Concurență sub Unix la nivel de procese şi threaduri

1 Concurență la nivel de proces Unix 3
1.1 Semnale 3
1.1.1 Conceptul de semnal Unix 3
1.1.1.1 Semnale Unix 3
1.1.1.2 Lista semnalelor 4
1.1.1.3 Tratarea semnalelor 5
1.1.2 Apelul sistem signal şi exemple 5
1.1.2.1 Apelul signal 5
1.1.2.2 Utilizarea semnalelor pentru prevenirea stării zombie 6
1.1.2.3 Utilizarea semnalelor la blocarea tastaturii 7
1.1.3 Alte apeluri în contextul semnalelor 9
1.1.3.1 kill şi raise 9
1.1.3.2 alarm şi pause 10
1.1.3.3 abort şi sleep 10
1.1.4 Seturi de semnale 10
1.1.5 Scenarii posibile de folosire a semnalelor 12
1.1.5.1 Ignorarea unui semnal 12
1.1.5.2 Terminare cu acțiuni premergătoare 12
1.1.5.3 Manevrarea reconfigurărilor dinamice 13
1.1.5.4 Rapoarte periodice de stare 13
1.1.5.5 Activarea / dezactivarea depanării 14
1.1.5.6 Implementarea unui timeout 15
1.2 Comunicații între procese Unix (IPC) 16
1.2.1 Structuri, apeluri sistem şi concepte comune 16
1.2.1.1 Drepturi de acces la IPC 17
1.2.1.2 Cheie de identificare 17
1.2.1.3 Apeluri sistem 19
1.2.2 Comunicarea prin memorie partajată 20
1.2.2.1 Definire şi apeluri sistem 20
1.2.2.2 Creare, deschidere, ataşare, control 21
1.2.2.3 Exemplu: linii care repetă acelaşi caracter 22
1.2.2.4 Problema foii de calcul rezolvată prin memorie partajată 25
1.2.3 Comunicarea prin cozi de mesaje 29
1.2.3.1 Flux de octeți 29
1.2.3.2 Schimburi de mesaje 32
1.2.3.3 Cozi de mesaje Unix 33
1.2.3.4 Acces şi operații asupra cozii 34
1.2.3.5 Un exemplu de utilizare a cozilor de mesaje 36
1.3 Semafoare 39
1.3.1 Conceptul de semafor 39
1.3.2 Semafoare Unix 40
1.3.2.1 Structura setului de semafoare Unix 40
1.3.2.2 Crearea unui set şi accesul la un set existent 41
1.3.2.3 Operații asupra semafoarelor Unix 41
1.3.2.4 Controlul seturilor de semafoare 42

1.3.3 Exemple de utilizare a semafoarelor Unix 43
1.3.3.1 Semafor binar şi secțiune critică 43
1.3.3.2 Problema producătorului şi consumatorului 44
1.3.3.3 Utilizarea semafoarelor la o foaie de calcul tabelară 47
1.3.3.4 Blocarea fişierelor folosind semafoare 52
1.3.3.5 Bibliotecă de emulare a semafoarelor teoretice 54
1.3.3.6 Blocarea fişierelor cu semafoare (2) 58
1.3.3.7 Client / server cu acces exclusiv la memoria partajată 59
1.3.3.8 Zone tampon multiple 62
2 Concurență la nivel de threaduri Unix 68
2.1 Relația procese - thread-uri 68
2.1.1 Definirea procesului 68
2.1.2 Reprezentarea în memorie a unui proces 68
2.1.3 Definiția threadului 70
2.2 Thread-uri pe platforme Unix: Posix şi Solaris 71
2.2.1 Caracteristici şi comparații Posix şi Solaris 71
2.2.1.1 Similarități şi facilități specifice 72
2.2.2 Operații asupra thread-urilor: creare, terminare 72
2.2.2.1 Crearea unui thread 72
2.2.2.2 Terminarea unui thread 73
2.2.2.3 Aşteptarea terminării unui thread 75
2.2.2.4 Un prim exemplu 75
2.2.3 Instrumente standard de sincronizare 78
2.2.3.1 Operații cu variabile mutex 78
2.2.3.2 Operații cu variabile condiționale 80
2.2.3.3 Operații cu semafoare 83
2.2.3.4 Blocare de tip cititor / scriitor (reader / writer) 85
2.2.4 Exemple de sincronizări 86
2.2.5 Obiecte purtătoare de atribute Posix 88
2.2.5.1 Inițializarea şi distrugerea unui obiect atribut 89
2.2.5.2 Gestiunea obiectelor purtătoare de atribute thread 90
2.2.5.3 Gestiunea obiectelor purtătoare de atribute mutex 91
2.2.5.4 Gestiunea obiectelor purtătoare de atribute pentru variabile condiționale 92
2.2.5.5 Purtătoare de atribute pentru obiecte partajabile reader / writer 92
2.2.6 Planificarea thread-urilor sub Unix 92
2.2.6.1 Gestiunea priorităților sub Posix 93
2.2.6.2 Gestiunea priorităților sub Solaris 93
2.2.7 Problema producătorilor şi a consumatorilor 93

1 Concurență la nivel de proces Unix

1.1 Semnale

1.1.1 Conceptul de semnal Unix

1.1.1.1 Semnale Unix

Un *semnal* este o întrerupere software pe care un proces o primeşte spre a fi informat de apariția unuoi eveniment. Semnalele se folosesc în general pentru situații de excepție, alarme, terminări neașteptate şi, mai rar, pentru comunicația între procese. Ele reprezintă calea cea mai simplă, mai veche şi mai rigidă de comunicație.

Fiecare semnal are un nume prefixat de SIG, numărul semnalului fiind între 1 şi 31. Informația primită de proces prin semnal este minimă: doar tipul semnalului.

Cauzele semnalelor pot fi grupate astfel:

1. Erori apărute în timpul execuției: adresări înafara spațiului de memorie, scriere într-o zonă de memorie care este read-only pentru proces, diverse erori hard etc.
2. Erori soft la nivelul unor aspeluri istem: inexistența funcției sistem, scriere în pipe fără să existe proces de citire sau invers, parametri de apel eronați, inexistența unor resurse etc.
3. Comunicarea unui proces cu un alt proces.
4. Comanda `kill`, terminarea forțată a unui proces, alarmă.
5. Intreruperi lansate de către utilizator prin Ctrl-\, Ctrl-Break, DEL, deconectare terminal etc.
6. Terminarea unui proces fiu (execuția `exit`)

Un semnal este *generat* spre un proces, sau *transmis* unui proces în momentul apariției evenimentului care provoacă semnalul. Un semnal este *livrat* unui proces când acțiunea ataşată semnalului este executată. Intre cele două stări, semnalul se consideră *nerezolvat*.

Trebuie reamintit faptul că procesul nu ştie de unde primeşte semnalul! Este posibil ca un proces să blocheze livrarea unui semnal. Dacă procesul blochează semnalul dar el este generat spre proces, atunci semnalul rămâne nerezolvat fie până la deblocarea lui, fie până la schimbarea rezolvării în ignorarea semnalului.

Din rațiuni didactice, exemplele alese de noi pentru lucrul cu semnale sunt relativ simple. Trebuie însă să menționăm faptul că semnalele sunt folosite în aplicații relativ complexe. Pentru aplicații reale se impune o documentare exactă asupra semnalelor pentru versiunea Unix folosită.

1.1.1.2 Lista semnalelor

Am încercat o reunire a numelor de semnale folosite în principalele familii de sisteme Unix. Cele 31 de semnale (la unele am indicat şi numărul), definite în headrul <`signal.h`>, sunt:

- SIGHUP (1) primit de fiecare proces la deconectarea terminalului lui de control.
- SIGINT (2) primit de fiecare proces la apăsarea Ctrl-C sau DEL.
- SIGKILL (9 - hard termination signal) folosit pentru terminarea de urgență a unui proces. În absența urgenței, se recomandă SIGTERM.
- SIGTERM (15 - soft termination signal) oprire prin `kill` sau `shutdown`.
- SIGCHLD (20) este trimis de un proces fiu spre părinte atunci când el se termină sau când este oprit.
- SIGUSR1 şi SIGUSR2 (30, 31) semnale utilizator, care pot fi folosite la comunicarea între procese.
- SIGABRT este generat de apelul `abort` şi provoacă terminarea anormală a procesului.
- SIGALRM (14) este trimis când orologiul de alarmă, poziționat prin `alarm`, constată că timpul a expirat.
- SIGVTALARM generat la expirarea timpului unui ceas virtual poziționat prin `settimer`.
- SIGPROF generat la expirarea timpului unui orologiu poziționat prin `settimer`.
- SIGBUS Detectarea unei erori hard.
- SIGCONT este transmis unui proces oprit când se doreşte continuarea sa.
- SIGEMT transmis de hard la o eroare de implementare.
- SIGFPE eroare hard de virgulă flotantă.
- SIGILL transmis la detectarea unei instrucțiuni ilegale.
- SIGINFO generat de driver-ul de terminal la tastarea lui Ctrl-T. Drept urmare, sunt afişate informații de stare a proceselor aflate în background.
- SIGIO (23) indică apariția unui eveniment asincron de I/O.
- SIGIOT detectare eroare hard dpendantă de implementare.
- SIGPIPE primit de procesul care scrie într-un pipe dar nu există proces care să citească din pipe.
- SIGPOOL specific SVR4 generat la apariția unui eveniment I/O.
- SIGPWR utilizat frecvent de surse de curent UPS la detectarea unei căderi de tensiune.
- SIGQUIT generat la apăsarea Ctrl-\.
- SIGEGV referire la un spațiu înafara zonei de adresare (segment violation).
- SIGSTOP pune procesul în stare de aşteptare, până la un alt semnal, de continuare sau de terminare.
- SIGSYS argumente greşite la apeluri sistem.
- SIGTRAP trimis la modul trap după fiecare instrucțiune.
- SIGTSTP transmis la apelul lui Ctrl-Z
- SIGTTIN generat când un proces în background încearcă să citească de la terminalul său de control.

- SIGTTOU generat când un proces în background încearcă să scrie pe terminalul său de control.
- SIGURG generat la apariția unei urgențe. De regulă este generat la recepția incorectă a datelor la o conexiune în rețea.
- SIGWINCH la modificarea dimensiunilor unei ferestre prin `ioctl`, este generat spre procesele în background.
- SIGXCPU generat la depăşirea limitei de timp CPU fixată prin `setrlimit`.
- SIGXFSZ generat la depăşirea limitei de fişiere admisă prin `setrlimit`.

1.1.1.3 Tratarea semnalelor

Tratarea unui semnal se poate face în trei moduri:
- Ignorarea semnalului şi continuarea activității procesului. Doar SIGKILL nu poate fi ignorat.
- Sarcina tratării semnalului este lăsată pe seama nucleului. În acest caz, semnalele SIGCHLD, SIGPWR, SIGINFO, SIGURG şi SIGWINCH sunt ignorate, SIGCONT continuă procesul oprit, iar toate celelalte semnale duc la terminarea procesului.
- Tratarea semnalului printr-o procedură proprie, care este lansată în mod automat la apariția semnalului. La terminarea procedurii, procesul este reluat din punctul din care a fost întrerupt.

Modul de tratare a semnalelor este transmis proceselor fii. Funcțiile `exec*` lasă tratarea în seama nucleului, chiar dacă înainte de `exec` a fost prevăzut altfel.

Nucleul pune la dispoziția utilizatorilor patru funcții principale de tratare a semnalelor:
1. `signal` pentru specificarea modului de tratare;
2. `kill` pentru trimiterea de către un proces a unui semnal către alt proces;
3. `pause` pune în aşteptare un proces până la sosirea unui semnal;
4. `alarm` pentru generarea semnalului SIGALARM.

Pe lângă acestea, mai există şi alte apeluri utile în contextul semnalelor, aşa cum se va vedea în cele ce urmează.

1.1.2 Apelul sistem signal şi exemple

1.1.2.1 Apelul signal

Aces apel sistem presupune citarea headerului <signal.h>. Prototipul apelului este:

```
void (*signal (int semnal, void (*functie)(int)))(int);
```

Deci, mai clar vorbind, signal are două argumente: numărul semnalului şi un nume de funcție:

`semnal` este numărul semnalului (constanta semnal) pentru care se stabileşte modul de tratare.
`functie` poate avea una dintre următoarele trei valori:

- `SIG_DFL` specifică tratarea implicită a semnalului de către nucleu, în modalitatea descrisă mai sus.
- `SIG_IGN` indică ignorarea semnalului, acţiune de la care fac excepţie SIGKILL şi SIGSTOP.
- `nume` este numele efectiv al unei funcţii definite de către utilizator. Această funcţie are ca şi parametru un întreg care conţine numărul semnalului apărut. Ea (funcţia) va intra în lucru în momentul tratării semnalului pentru care este activată.

Valoarea întoarsă de către funcţia signal este adresa rutinei anterioare de tratare a semnalului.

1.1.2.2 Utilizarea semnalelor pentru prevenirea stării zombie

Serverele concurente transmit sarcina rezolvării fiecărei cereri unui alt proces. Schema generală de funcţionare a unui server concurent este indicată în programul ?. Am ales pentru prezentare comunicarea prin *socket*,

```
// Partea de initializare a comunicatiei

s = socket ( … );
bind ( … );
listen ( … );

/* Urmeaza partea ciclica a serverului */

for ( ; ; ) {
  f = accept ( s, … );
    // S-a primit o noua cerere de la un client

  if ( fork () == 0 ) {

    // Codul procesului fiu, cel care rezolva efectiv cererea
  else {

    // Codul serverului, care asteapta o noua cerere

  }

   // Partea de terminare a activitatii serverului

}
```

Programul ? Schema unui server concurent

Termenul de "*zombie*" denumete acele procese de sub Unix care au fost lansate de către un proces "tată" şi care "tată" şi-a terminat activitatea fără să mai aştepte terminarea procesului "fiu".

În general, atunci după ce un proces lansează, cu `fork`(), un proces fiu, cele două evoluează în paralel. Atunci când procesul fiu şi-a terminat activitatea, el face un

apel sistem `exit`, prin care îşi anunță părintele că şi-a încheiat activitatea. La rândul lui, la un moment dat părintele trebuie să execute un apel sistem `wait` pentru a aştepta terminarea procesului fiu. Acest apel poate fi făcut fie înainte, fie după terminarea procesului fiu. Un proces fiu care s-a terminat, este trecut de către nucleu în starea *zombie*, până când părintele execută apelul sitem `wait`.

În multe servere concurente, apelul sistem `wait` nu poate fi practic executat decât dacă serverul (procesul părinte) rămâne în aşteptare numai după un anumit fiu. Ori această cerință este imposibilă de impus serverelor care aşteaptă să fie contactate în orice moment de către mulți clienți.

