Sisteme de operare

Cursul 6 - Comunicarea între procese

Drăgulici Dumitru Daniel

Facultatea de matematică și informatică,
Universitatea București

2008

Cuprins

Comunicarea intre procese

Probleme importante legate de comunicarea între procese:
- cum poate un proces să trimită informația către altul;
- evitarea situației când procesele își afectează reciproc, într-un mod nedorit, executarea activităților critice (de ex. două procese încearcă în același timp să aloce ultimul MB de memorie);
- serializarea corectă atunci când există anumite dependențe: dacă procesul A produce datele și procesul B le tipărește, B va trebui să aștepte ca A să producă date înainte de a începe să tipărirească.

Ultimile două probleme se regăsesc și la thread-uri. Prima problemă (trimiterea informației) este ușor de rezolvat în cazul thread-urilor unui același proces, deoarece ele partajază un spațiu comun de adrese; dacă thread-urile sunt din procese diferite, abordarea este aceeași ca în cazul proceselor.

În cele ce umează vom trata cazul proceselor, dar notăm că aceleași probleme și soluții există și în cazul thread-urilor.

Cuprins

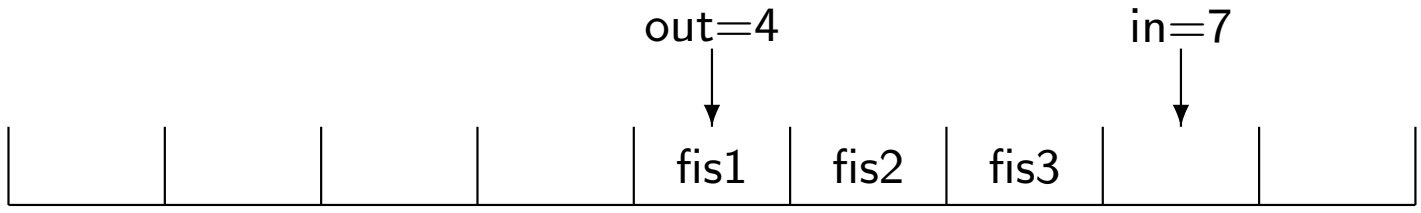
Condiții de cursă

Condiții de cursă (**race conditions**) = situații în care două sau mai multe procese citesc sau scriu date partajate și rezultatul final depinde de cine se execută exact când.

Detectarea greșelilor în programare care conțin condiții de cursă este dificilă - rezultatele celor mai multe teste de execuție sunt bune, dar odată la foarte mult timp ceva ciudat și inexplicabil se întâmplă.

Condiții de cursă

Exemplu:

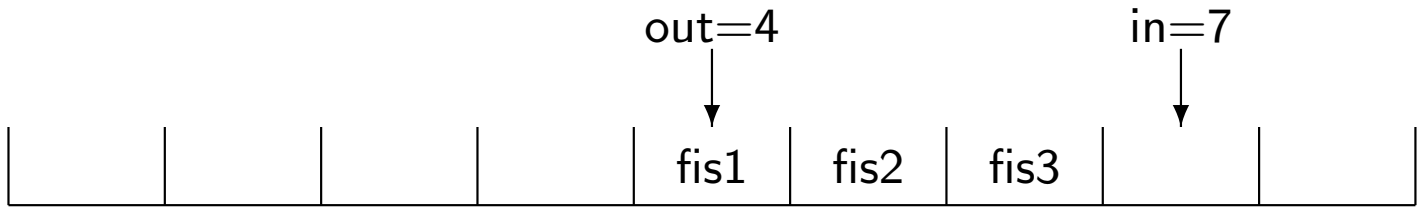


Presupunem că printarea fișierelor se face folosind:
- un tabel organizat ca o coadă, cu intrari pentru fișierele ce urmează a fi printate; el are intrările numerotate 0, ..., N-1;
- două variabile "in" și "out", reținând prima poziție ocupată, respectiv prima poziție liberă; ele sunt accesibile tuturor proceselor (de ex. se află într-un fișier cu două cuvinte);
- un proces care periodic verifică dacă există fișiere în coadă (in $\neq$ out) și dacă da, printează fișierul de pe poziția "out" și incrementează "out" cu 1 modulo N ("`out=(out+1)%N`");

Presupunem ca N=100 iar la momentul curent in=7, out=4.

Condiții de cursă

Exemplu:



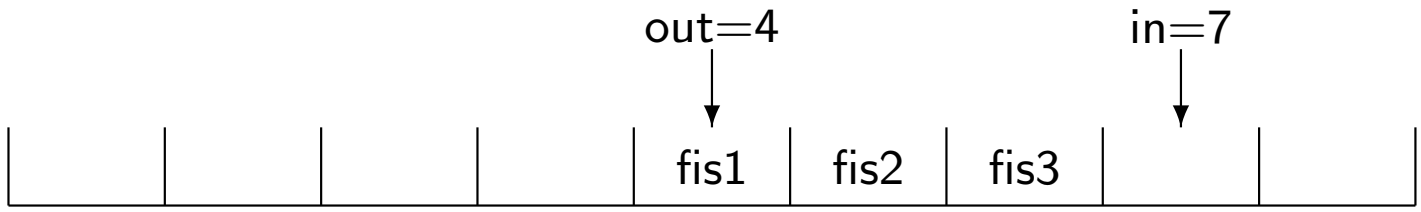
Presupunem că avem două procese (de ex. comenzi de printare de la doi utilizatori) care urmăresc să trimită la printare câte un fișier:

```
proces A: citeste(&urmatorul,in);
          insereaza(fis_a,tabel[urmatorul]);
          scrie((urmatorul+1)%N,in);

proces B: citeste(&urmatorul,in);
          insereaza(fis_b,tabel[urmatorul]);
          scrie((urmatorul+1)%N,in);
```

Condiții de cursă

Exemplu:

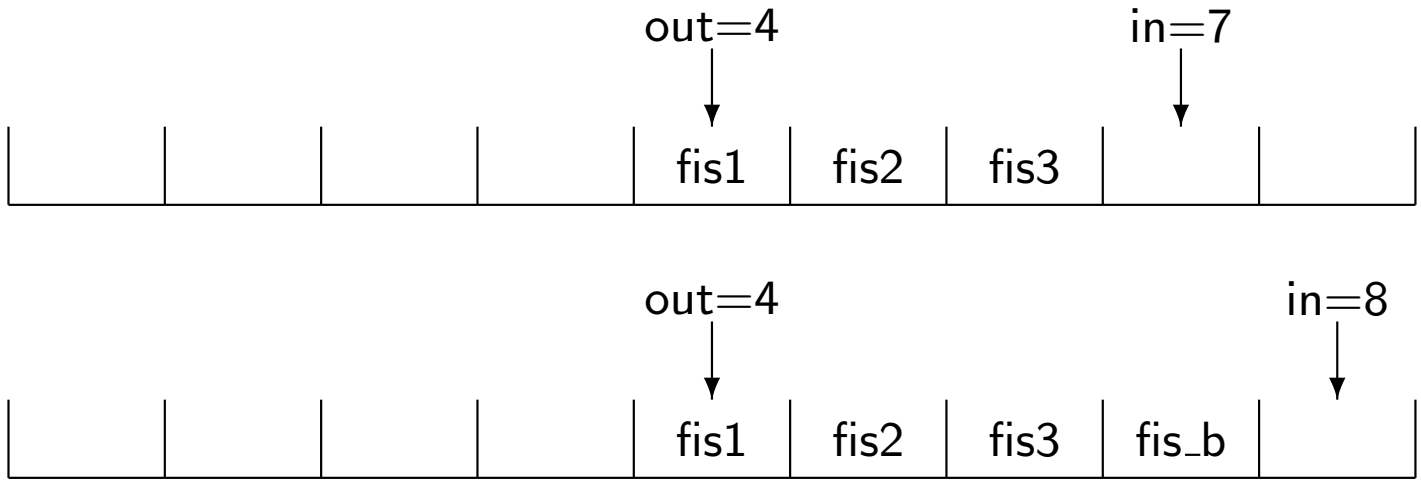


Executate în paralelism intercalat, planificatorul poate decide următoarea secvență de executare a celor două procese:

```
proces A                                                        proces B
-------------------------------------------------------------------------
citeste(&urmatorul,in);                 |
 (urmatorul=7)                          |
-------------------------------------------------------------------------
                                        |       citeste(&urmatorul,in);
                                        |        (urmatorul=7)
-------------------------------------------------------------------------
insereaza(fis_a,tabel[urmatorul]);      |
scrie((urmatorul+1)%N,in);              |
 (tabel[7]=fis_a, in=8)                 |
-------------------------------------------------------------------------
                                        |  insereaza(fis_b,tabel[urmatorul]);
                                        |  scrie((urmatorul+1)%N,in);
                                        |   (tabel[7]=fis_b, in=8)
-------------------------------------------------------------------------
```

Condiții de cursă

Exemplu:



Consecință: fișierul "fis_a" nu se mai printează (utilizatorul care a dat comanda așteaptă degeaba lângă imprimantă).

Cuprins

Regiuni critice

Regiune critică (**critical region**) sau **secțiune critică** (**critical section**) = parte din program care accesează o resursă partajată (structură de date sau dispozitiv).

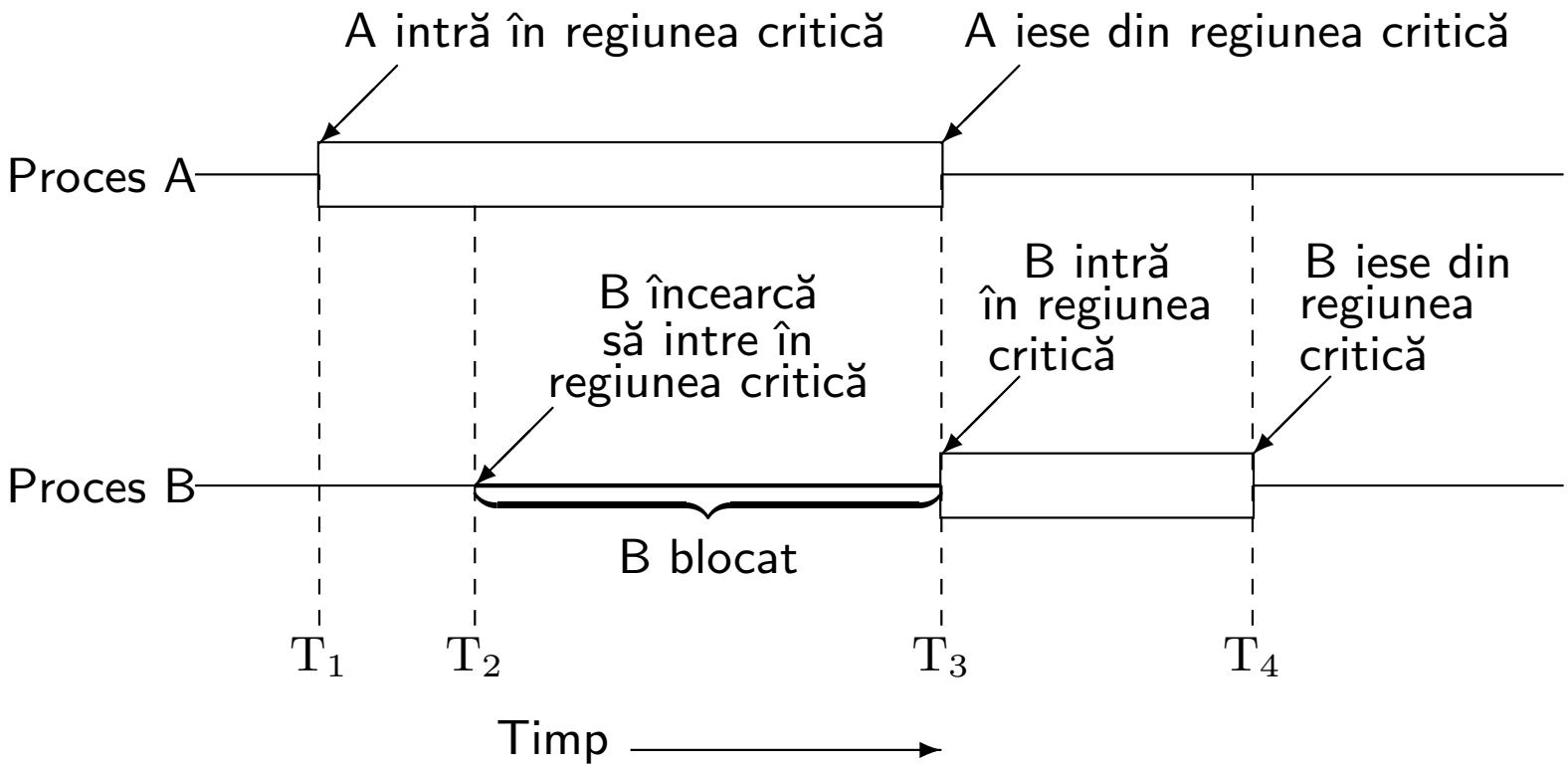
Excluderea mutuală (**mutual exclusion**) = împiedicarea folosirii simultane de către mai multe procese a unei resurse partajate.

Dacă am putea face ca oricare două procese ce partajază anumite resurse să nu fie niciodată în același timp în regiunile lor critice, am putea evita cursele. Deși această cerință evită condițiile de cursă, ea nu este suficientă pentru ca procesele paralele să coopereze corect și eficient folosind datele partajate. Pentru a avea o soluție bună, trebuie îndeplinite patru condiții:

1. Oricare două procese nu se pot afla simultan în regiunile lor critice.
2. Nici un fel de presupunere nu se poate face asupra vitezelor sau numărului de procesoare.
3. Nici un proces care rulează în afara regiunii sale critice nu poate bloca alt proces să intre în regiunea sa critică.
4. Nici un proces nu trebuie să aștepte la infinit pentru a intra în regiunea sa critică.

Regiuni critice

Exemplu de comportament pe care ni-l dorim:



Regiuni critice

În cele ce urmează prezentăm câteva metode de realizare a excluderii mutuale, a.î. mai multe procese ce accesează o resursă partajată sa nu se poata afla simultan în regiunea lor critică.

Cuprins

Dezactivarea întreruperilor

Metoda: procesul dezactivează întreruperile imediat ce a intrat în regiunea critică și le reactivează imediat înainte de a ieși din ea.

Astfel, pe parcursul regiunii critice procesul nu mai poate fi întrerupt de nucleu pentru a comuta procesorul la alt proces (comutarea între procese este rezultatul întreruperilor de ceas sau de alt tip, iar acestea sunt dezactivate) - deci nici un alt proces nu îl mai poate incomoda.

Dezactivarea întreruperilor

Avantaje/dezavantaje:
- nu este bine sa dai proceselor utilizator puterea de a dezactiva întreruperile; dacă nu le mai reactivează ?
- dacă sistemul are mai multe procesoare, dezactivarea întreruperilor afectează doar procesorul care a executat instrucțiunea de dezactivare; celelalte procesoare vor rula în continuare procese, care ar putea accesa resursele partajate;
- pentru nucleul SO este convenabil să dezactiveze întreruperile cât timp execută câteva instrucțiuni de actualizare a unor variabile sau liste; dacă de ex. ar apărea o întrerupere cât timp lista proceselor gata de execuție este într-o stare inconsistentă, ar putea apărea condiții de cursă.

Concluzie: metoda este adesea utilă în cadrul nucleului SO, dar nu este adecvată ca mecanism general de excludere mutuală pentru procesele utilizator.

Cuprins

Variabile zăvor

Metoda: se folosește o variabilă partajata (**zăvor**) având valoarea inițială 0; când un proces vrea să intre în regiunea critică, verifică zavorul; dacă este 0, îl face 1, intră în regiunea critică, iar la sfârsit îl face 0; dacă este deja 1, așteaptă să devină 0.

Deci zăvor $= 0$ înseamnă că nici un proces nu este în regiunea critică, zăvor $=$ 1 înseamnă că există un proces aflat în regiunea critică.

Aceasta metodă conține aceeași greșala fatală ca în exemplul anterior cu printarea fișierelor: de ex. un proces poate observa că zăvor $= 0$, intră pe ramura ce duce către regiunea critică, și înainte de a apuca să facă zăvor $= 1$, planificatorul îl întrerupe, comută la alt proces, care observând că zăvor $= 0$ (încă) intră pe ramura ce duce către regiunea critică, etc. - astfel ambele procese se vor putea afla simultan în regiunea critică.

Următoarele metode de obținere a excluderii mutuale sunt corecte și se bazează pe așteptarea ocupată.
În esență, ele fac următorul lucru: când un proces vrea să intre în regiunea sa critică, verifică dacă accesul îi este permis; dacă nu, așteaptă să îi fie permis, executând o buclă strânsă.

Cuprins

Alternarea strictă

Metoda folosește o variabilă partajată care reține al cui este rândul, iar fiecare proces înaintea regiunii critice așteapta să aibe valoarea potrivită, intră în regiunea critică, iar la ieșire îi schimbă valoarea pentru a-i da voie procesului celălalt.

Alternarea strictă

Exemplu (atenție la " ;" de după instrucțiunile `while` interioare !):

```
while(TRUE){
  while(turn!=0); /* loop */
  critical_region();
  turn=1;
  noncritical_region();
}
```

procesul (a)

```
while(TRUE){
  while(turn!=1); /* loop */
  critical_region();
  turn=0;
  noncritical_region();
}
```

procesul (b)

Metoda evită cursele, dar nu garantează respectarea condiției 3 de mai înainte (anume, nici un proces care rulează în afara regiunii lui critice nu poate bloca alte procese).

Alternarea strictă

Exemplu (atenție la ";" de după instrucțiunile `while` interioare !):

```
while(TRUE){
  while(turn!=0); /* loop */
  critical_region();
  turn=1;
  noncritical_region();
}
```

procesul (a)

```
while(TRUE){
  while(turn!=1); /* loop */
  critical_region();
  turn=0;
  noncritical_region();
}
```

procesul (b)

De exemplu, mai sus este posibil următorul scenariu:
- (b) face "`turn=0`", apoi intră în "`noncritical_region()`", care presupunem că durează mult;
- între timp (a) (având "`turn=0`"), efectuează repede "`critical_region()`", "`turn=1`", "`noncritical_region()`", apoi ajunge la "`while(turn!=1)`";
- întrucât (b) încă rulează "`noncritical_region()`", durează mult până va face "`turn=0`", așa că (a) va aștepta mult până să intre iar în "`critical_region()`".

Deci (b), aflat în regiunea necritică, îl împiedică pe (a) să intre în regiunea critică.

Alternarea strictă

Astfel, metoda nu este utilă când unul dintre procese este mult mai lent decât celalalt.

De asemenea, metoda impune proceselor să alterneze strict efectuarea regiunilor critice; aceasta poate cauza limitări - de exemplu, în exemplul anterior cu printarea fișierelor, alternarea strictă ar împiedica un proces să înregistreze consecutiv două fișiere la printare.

Verificarea în continuu a unei variabile până când are o anumită valoare s.n. **așteptare ocupată** (**busy waiting**); ea trebuie în general evitată, deoarece irosește timp pe procesor; este utilă doar când probabilitatea ca așteptarea să fie scurtă este rezonabilă.

Un zăvor care folosește așteptarea ocupată se numește **spin lock**.

Cuprins

Soluția lui Peterson

Metode de obținere a excluderii mutuale fără alternare strictă au obținut T.Dekker (matematician olandez), apoi (mai simplu) G.L.Peterson (în 1981).

