Sisteme de Operare

## Model de subiect - de la ultimul curs

* Procese zombie: definiție și exemplu de cod care să producă un proces zombie

In cazul in care un proces copil se termina iar procesul parinte apeleaza wait (indiferent de momentul terminarii procesului copil) starea procesului la terminare este transmisa in modul amintit in paragraful anterior catre procesul parinte. In cazul in care procesul copil isi incheie executia iar procesul parinte nu apeleaza wait, procesul copil devine un proces zombie.

* Câte procese și câte fire sunt create?

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

fork()

pthread\_create()

fork()

}

Dacă presupunem că la fork nu se copiază firele de execuție, avem 16 procese și 10 fire.

* Avem o anumită arborescență, care este secvența de cod care o produce?

1

/ | \

2 3 4

pid = fork()

If (pid != 0) {

pid = fork()

If (pid != 0) {

fork()

}

}

* Se dă următoarea listă de procese cu timpii necesari pentru execuție:

|  |  |
| --- | --- |
| Proces | CPU |
| P0 | 9 |
| P1 | 13 |
| P2 | 8 |
| P3 | 4 |
| P4 | 7 |

Aplicați algoritmul de scheduling Round Robin cu o cuantă de timp q = 3

P0 P1 P2 P3 P4 P0 P1 P2 P3 P4 P0 P1 P2

3 6 9 12 15 18 21 24 25 28 31 34

* Se dă următoarea secvență de accesare a paginilor: 1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5

Avem 3 page frame-uri.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 4 | 4 | 4 | 1 | 1 | 1 | 4 | 4 |
|  | 2 | 2 | 2 | 5 | 5 | 5 | 2 | 2 | 2 | 5 |
|  |  | 3 | 3 | 3 | 6 | 6 | 6 | 3 | 3 | 3 |

Se dă următoarea secvență: 1, 2, 3, 1, 4, 2, 5, 3, 6, 4, 4, 5, 4, 6

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 1 | 1 |  | 1 | 1 | 5 | 5 | 5 | 4 |  | 4 |  |  |
|  | 2 | 2 |  | 4 | 4 | 4 | 3 | 3 | 3 |  | 5 |  |  |
|  |  | 3 |  | 3 | 2 | 2 | 2 | 6 | 6 |  | 6 |  |  |

*A****lock****occurs when multiple processes try to access the same resource at the same time.*

*One process loses out and must wait for the other to finish.*

*A****deadlock****occurs when the waiting process is still holding on to another resource that the first needs before it can finish.*

*So, an example:*

*Resource A and resource B are used by process X and process Y*

* *X starts to use A.*
* *X and Y try to start using B*
* *Y 'wins' and gets B first*
* *now Y needs to use A*
* *A is locked by X, which is waiting for Y*
* Considerați problema filozofilor și soluția propusă mai jos:

do {

wait(chopstick[i])

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

/\* \*/

signal(chopstick[i])

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

} while(true);

Ce fenomen/problemă apare?

Modificați codul astfel încât filozofii pari iau bețișorul din stânga iar cei impari pe cel din dreapta. Arătați că nu mai apare fenomenul.

Arătați dacă noua soluție satisface cele 3 proprietăți (excluziune mutuală, progres, timp finit de așteptare).

Dacă fiecare filozof execută wait(chopstick[i]) și apoi este întrerupt, poate apărea fenomenul de **deadlock** și tot procesul se blochează.

Modificarea propusă ar însemna să înlocuim liniile 2-3 cu:

If (i % 2 == 0) {

wait(chopstick[i]) // stanga

wait(chopstick[(i + 1) % n]) // dreapta

} else {

wait(chopstick[(i + 1) % n]) // dreapta

wait(chopstick[i]) // stanga

}

Chiar dacă toți filozofii execută prima linie din if (respectiv din else), nu o să apară deadlock, pentru că o să fie cel puțin un filozof care va putea să ridice ambele bețișoare.

Proprietăți satisfăcute:

* Excluziunea mutuală exista de la început, garantată de implementarea wait-ului (nu era posibil ca doi filozofi să ridice același chopstick simultan)
* Progresul este garantat acum, nu mai există deadlock, întotdeauna mănâncă cel puțin un filozof
* În cazul discutat mai devreme, în care toți filozofii încearcă să mănânce în același timp, este posibil ca singurul filozof care mănâncă să lase bețișoarele și apoi să le ridice din nou, luând posibilitatea altui filozof să le ridice.
* Fie A o matrice de întregi din N^{10x10}, reținută linie cu linie contiguu în memorie.

Avem 3 page frame-uri disponibile.

O pagină de memorie ține 10 întregi.

Prog 1: intră în totalitate într-o pagină P10

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

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

A[i][j] = 0;

}

}

Câte pagini are nevoie programul?

10 de la matrice + 1 pentru cod = 11 pagini

Cum arată diagrama Gant?

Codul accesează matricea în ordinea în care e în memorie: A[0][0], …, A[0][9]

Paginile matricei A: 0 1 2 3 4 5 6 7 8 9

Cu 3 frame-uri: tot timpul rețin într-un frame programul, și apoi încarc matricea în celelalte două:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **10** | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 | 10 |
|  | **0** | 0 | **2** | 2 | **4** | 4 | **6** | 6 | **8** | 8 |
|  |  | **1** | 1 | **3** | 3 | **5** | 5 | **7** | 7 | **9** |

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

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

A[i][j] = 0;

}

}

Paginile din matrice vor fi accesate secvențial:

1, 2, 3, …, 7, 8, 9, 1, 2, 3…

Avem de 10 ori mai multe page fault-uri.