Incepând cu Unix System V Release 4 (SVR4), s-a introdus o posibilitate simplă de evitare a apariției proceselor *zombie*. Este suficient ca în partea de inițializare a serverului concurent (deci înainte de începerea creării proceselor fii de servire a cererilor), să apară secvența din programul ?.

```
# include <signal.h>

…

signal(SIGCHLD, SIG_IGN)
```

Programul ? Secvență de evitare "zombie" stil Unix System V

Prin apelul `signal` se cere procesului respectiv să ignore (SIG_IGN) semnalul de terminare a proceselor fii (SIGCHLD). Acest mecanism este de asemenea valabil, printre altele, pentru toate versiunile de sistem de operare LINUX

Varianta Unix BSD (Berkeley Software Distribution) nu admite o soluție aşa de simplă pentru evitarea proceselor "*zombie*". Aceasta pentru că la această variantă de Unix efectul unui apel sistem `signal` este valabil o singură dată.

Rezolvarea problemei în acest caz se poate face după o schemă ca cea din programul ?.

```
#include <signal.h>
…
void waiter(){          // Functie de manipulare a apelurilor signal
                    // Trateaza un semnal
  wait(0);         // Sterge fiul recent terminat
  signal(SIGCHLD, waiter);    // Reinstalare handler signal
}//waiter

signal(SIGCHLD, waiter);      // Plasat în partea de initializare
```

Programul ? Evitare „zombie" stil BSD

1.1.2.3 Utilizarea semnalelor la blocarea tastaturii

Programul ? permite blocarea tastaturii de către utilizatorul real. Deblocarea se realizează numai atunci când de la tastatură se dă parola utilizatorului real.

```
#define _XOPEN_SOURCE 0
```

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include<string.h>
#include <pwd.h>
#include <shadow.h>
#include <signal.h>
   main() {
      char *cpass, pass[15];
      struct passwd *pwd;
      struct spwd *shd;
      signal(SIGHUP, SIG_IGN);
      signal(SIGINT, SIG_IGN);
      signal(SIGQUIT, SIG_IGN);
      setpwent();
      pwd = getpwuid(getuid());
      endpwent();
      setspent();
      shd = getspnam(pwd->pw_name);
      endspent();
      setuid(getuid()); // Redevin userul real
      for ( ;; ) {
         strcpy(pass, getpass("...tty LOCKED!!"));
         cpass = crypt(pass, shd->sp_pwdp);
         if (!strcmp(cpass, shd->sp_pwdp))
            break;
      }//for
   }//main
//lockTTY.c
// Compilare: gcc -lcrypt -o lockTTY lockTTY.c
// User root: chown root.root lockTTY
// User root: chmod u+s lockTTY
//  Executie: ./lockTTY
```

Programul ? Sursa `lockTTY.c`

Părțile specifice semnalelor sunt prezentate de specificarea headerului din linia 8 şi în cele trei linii cu `signal`, . care comandă ignorările celor trei semnale care ar putea elibera în mod forțat tastatura.

După ignorarea semnalelor, se deschide accesul la fişierul `/etc/passwd`. Se extrage în structura punctată de `pwd` linia din acest fişier corespunzătoare utilizatorului care a lansat programul. Apoi se închide accesul la acest fişier.

Urmează deschiderea accesului la fişierul `/etc/shadow`, unde sunt memorate parolele criptate ale utilizatorilor. Se extrage în structura punctată de `shd` linia din acest fişier care corespunde numelui utilizatorului curent. Apoi se închide accesul la acest fişier.

Important pentru securitatea sistemului: Accesul la fişierul /etc/shadow este permis numai utilizatorului root! Prin urmare, acest program va funcționa numai dacă:

1. Proprietarul lui este userul root, care va atribui programului dreptul de *setuid* spre a putea fi folosit de către orice user din sistemul lui propriu de directori.
2. Partiția în care este găzduit sistemul de fişiere al userului curent trebuie să aibă, conferit de adminstrator, dreptul de a executa programe cu atributul setuid.

Incălcarea uneia dintre aceste două cerințe va provoca: "`Segmentation fault (core dumped)`".

Apelul setuid(getuid()) are menirea de a reveni programul la starea non-setuid, adică să se comporte ca şi un program obişnuit (setuid a fost necesar doar pentru accesul la `/etc/shadow`).

In continuare, programul execută un ciclu infinit în care:

- Afişează prompterul "`...tty LOCKED`!!" şi aşteaptă de la tastatură o parolă, pe care o depune, în clar, în vectorul `pass`.
- Parola în clar este criptată şi pointerul `cpass` indică stringul în care sistemul a plasat această criptare.
- Se compară criptarea a ceea ce s-a dat de la tastatură cu ceea ce se află în `/etc/shadow` corespunzător parolei criptate a userului curent. In caz de concordanță programul se termină şi implicit tastatura este deblocată. Dacă cele două criptări nu concordă, atunci ciclul se reia de la afişarea prompterului "`...tty LOCKED`!!".

1.1.3 Alte apeluri în contextul semnalelor

1.1.3.1 kill şi raise

Apelul sistem `kill` transmite un semnal la un proces sau la un grup de procese. Funcția `raise` permite unui proces să-şi trimită lui însuşi un semnal. Prototipurile celor două apeluri sunt:

```
int kill(pid_t pid, int semnal);
int raise(int semnal);
```

Ambele funcții întorc 0 la succes sau -1 la eroare.
Semnificațiile parametrilor sunt:

- `semnal` este numărul semnalului care se doreşte a fi transmis.
- `pid` este interpretat, în funcție de valoarea lui, astfel:
  - `pid` > 0 înseamnă că semnalul va fi transmis procesului al cărui pid este cel specificat.
  - `pid` == 0 indică trimiterea semnalului la toate procesele din acelaşi grup cu emițătorul şi la care emițătorul are dreptul să trimită semnalul.
  - `pid` < -1 indică faptul că semnalul va fi transmis tuturor proceselor pentru care GID == -`pid` şi la care emițătorul are dreptul să trimită semnalul.
  - `pid` == -1 este folosit, la unele versiuni, de către superuser pentru broadcast.

Programul ? prezintă un exemplu de folosire a `kill`, şi anume pentru a realiza `raise` prin `kill`.

```
// Implementarea raise folosind kill
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>
```

```
int raise(int sem) {return(kill(getpid(),sem));}
```

Programul ? Implementare `raise` prin `kill`

1.1.3.2 alarm şi pause

Funcția `alarm` permite poziționarea unui *timer*. La expirarea timpului, va fi generat SIGALRM. Dacă semnalul este ignorat sau nu este captat, acțiunea implicită este de a se termina procesul.
Prototipul ei este

```
unsigned int alarm(unsigned int secunde);
```

Intoarce numărul de secunde rămase de la cererea anterioară `alarm`.
`secunde` indică numărul de secunde după care se va declanşa SIGALRM.

Funcția `pause` suspendă (pune în starea WAIT) procesul apelant până la primirea unui semnal. Prototipul ei este:

```
int pause(void);
```

Din acest apel se revine dacă se revine din rutina de tratare a semnalului. Această funcție este folosită în special în combinație cu alarm.

NU este necesar un `pause` în cazul apelurilor `read`, `msgrecv` şi `wait`.

1.1.3.3 abort şi sleep

Apelul sistem `abort` provoacă terminarea anormală a procesului apelator. Prototipul ei este:

```
void abort(void);
```

Apelul sistem `sleep` pune în aşteptare procesul apelator până la scurgerea unui anumit interval de timp. Prototipul ei este:

```
unsigned int sleep(unsigned int secunde);
```

Intoarce 0 la scurgerea intervalului secunde specificat, sau întoarce numărul de `secunde` rămase de scurs dacă procesul a fost relansat mai repede printr-un semnal.

1.1.4 Seturi de semnale

Versiunile mai noi de Unix au implementate funcții care lucrează pe *seturi de semnale.*

Un set de semnale este definit de tiul de date sigset_t şi de cinci funcții care lucrează cu acest set:

```
int sigemptyset(sigset_t *set);     // invalideaza semnalele setului
int sigfillset(sigset_t *set);      // valideaza semnalele setului
int sigaddset(sigset_t *set, int semnal);   // adauga semnal la set
int sigdelset(sigset_t *set, int semnal);   // sterge semnal din set
int sigismember(sigset_t *set, int semnal); // confirma sau infirma
                                        // prezenta semnalului in set
```

Masca de semnale asociată unui proces constă din setul de semnale blocate spre a fi livrate procesului. Prin intermediul apelului sistem `sigprocmask` se poate manevra această mască:

```
int sigprocmask(int mod, sigset_t *set, sigset_t *vset);
```

Intoarce 0 la succes şi -1 la eroare. Semnificația parametrilor este:

- Dacă vset != NULL provoacă întoarcerea măştii curente în spațiul indicat de set.
- Dacă `set` == NULL atunci masca de semnale nu se modifică.
- Dacă `set` != NULL atunci, în funcție de valoarea constantei mod, avem:
  - SIG_BLOCK se reuneşte masca veche cu cea indicată prin `set`.
  - SIG_UNBLOCK masca veche este intersectată cu cea indicată de `set`.
  - SIG_SETMASK înlocuieşte masca veche cu cea indicată de `set`.

Aflarea setului de semnale nerezolvate (apărute în timpul blocării) se face cu ajutorul apelului sistem `sigpending`, a cărui prototip este:

```
int sigpending(sigset_t *set);
```

Intoarce 0 la succes şi -1 la eroare. În caz de succes va depune în mulțimea *`set` semnalele nerezolvate.

Realizarea de regiuni critice poate fi făcută cu ajutorul modificării măştilor de semnale. Realizarea unei astfel de regiuni critice se poate face astfel:

```
sigset_t newmask, oldmask;
- - -
sigemptyset(&newmask);
sigaddset(&newmask, SIG__);

// Ocupare regiune critica prin mascare SIG__ şi salvare masca
if(sigprocmask(SIG_BLOCK,&newmask,&oldmask)<0) {err_sys…}

  /s regiune critica

// Deblocare SIG__ şi restaurare masca
if(sigprocmask(SIG_SETMASK,&oldmask, NULL)<0) {err_sys…}
  pause();
- - -
```

Se observă că restaurarea şi punerea procesului în stare de aşteptare se face prin două apeluri sistem independente, între care se poate interpune un semnal, ceea ce ar putea conduce la necazuri :(…

Inconvenientul poate fi remediat prin apelul sistem `sigsuspend`:

```
int sigsuspend (sigset_t *semmask);
```

Prin acest apel sistem masca este poziționată la valoarea indicată prin `semmask`, iar procesul este pus în aşteptare până când se captează un semnal.

1.1.5 Scenarii posibile de folosire a semnalelor

În cele ce urmează vom examina şase utilizări tipice de semnale, pentru care vom preznta schema cadru de utilizare.

1.1.5.1 Ignorarea unui semnal

Un proces poate decide ignorarea unui semnal, cu excepția lui SIGKILL. Schema de funcționare este:

```
main() {
- - -
   signal(SIGINT, SIG_IGN);
- - -
// Aici se lucreaza cu ignorarea semnalului SIGINT,
// si în acelasi mod se poate ignora orice alt semnal
}
```

Programul ? Ignorarea unui semnal

1.1.5.2 Terminare cu acțiuni premergătoare

Uneori la primirea unui semnal, cum ar fi SIGINIT, este nevoie ca înainte de terminare (acțiunea propriu-zisă a semnalului) să fie executate unele acțiuni premergătoare. Astfel de acțiuni posibile ar fi: ştergerea fişierelor temporare, a cozilor de mesaje sau a segmentelor de memorie partajată, aşteptarea terminării fiilor etc. O schemă tipică de lucru în acest caz ar fi cea din programul ?.

```
// Variabile globale
   int fiu;

   void terminare(int semnal) {
      unlink("/tmp/FisierTemporar");
      kill(fiu, SIGTERM);
      wait(0);
      fprintf(stderr, "Terminarea programului\n");
      exit(1);
}//functia de tratare a semnalului

   main() {
      signal(SIGINT, terminare); // Anuntare functie tratare semnal
- - -
//  Creare fisier temporar
      open("/tmp/FisierTemporar", O_RDWR | O_CREAT, 0644);
- - -
//  Creare proces fiu
      fiu = fork();
- - -
```

```
}
```

Programul ? Acțiuni înaintea terminării prin semnal

In cazul terminărilor normale: `exit` sau `return` din funcția `main` etc. se poate foloseşte apelul sistem `atexit`(). Prototipul acestei apel este:

```
int atexit(void (*functie)(void))
```

Apelul înregistrează (anunță sistemul de operare că) functie care va intra în lucru înainte de o terminare normală. Aceasta permite întreținerea unei stive de apeluri de funcții pentru ştergeri înaintea terminărilor. Din păcate, functie înregistrată prin atexit ea nu intră în lucru la nici o terminare anormală, cum este cazul la unele semnale.

1.1.5.3 Manevrarea reconfigurărilor dinamice

Multe programe îşi citesc parametrii de configurare dintr-un fişier special destinat. Citirea acestuia se face în momentul lansării programului. Dacă se modifică acest fişier, atunci este necesară restartarea programului spre a lua în calcul noile configurări.

Utilizarea unui handler de semnal poate evita restartarea. Astfel, după modificarea fişierului, se trimite prin `kill` un semnal procesului şi acesta îşi reciteşte fişierul de reconfigurare. Scenariul de lucru este prezentat în programul ?

```
void citeste_configuratia(int semnal) {
   int fd;
   fd = open("FisierConfigurare",O_RDONLY);
   // citirea parametrilor de configurare
- - -
   close(fd);
   signal(SIGHUP, citeste_configuratia);
}//functia de citire a fisierului de configurare

main() {
   // citirea initiala a configuratiei
   citeste_configuratie();

   while(1) {
    // ciclu executie
   - - -
  }
}
```

Programul ? Citirea dinamică a configurărilor

Pentru a provoca recitirea configurărilor, se va da comanda:

```
$  kill -SIGHUP  pidulprocesului
```

1.1.5.4 Rapoarte periodice de stare

Un program care lucrează un timp îndelungat, poate acumula o cantitate mare de informații interne: diverse informații statistice, de rutare, rezultate intermediare etc. Instalând un handler de semnal, se poate provoca, după dorință, afişarea acestor informații de stare şi eventual depunerea lor într-un fişier jurnal. Dacă programul eşuează, datele din jurnal pot fi analizate ulterior în vederea depanării. In acest context, oferim o modalitate ca utilizatorul să stabilească, on-line, fiecare moment de afişare / salvare.

Programul trebuie să-şi plasze aceste informații de stare ca şi globale, spre a fi accesibile atât din programul propriu-zis care le crează, cât şi din handlerul de semnal. Un scenariu simplu este cel din programul ?

