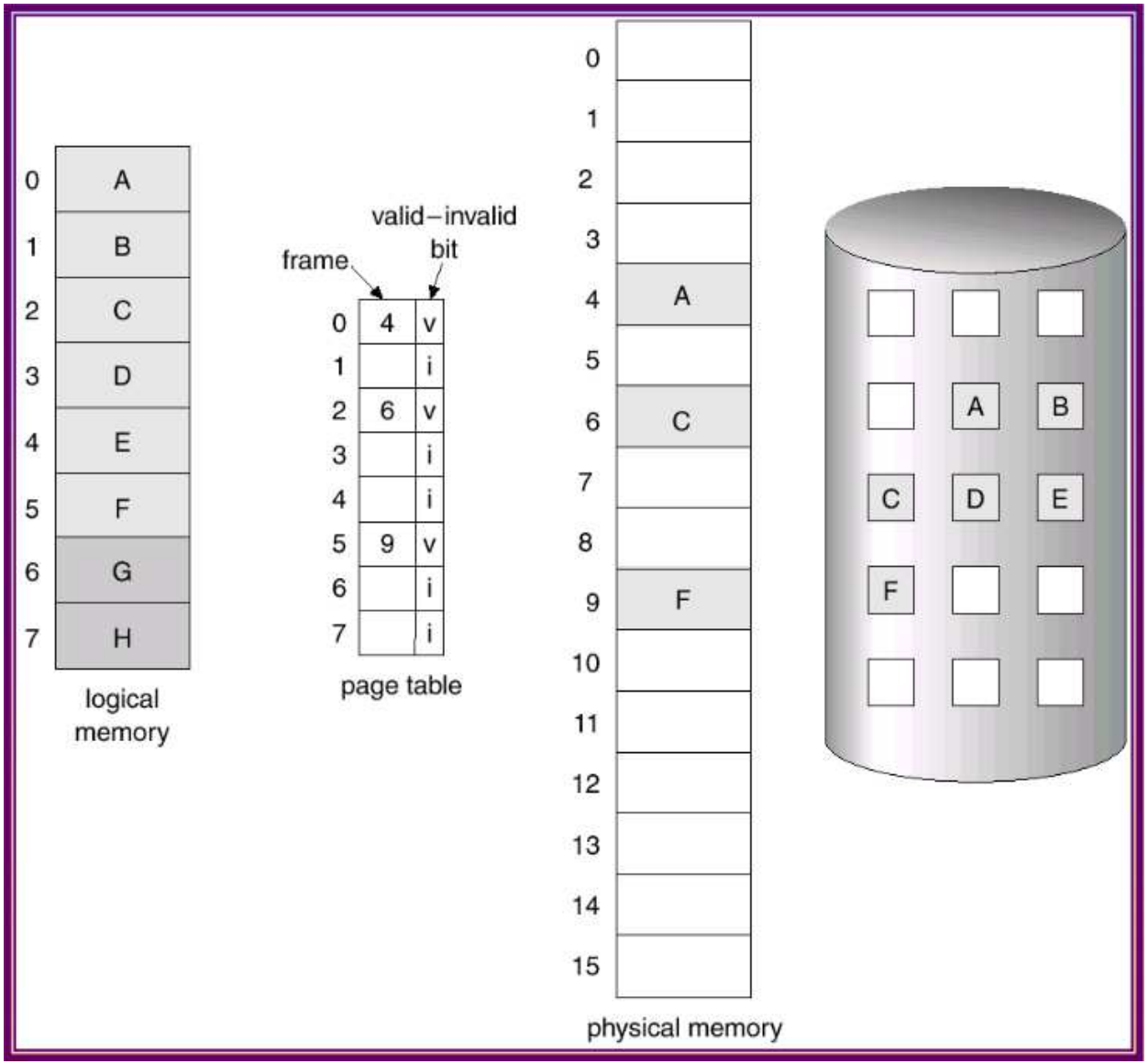
# Spațiul de swap



# Spațiul de swap (2)

- Memoria secundară
- Suport pentru paginile de memorie
  - anumite pagini de memorie pot fi evacuate pe disc
- Paginile vor fi aduse de pe disc în memorie atunci când este necesar
  - algoritmi de înlocuire a paginilor
- Swap in/swap out

# Demand paging



# Demand paging (2)

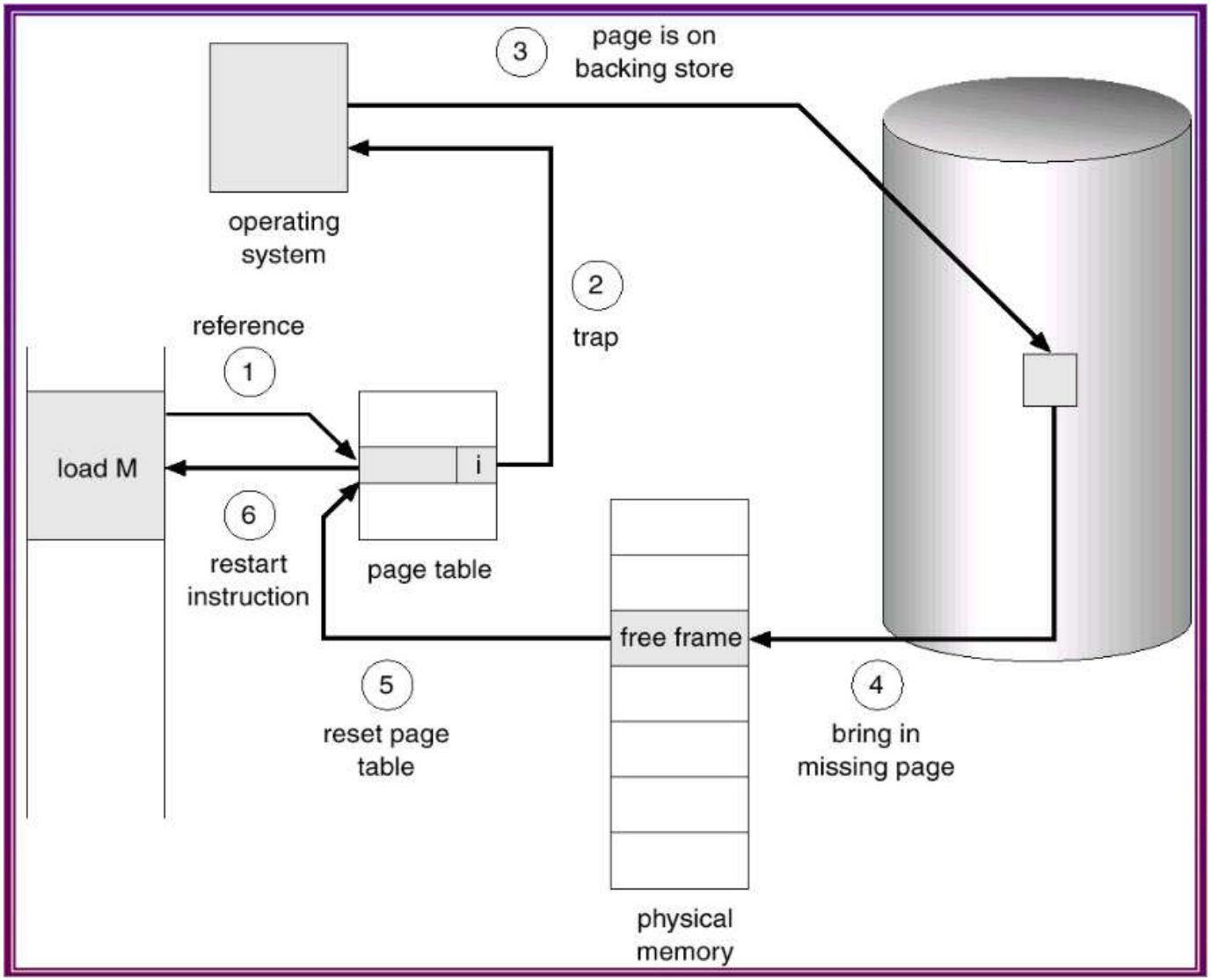
- La încărcarea unui program nu se încarcă tot programul
- Lazy swapper/pager

