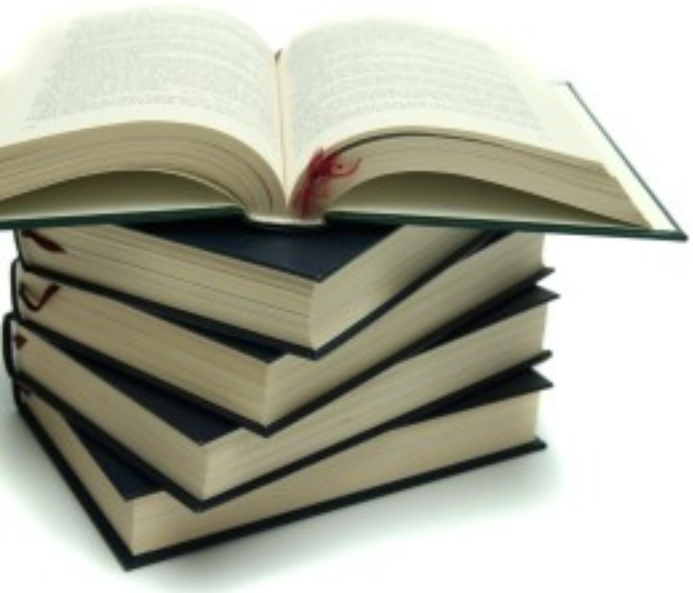
page crunch

Cursul 6



6

Gestiunea memoriei

30 martie 2010

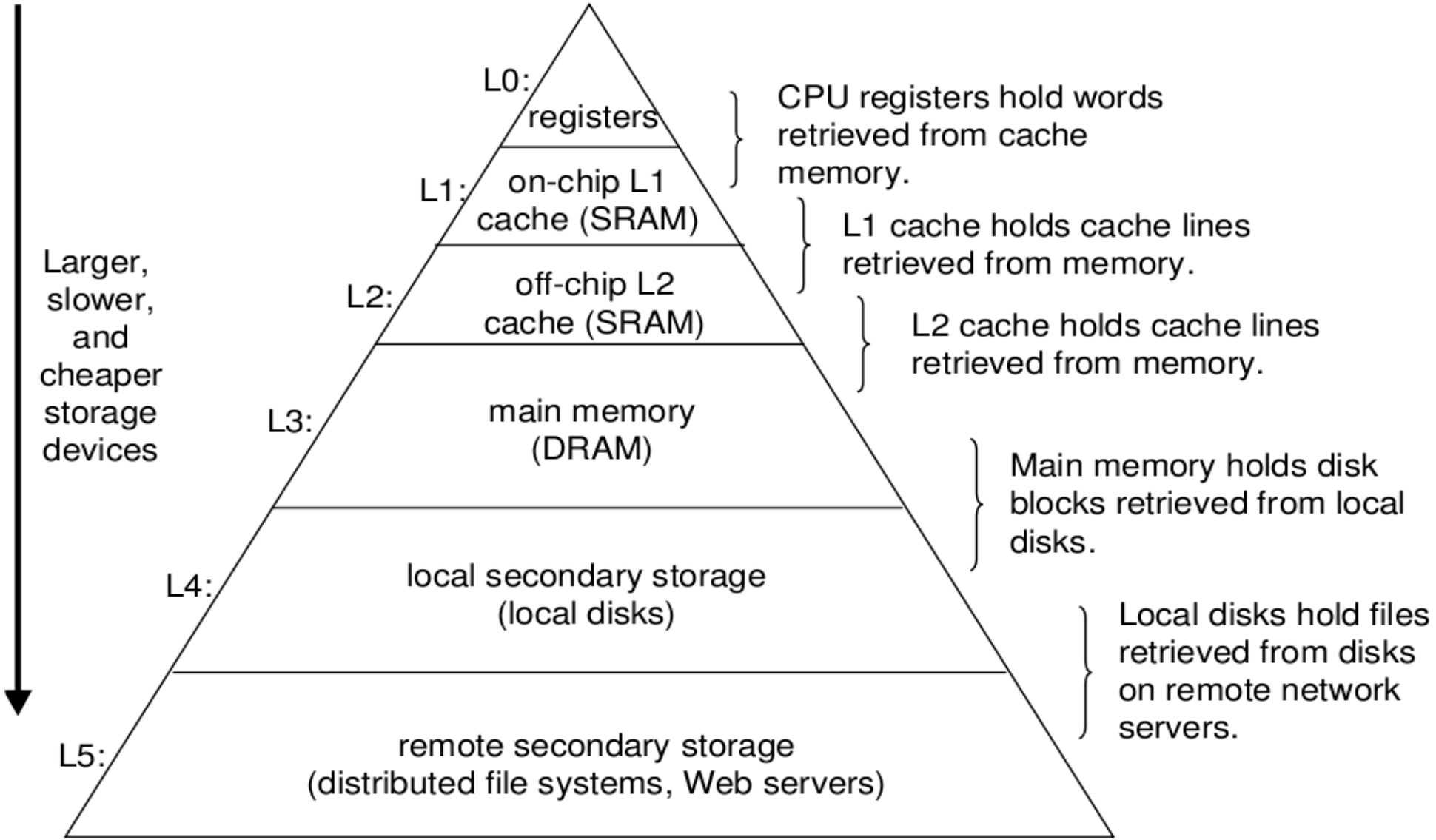
Suport curs 6

- OSC
  - Capitolul 8 – Main Memory
- MOS
  - Capitolul 4 – Memory Management
    - Secțiunile 4.1, 4.2, 4.3, 4.8
- Ulrich Drepper - What every programmer should know about memory

Cuprins

- Gestiunea memoriei și a accesului
- Protecția memoriei
- Asocierea adreselor (binding)
- Alocarea memoriei contigue
- Paginarea
- Segmentarea
- Gestiunea memoriei la x86

Ierarhia memoriei



Memoria principală

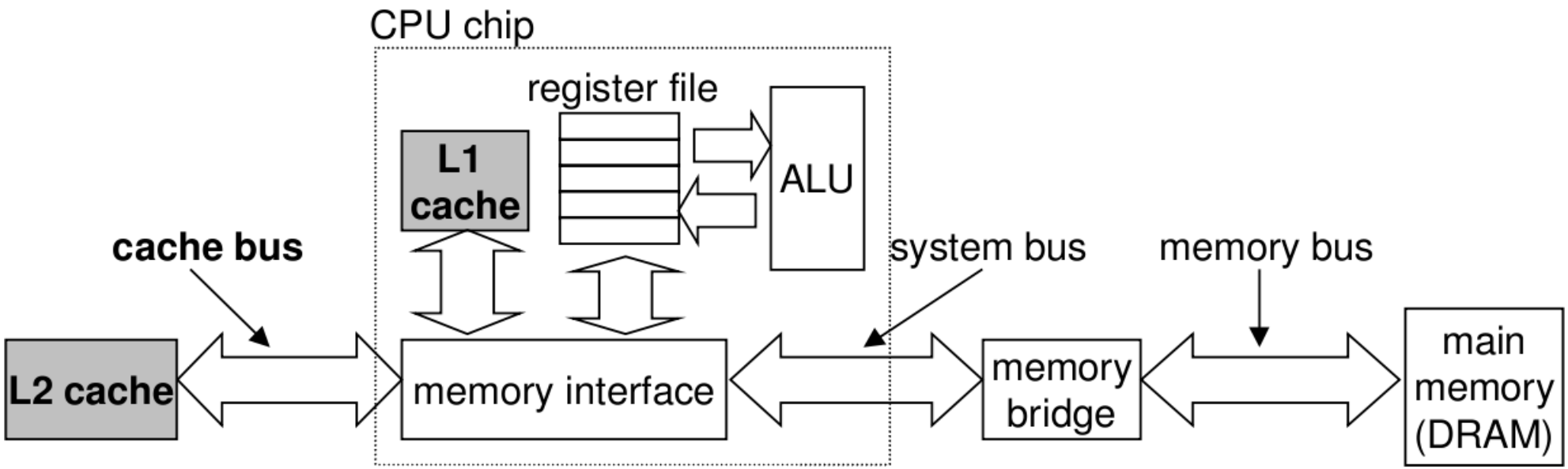
- De obicei RAM (Random Access Memory)
- Menținerea mai multor procese
- Ce tipuri informații sunt menținute în memoria principală?
  - instrucțiuni (cod)
  - date
- Cum se accesează memoria?
  - prin intermediul adreselor de memorie
  - memoria este un vector de octeți, fiecare cu o adresă proprie

Gestiunea memoriei

- Nu interesează modul în care se generează o adresă de memorie (contor de program, adresare directă etc.)
- Interesează secvența de adrese de memorie generate de un program

- Subsistemul de gestiune a memoriei
  - alocarea/dezalocarea memoriei
  - translatarea adreselor virtuale
- Suport hardware

