Curs 10

Memorie virtuala

= separarea memoriei logice ale user-ului de memoria fizica
= doar o parte din program trebuie sa fie incarcat in memorie pt executie
= adresele logice pot fi mai mari decat cele fizice
= permite spatiilor de adrese sa fie partajate de mai multe procese
= permite mai multe programe sa se execute concurent

Virtual adress space

=vizualizare logica a proceselor stocate in memorie
=incepe de la 0 si sunt contigue pana la finalul spatiului
=memoria fizica e organizata in page frames
=MMU mapeaza adresele logice cu cele fizice
! Memoria virtuala => implementata prin paginare sau segmentare la cerere !
Paginarea la cerere => poate pune tot procesul in memorie la load time
=> poate aduce doar o pagina in memorie daca e necesar
=> similara cu paginarea cu swapping
=> daca pagina este ceruta se va face o referinta catre aceasta (daca referinta e invalida => abort, daca pagina nu e in memorie => se aduce in memorie)
Lazy swapper = nu aduce o pagina in memorie decat in cazul in care e nevoie

Valid-Invalid Bit

Fiecare intrare dintr-un tabel de pagini are un bit valid-invalid. (valid => pagina este in memorie = memory resident, invalid => pagina nu e in memorie)
In timpul translatiei MMU de adrese, daca un valid-invalid bit din intrarea intr-o tabela de pagina este invalida => page fault.
Tratarea page-fault:

1. If there is a reference to a page, first reference to that page will trap to operating system
   - Page fault
2. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
3. Find free frame
4. Swap page into frame via scheduled disk operation
5. Reset tables to indicate page now in memory
   Set validation bit = **v**
6. Restart the instruction that caused the page fault

Free-Frame List

Page-fault => SO trebuie sa aduca pagina ceruta din spatiul de stocare secundar in memoria principala

=>cele mai multe sisteme de operare au un free-frame list (mai multe frame-uri care sunt libere si indeplinesc cerinte pentru un astfel de request)

Deman Paging – Stages

1. Trap pentru SO
2. Se salveaza registrii userului si starea procesului
3. Se verifica daca intreruperea a fost un page fault
4. Se verifica ddaca pagina referita era legala si se determina locatia paginii
5. Se face o citire de pe disk pentru un frame liber
6. Se asteapta in coada pana cand e rezolvat acest request
7. Seek/latency time pt device
8. Se incepe transferul paginii catre un frame liber
9. Cat se asteapta CPU se aloca altui user
10. Se primeste o interrupere de pe disk dupa ce se termina procesul
11. Se salveaza registrii si starea procesului pt userul care a rulat last time
12. Se determina daca intreruperea a fost de pe disk
13. Se corecteaza tabela de pagini
14. Se asteapta ca CPU sa fie alocat pt procesul initial
15. Se restaureaza registrii, starea procesului, noua tabela de pagini, se reia rularea

Exemplu:

- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed
  - Read the page – lots of time
  - Restart the process – again just a small amount of time
- Page Fault Rate $0 \le p \le 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

  EAT = (1 – p) x memory access
  + p (page fault overhead
  + swap page out
  + swap page in)

Copy-on-Write => acorda acces si parintelui si copilului (la nivel de procese) sa partajeze aceleasi pagini in memorie. (daca unul dintre procese modifica o pagina partajata, doar atunci aceasta e copiata)
=>se permite astfel o creare mai eficienta de procese

OBSERVATIE:
Paginile libere se gasesc in general in pool uri de zero-fill-on-demand pages.

IMPORTANT

`vfork()` variation on `fork()` system call has parent suspend and child using copy-on-write address space of parent

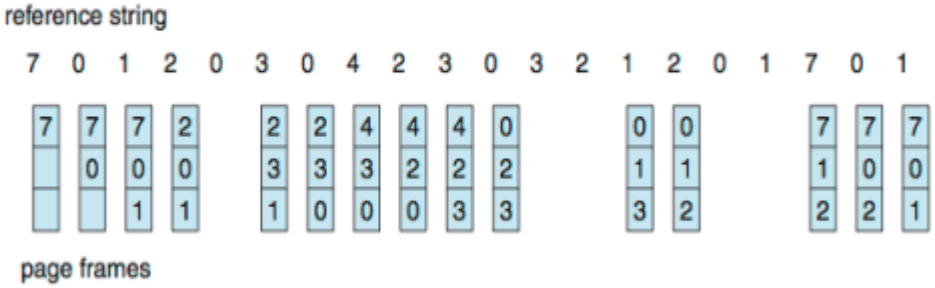
- Designed to have child call `exec()`
- Very efficient

Inlocuire de pagina
Previne solicitarea memoriei si utilizeaza modificarea unui bit (dirty bit) pentru a reduce numarul mare de transferuri de pagini (doar cele modificate sunt scrise pe disk)