- Se creează tabela de pagini
- Paginile sunt marcate invalide
- Referirea unei pagini marcate ca invalidă produce o eroare de pagină (page fault)
- Pagina este adusă în memoria

# Demand paging (3)

- Se folosește chiar dacă există memorie suficientă
  - diminuează timpul de creare a procesului
  - la început este posibil să se genereze mai multe page-fault-uri <- prepaging
- Trei tipuri de intrări de pagini
  - pagini valide rezidente
  - pagini marcate invalide dar prezente în swap – demand paging
  - pagini invalide

# Page fault


3
page is on
backing store
operating
system
2
trap
reference
1
load M
i
6
restart
instruction
page table
free frame
5
reset page
table
4
bring in
missing page
physical
memory

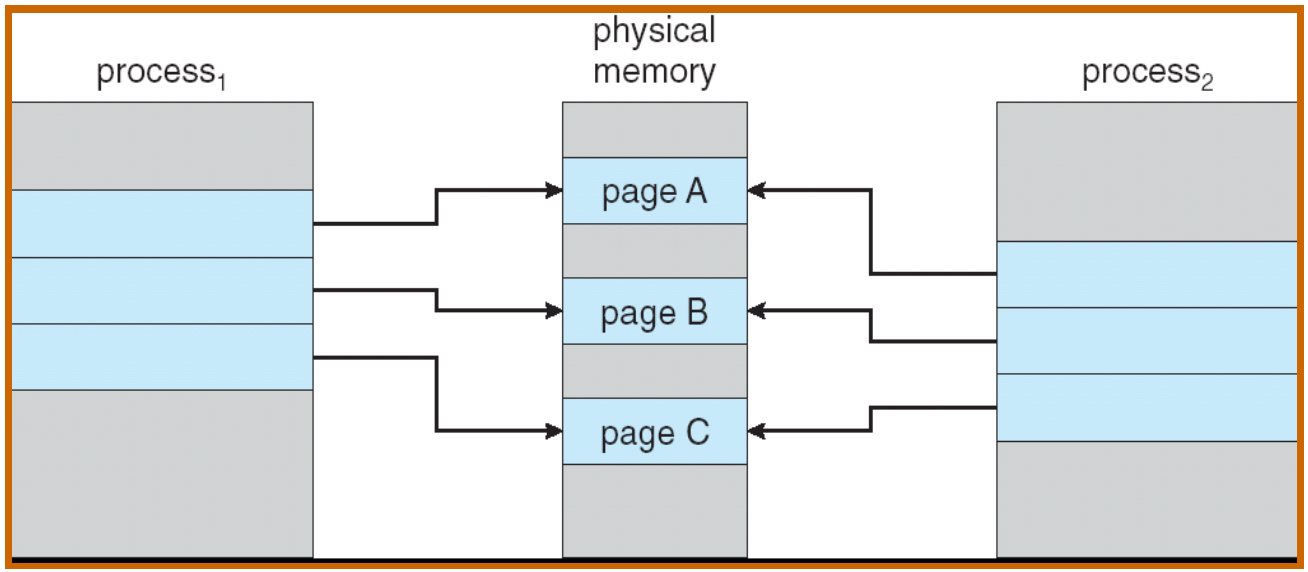
# Page fault (2)

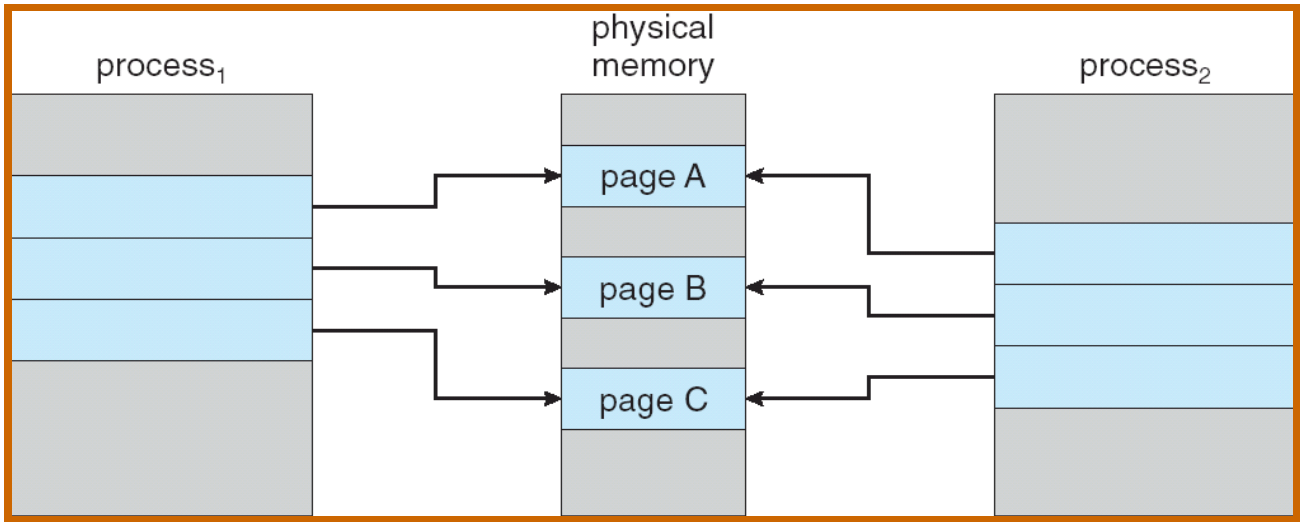
- Excepție cauzată de absența unei pagini fizice
- Verificare tabelă de pagini
  - referință invalidă -> abort
  - pagina nu este în memorie
- Căutare pagină fizică liberă
- Aducere pagină din swap (swap in)
- Resetare tabelă (bitul de validitate activ)

