Interactiunea intre procese

Zone critice

- Partea programului in care este apel spre date utilizate comun, se numeste zona critica ori sectie critica. Daca este posibil de a evita aflarea a doua procese concomitent in zona critica, se va evita concurenta.
- Pentru evitarea concurentei e necesar de a efectua 4 cerinte:
- 1. Doua procese nu trebuie concomitant sa se afle in zone critice.
- 2. In program nu trebuie sa fie presupuneri a viterei ori cantitatii de procese.
- 3. Procesul, care se afla in afara zonei critice, nu poate bloca alte procese.
- 4. Este imposibila situatia in care procesul se afla vesnic in asteptarea intrarii in zona critica.

Excluderea reciproca cu asteptare activa

Cea mai simpla solutie este interzicerea tuturor intreruperilor la intrarea procesului in zona critica si permiterea intreruperilor la iesirea din aceasta zona. Daca intreruperea este interzisa, este imposibila intreruperea dupa timer. Intrucit procesorul trece de la un proces la altul numai dupa intrerupere, interzicerea intreruperilor exclude posibilitatea de a trece de la un proces la altul. Astfel proces poate sa prelucreze si salveze datele fara a concura cu alte procese.

Blocarea cu ajutorul variabilei binare

- Variabila blocarii initial este 0. Daca procesul doreste sa intre in zona critica, el preventive citeste variabila. Daca variabila este 0, procesul o trece in 1 si intra in zona critica. Daca variabila este 1 atunci procesul asteapta, pina valoare ei nu va fi 0. Astfel 0 inseamna ca nici un proces nui in zona critica, iar 1 inseamna ca careva proces este in zona critica.

Alternare Stricta

Initial ambele procese se afla in afara zonelor critice. Procesul 0 apeleaza enter_region, genereaza elementele tabloului si trece variabila turn in 0. Deoarece procesul 1 nu este interest in trecere in zona critica, procedura se realizează. Acum daca procesul 1 va apela enter_region, el va astepta pina interested(0) va avea valoarea False, asta se va intimpla numai atunci cind procesul 0 va apela functia leave_region, pentru a iesi din zona critica.

Daca ambele procese au apelat enter_region practic concomitant, ambii salveaza numarul sau in turn. Se va salva numarul acelui proces, care era al doilea, iar cel precedent va fi pierdut. Acest proces va ramine in ciclu si va astepta pina celalalt proces va iesi din regiunea critica.

```
while(TRUE) {
while(turn!=0) /*loop*/;
critical region();
Turn=1;
noncritical region();
}
while(TRUE) {
while(turn!=0) /*loop*/;
critical region();
turn=0;
noncritical region ();
}
```

Variabila turn initial egala cu 0, depisteaza al cui este rindul de a intra in zona critica. Initial procesul 0 controleaza valoarea turn, citeste 0 si intra in zona critica. Procesul 1 controleaza valoarea turn, citeste 1 si intra in ciclu, permanent controlind cind valoarea lui turn va fi 0. Acest control permant se numeste asteptare activa (buclă de așteptare).

Dezavantajul acesta este cheltuiala timpului procesorului. Blocarea, folosind asteptarea activa, se numeste spin-block.

Aceasta situatie incalca a treia cerinta formulate mai sus, un proces ce nu se află în zona critică este blocat de altul.

Algoritmul lui Peterson

Initial ambele procese se afla in afara zonelor critice. Procesul 0 apeleaza enter_region, genereaza elementele tabloului si trece variabila turn in 0. Deoarece procesul 1 nu este interest in trecere in zona critica, procedura returneaza. Acum daca procesul 1 va apela enter_region, el va astepta pina interested(0) va avea valoarea False, asta se va intimpla numai atunci cind procesul 0 va apela functia leave_region, pentru a iesi din zona critica.

