Sisteme de Operare

Exceptii si intreruperi

Exceptiile sunt intreruperi *software* ale unui program cauzate de **requestul unui user** sau de o **eroare**.

Exemple: *division by zero, segmentation fault*.

Rolul unei intreruperi este de a *opri* executia normala a unui program pentru a gestiona evenimente sau conditii specifice care necesita **atentie imediata** sau **procesare prioritara**.

Cum raspunde kernelul la o **intrerupere hardware**?

CPU intra in **kernel mode**, stocheaza **starea sa interna** in registrii. In plus, se memoreaza si **tipul exceptiei** intr-un registru.

Apeleaza un **handler general** pentru a gasi *(folosind registrul cu exceptia ca index)* un **handler specific** care poate trata exceptia din cod.

Acum ca avem **handlerul**, putem rula **cod C,** iar **kernel-ul**:

- Decide daca exceptia e **fatala**, handlerul **termina** executia procesului
- Daca **nu**, **inlatura** exceptia, **reda controlul** handlerului de nivel jos si **restaureaza datele** initiale in CPU

Paginare vs segmentare:

Paginare:

- pagini de dimensiune fixa

- cauzeaza fragmentare interna(spatiu nefolosit in pagina)

- invizibila

pentru programator(gestionata de hardware/OS)

Segmentare:

- segmente de dimensiune variabila

- cauzeaza fragmentare externa(spatii libere intre segmente)

- vizibila pentru programator(segmentele sunt definite logic)

Paginarea e ***eficienta pentru gestionarea fizica*** a memoriei, **ignora** logica programului.

Segmentarea ***respecta strucutra logica*** a programului, dar e **greu de gestionat** din cauza dimensiunilor variabile.

Fragmentare

Fragmentarea semnaleaza ca memoria **nu** este optimizata (spatiul devine fragmentat in blocuri mici si discontinue, astfel, se pierde memorie).

Fragmentarea interna se produce **in interiorul unui program**, cand unui proces i se aloca mai *multa memorie* decat are nevoie.

Fragmentarea externa are loc **in cadrul sistemului**, cand memoria *nu este contigua* si *exista multe blocuri mici* de memorie.

Segmentare

Segmentarea e procesul cand memoria logica **se imparte** in segmente cu dimensiune **variabila**, devenind ca un tablou **unidimensional**.

Segment table consuma mai putin spatiu decat **page table**.

Segmentarea ofera user-ului viziunea procesului pe care **paginarea nu o ofera**.

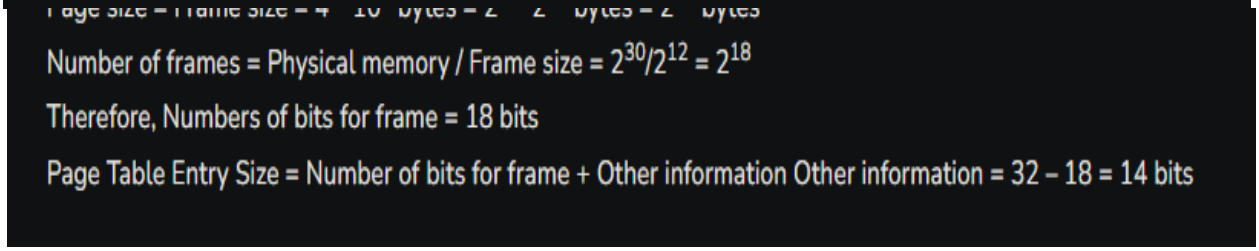
Paginare

Paginarea este o schema de **gestionare a memoriei** care *elimina necesitatea* unei alocari **contigue** a memoriei fizice. Procesul de recuperare sub forma de pagini din stocarea secundara in memoria principala este cunoscut ca **paginare**.

Frame este o unitate **fizica** de stocare, iar *pagina este plasata* intr-un **frame**.

Poate elimina **fragmentarea externa** (imparte memoria in cadre fizice de dimensiune fizica, astfel incat orice cadru liber poate fi alocat, indiferent de cerinta de memorie, „fara gauri” neutilizabile).

Physical Address Space = 64MB = 2^{26}B
Virtual Address = 32-bits, ∴ Virtual Address Space = 2^{32}B
Page Size = 4KB = 2^{12}B
Number of pages = $2^{32}/2^{12} = 2^{20}$ pages.
Number of frames = $2^{26}/2^{12} = 2^{14}$ frames.
∴ Page Table Size = 2^{20}×14-bits ≈ 2^{20}×16-bits ≈ 2^{20}×2B = 2MB.



Present (valid) bit este un **indicator** in cadrul paginarii in SO. Acest bit este asociat cu fiecare pagina alocata in *spatiul de adrese virtual al unui proces* si indica daca pagina **este sau nu incarcata** in RAM.