# Copy-on-write

- fork()
  - se creează un spațiu de adresă nou pentru procesul copil
  - se realizează o copie a procesului părinte
  - se consumă timp
  - de obicei se execută exec() -> copierea a fost inutilă

# Copy-on-write (2)

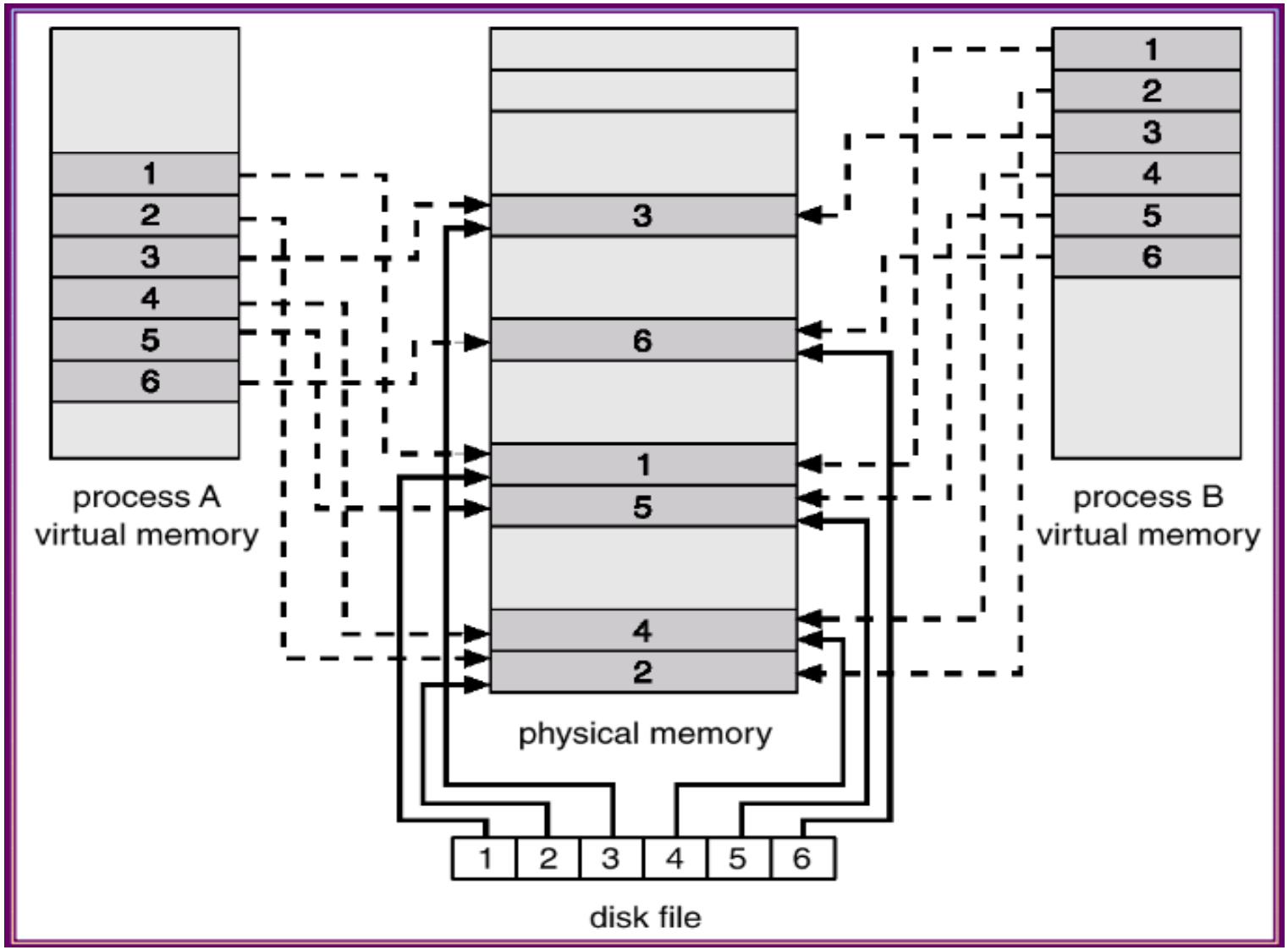


# Copy-on-write (3)

- Se partajează paginile între procesul fiu și procesul părinte
  - paginile sunt marcate read-only
- În momentul unui acces de tip scriere se generează page fault
- Se creează o copie a paginii în procesul copil configurată read-write
- Pagina procesului părinte e configurată read-write

# Maparea fișierelor în memorie



# Maparea fișierelor în memorie (2)

- Un bloc de date al unui fișier este mapat într-o pagină/set de pagini
- Inițial se folosește demand paging
- Lucrul cu fișiere se realizează prin operații de acces la memorie
  - scrierea la o adresă de memorie înseamnă scrierea în fișier
  - scrierile nu sunt imediate/sincrone (se folosesc apeluri msync)
  - memoria este folosită pe post de cache

# Maparea fișierelor în memorie (3)

- Nu este nevoie de apeluri de sistem read/write
- Se pot partaja date între procese
  - pagina virtuală a fiecărui proces referă aceeași pagină fizică

# Maparea fișierelor în Linux

- Apelul mmap
  - `mmap (NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0)`
  - dacă se folosește MAP_ANONYMOUS nu există fișier de suport
- Poate fi folosit pentru partajarea datelor
- munmap/msync/mprotect