Algoritmul lui Peterson este format din două proceduri ce trebuie executate în fiecare proces la intrarea, respectiv ieșirea din regiunea critică și este ilustrat mai jos (fiecare proces apelează procedurile cu numărul său de ordine, 0 sau 1):

Soluția lui Peterson

```
  /* date partajate */
#define FALSE 0
#define TRUE 1
int turn;                          /* al cui e randul ? */
int interested[2]={FALSE,FALSE}; /* toate valorile sunt initial 0 (FALSE) */

 /* proceduri prezente in fiecare proces */
void enter_region(int process){    /* procesul este 0 sau 1 */
  int other;                       /* numarul celuilalt proces */
  other=1-process;
  interested[process]=TRUE;        /* arata ca esti interesat */
  turn=process;                    /* te inscrii la rand */
  while(turn==process && interested[other]==TRUE); /* cicleaza instr.vida */
}
void leave_region(int process){    /* process: cine pleaca */
  interested[process]=FALSE;       /* indica plecarea din regiunea critica */
}
```

Obs: înainte de a intra în regiunea critică, un proces "process" își manifestă interesul setând "interested[process]=TRUE", iar după ieșirea din regiunea critică setează "interseted[process]=FALSE"; "turn" reține ultimul proces interesat; un proces interesat intră în regiunea critică doar dacă celălalt nu este interesat (în particular nu este în secțiunea sa critică), sau este interesat dar s-a anunțat în "turn" mai târziu.

Cuprins

Instrucțiunea TSL

Multe calculatoare (mai ales cele cu mai multe procesoare) au o instrucțiune hardware **TSL** (**Test and Set Lock - testează și setează zăvor**):

```
TSL RX,LOCK
```

care citește în registrul "RX" conținutul zăvorului "LOCK" (word aflat în memorie), apoi salvează o valoare nenulă în zăvor; citirea și salvarea sunt **indivizibile** (nici un procesor nu poate accesa word-ul din memorie până nu se termină instrucțiunea).

Procesorul care execută "TSL" blocheaza magistrala memoriei pentru a interzice altor procesoare să acceseze memoria pâna când se termină instrucțiunea.

Metoda de obținere a excluderii mutuale bazată pe "TSL" folosește o variabilă partajată "LOCK" pentru a coordona accesul la resursele partajate; când "LOCK" este 0, orice proces o poate seta la 1 cu "TSL", apoi să acceseze resursele partajate, apoi să o reseteze la 0 cu "move"; dacă un proces vrea să acceseze resursele partajate dar "LOCK" este 1, execută așteptare ocupată până devine 0.

Instrucțiunea TSL

Exemplu (într-un limbaj assembler fictiv dar tipic):

```
enter_region:
 TSL REGISTER,LOCK  | copiaza zavorul in registru, apoi seteaza zavorul la 1
 CMP REGISTER,#0    | zavorul a fost 0 ?
 JNE enter_region   | daca nu a fost 0, cicleaza (asteptare ocupata)
 RET                | return catre apelant; se intra in regiunea critica

leave_region:
 MOVE LOCK,#0       | salveaza 0 in zavor
 RET                | return catre apelant
```

Fiecare proces, înainte de secțiunea critică va apela "`enter_region()`", la revenire va executa (cu zăvorul setat la 1) regiunea critică, apoi va apela "`leave_region()`".

Faptul că instrucțiunea "TSL" execută citirea și modificarea zăvorului în mod hardware indivizibil, elimină posibilitatea că între cele două operații sistemul să comute la alt proces care să consulte / modifice și el zăvorul - deci nu apar condiții de cursă.

Atât soluția lui Peterson cât și cea care folosește TSL sunt corecte, dar necesită așteptarea ocupată - procesele așteapta permisiunea de a intra în regiunea critică ciclând într-o buclă stransă.

Pe langă faptul că irosește timp pe procesor, așteptarea ocupată poate avea efecte neașteptate. De ex. presupunem că avem doua procese: H cu prioritate mare și L cu prioritate mică, iar regulile de planificare sunt a.î. H să fie executat de fiecare dată când este în starea de gata de execuție (ready). Atunci, dacă la un moment dat L este în regiunea critică iar H devine ready (de exemplu a terminat o operație de I/O), planificatorul comută la H, care începe așteptarea ocupată, și atunci L nu va termina niciodată secțiunea critică (deoarece în timpul ciclului de așteptare H nu ajunge niciodată sleeping, și astfel procesorul este alocat numai lui H), iar H va aștepta la nesfârșit.

Această situație se numește uneori **problema inversiunii de prioritate** (**priority inversion problem**).

În continuare vom prezenta câteva primitive de comunicare interprocese care fac ca procesul să se blocheze în loc să consume timp procesor, atunci când nu i se permite intrarea în regiunea critică.

Cuprins

Sleep și Wakeup

Unele dintre cele mai simple primitive de comunicare între procese sunt perechea "sleep" și "wakeup".

"sleep" este un apel sistem ce blochează procesul apelant (în starea suspendat), până când un alt proces îl trezește.

"wakeup" trezește un proces (primit ca parametru) din suspendarea produsă de un "sleep".

Alternativ, atât "sleep" cât și "wakeup" au fiecare câte un parametru constând într-o adresă de memorie utilizată pentru a împerechea apelurile de "sleep" cu cele de "wakeup".

Pb. producator-consumator cu "sleep/wakeup"

Aplicație la problema **producator-consumator** (**producer-consumer**) (cunoscută și ca **problema zonei tampon finite** (**bounded-buffer**)): două procese partajază o zonă tampon cu lungime fixă; unul (producatorul) introduce date în el, celălalt (consumatorul) le extrage. Problema se poate generaliza la m producatori și n consumatori.

Dacă producatorul vrea să introducă un element iar zona e plină, sau dacă consumatorul vrea sa extragă un element iar zona e goală, apar probleme; în fiecare caz procesul trebuie să se blocheze până cand celalalt crează condițiile necesare, apoi să fie trezit de acesta. De asemenea, pot apărea condiții de cursă similare celor din exemplul de mai devreme cu printarea fișierelor.

Pb. producator-consumator cu "sleep/wakeup"

O rezolvare INCORECTA ce utilizează "sleep" și "wakeup" este următoarea:

```
  /* date partajate */
#define N 100  /* numarul de locatii din zona tampon */
int count=0;   /* numarul de elemente din zona tampon */
```

```
 /* procesul producator */
void producer(void){
  int item;
  while(TRUE){
    item=produce_item();
    if(count==N)sleep();
    insert_item(item);
    count=count+1;
    if(count==1)wakeup(consumer);
  }
}
```

```
/* procesul consumator */
void consumer(void){
  int item;
  while(TRUE){
    if(count==0)sleep();
    item=remove_item();
    count=count-1;
    if(count==N-1)wakeup(producer);
    consume_item(item);
  }
}
```

Apelurile "sleep" și "wakeup" nu sunt în biblioteca standard C, dar în UNIX/Linux se pot implementa prin trimiterea cu apelul "kill()", sau autotrimiterea cu apelul "raise()" a semnalelor SIGSTOP și SIGCONT de suspendare / trezire a procesului (a se vedea cursul 5).

Pb. producator-consumator cu "sleep/wakeup"

O rezolvare INCORECTA ce utilizează "sleep" și "wakeup" este următoarea:

```
  /* date partajate */
#define N 100  /* numarul de locatii din zona tampon */
int count=0;   /* numarul de elemente din zona tampon */
```

```
 /* procesul producator */
void producer(void){
  int item;
  while(TRUE){
    item=produce_item();
    if(count==N)sleep();
    insert_item(item);
    count=count+1;
    if(count==1)wakeup(consumer);
  }
}
```

```
/* procesul consumator */
void consumer(void){
  int item;
  while(TRUE){
    if(count==0)sleep();
    item=remove_item();
    count=count-1;
    if(count==N-1)wakeup(producer);
    consume_item(item);
  }
}
```

Funcția "`produce_item()`" generează următorul element, "`insert_item()`" introduce un elemet în zona tampon, "`remove_item()`" scoate un element din zona tampon, "`consume_item()`" efectuează o prelucrare oarecare, de ex. tipărește un element.

Pb. producator-consumator cu "sleep/wakeup"

O rezolvare INCORECTA ce utilizează "sleep" și "wakeup" este următoarea:

```
  /* date partajate */
#define N 100  /* numarul de locatii din zona tampon */
int count=0;   /* numarul de elemente din zona tampon */
```

```
 /* procesul producator */
void producer(void){
  int item;
  while(TRUE){
    item=produce_item();
    if(count==N)sleep();
    insert_item(item);
    count=count+1;
    if(count==1)wakeup(consumer);
  }
}
```

```
/* procesul consumator */
void consumer(void){
  int item;
  while(TRUE){
    if(count==0)sleep();
    item=remove_item();
    count=count-1;
    if(count==N-1)wakeup(producer);
    consume_item(item);
  }
}
```

Când producatorul detectează în final "`count==1`", înseamnă că înainte de inserarea item-ului curent zona tampon era goală, deci consumatorul care o golise intrase în "sleep", și atunci (având deja un element în zona tampon) îl trezește cu "`wakeup(consumer)`". El mai poate eventual să insereze câteva elemente înainte ca planificatorul să comute pe consumator.

Pb. producator-consumator cu "sleep/wakeup"

O rezolvare INCORECTA ce utilizează "sleep" și "wakeup" este următoarea:

```
  /* date partajate */
#define N 100  /* numarul de locatii din zona tampon */
int count=0;   /* numarul de elemente din zona tampon */
```

```
 /* procesul producator */
void producer(void){
  int item;
  while(TRUE){
    item=produce_item();
    if(count==N)sleep();
    insert_item(item);
    count=count+1;
    if(count==1)wakeup(consumer);
  }
}
```

```
/* procesul consumator */
void consumer(void){
  int item;
  while(TRUE){
    if(count==0)sleep();
    item=remove_item();
    count=count-1;
    if(count==N-1)wakeup(producer);
    consume_item(item);
  }
}
```

Similar, când consumatorul detectează "count==N-1", înseamnă că înainte de consumarea item-ului curent zona tampon era plină, deci producatorul care o umpluse intrase în "sleep" și atunci îl trezește.

Pb. producator-consumator cu "sleep/wakeup"

```
/* producator */
while(TRUE){
  item=produce_item();
  if(count==N)sleep();
  insert_item(item);
  count=count+1;
  if(count==1)wakeup(consumer);
}
```

```
/* consumator */
while(TRUE){
  if(count==0)sleep();
  item=remove_item();
  count=count-1;
  if(count==N-1)wakeup(producer);
  consume_item(item);
}
```

În rezolvarea anterioară condiția de cursă poate apărea deoarece accesul la "count" nu este restricționat. Putem avea de ex. următorul scenariu:
- consumatorul detectează "count==0" și se înscrie pe ramura "DA"; înainte de a intra în "sleep", planificatorul comută la producator;
- producatorul inserează un element, incremenetează "count", și constatând că "count==1" trimite semnalul de trezire către consumator; planificatorul comută la consumator;
- consumatorul primește semnalul de trezire, dar nefiind logic suspendat (nu a ajuns încă la "sleep"), semnalul se pierde; apoi consumatorul ajunge și intră în "sleep";
- după un timp producatorul umple zona tampon și intră și el în "sleep".
Astfel, ambele procese ramân blocate definitiv.

Pb. producator-consumator cu "sleep/wakeup"

Esența problemei este ca un semnal de trezire venit când procesul nu era suspendat logic este pierdut. Am putea adăuga un **bit de așteptare a trezirii** (**wakeup waiting bit**), care devine 1 când procesul primește un semnal de trezire și nu este suspendat logic; ulterior, dacă procesul încearcă un "sleep" și bitul este 1, apelul sleep nu va suspenda procesul ci doar va face bitul 0. Aceasta rezolvă problema în cazul a două procese, dar pentru mai multe procese (și deci mai multe semnale ce pot veni când nu sunt așteptate) un singur bit nu este suficient; am putea adăuga mai multi biti, dar problema de principiu rămâne. Pentru rezolvarea ei au fost introduse semafoarele ...

Cuprins

Semafoare

Semaforul a fost introdus de E.W.Dijkstra în 1965, ca o variabilă întreagă utilizată pentru numărarea semnalelor de trezire venite ce așteaptă să fie folosite; valorile sale sunt ≥ 0.

Dijkstra a propus două operații "down" și "up" (ce generalizează "sleep" și "wakeup") definite astfel:
- "down" asupra unui semafor verifică dacă semaforul are valoarea > 0;
  dacă da, o decrementează cu 1 (deci folosește unul dintre semnalele de trezire), iar procesul apelant continuă (nu se blochează);
  dacă nu, procesul apelant se blochează (în starea sleeping) (fără a încheia deocamdată operația "down");
  verificarea valorii, decrementarea ei și eventuala blocare a procesului sunt efectuate ca o singură **acțiune atomică**;

Semafoare

Semaforul a fost introdus de E.W.Dijkstra în 1965, ca o variabilă întreagă utilizată pentru numărarea semnalelor de trezire venite ce așteaptă să fie folosite; valorile sale sunt ≥ 0.

Dijkstra a propus două operații "down" și "up" (ce generalizează "sleep" și "wakeup") definite astfel:
- "up" asupra unui semafor incrementează valoarea sa cu 1, iar dacă unul sau mai multe procese erau blocate în așteptarea încheierii unei operații "down" la acel semafor, unul dintre ele este ales de sistem (de ex. aleator) și i se permite să încheie operația "down" (el va decrementa semaforul la loc);
  astfel, după un "up" pe un semafor ce avea procese în așteptare la el, semaforul rămâne tot 0 dar vor fi cu 1 mai puține procese în așteptare la el;
  incrementarea semaforului și eventuala trezire a unui proces se desfășoară de asemenea atomic;
  nici un proces nu se va bloca efectuând "up" (așa cum înainte nici un proces nu se bloca efectuând "wakeup").

Semafoare

Semaforul a fost introdus de E.W.Dijkstra în 1965, ca o variabilă întreagă utilizată pentru numărarea semnalelor de trezire venite ce așteaptă să fie folosite; valorile sale sunt ≥ 0.

Dijkstra a propus două operații "down" și "up" (ce generalizează "sleep" și "wakeup") definite astfel:
- când o operație atomică asupra unui semafor a început, se garantează că nici un proces nu poate accesa semaforul până când operația nu se încheie sau nu se blochează; atomicitatea este esențială pentru rezolvarea problemelor de sincronizare și pentru evitarea condițiilor de cursă.

Semafoare

Uneori în loc de "down" și "up" se folosesc denumirile "P", respectiv "V" (denumiri introduse de Dijkstra în lucrarea sa originală, cu semnificație în limba olandeză).

Semafoare

În mod normal, operațiile "up" și "down" sunt implementate ca apeluri sistem.

Implementarea lor într-un mod atomic este esențială; în acest scop se dezactivează pentru scurt timp întreruperile, câta vreme se testează semaforul, se actualizează și, dacă este necesar, se trece procesul în adormire - toate aceste acțiuni necesită puține instrucțiuni și astfel nu apar probleme cauzate de dezactivarea întreruperilor.

Dacă sunt folosite mai multe procesoare, fiecare semafor trebuie protejat de o variabilă zăvor folosind instrucțiunea "TSL", pentru a permite accesul unui singur procesor la semafor la un moment dat. Astfel apare o așteptare ocupată, dar ea este foarte scurtă (operațiile asupra semaforului durează puțin).

Pb. producator-consumator cu semafoare

O rezolvare a problemei producator-consumator cu semafoare este:

```
  /* date partajate */
#define N 100           /* numarul de locatii din zona tampon */
typedef int semaphore; /* semafoarele sunt un tip special de int */
semaphore mutex=1;      /* controleaza accesul la zona critica */
semaphore empty=N;      /* numara locatiile libere din zona tampon */
semaphore full=0;       /* numara locatiile ocupate din zona tampon */
```

```
 /* procesul producator */
void producer(void){
  int item;
  while(TRUE){
    item=produce_item();
    down(&empty);
    down(&mutex);
    insert_item(item);
    up(&mutex);
    up(&full);
  }
}
```

```
/* procesul consumator */
void consumer(void){
  int item;
  while(TRUE){
    down(&full);
    down(&mutex);
    item=remove_item();
    up(&mutex);
    up(&empty);
    consume_item(item);
  }
}
```

Pb. producator-consumator cu semafoare

Cu "`down(&empty)`" / "`up(&empty)`" se decrementează / incrementează numărul locurilor libere, cu "`down(&full)`"/"`up(&full)`" se decrementează / incrementează numărul locurilor ocupate, iar cu "`down(&mutex)`" / "`up(&mute)`" se marchează intrarea / ieșirea din regiunea critică.

Semafoare

Semafoarele cu valoarea inițială 1 care sunt folosite de două sau mai multe procese pentru a asigura accesul unuia singur în regiunea lui critică la un moment dat se numesc **semafoare binare** (**binary semaphores**).

Dacă fiecare proces efectuează un "down" asupra semaforului binar imediat înainte de a intra în regiunea lui critică și un "up" asupra lui imediat după ieșirea din regiunea critică, excluderea mutuală este garantată.

Semafoare

În exemplul anterior am folosit semafoarele în două moduri diferite:
- "`mutex`" a fost folosit pentru excluderea mutuală - el garanează că doar un singur proces va accesa zona tampon și variabilele asociate acestuia la un moment dat;
- "`full`" și "`empty`" au fost folosite pentru **sincronizare** - ele garantează că anumite secvențe de operații pot sau nu pot apărea; în cazul nostru, ele asigură că producatorul nu se mai execută dacă zona tampon este plină, iar consumatorul nu se mai execută dacă zona tampon este goală.

Cuprins

Mutex

Un **mutex** este o variabilă cu două stări posibile: deschis/închis și în legatură cu care sunt definite două operații (proceduri):
- "mutex_lock": dacă mutex-ul este deschis, el se închide, iar apelantul continuă (nu se blochează);
  daca mutex-ul este închis, apelantul este blocat (la mutex-ul respectiv);
- "mutex_unlock": dacă mutex-ul este deschis, nu face nimic;
  dacă mutex-ul este închis, îl deschide, apoi alege aleator pe unul din cei care erau blocați la el (dacă există) și îl trezește - acesta finalizează închiderea, a.î. în final mutex-ul rămâne tot închis; dacă nu era nimeni blocat la mutex, acesta rămâne deschis;
  în ambele cazuri, cel care a apelat "mutex_unlock" continuă (nu se blochează).

Mutex-urile sunt versiuni simplificate de semafoare, pe care le folosim atunci cand nu vrem să numărăm ci doar să obținem excluderea mutuală; stările deschis / închis semnifică permisiunea / interzicerea de a intra în regiunea critică; înainte de a intra în regiunea critică se execută "mutex_lock" asupra unui mutex, iar după ieșirea din regiunea critică se excută "mutex_unlok" asupra acestuia.

Mutex

Mutex-urile sunt ușor și eficient de implementat și de aceea sunt folosite mai ales la excluderea mutuală a thread-urilor implementate în spațiul utilizator (a se vedea cursul 5).

Dacă este disponibilă o instrucțiune TSL, mutex-urile se pot implementa în spațiul utilizator astfel:

```
mutex_lock:
 TSL REGISTER,MUTEX    | copiaza MUTEX in REGISTRU si seteaza MUTEX la 1
 CMP REGISTER,#0       | MUTEX a fost 0 ?
 JZE ok                | daca MUTEX a fost 0 (deci deschis), iesire
 CALL thread_yield     | MUTEX a fost ocupat, cedeaza procesorul
 JMP mutex_lock        | (la reprimirea procesorului) incearca iar
ok: RET                | revenire in apelant (care va intra in reg. critica)

mutex_unlock:
 MOVE mutex,#0         | pune 0 in MUTEX
 RET                   | revenire in apelant
```

Mutex

Procedurile sunt asemănatoare cu "enter_region" și "leave_region" din exemplul anterior de excludere mutuală bazat pe TSL, dar nu face așteptare ocupată până când planificatorul comută pe alt proces care eliberează zavorul ci cedează procesorul voluntar ("thread_yield").