Un **tabel de pagini inversate (IPT)** este o structura de date utilizata pentru a **mapa** paginile de *memorie fizica* la paginile de *memorie virtuala*.

IPT este folosit pentru **a gestiona** spatiul de schimb si pentru **a permite** procesului sa utilizeze **mai multa memorie** decat este **disponibila fizic** in RAM, permitand astfel **functionarea eficienta** a sistemului de operare si a aplicatiilor.

Memorie virtuala

Virtual address space este o camera de adrese care se refera de obicei la referinta diferitelor sloturi de memorie virtuala alocate diferitelor procese.

Avantaj: numai o parte din program trebuie sa fie in **memoria de executie**, pentru un proces mai eficient, **spatiul de adresa logica** poate fi **mai mare** decat cel **fizic**, cu ajutorul **demand pagining/demand segmentation.**

Demand pagini este tehnica folosita in *sisteme de memorie virtuala* unde paginile **intra in memoria principala** cand sunt **chemate** sau **CPU**-urile **au nevoie** de ele.

Page-fault este momentul cand pagina **nu** e gasita in main memory.

Kernel-ul face o **intrerupere** cand intampina **page-fault**, **trateaza intreruperea** si **afla cauza**, se *aloca pagina in tabel, se updateaza* tabelele si *se continua* executia.

Solutii restartare instructiuni:

- Microcodul CPU **stocheaza starea interna pe stiva** cand apare page-fault.
- Microcodul CPU **are capacitatea de roll-back** la starea initiala.

Performance of Demand Paging

Effective Access Time (EAT) = (1-p) * Memory Access Time + p * Page Fault Time

Page Fault Time = page fault overhead + swap out + swap in + restart overhead

---

Procese

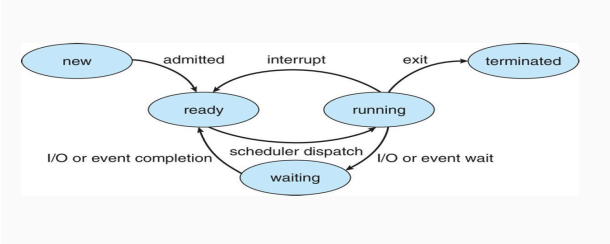
Procesul este o instanta a programului in executie. Programul devine **proces** atunci cand un fisier executabil e incarcat in memorie.

Programul este o entitate *pasiva* stocata pe disk, iar **procesul** este o entitate *activa*.

Procesele multitasking sunt tratate de parca ar avea un *CPU propriu*.

Un proces contine mai multe parti: text section, heap, stack, data section, program counter.

Un proces/thread are *5 stari*: nou, terminat, in executie, pregatit si in asteptare.



Process Scheduler alege dintre procesele disponibile pentru urmatoare executie pe CPU.

Context switch se intampla cand trecem de la un proces la altul.

Daca costul de creare al proceselor e prea mare, aplicatiile pot sa nu foloseasca *eficient* procesoarele. Daca timpul de *creare* al unui proces e ***comparabil*** cu timpul de *filtrare* al unui cadran, e posibil ca filtrarea intregii imagini cu 4 procese *sa dureze mai mult* decat filtrarea ei cu un *singur* proces(„*slowdown*”).

Threads

Un **thread** este un *punct de executie* cu **context redus** in cadrul programului.

Fiecare **thread** contine: *structuri de date necesare* (stiva) si *o copie a contextului de executie a aplicatiei* (CPU).

Beneficii: mai ieftine decat procesele, thread switching < context switching, iar procesorul poate sa se foloseasca de arhitectura multicore.

Este mai ieftin sa schimbi 2 threads decat 2 procese, deoarece nu se mai schimba si **PCB** (Process Control Block), doar **TCB** (Threads Control Block).

Threads ruleaza ***asincron*** si pot *pierde procesorul*, deci, accesul la datele partajate trebuie ***sincronizat*** cand se doreste IPC.

Avantaje threaduri kernel fata de procese:

COMUNICARE / **PARTAJARE DATE**: Datele din procesul in care exista threadul sunt date gloabale(nu e nevoie de IPC, se obtin direct).

RESURSE: Este mai rapid sa facem switch intre threaduri decat intre procese.

RESURSE: Threadurile consuma mai putine resurse cand sunt create decat un process. Cand creezi un procez, *trebuie creat PCB*, dar cand creezi un thread, *PCB ramane la fel*, doar **TCB** este schimbat.

Avantaje thread user fata de cele kernel:

PORTABILITATE: Threadurile user pot fi implementate (portate) mai usor in alte sisteme de operare decat cele kernel.

RESURSE: Cand facem context switch, este bine sa facem pe cele user, decat kernel, fiind **overhead mai mic** (consuma mai putine resurse).

