UNIVERSITATEA „POLITEHNICA” BUCURESTI

FACULTATEA DE ELECTRONICA, TELECOMUNICATII SI TEHNOLOGIA INFORMATIEI

**Sisteme de Operare**

- Comunicarea între procese în medii concurențiale -

Studenți : Alexandru Alin Florian

 Bucică Marius Emilian

 Grigore Andrei

 Grupa : 434 Aa

**Cuprins**

1. **Comunicații între procese și excludere reciprocă**
2. **Metode de excludere mutuală cu așteptare**
3. **Excluderea mutuală fără așteptare activă**
4. **Bibliografie**

**COMUNICAŢII ÎNTRE PROCESE ŞI EXCLUDERE RECIPROCĂ**

**(Grigore Andrei)**

În sistemele de operare multitasking, procesele care lucrează împreună îşi dispută resurse comune, pe care doar unul dintre ele le poate folosi deplin la un moment dat. Natura resursei comune nu schimbă natura problemei de comunicaţie între procese. Resursa poate fi o parte de memorie, poate fi un fişier sau un port de acces la operaţii de intrare/ieşire.

În cadrul oricărui sistem UNIX pot rula mai multe programe în regim concurent, regăsite sub numele de *procese*. Procesele pot fi programele utilizator, precum şi o serie

de procese speciale. Aceste procese speciale rulează în fundal (adică nu interacţionează cu utilizatorul), cu rolul de a asigura diverse servicii (cum ar fi tipărirea la imprimantă, bazele de date, server Web ş.a.m.d). Aceste procese poartă denumirea de *daemoni*. Un proces se află la un moment dat într-o anumită stare, după cum vom vedea mai jos. În mod normal, fiecare proces va fi programat să ruleze o perioadă foarte scurtă de timp, după care este trecut într-o coadă de aşteptare, şi aşa mai departe. Stările posibile ale unui proces sunt următoarele:

* rulare (*running*), starea în care procesul primeşte o cuantă de timp pentru a fi

executat în cadrul procesorului (notată cu "R");

* aşteptare (*sleep*) în vederea căpătării unei cuante de timp procesor (notată cu "S");
* aşteptare (*wait*) în vederea realizării unei operaţii de intrare/ieşire (aceste operaţii

fiind considerate mari consumatoare de timp, procesul va fi pus în stare de aşteptare până la terminarea respectivei operaţiuni) (notată cu "D");

* oprit temporar (*stopped*), stare în care procesul nu va fi programat temporar pentru

execuţie (notată cu "T");

* terminare (*terminate*), sistemul pregătind eliminarea procesului din memorie, urmând ca acesta să dispară complet;
* *zombie*, stare în care un proces trece atunci când procesul său părinte nu ia determinat corect încetarea execuţiei sau zona de memorie pe care a ocupat-o nu a putut fi eliberată, ocupând astfel inutil loc în coada de aşteptare (notată cu "Z").

Presupunem că următorul loc liber în coadă este în poziţia 7 şi că are loc următoarea suită de evenimente :

* procesul A citeşte variabila "intrare" şi o memorează într-o variabilă locală, numită URM\_LOC\_LIBER;
* procesul A trece în aşteptare;
* un alt proces, procesul B, intră în execuţie şi scrie numele fişierului de imprimat în poziţia 7, memorată;
* procesul B actualizează la 8 variabila "intrare";
* procesul B trece în aşteptare;
* procesul A redevine activ şi scrie în poziţia 7 din coadă numele propriului fişier, peste ceea ce scrisese procesul B.

Daemon-ul nu va sesiza nici o eroare, dar procesul B nu va putea fi lansat.

Aceste situaţii în care diverse procese exploatează resurse comune, iar rezultatul depinde de procesul rulat şi de momentul când a rulat, se numesc condiţii conflictuale.

Într-o structură de calcul tradiţională, pe arhitectură von Neumann, procesele execută succesiv diferite zone din codul de program propriu. O parte din timp, un proces execută operaţii care nu duc la un conflict de resurse. Vor există ulterior şi momente în care este necesar accesul la resursele comune, când pot apărea conflicte. Partea programului în care se pot produce conflicte este numită secţiune critică.