Comunicarea proces-memorie



Accesul la memorie

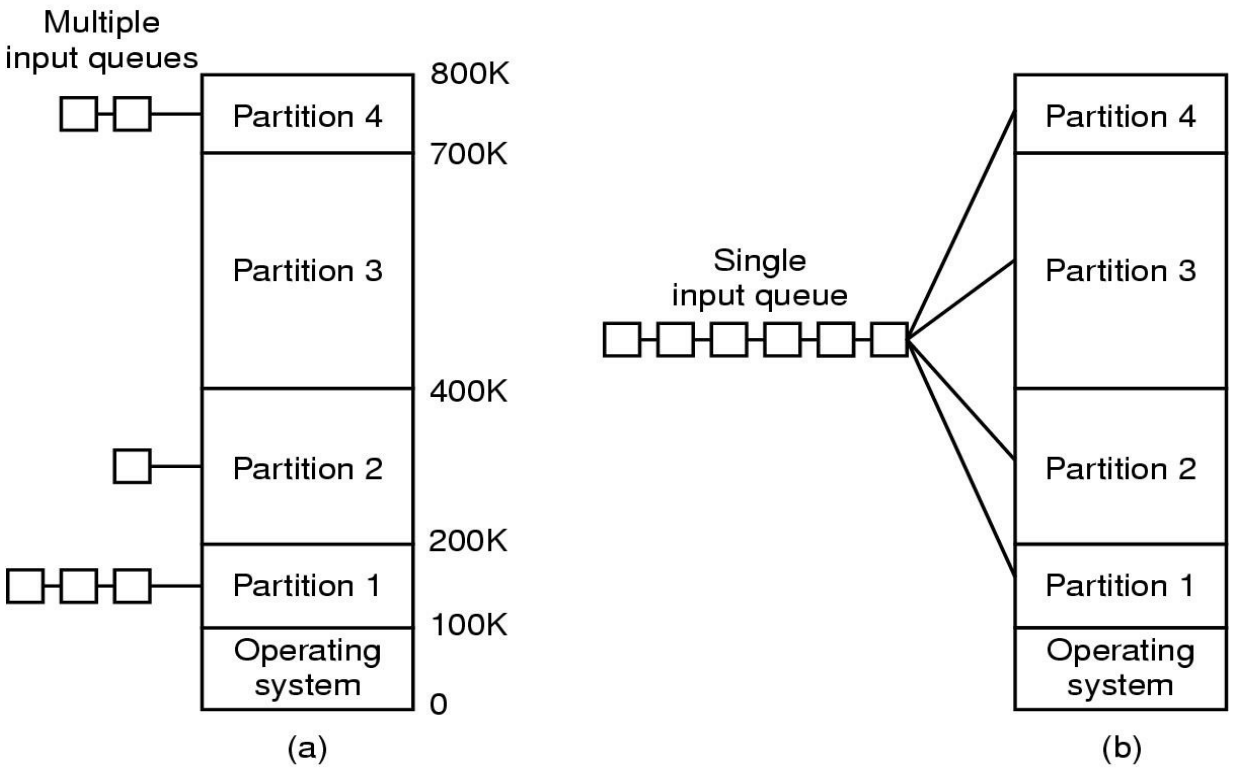
- Anumite arhitecturi permit accesarea directă a memoriei (x86)
- Altele (MIPS) folosesc instrucțiuni de tip load/store
- Valorea citită este stocată într-un registru
- Procesorul poate efectua calcule doar cu registrele
- Registru specializat pentru citirea următoarei instrucțiuni (IP/PC)

Memoria cache

- Cache între procesor și memoria principală (SRAM)
- Fiecare locație din cache are un tag asociat unui set de locații din memoria principală (linii de cache)

- Înainte de accesarea memoriei principale, procesorul accesează memoria cache
  - cache hit/cache miss
  - hit rate
- În caz de miss, se alocă o intrare nouă

Sisteme multiprogramate cu partiții fixe



(a) Cozi pentru fiecare partiție

- Job-urile aşteaptă chiar dacă avem memorie disponibilă

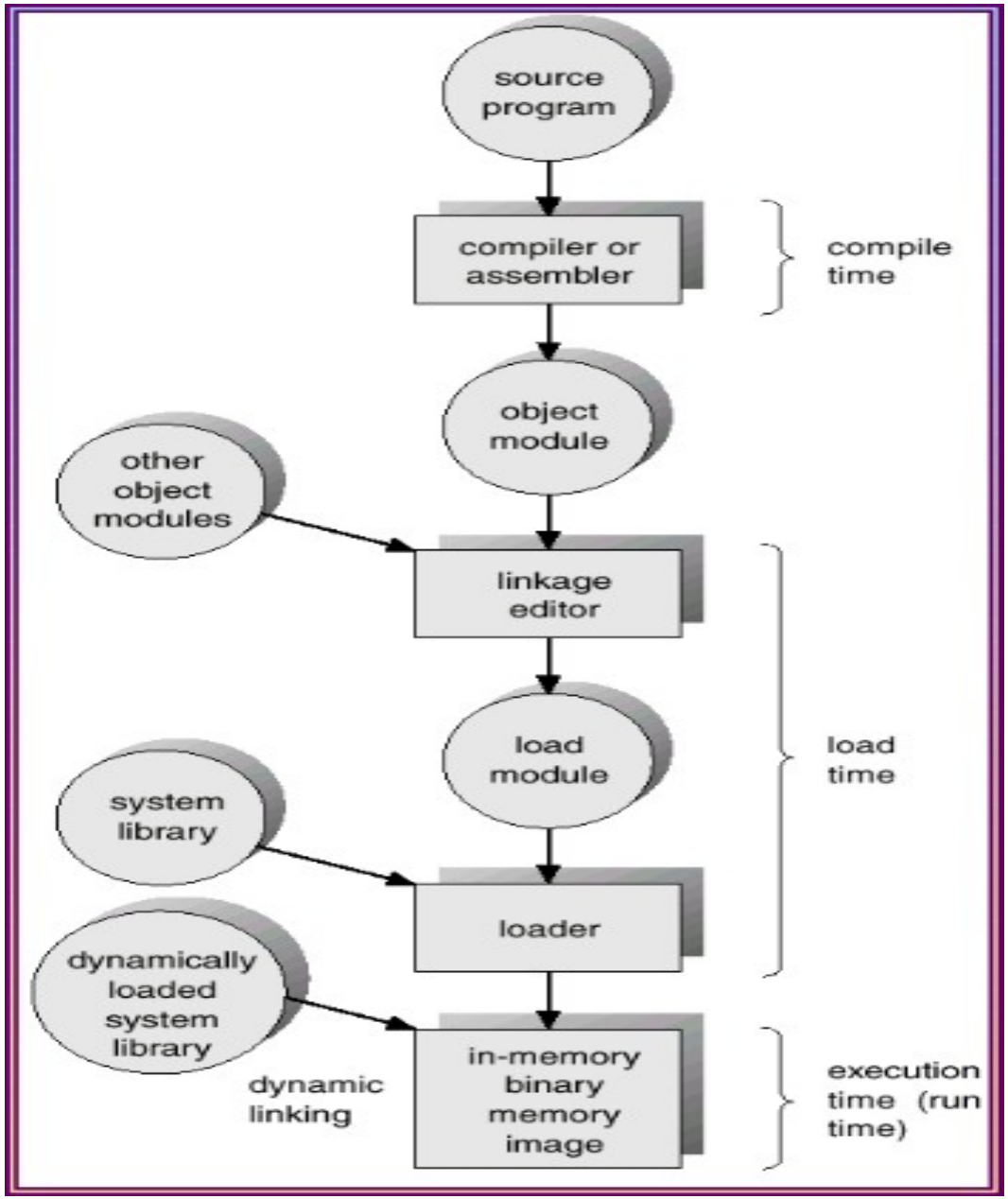
(b) O singură coadă pentru toate partițiile

Address binding

- Adresele asociate unui program vor trece prin diverse faze de “binding” până la încărcarea în memorie
  - compilatorul va asocia un simbol unei adrese relocabile (14 octeți de la începutul modulului)
  - linkerul sau loaderul va asocia adresa relocabilă la o adresă absolută (spre exemplu 74014)

- Asocierea poate fi făcută în diverse faze
  - la compilare – se cunoaște la compilare localizarea în memorie a procesului; se generează cod absolut (.COM)
  - la încărcare – compilatorul generează cod relocabil
  - la execuție – o biblioteca poate fi încărcatț pe parcursul execuției

Address binding (2)



Linking

- Static linking – rutinele bibliotecilor sunt adăugate executabilului
  - fișiere obiect între ele și cu biblioteci statice
  - exemplu: fiecare proces deține o copie a imaginii funcției printf
  - se rezolvă adresa fiecărei funcții la fiecare instrucțiune **call** care o apelează
- Dynamic linking
  - bibliotecile se pot încărca la on-the-fly: necesita relocare la run-time
  - toate procesele folosesc o singură copie a codului bibliotecii – biblioteci partajate
  - necesită suportul sistemului de operare

Relocare la linking

```
#include <stdio.h>

int a;

int main()

{

a=4;

printf("a=%d\n", a);

}
```

```
Sections:
Idx Name          Size      VMA       LMA       File off

  0 .text         00000030  00000000  00000000  00000040
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY,
  1 .data         00000000  00000000  00000000  00000070
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000070
                  ALLOC
  6 .rodata       00000006  00000000  00000000  000026a0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
```

```
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000008 R_386_32          a
00000015 R_386_32          a
0000001a R_386_32          .rodata
0000001f R_386_PC32        printf
```

```
SYMBOL TABLE:
00000000 g     F .text   0000002a main
00000004       O *COM*   00000004 a
00000000         *UND*   00000000
printf
```

```
Contents of section .rodata:
 0000 613d2564 0a00            a=%d..
```

Relocare la linking (2)

```
Disassembly of section .text:

00000000 <main>:

int a;

int main()
{
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 ec 08                sub    $0x8,%esp
        a=4;
   6:   c7 05 00 00 00 00 04    movl   $0x4,0x0
   d:   00 00 00
        printf("a=%d\n", a);
  10:   83 ec 08                sub    $0x8,%esp
  13:   ff 35 00 00 00 00       pushl  0x0
  19:   68 00 00 00 00          push   $0x0
  1e:   e8 fc ff ff ff          call   1f <main+0x1f>
  23:   83 c4 10                add    $0x10,%esp
}
  26:   89 ec                   mov    %ebp,%esp
  28:   5d                      pop    %ebp
  29:   c3                      ret
  2a:   8d b6 00 00 00 00       lea    0x0(%esi),%esi
```

```
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000008 R_386_32          a
00000015 R_386_32          a
0000001a R_386_32          .rodata
0000001f R_386_PC32        printf
```

