Tema de casa

2013

Sisteme de operare

Comunicare Interprocese

Studenti: Chelaru Camelia

Giana Catalin

Grupa 433A

CUPRINS:

A. Introducere generala – Prezentare tema

Actuala tema se cheama Comunicarea intreprocese, definind pe scurt un concept de baza specific oricarului sistem de operare cum ar fi procesele. Fiind un element fundamental sistemului de operare, procesele alcatuiesc un domeniu vast de informatie, iar in lucrare ce urmeaza vor fi prezenate doar concepte de baza despre cum comunica procesele si ce solutii de pot implementa.

Partea I prezinta o scurta introducere in ceea ce insemna proces, fir de executie, necesitatea comunicarii intre procese, regiunea critica si metode de combatere a interblocarilor si buna functionare a sistemului de calcul. In partea a II-a se exemplifica 3 probleme generale care se intalnesc in executia proceselor care partajeaza aceleasi reusurse ale sistemului si idei de solutionare.

B. SUBIECTE

1.1 Introducere – Procese. Regiuni critice

Proces

Mai este intalnit in domeniu si sub nume de task, care defineste cel mai simplu proces elementar, care este independent, nu mai poate fi descompus in alte procese.

Procesul e definit in literetura de specialitate ca *“ o abstractizare a unui program in executie” [1- pag 67]* cu anumiti parametrii specifici: un spatiu de adrese din care programul aferent poate sa citeasca sau sa scrie.

Procesele trebuie sa comunice intre ele pentru a evita interblocarile (asemanator in practica cu un blocaj in trafic), situatie care apare atunci cand procesele ce ruleaza nu avanseaza unul din cauza altuia. O solutie pentru a lega doua procese ar fi o conducta (pipe). *“O astfel de componenta este un pseudofisier”* in care de scriu/ citesc date de catre procesele in cauza.

Procesele nu sunt de sine statatoare, ele pot sa interactioneze intre ele, in sensul ca datele de iesire dintr-un proces pot fi date de iesire pentru altul, acest lucru faca comunicarea indispensabila, pentru sincronizarea activitatilor. Aceasta comunicare face posibila acest mecanism, fara a partaja acelasi spatiu de adrese.

In engleza **Comunicarea interprocese** se denumeste ca IPC (InterProcess Communication)

Procesele dintr-un SO pot fi: *(taxonomie preluata din [2 pag 107])*

a) procese independente

- Daca execuţia unui proces nu poate afecta sau nu este afectata de catre execuţia altor procese din sistem, atunci procesul este independent.
- Evident, un proces care nu partajeaza resurse (date) cu alt proces este independent.

b) procese cooperante – in caz contrar, fiind procesul care influenţeaza sau este influenţat de alte procese care ruleaza pe acelaşi sistem sau pe alte sisteme.

Regiunea Critica

Cateodata un proces trebuie sa acceseze memoria partajata sau fisiere. Portiunea programului unde memoria partajata este accesata pentru citire sau scriere se numeste **regiunea critica sau sectiune critica.**

Practic, sectiunea critica poate fi privita ca "*actualizarea unei variabile partajate" [3]*

De exemplu, atunci cand doua procese vor sa acceseze memoria partajata in acleasi timp. Conditia optima ar fi ca cele doua procese sa nu fie in acelasi timp in regiunilor lor critice, o solutie implementata ar fi **excluderea mutuala** (un proces oarecare foloseste doar el aceea resursa partajata). Probleme apare atunci cand unele procese nu termina operatiile cu unele variabile globale, pe care le acceseaza si alte procese. Cand un proces intra intr-o secţiune critica, el trebuie sa execute complet toate instrucţiunile secţiunii critice, inainte ca alt proces sa o poata accesa.

Numai procesului care executa o secţiune critica ii este permis accesul la variabila partajata, in timp ce tuturor celorlalte procese le este interzis accesul

De aici rezulta ca e necesar transmiterea de mesaje suplimentare, semnalizarea terminarii unor procese, si astfel orice solutie implementata trebuie sa respecte urmatoarele 4 **conditii de cooperare corecta** ca sa nu apare erori de executie**.** [1 – pag 96]:

1. Oricare doua procese nu se pot afla simultan in regiunile lor critice.

2. Nici un fel de presupunere nu se poate face asupra vitezelor sau numarului de procesoare.

3. Nici un proces care ruleaza in afara regiunii sale critice nu poate bloca alt proces sa intre in regiunea sa critica.

4. Nici un proces nu trebuie sa astepte la infinit pentru a intra in regiunea sa critica.

1.2 Excluderea mutuala folosind asteptarea ocupata

Daca doua procese M si N vor sa comunice, trebuie sa faca shcimb de mesaje, iar pentru aceasta trebuie sa existe un "*link de comunicatie"* [2] intre cele doua procese.