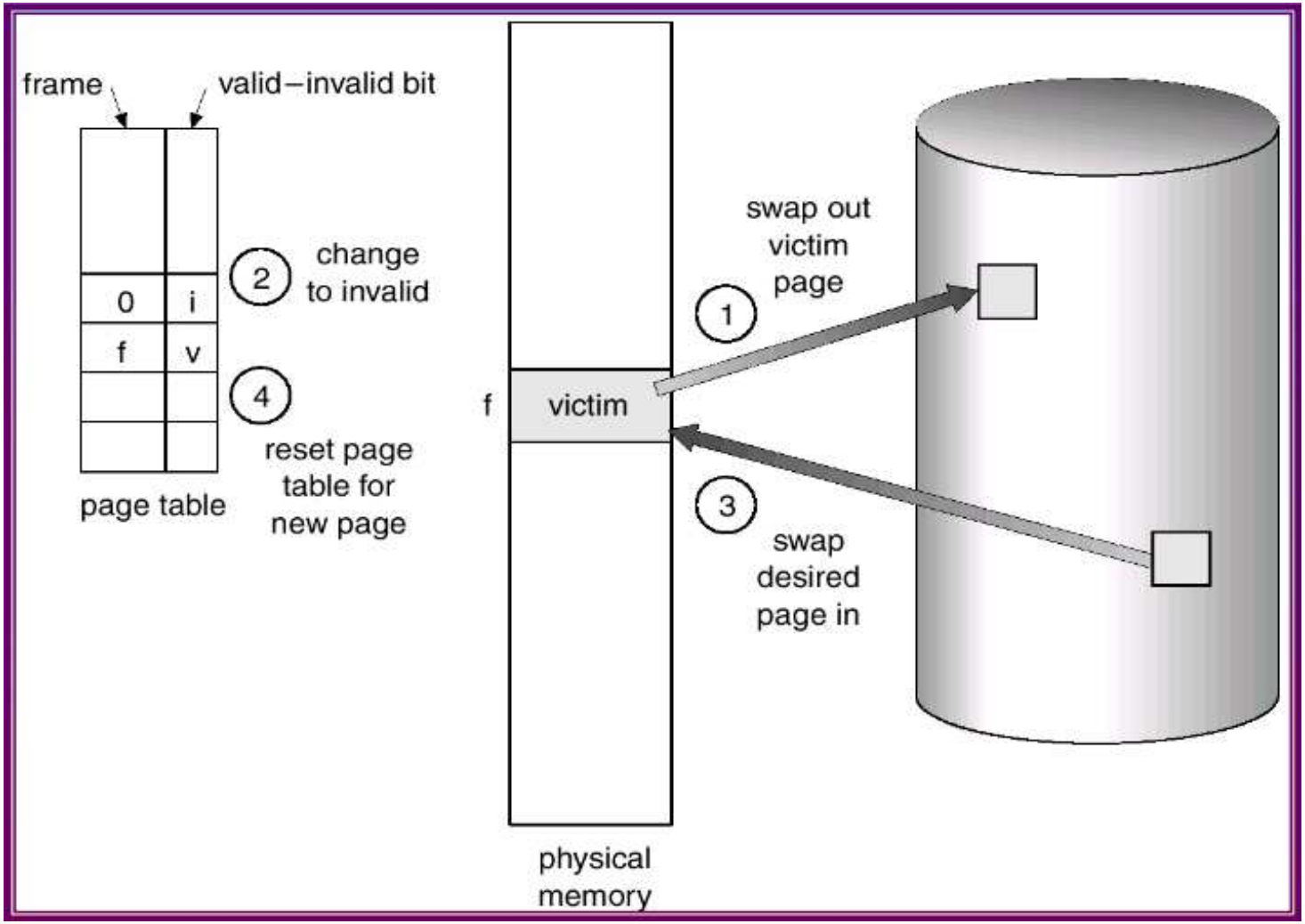
# Maparea fișierelor în Windows

- Folosită și pentru implementarea memoriei partajate
- CreateFileMapping
  - dacă se folosește INVALID_HANDLE_VALUE, nu se folosește fișier de suport
- MapViewOfFile
- UnmapViewOfFile

# Algoritmi de înlocuire a paginilor

- Situație
  - se generează page fault
  - nu există frame liber
  - ce frame este eliberat?
- Algoritmul optim, Not Recently Used
- FIFO, Least Recently Used
- Second Chance, Algoritmul ceasului
- Working Set, WSClock

# Înlocuirea paginii



# Analiza algoritmilor de înlocuire a paginii

- Se folosește un șir de referință
- Un număr din șir reprezintă o pagină logică accesată
- Se contorizează page-fault-urile
- Un page fault apare dacă se accesează o pagină absentă din memorie

# Algoritmul optim

- Teoretic (similar cu SJF)
- Fiecare pagină are asociat numărul de accese ale programului până la referirea acesteia
- Se înlocuiește pagina care va fi referită cel mai târziu

# Not recently used

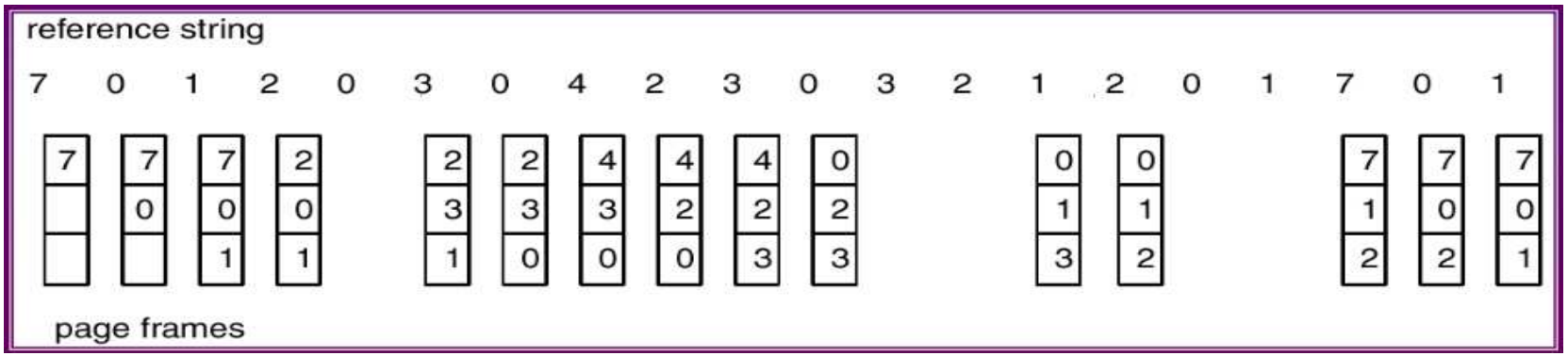
- Fiecare pagină are asociați biții R (referenced) și M (modified)
- Pagina inițial inaccesibilă
- La page-fault se marchează read-only și se activează bitul R
- La scriere se generează un nou page-fault
  - se marchează pagina read-write
  - se activează bitul M

# Not recently used (2)

- Bitul R este dezactivat periodic (cleared)
  - Se deosebesc paginile referite recent
- 4 clase de pagini
  - Clasa 0: R=0, M=0
  - Clasa 1: R=0, M=1
  - Clasa 2: R=1, M=0
  - Clasa 3: R=1, M=1
- Se înlocuiește o pagină din clasa cea mai mică