```
// informații globale de stare
   int  numar;

void  tipareste_stare(int semnal) {

// Tipareste informatiile de stare solicitate
   printf("Copiat %d blocuri\n", numar);
   signal(SIGUSR1, tipareste_stare);
}//handlerul de semnal

main() {
   signal(SIGUSR1,tipareste_stare);
   - - -
   for(numar=0; numar < UN_NUMAR_MARE; numar++) {
     - - - // citeste bloc de pe un stream
     - - - // prelucreaza bloc
     - - - // tipareste bloc
   }//for
}//main
```

Programul ? Tipăririrea unor rapoarte de stare

Pentru a provoca tipărirea unui raport, se va da comanda:

```
$  kill -SIGUSR1  pidulprocesului
```

1.1.5.5 Activarea / dezactivarea depanării

Multe dintre programe sunt presărate, mai ales în faza de testare, cu o serie de instrucțiuni `printf` (`fprintf(stderr ... )`) care să afişeze urme ale execuției. Folosind un semnal şi o variabilă întreagă pe post de comutator boolean, afişarea se poate activa / dezactiva alternativ. Exemplul din programul ? oferă o astfel de posibilitate, în care variabila bascula are rolul de comutator, iar semnalul SIGHUP anunță momentele de comutare

```
int bascula;

void debugNOdebug(int semnal) {
   bascula ^= 1; // schimba starea indicatorului de depanare
   signal(SIGHUP, debugNOdebug);
}

main() {
   // Initial fara depanare
```

```
   bascula = 0;
   // instalare handler semnal */
   signal(SIGHUP, debugNOdebug);
- - -
   // tiparirile de depanare trebuie sa fie de forma:
   if (bascula) {printf(...) sau fprintf(stderr ... );}
- - -
}
```

Programul ? Activarea / dezactivarea depanării

Pentru a inversa starea de depanare, se va da comanda:

```
$  kill -SIGHUP  pidulprocesului
```

1.1.5.6 Implementarea unui timeout

Exemplul cadru care urmează arată cum se poate evita de către un proces aşteptarea indefinită.

Vom construi o funcție, `int t_gets(s,t)`, care citeşte de la intrarea standard un string şi-l depune la adresa `s`, însă pentru aceasta nu aşteaptă mai mult de `t` secunde.

Funcția `t_gets` va întoarce:

- -1 la întâlnirea sfârşitului de fişier;
- -2 la depăşirea celor `t` secunde (timeout);
- lungimea şirului citit în celelalte cazuri.

Problema esențială la această funcție este aceea că, în caz de timeout, programul să revină la contextul dinaintea lansării operației de citire propriu-zise. Pentru implementarea acestei facilități trebuie folosite două apeluri sistem specifice:

```
int setjmp(jmp_buf tampon);
void longjmp(jmp_buf tampon, int valoare);
```

Funcția `setjmp` copiază contextul execuției (stiva etc.) în `tampon`, după care întoarce valoarea zero. Ulterior, folosind funcția `longjmp` se reface din `tampon` contextul copiat prin `setjmp`, după care `setjmp` întoarce încă odată (după refacerea contextului) `valoare` dată ca al doilea parametru la `longjmp`.

Această pereche de funcții acționează la nivel fizic şi permit salvarea - refacerea contextului la diverse momente ale execuției, făcând “uitate” părți din execuție.

Programul ? prezintă funcția `t_gets`, împreună cu un program de test.

```
#include <stdio.h>
#include <setjmp.h>
#include <sys/signal.h>
#include <unistd.h>
#include <string.h>
```

```
jmp_buf tampon;

void handler_timeout (int semnal) {
   longjmp (tampon, 1);
}//handler_timeout

int t_gets (char *s, int t) {
   char *ret;
   signal (SIGALRM, handler_timeout);
   if (setjmp (tampon) != 0)
      return -2;
   alarm (t);
   ret = fgets (s, 100,stdin);
   alarm (0);
   if (ret == NULL)
      return -1;
   return strlen (s);
}//t_gets

main () {
   char s[100];
   int v;
   while (1) {
      printf ("Introduceti un string: ");
      v = t_gets (s, 5);
      switch (v) {
         case -1:
            printf("\nSfarsit de fisier\n");
            return(1);
         case -2:
            printf ("timeout!\n");
            break;
         default:
            printf("Sirul dat: %s a.Are %d caractere\n", s, v-1);
      }//switch
   }//while
}//t_gets.c
```

Programul ? Implementarea unui timeout

1.2 Comunicații între procese Unix (IPC)

Incepând cu Unix System V s-au dezvoltat, pe lângă mecanismele clasice de comunicare prin pipe şi FIFO, o serie de mecanisme reunite sub sigla IPC (Inter Process Communications system). Aceste mecanisme IPC sunt:

- comunicare prin *memorie partajată*;
- comunicare prin *cozi de mesaje*.
- comunicarea prin *semafoare*;

1.2.1 Structuri, apeluri sistem şi concepte comune

După cum se va vedea, între aceste mecanisme există similarități, referitoare la fişierele header de inclus, la constantele predefinite de comandă a acțiunilor, la numele apelurilor sistem şi la semantica apelurilor sistem.

In plus, există două concepte comune celor trei IPC-uri: *structura drepturilor de acces* la IPC şi noțiunea de *cheie de identificare*.

1.2.1.1 Drepturi de acces la IPC

Indiferent de tipul IPC, în headerul `<sys/ipc.h>` este descrisă o structură de date folosită de nucleul Unix pentru gestionarea IPC. Structura este definită astfel:

```
struct ipc_perm {
  ushort uid;           // identificatorul proprietarului
  ushort gid;           // identificatorul grupului proprietar
  ushort cuid;          // identificatorul creatorului
  ushort cuid;          // identificatorul grupului creator
  ushort mode;          // drepturile de acces la IPC
  ushort seq;           // numar de secventa al descriptorului
  key_t  key;           // cheia de identificare a IPC-ului
}
```

Toate câmpurile, cu excepția lui `seq` sunt inițializate în momentul creării structurii IPC.

Drepturile de acces la IPC sunt specificate în câmpul `mode` şi sunt similare cu drepturile de acces la fişierele Unix: *r* pentru citire, *w* pentru scriere şi *x* pentru execuție. Fiecare dintre acestea sunt prezente în trei categorii: user, grup şi restul utilizatorilor. Singura deosebire este că dreptul *x* nu este folosit.

Cozile de mesaje şi memoria partajată folosesc drepturile *read* şi *write*, iar semafoarele *read* şi *alter*. Pentru specificarea drepturilor se pot folosi fie cifrele octale, fie constante specifice. Tabelul care urmează prezintă constantele de specificare a drepturilor, pe grupe de utilizatori.

	Memorie partajată	Cozi de mesaje	Semafoare
Proprietar u	SHM_R SHM_W	MSG_R MSG_W	SEM_R SEM_A
Grup g	SHM_R >>3 SHM_W >>3	MSG_R >>3 MSG_W >>3	SEM_R >>3 SEM_A >>3
Alții o	SHM_R >>6 SHM_W >>6	MSG_R >>6 MSG_W >>6	SEM_R >>6 SEM_A >>6

1.2.1.2 Cheie de identificare

In mod normal, la crearea unui IPC nucleul întoarce un handle descriptor al canalului IPC. Din păcate, acesta nu poate fi folosit decât în procesul curent şi descendenții acestuia. Pentru a permite comunicarea prin IPC din procese independente, este definită *cheia de identificare a IPC*.

O astfel de cheie este un întreg lung (`key_t`), care identifică în mod unic un IPC. Valoarea acestei chei este stabilită de comun acord de către proiectanții serverului şi a clienților IPC. In momentul creării, valoarea cheii este transmisă nucleului, iar

clienții utilizează această cheie pentru a-şi deschide în propriile programe conexiunea la IPC.

In cazul mecanismelor IPC, fiind vorba de comunicații pe aceeaşi maşină, pe post de nume se află această *cheie* de recunoaştere a canalului IPC. Astfel, la crearea canalului IPC (de obicei de către server) se fixează valoarea acestei chei, iar apoi fiecare dintre utilizatorii acestui canal (clienții) trebuie să furnizeze la deschidere această cheie.

Această relație între client şi server se poate realiza prin două moduri:

1. Proiectantul serverului şi proiectanții clienților se înțeleg asupra unui întreg care să aibă rolul de *constantă unică a canalului* (*cheie)* de recunoaştere a IPC. Apoi, la creare serverul furnizează valoarea acestei chei, iar fiecare client îşi deschide canalul furnizând aceeaşi cheie.

2. La crearea structurii IPC se specifică drept cheie constanta simbolică IPC_PRIVATE. In această situație, canalul IPC este identificat prin *descriptorul* (handle) al canalului IPC. Pentru aceasta, proiectantul trebuie să găsească o modalitate de a transmite clienților acest descriptor - de exemplu scrierea acestui handle într-un fişier accesibil tuturor protagoniştilor. Aceştia vor folosi descriptorul ca pe un canal gata deschis. Această modalitate este destul de rar folosită.

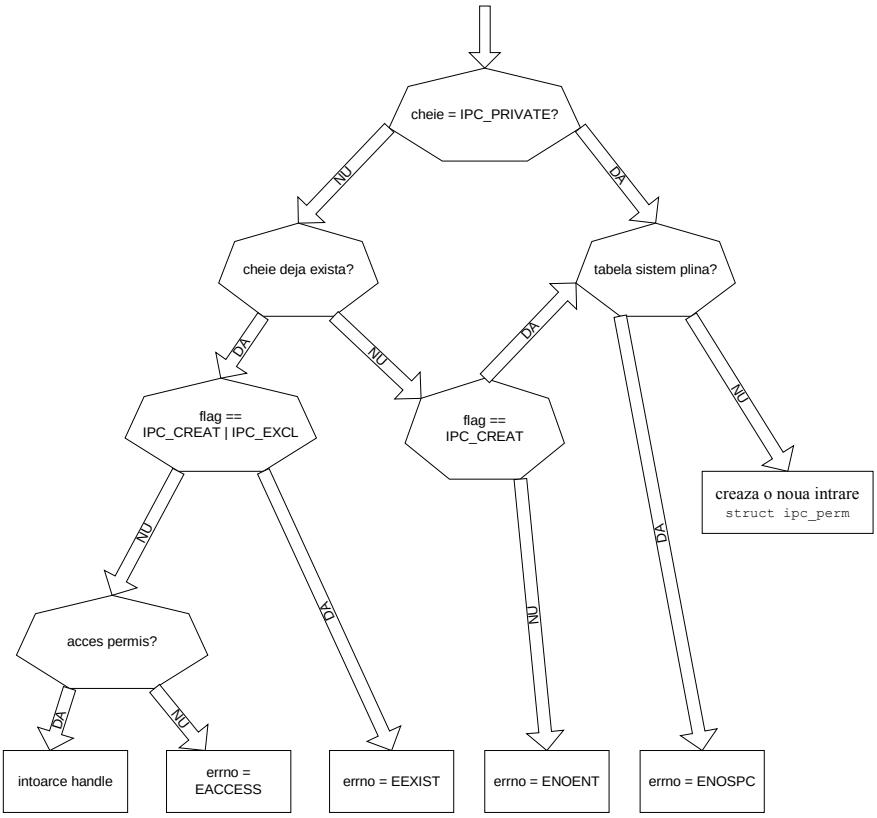

Figura ? Crearea unui canal nou IPC

Deci, diferența între cele două metode este aceea că în primul caz fiecare protagonist îşi face propria deschidere de canal (deci vor primi descriptori de canal diferiți), dar

vor furniza aceeaşi cheie, convenită în prealabil. In ccazul al doilea, există un unic întreg descriptor al canalului, folosit de către toți protagoniştii.

Regulile valabile la crearea unui nou IPC sunt prezentate în fig. ?

Principala dificultate a folosirii unei chei comune acceptate de protagoniştii la comunicație este aceea că există posibilitatea ca aceeaşi cheie să fie asociată deja la un alt canal IPC. Pentru a evita această situație, se oferă apelul sistem:

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok (char *cale, char proiect);
```

În `<sys/types.h>` se defineşte `key_t` ca un întreg pe 32 de biți. cale indică locul în structura de directori unde se află programele /datele IPC-ului dorit. proiect este un caracter furnizat să diferențieze (eventual) mai multe proiecte situate în acelaşi loc. `ftok` combină inodul lui cale cu proiect şi furnizează un număr pe post de cheie.

1.2.1.3 Apeluri sistem

Ca şi moduri de utilizare, aceste trei mecanisme IPC au multe similarități. Tabelul următor prezintă, pe categorii, apelurile sistem utilizate pentru IPC.

Cele trei apeluri `*ctl` accesează şi modifică intrarea `struct ipc_perm` a canalului IPC. Cele trei apeluri `*get` obțin un descriptor al IPC. Acestea au ca argumente *cheia de identificare IPC* şi un argument `flag` care defineşte drepturile de acces la IPC ale celor trei categorii de utilizatori.

	memorie partajată	cozi de mesaje	semafoare
fişier header	`<sys/shm.h>`	`<sys/mesg.h>`	`<sys/sem.h>`
apel sistem de creare sau deschidere	`shmget`	`msgget`	`semget`
apel sistem de control a operațiilor	`shmctl`	`msgctl`	`semctl`
apel sistem pentru operaþii IPC	`shmat` `shmdt`	`msgsnd` `msgrcv`	`semop`

Pentru crearea canalului - unul dintre apelurile `shmget`, `semget` sau `msgget`, trebuie furnizat pentru `flag`:

```
IPC_CREAT | IPC_EXCL | drepturi_de_access
```

In urma apelului se crează o structură de date, după caz `struct shmid_ds`, `struct semid_ds`, `struct msgid_ds`, care defineşte canalul. Structura este tipică fiecărui tip de canal, dar include în ea un câmp de tip `struct ipc_perm`, precum şi informații de dimensiune şi temporale asupra canalului definit.

1.2.2 Comunicarea prin memorie partajată

1.2.2.1 Definire şi apeluri sistem

Unul dintre principiile programării Unix (şi nu numai) este acela că *"fiecare proces îşi accesează în mod exclusiv propriul spațiu de memorie internă*". Uneori, în aplicațiile care folosesc IPC, se impune un schimb rapid a câtorva informații. In acest scop s-a definit conceptul de memorie partajată.

Ea permite ca două sau mai multe procese să folosească acelaşi segment de memorie.
Principiul comunicării prin memorie partajată constă în:

1. Un proces crează un segment de memorie partajată. Descriptorul zonei este trecut în structura de date ataşată zonei.
2. Procesul creator asociază segmentului de zonă partajată o cheie numerică pe care toate procesele care accesează zona trebuie să o cunoască.
3. Procesul creator asociază zonei drepturile de acces.
4. Orice proces care doreşte să acceseze segmentul de memorie partajată, inclusiv creatorul, trebuie să şi-o ataşeze la spațiul lui virtual de adrese. In urma acestei ataşări obține adresa zonei de memorie partajată.

Acest principiu este ilustrat în fig. ?.

