

CHAPTER 4

Thread uri

Proces = un singur punct de executie in aplicatie (o singura instanta in rularea aplicatiei)

! Unele aplicatii pot profita de existenta mai multor puncte de executie simultanasa in cadrul aplicatiei

- Creearea de puncte multiple de executie in program se poate face cu procese care partajeaza data
 - Procesele au date private
 - Comunica intre ele prin IPC (memoria partaja)
 - Daca exista capacitate de multiprocesare, procesele pot rula simultan pe mai multe procesoare
- Puncte nevrangice
 - Creearea proceselor
 - Context switch ul

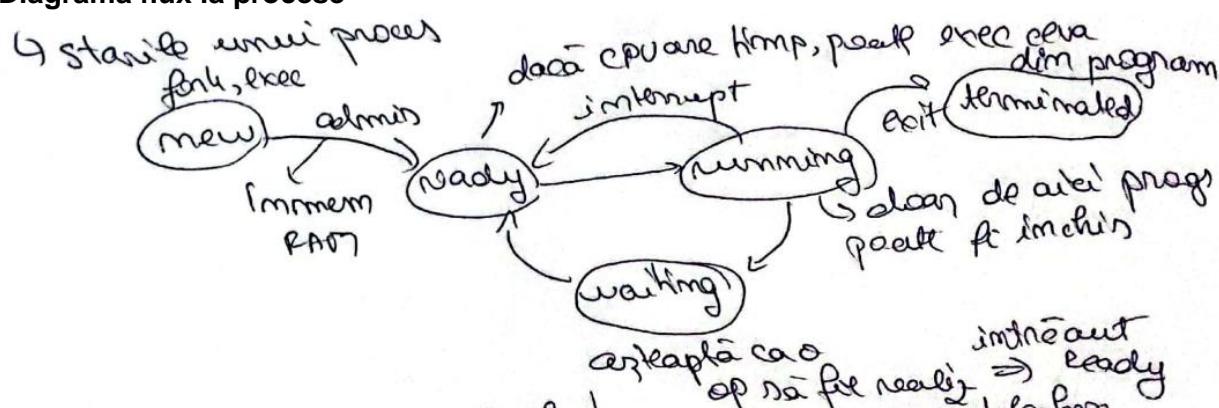
1. Creearea proceselor

- a. Contextul unui proces este stocat in PCB Kernel
- b. Un PCB este format din:
 - i. Process state(new, ready, waiting, ...)
 - ii. Process number (pid-ul)
 - iii. Program counter
 - iv. Registrii
 - v. Memory limits
 - vi. Lista de fisiere deschise
- c. Din cauza marimii PCB-ului, este un proces de cost semnificativ atunci cand se creeaza unul nou
- d. Daca acest cost e prea mare, aplicatiile pot sa nu foloseasca eficient procesoarele (Este posibil sa aparata fenomenul de "slowdown")

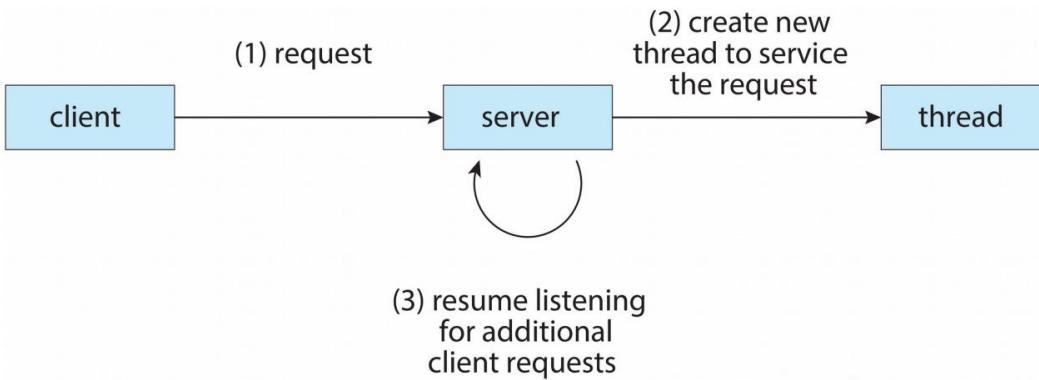
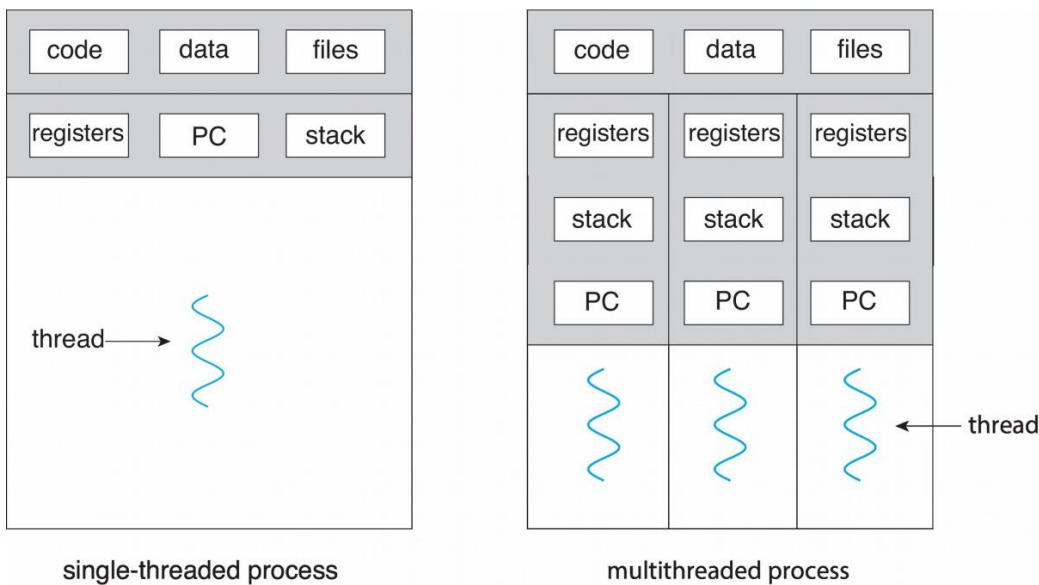
2. Context-switch

- a. Salvarea contextului unui proces si incarcarea contextului unui nou proces poate implica un cost semnificativ avand in vedere bogatia de informatii din PCB
- b. Daca timpul de context switch e mare, viteza de transfer a datelor e serios afectata

Diagrama flux la procese



Arhitecturi multithread



Beneficii:

- Responsiveness - daca un proces se blocheaza, celelalte continua
- Resource Sharing - impart din resurse
- Economy - mai ieftin decat creearea de procese
- Scalability - procesul se poate folosi de arhitecturile multicore

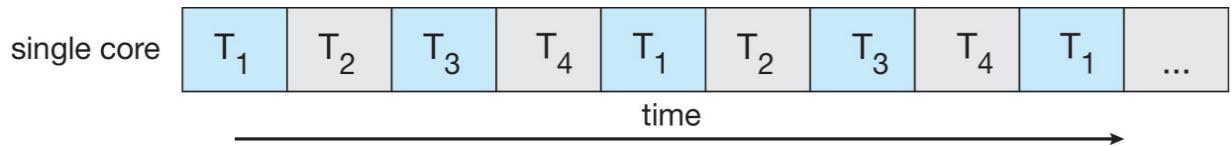
Multicore programming

- Apar urmatoarele probleme:
 - Dividing activities
 - Data dependency
 - Balance
 - Testing and debugging
 - Data splitting

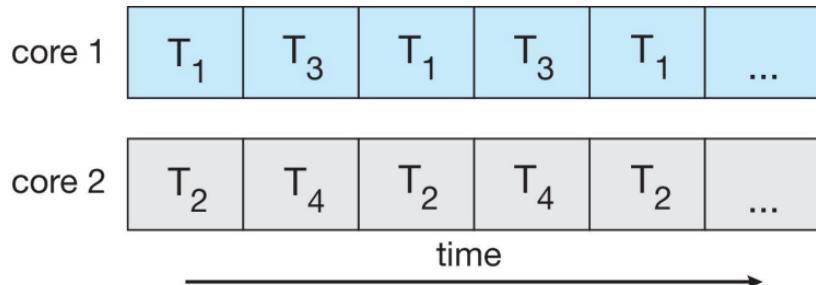
Paralelism = Un sistem poate sa execute mai mult de un task in acelasi timp

Concurrenta = suporta sa faca progres in acelasi timp pe mai mult procese (este facut prin scheduler)

Concurrenta pe un singlecore

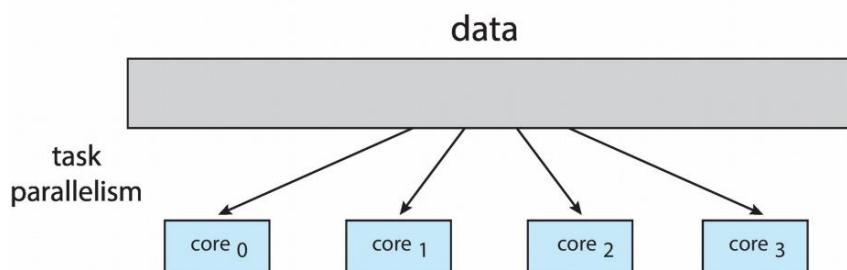
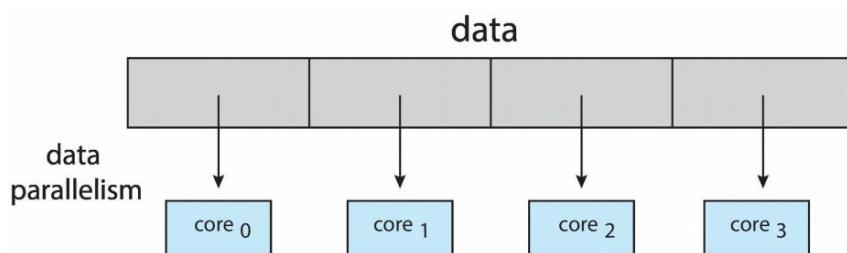


Paralelismul pe multicore



Tipuri de paralelism

- Data parallelism
 - Distribuie subseturi de date identice de-a lungul tuturor proceselor
- Task parallelism
 - Distribuie task urile



Legea lui Amdahl

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

Unde S = serial portion si N = processing cores

- Identifica imbunatatirile la performanta daca am adauga mai multe core ur la o aplicatie cu portiuni serializate si paralele
- Exemplu: O aplicatie e 75% paralela si 25% serializata si o trecem de la un core la 2: Speedup =< $1/(1/4 + (1 - 1/4) * (1/2)) = 1/(1/4 + 3/18) = 8/5 = 1.6$ times, deci se imbunatatestte cu 60%
- Cu cat N se apropiie de infinit, speedup ul incepe sa tinda la $1/S$
- Partea serializata a unei aplicatii are efecte disproportionala asupra performantei fata de adaugarea de core ur

Legea lui Gustafson

$$Speedup = 1 / (s + p / N)$$

- Presupunere implicita: p e independent de N (adica dimensiunea problemei e fixa)
- Presupunere nerealista in practica, unde dimensiunea problemei scaleaza cu numarul de procesoare => o noua abordare, in care se presupune ca timpul de rulare e constant, nu dimensiunea problemei
- Acum, se condiera ca p variaza liniar cu N

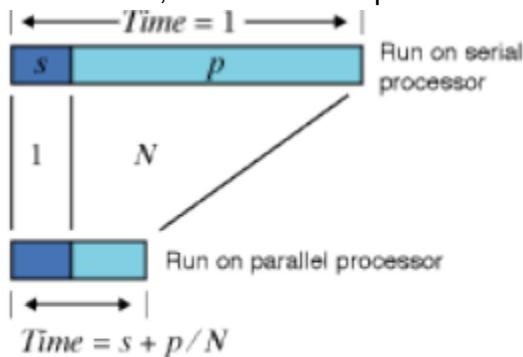


FIGURE 2a. Fixed-Size Model: $Speedup = 1 / (s + p / N)$

- Obtinem astfel o noua definitie pentru speedup

$$Scaled\ speedup = s + pN = s + (1 - s)N = N + (1 - N)s$$

- Daca s e constant , dependenta de N e liniara cu panta $1-s$

- Schimbare de paradigma: Scaled speedup-ul este de fapt "slowdown-ul" teoretic pe care il inregistreaza un program paralel atunci cand ar rula ipotetic pe o masina cu un singur procesor
- Concluzii:
 - Accentul folositii unui numar crescut de procesoare cade pe necesitatea de a rula programe mai mari in acelasi timp daca se poate, nu pe scaderea timpului de executie pe o problema de dimensiune fixa
- OBS: Nu orice problema poate fi marita arbitrar, ca atare exista si limitari ale legii lui Gustafson

Fire de executie

- Modalitate de a reduce costurile crearii punctelor de executie in aplicatie si a schimbarii contextelor de executie intre ele
- Idee: Puncte de executie multiple din aplicatie partajeaza o parte din contextul programului, DAR fiecare punct are o copie individuala a unui subset al contextului a aplicatiei si structuri de date (ex: stiva)
- Apare o noua abstractie de nivel inalt, firul de executie, care este un punct de executie cu context redus in cadrul programului (i se mai spune si "proces usor")
- Un TCB (Thread Control Block) are:
 - Thread Identifierul (id-ul thread-ului)
 - Stack pointer
 - Program counter
 - State (running, waiting ...)
 - Valorile registrilor
 - Pointer catre PCB-ul procesului pe care traieste

Caracteristici

- Ruleaza sequential, au program counter si stiva proprie
- Multiplexeaza accesul la CPU ca si procesele
- Pot crea alte thread-uri
- Pot executa apeluri sistem
- Analogie:
 - Thread-ul fata de proces este ceea ce procesul este fata de procesor
 - Procesul actioneaza ca un procesor virtual pe care ruleaza threadul

Diferente fata de procese

- Threadurile aceliasi proces partajeaza spatiul de adresa al procesorului (de ex, partajeaza variabilele globale) => un thread poate distruge usor un alt thread, deoarece nu exista protectia MMU ca in cazul proceselor diferite
- Lipsa protectiei intre thread-uri:
 - E inevitabile
 - Nu e necesara
 - Nici nu e de dorit

- Alte resurse partajate: acelasi set de fisiere deschise, timere, semnale , etc.

Alte caracteristici

- Stari (la fel ca si la procese):
 - In rulare / running
 - Gata de rulare / ready
 - Blocat
 - Terminat / terminated
- Modele de utilizare
 - Cooperativ, lucru in echipa (ex. Filtrarea de imagini)
 - Master -> Slave/Worker (ex. Server)
 - Pipeline (ex. Producator - consumator)
- Avantaj principal: datele partajate sunt datele globale din proces (nu e nevoie de setarea unor mecanisme IPC de tipul memoriei partajate)
 - Buffer global pentru producator-consumator
 - Argument puternic pentru sistemele multiprocesor

Design ul pachetelor de thread-uri

Pachet de thread uri = colectie de primitive (apeluri de biblioteca) pentru lucrul cu thread-uri

1. Gestiunea thread urilor
 - a. Creeare thread
 - i. Primeste ca argument functia, o stiva si o prioritate de planificare
 - ii. Intoarce un TID (Thread ID)
 - b. Terminare thread:
 - i. Explicit prin apel "exit" sau semnal de tip kill de la alt thread / proces
 - c. Primitive pentru mecanisme, necesara datorita existentei datelor partajate (uzual mutex - uri, dar in mod notabil si variabile de conditie folosite in conexiune cu un mutex)
 - d. Ex: acquire/release source folosind mutex + condition variable
2. Planificarea
 - a. Aceiasi algoritmi ca si la procese, vom discuta la planificarea proceselor/thread urilor
3. Probleme de reentranta
 - a. Scenariu: doua thread uri T1 si T2 care executa concurrent apele la sistem. T1 reuseaza. T2 esueaza.
 - b. Daca T1 nu evaluateaza errno inainte ca T2 sa execute, T1 va crede eronat ca apelul sau la sistem a esuat
 - c. Problema principală: errno e variabila globala
 - d. Solutii posibile:
 - i. Protejarea errno cu mutex uri
 - ii. Creearea unei copii private a lui errno => Salvam in TLS (Thread - Local Storage)

Implementarea thread urilor kernel

- Pachete de threaduri se pot implementa în kernel sau în spațiul utilizator
- Threadurile kernel separă camurile din PCB care ajută la crearea unui punct de execuție și le stochează într-un TCB
- Astfel, un proces cu un singur punct de execuție este reprezentat în Kernel de un PCB și un TCB
- Operația de creare a unui thread este un apel sistem
 - Aloca un TCB
 - Aloca stive kernel și user
 - Le legă la PCB-ul procesului în care s-a făcut apelul

Costuri threaduri kernel

Costul creării unui thread kernel <<< Costul creării unui proces

- Se aloca și se initializează doar TCB-ul și stivele
- Restul contextului este deja în PCB
- Context switch-ul între threadurile aceluiași proces durează mult mai puțin decât schimbarea contextului între două procese (la primul se schimbă doar TCB-ul, iar la al doilea și PCB-ul)

Planificarea kernel thread urilor pentru execuție

- Threadurile rulează asincron unele fata de celelalte și pot pierde procesorul la fel ca și procesele => accesul la date partajate trebuie sincronizat când se dorește IPC
- Planificatorul kernel alege urmatorul thread care trebuie să ruleze => dacă aplicația are propria politică de planificare trebuie să o comunice într-un fel sau altu kernelului
- În cazul multiprocesoarelor, planificatorul poate asigna mai multe CPU-uri unui singur proces pentru ca threadurile să ruleze în paralel ("gang scheduling")
- Dacă un thread se blochează în kernel și cuanta de timp alocată procesului nu a expirat, planificatorul căuta în lista de TCB-uri un thread gata de rulare din același proces și îi acordă procesorul

Protectia kernel thread urilor

- Observația generală despre threaduri este valabilă și pentru threadurile kernel
- Un thread poate corupă stiva altui thread => se distrug date private ale altui thread
- Soluție: Implementarea stivelor în spații de adresa diferite, dar asta mărește costul context switching-ului. Mai exact, ar fi nevoie de salvarea și reincarcarea contextelor de execuție referitoare la gestiunea memoriei, pe lângă TCB

Dezavantajele kernel thread urilor

- Deși mai puțin costisitoare ca procesele, au anumite aspecte care le fac nepotrivite pentru utilizatori
- 1. Thread_create este apel sistem => costisitor pentru procese care creează multe threaduri
- 2. Context switching-ul de threaduri necesită intrarea și ieșirea în/din kernel mode => overhead aditional context switching-ului obisnuit
- 3. Implementarea în kernel este inflexibilă

- a. Impune un model de threaduri care nu e potrivit pentru oricare aplicatie
- b. Codul planificatorului (scheduler) nu e accesibil (fiind in kernel) => greu de adaptat pentru cerintele specifice ale unei aplicatii anumite (politica de planificare nu se poate schimba usor)

Trap = apel sistem

User level threads

- Daca planificatorul si TCB urile sunt implementate in spatiul utilizator costurile scad pentru ca nu mai e nevoie de traversarea granitei kernel/user
- Comparatie calitativa, intr-un ex. In care un apel de procedura costa 7 usec, iar un apel sistem (trap) 19 usec

Operatie	Thread user	Thread kernel	Proces Unix
fork	34 usec	948 usec	11300 usec
IPC synch	37 usec	441 usec	1840 usec

Caracteristici

- Nu apeleaza serviciile kernel pentru creare si context switch => aceste operatii sunt foarte rapide
- Aplicatiile pot furniza propriul planificator customizat conform cerintelor specifice
- Nu necesita nici un fel de suport explicit din partea kernelului; procesul e privit ca un procesor virtual pentru threaduri
- Fiind mult mai rapide, de regula threadurile user se implementeaza deasupra celor kernel => planificatorul de threaduri user trateaza threadurile kernel ca pe procese virtuale si multiplezeaza mai multe threaduri user pe unul sau mai multe thread-uri kernel
- Multithreading models:
 - Many-to-One
 - One-to-One
 - Many-to-Many

Many-to-One

- Many user level threads mapati la One kernel thread
- Blocarea unui thread poate duce la blocarea tuturor
- Mai multe threaduri nu pot rula in paralel pentru ca doar unu poate fi in kernel la un moment dat
- Foarte putine folosesc modelul asta: Solaris Green Threads, GNU Portable Threads

One-to-One

- One user level thread mapat la One kernel thread
- Creand un user-level thread inseamna si creearea unui kernel
- Mai multa concurrenta decat Many-to-One
- Numar de threaduri per proces mai redus

- Ex: Windows si Linux

Many-to-Many

- Many user level threads mapati la Many kernel threads
- Permite sistemelor de operare sa creeze un numar suficient de kernel threads
- Ex: Windows cu ThreadFiber package. De altfel, nu o arhitectura foarte comună

Two-level

- O combinatie de Many-to-Many cu One-to-One, adica e M-to-M doar ca permite unui thread user sa fie mapat de un singur la unul kernel

Probleme comune thread urilor kernel si user

1. Reentrantă
 - a. Multe apeluri de biblioteca sunt nereentrantă, iar apelurile reentrantă sunt desemnate în manual ca fiind "thread safe"
 - b. Ex: errno, malloc, apeluri stdio sunt afectate
2. Tratarea semnalelor
 - a. Ex: Un thread tratează un semnal în timp ce altul vrea ca semnalul respectiv să termine aplicația
 - b. Se poate întâmpla dacă se folosesc apărutii biblioteca și runtime user împreună
 - c. Problema deriva din faptul că semnalele sunt definite per proces și nu per thread

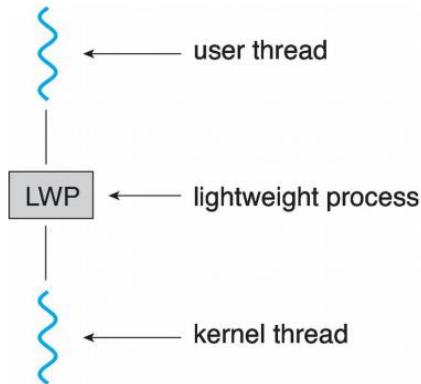
Dezavantajele thread urilor utilizator

- Determinate de faptul că existența lor este necunoscută kernelului
- 1. Dacă un thread user executa un apel sistem care se blochează în kernel, planificatorul kernel (agnostic cu privire la threadurile user) consideră că întreg procesul s-a blocat și aloca CPU altui proces, chiar dacă procesul curent mai are și alte threaduri gata de rulare și nu s-a consumat toată cantitatea de timp
- 2. Dacă un thread user comite page fault (=acces la pagina de memorie inexistentă/nealocată) => același efect ca și sus, planificatorul kernel alege alt procesor în vreme ce pagina de memorie este adusă de pe disc => adică rulează cu mai puține CPU decât este necesar
- 3. Neexistând cunoștință despre existența threadurilor user, kernelul poate lua procesorul unui thread user care detine un spinlock pe care nu l-a eliberat => scădere dramatică a performanței pentru aplicațiile care folosesc threaduri user paralele. Efectul este mai dramatic dacă schedulerul kernel alege să ruleze alte threaduri care vor să obțină același spinlock
- Problema de fond: Lipsa de coordonare între schedulerul kernel și implementarea pachetului de threaduri user

Scheduler activations

- În M-to-M și 2-level au nevoie de comunicare pentru a menține numărul optim de threaduri
- De obicei, se folosește o structură de date intermedie între user și kernel threads și se numește lightweight process (LWP)

- Scheduler activations asigura upcall uri, mecanisme de comunicare de la kernel la upcall handler in libraria thread urilor
- Comunicarea permite aplicatiei sa mentina un numar corect de kernel threads



Scheduler activations - notiuni curs

- Metoda de coordonare kernel <-> pachet thread uri utilizator
- Model: aplicatia ruleaza pe un multiprocesor virtual
 - Existaapeluri sistem pentru a suplimenta/diminua numarul de procesoare virtuale alocate de kernel
 - Kernelul decide daca onoreaza cererea sau nu
- Scheduler activation e aprozimarea unui kernel thread
 - Are stive kernel si user
 - Ofere context de executie pentru un thread user
 - In plus, ofera conceptul de upcall pentru evenimente de mai mult tipuri
 - Fiecare upcall creeaza o noua activare (optimizare: folosirea vechilor activari)

Tipuri de evenimente upcall

1. Adauga un procesor
 - a. Consecinta este executia unui thread user
2. Procesor preemptat
 - a. Adauga threadul user care se executa in activarea care a pierdut CPU in coada de threaduri gata de rulare
3. Activare blocata
 - a. Activare s-a blocat si nu mai utilizeaza procesorul
4. Activare deblocata
 - a. Pun in lista ready threadul care se executa in contextul activarii blocate

Cum functioneaza la ... ?

1. Pornirea procesorului
 - a. Kernelul aloca o activare + notifica aplicatia (upcall "adauga procesor") dupa ce i-a asigurat un CPU
 - b. Sistemul de gestiune al thread urilor user primeste notificarea si foloseste noua activare drept context de executie pentru initializarea sa si a thread urilor main
2. Crearea de suplimentare a concurentei ("adauga procesor" sau "activare blocata")

- a. Kernelul salveaza starea threadului user in activarea curenta
- b. Aloca o noua activare si cheama aplicatia in contextul noii activari

END OF CHAPTER 4

CHAPTER 6

DEFINITII IPC

IPC = Inter Process Communication

procese independente – procese care nu partajeaza resurse

procese dependente – partajeaza resurse pt a-si indeplini obiectivele

sectiune/regiune critica – portiune de cod care accesea o resursa partajata

race condition – situatie in care executia intreatesuta (interleaved) a mai multor procese care accesea o resursa partajata induce rezultate nedeterministe

excludere mutuala – situatie in care cel mult un proces are acces la un moment dat la o resursa partajata

deadlock – situatie in care nici un proces nu poate continua pt. ca resursele de care are nevoie sunt detinute de un alt proces

pp. implicit accesul mutual exclusiv la resursele partajate si o forma sau alta de asteptare circulara (circlu intr-un graf de alocare a resurselor)

sincronizare – cerinta ca un proces sa fi atins o anumita etapa in calculul sau inainte ca alt proces sa poata continua

starvation – resursele necesare unui proces nu ii sunt niciodata puse la dispozitie

ex: un proces de prioritate mica PI este impiedicat sa accesze o resursa care e constant obtinuta de un proces de prioritate mare Ph . spunem ca PI “moare de foame”

Cerinte necesare unei solutii corecte de partajare a resurselor de catre procese concurente nu e permisa nici o presupunere vis-à-vis de viteza relativa de executie a proceselor concurente

excludere mutuala – cel mult un proces poate fi in sectiune critica la orice moment dat

asteptare limitata – daca un proces solicita accesul in sectiune critica trebuie sa i se garanteze ca-l va obtine candva

progres – un proces care ruleaza in afara sectiunii critice nu trebuie sa blocheze alt proces care vrea sa intre in sectiunea critica

General structure of process P_i

```
while (true) {  
    entry section  
    critical section  
    exit section  
    remainder section  
}
```

SOLUTII CRITICAL SECTION

Alternanta stricta

```
int turn = 0;
```

```
P0 () {  
    while(1) {  
        while(turn != 0) ;  
        sectiune_critica();  
        turn = 1;  
        sectiune_necritica();  
    }  
}
```

```
P1 () {  
    while(1) {  
        while(turn != 1) ;  
        sectiune_critica();  
        turn = 1;  
        sectiune_necritica();  
    }  
}
```

busy-waiting = procesele asteapta sa le vina randul sa intre in sectiunea critica, ocupand procesorul

! Ineficient pe sisteme multicore, sectiunea critica ar trebui sa fie cat mai scurta la busy-waiting !
Corectitudine:

1. Are exclusiune mutuală
2. Asteptarea limitata e respectata datorita alternantei stricte
3. Cerinta de progres totusi nu e respectata pentru ca unul din procese poate sa petreaca foarte mult timp inafara zonei critice, in mod nejustificat

Algoritmul lui Peterson

```

const int N = 2;

bool flag[N];

int turn;

```

<pre> void P1(int i) { j = 1 - i; flag[i] = true; turn = j; while(flag[j] && (turn == j)) ; sectiune_critica(); flag[i] = false; sectiune_necritica(); } </pre>	<pre> void P1(int j) { i = 1 - j; flag[j] = true; turn = i; while(flag[i] && (turn == i)) ; sectiune_critica(); flag[j] = false; sectiune_necritica(); } </pre>
---	---

SC = sectiune critica

Corectitudine:

1. Excludere mutuală
 - a. Pi intra în SC doar dacă $\text{flag}[j] == \text{false}$ sau $\text{turn} == i$
 - b. pp atât P0 cat și P1 sunt simultan în SC \Rightarrow ambele au executat liniile 7 & 8, deci $\text{flag}[0] == \text{flag}[1] == \text{true}$
 - c. cf. celor două ipoteze de mai sus, P0 și P1 nu puteau executa linia 9 în același timp, pt că turn nu poate fi 0 sau 1 simultan \Rightarrow unul dintre procese, sa zicem Pj trebuie să fi executat cu succes linia 9 (instructiunea while) în vreme ce celalalt a executat $\text{turn} = j$;
 - d. dar, în acest moment, $\text{flag}[j]=\text{true} \& \& \text{turn} == j$ și Pi trebuie să aștepte pana cand Pj ieșe din SC \Rightarrow excluderea mutuală e indeplinită
2. Progres
 - a. Pi nu poate intra în SC doar dacă $\text{flag}[j]=\text{true}$ și $\text{turn} == j$
 - b. dacă Pj nu este interesat să intre în SC $\text{flag}[j] == \text{false}$ și Pi poate intra în SC
 - c. altfel, la ieșirea din SC, Pj setează $\text{flag}[j] = \text{false}$ și poate petrece oricât timp în sectiune necritică fără să impiedice Pi să intre în SC
3. Așteptare limitată
 - a. dacă Pj vrea să intre în SC, setează $\text{flag}[j]=\text{true}$ dar și $\text{turn} = i$
 - b. pt că Pi nu schimba valoarea lui turn în linia 9 (instructiunea while), va intra în SC după cel mult o intrare a lui Pj în SC

!!! Aceasta soluție este inconsistentă pentru multithreading, pentru că contează ordinea operațiilor !!!

Pentru a ne asigura că algoritmul lui Peterson funcționează, trebuie să ne folosim de Memory Barrier.

Memory barrier

Memory model = „promisiunile memoriei” facute de un computer către aplicații

Memory model sunt de 2 tipuri:

1. Strongly ordered – unde o schimbare în memoria unui procesor este imediat vizibilă de către celelalte procesoare

2. Weakly ordered – unde o schimbare in memoria unui procesor nu este neaparat vizibila imediat tuturor celorlalte procesoare

Memory barrier = instructiune care forteaza ca orice schimbare facuta in memorie sa fie facuta vizibila tuturor procesoarelor

!! Memory barrier urile obliga programul sa execute mai intai o bucată anume de modificari in memorie, apoi urmand sa incarce restul de chestii

Synchronization hardware pentru critical section

1. Hardware instructions

a. Test-and-Set

```
boolean test_and_set (boolean *target)
{
    boolean rv = *target;
    *target = true;
    return rv;
}
```

- Proprietati:
- Executat atomic
- Returneaza valoarea originala a parametrului
- Seteaza noua valoare a parametrului la TRUE

SOLUTIE:

```
do {
    while (test_and_set(&lock))
        ; /* do nothing */

    /* critical section */

    lock = false;
    /* remainder section */
} while (true);
```

SPINLOCKS cu test_and_set:

Locks = concept de nivel inalt pentru protectia spatiului critic

Ofera două operații:

- ~ acquire (apelat la intrarea în SC)
- ~ release (apelat la ieșirea din SC)

! Cand un proces are lock-ul, celelalte sunt in busy waiting („spinning in the while”, de unde și denumirea) !

b. Compare-and-Swap

```

int compare_and_swap(int *value, int expected, int new_value)
{
    int temp = *value;
    if (*value == expected)
        *value = new_value;
    return temp;
}

```

- Proprietati:
- Executat atomic
- Returneaza valoarea initiala a primului parametru
- Seteaza valoarea primului parametru la valoarea celui de-al treilea, dar numai daca `*value == expected` e adevarat
- Schimbul se executa doar in cazul de mai sus

SOLUTIE:

Int lock = 0;

```

while (true) {
    while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */

    /* critical section */

    lock = 0;

    /* remainder section */
}

```

BOUNDED-WAITING (producator – consumator) cu compare_and_swap

```

while (true) {
    waiting[i] = true;
    key = 1;
    while (waiting[i] && key == 1)
        key = compare_and_swap(&lock,0,1);
    waiting[i] = false;
    /* critical section */
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = 0;
    else
        waiting[j] = false;
    /* remainder section */
}

```

!! AMBELE SE EXECUTA ATOMIC (FARA INTRERUPERE) !!