În cazul thread-urilor implementate în spațiul utilizator, când thread-urile nu sunt întrerupte de nucleu (care nu știe de existența lor) acestea nu pot face așteptare ocupată, deoarece ar cicla la infinit - așa cum am spus în cursul 5, ele trebuie să renunțe voluntar la procesor (iar executivul procesului gazdă va planifica alt thread sau tot același).

Cum "thread_yield" este doar un apel către executivul procesului, operațiile "mutex_lock" și "mutex_unlock" nu necestă apeluri sistem, și astfel thread-urile implementate în spațiul utilizator se pot sincroniza în întregime în spațiul utilizator.

Mutex

În toate metodele prezentate până acum (algoritmul lui Peterson, semafoarele, etc.) este necesar ca anumite locații din memorie folosite de metoda (ex. semafoarele) să fie partajate între procese / thread-uri.

Pentru a putea fi partajate între mai multe procese, locațiile respective nu se pot afla în spațiul de adrese al unui proces (spațiile de adrese ale proceselor sunt în mod normal disjuncte). Ele trebuie stocate în nucleul SO sau, în sistemele care permit acest lucru (ex. UNIX/Linux) plasate în zone speciale de memorie partajată care deși fizic sunt unice, logic pot fi privite de mai multe procese ca fiind în spațiul lor de adrese.

Când mai multe procese pot avea zone comune de memorie partajată, diferența față de thread-uri este mai neclară, dar totuși există: deși acum au o parte din spațiul de adrese comun, procesele au în continuare fișiere deschise diferite, alarme programate diferite, alte caracteristici specifice la nivel de proces diferite (thread-urile unui același proces le partajaza și pe acestea); în plus, comutarea între ele făcută de nucleu este în continuare mai lentă decât comutarea thread-urilor implementate în spațiul utilizator și făcută de procesul gazdă.

Cuprins

Monitoare

Monitorul este o primitivă de sincronizare de nivel înalt (introdusă de Hoare (1974) și Brinch Hansen (1975)), constând într-un pachet special de proceduri și structuri de date, cu proprietatea că:
- procesele pot apela procedurile unui monitor, dar nu pot accesa direct structurile sale de date;
- într-un monitor doar un proces poate fi activ la un moment dat (dacă între timp alt proces încearcă, el este blocat automat până monitorul e liber).

Monitoarele sunt concepte ale limbajelor de programare, tratate de compilator - de exemplu compilatorul văzând că o procedură este definită într-un monitor, tratează apelurile ei altfel. Este sarcina compilatorului, nu a programatorului, să implementeze excluderea mutuală asupra procedurilor din monitor (compilatorul adaugă automat codul necesar - de exemplu folosește un semafor binar sau un mutex).

Monitoarele ne scutesc de dificultățile explicitării tuturor pașilor elementari ai unui algoritm de excludere mutuală - de exemplu, în rezolvarea problemei producator-consumator cu semafoare prezentată mai devreme, dacă cele două operații "down" din codul producatorului erau inversate, apăreau condiții de cursă (exercitiu!). Ele ne ofera câteva proceduri cuprinzătoare cu care putem implementa algoritmul în mai puțini pași. Pur și simplu vom implementa regiunile critice ca proceduri de monitor.

Monitoare

Deși monitoarele ne oferă excluderea mutuală, ele trebuie îmbogățite și cu un instrument de sincronizarea a proceselor - să putem bloca procesul apelant în așteptarea unor semnalizări din partea altor procese privind anumite evenimente.
De exemplu, în problema producator-consumator putem transforma testele de zona tampon plină și goala în proceduri de monitor, dar cum se blocheaza producatorul dacă zona tampon e plină ?

Monitoare

Soluția constă în adăugarea la monitor a unor **variabile de condiție** (**condition variables**) împreună cu următoarele operații asupra lor, efectuabile din procedurile monitorului:
- "wait": blochează procesul apelant, la variabila condiție respectivă; monitorul devine astfel liber și alt proces poate intra în el;
- "signal": planificatorul alege unul dintre procesele blocate la variabila condiție respectivă și îl trezește;
variabila condiție nu acumulează semnalizări ca semafoarele, deci dacă nu există procese blocate la ea, semnalizarea cu "signal" se pierde.

Pentru a evita situația când și procesul care a făcut "signal" și procesul trezit rămân simultan active în cadrul monitorului, s-au propus mai multe soluții:
- (Hoare): procesul trezit continuă, procesul care a facut "signal" se blochează (e mai complicat);
- (B.Hansen): se specifică necesitatea ca după un "signal" procesul să părăsească imediat monitorul - deci folosirea lui "signal" să fie permisă doar la sfârșitul procedurilor monitorului (e mai simplu); această soluție va fi adoptată în cele ce urmează;
- procesul care a facut "signal" continuă, dar numai după ce acesta va părăsi monitorul procesul trezit își va continua execuția.

Monitoare

Rezolvarea problemei producător-consumator cu monitoare, în limbajul Pidgin Pascal (un limbaj imaginar):

```
monitor ProducerConsumer
  condition full, empty;
  integer count;
  procedure insert(item:integer);
  begin
    if count=N then wait(full);
    insert_item(item);
    count=count+1;
    if count=1 then signal(empty);
  end;
  function remove:integer;
  begin
    if count=0 then wait(empty);
    remove=remove_item;
    count=count-1;
    if count=N-1 then signal(full);
  end;
  count=0;
end monitor;
```

```
procedure producer;
begin
  while true do
    begin
      item=produce_item;
      ProducerConsumer.insert(item);
    end
end;

procedure consumer;
  begin
    while true do
      begin
        item=ProducerConsumer.remove;
        consume_item(item)
      end
  end;
```

Monitoare

Operațiile "wait" și "signal" seamănă cu "sleep" și "wakeup", dar pentru că se fac în cadrul monitorului, evită cursa care apărea la primele când între momentul detectării zonei tampon pline / goale de către producator / consumator și intrarea lor în "sleep" procesele puteau fi comutate ajungându-se la inconsistențe.

Monitoare

Subliniem că monitoarele, fiind concepte ale limbajelor de programare tratate de compilator, nu pot fi folosite decât în limbajele ce oferă suport nativ pentru ele (Java, C# și alte limbjaje ce folosesc .NET, Ada, Ruby, Pascal-FC, etc).

În plus, deși funcționalitatea lor este integrată în programul utilizator de către compilator, și sistemul gazdă trebuie să îndeplinească anumite cerințe, de ex. să conțină mecanismele folosite de compilator pentru implementarea monitoarelor: semafoare, TSL, etc.

O rezolvare a problemei producator-consumator cu monitoare în limbajul Java este descrisa în lucrarea lui A.S. Tanenbaum "Sisteme de operare moderne".

Cuprins

Transfer de mesaje

Transferul de mesaje (**message passing**) este o metodă de comunicare între procese care folosește două primitive "send" și "receive", implementate ca apeluri sistem (asemănător semafoarelor și spre deosebire de monitoare) definite astfel:
- "send(destination,&message)": trimite mesajul la destinația "destination";
- "receive(source,&message)": preia un mesaj provenit de la sursa "source" (care poate fi nespecificată și atunci e vorba de orice sursă); dacă nici un mesaj nu este disponibil, apelantul se blochează până la primirea unui mesaj (se poate alege și să se revină din apel imediat, cu cod de eroare).
(parametrul "message" este de fapt o adresă din spațiul de adrese propriu, de unde se ia / unde se pune mesajul).

Transfer de mesaje

Există mai multe variante de implementare a transferului de mesaje:

- Implementarea în SO a unei structuri de date speciale numită **casuță postală** (**mail box**) având un număr fix de locuri de mesaj și un identificator unic - el este indicat ca sursă/destinație la "send" și "receive"; casuța poștală stochează temporar mesajele trimise și încă nepreluate; când un proces încearcă să trimită un mesaj într-o căsuță poștală plină, se blocheaza până se eliberează un loc.

Această implementare seamană cu mecanismul semafoarelor, numai că un mesaj adăugat unei casuțe poștale poate conține mai multă informație decât valoarea +/- 1 adăugată unui semafor; în plus, semaforul nu are o capacitate maximă fixată.

- O abordare opusă căsuțelor poștale este eliminarea stocării temporare a mesajelor iar dacă un proces face "send" și destinatarul nu este deja în "receive", el se blochează (până ce destinatarul face "receive" și preia mesajul). Astfel, primul proces care ajunge la un "send" sau "receive" se blochează până când celalalt ajunge la operația dual, după care își transferă mesajul și repornesc simultan.

Aceasta este o strategie de sincronizare între procese (s.n **rendezvous**). Este mai ușor de implementat decât stocarea temporară a mesajelor, dar emițătorul și receptorul vor fi forțați să păstreze ritmul celui mai lent.

Transfer de mesaje

Transferul de mesaje este folosit și pentru comunicarea între procese aflate pe mașini diferite legate într-o rețea. În acest caz se pot pierde mesaje și trebuie implementat un protocol de comunicare între procese. De exemplu:
- la fiecare mesaj primit, receptorul trimite emițătorului un mesaj de **confirmare pozitivă** (**acknowledgement**); dacă emițătorul nu îl primește într-un anumit interval de timp (mesajul de confirmare pozitivă se poate pierde și el), retrimite mesajul;
- pentru a nu confunda mesajele noi cu cele vechi retransmise și pentru a reconstitui ordinea logică a mesajelor, ele se pot asocia cu numere succesive dintr-o secvență.

De asemenea, într-o rețea se pune problema unui mod de a numi procesele, a.î. specificarea sursei / destinației să fie neambiguă și a unui mod de **autentificare** (**authentification**) a.î. un client să fie sigur că el comunică cu adevăratul server și nu cu un impostor.

Transfer de mesaje

Prezentăm în continuare o rezolvare a problemei producator-consumator cu transfer de mesaje, având la bază următoarele idei:
- atât producatorul cât și consumatorul au câte o căsuță poștală de capacitate N; în ea sunt stocate mesajele primite și încă nepreluate;
- mesajele pot fi pline sau goale; producătorul preia mesaje goale și le retrimite pline, consumatorul invers (deci în sistem vor fi mereu exact N mesaje); inițial consumatorul trimite producatorului N mesaje goale;
- dacă producatorul sau consumatorul apelează "receive" iar căsuța lor poștală este goală, se blochează până vine un mesaj; datorită numărului fix de mesaje din sistem, nici unul dintre ei nu riscă să se blocheze în "send".

Transfer de mesaje

Rezolvarea problemei producător-consumator cu transfer de mesaje:

```
#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);
  }
}
```

Cuprins

Bariere

Barierele sunt un mecanism de sincronizare folosit atunci când mai multe procese execută activități împărțite în etape și nici un proces nu are voie să treacă la o etapă nouă până când toate celelalte procese nu sunt și ele gata să treacă la etapa respectivă.

În acest scop, între etape sunt plasate bariere - practic, apeluri ale unei primitive "barrier" (implementată în general prin intermediul unei funcții de bibliotecă), care blochează procesul apelant până ce și celelalte procese ajung în punctul respectiv - din acel moment vor reporni toate.

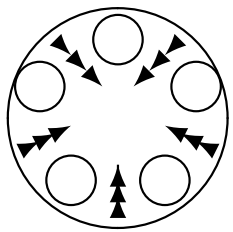
Exemplu: avem de calculat elementele unui șir recursiv de matrici; fiecare matrice se calculează pe baza precedentei: $\mathrm{A_n}$=f($\mathrm{A_{n-1}}$); matricile sunt mari, iar pentru eficiență se folosesc procese paralele care calculează părți din matricea curentă $\mathrm{A_n}$; aceste procese trebuie să se aștepte între ele la sfârșitul fiecărei etape n, deoarece nici o parte din $\mathrm{A_{n+1}}$ nu poate fi calculată până ce $\mathrm{A_n}$ nu este în întregime calculată.

Cuprins

Cuprins

Problema filozofilor care manancă

Problema filozofilor care manancă (**dining philosophers problem**) este o problemă de sincronizare propusă și rezolvată de Edsger Dijkstra în 1965, constând în urmatoarele:
- la o masă circulară stau 5 filozofi; în fața fiecăruia este o farfurie cu spaghetti; între oricare doi este o furculiță; fiecare are nevoie de ambele furculițe alăturate pentru a mânca (uneori problema este formulată cu filozofi chinezi având în față câte o farfurie cu orez și între oricare doi câte un băț și având nevoie fiecare de ambele bețe alăturate pentru a putea manca);
- viața unui filozof constă în perioade de hrănire și perioade de gândire care alternează; când unui filozof îi este foame, încearcă să ia furculițele alăturate, câte una odată (nu contează ordinea), apoi manancă o vreme, apoi eliberează furculițele, și iar gândește.
- cerința: un program pentru fiecare filozof care face ce trebuie, fără să se împotmolească.



Problema filozofilor care manancă

Dificultăți de implementare:

- putem scrie programul a.î. dacă filozofului i se face foame să aștepte furculița stângă, să o ia, apoi să o aștepte pe cea dreaptă, să o ia, apoi să manânce, apoi să elibereze furculițele; atunci, dacă tuturor li se face foame simultan, vor lua furculița stângă (e liberă), apoi vor aștepta la nesfârșit pe cea dreaptă (vecinul nu o eliberează până nu manancă); astfel se ajunge la **interblocare** (**deadlock**);

Problema filozofilor care manancă

Dificultăți de implementare:

- putem modifica programul a.î. după preluarea furculiței stângi filozoful să verifice dacă cea dreaptă este disponibilă, iar dacă nu să o pună jos pe cea stângă, apoi să aștepte un interval de timp fixat, apoi să încerce din nou; atunci, dacă toți filozofii încep acțiunea simultan, vor lua furculița stângă, văzând că nu e liberă cea dreaptă o vor pune jos, apoi după acel interval de timp iar iau furculița stângă, etc; astfel se ajunge la **livelock**, caz particular de **infometare** (**starvation**) - procesele, în așteptarea resursei, își schimbă starea la infinit fără a progresa;

Problema filozofilor care manancă

Dificultăți de implementare:

- putem modifica programele a.î. fiecare filozof, după eșuarea preluării furculiței drepte și eliberarea celei stângi, să aștepte un interval de timp aleator (nu fixat) pâna să încerce din nou; atunci șansa să nu se schimbe nimic este din ce în ce mai mică odată cu trecerea timpului (în aplicațiile unde nu este o problemă dacă încercăm din nou mai târziu soluția este folosită, de ex. în rețeaua locala Ethernet dacă două calculatoare trimit un pachet în același timp, fiecare așteaptă un interval de timp aleator și încearcă din nou); vrem însă o soluție sigură;

Problema filozofilor care manancă

Dificultăți de implementare:

- am putea proteja toate acțiunile unui filozof legate de mâncare (preluarea furculiței stângi, a celei drepte, mâncatul, eliberarea furculițelor) cu un mutex; atunci nu mai apare interblocarea sau infometarea, soluția este corectă dar nu și performantă, deoarece doar un filozof ar putea mânca la un moment dat, deși ar trebui să poata doi (deoarece există 5 furculițe).

Problema filozofilor care manancă

Soluția următoare este corectă și asigură paralelismul maxim pentru un număr arbitrar de filozofi.

Se folosește un vector "`state`" cu stările fiecărui filozof: `THINKING`, `HUNGRY` (dorește să mănânce și încearcă să preia furculițele), `EATING`.

Un filozof poate trece în starea `EATING` doar dacă nici unul din vecinii săi nu este în starea `EATING`.

Se folosește un semafor "`mutex`" pentru a proteja regiunile critice (ele sunt cuprinse între "`down(&mutex)`" și "`up(&mutex)`").

De asemenea, se folosește un vector "`s`" de semafoare, câte unul pentru fiecare filozof, a.î. orice filozof înfometat să se poată bloca dacă vreun vecin al său mănâncă.

Fiecare proces-filozof execută procedura "`philosopher()`" ca pe un cod principal, restul fiind proceduri obișnuite (nu procese separate).

Problema filozofilor care manancă

```
 /* date partajate */
#define N 5                     /* numarul de filozofi */
#define LEFT (i+N-1)%N          /* indexul vecinului stang al lui i */
#define RIGHT (i+1)%N           /* indexul vecinului drept al lui i */
#define THINKING 0              /* filozoful gandeste */
#define HUNGRY   1              /* filozoful incearca sa ia furculitele  */
#define EATING   2              /* filozoful mananca */

typedef int semaphore;          /* semafoarele sunt un tip special de intreg */
int state[N]={0,0,0,0,0};       /* vector pentru a retine starea fiecaruia */
semaphore mutex=1;              /* excludere mutuala pentru regiunile critice */
semaphore s[N]={0,0,0,0,0}; /* cate un semafor pentru fiecare filozof */

 /* programul unui filozof */
void philosopher(int i){        /* i: numarul filozofului, de la 0 la N-1 */
  while(TRUE){                  /* repeta la infinit */
    think();                    /* filozoful gandeste */
    take_forks();               /* preia 2 furculite sau blocheaza-te */
    eat();                      /* filozoful mananca */
    put_forks();                /* depune ambele furculite */
  }
}
```

Problema filozofilor care manancă

```
void take_forks(int i){      /* i: numarul filozofului, de la 0 la N-1 */
  down(&mutex);              /* intra in regiunea critica */
  state[i]=HUNGRY;           /* inregistreaza ca filozofului i ii e foame */
  test(i);                   /* incearca sa obtina 2 furculite */
  up(&mutex);                /* iese din regiunea critica */
  down(&s[i]);               /* blocare daca furculitele n-au fost obtinute*/
}

void put_forks(int i){       /* i: numarul filozofului, de la 0 la N-1 */
  down(&mutex);              /* intra in regiunea critica */
  state[i]=THINKING;         /* filozoful a terminat de mancat */
  test(LEFT);                /* vezi daca vecinul stang poate manca acum */
  test(RIGHT);               /* vezi daca vecinul drept poate manca acum */
  up(&mutex);                /* iesire din regiunea critica */
}

void test(int i){            /* i: numarul filozofului, de la 0 la N-1 */
  if(state[i]==HUNGRY&&state[LEFT]!=EATING&&state[RIGHT]!=EATING){
    state[i]=EATING;
    up(&s[i]);
  }
}
```

Problema filozofilor care manancă

Obs: Funcția "`test()`" este tentativa de a mânca; apelul ei, ca și schimbările de stare se fac în zonă de excludere mutuală, a.î. când un proces face aceste operații, celelalte procese au o stare fixată - astfel nu pot apărea inconsistențe.

Cuprins

Problema cititorilor și scriitorilor

Problema filozofilor care mănâncă modelează procese aflate în competiție pentru obținerea accesului exclusiv la un număr mic de resurse, ca dispozitivele de I/O.

Problema cititorilor și scriitorilor modelează accesul la o baza de date.

Problema cititorilor și scriitorilor (**Readers-writers problem**, Courtois et al., 1971):
- considerăm o bază de date, cu mai multe procese concurente, unele care citesc, altele care scriu în ea;
- este permisă citirea simultană de către mai multe procese, dar cât timp un proces scrie, nici un alt proces nu poate citi sau scrie.

Cum putem programa cititorii și scriitorii ?