Algoritmi folositi pentru page replacing

FIFO

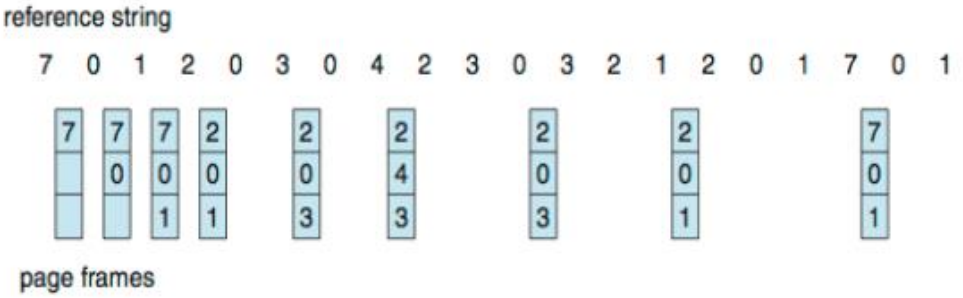
- Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1
- 3 frames (3 pages can be in memory at a time per process)



15 page faults

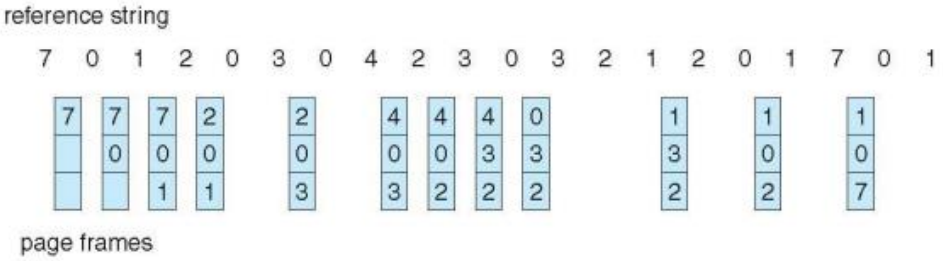
OPT Algorithm (optimal algorithm) (nu are anomalia lui Belady)

- Replace page that will not be used for longest period of time
  - 9 is optimal for the example
- How do you know this?
  - Can't read the future
- Used for measuring how well your algorithm performs



Least Recently Used (LRU) (nu are anomalia lui Belady)

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Capitalizes on the locality of reference principle
- Associate time of last use with each page



- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used

Reference Bit = R = fiecare pagina are un bit de referinta (0 daca pagina nu are referinta, 1 daca are referinta)

Counting algortihms

Tin un contor cu numarul de referinte care s-au facut pentru fiecare pagina (adauga bitul de referinta R la fiecare ceas).

Exemple:

1. Least Frequently used (LFU) SAU Not Frequently Used (NFU) (inlocuiesc paginile cu cel mai mic count)
2. Most Frequently Used (MFU) (se bazeaza pe faptul ca paginile cu cel mai mic count urmeaza sa fie utilizate)

Aging

->NFU=>reflecta mai bine localitatea temporala

->LRU=>aproximeaza mai bine

-> la fiecare ceas, count-ul se shifteaza la dreapta cu 1 bit si se aduna cu R cel mai semnificativ bit (bitul de referinta)

->page fault => se elimina frame-ul cu contorul cel mai mic

Page Buffering Algorithms

- ➔ Se pastreaza un grup de frame-uri libere
- ➔ Pastram frame-ul disponibil pentru cand avem nevoie, nu cel gasit in urma unui fault
- ➔ Punem pagina in frame si adaugam inapi in grupul de frame uri, iar cand ne este convenabil, eliberam iar
- ➔ Pastram o lista cu paginile modificate (cand e posibil scriem paginile si le setam ca non-dirty)
- ➔ Pastram continutul frame-urilor libere pentru a gestiona mai usor faulturile si pentru a verifica daca exista referinta acolo

Alocarea de frameuri

Fiecare proces are nevoie de un numar minim de frame-uri (numarul maxim poate fi numarul total de frame-uri din sistem).

Alocare fixa

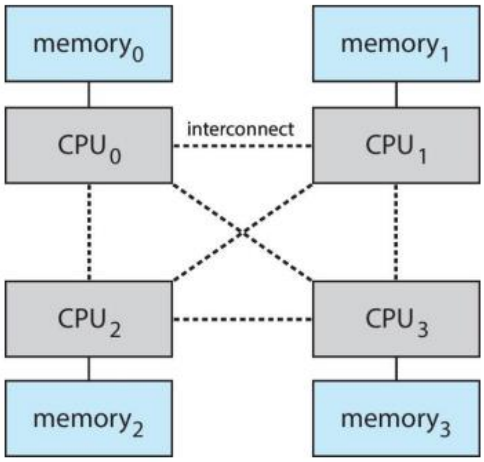
- ➔ Alocare egala pentru fiecare proces (100 frameuri, 5 procese => 20 de frame uri per proces)
- ➔ Alocare proportionala (in functie de marimea fiecarui proces). Formula pentru alocare **(a_i pentru procesul p_i = s_i / S * m) unde s_i este dim. acelui proces, S suma dimensiunilor proceselor, m nr total de frameuri**

Global replacement = procesul selecteaza un locuitor pentru frame din multimea de frame-uri (deci un proces poate lua un frame de la alt proces) (timp mai mare de executie si un debit mai mare)

Local replacement = fiecare proces selecteaza un frame din multimea sa de frame-uri disponibile (performanta mai mare, posibil sa se incerce sa se utilizeze memorie prea putina)

Non-Uniform Memory-Access

Cele mai multe sisteme sunt de tip NUMA = viteza cu care e accesata memoria variaza.