VARIABILE ATOMICE

- Asigura updateuri atomice pe date basic precum int si bool
- Exemplu, fie sequence o variabila atomica si increment o functie, daca facem increment(&sequence) suntem siguri ca &sequence se incrementeaza fara intrerupere
- O implementare a functiei increment poate fi urmatoarea:

```

void increment(atomic_int *v)
{
    int temp;
    do {
        temp = *v;
    }
    while (temp != (compare_and_swap(v,temp,temp+1)));
}

```

Suport hardware RISC pentru sectiuni critice

TAS (test_and_swap) si CAS (compare_and_swap) sunt specifice procesoarelor CISC, dar sevenetele atomice de tip read-modify-write nu se pot implementa pe procesoare RISC (arhitecturi load/store)

Operatii speciale RISC:

* Load Linked (LL)

* Store Conditional (SC)

LL = incarca o variabila din memorie intr-un registru CPU si apoi verifica activ daca variabila din memorie este modificata de alte procesoare

SC = verifica daca au existat modificarile variabilei de memorie la ultimul LL

Daca nu exista se salveaza 1 pe registru, altfel stocarea esueaza , iar valoarea registrului se seteaza pe 0

Implementare atomica cu LL/SC

- Mecanism folosit pentru sincronizarea wait-free/lock-free
- Simuleaza executia unei tranzactii din baza de date conform principiilor ACID (Atomicity, Consistency, Isolation, Durability)

1 increment:

```
2    ll      $2, count      ; incarca valoarea counter-ului
3    addu   $3, $2, 1       ; incrementa valoarea counter
4    sc     $3, count      ; incercă să stocheze noua valoare
5    beq    $3, zero, increment ; zero înseamnă exec
6    j      $31            ; return din rutina
```

Proprietati ACID:

* Atomicitate – modificarile (incrementarea counter-ului) sunt executate ca si cand operatia ar fi atomica, adica fie toate sunt executate, fie niciuna

* Consistenta – datele (counter-ul) sunt intr-o stare consistenta cand tranzactia incepe si cand se termina, adica counter-ul incrementat corect, chiar daca se executa mai multe tranzactii simultan, eventual intreatesut

* Izolare – starea intermedia a tranzactiei e invizibila altor tranzactii

* Durabilitate – dupa incheierea cu succes a tranzactiei, datele sunt salvate in memoria principală RAM si modificarea e finala

Comparatie CAS vs LL/SC

- Daca au aparut modificarile counter-ului, SC va esua garantat, chiar daca valoarea initial citita de LL a fost restaurata
- Daca se incercă secvanta de operatii cu CAS, semantica LL/SC e mai puternica decat CAS
- Atât CAS cat si LL/SC se pot folosi pentru implementarea sincronizarii wait-free

Suport hardware pentru multiprocesoare

- Felul in care se interconecteaza procesoarele individuale in sistemele multiprocesor determina in general si tipul de acces la memorie: UMA, NUMA, NORMA

- In sistemele UMA/NUMA procesoarele partajeaza datele prin variabile partajate de memorie accesate sincronizat (ex. cu ajutorul lock urile)

- In sistemele NORMA accesul coordonat la date se face prin message passing

Multiprocesoare cu o singura magistrala

- BUS ul unic e principala limitare pentru cresterea nr de CPU uri

- Folosirea cache urilor reduce traficul si permite cresterea de procesoare in UMA

- Existenta cache urilor in general duce la copii multiple ale aceleiasi locatii din memorie, adica potential de inconsistenta daca un CPU modifica datele
- Totusi, accesul uncached la datele partajate nu e indicat deoarece scade viteza accesului la date si creste traficul pe bus, ceea ce incetineste accesul tuturor
- Solutia vine prin folosirea unor protocoale de mentinere a coerentei cache-urilor locale
- Ex: Snooper Coherency Protocol
 - * Controlerul cache-urilor monitorizeaza bus-ul pt traficul care afecteaza datele din cache urile lor
 - * Read miss : controllerul localizeaza o copie actualizata a datelor
 - * Write : exista doua protocoale posibile
 - a) write-invalidate : invalideaza toate copiile celorlalte procesoare inainte de a scrie datele in cache-ul local (ex: Intel MESI)
 - b) write-update: scrie datele modificate pe bus printre-o operatie de tip broadcast, astfel incat sa se updateze peste tot copiile

Instructiuni atomice multiprocesor

- Se folosesc instructiuni atomice de tip TAS / CAS
- Cand un CPU executa TAS, CPU ul acela primeste exclusiv magistrala pe durata executiei instructiunii, se executa ciclul read-modify-write si dupa aceea se deblocheaza bus-ul pentru uzul altor procesoare
- Intr-un ciclu read-modify-write se citeste si se scrie o valoare in mod atomic, iar datele nu sunt cached, astfel avem urmatoarele consecinte:
 - => operatie mai costisitoare decat operatiile de citire/scriere obisnuite
 - => magistrala e ocupata
 - => operatiile TAS afecteaza semnificativ rata de transfer pe magistrala
 - => alte consecinte privind implementarea spinlock urilor in sisteme multiprocesor

```

void acquire(char *lock_ptr)
{
    disable_interrupts();
    while(tas(lock_ptr))
        ;
}
void release(char *lock_ptr)
{
    *lock_ptr = 0;
    enable_interrupts();
}

```

OBS:

- Dezactivarea intreruperilor in acquire impiedica pierderea procesorului pe durata detinerii lock ului

- Pierderea procesorului ar inseamna ca procese/thread-uri de pe alte CPU nu pot intra in sectiunea critica din cauza unui proces/thread care nu ruleaza
- Pe durata detinerii unui lock, celelalte procesoare executa TAS in bucla si consuma inutil largimea de banda a magistralei afectand procesoarele care nu sunt implicate in accesul la sectiunea critica
- La release, procesorul care detine lock-ul concureaza cu celelalte procesoare pentru accesul la magistrala inainte de a putea ceda lock-ul
- Solutia ar fi implementarea lui test-and-test-and-set (TATAS)

TATAS

- Traficul pe bus se poate reduce daca procesoarele cicleaza in acquire pe o copie locala a spinlock-ului
- Pentru ca release este in fond o operatie de scriere a spinlock-ului, protocolul de coerență snoop detecteaza scrierea si invalideaza copiile locale ale celorlalte procesoare (in cazul protocolului write-validate)
- Cand celelalte procesoare acceseaza din nou spinlock-ul => cache miss
- La cache miss copia locala se va actualiza si va reflecta starea de lock free
- Procesorul incerca din nou sa execute TAS sperand ca acum lock-ul e probabil free

```

void acquire(char *lock_ptr)
{
    disable_interrupts();
    while(*lock_ptr || tas(lock_ptr))      Obs: codul se bazeaza pe evaluarea
        ;                                scurt circuitata a conditiilor in C
}
void release(char *lock_ptr)
{
    *lock_ptr = 0;
    enable_interrupts();
}

```

TATAS cu backoff

- Introducem un delay inainte de a incerca din nou operatia TAS
- Intarzierea poate fi:
 - Statica
 - Fiecare CPU are un slot
 - Merge bine pentru multe CPU-uri
 - Intarzie nejustificata un singur CPU chiar daca lock-ul e liber
 - Dinamica
 - Toate CPU-urile aleg o intarziere mica, ceea ce duce la coliziuni
 - Dupa detectarea coliziunilor, fiecare CPU maresteste intarzierea

- Overhead ul intarzierilor mici de la inceput face metoda mai costisitoare decat alocarea statica a intarzierilor

OBS:

Folosirea protocolului write-update poate imbunatatii performanta TATAS prin reducerea traficului pe bus

```
void acquire(char *lock_ptr)           void release(char *lock_ptr)
{
    disable_interrupts();             {
    while(*lock_ptr || tas(lock_ptr))   *lock_ptr = 0;
    {                                enable_interrupts();
        while(*lock_ptr)
            ;
        delay();
    }
}
}
```

Sincronizare wait-free

- Se pot folosi instructiuni hardware de tipul CAS/CAS2 sau LL/SC pentru a implementa
- Instructiunile atomice simplifica IPC pe multiprocesoare, dar au si dezavantaje:
 - Pot avea performanta redusa daca sunt folosite neglijent
 - Daca un proces/thread pierde CPU sau e forcat sa astepte mult in timp ce are lock ul, celelalte procese nu pot avansa
 - Daca un proces/thread se termina anormal in timp ce detine lock ul, nici un alt proces nu poate intra in sectiunea critica
 - Inversarea prioritatilor - cand un proces/thread cu prioritate mica detine lock ul si impiedica un proces cu prioritate mare sa intre in sectiunea critica
- Totusi, incercă sa rezolve aceste dezavantaje folosind o forma de control optimist al concurentei
- IDEE CENTRALA: se incercă executia operatiei, dar se lasă datele intr-o stare consistentă daca operatia esueaza
- Model Herlihy:
 - Sistem cu n procese secentiale
 - Obiect concurrent = ({tip + valori}, set operatii, specificatie secentiala)
 - Obiect concurrent neblocant - procesul care executa una dintre operatiile sale trebuie sa o termine intr-un numar finit de pasi
 - Obiect concurrent wait-free - fiecare proces din sistem trebuie sa termine operatie intr-un numar finit de pasi
 - conditia de operare nonblocanta inseamna ca unele procese/threaduri vor face intotdeauna progres indiferent de intarzierile sau opririle forcate ale altor procese/threaduri

- conditia de operare wait-free e mai puternica => toate procesele/threadurile care nu sunt oprite fac progres indiferent de terminarile anormale sau intarzierile altor procese/threaduri
- ambele conditii exclud folosirea sectiunilor critice ca metoda de implementare, pentru ca un proces/thread care se termina abnormal (cu eroare) in mijlocul unei sectiuni critice va bloca pt totdeauna celelalte procese care asteapta sa intre in sectiunea critica
- rezultat teoretic important: este imposibil sa se construiasca implementari neblocante sau wait-free ale tipurilor de date simple folosind operatii de citire/scriere, TAS, fetch-and-add, swap de memorie cu registre (eg, instructiunea Intel xchg)

Mutex locks

- Cel mai simplu mutex lock este o variabile booleana care indica faptul ca lock ul este liber sau nu
- Pentru a proteja sectiunea critica ne putem folosi de acquire si release, insa acestea trebuie sa fie atomice
- Insa, solutia cu mutex implica folosirea busy waiting ului,, astfel ca intra in categoria spinlock ului

```
while (true) {
    acquire lock
    critical section
    release lock
remainder section
}
```

Sleep/wakeup

- Spinlock urile sufera de busy-waiting si ca atare de problemele discutate anterior (ex. Inversiunea de prioritati)
- Busy-waiting ul poate fi inlocuit de primitiva sleep , care blocheaza procesele care nu au dreptul de a intra in sectiunea critica
- Cand procesul aflat in sectiunea critica o paraseste, apeleaza primitiva wakeup care trezeste toate procesele blocate in sleep
- In continuare, problema producator-consumator implementata cu sleep/wakeup

#define N 100

Int count = 0;

```

Void producer(void){
Int item;

While (TRUE){
produce_item(&item);           //generate next item
If (count == N) sleep();       // if buffer full, go to sleep
enter_item(item);             //else, put item in buffer
Count = count + 1;            //increment number of items in buffer
If (count == 1) wakeup(consumer); // if buffer was empty, wakeup consumer
}

```

```

Void consumer(void){
Int item;

While (TRUE) {
If (count == 0) sleep();        //if buffer empty, go to sleep
remove_item(&item);           //else, take an item
Count = count - 1;             // decrement the number of items in buffer
If (count == N-1) wakeup(producer); // if buffer was full, wakeup producer
consume_item(item);           //consume the item
}

```

Race condition la solutia sleep/wakeup

- Presupunem ca buferul e gol, consumatorul a citit count == 0 si inainte sa execute sleep schedulerul da controlul producatorului
- producatorul produce un obiect, observa ca buferul era gol, crede ca procesul consumator executa sleep si apeleaza wakeup pt a-l notifica
- cand revine pe CPU, consumatorul executa sleep => notificarea wakeup s-a pierdut !
- pt ca procesul consumator nu a apucat sa consume obiectul produs, producatorul continua sa produca obiecte pana umple buferul complet si e nevoie sa apeleze sleep - in acest moment, ambele procese sunt blocate => deadlock !
- problema esentiala: accesul nesincronizat la variabila count
- solutie posibila: wakeup seteaza un waiting bit pe care consumatorul il citeste si nu intra in sleep
- Limitare neplacuta: in general e nevoie de un numar arbitrar de waiting bits => e necesara o primitiva speciala (semafoare)

Semafoare

- Structura de date speciala (Dijkstra 1965)
- Are un contor care numara wakeup urile
- Are coada de procese blocate in sleep
- Are 2 operatii:
 - down

Operatie care decrementeaza ATOMIC contorul daca e > 0 si permite procesului sa continue

Daca contorul e 0, blocheaza procesul si il pune in coada

- up

Verifica daca exista procese blocate in coada, iar daca avem alege unu si il deblocheaza, altfel incrementeaza atomic contorul

- Un semafor cu contor initializat 1 este de fapt un mutex (semafor binar)
- up/down generalizeaza sleep/wakeup DAR sunt operatii ATOMICE
- OBS: up nu blocheaza niciodata procese !!

Producator-consumator cu semafoare

```
item_t buffer[N]
semaphore mutex = {1}, space = {N},
items = {0};

void producer(void)
{
    item_t item;
    while(true) {
        produce(&item);
        down(space);
        down(mutex);
        put(buffer, &item);
        up(mutex);
        up(items);
    }
}
```

```
void consumer(void)
{
    item_t item;
    while(true) {
        down(items);
        down(mutex);
        get(buffer, &item);
        up(mutex);
        up(space);
        consume(&item);
    }
}
```

Conditii implementare semafoare

- Trebuie sa garantam ca oricare 2 procese nu pot executa up si down in acelasi timp pe acelasi semafor
- Astfel, implementarea devine si ea la randu ei o problema de critical section
- Am putea sa o facem cu busy-waiting , insa implementarea ar trebui sa fie scurta, iar daca o aplicatie sta mult in critical section, atunci nu este o alegere buna
- Astfel ca pentru fiecare semafor vom face un waiting queue si fiecare intrare va avea un value si un pointer catre urmatoarea inregistrare din lista
- Avem 2 operatii:
 - Block - pune procesul care invoca operatia pe lista potrivita
 - Wakeup - sterge unu din procese din waiting queue si il pune in ready queue

```

typedef struct {
    int value;
    struct process *list;
} semaphore;

down(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}

up(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup (P);
    }
}

```

Exemplu de greseli producator - consumator

- Producatorul inverseaza secventa de accesare a semafoarelor
- Daca bufferul e plin, producatorul se blocheaza in operatia down(space), iar mutex = 0
- Consumatorul porneste si executa down(items) si down (mutex) => Deadlock ! (mutex = 0)
- Concluzie : e nevoie de mecanisme de sincronizare de nivel inalt asistate de compilator!

Monitors

- O abstractizare de nivel inalt care asigura un mecanism bun pentru sincronizarea proceselor
- Abstract data type, variabilele interne pot fi accesate doar de codul din procedura
- Numai un singur proces poate fi activ in monitor la un moment dat
- Pseudocodul unui monitor:

```

monitor monitor-name
{
    // shared variable declarations
    procedure P1 (...) { .... }

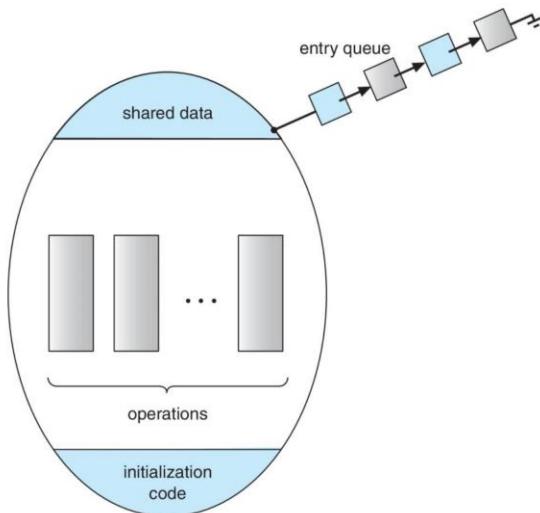
    procedure P2 (...) { .... }

    procedure Pn (...) {.....}

    initialization code (...) { ... }
}

```

O schema a monitorului:



Asistenta compilatorului

- Procedurile monitorului sunt instrumentate de compilator sa execute o secventa speciala de apelare
- Cand un proces apeleaza o procedura din monitor, primele instructiuni ale procedurii verifica daca exista alt proces activ in interiorul monitorului. Daca da, procesul e suspendat pana cand celalalt paraseste monitorul, iar daca nu, procesul apelant poate intra in monitor. Implementarea se face in mod uzual folosind un monitor
- Avantaj: sectiunile critice se implementeaza ca proceduri de monitor, iar compilatorul genereaza automat cod de excludere mutuala
- Implementare:

Semaphore mutex

Mutex = 1

down(mutex)

...;

```
Body of P;  
...  
up(mutex);
```

!Excluderea mutuală e cu siguranta aplicată în interiorul unui monitor

Variabile conditie

- Problema: Ce se întâmplă dacă un proces aflat în interiorul monitorului trebuie să se blocheze?
- Solutie: variabile conditie
- Acestea suportă 2 operații:
 - Wait - se blochează procesul apelant și eliberează monitorul pentru ca alte procese blocate la intrarea în monitor să poată accesa monitorul
 - Signal - trezeste procesul care a apelat anterior wait pe această variabilă conditie

Problema signal:

Dacă trezeste un proces care a apelat anterior wait, vor exista simultan două procese în interiorul monitorului

Solutii:

- a) Hoare - procesul care executa signal e suspendat, și celalalt proces e lăsat să ruleze în monitor
- b) Hansen - signal nu se poate executa decât ca ultima operație a unei proceduri de monitor (adică procesul care cheamă signal parasește imediat monitorul după execuția operației)

Uzual, se folosește Hansen pentru simplitatea implementării

OBS:

- Variabilele conditie nu contorizează "semnalele", adică o operație signal pe o variabilă conditie pe care nu așteaptă nimănii să pierde => wait trebuie executat înainte de signal
- wait/signal aproximează comportamentul sleep/wakeup, DAR race conditionul sleep/wakeup este înălțat de excluderea mutuală automată a accesului în monitor
- Monitoarele sunt un concept de nivel de limbaj de programare, spre deosebire de semafoare care pot fi implementate și ca apeluri de biblioteca
- Atât semafoarele cât și monitoarele funcționează doar pe sisteme cu memorie partajată , NU pe sisteme distribuite

Monitor implementation folosind semafoare

Semaphore mutex; //initial 1

Semaphore next; // initial 0

Int next_count = 0; //numarul de procese în coada

Down (mutex);

...

Body of P

...

if(next_count>0)

up(next);

Else

 up(mutex);

Sa-mi bag pula in monitoare, de la slide-ul 89 pana la 95 nu am mai scris

Liveness

- Procesele pot sa aiba de asteptat un timp infinit pana primesc lock-ul
- Asteptarea asta infinita incalca progresul si bounded-waiting-ul puse ca si criteriu
- Liveness = un set de proprietati pe care un sistem trebuie sa le aiba ca sa se asigura ca procesele fac progres
- Asteptarea infinita e un exemplu de liveness esuat

Deadlock = 2 sau mai multe procese asteapta la infinit un eveniment care poate fi facut de un proces care asteapta si el

P_0

down (S) ;

down (Q) ;

...

up (S) ;

up (Q) ;

P_1

down (Q) ;

down (S) ;

...

up (Q) ;

up (S) ;

Pe acest exemplu, consideram ca P_0 executa $\text{down}(S)$ si P_1 executa $\text{down}(Q)$, astfel ca P_0 trebuie sa astepte sa fie eliberat Q , iar P_1 asteapta sa se elibereze S , dar cum niciunul dintre ele nu poate continua, am ajuns astfel intr-o situatie de deadlock

Conditii necesare pentru deadlock

- Excludere mutuală - doar un proces/thread poate utiliza o resursă la un moment dat
- Hold & wait - procesul/threadul detine o resursă și asteapta să obtină și alte resurse detinute de alte procese/threaduri
- Fără preemptiune - resursele detinute nu pot fi confiscate forțat, ci doar cedate voluntar la finalul task-ului
- Asteptarea circulară - un set de procese/threaduri aflate într-un lanț de asteptare

Producerea unui deadlock presupune ca toate cele 4 conditii sa fie respectate !

OBS: (4) => (2), adica nu toate conditiile sunt independente unele de celelalte

- Alte forme de deadlock:

 Starvation -indefinite blocking

- Un proces poate să nu mai fie niciodată scos din coada semaforului în care este suspendat

 Priority inversion

 * O problema de programare atunci cand un proces cu prioritate scazuta tine lock-ul necesar unui proces cu prioritate mai mare

* Se rezolva prin priority-inheritance protocol
Incidentul Mars Pathfinder exemplifica asta si il poti citi la pagina 100

END CHAPTER 6

CHAPTER 7

Probleme clasice de sincronizare:

1. Bounded-buffer problem
2. Readers and Writers Problem
3. Dining-Philosophers Problem

READERS-WRITERS PROBLEM

- Un data set e impartit de mai multe procese concurente impartite in doua categorii:
 - Readers - pot doar sa citeasca data set ul, nu pot sa il updateze
 - Writers - pot sa si citeasca si sa scrie
- Problema: Permiterea mai multor cititori sa citeasca in acelasi timp, iar la writeri sa permitem unui singur sa aiba acces la data in momentul ala
- Datele impartite vor fi:
 - Data set ul
 - Semaforul rw_mutex initializat cu 1
 - Semaforul mutex initializat cu 1
 - Integer read_count initializat cu 0

• Structura unui writer

```
while (true) {
    down(rw_mutex);

    ...
    /* writing is performed */

    ...
    up(rw_mutex);
}
```

- Structura unui reader

```

while (true) {
    down(mutex);
    read_count++;
    if (read_count == 1) /* first reader */
        down(rw_mutex);
    up(mutex);

    ...
/* reading is performed */

    ...

    down(mutex);
    read_count--;
    if (read_count == 0) /* last reader */
        up(rw_mutex);
    up(mutex);
}

```

Variatii ale problemei:

- Deoarece problema de mai sus poate duce la starvationul writerului, astfel ca ea se va numi First reader-writer problem
- Second reader-writer problem e o variație care spune că odată ce un writer a inceput să scrie, niciun nou reader nu mai are voie să citească
- Ambele pot duce la starvation, care pot duce la mai multe variații
- Problema e rezolvată pe unele sistemele de kernel care da lock-urile reader-writer

Dining-philosophers Problem

- N filozofi sunt la masa și vor să manance.
- Starile lor sunt fie THINKING, fie EATING
- Pentru a manca ei au nevoie de 2 betisoare, însă sunt doar 5 betisoare
- Problema poate fi văzută și ca mancarea = data set-ului și cele 5 betisoare văzute ca un semafor initializat cu 1
- Solutie cu semafoare:

```

#define THINKING      0
#define HUNGRY        1
#define EATING         2

int state[N];
semaphore_t  mutex = {1};
semaphore_t  ph[N]; // toate initializeaza cu 0

void philosopher(int i) {
    while(true) {
        think();
        take_forks(i);
        eat();
        put_forks(i);
    }
}

void take_forks(int i) {
    down(mutex);
    state[i] = HUNGRY;
    test(i);
    up(mutex);
    down(ph[i]);
}

void put_forks(int i) {
    down(mutex);
    state[i] = THINKING;
    test((i - 1) % N);
    test((i + 1) % N);
    up(mutex);
}

void test(int i) {
    if(state[i] == HUNGRY &&
       state[(i-1)%N] != EATING) &&
       state[(i+1)%N] != EATING)
    {
        state[i] = EATING;
        up(ph[i]);
    }
}

```



- Solutie cu monitor

```

monitor DiningPhilosophers
{
    enum {THINKING, HUNGRY, EATING} state [5];
    condition self [5];

    void pickup (int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING) self[i].wait;
    }

    void putdown (int i) {
        state[i] = THINKING;
        // test left and right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
    }

    void test (int i) {
        if ((state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING) ) {
            state[i] = EATING ;
            self[i].signal () ;
        }
    }

    initialization_code() {
        for (int i = 0; i < 5; i++)
            state[i] = THINKING;
    }
}

```

Corectitudine

- Fiecare filozof i invoca operatia pickup() si putdown() in aceasta ordine, astfel ca o sa fie imposibil sa intre in deadlock, insa este posibil sa avem starvation

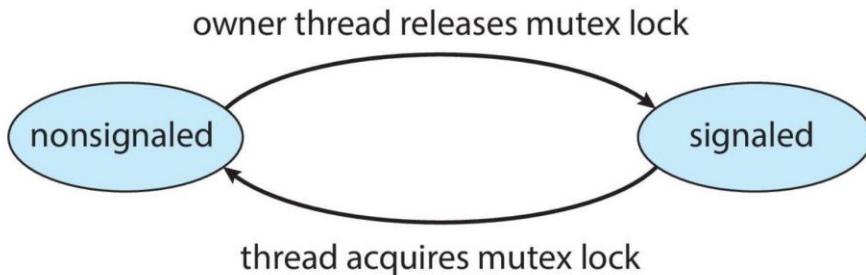
KERNEL Synchronization - Windows

- Foloseste masti pentru a proteja accesul la resurse globale pe sisteme cu un singur procesor
- Foloseste spinlock pe sistemele cu mai multe procesoare
- Totodata, asigura dispatcher objects in user-land care pot sa aiba rol de mutex, semafor, evenuri si timere

Event = este ceva foarte asemănător cu condition variable

Timer = notifică unul sau mai multe threaduri când le-a expirat timpul

- Dispatcher objects sunt fie în signaled-state (object available), fie în non-signaled state (thread will block)



KERNEL Synchronization - Linux

- Înainte de 2.6 nu avea intreruperi, însă după v2.6 a devenit fully preemptive
- Linux asigură semafoare, atomic integers, spinlocks și Reader-writer ambele variante
- Pe un sistem cu un singur procesor, spinlockurile sunt înlocuite de activarea și dezactivarea preemptivness-ului
- Atomic variables

```
atomic_t counter;
int value;
```

<i>Atomic Operation</i>	<i>Effect</i>
<code>atomic_set(&counter, 5);</code>	<code>counter = 5</code>
<code>atomic_add(10, &counter);</code>	<code>counter = counter + 10</code>
<code>atomic_sub(4, &counter);</code>	<code>counter = counter - 4</code>
<code>atomic_inc(&counter);</code>	<code>counter = counter + 1</code>
<code>value = atomic_read(&counter);</code>	<code>value = 12</code>

Sincronizare în Solaris

- Lockuri adaptive, variabile de condiție, semafoare, reader-writer locks, turnstiles
- Lockuri adaptive:
 - Folosite pentru secțiuni critice scurte
 - Comportament polimorf spinlock-semafor
 - Dacă un thread vrea acces la date protejate de un lock detinut de un alt thread care rulează pe alt CPU, lock-ul se comportă ca un spinlock
 - Dacă lock-ul e detinut de un thread care nu rulează momentan, atunci thread-ul care încearcă să obțina va fi pus în sleep, adică lock-ul se comportă ca un semafor

- Pe un sistem uniprocesor, cand un thread incearca sa obtina un lock detinut de alt thread, comportamentul lock ului e intotdeauna de tip semafor

Sincronizare POSIX

- Asigura mutex, semafoare si condition variables
- Folosit foarte mult pe UNIX, Linux si macOS
- Mutex locks:

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex,NULL);

/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

- Semafoare
 - a) Named - pot fi folosite de procese cu care nu au legatura

```
#include <semaphore.h>
sem_t *sem;

/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);

/* acquire the semaphore */
sem_wait(sem);

/* critical section */

/* release the semaphore */
sem_post(sem);

b) unnamed
```

```

#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);

/* acquire the semaphore */
sem_wait(&sem);

/* critical section */

/* release the semaphore */
sem_post(&sem);

```

- Condition variables

```

pthread_mutex_t mutex;
pthread_cond_t cond_var;

pthread_mutex_init(&mutex,NULL);
pthread_cond_init(&cond_var,NULL);

```

- Pentru ca un thread sa astepte dupa o conditie folosim pthread_cond_wait() si pentru a da signal la un condition variable folosim pthread_cond_signal()

Memorie tranzactionala

- Un thread termina modificarile operate pe o zona de memorie partajata fara a se coordona cu alte threaduri
- Se inregistreaza fiecare operatie de citire/scriere intr-un log
- La finalul tranzactiei se incarca operatia de COMMIT
 - Daca reuseste, atunci tranzactia e validata si modificarile devin permanente
 - Daca esueaza, tranzactia e abandonata si re-executata pana reuseste
- Beneficiu major: grad crescut de concurrenta, threadurile nu au nevoie sa se sincronizeze prin lock-uri la accesul memoriei partajate

Suport hardware pentru memorie tranzactionala

- Accesibila prin intermediul interfetelor HLE (Hardware Lock Elision) si RTM (Restricted Transactional Memory)
- HLE adauga doua prefixuri : XACQUIRE si XRELEASE
 - Se pot folosi doar pentru anumite instructiuni care trebuie prefixate explicit cu LOCK

- Permite executia optimista a sectiunii critice sarind scrierea lock ului, astfel incat acesta apare liber pentru alte thread uri
- O tranzactie esuata determina reluarea operatiei de la instructiunea prefixata cu XACQUIRE
- RTM adauga la ISA trei noi instructiuni:
 - XBEGIN - marcheaza inceputul zonei de memorie tranzactionala
 - XEND - marcheaza sfarsitul zonei de memorie tranzactionala
 - XABORT - abandoneaza explicit o tranzactie
 - Esecul tranzactiei redirecteaza executia catre codul specificat de instructiunea XBEGIN, iar codul de eroare e stocat in registrul EAX
- Instructiunea XTEST permite testarea starii procesorului (este sau nu in mijlocul executiei unei regiuni de memorie tranzactionala)
- Exemplu RTM:

```

void elided_lock_wrapper(lock) {
    if(_xbegin() == _XBEGIN_STARTED) { // start tranzactie
        if(lock e liber)
            return; // executa SC in tranzactie
        _xabort(0xff); // abandoneaza tranzactia
    }
    obtine lock
}

void elided_unlock_wrapper(lock) {
    if (lock e liber)
        _xend(); // comite tranzactia
    else
        elibereaza lock;
}

```

END CHAPTER 7

CHAPTER 5

CPU Scheduling - concepte basic

- Maximizarea folosirii CPU ului se obtine cu multiprogramming
- CPU - I/O Burst Cycle = proces de executie care consta intr-un ciclu de CPU execution si asteptarea dupa I/O
- CPU burst e urmat de I/O burst
- Problema principala o reprezinta distributia CPU burst urilor

