**Probleme clasice de coordonare si sincronizare procese**

**Problema producator-consumator**

Cunoscută și sub numele de buffer de legatură, doua procese împart același buffer de mărime fixă.

Producatorul pune informație în buffer.

Consumatorul scoate acestă informație. Putem aveam n producători și m consumatori.

Problemele apar atunci când producătorul vrea să pună o nouă informație în buffer, dar acesta este deja plin. Soluția este intrarea producătorului în repaus și pornirea lui atunci când consumatorul a scos una sau mai multe informații. În același mod, în cazul în care consumatorul vrea să scoată o informație din buffer și constată că buffer-ul este gol, intră în repaus până când producatorul introduce ceva în buffer, moment în care repornește.

Pentru a ține evidența numărului de informații din buffer este nevoie de o variabilă – countul. Dacă N este numărul maxim de informații pe care le poate susține buffer-ul, codul producătorului va verifica mai întâi dacă în count se află N informații. Dacă da, producatorul va intra în repaus; dacă nu, producătorul va adaugă o informație și va mări countul.

Pentru consumator este similar: mai întâi se testează countul pentru a se vedea dacă este 0. Dacă da, intră în repaus; dacă este mai mare de zero, scoate o informație și scade countul. Fiecare dintre procese verifică să vadă dacă celălalt ar trebui să fie în repaus, și dacă nu, il repornește.

Pentru a exprima în C apeluri ale sistemului cum sunt SLEEP și WAKEUP vor fi reprezentate sub formă de apeluri către rutinele de bibliotecă. Ele nu fac parte din biblioteca C standard, dar este foarte posibil ca ele să fie disponibile pe orice alt sistem care efectiv a avut aceste apeluri ale sistemului. Procedurile enter\_item și ­remove\_item, care nu sunt arătate, se ocupa cu evidența informațiilor introduse și scoase din buffer.

#define N 100 /\* număr sloturi în buffer \*/

int count = 0; /\* număr element în buffer\*/

void producer(void)

{

while (TRUE) { /\* se repetă pentru totdeauna\*/

produce\_item(); /\* se generează elemental următor \*/

if (count == N) sleep(); /\* dacă bufferul este plin, sleep \*/

enter\_item(); /\* pune element în buffer \*/

count = count + 1; /\* incrementează numărul de elemente din buffer \*/

if (count == 1) wakeup(consumer); verifică dacă bufferul era gol \*/

}

}

void consumer(void)

{

while (TRUE) { /\* se repetă pentru totdeauna \*/

if (count == 0) sleep(); /\* /\* dacă bufferul este gol, sleep \*/

remove\_item(); /\* scoate element din buffer \*/

count = count – 1; /\* decrementează numărul de elemente din buffer \*/

}

}.

Condiția de competiție poate să apară datorită accesului nelimitat la count. Buffer-ul este gol și consumatorul tocmai a citit contul pentru a vedea dacă este 0. În acest moment temporizatorul decide să oprească temporar rularea consumatorului și să înceapă rularea producatorului. Producatorul introduce o informație în buffer, mărește contul și observă că în acest moment are valoarea 1. Raționând că valoarea contului era 0 și consumatorul trebuie să fie în repaus, producatorul apelează wakeup pentru a reporni consumatorul.

Din păcate, consumatorul înca nu este în mod logic în repaus, astfel încat semnalul de repornire este pierdut. La urmatoarea rulare a consumatorului acesta va verifica valoarea countului citit anterior , va constata ca este 0 și va intra în repaus. Mai devreme sau mai tarziu producatorul va umple buffer-ul și va intra și el in repaus. Ambele vor ramane în această stare.

Esența acestei probleme este faptul că o comandă de repornire trimisă unui proces care (înca) nu se află în repaus este pierdută. Dacă aceasta nu s-ar pierde, totul ar funcționa. O remediere rapidă ar fi să se modifice regulile pentru a adăuga în acest cadru un bit de repornire aflat în așteptare. Când o comandă de repornire este trimisă unui proces care nu se află încă în stare de repaus, acest bit este pornit. Mai târziu, când procesul încearcă să intre în repaus, dacă acest bit este pornit, va fi oprit, dar procesul va rămâne pornit. Acest bit este un fel de bancă pentru semnale de repornire.