Daca ambele procese au apelat enter_region practic concomitant, ambii salveaza numarul sau in turn. Se va salva numarul acelui proces, care era al doilea, iar cel precedent va fi pierdut. Acest proces va ramine in ciclu si va astepta pina celalalt proces va iesi din regiunea critica.

```
#define FALSE 0
#define TRUE 1
fdefine N 2 /* Numarul de procese */
int turn: /* Cine e la rînd */
int interested[N]; /* Inițial variabelile sunt 0 (FALSE) */
void enter_region(int process); /* Procesul 0 sau 1 */
{
int other; /* Numarul procesului doi */
other = 1 - process; /* Primul proces */
interested[process] = TRUE; /* Indicatorul de interese*/
turn = process; /*Instalarea flagului*/
while (turn == process && interested[other] == TRUE) /*Operator pustiu */;
}
void leave_region(int process) /* process: proesul părăsește zona critică */
{
interested[process] = FALSE; /*Indicatorul ieșirii din zona critică */
}
```

Comanda TSL (Test and Set Lock- control si blocare)

In registru RX se salveaza continutul cuvintului memoriei lock, iar in celula memoriei lock se salveaza o oarecare valoare nenula. Se garanteaza in timpul operatiei de citire si salvare a cuvintului niciun proces nu poate apela cuvintul din memorie, pina comanda nu va fi efectuata. Procesorul care efectuiaza comanda TSL, blocheaza magistrala de memorie.

Enter_region: TSL REGISTER.LOCK; valoarea lock se copie in registru, valoarea variabilei devine 1

GMP REGISTER#0; valoarea veche lock se compara cu 0

JNE enter_region; Daca nui zero, atunci blocarea deja era initiate, ciclul nu se termina

RET; returnarea la programul ce apeleaza, procesul a intrat in zona critica

leave_region:MOVE LOCK#0; salvarea 0 in lock

RET

Inainte de a intra in zona critica procesul apeleaza procedura enter_region, care efectueaza asteptarea activa pina la eliminarea blocarii, apoi ea instaleaza blocarea si returneaza. La iesirea din zona critica procesul apeleaza procedura leave_region, plasind 0 in variabila lock.

Pentru ca totul sa fie correct procesul trebuie sa apeleze functiile la timp.

Primitivele interactiunii intre procese

Primitivele interactiunii intre procese utilizate in loc de cicluri de asteptare, in care in zadar se cheltuie timpul. Aceste primitive blocheaza procesele in cazut interzicerii intrarii in zona critica. Cele mai simple sunt sleep si wakeup. Sleep este o cerere a sistemului in rezultatul carei procesul care apeleaza este blocat, pina nu-l va porni alt proces.La cererea wakeup este un parametru, procesul, care trebuie pornit. E posibila existenta inca unui parametru celula memoriei utilizata pentru cereri.

Problema Producator Consumator

```
#define N 100 /* Максимальное количество элементов в буфере */
int. count - 0: /* Текущее количество элементов в буфере */
void producer(void)
{
Int item;
while (TRUE) { /* Повторять вечно */
item = produce_item(); /* Сформировать следующий элемент */
if (count == N) sleepO; /* Если буфер полон, уйти в состояние ожидания */
insert_item(item); /* Поместить элемент в буфер */
count = count +1; /* Увеличить количество элементов в буфере */
if (count == 1) wakeup(consumer); /* Был ли буфер пуст? */
 } }
void consumer(void)
{
int item;
while (TRUE) { /* Повторять вечно */
if (count == 0) sleepO; /* Если буфер пуст, уйти в состояние ожидания */
item = remove_item( ); /* Забрать элемент из буфера */
count = count – 1; /* Уменьшить счетчик элементов в буфере */
if (count == U - 1) wakeup(producer); /* Был ли буфер полон? */
consume_item(item); /* Отправить элемент на печать */ }}
```

Daca consumatorul nu era in starea de asteptare atunci semnalul de activare a fost in zadar. Cind conducerea trece la consummator el se va intoarce la valoare cindva citita din count, va afla ca este egala cu 0 si va trece in starea de asteptare. Consumatorul va umple bufferul si va trece tot in asteptare.

Esenta acestei probleme este ca daca semnalul de activare trece la procesul ce nui in asteptare acesta dispare in zadar. Rezolvarea acestui dezavantaj poate fi prin adaugarea bitului de asteptare a activarii.

Semafoare

Semafoarele sunt variabilile ce pot fi 0 in cazul cind nus semnale de activare salvate ori un numar pozitiv ce corespune numarului de semnale existente.