CPU Scheduler

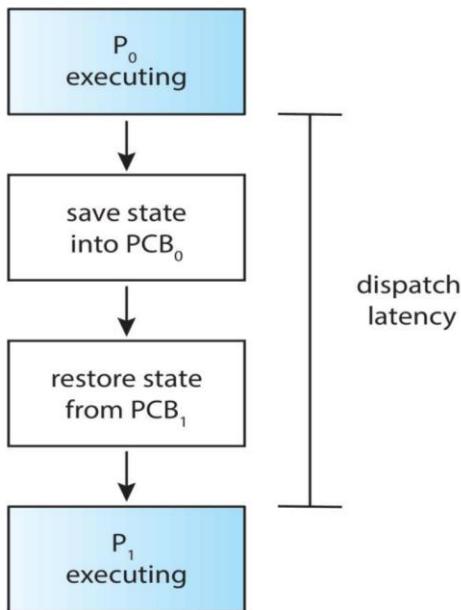
- Selecteaza din procesele ready si le aloca unui CPU core
- !! Coada de procese ready poate fi aranjata in mai multe moduri
- CPU Scheduler ul ia decizii atunci cand un proces:
 - Trece din running in waiting
 - Trece din running in ready
 - Trece din waiting in ready
 - Se termina
- Pentru prima si ultima situatie nu exista alegeri legate de scheduling, pentru ca un nou proces trebuie ales pentru executare
- Insa, pentru al doilea si al treilea exista alegeri

Preemptive si Nonpreemptive Scheduling

- La prima si la a patra avem nonpreemptive, insa la a doua si a treia avem preemptive
- Sub scheduler nonpreemptive, odata ce CPU ul e alocat unui proces, acesta tine procesul pana cand fie se termina, fie trece in waiting state
- Toate sistemele de operare moderne folosesc algoritmi de scheduling preemptivi
- Totusi, preemptive scheduling ul poate duce la un race condition atunci cand data e impartasita de-a lungul mai multor procese

Dispatcher

- Dispatcher module ul ofera control CPU ului catre procesul ales, iar acest lucru implica:
 - Switching context
 - Switching la user mode
 - Sarirea la locatia din user program ca sa il restarteze
- Dispatch latency = timpul cat dureaza oprirea unui proces si inceperea rularii altuia (practic durata evenimentului descris mai sus)



Criterii de scheduling

- CPU utilization - sa tinem CPU ul cat se poate de ocupat
- Throughput - Numarul de procese care se termina per unitate de timp
- Turnaround time - Timpul necesar sa execute un anume proces
- Waiting time - Timpul pe care un proces il petrece in ready queue
- Response time - Timpul de cand un proces a fost submis pana cand e produs primul response
- Astfel, noi vom dori sa optimizam urmatoarele lucruri:
 - Max CPU utilization
 - Max throughput
 - Min turnaround time
 - Min waiting time
 - Min response time

ALGORITMI DE SCHEDULING

FIRST COME, FIRST SERVED (FCFS)

- Presupunem ca primim procesele P1,P2 si P3 in ordinea asta si burst time de 24, 3 si 3
- Scheduler ul nostru va arata astfel:



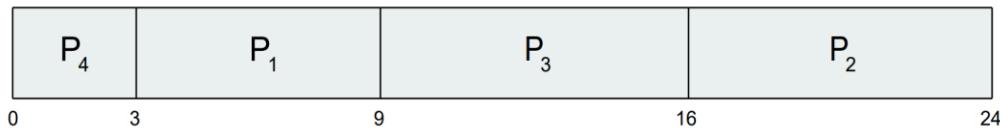
- Si vom avea waiting time $P_1 = 0$; $P_2 = 24$; $P_3 = 27$, avand astfel un timp mediu de asteptare egal cu 17

- OBS: Daca am fi primit mai intai P2, P3 si dupa P1, am fi avut waiting time mediu de 3 !
- OBS2: Apare convoy effect ul - procesele scurte in spatele celor lungi. Un exemplu ar putea fi un CPU-bound proces si multe procese I/O-bound

SHORTEST JOB FIRST (SJF)

- I se mai spune si Shortest Job Next (SJN)
 - Initial folosit pentru sisteme batch, cand timpii de rulare sunt cunoscuti a priori
 - Algoritmul este simplu, se alege cel mai scurt job primul, astfel ca aceste este si optimal:
 - Pp t_1, t_2, \dots, t_n timpii de rulare
 - Job 1 termina la t_1
 - Job 2 termina la $t_1 + t_2$
 - ...
 - Job n termina la $t_1 + t_2 + \dots + t_n$
- $$\Rightarrow \text{timp mediu de asteptare} = \frac{1}{n}(n t_1 + (n-1) t_2 + \dots + t_n)$$

- OBS: SJF e optimal DOAR daca joburile sunt disponibile simultan !!
- Ex de SJF: Primim procesele P1 (6), P2(8), P3(7) si P4(3). Le ordonam conform SJF si obtinem P4, P1, P3, P2



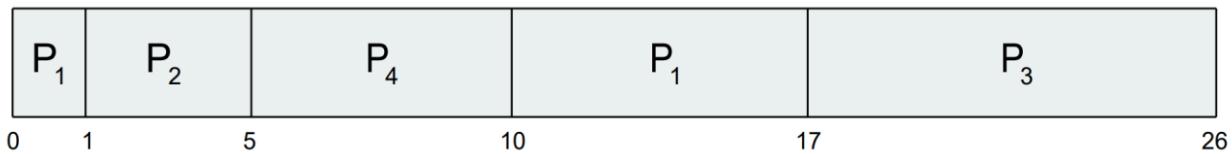
- Obtinem un average time egal cu 7

Shortest Remaining Time First (SRTF)

- Este varianta preemptive a lui SJF
- De fiecare data cand un nou proces ajunge in ready queue, se ia o decizie de scheduling care se face cu ajutorul SJF ului
- Exemplu de SRTF:

<u>Process</u>	<u>Arrival Time</u>	<u>Burst Time</u>
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

O sa obtinem urmatoarea diagrama Gantt:



Si vom obtine un average waiting time = $[(10-1) + (1-1) + (17-2) + (5-3)] / 4 = 26/4 = 6.5$

ROUND ROBIN (RR)

- Fiecare proces primeste o unitate mica de timp pe CPU (time quantum notat cu q) de obicei intre 10 si 100 milisecunde. Dupa acest timp, procesul este preempted si adaugat la finalul ready queue ului
- Performanta RR ului este data de cuanta de timp, astfel daca avem q mare ajungem sa avem un FCFS, iar daca avem un q mic facem intr-adevar un RR
- OBS: Cuanta de timp trebuie sa aiba grija si la context switch, intrucat daca este prea mica nu o sa functioneze algoritmul

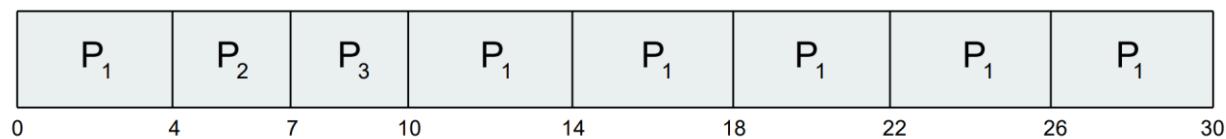
Observatii:

- Run/ready queue e implementata circular, mutarea procesului la finalul listei revine la a muta un pointer circular
- Problema mare este alegerea cuantei de timp. Daca avem 20 ms cuanta si 5 ms context switch ul, ajungem sa avem un overhead de 20%, iar daca avem cuanta de 500ms si 10 useri, primul user va vedea instant o parte din rezultate, in timp ce al 10-lea va astepta 5 secunde. Astfel, cuanta mica duce la reducerea eficientei utilizarii CPU ului , iar cuanta mare duce la reducerea timpul de raspuns pentru programe interactive

Exemplu de Round Robin cu cuanta de 4

<u>Process</u>	<u>Burst Time</u>
P_1	24
P_2	3
P_3	3

Pentru aceste procese si cuanta de 4 vom avea urmatorul Gantt chart:



Timpul de asteptare mediu va fi 5.66

De obicei, va avea tunraround time mai mare ca SJF, dar va avea un raspuns mult mai bun (interactivitate)

PLANIFICAREA CU PRIORITATI

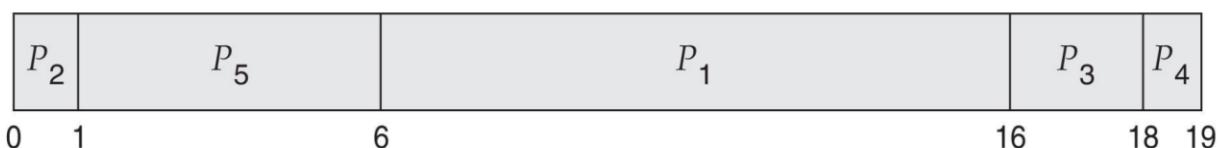
- RR presupune ca toate procesele au aceeasi prioritare, ceva nerealist
- Cuantificarea "importantei" se face cu ajutorul prioritatilor
- Procesul cel mai important => Prioritate maxima
- Planificatorul va alege intotdeauna procesul cu prioritate maxima
- Pentru a impiedica procesele cu prioritate maxima sa ruleze indefinit , planificatorul poate reduce prioritatea procesului care ruleaza cu fiecare tact de ceas
- Cand prioritatea scade sub cea a urmatorului proces, facem context switch si continuam cu ala
- Prioritatile pot fi statice sau dinamice, iar pentru asignarea dinamica am putea sa dam prioritate mare proceselor I/O bound pentru ca ruleaza putin si sa poata continua cu lansarea urmatoarelor cereri I/O, astfel ca am putea sa le dam prioritatea $1/f$, unde f e fractiunea din cuanta de CPU utilizata la ultima rulare, adica cu cat ruleaza mai putin cu atat are prioritate mai mare
- O alta abordare e creearea unor clase de prioritate:
 - N clase de prioritate
 - Se ruleaza RR pe procesele din clasa maxima
 - Daca nu mai exista alte procese acolo, mergem la urmatoarea clasa
 - Si aici e nevoie de o ajustare a prioritatilor pentru a evita starvation in clasele de prioritate mica

Exemplu de Planificare cu prioritati

In exemplu nostru cu cat e mai mica prioritatea, cu atat o planificam mai rapid

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Diagrama Gantt:



In acest exemplu vom obtine un average waiting time de 8.2

Exemplu de Planificare cu prioritati, dar de data asta cu Round Robin de cuanta 2

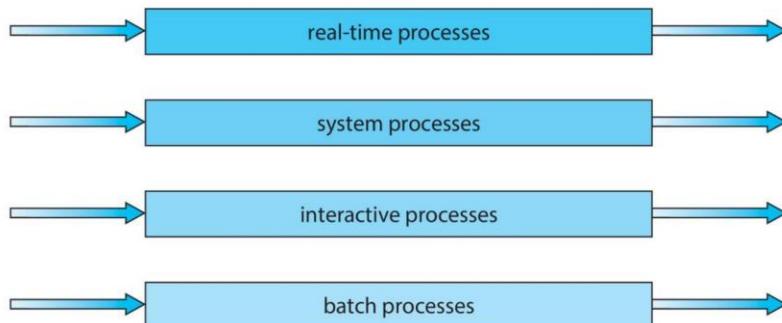
<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

P_4		P_2	P_3	P_2	P_3	P_2	P_3	P_1	P_5	P_1	P_5		
		0	7	9	11	13	15	16	20	22	24	26	27

Multilevel Queue

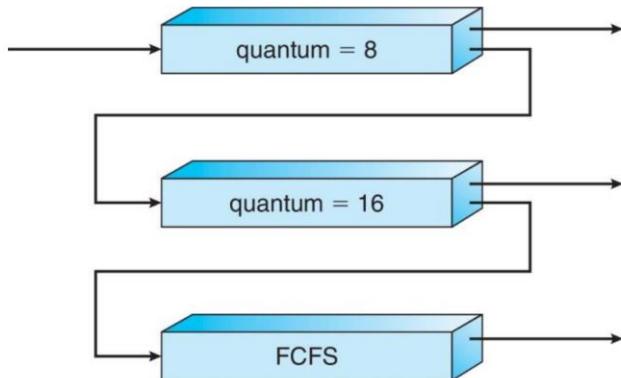
- Ready queue ul va fi format din mai multe queue uri
- Putem definiti multilevel queue scheduler ul dupa urmatorii parametrii:
 - Numar de cozi
 - Scheduling algorithm ul pentru fiecare coada
 - Metoda folosita sa aleaga in ce coada pune un proces anume
 - Scheduling ul intre cozi
- Cu priority scheduling, putem sa avem cozi care ne dau prioritatea (adica metoda gandita mai sus cu n clase de prioritate)
- O metoda de a prioritiza procesele poate fi:

highest priority



lowest priority

- O variație la Multilevel queue e să adaugăm și un Feedback, astfel încât la definitia lui adaugăm și metodele de a să cunoască upgradam/downgradam un proces. Totodată, poate fi implementată și o metodă de aging.
- Exemplu de Multilevel Feedback Queue:



Daca un proces trece de cuanta de 8, merge in coada cu cuantum 16, iar daca nici acolo nu se termina ajunge in coada de FCFS, unde cu siguranta va fi executat

THREAD SCHEDULING

- Trebuie facuta distinctie intre user-level si kernel-level threads
- Cand kernel thread uri sunt suportate, o sa avem doar kernel threads, nu si procese
- La Many-to-One and Many-to-Many models, thread library schedules user-level thread urile sa ruleze pe LWP.
 - Cunoscut ca si process-contention scope (PCS) , deoarece competitia se intampla doar la nivel de proces
 - De obicei facut de programator prin setarea prioritatii
- Kernel threadul scheduled pe un CPU disponibil e un system-contention scope (SCS), deoarece competitia se intampla la nivel de system

Pthread Scheduling

- API ul specifica daca vrea PCS sau SCS in timpul creeari
 - PTHREAD_SCOPE_PROCESS foloseste PCS
 - PTHREAD_SCOPE_SYSTEM foloseste SCS
 - OBS: Poate fi limitat de sistem ! Linux si macOS permit doar SCS

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
    int i, scope;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* first inquire on the current scope */
    if (pthread_attr_getscope(&attr, &scope) != 0)
        fprintf(stderr, "Unable to get scheduling scope\n");
    else {
        if (scope == PTHREAD_SCOPE_PROCESS)
            printf("PTHREAD_SCOPE_PROCESS");
        else if (scope == PTHREAD_SCOPE_SYSTEM)
            printf("PTHREAD_SCOPE_SYSTEM");
        else
            fprintf(stderr, "Illegal scope value.\n");
    }
}
```

```

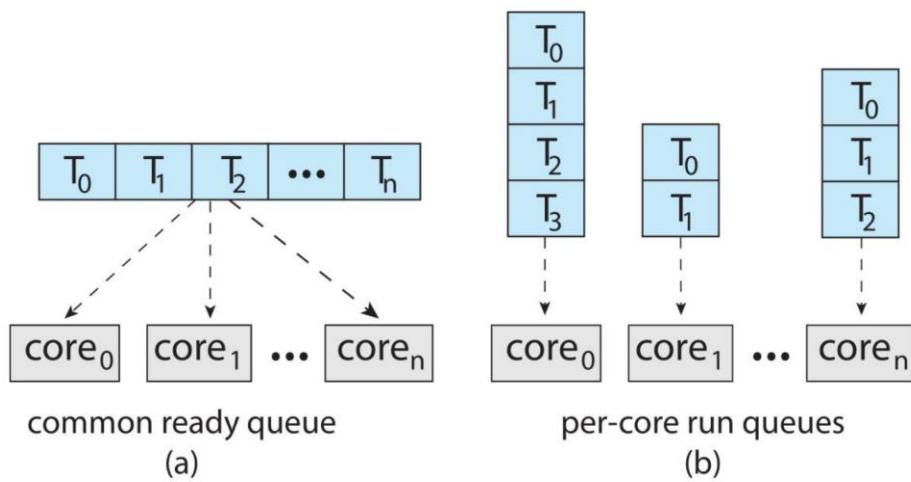
/* set the scheduling algorithm to PCS or SCS */
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
/* create the threads */
for (i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], &attr, runner, NULL);
/* now join on each thread */
for (i = 0; i < NUM_THREADS; i++)
    pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this function */
void *runner(void *param)
{
    /* do some work ... */
    pthread_exit(0);
}

