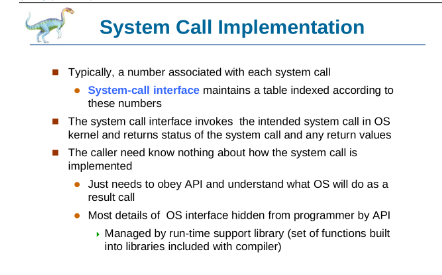
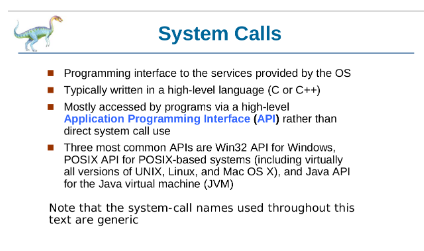
**CURS 2**

****

API (Application Program Interface)

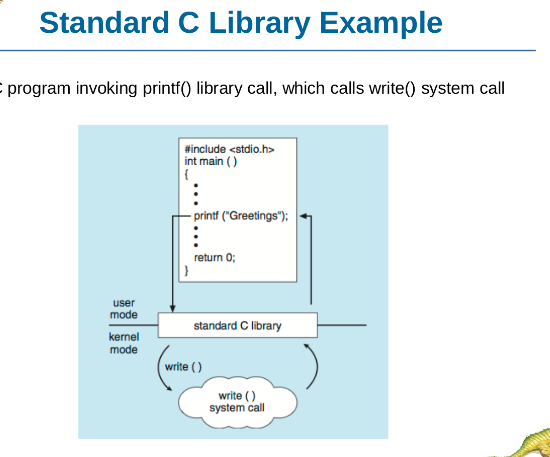
**how one piece of software can interact with another**

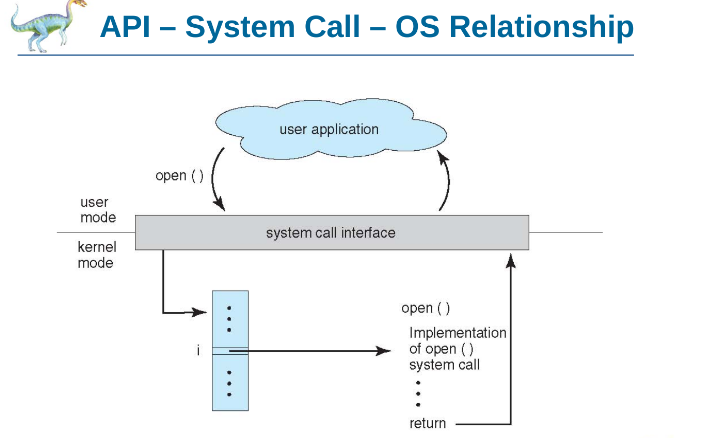
ABI( Application Binary Interface)

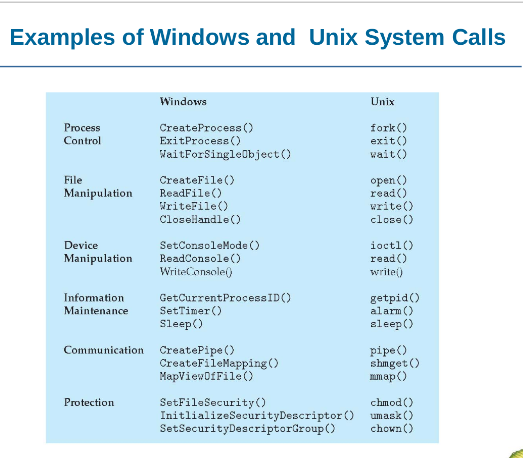
**analogous to an API but expressed in compiled code instead of source code.**

**USERLAND vs KERNEL**

**API vs ABI**







*batch=”non interactive execution”(procese care se executa in fundal, “with later execution”)*

|  |  |
| --- | --- |
| * **printf(...)** * **POSIX(Portable Operating System Interface)** * **WIN32API** * **ret = read(fd, buf, count)** * **struct sys\_read\_args{**   + **int fd; ….** * **struct sys\_read\_args \*v**   + **v = malloc(sizeof \*v)**   + **v->fd=fd**   + **v->buf=buf**   + **v->count=count**   + **sys\_read(v)** | **write(2)** |

**Fișiere**

**-permisiuni**

**USER GROUPS OTHERS**

**twx r— —**

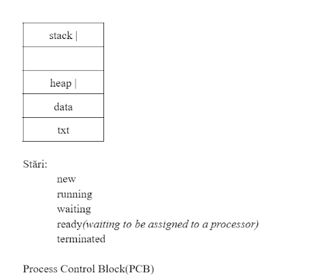
**111 100 000**

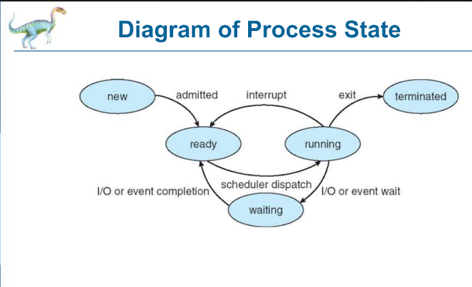
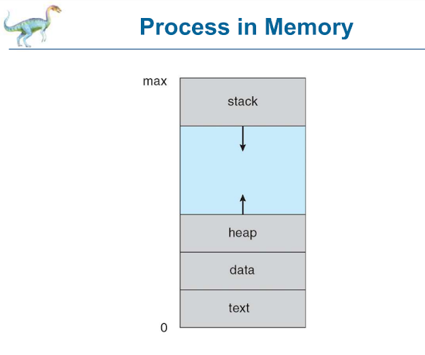
**7 4 0**

**CURS 3**

**PROCESSING**

→PROCESE





Stări:

new*(process created)*

running*(instructions executed)*

waiting*(waiting for event to occur)*

ready*(waiting to be assigned to a processor)*

terminated

Process Control Block(PCB)

|  |
| --- |
| process state |
| process number |
| program counter |
| registers |
| memory limits |
| list of open files |
| … |

→ **THREATS**

!process has a single thread of execution

**→ PROCESS SCHEDULING**



**→ short-term scheduler(CPU scheduler):**

* ce proces trebuie executat imediat, bazându-se pe criterii(prioritatea procesului/timp execuție);
* rulează la intervale scurte de timp

**→ long-term scheduler(job scheduler) ⇔ load scheduler:**

* the degree of multiprogramming;
* planificarea proceselor( decide care proces trebuie să intre în memoria principală si care vor fi in memoria secundară sau in swap);
* rulează la intervale mai mari de timp;
* impact redus asupra performanțelor sistemului

→ **medium-term scheduler(swapper):**

* procese care au fost suspendate pentru o perioadă scurtă de timp
* decide daca acestea trebuie readuse in memoria principala sau daca trebuie sa aștepte încă
* ordinea in care procesele vor fi executate, tinant cont de criterii precum prioritatile proceselor, resurse sau restrictii;

scop→ maximizeaza utilizarea resurselor sist si optimizeaza performanța sistemului in ansamblu

* **I/O bound process**(ex. jocurile video)
* **CPU bound process**(ex: resizing images)

* **single foreground process**(via user interface)
* **multiple background processes**

**ssh-modul standard de a ne conecta la celelalte calculatoare**

*Secure Shell este un protocol de rețea criptografic ce permite ca datele să fie transferate folosind un canal securizat intre dispozitive de rețea.*

UNIX examples:

fork()

*pid = fork()*

*-> creaza un nou proces->child*

*->intoarce pid-ul copilului, copilul primeste 0*

exec()

wait()

pid = *wait(&retval) -> pid primeste pid-ul copilului*

*1: fork()*

*2:fork()*

*3:fork()*

*=> 2^3 - 1*

*(la fiecare fork(), fiecare “nod” existent face un copil)*

*pid = fork()*

*if(pid > 0){*

*pid1 = fork();*

*if(pid1>0)*

*fork();*

*}*

*wait(null)*

*wait(null)*

*wait(null)*

**→PROCESS TERMINATION**

**→ cascading termination:**

* terminarea unei serii de procese ca rezultat al terminarii unui proces părinte
* În acest caz, atunci când procesul părinte se termină, el trimite semnale de terminare tuturor proceselor copil afiliate acestuia, care la rândul lor, pot trimite semnale de terminare altor procese. Acest lucru se continuă până când toate procesele sunt terminate, făcându-se o curățare completă a sistemului

**→PROCES ZOMBIE**

* Un proces care s-a încheiat dar nu a fost așteptat
* Dacă un proces zombie rămâne și orfan, este adoptat de init și apoi încheiat/
* Un proces zombie este un proces care a finalizat executia, dar informatiile sale despre proces inca sunt retinute in sistem de catre parintele sau.
* Acest proces nu mai poate fi accesat sau manipulat, dar ocupa inca resurse ale sistemului, cum ar fi descriptoarele de fisiere si memoria, si nu poate fi terminat pana cand parintele sau nu il marcheaza ca fiind terminat.

*pid\_t pid = fork();*

*if (pid == 0) {*

*printf("Child process started with PID %d\n", getpid());*

*sleep(5);*

*printf("Child process is now exiting\n");*

*return 0;*

*} else {*

*printf("Parent process is waiting for child process to terminate\n");*

*wait(NULL);*

*printf("Child process has terminated, but the parent process does not properly handle its status\n");*

*sleep(5);*

*printf("Parent process is now exiting\n");*

*return 0;*

*}*

* In acest exemplu, deoarece procesul părinte nu gestionează starea procesului fiu (de exemplu, prin apelul waitpid()), acesta devine un proces zombie și persistă în sistem până când procesul părinte nu se termină.

**→PROCES ORFAN**

* Un proces al cărui părinte și-a terminat execuția
* Își pierde legătura în ierarhia de procese
* Este adoptat de un proces dedicat-init (procesul principal al sistemului), care devine parintele lor implicit si se ocupa de terminarea acestora.
* Un proces orfan este un proces care a fost creat de un alt proces, dar care nu mai are niciun parinte care sa-l supravegheze sau sa-i gestioneze terminarea. Acest lucru poate duce la crearea unui proces orfan care sa ramana activ pe sistem, consumand resurse, chiar daca procesul parinte a fost terminat.

*if (pid == 0)*

*{*

*printf("Child process with PID: %d\n", getpid());*

*printf("Child process's parent PID: %d\n", getppid());*

*sleep(5);*

*printf("Child process's parent PID after 5 seconds: %d\n", getppid());*

*}*

*else if (pid > 0)*

*{*

*printf("Parent process with PID: %d\n", getpid());*

*printf("Parent process's parent PID: %d\n", getppid());*

*sleep(1);*

*printf("Parent process is terminating...\n");*

*}*

*return 0;*

* În acest exemplu, procesul copil este creat prin apelarea funcției fork(). După ce procesul părinte se termină, procesul copil rămâne fără un proces părinte și devine un proces orfan.

**→ PROCES DAEMON**

* Un proces daemon este un proces care rulează în fundal și nu are nicio interacțiune directă cu utilizatorul.
* Acest tip de proces rulează în mod constant, fără a fi nevoie să fie lansat manual sau interacționează cu utilizatorul prin intermediul unei interfețe grafice sau a unei linii de comandă.
* Se termină doar atunci când sistemul este shutdown.
* De exemplu, serviciile de sistem, cum ar fi print server-ul sau serviciul de planificare a sarcinilor, sunt de obicei procese daemon.

*pid\_t pid;*

*pid = fork();*

*if (pid > 0) {*

*// Procesul parinte se termina*

*exit(0);*

*} else if (pid == 0) {*

*// Procesul copil devine un daemon*

*setsid();*

*chdir("/");*

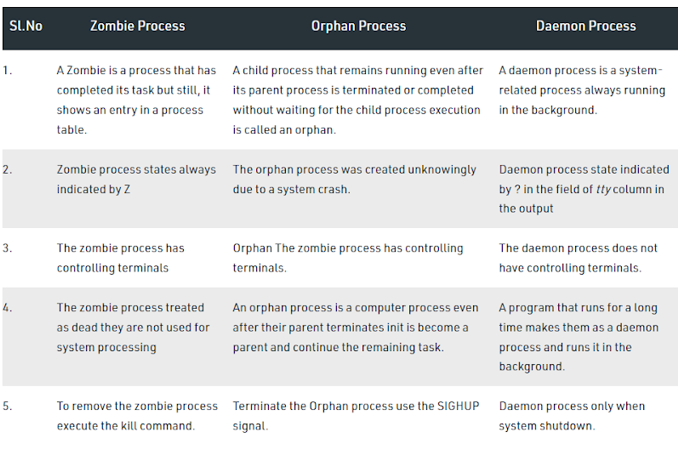
*umask(0);*

*while (1) {*

*// Codul procesului daemon*

*}*

*}*



Exemplu:

→ Google Chrome Browser → multiproces cu 3 tipuri de procese

→ Browser process(interfața userului, disk, I/O)

→Render process(pag web, HTML, Javascript)

→Plus-in process

!!!Fiecare ‘tab deschis’ este un proces separat!!!