Situaţiile generatoare de erori sunt relevate de exemplul precedent în care procesul B începe folosirea variabilei comune înainte ca procesul A să fi terminat accesul la resursă. Există multe alte cazuri de memorie comună, fişiere comune şi orice alte resurse comune. Soluţia de prevenire a acestor condiţii conflictuale este de a opri accesul la resurse a mai mult de un proces în acelaşi timp. Evităm conflictele numai dacă doua procese nu intră în propriile secţiuni critice în acelaşi timp. Cu alte cuvinte e necesară o excludere reciprocă : o metodă de asigurare că, în momentul în care un proces accesează o resursă comună, toate celelalte procese vor fi excluse de la folosirea acelei resurse. Alegerea operaţiilor primitive care să implementeze metode pentru excludere reciprocă este o componentă principală în proiectarea unui sistem de operare şi un subiect de care ne vom ocupa în continuare. Această condiţie nu este suficientă pentru a evita conflictele, pentru a avea procese paralele care să coopereze corect şi eficient în utilizarea resurselor comune. În cele ce vor urma, se vor respecta o serie de condiţii de proiectare impuse pentru a avea o bună cooperare între procesele care rulează simultan:

* să nu existe 2 procese simultan în propriile secţiuni critice.
* procesele să fie determinate, adică să producă rezultate identice, indiferent de viteză de calcul sau necesar de CPU.
* nici un proces care rulează în afara secţiunii critice să nu blocheze alte procese.
* nici un proces să nu aştepte arbitrar de mult până la intrarea în secţiunea critică proprie.

Fiecare proces este identificat printr-un *identificator de proces* (*PID Process Identifier*), un număr întreg mai mare decât 1. În mod normal, procesele sunt interactive, adică comunică cu utilizatorul prin intermediul terminalului asociat în cazul programelor cu interfaţă tip linie de comandă (mod text) sau prin intermediul mediului grafic X Window în

cazul programelor dotate cu o asemenea interfaţă. Vom numi acest tip de procese ca fiind în primplan (*foreground*).

O altă categorie de procese sunt acelea care nu interacţionează cu utilizatorul, fiind vorba în general de daemonii menţionaţi mai sus. Spunem că aceste procese rulează în fundal (*background*).

Filosofia UNIX privind modul de viaţă al proceselor este că orice proces este născut de un alt proces, denumit *proces părinte* (identificatorul acestuia este denumit *PPID Parent PID*). La momentul pornirii sistemului, se crează un pseudoproces având PID egal cu zero, care lansează în execuţie procesul *init*, acesta din urmă având PID egal cu unu. Acesta va lansa alte procese, care vor lansa la rândul lor altele etc., astfel încât orice proces care rulează pe maşină are ca strămoş pe *init*.

Fiecare proces deţine un set de drepturi şi proprietăţi, acestea moşteninduse de la părinte la copil. Desigur, procesele copil şi părinte pot funcţiona şi independent unul de celălalt, dar există şi situaţii în care moartea unuia va conduce la supravieţuirea celuilalt.

Dacă un proces îşi pierde părintele, atunci PPIDul său va fi automat considerat ca fiind

egal cu 1 (cu alte cuvinte, părintele său devine *init*).

Procesele reprezintă imaginea dinamică (încărcată în memorie) a unui program, iar acel program este în fapt un fişier executabil deţinut de un utilizator. Astfel, şi procesul va avea un proprietar şi va avea apartenenţă la un grup. Drepturile de acces ale procesului şi controlul său depinde aşadar de drepturile pe care le are proprietarul.

Utilizatorii obişnuiţi îşi pot controla doar propriile procese. Utilizatorul *root* poate controla activitatea tuturor proceselor de pe maşină.

Lista de procese care rulează la un moment dat poate fi consultată prin intermediul comenzii ps. Argumentele uzuale sunt:

* a, are ca efect afişarea tuturor proceselor (nu doar cele aparţinând utilizatorului curent)
* u, realizează afişarea într-un format extins, incluzând şi numele utilizatorului care deţine procesele precum şi starea acestora
* x, conduce la afişarea şi acelor procese care nu au asociat un terminal
* w, afişează informaţiile chiar dacă depăşesc lungimea liniei

Fără argumente, ps are ca efect afişarea proceselor interactive pe care le deţine utilizatorul curent (cel care lansează comanda).

Pentru a vedea lista completă de procese care rulează pe maşină, vom folosi comanda ps aux.

Utilă este şi comanda top, care afişează un "top" al proceselor, ordonate în funcţie de timpul procesor consumat. Vor fi afişate de asemenea şi informaţii privind gradul de utilizare a memoriei.