Dezavantaje threads user si cum pot fi rezolvate:

-Nu putem beneficia de **paralelism** cand folosim threaduri user deoarece kernel-ul gestioneaza procesul ca un singur thread, iar threadurile user nu pot rula simultan pe mai multe nuclee.

-**Solutie**: am putea mapa threadurile user la threaduri kernel, asa beneficiem de parallelism

-Nu se **coordoneaza** bine cu kernelul

-Intr-un program in care sunt folosite semnale, acestea sunt **destinate** pentru **PROCESE**, nu **THREADS**, ceea ce ar crea probleme in executie.

-**Solutie**: Desemnam un **singur** THREAD care se ocupa de semnal.

Threadurile kernel sunt create prin thread_create(syscall), iar daca un proces are multe threads, va fi costisitor sa faca de fiecare data apelul(nu se coordoneaza bine cu kernel).

Scheduler activation

Scheduler activation reprezinta ***aproximarea*** unui thread kernel.

Are **stive kernel** si **user**, ofera conceptul de ***upcall*** pentru evenimente, fiecare ***upcall*** creeaza o noua activitate.

Upcall este un *mecanism de comunicare* intre ***kernel*** si ***upcall handler****.*

Tipuri de **upcalls**:

- ***Adauga procesor*** – consecinta este executia unui thread user
- ***Procesor preemtat*** – adauga threadul user care se executa in activarea care a pierdut CPU in coada de threads gata de rulare
- ***Activare blocata*** – activarea s-a blocat (apel sistem blocant) si nu mai utilizeaza procesorul
- ***Activare deblocata*** – pune in lista gata de rulare threadul care se executa in contextul activarii blocate

---

Semnale

Semnalele sunt utilizate in sisteme UNIX pentru a notifica un proces ca un eveniment particular a avut loc.

Un **signal handler** este folosit sa *proceseze* semnale:

- Semnalul este ***generat*** de un eveniment
- Semnalul este ***trimis*** catre proces
- Semnalul este ***gestionat*** de unul dintre signal handlers (*default/user-defined*)

Fiecare semnal are *default handler* pe care kernelul il ruleaza cand se gestioneaza semnale.

User-defined signal handler poate *suprascrie* **default handler.**

Pentru **single-threaded**, semnalul este *trimis* procesului.

Legea lui Amdahl

Identifica imbunatatirile performantei prin adaugarea de core-uri unei aplicatii care are componente in serie si paralele.

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

S – portiunea in serie

N – core-urile care proceseaza

Daca avem 75% paralel, 25% in serie, de la 1 la 2 core-uri rezulta speedup de 1.6 ori mai mare.

VFork

Vfork a fost creat pentru a eficientiza fork:

pid_t vfork(void)

Copilul **imparte aceeasi memorie** cu parintele pana acesta da exec sau exit(copilul).

A fost creat **inainte** sa fie implementat **COW**(copy on write), copilul copia intreaga memorie a parintelui si era **costisitor**, **vfork** a fost implementat pentru cazurile cand copilul trebuia sa se execute *imediat*, fara *sa copieze* memoria parintelui.

O **limitare** este ca nu trebuia *sa modifice* memoria parintelui. Orice instructiune din copil poate ***sa strice*** procesul parinte si starea acestuia. In plus, copilul **trebuie** sa dea imediat *exec* sau *exit*.

I-Node (struct)

I-Node este structura de date fundamentala care e utilizata pentru a stoca informatiile despre fisier sau director. Ajuta la **organizarea** sistemului de fisiere in SO-urile UNIX.

I-Node contine: *tipul fisierului*, *drepturile asupra fisierului* si *dimensiunea in octeti*.

I-Node este structura care face posibil sa existe fisiere memorate pe calculator.

V-Node (obiect)

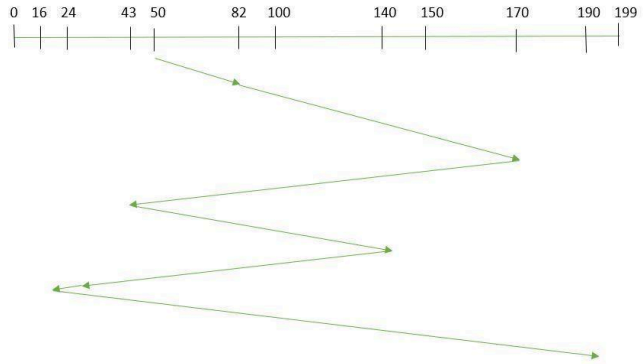
V-node este un obiect de memorie **KERNEL** care *deschide*, *citeste*, *scrie*, *inchide* si face actiuni similare pe fisiere in *sisteme UNIX*. Poate fi definit ca o *abstractizare* a **I-Node**-ului.

V-node exista doar cand fisierul este **deschis**.

