

# Sisteme de Operare

## Administrarea memoriei partea a III-a

**Cristian Vidrașcu**

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

- Memoria virtuală (continuare)
  - Paginarea la cerere
  - MMU
  - Algoritmi de înlocuire a paginilor
  - Fenomenul de *trashing*
  - Segmentarea la cerere
- Concluzii legate de tehnicile de administrare a memoriei
- Tehnici recente pentru administrarea memoriei
- Administrarea memoriei în Linux și Windows

## ➤ **Paginarea (cu încărcare) la cerere**

= paginarea combinată cu tehnica de *swapping*

- **Ideea:** aducerea paginilor de pe disc în memorie doar în momentul când sunt referite (“când e nevoie de ele”)
- Astfel se elimină restricția ca programul să fie prezent în întregime în memorie pentru a putea fi executat
- Motive de eliminare a acestei restricții:
  - programele conțin zone de cod ce tratează cazuri mutual-exclusive (în funcție de datele de intrare), sau zone care se execută la momente de timp mutual-exclusive
  - unele programe conțin zone de date f. mari (e.g. compilatoarele)
  - Programele pot avea multe rutine de tratare a unor erori posibile, ce vor fi apelate în timpul execuției doar dacă apar acele erori

## ➤ Paginarea la cerere (cont.)

### – Avantaje:

- programul este prezent doar parțial în memorie
- mai puține operații I/O decât la transferul proceselor în întregime (de la schema de administrare cu *swapping* pur)
- mai puțină memorie necesară la un moment dat
- mai multe procese la un moment dat (crește gradul de multi-programare)
- nu mai e nevoie de tehnica de programare a *overlay*-urilor, efortul programatorului fiind preluat de S.O.

### – Dezavantaje:

- complexitatea hardware și software a acestui mecanism de gestiune a memoriei

## ➤ **Paginarea la cerere (cont.)**

- Când este referită o pagină, trebuie verificat dacă:
  - Este o **referință** validă?  
Cu alte cuvinte, procesul are dreptul de a accesa  
acea adresă?
  - Pagina referită este în memorie?  
Dacă nu, caută un cadru de pagină liber și încarcă  
pagina de pe disc în el.

# Paginarea la cerere /4

## ➤ Paginarea la cerere (cont.)

- Pentru a păstra evidența paginilor aflate în memorie, se asociază, pentru fiecare pagină, un bit **validă sau invalidă** în *tabela de mapare a paginilor*

Pagina	Numărul cadrului	Bit (in)validă
1	300	1
2	85	0

## ➤ **Erori de pagină**

- Semnificația bitului: 1 – pagina este prezentă în memorie; 0 – pagina nu este în memorie
- **Eroare de pagină** (*page fault*): atunci când se încearcă accesarea unei pagini marcate ca invalidă (bitul este 0); pagina nu este în memorie și va trebui adusă de pe disc
- Se generează o **întrerupere de pagină** (PFI=*page fault interrupt*), de tip sincron, care este transparentă pentru utilizator și are o prioritate superioară celorlalte întreruperi; ea va întrerupe execuția programului și S.O.-ul va executa handler-ul asociat acestei întreruperi

## ➤ **Erori de pagină (cont.)**

– Rutina de tratare a întreruperii de pagină execută:

- Se examinează adresa solicitată pentru a vedea dacă este o adresă permisă; dacă nu este, procesul va fi terminat anormal
- Se caută un cadru de pagină liber în memoria principală și se solicită o operație I/O care va aduce pagina de pe disc în cadrul liber găsit; pentru aceasta se mai folosește și o *tabelă de mapare pe disc*, ce conține adresa de pe disc a fiecărei pagini virtuale
- După încărcarea paginii, se actualizează *tabela de mapare a paginilor* pentru a indica faptul că pagina este validă
- Apoi este reluată execuția procesului – instrucțiunea oprită este restartată (toată această procedură este transparentă pentru proces)



## ➤ **Erori de pagină (cont.)**

- Este posibil să apară *mai multe* erori de pagină la execuția unei singure instrucțiuni – cazul extrem: codul instrucțiunii “călare” pe 2 pagini, cu 2 operanzi fiecare “călare” pe 2 pagini, cu adresare indirectă/indexată – pot apare 3 întreruperi PFI
- Inconvenientul acestei metode: *overhead*-ul implicat în cazul erorilor de pagină (necesită accese la disc, plus folosirea CPU pentru ajustarea tabelelor)
- O memorie fizică mică și prea multe procese în sistem, ori cu o localitate proastă a datelor și/sau codului, pot conduce la un număr f. mare de întreruperi PFI, deci la apariția fenomenului de **trashing** (i.e. fenomenul de sufocare a SC-ului cu rezolvarea erorilor de pagină, în loc de a-și folosi resursele pentru execuția job-urilor utile)

## ➤ **Paginarea la cerere (cont.)**

- O altă problemă: la o eroare de pagină, ce facem dacă nu găsim nici un cadru fizic liber?
- Soluția: sacrificarea unei pagini din memorie – mutarea ei pe disc pentru a elibera spațiu în memorie pt. pagina ce se încarcă
- Cum alegem victima pentru *page swapping*?  
Există mai multe criterii de alegere, mai mulți algoritmi pe care-i vom discuta puțin mai târziu
- Ideea: folosirea unui algoritm de înlocuire a paginilor care **să minimizeze numărul de erori de pagină** (i.e., de întreruperi PFI)
- În plus, se poate preveni supra-alocarea de memorie unui anumit proces – rutina de tratare a PFI poate decide să facă niște înlocuiri chiar dacă mai sunt cadre fizice libere

## ➤ **Paginarea la cerere (cont.)**

- O altă problemă: pe lângă bitul de pagină (in)validă, mai este nevoie de un bit de “tranzit” care să indice starea de pagină în curs de încărcare de pe disc în memorie
- Motivul: transferul de pe disc durează f. mult, timp în care CPU execută alt proces, și este posibil ca acesta să aleagă ca victimă pentru sacrificiu tocmai pagina în curs de încărcare
- Regulă: când aleg victima, evit paginile aflate în starea de tranzit
- *Observație*: tabela de mapare a paginilor unui proces este păstrată și ea în memoria acelui proces; pagina ce conține tabela nu trebuie evacuată din memorie de către alg. de *page swapping*, i.e. ea este “încuiată” pe toată durata execuției procesului, pe când paginile în tranzit sunt “încuiate” doar temporar, pe durata încărcării

# Paginarea la cerere /10

## ➤ **Paginarea la cerere (cont.)**

- O altă optimizare: fiecărei pagini  $i$  se asociază în tabela de mapare a paginilor un al 3-lea bit, bitul *dirty*, care determină dacă pagina a fost vreodată modificată de la ultima încărcare de pe disc
- Inițial, la încărcare, bitul *dirty* este pus pe zero, iar apoi orice scriere în acea pagină îl setează pe 1 (se face  $\text{bit} := \text{bit} \text{ or } 1$ )
- Rostul acestui bit *dirty* – optimizarea operațiilor I/O: dacă pagina nu a fost modificată de la ultima încărcare de pe disc, atunci ea nu mai trebuie transferată pe disc atunci când este aleasă drept victimă de către algoritmul de *page swapping*