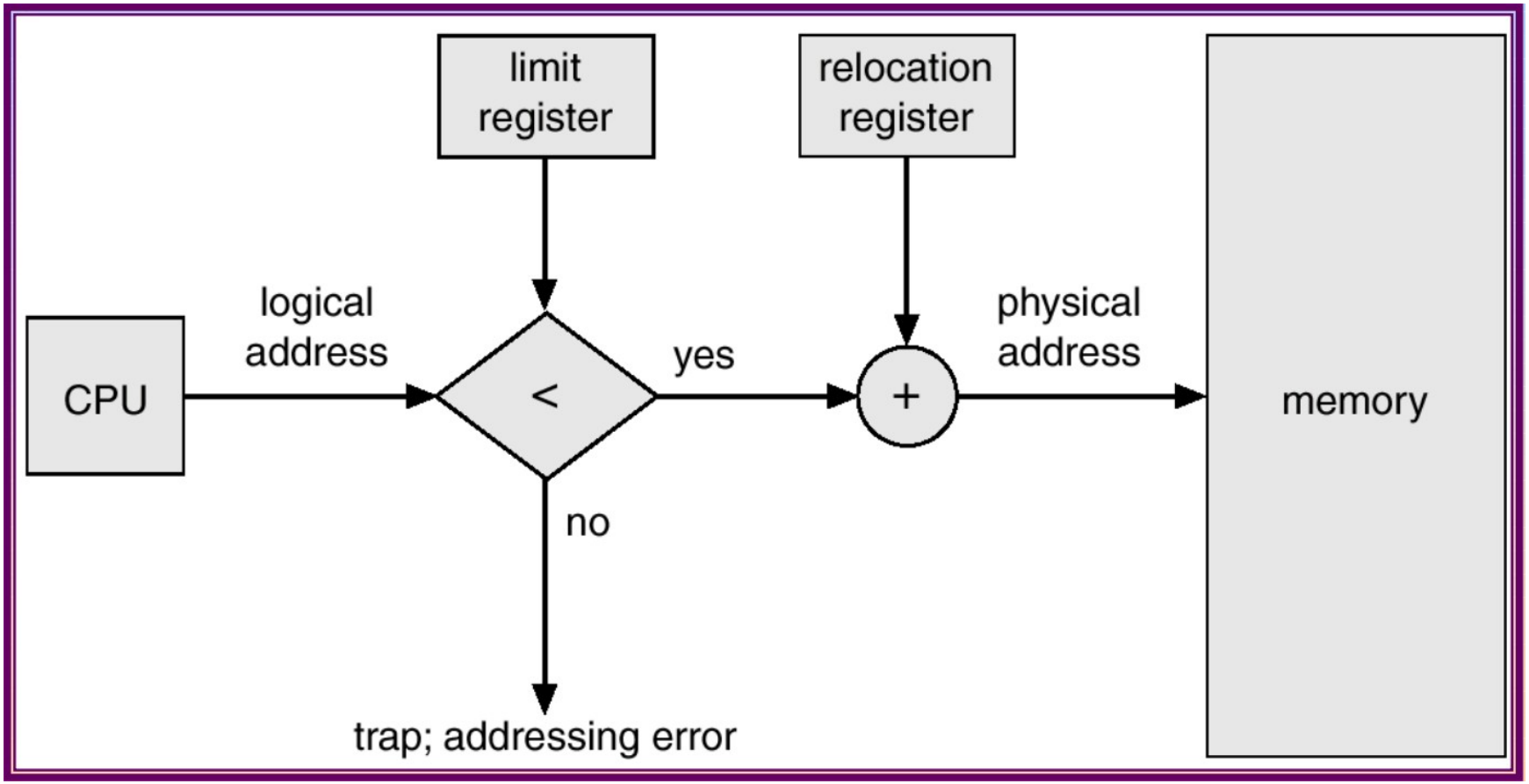
Spațiul de memorie

- Fiecare proces are asociat un spațiu de memorie
- Nucleul are un spațiu de memorie propriu

- Trebuie protejat accesul la spațiul de memorie al nucleului
- Spațiile de memorie ale proceselor trebuie protejate unele de celelalte

- Spațiu de memorie = spațiu de adrese valide ce pot fi accesate de un proces

Protecția spațiului de memorie



- Registru bază
- Registru limită

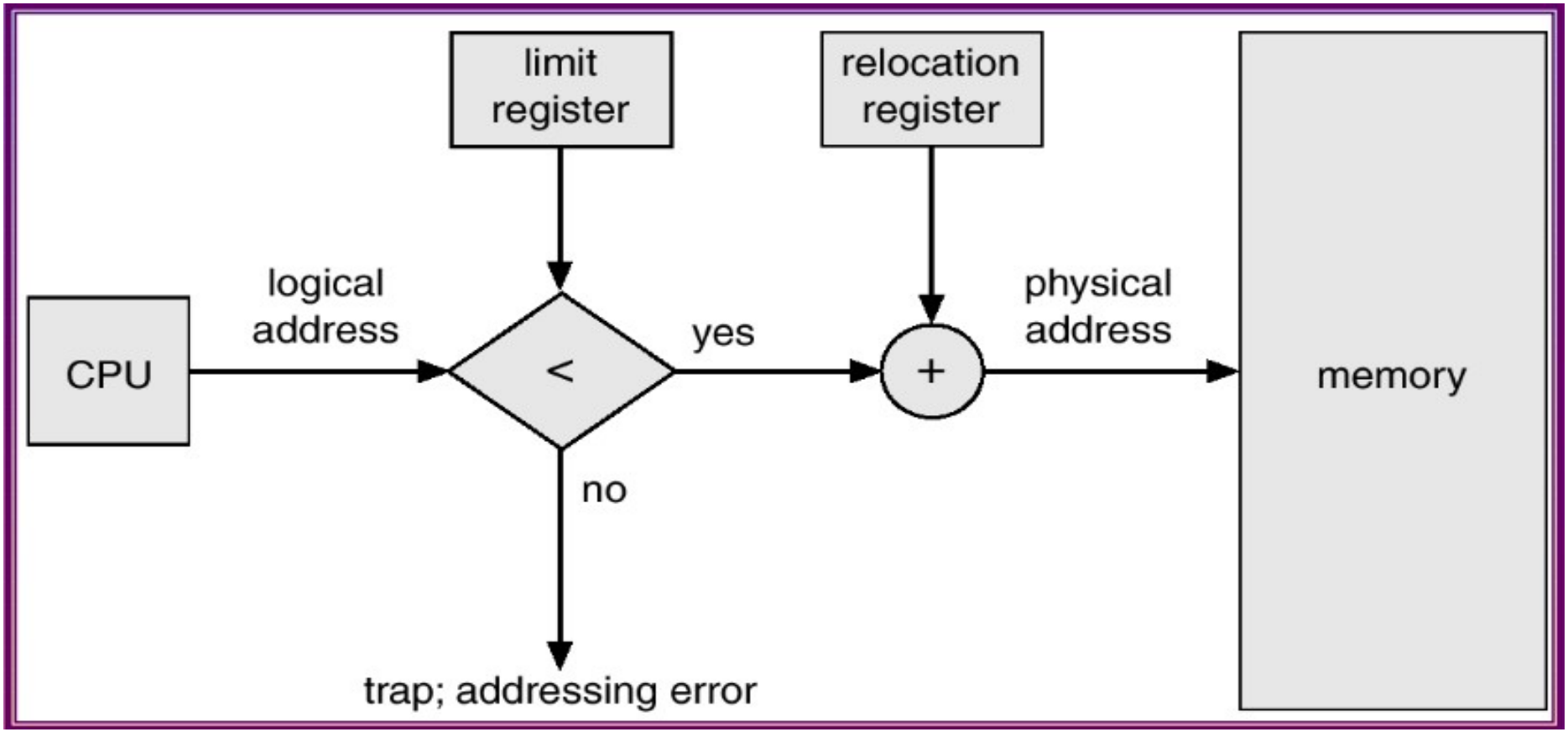
Adresare logică

- Adresele generate de procesor se numesc adrese logice
- Adresele folosite de unitatea de memorie se numesc adrese fizice
- În cazul relocării la execuție adresa logică diferă de adresa fizică
  - adresa logică este o adresă virtuală
  - spațiu virtual de adrese
  - spațiu fizic de adrese
- MMU (Memory Management Unit) - mapare adrese virtuale – adrese fizice

Alocarea memoriei contigue

- Spațiul fizic este împărțit în două partiții
  - kernel
  - procese utilizator
- Alocarea trebuie efectuată eficient
- În cazul alocării de memorie contiguă, fiecare proces rezidă într-o secțiune contiguă de memorie
  - folosirea unui registru de relocare și a unui registru limită

Alocarea memoriei contigue (2)



Algoritmi de alocare a memoriei

- Se mențin informații despre blocurile libere și blocurile alocate
- First-fit: se alocă primul bloc liber suficient de mare
- Best-fit: cel mai mic bloc suficient de mare
- Worst-fit: cel mai mare bloc
- First-fit și best-fit sunt superiori worst-fit (viteză și folosirea memoriei)

Fragmentarea memoriei

- Fragmentare externă
  - există spațiu pentru o nouă alocare dar nu este contiguu
- Fragmentare internă
  - se alocă un spațiu mai mare decât cel necesar
- Soluții
  - compactarea blocurilor libere prin glisare
  - folosirea de spații fizice non-contigue de memorie (paginare, segmentare)

Paginarea

- Permite spațiu fizic non-contiguu de memorie
- Se elimină problema găsirii unui bloc de dimensiune potrivită pentru alocare
- Paginile au o dimensiune fixă (4KB - x86)
- Paginile fizice sunt denumite “frames”
- Paginile virtuale sunt denumite “pages”

Paginarea (2)