Pb. cititorilor și scriitorilor

```
typedef int semaphore;
semaphore mutex=1;                  /* controleaza accesul la "rc" */
semaphore db=1;                     /* controleaza accesul la baza de date */
int rc=0;                           /* nr. celor care citesc sau vor s-o faca */
void reader(void){
  while(TRUE){                      /* repeta la infinit */
    down(&mutex);                   /* obtine acces exclusiv la rc */
    rc=rc+1;                        /* un cititor in plus */
    if(rc==1)down(&db);             /* daca acesta este primul cititor ... */
    up(&mutex);                     /* elibereaza accesul exclusiv la "rc" */
    read_data_base();               /* acceseaza datele */
    down(&mutex);                   /* obtine acces exclusiv la rc */
    rc=rc-1;                        /* un cititor mai putin acum */
    if(rc==0)up(&db);               /* daca acesta este ultimul cititor ... */
    up(&mutex);                     /* elibereaza accesul exclusiv la "rc" */
    use_data_read();                /* nu este in regiunea critica */
  }
}
void writer(void){
  while(TRUE){                      /* repeta la infinit */
    think_up_data();                /* nu este in regiunea critica */
    down(&db);                      /* obtine acces exclusiv */
    write_data_base();              /* actualizeaza datele */
    up(&db);                        /* elibereaza accesul exclusiv */
  }
}
```

Problema cititorilor și scriitorilor

Obs: Primul cititor care intră în baza de date face "`down(db)`"; în continuare alți cititori pot intra imediat (deoarece "`rc`" nu mai este 1 deci nu mai fac "`down(&db)`") - ei doar incrementează "`rc`"; în schimb dacă vine un scriitor, el este blocat în "`down(db)`" (pe care-l face necondiționat); pe masură ce cititorii pleacă, decrementează "`rc`", iar ultimul (deoarece "`rc`" este 0) face și "`up(&db)`", permițând astfel unui scriitor blocat, dacă există, să intre.

Dacă un scriitor accesează baza de date, face "`down(&db)`" necondiționat și astfel orice alt cititor sau scriitor care va încerca accesul va fi blocat pâna când scriitorul face "`up(&db)`".

Problema cititorilor și scriitorilor

Deficiența a soluției de mai sus: dacă o citire durează 5 secunde iar cititorii (re)vin la fiecare 2 secunde, odată ce a intrat un prim cititor vor fi în permanență cititori activi în baza de date, a.î. dacă între timp vine un scriitor el este blocat veșnic.

O soluție: rescrierea programului a.î. la sosirea unui cititor, dacă era un scriitor în așteptare, cititorul este blocat în spatele scriitorului (în loc să fie acceptat imediat) - astfel, un scriitor va trebui să aștepte doar ieșirea cititorilor activi în momentul sosirii lui (exercițiu !); metoda însă micșorează concurența, deci are o performanță mai slabă.

Courtois et al. prezintă o soluție care dă prioritate scriitorilor.

Cuprins

Problema frizerului somnoros

Problema frizerului somnoros (**The Sleeping Barber Problem**) (atribuită adesea lui Edsger Dijkstra, 1965):
- într-o frizerie, există un frizer, un scaun de tuns și n scaune de așteptare;
- dacă nu sunt clienti, frizerul doarme pe scaunul de tuns;
- dacă sosește un client când frizerul doarme, îl trezește iar frizerul începe să-l tundă; dacă sosește un client când frizerul tunde, fie stă pe un scaun de așteptare, dacă există scaune goale, fie părăsește frizeria, dacă nu există scaune goale.

Cerința constă în a programa frizerul și clientii fără a intra în condiții de cursă.

Problema frizerului somnoros

O implementare incorectă poate conduce la interblocare sau infometare, de ex:
- frizerul poate aștepta la nesfârșit un client iar clientul așteaptă la nesfârșit frizerul (interblocare);
- clienții pot să nu decidă abordarea frizerului într-un mod ordonat, a.î. unii clienți pot să nu aibe șansa de a se tunde deși au așteptat (înfometare).

În formularea dată, problema implică un singur frizer ("**the single sleeping barber problem**"); ea se poate generaliza considerând mai mulți frizeri ce trebuie coordonați printre clienții în așteptare ("**multiple sleeping barbers problem**").

Problema este similară multor situații de formare de cozi, de ex. serviciul de relații cu clienții format din mai multe persoane și care dispune de un sistem de apel în așteptare computerizat pentru reținerea unui număr limitat de apeluri primite.

Problema frizerului somnoros

Soluția următoare folosește:
- un semafor "`customers`", care numără clienții în așteptare (se exclude clientul de pe scaunul de tuns, care nu așteaptă);
- un semafor "`barbers`", care numară frizerii ce nu fac nimic, în așteptarea clienților (0 sau 1);
- un semafor "`mutex`", folosit pentru excluderea mutuală;
- o variabiă "`waiting`", care numără și ea clienții în așteptare; ea este de fapt o copie a lui "`customers`" și este necesară deoarece nu putem citi valoarea curentă a unui semafor.

Frizerul execută procedura "`barber()`", iar fiecare client, când vine, execută procedura "`customer()`":

Problema frizerului somnoros

```
 /* date partajate */
#define CHAIRS 5              /* nr. de scaune pentru clientii care asteapta */
typedef int semaphore;
semaphore customers=0;        /* nr. de clienti care asteapta */
semaphore barbers=0;          /* nr. de frizeri care asteapta clientii */
semaphore mutex=1;            /* pentru excluderea mutuala */
int waiting=0;                /* nr. de clienti care asteapta */
```

Problema frizerului somnoros

```
 /* procedura executata de frizer */
void barber(void){
  while(TRUE){
    down(&customers);       /* dormi, daca nr. de clienti este 0 */
    down(&mutex);           /* obtine acces la "waiting" */
    waiting=waiting-1;      /* decrementeaza nr. de clienti care asteapta */
    up(&barbers);           /* un frizer gata sa tunda */
    up(&mutex);             /* elibereaza "waiting" */
    cut_hair();             /* tunde (in afara regiunii critice) */
  }
}
 /* procedura executata de fiecare client */
void customer(void){
  down(&mutex);             /* intra in regiunea critica */
  if(waiting<CHAIRS){       /* daca nu sunt scaune libere, pleaca */
    waiting=waiting+1;      /* incrementeaza nr. de clienti care asteapta */
    up(&customers);         /* trezeste frizer daca e nevoie */
    up(&mutex);             /* elibereaza accesul la "waiting" */
    down(&barbers);         /* dormi, daca nr. de frizeri liberi este 0 */
    get_haircut();          /* ia loc si fii tuns */
  } else {
    up(&mutex);             /* frizeria este plina, nu astepta */
  }
}
```

Problema frizerului somnoros

Obs1: în codul clientului nu există buclă, deoarece fiecare se tunde doar o dată; în codul frizerului există buclă, pentru a prelua următorul client.

Obs2: soluția de mai sus este preluată din lucrarea lui A.S. Tanenbaum "Sisteme de operare moderne"; ea are anumite deficiențe, de ex. este posibil ca un nou client să fie servit în timp ce clientul anterior încă execută "`get_haircut()`" (este în afara regiunii critice); ar trebui adaugată și o sincronizare client-frizer la sfârșitul procedurilor "`cut_hair()`" și "`get_haircut()`" (exercitiu !).

Cuprins

Cazul UNIX/Linux

Comunicarea între procese în UNIX/Linux se poate face prin:
- semnale;
- IPC: segmente de memorie partajate, vectori de semafoare, cozi de mesaje;
- fișiere tub (pipe).

Cuprins

Semnale

Semnalele (**signal**) sunt entități ce au ca singură informație asociată un cod numeric întreg (tipul semnalului).

Codurile semnalelor sunt de la 1 la NSIG-1 și se pot desemna prin mnemonice scrise cu majuscule și care încep cu "SIG" (SIGINT, SIGQUIT, etc.); atât NSIG cât și menmonicele sunt constante simbolice definite în "signal.h".

Semnalele pot fi primite de procese, fiind trimise de alte procese sau generate de anumite evenimente; când la un proces ajunge un semnal, acesta nu cunoaște expeditorul sau evenimentul generator ci doar tipul semnalului (codul numeric, singura informație asociată semnalului).

Semnale

Un semnal poate fi trimis de un proces altor procese cu apelurile:

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

$\Longrightarrow$ dacă "pid" este > 0, se trimite semnalul "sig" procesului cu PID-ul "pid";

dacă "pid" este $= 0$, se trimite "sig" tuturor proceselor din același grup cu procesul expeditor;

dacă "pid" este $=$ -1, se trimite "sig" tuturor proceselor pentru care procesul apelant are dreptul să trimită semnale, mai puțin procesului "init" care are PID-ul 1 (cu unele excepții);

dacă "pid" este $<$ -1, se trimite "sig" tuturor proceselor din grupul al cărui lider are PID-ul -PID;

dacă "sig" este 0, nu se transmite nici un semnal, dar se face verificarea erorilor (astfel, putem verifica dacă un proces există trimițându-i 0 și verificând codul de eroare furnizat de "kill()" în errno);

procesul expeditor poate trimite un semnal doar dacă este proces privilegiat sau dacă UID-ul sau EUID-ul lui coincide cu UID-ul sau "saved set-user-ID"-ul procesului destinatar; în cazul semnalului SIGCONT este suficient ca expeditorul și destinatarul să fie în aceeași sesiune;

la succes returnează 0, la eșec returnează -1;

errno posibile: EINVAL, EPERM, ESRCH.

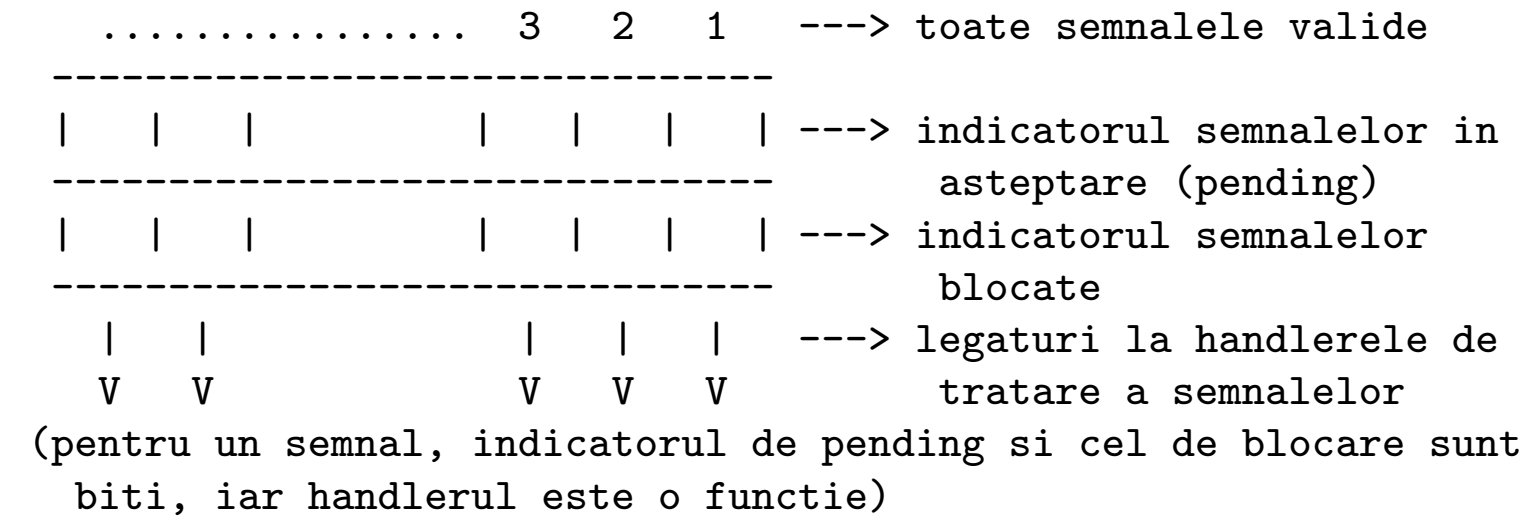
Semnale

```
#include <signal.h>
int raise(int sig);
```

$\Longrightarrow$ se trimite semnalul "sig" însuși procesului curent;
este echivalent cu "kill(getpid(), sig);"
la succes returnează 0, la eșec returnează $\neq 0$.

Semnale

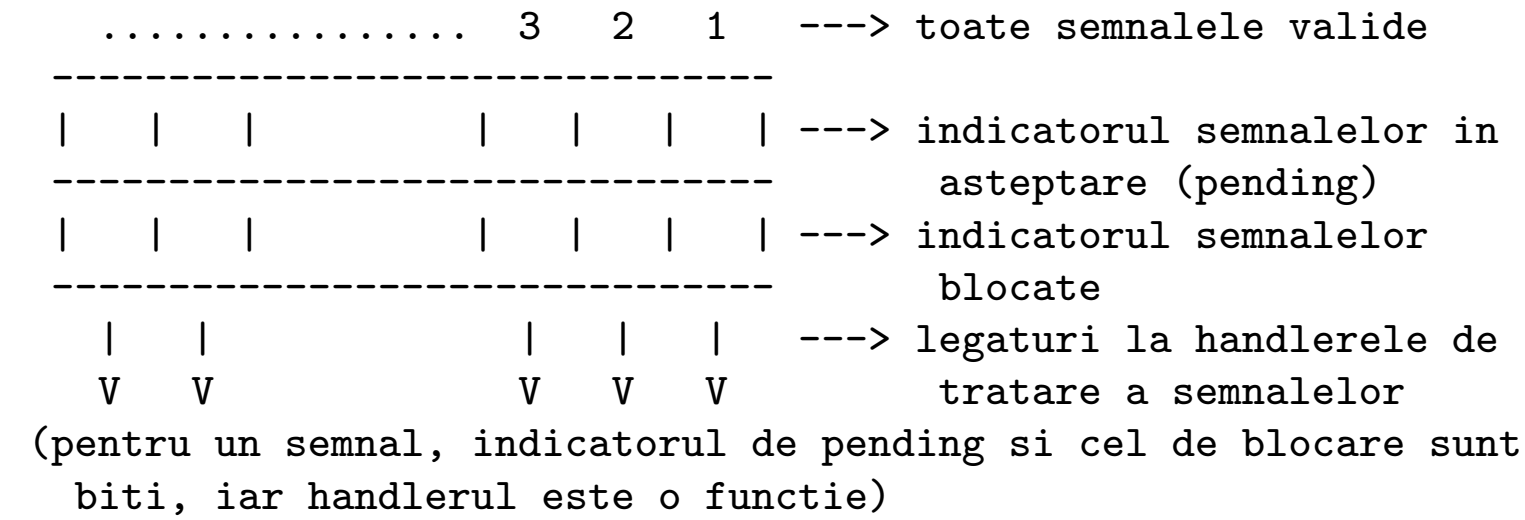
Pentru a gestiona semnalele primite, un proces folosește o structură având cel puțin următoarele câmpuri:



Când la un proces ajunge un semnal n:
- bitul de pending al lui n devine 1; dacă pe perioada cât acest bit este 1 mai vine un semnal n, acesta se va pierde (din fiecare semnal poate fi la un moment dat în pending doar un singur exemplar);

Semnale

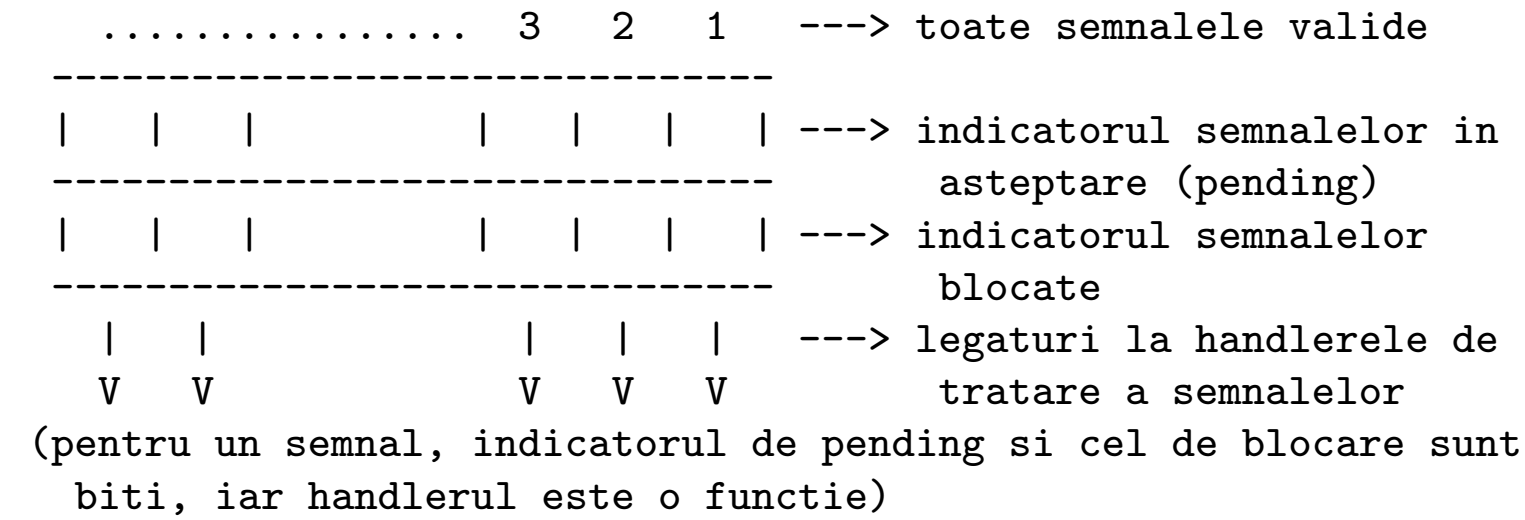
Pentru a gestiona semnalele primite, un proces folosește o structură având cel puțin următoarele câmpuri:



Când la un proces ajunge un semnal n:
- dacă bitul de blocaj al lui n este 1, semnalul n nu va fi tratat (el rămâne în pending); dacă acest bit e 0, semnalul n va fi tratat atunci când va fi posibil (a se vedea mai jos);

Semnale

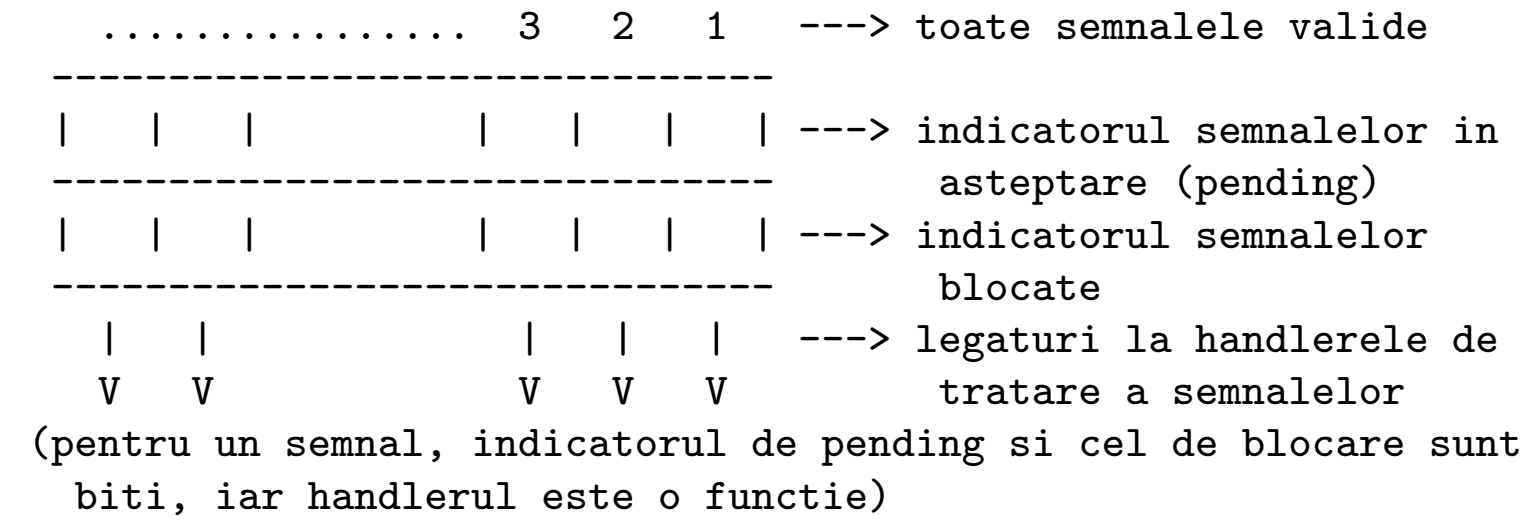
Pentru a gestiona semnalele primite, un proces folosește o structură având cel puțin următoarele câmpuri:



Când la un proces ajunge un semnal n:
- în general un proces tratează semnalele aflate în pending doar atunci când este în user mode, deci atunci când execută instructiuni ale utilizatorului; astfel, când procesul execută un apel sistem, el nu tratează semnalele (toate sunt blocate temporar);

Semnale

Pentru a gestiona semnalele primite, un proces folosește o structură având cel puțin următoarele câmpuri:



Când la un proces ajunge un semnal n:
- tratarea unui semnal n aflat în pending constă în apelarea handlerului asociat; chiar de la începutul executării handlerului bitul de pending al lui n se repoziționează pe 0 (deci procesul poate primi imediat un alt n, care va intra în pending); pe parcursul executării handlerului n este în principiu blocat suplimentar, astfel că dacă între timp vine un nou n, el va fi blocat în pending până la sfârșitul executării handlerului, când va putea fi tratat lansând din nou handlerul; un handler poate fi însă instalat și a.î. acest blocaj suplimentar să nu mai apară (a se vedea mai jos).