Excludere mutuala are rolul de a **"exclude temporar accesul altor procese la o variabila partajata."** [3 – pag 39].

Din acest punct de vedere, mecanismul de baza pe care il urmareste un proces este:

-protocol de negociere , in urma caruia invingatorul continua execuţia;

-regiunea critica care permite utilizarea exclusiva a resursei;

-protocol de cedare , dupa care detinatorul se deconecteaza.

Solutiile propuse pot fi pe baza hard (validare/invalidare de intreruperi sau instrucţiuni *Test and Set, Dezactivarea Intreruperilor*) si soft (Peterson si Lampert).

Dezactivarea intreruperilor

Ipoteza: Porneste de la premisa ca orice procesor dispunde de instruciuni de dezactivare a intreruperilor.

Functionare: Algorimul este cel mai simplu, un proces dezactiveaza intreruperile imediat cum a accesat zona critica si le reactiveaza iar dupa ce a terminat excutia cu succes. Asa, procesul detine controlul sistemului de calcul.

Dezavantaje: nu mai este posibila comutare intre procese, pentru ca nu mai avem intreruperi de ceas, si apar probleme grave daca nu ar fi posibila reactivarea lor.

Variabile zavor

Ipoteza: Se utilizeaza o singura variabila partajata intre procese, care are valoarea 0 atunci cand niciun proces nu foloseste resurselor comune si valoarea 1 atunci cand cel putin un proces foloseste memoria partajata.

Functionare: La intrarea in regiunea critica, se testeaza aceasta variabila, daca ea este 0 atunci se poate accesa zona, si dupa acces variabila devine 1, altfel se asteapta pana devine 0. La iesirea variabila se reseteaza la valoarea implicita 0.

Dezavantaj: Metoda e ineficienta atunci cand doua procese intra simultan in regiunea critica, nu se rezova excluderea reciproca

Instrucțiunea Test and Set (TS)

Ipoteza: Este o solutie hard pentru ca are la baza instructiunea TSL RX, LOCK, prezenta in orice calculator modern. Ea are ca operand adresa (registrul RX) unei variabile de control(zavorul LOCK), pentru intrarea si iesirea din regiunea critica.

Functionare: ce face instructiunea TSL? -compara valoarea operandului cu o valoare constanta (de exemplu 0 pentru BUSY) si seteaza flagurile de conditie ale procesorului, dupa care seteaza operandul la valoarea BUSY. Atunci cand apare instructiunea TSL, magistrala memoriei se blocheaza pana la terminarea instructiunii.

COD: LOAD R, operand //copierea operandului in registru

CMP R, BUSY // compara zavorul cu Busy

flag ZERO=1 // se face doar daca R=BUSY

STORE operand,BUSY // daca R nu e BUSY, face operandul BUSY

Dezavantaj: folosirea asteptatii ocupante, asteptarea in bucla pana cand accesul ii este permis induce timp pierdut pe procesor

Alternarea stricta (Busy-Wait)

Ipoteza: este un protocol de asteptare in excluderea mutuala

Functionare: Avem o variabila "turn" initializata cu valoarea 0. Ea determina cine are voie sa intre in regiunea critica si are rol in actualizarea memoriei. Daca turn este 0 se intra in regiunea critica, daca nu, atunci celelalte procese intra intr-o bucla locala, astfel variabila "turn"este ascultata non stop de catre sistem.

Dezavantaj: consuma timp de execuție al procesorului de catre un proces care nu avanseaza, atunci cand procesele in cauza au timp de executie diferiti, procesele trebuie sa se astepte unele pe altele, pentru ca executa alternant bucle in zona critice si noncritice cand unul cand altul. (se impune o alternare stricta)

1.3 Excluderea mutuala fara asteptare activa

Exemplele precedente, cer activitate inutila pentru asteptarea eliberarii resurselor si astepe timp pierdut. Solutia : blocarea procesului pentru care resursa nu edisponibila. Acesta nu va efectua activitate inutila cit timp va astepta. Daca procesul nu e blocat, activitatile de testare inoportuna pot crea blocaj, daca activitatile cu prioritate mare nu pot fi desfasurate, pentru ca resursele exista deja la activitati prioritate mica (L)(mecanismul de prioritati impiedica circulatia taskurilor). Am putea astfel bloca procesului cu prioritate mare(H).