# Paginarea la cerere /11

- **Concluzii – întrebări legate de paginarea la cerere:**
  - Cum împiedicăm utilizatorii să acceseze datele protejate?
    - drepturi de acces specificate la nivel de pagină
  - Dacă o pagină este prezentă în memorie, cum o găsim?
    - tabela de mapare a paginilor (în memoria fizică)
  - Dacă o pagină nu este prezentă în memorie, cum o găsim?
    - tabela de mapare (a paginilor) pe disc
  - Când este adusă o pagină în memorie?
    - politici de încărcare – la cerere (atunci când este nevoie de ea)
  - Dacă o pagină este adusă în memorie, unde o punem?
    - politici de plasare
  - Dacă o pagină este evacuată din memorie, unde o punem?
  - Cum decidem care pagini să fie evacuate din memorie?
    - politici de înlocuire – algoritmi de *page swapping*

# Paginarea la cerere /12

- **Mecanisme – pentru suportul paginării la cerere:**
  - **Suportul hardware** – pe lângă translatarea dinamică a adreselor necesară pentru suportul paginării sau segmentării (e.g. tabelele de mapare):
    - Mecanism de generare a erorilor de pagină (i.e. PFI) în situația accesării paginilor absente din memorie
    - Instrucțiuni restartabile
  - **Suportul software:**
    - Structuri de date pentru suportul politicilor de înlocuire, încărcare și plasare a paginilor în memorie
    - Structuri de date pentru localizarea în memoria secundară (i.e., pe disc) a paginii dorite

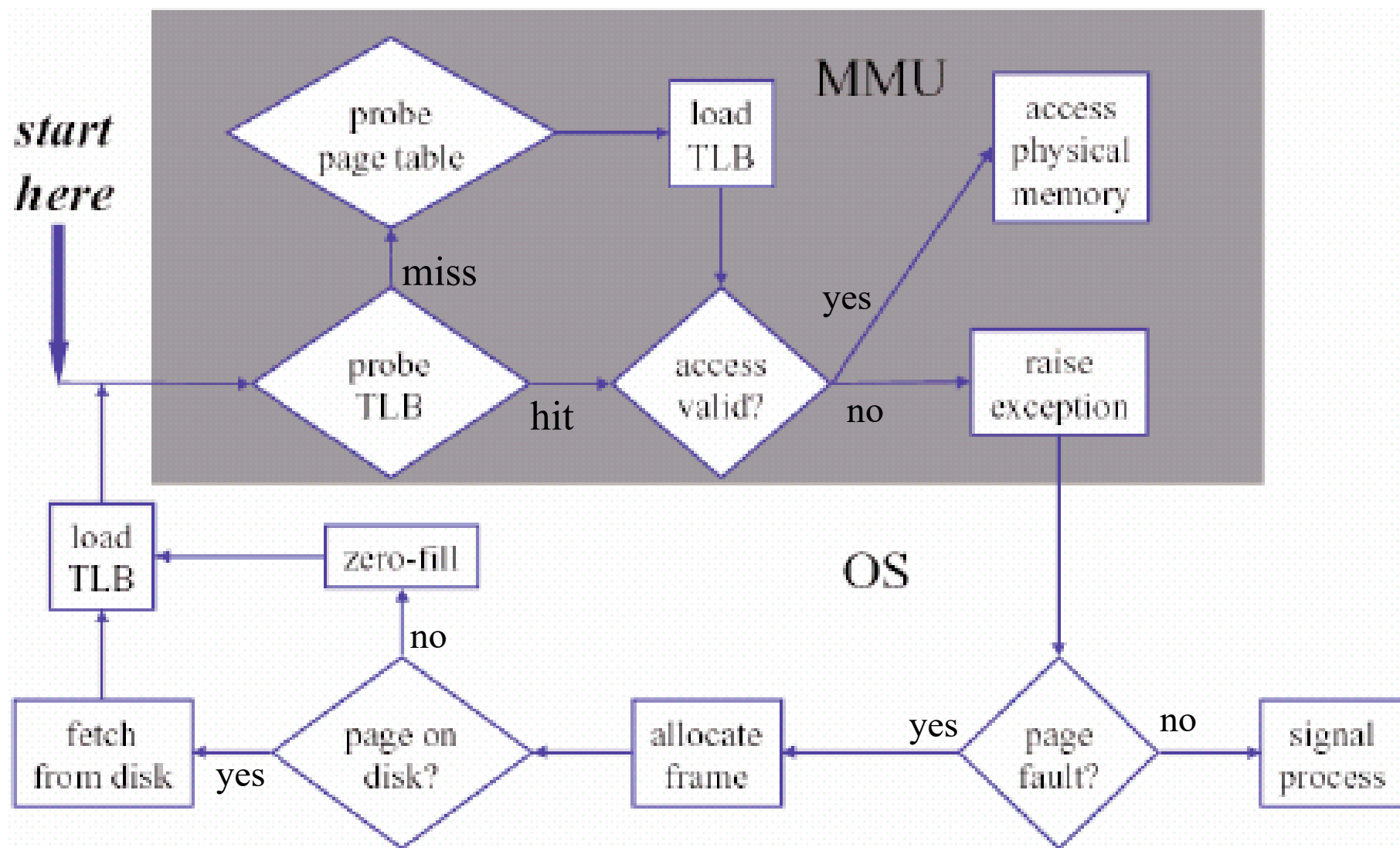
- **MMU (= Memory Management Unit**, located in the CPU)
  - Input: adrese virtuale
  - Output: adresele fizice asociate, sau diverse situații excepționale, de violare a accesului (diverse excepții, care întrerup procesorul, i.e. execuția instrucțiunii ce cauzează acea excepție)
  - Situații (tipuri de excepții) de violare a accesului:
    - pagină absentă din memorie (**excepție PFI**)
    - acces în mod utilizator vs. mod kernel
    - lipsa dreptului de read a memoriei
    - lipsa dreptului de write a memoriei
    - lipsa dreptului de execute a memoriei (pt. paginile de cod)
    - alte excepții, e.g. *guard page* și *COW page* (Copy-On-Write)

## ➤ MMU (cont.)

- SO-ul controlează operarea MMU pentru a selecta:
  1. Submulțimea de posibile adrese virtuale ce sunt valide pentru fiecare proces (i.e., spațiul virtual de adrese al procesului)
  2. Traslatările (mapările) fizice pentru acele adrese virtuale
  3. Modurile de acces permis la acele adrese virtuale (read/write/execute)
  4. Setul specific de translatări valabil la un moment dat (este necesar un *context-switch* rapid de la un spațiu de adrese la altul)
- MMU finalizează un acces la memorie numai dacă SO-ul spune că “referința e OK” (altfel, MMU produce o excepție)



➤ **MMU – finalizarea unui acces la memorie:**



# Alg. de page swapping /1

## ➤ Algoritmi de înlocuire a paginilor

- Alg. de înlocuire a unei pagini neutilizate recent: NRU
- Alg. de înlocuire în ordinea încărcării: FIFO
- Alg. de înlocuire a celei mai puțin utilizate recent pagini: LRU
- Aproximări LRU pentru implementări
- Alte abordări
- Implementări reale