**Problema producator-cosumator cu ajutorul semafoarelor**

Semafoarele rezolvă problema repornirilor pierdute, este foarte important ca acestea să fie implementate ca indivizibile. În mod normal ele implementeaza apeluri ale sistemului, cu sistemul de operare dezactivând pentru foarte scurt timp toate întreruperile în timp ce testează semaforul, o actualizeaza și trece procesele în stare de repaus, dacă este necesar. Deoarece toate aceste acțiuni au nevoie doar de câteva instrucțiuni, dezactivarea întreruperilor nu cauzează nici o problema. Dacă se utilizează mai multe procesoare, fiecare semafor ar trebui protejat de o variabilă de blocare, utilizandu-se instrucțiunile TSL pentru a ne asigura că semaforul este verificat doar de un procesor odată. Operația de semafor durează doar câteva microsecunde, în timp ce producatorul sau consumatorul ar putea dura foarte mult.

#define N 100 /\* număr sloturi în buffer \*/

typedef int semaphore ; /\* semafoarele sunt un tip special de întregi \*/

semaphore mutex = 1; /\* accesul controlerului în regiunea critică \*/

semephore empty = N; /\* numărul sloturilor goale de pe buffer \*/

semaphore full = 0; /\* numărul sloturilor pline de pe buffer \*/

void producer()

{

int item;

while(TRUE) /\* se repetă pentru totdeauna \*/

{

produce\_item(&item); /\* generează ceva pentru a pune în buffer \*/

down(&empty); /\* decrementează variabila empty \*/

down(&mutex); /\* intră în regiunea critică \*/

enter\_item(item); /\* pune noul element în buffer \*/

up(&mutex); /\* părăsește regiunea critică \*/

up(&full); /\* incrementează variabila full \*/

}

}

void consumer()

{

int item;

while(TRUE) /\* se repetă pentru totdeauna \*/

{

down(&full); /\* incrementează variabila count \*/

down(&mutex); /\* intră în regiunea critică \*/

remove\_item(&item); /\* scoate un item din buffer \*/

up(&mutex); /\* părăsește regiunea critică \*/

up(&empty); /\* incrementează variabila empty \*/

consume\_item(item); /\* folosește elementul\*/

}

}

Această soluție utilizeaza trei semafoare: unul numit full pentru numărarea slot-urilor pline, unul numit empty pentru numărarea slot-urilor goale și unul mutex pentru a se asigura că producatorul și consumatorul nu accesează buffer-ul în același timp. Full este inițial 0, empty este inițial egal cu numarul de slot-uri din buffer, iar mutex este inițial 1. Barierele care încep de la 1 și sunt utilizate de 2 sau mai multe procese pentru a se asigura că doar unul dintre ele poate să intre în zona sa critică la un moment dat se numesc semafoare binare. Dacă fiecare proces execută o comandă DOWN chiar înainte de a intra in zona sa critică, și o comandă UP imediat după ieșire, este garantată excluderea reciprocă.

Într-un sistem care utilizează semafoare, modul natural de a ascunde întreruperile este asocierea unui semafor, inițial setată la 0, pentru fiecare dispozitiv I/O. Imediat dupa pornirea unui dispozitiv I/O, procesul administrator execută o comandă DOWN la semafor asociată, blocându-se imediat. Când se produce întreruperea, un remediator al întreruperii execută o comanda UP la semaforul asociat, lucru care face ca procesul relevant să fie din nou pregătit pentru rulare. În acest model se face executarea unei comenzi UP la bariera dispozitivului astfel încat la pasul 7 administratorul să poată rula managerul dispozitivului. Bineînțeles, dacă în acest moment existp mai multe procese pregatite, administratorul poate să decidă rularea unui proces mai important mai întâi.

Semaforul mutex este utilizat pentru excluderea reciprocă. Este astfel proiectată încât să garanteze faptul că un singur proces va scrie sau va citi în buffer și variabilele asociate la un moment dat. Excluderea reciprocă este necesară pentru evitarea haosului.