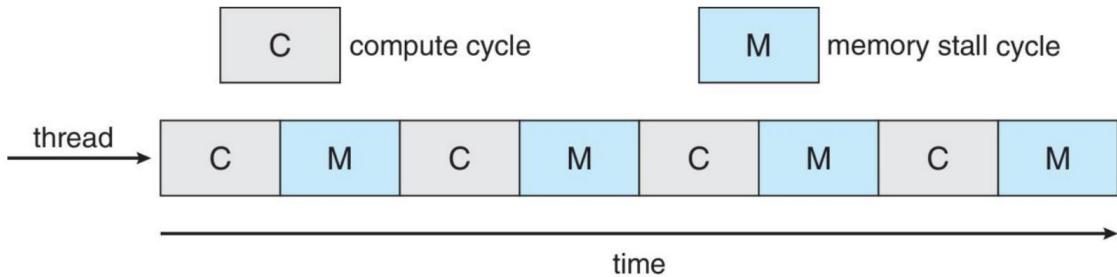
```

Multiple Processor Scheduling

- Devine mai complicat cand avem mai multe CPU uri
- Multiprocess poate sa aiba oricare din urmatoarele arhitecturi:
 - Multicore CPUs
 - Multithreaded cores
 - NUMA systems
 - Heterogeneous multiprocessing
- Simetric multiprocessing (SMP) este atunci cand fiecare procesor isi face singur planificare
- Toate thread uri pot sa fie intr-un ready queue normal, insa fiecare proces poate sa aibe o coada privata de thread uri

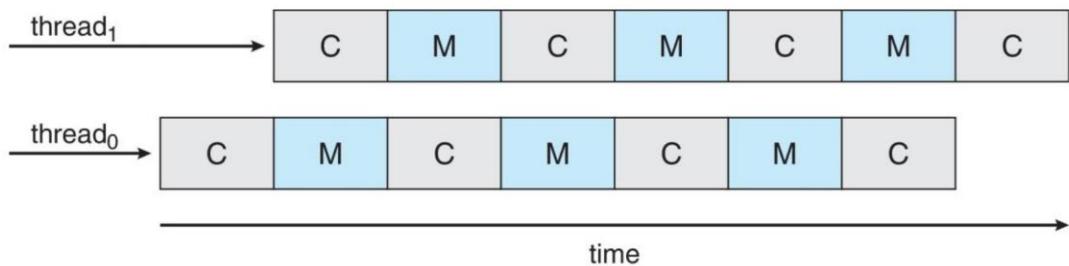


- Un trend recent e sa pui mai multe processor core uri pe un singur chip, deoarece este si mai rapid si consuma si mai putina electricitate. Totodata, si numarul de thread uri per core a crescut, astfel folosinduse de memory stall ca sa faca progres pe un alt thread pana isi ia ce are nevoie din memorie

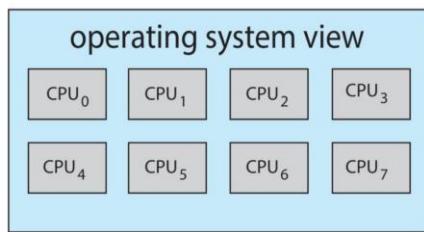
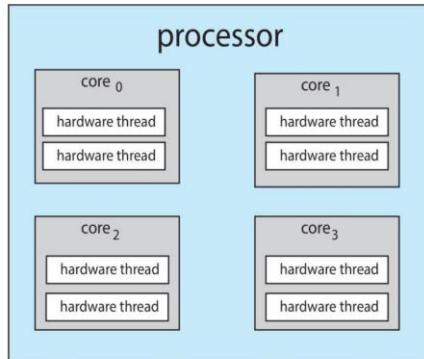


Multithreaded Multicore System

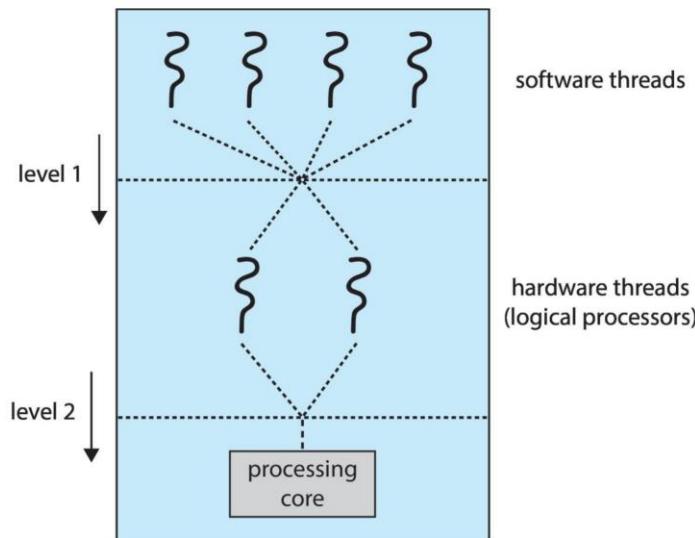
- Fiecare core are > 1 hardware threads
- Daca unu din ele are memory stall, trece la urmatorul



- Chip-multithreadind (CMT) acorda fiecarui core mai multe hardware threads. Ex: Pe un quad core system cu 2 hardware threads per core, sistemul de operare va vedea 8 logical procesoare



- 2 nivele de scheduling:
 - 1) Sistemul de operare decide care software thread il pune pe logical CPU
 - 2) Cum fiecare core decide ce hardware thread sa ruleze



Multiple-Processor Scheduling - Load Balancing

- Daca folosim SMP , atunci avem nevoie sa mentinem toate CPU urile incarcate pentru eficienta
- Load balancing ul incearca sa rezolve aceasta problema si sa imparta in mod egal
- Push migration ul este un task periodic care verifica incarcarea pe fiecare CPU si daca gaseste si daca gaseste il trimitre pe proces de la CPU supraincarcat catre altele

- Pull migration e procesor prin care procesoarele care nu fac nimic iau procese din waiting queue de la procesorul supraincarcat

Multiple-Processor Scheduling - Processor Affinity

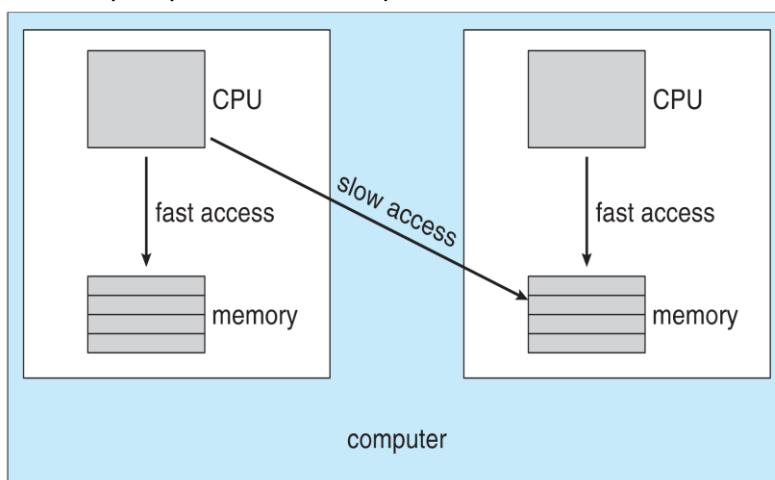
- Cand un thread ruleaza pe un CPU , memoria thread ului ramane stocata in cache-ul CPU-ului, astfel ca am putea sa spunem ca thread are o "afinitate" la acest procesor
- Load balancing-ul poate sa afecteze aceasta afinitate cand un thread este mutat intre procesoare
- Astfel, putem defini două afinitati:

Soft affinity = sistemul de operare incercă să îl pastreze pe același procesor

Hard affinity = sistemul de operare permite specificarea exactă a procesoarelor pe care dorim să rulăm procesul

NUMA și CPU Scheduling

- Dacă sistemul de operare este NUMA-aware, atunci îl poate să îi asigneze memoria cea mai apropiată de CPU-ul pe care rulează thread-ul



NORMA Scheduling

- În sistemele distribuite, fiecare procesor executa local algoritmul de planificare
- Q: Cum se comportă un grup de procese care interacționează între ele și rulează pe mașini diferite

ex: procesele A, B rulează pe P1; C și D pe P2

mașinile P1 și P2 rulează RR cu q=10ms

A și C rulează în cuantele pare, B și D în cele impare

A porneste și cheama D în cuanta pară, D nu rulează (C rulează în cuantele pare)

dupa 10ms, D primește mesajul lui A și raspunde; cuanta e impara, A nu rulează (B rulează)

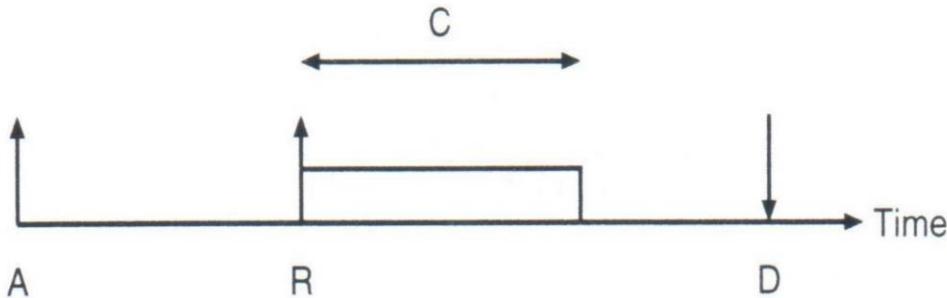
dupa inca 10 ms, A primește mesajul lui D

schimbul de mesaje are un overhead de 20 ms

- Idee: procesele care comunică frecvent să ruleze simultan pe mașini diferite
- Astfel, apare ideea de co-scheduling cu matrici:
- fiecare coloană conține procesele rulate pe un anume procesor (mașina)
- fiecare linie conține procesele care rulează simultan procesoare
- i.e., coloanele reprezintă procesoare (mașini), liniile cuante de timp
- fiecare procesor planifică RR local
- procesoarele își sincronizează cuantele RR folosind bcast (un ceas sincron ca la SIMD, posibil implementat prin NTP)
- co-scheduling: toate procesele unui grup au alocate aceleasi cuante de timp (i.e., apar pe aceleasi linii ale matricii in diferite coloane) => maximizarea paralelismului disponibil și a throughput-ului comunicatiei

Planificare de timp real

- Sistem de timp real = sistem în care corectitudinea executiei unui program nu depinde doar de corectitudinea rezultatelor calculate ci și de timpul la care sunt livrate rezultatele
- Se împart în 2
 - Hard - Toate rezultatele trebuie livrate la timp
 - Soft - Rezultatele calculate pot fi acceptate și cu întâzieră
- Task = o singura executie a unui bloc de cod
- Timp de sosire al taskului = timpul la care sistemul devine conștient că taskul trebuie executat
- Taskurile periodice sosesc la interval regulate de timp, cu $T(t)$ timpul fix inter-sosiri (perioada) și $C(t)$ timpul de calcul
- Taskurile aperiodice:
 - Caracterizate de rate de sosiri stochastice
 - Pot veni "în rafala" => supraincarcare sistem
 - În consecință, se presupune că există un timp minim între sosirile succesive ale aceluiasi task
 - Asemenea task uri se numesc sporadice, însă și acestea pot veni într-un interval scurt de timp, iar dacă sistemul nu are suficiente resurse se ajunge la nerespectarea dealine urilor și spunem că sistemul se află într-o supraincarcare trecatoare
- Timpul de eliberare = timpul la care taskului î se permite să pornească executia
- Deadline = timpul până la care trebuie să se termine executia task ului
- Diagrama de timp : Sosire -> Release -> Calcul -> Deadline



Slide urile 56-60 nu sunt aici

Rate Monotonic (RM)

- Algoritm pentru taskuri periodice (intersante pentru modelarea usoara si folosirea pe scara larga in industrie/automatizari)
- RM este un algoritm offline de planificare a unor taskuri care respecta urmatoarele constrangeri:
 - 1) Cererile taskurilor cu deadline hard sunt periodice, cu interval constant intre cereri
 - 2) Deadlineurile constau exclusiv in constrangeri de executabilitate, adica fiecare task trebuie sa termine inainte sa apara urmatoarea cerere pentru task
 - 3) Taskurile sunt independente, cererile pentru un anumit task de initiere sau terminarea cererilor pentru alte taskuri
 - 4) Timpul de executie al fiecarui task e constant (nu variaza in timp); timpul de executie inseamna timpul CPU necesar sa execute taskul fara intrerupere
 - 5) Orice task neperiodic din sistem este special: e fie rutine de initializare, fie de recuperare din eroare. Inlocuiesc taskurile periodice cand ruleaza, dar nu deadlinurile hard

ALGORITMUL RM

- Taskurile au asignate prioritati statice, adica taskurile cu rate de sosiri mari au prioritate mare
- Taskurile se executa preemptiv
- Planificatorul alege intotdeauna taskul cu prioritate maxima
- Utilizarea U a procesorului de catre n taskuri :

$$U = \sum_{i=1}^n \frac{C_i}{T_i}$$

LEMA:

Daca $U < \ln 2$, planificarea RM e fezabila (toate deadlineurile sunt respectate)

In practica:

$U < \ln 2$ e o cerinta conservatorie, exista seturi de taskuri care se pot planifica RM si a caror utilizare depaseste $\ln 2$.

- Exemplu RM

Avem un sistem multimedia care are un task audio cu perioada $T_a = 10$ ms si $C_a = 2$ ms si un task video cu $T_v = 30$ ms si $C_v = 10$ ms $\Rightarrow U_a = 2/10 = 0.2$ si $U_v = 10/30 = 0.30 \Rightarrow U_{\text{total}} = 0.53 < \ln 2 = 0.69$, adica avem taskuri planificabile

Taskul audio are frecventa maxima (perioada cea mai scurta), deci are si prioritate maxima

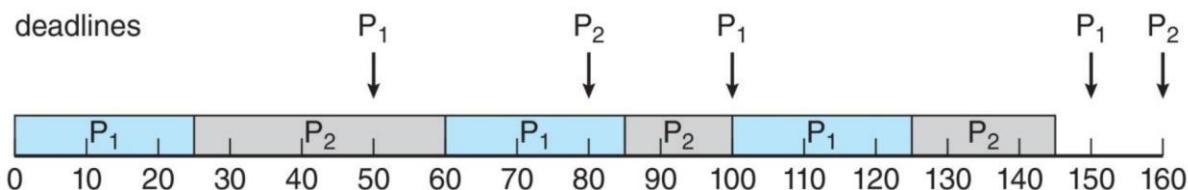
Presupunem ca $C_v = 20$ ms $\Rightarrow U_v = 20/30 = 0.66 \Rightarrow U_{\text{total}} = 0.86 > \ln 2$. Totusi, o diagrama de timp arata ca taskurile sunt planificabile RM daca taskul audio are prioritate maxima

Earliest Deadline First (EDF)

- Taskul cu deadline-ul cel mai apropiat are prioritate maxima
- Un set de n taskuri e planificabil daca

$$U = \sum_{i=1}^n \frac{C_i}{T_i} \leq 1$$

- Ambele exemple de la RM sunt planificabile si EDF
- Prioritatile sunt acordate dupa deadline, astfel ca daca deadline-ul e mai devreme are prioritate mai mare si daca deadline e mai tarziu are prioritate mai mica
- Figura:



LEAST SLACK FIRST (LSF)

- Slack = durata maxima de timp cu care un task poate fi intarziat fara a-si pierde deadline-ul
- LSF alege pentru rulare taskul cu slack-ul ce mai mic

Planificarea taskurilor aperiodice

- EDF si LSF se pot folosi si pentru planificarea taskurilor aperiodice
- Se poate demonstra faptul ca EDF e optimal
 - Daca orice alt algoritm produce o planificare fezabila pentru un set de taskuri ale caror timpi de sosire, timpi de calcul si deadline-uri sunt cunoscute, atunci si EDF o poate face
- Exista o demonstratie similara de optimalitate si pentru LSF

Proportional Share Scheduling

- T shares sunt allocate tuturor proceselor din sistem
- O aplicatie primeste N shares , unde $N < T$, astfel ca fiecare aplicatie va primi N/T din timpul total pe procesor
- Planificarea de timp real functioneaza bine in sisteme embedded (mix-ul de job-uri e cunoscut a priori)
- In medii dinamice sistemul poate deveni supraincarcat, iar aplicatiile de timp real nu-si mai pot respecta deadline urile
- Solutiile de planificare bazate pe prioritati nu sunt intotdeauna utile, de exemplu daca avem un task computational intensiv (video) de prioritate mare intarzie indefinit taskurile non real-time de prioritate mica (compilari)
- Abordare noua in planificarea de procese:
 - Renuntam la conditia ca toate procesele sa termine cu siguranta inainte de deadline
 - Planificatorul controleaza rata de executie a proceselor variind “proportia” de resurse (timp CPU) pe care procesele o primesc

Planificarea de tip loterie

- Un algoritm randomizat de alocare proportionala a resurselor in general: timp CPU, largime de banda I/O, alocare de memorie si accesul la lockuri
- Controleaza ratele relative de executie ale proceselor variind procentul de alocare al resursei (ex. Timpul CPU pe care il primeste procesorul)
- Rata de “consum” a resursei de catre un proces activ este proportionala cu procentul de alocare a resursei respective detinut de proces

Tichete de loterie

- Drepturile asupra resurselor sunt exprimate prin intermediul unor tichete de loterie
- Alocarea resursei se face prin organizarea unei loterii:
 - Resursa e acordata procesului care detine tichetul castigator
 - Alocarea e proportionala cu numarul de tichete detinut de proces
- Tichetele de loterie reprezinta dreptul de a folosi o fractiune din resursa (suma tichetelor detinute de un proces defineste procentul total de utilizare a resursei la care are dreptul procesul). Totodata, acestea sunt abstracte, relative si uniforme:
 - Abstracte: cuantifica dreptul de folosire a resursei independent de detalii hardware
 - Relative: fractiunea de resursa pe care o cuantifica variaza proportional cu numarul total de tichete detinute:
 - Variatia e dinamica, in functie de competitia la resursa
 - Un proces va obtine o fractiune mai mare a unei resurse cu competitie mica decat in cazul uneia cu competitie mare
 - In cel mai rau caz, un proces primeste o fractiune de resursa proportionala cu nr sau de tichete

- Uniforme: indiferent de tipul de resursa, procentele de alocare ale proceselor sunt reprezentate omogen prin tichete

Algoritm

- La inceputul fiecarei cuante, planificatorul organizeaza o loterie si resursa (CPU ul) e acordata detinatorului tichetului castigator
- Tichetul castigator este generat aleator conform unei distributii uniforme
- Apoi se cauta in lista de procese ready participantul care detine tichetul castigator

Exemplu: Numar total de tichete = 20, si avem 5 procese cu [10,2,5,1,2] tichete si alegem un numar random intre 0 si 19 si neiese 15

S1 = 10 ; S1 > 15? NU

S2 = 12 ; S2 > 15? NU

S3 = 17 ; S3 > 15? DA => procesul numarul 3 primeste procesorul

Analiza algoritmului

- Running time:
 - Generarea de nr aleatoare e rapida
 - Traversarea listei de procese O(n)
 - Acumularea sumei partiale
- Optimizari pentru numarul de procese ce trebuie examinate:
 - Pt distributie inegală, ordonam descrescator lista de procese conform numarului de tichete detinute => lungimea medie de cautare scade pentru ca procesele cu numar mare de tichete sunt frecvent selectate
 - Pentru n mare, se poate folosi un arbore cu sume partiale in nodurile interne si numarul de tichete ale proceselor pe frunze => localizarea tichetului castigator se face pornind din radacina catre procesul castigator , complexitate O(log n)

Caracteristici LS

- Fair din punct de vedere probabilistic
 - Dupa o perioada suficienta de timp, valoarea estimata a fractiunii de alocare a resursei este proportionala cu numarul de tichete detinut de proces
 - OBS: nu e garantat ca un proces cu t tichete din totalul T va primi exact t / T din utilizarea resursei
 - Diferenta scade insa pe masura ce numarul de alocari creste
 - Explicatie: Fie X = numarul de loterii castigate de un proces, variabila aleatoare cu distributie binomiala
 - Fie p = t/T probabilitatea , iar dupa n loterii identice vom avea E[x] = np , iar Var(x) = np(1-p)

Adica throughput ul procesului e proportional cu numarul de tichete detinut, iar acuratetea estimarii se imbunatatesta cu viteza lui radical din n

(1) fair dpdv probabilistic

- fie $Y = \text{nr de loterii necesare ca un proces sa castige prima data}$ loteria, variabila aleatoare cu distributie geometrica
- $E[Y] = \frac{1}{p}$ iar $Var(Y) = \frac{1-p}{p^2}$
⇒ timpul mediu de raspuns este invers proportional cu nr de tickete detinut de proces
- Tichetele se pot transfera de la un proces la altul atunci cand acesta se blocheaza dintr-un motiv oarecare
- Inflatia de tickete: Alternativa la transferul de tickete
 - Procesul isi marea dreptul la resurse prin creearea artificiala de tickete pentru sine
 - Metoda se poate folosi intre procese care au incredere reciproca intre ele
 - Prin inflatie/deflatie de tickete procesele controleaza alocarea resursei fara sa comunice explicit intre ele
 - In general este de evitat, pentru ca poate conduce la monopolizarea resursei
- Tickete de compensare
 - Cand un proces consuma doar o fractiune f din cuanta alocata pentru utilizarea resursei, el primeste un ticket de compensare
 - Ticketa de compensare ii creste procesului valoarea cu $1/f$ pana cand procesul primeste urmatoarea cuanta
 - Astfel, consumul resursei de f^*p este ajustat cu $1/f$ ca sa corespunda fractiunii reprezentate de p
 - In absenta unei astfel de metode, procesul care nu consuma intreaga fractiune alocata utilizeaza resursa mai putin decat are dreptul

Ramas la slide 83, de aici nu mai zice el nimic, asa ca citesti din slide uri

END OF CHAPTER 5

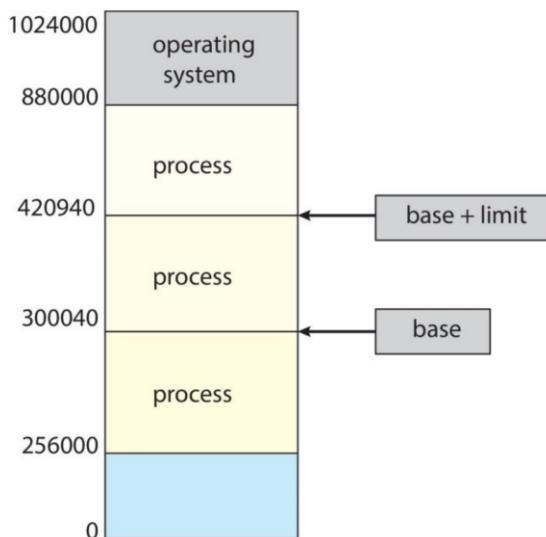
CHAPTER 9

Gestiunea memoriei

- Controleaza partile din memoria RAM utilizate si respectiv libere (neutilizate)
- Alocă/dealocă memorie pentru procese
- Gestionă spațiul de swap
- Swap = schimb de segmente de memorie între disc și memoria principală atunci când aceasta nu poate săține toate procesele rezidente
- În absența multiprogramării și a swapping-ului:
 - Un singur proces este rezident în memoria principală la un moment dat
 - aproape toată memoria principală este disponibilă procesului (mai puțin memoria ocupată de sistemul de operare)

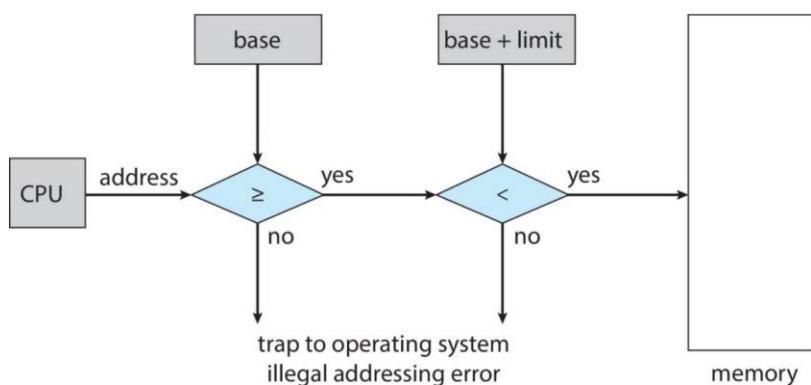
Background

- Programul trebuie sa fie adus de pe disc in memorie si pus intr-un proces pentru ca el sa ruleze
- Memoria principală și registrii sunt singura zona de memorie accesibila direct din CPU
- Unitatea de memorie vede doar un flux de adrese + read requests sau adrese + data and write request
- Accesul la registrii este facut intr-un singur CPU clock (sau mai putin)
- Memoria principală poate dura mai multe cicluri, cauzand un stall (o blocare)
- Cache-ul sta intre memoria principală și registrii CPU
- Protectia memoriei trebuie sa existe pentru a opera corect:
 - Trebuie sa ne asiguram ca un proces poate accesa doar adresele din address space ul lui
 - Putem sa protejam folosind o pereche de base si limit registers care sa defineasca spatiul unui proces



Hardware address protection

- CPU trebuie sa verifice orice acces la memorie generat in user mode si sa fie sigur ca e intre base si limit pentru acel user



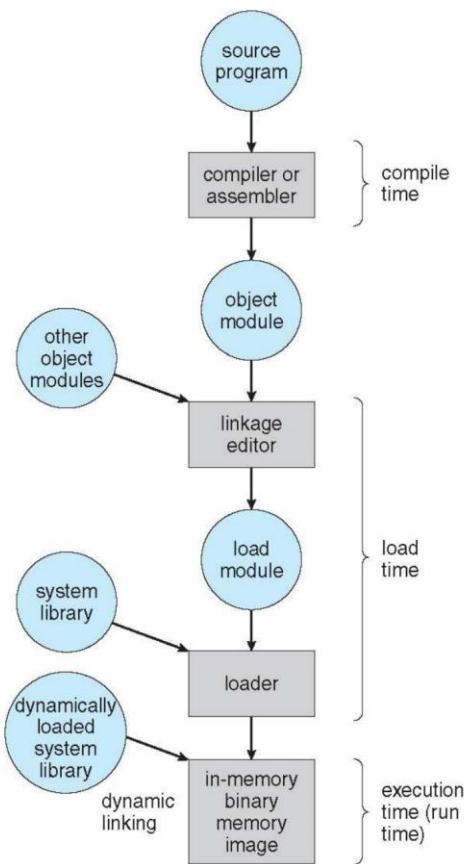
- Instructiunile de incarcare a base ului si limit ului sunt privilegiate