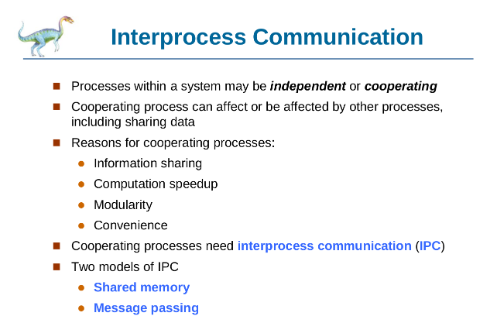
**→COMUNICAREA INTERPROCESS(IPC)**

Procese:

(1) independent,

(2) cooperating→sunt afectate/pot afecta alte procese

→avantaje: information sharing, computation speedup, modularity, convenience



**IPC:** → **Shared memory**

→ permite mai multor procese sa aibă acces la același spațiu de mmeorie

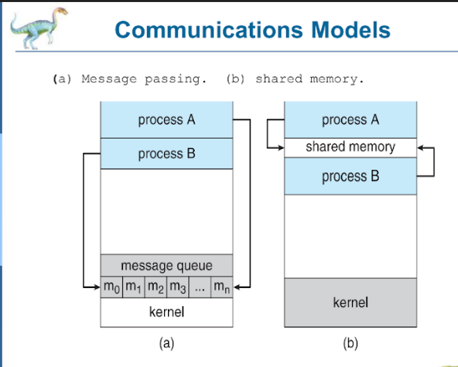
→ crește performanța și comunicare interprocess(IPC)

         → **Message passing**

→procesele pot transmite informații sau date unui alt proces prin intermediul unei SD numite mesaj

→ procesele nu au acces direct la memoria celuilalt proces, transmit mesajul printr-o coadă de mesaje

→ abstractizare + control strict asupra accesări resurselor + gestionare mai bună a erorilor



**→ SOCKETS**

**→** interfață de comunicare între 2 procese prin intermediul rețelei

→ permite transmisia de date între procese prin conexiuni de rețea

→ “canal” prin care procesele pot comunica între ele

→ 3 tipuri(protocoale de comunicare specifice): (1)Connection-oriented(TCP), (2) Connectionless(UDP), (3)MulticastSocket

→**REMOTE PROCEDURE CALLS(RPC)**

→permite unui program să execute apeluri de proceduri la distanță prin intermediul unui sistem de mesagerie

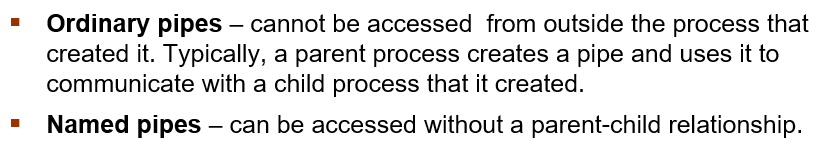
→ folosit pt construirea sistemelor distribuite; programe separate utilizate pentru sarcini diferite, dar comunică între ele pentru a realiza o anumită funcționalitate

**→PIPES**

→se comporta ca o cale de comunicare(cele mai simple)

→creează un canal între 2 procese prin care pot să schimbe informații

→ utilizate adesea în shell-urile Unix



*$ ls | grep \*.tgz*

*ls → listeaza la stdout*

*grep = general regular expressions prep;*

*| ; 2 procese legate printru-un pipe, direcția fiind →*

*\*.tgz -> fisiere de tip tgz*

*→comanda din stanga ia la stdout si → stdin in dreapta*

*$ cat names.txt | grep Paul | grep -v “Irofti”*

*cat = afiseaza pe ecran continutul unui fisier*

*-v = negatie*

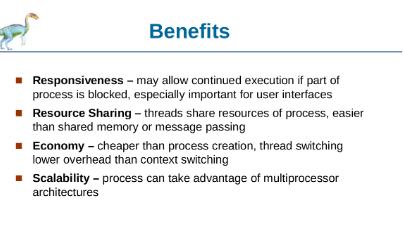
*→ afiseaza “ toti Paul din fisier in afara de Paul Irofti”*

**CURS 3**

**THREADS**⇔ fire de executie

PCB: regs, PC, files, PID…..

|  |  |
| --- | --- |
| T1 | T2 |
| main()  PC\_T1  stack  regs | spellcheck(word)  PC\_T2  <- TID  stack  regs |



PC= program counter

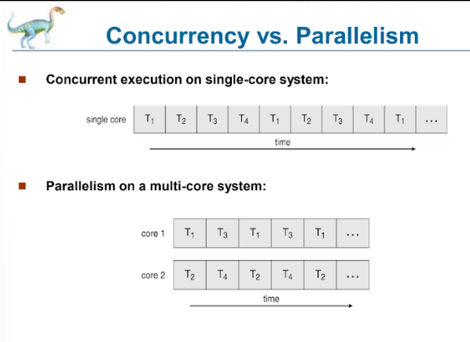
Ex: Pentru un spellcheck, ce introduce utilizatorul se va situa in heap(fiind alocat dinamic); data este static, iar txt este codul.

→**MULTICORE PROGRAMMING**

**→**Data parallelism

→Task parallelism

→**CONCURRENCY vs. PARALLELISM**



→ **CONCURENȚĂ ⇔ structură, coordonare**

→ O resursa limitata, mai multi vor sa o foloseasca

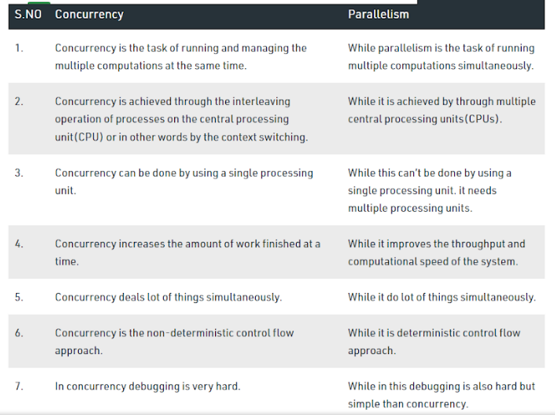
→mai multe task-uri sunt executate independent, dar nu pot fi rulate in același timp

→ creează impresia de paralelism, dar nu e

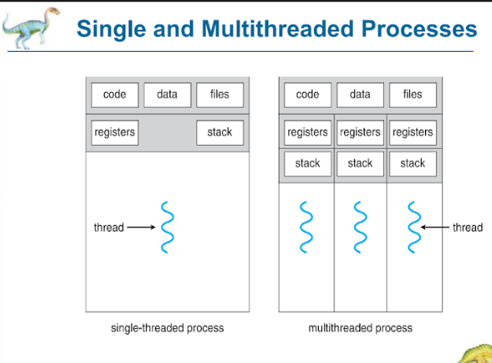
→**PARALELISM ⇔ execuție**

→executarea mai multor task-uri în același timp; fiecare task pe un procesor/core separat

!!!Paralelismul este o formă de concurență, dar nu toate formele de concurență implică execuție paralelă



→ SINGLE & MULTITHREADED PROCESSES

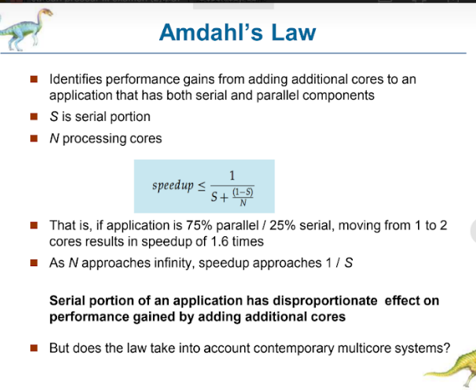


**→ AMDAHL’S LAW**

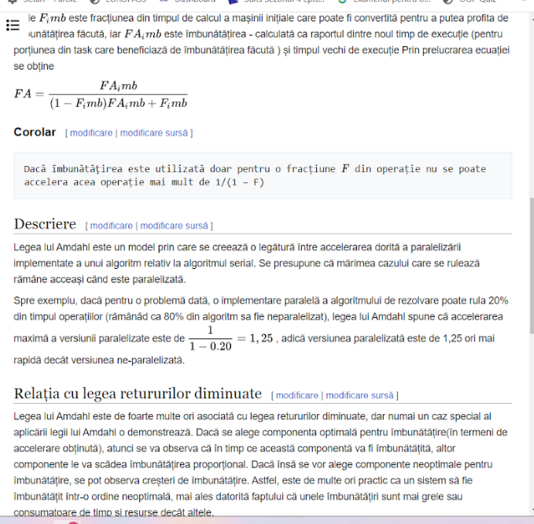
**→**PRINCIPIU: creșterea performanței prin adăugarea de resurse(procesoare suplimentare)

→Legea afirmă că o anumită porțiune a aplicației trebuie să fie serială și nu poate fi paralelizată, iar adăugarea de procesoare suplimentare nu va îmbunătăți performanța acestei secțiuni seriale în mod proporțional

→Eficiența globală a sistemului este limitată de această porțiune serială și nu poate fi îmbunătățită în mod nelimitat prin adăugarea de resurse.



*Spre exemplu, dacă pentru o problemă dată, o implementare paralelă a algoritmului de rezolvare poate rula 20% din timpul operațiilor (rămânând ca 80% din algoritm sa fie neparalelizat), legea lui Amdahl spune că accelerarea maximă a versiunii paralelizate este de*

*, adică versiunea paralelizată este de 1,25 ori mai rapidă decât versiunea ne-paralelizată.*

→ **USER THREADS**  ⇔ fire de execuție controlate și gestionate de utilizator

→pot fi utilizate pentru a implementa multithreading

→mai ușor de gestionat( controlate în întregime de utilizator)

→ user-level threads lib

→ ex cu relatie de tip n:1

* posix pthdreads ( POSIX = Portable Operating System Interface)
* windows thdreads
* java threads

→**KERNEL THREADS** ⇔ fire de execuție controlate și gestionate de sist de operare

→ implementate direct în kernel

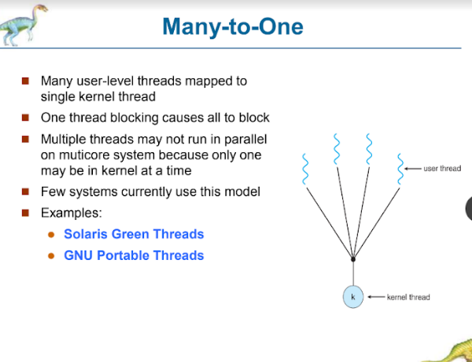
→pot efectua o varietate de activități( gestionarea memoriei/fișierelor/traficului de rețea)

→ transparente pentru utilizator

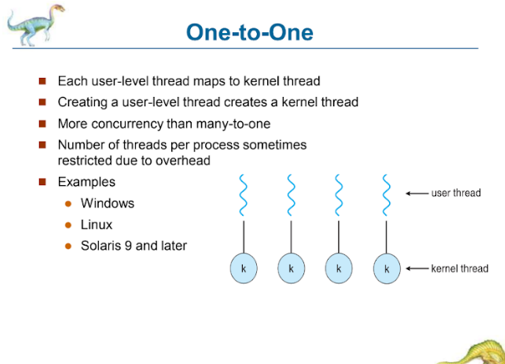
→ex relatie de tip 1:1

→ **MULTITHREADING MODELS**

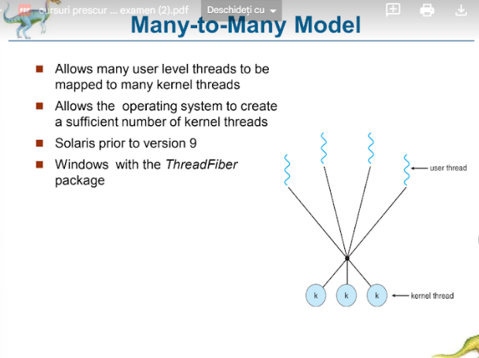
→ M:1



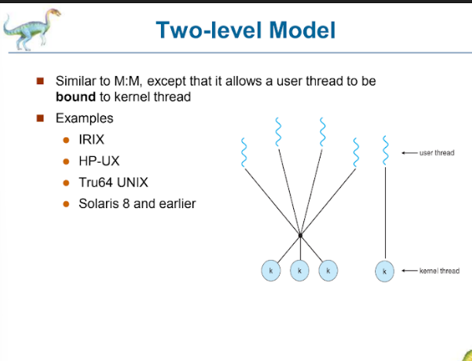
→1:1



→M:M



→ 2-level model



JOIN!! ⇔ wait-ul de la procese

**CURS 5 - SINCRONIZARE**

**→CRITICAL SECTION PROBLEM**

→ mai multe procese  accesează și modifică același set de date/resurse

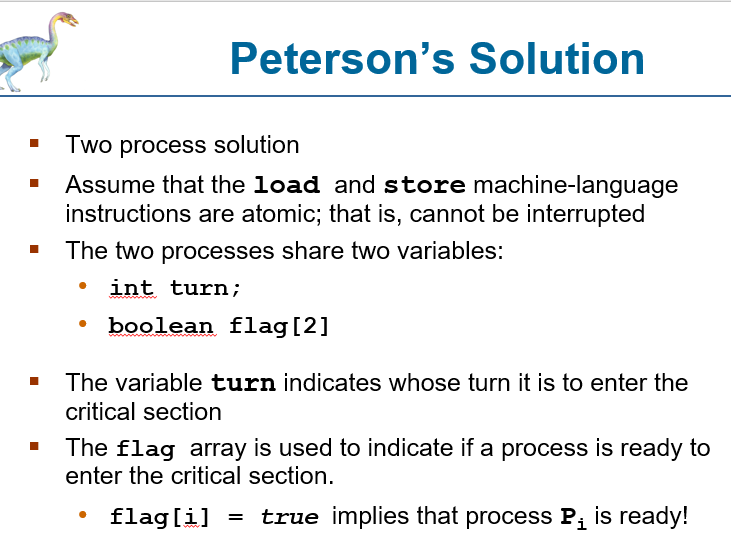
→solutie: sincronizare

→ permite unui singur proces să acceseze resursele la un moment dat

→ **SOLUȚIA PETERSON→software**

→ pentru probl secțiunii critice in sisteme cu 2 procesoare

→constă în utilizarea a două variabile flag(una pentru fiecare proces) care indică dacă procesul respectiv încearcă să intre în secțiunea critică și un turn care indică procesul care are acces la secțiunea critică





→**soluție harware**

→ operație atomică(nu poate fi întreruptă)