Semnale

Pentru fiecare semnal există handlere implicite (ale sistemului), dar pentru majoritatea putem instala handlere proprii (excepții: SIGKILL, SIGSTOP, uneori SIGCONT);

Pentru un semnal putem reinstala handlerul implicit, desemnat prin SIG_DFL (care însă pentru fiecare semnal poate înseamna altceva), sau un handler de ingorare al sistemului, desemnat prin SIG_IGN (pentru cele două constante simbolice putem include "signal.h");

Semnale

SUBLINIEM:
- dacă un proces primește un semnal neblocat n și este într-o zonă unde execută instrucțiuni ale utilizatorului (indiferent unde), el va fi întrerupt și se va executa handlerul h asociat lui n; apoi, dacă h nu a cerut terminarea programului, acesta se reia de unde s-a întrerupt;

Semnale

SUBLINIEM:

- pe parcursul executării lui h mai poate veni un semnal neblocat p și atunci sunt posibile cazurile:

  • dacă h este un handler al sistemului (SIG_DFL sau SIG_IGN): atunci p rămâne în pending până la sfârșitul lui h, deoarece pe perioada lui h procesul este în kernel mode și toate semnalele sunt blocate temporar; după terminarea lui h blocajul suplimentar dispare și p va fi tratat - se va lansa handlerul f al lui p;

Semnale

SUBLINIEM:

- pe parcursul executării lui h mai poate veni un semnal neblocat p și atunci sunt posibile cazurile:
  - dacă h este un handler al utilizatorului (chiar și funcția cu corp vid), atunci sunt posibile subcazurile:
    - dacă p = n:

atunci noul n nu se pierde (pentru că bitul de pending al lui n s-a repoziționat pe 0 chiar de la începutul lui h), dar rămâne în pending pe toata perioada lui h (pentru că n este blocat temporar); după terminarea lui h blocajul suplimentar asupra lui n dispare și noul n va fi tratat (se va lansa iar h); am presupus însă că h nu a fost instalat cu eliminarea blocajului suplimentar, cum am spus mai sus ca se poate;

Semnale

SUBLINIEM:

- pe parcursul executării lui h mai poate veni un semnal neblocat p și atunci sunt posibile cazurile:
  - dacă h este un handler al utilizatorului (chiar și funcția cu corp vid), atunci sunt posibile subcazurile:
    - dacă p $\neq$ n:

atunci h se întrerupe temporar (pentru că sunt instrucțiuni ale utilizatorului), se execută f, apoi se continuă h de unde a rămas;

(în cele de mai sus am presupus că h și f nu produc terminarea programului).

Semnale

SUBLINIEM:

Astfel, presupunând că n și p nu sunt blocate iar h și f sunt handlere ale utilizatorului care nu termină programul și blochează semnalul pentru care au fost lansate pe perioada execuției lor, dacă procesul primește foarte repede secvența n, n, p, va executa: un început de h, apoi un f, apoi sfârșitul primului h, apoi un alt h.

Semnale

Pentru a manipula semnale din program, avem următoarele instrumente (furnizate de "signal.h"):

`#include <signal.h>`

`sigset_t`
 $\Longrightarrow$ tipul "mulțime de semnale";

`int sigemptyset(sigset_t *set);`
 $\Longrightarrow$ setează `*set := {}`;
 returnează: 0=succes, -1=eșec;

`int sigfillset(sigset_t *set);`
 $\Longrightarrow$ setează `*set := {1, ..., NSIG-1}`;
 returnează: 0=succes, -1=eșec;

`int sigaddset(sigset_t *set, int signum);`
 $\Longrightarrow$ adaugă `*set := *set` $\cup$ `{signum}`;
 returnează: 0=succes, -1=eșec;

`int sigdelset(sigset_t *set, int signum);`
 $\Longrightarrow$ elimină `*set := *set` $\setminus$ `{signum}`;
 returnează: 0=succes, -1=eșec;

`int sigismember(const sigset_t *set, int signum);`
 $\Longrightarrow$ testeaza dacă `signum` aparține `*set`;
 returneaza: 0=nu aparține, 1=aparține, -1=eșec;

errno posibile: EINVAL.

Semnale

Putem bloca/debloca semnale (modificând linia a 2-a, a indicatorilor semnalelor blocate, din structura de gestiune a semnalelor primite de procesul curent) cu apelul:

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

$\Longrightarrow$ setează/consultă masca procesului de blocare a semnalelor;
`*set`=noua mască (dacă `set`=NULL, nu se schimbă masca);
`*oldset`=aici se recuperează vechea mască (dacă `oldset`=NULL, nu se mai recuperează);
`how`=poate fi indicat prin următoarele constante simbolice (furnizate de "signal.h"):
SIG_SETMASK $\Rightarrow$ noua mască devine `*set`
SIG_BLOCK $\Rightarrow$ noua mască devine `*oldset` $\cup$ `*set`
SIG_UNBLOCK $\Rightarrow$ noua mască devine `*oldset` $\setminus$ `*set`
returnează: 0=succes, -1=esec;
nu pot fi blocate SIGKILL și SIGSTOP (încercarea eșuează, apelul returnează -1 și setează errno=EINVAL);

Semnale

Putem afla semnalele în pending la procesul curent (consultând prima linie, a indicatorilor semnalelor în pending, din structura de gestiune a semnalelor primite) cu apelul:

```
#include <signal.h>
int sigpending(sigset_t *set);
```

$\Longrightarrow$ pune în `*set` mulțimea semnalelor aflate în pending în acel moment; returnează: 0=succes, -1=esec;

Semnale

Pentru majoritatea semnalelor se pot instala handlere utilizator.

Pentru a putea fi folosită ca handler, o funcție trebuie să aibe un singur parametru de tip "int" și să returneze "void", de exemplu:

```
void h(int n){...}
```

Când procesul va primi semnalul asociat, el va fi transmis ca parametru acestei funcții. Astfel, dacă aceeași funcție este instalată ca handler pentru mai multe semnale, ea va ști la fiecare apel semnalul pentru care a fost apelată (și poate fi scrisă să facă ceva diferit în fiecare caz).

Putem instala un handler utilizator pentru un semnal cu apelurile "signal()" și "sigaction()":

Semnale

Apelul "signal()" este mai simplu:

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

$\Longrightarrow$ se instalează funcția specificată de "handler" pentru semnalul "signum";

"handler" poate specifica o funcție utilizator (cu signatura menționată), poate fi SIG_IGN (handler-ul de ignorare) sau SIG_DFL (handler-ul implicit al semnalului "signum"); "signum" nu poate fi SIGKILL sau SIGSTOP;

"signal()" returnează adresa vechiului handler folosit pentru semnalul "signum", sau SIG_ERR în caz de eroare;

în unele implementari UNIX/Linux, handler-ele funcții utilizator sunt instalate a.i. blochează temporar pe perioada apelului semnalul pentru care au fost instalate sau reinstalează pentru semnalul respectiv, chiar de la începutul apelului, handlerul SIG_DFL; în acest caz, dacă dorim să instalam handler-ul permanent, punem la începutul său un apel de reinstalare a sa pentru același semnal: `void h(int n){signal(n,h); ...}`
(dacă în plus pe perioada apelului "n" este blocat, nu există riscul ca un nou "n" să întrerupă "h" între "{" și "`signal(n,h);`" iar pentru el să se execute SIG_DFL); dacă nu vrem să ne bazăm pe aceste comportamente implicite, folosim "sigaction()".

Semnale

Apelul "sigaction()" permite specificarea unor setări mai fine, cu ajutorul unei structuri "sigaction":

```
#include <signal.h>
int sigaction(int signum,
              const struct sigaction *act,
              struct sigaction *oldact);
```

$\Longrightarrow$ dacă "act" este $\neq$NULL, se instalează acțiunea descrisă de structura "*act" pentru semnalul "signum";
dacă "oldact" este $\neq$NULL, este recuperată vechea acțiune în "*oldact";
"signum" nu poate fi SIGKILL sau SIGSTOP;

Semnale

```
#include <signal.h>
struct sigaction{
  void (*sa_handler)(int);
   ...
  sigset_t sa_mask;
  int sa_flags;
   ...
};
```

acțiunea descrisă de o structură "sigaction" conține următoarele informații:
`sa_handler` = pointer la funcția handler;
`sa_mask` = o mască de semnale adăugate la masca de semnale blocate a procesului, pe perioada executării handlerului;
`sa_flags` = 0 sau disjuncție pe biți de mai multe opțiuni;

Semnale

```
#include <signal.h>
struct sigaction{
  void (*sa_handler)(int);
   ...
  sigset_t sa_mask;
  int sa_flags;
   ...
};
```

câteva dintre opțiunile utilizabile la `sa_flags`:

SA_ONESHOT, SA_RESETHAND = după o singură execuție a noului handler se va reinstala automat handlerul implicit; reinstalarea va avea loc chiar de la începutul executării noului handler (a se vedea funcția "signal");

SA_NOMASK, SA_NODEFER = semnalul căruia i s-a asociat handlerul nu va mai fi blocat pe perioada executării sale (deci handlerul se va putea întrerupe de către el însuși);

SA_RESTART = anumite apeluri sistem vor fi restartabile după primirea semnalului; de exemplu, dacă procesul doarme într-un "wait()", la primirea semnalului va executa handler-ul apoi va continua să doarmă în acel "wait()" (altfel, după executarea handler-ului nu se reia "wait()" ci se trece mai departe);

Semnale

Dacă am asociat unui semnal un handler utilizator ce modifică niște variabile globale care influiențează cursul ulterior al execuției programului, este foarte important să știm de câte ori și în ce momente ale execuției va veni semnalul respectiv (de asta depinde treseul execuției).

Totuși, la momentul scrierii programului, nu putem anticipa ce semnale va primi el pe parcursul execuției și în ce momente - asta se decide abia la run-time.

De aceea, în general asemenea programe sunt greu de scris, deoarece trebuie să avem în vedere toate cazurile posibile.

Semnale

Variabilele globale modificate de handlerele utilizator asociate semnalelor e bine să fie declarate cu calificatorul "volatile" - aceasta este o indicație către compilator că variabila ar putea fi modificată oricând pe altă cale decât cursul normal al execuției descris în program (eventual ar putea fi modificată din afara acestui program).

Compilatorul interpretează această indicație prin aceea că nu optimizează accesarea variabilei respective în memorie; mai exact orice accesare / modificare a variabilei va fi făcută cu citirea și scrierea valorii ei curente în memorie (valoarea curentă nu va fi menținută mai mult timp în regiștri).

Semnale

De exemplu secvența:

```
int a;
int x=1,y;
  ...
a=x;
a=a+10;
y=a;
```

va fi tradusă la compilarea cu "gcc -O3 -S" în :

```
movl    x, %eax
addl    $10, %eax
movl    %eax, a
movl    %eax, y
```

deci, dacă vine un semnal între "a=x" și "a=a+10" iar handler-ul dă lui "a" valoarea 100, în "y" va ajunge tot 11 (în loc de 110), deoarece valoarea curentă a lui "a" este menținută în registrul "%eax" și nu este recitita din memorie;

Semnale

În schimb secvența:

```
volatile int a;
int x=1,y;
  ...
a=x;
a=a+10;
y=a;
```

va fi tradusă la compilarea cu "gcc -O3 -S" în :

```
movl    x, %eax
movl    %eax, a
movl    a, %eax
addl    $10, %eax
movl    %eax, a
movl    a, %eax
movl    %eax, y
```

deci, un semnal care vine între aceste instrucțiuni sursă are șanse mai mari să prindă valoarea curentă a lui "a" în memorie (nu în registru).

Semnale

Alte apeluri utile:

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds);
```

$\Longrightarrow$ procesul curent doarme până trec "`seconds`" secunde sau primește un semnal neignorat;

returnează 0 dacă a trecut timpul cerut sau numarul de secunde neconsumate altfel;

```
#include <unistd.h>
void usleep(unsigned long usec);
```

$\Longrightarrow$ procesul curent doarme până trec (cel puțin) "`usec`" microsecunde (durata poate crește în funcție de încărcarea sistemului sau alți factori) sau primește un semnal neignorat;

Semnale

Putem adormi procesul în așteptarea unui semnal cu apelurile:

```
#include <unistd.h>
int pause(void);
```

$\Longrightarrow$ procesul adoarme până primește un semnal neignorat - deci neblocat și cu alt handler decat SIG_IGN (excepție: dacă semnalul este SIGCONT cu handlerul SIG_DFL, procesul rămâne adormit);

returnează după ce handler-ul semnalului a facut return - anume "pause()" returnează -1 și setează errno = EINTR;

```
#include <signal.h>
int sigsuspend(const sigset_t *mask);
```

$\Longrightarrow$ instalează temporar (până la ieșirea din apel) "`*mask`" ca mască de blocare a semnalelor și adoarme procesul până la primirea unui semnal neblocat de "`*mask`"; acțiunea este ATOMICĂ (cele două efecte sunt realizate printr-o singură operație, deci nu există riscul să se trateze un semnal neblocat de "`*mask`" între momentul instalării acestei măști și adormirea procesului);

la primirea unui semnal neblocat de "`*mask`" comentariile și return-ul sunt ca la "pause";

Semnale

Putem programa primirea de către procesul curent a unui semnal SIGALRM după un anumit interval de timp cu apelul:

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

$\Longrightarrow$ programează trimiterea unui semnal SIGALRM către procesul apelant după trecerea a "`seconds`" secunde (apelul nu blocheaza procesul în așteptarea semnalului, ci procesul își continuă execuția iar după trecerea intervalului de timp primește semnalul);

dacă anterior a mai fost efectuat un apel "alarm()", el este anulat (numărătoarea descrescatoare a secundelor se reia de la "`seconds`");

dacă apelam "alarm()" cu parametrul "`seconds`"=0 doar se anulează orice apel anterior;

returnează numărul de secunde rămase de numărat în urma apelului anterior, sau 0 dacă nu a existat un apel anterior.

Obs: apelul "sleep()" poate fi implementat folosind SIGALRM; de aceea mixarea apelurilor "alarm()" și "sleep()" este contraindicată.

Semnale

Câteva semnale importante și semnificațiile lor:

SIGHUP $= 1$

Când procesul lider al unei sesiuni se termină, acest semnal este trimis automat tuturor proceselor din sesiunea respectivă.

Handler implicit: terminarea.

Astfel, când ne delogam (deci login shell-ul, care era lider de sesiune se termină), toate procesele lansate prin comenzi shell se termină automat. Putem însă lansa un program "prog" cu comanda "nohup prog" și atunci el va fi "vaccinat" la semnalul SIGHUP iar cand îl va primi nu se va termina - astfel putem lansa programe care să ruleze și dupa ce ne-am delogat (de exemplu de pe-o zi pe alta). Ieșirea standard și ieșirea standard de eroare ale unui program lansat cu "nohup" sunt redirectate automat spre un fișier "nohup.out" aflat în directorul curent (sau în directorul home, dacă cel curent nu oferă drept de scriere), cu excepția cazului când s-a indicat explicit o altă redirectare (de exemplu dăm: "nohup prog $>$ f"), caz în care se respectă această redirectare.

Semnale

Câteva semnale importante și semnificațiile lor:

SIGINT $= 2$, SIGQUIT $= 3$

Când tastăm de la un terminal Ctrl-c, respectiv Ctrl-\ (de fapt caracterele logice `intr` și `quit`), primul, respectiv al doilea semnal este trimis către toate procesele aflate în foreground la terminalul respectiv.

Handler implicit: terminarea (în cazul lui SIGQUIT se face și un fișier "core" cu imaginea memoriei).

Semnale

Câteva semnale importante și semnificațiile lor:

SIGKILL $= 9$

Este neblocabil și nu îi putem asocia alt handler decât cel implicit.

Handler implicit: terminarea.

Acest semnal este folosit de regulă pentru a termina explicit un proces dat.

Semnale

Câteva semnale importante și semnificațiile lor:

SIGUSR1 = 10, SIGUSR2 =12

Handler implicit: terminarea.

De regulă sunt folosite de programatori pentru scopuri particulare.

Semnale

Câteva semnale importante și semnificațiile lor:

SIGSEGV $= 11$

Când un proces încearcă să acceseze cu instrucțiuni din programul utilizator (fiind deci în user mode) o zonă din afara spațiului de adrese al procesului primește semnalul SIGSEGV.

Handler implicit: terminarea.

Se poate instala un handler utilizator sau chiar SIG_IGN și atunci procesul continuă, doar instrucțiunea care a încercat accesul ilegal nu este efectuată.

Semnale

Câteva semnale importante și semnificațiile lor:

SIGALRM $= 14$

Handler implicit: terminarea.

De regulă este folosit de către un program pentru a-și planifica un comportament anume după un anumit interval de timp, indiferent de momentul unde se află cu execuția (cu ajutorul unui handler utilizator asociat semnalului).

Semnale

Câteva semnale importante și semnificațiile lor:

SIGTERM $= 15$

Handler implicit: terminare

Semnale

Câteva semnale importante și semnificațiile lor:

SIGCHLD $= 17$

Cand un proces se termină, părintele său primește un semnal SIGCHLD.

Handler implicit: ignorare

Semnale

Câteva semnale importante și semnificațiile lor:

SIGSTOP = 19, SIGCONT = 18

Semnale folosite la suspendarea / reluarea explicită a unor procese (mecanisme de job control).

Handler implicit: suspendare, respectiv reluare.

SIGSTOP nu poate fi blocat și nu i se poate asocia alt handler decât cel implicit; lui SIGCONT pe anumite sisteme nu-i putem asocia alt handler decât cel implicit.

Semnale

Din linia de comandă shell putem transmite semnale către procese cu comanda:

```
kill -semnal proces
```

unde "`semnal`" este codul numeric sau sfârșitul mnemonicului unui semnal (partea fără "SIG"), iar "`proces`" este PID-ul procesului destinatar; în general, comanda reușește dacă proprietarul său real coincide cu cel al procesului destinatar sau este root; mai sunt si alte detalii.

Cu comanda

```
kill -l
```

se afisază o listă cu toate semnalele implementate în sistemul curent.

Semnale

Exemplu:

Un proces generează un copil și îi trimite 2000 de semnale SIGUSR1; părintele numără câte a trimis, copilul numără câte a primit; fiecare, când a ajuns la 2000 afișază un mesaj și se termină; pentru a nu se pierde semnale, părintele nu trimite un nou semnal până nu primește confirmarea primirii celui anterior, sub forma tot a unui semnal SIGUSR1.

Variantă INCORECTĂ:

```
#include<sys/types.h>
#include<unistd.h>
#include<signal.h>
#include <sys/wait.h>
#include<stdio.h>
#include<stdlib.h>
volatile int nr; char *nume;
void f(int n){signal(n,f);}
void g(int n){printf("%s: %d\n",nume,nr); exit(0);}
int main(){
  pid_t p;
  signal(SIGUSR1,f); signal(SIGINT,g); nr=0;
  if(p=fork()){
    nume="parinte";
    while(nr<2000){kill(p,SIGUSR1); ++nr; pause();}
    wait(NULL);
    printf("final parinte\n");
  }else{
    nume="copil"; p=getppid();
    while(nr<2000){pause(); ++nr; kill(p,SIGUSR1);}
    printf("final copil\n");
  }
  return 0;
}
```

Semnale

Comentarii:
- la fiecare iterație (în total 2000), părintele trimite un semnal, îl numără, apoi adoarme în "pause()" așteptand confirmarea;
  când aceasta vine, iese din "pause()" și executa handler-ul "f()" care nu face nimic deosebit (doar se reinstalează - am presupus că în sistemul curent "signal()" instalează handlere doar pentru o singură folosire) - rolul lui "f()" este doar de a întrerupe "pause()";
  asemănător copilul, la fiecare iterație (în total 2000), așteaptă în "pause()" un semnal, când acesta vine iese din "pause()" (și execută "f()"), apoi îl numără și trimite părintelui confirmarea;

- înainte de afișarea mesajului final, părintele așteaptă terminarea copilului ("wait(NULL)"); altfel, părintele s-ar putea termina primul și atunci fiul s-ar muta în background și n-ar mai putea afișa mesajul său final (decât dacă terminalul este setat "stty -tostop");

- în caz de interblocare, ambele procese pot fi terminate cu un Ctrl-c de la terminal - ele, fiind în forground, vor primi semnalul SIGINT, iar handler-ul asociat va afișa semnalele numărate pâna la acel moment și va face "exit()";

Semnale

Comentarii:
- la rulare se constată că uneori se transmit / receptionează toate cele 2000 semnale iar procesele se termină normal, alteori ele se interblochează fără a le trimite / receptiona pe toate;

- un scenariu care duce la interblocare: părintele trimite un semnal, și înainte de a ajunge la "pause()" primește confirmarea; atunci execută "f()" (care nu face nimic deosebit), iar când ajunge la "pause()" va aștepta confirmarea care venise deja; astfel, părintele nu va trece la iterația următoare să mai trimită un semnal, iar copilul nu mai trimite o confirmare până nu primește un semnal; astfel ambele procese ajung să doarmă într-un "pause()";

- într-o altă varintă a programului, copilul ar putea trimite confirmarea chiar din handler-ul care-l trezește din "pause()"; atunci ar apărea în plus și riscul numărării greșite a semnalelor, în urma efectuării mai multor schimburi de semnale între un "pause()" și un "++nr" din aceeași iterație a copilului;

- în plus, dacă handler-ele s-ar fi instalat în cele două procese după "fork()", exista riscul ca părintele să înceapă trimiterea semnalelor înainte ca copilul să-și instaleze handler-ul (și ar fi folosit handler-ul implicit al lui SIGUSR1, care termina procesul);

- cauza interblocării sau numărării greșite este că semnalele nu sunt tratate întotdeauna în locul unde sunt așteptate.

Variantă CORECTĂ:

```
#include<sys/types.h>
#include<unistd.h>
#include<signal.h>
#include <sys/wait.h>
#include<stdio.h>
#include<stdlib.h>
volatile int nr; char *nume;
void f(int n){signal(n,f);}
void g(int n){printf("%s: %d\n",nume,nr); exit(0);}
int main(){
  pid_t p; sigset_t ms;
  signal(SIGUSR1,f); signal(SIGINT,g);
  sigemptyset(&ms); sigaddset(&ms,SIGUSR1); sigprocmask(SIG_SETMASK,&ms,NULL);
  sigemptyset(&ms); nr=0;
  if(p=fork()){nume="parinte";
               while(nr<2000){kill(p,SIGUSR1); ++nr; sigsuspend(&ms);}
               wait(NULL);
               printf("final parinte\n");
  }else{nume="copil"; p=getppid();
        while(nr<2000){sigsuspend(&ms); ++nr; kill(p,SIGUSR1);}
        printf("final copil\n");
  }
  return 0;
}
```

Semnale

Comentarii:

în afara lui "sigsuspend()" SIGUSR1 este blocat (deci dacă vine rămâne în pending), iar în "sigsuspend()" este deblocat;
blocarea / deblocarea și intrarea în așteptare sunt efectuate de "sigsuspend()" în mod atomic, a.î. nu există riscul ca vreun SIGUSR1 să fie tratat între deblocare și intrarea în așteptare; în plus, logica programului face ca fiecare proces să nu poată trimite două semnale fără să primească un semnal între ele, astfel că destinatarul nu poate pierde un semnal pentru că avea deja unul netratat în pending.

Semnale

Exemplu: utilizare "sigaction()" și diverse fenomene legate de semnale:

```
#include<sys/types.h>
#include<unistd.h>
#include<signal.h>
#include<stdio.h>
void h(int n){printf("inceput %d\n",n); sleep(1); printf("sfarsit %d\n",n);}
int main(){
  pid_t p; int i; struct sigaction s;
  sigset_t ms;
  sigemptyset(&ms); for(i=1;i<=3;++i)sigaddset(&ms,i);
  sigprocmask(SIG_SETMASK,&ms,NULL);
  s.sa_handler=h; s.sa_flags=0;
  s.sa_mask=ms; sigaction(1,&s,NULL);
  sigdelset(&s.sa_mask,3); sigaction(2,&s,NULL); sigaction(3,&s,NULL);
  if(fork()){
    sleep(4); sigpending(&ms);
    for(i=1;i<NSIG;++i)if(sigismember(&ms,i)) printf("%d ",i); printf("\n");
    sigemptyset(&ms); sigprocmask(SIG_SETMASK,&ms,NULL);
  }else{p=getppid();
        kill(p,1); kill(p,1); kill(p,1); sleep(1);
        kill(p,2); sleep(1); kill (p,3);
  }
  return 0;
}
```

Semnale

Comentarii:

- la rulare afișază:

```
1 2 3
inceput 1
sfarsit 1
inceput 2
inceput 3
sfarsit 3
sfarsit 2
```

- inițial părintele și-a blocat semnalele 1,2,3, apoi a adormit 4 secunde, timp în care copilul i-a trimis mai multe din aceste semnale; părintele, avându-le blocate, nu s-a trezit din "sleep()" din cauza lor (a dormit toate cele 4 secunde); mai mult, ultimele două 1 trimise de copil s-au pierdut, deoarece părintele avea deja un 1 in pending;

- după cele 4 secunde, părintele și-a consultat semnalele în pending și le-a afișat, apoi a deblocat semnalele 1,2,3; atunci, semnalele 1,2,3 din pending au început să fie tratate (să li se apeleze handler-ul "f()");

Semnale

Comentarii:

- la rulare afișază:

```
1 2 3
inceput 1
sfarsit 1
inceput 2
inceput 3
sfarsit 3
sfarsit 2
```

- semnalele din pending au fost tratate în ordine crescatoare, numai că odată lansat "f(1)" semnalele 2,3 erau blocate (pentru 1 "f()" a fost instalat a.î. pe perioada execuției sale să fie blocate suplimentar semnalele 1,2,3), așa că "f(1)" s-a executat neîntrerupt; odată lansat "f(2)", deoarece era un 3 în pending, nu era blocat (pentru 2 "f()" a fost instalat a.î. pe perioada executării sale să fie blocate suplimentar doar 1,2) iar execuția lui "f()" se face în user mode, "f(2)" s-a întrerupt ca să se execute "f(3)", iar după revenire s-a continuat.

Semnale

Exemplu: folosire "alarm()":

```
#include<signal.h>
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>

void h(int n){printf("Timpul a expirat\n"); exit(1);}

int main(){int c;
  signal(SIGALRM,h);
  printf("Asa e ? (d/n): "); alarm(10);
  scanf("%c",&c);
  alarm(0);
  printf("Multumesc pentru raspuns !\n");
  return 0;
}
```

Comentariu: după afișarea întrebării "`Asa e ? (d/n): `", dacă utilizatorul răspunde în < 10 secunde, se afișază "`Multumesc pentru raspuns !\n`", alarma se anulează iar procesul se termină; dacă nu raspunde în timp util, procesul primește alarma, handler-ul afișază "`Timpul a expirat\n`" iar procesul se termină (nu mai este afișat "`Multumesc...`").

Semnale

Exemplu: folosirea "alarm()" și "sigsetjmp()"/"siglongjmp()" la implementarea manuală (făcută explicit de programator) a thread-urilor în spațiul utilizator.

Este urmat tiparul unui exemplu similar din cursul 5, dar se folosesc "sigsetjmp()/siglongjmp()" în loc de "setjmp()/ longjmp()" (se pot folosi și ultimele, nu este esențial), iar thread-urile nu mai cedează voluntar procesorul prin apeluri de tip "thread_yield()" ci sunt comutate automat la anumite intervale de timp, folosind semnale SIGALRM și apeluri "alarm()".

Semnale

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<stdarg.h>
#include<setjmp.h>
#include<signal.h>
#include<unistd.h>

/* executiv */

#define MAXTHREADS 100

struct thread_table_entry{
  sigjmp_buf j[2]; int s;
  void (*f)();
} thread_table[MAXTHREADS];
int nthreads, tcurrent;
sigjmp_buf jscheduler, jmain;

unsigned int thread_set[MAXTHREADS], nthreadset;
```

Semnale

```
int thread_create(void (*pf)()){
  int i,k;
  if(nthreadset==nthreads)return -1;
  for(i=0;i<nthreads;++i){
      for(k=0;k<nthreadset;++k)if(thread_set[k]==i)break;
      if(k==nthreadset)break;
  }
  thread_table[i].f=pf;
  thread_table[i].s=0;
  thread_set[nthreadset]=i; ++nthreadset;
  return i;
}

void thread_terminate(){
  int k;
  if(nthreadset==0)return;
  alarm(0);
  for(k=0;k<nthreadset;++k)if(thread_set[k]==tcurrent)break;
  --nthreadset; thread_set[k]=thread_set[nthreadset];
  siglongjmp(jscheduler,1);
}
```

Semnale

```
void schedule_threads(){
  if(nthreadset==0){tcurrent=-1; siglongjmp(jmain,1);}
  tcurrent=thread_set[rand()%nthreadset];
  alarm(1);
  if(thread_table[tcurrent].s==0)
    {thread_table[tcurrent].s=1; siglongjmp(thread_table[tcurrent].j[0],1);}
  else siglongjmp(thread_table[tcurrent].j[1],1);
}

void thread_handler(int n){
  signal(n,thread_handler);
  if(!sigsetjmp(thread_table[tcurrent].j[1], 1)) siglongjmp(jscheduler,1);
}
```

Semnale

```
void initialize_threads(unsigned int nt, unsigned int dim){
 static int dimaux=0;
 unsigned char buf[1024];
 if(dimaux==0){
   if(nt>MAXTHREADS || dim==0)exit(1);
   srand(time(NULL));
   nthreads=nt; dimaux=dim;
   nthreadset=0; tcurrent=-1;
   signal(SIGALRM,thread_handler);
 }
 if(dim>0)initialize_threads(nt,dim-1);
 else if(nt>0)initialize_threads(nt-1,dimaux);
 if(dim==0)
  if(nt==nthreads){dimaux=0; if(sigsetjmp(jscheduler, 1)) schedule_threads(); }
  else if(sigsetjmp(thread_table[nt].j[0], 1)){
            (*thread_table[nt].f)();
          thread_terminate();
        }
}

void start_threads(){if(!sigsetjmp(jmain, 1))siglongjmp(jscheduler,1);}
```

Semnale

```
/* aplicatie */

int vglobal;
void f2(){
  int vf2=20,i;
  for(i=0;i<4;++i){
     sleep(1);
     printf("f2: vf2=%d, vglobal=%d\n",++vf2,++vglobal);
  }
}
void f1(){
  int vf1=10; int j;
  thread_create(f2);
  for(j=0;j<2;++j){
     sleep(1);
     printf("f1: vf1=%d, vglobal=%d\n",++vf1,++vglobal);
  }
}
int main(){
  initialize_threads(2,1);
  vglobal=0;
  thread_create(f1); start_threads();
  return 0;
}
```

Semnale

La rulare programul poate afișa de exemplu:

```
f1: vf1=11, vglobal=1
f2: vf2=21, vglobal=2
f2: vf2=22, vglobal=3
f2: vf2=23, vglobal=4
f1: vf1=12, vglobal=5
f2: vf2=24, vglobal=6
```

Pentru alte comentarii, a se vedea cursul 5.

Exemplu: gestionarea erorilor fatale (involuntare) (a se vedea cursul 5):

```
#include<setjmp.h>
#include<signal.h>
#include<stdio.h>
sigjmp_buf stare;
void h(int n){siglongjmp(stare,1);}
int main(){
  int n,*p=NULL;
  signal(SIGSEGV,h);
  if(!sigsetjmp(stare, 1))
    n=*p;
  else
    fprintf(stderr,"Eroare.\n");
  return 0;
}
```

Comentariu: la prima întâlnire a apelului "sigsetjmp(stare, 1)" acesta salvează în "stare" contextul de dinaintea operației riscante și returnează 0; atunci se intră pe ramura operației riscante "n=*p;", aceasta generează o eroare fatală - accesarea prin instrucțiuni utilizator a unei zone din afara spațiului de adrese al procesului - procesul primește semnalul SIGSEGV, iar handler-ul asociat restaurează contextul salvat în "stare"; atunci se întâlnește pentru a doua oară apelul "sigsetjmp(stare, 1)", acum acesta returnează 1 (al doilea parametru al lui "siglongjmp()") și astfel se intră pe ramura "else".

Cuprins

IPC

IPC (**Inter Process Communication**) desemnează o categorie de entități prin intermediul cărora se poate realiza comunicarea între procese; ele pot fi de trei tipuri:
- segment de memorie partajată (shared memory segment);
- vector de semafoare (semaphore set);
- coadă de mesaje (message queue).

Orice IPC are un **tip** (din cele 3), o **cheie externă** (identificator numeric la nivelul instanței UNIX/Linux în care se află) și niște **identificatori numerici interni** (la nivelul diverselor procese care îl folosesc). Identificatorii numerici externi sunt de tip "key_t", definit în "sys/types.h".

Deși IPC-urile nu sunt tratate ca fișiere, ele au multe atribute asemanatoare: proprietar, drepturi de acces, momentul ultimului acces, momentul ultimei modificări, etc.

IPC

IPC-urile existente în sistem se pot vizualiza cu comanda shell "`ipcs`" (fără opțiuni le afișază pe toate, cu opțiunile "`-m`" / "`-s`" / "`-q`" afișază numai segmentele de memorie partajată / vectorii de semafoare / cozile de mesaje) și se pot șterge cu comanda shell "`ipcrm`" (în forma "`ipcrm -M key`" / "`ipcrm -S key`" / "`ipcrm -Q key`" șterge segmentul de memorie partajată / vectorul de semafoare / coada de mesaje cu cheia "`key`"); IPC-rile pot fi șterse doar de proprietarul lor sau "root".

IPC

Înainte de a crea un IPC, trebuie să construim o cheie pentru el; pentru aceasta folosim apelul:

```
# include <sys/types.h>
# include <sys/ipc.h>
key_t ftok(const char *pathname, int proj_id);
```

$\Longrightarrow$ calculează și returnează o cheie plecând de la numărul de disc și numărul de i-nod al fișierului cu numele și calea specificate de "`pathname`" (trebuie să fie un fișier existent și accesibil) și de la octetul low al întregului "`proj_id`" (care trebuie să fie nenul); nu contează conținutul fișierului; rezultatul va fi același pentru toate "`pathname`"-urile care se referă la același fișier, dacă folosim același "`proj_id`"; în caz de eroare returnează -1 iar errno este setat ca la apelul "stat()" (a se vedea cursul despre gestiunea fișierelor).

Cuprins

IPC - segmente de memorie partajată

Un proces poate crea și/sau poate obține un identificator intern propriu pentru un segment de memorie partajată cu apelul:

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, int size, int shmflg);
```

$\Longrightarrow$ returnează un identificator numeric al segmentului de memorie partajată cu cheia "`key`", care va fi intern la nivelul procesului apelant;

"`key`" poate fi IPC_PRIVATE sau o cheie;

"`size`" este o dimensiune în bytes;

"`shmflg`" poate fi 0 sau o disjuncție pe biti de constantele simbolice IPC_CREAT, IPC_EXCL și constantele simbolice ce descriu drepturile fișierelor create cu "open()" pentru proprietar, grup, alții (de ex. pentru drepturi de read/write pentru proprietar putem adăuga S_IRWXU) - a se vedea cursul despre gestiunea fișierelor;

IPC - segmente de memorie partajată

Un proces poate crea și/sau poate obține un identificator intern propriu pentru un segment de memorie partajată cu apelul:

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, int size, int shmflg);
```

$\Longrightarrow$

dacă "key" este IPC_PRIVATE, sau dacă "key" nu este IPC_PRIVATE dar nu există segmente cu cheia "key" și în "shmflg" apare IPC_CREAT, atunci se crează un segment nou (în cazul IPC_PRIVATE el va fi accesibil doar procesului care l-a creat și descendentilor lui); segmentul va avea ca proprietar proprietarul efectiv al procesului, dimensiunea "size" rotunjită la un multiplu al valorii PAGE_SIZE și drepturile date de ultimii 9 biți semnificativi ai constantelor referitoare la drepturi prezente în "shmflg"; "size" trebuie sa fie în intervalul [SHMMIN, SHMMAX]; apelul returnează un identificator intern pentru acest segment;

IPC - segmente de memorie partajată

Un proces poate crea și/sau poate obține un identificator intern propriu pentru un segment de memorie partajată cu apelul:

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, int size, int shmflg);
```

$\Longrightarrow$

dacă există un segment cu cheia "`key`" iar în "`shmflg`" apare atât IPC_CREAT cât și IPC_EXCL, apelul eșueaza (cu errno = EEXIST);

dacă există un segment cu cheia "`key`" și nu am folosit IPC_CREAT, apelul verifică dacă procesul apelant are drepturile necesare pentru a-l accesa și dacă "`size`" este $\leq$ dimensiunea cu care a fost creat segmentul; în caz afirmativ apelul returnează un identificator intern pentru acest segment;

IPC - segmente de memorie partajată

Un proces poate crea și/sau poate obține un identificator intern propriu pentru un segment de memorie partajată cu apelul:

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, int size, int shmflg);
```

$\Longrightarrow$

la eșec, apelul returnează -1; errno posibile: EACCES, EEXIST, EINVAL, ENFILE, ENOENT, ENOMEM, ENOSPC;

IPC - segmente de memorie partajată

Odata ce a obținut un identificator intern pentru un segment, procesul îl poate atașa în spațiul său de adrese la o anumită adresă (după care îi poate accesa conținutul începând de la acea adresă cu instrucțiuni obișnuite) folosind apelul:

```
#include <sys/types.h>
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

$\Longrightarrow$ atașaza segmentul identificat de "`shmid`" la spațiul de adrese al procesului apelant; adresa de atașare este specificată de "`shmaddr`", după următoarele criterii:
- dacă "`shmaddr`" este NULL, sistemul alege o adresă (nefolosită); este varianta recomandabilă;
- dacă "`shmaddr`" nu este NULL iar SHM_RND este prezent în "`shmflg`", adresa de atașare este "`shmaddr`" rotunjită prin lipsă la cel mai apropiat multiplu de SHMLBA; daca SHM_RND este absent, "`shmaddr`" trebuie să fie o adresă aliniată la nivel de pagină, și la ea se va atașa segmentul;

"`shmflg`" poate fi 0 sau o disjuncție pe biți de constantele simbolice SHM_RND (cu sensul menționat mai sus), SHM_RDONLY (și atunci procesul apelant va putea accesa segmentul doar în citire - altfel îl poate accesa și în citire și în scriere);

IPC - segmente de memorie partajată

Odata ce a obținut un identificator intern pentru un segment, procesul îl poate atașa în spațiul său de adrese la o anumită adresă (după care îi poate accesa conținutul începând de la acea adresă cu instrucțiuni obișnuite) folosind apelul:

```
#include <sys/types.h>
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

$\Longrightarrow$

procesul trebuie să aibe drepturile necesare tipului de acces intenționat (citire sau citire și scriere);

la succes apelul "shmat()" returneaza adresa de atașare, la eșec returnează -1;

errno posibile: EACCES, EINVAL, ENOMEM.