- OBS: Kernelul are acces nerestricționat la întreaga memorie, deopotrivă zona de memorie a sistemului de operare cat și a proceselor

Address Binding

- Programele de pe disk , ready să fie aduse în memorie să execute un input trebuie să fie loadate de la adresa 0000
- Este inconvenabil să avem primul proces user la adresa fizica 0000 mereu, astfel că adresele pot fi reprezentate în diferite stagiile vieții lor :
 - Source code address , de obicei simbolic
 - Compiled code addressul care se binduieste de o adresa mutabile (adică "14 bytes de la inceputul modulului")
 - Linker sau loader vor bindui adresele mutabile la adrese absolute (adică 74014)
 - Fiecare binding mapează un address space de altu

Binding of instructions and Data to memory

- Address binding de instructiuni și date la adrese de memorie se poate întâmpla la 3 stagiile diferite:
 - Compile time : Dacă stim locația memoriei a priori, absolut codul va fi generat și va trebui codul recompliat dacă locația de start se schimbă
 - Load time: Trebuie să genereze cod mutabil dacă locația din memorie nu este cunoscută la compilare
 - Execution time: Bindingul întârzie run time-ul dacă procesul poate fi mutat în timpul executiei de la un segment de memorie la altu, însă nu e nevoie de hardware suport pentru mapările de adrese (adică avem nevoie de registrii de base și limit)



Logical vs Physical Address Space

- Conceptul de logical address space care e legat de un physical address space este foarte important pentru o managementare buna a memoriei

Logical address = generat de CPU, se mai numesc si virtual address

Physical address = adresa vazuta de unitatea de memorie

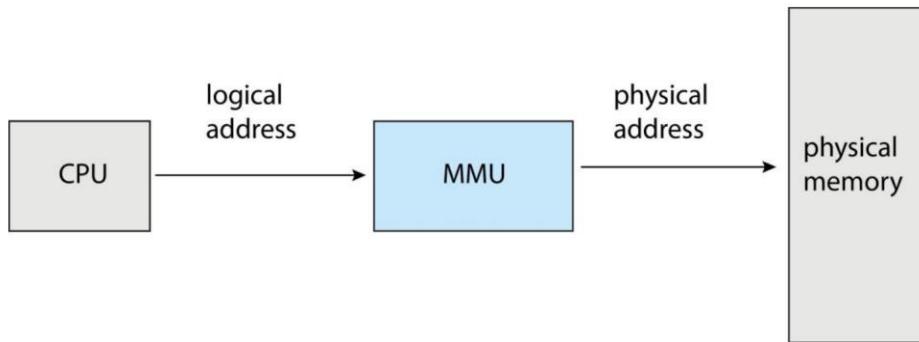
- Logical si physical addresses sunt aceleasi in compile-time si in load-time address-binding schemes; logical/virtual si physical addresses difera doar la execution-time address-binding scheme

Logical address space = set de adrese logica generate de un program

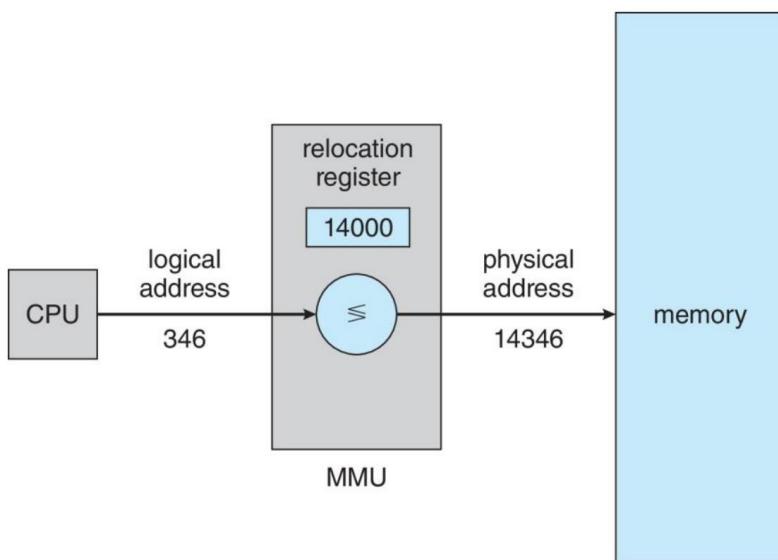
Physical address space = set de toate adresele fizice generate de un program

Memory Management Unit (MMU)

- Device hardware care la run-time mapeaza adresele virtuale de cele fizice



- Sa consideram o schema simpla, care este o generalizare a schemei lui base-register
- Base registerul va deveni acum relocation register
- Valoarea din relocation register va fi adaugata la fiecare adresa generata de un proces user la un moment dat si trimis in memorie
- Progmul user se ocupa de logical addresses ; el nu va lucra niciodata cu adrese fizice
 - Execution-time binding ul apare atunci cand se face o referinta la locatia din memorie
 - Logical addresses sunt legate de physical addresses



Dynamic loading

- Tot programul va trebui sa fie in memorie ca acesta sa fie executat
- Routine nu e incarcata pana nu e chemata
- Pentru o utilizare mai buna a memory space ului, rutinele nefolosite nu vor fi incarcate
- Toate rutinele sunt pastrate pe disc intr-un format mutabil
- Este foarte folositor pentru cand avem bucati mari de cod de care e nevoie ca sa se trateze cazuri rare
- Nu avem nevoie de suport de la sistemul de operare
 - E implementat prin program design

- Sistemul de operare poate sa ajute prin asigurarea de librarii care implementeaza dynamic loading
- Static linking - librarii de sistem si cod de program combinat de loader in imaginea binara a programului
- Dynamic linking - linking amanat pana la execution time
- O bucată mică de cod folosită să gasească memoria potrivită - resident library routine
- Stub = o bucată mică de cod
- Stub se înlocuiește cu adresa rutinei și o se executa
- Sistemul de operare verifică dacă o rutină este în memoria proceselor, iar dacă nu este în address space, îl adaugă
- Dynamic linking este foarte folositor în special pentru librarii
- OBS: spre deosebire de dynamic loading, dynamic linking are nevoie de support OS (presupune protecția memoriei între procese)

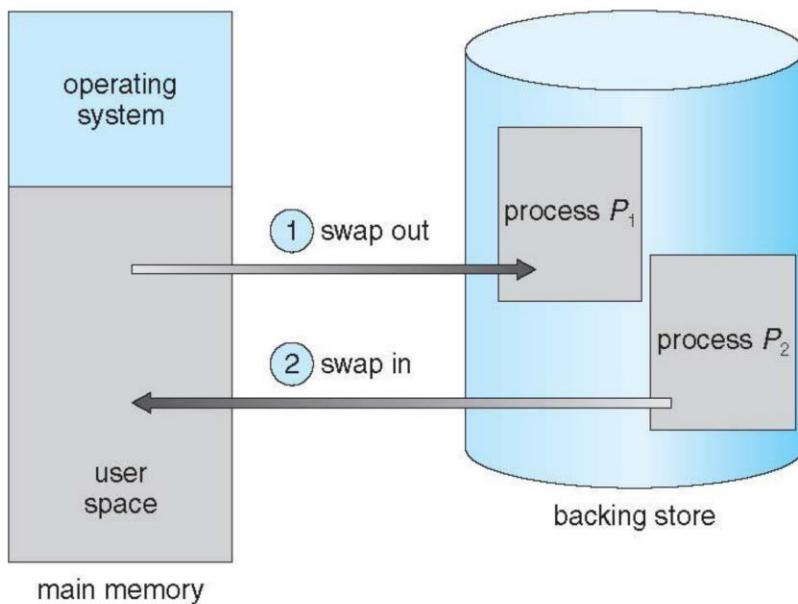
Swapping

- Un proces poate fi swapped temporar din memorie spre un backing store, apoi adus înapoi în memorie pentru a își continua execuția
 - Totalitatea spațiului fizic de adrese de memorie poate să intreacă memoria fizică
 - Creste gradul de multiprogramare

Backing store = un disk rapid și mare destul să poată sătine toate copiile de imagini de memorie a tuturor utilizatorilor și trebuie să aibă și acces direct la acestea

Roll out, roll in = varianta de swapping folosită pentru a face priority scheduling ; un proces cu prioritate mai mică va fi swapped cu un proces cu o prioritate mai mare ca să fie încărcat și executat acesta

- O parte foarte mare din swap time este reprezentată de transfer time; timpul total de transfer este proporțional cu cantitatea de memorie swapped
- Sistemele mențin un ready queue de procese pregătite să ruleze care arată memory images pe disk



Context Switch Time including Swapping

- Daca urmatoarele procese care urmeaza sa fie puse pe CPU nu sunt in memorie, trebuie sa facem swap cu un proces de acolo
- Context switch time ul poate sa fie foarte mare
- 100MB process swapping pe hard disk cu o rata de 50MB/sec :
 - Swap out time : 2000ms
 - Plus swap in de acelasi timp : 2000ms
 - Se ajunge la un context switch de 4000ms (4 sec)
- Se poate reduce dimensiunea daca stim exact cată memorie e cu adevarat folosita, spre exemplu putem folosi system callurile request_memory() și release_memory()
- Alte constrangeri ale swapping ului:
 - Asteptarea după I/O - nu poti face swap out intrucat I/O ul s-ar intampla în procesul incorrect
 - Sau intotdeauna transfera I/O ul spre kernel space , apoi adauga I/O device (e cunoscut si ca double buffering)
- Swapping ul standard nu e folosit in sistemele moderne, insa exista versiuni modificate foarte folosite, precum sa faci swap doar cand memorie libera e foarte mica sau cand faci swap doar pe parti ale procesului, in sisteme care implementeaza memorie virtuala

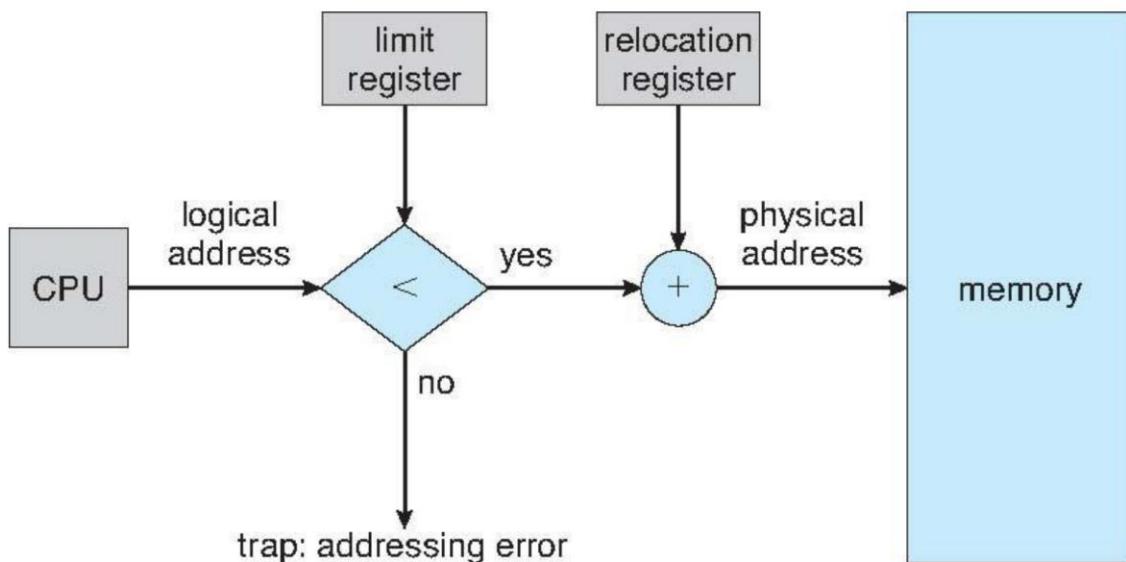
Swapping on Mobile Systems

- De obicei nu e suportat, deoarece se bazeaza pe flash memory care:
 - Foloseste putina memorie
 - Are un numar limitat de write cycles
 - Are un throughput slab intre flash memory si CPU pe mobile

- În schimb se folosesc alte metode, precum iOS care întreaba aplicațiile dacă pot să renunțe din spațiu alocat, android îl e mai brutal și termină aplicații dacă memorie e foarte mică

Contiguous Allocation

- Memoria principală suportă și procesele sistemelor de operare și pe cele ale utilizatorilor
- Avem resurse limitate, deci trebuie să le alocăm eficient
- Contiguous allocation este una din primele metode
- Memoria principală este împărțită obișnuit în două secțiuni:
 - Resident operating system, de obicei situat la început în memoria și cu vectorul de interrupții
 - Procesele utilizatorului sunt situate în spatele secțiunii sistemului
 - În fiecare proces este continut într-o singură secțiune contiguă a memoriei
- Registrile de relocație sunt folosite pentru a proteja procesele utilizatorului între ele și schimbarea adreselor sistemelor de operare
 - Registrul base conține valoarea celei mai mici adrese fizice
 - Registrul limită conține un interval de adrese logice, fiecare adresa logică trebuie să fie mai mică decât registrul limită
 - MMU mapează adresele logice în mod dinamic
 - Poate să permită cărui cod kernel să fie temporar și kernel să schimbe dimensiunea



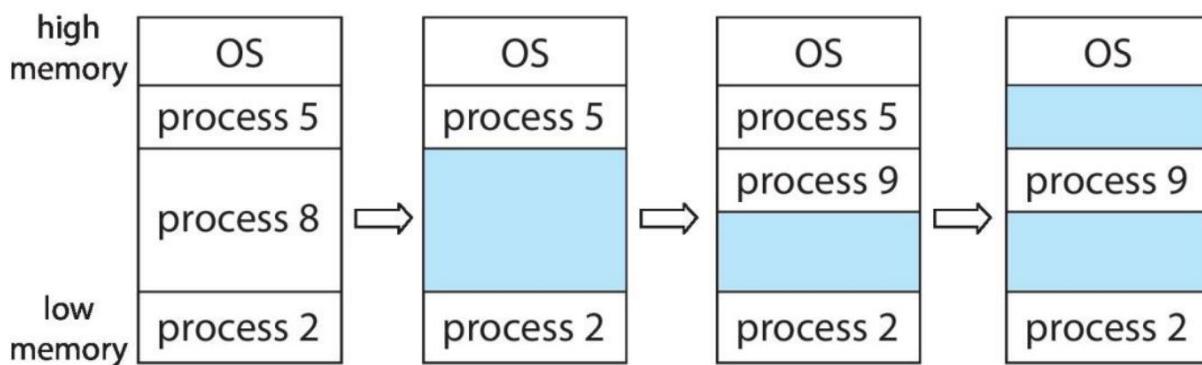
Alocarea memoriei cu partitii fixe

- Memoria principală este împărțită într-un număr fix de partiții
- În fiecare partiție este situat un singur proces => gradul de multiprogramare este limitat de numărul de partiții
- Procesele care sosesc în sistem sunt puse într-o coadă de așteptare pentru partiții libere; partiția trebuie să fie suficient de mare ca să conțină programul
- Odată disponibilă, programul se încarcă în partiție și se execută

- La terminare, partitia se elibereaza si devine disponibila pentru alt proces
- Problema: dimensiunea fixa a partitiilor implica existenta unui spatiu neutilizat (“pierdut”) de catre proces pe durata rularii (“fragmentare interna”)
 - Solutia 1: Cozi separate pentru fiecare partitie (dezavantaj: cozi de partitii mari goale, cozi de partitii mici aglomerate)
 - Solutia 2: Coada unica; se alege cel mai mic proces care incape in partitia eliberata (dezavantaj: discriminarea proceselor mici)

Variable partition

- Alocarea de partitii multiple:
 - Gradul de multiprogramare e limitat de numarul de partitii
 - Variable partition sizes pentru eficienta (Dimensionat dupa nevoile procesului)
 - Hole - block de memorie disponibil, hole uri sunt de marimi diferite peste tot in memorie
 - Cand un proces ajunge, e alocat unui hole destul de mare cat sa intre in el
 - Cand procesele ies din partitie o si elibereaza si mai elibereaza indirect si partitiile adicente
 - Sistemele de operare tin informatie despre allocated partitions si free partitions (hole)



Problema Dynamic Storage-Allocation ului

- Cum putem satisface un request de marime n dintr-o lista de hole uri libere?
 - First-fit : Alocam primul hole destul de mare
 - Best-fit: Alocam cel mai mic hole care e destul de mare; Trebuie sa ne uitam prin toata lista, insa produce si cel mai mic posibil
 - Worst-fit: Alocam cel mai mare hole; Trebuie sa ne uitam prin toata lista si vom produce cel mai mare hole
 - Next-fit: Similar cu first-fit, dar cauta din punctul in care a ramas ultima cautare
 - Quick-fit : mentine liste separate pentru cele mai frecvent cerute dimensiuni de segmente. Foarte rapid, insa nu e folosit foarte des pentru procese, ci e pentru alea chestii cum ar fi pachete de retea Ethernet, acolo unde stim dimensiunea a priori
- First-fit si best-fit sunt mai bune decat worst-fit cu privinta la viteza si folosirea spatiului

Fragmentation

External fragmentation = tot spatiul de memorie existent pentru a satisface un request, dar nu e contiguous

Internal fragmentation = memoria alocata poate sa fie mai mare decat memorie ceruta, astfel ca diferenta este un spatiu nefolosit

- O analiza a lui First-fit demonstreaza ca daca avem N blockuri alocate, $0.5N$ blockuri vor fi pierdute prin fragmentare
 - $\frac{1}{3}$ pot sa devina unusable \rightarrow 50-percent rula
 - Explicatie: presupunem ca sistemul este in echilibru, un proces oarecare in mijlocul memoriei
 - Pe durata executiei $\frac{1}{2}$ din operatii cu segmentele de deasupra lui sunt alocari, $\frac{1}{2}$ dealocari
 - $\frac{1}{2}$ din timp procesul are ca vecin alt proces, $\frac{1}{2}$ din timp are un segment neutilizat
 - Pe medie, $\frac{1}{2}$ din blocurile alocate sunt gauri

Analiza fragmentarii externe

- f = fractiunea de memorie aferenta gaurilor
- s = dimensiunea medie a n procese
- $k * s$ = dimensiunea medie a gaurilor, $k > 0$
- m = dimensiunea memoriei in biti
- ⇒ $n/2$ gauri de memorie ocupa $m - n*s$ biti
- ⇒ $(n/2)*k*s = m - n*s$
- ⇒ $m = n*s*(1 + k / 2)$
- ⇒ $f = ((n / 2) * k*s) / m = k / (k + 2)$
- ⇒ pt $k = \frac{1}{2}$ (dimensiunea gaurii e $\frac{1}{2}$ din dimensiunea medie a procesului)
 $f = 20\% \text{ pierdere de memorie}$
- pt. $k = \frac{1}{4} \Rightarrow f = 11\%$

- External fragmentation ul poate fi redus prin compaction
 - Punem toate bucatile de free memory intr-un singur block mare
 - Compactarea e posibila doar daca relocarea e dinamic si e facuta la execution time
 - Apare o problema la I/O
 - Se blocheaza un process in memorie in timp ce e in I/O
 - Se face I/O numai in bufferele sistemelor de operare

- În general fragmentarea externă se rezolvă prin alocarea necontigua a spațiului logic (virtual) de adrese
 - Solutii complementare: segmentarea si/sau paginarea
 - Paginarea rezolvă în bună măsură problema fragmentării

Paging

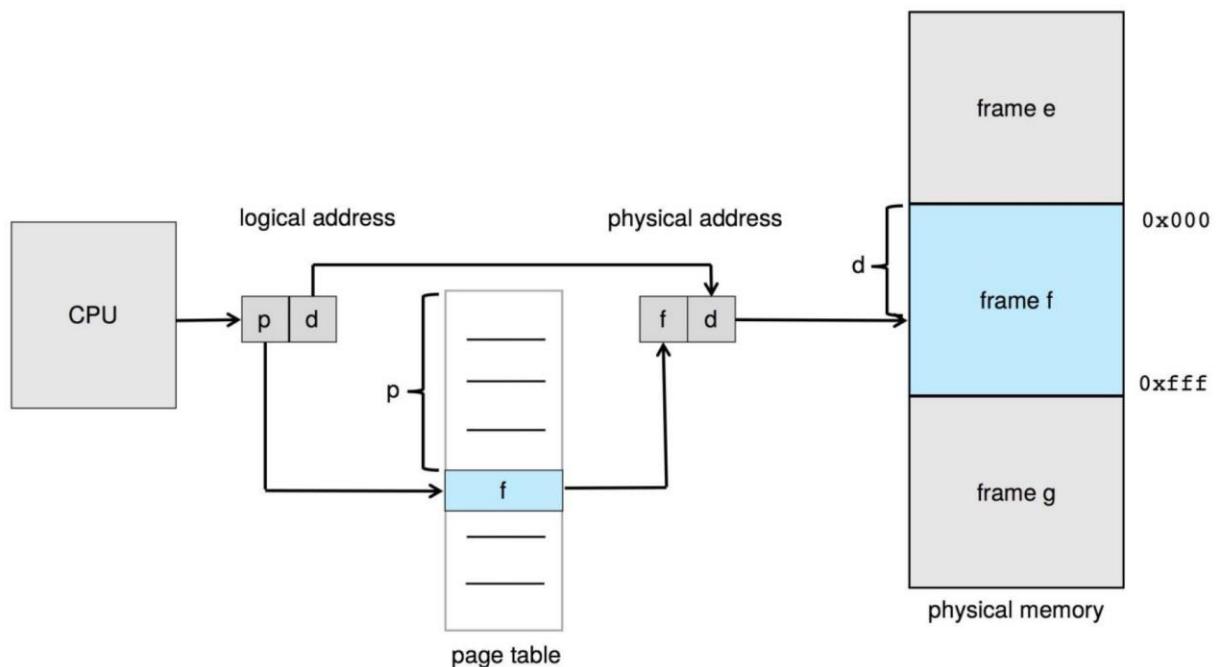
- Spațiul de adrese fizice al unui proces poate să fie noncontiguos (neînvecinat); procesul e alocat atunci și available pentru a evita external fragmentation și pentru a evita bucati de memorie diferite
- Impartim memoria fizică în blockuri de marimi fixe pe care le numim frames. Marimea o să fie o putere a lui 2, între 512 bytes și 16 MBytes
- Impartim memoria logică în blockuri de marimi fixe pe care le numim pages
- Tinem cont de toate frame-urile libere
- Ca să rulăm un program de marime N pages, trebuie să gasim N free frame-uri care să încarce programul
- Vom seta un page table care să traduca din adrese logice în adrese fizice
- Backing store-ul trebuie să asemenea să fie împărțit în pages
- Însă tot o să avem fragmentare internă
- La încarcarea programelor în memorie nu e nevoie de relocare
 - Fiecare proces are tabela de pagini proprii spațiului sau de adrese virtual
 - Spațiul de adrese virtual e contiguu (apropiat), alocarea efectivă de memorie fizică nu e contiguă
- Permite implementarea sistemelor de memorie virtuală
 - Spațiul virtual de adresa e mai mare decât spațiul de adrese fizice (adică memoria calculatorului)
 - Ex: spațiu de adresa pe 64 de biti (2EB, 2048 PB) chiar dacă memoria e semnificativ mai mică
 - O parte din pagini sunt rezidente în memorie, restul se află pe disc
- Se potriveste multiprogramarii
 - Accesul la paginile nerezidente în memorie (page-fault) determină oprirea procesului până la încarcarea paginii de pe disc în memorie, prilej ideal pentru a oferi procesorul altui proces

Address Translation Scheme

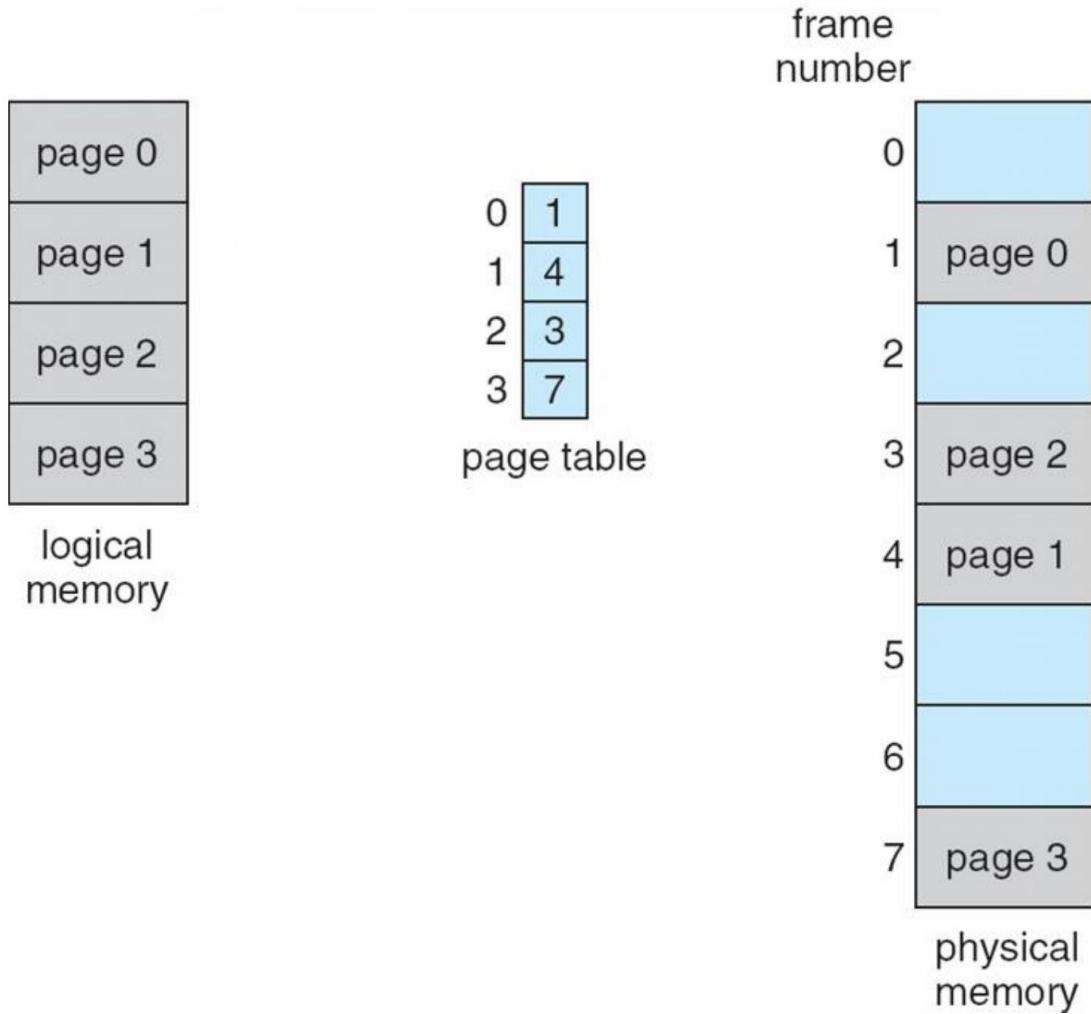
- Adresele generate de CPU sunt împărțite în :
 - Page number (p) - folosit ca și index într-un page table care conține adresele de bază dintr-un frame f în memoria fizică
 - Page offset (d) - combinat cu baza adresei definesc adresa fizică de memorie trimisă către memory unit
 - Perspectiva alternativă, analogie cu relocarea: f este base register, iar limita este data de dimensiunea paginii/frame ului

page number	page offset
p	d
m -n	n

- Pentru un spatiu de adrese logice de 2^m si page size 2^n

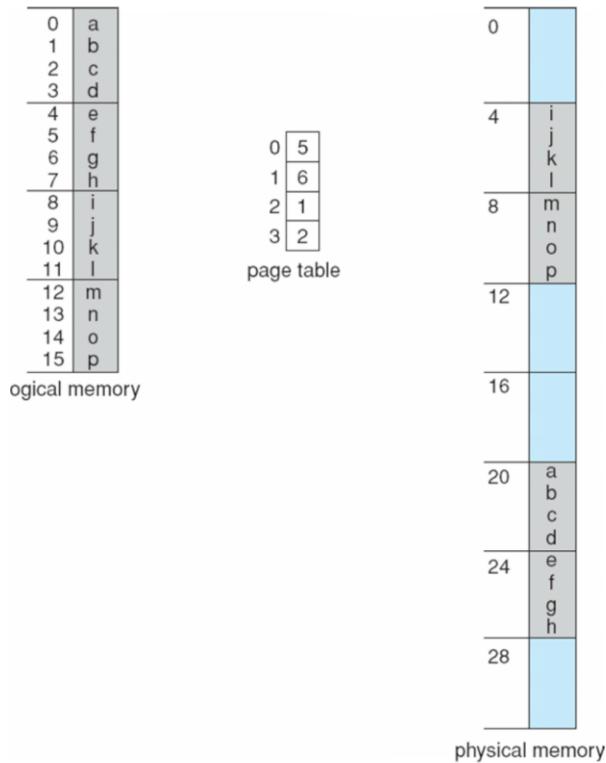


Paging model



Paging example

- Adrese logice: n=2 si m=4. Folosim un page size de 4 bytes si o adresa fizica de 32 de bytes (8 pages)



Paging – Calculating internal fragmentation

- Paginarea exclude fragmentarea externa (orice frame liber poate fi alocat unui proces care are nevoie de el)
- În schimb, nu elimină complet fragmentarea internă, de ex:
 - Page size = 2048
 - Process size = 72,766 bytes
 - Înseamnă că o să ne folosim de 36 de pagini, dar ultima nu o să fie toată folosită
 - Fragmentarea internă = $2048 - \text{cat folosim} (\text{adică } 1086 \text{ bytes}) = 962 \text{ bytes}$
- Worst case fragmentation: 1 frame - 1 byte
- On average fragmentation: $\frac{1}{2}$ frame size

Analiza optimalității dimensiunii paginii

- Q: Care e dimensiunea optimă a paginii?
- Pagini mici => fragmentare internă mică, dar tabela de pagini mare (cu multe intrări)
- Pagini mari => Tabela de pagini mai mică, dar și I/O mai eficient cu discul
- Analiza sumară:
 - Presupunem că s = dimensiunea medie a unui proces și p = dimensiunea paginii în bytes
 - Adică ne vor trebui s/p pagini per proces și s^*e / p bytes per tabelă de pagini, unde e = numărul de bytes din PTE (Page Table Entry)
 - În medie, spațiul de memorie pierdut din cauza fragmentării interne a ultimei pagini este $p/2$, adică vom avea overhead egal cu $s^*e / p + p/2$

- obs: cei doi termeni ai overheadului evolueaza contrar in functie de valoarea lui p
 - creste p , scade dimensiunea tableei de pagini si creste fragmentarea interna
 - scade p , scade fragmentarea interna si creste dimensiunea tableei de pagini
- pt a determina valoarea optima a lui p , derivam si rezolvam ecuatia

$$-\frac{s \cdot e}{p^2} + \frac{1}{2} = 0$$

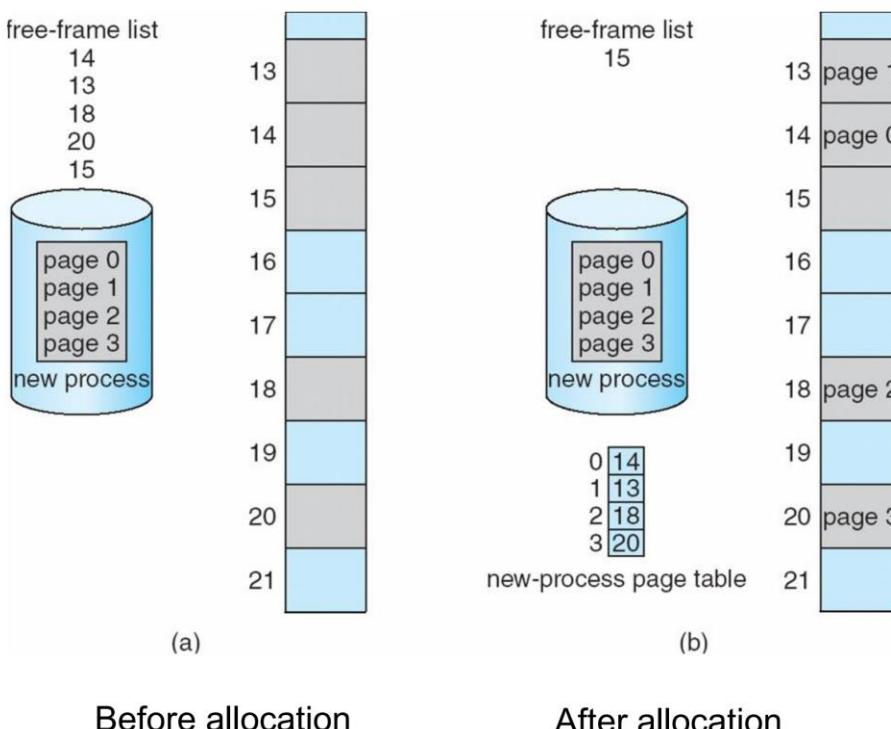
$$\Rightarrow p = \sqrt{2se}$$

- ex:

$$s = 512 \text{ KB si } e = 4 \text{ bytes} \Rightarrow p = \sqrt{2 * 2^2 * 2^{19}} = 2^{11} = 2KB$$

- in general, se iau in considerare si alti factori (eg, viteza discului) pt stabilirea dimensiunii optime a paginii

Free frames



Before allocation

After allocation

Idei principale ale paginarii

- Separare clara intre perspectiva programatorului asupra memoriei si memoria fizica
 - Programatorul vede memoria ca un spatiu contiguu(invecinat) care contine programul
 - In realitate, programul este “imprastiat in memoria fizica, paginile sale amestecandu-se cu paginile altor programe