Cealaltă utilizare a semafoarelor este pentru sincronizare. Semafoarele full și empty sunt necesare pentru a garanta că anumite secvențe de evenimente apar sau nu. În acest caz ele se asigură că producatorul nu mai rulează în momentul în care buffer-ul e plin și că consumatorul nu mai rulează în momentul când buffer-ul e gol. Aceasta utilizare este diferită de excluderea reciprocă.

**Problema producător-consumator cu transmiterea mesajelor**

Presupunem că toate mesajele sunt de aceeași mărime și că mesajele trimise , dar neprimite încă sunt amortizate automat de către sistemul de operare. În acest caz, este utilizat un total de N mesaje, analog numărului N de slot-uri dintr-un buffer al memoriei comune. Consumatorul începe prin trimiterea de N mesaje goale producătorului. De fiecare dată când producatorul are de furnizat o informație consumatorului, ia un mesaj gol și il trimite înapoi plin. Numarul total de mesaje din sistem rămâne constant în timp, astfel încât ele pot fi stocate într-o anumită cantitate de memorie, stiută dinainte.

Dacă producatorul lucrează mai repede decât consumatorul, toate mesajele vor sfârși prin a fi pline așteptând consumatorul; producătorul va fi blocat așteptând ca un mesaj gol să se întoarcă. În cazul în care consumatorul lucrează mai repede, situația se va inversa: toate mesajele vor fi goale așteptând ca producatorul să le umple; consumatorul va fi blocat așteptând un mesaj plin.

#define N 100

void producer()

{

int item ;

message m;

while (TRUE) {

produce\_item(&item);

receive(consumer,&m);

build\_message(&m,item);

send(consumer,&m);

}

}

void consumer ()

{

int item,I;

message m;

for(I=o;I<N;I++) send (prodicer,&m)

while(TRUE) {

receive(producer,&m);

extract\_item(&m,&item);

send(producer,&m);

consume\_item(item);

}

}

Un mod de a dresa mesajele este de a atribui fiecărui proces o adresă unică și de a programa mesajele de-a se adresa proceselor. Un alt mod este să fie făcută o noua structură de date numită cutie poștală. O cutie poștală este un loc în care se amortizează un anumit număr de mesaje, număr care de obicei se specifică atunci când aceasta este creată. Atunci când sunt utilizate cutiile poștale parametrii de adresă din apelurile SEND și RECEIVE sunt cutiile poștale și nu procesele. Când un proces încearcă să trimită ceva unei cutii poștale deja pline este suspendat până când un mesaj este scos din cutia poștală respectiva făcând loc pentru unul nou.

Pentru problema producator-consumator, atât producătorul cât și consumatorul vor creea cutii poștale destul de mari încât să încapă N mesaje. Producătorul va trimite mesaje conținând date către cutia poștală a consumatorului, iar consumatorul va trimite mesaje goale către cutia poștală a producatorului. Atunci când sunt folosite cutii poștale mecanismul de amortizare este clar: cutia poștală destinatară reține mesajele care au fost trimise procesului destinatar dar care n-au fost acceptate încă.

Extrema opusă cutiilor poștale este eliminarea oricărei amortizări. Atunci când acest tip de abordare este urmat, dacă operația SEND este executată înaintea operației RECEIVE procesul expeditor este blocat până la încheierea operației de primire, moment în care mesajul poate fi copiat direct din expeditor în receptor fără nici un fel de amortizare intermediară. În același mod, dacă operația RECEIVE este executată mai întâi receptorul este blocat până la efectuarea operației de primire. Această strategie este cunoscută sub numele de rendezvous. Este mai usor de implementat decât o schema de mesaj amortizat dar este mai puțin flexibilă de vreme ce receptorul și expeditorul sunt obligați să ruleze în pas de blocaj.

