Sisteme de Operare

Administrarea memoriei
partea a II-a

Cristian Vidrașcu

https://profs.info.uaic.ro/~vidrascu

Cuprins

Alocarea memoriei (continuare)

- Scheme de alocare necontigue
  - Paginarea
  - Segmentarea
  - Segmentarea paginată

(va urma)

- Scheme de alocare cu memorie virtuală

Alocarea paginată /1

- **Paginarea** (paginare pură, fără *swapping*)
  - permite ca memoria alocată unui proces să fie necontiguă
  - paginare hardware
    - memoria fizică este împărțită în blocuri de lungime fixă, numite **cadre de pagină** (*frames*), sau *pagini fizice*
    - lungimea cadrelor de pagină este o putere a lui 2 și este o constantă a SC-ului (e.g. pentru arhitecturile Intel este 4 Ko)
    - memoria logică a unui proces (zona de cod + zona de date) este împărțită în blocuri de lungime fixă (egală cu lungimea cadrelor de pagină), numite **pagini** (*pages*), sau *pagini virtuale*

Alocarea paginată /2

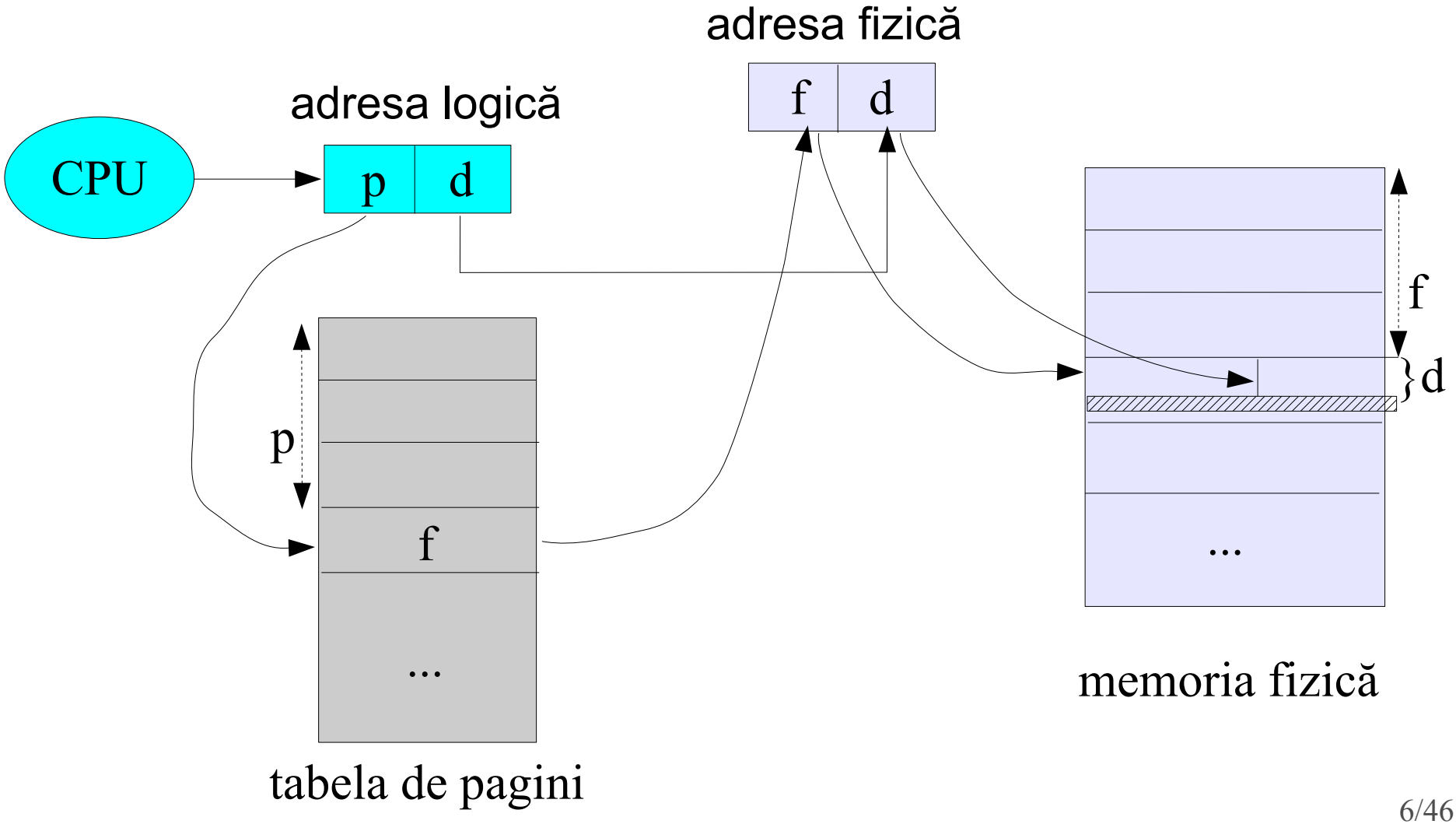
- **Paginarea (cont.)**
  - paginare hardware
    - pentru execuția procesului, paginile sale trebuie încărcate în orice cadre de pagină libere (nu neapărat contiguu!)
    - SO-ul păstrează evidența cadrelor de pagină libere
    - pentru a rula un program ce necesită N pagini, este nevoie să se găsească și să se mapeze N cadre libere
    - Avantaj: datorită alocării necontigue, se elimină fragmentarea *externă*, i.e. nu rămân zone de memorie nealocate ce nu pot fi folosite
    - Apare însă fragmentarea *internă* a memoriei – poate rămâne spațiu nefolosit, dar alocat proceselor (deoarece dimensiunea procesului nu este un multiplu exact de lungimea paginilor)

Alocarea paginată /3

- **Paginarea (cont.)**
  - Adresele logice sunt formate din 2 componente: numărul de pagină (p) și deplasamentul în pagină (d)
  - Adresele fizice sunt formate din 2 componente: numărul de cadru (f) și deplasamentul în cadru (d)
  - Pentru ***maparea*** paginilor în cadre de pagină (i.e. translatarea adreselor logice în adrese fizice) se folosește o *tabelă de pagini* per proces, care păstrează pentru fiecare pagină, adresa de bază a cadrului asociat ei (în care este încărcată pagina respectivă)
  - Mai precis, pentru translatare se folosește numărul de pagină drept index în tabela de pagini, conform schemei următoare