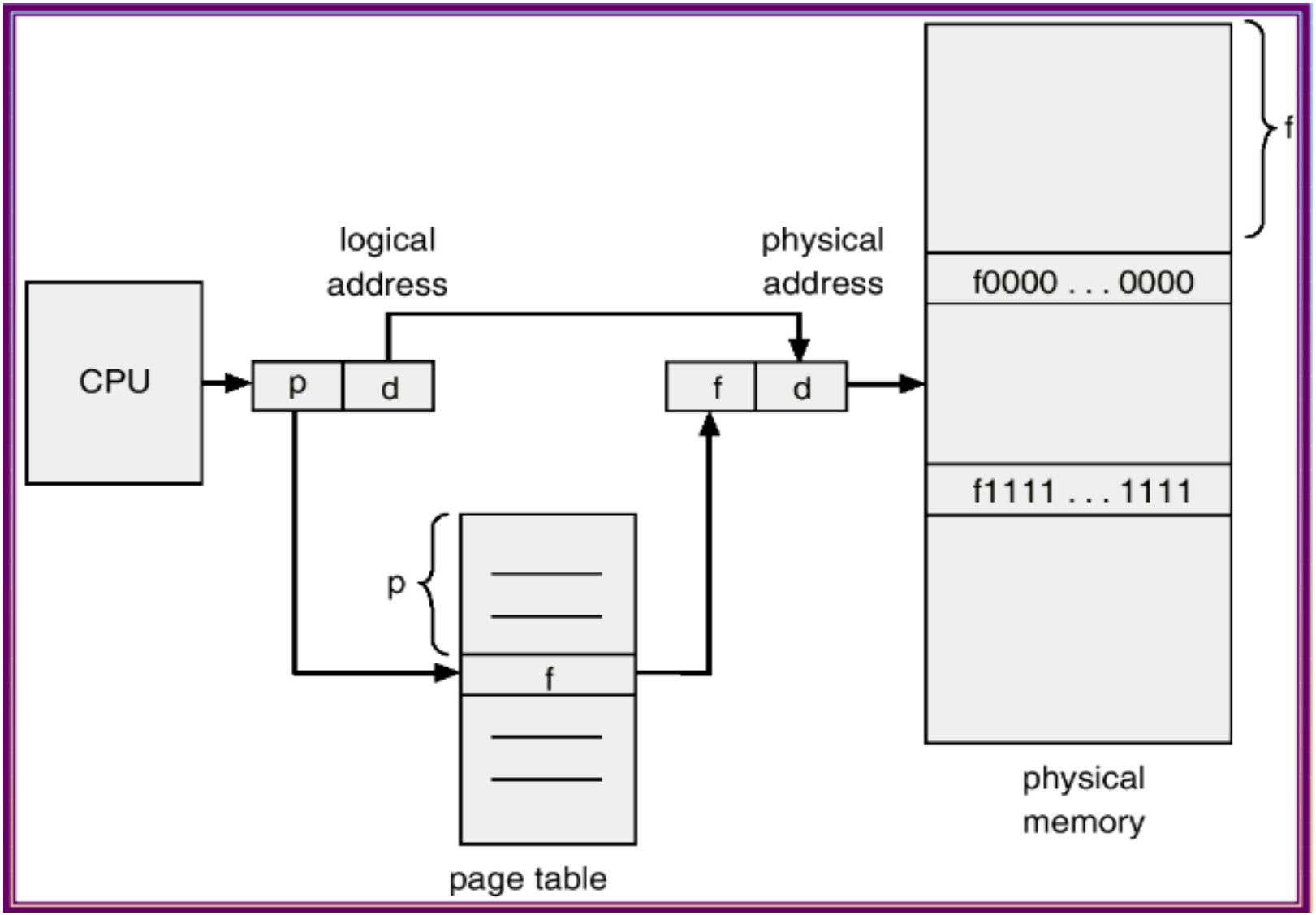
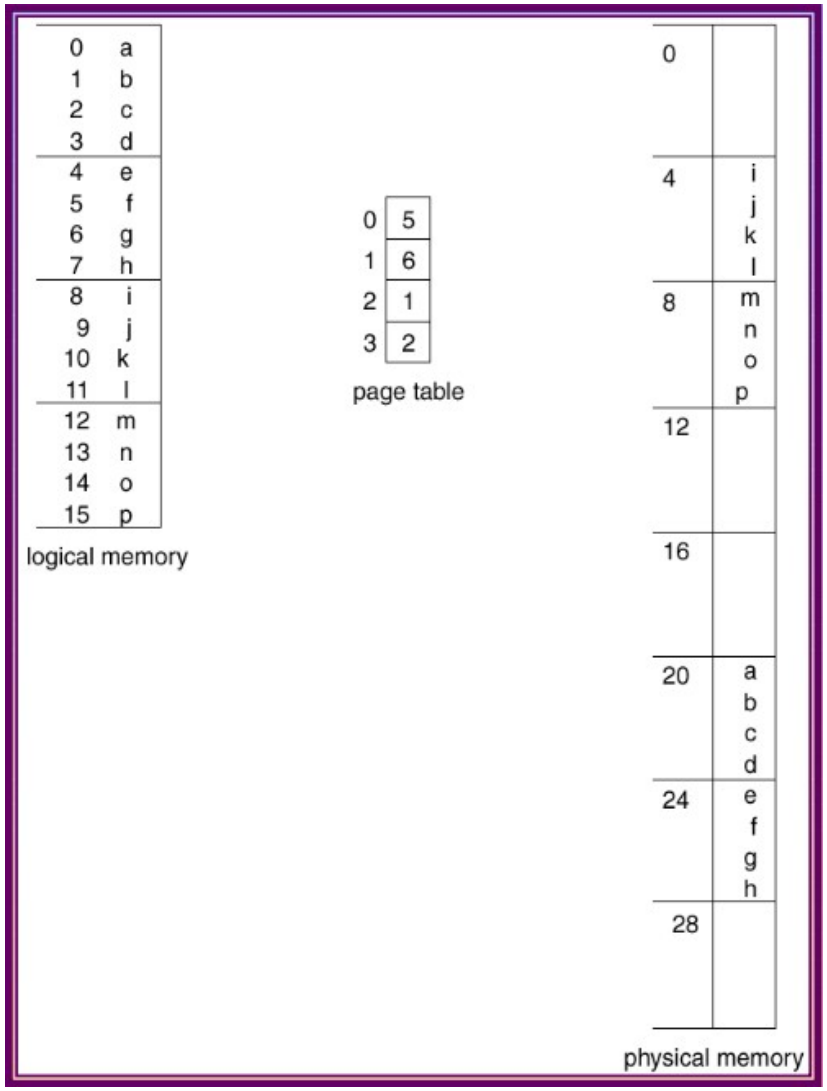

CPU
logical address
p
d
physical address
f
d
f
f0000 . . . 0000
f1111 . . . 1111
p
f
page table
physical memory

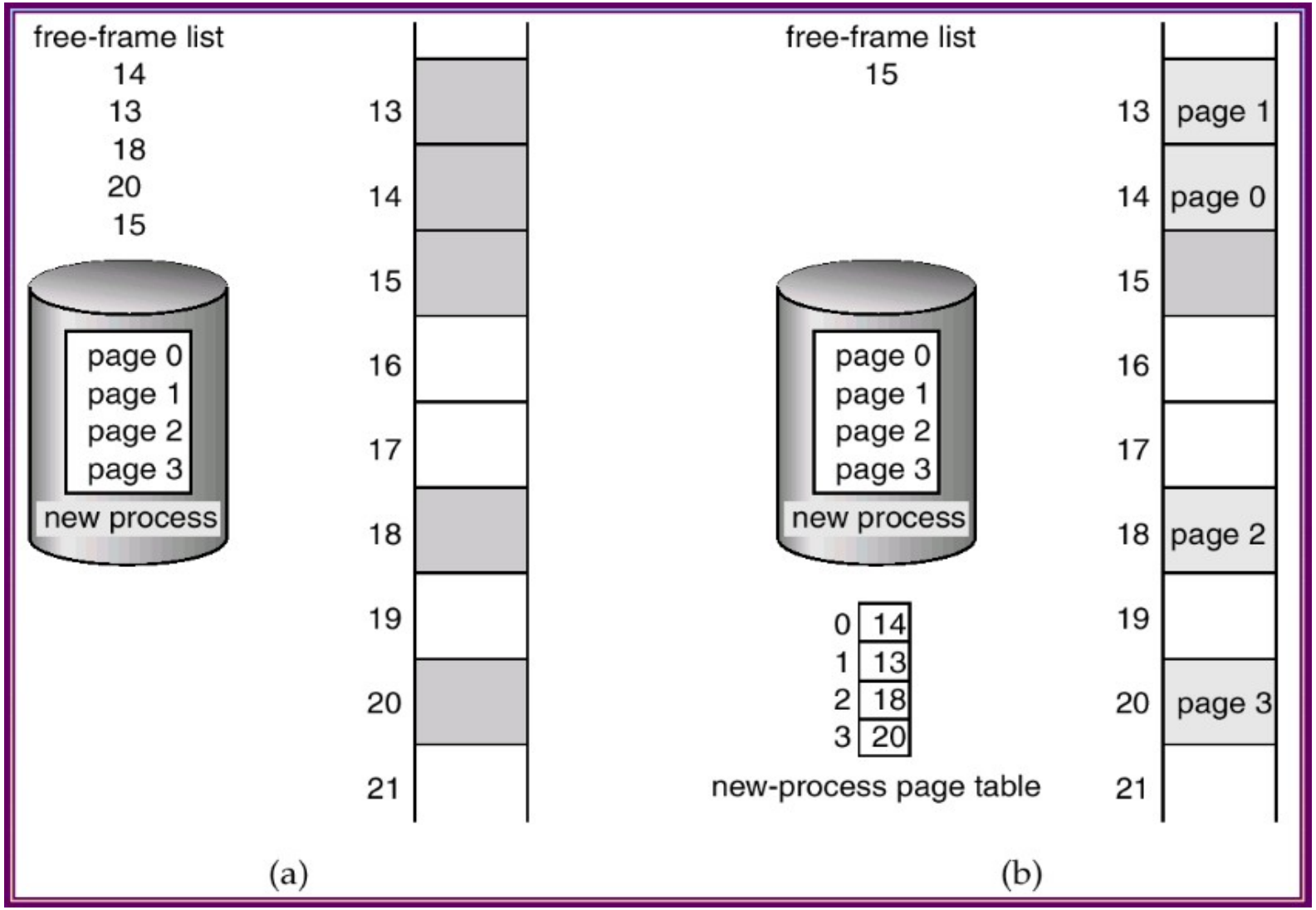
Tabela de pagini

- Adresa virtuală – număr de pagină (p) + deplasament (d)
- Numărul paginii este un index în **tabela de pagini**
- Tabela de pagini conține adresa fizică asociată fiecărei pagini (adresa frame-ului)
- Dimensiunea paginii (virtuale și fizice) este dată de hardware (512B – 4MB) (huge pages - 256MB)

Tabela de pagini (2)