# FIFO

- Se menține o listă cu paginile din memorie
- Noile pagini sunt adăugate la sfârșitul listei
- Se înlocuiește prima pagină din listă (cea mai veche)



# Anomalia lui Belady


Șirul de referință: 1,2,3,4,1,2,5,1,2,3,4,5

# Second chance

- Variantă modificată a FIFO
- Se ține cont de bitul R
- Se inspectează prima pagină din listă
  - dacă R=0 pagina este selectată pentru înlocuire
  - dacă R=1, pagina este mutată la coada listei și R=0

# Algoritmul ceasului

- Similar cu second chance
- Paginile sunt ținute într-o listă circulară
- La page fault se analizează pagina din dreptul “brațului”
  - R=0, pagina este selectată pentru înlocuire
  - R=1, se resetează R și se avansează “brațul”

# Least recently used

- Se selectează pagina referită de cele mai puține ori
- Fiecărei pagini i se asociază un contor de utilizare
  - contorul este incrementat la fiecare instrucțiune
- Contor mai mic înseamnă pagină referită de mai puține ori

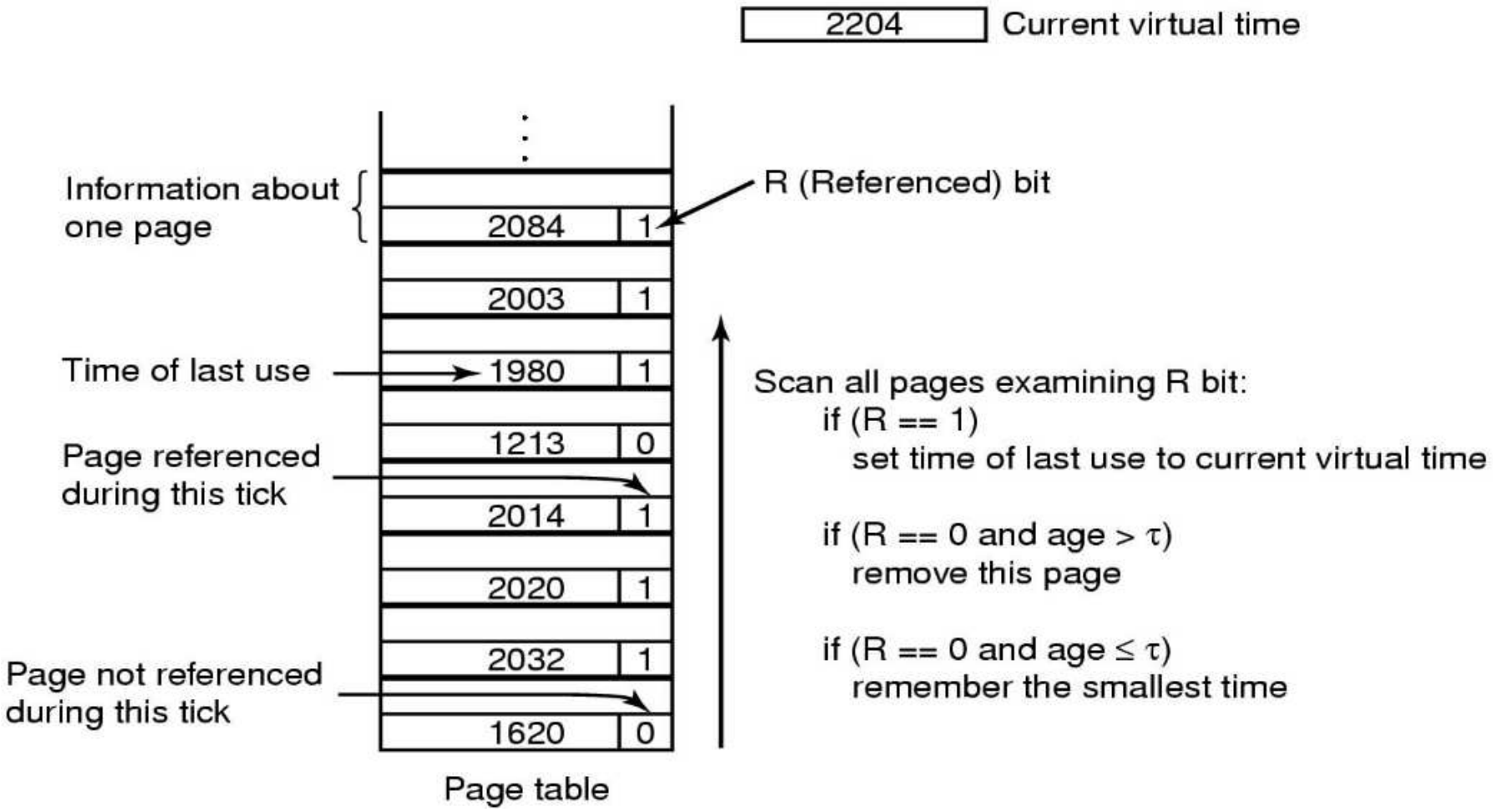
# LRU-variante

- Not Frequently Used
  - la fiecare întrerupere de ceas
    - se scanează toate paginile din memorie
    - se adaugă la contor bitul R
- Aging
  - la fiecare întrerupere de ceas
    - se scanează toate paginile din memorie
    - se deplasează contorul la dreapta cu un bit
    - se setează bitul cel mai semnificativ pe valoarea lui R