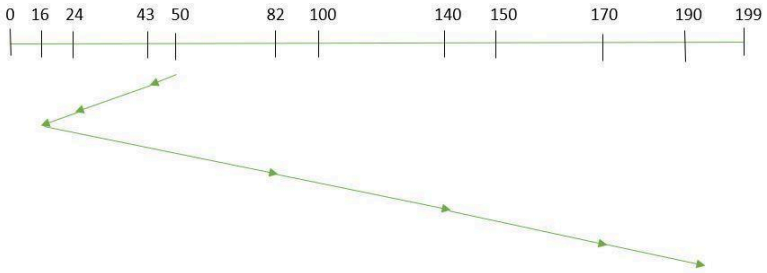
V-Node este obiectul care face posibila *citirea*/*scrierea*, *deschiderea*/*inchiderea* fisierelor. (Parte dinamica)

Disk Scheduling Algorithms

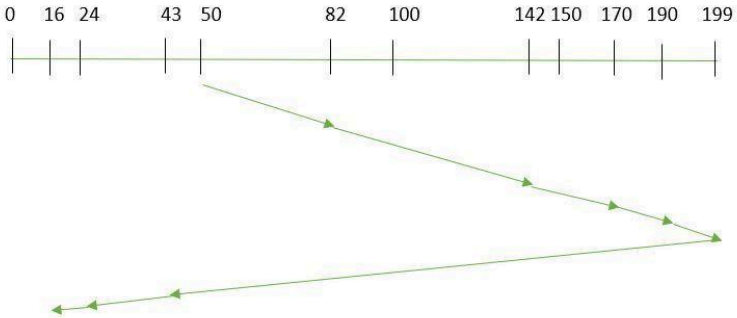
FCFS (First Come First Serve) – se ia in ordine



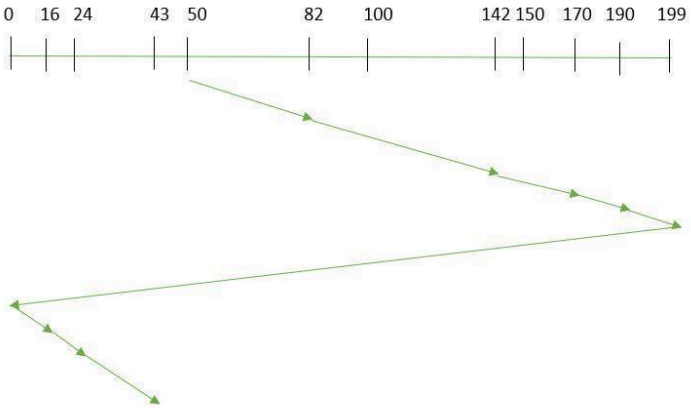
SSTF (Shortest Seek Time First) – se duce la cel mai aproape



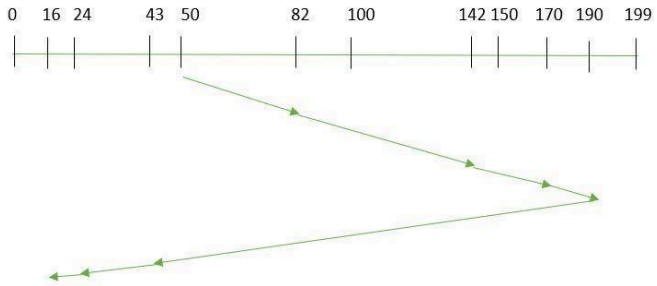
SCAN – se duce in pana la final (trece prin date), apoi pe rand pana la cel mai mic



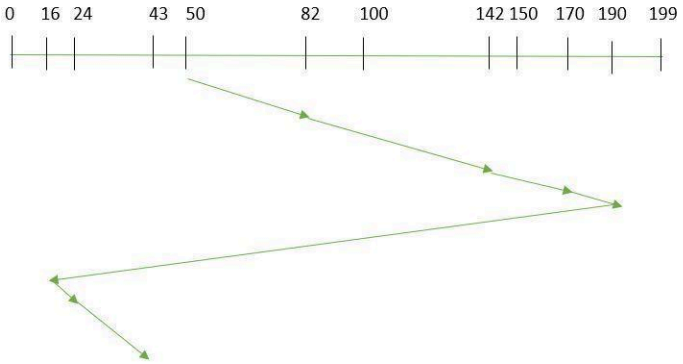
C-SCAN – se duce in dreapta (la final chiar daca nu e nevoie) si apoi in stanga la 0 si apoi urca



LOOK – se duce la maxim (prin date) si apoi se duce pana la minim (prin date)



C-LOOK – se duce la maxim(trece) apoi se duce la minim(nu) si urca pana la ultimul(trece)



Problema cu pagini

Spatiu de adrese pe 43 biti => Spatiul are capacitatea de 2^43 bytes.