Alocarea paginilor fizice



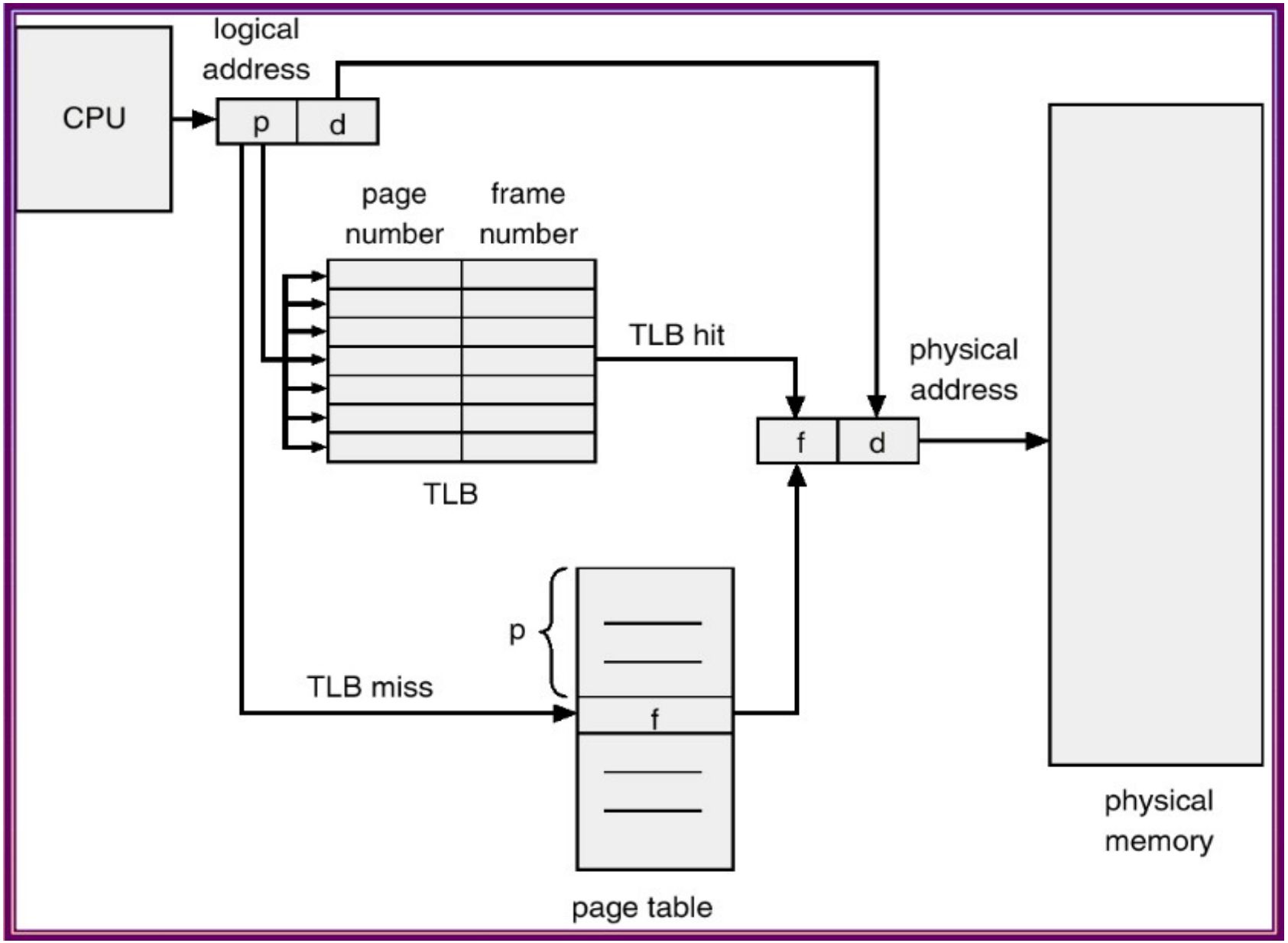
Suportul hardware

- În general, un sistem de operare alocă o tabelă de pagini pentru fiecare proces
- Pointer-ul la tabela de pagini a procesului este stocat în PCB
- O soluție este folosirea de registre pentru stocarea tabelei de pagini
  - problemă: dimensiunea mare
- PTBR – page table base register
  - punctează către adresa de memorie unde este stocată tabela de pagini
  - problemă: două accese la memorie
  - soluția: TLB

TLB

- Translation Look-aside Buffer
- Cache rapid dar de dimensiune mică și cost ridicat
- Memorie asociativă rapidă
  - tag (cheie) – page number
  - valoare – frame number
- Dimensiune tipică între 64 și 1024 de intrări

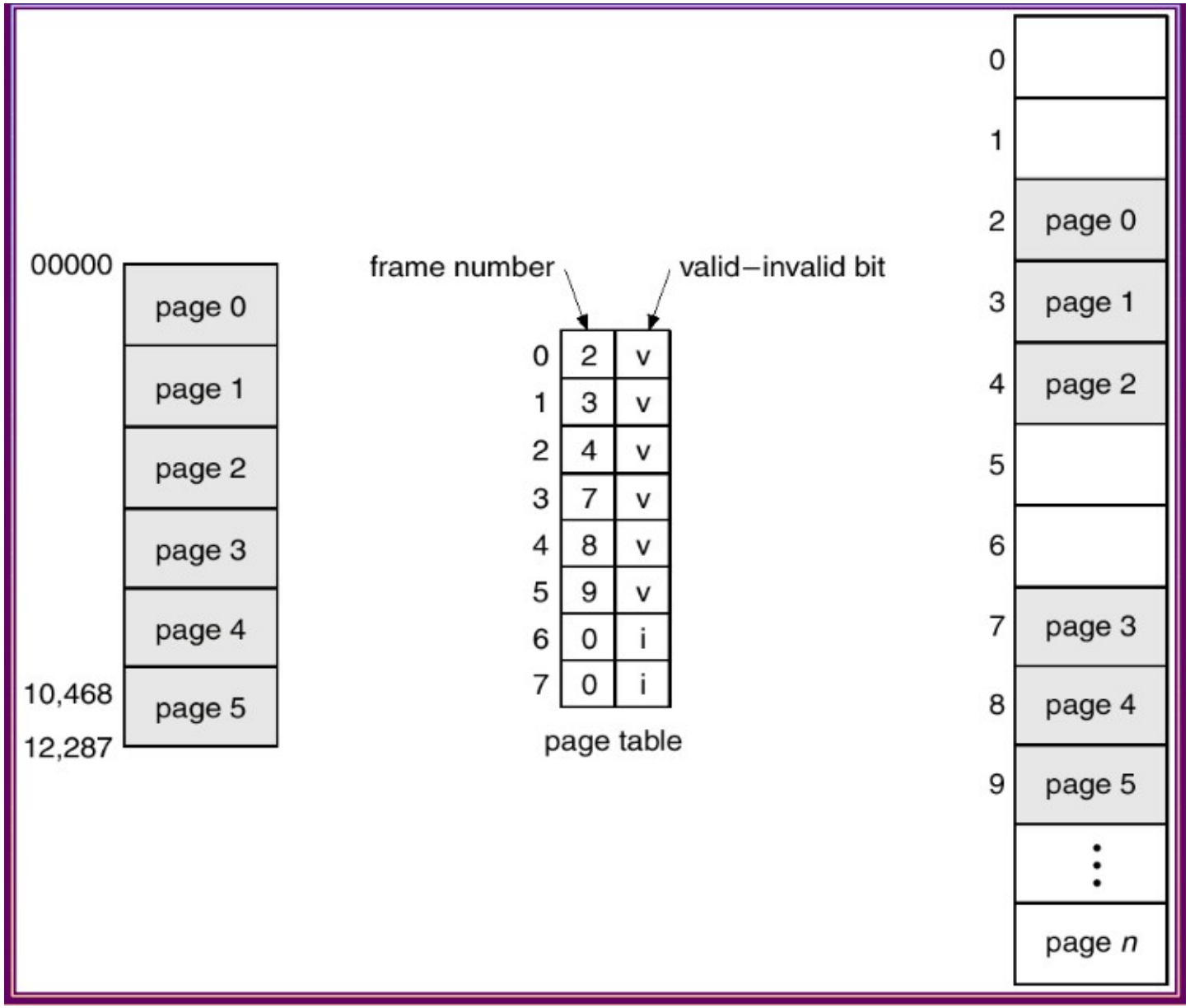
TLB (2)


CPU
logical address
p
d
page number
frame number
TLB hit
physical address
f
d
TLB
p
TLB miss
f
page table
physical memory

Exercițiu TLB

- Date
  - TLB hit rate: 75%
  - Timp de acces la memorie: 60ns
  - Timp de acces la TLB: 10ns
- Care este media timpului de acces la memorie?
- Dar dacă TLB hit rate este 95%?

Protecția memoriei paginate



Partajarea memoriei

- Avantaj al paginării
- Fiecare proces care provine din același executabil poate partaja paginile fizice
  - paginile de date sunt distincte
- Codul poate fi partajat dacă nu trebuie relocat
- Codul partajat trebuie să fie read-only
- Mecanismul de memorie partajată este implementat folosind pagini partajate