*boolean TestAndSet(boolean \*target){*

*boolean rv = \*target;*

*\*target = TRUE;*

*return rv;}*

→ soluția

*do{*

*while(TestAndSet(&lock))*

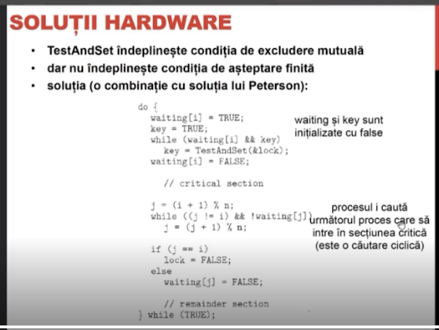
*; //do nothing*

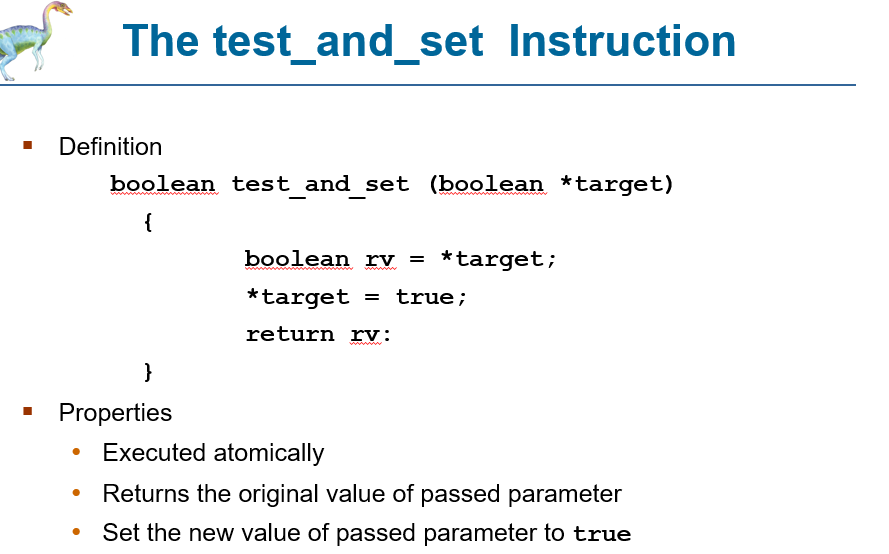
*//critical section*

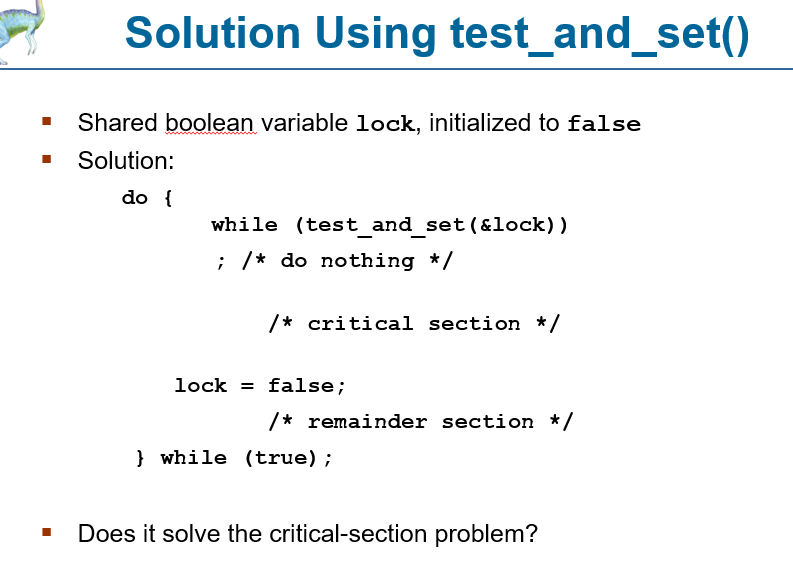
*lock = FALSE;*

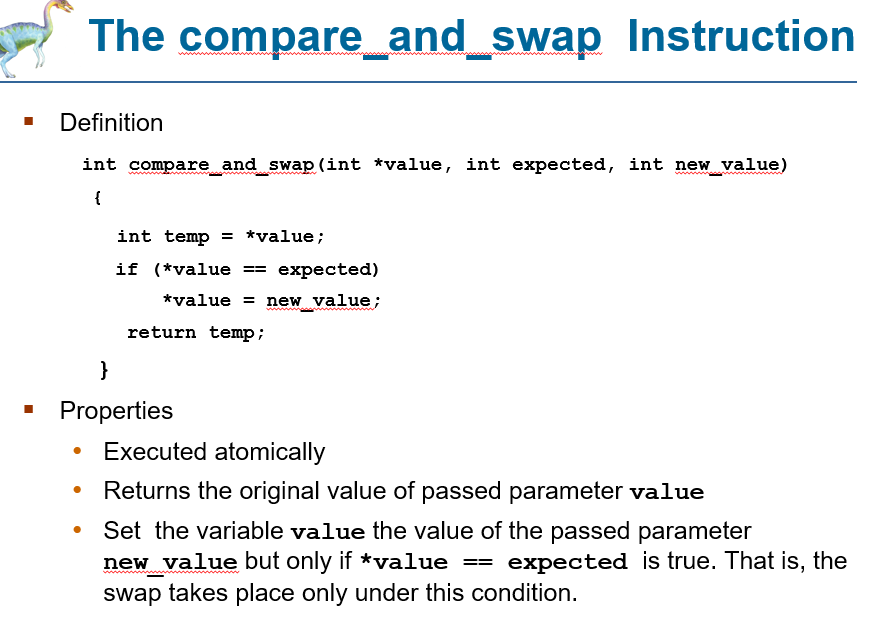
*//remander section*

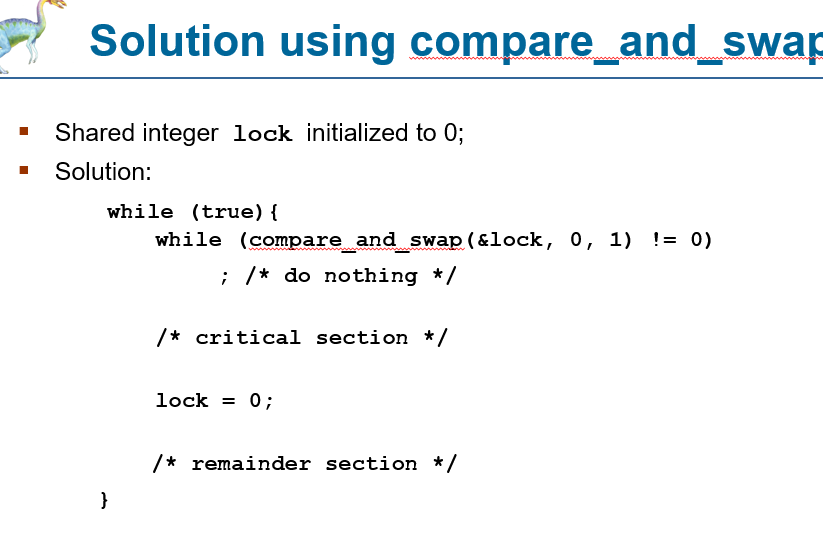
*} while(TRUE);*

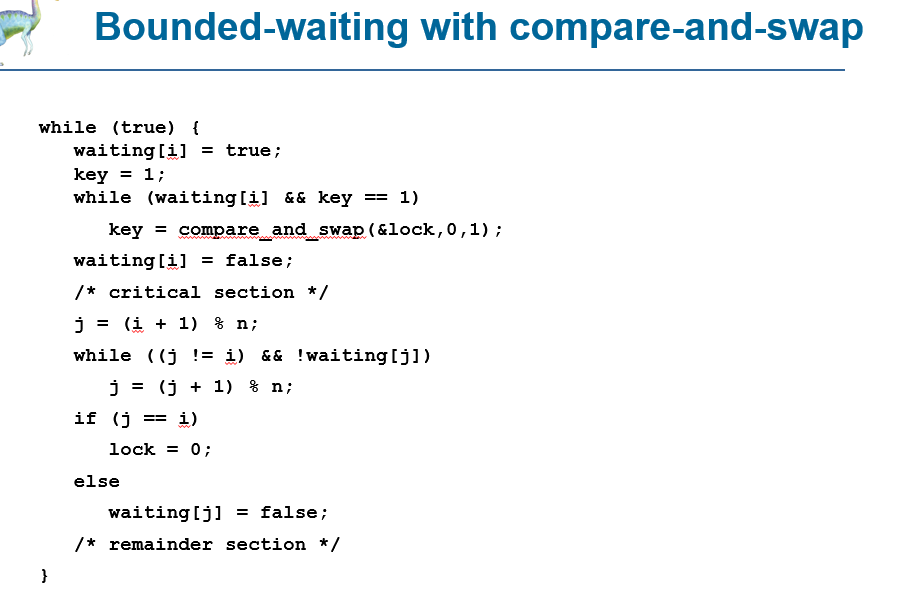


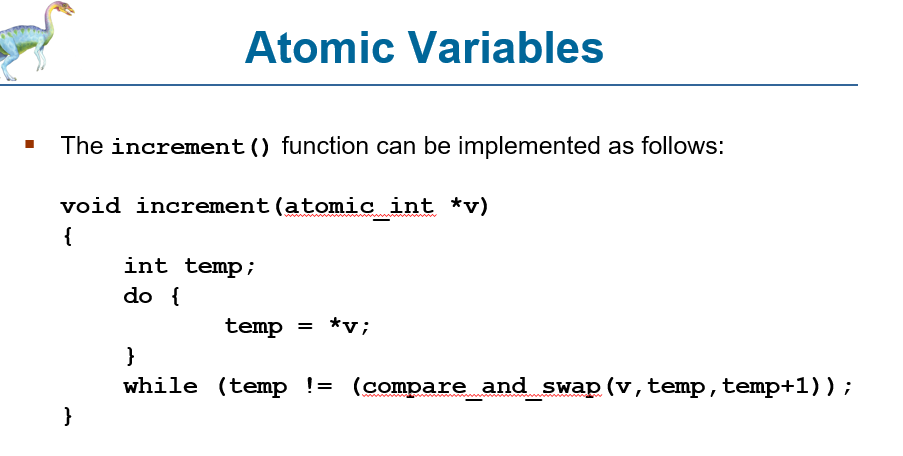












→ **MUTEX ⇔ MUTual EXclusion**

**→**mecanism de sincronizare pentru a controla accesul la resurse critice de mai mulți procese sau thread-uri în același timp

→asigură că doar un singur thread poate accesa resursa critica la un moment dat, evitând astfel conflictele de acces

→funcționează prin acordarea accesului exclusiv la resursă numai unui singur thread la un moment dat, în timp ce alte thread-uri sunt blocate sau așteaptă accesul.

**→SEMAFOARE**

→ instrumente utilizate în sistemele de operare pentru a controla accesul la resursele sistemului între mai multe procese sau fire de execuție

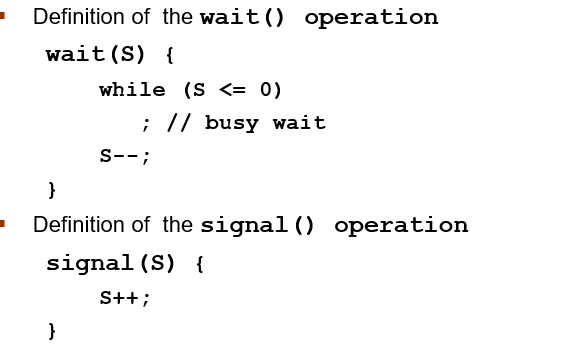
→utilizate pentru a evita conflictele de acces la resursele sistemului, cum ar fi fișierele sau memoria partajată, și pentru a coordona acțiunile dintre procese

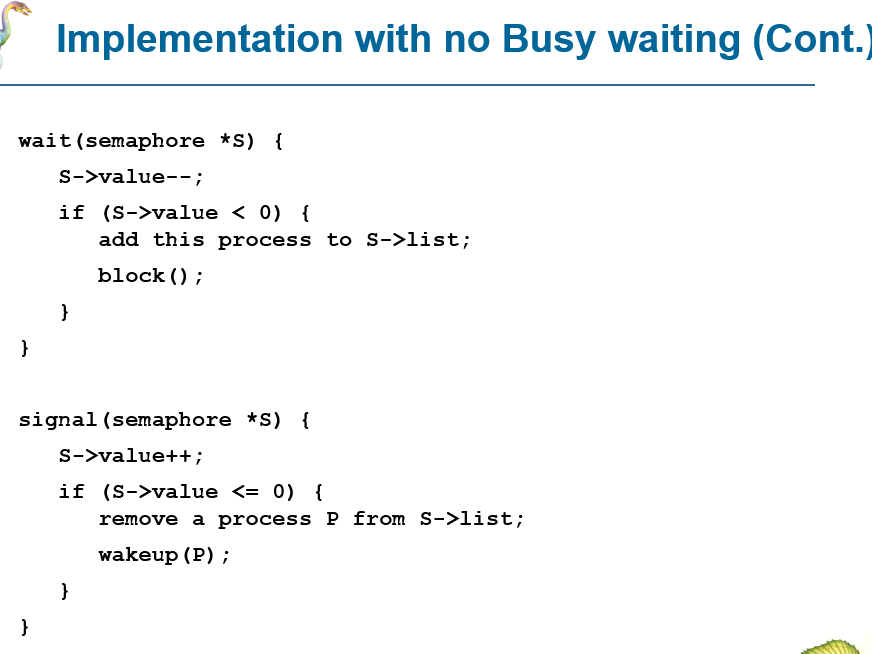
→poate fi setat sau resetat, în funcție de starea accesului la resursele sistemului

→ când un proces încearcă să acceseze o resursă protejată prin semafor, acesta blochează sau așteaptă până când semaforul este setat, permitând accesul la resursă

→ față de mutex, avem wait() si signal()

→ În contrast cu mutex-ul, care permite doar accesul unei singure unități de execuție la un moment dat, semafoarele permit accesul unei unități de execuție specificate la o resursă, ținând cont de limitările de capacitate ale acestei resurse.