Djikstra a propus 2 operatii, down si up. Down compara semaforul cu 0. Daca e mai mult de 0 atunci operatia down il micsoreza si returneaza conducerea procesului. Daca e 0 atunci down nu returneaza conducerea procesului, il trece in starea de asteptare.

Problema producator consummator cu semafoare

- fdefine N 100 /* количество сегментов в буфере */
- typedef int semaphore; /* семафоры - особый вид целочисленных переменных */
- semaphore mutex =1; /* контроль доступа в критическую область */
- semaphore empty » N; /* число пустых сегментов буфера */
- semaphore full - 0; /* число полных сегментов буфера */
- void producer(void) {
- int item;
- while (TRUE) { /* TRUE - константа, равная 1*/
- item = produce_item(); /* создать данные, помещаемые в буфер */
- down(&empty); /* уменьшить счетчик пустых сегментов буфера */
- down(tautex); /* вход в критическую область */
- insert_item(item); /* поместить в буфер новый элемент */
- up(&mutex); /* выход из критической области */
- upt&full); /* увеличить счетчик полных сегментов буфера */
- void consumer(void) {
- int item;
- while (TRUE) { /* бесконечный цикл */
- downC&full); /* уменьшить числа полных сегментов буфера */
- down(&mutex); /* вход в критическую область */
- item = remove_item(); /* удалить элемент из буфера */
- up(&mutex); /* выход из критической области */
- up(Sempty); /* увеличить счетчик пустых сегментов буфера */
- consume_item(item); /* обработка элемента */
- }}

Se utilizeaza 3 semofoare, unul pentru calculul zonelor ocupate in buffer(full), unul pentru nr de segmente goale(empty), al treilea pentru excluderea accesului concomitant la buffer(mutex). Valoarea counterului initial e 0 , empty este egal cu numarul de segmente libere, iar mutex este 1. Semafoarele initial egale cu 0, utilizate pentru excluderea aflarii in zona critica a doua procese concomitant, se numesc semafoare binare.

Semaforul mutex se utilizeaza pentru realizarea blocarii concomitente, adica a apelarii concomitente la buffer si legat de variabilele a celor doua procese.

Semafoarele full si empty sunt necesare pentru a garanta ca anumite secvente se realizeaza ori nu.

Mutex (mutual exclusion – excluderea mutuala)

Mutex este variabila care se afla in una din doua stari:

Blocata ori neblocata . Pentru descrierea mutexului e necesar un bit, care este 0 daca nui blocat, celelalte valori inseamna stare blocata. Valoarea mutexului este definite de doua functii, daca procesul vrea sa intre in zona critica, el apeleaza procedura mutexjock. Daca mutexul nu este blocat cererea se efectueaza si threadul care apeleaza poate intra in zona critica.

Daca mutexul este blocat, threadul care apeleaza este blocat pina celalalt aflat in zona critica nu va iesi din ea apelind procedura mutex_unlock. Daca mutex blocheaza citeva threaduri atunci aleatoriu se alege unul.

```
mutexjock:
TSL REGISTER.MUTEX ; Valoarea veche din mutex se copie in registru si devine 1
CMP REGISTER.#0; compararea valorii vechi cu 0
JZE ok ; daca era 0 atunci mutex nu era blocat, return
CALL thread_yield ; mutex este ocupat, trece la alt thread
JMP mutex lock ; incercare mai tirzie
ok: RET ; return, intrare in zona critica
mutex_unlock:
MOVE MUTEX.#0 ; mutex devine 0
RET ; return
```

Monitoarele

Un monitor este asociat cu o valoare specifica si cu o functie de sistem care controleaza accesul la aceasta valoare. Atunci când un fir de executie retine monitorul pentru a accesa valoarea specificată celelalte fire de executie sunt blocate si nu au acces la această valoare. Un fir de executie poate prelua un monitor numai atunci când monitorul este liber, si îl eliberează atunci când dores te, când nu mai are necesitatea de acest fir de executie. Monitoare pot fi metode sau un cod al programului. Monitoarele se crează folosind cuvântul cheie synchronized .