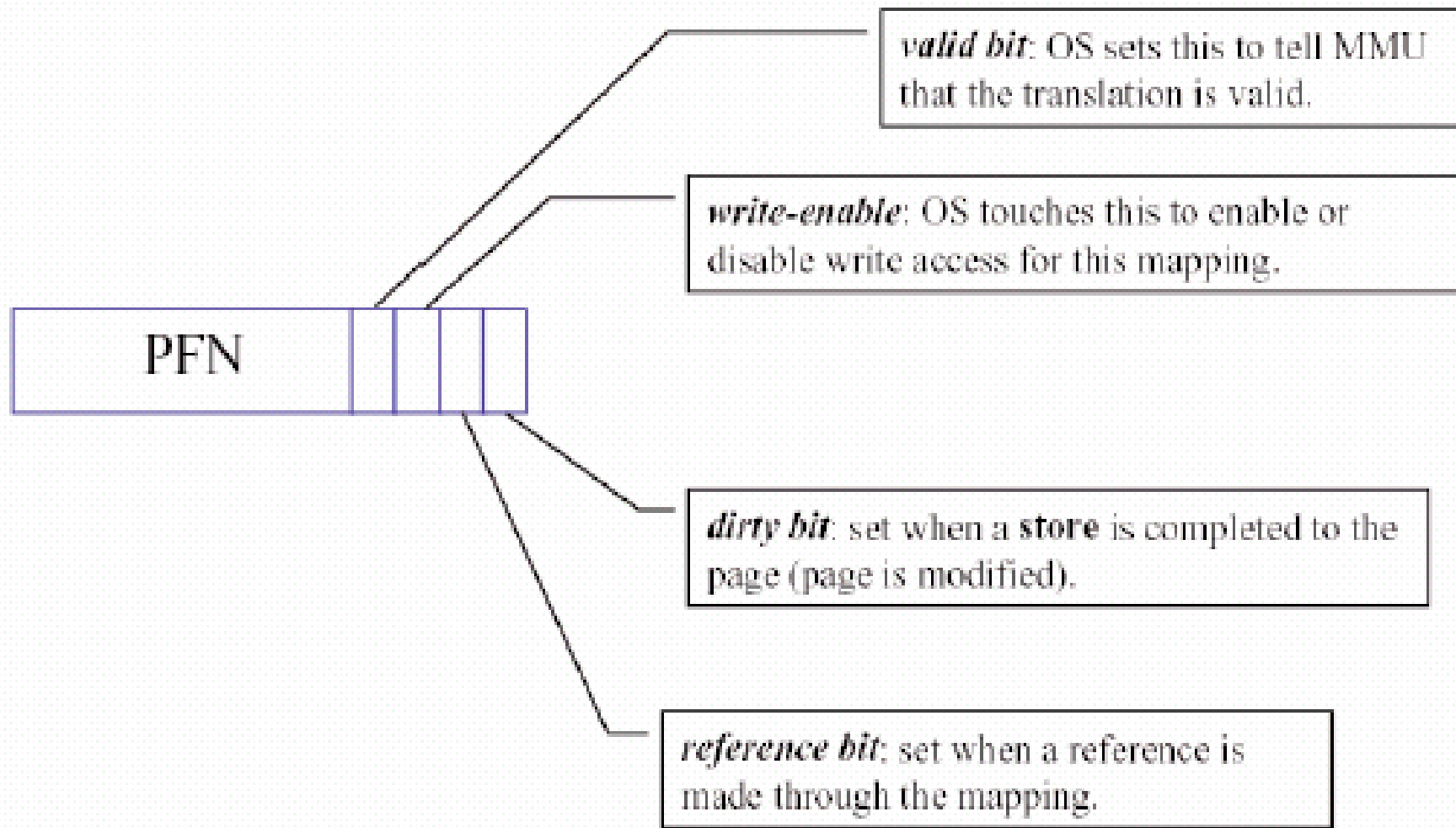
# Alg. de page swapping /2

## ➤ **Politica de înlocuire a paginilor**

- Când nu mai sunt disponibile cadre de pagină libere, SO-ul trebuie să înlocuiască o pagină (**victima**), înlăturând-o din memorie pentru a rămâne rezidentă doar pe disc (**depoziul pentru copia de siguranță**) și scriindu-i conținutul înapoi pe disc dacă fusese modificată după încărcarea în memorie (**pagina dirty**)
- **Algoritmul de înlocuire** – are drept scop alegerea celei mai bune victime, metrica utilizată de obicei pentru “cea mai bună” fiind reducerea ratei erorilor de pagină (i.e., se urmărește minimizarea numărului de întreruperi PFI)

# Alg. de page swapping /3

## ➤ Intrările în tabela de mapare a paginilor



# Alg. de page swapping /4

- **Problema *caching*-ului paginilor** (politica de înlocuire)
  - Fiecare thread/proces “produce” un flux de referințe la paginile virtuale din propriul spațiu de adresare
    - Vom modela execuția ca o **secvență de referințe la pagini**:  
e.g. “1,2,3,1,2,3,4,2,3,5,...”
  - S.O. încearcă să minimizeze numărul de erori de pagină produse
    - **Mulțimea de lucru** (*working set*) = mulțimea de pagini folosite efectiv de fiecare proces, se modifică în timp relativ încet
    - SO-ul încearcă să “aranjeze” mulțimea de pagini rezidente în memorie a fiecărui proces activ astfel încât să aproximeze cât mai bine mulțimea sa de lucru
  - Factorul determinant pt. succes este politica de înlocuire folosită
    - Aceasta determină **mulțimea de pagini rezidente** în memorie

# Alg. de page swapping /5

## ➤ Algoritmul optim (inutilizabil)

- Răspunsul optim la întrebarea **care** pagină o aleg drept victimă? este simplu, dar imposibil de realizat:  
*se alege acea pagină care va fi solicitată cel mai târziu în viitor*
- Inutilizabil deoarece răspunsul nu poate fi prevăzut în avans – evoluția unui program nu este previzibilă, ea depinde de datele concrete asupra cărora operează
- Au existat încercări, dar nu erau practice:  
'66 L.A. Belady – un model de evidență statistică prin care se putea prevedea *cu o oarecare probabilitate* care este pagina ce va fi solicitată cel mai târziu

## ➤ Algoritmi practici

- Metode folosite: NRU, FIFO, LRU și alte variante
- Observație: indiferent de alg. de decizie folosit, trebuie avut în vedere faptul că, pentru implementare, gestiunea unei structuri de date care să permită decizia, trebuie făcută *la fiecare acces la memorie*
- Prin urmare, nu este permisă nici măcar gestiunea unei liste liniare simplu înlănțuite (!), ci este nevoie de mecanisme mult mai ușor de gestionat
- De multe ori aceste metode se implementează prin hardware, făcându-se uneori anumite compromisuri

# Alg. de page swapping /7

## ➤ Algoritmul NRU (= Not Recently Used)

- Ideea: înlocuirea unei pagini care nu a fost utilizată recent
- Fiecare pagină fizică are asociați doi biți în tabela de paginare, folosiți pentru a decide ce pagină se va evacua
- Bitul de *referire* este resetat (pus pe 0) la încărcarea paginii și este setat (pus pe 1) la fiecare acces (referire) la pagină
- Periodic (de obicei la 20ms) se face operația de *clearing bits*, prin care biții de referire ai tuturor paginilor sunt resetați
- Al doilea bit este bitul de *modificare* (bitul *dirty*), ce este resetat la încărcarea paginii și setat la scrierea în pagină