***Monitor***

**→DEADLOCK⇔ circular waiting**

→situatie in care doua sau mai multe procese sau fire de executie sunt blocate in mod permanent fiecare asteptand celalalt sa elibereze o resursa

→ imposibilitatea de a continua executia normala a proceselor

→poate cauza blocarea intregului sistem

→se poate produce in situatii in care fiecare proces detine o resursa necesara celuilalt proces si asteapta sa primeasca acces la resursa detinuta de celalalt proces

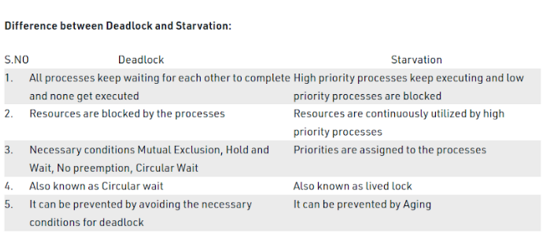
**→STARVATION⇔ lived lock**

→ un proces sau un grup de procese nu reușesc să obțină accesul la resursele necesare într-un mod rezonabil de timp

→cauzat de o schema de planificare defecta sau de un proces care monopolizează resursele, ceea ce împiedică alte procese să progreseze și să termine în mod rezonabil

→poate duce la creșterea duratei de execuție a proceselor și la o utilizare ineficientă a sistemului.

→ se soluționează cu **AGING**(gradual, crește prioritatea proceselor care așteaptă deja de mult timp)



**→PRIORITY INVERSION**

→ un proces cu prioritate mai mare este blocat din cauza unui proces cu prioritate mai mică

→ afectează timpul de execuție pentru procesul cu prioritate mare

→ comportament impredictibil al sistemului și întârzieri în execuția proceselor importante

→ modalități de evitare:

* priorități crescătoare( procesele cu prioritate mare primesc acces la resurse înaintea proceselor cu prioritate mică)
* prioritate dinamică
* cerere-ofertă
* protocolul Ceiling and Delegation; stabilește un plafon de prioritate pentru fiecare resursă
* blocare temporară( procesul cu prioritate mai mică poate fi blocat temporar până când cel cu prioritate mai mare are acces la resurse)

**→ PROBLEME CLASICE DE SINCRONIZARE**

**→ BOUNDED-BUFFER ⇔ producer-cosumer problem**

**→** implică 2 sau mai multe procese care colaborează pentru a transfera obiecte într-un buffer(zonă memorie) limitată

→ se concentrează pe gestionarea accesului la buffer și la evitarea blocării bufferului din cauza accesului incorect

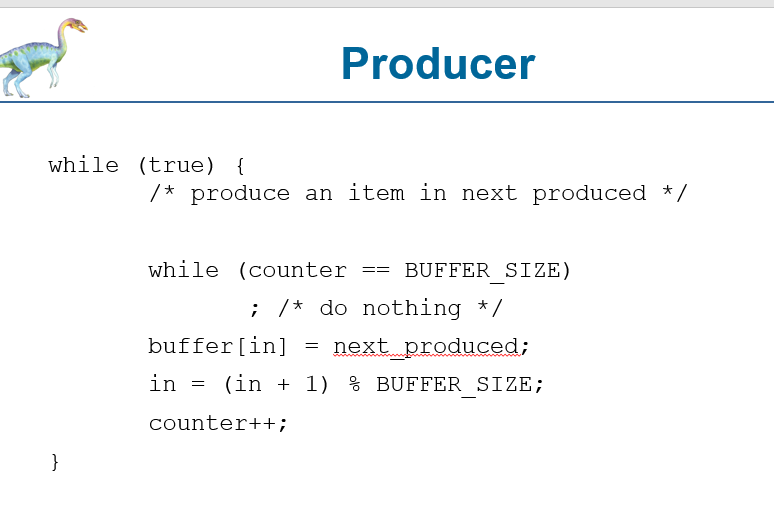
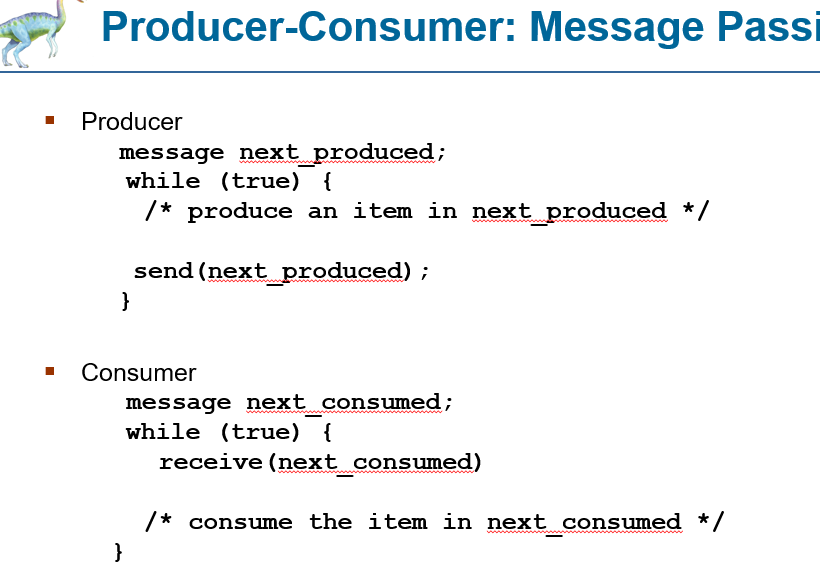
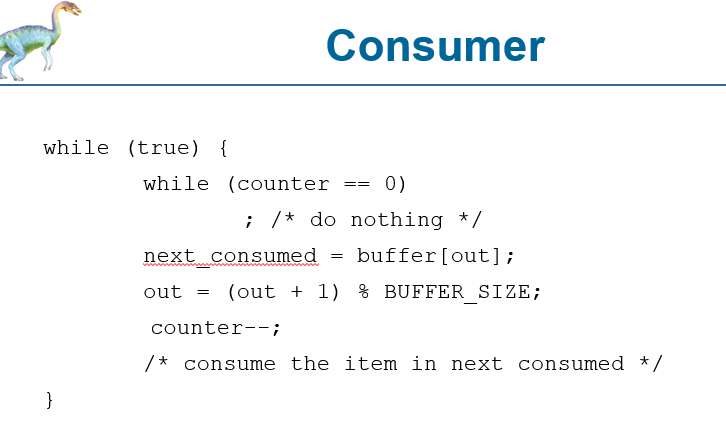
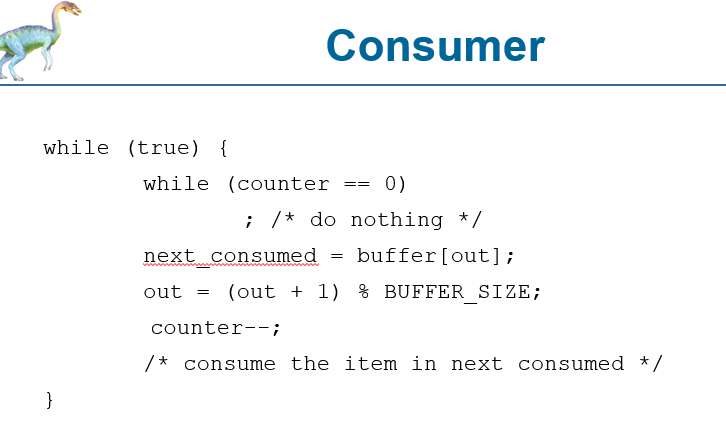
**→** producatorii ‘produc’ un produs, consumatorii ‘consuma’ produsul, dar ambii folosesc unul din containere de fiecare dată

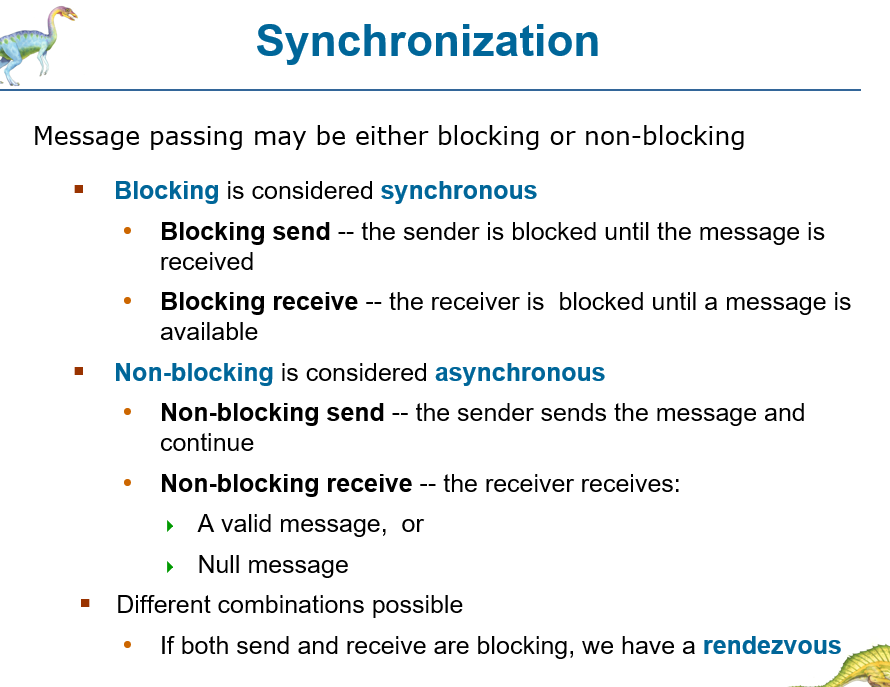
→ solutie:

→ 3 semafoare ‘mutex’(pentru acces la buffer,initilizat la 1), ‘full’ (cate poz sunt ocupate, initilizat la 0) și’empty’(cate poz sunt libere, initilizat la n)

→Producătorii vor aștepta până când este disponibil un slot liber în buffer, iar consumatorii vor aștepta până când există cel puțin un element în buffer.

→Când un producător adaugă un element în buffer, va semnaliza un consumator, iar când un consumator scoate un element din buffer, va semnaliza un producător.





**→ READERS AND WRITERS (!citirea doar cand nu se scrie)**

**→**multiple procese sau thread-uri care încearcă să obțină acces la o resursă partajată, unde unele vor să citească sursa(readers doar citesc), alții să o scrie(writers pot să scrie/citeasca)

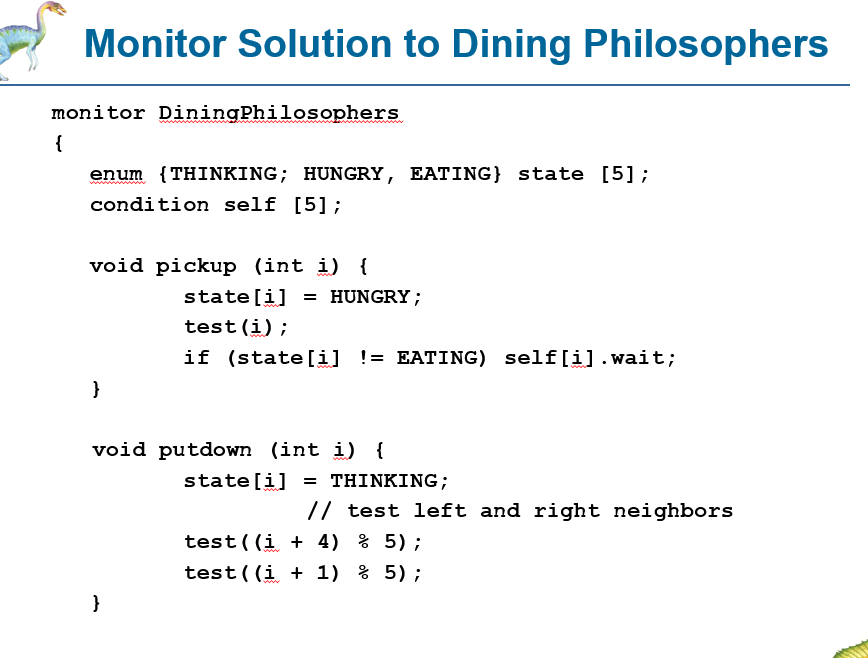
→ cerința este să se coordoneze accesele ținând cont că un singur writer poate avea acces la o zona partajată la un moment dat

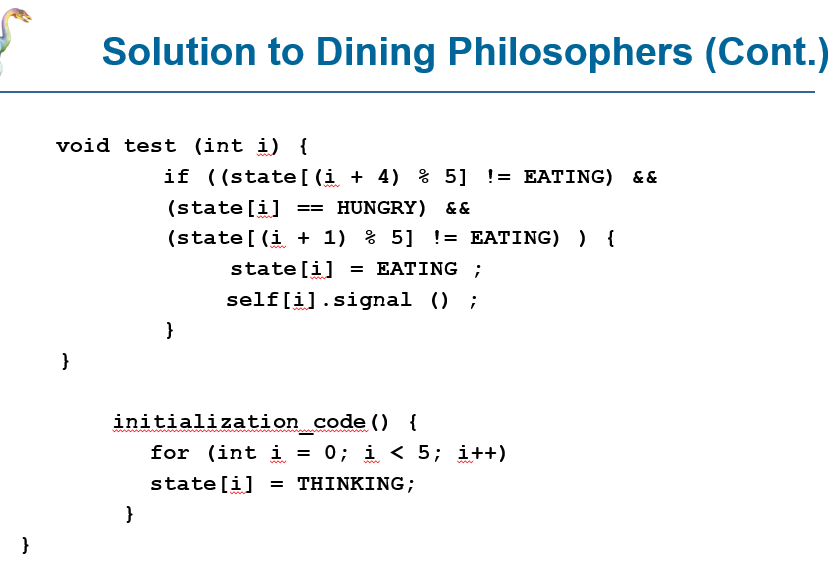
→ solutie: semafor pentru a controla accesul scriitorilor si un contor pentru a tine evidența nr de cititori curenți