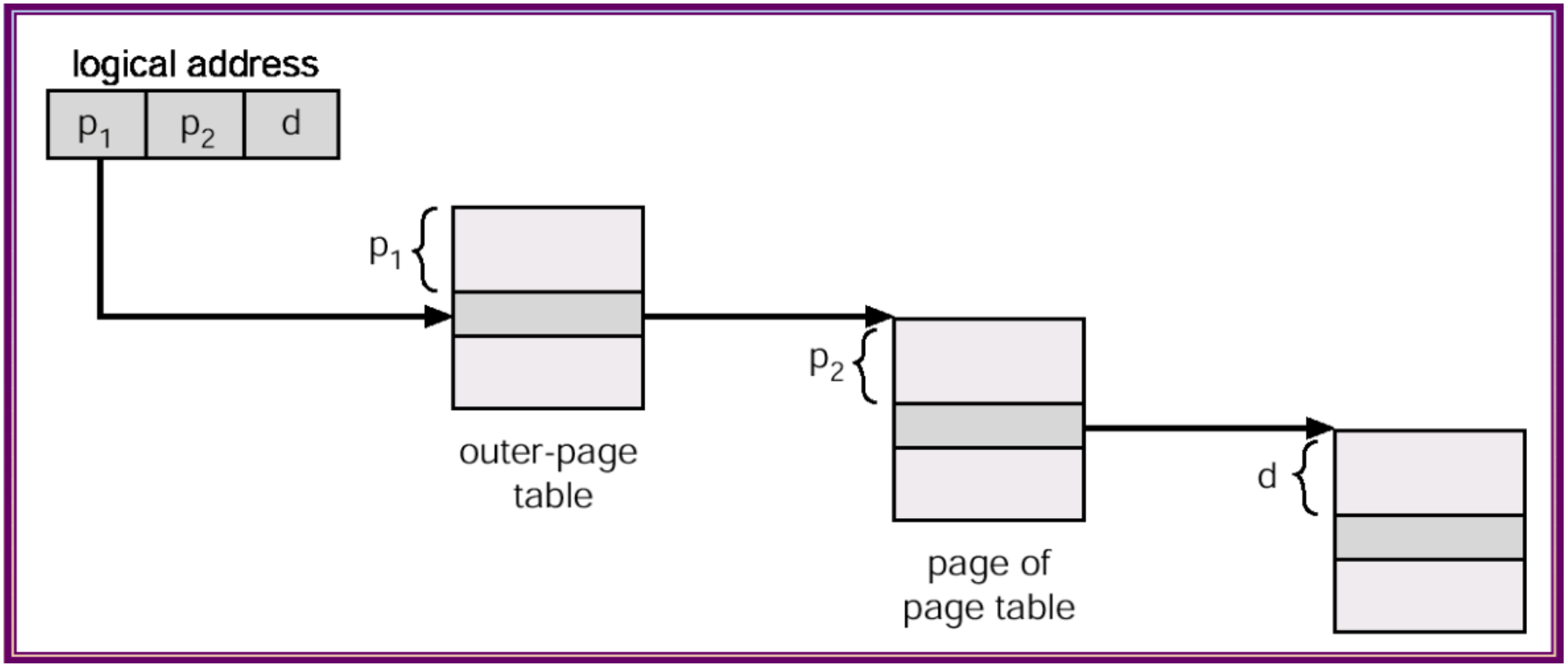
Paginarea ierarhică

- Situație: sistem cu spațiu de adresă pe 32 de biți, dimensiunea unei paginii de 12 biți
  - tabela de pagini conține 2^32 / 2^12 = 2^20 intrări (1 milion)
  - fiecare intrare ocupă 4 octeți -> 4 MB ocupați doar cu menținerea tabelei de pagini
- Soluție:
  - paginarea ierarhică
    - împărțirea tabelei de pagini în componente mai mici

Paginarea ierarhică (2)

- Pe un sistem pe 32 de biți se poate opta pentru împărțirea numărului de pagină
  - Număr de pagină de 10 biți
    - index în outer page table
  - Offset de pagină de 10 biți
    - index în page table
- Outer page table nu conține pointeri către page table în pozițiile unde adresele sunt invalide
  - economisire spațiu

Paginarea ierarhică (3)



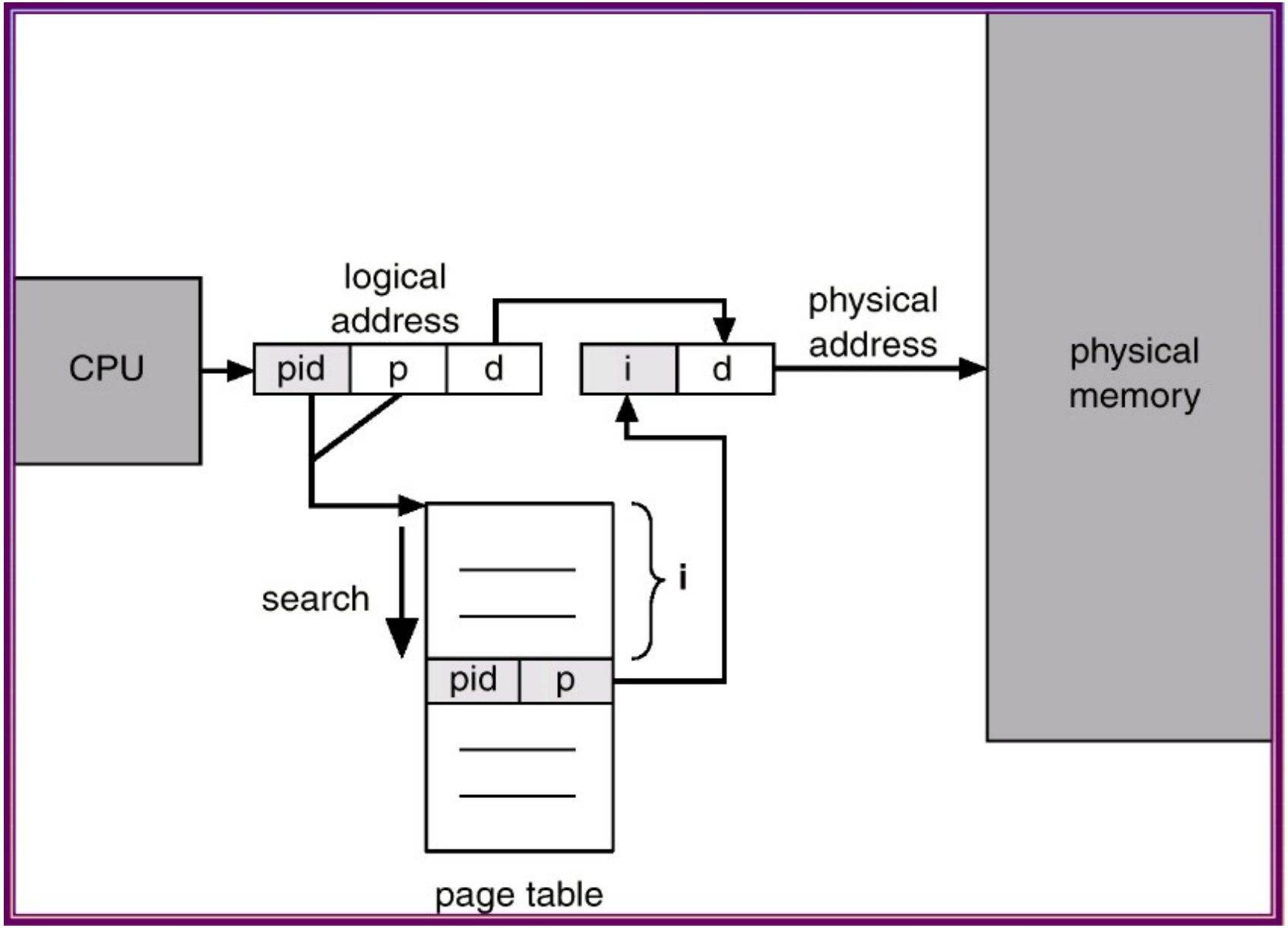
Exercițiu

- Date
  - Spațiu virtual de adresare de 32 de biți
  - Pagina de 4K
  - Spațiul virtual al unui proces
    - `[.text 1020 pagini]`
    - `[.data 4 pagini]`
    - `[4096 pagini spatiu nemapat]`
    - `[.stack 16 pagini]`
  - O pagină din tabela de pagini conține 1024 de intrări
- Cât spațiu ocupă tabela de pagini în cazul folosirii:
  - paginării neierarhice
  - paginării ierarhice pe două niveluri (10 biți, 10 biți, 12 biți)

Tabela de pagini inversată

- Altă soluție la dimensiunea mare a tabelei de pagini
- Există o intrare pentru fiecare pagină fizică (frame din sistem)
- Intrarea conține
  - pid-ul procesului
  - adresa paginii logice stocate în pagina fizică
- Se folosește mai puțin spațiu
- Căutare mai costisitoare – tabele hash

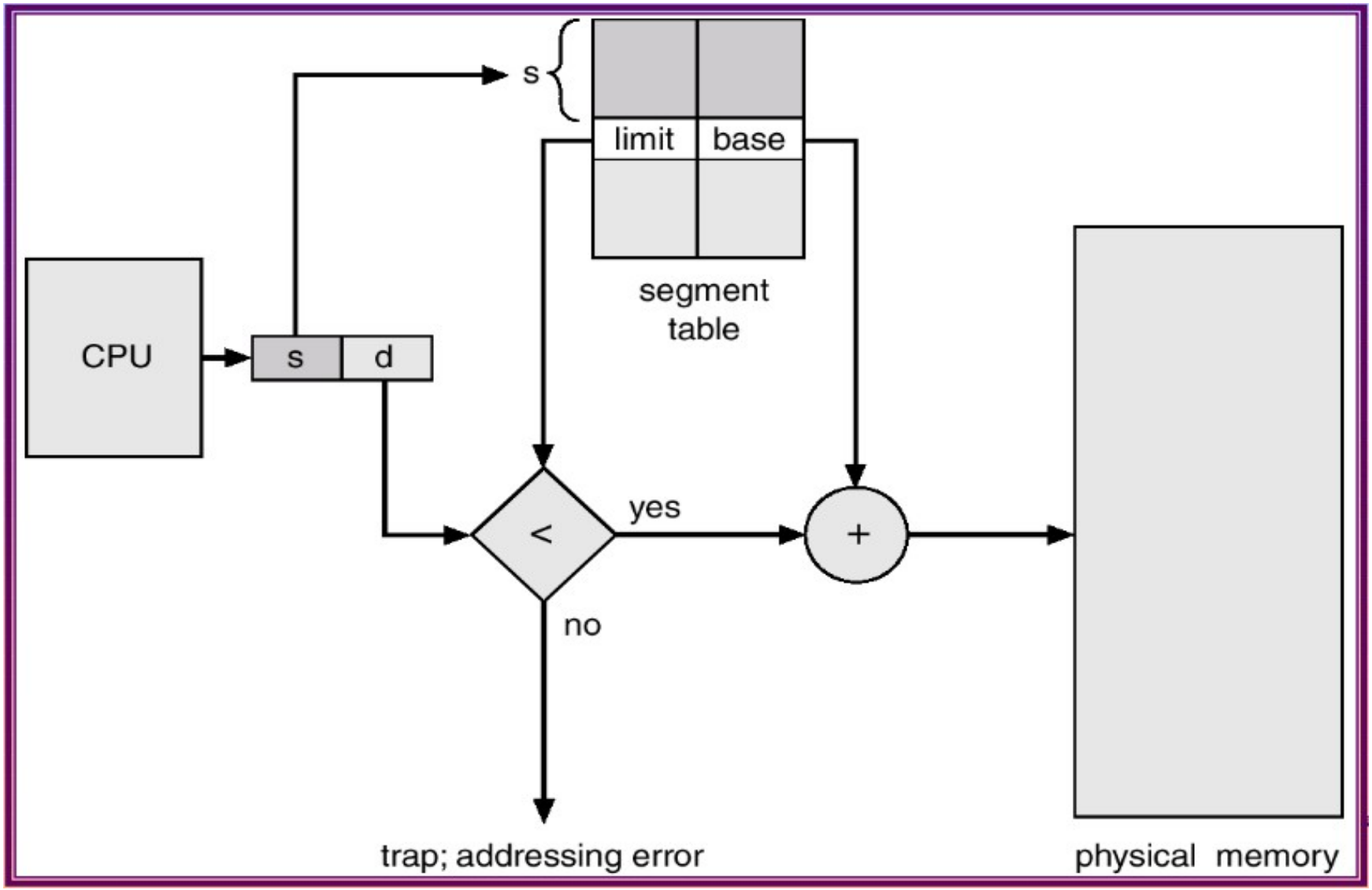
Tabela de pagini inversată (2)