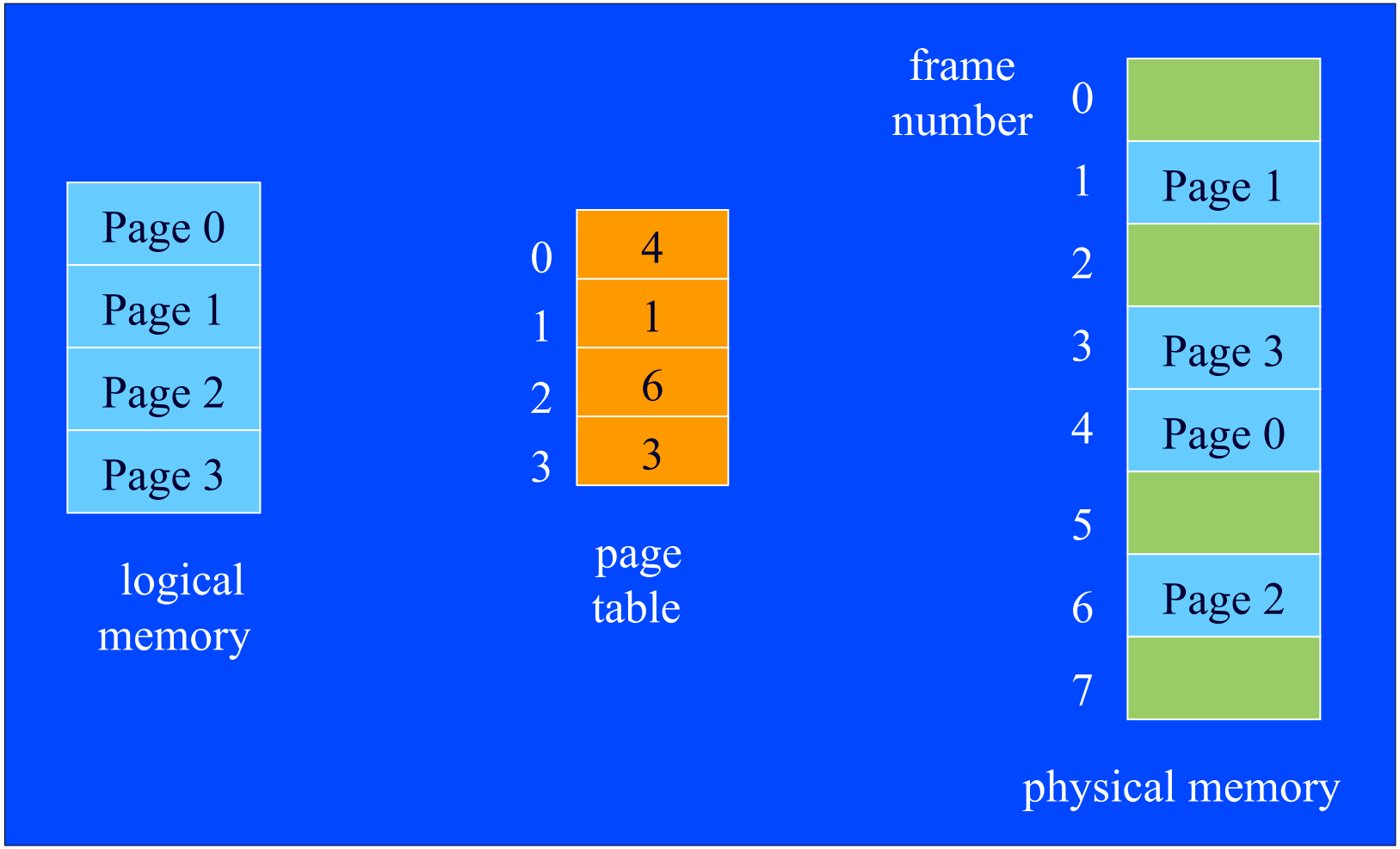
Alocarea paginată /4

Translatarea adresei logice în adresă fizică:



Alocarea paginată /5

Exemplu:



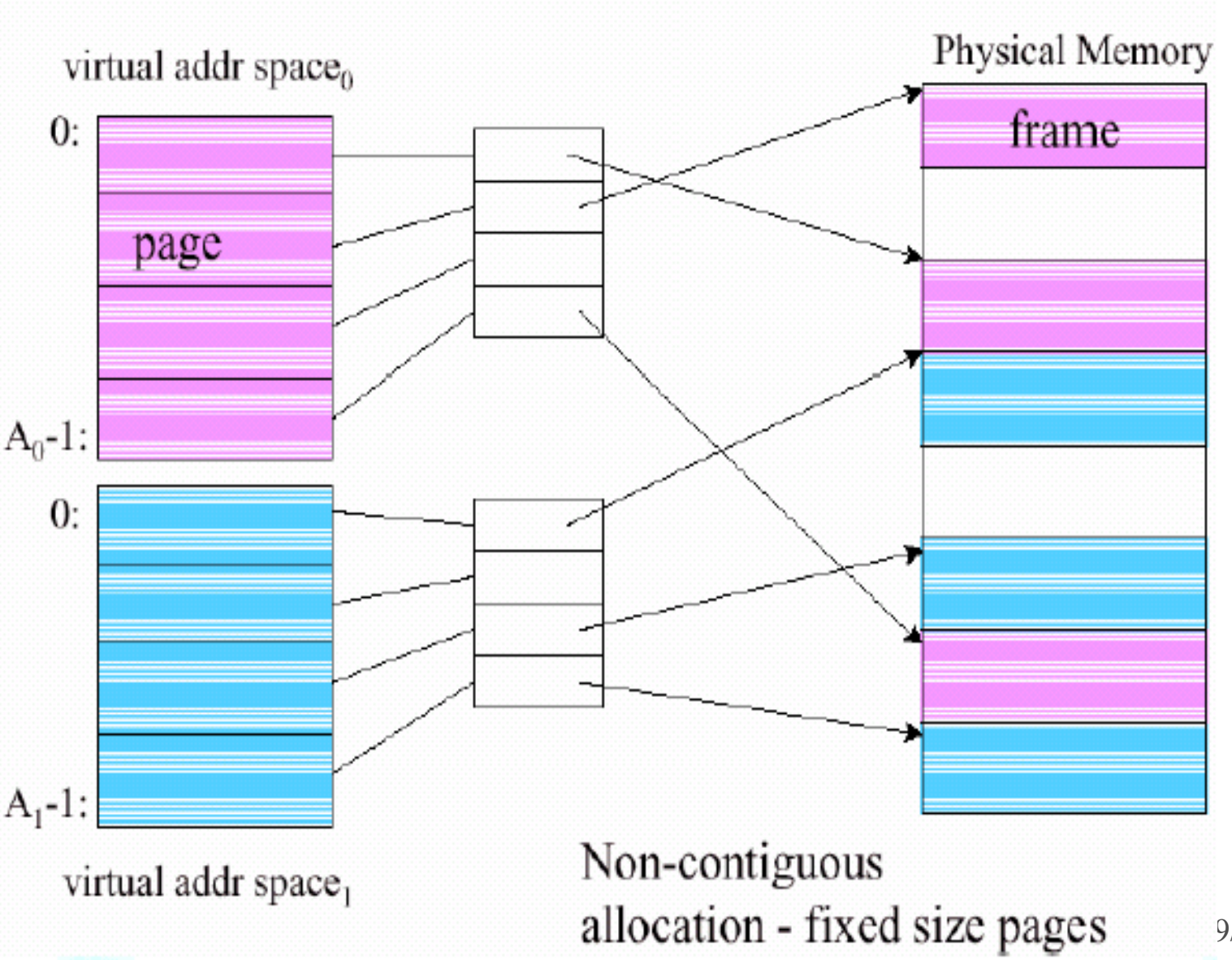
Alocarea paginată /6

- **Paginarea (cont.)**
  - Alt avantaj: suportă *partajarea memoriei* – două sau mai multe procese pot "vedea" aceeași zonă de memorie prin maparea (i.e., încărcarea) câte unei pagini virtuale din fiecare proces în același cadru fizic
  - Cod reentrant = cod nemodificabil (poate fi partajat)
  - Dezavantaj: programul trebuie, încă, să fie rezident în întregime în memoria principală
    - Înlăturarea acestui neajuns: *paginarea la cerere* (folosind tehnica de swapping, la nivel de pagină)
  - Alt dezavantaj: fiecare acces la memorie presupune un acces suplimentar la tabela de pagini din memorie, pentru calculul de adresă

Alocarea paginată /7

- **Exemplu de alocare paginată**

E incompletă,
în sensul că nu toate paginile virtuale "posibile" sunt și folosite efectiv (i.e., mapate/încărcate în memoria fizică)



Alocarea paginată /8

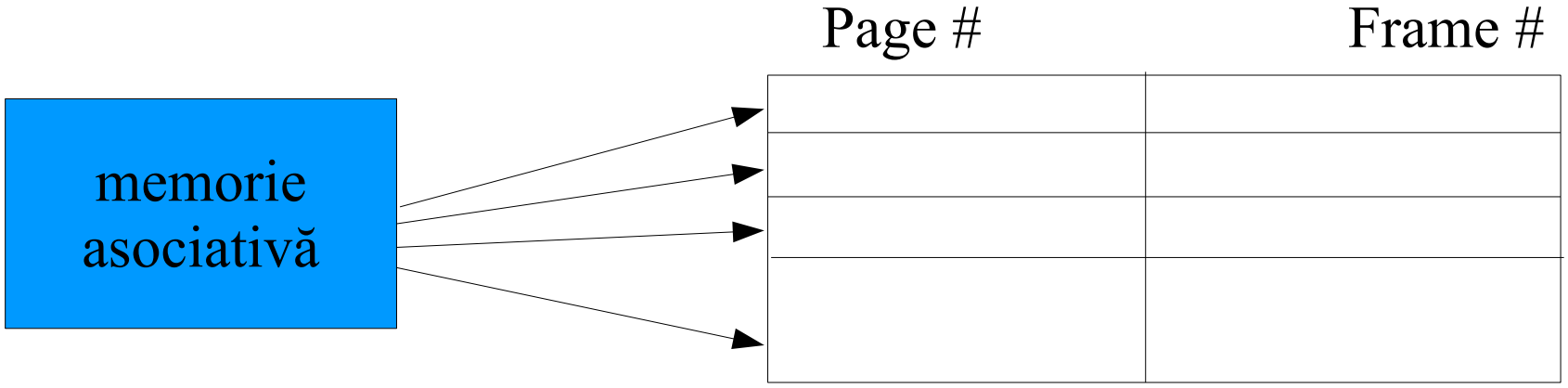
- **Implementări pentru tabelele de pagini**
  - Păstrarea tabelelor de pagini se poate face în: regiștri hardware sau memoria principală
  - Soluția cu regiștrii hardware este mult mai rapidă, dar devine prea costisitoare pe măsură ce crește dimensiunea tabelei de pagini;
    în plus, accesul regiștrilor hardware este privilegiat
  - Problema soluției cu memoria principală: accese multiple la memorie pentru a accesa o informație din memorie (întâi trebuie accesată tabela de pagini, pentru aflarea adresei fizice asociate adresei logice dorite, și abia apoi se accesează memoria fizică la adresa aflată pe baza translatării)