Una dintre metodele de comunicare clasice UNIX între procesele care rulează pe maşină o constituie *semnalele*. Un semnal este o valoare numerică având o anumită semnificaţie. Ele pot anunţa anumite evenimente apărute în sistem, cum sunt cele de natură hardware (instrucţiune ilegală, întreruperea tensiunii de alimentare etc.) sau software (de exemplu, atunci când un proces încearcă să acceseze o zonă de memorie care nu îi este permisă). Tot semnale primeşte un proces atunci când un proces înrudit îşi încheie execuţia sau la apariţia unor operaţii de intrare/ieşire. De asemenea, utilizatorii pot trimite (desigur, doar proceselor pe care le deţin) direct sau indirect semnale. Astfel, un proces cu probleme poate fi oprit trimiţândui un semnal. Prin apăsarea combinaţiei CTRL+C, de exemplu, se întrerupe execuţia procesului curent prin trimiterea unui semnal către acesta. Procesele pot avea definite anumite comportamente pentru fiecare semnal în parte, sau poate ignora semnalele primite, cu excepţia câtorva, şi anume SIGKILL şi SIGSTOP.

Semnalele principale sunt :

|  |  |  |
| --- | --- | --- |
| **Denumire semnal** | **Valoare semnal** | **Descriere** |
| SIGHUP | 1 | *Hangup*, semnalizează terminarea execuţiei procesului părinte. Este utilizat de mulţi daemoni pentru a determina recitirea fişierelor de configurare etc. |
| SIGINT | 2 | întreruperea procesului (de la tastatură) |
| SIGQUIT | 3 | încetarea execuţiei procesului (de la tastatură, în mod uzual combinaţia CTRL+C) |
| SIGILL | 4 | procesul a efectuat o operaţie invalidă |
| SIGKILL | 9 | oprirea forţată a procesului |
| SIGSEGV | 11 | referinţă invalidă |
| SIGPIPE | 13 | comunicaţie prin *pipe* întreruptă |
| SIGTERM | 15 | terminarea procesului |
| SIGUSR1SIGUSR2 | 1617 | semnale definite de utilizator |
| SIGCHLD | 18 | procesul copil şi-a încheiat execuţia |
| SIGSTOP | 23 | opreşte temporar execuţia procesului |
| SIGCONT | 25 | continuă execuţia procesului după ce acesta a fost oprit temporar |

Trimiterea unui semnal către un proces se face prin comanda:

kill -*semnal nr\_proces*

Semnalul poate fi specificat fie prin valoarea sa numerică, fie prin denumirea sa. Spre exemplu, comanda kill -SIGKILL 3419 (sau kill -9 3419) va trimite semnalul SIGKILL procesului având PID egal cu 3419, ceea ce va provoca încetarea execuţiei acestuia.

**METODE DE EXCLUDERE MUTUALA CU AŞTEPTARE**

În metodele cu aşteptare, în mod uzual dialogul interprocese este conceput de utilizator.

INHIBAREA ÎNTRERUPERILOR

Inhibarea întreruperilor este o metodă foarte simplă de reglementare a comunicaţiilor între procese. În acest caz, fiecare proces blochează mecanismul de lansare a întreruperilor imediat după ce intră în zona sa critică şi deblochează mecanismul imediat la ieşirea din zona critică. Unitatea centrală poate schimba procesele pentru care lucrează doar ca rezultat al unei întreruperi de ceas. Cu întreruperile inhibate nu va sosi nici o întrerupere de ceas, astfel că procesorul nu va avea posibilitatea de a schimba procesul curent. Odată ce un proces a inhibat întreruperile, poate accesa resursele comune fără teama că poate ajunge la o stare conflictuală.

Această soluţie nu este aplicată în practică deoarece dacă o întrerupere de acest gen soseşte şi nu este luată în considerare, poate apărea o funcţionare imprevizibilă din gestionarea defectuoasă a sistemului. Ea este acceptabilă numai integrată în sisteme de operare, dar nu şi pentru procese ale aplicaţiilor.



VARIABILE DE ÎNCHIERE-POARTĂ

Se consideră o variabilă globală, comună tuturor proceselor, numită POARTĂ, iniţial nulă. Când un proces vrea să între în zona critică, testează POARTA. Dacă POARTA e nulă (deschisă), procesul trece variabila în "1" (închide POARTA), şi intră în zona critică. Dacă POARTA e "1" (închisă), aşteaptă pană se deschide. Acesta înseamnă că atunci când o POARTA e deschisă, nici un proces nu se află în zona critică şi atunci când POARTA e închisă, un proces se află în zona sa critică.

Să presupunem că un proces testează poarta şi o vede deschisă, dar înainte de a o închide, intră în lucru un alt proces, care închide poarta. La revenirea primului proces în lucru, va intra şi acesta în zona critică, deoarece a memorat că poarta e deschisă şi apare conflictul. Presupunând că primul proces testează variabila din nou când revine, problema nu este rezolvată pentru că situaţia anterioară poate apărea în continuare, legat de acest ultim moment de testare.