Problema consummator producator prin monitoare

- public class ProducerConsumer {
- static final int N – 100; //marimea bufferului
- static producer p = new producer();
- static consumer c = new consumer();
- static our_monitor mon = new our_monitor();
- public static void main(String args[]) {
- p.start();
- c.start();
- }
- static class producer extends Thread {
- public void run() {
- int item;
- while (true) {
- item = produce_item();
- mon.insert(item);
- }

```
• private int producejtem() { ... }
• static class Consumer extends Thread {
• public void run() {
• int item;
• while (true) {
• item = mon.remove();
• Consume Jtem (item);
• private void consume_item(int item) { ... }
• static class our_monitor{
• private int buffer[] = new int[N]:
• private int count = 0, lo = 0, hi =  0;
• public synchronized void insert(int val) {
• if (count == N) go_to_sleep();
• buffer [hi] = val;
• hi = (hi+l)%N;
• count = count+1;
• if (count == 1) notify():.
• }
```

```
• public synchronized int remove( ) {
• int val:
• if (count == 0) go_to_sleep():
• val = buffer [lo]:
• lo = (lo+l)%N;
• count = count -1;
• if (count == N -1) notify();
• return val;
• }
• private void go_to_sleep() { try{wait():} catch(interruptedException exc) {};}
•  }}
```

Transmiterea mesajelor

Transmiterea mesajelor este metoda interactiunii intre procese folosind doua primitive: send si receive.

Sent(destination, Smessage);

Receive(source, Smessage);

Prima cerere transmite mesajul la destinatarul, iar a doua primeste mesajul de la sursa. Daca mesaj nui , a doua cerere este blocata pina la aparitia mesajului ori este returnat codul erorii.

Problema consumator producator prin transmiterea mesajelor

```
#define N 100
void producer(void)
{ int item;
message m; while (TRUE) {
item = produce_item();
receive(consumer, &m);
build_message(&m, item); send(consumer, &m);
void consumer(void)
{
int item, i;
message m;
 for (i = 0; i < N; i++) send(producer, &m);
while (TRUE) {
receive(producer, &m);
item = extract_item(&m);
send(producer, &m);
consume_item(item);}}
```

Bariere

La realizarea metodei de sincronizare bariere problema se împarte în etape (faze) de realizare si este important ca toate firele de executie sa finalizeze etapa curenta si numai apoi sa treaca la etapa urmatoare. Pentru a realiza această metodă de sincronizare trebuie sa cunoastem numarul firelor de executie care participa în etapa curenta.

Problema Cina filosofilor

Cinci filozofi chinezi îşi petrec viaţa gândind şi mâncând în jurul unei mese rotunde înconjurată de cinci scaune, fiecare filozof ocupând un scaun . În centrul mesei este un platou cu orez şi în dreptul fiecărui filozof se află o farfurie. În stânga şi în dreapta farfuriei câte un beţişor. Deci, în total, cinci farfurii şi cinci beţişoare. Un filozof poate efectua două operaţii: gândeşte sau mănâncă. Pentru a putea mânca, un filozof are nevoie de două beţişoare, unul din dreapta şi unul din stânga. Dar acesta poate ridica un singur beţişor odată. Problema cere o soluţie pentru această cină.

Trebuie rezolvate două probleme importante cum ar fi:

- interblocarea care poate să apară. De exemplu, dacă fiecare filozof ridică beţişorul din dreapta sa, nimeni nu mai poate să-l ridice şi pe cel din stânga şi apare o situaţie clară de aşteptare circulară, deci de interblocare.
- problema înfometării unui filozof care nu poate ridica niciodată cele două beţişoare

- #define N 5 /* nr de filosofi*/
- #define LEFT (i+N)%N /* numarul vecinului sting filosofului cu numarul i */
- fdefine RIGHT (i+1)%N /* numarul vecinului drept filosofului cu numarul i */
- #define THINKING 0 /* filosoful mediteaza*/
- #define HUNGRY 1 /*fislosof flamind*/
- #define EATING 2 /* filosoful maninca*/
- typedef int semaphore;
- int state[N]; /* starea fiecarui filosof*/
- semaphore mutex = 1; /* excluderea reciproca*/
- semaphore s[N]; /* fiecarui filosof un semafor*/
- void philosopher(int i) {
- while (TRUE) {
- Think(); /* filosof mediteaza*/
- take_forks(i);//primește 2 furculițe sau e blocat
- eat(); //mănîncă
- put_forks(i);//pune furculitele pe masă}}