Alocarea paginată /9

- **Tabela de pagini în memorie**
  - O tabelă de pagini per proces
  - Un registru, PTBR (*page table base register*), conține adresa de bază (i.e., de început) în memorie a tabelei de pagini (e.g. registrul CR3 în cazul CPU x86)
  - Un alt registru, PTLR *(page table length register*), indică dimensiunea tabelei de pagini
  - Problemă: fiecare acces la date sau instrucțiuni "costă" *două* accese de memorie
  - Soluția: TLB (*traslation look-aside buffer*)

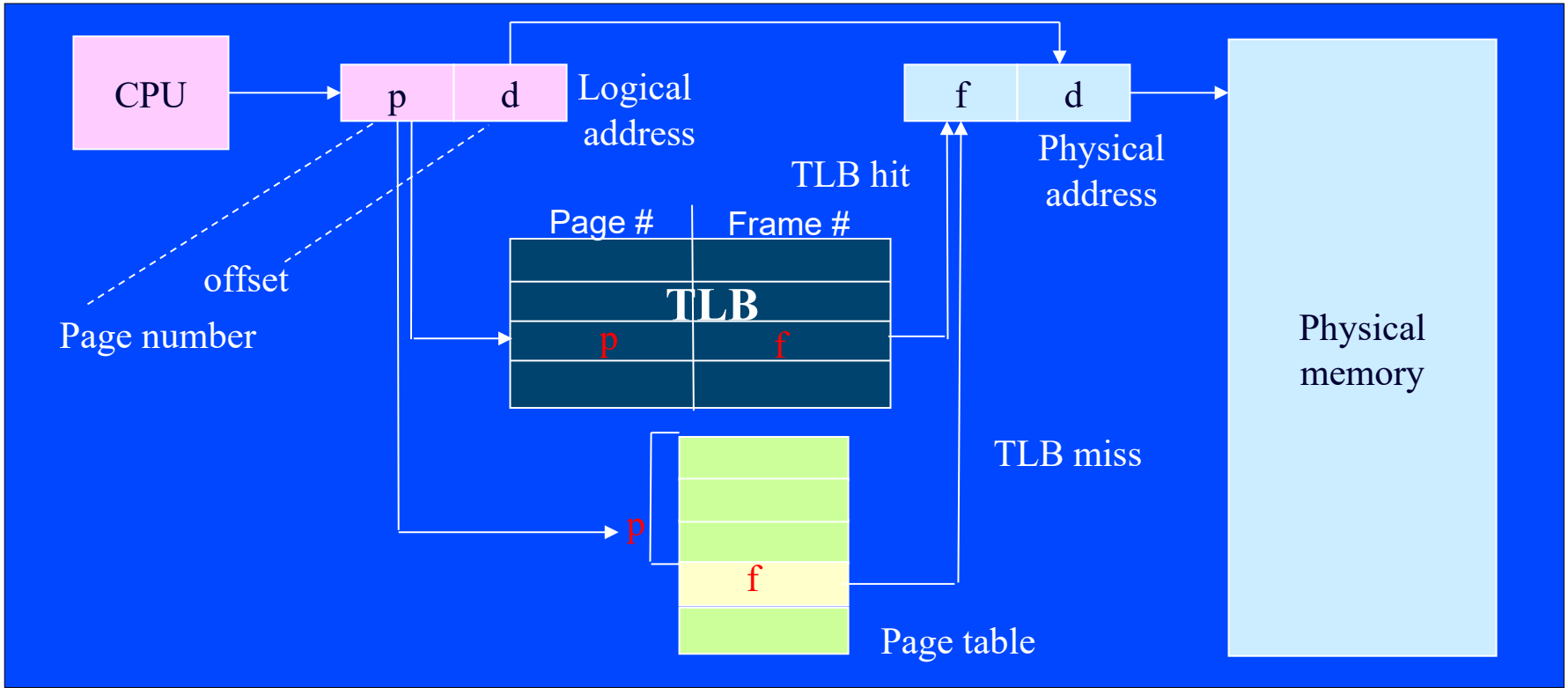
Alocarea paginată /10

- **Traslation Look-aside Buffer (TLB)**
  - Este o memorie specială de dimensiune mică, numită *memorie asociativă* (hardware special, scump)
  - Calitatea ei fundamentală este *adresarea prin conținut*, și nu prin adresă: găsește locația care are un conținut specificat, căutând simultan (i.e., *în paralel*) în toate locațiile ei



Traslation Look-aside Buffer (TLB) (cont.)

– TLB are rol de *cache* pentru tabela de pagini din memorie: când se încearcă aflarea numărului de cadru asociat unui număr de pagină virtuală, aceasta este căutată mai întâi în TLB, iar dacă nu-i găsită în TLB, este căutată în tabela de pagini din memorie



Alocarea paginată /12

- **Traslation Look-aside Buffer (TLB) (cont.)**
  - Performanța adusă de folosirea TLB este dată de:
  - Hit_ratio = procentajul de găsiri în TLB a paginii căutate
  - ***Timpul efectiv de acces*:**

$$EAT = 1 * Memory\_AT + (Hit\_ratio * TLB\_AT + + (1 - Hit\_ratio) * (TLB\_AT + Memory\_AT))$$

  - Valori uzuale: Memory_AT >> TLB_AT și Hit_ratio ≈ 90-95% și ca urmare EAT este foarte apropiat de 1* Memory_AT

  - În tabela de pagini, paginile mai au asociați și alți biți de informație folosiți pentru **protecția memoriei** – restricții de acces (ReadAccess, WriteAccess, ExecuteAccess, ș.a.)

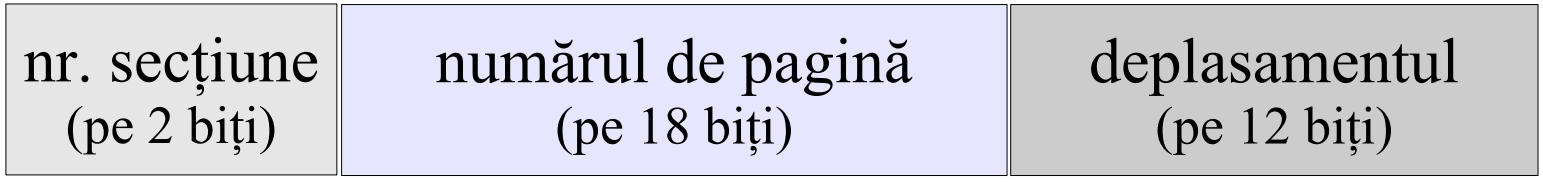
Alocarea paginată /13

- **Tabele de pagini pe nivele multiple**
  - Tabelele de pagini pot avea dimensiuni mari
  - Presupunem un spațiu de adresare de 2^{32} octeți, adică 4 Go (e.g. cazul arhitecturii x86 – pe 32 biți)
  - Dacă cadrul de pagină are 4 Ko, atunci numărul de intrări în tabela de pagini este de cca. un milion (!) (mai exact, în acest caz tabela are 2^{20} = 1.048.576 de intrări)
  - O tehnică utilizată pentru reducerea dimensiunii tabelei constă în utilizarea unei **tabele de pagini pe nivele multiple**