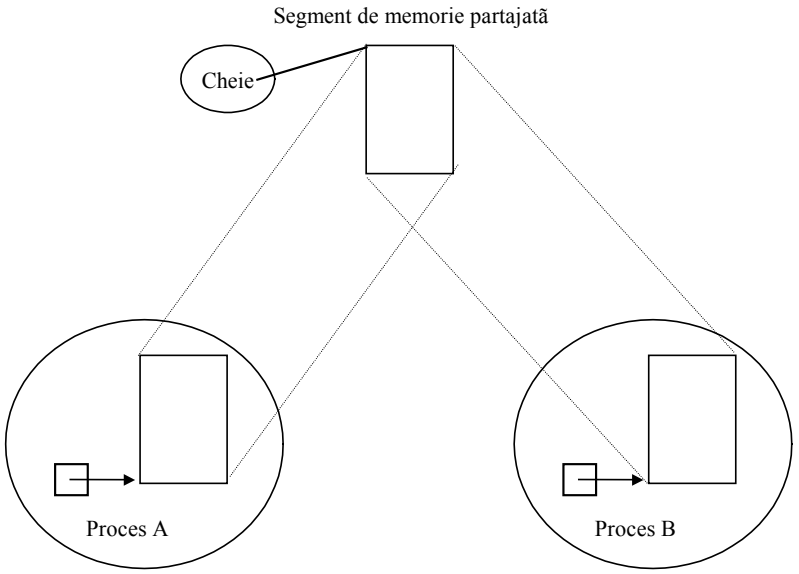

Figura ? Mecanismul de memorie partajată

Nucleul întreține, pentru fiecare segment de memorie partajată, următoarea structură:

```
struct shmid_ds {
  struct ipc_perm shm_perm;   // operatii permise
  int shm_segz;  //lungime segment de memorie partajata
```

```
  ushort shm_cpid;      // pid proces creator
  ushort shm_opid;      // pid ultim proces operator
  ulong shm_nattch;     // numarul curent de atasari
  time_t shm_atime;     // momentul ultimei atasari
  time_t shm_dtime;     // momentul ultimei detasari
  time_t shm_ctime;     // momentul ultimei modificari
};
```

Nucleul Unix impune o serie de limitări numerice privind memoria partajată:

- maximum 131072 octeți / segment
- minimum 1 octet / segment
- maximum 100 zone partajate / sistem
- maximum 6 zone partajate / proces

1.2.2.2 Creare, deschidere, ataşare, control

Crearea unui segment şi **deschiderea** accesului la un segment de memorie partajată se face cu apelul sistem shmget. Prototipul apelului este:

```
int shmget (key_t cheie, int lungime,int flag);
```

Funcția întoarce un număr întreg, descriptor al segmentului, pe care-l vom nota în cele ce urmează cu `shmid`. În caz de eroare întoarce valoarea -1 şi poziționează variabila `errno`.

`cheie` este cheia de acces la zona partajată.

`lungime` este, în cazul creării, dimensiunea în octeți a segmentului; la deschiderea unui segment existent se pune 0.

`flag` indică, pentru creator, constanta IPC_CREAT combinată cu drepturile de acces. Pentru deschiderea accesului la un segment existent, se pune 0, iar sistemul de operare controlează dacă accesul este permis, confruntând drepturile specificate de creator cu drepturile de acces ale procesului (proprietar de segment, membru al grupului, alții).

Ataşarea unui unui proces la un segment de memorie partajată se face prin apelul sistem `shmat`, care are prototipul:

char* shmat (int shmid, char* adresmem, int flag);

Funcția întoarce adresa segmentului de memorie partajată indicat prin descriptorul `shmid`.

Dacă `adresmem != 0`, atunci adresa segmentului va fi rotunjită la un anumit multiplu de octeți. In general, acest argument are valoarea 0.

Argumentul `flag` indică dreptul de acces la segment, astfel:
- dacă `flag` == 0, se permite acces atât în citire cât şi în scriere;
- dacă `flag` == SHM_RDONLY, atunci se permite numai accesul în citire.

Detaşarea unui proces de la un segment de memorie se face prin apelul:

```
int shmdt (char* adresa);
```

Parametrul adresa fiind cel obținut printr-un apel sistem shmat.

Asupra unui segment de memorie se pot aplica o serie de **controale**, materializate prin efectuarea de operații asupra structurii `shmid_ds` definită mai sus. Pentru aceasta se utilizează apelul sistem:

```
int shmctl (int shmid, int comanda, struct shmid_ds *buf);
```

Prin intermediul ei se pot obține informațiile din câmpurile structurii indicate prin descriptorul `shmid`, se pot modifica unele dintre acest informații sau se poate şterge structura desfiinţând astfel segmentul de memorie partajată.

`comanda` indică o acțiune printr-una dintre valorile:

- IPC_STAT transferă conținutul structurii la adresa buf.
- IPC_SET actualizează structura în conformitate cu conținutul structurii de la adresa buf (dacă procesul are acest drept).
- IPC_RMID şterge segmentul de memorie partajată.
- SHM_LOCK blochează accesul la segmentul de memorie partajată.
- SHM_UNLOCK deblochează accesul.

1.2.2.3 Exemplu: linii care repetă acelaşi caracter

Primul exemplu de utilizare a memoriei partajate este următorul. Se defineşte un segment de memorie partajată care conține un întreg `n` şi un caracter `c`.

Serverul va crea segmentul de memorie partajată, apoi va tipări din când în când linii formate numai din caracterul `c` repetat de `n` ori, după care va schimba caracterul în '`S`' şi va modifica aleator valoarea lui `n`.

Clientul va deschide segmentul şi din când în când va tipări câte o linie de `n` caractere `c`, apoi va schimba caracterul în '`C`' şi aleatr va da o altă valoare pentru `n`.

Programul va consta din cinci texte sursă. Primul dintre ele dă funcția `err_sys.c`, care va fi folosită de multe ori în cele ce urmează. Ea "tratează", într-o manieră "grosolană" apariția unor situații de eroare. Textul ei sursă este dat în programul ?

```
void err_sys (char *c) {
   perror (c);
   exit (1);
}//err_sys.c
```

Programul ? Funcția `err_sys`

Segmentul de memorie partajată este definit prin programul ?, care conține şi citarea headerelor necesare.

```
#include <stdio.h>
#include <errno.h>
   extern int errno;
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
#include "err_sys.c"
//Antet: defineste structura unui segment de memorie partajata
   typedef
      struct segment_shm {
         int n;
         char c;
      } segment_shm;//shm_info
#define CHEIE     (key_t) (1234)
#define LUNGIME   sizeof(struct segment_shm)
//shm_litera.h
```

Programul ? Fişierul header `shm_litera.h`

Sursa serverului este dată de către programul ?.

```
// Creaza un segment de memorie partajata indicat de mp.
// Dupa creare, aleator intre 1 si 10 secunde, ia din segment un
intreg n,
// un caracter c si tipareste o linie in care c apare de n ori
#include "shm_litera.h"
int main () {
   int i, shmd;
   segment_shm *mp;
   if ((shmd = shmget (CHEIE, LUNGIME, IPC_CREAT |0666)) < 0)
      err_sys("Nu se poate crea cu shmget");
   if ((mp = (segment_shm *) shmat (shmd, 0, 0)) == NULL)
      err_sys ("Nu se poate atasa adresa");
// Fixeaza continut implicit pentru segment
   mp->n=1+rand()%80;
   mp->c='S';
// Ciclul de schimbari si tipariri
   for(;;sleep(rand()%3)) {
      for (i = 0; i < mp->n; i++)
         putchar (mp->c);
      putchar ('\n');
      mp->n=1+rand()%80;
      mp->c='S';
   }//for
}//shm_serv_litera.c
```

Programul ? Fişierul sursă `shm_serv_litera.c`

Pentru a putea testa mai uşor serverul, la crearea segmentului nu s-a folosit flagul IPC_EXCL. Aceasta permite recrearea ori de câte ori a segmentului de memorie partajată.

Programul ? prezintă sursa clientului.

```
// Deschide un segment de memorie partajata indicat de mp.
// Dupa deschidere, tipareste o linie cu 'C' de n ori,
```

```
// schimba aleator n, asteapta un timp, apoi reia tiparirea
#include "shm_litera.h"
   main () {
      int i, shmd;
      segment_shm *mp;
      if ((shmd = shmget (CHEIE, 0, 0)) < 0)
         err_sys ("Nu se poate deschide shm");
      if ((mp = (segment_shm *) shmat (shmd, 0, 0)) == NULL)
         err_sys ("Nu se poate atasa adresa");
   //Fixeaza aleator, continut pentru segment si tipareste
      for(;;sleep(rand()%2)) {
         for(i=0;i<mp->n;i++)
            putchar(mp->c);
         putchar('\n');
         mp->n = 1+rand()%50;
         mp->c = 'C';
      }//for
   }//shm_clie_litera.c
```

Programul ? Fişierul sursă `shm_clie_litera.c`

După compilarea clientului şi a serverului, o posibilă execuție paralelă a două procese ar putea apare ca în fig. ? şi fig. ?. Cele două procese au fost lansate (relativ) în acelaşi moment.

Pentru a se obține efecte mai spectaculoase, recomandăm să se creeze mai mulți clienți, fiecare depunând o altă literă în segment. De asemenea, momentele de aşteptare între două accese a fiecărui proces să fie aleatoare. Ce s-ar putea obține? De exemplu linii care să nu fie formate dintr-o aceeaşi literă, deoarece pe timpul tipăririi unei linii s-ar putea ca alt proces să modifice caracterul `c`!

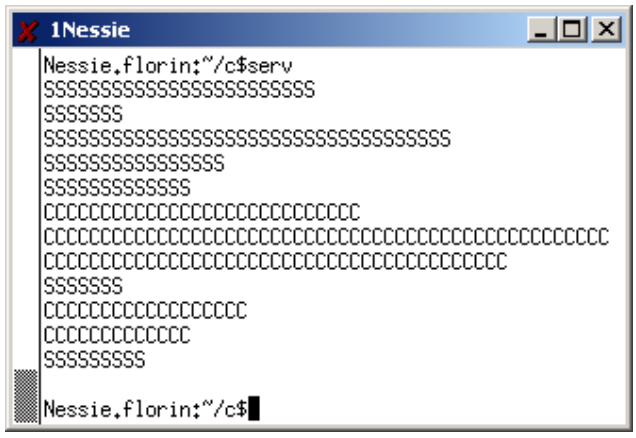

Figura ? O posibilă evoluție a serverului litere

După cum se poate vedea, atât clientul cât şi serverul evoluează indefinit şi ele pot fi oprite doar din exterior. Din această cauză, nu se face nici detaşarea segmentului de la proces, nici distrugerea segmentului de zonă partajată. In mod implicit, la terminarea procesului are loc şi detaşarea. Pentru a nu ține încărcat sistemul de operare cu segmentul de memorie partajată, trebuie rulat programul ? care şterge segmentul din sistem.

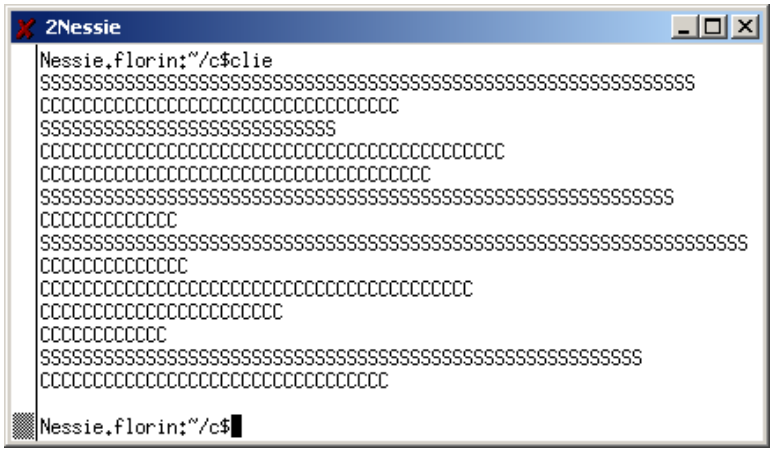

Figura ? O posibilă evoluție a unui client litere

```
// Sterge segmentul de memorie partajata
#include "shm_litera.h"
   int main () {
      int i, shmd;
      segment_shm *mp;
      if ((shmd = shmget (CHEIE, LUNGIME, IPC_CREAT |0666)) < 0)
         err_sys("Nu se poate crea cu shmget");
      shmctl (shmd, IPC_RMID, NULL);
   }//shm_rm_litera.c
```

Programul ? Stergerea unui segment de memorie partajată

1.2.2.4 Problema foii de calcul rezolvată prin memorie partajată

In diversele exemple de aplicații care vor urma, în care sistematic participă mai multe procese, va fi oportun să se poată urmări evoluția în timp a acestor procese. Pentru o abordare uniformă a exemplelor, vom folosi o unică funcție / metodă care să furnizeze, în orice moment, atât informațiile de identificare ale unui proces, cât şi informațiile temporale - indicarea datei şi orei exacte a apelului – pentru proces.

In acest scop vom folosi o funcția descrisă în programul ?.

```
#include <stdio.h>
#include <unistd.h>
#include <time.h>
#define SMAX 100
   char *semnaturaTemporala() {
      static char Faras[SMAX];
      char host[SMAX];
      struct tm  *T;
      time_t      Local;
      Local= time(NULL);
      T =localtime(&Local);
      gethostname(host, SMAX);
      sprintf( Faras, "Host: %s PPID: %d PID: %d Time: %s", host,
getppid(), getpid(), asctime(T));
      return(Faras);
   }//semnaturaTemporala
```

Programul ? Fişierul `semnaturaTemporala.c`

Problema foii de calcul pe care ne-o imaginăm este următoarea. Se defineşte un segment de memorie partajată care conține o foaie de calcul formată din doi vectori a câte patru elemente, ca în fig. ?.

index	s – semnătură	v – valoare
0		v[0]
1		v[1]
2		v[2]
3		v[3]=v[0]+v[1]+v[2]

Figura ? Structura unui segment de memorie partajată

Asupra acestui segment de memorie partajată acționează două categorii de procese: scriitori şi cititori.

Un proces scriitor execută, repetat, de un număr fixat de ori, următorii paşi:

1. generează aleator un număr *i* între 0 şi 2;
2. generează un număr aleator *x* între 1 şi 99;
3. `s[i] = semnaturaTemporala();` // acțiune asupra memoriei partajate;
4. `v[3] = v[3] - v[i] + x;` // acțiune asupra memoriei partajate;
5. `v[i] = x;` // acțiune asupra memoriei partajate;
6. `s[3] = semnaturaTemporala()` // acțiune asupra memoriei partajate;
7. staționează un interval aleator de timp.

Un proces cititor execută, repetat, de un număr fixat de ori, următorii paşi:

1. citeşte din memoria partajată vectorii `s` şi `v` (foaia de calcul);
2. afişează pe ieşirea standard `semnaturaTemporala()` (a cititorului), vectorii `s` şi `v` citiți;
3. semnalează pe ieşirea standard eventualele inconsistențe ale foii de calcul: fie că `s[3]` nu coincide cu nici unul dintre `s[0]`, `s[1]`, `s[2]`, fie că `v[0]+v[1]+v[2]` este diferit de `v[3]`.
4. staționează un interval aleator de timp.

Acest exemplu poate releva obținerea de rezultate inconsistente (vezi [?]) datorate multiaccesului nesincronizat la o aceeaşi locație. Astfel, paşii 3, 4, 5 şi 6 ale unui proces scriitor pot fi executați întrețesut cu aceiaşi paşi ai altui proces scriitor.