→ un nou reader poate să acceseze zona partajată atât timp cât există deja un reader curent

→dacă un writer solicită accesul, acesta trebuie să aștepte până când nu mai există niciun reader curent

**→ DINING-PHILOSOPHERS**

****



**→ SLEEPING BARBER PROBLEM**

→ problemă  de concurență

→ un frizer, N scaune;

→dacă nu are clienți, acesta adoarme,

→dacă apare un client, se trezește și îl tunde,

→dacă mai mulți clienți apar în același timp, poate tunde numai unul

→ modelează astfel încât frizerul să servească clienții în ordinea în care au ajuns

→ solutie:

→ semafoare pentru coordonarea activității clienților: 1 semafor pentru clienți și 1 pentru frizer pentru a controla accesul la resurse( scaune/fotoliu)

→semaforul pentru clienți este inițializat, frizerul verifică dacă există clienți așteptând

→ dacă există clienți, eliberează semaforul și tunde clientul, astfel indică faptul că un scaun este acum liber

→dacă nu există, se culcă

*Semaphore Customers = 0;*

*Semaphore Barber = 0;*

*Mutex Seats = 1;*

*int FreeSeats = N;*

*Barber {*

*while(true) {*

*/\* waits for a customer (sleeps). \*/*

*down(Customers);*

*/\* mutex to protect the number of available seats.\*/*

*down(Seats);*

*/\* a chair gets free.\*/*

*FreeSeats++;*

*/\* bring customer for haircut.\*/*

*up(Barber);*

*/\* release the mutex on the chair.\*/*

*up(Seats);*

*/\* barber is cutting hair.\*/*

*}*

*}*

*Customer {*

*while(true) {*

*/\* protects seats so only 1 customer tries to sit*

*in a chair if that's the case.\*/*

*down(Seats); //This line should not be here.*

*if(FreeSeats > 0) {*

*/\* sitting down.\*/*

*FreeSeats--;*

*/\* notify the barber. \*/*

*up(Customers);*

*/\* release the lock \*/*

*up(Seats);*

*/\* wait in the waiting room if barber is busy. \*/*

*down(Barber);*

*// customer is having hair cut*

*} else {*

*/\* release the lock \*/*

*up(Seats);*

*// customer leaves*

*}*

*}*

*}*

**→ MONITOARE**

→ mecanism de sincronizare

→ gestionarea accesului la resurse comune in sist operare multithread

→ definește o secțiune critică în care singura instanță poate accesa o resursă dată în același timp, fiecare alt proces fiind în așteptare până când procesul curent își termină accesarea la resursa respectivă

→ protecție a resurselor

→ garantează că nu va fi accesată în mod incorect

*pthread\_mutex\_t lock;*

*pthread\_cond\_t cond;*

*int buffer = 0;*

*void \*producer(void \*arg) {*

*while (1) {*

*pthread\_mutex\_lock(&lock);*

*while (buffer != 0) {*

*pthread\_cond\_wait(&cond, &lock);*

*}*

*buffer = 1;*

*printf("Produced\n");*

*pthread\_cond\_signal(&cond);*

*pthread\_mutex\_unlock(&lock);*

*sleep(rand() % 3);*

*}*

*}*

*void \*consumer(void \*arg) {*

*while (1) {*

*pthread\_mutex\_lock(&lock);*

*while (buffer == 0) {*

*pthread\_cond\_wait(&cond, &lock);*

*}*

*buffer = 0;*

*printf("Consumed\n");*

*pthread\_cond\_signal(&cond);*

*pthread\_mutex\_unlock(&lock);*

*sleep(rand() % 3);*

*}*

*}*

*int main(int argc, char \*argv[]) {*

*pthread\_t producer\_thread, consumer\_thread;*

*pthread\_mutex\_init(&lock, NULL);*

*pthread\_cond\_init(&cond, NULL);*

*pthread\_create(&producer\_thread, NULL, producer, NULL);*

*pthread\_create(&consumer\_thread, NULL, consumer, NULL);*

*pthread\_join(producer\_thread, NULL);*

*pthread\_join(consumer\_thread, NULL);*

*pthread\_mutex\_destroy(&lock);*

*pthread\_cond\_destroy(&cond);*

*return 0;*

*}*

→ implementează un monitor simplu care controlează accesul la un buffer comun între două fire de execuție (un producător și un consumator)

**→TRANSACTIONAL MEMORY(™)**

→gestionarea accesului concurent la memoria partajată

→ tranzacții; fiecare tranzacție = lot de operații care se execută împreună

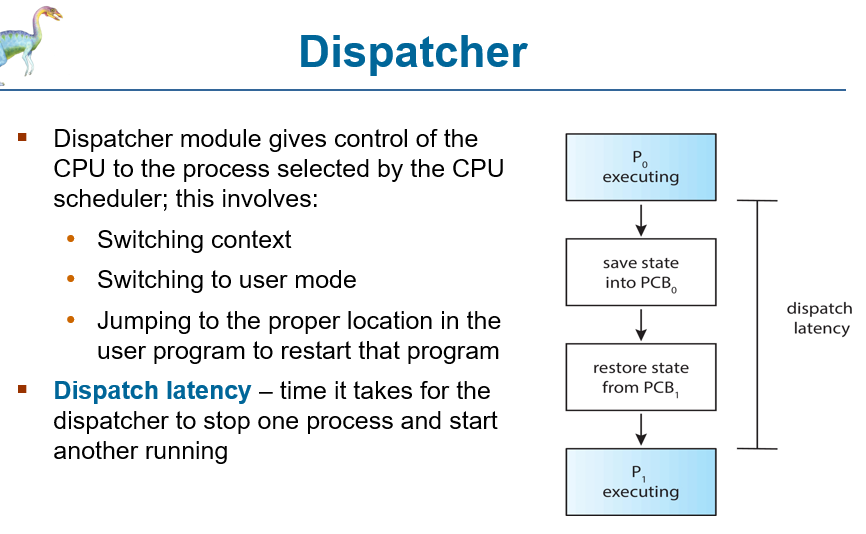
→dacă orice operație în tranzacție eșuează, toate modificările făcute de acea tranzacție sunt anulate

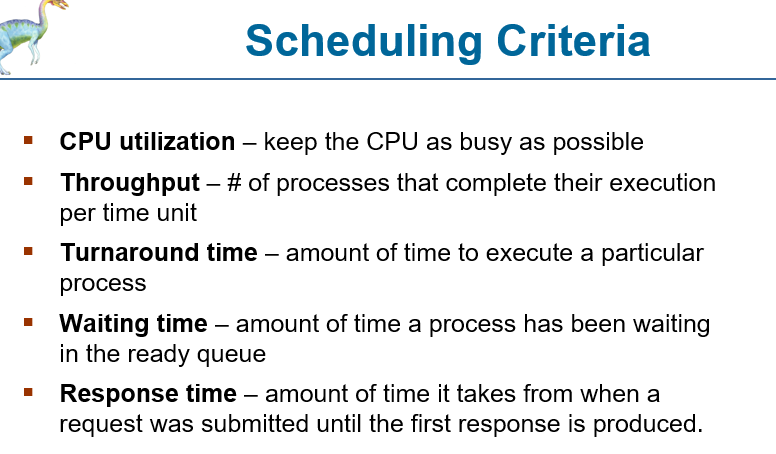
**→OPEN MP (Open Multi-Processing)**

**→**API standard pentru programare paralelă (parallel programming) care se concentrează pe a facilita programarea paralelă prin partajarea memoriilor și sincronizarea proceselor

→"thread-level parallelism", în care se creează mai multe fire de execuție (threads) care rulează concomitent, fiecare fiind responsabil cu o parte din sarcinile programului

**CURS 6 - CPU(Central Processing Unit)**





→ **CPU SCHEDULER**

**→ short-term scheduler**

**→**gestionează planificarea proceselor ce rulează pe CPU

→ oferă o utilizare eficientă a CPU

→ bună alocare a resurselor la procese și aplicații cu priorități diferite

→ scopul e să fie CPU mereu ‘ocupat’

→ nr proceselor care își completează execuția pe unitate de timp trebuie să fie maximizată

→ timpul ca un proces să-și termine execuția trebuie să fie minimizat( la fel și timpul de așteptare și timpul de răspuns)

→ criterii

→ utilizare CPU-ului

→nr de procese realizate într-o unitate de timp

→ timpul de așteptare/raspuns…

→**FIRST-COME, FIRST-SERVED(FCFS) SCHEDULING**

**→**algoritm de planificare care asigură accesul la CPU în ordinea în care procesele au intrat în sistem

→ implementat cu o coadă FIFO

→ poate duce la blocare/asteptari lungi in cazul proceselor cu durată lungă de asteptare

→**SHORTEST-JOB-FIRST(SJF) SCHEDULING**

→algoritm de programare a CPU în care procesele sunt programate în funcție de durata lor estimată sau reală

→procesele cu durata cea mai scurtă sunt programate înaintea celorlalte, astfel încât să se reducă timpul total de așteptare

→ cel mai mic timp de așteptare mediu

→ poate duce la starvation dacă apar procese mai scurte

→ folosit pentru long-term scheduling

→varianta acestui algoritm care ia în considerare durata reală a proceselor este cunoscută sub numele de Shortest-Remaining-Time-First (SRTF) scheduling.

**→ LONGEST JOB FIRST(LJF)**

→opus SJF; primul proces va fi cel cu cel mai lung timp

→timp mediu de așteptare foarte ridicat

**→PRIORITY SCHEDULING**

→ cele mai importante procese sunt executate înainte

→ cand se intalnesc procese cu aceeasi prioritate, se aplică FCFS

→ problemă starvation

**→ ROUND ROBIN**

**→** fiecare proces e executat pentru o anumită perioadă(quantum), apoi trece la urmatorul din listă

**→**daca termina mai repede, trece la urmatorul, nu asteapta quantumul

→ reduce blocarea proceselor ⇒ eficiența sistemului

**→ MULTILEVEL QUEUE**

→ folosește mai multe cozi de procese

→ fiecare coadă are o prioritate diferită

→ de ex, putem impărți în foreground(interactive) și background(batch)

– fiecare coadă folosește propriul alg de scheduling (RR, FCFS)

**→MULTILEVEL FEEDBACK QUEUE SCHEDULING**

→ asemănător Multilevel queue, dar aici procesul poate fi mutat între cozi in functie de performanta lor

→ avantaj: flexibilitate, dezavantaj: CPU overheads



**→ MULTIPLE-PROCESSOR SCHEDULING**

**→** constă in asignarea proceselor catre mai multe procesoare pentru a fi executate in paralel

→ maximizează utilizarea resurselor

→reduce timpul total de executie

**→ PRIORITY-BASED SCHEDULING**

→Procesele cu prioritate mai mare vor fi execute înaintea celor cu prioritate mai mică

→cu excepția cazului în care un proces cu prioritate mai mare începe să aștepte pentru o resursă liberă sau pentru un alt proces. În acest caz, un proces cu prioritate mai mică poate să fie executat în timp ce procesul cu prioritate mai mare așteaptă.

→ **EARLIEST DEADLINE FIRST SCHEDULING(EDF)**

**→** algoritm de planificare(dinamic) a proceselor care prioritizează după criteriul deadline-ului cel mai apropiat

→ procesul cu deadline-ul cel mai apropiat este executat înaintea tuturor

→ cel mai eficient pentru real-time

→ **VIRTUALIZATION & SCHEDULING**

→ virtualizarea permite crearea mai multor sisteme de operare virtuale(mașini virtuale) pe aceeași infrastructură hardware

→ aici scheduling va însemna planificarea proceselor pe mai multe mașini virtuale

**CURS 8 - MAIN MEMORY**

→ **MEMORY MANAGEMENT UNIT**

→ controlează accesul proceselor la memorie

→mapează adresele virtuale utilizate de procese la adresele fizice reale din RAM

→**RELOCATION REGISTER**

→gestionează locația fizică a adreselor din memoria secundară

→poate fi programat pentru a adăuga sau a scădea o valoare fixă de la fiecare adresă din memoria secundară, astfel încât să se poată realoca memoria între procese sau să se poată gestiona zone separate de memorie pentru fiecare proces

→**DYNAMIC LINKING**

→încarcarea bibliotecilor software în memoria unui sistem la runtime(nu la link-time cand se compilează codul-sursa)

→permite împărțirea codului, astfel încât să nu trebuiască recompilată întreaga aplicație

→**SWAPPING**

→atunci când procese/programe rulează simultan

→maximizează eficiența; permite un nr mai mare de procese să ruleze simultan fără a fi limitate de memoria disponibilă

→consta în transferarea proceselor(in loc de a fi in permanență în RAM)