*Pagini de 8 KB => 8 * 1028 byte => 8 * 2^10 byte = 2^ 13 bytes*

PTE uri (intrari in tabela de pagini) de 8 bytes.

10 KB de text și date

5 KB de stiva ramasi

Nr pagini = (spatiu de adrese) / (marimea unei pagini)

Nr pagini = 2^43/2^13 = 2^30 pagini

PTE-uri de 8 bytes si 2^30 pagini=> 2^30*8 bytes = 8GB (2^10 bytes = 1 KB; 2^20 bytes = 1 MB; 2^30 bytes = 1 GB)

Tabela de pagini pe 3 niveluri cu nr egal de biti pe fiecare nivel

2^43 bytes de spatiu => 43 de biti total

2^13 bytes spatiul unei pagini => 13 biti pentru pagini

Raman 43-13 = 30 biti pentru tabela

3 niveluri cu nr egal biti => 30/3 = 10 biti/nivel => consuma 2^10 bytes/nivel

Schema adresei:

[Index nivel 1(10 biti) | Index nivel 2 (10 biti) | Index nivel 3 (10 biti) | Offset (13 biti)]

Mapare:

-fiecare nivel are index de la 0 la 1023, iar ultimul index indica catre o tabela de nivel urmator

Nivel 1 (L1):

Index 1023(ultimul index, stiva e la capat) indica catre o tabela de nivel 2 (L2_1023)

Nivel 2 (L2):

Index 1023(ultimul index, stiva e la capat) indica catre o tabela de nivel 3 (L3_1023)

Nivel 3 (L3):

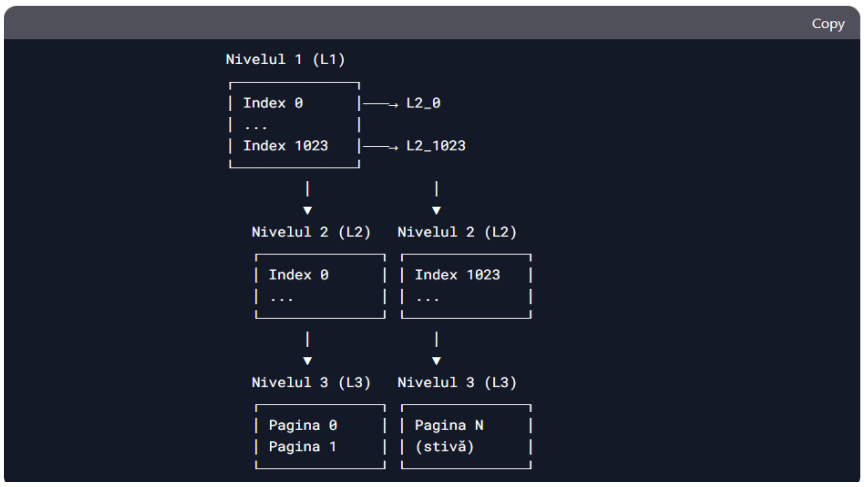
Contine intrarea pentru ultima pagina (stiva).

5. Numărul de tabele în funcție de regiuni

Componentă	Nivel 1 (L1)	Nivel 2 (L2)	Nivel 3 (L3)
Text/Date (10 KB)	1	1 (L2_0)	1 (L3_0)
Stivă (5 KB)	1	1 (L2_1023)	1 (L3_1023)
Total	**1**	**2**	**2**

Nivelul 1 are o **singura** tabela deoarece este **UNIC** pentru intregul proces. Este tabela de pagini principala, stocata intr-o singura pagina fizica de 8KB. Toate adresele virtuale ale procesului folosesc aceeasi tabela L1.

6. Reprezentare grafică



5 tabele de 8KB => 5*8KB=40 KB folositi

Implementarea tabelei liniare si memoria fizica necesara

Solutie teoretica: facem swap din memoria principala in cea secundara, sacrificand timpul de executie, cea secundara avand o viteza mai mica fata de memoria principala. In practica e aproape imposibil sa punem 8GB in 128MB RAM(fiind doar atat in memoria fizica).

Solutie realista: Folosirea unei tabele ierarhice (ca cea de sus), care reduce memoria necesara la 40KB.

Lottery Scheduling

Alocarea resursei se face prin organizarea unei **loterii**: Resursa e acordata procesului care detine tichetul **castigator**. Alocarea e **direct proportionala** cu *numarul de tichete* detinute de proces (suma lor defineste procentul total de utilizare a resursei la care are dreptul procesul).

Se ia un numar **random** de la 0 la n-1 (numarul tichetelor) folosind **10 instructiuni RISC**, apoi trece prin fiecare proces si verifica daca ***suma pana in prezent*** este **> *numarul ales*.** Daca da => **aloca** resursa, apoi repeta luand alt numar random.