Alocarea paginată /14

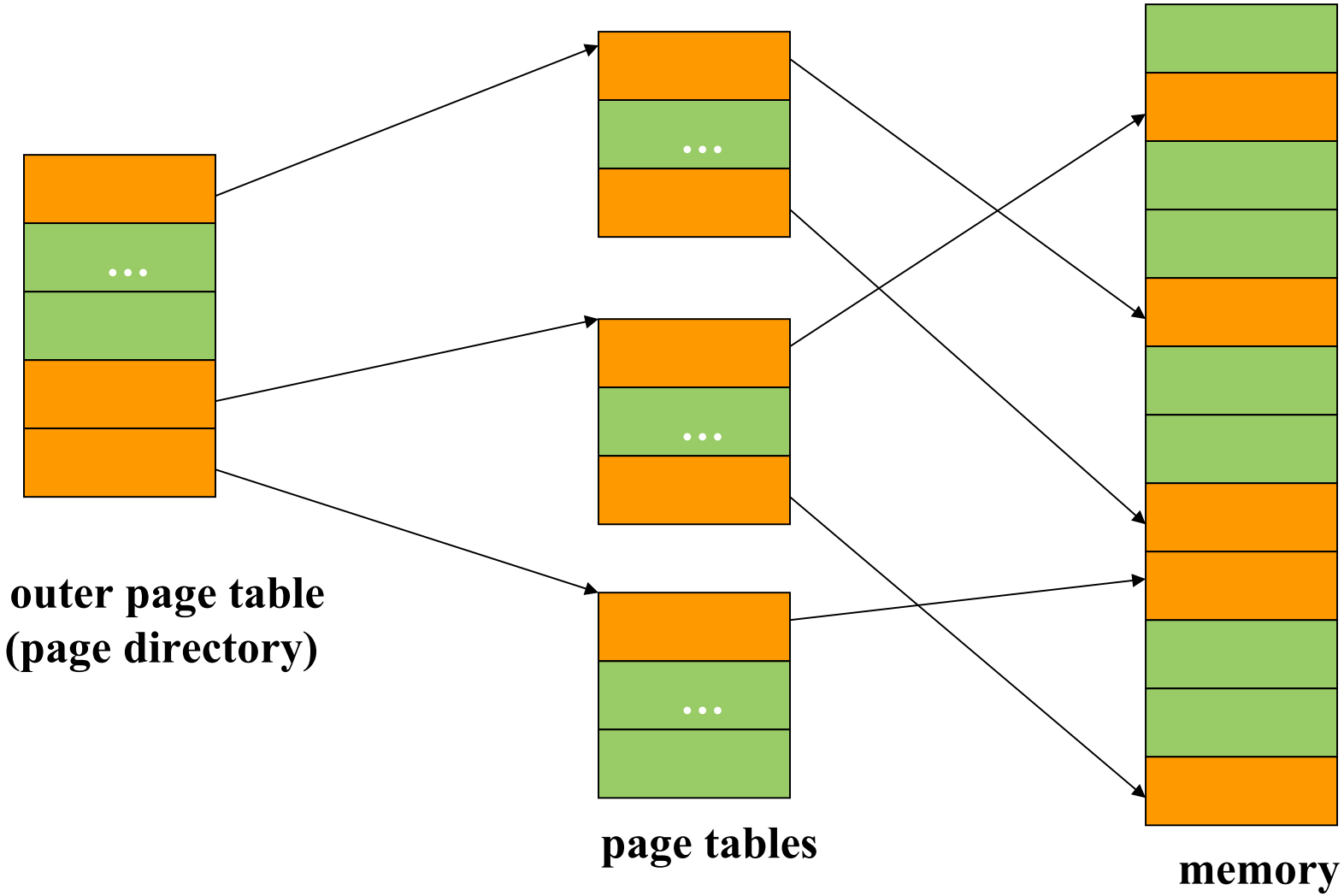
- **Tabele de pagini pe nivele multiple – exemplu:**
  - Spațiul de adrese: 2^{32} octeți, dimensiunea paginii: 4096 octeți
  - Deci vom avea 1M (i.e., 1.048.576) de intrări în tabela de pagini
  - O adresă logică arată astfel:

  - Partiționăm tabela de pagini în 4 "bucăți" (secțiuni), fiecare secțiune având 256K (i.e., 262.144) de intrări
  - Alocăm numai câte secțiuni sunt necesare, nu pe toate 4 !
  - Acum o adresă logică arată astfel:



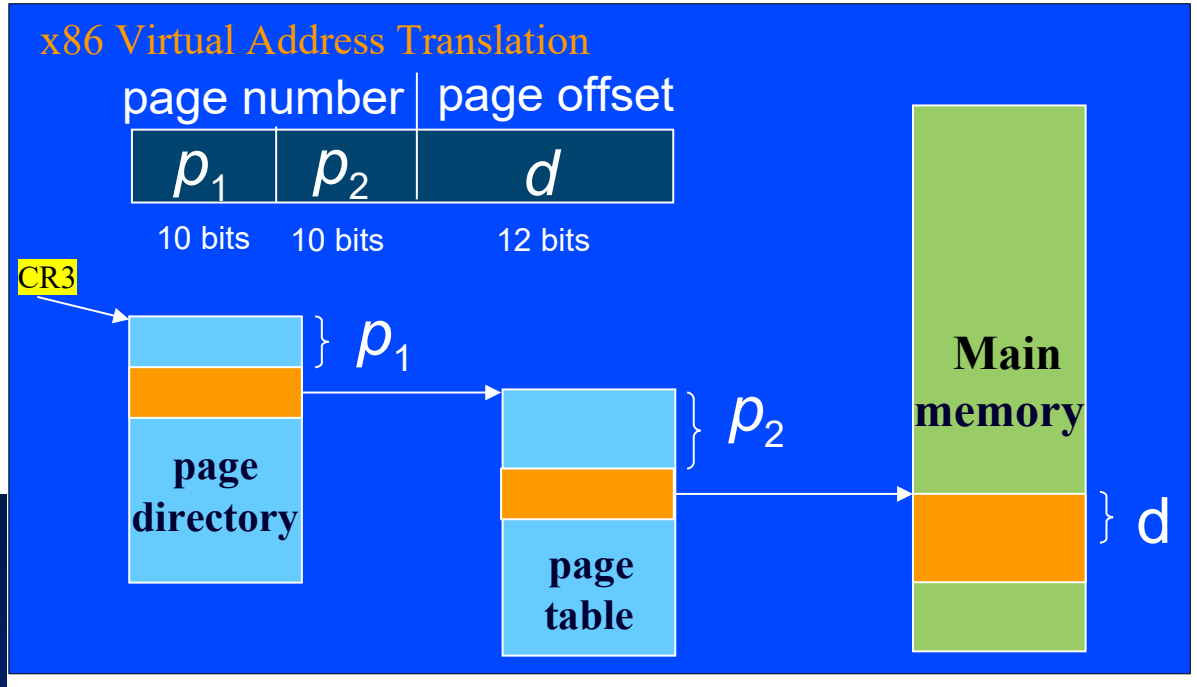
Alocarea paginată /15

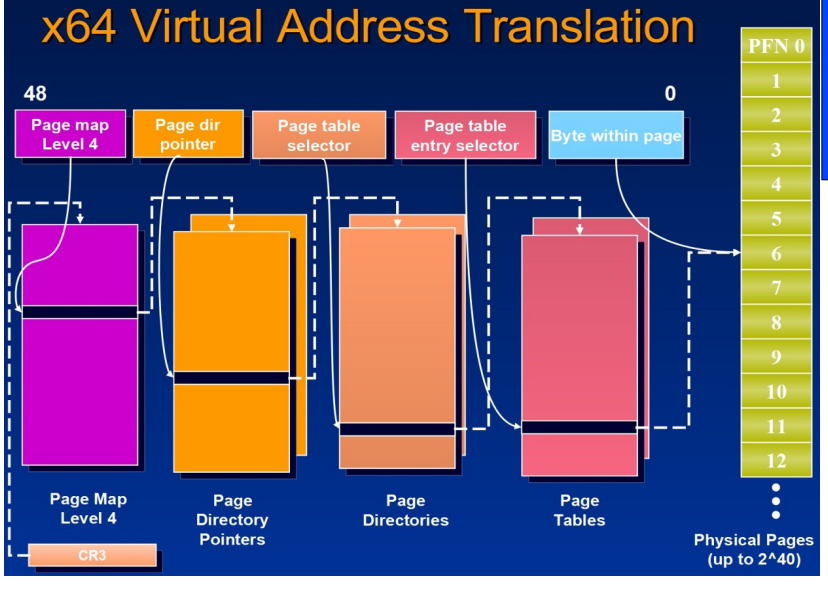
- **Tabele de pagini pe nivele multiple – exemplu:**



Tabele de pagini pe nivele multiple (cont.)

Translatarea adreselor pentru un CPU x86 (i.e., pe 32 biți) → 2 nivele de paginare:





Translatarea adreselor pentru un CPU x64 (i.e., pe 64 biți) → se utilizează doar 48 de biți pentru adresele logice și tabele de paginare cu 4 nivele (!)