IPC - segmente de memorie partajată

Procesul poate accesa segmentul atașat prin instrucțiuni obișnuite, folosind de exemplu pointeri, pâna îl detașază cu apelul:

```
#include <sys/types.h>
#include <sys/shm.h>
int shmdt(const void *shmaddr);
```

$\Longrightarrow$ detașază segmentul atașat spațiului de adrese al procesului apelant la adresa "shmaddr" (returnată în prealabil de "shmat()");

returnează 0 la succes și -1 la eșec;

errno posibile: EINVAL.

IPC - segmente de memorie partajată

Atributele unui segment se pot gestiona cu apelul:

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

$\Longrightarrow$ consultă/setează atributele segmentului cu identificatorul intern "`shmid`" (inclusiv poate marca segmentul pentru distrugere);

IPC - segmente de memorie partajată

Atributele unui segment se pot gestiona cu apelul:

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

$\Longrightarrow$

"buf" este adresa unei structuri de tip "shmid_ds", definit în "sys/shm.h" astfel:

```
struct shmid_ds {
  struct ipc_perm shm_perm;    /* Ownership and permissions */
  size_t          shm_segsz;   /* Size of segment (bytes) */
  time_t          shm_atime;   /* Last attach time */
  time_t          shm_dtime;   /* Last detach time */
  time_t          shm_ctime;   /* Last change time */
  pid_t           shm_cpid;    /* PID of creator */
  pid_t           shm_lpid;    /* PID of last shmat()/shmdt() */
  shmatt_t        shm_nattch;  /* No. of current attaches */
  ...
};
```

IPC - segmente de memorie partajată

Atributele unui segment se pot gestiona cu apelul:

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

$\Longrightarrow$

tipul structură "ipc_perm" este definit în "sys/ipc.h" astfel (am evidențiat cu /* setabil */ câmpurile setabile cu IPC_SET):

```
            struct ipc_perm {
                key_t key;            /* Key supplied to shmget() */
/* setabil */   uid_t uid;            /* Effective UID of owner */
/* setabil */   gid_t gid;            /* Effective GID of owner */
                uid_t cuid;           /* Effective UID of creator */
                gid_t cgid;           /* Effective GID of creator */
/* setabil */   unsigned short mode;  /* Permissions + SHM_DEST and
                                         SHM_LOCKED flags */
                unsigned short seq;   /* Sequence number */
    };
```

IPC - segmente de memorie partajată

Atributele unui segment se pot gestiona cu apelul:

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

$\Longrightarrow$

"cmd" poate avea valorile:

IPC_STAT = copiază atributele segmentului în structura pointată de "buf" (apelantul trebuie să aibe drept de citire pe segment);

IPC - segmente de memorie partajată

Atributele unui segment se pot gestiona cu apelul:

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

$\Longrightarrow$

"cmd" poate avea valorile:

IPC_SET = setează anumite atribute ale segmentului cu cele date prin structura pointată de "buf";

se pot modifica atributele: "shm_perm.uid", "shm_perm.gid", și ultimii 9 biți semnificativi ai lui "shm_perm.mode";

procesul apelant trebuie să fie privilegiat, sau proprietarul său efectiv trebuie să coincida cu proprietarul ("shm_perm.uid") sau creatorul ("shm_perm.cuid") segmentului;

IPC - segmente de memorie partajată

Atributele unui segment se pot gestiona cu apelul:

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

$\Longrightarrow$

"cmd" poate avea valorile:

IPC_RMID = marchează segmentul pentru distrugere (practic, un flag nestandard "SHM_DEST" al lui "shm_perm.mode" va fi setat);
procesul apelant trebuie să fie privilegiat sau proprietarul sau creatorul segmentului;

IPC - segmente de memorie partajată

Atributele unui segment se pot gestiona cu apelul:

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

$\Longrightarrow$

Observații:

1. apelurile "shmget()", "shmat()", "shmdt()", "shmctl()" actualizează automat anumite atribute ale segmentului la care se referă.
2. Un segment este distrus efectiv doar în momentul când sunt îndeplinite simultan două condiții: este marcat pentru distrugere și nici un proces nu mai este atașat la el (i.e. "`shm_nattch`" devine 0); apelurile "shmat()" / "shmdt()" influiențează "`shm_nattch`";
3. Un segment nemarcat pentru distrugere persistă în sistem chiar dacă nu mai sunt procese atașate la el și își păstreaza conținutul (poate fi regăsit de procese ce se atașază ulterior la el) - deci, dacă vrem ca segmentele să dispară, trebuie distruse explicit;
4. În Linux un proces se poate atașa la un segment chiar dacă este marcat pentru distrugere (în unele implementari UNIX nu se poate).
5. la "fork()" se moștenesc segmentele atașate, la "exec()" și "exit()" segmentele atasate se detașază;

IPC - segmente de memorie partajată

Atributele unui segment se pot gestiona cu apelul:

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

$\Longrightarrow$

Apelul "shmctl()" returnează 0 la succes și -1 la eșec;
errno posibile: EACCES, EFAULT, EIDRM, EINVAL, ENOMEM, EOVERFLOW, EPERM.

IPC - segmente de memorie partajată

Exemplu: problema producator-consumator cu un segment de memorie partajată și sincronizare prin semnale:

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<unistd.h>
#include<sys/stat.h>
#include<fcntl.h>
#include<signal.h>
#include<sys/wait.h>
#include<stdlib.h>
#include<stdio.h>
```

IPC - segm. de mem. partajată

```
#define N 10
pid_t p,c;
int shmid; int *buf;
sigset_t ms;
void finalizare(int);
void f(int sig){signal(sig,f);}
void g(int sig){if(p==getpid()){wait(NULL); finalizare(0);} else exit(0);}
void initializare(){
  if((shmid=shmget(IPC_PRIVATE,(N+1)*sizeof(int),S_IRWXU))==-1)
       {perror("shmget"); finalizare(1);}
  if((buf=(int*)shmat(shmid,NULL,0))==(int *)-1)
       {perror("shmat"); finalizare(2);}
  signal(SIGUSR1,f); signal(SIGINT,g);
  sigemptyset(&ms); sigaddset(&ms, SIGUSR1);
  sigprocmask(SIG_SETMASK,&ms,NULL);
  sigemptyset(&ms);
}
void finalizare(int n){
  switch(n){
    default:
              shmdt(buf);
    case 2:  shmctl(shmid,IPC_RMID,NULL);
    case 1:  ;
  }
  exit(n);
}
```

IPC - segm. de mem. partajată

```
int main(){
  int b,v;
  initializare();  p=getpid();
  b=v=0; buf[N]=0; /* buf[N]: count; buf[0] ... buf[N-1]: buffer */
  if(c=fork()){ /* parinte = producator */
    int item; item=0;
    while(1){
      ++item;                          /* item=produce_item() */
      if(buf[N]==N)sigsuspend(&ms);    /* if(count==N)sleep() */
      buf[b]=item; b=(b+1)%N;          /* insert_item(item) */
      ++buf[N];                        /* count=count+1 */
      if(buf[N]==1)kill(c,SIGUSR1);    /* if(count==1)wakeup(consumer) */
    }
  }else{       /* copil = consumator */
    int item;
    while(1){
      if(buf[N]==0)sigsuspend(&ms);    /* if(count==0)sleep() */
      item=buf[v]; v=(v+1)%N;          /* item=remove_item() */
      --buf[N];                        /* count=count-1 */
      if(buf[N]==N-1)kill(p,SIGUSR1);/* if(count==N-1)wakeup(producer) */
      printf("%d\n",item);             /* consume_item(item) */
    }
  }
  finalizare(0);
  return 0;
}
```

IPC - segmente de memorie partajată

Comentarii:

- Structura programului de mai sus urmează tiparul rezolvării date în secțiunea "Sleep și Wakeup", dar elimină condițiile de cursă apărute acolo, deoarece în afara apelurilor "sigsuspend()" semnalul SIGUSR1 este blocat și doar în "sigsuspend()" este neblocat - deci dacă vine vreun asemenea semnal, rămâne în pending până în locul unde este așteptat (adică în "sigsupend()"); în plus, logica programului face ca nici un proces să nu trimită un nou SIGUSR1 până nu primește un SIGUSR1, a.î. nu este posibilă pierderea de semnale (care ar putea apărea dacă s-ar primi un nou SIGUSR1 cât timp există unul în pending).

- Deoarece ambele procese efectuează cicluri infinite, ele trebuie terminate forțat - în acest scop se va tasta Ctrl-C și atunci ambele vor primi câte un semnal SIGINT (ambele sunt în foreground), iar handler-ul asociat le va termina; în plus, în procesul părinte, acest handler așteaptă terminarea copilului (dacă părintele se termină primul, copilul se mută în background și n-ar mai putea putea afișa ultimile informații pe terminal decât dacă terminalul este setat "stty -tostop") și dezalocă resusele partajate.

- Zona tampon partajată este organizată ca o coadă alocată într-un vector parcurs circular - indicii bază "b" (unde se introduce) și vârf "v" (de unde se extrage) cresc cu 1 modulo N.

Cuprins

IPC - vectori de semafoare

Asupra semafoarelor UNIX/Linux se pot efectua operațiile "up", "down" (desrise mai înainte) și o operație nouă, anume adormirea procesului pâna valoarea semaforului devine 0 (în urma unor operații "down" efectuate de alte procese).

IPC - vectori de semafoare

Un proces poate crea și/sau poate obține un identificator intern propriu pentru un vector de semafoare cu apelul:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t key, int nsems, int semflg);
```

$\Longrightarrow$ returnează un identificator numeric al vectorului de semafoare cu cheia "`key`", care va fi intern la nivelul procesului apelant;

parametrii "`key`" și "`semflg`" au aceleași valori și semnificații ca parametrii "`key`" și "`shmflg`" de la "shmget()";

"`nsems`" reprezintă nr. de semafoare create în vector (dacă se crează un vector - atunci acest nr. trebuie să fie $\leq$ valoarea SEMMSL) sau folosite din vector (dacă se acceseaza un vector existent - atunci acest nr. trebuie să fie $\leq$ nr. specificat la crearea vectorului); dacă se accesează un vector existent, acest nr. poate fi 0 (don't care);

la eșec, apelul returnează -1; errno posibile: EACCES, EEXIST, EINVAL, ENOENT, ENOMEM, ENOSPC.

IPC - vectori de semafoare

Asupra unui vector de semafoare (pentru care procesul a obținut în prealabil un identificator intern) se poate face un vector de operații, cu apelul:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

$\Longrightarrow$ aplică vectorului de semafoare identificat de "`semid`" vectorul de operații pointat de "`sops`" și având lungimea "`nsops`";

IPC - vectori de semafoare

Asupra unui vector de semafoare (pentru care procesul a obținut în prealabil un identificator intern) se poate face un vector de operații, cu apelul:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

$\Longrightarrow$

o operație asupra unui semafor este indicată printr-o structură "sembuf" definită astfel:

```
struct sembuf{
     unsigned short sem_num;
     short          sem_op;
     short          sem_flg;
}
```

membrii structurii "sembuf" au semnificațiile:

sem_num: indicele semaforului din vector căruia i se aplică operația;

IPC - vectori de semafoare

Asupra unui vector de semafoare (pentru care procesul a obținut în prealabil un identificator intern) se poate face un vector de operații, cu apelul:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

$\Longrightarrow$

o operație asupra unui semafor este indicată printr-o structură "sembuf" definită astfel:

```
struct sembuf{
     unsigned short sem_num;
     short          sem_op;
     short          sem_flg;
}
```

membrii structurii "sembuf" au semnificațiile:

sem_op: $> 0 \Rightarrow$ incrementare cu valoarea respectivă (în sensul "up");
$< 0 \Rightarrow$ decrementare cu valoarea respectivă (în sensul "down");
$= 0 \Rightarrow$ procesul adoarme pâna valoarea devine 0;

în primele două cazuri, procesul trebuie să aibe drept de modificare asupra vectorului de semafoare; în ultimul caz trebuie sa aibe drept de citire;

IPC - vectori de semafoare

Asupra unui vector de semafoare (pentru care procesul a obținut în prealabil un identificator intern) se poate face un vector de operații, cu apelul:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

$\Longrightarrow$

o operație asupra unui semafor este indicată printr-o structură "sembuf" definită astfel:

```
struct sembuf{
     unsigned short sem_num;
     short          sem_op;
     short          sem_flg;
}
```

membrii structurii "sembuf" au semnificațiile:

sem_flg: poate fi 0 sau disjuncție pe biți de constantele simbolice:

IPC_NOWAIT $\Rightarrow$ operația este neblocantă (în loc să adoarmă procesul, operațiile nu se efectuează, apelul returnează -1 și setează errno = EAGAIN); notăm că operațiile ce pot adormi procesul sunt sem_op < 0 și sem_op $= 0$;

SEM_UNDO $\Rightarrow$ operația va fi anulată la terminarea procesului care a efectuat-o;

IPC - vectori de semafoare

Asupra unui vector de semafoare (pentru care procesul a obținut în prealabil un identificator intern) se poate face un vector de operații, cu apelul:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

$\Longrightarrow$

apelul returnează 0 la succes și -1 la eșec; errno posibile: E2BIG, EACCES, EAGAIN, EFAULT, EFBIG, EIDRM, EINTR, EINVAL, ENOMEM, ERANGE.

Observații esențiale:

- mulțimea operațiilor din vectorul specificat se efetuează ATOMIC;
- dacă apelul reușește, se garantează că toate operațiile au reușit, iar dacă apelul eșuează, se garantează ca nici una din operații nu s-a efectuat;

IPC - vectori de semafoare

Atributele unui vector de semafoare se pot gestiona cu apelul:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, ...);
```

$\Longrightarrow$

"`semid`" este identificatorul intern al unui vector de semafoare;

"`semnum`" este indicele unui semafor din acest vector (numărând de la 0);

"`cmd`" este tipul operației efectuate;

în funcție de valoarea lui "`cmd`", poate exista și un al 4-lea parametru, pe care-l vom numi "`arg`";

IPC - vectori de semafoare

Atributele unui vector de semafoare se pot gestiona cu apelul:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, ...);
```

$\Longrightarrow$

"cmd" poate fi:
GETNCNT = apelul returnează nr. de procese ce asteaptă incrementarea semaforului de indice "semnum";
GETZCNT = apelul returnează nr. de procese ce așteapta ca valoarea semaforului de indice "semnum" să ajungă 0;
GETVAL = apelul returnează valoarea semaforului de indice "semnum";
GETALL = apelul furnizează valorile tuturor semafoarelor din vector în componentele vectorului pointat de "arg", care trebuie să fie de tip "unsigned short *"; parametrul "semnum" este ignorat;
GETPID = apelul returnează PID-ul ultimului proces care a efectuat o operație asupra semaforului de indice "semnum";

IPC - vectori de semafoare

Atributele unui vector de semafoare se pot gestiona cu apelul:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, ...);
```

$\Longrightarrow$

SETVAL = valoarea semaforului de indice "`semnum`" devine "`arg`", care trebuie să fie de tip "`int`"; se anulează setarile "`undo`" pentru semaforul modificat în toate procesele; returnează 0=succes, -1=eșec;

SETALL = valorile tuturor semafoarelor din vector sunt setate cu componentele vectorului pointat de "`arg`", care trebuie să fie de tip "`unsigned short *`"; parametrul "`semnum`" este ignorat; alte detalii sunt ca la SETVAL;

IPC - vectori de semafoare

Atributele unui vector de semafoare se pot gestiona cu apelul:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, ...);
```

$\Longrightarrow$

IPC_STAT = furnizează în structura pointată de "arg", care trebuie să fie de tip "struct semid_ds *", atributele vectorului de semafoare; parametrul "semnum" este ignorat;

IPC_SET = setează unele atribute ale vectorului de semafoare conform structurii pointate de "arg", care trebuie să fie de tip "struct semid_ds *"; se pot modifica doar membrii "sem_perm.uid", "sem_perm.gid" și ultimii 9 biți semnificativi ai lui "sem_perm.mode"; parametrul "semnum" este ignorat;

tipul structură "semid_ds" este definit în "sys/sem.h" astfel:

```
struct semid_ds {
    struct ipc_perm sem_perm;  /* Ownership and permissions */
    time_t          sem_otime; /* Last semop time */
    time_t          sem_ctime; /* Last change time */
    unsigned short  sem_nsems; /* No. of semaphores in set */
};
```

tipul structură "ipc_perm" este definit in "sys/ipc.h" și a fost descris mai sus, la apelul "shmctl()";

IPC - vectori de semafoare

Atributele unui vector de semafoare se pot gestiona cu apelul:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, ...);
```

$\Longrightarrow$

IPC_RMID = distruge imediat vectorul de semafoare, trezind toate procesele adormite într-un "semop()" la el (lor, "semop()" le va returna eroare și va seta errno = EIDRM); parametrul "semnum" este ignorat;

IPC - vectori de semafoare

Atributele unui vector de semafoare se pot gestiona cu apelul:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, ...);
```

$\Longrightarrow$

în cazurile GETNCNT, GETZCNT, GETVAL, GETALL, GETPID, IPC_STAT procesul trebuie să aibe drept de citire asupra vectorului de semafoare respectiv;
în cazurile SETVAL, SETALL procesul trebuie să aibe drept de modificare asupra vectorului de semafoare respectiv;
în cazurile IPC_SET, IPC_RMID procesul trebuie să fie privilegiat sau proprietarul său efectiv trebuie să coincida cu proprietarul sau creatorul vectorului de semafoare;

în caz de eșec apelul "semctl()" returnează -1, iar în caz de succes returnează ce am menționat deja când "cmd" este GETNCNT, GETPID, GETVAL sau GETZCNT, și 0 când "cmd" are alta valoare;

errno posibile: EACCES, EFAULT, EIDRM, EINVAL, EPERM, ERANGE;

În general, apelurile legate de vectorii de semafoare modifică automat anumite atribute ale acestora.