1.3.1 Sleep si wake-up

Ipoteza: In acest tip de protocol, procesul care nu are resursa disponibila este suspendat (adormit) ai introdus intr-o coada de aşteptare. Apoi este trezit, cand resursa devine disponibila.

Functionare: Apar doua primitive: SLEEP care autoblocheaza procesul curent si WakeUp care trezeste/activeaza procesul. Ele sunt rutine ale sistemului. Pentru a exemplifica procedeul folosim asemanarea cu operatii de scriere/citire. Scriere= inserarea unui element, iar citire= extragerea unui element. Fie o zona de memorie, denumita zona tampon finita, care este partajata intre doua procese, procesul M scrie date si alt proces N preia/citeste date tot din aceasta zona. Ce se intampla atunci cand zona este plina si nu se mai poate scrie, sau cand nu ai date de citit, zona este goala? Ambele procese in situatiile prezentate vor fi blocate si se suspenda si vor fi trezite atunci cand procesul complementar isi va face treaba, sa citeasca date pentru golirea zonei si respectiv sa scrie date pentru umplerea zonei.

Pentru a testa daca zona este sau nu goala, se testeaza o variabila cout care retine numarul de elemente maxime din zona tampon(N). apare 3 situatii:
-Daca count=N atunci se suspenda procesul M
-Daca count=0 se suspenda procesul de citire N
-Daca count are o valoare oarecare intre 0 si N, atunci se va scrie pentru M si contorul va fi incrementat, si pentru N se va citi si contorul decrementat.

Avantaj: In sistemul uniprocesor este folosit totdeauna SLEEP-WAIT, deoarece nu se consuma timp procesor inutil, blocheaza primitvele.

Dezavantaj: un proces merge la culcare cand celalalt vrea sa-l trezeasca, deci se pierde trezirea, situatia care apare la un acces dublu la zona tampon, atunci cand scriem si citim in acelasi timp, urmand blocarea ambelor procese.

1.3.2 Semaforul

Ipoteza: *Semafoarele sunt obiectele de sincronizare cele mai des şi mai mult folosite* [3]. Este o generalizare a situatie de sleep si wake-up, avantajul adus fiind semnalul de trezire care nu se mai pierde de drum.

Functionare: Un semafor este de fapt o variabila partajata care ia valori diferite de 0. Asupra unui semafor se pot executa doua operaţii de baza:

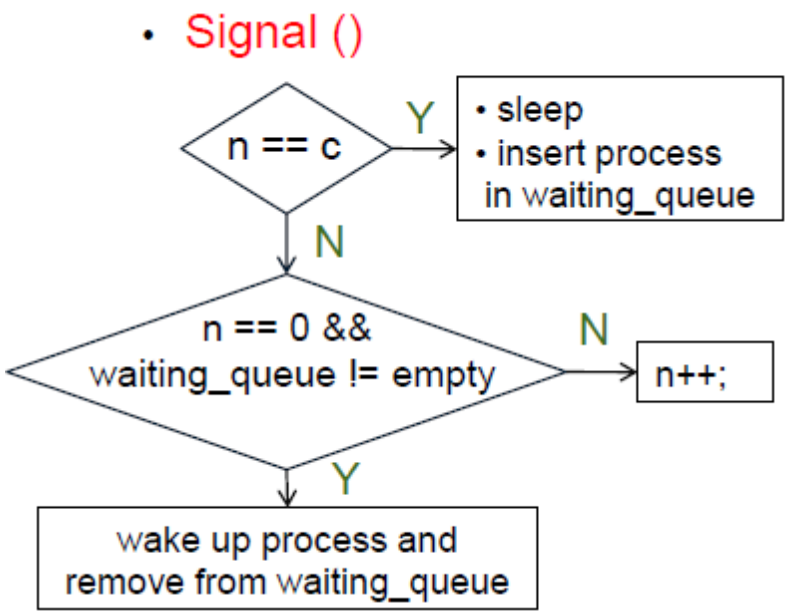
-decrementarea variabilei semafor cu 1 (DOWN), adica procsesul actual isi continua executia

-incrementarea variabilei semafor cu 1 (UP), adica vor fi mai putine procese care asteapta la acel semafor

Pentru exemplificarea celor doua situatii vom folosti doua primitive, asa cum se vede in schema exmplificata mai jos, [preluata din 4]

wait(s) - incearca sa decrementeze valoarea variabilei semafor s cu 1 şi daca variabila este 0 procesul ramane in aşteptare pana cand $s \neq 0$.

signal(s) – incrementeaza cu 1 semaforul s.