ALTERNARE FORŢATĂ CU VARIABILĂ COMUTATOR COMUNĂ



Fie variabila *comutator* comună şi iniţial nulă. La început procesul (a) inspectează variabila *comutator*; când acesta este "1" procesul (a) intră în zona critică. Procesul (b) găseşte variabila *comutator* pe "1" şi aşteaptă ieşirea procesului (a) din zona critică. Procesul (a), când iese din zona critică, pune variabila *comutator* pe "2" şi poate intra procesul (b) în zona critică. Continua aşteptare în bucla închisă a variabilei *comutator* este un dezavantaj pentru că ocupă timpul CPU. Comutarea proceselor se face la ieşirea unuia dintre procese din zona critică. Să presupunem, într-un caz defavorabil că, după ce procesul (a) iese din zona critică şi comută variabila, procesul (b) intră rapid în zona critică, iese rapid din zona sa critică şi comută variabila comună pe "1". Procesul (b) trece apoi rapid printr-o zonă necritică şi doreşte să intre iar într-o zonă critică. Pentru aceasta procesul (b) va aştepta ca procesul (a) (presupus mai lent) să işi termine zona necritică.

ALTERNARE CU VARIABILĂ LOCALĂ VIZIBILĂ



ALTERNARE CU VARIABILĂ LOCALĂ MODIFICATĂ



Se elimină blocarea în aşteptare infinită de la soluţia anterioară.

**(Sfârșit Grigore Andrei)**

METODA DEKKER **(Bucică Marius Emilian)**



METODA DEKKER

O variabila locala care exprima dorinta de a rula zona critica si o variabila *comutator* care reflecta daca dorinta este simultana,pentru rezolvarea conflictului sunt folosite de catre Metoda Dekker.Acesta metoda realizeaza excluderea reciproca.Această metodă realizează excluderea reciprocă. Variabila comutator face să se evite intrarea în sincronism şi conduce la alternare. Orice conflict se rezolvă în timp finit. Alternanta se aplică numai la coincidenta int - int; cine câştigă primul variabila *comutator*  execută zona critică proprie.

METODA PETERSON

O metodă mai simplă decât Dekker pentru o excludere mutuală:

#define FALS 0, TRUE 1

#define N 2

int activ; /\* variabila care reţine numărul procesului ce se află în zona critică \*/

int vreau[N];

{ vreau[0]=0; vreau[1]=0;

alt\_proces=not ast\_proces;

vreau[proces]=TRUE;

vreau(ast\_proces)="1"; /\* procesul curent doreste să intre în zona critică \*/

activ=ast\_proces;

while((activ==ast\_proces) && (vreau(alt\_proces)==1))

{} /\* sunt pe cale să intru şi altul vrea să intre \*/

}

{ vreau[ast\_proces]=0; } /\* indică ieşirea din zona critică \*/

Înainte de a intra în zonă, fiecare proces va apela funcţia de intrare în zonă, aşteptând până când intrarea este sigură. Apelarea se face cu numărul procesului ca parametru. După ce iese din zona critică, va apela rutina de ieşire din zona. Sa presupunem cel mai dezavantajos caz, când ambele procese vor apela, în acelaşi timp, rutina de intrare: ele vor fi scrise în "activ", dar doar ultimul va fi reţinut şi va ieşi imediat din "while", celălalt va aştepta ieşirea concurentului din zona critică.

EXCLUDEREA CU AJUTORUL INSTRUCŢIUNILOR SPECIALE DE BLOCARE A MAGISTRALEI

 Majoritatea microprocesoarelor au o instrucţiune specială numită TSL - Test and Set Lock, utilizată pentru lucrul cu mai multe CPU. Această instrucţiune determină microprocesorului să emită la un pin special un semnal LOCK, folosit de circuitele hardware pentru interzicerea preluării magistralei de sistem de către CPU concurente. TSL este utilă la execuţia instrucţiunilor cu mai multe faze de acces la resurse, de ex. instrucţiuni Read/Modify/Write, care prezintă fragilitatea de a fi întrerupte în execuţie de procese concurente. Acestea pot corupe resursele procesului întrerupt, care işi reia activitatea în condiţii imprevizibile.

Pentru a folosi TSL, vom folosi o variabilă comună numită FLAG, pentru a sincroniza accesul la o memorie comună. Când variabila FLAG este "0", orice proces are acces la memoria comună. Primul proces care solicită resursă comună va bloca accesul celorlalte procese concurente, punând variabila FLAG pe "1". Eliberarea resursei comune se realizează prin iniţializarea variabilei FLAG.