→**PAGING**

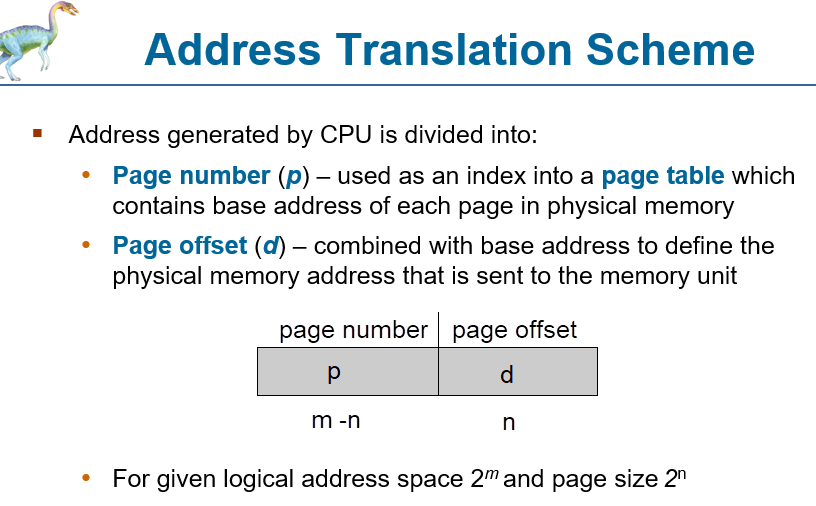
→divide memoria fizică în blocuri de dimensiune fixă ‘frames’

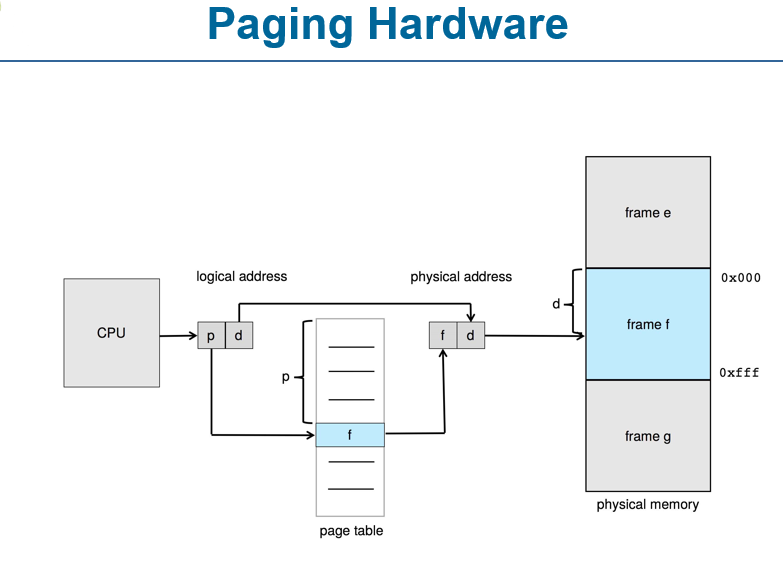
→  - | | -                 logică în blocuri de aceeași dimensiune ‘pagini’

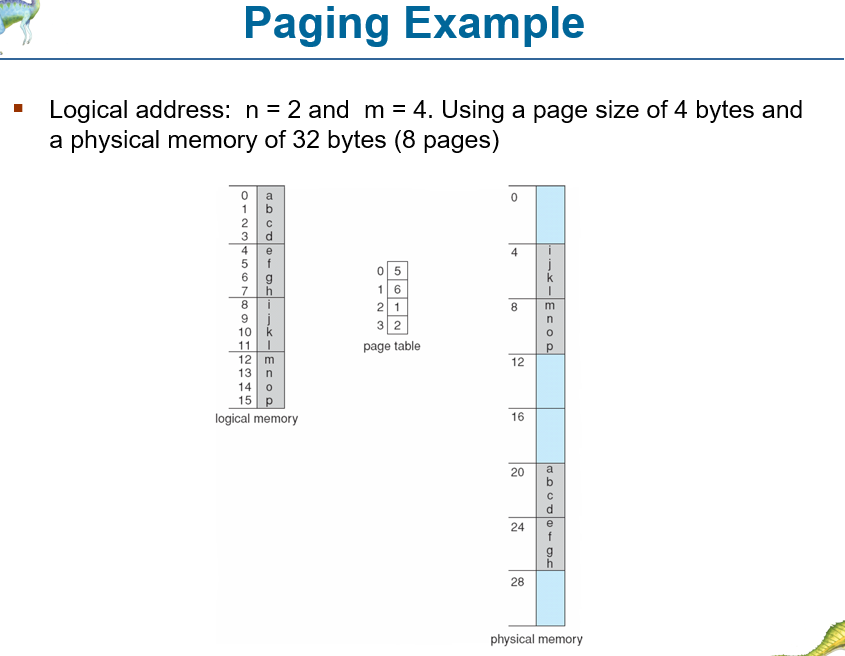
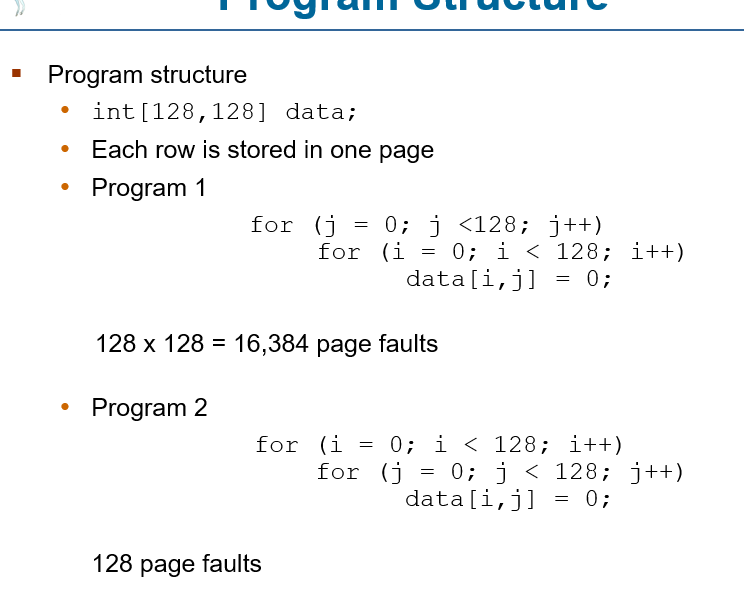
→ sistemul de operare mapează aceste pag în memoria fizică atunci când sunt necesare

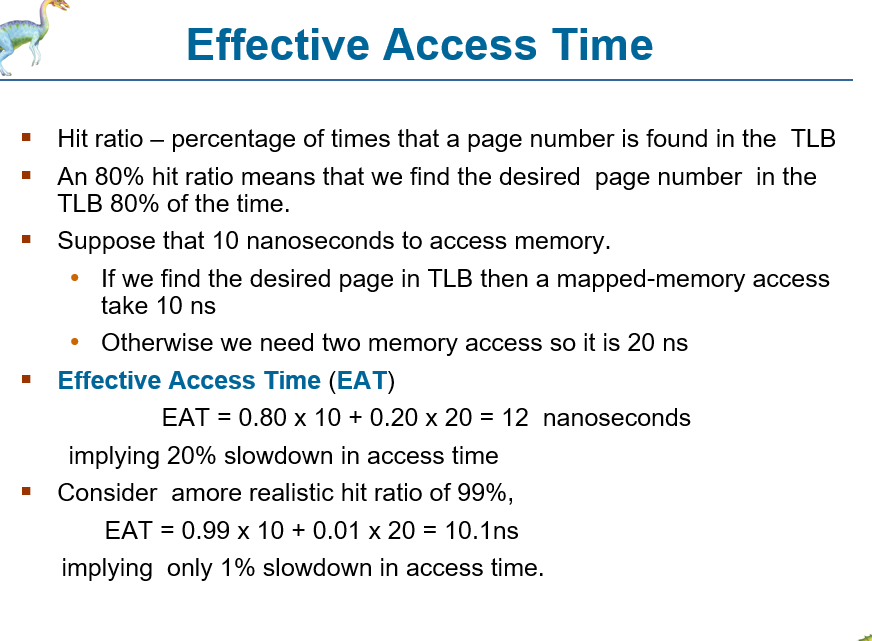
**CURS 9 - MEMORIE VIRTUALĂ**

→ separarea user logical memory de physical memory





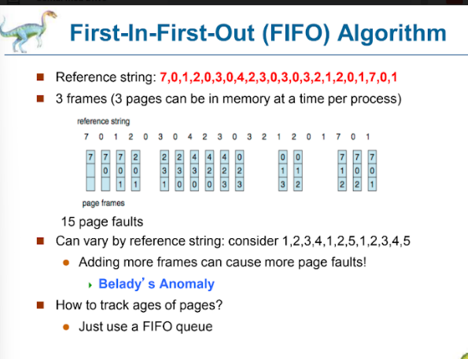
 



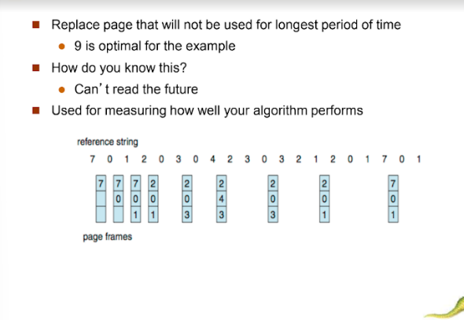
→**COPY ON WRITE**

→ crearea unei copii a unui segm de memorie când se dorește modificarea acestuia

**→ FIFO (FIRST IN, FORST OUT)**



⇒ 15 pages fault(nu număram acolo unde nu se modifică stiva!)

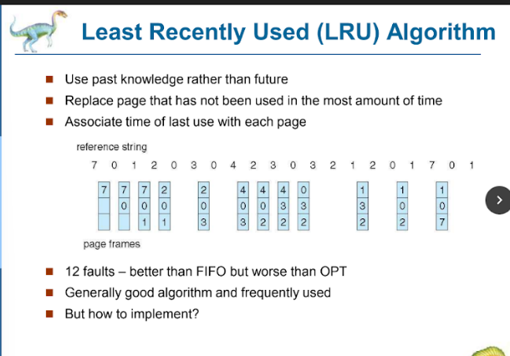
**→ALGORITM FIFO - OPTIM** **** ⇒ 9 pages fault

→replace page that will not be used for longest period of time

→!!!atunci când înlocuim, verificăm ce varianta este cel mai putin folosita in viitor si pe aceea o inlocuim, acolo unde se afla deja pe stiva→pass

(practic, aici ne uitam in viitor)

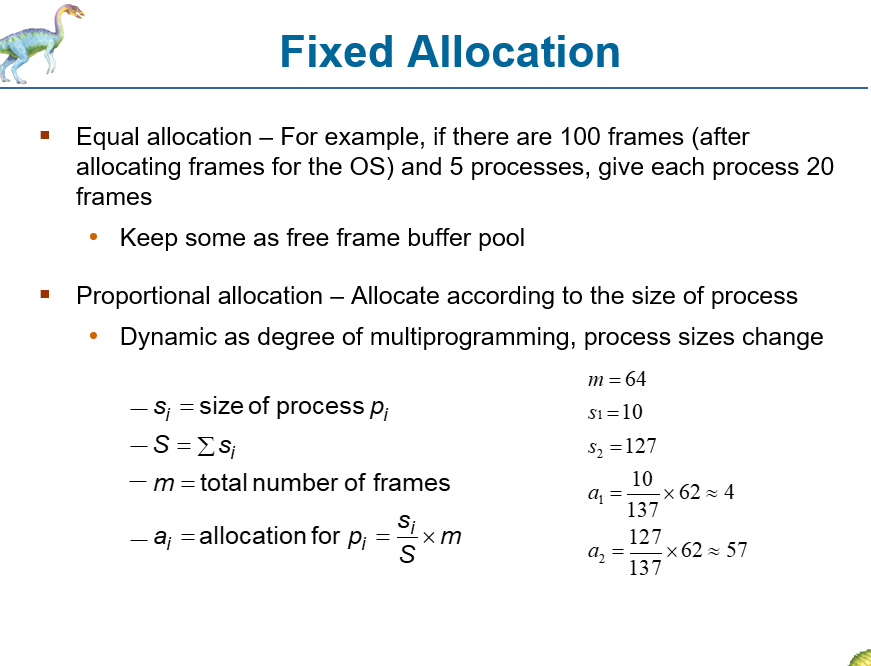
→**ALGORITM LEAST RECENTLY USED (LRU)**

 ⇒ 12 pages fault

→(mai bun decât FIFO simplu, dar mai prost decât FIFO optim)

→replace page that is least recently used

(aici ne vom uita in trecut)



**CURS 11**

→ metadatele conțin informații despre date

/proc   → arborescenta din linux pentru procese

Single Level Directory

→ Naming problem

→ directoare denumite diferit

→ folosim hash-uri(coliziune de date) ; ex: poze repetitive pe wapp, pointer spre aceeasi poza in loc de poza resalvata

IPFS(InterPlanetary File System)- protocol de comunicații și rețea descentralizată peer-to-peer pentru stocarea și partajarea datelor printr-un sistem de fișiere distribuit.

→ Grouping problem

→Mount Point

ex: stick in calculator; apare o litera-> unde a aparut acea noua partitie

*What is meant by mount point?*

*A mount point is* ***a directory or file at which a new file system, directory, or file is made accessible****. To mount a file system or a directory, the mount point must be a directory; and to mount a file, the mount point must be a file.*

→Unmount (ex: eject la stick)

**CURS 12**

Layered File System

Applications programs

|

logical file system → cum tinem directoarele, manage metadata information

→ ex: ext pt Linux;

|

file-organization mode → stiu ce e un fisier(dimensiune, unde, cand a fost creat etc.)

|

basic file system → buffere memorie si cache

|

I/O control

|

devices

PCB/TCB/FCB

→A process control block (**PCB**) is **a data structure used by computer operating systems to store all the information about a process**. It is also known as a process descriptor.

→A trusted computing base (**TCB**) is **everything in a computing system that provides a secure environment for operations**. This includes its hardware, firmware, software, operating system, physical locations, built-in security controls, and prescribed security and safety procedures.