Procedura wait( ) este asemanatoare in conditiile in care prima conditiie este n==0, a doua conditie este n==c, si operatia executata este n--.

Unde c= capacitatea, numarul de resurse libere, iar n este numarul total de resure utile.

Dezavantaje consum de timp procesor de catre procesele aflate in aşteptare. Sau amanarea unui proces un timp oarecare pentru obţinerea unui semafor, datorita faptului ca nu este nici o organizare intre procesele care aşteapta la semafor.

1 .3.3 Mutex

Ipoteza: Mutex sau MUTual Exclusion locks sunt cele mai simple sisteme de sincronizare. Un mutex este folosit pentru a asigura acces exclusiv la datele impartite intre firele de executie.

Functionare :

-operaţia de acces şi obţinere a mutex-ului, notata *cu ocupare (mutex)* sau *lock(mutex)* ;

-operaţia de eliberare a mutex-ului, notata cu *eliberare (mutex)* sau *unlock(mutex)*.

Avantaje: Numai un singur thread (fir de executie) poate avea mutexul securizat la un anumit timp. Firele de executie care incearca sa blocheze un mutex deja blocat vor ramane blocate pana cand threadul care detine mutexul il va debloca. Cand thread-ul deblocheaza mutexul, procesul cu prioritatea cea mai mare care asteapta sa blocheze mutexul va devine noul proprietar al mutexului. In acest fel, firele de executie vor fi executate in ordinea prioritatilor lor.

Pe majoritatea procesoarelor, achizitia unui mutex nu necesita accesarea kernelului pentru a gasi un mutex liber. Ceea ce permite acest lucre este algoritmul de comparare si schimbare la un procesor X86 si conditia de incarca/pastreaza pe majoritatea procesoarelor RISC.

Accesarea kernelului este facuta la momentul achizitiei numai daca mutex-ul este deja tinut astfel incat firul sa acceseze o lista blocata; accesarea kernelului este facuta la iesire daca alte fire asteapta sa fie deblocate pe acel mutex. Acest lucru permite achizitia si eliberarea unei sectiuni critice sa fie foarte rapide si suporta munca depusa de sistemul de operare pentru a rezolva disputa.

O functie de verificare a blocarii (pthread_mutex_trylock) poate fi folosita pentru a testa daca mutexul este in present blocat sau nu. Pentru cele mai bune performante, timpul de executie al sectiunii critice trebuie sa fie mic si de o durata limitata. O conditie variabila ar trebui sa fie folosita daca firul se blocheaza in sectiunea critica.

Dezavantaj: Implicit, daca un thread cu o prioritate mai mare decat detinatorul mutexului incearca sa blocheze un mutex, atunci prioritatea efectiva a detinatorului curent este crescuta la cea a threadului cu prioritate mai mare care asteapta blocarea mutexului. Prioritatea efectiva a detinatorului curent este din nou ajustata cand se deblocheaza mutexul; noua sa prioritate este maximul prioritatii dintre a sa si cea a threadurilor care blocheaza mutexul direct sau indirect.

Codul pentru mutex_ lock si mutex_unlock este prezentat in sectiunea urmatoare :

Mutex_lock:

```
TSL  registru, mutex        // copiaza mutex in registru si seteaza mutex la 1
CMP registru, #0            // verifica daca mutex-ul a fost 0
JZE ok                      // daca mutex era 0, atunci era deschis, se face iesirea
CALL thread_yield           //mutex-ul e ocupat; planifica alt fir de executie
```

```
        JMP mutex_lock          // sare mai tarziu la eticheta mutex_lock
Ok:     RET                     //intoarcere la apelant; s-a intrat in regiunea critica

Mutex_unlock:
        MOV mutex, #0           // scrie 0 in mutex
        RET                     // intoarcere la apelant
```

1.3.4 Monitoare

Ipoteza: Monitoarele sunt niste primitive de sincronizare de sincronizare de nivel mai inalt propuse de Hoare(1974) si Brinch Hansen (1975) pentru a usura scrierea unor programe correct, existand diferente minore intre conceptele celor 2.