O soluţie de folosire a acestei instrucţiuni astfel încât să nu existe conflicte este dată de 2 rutine asamblor:

enter\_region: CHECK: MOV AL,1 ; interzice accesul altor procese

LOCK: XCHG FLAG,AL ; TSL

TEST AL,AL ; examinează flag-ul din FLAG

JNZ CHECK

…………………………………

 zonă critică

 …………………………………

leave\_region: MOV FLAG,#0 ; iniţializează FLAG

Procesele vor apela enter\_region înainte de a intra în zona critică, vor emite un semnal de aşteptare tuturor celorlalte procese ce vor încerca accesul la resursele comune. Buna funcţionare depinde de corectitudinea de apelare (la momentele de timp necesare) a rutinelor de intrare, respectiv părăsire a zonei critice.

**EXCLUDEREA MUTUALĂ FĂRĂ AŞTEPTARE ACTIVĂ**

Metodele precedente, inclusiv cele care rezolvă corect problema, cer activitate inutilă pentru aşteptarea eliberării resurselor. Soluţia este blocarea procesului pentru care resursa nu e disponibilă. Acesta nu va efectua activitate inutilă cît timp va aştepta.

Dacă procesul nu e blocat, activităţile de testare inoportună pot crea blocaj, dacă activităţile cu prioritate mare nu pot fi desfăşurate, pentru că resursele există deja la activităţi cu prioritate mică (mecanismul de priorităţi impiedică circulaţia taskurilor). Soluţia este blocarea procesului cu prioritate mare.

EXCLUDEREA PRIN SLEEP-WAKEUP

Se folosesc primitive sistem speciale:

SLEEP - se autoblochează procesul curent.

WAKEUP(p) - trezeşte (activează) procesul p.

**Exemplu**: Avem două procese şi o magazie de capacitate finită:

1. procesul executant realizează produse şi le pune în magazie.



1. procesul beneficiar preia produsele din magazie.

Executantul se va autobloca din producţie în cazul în care magazia e plină.

Beneficiarul se va autobloca din consum în cazul în care magazia e goala.

Executantul şi beneficiarul se vor “trezi” unul pe celălalt în cazul înlăturării problemei care a dus la o blocare.

Totuşi, prin *time-sharing*, se poate ajunge la un acces dublu la o magazie goală: beneficiarul vede magazia goală n=0, dar dispecerul începe procesul executant, care produce şi introduce în magazie, pe care o reactualizează la n=1 şi trece la trezirea beneficiarului. Semnalul de trezire se pierde, pentru că beneficiarul nu este blocat. Procesul beneficiar revine în execuţie prin dispecer şi testează valoarea n, citită înainte de a i se prelua controlul (n=0). Drept rezultat, beneficiarul execută primitiva SLEEP şi se autoblochează. Executantul va umple magazia şi se va autobloca şi el, astfel că ambele procese vor fi blocate.

Soluţie : memorarea bitului de trezire, dacă se trimite unui proces treaz. Ulterior, dacă acesta vrea să se autoblocheze, nu o va face, dar va şterge bitul de memorare a trezirii. Fiecare proces care poate trezi un proces încă treaz va trebui să işi aibă bitul memorat, impreună cu adresa procesului la care se referă.

SEMAFOARE

In 1965 Dijkstra a propus folosirea unei variabile pentru controrizarea wakeup-urilor salvate pentru viitoarea utilizare:semafor.

Semafor = 0 : nici un wakeup nu a fost salvat

Semafor = n > 0 : n wakeup-uri au fost salvate

Se folosesc două proceduri : DOWN şi UP.

Semafor: variabila care numără trezirile pentru a ţine cont de ele apoi. De fapt, un contor.

DOWN:

● verifică valoarea semaforului

● dacă semafor > 0, decrementează semaforul şi continuă execuţia procesului

● dacă semafor == 0, execută sleep

● verificarea valorii, modificarea şi executarea sleep (dacă e cazul) formează o operaţie

atomică (o operaţie indivizibilă) => pe durata

executării acestei operaţii nici un alt proces nu are acces la semafor

● atomicitate este esenţială pentru evitarea condiţiilor de concurenţă

UP:

● incrementează valoarea semaforului

● dacă unul sau mai multe procese executau sleep determinat de semafor (incapabil să execute down), unul din ele va fi ales de sistem şi va executa down

● după up semaforul poate să rămână 0, dar vor fi mai puţine procese blocate de acel semafor

● up este de asemenea o operaţie atomică

● up nu poate bloca procesul respectiv

Dacă unul sau mai multe procese erau blocate de acest semafor (nu puteau să continue o operaţie DOWN(s) anterioară), unul din ele e ales de sistem şi i se permite sa-şi termine operaţia de DOWN care îl blocase => după o operaţie de UP asupra unui semafor cu procese blocate, semaforul va avea tot valoarea 0, dar se reduce numărul de procese blocate de el.

Se foloseste DOWN pentru blocarea proceselor şi va micşora numărul de activări memorate.