Se observa ca pentru o performanta optima presupune ca un CPU sa acceseze memoria cea mai apropiata.

Thrashing

Pentru un proces cu pagini putine => multe page-faulturi (fie la gasirea unei pagini care e inexistenta, fie la inlocuirea unui frame existent, etc) ce duce la o utilizare slaba a CPU

Thrashing = un proces este ocupat sa schimbe pagini in and out

Locality model = procesul migreaza dintr-un loc in altul in memorie, iar locatiile se pot suprapune. Thrashingul apare in momentul in care marimea locatiilor e mai mare decat dimensiunea intregii memorii.

Working-Set Model

DELTA = Woring-set window = numar fix de referinte de pagini

WSS_i = working_set pt procesul i = numarul de pagini catre care au fost referinte in cel mai recent woking-set window

Daca DELTA = prea mic => nu se va cuprinde intreaga locatie
Daca DELTA = prea mare => include prea multe locatii
Daca DELTA = infinit => include tot programul

Fie D = suma tuturor frame-urilor cerute (WSS_i) > m(nr tot frames) => trashing

Page-Fault Frequency

Se stabileste un page fault frequency acceptabil si se foloseste politica local replacement. (prea mic => procesul pierde frame-ul, prea mare => procesul primeste un frame)

Alocarea memoriei kernel

Kernel Memory Allocator = KMA = obtine pagini de la alocatorul de pagini si le foloseste pt cererile de memorie neswapabila ale kernelului (unele trebuie sa fie contigue)

= trebuie sa fie rapid

= KMA e apelat din handlere de intrerupe care au constrangeri critice de timp

= KMA lent = degradarea performantei intregului sistem

= trebuie sa aiba interfata de programare simpla

= memoria se aloca intr-o parte a kernelului si se dealoca in alta (deci free() trebuie proiectat ai sa nu stie dimensiunea alocarii)

factor de utilizare=memorie_totala_ceruta/memoria_totala_necesara_cereri

KMA trebuie sa obtina noi pagini cand free list e goala si sa dea pagini inapoi cand nu sunt folosite (si sa compacteze blocuri nealocate adiacente pt a reusi sa satisfaca cereri mai mari)

Buddy System

Aloca memorie din segmentele de dimensiune fizica (alocarea memoriei se face folosind power-of-2 allocator).

Exemplu

- Split into A_L and A_R of 128KB each
  - One further divided into B_L and B_R of 64KB
    - One further into C_L and C_R of 32KB each – one used to satisfy request

Avantaj => zone nefolosite se unesc pentru a forma o zona mai mare

Dezavantaj => fragmentare

Implementare

➔ Managerul de memorie mentine free lists pentru toate puterile lui 2 pana la dimensiunea memoriei

Alocarea

Rotunjita la urmatoarea putere a lui 2
Daca nu exista in free lists segmentul corespunzator se merge la puteri mai mari. Blocurile mari alocate se sparg succesiv in 2 buddies

Dealocarea

Rotunjita la urmatoarea putere a lui 2
Daca segmentul dealocat are un buddy adiacent, se compacteaza intr-un segment mai mare (iar daca sgmentul rezultat are si el un buddy, se compacteaza in continuare)

regula de detectare buddies:

$$B_1, B_2 \text{ buddies} \Leftrightarrow size(B_1) = size(B_2) = 2^k \;\&\&\; 2^{k+1} | \; adr(B_1 + B_2)$$

Avantaje => dealocare rapida
Dezavantaje => fragmentare interna si externa

Slab allocator

Slab = una sau mai multe pagini fizice contigue
Cache = 1+ Slabs
Exista un singur cache pentru fiecare kernel.
Cache creat => obiectele sunt marcate cu free
Structuri stocate => obiectele sunt marcate cu used

Prepaging

- ⇨ Folosit pentru a reduce numarul de page faults la inceperea unui proces
- ⇨ Se face pt paginile unui proces dinainte sa fie accesate
- ⇨ Daca prepaged pages nu sunt folosite => ineficienta

TLB Reach = memoria accesibila din TLB ((TLB Size) * (Page Size))
= working setul fiecarui proces e stocat in TLB
= cresterea page size => fragmentare
= mai multe dimenisiuni pt pagini => fara fragmentare si eok

Priority Allocation

Ca alocarea proportionala doar ca se folosesc prioritati.
Un proces P_i generaza un page fault

- ➔ Selectezi inlocuitorul din frame-urile dispobibile
- ➔ Selectezi inlocuitorul un frame dintr-un proces cu prioritate mai mica

Memory compression

- ➔ Se compreseaza mai multe frame-uri intr-unul singur, si se reduce memoria utilizata de sistem, deci o alternativa pentru paginare este memory compression