Comunicarea interprocesorală dintre procesele utilizatoare din MINIX (si UNIX) se face prin pipes, care sunt de fapt cutii poștale. Singura diferență reală dintre un sistem de mesaj cu cutii poștale și mecanism pipe este acela că cel de-al doilea nu păstrează limitele mesajelor. Cu alte cuvinte dacă un proces scrie 10 mesaje de câte 100 bytes pentru un pipe și alt proces citește 1000 bytes de pe același pipe, cititorul va primi toate cele 10 mesaje dintr-o dată. În cazul unui adevarat sistem de mesaj fiecare READ ar trebui să returneze un singur mesaj. Bineînțeles dacă procesele cad de acord ca tot timpul să citeasc și să scrie pe pipe mesaje cu mărimă fixă sau să încheie fiecare mesaj cu un caracter special, nu vor exista nici un fel de probleme. Procesele care alcătuiesc sistemul de operare MINIX utilizează pentru comunicarea între ele o schemă de mesaj cu mesaje de mărime fixă.

**Problema bărbierului somnoros**

Un bărbier, un scaun, sala de așteptare cu n scaune.

* Nici un client?

Bărbierul doarme până când un client nou venit il trezește.

- Vine un client?

Barbierul doarme? Trezește-l!

Barbierul este ocupat și există scaune libere? Rezervă un scaun și așteaptă.

Bărbierul este ocupat și nu există scaune libere? Pleacă!

#define CHAIRS 5/\* numărul de scaune disponibile \*/

typedef int semaphore; semaphore customers = 0;/\* numărul de clienți care așteaptă \*/

semaphore barbers = 0; /\* numărul de bărbieri \*/

semaphore mutex = 1; /\* pentru excludere mutuală \*/

int waiting = 0; /\* clienții așteaptă (nu sunt bărbieriți) \*/

void Barber\_Thread(void) {

while (TRUE) {

down(&customers);/\* culcă-te dacă numărul de clienți e 0 \*/

down(&mutex); /\* obținerea accesului la 'waiting' \*/

waiting = waiting − 1;/\* decrementează contorul clienților în așteptare \*/

up(&barbers);/\* un bărbier e acum pregătit să bărbierească \*/

up(&mutex); /\* eliberează 'waiting' \*/

barbiereste(); /\* bărbierește (înafara regiunii critice) \*/ } }

void Customer\_Thread(void) {

down(&mutex); /\* intra în regiunea critică \*/

if (waiting < CHAIRS) {/\* dacă nu există scaune libere, pleacă \*/

waiting = waiting + 1;/\* incrementează contorul clienților în așteptare \*/

up(&customers);/\* trezește bărbierul dacă e nevoie \*/

up(&mutex); /\* eliberează accesul la 'waiting' \*/

down(&barbers);/\* culcă-te dacă nr. de bărbieri liberi e 0 \*/

get\_barbiereste(); /\* așează-te pt. a fi servit \*/

} else {

up(&mutex); /\* prăvălia e plină; nu aștepta \*/

}

}

Este o soluție bună? Nu chiar!

/\* Funcție call-back a procesului/firului de execuție care se ocupă cu eliberarea tichetelor, procesează toate mesajele recepționate și opțional trimite un răspuns \*/

TicketSystem\_MessageCallBack(int pid,int msg\_id, int \*response) {

\*response=NULL;

if (msg\_id==AQUIRE\_TICKET) {

down(&mutex);

if (waiting < CHAIRS) {

waiting = waiting + 1;

\*response=new\_semaphore(NON\_SIGNALED); /\* creează un semafor nou pentru a-i semnala clientului momentul când poate intra la bărbier \*/

customer\_queue\_add(response); /\* adaugă semaforul într-o coadă \*/

}

up(&mutex);

}

}

Customer\_Thread() {

int response;

send\_message(TICKET\_SYSTEM\_PID,AQUIRE\_TICKET,&response);

/\* trimite un mesaj procesului însărcinat cu eliberarea tichetelor pentru a i se asigna un tichet (semafor), care va fi memorat în response \*/

if (response==NULL) { return;}/\* tichetul nu a putut fi obținut \*/

up(&customers);

down(&response); /\* așteaptă rândul \*/

Getbarbiereste();

}

Barber\_Thread() {

while (TRUE) {

down(&customers);

down(&mutex);

waiting = waiting - 1;

customer\_queue\_signal(0); /\* anunță primul client din coadă că poate intra (de asemenea șterge-l din coadă) \*/

up(&mutex);

barbiereste();

}

}

**Problema cinei filozofilor chinezi**

Se consideră cinci filozofi care stau la o masă rotundă, fiecare având în față o farfurie cu mâncare. Între două farfurii vecine există o singură furculiță.