Se foloseste UP pemtru trezirea proceselor şi incrementează numărul de activări memorate.

Implementarea funcţiilor UP şi DOWN se face prin operaţii atomice, neîntreruptibile,care odată începute nu se mai opresc pînă cînd nu se sfirsesc. Apelul acestor funcţii se face prin "system call", în care OS va efectua el dezactivarea întreruperilor pe timpul testării semaforului, modificării lui eventuale, sau în timpul blocării procesului, pentru a efectua operaţia într-un tot. Toate aceste acţiuni se fac rapid, în citeva instrucţiuni şi nu deranjează sistemul.

La calculatoare cu mai multe CPU e necesar ca la fiecare CPU să se execute UP sau DOWN precedate de TSL, care să conducă la un arbitraj hardware între CPU pentru a asigura că numai un CPU examinează semaforul.

**Exemplu** : o magazie cu un producător şi un beneficiar.

Vom folosi 3 semafoare:

 full : număr de locuri ocupate în magazie.

 empty : număr de locuri libere în magazie.

 mutex : pentru a asigura accesul exclusiv al unui solicitant la magazie.

Iniţial valorile pentru cele 3 semafoare vor fi :

 full=0 ;magazia goală

 empty=n ;capacitatea magaziei

 mutex=1 ;permite accesul în zona critică

|  |  |
| --- | --- |
| Executant:--------->---------| || execută produsul| DOWN (&empty)| DOWN (&mutex)| pune produsul în magazie| UP (&mutex)| UP (&full)| | --------<---------- | Beneficiar:--------->----------| || DOWN (&full)| DOWN (&mutex)| ia produsul din magazie| UP (&mutex)| UP (&empty)| consumă produsul| | --------<---------- |

Semafoarele empty şi full realizează sincronizarea între procese, iar mutex realizează excluderea reciprocă a proceselor, în timpul accesului la magazie ( dacă mutex=0 înseamnă că un proces lucrează deja cu magazia, procesul care va face DOWN (&mutex) se va bloca, până ce procesul care lucrează cu magazia va elibera accesul făcând UP (&mutex) ).

Funcţionarea sincronizării se face pe baza algoritmului SLEEP / WAKEUP, dar semafoarele vor memora trezirile şi se va evita pierderea lor.

Semafoarele pot fi folosite în sistemul de operare pentru a ascunde întreruperile efectuate de sistem (pentru a le izola de utilizator).

Fiecare I/O are un semafor iniţializat pe 0. După startarea unui dispozitiv de I/O, se apelează DOWN pe acel semafor deci procesul se blocheaza. Cînd vine întreruperea de terminare a operaţiei de I/O, în rutina de tratare a ei se va efectua UP pe semaforul corespunzator, ceea ce va face ca procesul întrerupt să se activeze.

Implementare producator/beneficiar

cu semafoare

#define N 100

typedef int semaphore;

semaphore mutex = 1;

semaphore empty = N;

semaphore full = 0;

void producer(void)

{

int item;

while (TRUE) {

produce\_item(&item);

down(&empty);

down(&mutex);

enter\_item(item);

up(&mutex);

up(&full);

}

}

void consumer(void)

{

int item;

while (TRUE) {

down(&full);

down(&mutex);

remove\_item(&item);

up(&mutex);

up(&empty);

consume\_item(item);

}

}

**(Sfârșit Bucică Marius Emilian)**

MONITOARE **(Alexandru Alin Florian)**

Problema comunicatiei intre procese este posibil rezolvata de primitivele UP/DOWN, care asigura excluderea mutuala a proceselor fara activitate inutila.

Se pot produce unele erori de programare prin lasarea la dispozitie a unor IT ascunse. De exemplu inversarea urmatoare duce la DEADLOCK :

DOWN(&mutex)

DOWN(&empty)

In cazul in care magazia este este plina , empty =0 se blocheaza executantul, beneficiarul putand sa ajunga la DOWN(&mutex) si blocandu-se si el. Aceasta cauza a blocarii o reperzinta accesul utilizatorului la mecanismul intim de gestiune a proceselor, drept rezolvare se izoleaza utilizatorul de primitivele UP/DOWN.

 Un monitor reprezinta un modul de proceduri , variabile si structuri de date care realizeaza si izoleaza mecanismul de excludere reciproca. Procesele nu pot accesa structurile de date interne ale monitorului prin proceduri in afara acestuia, dar pot chema procedurile din monitor.