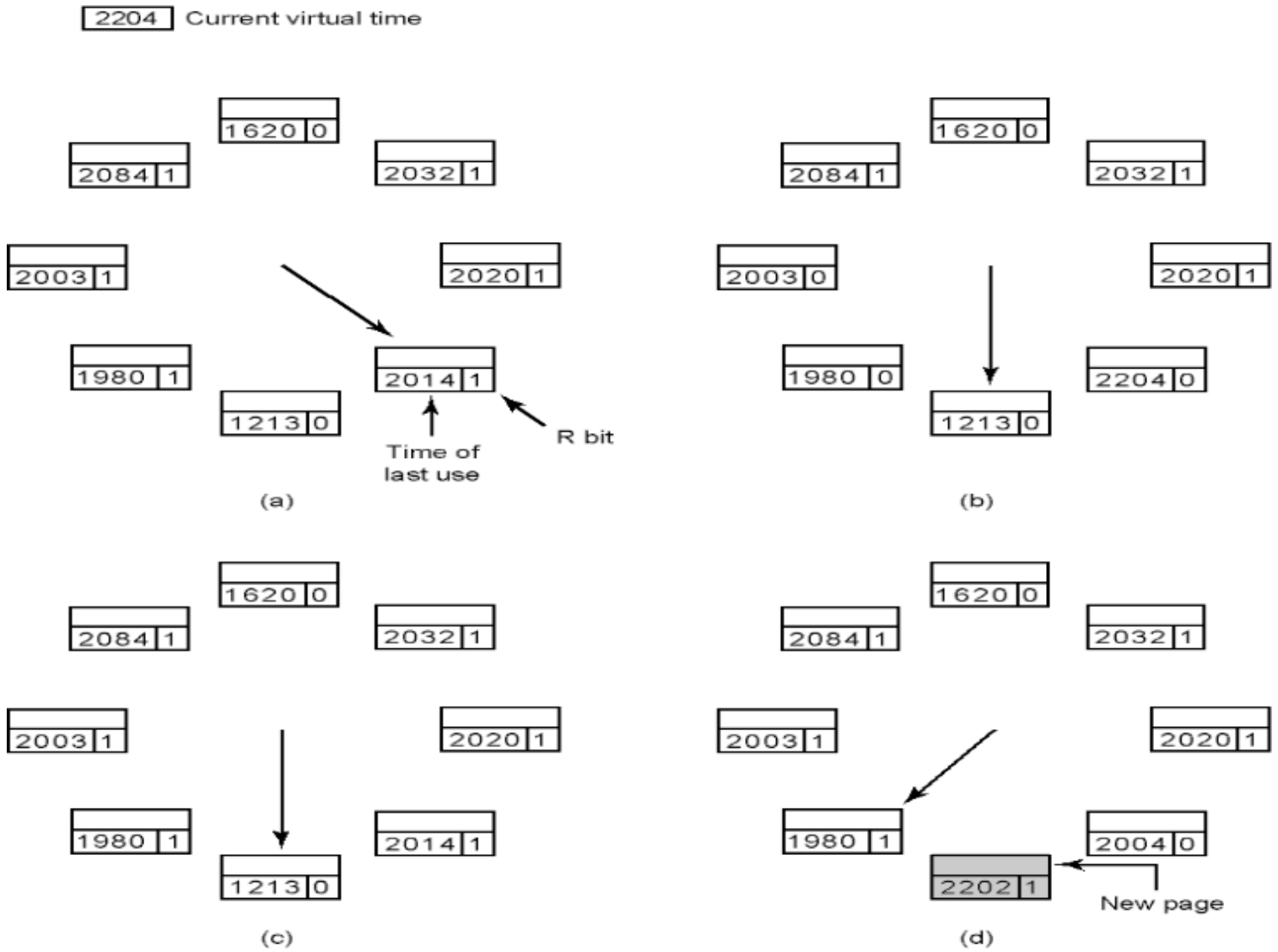
# Working set

- Working set
  - W(t, T) – paginile accesate de un proces în intervalul (t-T, t)
- Se marchează bitul R pentru paginile referite
- La fiecare întrerupere de ceas se dezactivează bitul R
- La page fault se scanează toate paginile
  - dacă R=1 se stochează într-un câmp (time of last use) asociat paginii timpul virtual al procesorului
  - dacă R=0 și age = timpul virtual curent – time of last use > T se selectează pagina pentru evacuare
  - dacă R = 0 și age = timpul virtual curent – time of last use < T, pagina nu este evacuată (decât dacă celelalte pagini sunt în aceeași situație și pagina are cel mai mic time of last use)

# Working set (2)



# WSClock


2204 Current virtual time
1620 0
2084 1
2032 1
2003 1
2020 1
1980 1
2014 1
1213 0
Time of last use
R bit
(a)
1620 0
2084 1
2032 1
2003 0
2020 1
1980 0
2204 0
1213 0
(b)
1620 0
2084 1
2032 1
2003 1
2020 1
1980 1
2014 1
1213 0
(c)
1620 0
2084 1
2032 1
2003 1
2020 1
1980 1
2004 0
2202 1
New page
(d)