- Diferenta de perspectiva e gestionata de MMU care mapeaza adresele logice/virtuale in adrese fizice (pagini -> frameuri)
- Aceasta mapare este ascunsa utilizatorului de catre kernel
- Pentru ca tabela de pagini e per proces, acesta nu poate accesa alte pagini decat paginile sale
- Sistemul de operare trebuie sa mentina contabilitatea frame urilor alocate si respectiv a celor libere, in general se foloseste o tabela de frame uri
- Sistemul de operare mentine cate o tabela de pagini pentru fiecare proces ca parte a PCB ului
- Aceasta se incarca in structurile de suport hardware pentru paginare ale procesorului (MMU) cand procesorul este ales pentru a rula de catre planificator
- In consecinta, paginarea afecteaza timpul de context-switch

Implementation of Page Table

- Page table ul e pastrat in memoria principală cu ajutorul
 - Page-table base register (PTBR) care pointueste catre page table
 - Page-table length register (PTLR) care indica marimea page table ului
- In aceasta schema, tot accesul la datele si instructiuni necesita doua accesari in memorie. Unul e pentru page table si unu pentru data/instruction
- Problema nevoii de a accesa de 2 ori memorie poate fi rezolvata folosind un special fast-lookup din memoria cache numit translation look-aside buffers (TLB) (numit si associative memory)

TLB

- Cache de mare viteza din MMU care mentine copiile celor mai des folosite intrari in tabela de pagini (PTEs)
- O intrare in TLB contine numarul de pagina si intrarea corespunzatoare din tabela de pagini si un bit de validitate care specifica daca intrarea e in uz sau nu
- TLB nu doar ca e mai rapid ca memoria principală, dar difera si ca functionare
 - Intrările in TLB sunt referite nu prin adresa ci prin continut
 - Fie pg numarul paginii asociate de intrare in TLB cu pt, intrarea din tabela de pagini
 - TLB ul compara in paralel pg cu toate numerele de pagina din toate intrările sale
 - Daca se gaseste o potrivire, valoarea pt corespunzatoare se foloseste de catre MMU pentru mapare

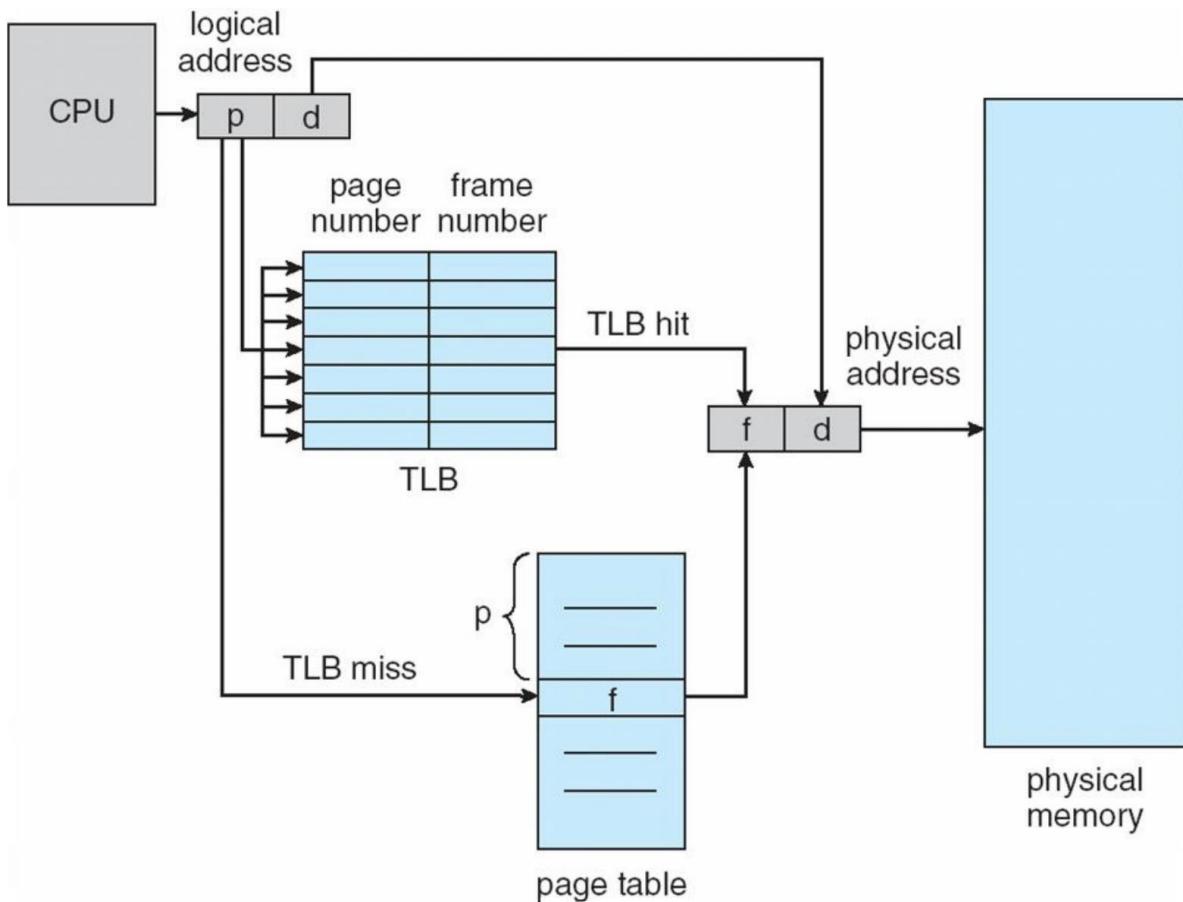
TLB miss

- Daca nu se gaseste nici o potrivire in TLB (TLB miss):

- MMU cauta in tabela de pagini din memoria principală o mapare valida și dacă o gaseste, înlocuiește o intrare din TLB cu noua mapare
 - Dacă intrarea înlocuită reprezintă eventual o pagina modificată, MMU marchează corespunzător intrarea din tabela de pagini
- Consecințele multiprogramării pentru operarea TLB-ului
 - La context-switch se schimbă tabelele de pagini => mapările din TLB nu mai sunt valide
 - Solutii posibile
 - Invalidarea întregului TLB, cu consecințe dramatice pentru performanța sistemului
 - Folosirea PID-ului la indexarea în TLB

Caracteristici TLB

- Unele TLB-uri stochează address-space identifiers (ASIDs) în fiecare TLB entry - ceea ce ajută la identificarea fiecarui proces care oferă address-space protection pentru acel proces, altfel ar fi nevoie de un flush la fiecare context switch
- TLB-urile sunt de obicei mici (64 până la 1024 de entry-uri)
- La un TLB miss, valoarea este încarcată în TLB pentru a o putea accesa mai rapid data viitoare, însă trebuie efectuate niste replacement policies și unele entry-uri ar putea fi fixate ca să nu fie înlocuite din TLB
- În procesoarele moderne, TLB face parte din instruction pipeline
- Unele procesoare au TLB separate pentru instrucțiuni și date



Effective Access Time

Hit ratio = procentul de cate ori un page number e gasit in TLB

- Un 80% hit ratio inseamna ca o sa gasim in TLB page number ul in 80% din cazuri
- Sa presupunem ca ne ia 10 nanosecunde sa accesam memoria
 - Daca gasim pagina in TLB si apoi mapata ne ia 10ns
 - Altfel, o sa trebuiasca sa accesam de doua ori memoria, deci o sa dureze 20 nanosecunde
- Effective Access Time (EAT)

$$EAT = 0.80 * 10 + 0.20 * 20 = 12$$
, ceea ce implica un slowdown de 20% la accesul in memorie
- Sa consideram un hit ratio mai realist de 99%

$$EAT = 0.99 * 10 + 0.01 * 20 = 10.1$$
 nanosecunde, ceea ce implica un slowdown de 1%

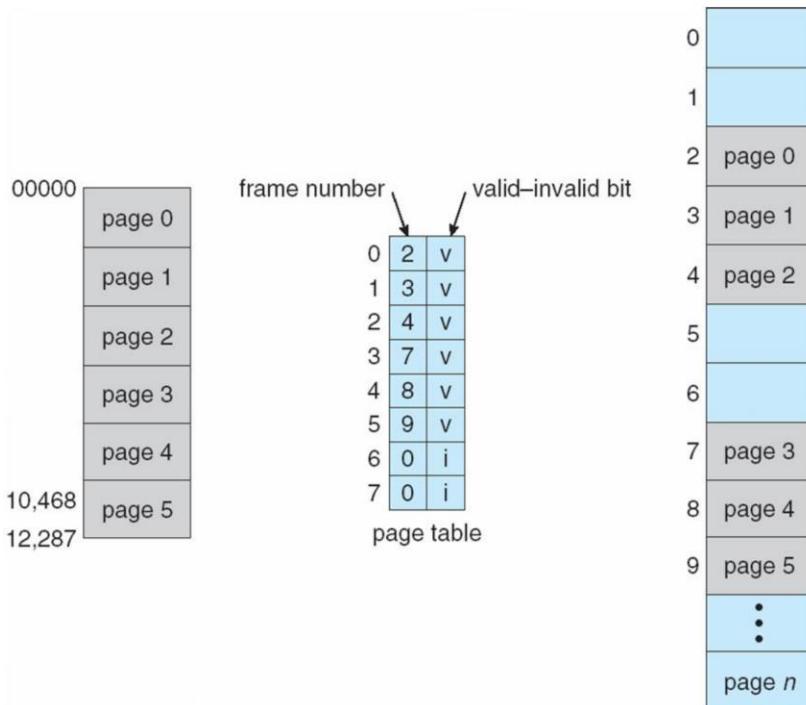
Informatie aditionala in PTE

- Present bit P
 - 1 daca frame ul corespunzator exista in memorie, 0 altfel

- Fiecare acces la memorie generat de proces trece prin MMU
 - P=1 in PTE ul corespunzator accesului => acces permis
 - P=0, MMU genereaza o exceptie numita page fault; kernelul trateaza aceasta exceptie aducand pagina corespunzatoare de pe disc si o incarca intr-un frame liber pe care il noteaza in PTE
- Astfel, doar o parte din proces e in memorie, restul e pe disc
- Referenced bit R - 1 daca pagina a fost referita (citita/scrisa)
- Modified bit M - 1 daca pagina a fost scrisa
- Permisii RWX - specifica daca pagina poate fi citita, scrisa, executata
- OBS: acesti biti ocupa spatiu in PTE => raman mai putini biti disponibili pentru adresare, adica se reduce spatiul de adresare virtuala

Memory protection

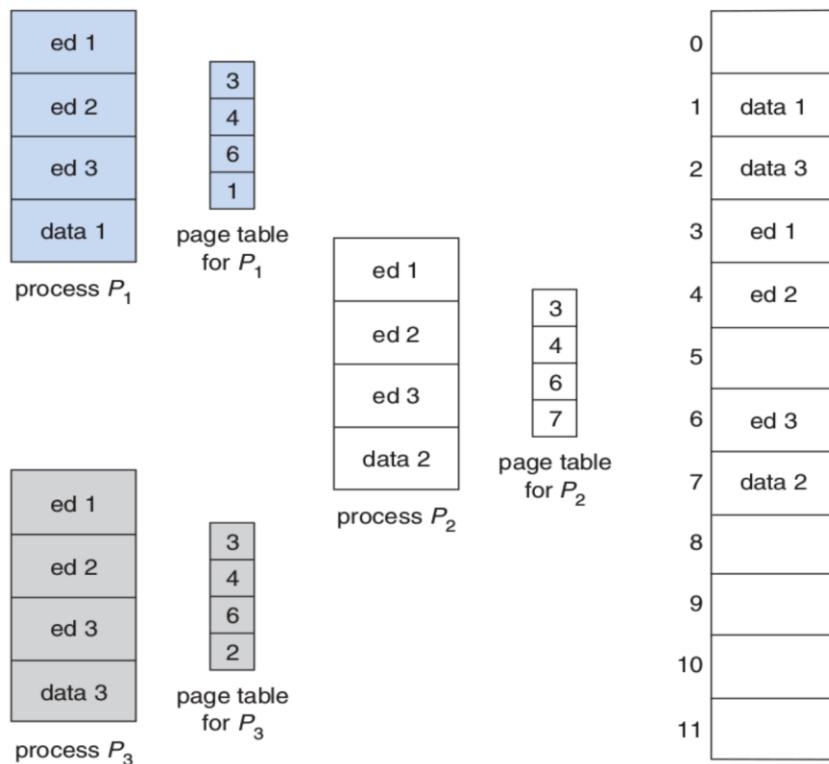
- Memory protection este implementata prin asocierea unui protection bit la fiecare frame ca sa indice ca e read-only sau read-write
 - Poate totodata sa adauge mai multi biti care sa indice ca e doar de executie si tot asa mai departe
- Valid-invalid bit atasat la fiecare intrare din page table:
 - Valid indica faptul ca pagina e in spatiul adreselor logice ale procesul, deci e un page valid
 - Invalid indica faptul ca pagina NU e in spatiul adreselor logice ale procesului
 - Sau se poate utiliza page-table length register (PTLR)



Shared pages

- Shared code
 - O singura copie read-only (reentrant) al unui cod impartasit intre procesoare

- Similar cu multiple threads impartind acelasi process space
- Totodata, e folositor si pentru comunicarea interprocese daca shareuirea de read-write pages e permisa
- Private code and data
 - Fiecare proces pastreaza o copie de cod si data
 - Paginile pentru codul si datele private pot aparea oriunde in spatiul adreselor logice
- Exemplu de shared pages



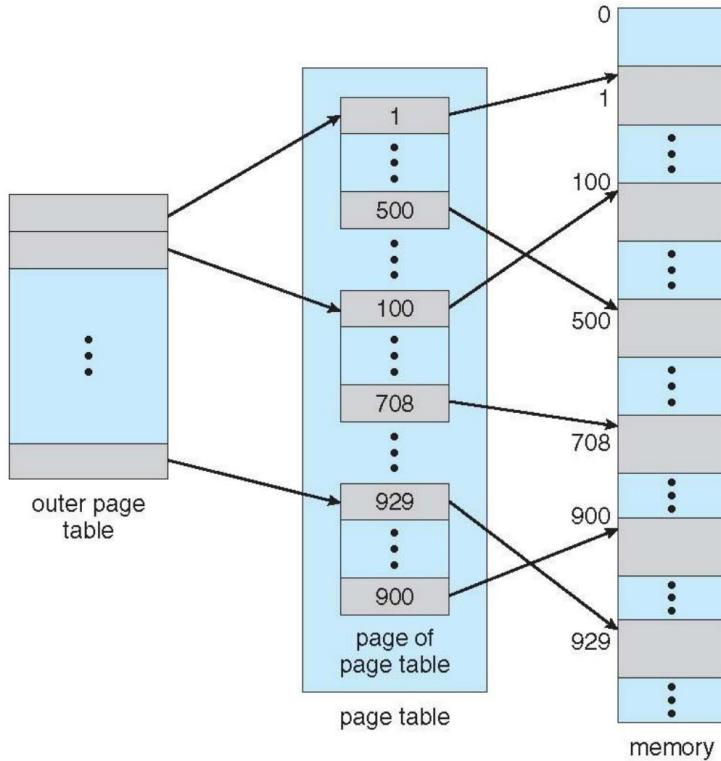
Structura Page Table urilor

- Structurile de memorie pentru paging pot deveni imense daca se folosesc metode directe
 - Consideram un spatiu logic de adrese de 32 de biti
 - Page size de 4KB (2^{12})
 - Page table poate avea 1 milion entries ($2^{32} / 2^{12}$)
 - Daca fiecare entry are 4KB \Rightarrow fiecare proces va avea 4MB din spatiul de adrese fizice doar pentru page table
 - O solutie simpla poate fi sa impartim page table ul in unitati mai mici:
 - Hierarchical Paging
 - Hashed Page Tables
 - Inverted Page Tables

Hierarchical Page Tables

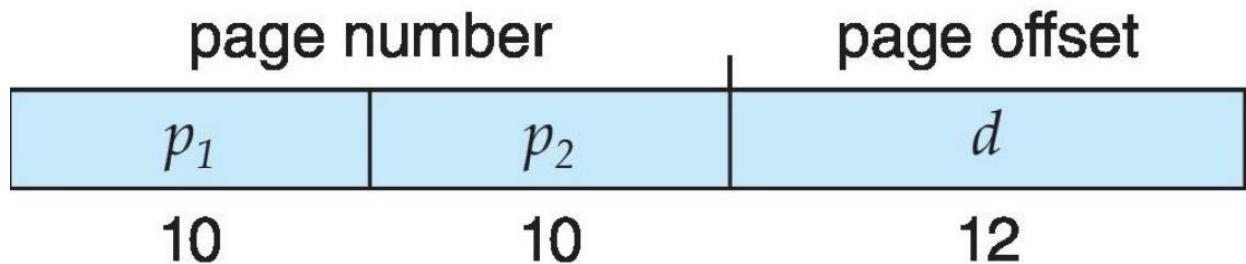
- Sparge spatiu de adrese logice in mai multe page table uri

- O tehnica simpla este un two-level page table
- Apoi vom parage ui page table ul

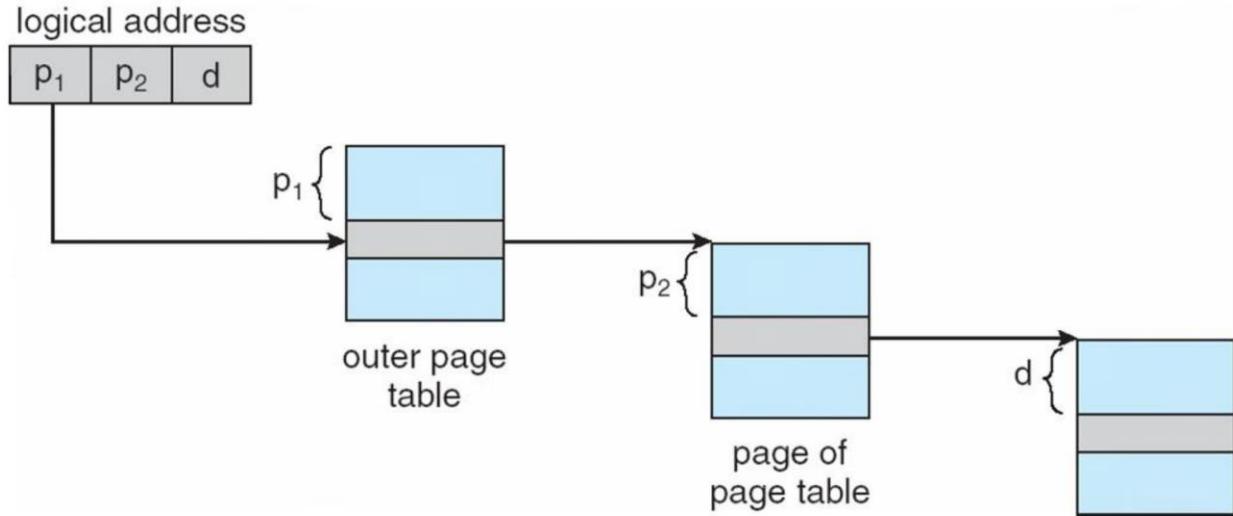


Two-level paging example

- O adresa logica (de 32 de biti) e impartita in:
 - 20 de biti page number ul
 - 12 biti page offset ul
- Cum page table ul e pageuit acum, page number ul mai este impartit si el in:
 - 10 biti page number
 - 10 biti page offset
- Deci, o sa avem urmatoarea adresa logica

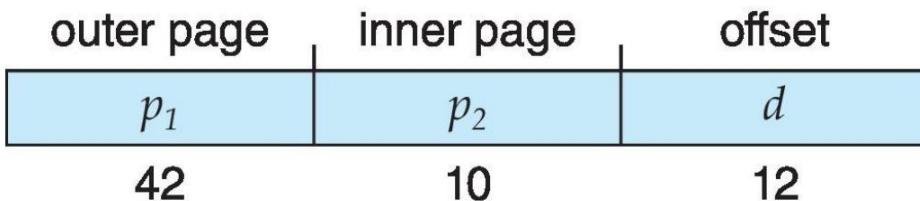


- Unde P_1 e un index catre un outer page table si p_2 este deplasarea din inner page table
- Mai este cunoscut si ca forward-amped page table

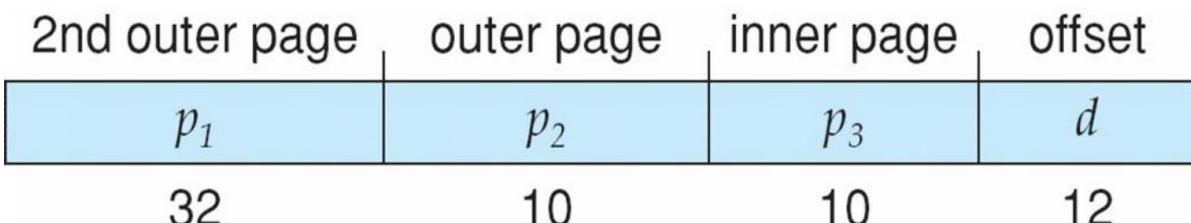


64-bit Logical Address Space

- Nici chiar un two-level nu e de ajuns
- Daca page size ul e 4KB
 - Atunci avem page table de 2^{52} entry uri
 - Daca am avea un 2-level scheme, inner page table uri ar putea avea 2^{10} 4-byte entries
 - Si adresele ar arata ceva de genu



- Outer page table are 2^{42} entryuri sau 2^{44} bytes
- Una dintre solutii ar fi sa mai adaugam un al doilea outer page table
- Insa in urmatorul exemplu in 2nd layer ar fi tot 2^{34} bytes in marime si e posibil sa fie nevoie de chiar 4 memory acces uri pentru a ajunge la o locatie in memorie



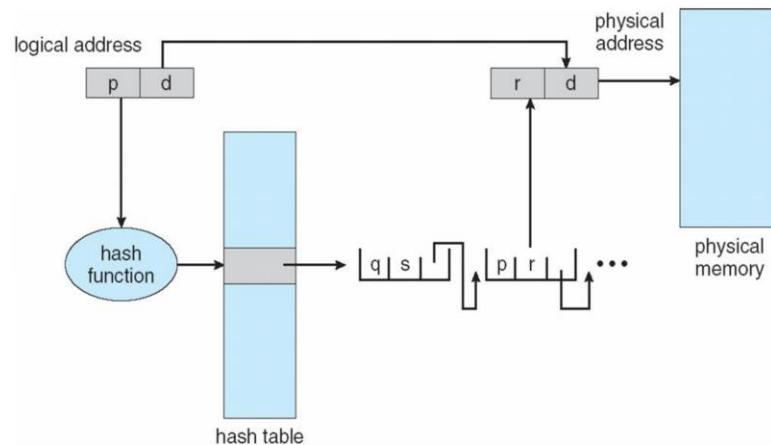
Zero level paging

- Paginare de nivel zero intalnita la unele procesoare RISC (MIPS R3000)
- MIPS R3000 implementeaza doar TLB si nu are hardware dedicat pentru cautarea in tabele de pagini

- La TLB miss se genereaza o exceptie si se da controlul sistemului de operare
- Handlerul de exceptie consulta tabelele de pagini din memorie si incarca noua mapare in TLB dupa ce in prealabil a scos o intrare daca nu era in spatiu. In plus, daca pagina accesata nu era in memorie, executa in prealabil si codul de page fault handler
- Motivatia pentru acest tip de design
 - Simplitatea chip ului
 - Studii de simulare care au calculat TLB miss rate ul si timpul de incarcare a unei intrari in TLB au aratat ca performanta e acceptabila

Hashed Page Tables

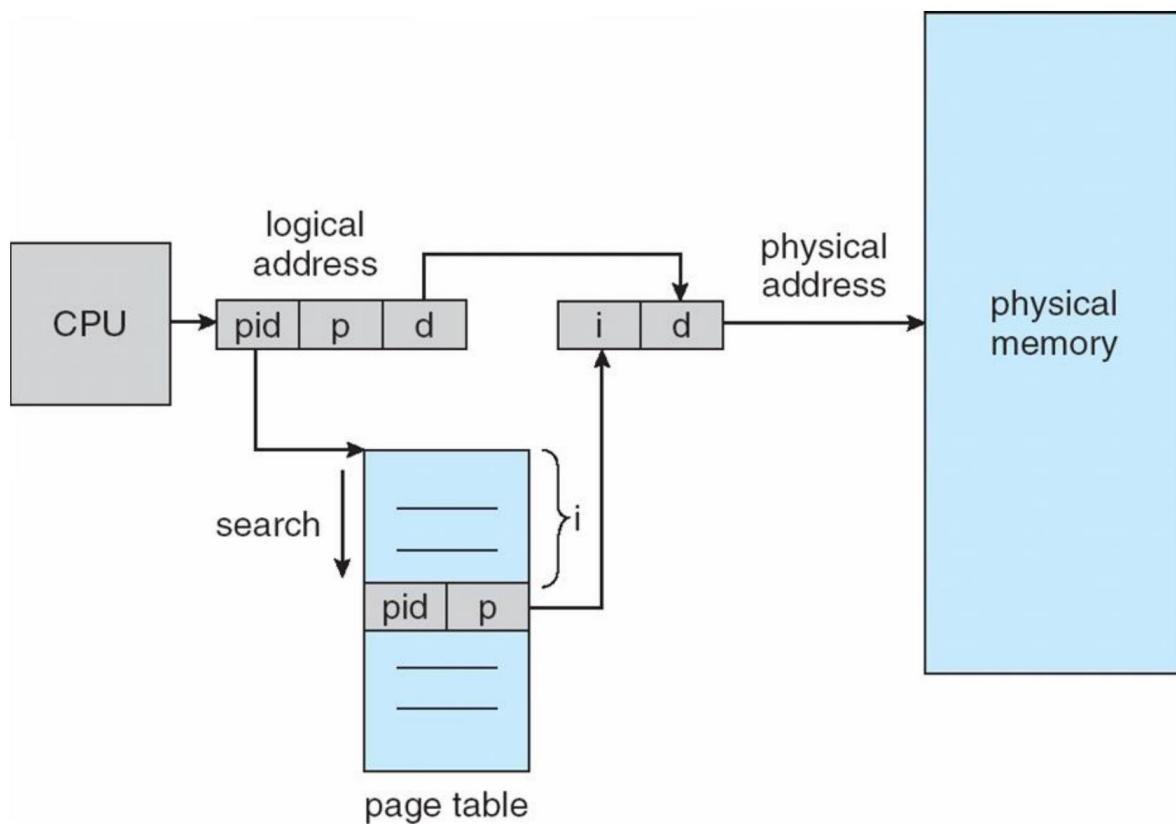
- Se folosete de obicei la adresele cu mai mult de 32 de biti
- Page number ul virtual e hash uit un page table. Acest page table contine un lant de element hash care acceaasi locatie
- Fiecare element contine:
 - 1) Virtual page number ul
 - 2) Valoarea maparii la page frame
 - 3) Un pointer catre urmatorul element
- Virtual page numbers sunt comparate in acest lant cand se cauta un match. Daca se gaseste, atunci se extrage frame ul corespunzator
- Variatia pe 64-bit se numeste clustered page tables
 - Similar to hashed, dar fiecare entry se refera la mai multe pagini (de ex 16) in loc sa se refere la unu singur
 - Foarte bun pentru adresele care nu sunt una langa alta



Inverted page table

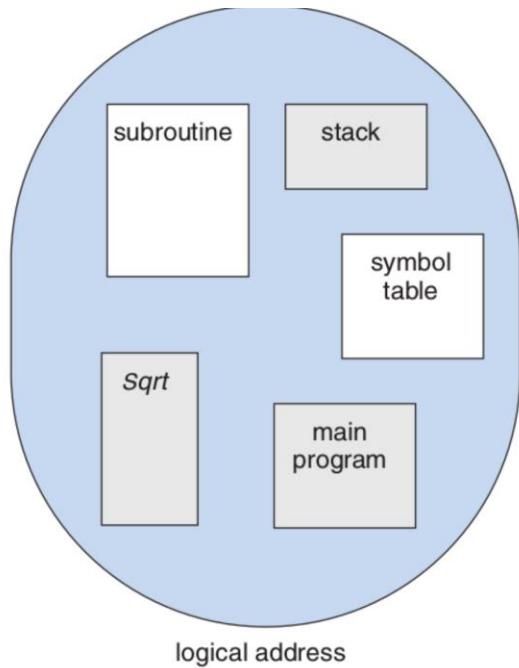
- In loc ca fiecare proces sa aiba un page table si sa tina cont de toate paginile logice posibile, se va uita direct dupa physical pages
- Un singur entry pentru fiecare real page din memorie
- Un entry consta din virtual address ul page ului stocat in acea memorie reala, cu informatii despre procesul care detine page ul
- Scade din memoria necesara stocarii pentru fiecare a unui page table, dar creste timpul necesar pentru a cauta un tabel cand un page reference se intampla

- Folosim hash table pentru a limita cautarea la unu singur sau cateva cel mult
- Referintele la adrese virtuale nemapate in tabela inversata rezulta in page faults



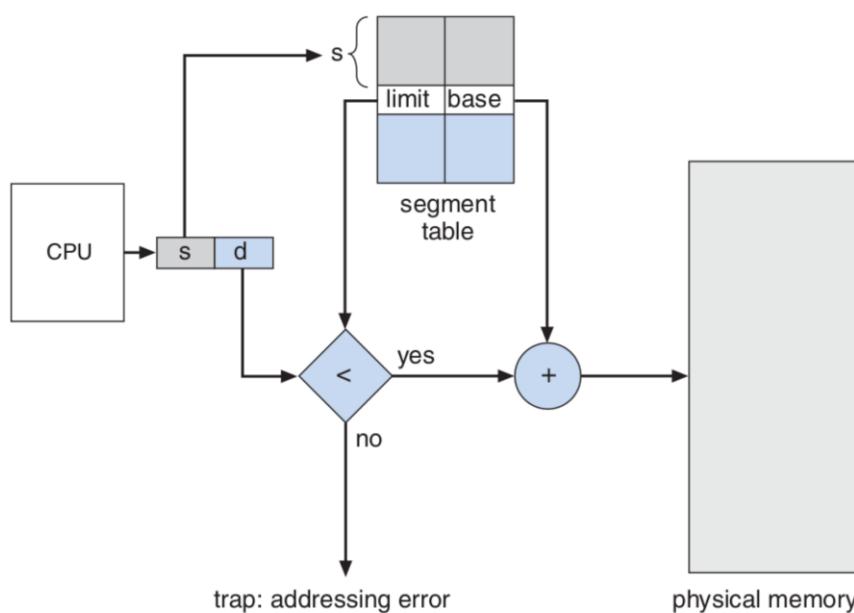
Segmentarea memoriei

- Paginarea ofera un tip de memorie virtuala unidimensională (adica exista un singur spatiu de adrese logice/virtuale)
- Viziunea programatorului asupra programului sau nu e liniara
 - Programul e o colectie de metode, proceduri, functii, structuri de date , etc
 - Colectie de segmente de memorie disparate, de dimensiuni variabile
 - Elementele unui segment sunt identificate prin offsetul fata inceputul segmentului
- Programatorul se refera la stiva, biblioteci, etc fara sa considere ce adrese de memorie ocupa aceste componente ale programului sau
- Spatiul log/virtual de adrese devine o colectie de segmente (ex. cod, variabile globale, heap, stive pentru thread uri, libc, etc)
 - Fiecare segment are un nume si o lungime
 - Adresarea se face prin nume segment + offset
 - Simplificare: adresa logica/virtuala devine <nr segment, offset>

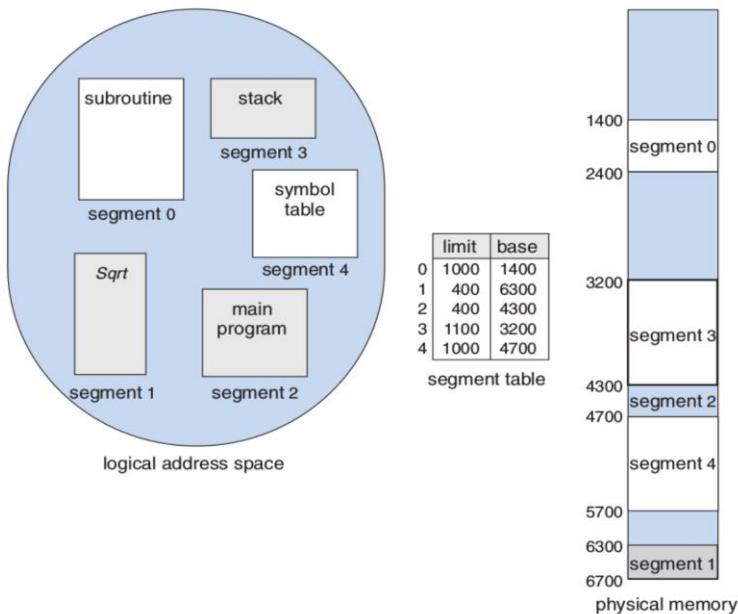


Suport hardware pentru segmentare

- Adresa logica/virtuala bidimensională, dar adresa fizică liniară
- Hardware-ul mapează adresa logică la cea fizică folosind o tabelă de segmente
- Fiecare intrare în tabelă are segment base și limită
- Numărul de segment indexează în tabelă, iar offset-ul din adresa logică trebuie să fie între 0 și limită



Exemplu de segmentare



Avantajele segmentarii

- Componentele logice ale programului pot avea dimensiuni arbitrar care se pot chiar modifica dinamic
- Spatiile aferente acestor componente logice sunt independente unele de altele => nu exista interferente atunci cand dimensiunile lor se modifica dinamic
- Existenta structurarii logice in spatii virtuale distincte faciliteaza partajarea de componentele intre programe
 - Fiecare componenta logica a programului poate fi partajata independent de celelalte

Caracteristici ale segmentarii memoriei

- Fiecare spatiu de adrese virtuale independent se numeste segment = secventa liniara de adrese de la 0 la o valoare maxima
- Segmentele diferite au in mod usual dimensiuni diferite
- Lungimea segmentelor poate varia dinamic
- Programatorul e constient de existenta segmentelor si de regula nu le amesteca
- Partajarea simplificata a componentelor programului prin de ex. Biblioteci partajate
- Linkeditarea simplificata a componentelor unui program:
 - Fiecare procedura are propriul segment, modificarile ulterioare de cod au efect local (nu necesita modificarile altor proceduri)
- Segmentele au protectie individuala, ceea ce ajuta la identificarea rapida a erorilor
 - Segment de cod RO si X, vector RW dar nu X

PAGINARE VS SEGMENTARE

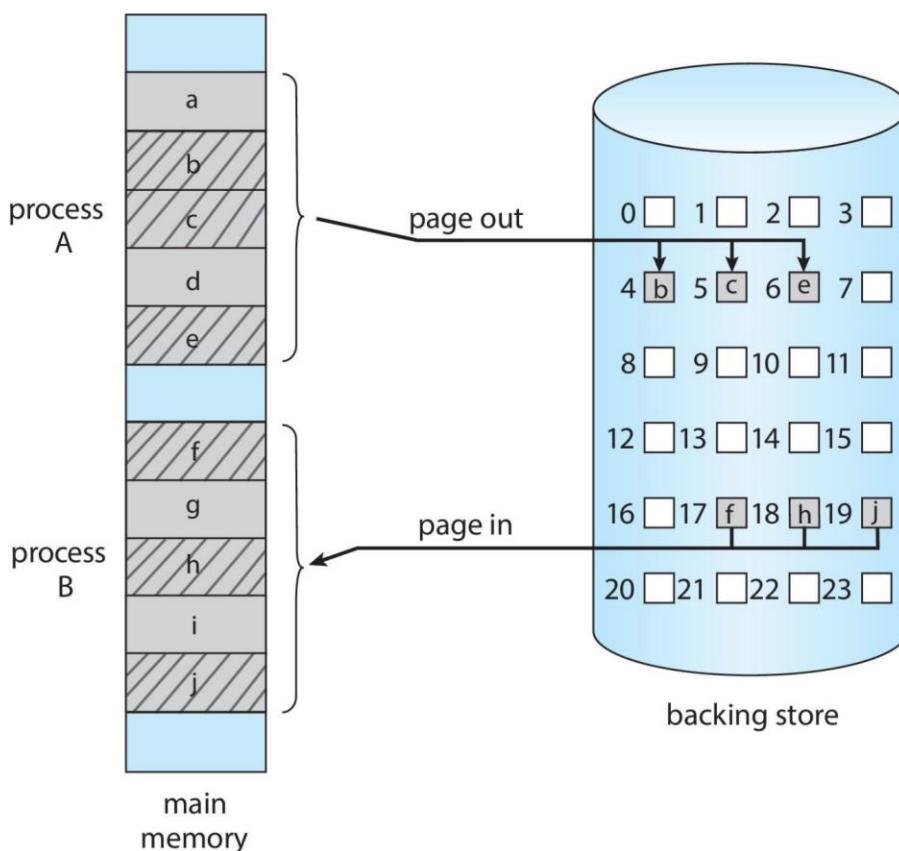
- Programatorul e constient de existenta segmentelor, dar nu si a paginarii
- Paginarea are loc intr-un singur spatiu de adrese virtuale
- In ambele cazuri, spatiul de adrese virtuale poate depasi dimensiunea memoriei fizice
- Paginarea nu distinge intre continutul paginilor => nu poate asigura protectie specifica

- Paginarea nu faciliteaza partajarea componentelor logice ale programului in niciun fel

Concluzii segmentare

- Paginarea e transparenta pentru programator si foloseste in principal pentru a putea incarca si rula programe mai mari decat memoria fizica
- Segmentarea ajuta programatorul sa imparta programul in componente logice distincte pe care le aloca in spatii virtuale independente, pe care le poate proteja si partaja cu usurinta
- Limitarile segmentarii pure
 - Segmentele au dimensiuni variabile, apar probleme de la multiprogramarea cu parti variabile, hole-uri, nevoie de compactare, etc
 - Daca segmentele sunt prea mari, e posibil sa nu incapa integral in memorie => apare ideea de a combina paginarea cu segmentarea, mai exact segmentarea cu paginarea

Exemplu de swapping cu paging



De la pagina 78 nu am mai scris

END CHAPTER 9

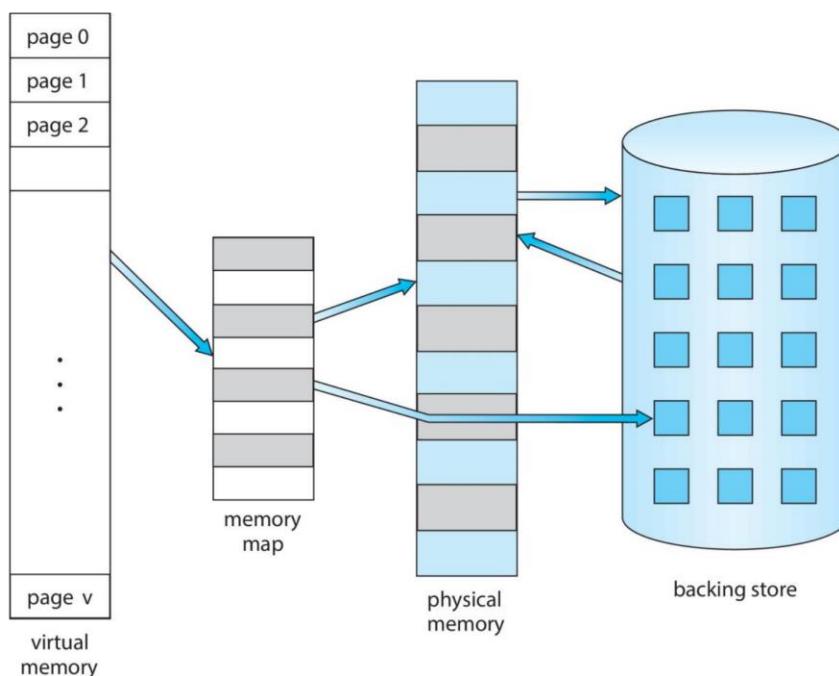
Virtual memory

Memoria virtuală = separarea unei memorii logice a userului fata de memoria fizica

- Numai o parte din program trebuie să fie în memorie la execuție
- Spatiul de adrese logic trebuie să fie astfel mult mai mare decât spațiul fizic de adrese
- Permite spațiilor de adrese să fie shareuite între procese
- Permite crearea proceselor într-un mod mai eficient
- Mai multe programe pot rula simultan
- Mai puțin I/O care trebuie să fie încarcat sau swapped processes

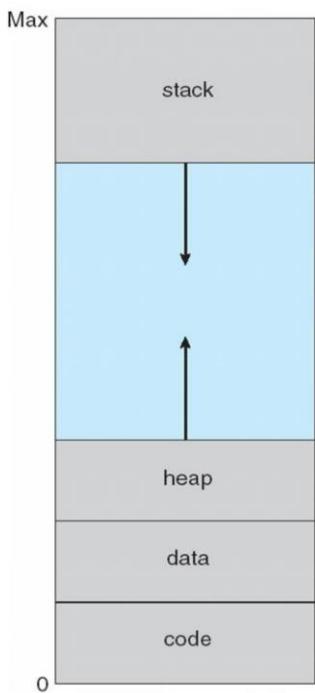
Spatiul de adrese virtuale = perspectiva logică despre cum un proces e stocat în memorie

- De obicei începe de la 0, și puntează o adresa la altă până la final
- În timp ce, memoria fizică este organizată în pagini
- MMU-ul trebuie să le mapeze de la logic la fizic
- Virtual memory poate fi implementat prin
 - Demand paging
 - Demand segmentation

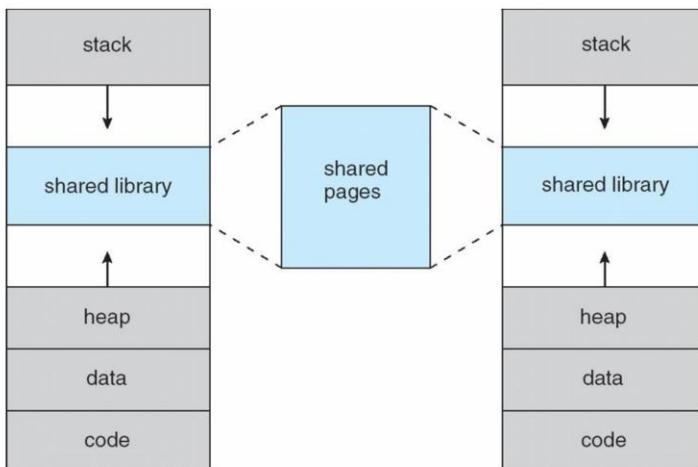


Virtual address space

- De obicei designează spațiul de adrese pentru stack să înceapă la o adresă maximă logică și să crească în jos, în timp ce heap-ul crește în sus
 - Maximiza folosirea spațiului de adrese
 - Adresele nefolosite între cele 2 sunt un hole
 - Nu e nevoie de memorie fizică până când heap-ul sau stack-ul crește la o anumită pagină



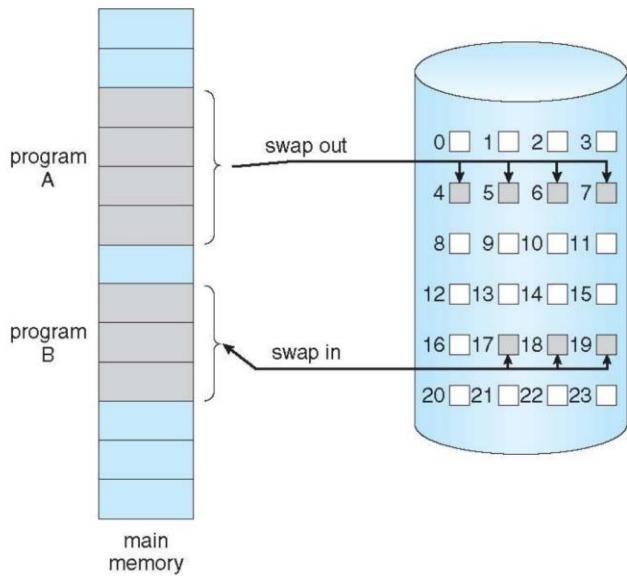
- Permite rarifierea spatiilor de adrese cu hole uri pentru crestere, librarii linkuite dinamic etc.
- Librariile de sistem sunt shareuite prin mapping ul in spatiul virtual de memorie
- Putem shareui si memorie prin maparea unor page uri de read-write pe virtual address space
- Paginile pot fi shareuite in timpul fork() ului, facand acest proces mai rapid



Demand Paging

- Poate sa aduca tot procesul in memorie la load time
- Sau poate sa aduca un page doar atunci cand e nevoie de el si astfel:
 - Avem nevoie de mai putin I/O, fara I/O inutil
 - Ne trebuie mai putina memorie
 - Raspuns mai rapid

- Mai multi useri
- Similar cu sistemul de paging cu swapping



- Avem nevoie de un page => facem o referinta catre el
 - Invalid reference => abort
 - Not-in-memory => bring to memory
- Lazy swapper = niciodata sa nu face swaps pe un page in memorie decat daca e necesara
 - Swapper ul care se ocupa cu paginile se numeste pager

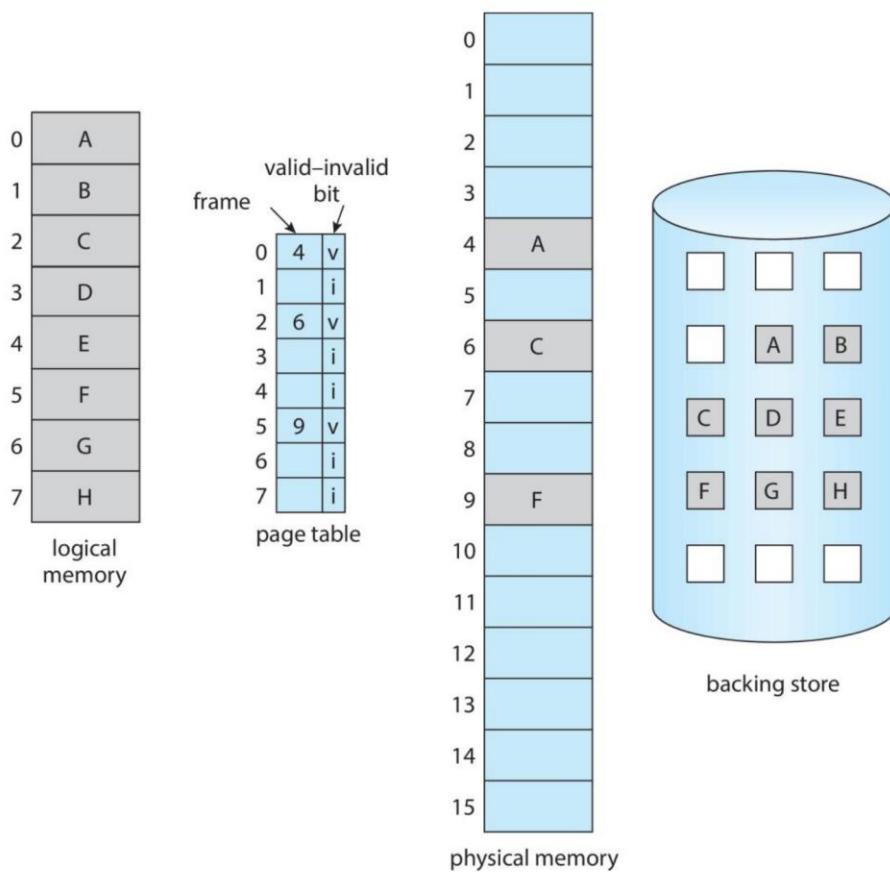
Basic concepts