Excluderea reciproca presupune ca in monitor sa fie activ un singur process la un moment dat, acest lucru este realizabil prin existent unor mijloace de examinare daca mai exista si alt proces activ la acel moment. Dacă exista, procesul ce încearcă intrarea în monitor se va suspenda pînă la eliberarea monitorului. Mijloacele prin care se implementează mecanismul folosesc de regulă semafoare, care nu sunt lăsate la îndemana programatorului decît prin monitor. Programatorul trebuie doar să stie că apelul zonelor critice trebuie facut prin monitor. Se realizează localizarea tratării excluderii reciproce într-o zona de program aflată sub control.

 Aceste procese trebuiesc sincronizate in timpul intereactionarii, astfel acestea se conditioneaza reciproc. Pentru fiecare conditie posibila se impune o variabila de conditie controlata de procedurile WAIT(var) si SIGNAL(var).

 Procedura WAIT este apelata in momentul in care monitorul se afla intr-o situatie in care nu poate continua, acest WAIT se face pentru o variabila de conditie numita “full”. Procesul se blochează şi părăseşte monitorul, permiţînd accesul la magazie altui proces. Deblocarea primului se poate realiza prin apelul procedurii SIGNAL(full).

După apelul procedurii SIGNAL procesul trebuie să părăsească imediat monitorul. Dacă variabila FULL este aştepată de mai multe procese, se va alege prin sistem procesul care se activează.

Rezolvarea problemei executant - beneficiar cu monitor:

|  |  |
| --- | --- |
| monitor execbenefprocedure intrăif (count == N) then WAIT(FULL); intră\_produs; count++; if (count == 1) then SIGNAL(EMPTY); end intră | procedure executant while(true) do produce; execbenef.intră; end while;end executant |
| procedure scoate if (count == 0) then WAIT(EMPTY); scoate\_produs; count--; if (count == N-1) then SIGNAL (FULL); end scoateend monitor | procedure beneficiar while (true) do execbenf.scoate; consuma; end while;end beneficiar; |

In cazul semafoarelor excluderea reciproca este proiectata de catre utilizator pe cand la monitoare aceasta se automatizeaza, functiile individuale SLEEP si WAKEUP sunt inlocuite de WAIT si SIGNAL.

Insa in cazul monitoarelor apar unele probleme printre care :

* Dificultatea de incorporare in compilatoare a secventei de cod care sa implementeze conceptul de monitor.
* Inca cazul sistemelor cu mai multe procese care partajeaza o memorie comuna este mai simpla rezolvarea cu semafoare.

Message Passing

Metoda Message Passing asigura excluderea reciproca si sincronizarea intre procese in sisteme distribuite folosindu-se apelarea procedurilor SEND (dest,&message) care blochează procesul emiţător pină la primirea mesajului de către destinatar si RECEIVE(source, &message) care blochează procesul receptor pînă la primirea mesajului de la sursa specificată.

La aceasta metoda pot aparea diverse probleme cum ar fi :

* perturbarea mecanismului de mesagerie, pot aparea pierderi de mesaje pe retea. Pentru verificarea transmisiei se va trimite un mesaj special ACK, in cazul in care emitatorul nu il primeste va fi reluata transmisia. Insa se poate pierde si mesajul ACK, problema care se rezolva prin acordarea unui numar de ordin a pachetului transmis,astfel receptia pachetului cu acelasi numar de ordine semnifica pierderea de ACK.
* In cazul adresarii pe retele mari pot surveni nedeterminari in cazul lansarii de nume identice in colturi diferite de retea. Pentru rezolvare se introduce domeniul, care semnifică un nume alocat unui grup de maşini si are un nume unic. Adresa va fi complet determinată de *nume\_proces/nume\_maşină/nume\_domeniu.*
* securitatea datelor prin autentificarea unui client pentru ca acesta sa nu fie interceptat de alt server.

Exemplu de message passing :

Remote Procedure Call (RPC) întilnită la sistemele de operare de tip client - server pentru sisteme distribuite (Local Procedure Call Manager - Windows NT).

Rularea unei proceduri locale poate fi expresia unei cereri de la unul din subsisteme sau de la un sistem din retea.

Caller - ul specifcă parametri, porneste operaţia şi se blochează pînă cînd primeste rezultatele operatiei.

La server mesajul ajunge la prelucrarea mesajului traducandu-l într-un apel local la o rutină standard, care se executa. E randul ambalării raspunsului la server şi trimiterea lui unde este desfacut şi obţinut de procesul client.

O problema la RPC ar fi ca parametrii procedurilor care trebuie prevăzuţi la client şi să ajungă la server pot avea reprezentări diferite, datorită construcţiei şi procesoarelor diferite.