Complexitate:

- **Parcurgerea listei – O(n)**
- **Arbore de intervale/echilibrat cu binary search – O(log n)**

U – utilizare totala

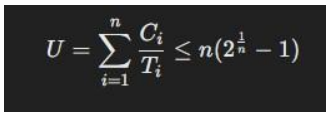
Ci – timpul de calcul al threadului *i*

Ti – perioada threadului *i*

RM (Rate Monotonic):

Se ia **cmmmc** al Ti si se **ordoneaza *crescator*** in functie de **Ti**, iar la fiecare Ti ,fiecare proces trebuie sa se execute de *Ci ori* in ordinea aflata anterior (pot aparea si spatii goale).

U = suma (Ci/Ti) < ln2



$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1)$$

EDF (Earliest Deadline First):

Ca la **RM**, doar ca **prioritatea** se face in functie de **cel mai apropiat deadline**. Cand se reia iar deadline-ul x, el devine 2x pentru ca trebuie sa ajunga la urmatorul deadline.

U = suma (Ci/Ti) <= 1

LSF (Least Slack First):

Se calculeaza pentru fiecare thread **slack = deadline – start time - remaining time**

Se calculeaza **mereu** slack-ul si se observa cel cu slack-ul **cel mai mic** si se executa pana altul devine cu slack **mai mic**.

Exercitiu

8. (14 pcte) O aplicatie multimedia contine trei thread-uri de timp real: T_v afiseaza stream-ul video iar T_s si T_d reprezinta canalele stang si respectiv drept ale sursei de sunet stereo. Caracteristicile de timp in milisecunde ale acestor thread-uri sunt:

Thread	Perioada	Timp de calcul
T_v	100	50
T_s	10	2
T_d	10	2

(a) (3 pcte) Se pot planifica aceste thread-uri conform algoritmului Rate Monotonic (considerati ca ln 2 = 0,69)? Comentati rezultatul obtinut (explicati scurt rezultatul, nu va marginiti sa raspundeti sec da/nu).

(b) (3 pcte) Se pot planifica aceste thread-uri folosind algoritmul Earliest Deadline First? Comentati rezultatul obtinut.

(d) (8 pcte) Fie doua thread-uri de timp real cu parametrii de timp exprimati in milisecunde ca in tabela de mai jos.

Thread	Timp de sosire	Deadline	Timp de calcul
T_1	0	17	10
T_2	2	10	5

Presupunand o cuanta de timp de 1 milisecunda, trasati diagrame de timp pentru executia acestor thread-uri conform algoritmilor de planificare Earliest Deadline First, respectiv Least Slack First (planifica primul thread-ul cu cel mai mic **slack**, unde slack este durata maxima de timp cu care un thread poate fi intarziat fara sa piarda deadline-ul).

a) Pentru a aplica RM, trebuie sa verificam inegalitatea:

U < n*(2^(1/n)-1)

U = utilizarea totala, adica suma raporturilor (timp calcul) / (perioada):

U = 50/100 + 2/10 + 2/10 = 0.9 < ln 2 (~0.69) **FALS** (nu putem aplica RM)

b) Earliest Deadline First: Conditia este U <= 1.

0.9 <= 1 => **Da**

c)

Thread	Timp de sosire	Deadline	Timp de calcul
T_1	0	17	10
T_2	2	10	5

Earliest Deadline First:

Timp = 0; Incepe executia T1 pana timp=2, ruleaza de la 0-2. A mai ramas 8 timp de calcul.

Timp = 2; Incepe executia T2 pana la timp 2+5. Ruleaza de la 2-7 (7 < 10 deadline)

Timp = 7; Incepe executia T1 pana la timp = 7+8(ramasi) = 15 (15 < 17 deadline)

Least Slack First (LSF)

Slack First = Deadline – Timp curent – Timp de calcul ramas

Timp = 0; Avem doar primul thread. Slack = 17 – 0 – 10 = 7

Timp = 2; Apare threadul 2. Slack = 10 – 2 – 5 = 3

Cum 3<7 => thread_2 ia CPU

Executa pana la timp = 2 + 5 = 7 (de aici preia thread_1)

IPC (Inter-Process Communication)

Procesele pot fi *independente* sau *cooperatoare*(au nevoie de IPC).

Tipuri de IPC: ***SHARED MEMORY*** si ***MESSAGE PASSING.***

Linkuri de comunicare:

Fizice – hardware bus, network, shared memory

Logice – direct/indirect, sincron/asincron

Pipes

Ordinary – relatie copil-parinte, comunicare unidirectionala

Named pipes – fara relatie, mai puternice si comunicare bidirectionala

---

Socket

Socket este un endpoint pentru comunicare.

3 tipuri: *Connection-Oriented* (TCP), *Conectionless* (UDP), *MulticastSocket* (class – datele pot fi trimise catre mai multi receptori)

RPC (Remote Call Procedure)

RPC este un mecanism princare un program/proces apeleaza functiile sau metode care ruleaza in **alt spatiu de adresa**, de obicei pe alt calculator sau in alt proces.