Aşa cum vom vedea în secțiunea dedicată semafoarelor, implementarea unei secțiuni critice pentru aceşti paşi va elimina rezultatele inconsistente.

In cele ce urmează vom prezenta o pereche de programe ce descriu scriitorul şi cititorul foii de calcul. Segmentul de memorie partajată este definit în fişierul header din programul ?, care conține şi citarea header-elor necesare.

```
#include <stdio.h>
#include <errno.h>
extern int errno;
```

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include "err_sys.c"
#include "semnaturaTemporala.c"
#define SMAX 1000
#define DSEMN 56
struct foaie_de_calcul {
   char s[4][DSEMN];
   int v[4];
}
#define LUNGIME sizeof(struct foaie_de_calcul)
#define CHEIE (key_t) (2234)
```

Programul ? Fişierul header `memPartFoaie.h`

Fişierul header defineşte foaia de calcul, lungimea acestuia şi defineşte cheia de identificare a segmentului.

Sursa creatorului de segment este dată în programul ?.

```
#include "memPartFoaie.h"

   main () {

      int i,shmd;
      struct foaie_de_calcul *mp;
      char s[SMAX];

      if ((shmd = shmget (CHEIE, LUNGIME, IPC_CREAT |0666)) < 0)
         err_sys("Nu se poate crea cu shmget");

      if((mp=(struct foaie_de_calcul *) shmat (shmd, 0, 0)) == NULL)
         err_sys ("Nu se poate atasa adresa");

      strcpy(s, semnaturaTemporala());
      strcpy(mp->s[0], s);
      strcpy(mp->s[1], s);
      strcpy(mp->s[2], s);
      strcpy(mp->s[3], s);
      mp->v[0] = mp->v[1] = mp->v[2] = mp->v[3] = 0;

      shmdt(mp);
   }
```

Programul ? Sursa `memPartCreareFoaie.c`

Acțiunea scriitorului este descrisă mai sus, iar sursa este prezentată în programul ?.

```
// 0
#include "memPartFoaie.h"
#define NR_ITER 100


   main () {

      int i,it, x, shmd;
      char s[SMAX];
      struct foaie_de_calcul *mp;
      if ((shmd = shmget (CHEIE, 0, 0)) < 0)
         err_sys ("Nu se poate deschide shm");
```

```
      if ((mp = (struct shm_info *) shmat (shmd, 0, 0)) == NULL)
         err_sys ("Nu se poate atasa adresa");

   // 1
      for(it=0;it<NR_ITER;sleep(rand()%2,it++)) {
         i = rand()%3;
         x = 1 + rand()%99;
         strcpy(s, semnaturaTemporala());

      // 2
         strcpy(mp->s[i], s);
         mp->v[3] = mp->v[3] - mp->v[i] + x;
         mp->v[i] = x;
         strcpy(mp->s[3],s);

      // 3
      }
      shmdt(mp);
      shmctl(shmd,IPC_RMID,0);
   // 4
   }
```

Programul ? Sursa `memPartScriitorFoaie.c`

Programul ? prezintă sursa cititor, care acționează conform celor de mai sus.

```
#include "memPartFoaie.h"
#define NR_ITER 50

   main () {

      int i, it,j,x, shmd;
      char s[SMAX];
      struct foaie_de_calcul *mp, *copie;
      if ((shmd = shmget (CHEIE, 0, 0)) < 0)
         err_sys ("Nu se poate deschide shm");

      if ((mp = (struct shm_info *) shmat (shmd, 0, 0)) == NULL)
         err_sys ("Nu se poate atasa adresa");

      copie=(struct foaie_de_calcul* )malloc(sizeof(struct
                                          foaie_de_calcul));
      for(it=0;it<NR_ITER;sleep(rand()%2,it++)) {
         for (j=0;j<4;j++) {
            copie->v[j]=mp->v[j];
            strcpy(copie->s[j],mp->s[j]);
         }

         strcpy(s, semnaturaTemporala());
         x = copie->v[3] - copie->v[0] - copie->v[1] - copie->v[2];
         i = 0;
         if (strcmp(copie->s[3], copie->s[0]) && strcmp(copie->s[3],
             copie->s[1]) && strcmp(copie->s[3], copie->s[2]))
            i= 1;
         printf("citire: %s", s);
         if (i) printf("\tSemnatura total eronata!");
         if (x) printf("\t%d este diferenta la total!",x);
         printf("\n");

         for(i=0; i<4; i++)
            printf("%s\t%d\n", copie->s[i], copie->v[i]);
         printf("\n");
      }
```

```
    shmdt(mp);
}
```

Programul ? Sursa `memPartCititorFoaie.c`

1.2.3 Comunicarea prin cozi de mesaje

1.2.3.1 Flux de octeți

Un *flux de octeți* este un canal unidirecțional de date, de dimensiune limitată, asupra căruia acționează două categorii de procese: scriitori şi cititori. Disciplina de acces la flux este FIFO, iar octeții în care se scrie, respectiv care sunt citiți, avansează în manieră circulară în buffer.

Pentru a înțelege mai exact această abordare, să presupunem că avem de-a face cu un buffer de *n* octeți, numerotați de la 0 la *n*-1. Pentru manevrarea eficientă a fluxului, este necesară întreținerea permanentă a trei parametri:

- *s* poziția curentă în care se poate scrie la următorul acces;
- *c* poziția curentă de unde se poate citi la următorul acces;
- *p* un boolean cu valoarea `true` dacă poziția de citire din buffer se află înaintea poziției de citire, sau valoarea `false` dacă în urma unei citiri sau scrieri s-a depăşit ultimul loc din buffer şi s-a continuat de la începutul lui.

In figurile ?, ?, ?, ?, sunt prezentate diverse ipostaze ale bufferului, condițiile ce trebuie îndeplinite de cei trei parametri şi modificările acestora în urma unei operații. Porțiunea haşurată indică zona octeților încă disponibili în buffer.

Figura ? prezintă bufferul gol fără nici un octet disponibil. Dacă fluxul este gol – nu conține nici un octet pentru citit - atunci procesele cititori aşteaptă până când un proces scriitor depune cel puțin un octet în flux.

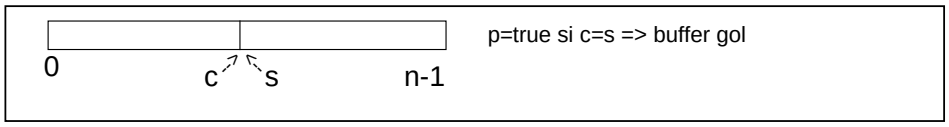

Figura ?Buffer gol

Figura ? prezintă bufferul plin, cu *n* octeți disponibili. Dacă fluxul este plin – nu mai este nici un loc pentru scriere – atunci procesele scriitori aşteaptă până când un cititor extrage cel puțin un octet.

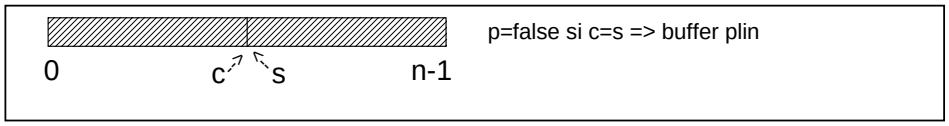

Figura ? Buffer plin

Figura ? prezintă condițiile necesare pentru a se putea scrie *o* octeți în buffer. Prima şi a treia situație provoacă doar avansarea indicelui *s* cu *o*, în timp ce scrierea din a doua situație îl modifică atât pe *s* cât şi pe *p*.

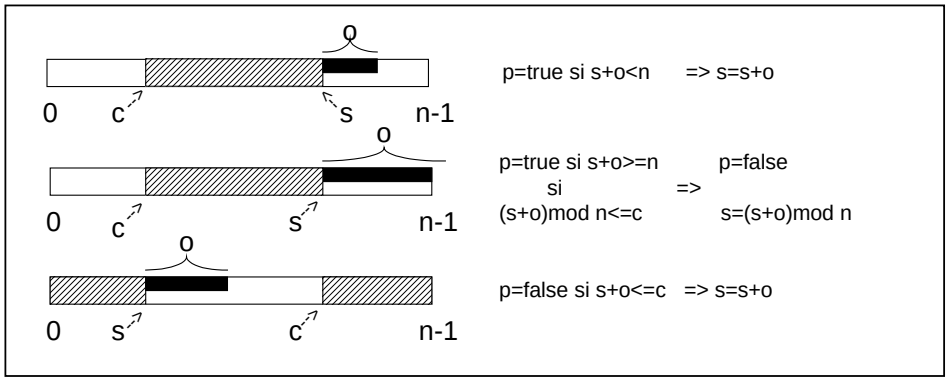

Figura ? Situații de scriere în buffer

In cazul fluxului de octeți, regulile de scriere sunt următoarele:

- Un octet odată scris în flux nu mai poate fi recuperat şi după el se va scrie octetul următor, în scrierea curentă sau în scrierea următoare.
- Un scriitor în flux precizează un număr *o* de octeți pe care doreşte să-i scrie. Dacă în flux mai există cel puțin o poziții libere, atunci scriitorul depune cei *o* octeți în flux. Dacă în flux există doar *r* octeți liberi, $0 < r < o$, atunci se vor depune în flux doar primii *r* octeți dintre cei *o* şi scrierea se consideră terminată. Rămâne la decizia procesului scriitor dacă doreşte sau nu să scrie în flux ceilalți *o-r* octeți printr-o scriere ulterioară.

Figura ? prezintă condițiile necesare pentru a se putea citi următorii *o* octeți. Primele două situații provoacă doar avansarea indicelui *c* cu *o*, în timp ce citirea în ultima situație îl modifică atât pe *c* cât şi pe *p*.

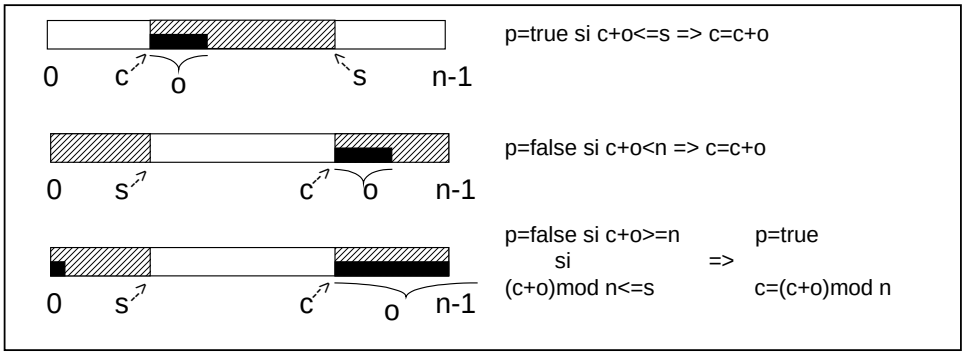

Figura ? Situații de citire din buffer

Regulile de citire din flux sunt următoarele:

- Un octet odată citit este eliminat din flux şi octetul următor este primul care va fi citit, în aceeaşi citire sau în citirea următoare.
- Un cititor din flux precizează un număr *o* de octeți pe care să-i citească. Dacă sunt disponibili cel puțin *o* octeți, atunci cititorul îi primeşte. Dacă în flux există doar *r* octeți, $0 < r < o$, atunci cititorul primeşte numai cei *r* octeți existenți şi citirea se consideră terminată. Rămâne la decizia procesului cititor dacă doreşte sau nu să obțină ceilalți *o-r* octeți printr-o citire ulterioară.

Atât pentru citire, cât şi pentru scriere, mai trebuie precizate încă două condiții şi anume:

- Toate citirile şi scrierile în flux sunt operații atomice: nici un alt proces nu poate întrerupe operația până când fluxul o declară terminată.
- Intre orice două operații consecutive lansate de un proces este posibil ca un alt proces să efectueze o operație asupra fluxului.

Pentru fluxul de octeți se cunosc şi alte denumiri, ca de exemplu *coadă cu lungime limitată, FIFO* [?].

Procesul cititor trebuie să conştientizeze faptul că în condițiile în care la flux sunt ataşați mai mulți cititori şi mai mulți scriitori, nu se poate decide de la ce procese scriitori provin octeții citiți, nici faptul că toți octeții citiți la un moment dat provin de la un proces scriitor sau de la mai mulți.

In acest sens, să considerăm un scenariu posibil. Să presupunem că există 3 procese scriitor A, B, C, care doresc să scrie următorii 9, 5 respectiv 6 octeți:

$a_1a_2a_3a_4a_5a_6a_7a_8a_9$
$b_1b_2b_3b_4b_5$
$c_1c_2c_3c_4c_5c_6$

Presupunem că procesele scriitori îşi țin evidența octeților scrişi efectiv în flux, aşa încât la o a doua scriere va depune în flux începând cu ultimul octet nescris.

Procesul cititor D doreşte să citească 4 octeți, iar procesul cititor E doreşte să citească 13 octeți din flux.

Presupunem că fluxul are capacitatea de 7 octeți.

Proces Doreşte să scrie *l* octeți	Octeți nescrişi	Conținut flux	Octeți citiți	Proces doreşte să citească *l* octeți
A 9	a_8a_9	$a_1a_2a_3a_4a_5a_6a_7$		
B 5 wait	$b_1b_2b_3b_4b_5$			
C 6 wait	$c_1c_2c_3c_4c_5c_6$			
			$a_1a_2a_3a_4a_5a_6a_7$	E 13
				D 4 wait
A 2	-	a_8a_9		
B 5	-	$a_8a_9b_1b_2b_3b_4b_5$		
		$b_3b_4b_5$	$a_8a_9b_1b_2$	D 4
C 6	c_5c_6	$b_3b_4b_5c_1c_2c_3c_4$		
C 2 wait				
			$b_3b_4b_5c_1c_2c_3c_4$	E 6
C 2	-	c_5c_6		

Lăsăm pe seama cititorului să simuleze accesele de mai sus la flux în diverse variante, în care nu toți corespondenții își verifică numărul de octeți schimbați cu fluxul, ci consideră că fiecare operație îi transferă exact atâția octeți câți a dorit să schimbe. De exemplu, în aceste condiții procesul A nu ar mai fi executat o a doua scriere de 2 octeți și ar fi rămas cu ei nescriși.

Acesta este mecanismul de comunicare prin flux de octeți. De regulă, prin intermediul lui se transmit la un moment dat puțini octeți în comparație cu dimensiunea fluxului. De asemenea, frecvențele citirilor și ale scrierilor sunt relativ apropiate. Din aceaste cauze, evoluția "relativ nefirească" exemplificată mai sus apare rar.

1.2.3.2 Schimburi de mesaje

Mecanismele de tip schimb de mesaje diferă esențialmente de fluxrile de octeți prin faptul că un mesaj ajunge la destinație complet, așa cum a cost creat la sursă. Deci nu are loc scriere parțială în canal sau citire parțială din canal, așa cum se petrec lucrurile la fluxul de octeți.

Din punct de vedere structural, un mesaj apare ca în fig. ?.

Destinație	Sursă	Tip mesaj	Conținut mesaj

Figura ? Structura unui mesaj

`Destinație` și `Sursă` sunt adrese reprezentate în conformitate cu convențiile de adresare ale implementării mecanismului de mesaje. `Tip mesaj` dă o caracterizare, în concordanță cu aceleași convenții, asupra naturii mesajului respectiv. Aceste trei câmpuri, eventual completate și cu altele în funcție de specific, formează *antetul mesajului*. Ultimul câmp conține mesajul propriu-zis.

Dacă sistemul de mesaje se adresează factorului uman, atunci informațiile sunt reprezentate sub formă de text ASCII, iar conținutul nu trebuie să aibă o structură rigidă. Dacă sistemul de mesaje se adresează proceselor, atunci câmpurile lui au structuri standardizate, evetual chiar conținut binar. Indiferent de natura corespondenților, scenariul comunicării prin mesaje este același:

- Emițătorul mesajului compune mesajul și îl expediază spre destinatar sau destinatari, după caz. In situația în care corespondenții se află pe același sistem de calcul, expedierea înseamnă depunerea lui într-o zonă specială numită *canal* sau *coadă de mesaje*.
- Sistemul care conține un destinatar recepționează mesajul. Recepția se face numai atunci când sistemul devine operațional. Pe perioada dintre expediere și recepție mesajul este păstrat temporar prin intermediul unui sistem de zone tampon.
- Atunci când procesul (utilizatorul) destinatar devine activ, sistemul îl anunță de primirea mesajului respectiv. De asemenea, fiecare proces activ controlează din când în când căsuța poștală sau canalul spre a sesiza apariția de noi mesaje.
- Atunci când procesul consideră de cuviință, își citește mesajul primit. In funcție de conținutul lui, procesul își poate modifica execuția ulterioară.

In funcție de numărul destinatarilor unui mesaj, se disting:

- Mesaje *unicast,* cu un destinatar unic: corespondență punct la punct.
- Mesaje *multicast,* în care există un grup de destinatari. Ei se stabilesc fie prin specificarea tuturor destinatarilor, fie există o listă predefinită de destinatari la care va fi trimis mesajul.
- Mesaje *broadcast,* în care mesajul este trimis automat tuturor destinatarilor care fac parte dintr-o anumită categorie: utilizatorii dintr-o rețea LAN sau WAN [11], angajații unui compartiment sau companii, proceselor care utilizează o anumită resursă etc.

Schimbul de mesaje este folosit pe scară largă mai ales în sistemele distribuite, deci programarea distribuită operează mai des cu acest mecanism. Există totuşi şi aplicații ce se derulează concurent pe acelaşi sistem şi care comunică între ele prin mesaje specializate.

Sistemele de operare îşi definesc propriile standarde de mesaje, specifice comunicării numai între procese. *Coada de mesaje* este un astfel de exemplu, asupra căruia vom reveni pe larg în cele ce urmează.

1.2.3.3 Cozi de mesaje Unix

O coadă de mesaje este o listă simplu înlănțuită de mesaje memorată în nucleu. Scriitorii şi cititorii cozii nu sunt dependenți temporal unii de alții. Nu există restricții de genul "un proces trebuie să aştepte să-i sosească un anumit mesaj", aşa cum se pot petrece lucrurile la pipe sau FIFO. Este deci posibil ca un proces să scrie nişte mesaje într-o coadă şi apoi să se termine. Este eventual posibil ca mai târziu un alt proces să le citească.

Pentru fiecare coadă de mesaje din sistem, nucleul întreține structura de informații `msgid_ds`, prezentată în continuare:

```
struct msgid_ds {
  struct ipc_perm msg_perm;// structura de drepturi a cozii
  struct msg *msg_first;   // pointer la prim mesaj din coada
  struct msg *msg_last;    // pointer la ultim mesaj din coada
  ushort msg_cbytes;       // nr. curent de octeți din coada
  ushort msg_gnum;         // nr. curent de mesaje din coada
  ushort msg_gbytes;       // nr. maxim octeți perm. in coada
  ushort msg_lspid;        // pid-ul ultimului mesaj emis
  ushort msg_lrpid;        // pid-ul ultimului mesaj receptat
  time_t msg_stime;        // ora ultimului mesaj emis
  time_t msg_rtime;        // ora ultimului mesaj receptat
  time_t msg_ctime;        // ora ultimei schimbari
};
```

Orice mesaj dintr-o coadă începe cu un întreg strict pozitiv lung care indică tipul mesajului, urmat de un şir nevid de octeți care reprezintă partea de date a mesajului. Lungimea acesteia este gestionată de către nucleu. Structura unui mesaj este dată în figura ?.

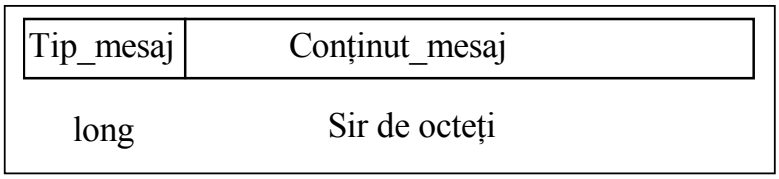

Figura ? Structura unui mesaj Unix

In figura ? este ilustrată o coadă care conține trei mesaje, unul de tip 200, iar celelalte două de tip 100.

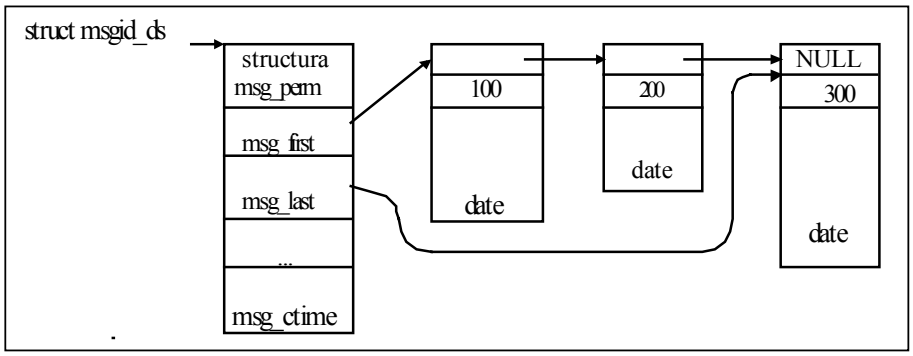

Figura ? O coadă de mesaje Unix

Ca şi la celelalte IPC-uri de sub Unix, există anumite limite, care eventual pot fi modificate de către administrator:

- 2048 numărul maxim de octeți/mesaj
- 4096 numărul maxim de octeți a unei cozi de mesaje
- 50 numărul maxim de cozi de mesaje admise în sistem
- 40 numărul maxim de mesaje

1.2.3.4 Acces şi operații asupra cozii

Crearea unei cozi noi şi *deschiderea* unei cozi existente se face folosind apelul sistem `msgget()`. Prototipul acesteia este:

```
int msgget (key_t cheie, int msgflag);
```

Funcția întoarce un întreg pozitiv, *descriptorul* cozii, pe care-l vom numi msgid. Prin el va fi identificată o coada de mesaje. In caz de eşec va întoarce valoarea –1.

`cheie` reprezintă cheia de identificare a IPC coada de mesaje, conform celor prezentate mai sus.

Pentru deschiderea unei cozi existente, `msflag` va avea valoarea 0. Pentru creare, `msflag` va avea valoarea:

```
IPC_CREAT | IPC_EXCL | drepturi_de_acces
```

Depunerea unui mesaj în coadă se face folosind apelul `msgsnd`, care are prototipul:

```
int msgsnd (int msgid, struct msgbuf *p, int l, int flag);
```

Intoarce 0 în caz de succes sau -1 în caz de eroare.

`msgid` este descriptorul cozii;

`p` este pointer spre mesajul de introdus. Structura msgbuf se defineşte astfel:

```
struct msgbuf {
  long mtype;     // tipul mesajului, numar strict pozitiv
  char mtext[1];  // tablou cu datele mesajului
};
```

Câmpul de tip al mesajului (`mtype`) trebuie completat înainte de introducerea în coadă.

În tabloul `mtext` utilizatorul poate pune orice configurație de octeți, deoarece nucleul nu controlează conținutul lui `mtext`.

`l` conține lungimea efectivă în octeți a mesajului.

`flag` are de cele mai multe ori valoarea 0. Uneori, când situațuia o cere, valoarea lui poate fi IPC_NOWAIT, cu efect de revenire imediată dacă nu este spațiu în coada de mesaje.

Citirea unui mesaj din coadă se face folosind apelul sistem `msgrcv`, al cărei prototip este:

```
int msgrcv(int msgid,struct msgbuf *p,int l,
                        int tipmesaj, int flag);
```

Funcția întoarce lungimea efectivă a conținutului mesajului.
p indică o zonă de memorie unde se va depune mesajul recepționat.
l indică dimensiunea maximă admisă a conținutului mesajului (indicat de p).
tipmesaj indică tipul mesajului care se doreşte a fi recepționat, astfel:

- `tipmesaj` == 0 înseamnă citirea primului mesaj din coadă;
- `tipmesaj` > 0 primul mesaj din coadă care este de tip tipmesaj
- `tipmesaj` < 0 primul mesaj din coadă care are tipul minim dar cel puțin egal cu -tipmesaj

flag indică ce se va face dacă mesajul cerut nu este în coadă. De cele mai multe ori are valoarea 0 şi indică aşteaptarea apariției mesajului solicitat. Dacă valoarea lui este IPC_NOWAIT, atunci nu se aşteaptă, ci se revine şi se semnalează eroare.

Controlul cozii se face folosind apelul sistem msgctl, al cărei prototip este:

```
int msgctl (int msgid, int comanda, struct msgid_ds *buffer);
```

Intoarce 0 în caz de succes şi -1 în caz de eroare.

`msgid` este descriptorul cozii de mesaje;

`buffer` este o structură martor a descriptorului cozii. Ea este folosită numai când parametrul comanda are anumite valori specifice.

`comanda` indică operația cerută. Valorile posibile sunt:
- IPC_STAT extrage structura descriptor a cozii în zona punctată de `buffer`.
- IPC_SET actualizează structura descriptor a cozii cu informațiile existente, în zona punctată de `buffer`.
- IPC_RMID şterge o coadă de mesaje din sistem.

1.2.3.5 Un exemplu de utilizare a cozilor de mesaje

Pentru a ilustra comunicarea prin schimburi de mesaje, propunem următoarea problemă. Se defineşte o coadă de mesaje care depozitează mesaje sub formă de stringuri. Deoarece sub Unix este nevoie de definirea unui tip de mesaj, convenim ca tipul să fie stabilit de către scriitorul mesajelor după primul caracter mesajului:
- tip = 1 dacă primul caracter este literă mică;
- tip = 2 dacă primul caracter este literă mare;
- tip = 3 dacă primul caracter este cifră zecimală;
- tip = 4 în celelalte cazuri.

O primă categorie de procese este formată din *scriitorii* în coada de mesaje. Acțiunile unui astfel de scriitor se desfăşoară în mod repetat, la fiecare iterație efectuând următoarele operații:
- citeşte o linie de la intrarea lui standard şi fixează tipul, ca mai sus;
- compune un mesaj format din `semnaturaTemporala()` (am definit la memoria partajată această funcție), urmată de linia citită;
- scrie mesajul în coadă;
- dacă linia citită reprezintă string-ul STOP, atunci îşi încheie execuția, altfel stă în aşteptare un interval aleator de timp.

A doua categorie de procese este constituită din *cititorii* de mesaje. Aceştia extrag, în mod repetat mesaje din coadă. La fiecare extragere execută următoarele acțiuni:
- citeşte un mesaj din coadă;
- adaugă la mesajul string citit propria lui `semnaturaTemporala();`
- afişează linia rezultată la ieşirea lui standard;
- dacă mesajul citit este stringul "STOP", atunci îşi încheie activitatea şi, eventual dacă platforma îi permite, închide şi distruge coada.

Implementarea constă în 5 programe scurte:
1. creatorul cozii;
2. scriitorul de mesaje în coadă;
3. cititorul primului mesaj din coadă;
4. cititorul primului mesaj care începe cu literă;
5. distrugătorul cozii.

Structura unui mesaj din coadă, este definită în fişierul header din programul ?.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include "err_sys.c"
#define DSEMN 56
#define SMAX 100
#define KEY 12345
typedef struct {        //definirea structurii
   long tip;
   char semn[DSEMN];    //care va fi trimisa prin coada
   char linie[SMAX];
} Mesaj;
```

Programul ? Fişierul header `antetCoadaMesaje.h`

In continuare, introducem sursele celor 5 programe programe, descrise pe scurt mai sus.

Programul ? creează coada de mesaje, după care îşi incheie execuția.

```
#include "antetCoadaMesaje.h"
main(){
   printf("\nCream coada de mesaje\n");
   if(msgget(KEY,IPC_CREAT|0666)<0) //crearea cozii
      err_sys("Eroare la crearea cozii de mesaje:");
}
```

Programul ? Sursa `coadaCreare.c`

Programul ? scrie mesaje in coadă, respectând specificațiile de mai sus.

```
#include "antetCoadaMesaje.h"
  #include "semnaturaTemporala.c"

   int coadaScriitor() {
      char buf[SMAX];
      int coadaId;
      Mesaj mesg;

      if ((coadaId=msgget(KEY,0))<0)
         err_sys("Eroare la obtinerea ID coada\n");
      for (;;) {
         gets(buf);
         if ((buf[0]>='a') && (buf[0]<='z'))
            mesg.tip=1;
         else
            if ((buf[0]>='A') && (buf[0]<='Z'))
               mesg.tip=2;
            else
               if ((buf[0]>='0') && (buf[0]<='9'))
                  mesg.tip=3;
               else
                  mesg.tip=4;
         strcpy(mesg.semn,semnaturaTemporala());
         strcpy(mesg.linie,buf);
         if(msgsnd(coadaId,&mesg,sizeof(Mesaj),0)<0)
            err_sys("Eroare la trimiterea mesajului:");
         if (!strcmp(buf,"STOP")) {
            printf("S-a citit STOP.\n");
            break;
         }
         sleep(rand()%10);
      }
```

```
   }

   main() {
      coadaScriitor();
   }
```

Programul ? Sursa `coadaScriitor.c`

Programul ? citeşte mesaje din coadă, în ordinea FIFO (pentru a obține mesajele din coadă în această ordine, se specifică valoarea 0 în al patrulea argument al apelului `msgrcv`).

```
#include "antetCoadaMesaje.h"
#include "semnaturaTemporala.c"

   int coadaCititor() {
      char buf[SMAX];
      int coadaId;
      Mesaj mesg;

      if ((coadaId=msgget(KEY,0))<0)
         err_sys("Eroare la obtinerea ID coada\n");
      for (;;) {
         if(msgrcv(coadaId,&mesg,sizeof(Mesaj),0,0)<0)
            err_sys("Eroare la receptionarea mesajului.");
         if (!strcmp(mesg.linie,"STOP")) {
            if (msgctl(coadaId,IPC_RMID,0)<0)
               err_sys("Eroare la stergerea cozii de mesaje.");
            else {
               printf("S-a receptionat STOP, coada stearsa.\n");
               break;
            }
         }
         else
            printf("|%s|%s|%s|\n",mesg.semn, mesg.linie,
                  semnaturaTemporala());
      }
   }

   main() {
      coadaCititor();
   }
```

Programul ? Sursa `coadaCititorFirst.c`

Programul care extrage din coadă un mesaj de un anumit tip, diferă de programul anterior prin faptul că procedura `coadaCititor` primeşte un parametru de tip `long`, reprezentând tipul mesajului solicitat. Acest tip se transmite ca al patrulea parametru, la apelul funcției `msgrcv`.