→A File Control Block (**FCB**) is a **file system structure in which the state of an open file is maintained**. A FCB is managed by the operating system, but it resides in the memory of the program that uses the file, not in operating system memory.

ex: One Drive( fac un mount point)

**File-System Implementation**

→ Boot Control Block: info needed by system to boot OS

maxim 4 partitii primare

→Volume Control Block: controleaza partitia respectiva, cate blocuri, dimensiune, unde kernel s.a.m.d

BSD /bsd

Linux /boot/vmlinuz

→FCB

→ inode

→ NFTS

Partiție

* raw (seq of blocks without file system)
* root partition

Log structured File System

→ Journaling

**CURS 14- recapitulare**

PCB

|  |
| --- |
| process state |
| process number |
| program counter |
| Registers |
| memory limits |
| list of open files |
| … |

sectoare>piste>cilindrii

**Process creation:**

fork()

exec()->utilizat dupa fork pentru a înlocui memoria procesului cu un program nou(suprascris)

!cascading termination

wait() → procesul parinte asteapta terminarea procesului parinte

proces zombi = nu pot elibera pcv, nu este invocat wait()

proces orfan = un parinte se termina fara sa fie apelat wait()

**Shared Memory**

**Fire de executie**

!program counter diferit

un proces poate avea mai multe thread-uri

stiva partajata → variabile locale

Legea Amdahl’s Law

Modele de executie: Many-to-One( → lb interpretate, ex:python), One-to-One

pthread\_create → doar trimite cererea

pthread\_join ⇔ wait

TLS(thread local storage) → fiecare thread are propria zona de data

Mutual exclusion → un singur proces in zona critica

Progres

Timp finit de asteptare

* Algoritm Peterson

**Semafoare:**

* 2 op (atomice): wait(), signal()
* mutex⇔ semafor cu S = 1

deadlock ⇔ 2 sau mai mule procese asteapta

starvation⇔infinite blocking(ex: probl dining philosophers)

RR → parcurge lista cu un time quantum (!!!! daca termina mai repede, trece la urmatorul, nu asteapta quantumul)

push(impinge informatii catre destinatie) vs pull(trage informatii de la sursa)

EARLIEST DEADLINE FIRST SCHEDULING(EDFS)

Memorie virtuala

Algoritm optim fifo

Thrashing → mult deodata si se opreste executia

→ încearcă să aloce mai multă memorie decât este disponibilă

REZOLVARE MODEL EXAMEN 2019

* VARIANTA 1

1. a. Ce este un proces zombie?

Un proces zombie este un proces terminat, dar care nu a fost complet curatat de sistemul de operare. Acest proces nu mai ruleaza, dar ramane in lista de procese si ocupa resurse de sistem, precum memoria. Un proces poate deveni zombie daca parintele sau nu a asteptat(wait()) sau nu a curatat datele procesului terminat. Solutia pentru acest tip de eroare este de a asigura ca fiecare proces isi asteapta copiii si ii curata inainte de a se termina.

b. Scrieti o secventa de cod care arata cand un proces devine zombie?

pid\_t child\_pid = fork();

if (child\_pid == 0) {

    // Proces copil

    printf("Child process with PID %d has been created\n", getpid());

    exit(0);

}

else {

    // Proces parinte

    printf("Child process with PID %d has become a zombie\n", child\_pid);

}

→ In acest exemplu, procesul copil se termină prin apelarea exit(0), dar procesul parinte nu apeleaza wait() pentru a recupera informatiile despre starea copilului terminat, astfel ca acesta devine un proces zombie.

1. for(i = 0; i<3; i++){

fork();

pthread\_create();

fork();}

1. Cate procese si fire de executie sunt create?

→ 2+2^3+2^5 fire execuție

→ 6 fork() ⇒ 2^6-1(fără procesul părinte)

1. Desenati arborescenta

     3.  probl filozofiilor → deadblock; proprietati

do{

wait(chopstick[i]);

wait(chopstick[(i+1)%n]);

/\*...\*/

signal(chopstick[i]);

signal(chopstick[(i+1)%n]);

}while(true);

1. Modifica solutia ridicand asimetric betisoarele; filozofii impari ridica intai betisorul din stanga, cei pari pe cel din dreapta. Arată ca nu mai apare fenomenul.

Solutia problemei “Dining Philosophers” este ridicarea bețișoarelor asimetric, astfel evitându-se deadlock-ul- filozofii impari ridica intai betisorul din stânga, cei pari pe cel din dreapta.Acest lucru poate fi implementat prin adaugarea unei conditii if inainte de a astepta fiecare betisor:

do{

if (i % 2 == 0) {

wait(chopstick[(i+1)%n]);

wait(chopstick[i]);

} else {

wait(chopstick[i]);

wait(chopstick[(i+1)%n]);

}

/*...*/

if (i % 2 == 0) {

signal(chopstick[(i+1)%n]);

signal(chopstick[i]);

} else {

wait(chopstick[i]);

wait(chopstick[(i+1)%n]);

}

}while(true);

1. Arătați dacă noua soluție satisface cele trei proprietăți: exclusivitate mutuală, progresul de timp și timp finit de așteptare.

→ exclusivitate mutuala(capacitatea de a garanta ca doar un filozof poate accesa furculitele la un moment dat)- wait/signal; Când un filozof dorește să ia în considerare două furculițe (chopstick[i] și chopstick[(i+1)%n]), el așteaptă pe ambele cu ajutorul apelului "wait". Astfel, în cazul în care alți filozofi încearcă să ia aceleași furculițe, vor aștepta până când primul filozof eliberează furculițele prin apelarea "signal".

→progres de timp: niciun filozof nu va fi blocat pentru totdeauna( nu e deadlock); această proprietate este satisfăcută deoarece un filozof nu poate ridica ambele betisoare (stânga și dreapta) simultan, ci doar un singur betisor la un moment dat. Acest lucru înseamnă că orice filozof care așteaptă un betisor poate fi sigur că va fi capabil să obțină unul în cele din urmă, astfel încât să poată termina de mâncat.

→ timp de așteptare: finit-mecanismul de așteptare-semnalare (wait-signal[op atomice]) utilizat pentru a gestiona accesul la tacâmuri; fiecare filozof nu trebuie să aștepte pentru mâncare pentru o perioadă nedeterminată de timp.

      4. matrice, 3 frame uri, pagini

P1:

for( i = 0; i < 10; i++)

for(j = 0; j < 10; j++)

A[i][j] = 0;

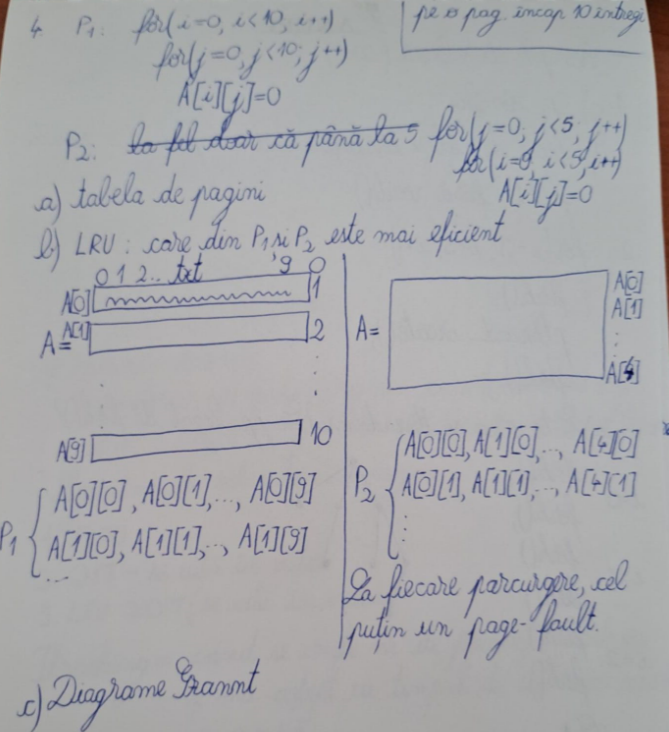
P2:

for( j = 0; j < 5; j++)

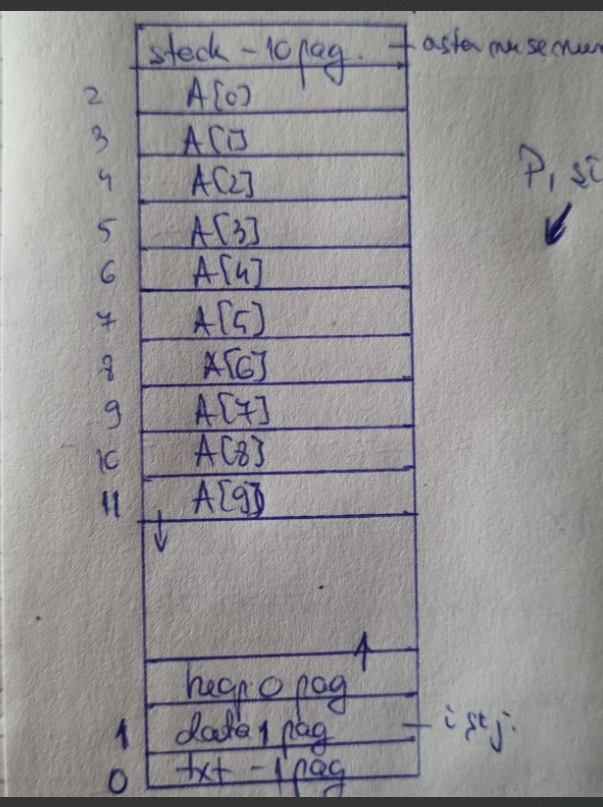
for( i = 0; i < 5; i++)

A[i][j] = 0;

1. Cum arata programul si datele repartizate pe pagini?

****

1. Folosind alg LRU(Least Recently Used) care este programul eficient?
2. Cum arată diagramele Gannt pt P1, P2?



Imparti intai in pagini si o sa ai 11 pagini. Pe o pagina incap 10 intregi si matricea are 10 linii. O sa mai fie o linie pentru cod=> 11 pagini. Avem la dispozitie 3 frame-uri. Unul este folosit pentru pagina cu codul. Mai raman 2 la care se aplica algoritmul LRU. Incarci in primele 2 frame-uri paginile primelor 2 linii. Cand ajungi la primul element de pe linia 3 in el. Alg LRU zice ca o elimini pe cea mai demult folosita. Adica prima, si asa mai departe. Initial, nu este nimic incarcat, adica la primul element de pe prima linie nu vom avea nimic. Trebuia sa aducem linia. La fel, la pimul de pe a doua linie. Primele 2 linii nu sunt by default.

* VARIANTA 2

1. Fie urmatoarea secvență:

for(i = 0; i<2; i++)

for(j=i;j>0;j--)

fork();

1. Câte procese sunt create?

2^(nr\_apeluri\_fork())-1 ⇒ 1

1. Desenați arborescența.

2. Considerați problema filozofilor și soluția propusă mai jos pentru n filozofi:

do{

wait(chopstick[i]);

wait(chopstick[(i+1)%n]);

/\* … \*/

signal(chopstick[i]);

signal[chopstick[(i+1)%n]);

} while(true);

1. Demonstrați că fenomenul de deadlock apare pentru n = 5 filozofi

→ soluția de mai sus permite maxim n = 4

→ pentru n = 5 se produce deadlock

→ dacă toți filozofii  ridică bățul din stânga, îl așteaptă pe cel din dreapta(care a fost deja ridicat de filozoful ‘vecin’) ⇒ deadlock

1. Dacă modificăm soluția a.i. fiecare filozof poate ridica bețele doar dacă ambele sunt disponibile, dispare deadlock, dar apare starvation. Arătați de ce.

3. Fie următoarea secvență de procese care apar la diferite momente de timp t

Proces t CPU

P0 0 5

P1 0 3

P2 2 8

P3 4 1

P4 6 7

1. Cum arată diagrama Gantt rezultată în urma aplicării algoritmului Round Robin nonpreemptive? ⇔ FCFS

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| P0 | P1 | P2 | P3 | P4 |

0 5                   8                         16                         17                    24

1. Dar pentru același algoritm în modul preemptive cu o cuantă de timp q = 2?

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| P0 | P1 | P2 | P0 | P3 | P1 | P4 | P2 | P0 | P4 | P2 | P4 | P2 | P4 |

0       2      4       6       8      9       10      12      14     15      17      19      21     23   24

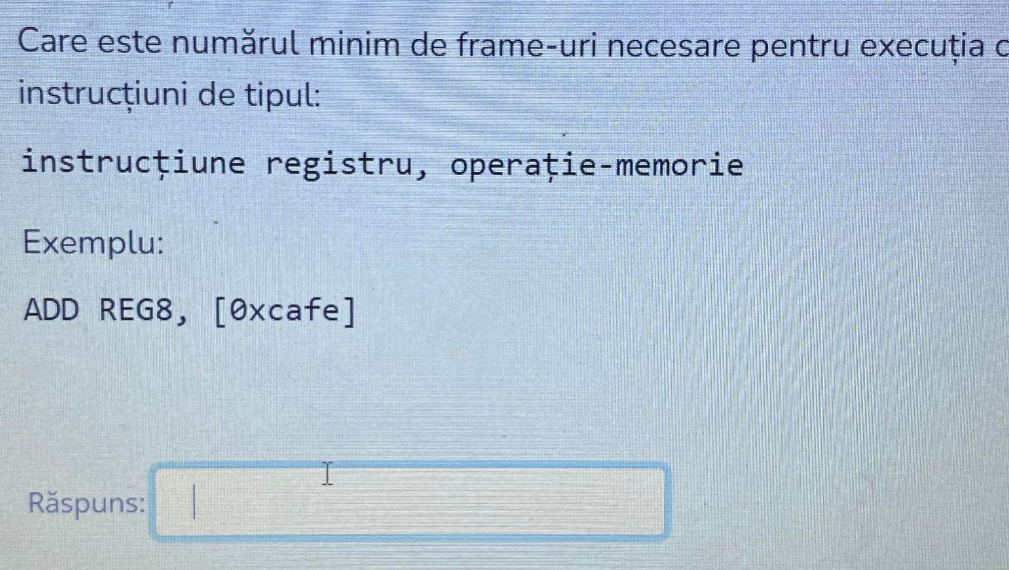
4. Care este numărul minim de frame-uri necesar pentru execuția corectă a proceselor pe un procesor cu instrucțiuni de tipul: instr reg, memop, memop(ex. add r8, [0xdead], [0xbeef]).