Functionare: Un monitor reprezinta o colectie de proceduri, variabile si functii care sunt grupate intr-un pachet, astfel incat procesele pot apela proceduri din monitor, dar nu si structurile de date interne acestuia.

Cea mai importanta caracteristica a unui monitor este aceea ca intr-un astfel de pachet poate fi activ un singur proces intr-un interval de timp dat. Acestea fiind niste structuri speciale compilatorul trateaza in mod diferit apelurile acestor proceduri fata de celalalte apeluri.

Asadar, in momentul in care se apeleaza o procedura din monitor, mai intai se verifica daca exista un alt proces activ. In cazul in care exista unul activ, va fi suspendat pana in momentul in care celalalt proces va parasi monitorul, altfel programul apelant poate intra in monitor.

Avantaje : Deoarece compilatorul si nu programatorul se ocupa de excluderea mutuala este mai putin probabil ca ceva rau sa se intample , desi programatorul trebuie sa cunoasca modul in care compilatorul obtine excluderea mutuala. Tot ce trebuie stiut este ca prin transformarea regiunilor critice in proceduri de monitor nu vor exista procese care sa isi execute regiunile critice in acelasi timp.

Dezavantaje: Chiar daca monitoarele ofera un mod usor de a obtine excluderea mutuala acest lucru nu este suficient, fiind necesara o alternativa ca procesele sa se blocheze doar atunci cand chiar nu mai

pot continua, lucru pentru care s-au introdus variabilele de conditie si 2 operatii wait si signal care verifica daca zona tampon este libera sau ocupata lasand sau nu procesul sa intre in monitor.

Un alt dezavantaj ar fi acela ca monitoarele nu pot fi aplicate in cazul unui sistem distribuit cu mai multe procesoare conectate printr-o retea locala, fiecare detinand propria memorie privata si nu poate fi folosit la schimbul de informatie intre masini.

1.3.5 Transferul de mesaje

Ipoteza: Aceasta metoda de comunicare foloseste 2 primitive, send si receive, acestea fiind spre deosebire de monitoare apeluri de sistem si nu constructii de limbaj, putand fi usor aranjate sub forma unor proceduri.

Functionare: Procedurile send si receive : " send (destination, & message)" si " receive (source, & message);" sunt folosite pentru a trimite un mesaj catre o anumita destinatie, respectiv pentru a primi un mesaj de la o anumita sursa. In cazul in care nu este disponibil niciun mesaj receptorul va ramane blocat pana la primirea unui mesaj.

- pentru a se proteja impotriva mesajelor pierdute, emitatorul si receptorul a fost introdus un semnal de confirmare speciala (acknowledgement). Daca nu se trimite ACK intr-un interval de timp, emitatorul va retransmite mesajul
- sistemele de mesaje trebuie sa trateze si autentificarea: sa se verifice daca mesajul este unul de send sau de receive.
- O alta problema este performanta : copierea mesajelor fiind mai lenta decat efectuarea unei operatii asupra unui semafor sau intrarea intr-un monitor.

Avantaje: in cazul transferurilor de mesaje se pot folosi casutele postale, acesta fiind un sistem de stocare temporara a mesajelor care este clar : casuta postala de la destinatie pastreaza mesaje care au fost trimise procesului destinatar, dar nu au fost inca acceptate.

Dezavantaje: Sistemele de transfer de mesaje au multe probleme de proiectare , care nu apar in cazul semafoarelor sau al monitoarelor.

1.3.6 Bariere

Ipoteza: Unele aplicatii sunt impartite pe bucati si nu pot trece mai departe pana cand toate procesele nu sunt gata pentru urmatoarea etapa. Acest lucru poate fi obtinut prin pozitionarea unei bariere la sfarsitul fiecare etape pentru a le delimita clar.

Functionare:

- Un process care ajunge la bariera este blocat pana ajung toate;
- Cand primul proces ajunge la bariera, acesta este suspendat pana in momentul in care ajung si celelalte;
- Cand toate ajung la bariera, toate sunt eliberate in urmatoarea etapa;

Avantaje: Acest sistem poate fi folosit in cazul in care exista foarte multe procese care trebuiesc executate cat mai repede. Cand toate procesele ajung la sfarsitul unei etape vor fi eliberate simultan in urmatoarea etapa.