- Cu swapping ul, pager va ghici care pagini vor fi folosite inainte sa fie schimbate din nou
- In schimb, pager ul le va aduce doar cand e nevoie de ele in memorie
- Cum determinam acest set de pageuri?
 - Avem nevoie de o noua functionalitate a MMU ului care sa implementeze demand paging
- Daca paginile de care e nevoie sunt deja in memorie, atunci nu e nicio diferență față de non demand-paging
- Însă, dacă o pagina de care avem nevoie nu e in memorie, atunci trebuie să o detectăm și să o încarcăm in memorie, fără să schimba comportamentul programului și fără a fi nevoie ca programatorul să schimbe codul

Valid-invalid bit

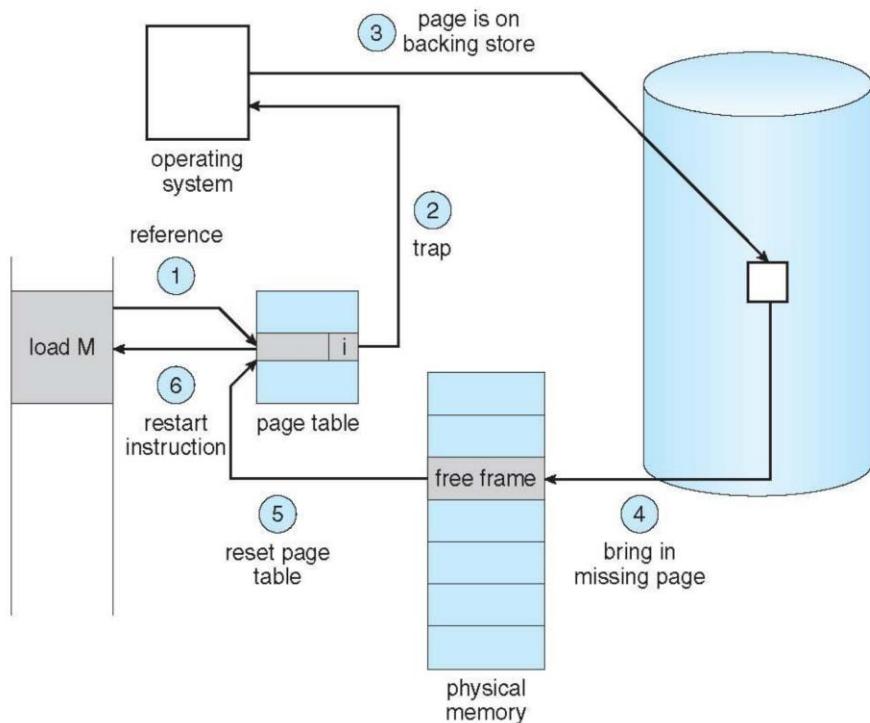
- Cu fiecare page table entry se asociază un valid-invalid bit
 - V => in-memory , adică e în memorie
 - I => not-in-memory
- Initial toate entryurile sunt puse la I

- În timp ce un MMU face address translation, dacă un valid-invalid bit e I , atunci avem page-fault



Pasi de tratare a page faultului

- Dacă e o referință către o pagină, prima referință către pagină va trapui sistemul => Page fault
- Sistemul de operare se uita la alt tabel să decida dacă
 - E invalid reference => abort
 - Sau doar nu e în memorie
- Gasim un frame liber
- Facem swap între page și frame prin scheduled disk operation
- Resetam tabela care să arate și noua pagină în memorie și ii punem la validation bit V
- Restartam instrucțiunea care ne-a dat page fault

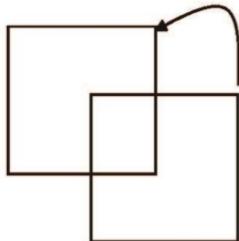


Aspecte ale demand pagingului

- Intr-un caz extrem in care incepem un proces fara page uri in memorie
 - Sistemul de operare seteaza instruction pointerul la prima instructiune care nu e in memorie => page fault
 - Si pentru fiecare proces page va face la fel prima oara
 - Pure demand paging
- In realitate, o instructiune data poate accesa mai multe pagini, adica mai multe page faults
 - Consideram fetch uirea si decodarea a instructiuni care aduna 2 numere din memorie si le stocheaza inapoi in memorie
 - Nu e asa o mare bataie de cap datorita locality of reference
- Hardware suportul necesar pentru demand paging
 - Page table cu valid/invalid bit
 - O memorie secundara (swap device cu swap space)
 - Instruction restart

Instruction restart

- Sa consideram o instructiune care poate accesa diferite locatii
 - Block move



-
- Auto increment/decrement location
- Ce facem , restartam instructiunea?

Solutii pentru restartarea instructiunii

- Depind de la un procesor la altu
- 1) PC ul e copiat intr-un registru intern CPU inaintea executiei fiecarei instructiuni + registrele interne care salveaza incremental/decrementul
- 2) Microcodul CPU stocheaza starea interna pe stiva cand apare page fault ul
- 3) Microcodul CPU are capacitate de roll-back la starea dinaintea executiei instructiunii (presupunem existenta unui checkpoint intern)

Free-frame list

- Cand un page fault se intampla, sistemul de operare trebuie sa aduca page ul de care e nevoie din secondary storage in main memory
- Cele mai mult sisteme de operare mentin un free-frame list, care e o gramasa de frame uri libere care pot satisface astfel de requesturi

head → **7** → **97** → **15** → **126** ... → **75**

-
- Sistemele de operare de obicei aloca free frames prin folosirea unei tehnici cunoscute ca zero-fill-on-demand , unde content ul frame urilor zerod-out inainte de a fi alocat
- Cand un sistem porneste, toata memoria disponibila e pusa pe free-frame list

Stages in Demand Paging - Worse Case

1. Trapuim sistemul de operare
2. Salvam registrii user si starea procesului
3. Determinam faptul ca intreruperea a fost un page fault
4. Verificam ca page reference ul e okay si determinant locatia lui pe disk
5. Citim din disk pe un free frame:
 - a) Asteapta intr-o coada pana cand read request ul se intampla
 - b) Asteapta sa faca seek device ul si/sau latency time ul
 - c) Incepe transferul paginii pe un frame liber
6. In timp ce asteapta, alocam CPU ul altor useri
7. Primește un interrupt de la disk I/O subsystem
8. Salveaza registrii si starea procesului altui user
9. Determina ca interruptul a fost de pe disc

10. Corecteaza page table ul si alte tabele ca sa arate ca page ul e in memorie
11. Asteapta CPU ul sa fie alocat din nou procesului asta
12. Reda registrii user, starea procesului si noul page table si apoi continua instructiunea

Performanca Iui Demand Paging

- Sunt 3 mari activitati
 - Intreruperea - cod facut cu grija, e nevoie de doar cateva sute de linii
 - Citirea page ului - foarte mult timp
 - Restartarea instructiunii - o bucată mică de timp
- Page Fault Rate $0 \leq p \leq 1$
 - Daca $p = 0$ nu avem page faults
 - Daca $p = 1$ toate referintele au fault
- Effective Access Time (EAT)
 - $EAT = (1 - p) \times \text{memory access}$
 - + $P(\text{page fault overhead} + \text{swap page out} + \text{swap page in})$

Exemplu de Demand Paging

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- $$\begin{aligned} EAT &= (1 - p) \times 200 + p(8 \text{ milliseconds}) \\ &= (1 - p \times 200 + p \times 8,000,000) \\ &= 200 + p \times 7,999,800 \end{aligned}$$
- If one access out of 1,000 causes a page fault, then

$$EAT = 8.2 \text{ microseconds.}$$

 This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
 - $$\begin{aligned} 220 &> 200 + 7,999,800 \times p \\ 20 &> 7,999,800 \times p \end{aligned}$$
 - $p < .0000025$
 - < one page fault in every 400,000 memory accesses

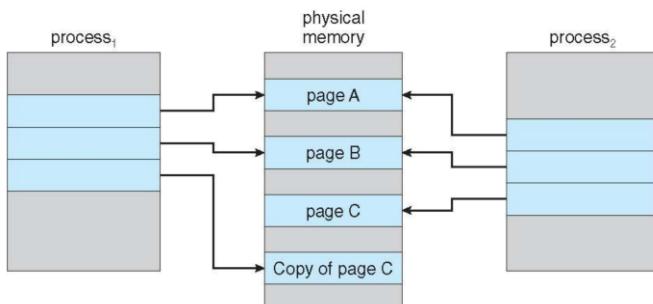
Optimizari pentru Demand Paging

- Swap space I/O faster than file system I/O even if on the same device
 - Swap allocated in larger chunks, less management needed than file system
- Copy entire process image to swap space at process load time
 - Then page in and out of swap space
 - Used in older BSD Unix
- Demand page in from program binary on disk, but discard rather than paging out when freeing frame
 - Used in Solaris and current BSD
 - Still need to write to swap space
 - ▶ Pages not associated with a file (like stack and heap) – **anonymous memory**
 - ▶ Pages modified in memory but not yet written back to the file system
- Mobile systems
 - Typically don't support swapping
 - Instead, demand page from file system and reclaim read-only pages (such as code)

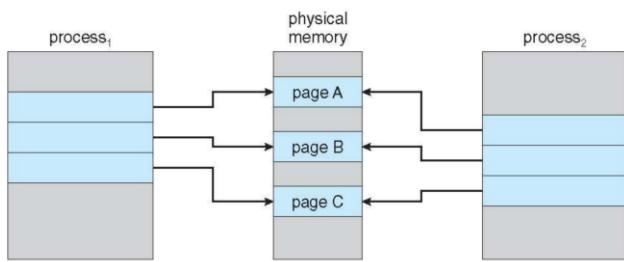
Copy-on-Write

- Copy-on-write (COW) permite si parintelui si copilului sa imparta initial aceleasi pagini in memorie. Daca vreunu din procese modifica o pagina, abia atunci va fi copiata
- COW permite o creearea de procese mult mai eficienta pentru ca vor fi copiate doar page uri modificate
- In general, paginile libere simt alocate dintr-un pool(gramada) de zero-fill-on-demand pages. Pool ul ar trebui sa aiba mereu free frames pentru ca sa aiba demand page execution rapid
- vfork() e o variatie de fork() in care parintele e suspendat in timp ce copilul foloseste copy-on-write address space al parintelui
 - Proiectat sa trebuiasca ca child ul sa foloseasca exec()
 - Foarte eficient

After Process 1 Modifies Page C



Before Process 1 Modifies Page C



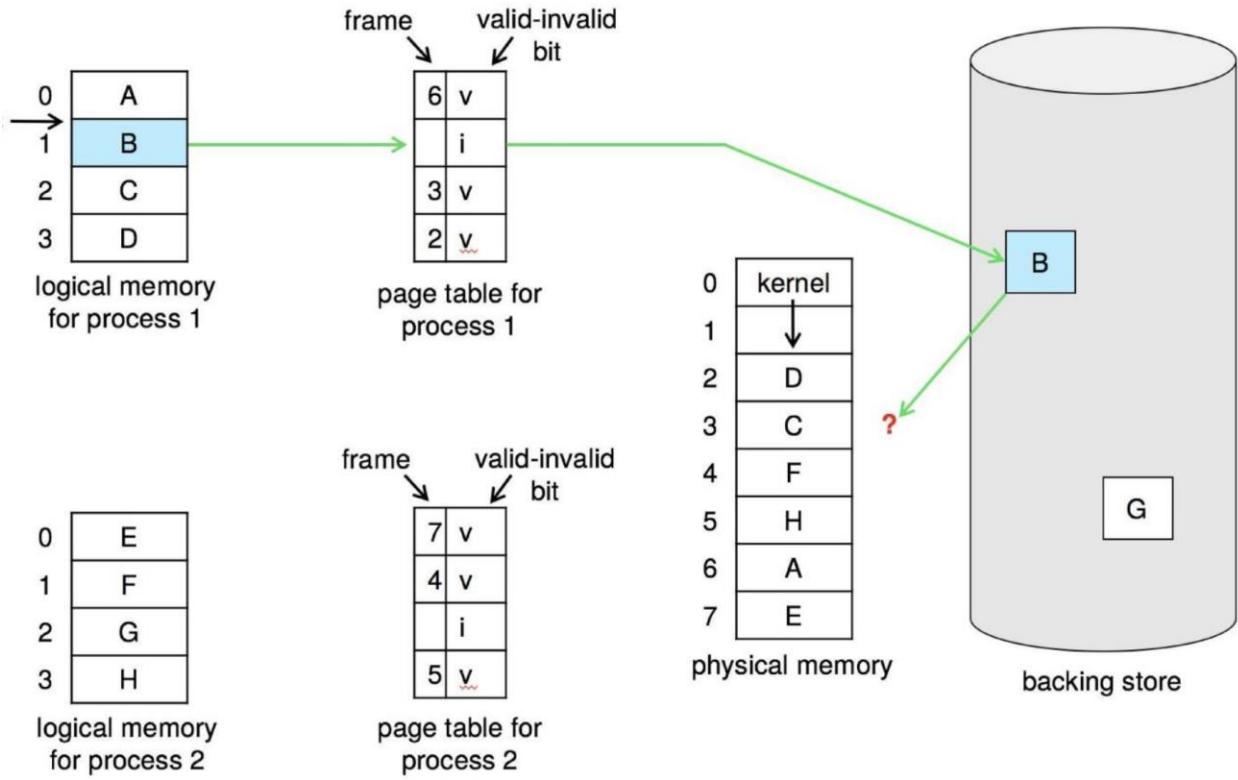
Ce se intampla totusi daca nu exista un free frame?

- Facem page replacement, adica cautam o pagina in memorie care nu e foarte folosita si o scoatem
 - Ce algoritm aplicam? Terminate? Swap out? Replace the page?
 - Vom vrea un algoritm care din care sa rezulte un numar minim de page faults

Page Replacement

- Previne over-allocation ul inafara memoriei modificand page-fault service ul astfel incat sa includa si page replacement
- Folosim modify/dirty bit ca sa reducem din overhead ul de pagini transferate, astfel numai paginile modificate vor fi scrise pe disk
- Page replacementul completeaza separarea intre memoria logica si cea fizica, o memorie virtuala mare poate fi aplicata unei memorii fizice mai mici

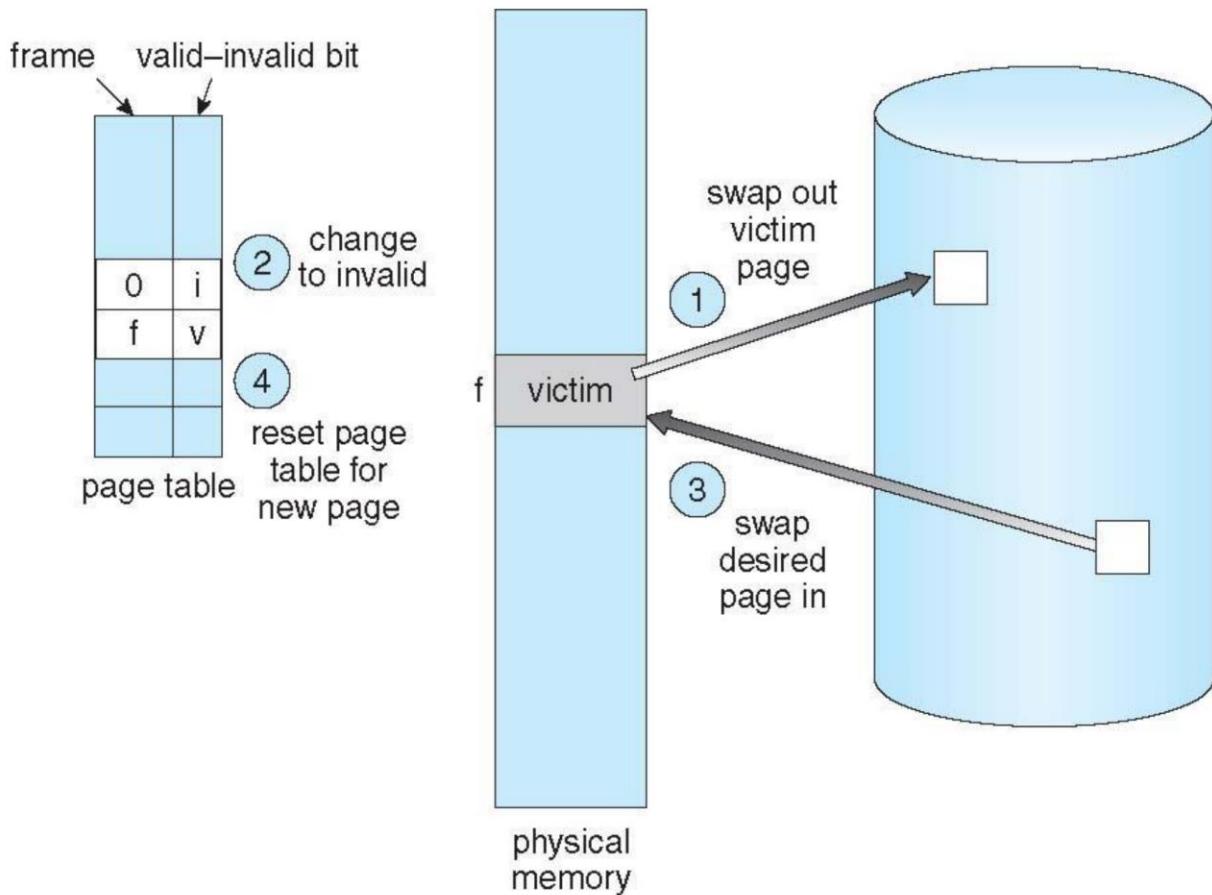
Need For Page Replacement



Basic Page Replacement

1. Gaseste locatia page ului dorit pe disk
2. Gaseste frame ul:
 - a. Daca e vreunul liber, il folosim
 - b. Daca nu avem niciunul liber, folosim un Page Replacement Algorithm sa selectam un victim frame
 - c. Scriem victim frame ul pe disc daca e dirty
3. Aduce page ul dorit pe noul frame liber; updateaza page si frame tables
4. Continua cu procesul de restartuire a instructiunii cauzate de trap

Page Replacement



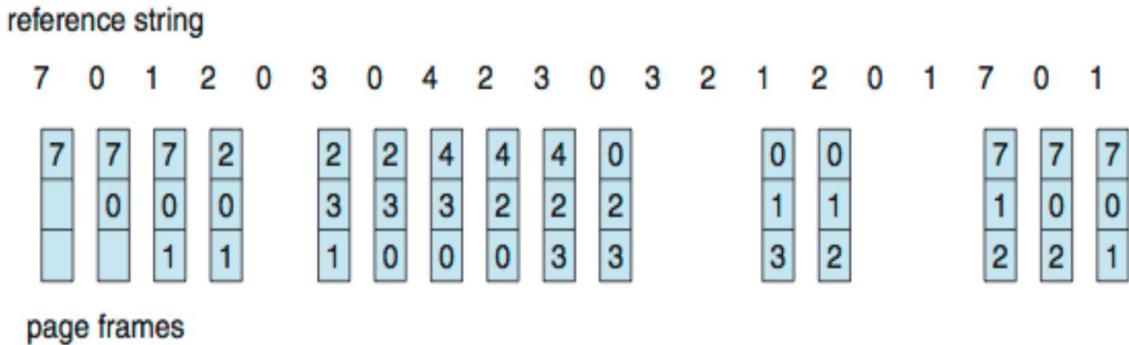
Algoritmi de Page si Frame Replacement

- Frame-allocation algorithms determine:
 - Cate frame-uri sa dea fiecarui proces
 - Care frame-uri sa le inlocuiasca
- Page-replacement algorithm
 - Vrea sa aiba cel mai mic page-fault rate si la prima accesare si la re-access
- In exemplile noastre, vom folosi urmatorul reference string de referenced page numbers

7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1

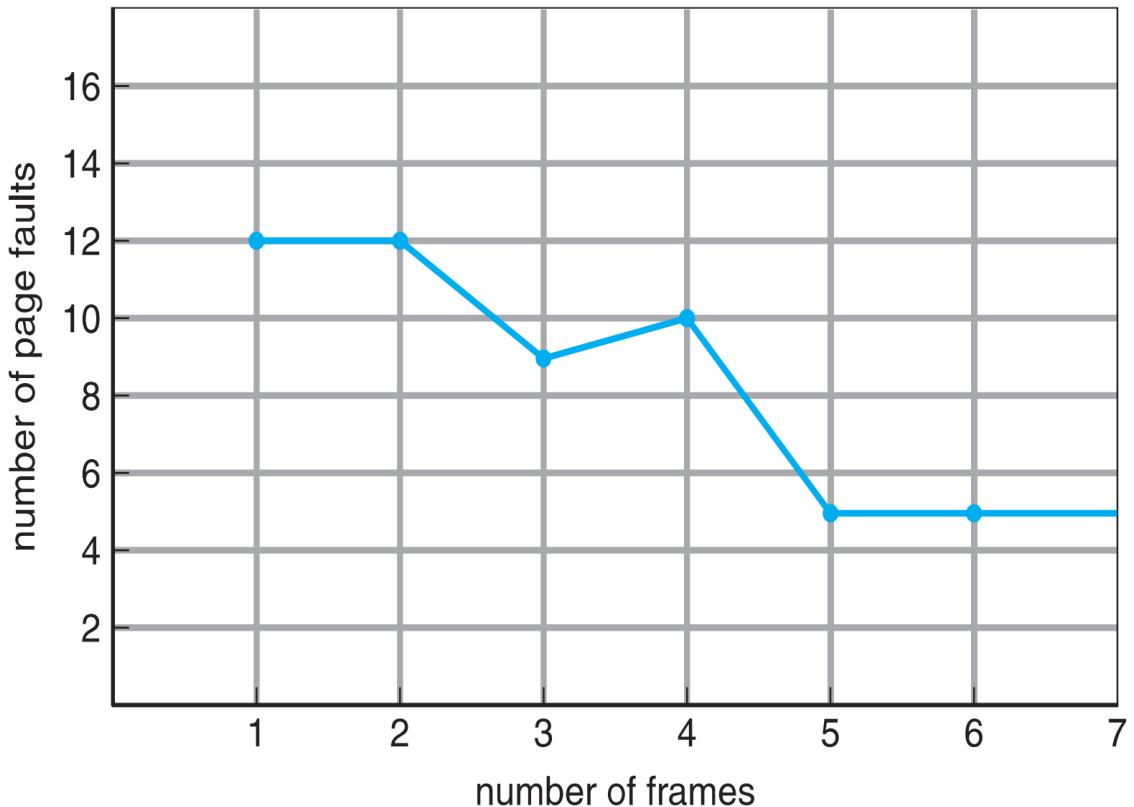
First-In-First-Out (FIFO)

- Consideram ca putem pastra doar 3 frame-uri/ 3 pagini in acelasi timp
- Considerand string-ul de mai sus in conformitate cu FIFO vom obtine:



15 page faults

- Poate varia in functie de reference string.
 - Adaugand mai multe frame uri putem obtine si mai multe page fault uri! Se numeste Anomalia lui Belady



Explicatie:

- FIFO nu e un algoritm de tip stiva, algoritmi pentru care $M(m,r)$ inclus egal in $M(m+1,r)$, unde $M(m,r)$ este multimea frame urilor dintr-o memorie de dimensiune m (numarul total de frame uri) dupa un numar r de referinte la memorie

- La marirea dimensiunii memoriei cu un frame si reexecutarea procesului, la orice moment in timp, toate paginile care erau in memorie dupa referinte in cazul memoriei mai mici sunt si acum in memorie avem in plus o pagina
- Ex algoritmi stiva: OPT(algoritmul optimal), LRU(Least Recently Used)
- Pentru algoritmii stiva e convenabil sa se reprezinte secventa de referinte la memorie prin distanta fata de varful stivei a locului paginii inainte de referinta
- Paginile nereferite inca nu sunt in stiva si au dista infinit

Algoritmi de tip stiva

- Secventa de distante fata de varful stivei nu depinde doar de secventa de referinte de memorie, ci si de algoritmul de paginare
- Proprietatile statistice ale secventei de distante produc performanta algoritmului de paginare
 - Daca densitatea de probabilitate a intrarilor din secventa de distante e "stransa" (adica majoritatea distantele sunt mai mici ca K, unde K e numarul de frame uri) sunt suficiente pentru o rata redusa de page fault uri
 - Daca densitatea de probabilitate e o functie "plata" => singura solutie pentru a evita un numar mare de page fault uri este sa ai tot atatea frame uri cate pagini sunt

Predictia ratei de page-fault uri

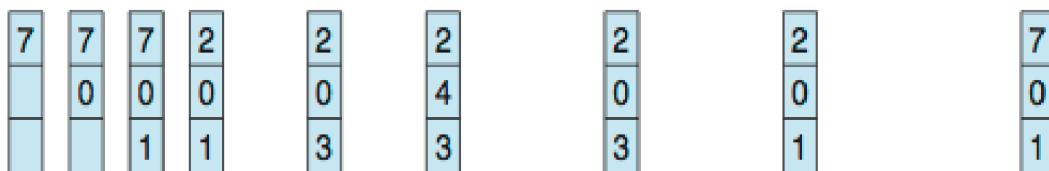
- Secventa de distante poate fi folosita la predictia ratei de page fault uri atunci cand variază numărul de pagini n
- Algoritm:
 - Scanează secvența de distante și calculează:
 - C_i = numărul de apariții ale distantei i în secvență
 - C_{inf} = numărul de apariții ale distantei infinit în secvență
 - Calculează vectorul F , unde F e suma tuturor C_i plus C_{inf} , adică numărul de page fault uri care apar datorită secvenței de distante într-o memorie cu m frame uri

Optimal (OPT) Algorithm

- Idee: schimbăm pagină ul care nu va fi folosit pentru cea mai lungă perioadă de timp
- Din pacate, nu putem prezice viitorul, astfel că OPT este folosit pentru a măsura cât de eficient e algoritmul nostru

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



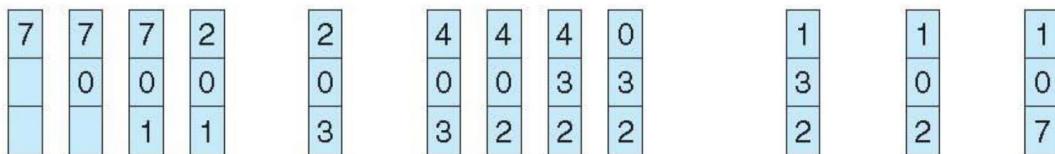
page frames

9 page faulturi

Least Recently Used (LRU) Algorithm

- Idee: Foloseste trecutul si inlocuieste pagina care nu a fost utilizata de cel mai mult timp
 - Se bazeaza pe locality of reference principle
 - Asociaza timpul de la ultima folosire cu fiecare page
- reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

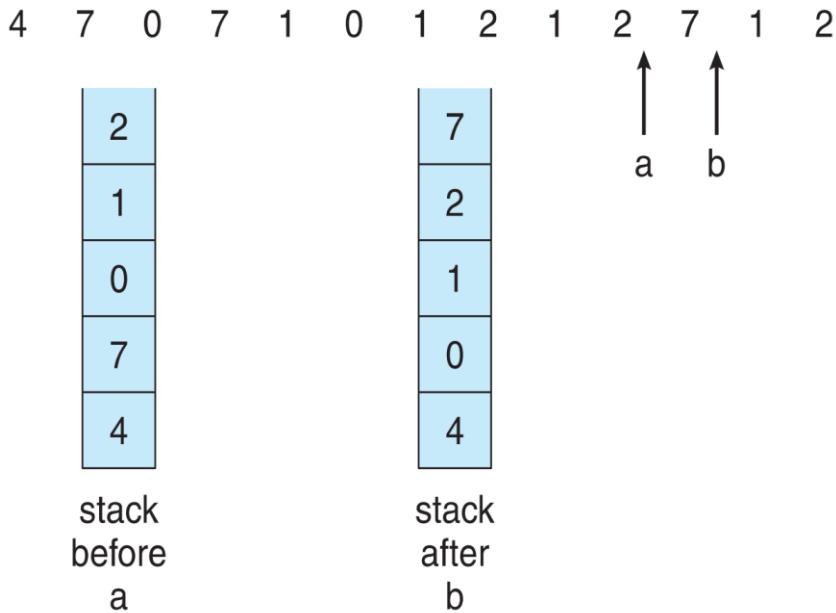


- Obtinem astfel 12 page faults, care e mai bine decat FIFO , dar mai porst decat OPT
- De obicei e un algoritm bun si folosit des

Implementare

- Counter implementation
 - Fiecare page entry are un conter, iar de fiecare data cand o pagina e referita copiem ceasul pe counter
 - Cand o pagina trebuie schimbată, se uita la countere si gaseste cea mai mica valoare (ne uitam la ceas, deci acum 15 min e mai mic decat acum 5 min)
 - Problema este ca trebuie sa cautam in tabel
- Stack implementation
 - Pastram un stack de page numbers intr-un format de double link
 - Atunci cand o page e referenced:
 - O mutam sus de tot
 - E nevoie sa schimbam 6 pointere
 - Fiecare update e mult mai scump, dar nu mai e nevoie de cautare
- LRU si OPT sunt 2 cazuri de algoritmi stiva care nu au anomalia lui Belady
- E necesare folosirea unei stive in care sa tinem cele mai recente page reference uri

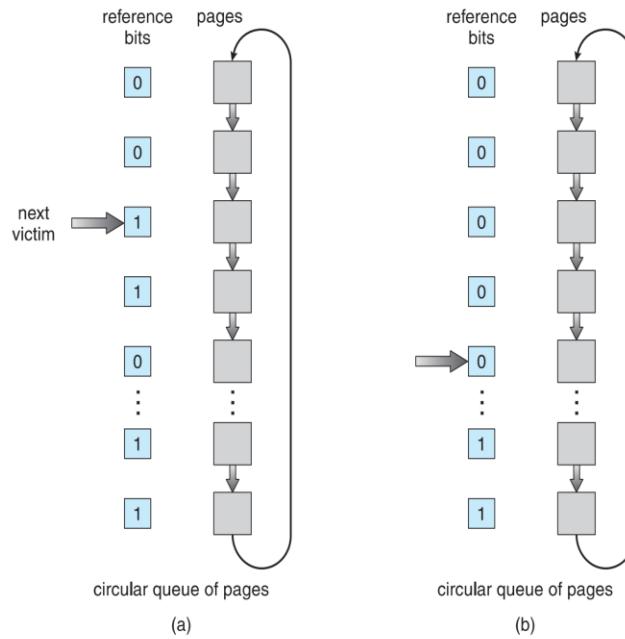
reference string



Algoritmi de aproximare a LRU

- LRU are nevoie de hardware special si foarte incet
- Avem nevoie de reference bit care e initial 0 si cand e chemat se pune 1, insa nu avem cum sa stim ordinea doar cu acesta
- Second chance algorithm
 - In general un FIFO, plus hardware cu reference bit
 - Clock replacement
 - Referenced bit = 0 => lasam pagina in memorie
 - Referenced bit = 1 atunci:
 - Punem reference bit ul la 0 si o lasam in memorie
 - Schimbam urmatoarea pagina, careia i se aplica aceleasi reguli

Second-chance Algorithm



- Second chance algorithm enhanced
 - Se mai numește și Not Recently Used (NRU)
 - Îmbunătățește algoritmul folosindu-se de reference bit și modify bit (daca e disponibil)
 - Algoritmul începe cu $R = 0$ și $M = 0$ și periodic R ul va fi setat la 0
 - Sa consideram urmatoarele perechi:
 - (0,0) nu a fost folosit recent, nu a fost modificat - cea mai bună pagină de înlocuit
 - (0,1) nu a fost folosit recent, a fost modificat - nu e la fel de bun, trebuie să scriem modificările înainte să il schimbam
 - (1,0) a fost folosit recent, dar nu modificat - probabil va fi folosit în curand
 - (1,1) a fost folosit recent, a fost modificat - probabil va fi folosit în curand și va fi nevoie să scriem modificările înainte să il schimbam
 - Cand se cheama page replacementul, folosim schema clock ului dar folosim cele 4 clase ale lio ca sa inlocuim page ul in cea mai joasa non-empty clasa (?)
 - S-ar putea sa fie nevoie de o cautare circulara a queue ului de cateva ori

Counting algorithm

- Păstrează un counter al numărului de referințe care au fost facuta către fiecare pagină (pur și simplu adaugăm R bit ul la fiecare clock tick). Nu este un algoritm foarte folosit
- Least Frequently Used (LFU) Algorithm (numit și Not Frequently Used (NFU)):
 - Schimba pagină cu cel mai mic count

- Nu reflecata corect locality of reference (Acum mult timp au fost pastrate niste frame uri in memorie, in timp ce frame uri recente si frecvente risca sa fie schimba. "NFU doesn't forget anything")
- Most Frequently Used (MFU) Algorithm
 - E bazat pe argumentul ca pagina cu cel mai mic count e probabil abia adusa si nu a avut timp sa fie folosita

Aging

- Ajuta NFU sa reflecte mai corect localitatea temporală și să aproximeze mai bine LRU
- La fiecare tact de ceas, contorul frame-ului este săvârșit la dreapta cu 1 bit, iar R se aduna la cel mai semnificativ bit al rezultatului shiftării => shiftarea micsorează ponderea acceselor departate în timp, iar adunarea lui R trebuie să actualizeze corect gradul de utilizare recentă a paginii
- La page fault se elimină frame-ul cu contorul cel mai mic
- Diferență față de LRU
 - La eliminarea a două pagini cu același prefix (ex 001) nu stim care pagina a fost referată ultima între tactul 1 și 2
 - Contorul frame-ului are reprezentare limitată (8 biti) => nu se distinge între două pagini cu contor 0, deși una poate a fost referată mai recent decât alta

Page Buffering Algorithms

- Păstrează un pool de free frameuri întotdeauna
 - Atunci când e nevoie de un frame disponibil, nu o să fie gasit la fault time
 - Citirea page-ului în free frame și apoi selectarea victimii pe care o să o scoatem să adauge la pool
 - Atunci când ne convine, dam afară victimă
- Posibil să păstrăm o listă de pagini modificate:
 - Atunci când backing store-ul este idle, scriem paginile acolo și le setăm ca non-dirty
- Posibil să păstrăm conținutul unui frame liber și să ne notăm ce este în el:
 - Dacă îl referențiem din nou înainte să fie folosit, nu mai e nevoie să încarcăm de pe disk