Avantaj fata de cele normale: model de programare centralizat, nicio diferenta intre apelurile locale si cele la distanta.

Un **stub** este o componenta software intermediara intre *programul-client* si *programul-server*.

Call by value si **copy/restore** sunt bune cu RPC, dar nu si **call by address**.

Protocoale de transport: **TCP** si **UDP**

Semantica RPC: exactly once, at least once, at most once.

De ce este **greu de implementat** *exactly once*?

- *Server crash*: rezolvare dificila, server-ul poate crapa inainte de servirea cererii, dupa servirea ei, dar si inainte de trasmiterea unui raspuns sau dupa transmiterea unui raspuns care se pierde.
- *Rezolvare*: *exterminarea*, *reincarnarea, reincarnarea delicata*, *expirarea* (nicio solutie nu e satisfacuta).

Structurarea sistemelor de operare

Monolithic Structure – Original Unix (Linux), **toate** serviciile si functionalitatile **implementate** in kernel.

- Comunicare rapida intre module, overhead mic al apelurilor intre module
- Numar mare de syscalls
- Program mare, greu de inteles

Microkernel (Mach), gestiunea proceselor si IPC **ruleaza** in kernel.

- Numar mic de syscalls
- Mai secure si reliable
- Kernel mic
- Comunicare costisitoare intre module
- Nu e performant

Loadable Kernel Modules (LKMs)

Abordare **OOP**, fiecare core e separat, similar cu layers.

Exemple: Linus, Solaris

System Boot

Bios este inlocuit de **UEFI** in sistemele moderne.

Grub este cel mai comun bootstrap loader.

PCB

PCB este o *structura de date* folosita pentru a *gestiona* si *pastra* informatii despre fiecare proces in executie.

PCB contine: PID, prioritatea procesului, starea si utilizarea memoriei.

Operatii idempotente

Efectul executiei *multiple* este *acelasi* cu cel al unei *singure* executii.

Exemplu *operatii idempotente*: setarea unui bit, citirea primului bloc de date dintr-un fisier.

Contra-exemplu *operatii idempotente*: incrementarea unui intreg, adaugarea datelor la sfarsitul unui fisier (operatia append).

Aplicatii concurente

Unele aplicatii profita de existenta *mai multor puncte de executie simultana* in cadrul aplicatiei (pe multiprocesoare).

Exemplu: procesor cu 4 core-uri si o aplicatie de filtrare imagini care imparte imaginea in 4 cadrane, fiecare core filtrand un cadran. Avantajul este descompunerea activitatilor mari in activitati mai simple care ruleaza simultan => reducerea timpului de rulare.

Exemplu: server de retea care asteapta cereri de la client, le proceseaza si trimite raspunsuri. Cu un singur punct de executie, procesarea ia mult timp, se intarzie mult tratarea altor cereri. **Solutia** este procesarea fiecarei cereri in puncte de executie diferite.

Definitii IPC

Procese independente – procese care nu partajeaza resurse.

Procese dependente – partajeaza resurse pentru a-si indeplini obiectivele (ex: imprimanta).

Sectiune critica – portiune de cod care acceseaza o resursa partajata.

Race condition – situatie in care executia intretesuta a mai multor procese care acceseaza 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 niciun proces nu poate continua pentru ca resursele de care are nevoie sunt detinute de alt proces.

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.

Asteptare limitata – daca un proces solicita accesul in sectiune critica trebuie sa i se garanteze ca il 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.

Cerinte necesare unei solutii corecte de partajare a resurselor de catre procese concurente

Excludere mutuala, ***asteptare limitata*** si ***progres***.

<u>Problema sectiunii critice cand 2 procese acceseaza o resursa partajata in acelasi timp</u>

Rezolvare:

1. Excludere reciproca - daca procesul Pi se executa in sectiunea sa critica, atunci niciun alt proces nu poate fi executat in sectiunile lui critice.

2. Progres – Daca niciun proces nu se afla in sectiunea sa critica si acolo exista unele procese care doresc sa intre in sectiunea lor critica, apoi selectarea procesului care va intra in sectiunea critica nu poate fi amanat pe termen nelimitat.

3. Bounded Waiting - o limita trebuie sa existe de cate ori altor procese li se permite sa intre in sectiunile lor critice, dupa ce un proces a facut o cerere de intrare in sectiunea sa critica si inainte ca cererea sa fie admisa.

O **bariera de memorie** este o instructiune care *forteaza* ca orice schimbare din memorie sa fie *propagata* (vizibila) celorlalte procesoare.

- **Strongly ordered** (modificarea memoriei de catre un procesor este propagata imediat catre celelalte)
- **Weakly ordered** (modificarea nu este propagata imediat)

Semafoare

Au **contor** care numara **wakeup**-urile.