# Alg. de page swapping /8

## ➤ Algoritmul NRU

- Paginile se împart la fiecare moment în patru clase, pe baza acestor biți:
  - Clasa 0 – biții=(0,0) : pagini nereferite și nemodificate
  - Clasa 1 – biții=(0,1) : pagini nereferite (de la ultimul *clearing*), dar modificate (de la încărcarea lor)
  - Clasa 2 – biții=(1,0) : pagini referite, dar nemodificate
  - Clasa 3 – biții=(1,1) : pagini referite și modificate
- Pagina “victimă” se alege din prima clasă nevidă (se caută întâi în clasa 0, apoi în clasa 1, ș.a.m.d.); dacă pagina aleasă este din clasa 1 sau 3, ea va fi salvată pe disc înainte de a fi înlocuită
- Avantaje: alg. f. simplu; nu-i optimal, dar e eficient în practică

# Alg. de page swapping /9

## ➤ Algoritmul FIFO (= First In First Out)

- Ideea: înlocuirea în ordinea încărcării paginilor
- Implementarea este simplă: se gestionează o listă FIFO cu paginile rezidente în memorie; actualizarea ei se face *la fiecare încărcare de pagină* (nu la fiecare acces la memorie!)
- Alegerea “victimei”: prima pagină din listă (i.e. se va evacua cea mai veche pagină din memorie)
- Bineînțeles, și aici se folosește bitul de *modificare* (bitul *dirty*), pentru a ști dacă pagina aleasă trebuie salvată pe disc înainte de a fi înlocuită

# Alg. de page swapping /10

## ➤ Algoritmul FIFO

- Exemplu: un program cu 5 pagini virtuale, cu secvența de referire “1,2,3,4,1,2,5,1,2,3,4,5”, iar memoria fizică: 3 cadre de pagină

Pagina solicitată	1	2	3	4	1	2	5	1	2	3	4	5	→ timp	
Pagina cea mai recentă	—	1	2	3	4	1	2	5	5	5	3	4		4
Lista FIFO:	—	—	1	2	3	4	1	2	2	2	5	3		3
Pagina cea mai veche	—	—	—	1	2	3	4	1	1	1	2	5		5
Înlocuire de pagină (PFI):	Da	Da	Da	Da	Da	Da	Da	Nu	Nu	Da	Da	Nu		
Memoria RAM:	Frame 0	—	1	1	1	4	4	4	5	5	5	5	5	
	Frame 1	—	—	2	2	2	1	1	1	1	3	3	3	
	Frame 2	—	—	—	3	3	3	2	2	2	2	4	4	

Rata erorilor de pagină:  $PFI\_ratio = 9/12 = 75\%$

# Alg. de page swapping /11

## ➤ Algoritmul FIFO

- Exemplu: același program cu 5 pagini virtuale, cu aceeași secvență de referire “1,2,3,4,1,2,5,1,2,3,4,5”, dar memoria fizică: 4 cadre

Pagina solicitată	1	2	3	4	1	2	5	1	2	3	4	5	
Pagina cea mai recentă	–	1	2	3	4	4	4	5	1	2	3	4	5
	–	–	1	2	3	3	3	4	5	1	2	3	4
Lista FIFO:	–	–	–	1	2	2	2	3	4	5	1	2	3
Pagina cea mai veche	–	–	–	–	1	1	1	2	3	4	5	1	2

Înlocuire de pagină (PFI):    Da   Da   Da   Da   Nu   Nu   Da   Da   Da   Da   Da   Da

**Memoria RAM:** ...    *Temă pentru acasă:* completați diagrama timp pt. memoria RAM!

Rata erorilor de pagină:     $PFI\_ratio = 10/12 = 83,33\%$

# Alg. de page swapping /12

## ➤ Algoritmul FIFO

- *Anomalia lui Belady*: deși bunul simț ne spune că șansa ca o pagină să fie înlocuită *scade* pe măsură ce *crește* numărul de cadre fizice, totuși nu se întâmplă așa în cazul alg. FIFO, după cum rezultă și din exemplul anterior
- Alt dezavantaj: (spre deosebire de LRU) algoritmul FIFO dezavantajează paginile “esențiale” (i.e. utilizate frecvent), ce vor fi în mod periodic evacuate pe disc și reîncărcate curând după evacuare

# Alg. de page swapping /13

## ➤ Algoritmul LRU (= Least Recently Used)

- Ideea: înlocuirea paginii cel mai puțin folosite în ultimul timp
- Cum? Se va folosi localitatea temporală/spațială a programului pentru a înlocui pagina cel mai puțin utilizată recent  
[principiul localității: o pagină ce a fost accesată des (rar) de către ultimele instrucțiuni, probabil va fi accesată des (rar) și în continuare]
- Pentru implementare este nevoie să se țină evidența utilizărilor paginilor, adică să ordonăm paginile după timpul celei mai recente referințe la ele – necesită deci gestiunea informației *la fiecare acces la memorie* și hardware f. costisitor
- Teoretic, gestionarea informației referitoare la utilizarea paginilor se poate face cu o coadă FIFO, cu observația că la accesarea unei pagini, ea este scoasă din coadă și mutată în vârful cozii, dar nu-i eficient practic

# Alg. de page swapping /14

## ➤ Algoritmul LRU

- Exemplu: același program cu 5 pagini virtuale, cu aceeași secvență de referire “1,2,3,4,1,2,5,1,2,3,4,5”  
memoria fizică: 3 cadre de pagină

Pagina solicitată	1	2	3	4	1	2	5	1	2	3	4	5	
Pagina cu cel mai recent acces	–	1	2	3	4	1	2	5	1	2	3	4	5
<b>Coadă LRU:</b>	–	–	1	2	3	4	1	2	5	1	2	3	4
Pagina cu cel mai vechi acces	–	–	–	1	2	3	4	1	2	5	1	2	3
Înlocuire de pagină (PFI):		Da	Da	Da	Da	Da	Da	Da	Nu	Nu	Da	Da	Da



**Memoria RAM:** ... *Temă pentru acasă:* completați diagrama timp pt. memoria RAM!

Rata erorilor de pagină:  $PFI\_ratio = 10/12 = 83,33\%$

# Alg. de page swapping /15

## ➤ Algoritmul LRU

- Exemplu: același program cu 5 pagini virtuale, cu aceeași secvență de referire “1,2,3,4,1,2,5,1,2,3,4,5” dar memoria fizică: 4 cadre de pagină

Pagina solicitată	1	2	3	4	1	2	5	1	2	3	4	5	 timp	
Pagina cu cel mai recent acces	—	1	2	3	4	1	2	5	1	2	3	4		5
Coadă LRU: 	—	—	1	2	3	4	1	2	5	1	2	3		4
	—	—	—	1	2	3	4	1	2	5	1	2		3
	Pagina cu cel mai vechi acces	—	—	—	—	1	2	3	4	4	4	5		1

Înlocuire de pagină (PFI):    Da   Da   Da   Da   Nu   Nu   Da   Nu   Nu   Da   Da   Da

**Memoria RAM:** ...    *Temă pentru acasă:* completați diagrama timp pt. memoria RAM!

Rata erorilor de pagină:     $PFI\_ratio = 8/12 = 66\%$



# Alg. de page swapping /16

## ➤ Algoritmul LRU

- Modalități de implementare a alg. LRU:
  - LRU cu contor de accese
  - LRU cu stivă
  - LRU cu matrice de referințe
- Multe SC-uri folosesc pentru paginare o *aproximare* a LRU
- Avantaje:
  - Nu prezintă *anomalia lui Belady* (se poate demonstra matematic că alg. LRU cu stivă are proprietatea de monotonie – nu funcționează mai rău dacă crește numărul de cadre de pagină ale memoriei fizice)
  - spre deosebire de alg. FIFO, alg. LRU avantajează paginile “esențiale” (utilizate frecvent), ce vor fi păstrate în memorie