 - În general folosit pentru a reduce slowdown-ul dacă victim frame-ul ales a fost incorect

Applications and Page Replacement

- Toți algoritmii astăzi se bazează pe sistemul de operare să ghicească către viitorul
- Unele aplicații au mai multe informații totuși, cum ar fi bazele de date
- Aplicațiile care necesită multă memorie pot să provoace double buffering
 - Sistemul de operare copiază o pagină în memorie ca I/O buffer
 - Aplicația păstrează pagină pentru a o folosi
- Sistemele de operare pot să ofere acces direct la disk, fără să incurce aplicațiile, prin raw disk mode
- Astfel, nu mai avem probleme de buffering, locking, etc.

Allocation of frames

- Fiecare proces are nevoie de un număr minim de frameuri
- De exemplu, IBM 370 are nevoie de 6 pagini să handle uiasca instrucțiunea SS MOVE:
 - Instrucțiunea are 6 bytes, deci se intinde pe 2 pagini
 - 2 pagini fac handle la from
 - 2 pagini fac handle la to
- Maximul este bine înțeles numărul total de frameuri din sistem
- Avem două metode de a aloca:
 - Fixed allocation
 - Priority allocation

Fixed allocation

- Putem avea alocare egală - adică dacă avem 100 de frameuri și 5 procese, atunci vom da 20 de frameuri fiecarui
- Sau putem avea alocare proporțională - alocăm în funcție de marimea procesului

– s_i = size of process p_i

$$m = 64$$

– $S = \sum s_i$

$$s_1 = 10$$

– m = total number of frames

$$s_2 = 127$$

– a_i = allocation for p_i = $\frac{s_i}{S} \times m$

$$a_1 = \frac{10}{137} \leftarrow 62 \cup 4$$

$$a_2 = \frac{127}{137} \leftarrow 62 \cup 57$$

Global vs Local Allocation

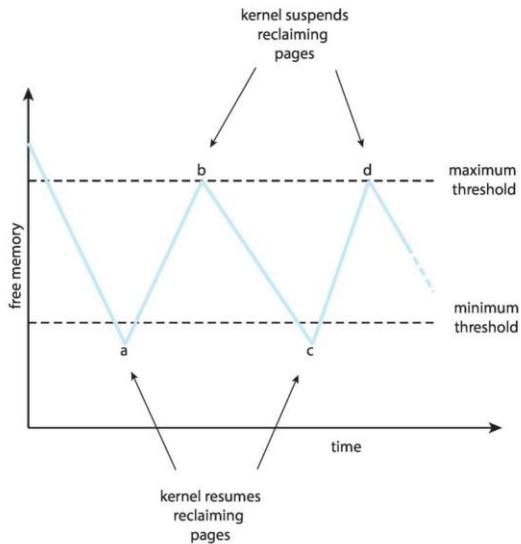
- Global replacementul este un proces care selectează un replacement frame din toate frameurile, un proces poate lua un frame dintr-un alt proces
 - Dar execution time-ul poate să difere drastic
 - Dar avem un throughput mai mare de obicei
- Local replacement - fiecare proces selectează numai din propriul set de frameuri
 - Performanță mai consistentă per-proces
 - Posibil să nu utilizăm toată memoria

Reclaiming Pages

- Strategie ca să implementăm global page-replacement
- Toate cererile la memorie sunt satisfăcute din free-frame list, în loc să aștepte să ajungă la 0 și după să facă page replacement
- Page replacement-ul va fi triggeruit când lista scade sub o anumită limită

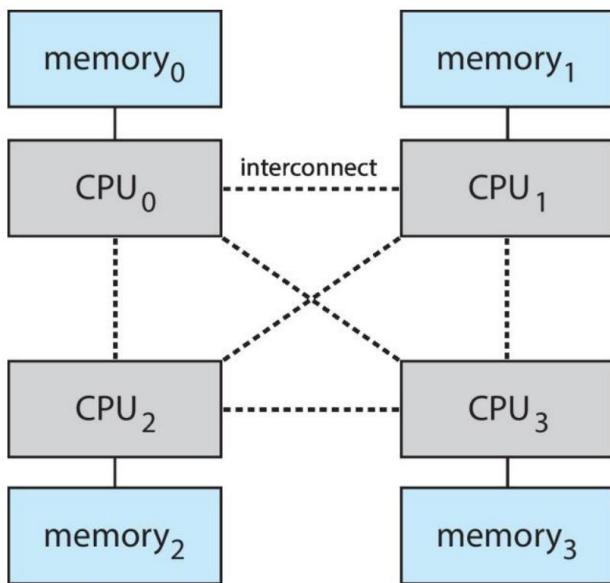
- Strategia asta incercă să se asigure ca mereu va fi suficientă memorie ca să se intamplate noi cereri

Reclaiming Pages Example



Non-uniform Memory Access

- Până acum, am presupus că toată memoria e accesibilă în mod egal
- Foarte multe sisteme însă sunt de tip NUMA - viteza accesului la memorie variază
- Arhitectura NUMA pentru multiprocesoare

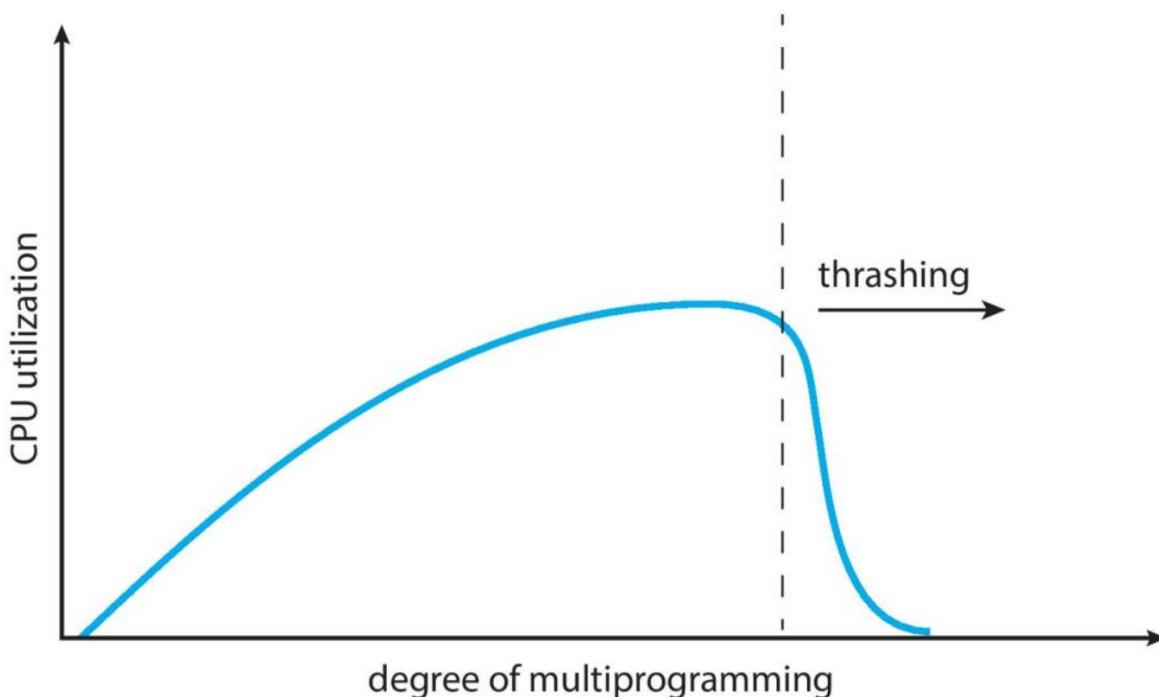


- Performanta optima vine prin alocarea memoriei cat mai apropiat de CPU ul pe care thread ul e programat
 - Totodata, ar trebui sa se poata programa thread uri pe acelasi system board daca se poate

Thrashing

- Daca un proces nu are suficiente pagini, page-fault rate ul va fi foarte ridicat
 - Page fault sa iei pagina
 - Sa faci replace la frame
 - Dar trebuie sa fii rapid sa adaugi inapoi un frame
 - Toate astea duc la:
 - Utilizare scazuta a CPU ului
 - Sistemul de operare va crede ca are nevoie sa creasca nivelul de multiprogramare
 - Un alt proces e adaugat la sistem

Thrashing = un proces e ocupat sa faca schimbari intre pagini

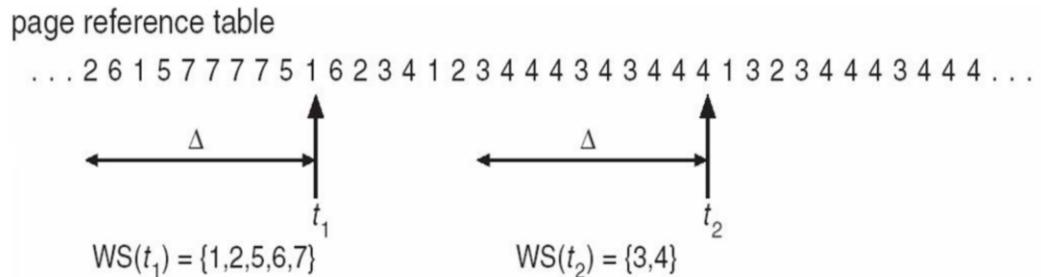


Demand Paging and Thrashing

- Cand functioneaza demand paging ul?
 - Functioneaza pe locality model
 - Procesele migreaza de la o locatie la alta
 - Locatiile se pot suprapune
- Cand se intampla thrashing ul?
 - Cand suma tuturor locatiilor e mai mare decat memoria totala
- Limiteaza efectele folosind local sau priority page replacement

Working-Set model

- $\Delta \equiv$ working-set window \equiv a fixed number of page references
Example: 10,000 instructions
- WSS_i (working set of Process P_i) = total number of pages referenced in the most recent Δ (varies in time)
 - if Δ too small will not encompass entire locality
 - if Δ too large will encompass several localities
 - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \sum WSS_i \equiv$ total demand frames
 - Approximation of locality
- if $D > m \Rightarrow$ Thrashing
- Policy if $D > m$, then suspend or swap out one of the processes

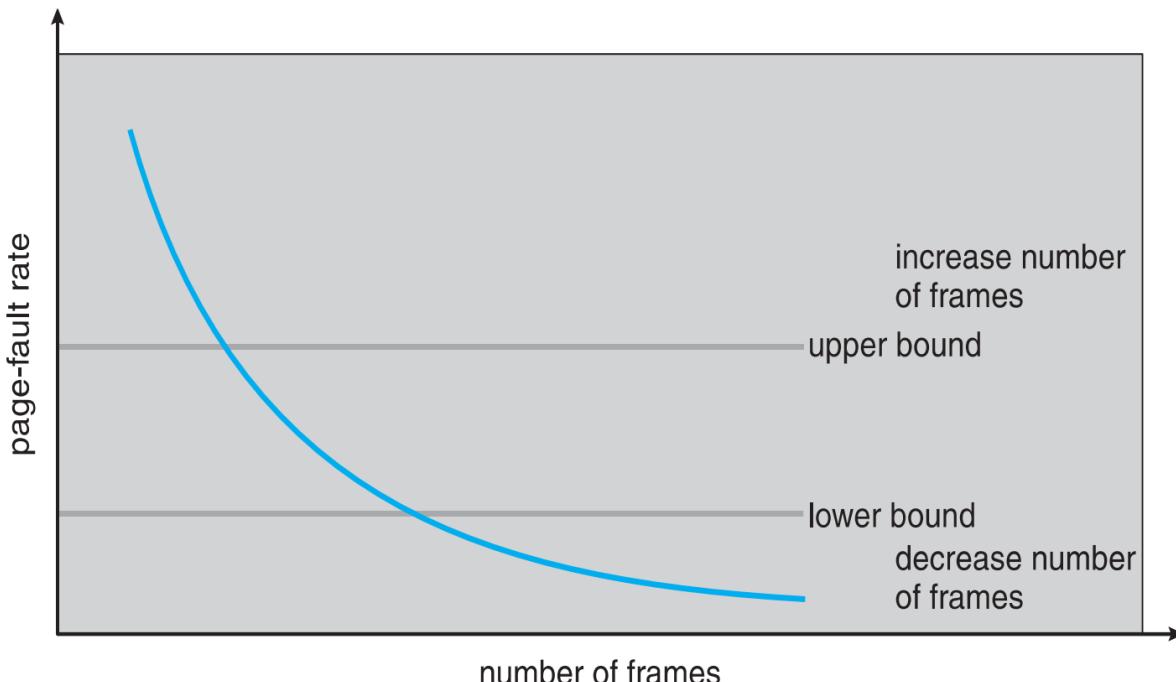


Keeping track of the working set

- Approximate with interval timer + a reference bit
- Example: $\Delta = 10,000$
 - Timer interrupts after every 5000 time units
 - Keep in memory 2 bits for each page
 - Whenever a timer interrupts copy and sets the values of all reference bits to 0
 - If one of the bits in memory = 1 \Rightarrow page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

Page-fault Frequency

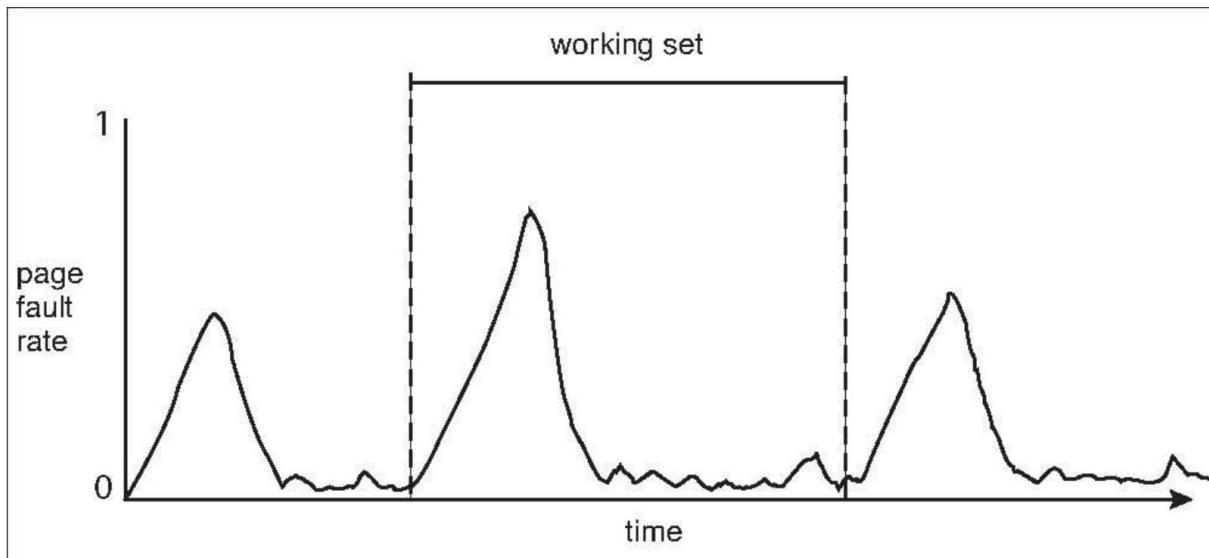
- O abordare mai directă decât WSS
- Setează o frecvență de page fault acceptabilă (PFF) și folosește policy-ul de replacement local
 - Dacă rata e prea mică, procesul pierde frame
 - Dacă rata e prea mare, procesul primește frame



Working Sets si Page Fault Rates

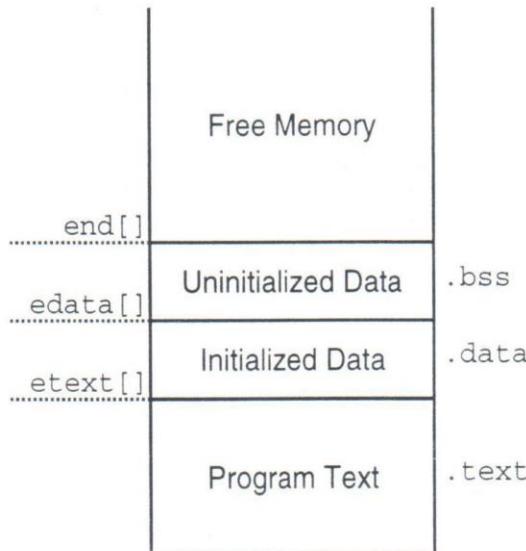
- Există o corelație directă între working set-ul unui proces și page fault frequency-ul lui

- Working set ul se schimba de-a lungul timpului



Alocarea memoriei kernel

- Harta memoriei dupa ce loaderul a incarcat imaginea kernelului
- Editorul de legaturi a generat simboluri care delimitaaza sectiunile imaginii kernel: _etext, _edata, _end
- Codul de initializare a sistemului poate folosi _end pentru a localiza adresa de start a memoriei libere
- Memoria disponibila se afla
 - Fie ca valoare pasata codului de bootstrap
 - Fie proband adresele dincolo de _end



- Memoria disponibila e impartita in frames si pusa in free list
 - Paginile sunt gestionate de un alocator de pagini (page-level allocator)

- În timpul executării, când kernelul are nevoie de memorie pentru structurile sale de date interne (PCB-uri, buffere de rețea, buffere pentru blocuri de disc și asta mai departe) cere o pagină (sau mai multe) alocatorului de pagini
- Structurile de date ale kernelului pot fi adresate virtual, dar nu sunt swapabile
 - Adică nu există page-fault pentru paginile kernel
 - Page-fault-urile ar fi critice pe durata tratarii intreruperilor sau în timpul secțiunilor critice

Alocatorul de memorie kernel

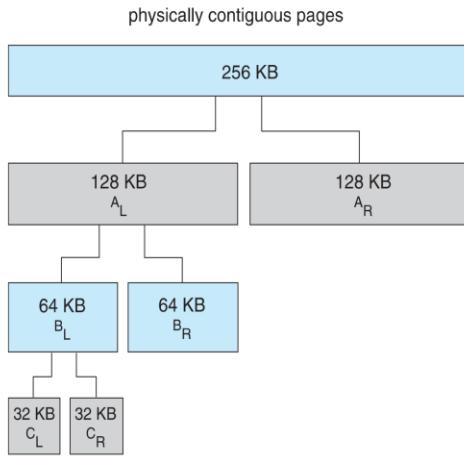
- Kernel Memory Allocator (KMA)
- Componentă a sistemului de operare care obține pagini de la alocatorul de pagini și le folosește pentru satisfacerea cererilor de memorie neswapabilă ale kernel-ului
- Trebuie să fie rapid
 - Stiva kernel este mică și echivalentul variabilelor automatice din programe folosește memorie KMA
 - KMA este apelat din handle-urile de intrerupere care au constraingeri critice de timp
 - KMA lent => degradarea performanței întregului sistem
- Trebuie să aibă interfață de programare simplă
 - De regulă, memoria se aloca într-o parte a kernel-ului și se dealoca din alta => free() trebuie proiectat astfel încât să nu stie dimensiunea alocării
- Trebuie să fie eficient
 - Să aibă factor de utilizare mare (ideal 100%, în practică 50% este acceptabil)
 - Factorul de utilizare e memoria totală cerută / memoria totală necesară pentru a satisface cererile
- Trebuie să fie capabil să obțina pagini noi de la alocatorul de pagini când free-list e goală, respectiv să dea înapoi pagini atunci când nu mai sunt folosite
- Trebuie să fie capabil să compacteze blocurile nealocate adiacente în vederea satisfacerii unor cereri viitoare de dimensiuni mai mari

Buddy system

- Aloca o memorie dintr-un segment fix care constă în mai multe pagini unele lângă altele în memoria fizică
- Memoria va fi alocată folosind power-of-2 allocator
 - Satisfac cererile în unități de mărimi de puteri ale lui 2
 - Cererea va fi aproximată la cea mai mare valoare a lui 2, care este cea mai apropiată
 - Când se cere o alocare mai mică decât ce este valabil, chunk-ul va fi împărțit în 2 bucăți de next-lower power a lui 2 (astea 2 se vor numi buddies)
 - Se va continua procesul până când se face un chunk de marime potrivită
- Spre exemplu, să presupunem că avem un chunk de 256kb și kernel-ul cere 21
 - Impartim în Aleft și Aright de căte 128kb
 - Mai impartim după Aleft în Bleft și Bright de 64kb
 - Mai impartim încă odată Bleft și obținem Cleft și Cright de căte 32kb și îl vom folosi pe unul din ele să satisfacă cererea

- Avantaj: se unesc rapid bucati mici in bucati mai mari
- Dezavantaj: fragmentation

Buddy System Allocator



Implementarea buddy systemului

- Managerul de memorie mentine free lists pentru toate puterile lui 2 pana la dimensiunea memoriei
- Alocarea:
 - Rotunjirea la urmatoarea putere a lui 2
 - Daca nu exista in free lists segmentul corespunzator se merge la puteti 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 adjacent, se compacteaza intr-un segment mai mare; daca segmentul rezultat are si el un buddy, atunci se compacteaza in continuare
- REGULA DE DETECTARE A BUDDIES
- B_1, B_2 buddies $\Leftrightarrow \text{size}(B_1) = \text{size}(B_2) = 2^k \ \&\& 2^{k+1} \mid \text{adr}(B_1 + B_2)$
- Exemplu de buddy sistem

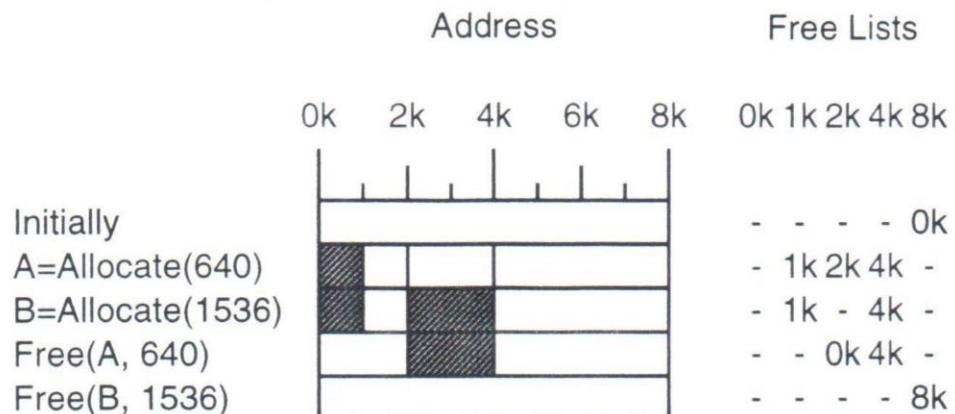


Figure 5: Buddy system operation.

Buddy system pros & cons

- Avantaje:
 - Deallocare rapida (segmentele compactabile se identifica prin adresa si buddies nu se cauta decat in lista segmentelor cu aceeasi dimensiune ca si segmentul eliberat)
- Dezavantaje
 - Fragmentare interna (din cauza rotunjirii de la alocare)
 - Fragmentare externa
 - Ex: dupa alocarea celor 1536 de bytes , o cerere de 5kb nu poate fi satisfacuta, desi exista 5KB liberi in total

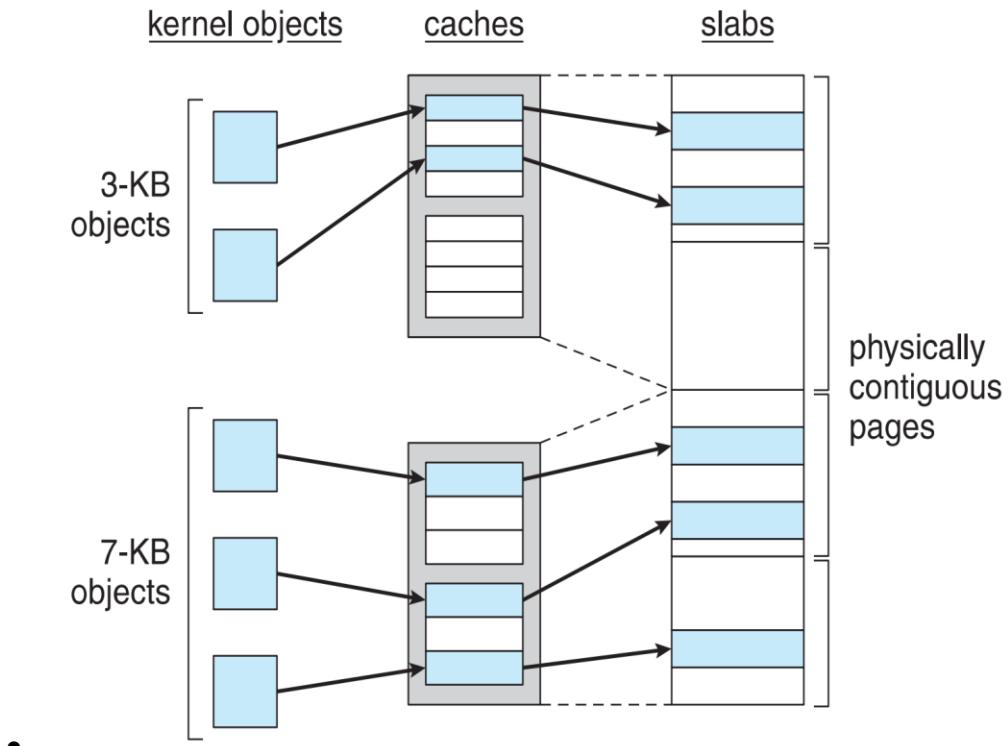
Slab allocator

- O strategie alternativa

Slab = unu sau mai multe pagini invecinate fizic

Cache = unu sau mai multe slab uri

- Un singur cache pentru fiecare structura de date unica din kernel
 - Fiecare cache e plin cu obiecte - instantele structurii de date
- Cand se creeaza, e plin de obiecte marcate ca free
- Cand sunt salvat si structuri, obiectele devin marcate ca used
- Cand un lab e plin de obiecte folosite, urmatorul obiect alocat dintr-un empty slab, iar daca nu mai sunt slab uri, alocam un nou slab
- Beneficiile includ nefragmentarea si indeplinirea rapida a cererilor de memorie



Slab Allocator in Linux

- For example process descriptor is of type `struct task_struct`
- Approx 1.7KB of memory
- New task -> allocate new struct from cache
 - Will use existing `free struct task_struct`
- Slab can be in three possible states
 1. Full – all used
 2. Empty – all free
 3. Partial – mix of free and used
- Upon request, slab allocator
 1. Uses free struct in partial slab
 2. If none, takes one from empty slab
 3. If no empty slab, create new empty



Slab Allocator in Linux (Cont.)

- Slab started in Solaris, now wide-spread for both kernel mode and user memory in various OSes
- Linux 2.2 had SLAB, now has both SLOB and SLUB allocators
 - SLOB for systems with limited memory
 - Simple List of Blocks – maintains 3 list objects for small, medium, large objects
 - SLUB is performance-optimized SLAB removes per-CPU queues, metadata stored in page structure

● De la pag 84 nu am mai scris, bag pula in ea de teorie

END CHAPTER 10

CHAPTER 13

De aici mai scriu doar ce are si el scris + algoritmii

Fisiere

- Abstractie de nivel de sistem de operare pentru stocarea persistenta a datelor
- La nivelul cel mai de jos, stocarea persistenta se face pe discuri (mai recent, memorii flash, SSD, NVRAM etc)
- Sistemul de fisiere
 - Componenta a sistemului de operare (adica parte a kernel ului)
 - Gestionarea mediul de stocare persistenta a datelor
 - Ofera la nivelul aplicatiei abstractia de fisier si apeluri sistem corespunzatoare
- Fisierele
 - Concret, containere pentru stocarea persistenta a datelor
 - Uzual referite prin nume (string ASCII) convertiti la o reprezentare interna a kernelului de catre sistemul de fisiere
 - Pardigma uzuala de folosire: open - read/write - close



File Attributes

- **Name** – only information kept in human-readable form
 - **Identifier** – unique tag (number) identifies file within file system
 - **Type** – needed for systems that support different types
 - **Location** – pointer to file location on device
 - **Size** – current file size
 - **Protection** – controls who can do reading, writing, executing
 - **Time, date, and user identification** – data for protection, security, and usage monitoring
 - Information about files are kept in the directory structure, which is maintained on the disk
 - Many variations, including extended file attributes such as file checksum
 - Information kept in the directory structure
- •

ARHITECTURA DISCURILOR

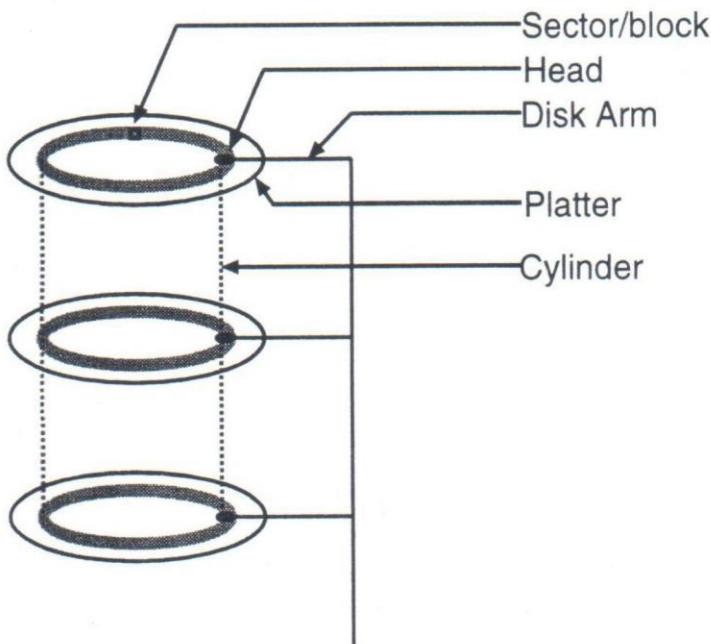


Figure 3: Geometry of a hard disk drive.

- Mediu de stocare format prin stivuirea unor platane pe un ax care formeaza un pachet
- Acest pachet este invartit in jurul axului la o viteza de rotatie constanta (5000 - 10000 rpm)
- Blocurile de disc (sectoare) situate la aceeasi distanta de centrul unui platan alcataiesc o pistă
- Setul de pistă aflate la aceeasi distanta fata de axul platanelor alcataieste un cilindru
- Toate datele dintr-un cilindru pot fi accesate simultan fara miscarea capului de citire
- Datele se citesc in multiplii de dimensiunea blocului (1kb - 4kb)

Citirea blocurilor

- Se muta capul de citire deasupra cilindrului care contine blocul de date (seek) - circa 5ms
- Se asteapta pana cand discul se roteste astfel incat datele da ajunga sub capul de citire (rotational delay) - circa 4ms pentru un disc cu 7200 de rpm
- Se transfera datele (transfer) prin alegerea capului de citire de deasupra pistei pe care se afla datele - circa 1ms pt 1kb
- Timpul mediu de acces random in general
 - $\text{timp seek} + \text{timp rotatie} + \text{timp transfer} = 9.1 \text{ ms}$
- Timpul mediu de acces random de pe acelasi cilindru
 - Timp rotatie + timp transfer
- Timpul mediu de citire a urmatorului bloc de pe aceeasi pistă
 - Timp transfer = 0.1 ms
- Idee centrala: minimizarea timpilor de seek si rotatie

ALGORITMI DE DISK SCHEDULING

- Incearca sa minimizeze timpul de cautare (seek time)

FCFS - simplu dar eficient

- Ex: coada de cereri blocuri aflate pe cilindrii 53,98,183,37,122,14,124,65,67 => capetele de citire se vor muta peste 640 de cilindrii

SSTF (Shortest seek time first)

- Serveste cererile de pe cilindrii cei mai apropiati de pozitia curenta a capetelor de citire
- Ex anterior: capetele de citire se muta peste 236 cilindrii
 - 53, 65, 67, 37, 14, 98, 122, 124, 183
- Tot nu e optimal, ex: 53,37,14,65,67,98,122,124,183 , adica doar 208 cilindrii parcursi
- Sufera de starvation: daca apar in permanenta cereri in apropierea capetelor de citire, cererile "indepartate" sunt intarziate indefinit

SCAN (algoritmul liftului)

- Bratul discului porneste de la un capat al acestuia catre celalalt si serveste cererile intalnite in cale
- Ajuns la capatul discului, o ia in sens invers
- Exemplu anterior ne da 203 salturi de cilindrii

- O cerere aparuta chiar inaintea capului de citire e servita imediat
- O cerere aparuta imediat in spatele capului e intarziata pana cand se intoarce capul de citire in sens contrar

C-SCAN (circular SCAN)

- Pentru o distributie uniforma a cererilor, cand bratul ajunge la capatul discului se intoarce, exista relativ putin cereri in fata capului pentru ca acestea tocmai au fost tratate
- Densitate mare de cereri noi e la capatul celalalt capat al discului unde cererile au asteptat cel mai mult
- Ofera timp de asteptare mai uniform (la momentul atingerii capatului discului, capul de citire se intoarce la celalalt capat)
- Trateaza cilindrii discului ca pe o lista circulara
- Ex anterior: bratul se muta peste 167 de cilindrii (se considera ca mutarea capului de citire la inceputul discutiei e o operatie foarte rapida)

LOOK

- SCAN si C-SCAN se muta capul de citire peste tot discul
- O implementare practica ia in calcul doar cererile pentru cilindrii situati intre numarul minim, respectiv maxim
- Algoritmii respectivi se numesc LOOK si C-LOOK
- Exemplu anterior: La cilindrul 183 bratul se intoarce imediat la cilindrul 14