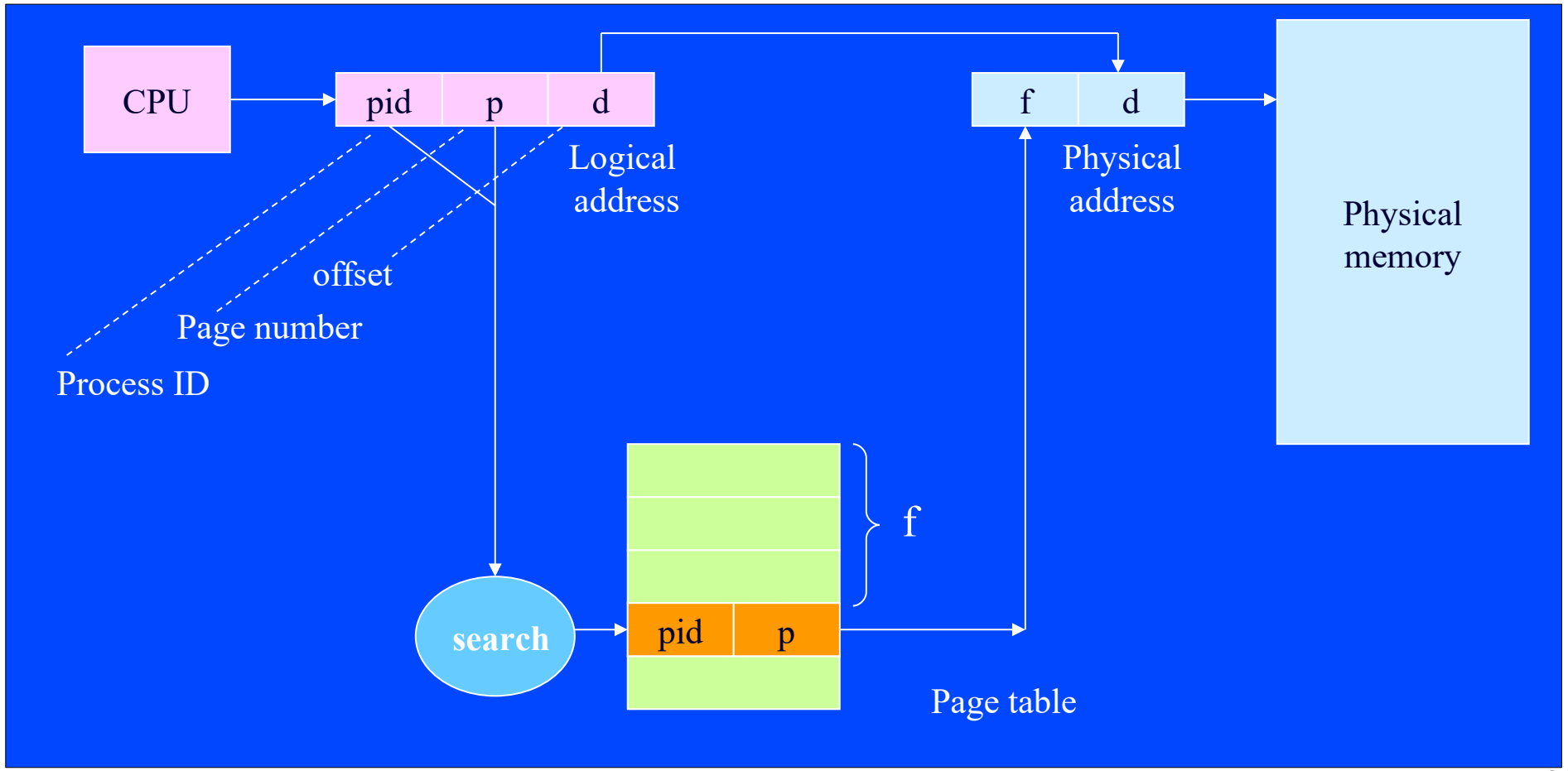
Notă: numărul de pagină virtuală (36 biți) se descompune în 4 grupuri (p_1, p_2, p_3 și p_4) de câte 9 biți fiecare.

Alocarea paginată /17

- **Tabele de pagini inverse**
  (utilizate de sistemele IBM RT, IBM AS/400, HP-UX)
  - O altă modalitate de reducere a dimensiunii tabelelor de pagini stocate în memorie
  - În loc de a avea o intrare în tabelă pentru fiecare pagină virtuală, avem câte o intrare pentru fiecare pagină fizică, ce conține o pereche de forma: (PID, număr pagină virtuală)
  - Când se translatează o adresă virtuală, se produce o căutare a numărului de pagină virtuală dorit, în tabela inversă de pagini, și se returnează numărul paginii fizice asociate, reprezentat de indexul intrării găsite

Alocarea paginată /18

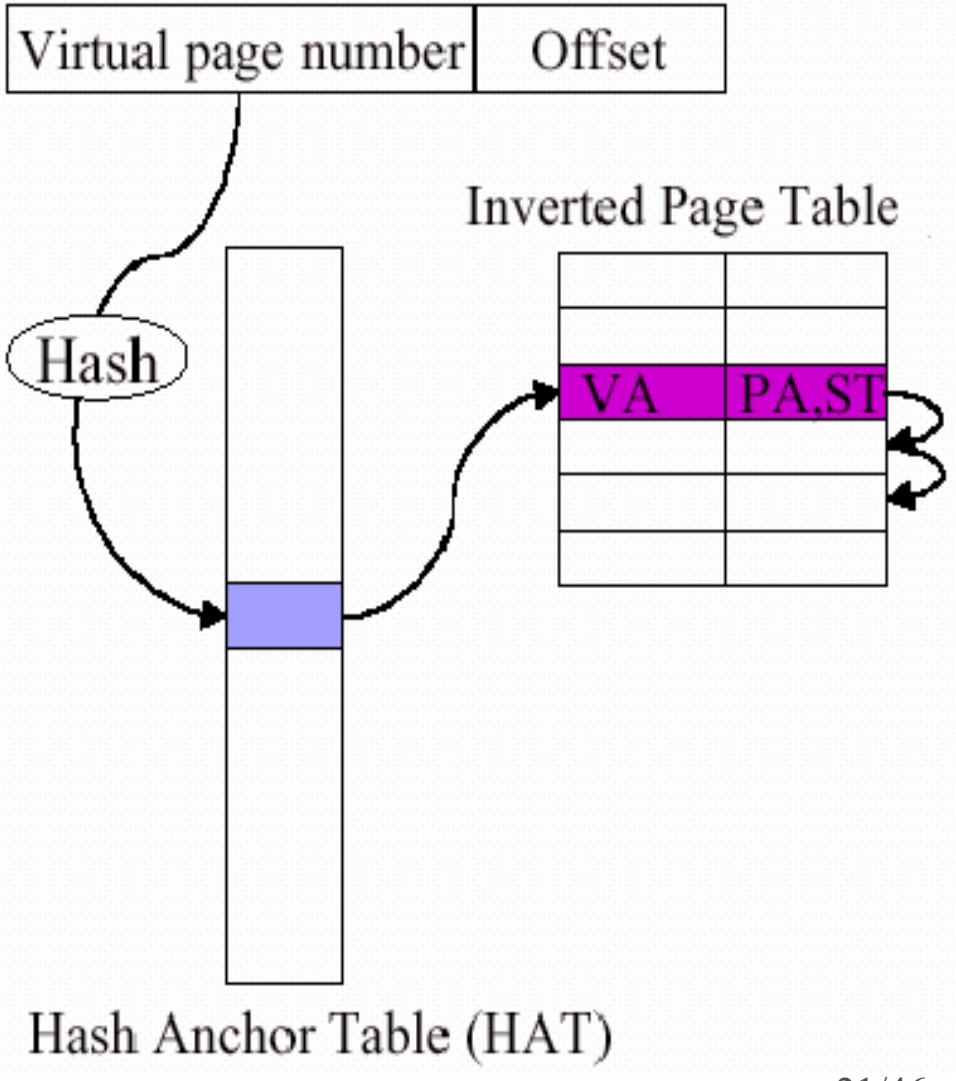
- **Tabele de pagini inverse** (cont.)



Alocarea paginată /19

- **Tabele de pagini inverse – dezavantaje:**
  - Prezintă dificultăți în suportul memoriei partajate (deoarece permite o singură adresă virtuală pentru o pagină fizică)
  - Căutarea este înceată (pentru că trebuie căutată întreaga tabelă)
    - Soluție: utilizarea tehnicii de *hashing*

Notă: despre funcții și tabele *hash* probabil ați învățat la disciplinele FAI sau PA; în caz că nu, puteți consulta următoarele referințe: funcție hash și tabelă hash.