Invităm cititorul să simuleze, cu ajutorul cozilor de mesaje, comunicarea de tip poşta electronică (pe aceeaşi maşină). Se consideră că, în acest caz, un mesaj va avea următoarea structură:

```
typedef struct {
   int uid; //poate identifica useruldestinatar al  mesajului
   char mesaj[100]; //continutul mesajului
} Mesaj;
```

1.3 Semafoare

1.3.1 Conceptul de semafor

Conceptul de *semafor* a fost introdus de Dijkstra, pentru a facilita sincronizarea proceselor, prin protejarea secțiunilor critice şi asigurarea accesului exclusiv la resursele pe care procesele le accesează.

Formal, un semafor se defineşte ca o pereche *(v(s),c(s))* unde *v(s)* este valoarea semaforului iar *c(s)* o coadă de aşteptare la semafor. Valoarea v(s) este un număr întreg care primeşte o valoare inițială $v_0(s)$ iar coada de aşteptare conține referințe la procesele care aşteaptă la semaforul s. Inițial coada este vidă, iar disciplina cozii depinde de sistemul de operare (LIFO, FIFO, priorități, etc.)

Pentru gestiunea semafoarelor, se definesc două operații indivizibile, notate: ***P(s)*** şi ***V(s)***. Operația ***P(s)***, executată de un proces ***A***, are ca efect decrementarea valorii semaforului s cu o unitate şi obținerea accesului la secțiunea critică, pe care acesta o protejează. Operația ***V(s)*** executată de procesul ***A*** realizează incrementarea cu o unitate a valorii semaforului s şi eliberarea resursei blocate. În unele lucrări [?], aceste primitve sunt denumite "WAIT" respectiv "SIGNAL". In figura ? şi figura ? sunt prezentate definițiile pseudocod ale celor două operații:

```
v(s)=v(s)-1;
if v(s) < 0 then
  begin
    STARE(A):=WAIT;
    c(s)<==A;          {procesulA intră în aşteptare}
    Se trece controlul la DISPECER
  end
else
  Se trece controlul la procesul A;
endif
```

```
v(s)=v(s)+1;
if v(s) <= 0 then
  begin
    c(s) ==> B;{se scoate din coadă un alt proces B}
    STARE(B):=READY;
    Se trece controlul la DISPECER
  end
else
  Se trece controlul la procesul A;
endif
```

Figura ? Operația *P(s)* executată către procesul *A*

Figura ? Operația *V(s)* executată către procesul *A*

1.3.2 Semafoare Unix

In sistemul de operare Unix mecanismul semafor este implementat de către nucleul Unix. Acesată implementare generalizează noțiunea clasică de semafor. Pe lângă fişierele header cunoscute, pentru semafoare se mai foloseşte headerul `<sys/sem.h>`.

1.3.2.1 Structura setului de semafoare Unix

Astfel, entitatea de lucru nu este semaforul, ci *setul de semafoare*. De asemenea, asupra unui set de semafoare pot fi efectuate mai multe operații simultan, iar valoarea de incrementare / decrementare poate fi mai mare ca 1. Toate operațiile rămân indivizibile, deci procesul execută toate operațiile cerute sau nici una.

Orice set de semafoare din sistem are asociată o *cheie de identificare,* aşa cum am descris în secțiunea despre IPC. Conform principiilor IPC de sub Unix, nucleul întreține, pentru un set de semafoare, următoarea structură de date:

```
struct semid_ds {
  struct ipc_perm sem_perm;   // operatiile permise
  struct sem      *sem_base;  // pointer la primul semafor din set
  ushort          sem_nsems;  // numarul de semafoare din set
  time_t          sem_otime;  // momentul ultimei operatii
  time_t          sem_ctime;  // momentul ultimei schimbari
};
```

Orice membru al unui set de semafoare are structura de mai jos:

```
struct sem {
  ushort semval;  // valoarea semaforului
  short sempid;   // pid-ul ultimei operatii
  ushort semncnt; // numar de procese in asteptare pana cand
                  // semval depaseste o valoare specificata
  ushort semzcnt; // numar de procese în asteptare pana cand
                  // semval ajunge la valoarea zero
};
```

De exemplu, dacă în set există două semafoare, atunci structura setului de semafoare este cea din figura ?.

Ca şi la memoria partajată şi la cozile de mesaje, avem o serie de limitări numerice. Principalele valori ale acestora sunt:

valoarea maximă a unui semafor: 32767;
nr. maxim de seturi de semafoare 10;
nr. maxim semafoare în sistem 60;
nr. maxim semafoare pe set 25;
nr. maxim de operații pe apel `semop` 10.

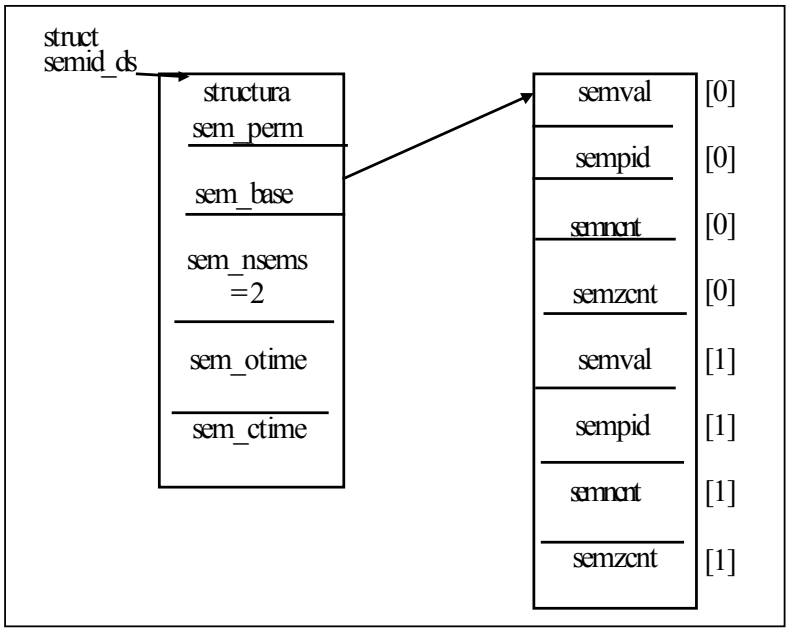

Figura ? Un set de două semafoare

1.3.2.2 Crearea unui set şi accesul la un set existent

Crearea unui set de semafoare sau accesarea unui set deja existent se face folosind apelul sistem `semget`, care atre prototipul:

```
int semget (key_t cheie, int nsemaf, int semflag);
```

Valoarea întoarsă este descriptorul grupului de semafoare pe care-l vom nota `semid` sau -1 în caz de eroare.

`nsemaf` indică, în cazul creării, numărul de semafoare din set. In cazul deschiderii unui grup deja creat, valoarea specificată este 0.

`semflag` conține flagurile deschiderii. În cazul creării se specifică faptul că este vorba de creare, eventual creare exclusivă (spre a se evita dubla creare) şi drepturile de acces la IPC-ul semafor (date prin trei cifre octale [?]). Valoarea constantei este:

```
IPC_CREAT |  [ IPC_EXCL ] | drepturi_de-acces
```

In cazul deschiderii unui set deja existent, valoarea constantei este 0.

1.3.2.3 Operații asupra semafoarelor Unix

Toate operațiile executate asupra unui set de semafoare sunt efectuate prin intermediul apelului sistem `semop`. Operațiile asupra valorilor semafoarelor din set sunt atomice. Prototipul apelului este:

```
int semop (int semid, struct sembuf* pointoper,
                              unsigned int noper);
```

Apelul întoarce 0 în caz de succes sau -1 în caz de eşec al apelului, caz în care poziționează variabila `errno`.

`semid` este întregul descriptor al setului de semafoare.

`noper` este numărul semafoarelor din set care urmează a fi modificate.

`pointoper` punctează spre un tablou cu elemente care indică modificările ce urmează a fi făcute. Un element modificator al acestui tablou are structura:

```
struct sembuf {
  ushort sem_num; // numarul semaforului din set
  short sem_op;   // operatia asupra semaforului, in concordanta
                  // cu valoarea sem_flg de la deschidere / creare
  short sem_flg;  // flagul operatiei: este fie 0 fie IPC_NOWAIT
};
```

Există trei acțiuni posibile asupra unui semafor, în funcție de valorile corespunzătoare ale sem_op, semval şi sem_flag, astfel:

1. Dacă (`sem_op`>0) sau ((`sem_op`<0) şi (`semval`>=-`sem_op`)) atunci `semval=semval+sem_op`

2. Dacă ((`sem_op`<0) şi (`semval`<-`sem_op`) şi (`sem_flag`==0)) atunci aşteaptă până când (`semval`>=-`sem_op`)

3. Dacă ((`sem_op`==0) şi (`semval`>0) şi (`sem_flag`==0)) atunci asteaptă până când (`semval`==0)

De fapt, avem de-a face fie cu creşterea / diminuarea valorii semaforului, fie cu aşteptarea ca valoarea semaforului să depăşească o valoare dată, fie să ajungă la valoarea zero. In celelalte cazuri (în special când `sem_flag` == IPC_NOWAIT), semop întoarce valoarea -1 şi poziționează `errno`.

1.3.2.4 Controlul seturilor de semafoare

Operațiile de control asupra unui set de semafoare sunt efectuate cu ajutorul apelului sistem `semctl`. Acesta are prototipul:

```
int semctl (int semid, int nrsem, int cmd, union semun arg);
```

Intoarce valoarea solicitată, cu excepția cazului GETALL când întoarce 0 sau la eroare când întoarce valoarea -1.

`semid` este întregul descriptor al setului de semafoare.

`nrsem` este numărul semaforului din set ale cărui informații se cer.

`arg` este parametrul care completează specificarea parametrului `cmd`. El are descrierea:

```
union semun {
  int val;                  // pentru SETVAL
  struct semid_ds *buff;    // pentru IPC_STAT si IPC_SET
  ushort *array;            // pentru GETALL si SETALL
}
```

`cmd` este comanda care indică operația dorită şi ea poate avea valorile:

- GETVAL întoarce valoarea `semval` a semaforului `nrsem`;
- SETVAL atribuie valorii `semval` a semaforului `nrsem` valoarea `arg.val`;
- GETPID întoarce valoarea `sempid` a semaforului `nrsem`;
- GETNCNT întoarce valoarea `semncnt` a semaforului `nrsem`;
- GETZCNT întoarce valoarea `semzcnt` a semaforului `nrsem`;
- GETALL întoarce valorile `semval` ale tuturor semafoarelor din set în `arg.array`;
- SETALL atribuie valorilor `semval` ale tuturor semafoarelor din set valorile corespunzătoare din arg.array;
- IPC_STAT depune valorile structuii `semid_ds` ale setului în `arg.buf`
- IPC_SET se actualizează valorile `sem_perm.uid`, `sem_perm.gid` şi `sem_perm.mode` ale setului cu cele din `arg.buf`.
- IPC_RMID şterge setul de semafoare din sistem.

1.3.3 Exemple de utilizare a semafoarelor Unix

1.3.3.1 Semafor binar şi secțiune critică

Un prim exemplu este definirea unui semafor binar care poate fi folosit pentru definirea de secțiuni critice, aşa cum am precizatmai înainte. Programul ? prezintă textul sursă `semaforBinar.c` care defineşte patru funcții: `creareSemBin()`, `stergereSemBin()`, `blocare()`, `deblocare()`.

```
# include <sys/types.h>
# include <sys/ipc.h>
# include "sys/sem.h"
# define KEY_SEM 3234  //cheile semafoarelor
//# include "err_sys.c"
   int semId; //identificator semafor

   void Operatie(int SemId, int op){
      struct sembuf SemBuf;
      SemBuf.sem_num=0;
      SemBuf.sem_op=op;
      SemBuf.sem_flg=0;
      if(semop(SemId,&SemBuf,1)<0)
         perror("Eroare la semop:");
   }
//operatia P
   void P(int SemId){
      Operatie(SemId,-1);
```

```
   }
//operatia V
   void V(int SemId){
      Operatie(SemId,1);
   }

   int creareSemBin() {
   //creare semafor
      if ((semId=semget(KEY_SEM,1,0666|IPC_CREAT))<0)
         err_sys("Eroare la crearea semaforului.\n");

   //initializare semafor cu valoarea 1
      if (semctl(semId,0,SETVAL,1)<0)
         err_sys("Eroare la initializarea semaforului.\n");
      return 0;
   }

   int stergereSemBin() {
   //stergere semafor
      if (semctl(semId,0,IPC_RMID,1)<0)
         err_sys("Eroare la stergerea semaforului.\n");
      return 0;
   }

//operatia de blocare
   void blocare() {
      P(semId);
   }

//operatia de deblocare
   void deblocare() {
      V(semId);
   }
```

Programul ? Fişierul header `semaforBinar.h`

Pentru utilizarea într-un program trebuie ca în faza de inițializare să se creeze semaforul, apoi să se folosească după caz blocarea şi deblocarea, iar la sfârşit să se distrugă semaforul.

De exemplu, dacă dorim să utilizăm acest semafor la protejarea foii de calcul din dată ca exemplu la memoria partajată, vom înlocui liniile comentate cu apeluri definite în fişierul antet `semaforBinar.h`, astfel:

```
// 0  ->    #include"semaforBinar.h"
// 1  ->    creareSemBin();
// 2  ->    blocare();
// 3  ->    deblocare();
// 4  ->    stergereSemBin();
```

Figura ? Aduăgarea apelurilor de protejare pentru foaia de calcul

1.3.3.2 Problema producătorului şi consumatorului

Ca un al doilea exemplu de folosire a semafoarelor am ales problema clasică Producător / Consumator, enunțată şi modelată înteoria generală pe care o vom implementa, respectând modelul teoretic.

Buferul accesat concurent de către producători şi consumatori reprezintă o resură *fifo*, de dimensiune `DimBuf`. Protejarea resursei comune se realizează folosind cele trei semafoare: `gol`, `plin` şi `mutex`, cu semnificația din soluția teoretică a problemei.

Implementarea problemei Producător - Consumator conține un fisier header (programul ?) şi trei programe sursă: programul care creează semafoarele folosite în exemplu (programul ?), programul producător (?) şi programul consumator (?).