# Alg. de page swapping /17

## ➤ **LRU cu contor de accese**

- Se implementează hard. CPU are un registru (de obicei pe 64 biți) numit *contor* cu rol de ceas logic – este incrementat la fiecare instrucțiune, sau acces la memorie
- În tabela de pagini există un câmp rezervat pentru a păstra valoarea acestui contor – la fiecare acces la o pagină, contorul este memorat în spațiul corespunzător acelei pagini
- Cu ajutorul acestor valori salvate, putem ști dacă o pagină a fost sau nu utilizată de la ultimul *clear*-ing al biților de referință, precum și în ce ordine (aproximativă) au fost accesate paginile
- Alegerea “victimei” pentru evacuare constă în căutarea în tabela de pagini a paginii cu cea mai mică valoare a contorului

# Alg. de page swapping /18

## ➤ **LRU cu stivă**

- Se folosește o stivă cu numerele de pagină
- Când este referită o pagină, ea este scoasă din stivă (doar dacă exista deja în stivă) și apoi este pusă în vârful stivei
- În orice moment, pagina de la baza stivei este pagina cea mai puțin utilizată recent
- Alegerea “victimei” pentru evacuare este simplă – pagina de la baza stivei, nu presupune o căutare, dar în schimb la fiecare acces este nevoie de căutarea paginii dorite în stivă pentru a o muta în vârful stivei

# Alg. de page swapping /19

## ➤ LRU cu matrice de referințe

- Se folosește o matrice binară (cu 0 și 1) de dimensiune  $n*n$ , unde  $n$  este numărul de pagini fizice ale memoriei SC-ului
- Inițial toate elementele matricii se pun pe 0
- În momentul când se referă pagina  $k$ , se pune mai întâi 1 peste tot pe linia  $k$ , iar apoi se pune 0 peste tot pe coloana  $k$
- În orice moment numărul de cifre 1 de pe o linie  $l$  ne arată ordinea de referire a paginii  $l$  – ultima pagină referită va avea  $n-1$  cifre de 1, penultima  $n-2$ , ș.a.m.d.
- Alegerea “victimei” pentru evacuare se face căutând linia din matrice cu cele mai puține cifre de 1 (cu cele mai multe zero-uri)
- Implementarea matricii de referințe se face prin hard

# Alg. de page swapping /20

## ➤ **Algoritmul LFU (=Least Frequently Used)**

- Ideea: înlocuirea paginii cel mai puțin folosite (per total, nu doar în ultimul timp)
- Alg. actualizează la fiecare acces de pagină un contor al numărului de accese, păstrat în tabela de pagini (contor ce nu este resetat periodic ca la LRU cu contor de accese)
- “Victima”: se alege pagina cu cel mai mic contor
- Dezavantaj: dacă o pagină este utilizată f. des în faza inițială a unui proces, iar apoi nu mai este utilizată deloc, ea rămâne totuși în memorie pentru că are o valoare f. mare a contorului

## ➤ **Algoritmul MFU (=Most Frequently Used)**

- dualul alg. LFU; se alege pagina cu cel mai mare contor

# Alg. de page swapping /21

## ➤ **Alte abordări:**

- NRU combinat cu FIFO
  - Ideea: se aplică întâi NRU, iar la nivelul fiecărei clase se aplică FIFO
- Metoda celei de-a doua șanse (e.g. Mach OS)
  - Ideea: se aplică FIFO cu următoarea modificare: dacă pagina cea mai veche are bitul de referință pus pe 1, atunci i se mai dă o șansă – bitul este pus pe 0 și pagina este mutată la sfârșitul listei (astfel devine cea mai recentă pagină); căutarea se reia cu noua listă, până se găsește o pagină cu bitul pus pe 0 – acea pagină este selectată pentru înlocuire
- Algoritmi de înlocuire a paginilor bazați pe prioritatea proceselor
- Algoritmi de înlocuire cu pagini de dimensiuni variabile

# Alg. de page swapping /22

## ➤ Implementări reale: **paged daemon**

- Majoritatea SO-urilor au unul sau mai multe procese de sistem responsabile cu implementarea politicii de înlocuire a paginilor pentru memoria virtuală
  - Un **demon** (*daemon*) este un proces de sistem autonom care execută periodic o anumită sarcină de administrare a SC-ului
- Demonul de paginare pregătește sistemul pentru evacuarea de pagini înainte ca să apară nevoia de evacuare
  - Demonul se “trezește” când cantitatea de memorie liberă devine mică
  - “Curăță” paginile *dirty* prin scrierea lor pe disc – *prewrite* sau *pageout*
  - Administrează liste ordonate cu candidații pentru evacuare
  - Decide cât de multă memorie să aloce pentru memoria virtuală

## ➤ Fenomenul de *trashing*

- **Trashing** = fenomenul de “sufocare” a sistemului atunci când comportamentul swapping devine excesiv  
(sistemul este ocupat predominant cu tratarea erorilor de pagină, în loc de a-și folosi timpul pentru execuția job-urilor utile)
- Paginarea la cerere funcționează datorită localității (temporale și spațiale) manifestate de programe, dar atunci când cerințele minimale de memorie nu pot fi satisfăcute, apar numeroase operații de swapping a paginilor
- Aceasta duce la o proastă utilizare a CPU și la f. multe operații I/O → SO-ul poate încerca să îmbunătățească utilizarea CPU prin *mărirea* numărului de procese



## ➤ Fenomenul de *trashing*

- **Mulțimile de lucru** (*working sets*) – sunt o modalitate utilă pentru controlul fenomenului de *trashing*
- O *mulțime de lucru* este mulțimea de pagini folosite de un proces la un moment dat (luând în calcul localitatea)
- Fenomenul de *trashing* va apare dacă suma dimensiunilor mulțimilor de lucru ale tuturor proceselor din sistem depășește limita impusă de cantitatea de memorie fizică din acel sistem
- În caz de nevoie, se pot suspenda unele procese pentru a se coborâ sub această limită

## ➤ Alte mecanisme de memorie virtuală

### – Segmentarea (cu încărcare) la cerere

= Segmentarea combinată cu tehnica de *swapping*

- **Ideea:** aducerea segmentelor de pe disc în memorie doar în momentul când sunt referite (“când e nevoie de ele”)
- Ridică probleme asemănătoare cu cele de la paginarea la cerere

### – Segmentarea paginată (cu încărcare) la cerere

= Segmentarea paginată combinată cu tehnica de *swapping*

## ➤ Planificarea schimburilor cu memoria

- **Întrebările gestiunii memoriei:** indiferent de metoda de alocare folosită, SO-ul trebuie să rezolve o serie de probleme cum ar fi:
  - **Cât?** – problema cantității de memorie alocată:
    - ➔ Alocare statică – toată memoria la început (e.g. alocarea pe partiții)
    - ➔ Alocare dinamică – doar necesarul curent (e.g. paginarea la cerere)
  - **Unde?** – problema locului liber pe care va fi plasat programul (apare la, e.g., alocarea cu partiții variabile) → Politici de plasare
  - **Când?** – problema momentului de încărcare a paginilor în cadre fizice (apare la alocarea cu paginare) → Politici de încărcare (*fetch*)
    - problema momentului de compactare (apare la, e.g., alocarea cu partiții variabile, sau la segmentarea nepaginată)
  - **Care?** – problema alegerii victimei la încărcarea paginilor (apare la, e.g., alocarea cu paginare la cerere) → Politici de înlocuire