## 

## Model de subiect 2019

Link: [Examen Irofti 2019.jpeg](https://drive.google.com/file/d/1tdxYxt6yowX1pQnnO_OlgafNv5IOoyij/view?usp=sharing)

Enunțuri și rezolvări:

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

Da, putem să reținem toate fișierele cu numele directorului lor în față (de exemplu, un director *dir* cu două fișiere *a* și *b* s-ar reține ca două fișiere */dir/a* și */dir/b*). Când ni se cere să listăm directorul */dir/*, putem afișa toate fișierele care încep cu prefixul */dir/*.

1. Fie următoarea arborescență de procese:

1

/ | \

2 3 4

| |

5 6

Scrieți o secvență de cod care reproduce această structură.

if (fork() != 0) {

// sunt in procesul 1

if (fork() != 0) {

// sunt tot in procesul 1

fork(); // l-am creat pe 4

} else {

// sunt in procesul 3

fork() // l-am creat pe 6

}

} else {

// sunt in procesul 2

fork() // l-am creat pe 5

}

1. Fie două procese Pi și 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; **// linia 13**

/\* remainder section \*/

} while (true);

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

1. Ce se întâmplă dacă eliminăm linia 13?
2. Arătați dacă soluția satisface cele trei proprietăți: exclusivitate mutuală, progres și timp finit de așteptare.
3. Fie două matrice A de dimensiune 6x6 și B de dimensiune 4x6, ținute contiguu 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.

Programul P1:

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

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

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

Programul 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?

Matricea A are nevoie de 6 pagini, una pentru fiecare linie:

A1: A[0][0] A[0][1] ...

A2: A[1][0] A[1][1] …

A3: ...

A4: ...

A5: …

Analog avem nevoie de 4 pagini pentru B.

Mai avem nevoie de încă două pagini, pe care le notăm P1 și P2, care rețin codul celor două programe.

1. Cum arată diagrama Gannt folosind algoritmul LRU cu o strategie de înlocuire a paginilor globală (în care ambele programe pot folosi toate frame-urile disponibile în sistem).

Inițial se încarcă P1 și A1 în memorie, apoi P2 și B1. Când P1 ajunge la următoarea linie din matrice, încarcă în memorie A2. Apoi vine B1, vrea să încarce B2, și este înlocuit A1 pentru că a fost utilizată cea mai demult.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| P1 | P1 | P1 | P1 | P1 | P1 | P1 | P1 | ... |
|  | A1 | A1 | A1 | A1 | B2 | B2 | B2 |
|  |  | P2 | P2 | P2 | P2 | P2 | P2 |
|  |  |  | B1 | B1 | B1 | A3 | A3 |
|  |  |  |  | A2 | A2 | A2 | B3 |

Procesul se repetă până când P2 termină cu B4.

Rămâne doar P1 care încarcă ultimele 2 pagini ale matricei A în memorie ca să le scrie.

1. 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?

Am putea aloca fiecărui program câte două frame-uri, într-un frame fiecare program își va ține codul și în celălalt va încărca, pe rând, liniile matricei. Frame-uri în plus nu ar ajuta programele, tot același număr de page fault-uri ar fi.

## **Examen seria 24**

1. a) Scrieți o secvență de cod care să ofere concurență și paralelism.

b) Prezentați un scenariu în care secvența s-ar putea executa complet paralel. Fără concurență.

1. Fie următoarea arborescență de procese:

1

/ \

2 3

/ \ / \

4 5 6 7

1. Fie două procese Pi și Pj care vor să acceseze zona critică, unde procesele împart lock (inițializat cu 0).
2. Arătați că soluția satisface cele 3 proprietăți: exclusivitate mutuală, progres și timp finit de așteptare.
3. Dați n exemplu arătând cum asigură linia 2 acces la zona critică.

do {

while (compare\_and\_swap(&lock, 0, 1))

; /\* do nothing \*/

/\* critical section \*/

lock = 0;

/\* remainder section \*/

} while (true);

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

|  |  |  |
| --- | --- | --- |
| Proces | t | CPU |
| P0 | 0 | 7 |
| P1 | 1 | 6 |
| P2 | 4 | 3 |
| P3 | 5 | 2 |
| P4 | 7 | 1 |

1. Cum arată diagrama Gantt rezultată în urma aplicării algoritmului Round Robin cu q = 2?
2. Dar pentru q = 4?
3. Care este timpul mediu de așteptare la punctul a)

**P2 rez**

#include <stdio.h>

#include <unistd.h>

int main()

{

\_\_pid\_t p2 = fork();

if(p2 > 0)//e parintele (1)

{

\_\_pid\_t p3 = fork();

if (p3 == 0){ //e copilul nr 3 al lui 1

\_\_pid\_t p6 = fork(); //3 face copil

if (p6 > 0){ //se executa 3

\_\_pid\_t p7 = fork();// 3 mai face un copil

}

}

}

else {//sunt copilul 2 al lui 1

\_\_pid\_t p4 = fork();//il creaza pe 4

if(p4 > 0){//sunt in 2, si il creez pe 5

\_\_pid\_t p5 = fork();

}

}

}