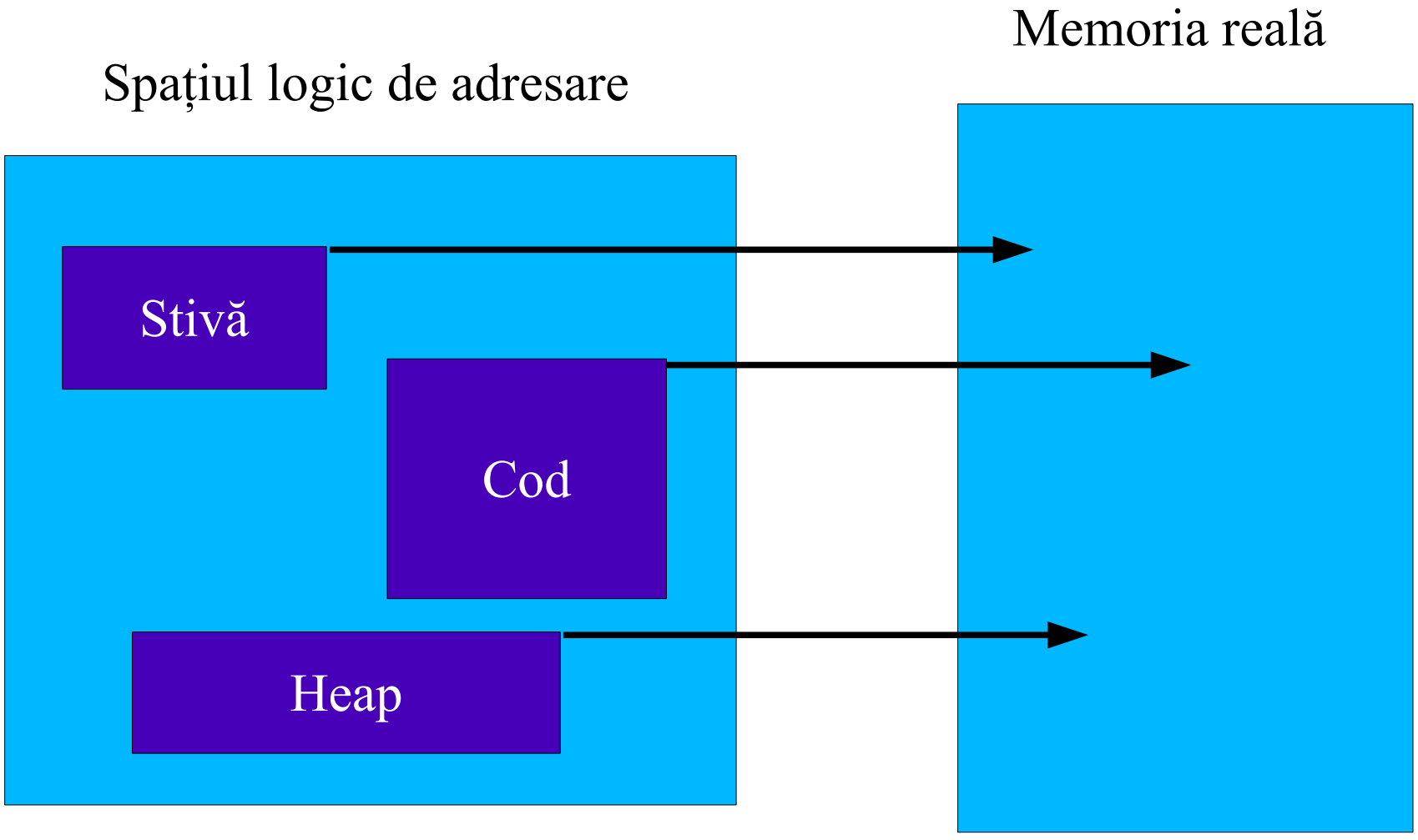
Segmentarea /1

- **Segmentarea** = "paginare" în segmente (i.e., pagini de dimensiuni variabile)
- Segmentarea reprezintă o vedere a memoriei d.p.d.v. al utilizatorului – o mulțime de "bucăți" de memorie de diverse dimensiuni:
  - programul principal
  - procedură
  - funcție
  - stivă
  - vector / matrice

Segmentarea /2

- Vederea utilizatorului asupra memoriei nu este aceeași cu organizarea memoriei fizice a SC-ului
- Fiecare din modulele și elementele de date ale programului (proceduri, funcții, stive, vectori, ...) este referit prin nume, fără ca utilizatorul să fie interesat ce adrese în memorie ocupă aceste elemente ale programului
- Spre deosebire, la paginare se practică o împărțire după organizarea memoriei fizice, transparentă pentru utilizator

Segmentarea /3



Segmentarea /4

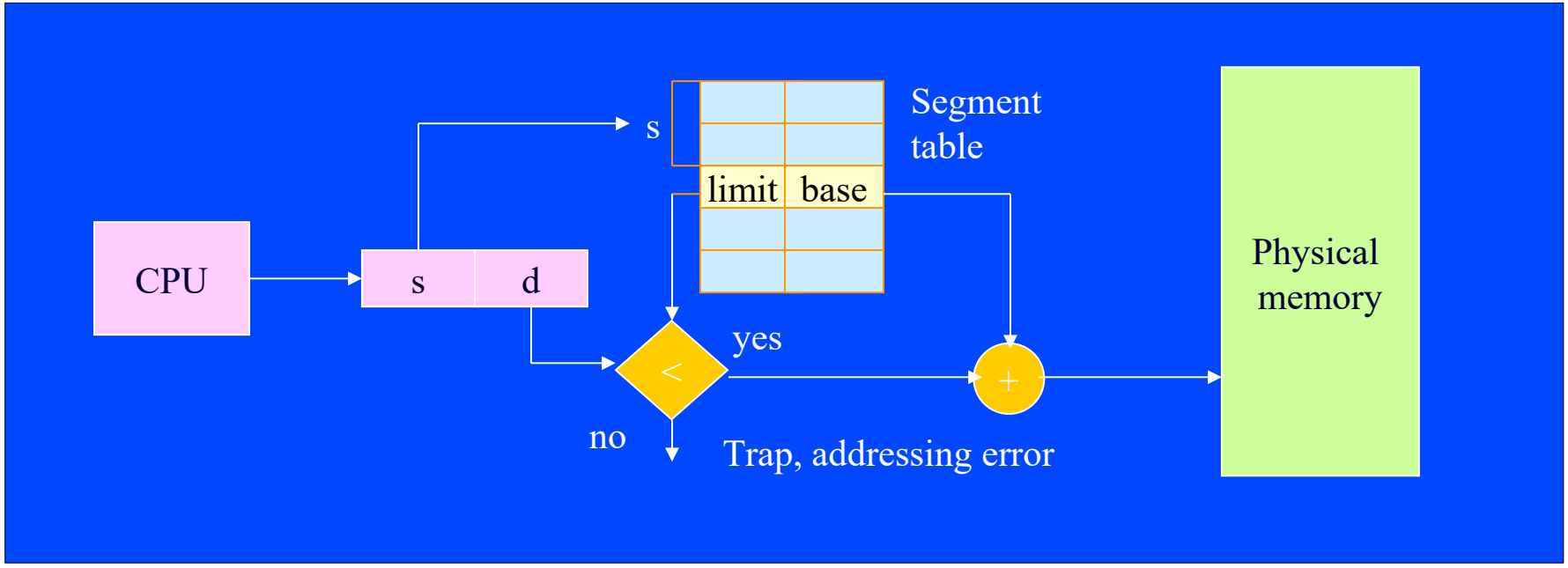
- **Segmentarea (cont.)**
  - Este o schemă de administrare a memoriei care suportă vederea utilizatorului asupra memoriei – programul este divizat în mai multe unități logice
  - Spațiul logic de adresare al programului constă dintr-o colecție de **segmente** (câte un segment pentru fiecare unitate logică din program)
  - Fiecare segment are un nume și o dimensiune
  - Adresele logice specifică atât numele segmentului, cât și deplasamentul în cadrul segmentului

Segmentarea /5

- **Segmentarea (cont.)**
  - Programatorul specifică fiecare adresă prin două componente: numele segmentului și deplasamentul, i.e. adresa virtuală = (segment , deplasament)
  - Deci programatorul vede spațiul virtual de adresare al programului ca un spațiu 2-dimensional, nu ca un spațiu 1-dimensional ca la celelalte metode
  - Fiecare proces are o *tabelă de segmente*, ce conține adresa de început a segmentului în memoria fizică

Segmentarea (cont.)