O alta problemă ar fi la erori deoarece specificul erorii este legat de structura serverului şi de faza la care s-a facut ruperea transmisiei astfel exista trei strategii de detectare:

1. "at least once": se efectuează de mai multe ori repetat
2. "at most once": se efectuează maxim o dată
3. "maybe": nu garantează nimic, dar este mai uşor de implementat

## Variante de structuri cu message passing

Exista doua variante de astfel de structuri : Mail Box (cutii postale) si fără bufferizare, tip RENDEZ-VOUS.

Mail Box (cutii postale) - sunt structuri de date de tip buffer pentru un număr de mesaje. Parametrii din SEND /RECEIVE vor fi în acest adrese de cutii poştale şi nu de proces. SEND la mail box plin blochează transmiterea,iar RECEIVE va scoate mesaje din cutia poştală şi va debloca SEND.

 In acest caz executantul si beneficiarul vor avea fiecare cate o cutie postala cu N locatii. Fiecare trimite la căsuţa poştală a celuilalt şi preia din cutia proprie,căsuţa poştală continad mesaje trimise destinatarului care nu au fost încă acceptate de acesta.

Rezolvarea cu message-passing a problemei executantului şi beneficiarului :

|  |  |
| --- | --- |
| Proc executant while(TRUE) executa(&item) receive(benef,&m) build\_message(&m,&item) send(benef,&m) endwhileendproc executant | proc beneficiar for(i=0;i<N;i++) send(exec,&m) while(TRUE) receive(exec,&m) preia(&m,&item) send(exec,&m) consumă item endwhileendproc beneficiar |

In acest caz executantul si beneficiarul au buffere cu cate N pozitii, executantul trimitand mesaje cu date în căsuţa poştală a beneficiarului aceasta continand mesaje ajunse la destinaţie dar încă neacceptate,iar acesta la randul lui trimite mesaje vide(ACK) în căsuţa poştală a executantului.

Odată cu obiectul produs executantul construieşte o descriere a acestuia, după care aşteaptă un mesaj de la beneficiar, construieşte un mesaj pe baza obiectului, trimite mesajul la beneficiar odată cu introducerea în magazie, iar beneficiarul trimite executantului N mesaje ACK pentru a permite umplerea magaziei, acesta aşteaptă mesaj de la un executant despre un produs, se face extragerea descrierii din mesaj,se trimite ACK pentru ultimul produs si se consumă produsul.

Construirea semafoarelor cu monitoare

Pentru excludere reciprocă se foloseşte semaforul mutex, iniţializat cu valoarea 1. Cand se apeleaza o procedura din monitor,va trebui executat întii DOWN(mutex), in cazul in care există alt proces în monitor,se ajunge la blocare(DOWN(0)), dacă nu există alt proces în monitor se decrementează în mutex şi execuţia continuă pînă la ultima linie de cod a procedurii,care trebuie să fie UP(mutex) care permite altui proces să intre în monitor.

Pentru testarea condiţiilor şi sincronizarea între procese se folosesc procedurile WAIT şi SIGNAL.

Implementarea primitivei WAIT implică 3 operaţii cu semafoare:

UP(mutex)

 DOWN(condiţie)

 ...

 ...

 DOWN(mutex)

 Implementarea primitivei SIGNAL implică o singură operaţie:

 UP(condiţie).

Exemplu pentru condiţia monitorului "execbenef":

|  |  |
| --- | --- |
| Monitor execbenef Proc intră if (count==N) WAIT(full) intră\_produs count++ if (count==1) SIGNAL(empty) End proc | program producător while(TRUE) produce execbenef.intră end whileend program |
|  Proc scoate if(count==0) WAIT(empty) scoate\_produs count-- if (count==N-1) SIGNAL(full) End procend monitor | program beneficiar while(TRUE) execbenef.scoate consuma end whileend program |

Este lansat primul procesul beneficiar ,in cazul in care contorul este nul,se aşteaptă (WAIT(empty)), cit timp este validă condiţia empty. Se lanseaza procesul producător şi el la un moment dat, produce şi intră în monitor,iar cand contorul ajunge la 1, semnalizează SIGNAL(empty)==UP(empty). Condiţia pe care este blocat beneficiarul devine falsă şi acesta revine, în acest moment existand două procese în monitor: cel blocat trezit si cel care a lansat SIGNAL dar care părăseşte imediat monitorul.

 Executorul părăseşte monitorul şi face UP (mutex) eliberînd resursele astfel beneficiarul întirzie în a procesa resursele.

**(Sfarsit Alexandru Alin Florian)**

**Bibliografie**

1. Kirch, O.; Dawson, T., *Linux Network Administrator’s Guide*, 2nd Edition, O’Reilly & Associates, Inc., 2000.
2. Wikipedia, the free encyclopedia that anyone can edit [ http://www.wikipedia.org ]
3. Curs Sisteme de Operare, 2010.