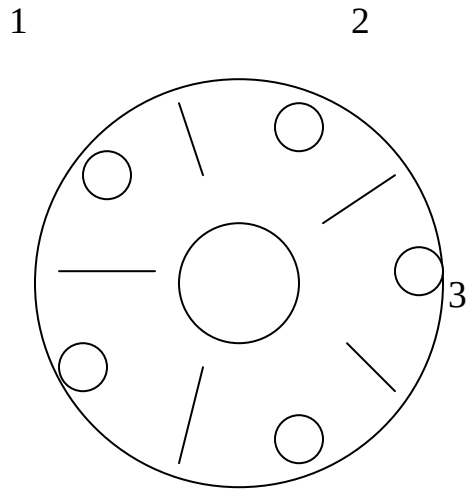
Dezavantaje: Niciun proces n+1 nu poate fi executat pana in momentul in care sunt finalizate toate procesele din iteratia n.

II. *Probleme clasice ale comunicarii interprocese*

2.1 Problema filozofilor

Enunt: Cinci filozofi chinezi stau la cina, fiecare cu cinci farfurii şi cinci furculite, dar pentru a manca un filozof are nevoie de doua furculite, una din dreapta şi una din stanga. Dar un filozof poate ridica o singura furculita odata. Problema cere o soluţie pentru aceasta cina.

Apare o abstractizare, adica, etapele unui proces sunt correlate: executia cu perioada de gandire a unui filozof, iar perioada de hranire cu accesul la resurse ale unui process.



Problema filozofilor chinezi.

Probleme intampinate:

-**Interblocarea**. De exemplu, daca fiecare filozof ridica beţişorul din dreapta sa, in acelasi timp toti, nimeni nu mai poate sa-l ridice şi pe cel din stanga, pentru ca nu mai mai ai ce si apare o asteptare circulara.

-Problema infometarii unui filozof care nu apuca sa ridice niciodata cele doua beţişoare. In engleaza **Starvation.**

Pasi: Se preia furculita din stanga. Se verifica daca furculita din dreapta este libera. Daca cea din drepta nu este libera, se lasa jos furculita luata din stanga ulterior, pentru a o folosi alt filozof. Se asteapta un timp, pe urma se reia ciclul

Fiind la o masa circulara, exista posibilitatea ca cei 5 filozofi sa inceapa sa manance in acelasi timp. Presupunad ca timpii de propagare sunt egali, toti vor executa acelasi actiuni in acleasi timp, adica vor ceda furculita din stanga in acelasi timp, si niciodata nu vor avea o furculita libera in dreapta. Se creeaza astfel o bucla infinita si niciun proces nu evolueaza. O solutie la problema de starvation poate fi una probabilistica, alegerea unor timpi aleatori pentru asteptarea reluarii buclelor. Solutia este deja aplicata in Ethernet atunci cand se reia transmiterea pachetelor. In aplicatiile unde ai nevoie de siguranta, timpi aleatori nu este o solutia viabila pentru starvation.

Propunere de algoritm pentru rezolzarea celor 2 probleme. Se urmareşte in ce stare poate fi un filozof, existand trei stari posibile: mananca, gandeşte şi este infometat. Unui filozof i se permite sa intre in starea „mananca” numai daca cel puţin unul din vecinii sai nu este in aceasta stare. Prin aceasta restricţie se previne interblocarea.

Parametrii implementarii:

stare[n] – un vector n-dimensional, in care pe poziţia i se gaseşte starea filozofului la un moment dat; aceasta poate fi:

0 pentru starea „gandeşte”

1 pentru starea „infometat”

2 pentru starea „mananca”

sem[n] – un vector n-dimensional, in care sem[i] este un semafor pentru filozoful i;

mutexfil – un mutex pentru excludere mutuala;

funcţia filozof(i) – este funcţia principala care coordoneaza toate celelalte funcţii şi care se refera la filozoful i;

funcţia ridica furculita(i) – este funcţia care asigura pentru filozoful i ridicarea ambelor furculite;

funcţia pune furculita i – este funcţia care asigura pentru fiecare filozof i punerea ambelor furculite pe masa;

funcţia test(i) – este funcţia care testeaza in ce stare este filozoful i.