Au **coada** de procese blocate in **sleep**.

Operatia **down** *decrementeaza atomic* **contorul** daca este **> 0** si permite procesului sa continue. Daca contorul e 0, *blocheaza* procesul si il pune in *coada*.

Operatia **up** *verifica* daca exista procese blocate in coada. Daca da, alege unul si il **deblocheaza**, daca nu, *incrementeaza atomic* **contorul**.

Un **semafor** cu *contor initial 1* este un **mutex** (semafor binar).

Up/down generalieaza **sleep/wakeup**, dar sunt operatii **atomice**.

Observatie: **up** nu blocheaza procese!

Conditii deadlock

1. Excludere mutuala – doar un proces/thread utilizeaza o resursa la un moment dat

2. Hold & wait – procesul/threadul detine o resursa si asteapta sa obtina alte resurse detinute de alte procese/threads

3. No preemtion – resursele obtinute de un proces/thread nu pot fi confiscate fortat de la un proces/thread care le detine, ci doar cedate voluntar la sfarsitul taskului.

4. Circular wait – un set de procese/threads aflate intr-un lant de asteptare (Pi asteapta o resursa detinuta de Pi+1).

Toate 4 trebuie satisfacute!

Inversiunea de prioritate

Reprezinta o situatie in care un proces cu o *prioritate mai mica intarzie* sau *blocheaza* executia unui proces cu o *prioritate mai mare*.

Fie P1(prioritate mare), P2(prioritate medie) si P3(prioritate mica).

P1 asteapta o resursa pe care o foloseste **P3** in executie, dar **P2** intervine si preia resursa inainte de a termina **P3**, astfel **P1** este intarziat.

Solutie: **priority inheritance**

Taskul care detine mutex-ul (semaforul) **mosteneste** prioritatea taskului cu prioritate mare *pe durata sectiunii critice*. Astfel, *taskul de prioritate mica primeste procesorul* in locul *taskului de prioritate medie*.

In aceste conditii, taskul de *prioritate mica* care si-a marit temporar prioritatea **termina sectiunea critica fara sa piarda procesorul**, **elibereaza mutex-ul** si **revine la prioritatea mica.**

Monitoare

Sunt o abstractizare la nivel inalt care ofera un mecanism *eficient* si *convenabil* de **sincronizare** a proceselor.

Avantajele folosirii monitoarelor sunt **evitarea folosirii incorecte** a *semafoarelor*, avand un singur semafor (mutex) si **suportul la nivel de limbaj** (synchronize).

O variabila conditie este un obiect in care un fir de executie se poate bloca pana cand un alt fir de executie notifica ca o singura conditie este indeplinita.

Se folosesc prin 2 operatii: **wait** si **signal**.

Wait – blocheaza procesul apelant si elibereaza monitorul pentru ca alte procese blocate la intrarea in monitor sa poata accesa monitorul.

Signal – trezeste procesul care a apleat anterior wait pe aceasta variabila conditie.

Mecanismul de suport hardware pentru protectia memoriei

Pentru fiecare **proces**, se aloca **base** si **limit** pentru a arata ***de unde pana unde*** are acces memoria.

Swapping

Swapping este o metoda in care se face **trecerea** din memoria **principala** in cea **secundara**. Ajuta in cazul in care sistemul **nu** are destula memorie RAM, facand un **sacrificiu la viteza** (memoria secundara fiind mai inceata decat cea principala).

Algoritmi de alocare dinamica a memoriei cu partitii fixe

First fit – se aloca primul bloc de memorie, care este mai mare decat se cere.

Best fit – se aloca cel mai mic bloc de memorie, mai mare decat se cere.

Worst fit – se aloca cel mai mare bloc de memorie disponibil.

Transformarea adresei logice in adresa fizica

Link: https://www.youtube.com/watch?v=l7HoguhFVQ4

Copy On Write (COW)

La **page sharing**, fiecare proces are acces la **aceleasi pagini** in memorie, cat timp aceasta parte **nu e modificata**. La modificare se copiaza intregul continut al paginii, iar **procesele nu vor mai avea acces** la aceeasi zona de memorie.

Algoritmi de page replacement: **FIFO, Least Recently Used, Second Chance, Opt.**

Thrashing se refera la o situatie in care sistemul de operare **foloseste mai mult timp si resurse** pentru a gestiona **swapping**-ul paginilor in si din memoria virtuala **decat pentru a efectua efectiv procesarea de date**. Acest lucru se intampla cand cantitatea de memorie fizica disponibila **nu este suficienta** pentru a sustine cerintele de memorie ale proceselor care ruleaza pe sistem.

Slab este una sau mai multe pagini fizice contigue.

Cache inseamna una sau mai multe **slab**-uri.