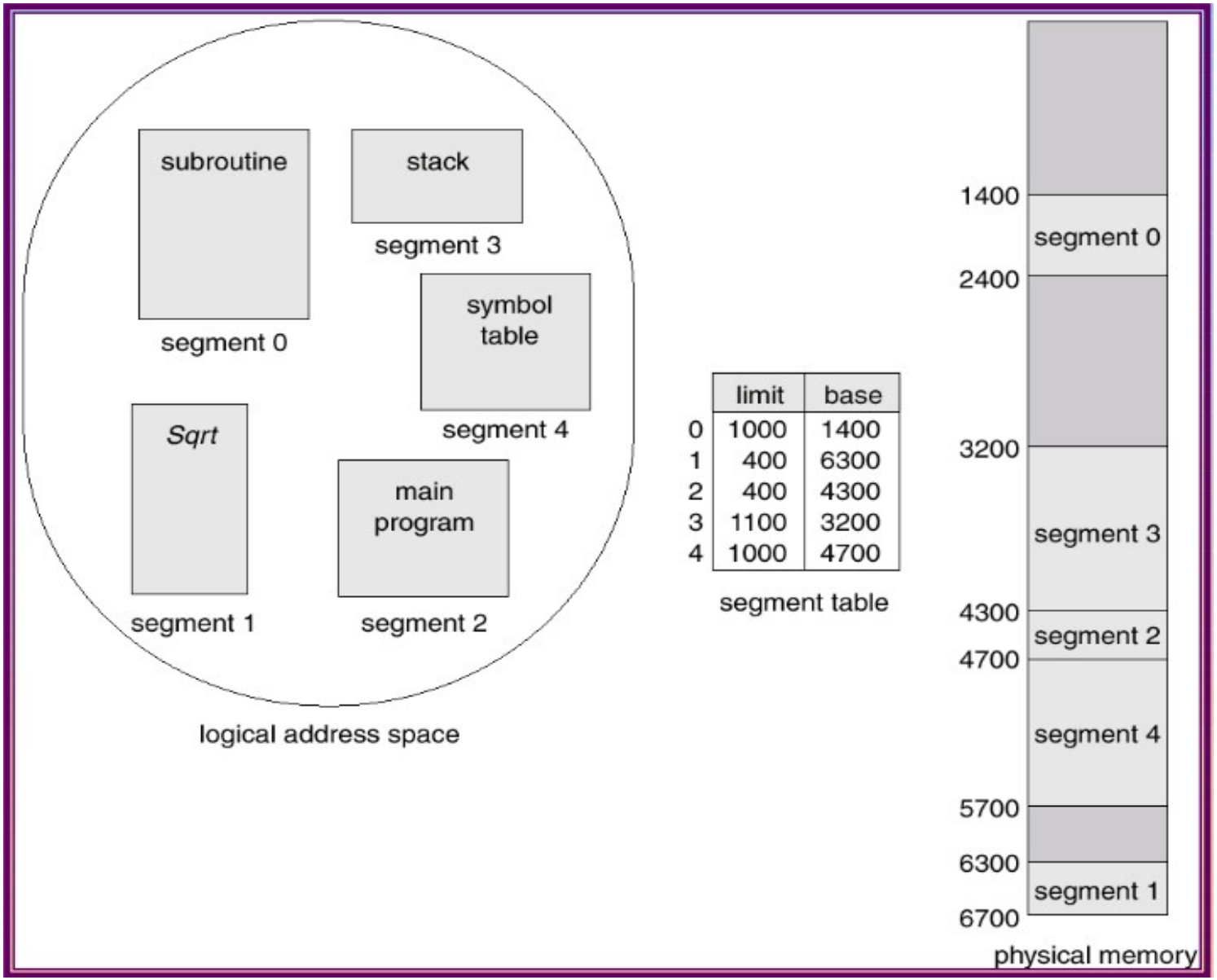
Segmentare

- Utilizatorul/programatorul vede memoria ca o colecție de zone (pentru funcții, date, stivă etc.)
- Segmentare – spațiul logic de adrese este o colecție de segmente
- O adresă logică este un tuplu
  - <număr segment, deplasament>
- Tabela de segmente realizează translatarea între adresa logică și adresa fizică
- Tabela de segmente conține o adresă de bază a segmentului și o adresă limită
- Numărul segmentului (din adresa logică) este un index în tabela de segmente

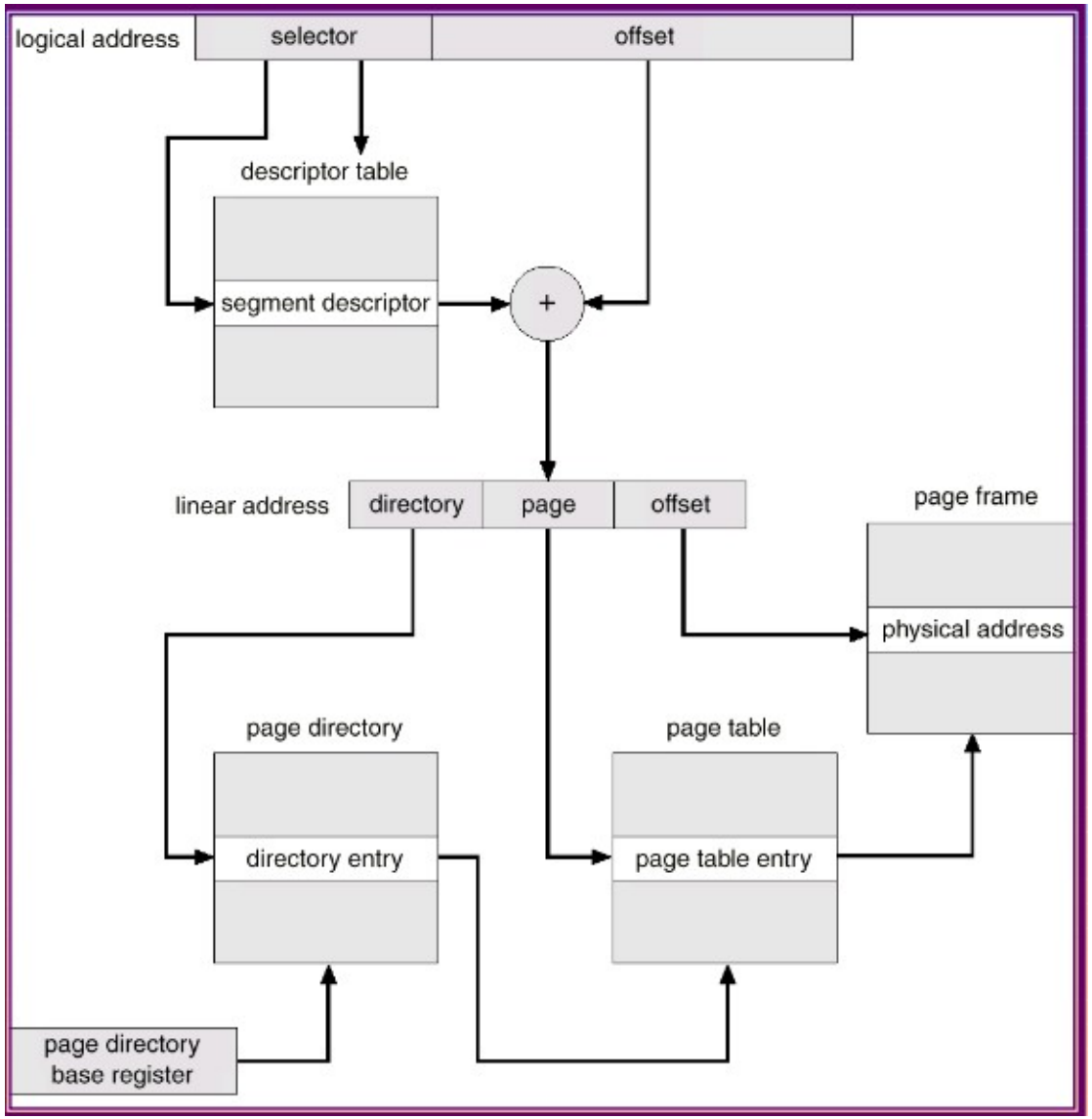
Segmentare (2)