Implementarea [3]:

```
#define n 5                     //am definit numarul de filozofi
#define stang(i+n-1)%n          //numarul vecinului din stanga filozofului i
#define drept(i+1)%n            //numarul vecinului din dreapta filozofului i
#define gandeste 0              //filozoful gandeste
#defineinfometat 1              //filozoful incearca sa ia furculite
#define mananca 2               //filozoful mananca
typedef int semafor;            //semafor de tip intreg
typedef int mutex;
int stare[n];                   //vector pentru a retine starea fiecaruia
mutex mutexfil=1                //excludere mutuala pt regiunile critice
semafor sem[n];                 //cate un semafor pt fiecare filozof
```

```
void filozof(int i)
while(i) {                    //repetarea la infinit
gandeste();                   //filozoful i gandește
ridicafurculita(i);           //filozoful i ridica cele doua furculite
mananca();                    //filozoful i mananca
punefurculita(i);             //filozoful i pune pe masa doua furculite

Void ridicafurculite(int i)
{wait(mutexfil);              //se intra in regiunea critica
stare[i]=infometat;           //filozoful i este in starea infometat
test(i);                      //incearca sa acapareze cele doua furculite
signal(mutexfil);             //se iese din regiunea critica
wait(sem[i]);}                //procesul se blocheaza daca nu se pot lua
                              cele doua furculite

void punefurculite(int i)
{wait(mutexfil);              //se intra in regiunea critica
stare [i]=gandeste;           //filozoful i a terminat de gandit
test(stang);                  //se testeaza daca vecinul din stanga
                              filozofului i mananca
test(drept);                  //se testeaza daca vecinul din //dreapta
```

```
                                filozofului i mananca
signal(mutexfil);               //se iese din regiunea critica
}
void test(int i);
{if stare [i]==
infometat&&stare[stang]!=
mananca&&stare[drept]!=
mananca)
{stare[i]=mananca;
signal(sem[i]);
}
}
```

2.2 Problema cititorilor si scriitorilor

Aceasta problema este un mod de rezolvare a problemei concurentei. Aceasta verifica situatia cand mai multe thread-uri impart aceeasi zona de memorie in acelasi interval de timp, unul citind si altul scriind.

Limitarea apare in momentul in care 2 procese vor sa acceseze aceeasi zona de memorie, unul pentru a scrie si unul pentru a citi lucru care este imposibil , existand maxim posibilitatea ca doi utilizatori sa citeasca in acelasi timp din aceeasi zona de memorie. Pentru a elimina aceasta situatie trebuie creata o incuietoare cititor-scriitor , structura de date ce rezolva una sau mai multe probleme .

Aceasta problema apare de in majoritatea cazurilor cand se face accesul la o baza de date. Exista multe situatii (ex. Rezervarea locurilor la cinema) cand mai multe procese doresc sa utilizeze baza de date in acelasi timp. Chiar daca se va permite ca mai multe procese sa citeasca date din acea baza de date, este inadmisibil permitem mai multor thread-uri sa scrie simultan.

Vom enumera 3 posibilitati ale acestei probleme :

1. Problema 1:

- o zona de memorie ;
- datele partajate se pot proteja cu ajutorul unui semafor mutex.
- Solutia nu este insa cea mai optima!

2. Problema 2:

- o zona comuna de memorie partajata de mutex;
- se adauga constrangerea ca nici un scriitor, o data adaugat in coada de asteptare, nu va astepta mai mult decat este necesar.
- Se mai numeste si preferinta scriitorilor.

3. Problema 3:

- Solutia propusa de cele doua probleme anterioare duce la infometare.
- La a treia problema cititor-scriitor se adauga constrangerea ca nici un thread nu este lasat sa ajunga la infometare

Vom exemplifica pe larg codul ultimei probleme (cu transmitere de mesaje)

```
void reader(int id){
 message rmsg;
 while(1){
            rmsg = id;
            send("ReadReq",rmsg);
            receive("Reader"+id,rmsg);
             DoReading();
             rmsg = id;
            send("Fini",rmsg);
 }
}
```

```
void writer(int id){
      message rmsg;
     while(1){
              rmsg = id;
              send("WriteReq",rmsg);
              receive("Writer"+id,rmsg);
               DoWriting();
              rmsg = id;
              send("Fini",rmsg);   }}
void controller(){
  while(1){
     if(count>0){  //no writer
               if (!empty("Fini")){
                        receive("Fini",msg);
                         count++;
 }
     else   if(!empty("WriteReq")) {
              receive("WriteReq",msg);
              id = msg.id;
              count -= MAXREADERS;      }
      else   if(!empty("ReadReq")) {
              receive("ReadReq",msg)
                count--;
              send("Reader"+msg.id, "OK");      }    }
     if(count==0) {                                        //only a writer
               send("Writer"+id, "OK");
                receive("Fini",msg);
               count = MAXREADERS;    }
     while(count<0) {                                  // 1 wrtr;
              n rdrs
               receive("Fini",msg);
```

```
            count++;
        }
    }
    }
```

2.3 Problema frizerului somnoros

Enunt: Frizeria are un frizer, un scaun de tuns si n scaune pentru clientii care asteapta. Daca nu sunt client, frizerul sta pe scaunul de tuns si adoarme.