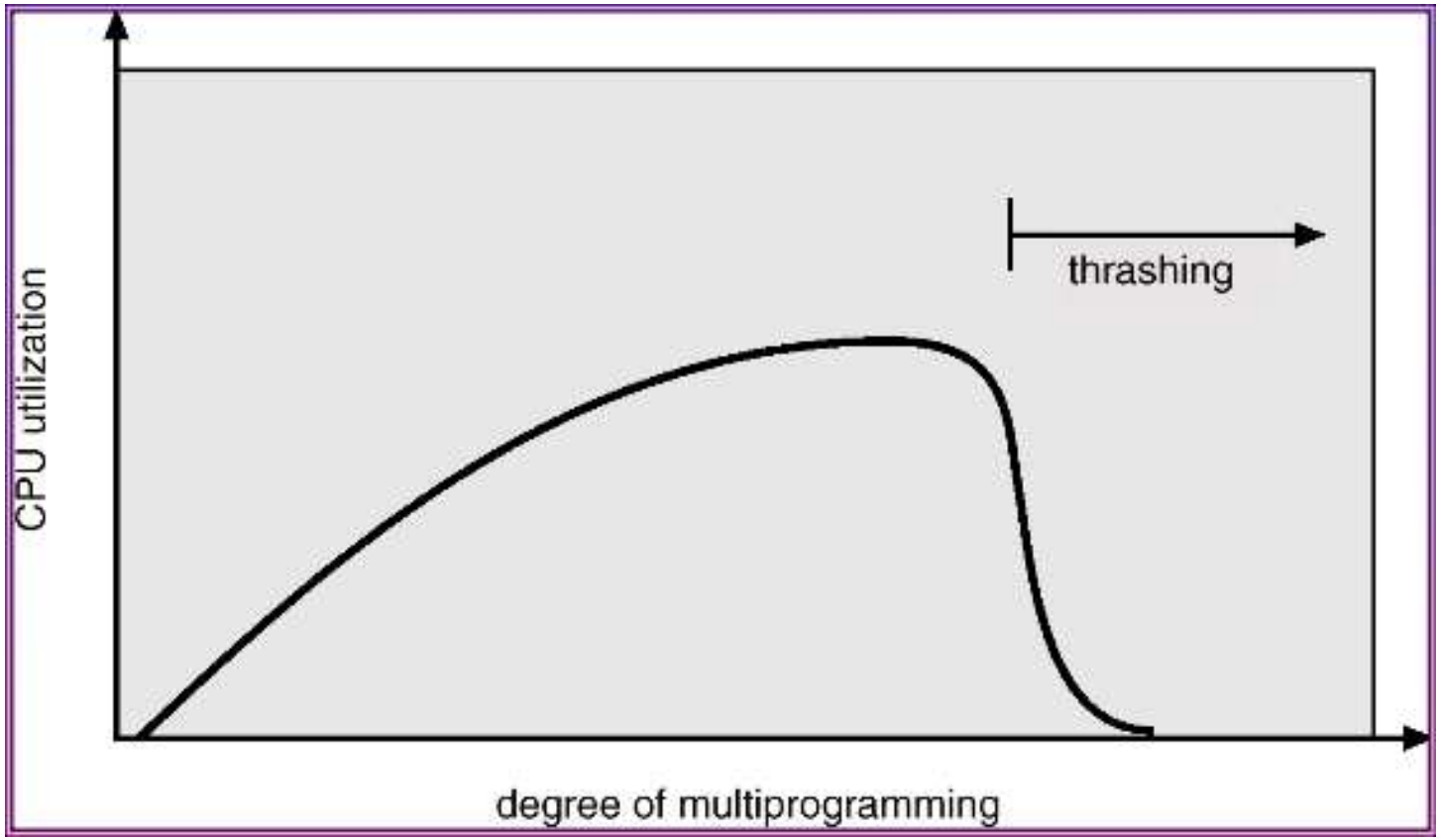
# Algoritmi de înlocuire a paginilor

- Optim – neimplementabil, util în testare
- NRU – primitiv
- FIFO – poate suferi de anomalia lui Belady
- Second change – FIFO îmbunătățit
- Ceasului – mai eficient decât second chance
- LRU – excelent, dificil de implementat fără hardware
- NFU – aproximare primitivă a LRU
- Aging – aproximare eficientă a LRU
- Working Set – algoritm bun, dar costisitor
- WSClock – înlătură problemele de performanță a Working Set

# Thrashing

- Dacă un proces nu are destule pagini, rata page-fault este ridicată
  - utilizare scăzută a procesorului
  - SO consideră necesară creșterea nivelului de multiprogramare
  - se adaugă un nou proces în sistem
- Thrashing
  - un proces este ocupat cu acțiuni de swap in/out ale paginilor proprii

# Thrashing



# Execuția programelor

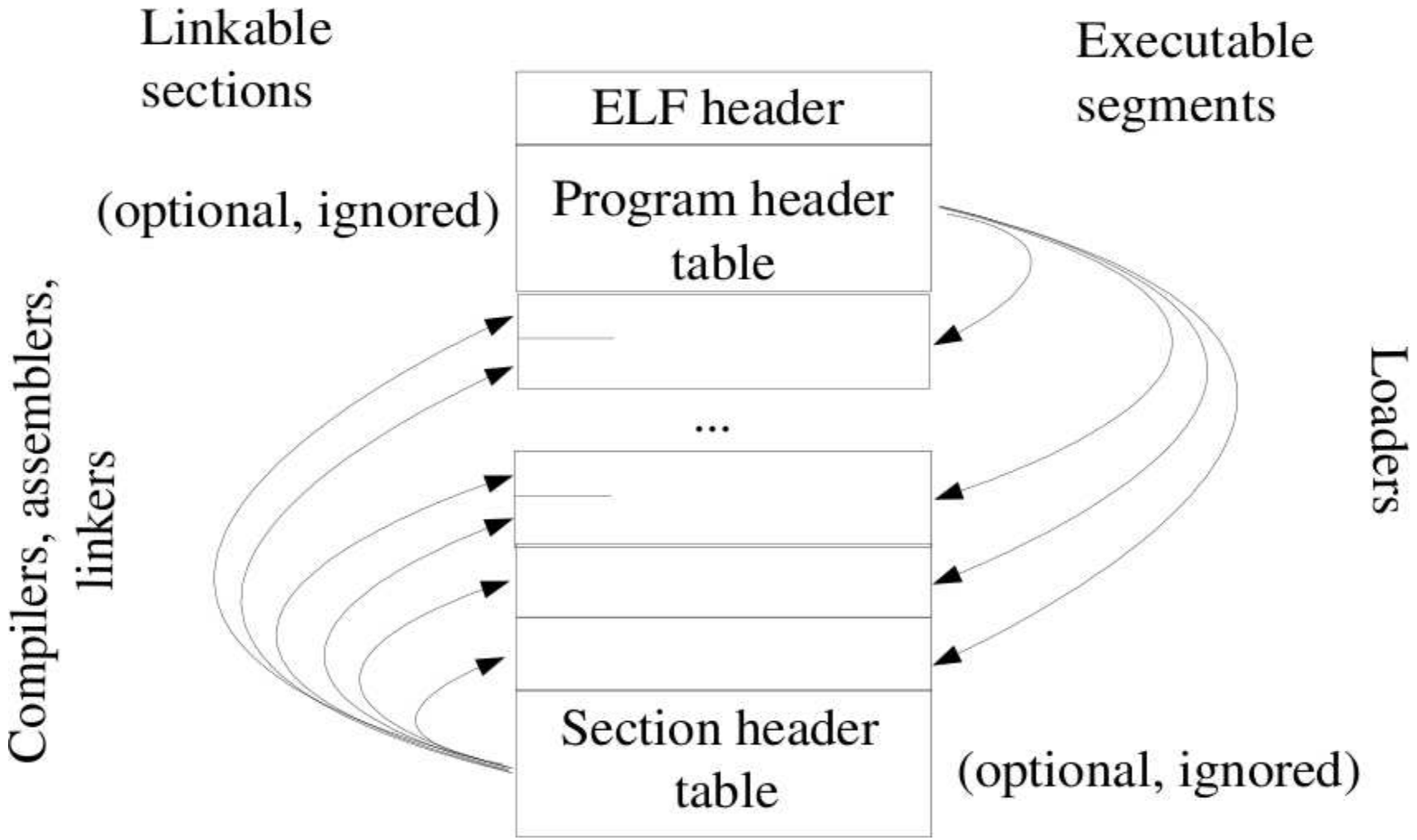
- Încărcarea programului în memorie se realizează prin maparea fișierului
  - segmentul de date read-write
  - segmentul de cod (text) – read-only (poate fi partajat)
- Este important formatul fișierului executabil

# Formatul a.out

- Utilizat pe majoritatea sistemelor Unix înainte de apariția ELF
- Antetul a.out:

```
int a_magic; // magic number
int a_text;  // text segment size
int a_data;  // initialized data size
int a_bss;   // uninitialized data size
int a_syms;  // symbol table size
int a_entry; // entry point
int a_trsize;  // text relocation size
int a_drsize;  // data relocation size
```

# Formatul ELF



# Formatul ELF (2)

- Executable and Linking Format
- Compilatoarele și asambloarele lucrează cu secțiuni
- Secțiunile sunt grupate în segmente
- Loader-ul mapează segmente ale fișierului în memorie