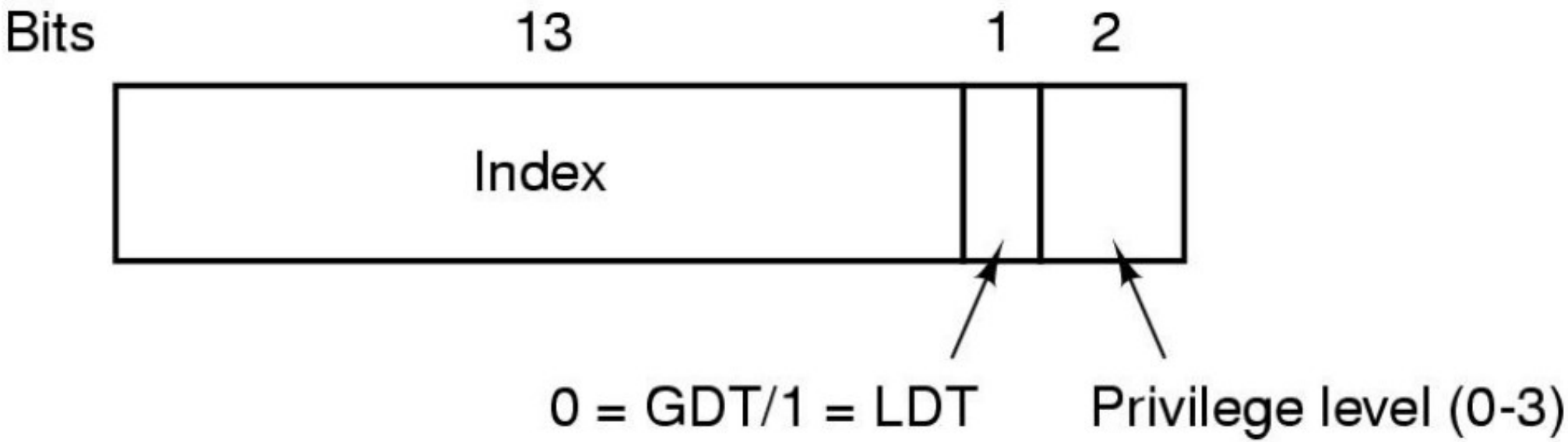
Segmentare (3)



Segmentare cu paginare (Intel)

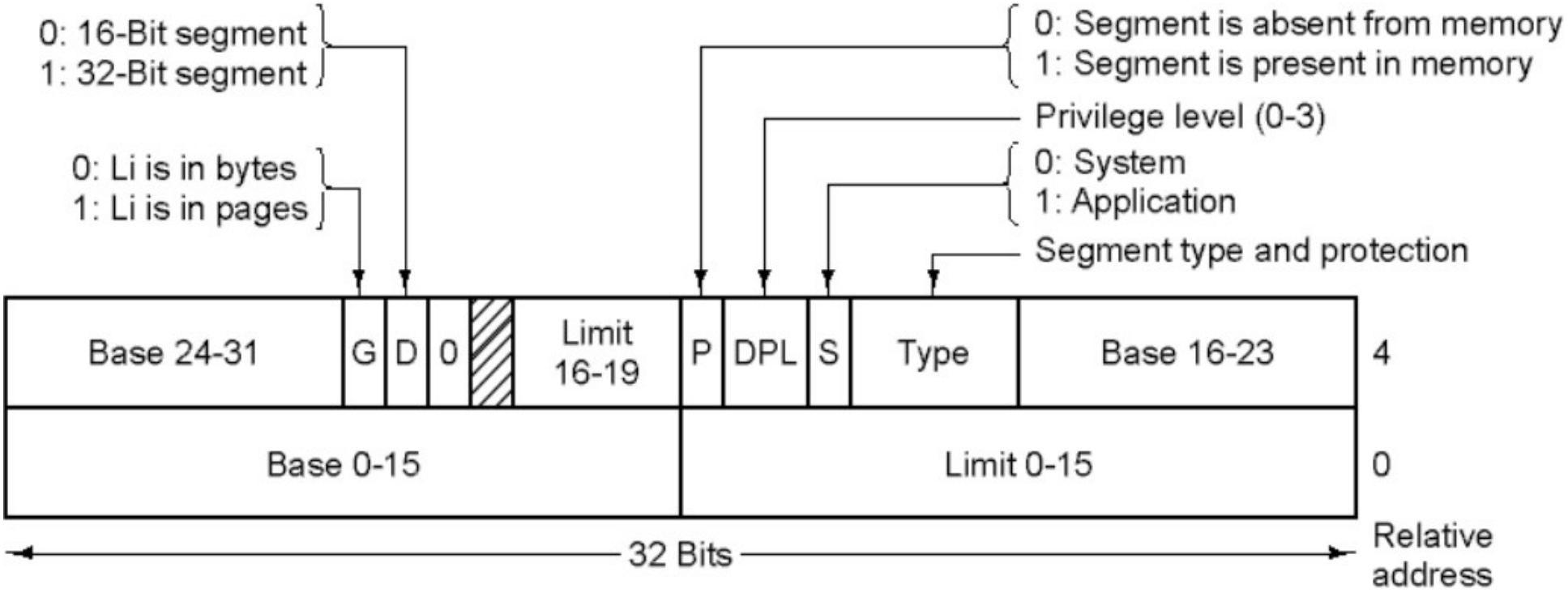


Selectorul de segment

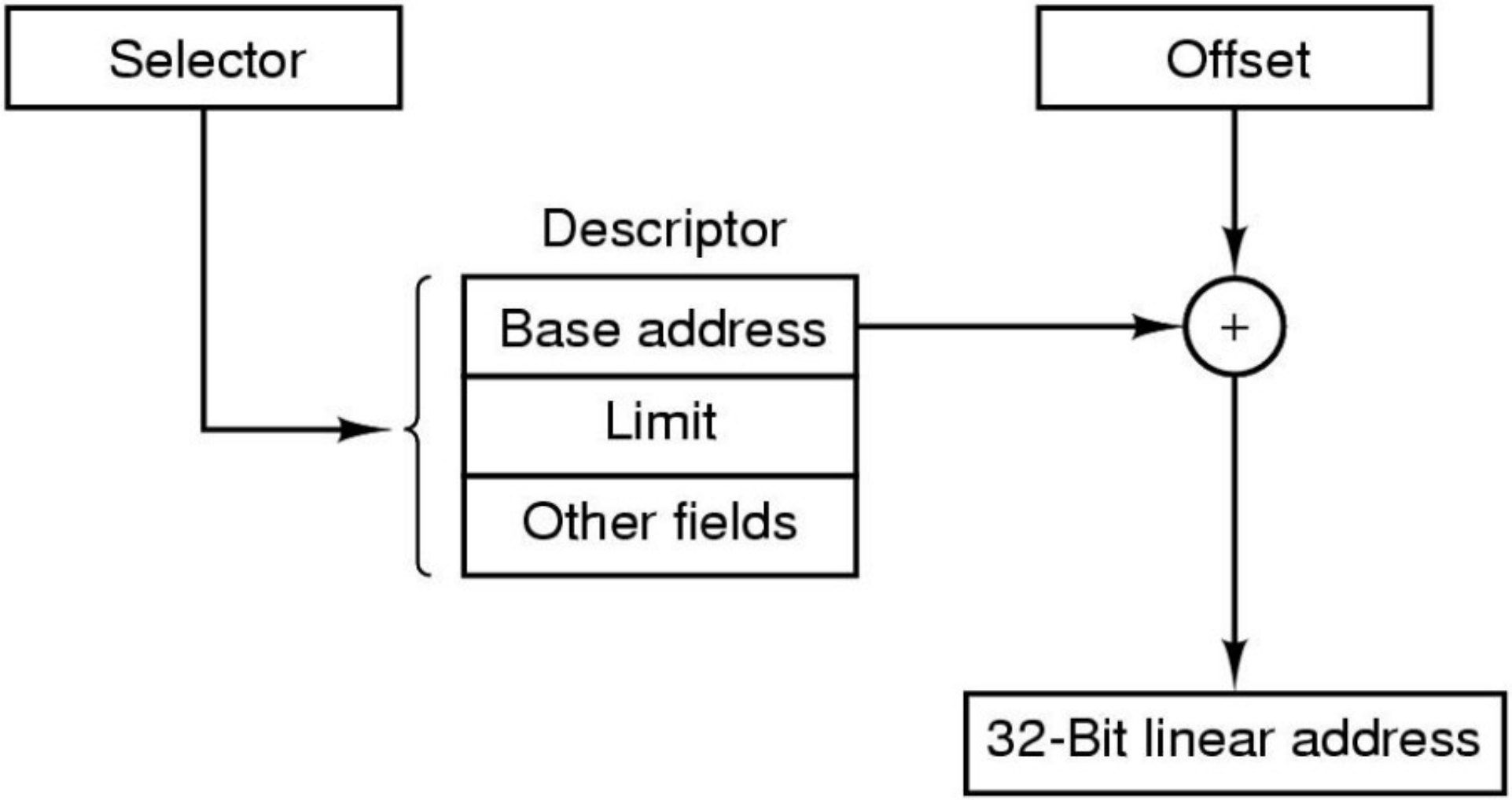


- CS pentru cod, DS pentru date, SS pentru stivă
- Indexează un descriptor în GDT sau LDT

Descriptorul de segment



Adresă liniară



Niveluri de protecție

- La x86 sunt 4 niveluri de protecție
- Apelurile inter-nivel se realizează prin operații de procesor specializate (call-gates)
  - selectorul indică un descriptor ce conține adresa de intrare și nivelul de protecție necesar
- Un apel de sistem înseamnă trecerea din nivelul de protecție 3 la nivelul 0

Cuvinte cheie

- Ierarhia de memorii
- Memoria principală
- Memoria cache
- Spațiu de memorie
- Address binding
- Dynamic linking
- Pagină virtuală (page)
- Pagină fizică (frame)
- Adresă logică
- Adresă fizică
- Alocare
- Fragmentare
- Paginare
- Tabelă de pagini
- TLB
- Paginare ierarhică
- Tabelă inversată
- Segmentare
- Selector de segment
- Descriptor de segment
- Adresă liniară

Exercițiu

- Câte elemente de fiecare tip din lista celor de mai jos conține un proces?
  - executabil
  - stivă
  - handler de semnal
  - tabelă de fișiere
  - spațiu de adresă
  - descriptor de fișier deschis
  - tabelă de pagini (discuție paginare ierarhică/neierarhică)

Exercițiu 2

- Un sistem fără TLB și cache accesează memoria folosind paginare neierarhică. Presupunând că timpul de acces la memorie este de 5ns și că o pagină de memorie ocupă 4KB, cât durează operațiile de mai jos? (sizeof(char) = 1)

```
char a[4096 * 16];
char b[4096 * 16];

/* init a[i] */

for (i = 0; i < 4096 * 16; i += 32) {
        b[i] = a[4096* 16 – i -1];
}
```

Întrebări

?