• void take_forks(int i)// i numarul filosofului flămînd activ
• {
• down(&mutex); //intrare în zona critică
• state[i] = HUNGRY; //se apreciază filosoful flămînd
• test(i);// încearcă să primească 2 furculițe
• up(&mutex); //ieșire din zona critică
• down(&s[i]); //blocat dacă nu a primit 2 furculițe
• void put_forks(i); { //nimărul filosofului activ care mănîmcă
• down(&nutex); //intrarea în zona critică
• state[i] = THINKING; //filosoful a terminat de mîncat
• test(LEFT); //pote mîmca vecinul din stînga
• test(RIGHT); //poate mînca vecinul din dreapta
• up(&mutex); //ieșirea din zona critică
• void test(i) //numarul filosofului activ care meditează
• {
• if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)
• stated] = EATING; // filosofii meditează
• up(&s[i]);
}}

Problema scriitorilor si cititorilor

- Problema modeleaza accesul la baza de date. Simuleaza utilizatorii care cer conexiunea la baza de date . Este permisa citirea concomitenta, dar nu scrierea, care trebuie sa fie unica. In momentul scrierii toate procesele trebuie terminate, chiar si cele care citesc.

Problema cititorilor si scriitorilor

- typedef int semaphore;
- semaphore mutex =1; //controlul de acces la citipori
- semaphore db = 1; //controlil de acces la baza de date
- int rc = 0; //numărul de cititori
- void reader(void)
- while (TRUE) {
- down(&mutex); //Accesul monopol la rc (cititori)
- rc = rc+1; //cu un cititor mai mult
- if (rc == 1) down(Sdb); //dacă cititorul este primul
- up(&mutex); //refuză la accesul monopol la rc
- read_data_base(); accesul la baza de date
- down(&mutex); primește accesul monopol la rc
- rc = rc-1; //cu in cititor mai puțin
- if (re == 0) up(&db); //dacă cititorul este ultimul
- up(&mutex); refuzî la accesul momopol la rc
- use data read(); //în afara zonei critice
- void writer(void)
- while (TRUE) {
- think_up_data(); // în afara zonei critice
- down(Sdb); //primește acces monopol
- write_data_base(); //înscrie datele în baza de date

Problema bărbierului

In frizerie este doar un barbier, scaunul lui si n scaune pentru client. Daca doritori sa utilizeze serviciile lui nus, atunci el doarme in scaunul sau. Daca intra un client el trebuie sal trezeasca. Daca clientul intra si vede ca barbierul e ocupat el ori asteapta pe scaun, in caz ca este loc, ori pleaca, daca nui loc.

Problema bărbierului

- #define CHAIRS 5 //numărul de scaune
- typedef int semaphore;
- semaphore customers = 0;// numărul de clienți care așteaptă
- semaphore barbers = 0; //numarul de barbieri care așteaptă clienți
- semaphore mutex = 1;//pentru excluderea interblocării
- int waiting = 0; //clienții care așteaptă
- void barber(void)
- {
- while (TRUE) {
- down(&customers);)//dacă clienți nu sunt, trece în starea de așteptare
- down(Smutex); //cere acces la clienții care așteaptă
- waiting = waiting – 1; //se micșorează numărul de clienți
- up(Sbarbers); //un barbier este gata de lucru
- up(Smutex); //interzice accesul la clienți

- cut hair(); //clientul este servit în afara zonei critice
- void customer(void)
- down(&mutex); //intrarea în zona critică
- if (waiting < CHAIRS) //dacă scaune libere nu sunt, pleacă
- waiting = waiting +1; //se adaugă un client în rînd
- up(Scustomers); //dacă barbierul doarme este trezit
- up(&mutex); //accesul interzis la clienți
- down(Sbarbers); //dacă barbierul e ocupat trece în starea de așteptare
- get_haircut(); //clientul este deservit
- } else {
- up(&mutex);//rîndul e plin și pleacă