– Calculul adresei fizice se face ca la paginare:
adresa fizică = adresa segmentului + deplasamentul



Segmentarea /7

- **Implementarea segmentării**
  - CPU generează adrese virtuale: (nume segment, deplasament)
    Intel x86: segmentele CODE, DATA, STACK
  - Se caută în tabela de segmente adresa de început în memorie a segmentului, iar apoi se adună deplasamentul pentru a obține locația dorită din memoria fizică
  - Ca și la paginare, se pot folosi regiștri hardware specializați, care păstrează translatarea segmentelor în adrese reale
    Intel x86: registrul CS – adresa segmentului de cod, DS – adresa segmentului de date, SS – adresa segmentului de stivă, ES – adresa segmentului de date suplimentar (extra-segmentul)
  - Probabil nu toate translatările pot fi păstrate în regiștri, dacă există un număr mare de segmente

Segmentarea /8

- **Segmentarea - avantaje:**
  - Divizarea logică a spațiului virtual: toate porțiunile unui segment au probabil același înțeles semantic
  - Memoria reală este împărțită în fragmente logice
  - Partajarea memoriei este simplă: mai multe segmente care au aceeași adresă fizică; în particular, partajarea *codului reentrant*
  - Protecția memoriei – fiecare segment primește anumite drepturi de acces, trecute în tabela segmentelor; la fiecare calcul de adresă se pot face și astfel de verificări
  - Alte lucruri, precum gestiunea limitelor vectorilor se pot face mai simplu prin această metodă de alocare a memoriei

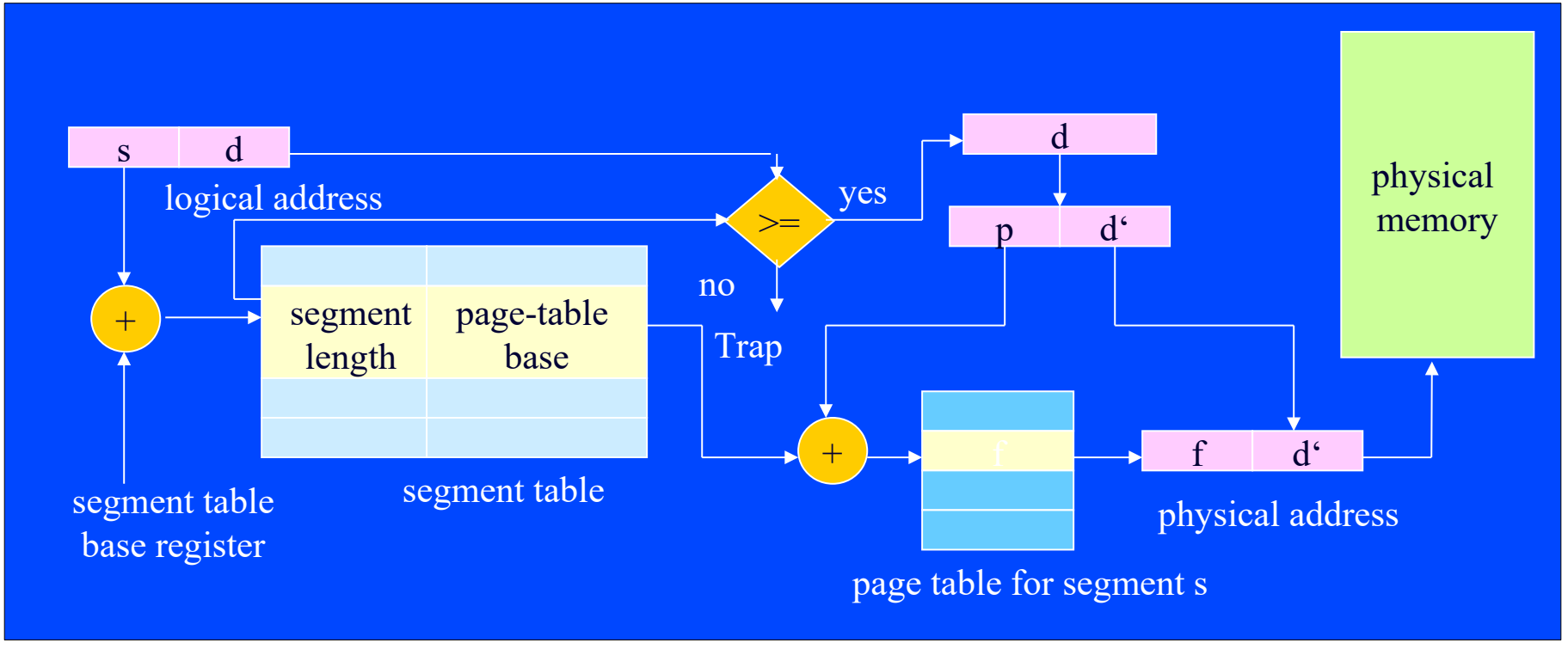
Segmentarea /9

- **Segmentarea - dezavantaje:**
  - Segmentele trebuie să încapă în memorie, deci nu pot avea lungimi mai mari decât dimensiunea memoriei fizice
  - Necesitatea ca fiecare segment să fie alocat într-o porțiune contiguă a memoriei fizice (folosind algoritmi gen FFA, BFA, WFA, ș.a.), plus dimensiunile variabile ale segmentelor, fac posibilă apariția *fragmentării externe* a memoriei
    - Pentru a înlătura fragmentarea se poate utiliza tehnica de *compactare* (i.e., defragmentarea memoriei)
    - O altă soluție: segmentarea paginată

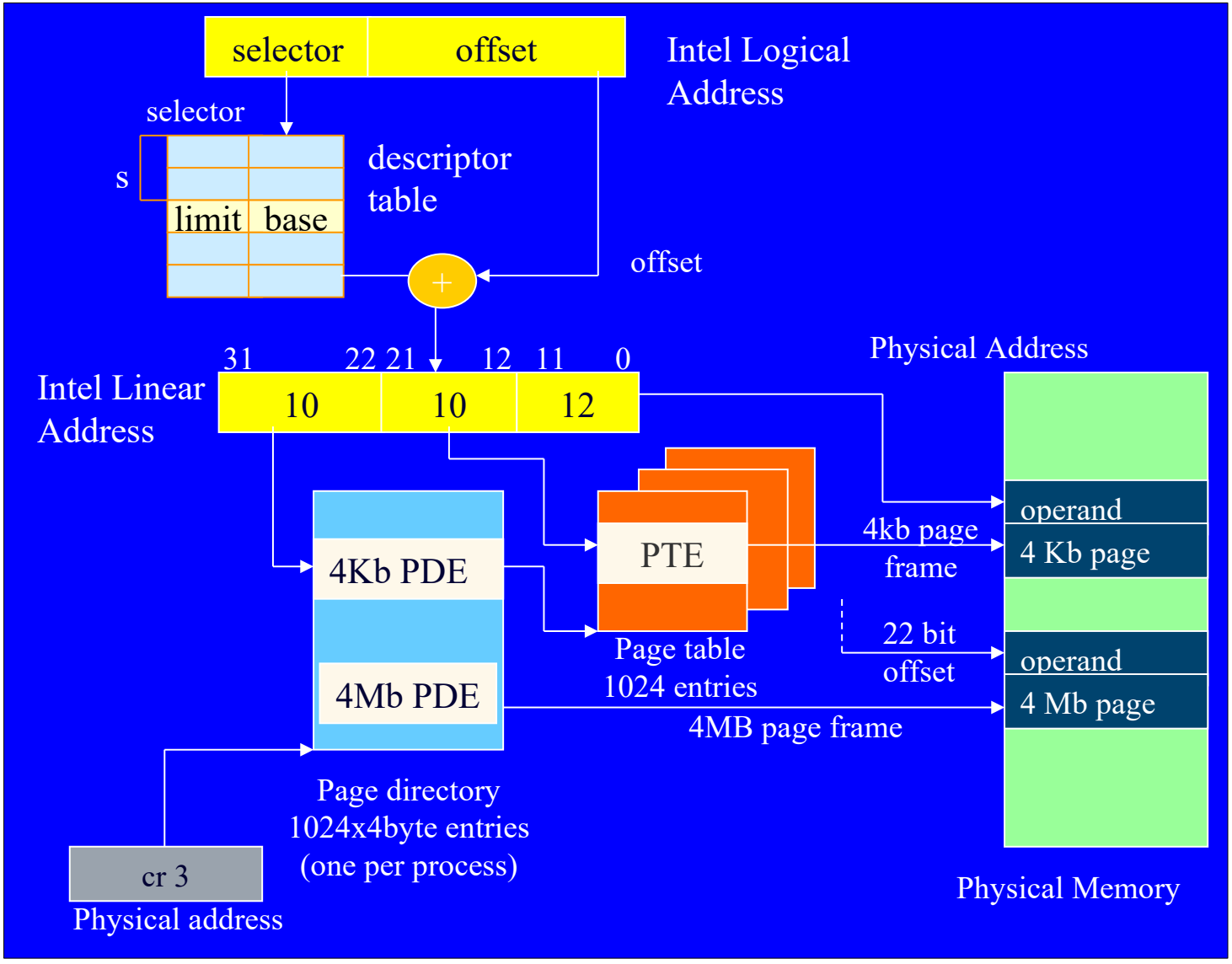
Segmentarea /10

- **Segmentarea paginată**
  - Introdusă de SO-ul MULTICS ('65-'69, AT&T + GE + MIT)
  - Fiecare segment este împărțit în pagini
  - Astfel se elimină două dezavantaje ale segmentării pure: alocarea contiguă și fragmentarea externă
  - Fiecare proces are o *tabelă de segmente* (care conține referințe către tabelele de pagini asociate segmentelor), iar fiecare segment are o *tabelă de (mapare a) paginilor* (în cadre de pagină din memoria fizică)
  - adresa virtuală = (segment, pagină, deplasament)
  - adresa fizică = (cadru de pagină, deplasament)

- **Segmentarea paginată (cont.)**
  - Folosită pe calculatorul GE 645 (cu SO-ul MULTICS)
    - Adrese logice = nr. segment: 18 biți + deplasament: 16 biți
    - Se utiliza un număr relativ mic de segmente de 64 Ko



Segmentarea paginată (cont.)