Si aici intre 0 si 2, pe acelasi rationament:

Ambele sunt cache-uite, 0 acces RAM direct

1 e cache-uita

Sau trebuie cautate pe ambele in RAM, deci 2 frame-uri, ca-s 2 locatii de memorie distincte, cu indexuri frame distincte poate.



Deci la asta as zice ca 1, ca tu faci un add intr un registru dintr o variabila tinuta in memorie la adresa sa zicem:

Virtual Index | Offset

Care fie ai noroc sa fie cache-uita => 0 acces RAM, aka niciun frame accesat

Fie esti mapata catre:

Frame Index | Offset

Deci minim 1 acces la un frame.

5. Fie un disk cu 5000 cilindrii și următoarea coadă de cereri I/O în așteptare 2000, 3000, 1200, 4, 2018. FIecare intrare reprezintă un cilindru, iar capul de citire al disk-ului se află la poziția 1000 și a fost înainte la poziția 314.

1. Începând de la poziția curentă, care este ordinea și distanțarea totală parcursă de cap pentru a satisface toate cererile din coadă folosind FCFS?

2000, 3000,1200,4,2018

1. Dar folosind SCAN?

2000,3000,2018,1200,4

In acest caz, se foloseste algoritmul First-Come-First-Served (FCFS) pentru a satisface cererile din coada. Astfel, capul de citire al disk-ului va parcurge urmatoarea secventa de cilindri pentru a satisface cererile:

1000 -> 2000 -> 3000 -> 1200 -> 4 -> 2018

Distantele parcurse sunt:

2000 - 1000 = 1000 cilindri pentru prima cerere

3000 - 2000 = 1000 cilindri pentru a doua cerere

1200 - 3000 = 1800 cilindri pentru a treia cerere (trebuie sa se intoarca la inceput, deci distanta totala va fi 5000 - 3000 + 1200 = 3200 cilindri)

4 - 1200 = 1196 cilindri pentru a patra cerere

2018 - 4 = 2014 cilindri pentru a cincea cerere

Distanta totala parcursa va fi 1000 + 1000 + 3200 + 1196 + 2014 = 7406 cilindri.

* VARIANTA 3

1. Puteți simula o structură de directoare pe mai multe nivele cu o structură cu un singur nivel? Daca da, explicați cum ați implementa o astfel de simulare și cum se comportă cu o structură pe mai multe nivele. Dacă nu, explicați ce vă împiedică să faceți acest lucru. Dați un exemplu.

Da, se poate simula o structură de directoare pe mai multe nivele cu o structură cu un singur nivel prin utilizarea unui sistem de nume în formă de cale (pathname). În acest caz, fiecare element din structura de directoare poate fi reprezentat prin numele acestuia, separând diferitele niveluri prin intermediul unui separator, de obicei "/" sau "", în funcție de sistemul de operare.

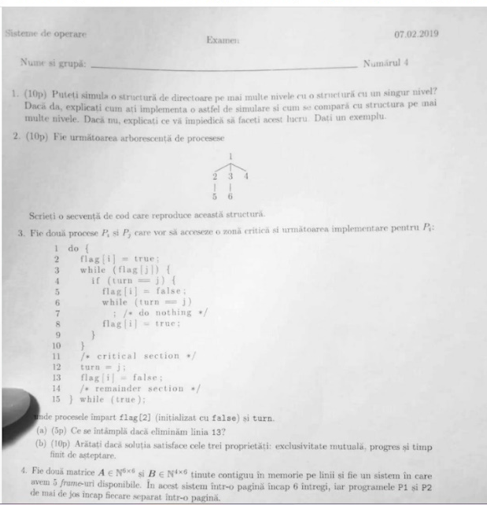
Pentru a implementa o astfel de simulare, se pot folosi mai multe câmpuri într-un registru sau structură de date pentru a ține evidența informațiilor despre fiecare element din structura de directoare. De exemplu, un registru poate conține numele elementului, tipul acestuia (director sau fișier), datele asociate cu fișierul (în cazul în care elementul este un fișier) și calea către element.

Această structură cu un singur nivel se comportă diferit față de o structură cu mai multe nivele în ceea ce privește accesarea și gestionarea elementelor. În cazul unei structuri cu mai multe nivele, elementele sunt organizate într-o structură ierarhică, în timp ce în cazul unei structuri cu un singur nivel, elementele sunt organizate linear.

Un exemplu de simulare a unei structuri de directoare pe mai multe nivele cu o structură cu un singur nivel poate fi reprezentat prin utilizarea unui sistem de fișiere pe un computer. În acest caz, fiecare fișier și director poate fi reprezentat prin numele acestuia, precedat de calea către acesta. De exemplu, "home/user/documents/letter.txt" poate reprezenta un fișier numit "letter.txt" situat în directorul "documents", care se află în directorul "user", care la rândul său se află în directorul "home".

Top of Form

1. Fie următoarea arborescență de procese:



Scrieți o secvență de cod care reproduce această structură.

1. Fie două procese Pi, Pj care vor să acceseze o zonă critică și următoarea implementare pentru Pi:

do{

flag[i] = true;

while(flag[j]){

if(turn == j){

flag[i] = false;

while(turn == j)

; /\* do nothing \*/

flag[i] = true;

}

}

/\* critical section \*/

turn = j;

flag[i] = false;

/\* remainder section \*/

}while(true);

unde procesele împart flag[2](inițializat false) și turn.

1. Ce se întâmplă dacă eliminăm linia 13?
2. Arătați dacă soluția satisface cele 3 proprietăți: exclusivitatea mutuală, progres și timp finit de așteptare.

4. Fie două matrice A și B ținute continuu în memorie pe linii și fie un sistem în care avem 5 frame-uri disponibile. În acest sistem într-o pagină încap 6 întregi, iar programele P1 și P2 de mai jos încap fiecare separat într-o pagină.

P1:

for(i=0; i< 6; i++)

for(j=0; j<6; j++)

A[i][j] = i+2\*j;

P2:

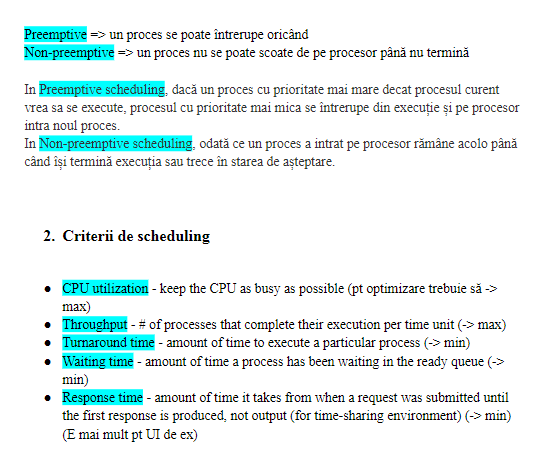
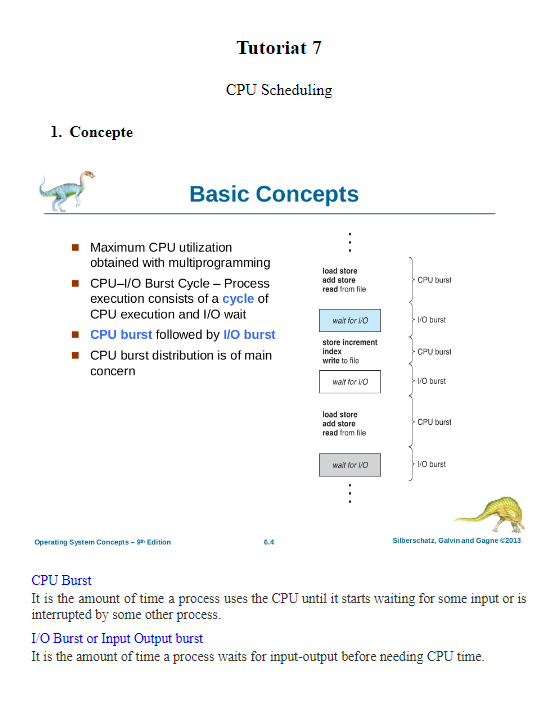
for(i=0; i<4; i++)

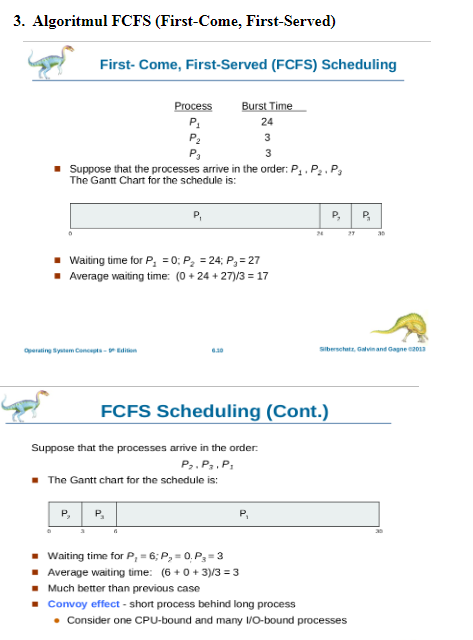
for(j=0; j<6; j++)

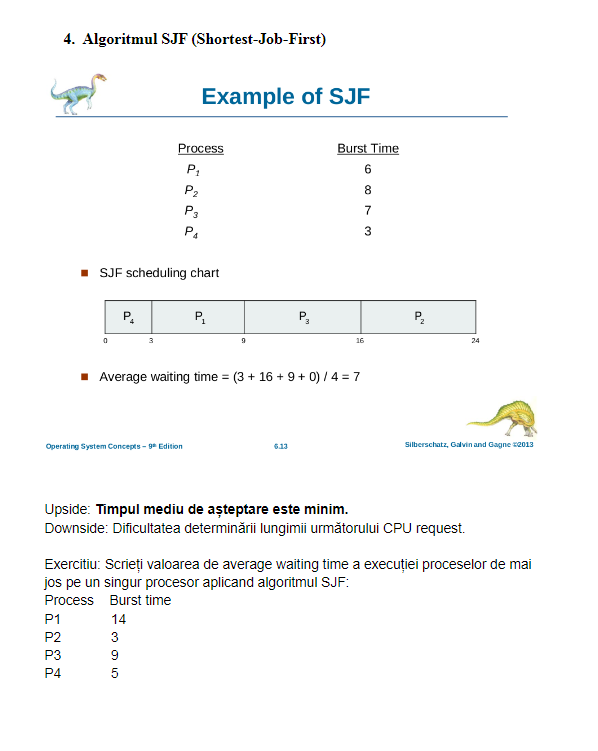
B[i][j] = i+j;

Presupunem că programele se execută concurent astfel: fiecare program stă pe procesor cât să ducă până la capăt instrucțiunea de la linia 4 o singură dată după care cedează locul concurentului.

1. Cum arată programele și datele repartizate pe pagini?
2. Cum arată diagrama Gannt folosind LRU cu o strategie de înlocuire a paginilor globală(în care ambele programe pot folosi toate frame-urile disponibile în sistem).
3. Cum ați aloca frame-urile pentru o strategie de înlocuire a paginilor locală(în care fiecare program primește un număr fix de frame-uri pe care doar el le poate utiliza). De ce?







**EDF**

Exemplu:

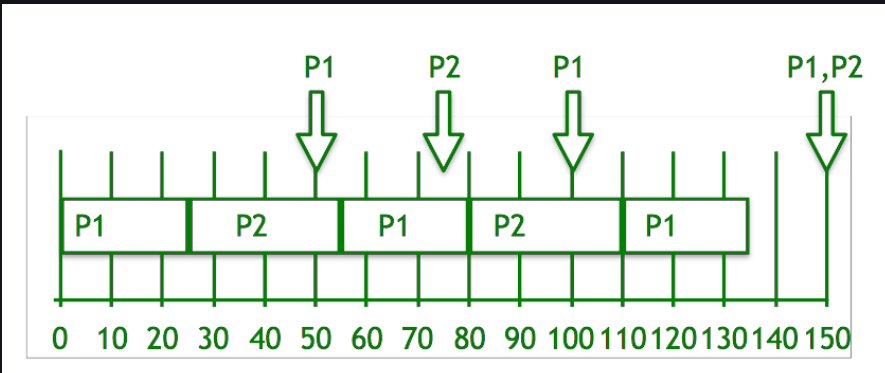
Luați în considerare două procese P1 și P2.

Fie perioada lui P1 p1 = 50

Fie timpul de procesare al lui P1 t1 = 25

Fie perioada lui P2 perioada2 = 75

Fie timpul de procesare al lui P2 t2 = 30



Pasi pentru rezolvare:

Termenul limită pentru P1 este mai devreme, deci prioritatea P1>P2.

Inițial, P1 rulează și își finalizează execuția de 25 de timp.

După 25 de ori, P2 începe să se execute până la 50 de ori, când P1 este capabil să se execute.

Acum, comparând termenul limită al lui (P1, P2) = (100, 75), P2 continuă să se execute.

P2 își finalizează procesarea la momentul 55.

P1 începe să se execute până la ora 75, când P2 poate să se execute.

Acum, comparând din nou termenul limită al lui (P1, P2) = (100, 150), P1 continuă să se execute.

Repetați pașii de mai sus...

În cele din urmă, la momentul 150, atât P1, cât și P2 au același termen limită, astfel încât P2 va continua să se execute până la momentul de procesare după care P1 începe să se execute.