IPC - vectori de semafoare

Exemplu: problema producător-consumator cu un segment de memorie partajată și sincronizare prin semafoare:

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<sys/sem.h>
#include<unistd.h>
#include<sys/stat.h>
#include<sys/wait.h>
#include<fcntl.h>
#include<signal.h>
#include<stdlib.h>
#include<stdio.h>
```

IPC - vectori de semafoare

```
#define N 10
pid_t p;
int shmid, semid; int *buf;
void finalizare(int);
void g(int sig){if(p==getpid()){wait(NULL); finalizare(0);} else exit(0);}
void initializare(){
  if((shmid=shmget(IPC_PRIVATE,N*sizeof(int),S_IRWXU))==-1)
       {perror("shmget"); finalizare(1);}
  if((semid=semget(IPC_PRIVATE,3,S_IRWXU))==-1)
       {perror("semget"); finalizare(2);}
  if((buf=(int*)shmat(shmid,NULL,0))==(int *)-1)
       {perror("shmat"); finalizare(3);}
  signal(SIGINT,g);
}
void finalizare(int n){
  switch(n){
    default:
              shmdt(buf);
    case 3:  semctl(semid,0,IPC_RMID);
    case 2:  shmctl(shmid,IPC_RMID,NULL);
    case 1:  ;
  }
  exit(n);
}
```

IPC - vectori de semafoare

```
int main(){
  int b,v; struct sembuf sop;
  initializare();
  p=getpid(); sop.sem_flg=0; /* sem_num: 0=mutex, 1=empty, 2=full */
  sop.sem_num=0; sop.sem_op=1; semop(semid,&sop,1);   /* mutex:=1 */
  sop.sem_num=1; sop.sem_op=N; semop(semid,&sop,1);   /* empty:=N */
  b=v=0; /* buf[0] ... buf[N-1]: buffer */
  if(fork()){ /* parinte = producator */
    int item; item=0;
    while(1){
      ++item;                                          /* item=produce_item() */
      sop.sem_num=1; sop.sem_op=-1; semop(semid,&sop,1);  /* down(&empty) */
      sop.sem_num=0; sop.sem_op=-1; semop(semid,&sop,1);  /* down(&mutex) */
      buf[b]=item; b=(b+1)%N;                          /* insert_item(item) */
      sop.sem_num=0; sop.sem_op=1; semop(semid,&sop,1);     /* up(&mutex) */
      sop.sem_num=2; sop.sem_op=1; semop(semid,&sop,1);      /* up(&full) */
    }
  }
```

IPC - vectori de semafoare

```
  else{      /* copil = consumator */
    int item;
    while(1){
      sop.sem_num=2; sop.sem_op=-1; semop(semid,&sop,1);   /* down(&full) */
      sop.sem_num=0; sop.sem_op=-1; semop(semid,&sop,1);  /* down(&mutex) */
      item=buf[v]; v=(v+1)%N;                          /* item=remove_item() */
      sop.sem_num=0; sop.sem_op=1; semop(semid,&sop,1);     /* up(&mutex) */
      sop.sem_num=1; sop.sem_op=1; semop(semid,&sop,1);     /* up(&empty) */
      printf("%d\n",item);                             /* consume_item(item) */
    }
  }
  finalizare(0);
  return 0;
}
```

IPC - vectori de semafoare

Comentarii:

- programul urmează tiparul soluției date în secțiunea "Semafoare";

- în locul semafoarelor "mutex", "empty", "full" s-a folosit un vector de 3 semafoare identificat intern prin "`semid`"; în el, semafoarele de indice 0, 1, 2 corespund respectiv lui "mutex", "empty", "full";
  pentru a face operațiile de "down" și "up" cu oricare dintre acestea, ca și pentru inițializările lui "mutex" și "empty" cu 1, respectiv `N` s-a folosit o aceeași structură "`sop`" inițializată în diverse feluri și apeluri "`semop()`";

- zona tampon este implementată în continuare explicit folosind un segment de memorie partajată, dar n-a mai fost nevoie să reținem acolo și numărul elementelor (semafoarele se ocupă cu numărarea);

- alte comentarii sunt ca la soluția dată în secțiunea "IPC - segmente de memorie partajată".

Cuprins

IPC - cozi de mesaje

Mesajele se transmit/recepționează prin intermediul unor structuri definite de utilizator, al căror prim câmp este obligatoriu de tip "long" (înseamnă tipul mesajului) și are o valoare > 0; celelalte câmpuri reprezintă conținutul mesajului.
În cele ce urmează, un asemenea tip structură va fi denumit generic "tip mesaj".

IPC - cozi de mesaje

Un proces poate crea și/sau poate obține un identificator intern propriu pentru o coadă de mesaje cu apelul:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
```

$\Longrightarrow$ returnează un identificator numeric al cozii de mesaje cu cheia "`key`", care va fi intern la nivelul procesului apelant;

parametrii "`key`" și "`msgflg`" au aceleași valori și semnificații ca parametrii "`key`" și "`shmflg`" de la "shmget()";

la eșec, apelul returnează -1; errno posibile: EACCES, EEXIST, ENOENT, ENOMEM, ENOSPC.

IPC - cozi de mesaje

Inserarea unui mesaj într-o coadă se poate face cu apelul:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd(int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg);
```

$\Longrightarrow$

"`msqid`" este identificatorul intern al cozii;

"`msgp`" este adresa structurii de "tip mesaj" conținând mesajul inserat;

"`msgsz`" este lungimea informației mesajului (deci fără lungimea câmpului ce conține tipul mesajului); trebuie să fie ≥ 0;

"`msgflg`" poate fi 0 sau IPC_NOWAIT;

IPC - cozi de mesaje

Inserarea unui mesaj într-o coadă se poate face cu apelul:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd(int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg);
```

$\Longrightarrow$

apelul inserează în coada o copie a structurii pointate de "`msgp`";

dacă există suficient spațiu în coadă, apelul returnează imediat cu succes; capacitatea cozii este desemnată de atributul său "`msg_bytes`"; la creare, această capacitate are valoarea de MSGMNB bytes, dar se poate modifica cu apelul "msgctl()";

IPC - cozi de mesaje

Inserarea unui mesaj într-o coadă se poate face cu apelul:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd(int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg);
```

$\Longrightarrow$

dacă nu există suficient spațiu în coadă, atunci:

- dacă am folosit IPC_NOWAIT apelul returnează imediat cu eșec și setează errno = EAGAIN;
- dacă nu am folosit IPC_NOWAIT, apelul adoarme procesul pâna se intră într-una din următoarele situații:
  - apare spațiu disponibil în coadă;
  - coada este distrusă; atunci apelul returnează cu eșec și setează errno = EIDRM;
  - un semnal întrerupe apelul; atunci apelul returnează cu eșec și setează errno = EINTR; apelurile "msgsnd()" nu sunt niciodată restartate după ce au fost întrerupte de un semnal, chiar dacă handler-ul semnalului a fost instalat cu "SA_RESTART";

IPC - cozi de mesaje

Inserarea unui mesaj într-o coadă se poate face cu apelul:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd(int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg);
```

$\Longrightarrow$

procesul apelant trebuie să aibe drept de scriere asupra cozii;

apelul "msgsnd()" returnează 0 la succes și -1 la eșec;

errno posibile: EACCES, EAGAIN, EFAULT, EIDRM, EINTR, EINVAL, ENOMEM.

IPC - cozi de mesaje

Extragerea unui mesaj dintr-o coadă se poate face cu apelul:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
ssize_t  msgrcv(int msqid, struct msgbuf *msgp, size_t msgsz,
                long msgtyp, int msgflg);
```

$\Longrightarrow$

"`msqid`" este identificatorul intern al cozii;

"`msgp`" este adresa structurii de "tip mesaj" în care se va prelua mesajul extras;

"`msgsz`" este lungimea informației mesajului (deci fără lungimea câmpului ce conține tipul mesajului); trebuie să fie ≥ 0;

"`msgtyp`" este tipul mesajului ce se dorește a fi extras;

"`msgflg`" poate fi 0 sau disjuncție pe biți de constantele simbolice IPC_NOWAIT, MSG_NOERROR și MSG_EXCEPT;

IPC - cozi de mesaje

Extragerea unui mesaj dintr-o coadă se poate face cu apelul:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
ssize_t  msgrcv(int msqid, struct msgbuf *msgp, size_t msgsz,
                long msgtyp, int msgflg);
```

$\Longrightarrow$

apelul citește din coadă un mesaj de tipul specificat de "`msgtyp`" în structura pointată de "`msgp`", eliminând mesajul citit din coadă;

IPC - cozi de mesaje

Extragerea unui mesaj dintr-o coadă se poate face cu apelul:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
ssize_t  msgrcv(int msqid, struct msgbuf *msgp, size_t msgsz,
                long msgtyp, int msgflg);
```

$\Longrightarrow$

dacă "`msgsz`" este $<$ lungimea mesajului ales pentru citire atunci:
- dacă nu am folosit MSG_NOERROR, mesajul nu este eliminat din coadă iar apelul returnează cu eșec setând errno = E2BIG;
- dacă am folosit MSG_NOERROR, mesajul este citit trunchiat (iar partea trunchiata se pierde);

"`msgtyp`" poate fi:

$0 \Rightarrow$ se citește primul mesaj din coadă;

$> 0 \Rightarrow$ se citește primul mesaj de tip "`msgtyp`" (dacă nu am folosit MSG_EXCEPT) sau care nu este de tip "`msgtyp`" (dacă am folosit MSG_EXCEPT);

$< 0 \Rightarrow$ se citește primul mesaj cu cel mai mic tip mai mic sau egal cu modulul lui "`msgtyp`";

IPC - cozi de mesaje

Extragerea unui mesaj dintr-o coadă se poate face cu apelul:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
ssize_t  msgrcv(int msqid, struct msgbuf *msgp, size_t msgsz,
                long msgtyp, int msgflg);
```

$\Longrightarrow$

dacă în coadă nu există nici un mesaj de tipul dorit atunci:
- dacă am folosit IPC_NOWAIT, apelul returnează imediat cu eșec și setează errno = ENOMSG;
- dacă nu am folosit IPC_NOWAIT, apelul adoarme procesul până se intră într-una din următoarele situații:

• un mesaj de tipul dorit este inserat în coadă;

• coada este distrusă; atunci apelul returnează cu eșec și setează errno = EIDRM;

• un semnal întrerupe apelul; atunci apelul returnează cu eșec și setează errno = EINTR; apelurile "msgrcv()" nu sunt niciodată restartate după ce au fost întrerupte de un semnal, chiar dacă handler-ul semnalului a fost instalat cu "SA_RESTART";

IPC - cozi de mesaje

Extragerea unui mesaj dintr-o coadă se poate face cu apelul:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
ssize_t  msgrcv(int msqid, struct msgbuf *msgp, size_t msgsz,
                long msgtyp, int msgflg);
```

$\Longrightarrow$

procesul apelant trebuie să aibe drept de citire asupra cozii;
în caz de succes, apelul "msgrcv()" returnează numarul de octeți de mesaj copiați efectiv în structura pointată de "msgp" (excluzând deci câmpul cu tipul mesajului), iar în caz de eșec returnează -1;
errno posibile: E2BIG, EACCES, EFAULT, EIDRM, EINTR, EINVAL, ENOMSG.

IPC - cozi de mesaje

Atributele unei cozi de mesaje se pot gestiona cu apelul:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

$\Longrightarrow$

"`msqid`" este identificatorul intern al unei cozi de mesaje;

"`cmd`" este tipul operației efectuate;

"`buf`" este adresa unei structuri în care se citesc / din care se setează atributele cozii;

IPC - cozi de mesaje

Atributele unei cozi de mesaje se pot gestiona cu apelul:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

$\Longrightarrow$

tipul structură "msqid_ds" este definit în "sys/msg.h" astfel:

```
struct msqid_ds {
    struct ipc_perm msg_perm;     /* Ownership and permissions
    time_t          msg_stime;    /* Time of last msgsnd() */
    time_t          msg_rtime;    /* Time of last msgrcv() */
    time_t          msg_ctime;    /* Time of last change */
    unsigned long   __msg_cbytes; /* Current number of bytes in
                                     queue (non-standard) */
    msgqnum_t       msg_qnum;     /* Current number of messages
                                     in queue */
    msglen_t        msg_qbytes;   /* Maximum number of bytes
                                     allowed in queue */
    pid_t           msg_lspid;    /* PID of last msgsnd() */
    pid_t           msg_lrpid;    /* PID of last msgrcv() */
};
```

tipul structură "ipc_perm" este definit în "sys/ipc.h" și a fost descris mai sus, la apelul "shmctl()";

IPC - cozi de mesaje

Atributele unei cozi de mesaje se pot gestiona cu apelul:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

$\Longrightarrow$

"cmd" poate fi:

IPC_STAT = furnizează în structura pointată de "buf" atributele cozii; procesul apelant trebuie să aibe drept de citire asupra cozii;

IPC_SET = setează unele atribute ale cozii conform structurii pointate de "buf"; se pot modifica doar membrii "msg_qbytes", "msg_perm.uid", "msg_perm.gid" și ultimii 9 biți semnificativi ai lui "msg_perm.mode";

IPC_RMID = distruge imediat coada, trezind toate procesele adormite într-o citire sau scriere la la (lor, apelurile le vor returna eroare și vor seta errno = EIDRM); parametrul "buf" este ignorat (poate fi chiar omis);

în cazurile IPC_SET si IPC_RMID procesul trebuie sa fie privilegiat sau proprietarul său efectiv trebuie să coincidă cu proprietarul sau creatorul cozii;

IPC - cozi de mesaje

Atributele unei cozi de mesaje se pot gestiona cu apelul:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

$\Longrightarrow$

apelul "msgctl()" returnează 0 la succes și -1 la eșec;

errno posibile: EACCES, EFAULT, EIDRM, EINVAL, EPERM;

În general, apelurile legate de cozile de mesaje modifică automat anumite atribute ale acestora.

IPC - cozi de mesaje

Exemplu: problema producător-consumator cu cu o coadă de mesaje:

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
#include<unistd.h>
#include<sys/stat.h>
#include<sys/wait.h>
#include<fcntl.h>
#include<signal.h>
#include<stdlib.h>
#include<stdio.h>
```

IPC - cozi de mesaje

```
pid_t p;
int msqid;
void finalizare(int);
void g(int sig){if(p==getpid()){wait(NULL); finalizare(0);} else exit(0);}
void initializare(){
  if((msqid=msgget(IPC_PRIVATE,S_IRWXU))==-1)
       {perror("msgget"); finalizare(1);}
  signal(SIGINT,g);
}
void finalizare(int n){
  switch(n){
    default:
              msgctl(msqid,IPC_RMID,NULL);
    case 1:  ;
  }
  exit(n);
}
```

IPC - cozi de mesaje

```
struct mesaj{long tip; int valoare;};
int main(){
  initializare(); p=getpid();
  if(fork()){   /* parinte = producator */
    int item; struct mesaj m;
    m.tip=1l; item=0;
    while(1){
      ++item;                              /* item=produce_item() */
      m.valoare=item;                      /* build_message(&m,item) */
      msgsnd(msqid,&m,sizeof(int),0);      /* send(consumer,&m) */
    }
  }else{      /* copil = consumator */
    int item; struct mesaj m;
    while(1){
      msgrcv(msqid,&m,sizeof(int),1l,0); /* receive(producer,&m) */
      item=m.valoare;                      /* item=extract_item(&m) */
      printf("%d\n",item);                 /* consume_item(item) */
    }
  }
  finalizare(0);
  return 0;
}
```

IPC - cozi de mesaje

Comentarii:

- programul urmează tiparul soluției date în secțiunea "Transfer de mesaje", însă consumatorul nu mai trimite producătorului mesaje goale - deci producătorul face doar "send" iar consumatorul doar "receive"; de aceea, numărul mesajelor din sistem nu este constant și astfel producătorul ar putea fi din când în când blocat în "send";

- observăm structura și mai simplă a programului: nu mai trebuie implementată explicit zona tampon (într-un segment de memorie partajată) și gestionarea ei ca o coadă alocată într-un vector parcurs circular, nu mai trebuie folosite instrumente auxiliare de sincronizare (semnale, semafoare) - deci coada de mesaje conține atât mecanismele de partajare a datelor cât și pe cele de sincronizare;

- alte comentarii sunt ca la soluția dată în secțiunea "IPC - segmente de memorie partajată".

IPC - cozi de mesaje

Instrumentul cozilor de mesaje este atât de puternic încât cu o singură coadă de mesaje putem stabili o comunicare între mai mult de două procese, fără a apărea confuzii între mesajele destinate unuia sau altuia - mesajele din coadă sunt delimitate logic (deci nu se pot amesteca conținuturile lor), iar tipul cu care sunt marcate face ca fiecare proces sa poată selecta doar mesajele adresate lui.

Prezentăm în continuare o variantă generalizată a problemei producător - consumator, în care avem mai mulți producători și mai mulți consumatori, fiecare producând / consumând doar item-uri de un anumit tip și comunicând printr-o singură coadă de mesaje fără a apărea confuzii între mesajele destinate unuia sau altuia:

IPC - cozi de mesaje

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
#include<unistd.h>
#include<sys/stat.h>
#include<sys/wait.h>
#include<fcntl.h>
#include<stdlib.h>
#include<stdio.h>
pid_t p;
int msqid;
void finalizare(int);
void initializare(){
  if((msqid=msgget(IPC_PRIVATE,S_IRWXU))==-1)
       {perror("msgget"); finalizare(1);}
}
void finalizare(int n){
  switch(n){
    default:
              msgctl(msqid,IPC_RMID,NULL);
    case 1:  ;
  }
  exit(n);
}
```

IPC - cozi de mesaje

```
struct mesaj{long tip; int valoare;};
void genprod(char *nume, long tip, int increment, int nr){if(!fork()){
    int item,i; struct mesaj m;
    m.tip=tip; item=0;
    i=0; do{
      item+=increment;                    /* item=produce_item() */
      m.valoare=item;                     /* build_message(&m,item) */
      msgsnd(msqid,&m,sizeof(int),0);     /* send(consumer,&m) */
      ++i;
    }while(i<nr);
    printf("Producator %s de tip %ld: trimise %d\n",nume,tip,i);
    exit(0);
}}
void gencons(char *nume, long tip, int nr){if(!fork()){
    int item,i; struct mesaj m;
    i=0; do{
      msgrcv(msqid,&m,sizeof(int),tip,0); /* receive(producer,&m) */
      item=m.valoare;                     /* item=extract_item(&m) */
      printf("%d\n",item);                /* consume_item(item) */
      ++i;
    }while(i<nr);
    printf("Consumator %s de tip %ld: primite %d\n",nume,tip,i);
    exit(0);
}}
```

IPC - cozi de mesaje

```
int main(){
  initializare(); p=getpid();
  genprod("A",1,1,2); genprod("B",1,10,3); genprod("C",2,100,5);
  gencons("x",1,5); gencons("y",2,1); gencons("z",2,4);
  while(wait(NULL)!=-1);
  finalizare(0);
  return 0;
}
```

IPC - cozi de mesaje

Comentarii:

- fiecare producător / consumator este lansat ca un proces copil și are un nume "`nume`", un tip "`tip`" al mesajelor pe care le produce / consumă și un anumit număr "`nr`" de mesaje pe care le produce / așteaptă; dacă se termină normal, fiecare afișaza câte mesaje a produs / primit;

- părintele comun așteaptă terminarea copiilor cu "`while(wait(NULL)!=-1)`"; aceasta nu este așteptare ocupată deoarece la fiecare iterație se blochează în "`wait(NULL)`" până se (mai) termină un copil - în total va cicla de atâtea ori câți copii are, apoi la iterația următoare, ne mai având copii, "`wait(NULL)`" îi va returna -1.

IPC - cozi de mesaje

La o rulare putem obține:

```
Producator A de tip 1: trimise 2
Producator B de tip 1: trimise 3
Producator C de tip 2: trimise 5
1
2
10
20
30
Consumator x de tip 1: primite 5
100
Consumator y de tip 2: primite 1
200
300
400
500
Consumator z de tip 2: primite 4
```

Cuprins

Fișiere tub

Fișierele tub (**conductă**, **pipe**) sunt un tip special de fișiere.

Investigarea lor necesită cunoștinte avansate despre fișiere și va fi facută în cursul despre gestiunea fișierelor.

Menționam doar că, spre deosebire de cozile de mesaje, secvențele de informații scrise în tuburi nu sunt delimitate logic ca mesajele (tubul reține o simplă succesiune de octeți) iar dacă mai multe secvențe de informații sunt scrise simultan de procese diferite, octeții lor se pot amesteca.