Arhitectura x86, începând de la micro-procesorul Intel 80386, permite prin hardware administrarea memoriei folosind segmentarea cu paginare pe 2 nivele.

- Memoria virtuală
  - Conceptul de memorie virtuală
  - Principiul localităţii

  (va urma)
  - Paginarea la cerere
  - Algoritmi de înlocuire a paginilor
  - Fenomenul de *trashing*
  - Segmentarea la cerere

Memorie virtuală /1

- **Memoria virtuală:** păstrează separarea spațiului de adresare logic (virtual) de cel fizic (real), introdusă de tehnicile de paginare și de segmentare pure, dar ...
- Acum, o parte din spațiul logic de adrese poate fi rezident, în fapt, pe disc (în situația în care memoria fizică este prea mică pentru a putea cuprinde în întregime spațiile logice de adrese ale tuturor proceselor active în sistem)

Memorie virtuală /2

- **Memoria virtuală**
  - SC-urile cu *memorie virtuală* au capacitatea de a adresa un spațiu de memorie mai mare decât memoria internă (i.e., RAM) disponibilă în sistemul respectiv
  - Ideea: *swapping*-ul pe disc al unor "bucăți" de memorie (tehnică introdusă de SO-ul ATLAS – 1960, Univ. Manchester, U.K.)
  - Avantaje:
    - Spațiul logic de adrese poate fi mult mai mare decât memoria fizică disponibilă
    - Este nevoie ca doar o parte a programului să fie rezident în memoria principală pentru a putea fi executat (deci nu mai este nevoie să fie prezent întregul program în memorie)

Memorie virtuală /3

- **Memoria virtuală**
  - Cum să o implementăm?
  - Este nevoie ca "bucăți" de memorie să fie *swap*-ate, la cerere, din memorie pe disc și invers
  - Implementare – tehnici de virtualizare folosite: **paginarea la cerere** (e.g. Windows, Linux) și **segmentarea la cerere** (e.g. OS/2)
  - Presupune mutarea, controlată de sistem, în susul și în josul ierarhiei de memorii (RAM ↔ disc)
  - Poate fi privită ca o automatizare a *overlay*-urilor (nu mai este sarcina programatorului de aplicații, ci a sistemului): SO-ul încearcă să determine în mod dinamic care "bucăți" încărcate anterior pot fi înlocuite de "bucăți" necesare acum

Memorie virtuală /4

- **Memoria virtuală**
  - Metoda funcționează doar datorită “localităţii” referințelor la memorie
  - Este adesea strâns legată de tehnica paginării / segmentării, deoarece necesită alocare necontiguă, tabelă de mapare, etc.: paginarea la cerere / segmentarea la cerere
  - Observație esențială: numai un subset din codul și, respectiv, datele unui program sunt necesare la un moment arbitrar de timp. Poate SO-ul să prezică care va fi acel subset la un moment următor, doar din observația comportamentului programului la momentele trecute de timp ?
    → Principiul localităţii (principiu euristic, desprins din practică)

Principiul localităţii /1

- **Principiul localităţii (vecinătăţii)** ('68 P.J. Denning)
  - Două tipuri de localizare a secvențelor de accesare a memoriei:
    - **Localitate temporală** – tendința de a accesa în viitorul apropiat locații accesate deja în istoria recentă
      (e.g. instrucțiuni repetitive, variabile utilizate frecvent)
    - **Localitate spațială** – tendința de a accesa în viitorul apropiat locații apropiate de alte locații accesate deja în istoria recentă
      (e.g. instrucțiuni secvențiale, structuri de date contigue – vectori, înregistrări, etc.)
      Se justifică astfel mutarea în "bucăți" mai mari pt. swapping

Principiul localităţii /2

- **Principiul localităţii** (cont.)
  - Datorită lui s-a impus programarea structurată (în locul programării cu goto), precum și structurarea datelor
  - *Morala*: dacă o matrice este introdusă în memorie pe linii (coloane), este bine să fie prelucrată tot pe linii (resp. pe coloane)

localitate bună:

```
for (i = 0; i++; i<n)
    for (j = 0; j++; j<m)
        A[i, j] = B[i, j]
```

localitate rea:

```
for (j = 0; j++; j<m)
    for (i = 0; i++; i<n)
        A[i, j] = B[i, j]
```

Assume: arrays laid out in rows

A[0,0]	A[0,1]
A[1,0]	A[1,1]
A[2,0]	A[2,1]

→ A[0,0] A[0,1] A[1,0] A[1,1] A[2,0] A[2,1]

Bad locality is a contributing factor in ***Thrashing*** (page faulting behavior dominates).

Explicații: a se vedea pe verso.

Principiul localităţii /3

➢ **Principiul localităţii** (cont.)

– Explicații suplimentare:

Exemplul anterior trebuie gândit pe matrici mari, e.g. o matrice 1024x1024 de numere de tip int (deci o celulă ocupă 4 octeți). Atunci fiecare linie a matricii va ocupa exact o pagină în memorie.
Întrucât matricile sunt "liniarizate" pe linii, nu pe coloane, atunci când compilatorul de C le alocă spațiu de stocare în memorie, rezultă că în codul din stânga, rata *miss*-urilor este egală cu 1024 / *numărul total de accese* , iar în codul din dreapta rata *miss*-urilor este: 1024x1024 / *numărul total de accese* ! Iar *numărul total de accese* este: 1024x1024, în ambele cazuri.

Observație finală: aceste rate de *miss*, calculate mai sus, se aplică în ambele situații, cea descrisă în acest curs (și anume, la calculul estimativ al duratei accesului la memorie, ținând cont de maparea adresei paginii virtuale în adresa paginii fizice, pe baza *cache*-ului TLB; revedeți formula de la slide-ul #14), cât și într-o situație descrisă în cursul viitor (și anume, la calculul ratei de *page fault*-uri, pentru un algoritm de *page-swapping* cum este alg. LRU, care păstrează cele mai recente valori, întocmai precum un *cache*).

Va urma:

- **Tehnici de implementare**
  - Paginarea cu încărcare la cerere
    (= paginarea combinată cu tehnica de *swapping*)
  - Segmentarea cu încărcare la cerere
    (= segmentarea combinată cu tehnica de *swapping*)
- **Algoritmi de *swapping* a paginilor/segmentelor**

Bibliografie

- **Bibliografie obligatorie**

  capitolele despre *gestiunea memoriei* din

  – Silberschatz : "*Operating System Concepts*"

  (cap.9, ultima parte: §9.3−8, din [OSC10])

  sau

  – Tanenbaum : "*Modern Operating Systems*"
  (cap.3, §3.3 și §3.7, din [MOS4])

Exercițiu de seminar

- Aplicație la: Alocarea paginată
  - **Enunț**: Se consideră un sistem cu paginare pură (i.e., nu la cerere) ce păstrează tabela de pagini în memorie.

    a) Dacă un acces la memoria fizică durează 225 ns (1 ns=10^{-9}s), cât timp va necesita referirea la o adresă logică ?

    b) Dacă se adaugă un registru asociativ (TLB), cu rol de *cache* a tabelei de pagini din memorie, și dacă 85% din toate referirile la tabela de pagini sunt găsite în acest registru asociativ, care va fi timpul efectiv de acces la o adresă logică ? (Presupunem că găsirea unei intrări din tabela de pagini în registrul asociativ, în caz că există, necesită timp zero.)

    c) Care este rata (exprimată în procente) de găsire în registrul asociativ a intrării căutate, minim acceptabilă pentru a avea un timp de acces efectiv nu mai mare de 243 ns ?

    Justificați răspunsurile.

  - **Rezolvare**: ?

Sumar

Alocarea memoriei (continuare)

- Scheme de alocare necontigue
  - Paginarea
  - Segmentarea
  - Segmentarea paginată

(va urma)

- Scheme de alocare cu memorie virtuală

Întrebări ?