Pasii implementarii:

- Cand un client soseste trebuie sa il trezeasca pe frizerul adormit
- Daca sosesc alti client, fie stau pe scaun, fie parasesc frizeria daca nu mai sunt locuri

Probleme intampinate :

- Frizerul trebuie programat astfel incat sa nu se intre in conditiile de cursa.
- Problema mai poate fi intampinata si atunci cand in cazul unei centrale telefonice sunt apeluri in asteptare.

Solutia si implementarea acestuia sunt descrise in paragraful, respective codul urmator : in momentul in care frizerul soseste la munca, executa procedura barber, ceea ce determina blocarea lui la semaforul customers pentru ca acesta are initial valoarea 0. Dupa acest procedeu frizerul adoarme si ramane asa pana la sosirea primului client

Parametrii implementarii :

- Customers – Numara clientii care asteapta (in afara de clientul care se afla pe scaunul de tuns)
- Barbers – numarul de frizeri (0 sau 1)- care nu fac nimic in asteptarea clientilor

- Mutex- pentru excluderea mutual
- Waiting – contor pentru clientii care asteapta (similar cu customers), folosim acest contor deoarece nu putem citi valoarea curenta a unui semafor

```
#define chairs 20                       // numarul de scaune pt clientii care asteapta
typedefint   semafor;
semaforcustomers = 0;                   //nr.de clienti care asteapta
semaforbarbers = 0;                     //nr.de frizeri care asteapta client      i
semaformutex =1;                        //pentru excluderea mutual
intwaiting=0;                           //clienti care asteapta
voidbarber(void)
{
    While (TRUE) {
        down (&customers);              //dormi daca nr. clienti este 0
        down (&mutex);                  //obtine acces la 'waiting'
        waiting = waiting-1;            //decrementeaza nr de clienti care
                                        asteapta
        up(&barbers);                   //un frizer este gata sa tunda
        up(&mutex);                     //elibereaza 'waiting'
        cut_hair();                     //tunde ( in afara regiunii critice)
```

```
        }
    }

    Voidcustomer(void)
    {
        down(&mutex);                   //intra in regiunea critica
        if(waiting < CHAIRS){           // dc nu sunt scaune libere, pleaca
            waiting = waiting +1;  //incrementeaza nr clienti care asteapta
            up(&customers);             // trezeste frizer daca este nevoie
            up(&mutex)                  //elibereaza accesul la ‘waiting’
            down(&barbers);             //dormi dc nr de frizeri liberi este 0
            get_haircut();              // ia loc si se tunde
        }else{
            up(&mutex);                 // frizeria este plina; nu astepta
    }
}
```

C. CONCLUZII:

Concluzionand, putem spune ca tema *"Comunicarea Interprocese"* este necesara pentru sisteme de calcul in vederea:

- Partajarii informatiilor (existenta unui mediu la care sa aiba acces mai mult utilizatori, asemanator cu spatiul WEB)
- Cresterea vitezei de procesare a informatiilor, a se vedea conceptul de **pseudoparalelism [1] (in timpul unei unitati de timp se pot executa mai multe programe) si multiprogramare.**
- Polimorfismul actiunilor, un utilizator de obicei vrea sa indeplineasca mia multe actiuni deodata: editarea, printare, sau compilare in acelasi timp.

BIBLIOGRAFIE:

[1] Sisteme de operare moderne – Andrew S. Tanenbaum, Editia a 2a, Editura Byblos, 2004

[2] Operating system concepts – Silberschatz Galvin Gagne, Sixth Edition, **JOHN WILEY** & **SONS, INC, 2002**

[3] Curs Sisteme de Operare – Sorin Adrian Ciureanu

[4] Freescale Soc Arhitectures a Practical Approach on 4G communication Systems

[5] Problema scriitor-cititor - Prep. Drd. Aritoni Ovidiu