## ➤ Planificarea schimburilor cu memoria (cont.)

### – Politici de schimb utilizate de gestiunea memoriei:

- **Politici de plasare** — utilizate de SO-uri și/sau programe (malloc/free)
  - algoritmi FFA, BFA, WFA
  - *Alocarea cu camere* (a se vedea studiul de caz: Linux), Google's algorithm *PartitionAlloc* (detalii: [aici](#)), ș.a.
- **Politici de încărcare (*fetch*)**
  - încărcarea la începutul execuției
  - încărcarea la cerere (i.e., pe parcursul execuției), cu varianta: încărcarea în avans (pe baza principiului localității)
- **Politici de înlocuire (*swapping*)**
  - alg. NRU, FIFO, LRU, ș.a.

## ➤ Planificarea schimburilor cu memoria (cont.)

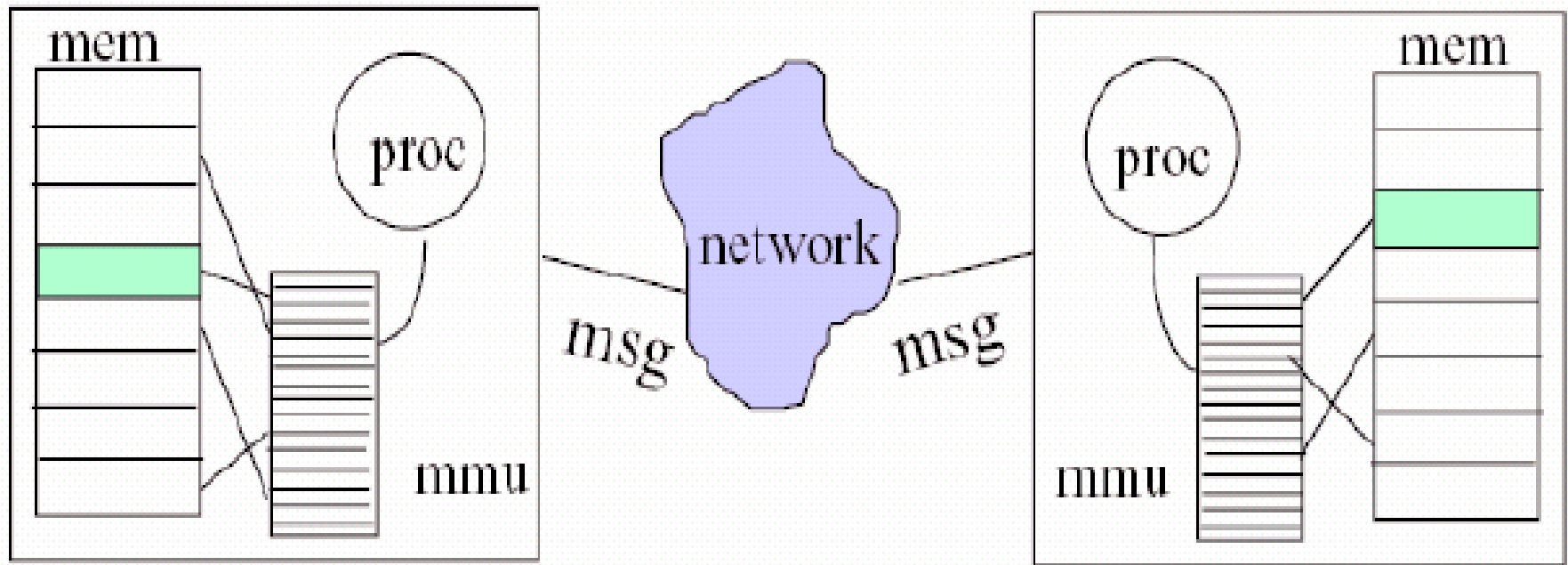
### – Tehnici de alocare utilizate:

	Alocare contiguă	Alocare necontiguă	Tip alocare
Memorie reală	Alocarea unică Alocarea cu partiții - fixe - variabile	Paginarea (pură) Segmentarea (pură) Segmentarea paginată (pură)	
Memorie virtuală	Alocarea cu swapping	Paginarea la cerere Segmentarea la cerere Segmentarea paginată la cerere	
Tip memorie			

- **SC au evoluat pe parcursul anilor**
  - Au apărut noi arhitecturi hardware / noi vederi ale “ierarhiei” de memorii
- **Memorie la distanță**
  - **NUMA** (Non-Uniform Memory Access)
    - Implementată pe AMD Opteron – 2003, Intel x86 – 2008  
([https://en.wikipedia.org/wiki/Non-uniform\\_memory\\_access](https://en.wikipedia.org/wiki/Non-uniform_memory_access))
  - **DSM** (Distributed Shared Memory)
  - **GMS** (Global Memory Systems)
    - Implementată pe Digital UNIX si FreeBSD ('98)

## Tehnici mai recente /2

- **Distributed Shared Memory (DSM)**
  - Permite utilizarea modelului de programare cu memorie partajată (spațiu de adresare partajat) într-un sistem distribuit (procesoare dotate doar cu memorie locală și o rețea de comunicație)



# Administrarea memoriei în Linux /1

- **Planificarea memoriei**

- Subsistemul de administrare a memoriei fizice din Linux are ca sarcină alocarea și eliberarea paginilor (fizice), a grupurilor de pagini și a blocurilor mici de memorie
- Linux-ul are mecanisme suplimentare pentru gestionarea memoriei virtuale, i.e. a memoriei mapate în spațiile de adrese logice ale proceselor aflate în curs de execuție în sistem