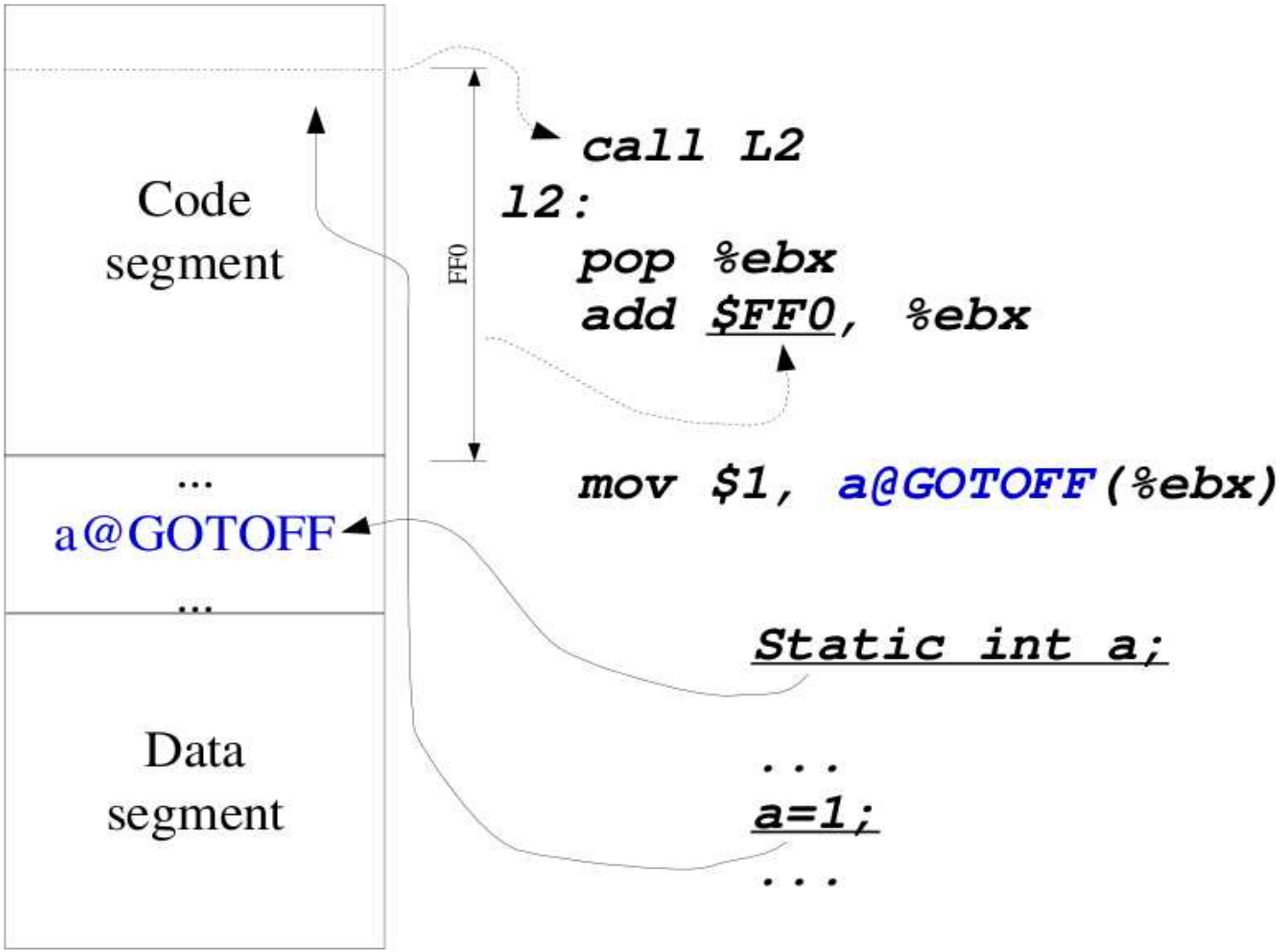
# Relocări ELF

- Relocare la linking
  - o secțiune cu relocări are asociată o secțiune de relocare
- Relocare la încărcare
  - relocare pentru obiecte partajate
    - relocări în segmentele .got și .plt
  - relocare pentru obiecte nepartajate
    - relocări în segmentul .text

# Global Offset Table

- Se dorește partajarea unor obiecte comune (biblioteci)
  - se alocă fiecărei biblioteci o zonă de memorie
    - nu funcționează cu încărcare dinamică
  - se folosesc relocări
    - nu se pot face relocări în .text sau .data
    - compilatorul generează cod independent de poziție (PIC) care accesează datele din GOT
    - relocarea se face în GOT

# Position Independent Code



# Procedure Linkage Table

- Tabelă de salturi prin indirectare
  - valoarea saltului este determinată din GOT
- Compilatorul folosește tabela pentru a apela proceduri din (alte) biblioteci partajate
- Aflarea efectivă a adresei nu se face la load-time ci la run-time (lazy procedure binding)

# Cuvinte cheie

- memorie virtuală
- spațiu de adresă
- stiva, heap, data, text
- swap
- demand paging
- swapper/pager
- page fault
- copy-on-write
- memory-mapped files
- memorie partajată
- mmap
- NRU, FIFO, LRU
- Working set, WSClock
- thrashing
- a.out, ELF
- GOT
- PIC
- PLT

# Exercițiu 1

- Dacă timpul copierii unei pagini este 1ms, ignorând timpul necesar altor operații (crearea de structuri interne, alocarea de pid-uri, copierea tabelelor de pagini etc.), care este timpul necesar apelului fork pe un proces cu un spațiu de adresă de 100MB? (pe un sistem de tip UNIX modern, cu dimensiunea unei pagini de 4K)
- Câte pagini virtuale, respectiv fizice pot partaja două procese? Dar două thread-uri?

# Exercițiu 2

- Care este numărul minim de page-fault-uri care vor avea loc în urma rulării următorului program? (o pagină ocupă 4 KB)

```
int *p, i, status;

p = (int *) mmap (NULL, sizeof(int) * 1024  * 1024, PROT_READ|PROT_WRITE, MAP_PRIVATE, 0, 0);
for (i = 0; i < 1024; i++)
        p[i*1024] = i;
switch(fork()) {
case -1:    /handle error */
case 0:     /* child process */
        for (i = 0; i < 512; i++)
                printf("p[%d] = %d\n", i*1024, p[i*1024]);
        for (i = 512; i < 1024; i++)
                p[i*1024] = 1024–i;
        exit(EXIT_SUCCESS);
        break;
default:
        break;
}

wait (&status);
for (i = 0; i < 1024; i++)
        p[i*1024] = i;
```

# Întrebări

?