Viața unui filozof constă în alternarea perioadelor când acesta mănâncă și respectiv gândește. Când unui filozof i se face foame, el încearcă să ia furculița din dreapta și furculita din stângă.

Dacă reușește, manâncă o perioadă apoi pune furculițele jos și continuă să gândească.

Implementarea cinei filozofilor

#define N 5

#define LEFT ((i-1)%N)

#define RIGHT ((i+1)%N)

#define THINKING 0

#define HUNGRY 1

#define EATING 2

typedef int semaphore;

int state[N];

semaphore mutex=1;

semaphore s[N];

void philosopher(int i) {

while (TRUE) {

think();

take\_forks(i);

eat();

put\_forks(i);

}

}

void take\_forks(int i) {

down(&mutex);

state[i] = HUNGRY;

test(i);

up(&mutex);

down(&s[i]);

}

void put\_forks(int i) {

down(&mutex);

state[i] = THINKING;

test(LEFT);

test(RIGHT);

up(&mutex);

}

void test(int i) {

if (state[i] == HUNGRY &&

state[LEFT] != EATING &&

state[RIGHT] != EATING) {

state[i] = EATING;

up(&s[i]);

}

}

**Problema rezervării biletelor**

Avem un calculator central conectat la terminale utilizate pentru rezervarea automată de locuri la un concert. Terminale plasate în birouri de rezervare care sunt răspîndite geografic. Când un client intră în birou, funcționarul afișează pe ecran situația din acel moment al rezervărilor. Clientul alege unul dintre locurile libere și funcționarul introduce la terminal numărul locului ales. Un bilet pentru acel loc este eliberat pe loc. În mod clar este necesară eliminarea situațiilor în care se fac rezervări multiple pentru același loc.

O primă încercare de soluționare: Presupunem că dorim să dam o soluție folosind un singur program. O soluție rezonabilă este să se scrie un modul handler de terminal care gestinonează input-ul și afișează informația pe un singur terminal. Având n terminale vom avea n instanțe ale acestui modul, care se execută concurent:

HANDLER1;

HANDLER2;

HANDLER3;

HANDLERn; // În pseudocod conținutul fiecărui handler va fi:

repeat

afișează locurile pe ecran;

read(ALEGERE\_CLIENT);

SEAT[ALEGERE\_CLIENT] := RESERVED;

Eliberează bilet forever

Solutia 1 nu este o soluție bună: user-ul nu este împiedicat să rezerve un loc deja rezervat; se pot rezerva simultan aceleași locuri fără ca sistemul să poată înlătura acest lucru;

Problema se rezolvă utilizând excluderea mutuală iar pentru implementarea ei cea mai simplă metodă este folosirea semafoarelor.

**Problema grădinii ornamentale**

* Oamenii intră într-o grădină ornamental prin una din cele 2 părți.
* Managementul vrea să știe câți oameni sunt în grădină în același timp.

Aceasta se poate generaliza la n părți în loc de 2.

Soluție: Un thread simulează venirea periodică a vizitatorilor în grădină în fiecare secundă dormind pentru câte o secundă și apoi invocând o metodă de incrementare pt obiectul de numărare.

class Counter {

int value = 0 ;

NumberCanvas display ;

Counter ( NumberCanvas n ) {

display.setvalue ( value ) ;

}

Void increment ( ) {

int temp = value ; // citește [ v ]

Simulate.HWinterrupt( ) ;

value=temp + 1 ; // citește [ v+1]

display.setvalue (value) ;

}

display = n ;

}

class Turnstile extends Thread {

NumberCanvas display ; // interfață

Counter people ; // share data

Turnstile(NumberCanvas n , Counter c ) { // constructor

display = n ;

people = c ;

}

public void run ( ) {

try {

display.setvalue(0) ;

for(int i = 1 ; i <= Garden .MAX; i++) {

Thread.sleep (500 ) ; //0.5 second

display.setvalue(i) ;

people.increment() ; //incrementează counterul

}

} catch(InterruptedException e ) { }

}

}