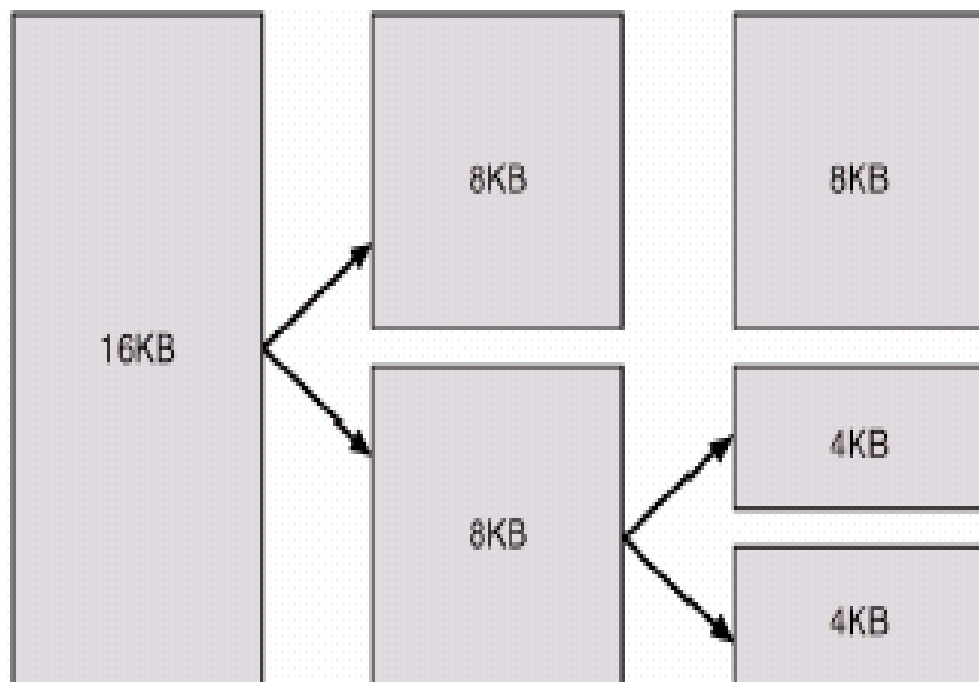
# Administrarea memoriei în Linux /2

- **Administrarea memoriei fizice**

- Alocatorul de pagini alocă și eliberează toate paginile fizice; poate alocă *la cerere* intervale contigue de pagini fizice (i.e. grupuri contigue de pagini), dar numai pentru cereri realizate în kernel-mode (pentru necesitățile nucleului).
- Practic, se alocă memorie fizică în avans, sistemul menținând astfel *free-memory pools* (i.e. intervale contigue de pagini fizice), din care poate satisface rapid cereri de alocare de diferite dimensiuni, cereri făcute de nucleu pentru a stoca structuri de date de diferite dimensiuni, sau pentru zone tampon folosite pentru I/O prin DMA
- Politici de plasare utilizate pentru cererile nucleului:
  - Alocatorul *power-of-2* – implementează alg. *alocarea cu camarazi*
  - Alocatorul *SLAB* – cu diverse variante de implementare

# Administrarea memoriei în Linux /3

- **Administrarea memoriei fizice (cont.)**
  - Alocatorul buddy-system (*alocarea cu camarazi*)

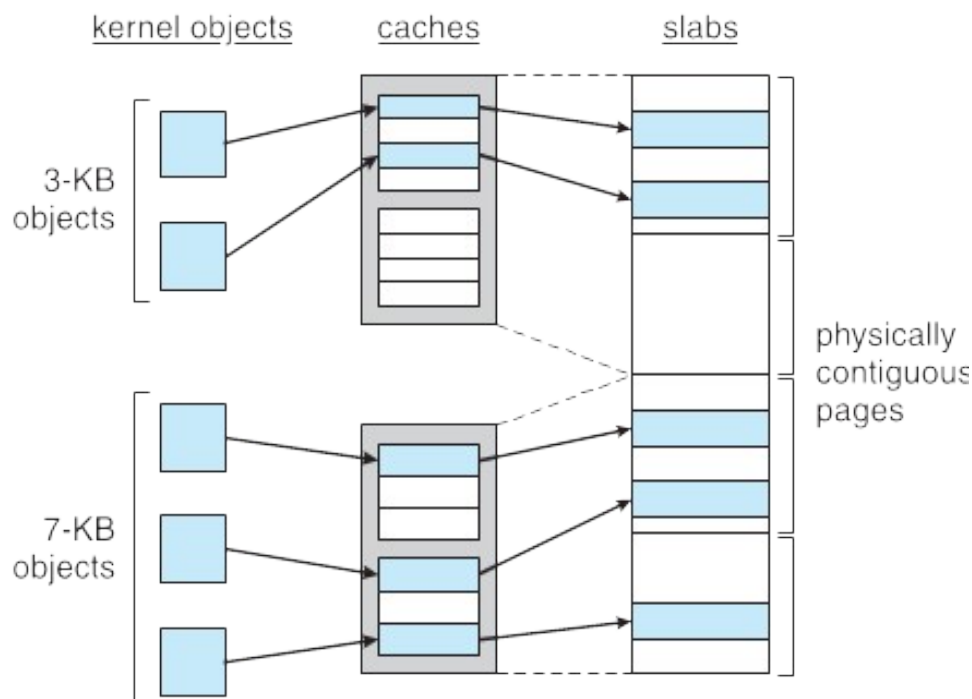


- Fiecare regiune de memorie alocabilă este asociată cu o regiune parteneră adiacentă
- Ori de câte ori două regiuni partenerere alocate sunt eliberate, ele sunt combinate într-o singură regiune liberă, dublă
- Dacă o cerere mică de memorie nu poate fi satisfăcută prin alocarea unei regiuni libere mici, o regiune liberă mai mare se împarte în 2 regiuni partenerere pentru a satisface cererea

# Administrarea memoriei în Linux /4

## • Administrarea memoriei fizice (cont.)

– Alocatorul SLAB, cu variantele SLOB și SLUB



- Un *slab* = un interval contiguu de pagini fizice
- Un *cache* = format din una sau mai multe unități *slab*, este folosit pentru a stoca toate instanțele unui tip de date utilizat de nucleu – obiecte marcate drept *free* sau *used*
- Avantaje: nu există risipă de spațiu prin fragmentare internă (precum la paginare, utilizată pentru user-mode); satisfacerea rapidă a cererilor de alocare, din *slab*-urile prealocate

*Notă:* pentru detalii suplimentare, citiți §10.8.2 din [OSC10].

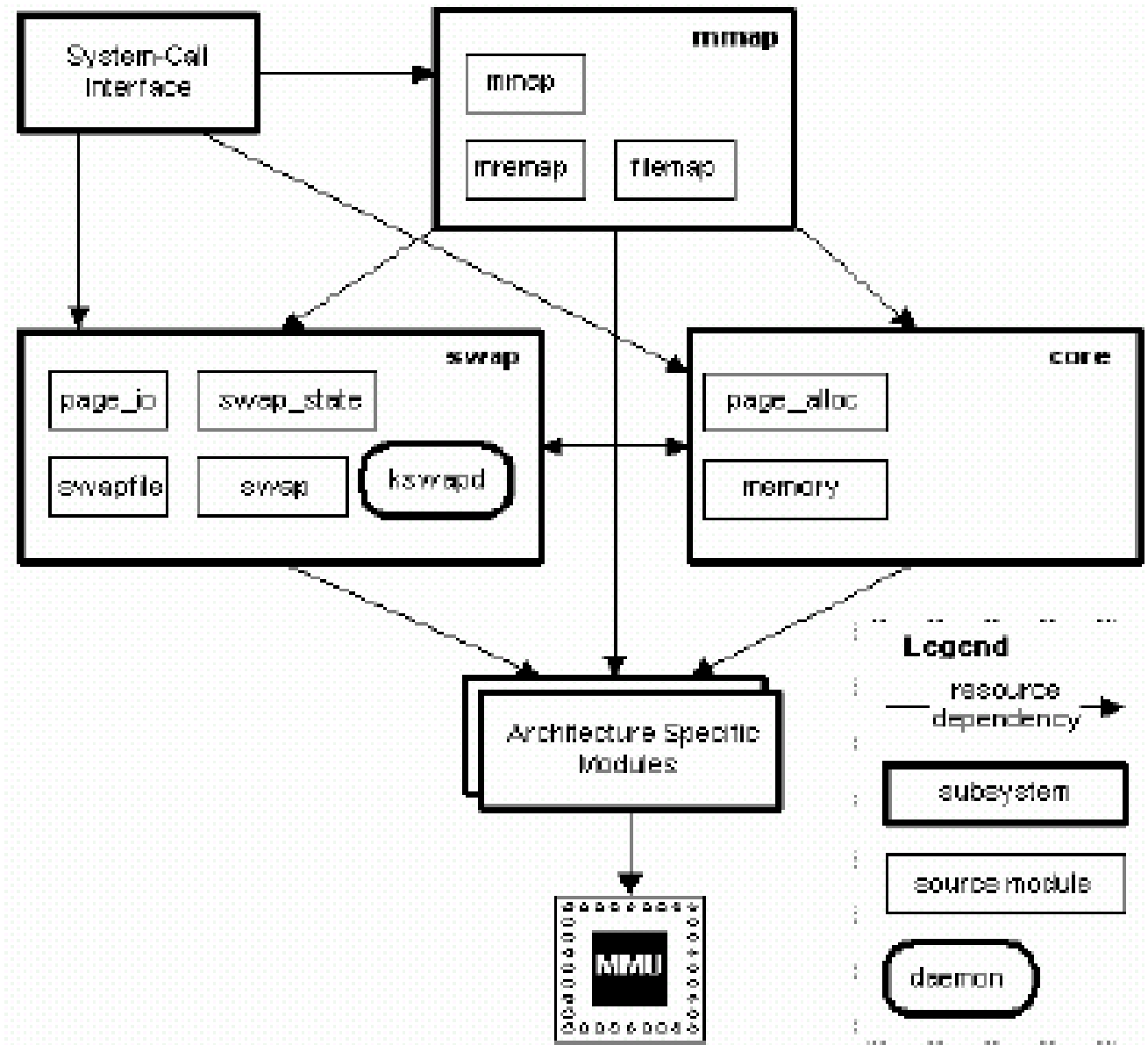
# Administrarea memoriei în Linux /5

- **Administrarea memoriei virtuale**

- Nucleul Linux rezervă o zonă de lungime constantă, dependentă de arhitectura SC, din spațiul virtual de adresare al fiecărui proces, pt. propriul său uz intern
- Această zonă rezervată de nucleu conține 2 regiuni:
  - O zonă statică ce conține o tabelă de pagini cu referințe la fiecare pagină fizică disponibilă în sistem, astfel încât să existe o translație simplă de la adrese fizice la adrese virtuale atunci când se rulează codul nucleului
  - Restul secțiunii rezervate nu este rezervată pentru un scop anume; intrările sale din tabela paginilor pot fi modificate pentru a indica spre orice alte zone din memorie

# Administrarea memoriei în Linux /6

- **Componente:**



# Administrarea memoriei în Windows /1

- **Administrarea memoriei** – Windows NT (2k/XP/.../10)
  - Administratorul de memorie lucrează cu procese, nu cu threads
  - Spațiul de adresare virtual este paginat cu încărcare la cerere, cu pagini de dimensiune fixă – 4 KB (max 64 KB pt. Itanium)
  - SO-ul poate folosi și pagini mari (cu dimensiunea de 4 MB) pentru a reduce spațiul ocupat de tabelele de paginare
  - Fiecare pagină virtuală poate fi: liberă (“neutilizată”), rezervată sau “angajată” (*committed*)
  - Paginile libere și cele rezervate au pagini *shadow* pe disc, iar referirile (accesul) la ele provoacă întotdeauna eroare de pagină
  - Se pot folosi maxim 16 fișiere de swap
  - Segmentarea nu este suportată
  - Tabelele de paginare sunt pe 2 nivele (la Windows pe 32 biți)

# Administrarea memoriei în Windows /2

- **Administrarea memoriei** (cont. – Windows pe 32 bits)
  - Adresele virtuale sunt pe 32 de biți (→ spațiu virtual de 4 GB)
  - Vârful (primii 64 KB) și baza (ultimii 64 KB) spațiului de adrese virtual al fiecărui proces, în mod normal nu sunt mapate
  - Începând de la 64 KB de la baza spațiului și până aproape de jumătate (2 GB) este zona utilizator privată – conține codul și datele programului
  - Ultima porțiune din jumătatea de jos conține date de sistem (contoare și *timer*-e) partajate read-only de toți utilizatorii
  - Jumătatea de sus (2 GB) a spațiului virtual conține SO-ul, inclusiv codul său, zona de date și diverse regiuni (*pools*), paginate și nepaginate (contigue), utilizate pentru obiectele sistemului; această porțiune nu poate fi scrisă și, în cea mai mare parte, nici citită, de către procesele utilizator

- **Bibliografie obligatorie**

capitolele despre *gestiunea memoriei* din

- Silberschatz : “*Operating System Concepts*”

(cap.10, prima parte: §10.1–6, din [OSC10])

sau

- Tanenbaum : “*Modern Operating Systems*”

(cap.3, §3.4-6, din [MOS4])

**Suplimentar:** capitolele studii de caz (despre Unix/Linux și respectiv Windows NT) din cele două cărți de mai sus



# Exerciții de seminar

## ➤ Aplicații la: Alocarea paginată la cerere

- Ex.1) **Enunț:** Considerăm un sistem cu paginare la cerere, cu strategia de swapping LRU, și un program ce trebuie rulat pe acest sistem, căruia SO-ul îi acordă 4 cadre (pagini fizice) pe toată durata execuției sale.

Spațiul virtual al programului are 5 pagini, iar secvența de acces la ele pe parcursul execuției sale este următoarea: 1,2,3,4,2,3,5,1,2,3,4,5.

Care este rata întreruperilor de pagină obținută la execuția acestui program? Justificați răspunsul, desenând diagrama timp a evoluției cozii LRU pentru acest program, precum și cea a ocupării cadrelor alocate acestui program, pe parcursul execuției sale.

- **Rezolvare:** ?

Pagina solicitată

	1	2	3	4	2	3	5	1	2	3	4	5
Pagina cu cel mai recent acces	—											
	—											
	—											
	—											

**Coadă LRU:**

Pagina cu cel mai vechi acces

Înlocuire de pagină (PFI):      ?   ?   ?   ...   ...   ...

# Exerciții de seminar

## ➤ Aplicații la: Alocarea paginată la cerere

– Ex.2) **Enunț:** Considerăm un sistem cu paginare la cerere, cu strategia de swapping LRU. Fie fragmentul de program alăturat, care inițializează elementele unei matrici  $A$  de dimensiune  $1000 \times 1000$ .

Spațiul virtual al programului are 1001 pagini: câte 1 pagină virtuală pentru fiecare linie a matricii  $A$ , plus 1 pagină pentru instrucțiuni și variabilele  $i$  și  $j$ .

La un moment dat memoria disponibilă este de 101 pagini fizice.

Câte PFI (întreruperi de pagină) vor fi necesare pentru execuția acestui program, în cele două variante a) și, respectiv, b) ?

Justificați răspunsurile.

a)

```
for(i=0; i<1000; ++i)
    for(j=0; j<1000; ++j)
        A[i][j] = 0;
```

b)

```
for(j=0; j<1000; ++j)
    for(i=0; i<1000; ++i)
        A[i][j] = 0
```

– **Rezolvare:** ?

# Sumar

- Memoria virtuală
  - Principiul localității
  - Paginarea la cerere
  - MMU
  - Algoritmi de înlocuire a paginilor
  - Fenomenul de *trashing*
  - Segmentarea la cerere
- Concluzii
- Tehnici mai recente pentru administrarea memoriei
- Administrarea memoriei în Linux și Windows

Întrebări ?