Comentariile din sursele programelor conțin explicații suplimentare privind funcționalitatea acestora.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include "semaforBinar.h"
#define KEY_GOL 1234
#define KEY_PLIN 1236
#define KEY_MUTEX 1235
#define DimBuf 10
#define TRUE 1
#define NumeFifo "FiFoProdCons"
```

Programul ? Fişierul header `antetProdCons.h`

```
#include "antetProdCons.h"
#include "err_sys.c"

   main(){

      int gol,plin,mutex;
   //cream semaforul "gol"
      if ((gol=semget(KEY_GOL,1,0666|IPC_CREAT))<0)
         err_sys("Eroare la crearea semaforului \"gol\".");

   //initializam semaforul "gol"
      if (semctl(gol,0,SETVAL,DimBuf)<0)
         err_sys("Eroare la initializarea semaforului \"gol\".");

   //cream semaforul "plin"
      if ((plin=semget(KEY_PLIN,1,0666|IPC_CREAT))<0)
         err_sys("Eroare la crearea semaforului \"plin\".");

   //initializare cu val 0
      if (semctl(plin,0,SETVAL,0)<0)
         err_sys ("Eroare la initializarea semaforului \"plin\".");

   //cream semaforul "mutex"
      if ((mutex=semget(KEY_MUTEX,1,0666|IPC_CREAT))<0)
         err_sys("Eroare la crearea semaforului \"mutex\".");

   //initializare cu val 1
      if (semctl(mutex,0,SETVAL,1)<0)
         err_sys ("Eroare la initializarea semaforului \"mutex\".");
   }
```

Programul ? Programul `initSemProdCons.c`

```
#include "antetProdCons.h"
```

```
#include "err_sys.c"
   int gol,plin,mutex,fw;

//creeza obiectul "obiect"
   int creeaza(int* obiect) {
      static int i=1;
      *obiect=i++;
      return 1;
   }

//depune "obiect" in fifo
   int depune(int obiect) {
      printf("Producatorul depune obiectul: %d\n",obiect);
      write(fw,&obiect,sizeof(int));
   //la al 30-lea obiect, producatorul isi incheie executia
      if ((obiect%30)==0) {
         if (semctl(plin,0,IPC_RMID,1)<0)
            err_sys("Eroare la stergerea semaforului plin.\n");
         if (semctl(gol,0,IPC_RMID,1)<0)
            err_sys("Eroare la stergerea semaforului gol.\n");
         if (semctl(mutex,0,IPC_RMID,1)<0)
            err_sys("Eroare la stergerea semaforului mutex.\n");
         close(fw);
         exit(1);
      }
      sleep(rand()%3);
      return 1;
   }

//rutina producator
   void Producator() {
      int obiect;
      printf("Procesul producator isi inceapa activitatea...\n");
      while (TRUE) {
         creeaza(&obiect);     //produce un nou obiect obiect
         P(gol);               //decrementeaza semaforul gol
         P(mutex);             //intra în regiunea critica
         depune(obiect);       //introduce obiectul obiect în
         V(mutex);             //iese din regiunea critica
         V(plin);              //incrementeaza semaforul plin
      }
   }

   main(){
      if ((fw=open(NumeFifo,O_WRONLY))<0)
         err_sys("Eroare la deschiderea fifo pentru scriere.\n");
      if((gol=semget(KEY_GOL,0,0))<0)
         err_sys("Eroare la obtinere Id semafor \"gol\".");
      if((plin=semget(KEY_PLIN,0,0))<0)
         err_sys("Eroare la obtinere Id semafor \"plin\".");
      if((mutex=semget(KEY_MUTEX,0,0))<0)
         err_sys("Eroare la obtinere Id semafor \"mutex\".");
      Producator();
   }
```

Programul ? Programul `producator.c`

```
#include "antetProdCons.h"
#include "err_sys.c"

   int gol,plin,mutex,fr;

//extrage obiectul "obiect" din fifo
```

```
   int extrage(int* obiect) {
      read(fr,obiect,sizeof(int));
      return 1;
   }

//afiseaza obiectul extras
   int consuma(int obiect) {
      printf("Consumatorul consuma obiectul: %d\n",obiect);
      if ((obiect%30)==0) {
         close(fr);
         exit(1);
      }
      sleep(rand()%3);
      return 1;
   }

//rutina consumator
   void Consumator() {
      int obiect;
      printf("Procesul consumator isi inceapa activitatea...\n");
      while (TRUE) {
         P(plin);              //decrementeaza semaforul plin
         P(mutex);             //intra în regiunea critica
         extrage(&obiect);     //scoate un obiect din recipient
         V(mutex);             //iese din regiunea critica
         V(gol);               //incrementeaza semaforul gol
         consuma(obiect);      //consuma obiectul
      }
   }

   main(){
      if ((fr=open(NumeFifo,O_RDONLY))<0)
         err_sys("Eroare la deschiderea fifo pentru citire.\n");
      if((gol=semget(KEY_GOL,0,0))<0)
         err_sys("Eroare la obtinere Id semafor \"gol\".");
      if((plin=semget(KEY_PLIN,0,0))<0)
         err_sys("Eroare la obtinere Id semafor \"plin\".");
      if((mutex=semget(KEY_MUTEX,0,0))<0)
         err_sys("Eroare la obtinere Id semafor \"mutex\".");
      Consumator();
   }
```

Programul ? Programul `consumator.c`

1.3.3.3 Utilizarea semafoarelor la o foaie de calcul tabelară

Acest exemplu este preluat din [?] şi utilizează memoria partajată şi semafoarele la o foaie de calcul. Să ne imaginăm o foaie de calcul dreptunghiulară de *m X n* elemente, în care fiecare element de pe ultima linie, cu excepția ultimului, conține totalul celorlalte elemente de pe coloana lui. Fiecare element de pe ultima coloană, cu excepția ultimului, conține suma celorlalte elemente de pe linia respectivă.

Asupra acestei foi acționează, în exemplul nostru la intervale aleatoare de timp, două procese: `modifica` şi `verifica`. Procesul `modifica` selectează aleator o linie şi o coloană din tabel. În această celulă a tabelului schimbă valoarea cu una nouă, după care reactualizează totalurile de pe linie şi de pe coloană.

Procesul `verifica` afişează periodic foaia de calcul şi verifică totalurile. În caz de neconcordanță afişează mesaje de eroare adecvate.

Merită să studiem puțin "granularitatea" partajării acestor servicii. Cel mai simplu mod de lucru este acela de a folosi un singur semafor, aşa încât la un moment dat asupra foii de calcul are acces un singur proces.

La o studiere mai atentă, se poate constata că această restricție poate fi relaxată simțitor. De exemplu, dacă `modifica` acționează asupra celulei `(i,j)`, atunci este necesară protecția numai a totalurilor de pe linia `i` şi de pe coloana `j`. Dacă procesul `verifica` controlează linia `k`, atunci trebuie protejată doar linia respectivă.

Pentru aceasta, sunt necesare $m + n$ semafoare, câte unul pentru fiecare linie şi fiecare coloană. Un proces de modificare va bloca o linie şi o coloană, iar un proces de control o linie sau o coloană. În fig. ? este prezentat ansamblul foii de calcul şi a semafoarelor aferente, în cazul în care acționează cele două procese descrise mai sus.

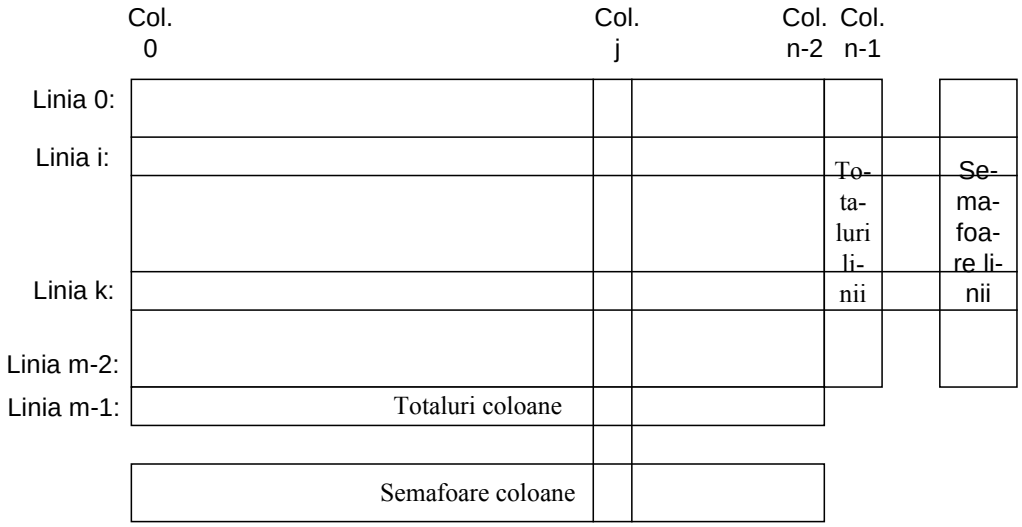

Figura ? O imagine a unei foi de calcul

Textul sursă al acestei aplicații este prezentat în programul ?.

```
/* Avem un tablou bidimensional de intregi care este retinut
   intr-o zona de memorie partajata. Tabloul bidimensional
   are proprietatea ca orice element de pe ultima linie este
   suma tuturor elementelor de pe coloana sa si respectiv
   un element de pe ultima coloana este suma tuturor elementelor
   de pe linia sa. Elementul din coltul dreapta jos nu este
   folosit. La aceasta zona de memorie au acces doua procese:
   unul care ia alege aleator un elemental tabloului bidimensional
   si il modifica la o valoare aleatoare si un alt proces care
   verifica daca sumele totale sunt corecte si tipareste
   elementele tabloului bidmensional. */
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
```

```
/* Blocheaza semaforul n din setul sem */
void
lock (int sem, int n)
{
  struct sembuf sop;
  sop.sem_num = n;
  sop.sem_op = -1;
  sop.sem_flg = 0;
  semop (sem, &sop, 1);
}

/* Deblocheaza semaforul n din setul sem */
void
unlock (int sem, int n)
{
  struct sembuf sop;
  sop.sem_num = n;
  sop.sem_op = 1;
  sop.sem_flg = 0;
  semop (sem, &sop, 1);
}

/* Creaza un set de n semafoare cu cheia k */

int
face_sema (key_t k, int n)
{
  int semid, i;

/* Verifica daca deja exista un semafor cu cheia k */
  if ((semid = semget (k, n, 0)) != -1)
/* Distruge semaforul existent */
    semctl (semid, 0, IPC_RMID, 0);
  if ((semid = semget (k, n, IPC_CREAT | 0600)) != -1)
/* Semaforul este blocat trebuie sa-l deblocam */
    for (i = 0; i < n; i++)
      unlock (semid, i);
  return semid;
}

/* Linia si coloana ce retine sume totale pe linii si coloane */
int linie_suma, coloana_suma;

/* Macrou de acces element din linia l, coloana c a tabloului t */
#define ELEM(l,c,t) (*((l)+((c)*NRCOL)+t))

int linie_sema, coloana_sema;

/* numarul de linii si de coloane */
#define NRLIN 8
#define NRCOL 8

/* cheia zonei de memorie partajata si a semaforului pentru linii
   CHEIE+1 este cheia semaforului pentru coloane */
#define CHEIE 4321


/*---------------- modifica_aleator() -------------------------*/
/* Selecteaza aleator un element al tabloului bidimensional t,
   insereaza in locul sau o valoare aleatoare si actualizeaza
   linia si coloana cu sumele totale */
```

```
void
modifica_aleator (int *t)
{
  int linie, coloana, valoare_veche, valoare_noua;

/* Ia un element aleator si il modifica */
  linie = rand () % (NRLIN - 1);
  coloana = rand () % (NRCOL - 1);
  valoare_noua = rand () % 1000;

/* Inceputul unei sectiuni critice */
  lock (linie_sema, linie);
  lock (coloana_sema, coloana);

/* Se modifica elementul din tabloul bidimensional */
  valoare_veche = ELEM (t, linie, coloana);
  ELEM (t, linie, coloana) = valoare_noua;

/* Se modifica sumele */
  ELEM (t, linie, coloana_suma) += (valoare_noua - valoare_veche);
  ELEM (t, linie_suma, coloana) += (valoare_noua - valoare_veche);

/* Sfirsitul sectiunii critice */
  unlock (linie_sema, linie);
  unlock (coloana_sema, coloana);

  usleep (5000);
}

/* ------------------- verifica_tipareste () ------------------ */
void
verifica_tipareste (int *t)
{
  int linie, coloana, suma, nr_gresit;
  static int nr_tablou = 0;   /* numara tablourile bidimensionale */

  nr_gresit = 0;
  nr_tablou++;
  for (linie = 0; linie < NRLIN; linie++) {
    suma = 0;
/* Inceputul sectiunii critice */
    lock (linie_sema, linie);
    for (coloana = 0; coloana < NRCOL; coloana++) {
      if (coloana != coloana_suma)
      suma += ELEM (t, linie, coloana);
      printf ("%5d", ELEM (t, linie, coloana));
    }
    if (linie != linie_suma)
      nr_gresit += (suma != ELEM (t, linie, coloana_suma));
/* Sfirsitul sectiunii critice */
    unlock (linie_sema, linie);
    printf ("\n");
  }

  for (coloana = 0; coloana < coloana_suma; coloana++) {
    suma = 0;
/* Inceputul sectiunii critice */
    lock (coloana_sema, coloana);
    for (linie = 0; linie < linie_suma; linie++)
      suma += ELEM (t, linie, coloana);
    nr_gresit += (suma != ELEM (t, linie_suma, coloana));
/* Sfirsitul sectiunii critice */
    unlock (coloana_sema, coloana);
```

```
  }
  if (nr_gresit)
    printf ("\nTabloul nr %d gresit\n", nr_tablou);
  if ((nr_tablou % 100) == 0)
    printf ("\nTabloul nr %d prelucrat\n", nr_tablou);
  printf ("\n-----------------------------\n");
  sleep (1);
}

/* --------------- main ------------------------------------ */
void
main ()
{
  int id, linie, coloana, *tablou;

  setbuf (stdout, NULL);

  linie_suma = NRLIN - 1;
  coloana_suma = NRCOL - 1;

/* Creaza un segment de memorie partajata */
  id = shmget (CHEIE, NRLIN * NRCOL * sizeof (int),
                                  IPC_CREAT | 0600);

  if (id < 0) {
    perror ("Eroare la crearea segmentului de memorie partajata");
    exit (1);
  }

  tablou = (int *) shmat (id, 0, 0);
  if (tablou < (int *) (0)) {
    perror (
          "Eroare la atasarea la segmentul de memorie partajata");
    exit (2);
  }

/* Initializeaza tabloul cu 0 */
  for (linie = 0; linie < NRLIN; linie++)
    for (coloana = 0; coloana < NRCOL; coloana++)
      ELEM (tablou, linie, coloana) = 0;

/* Creaza doua seturi de semafoare pentru linii si coloane */
  linie_sema = face_sema (CHEIE, NRLIN);
  coloana_sema = face_sema (CHEIE + 1, NRCOL);

  if ((linie_sema < 0) || (coloana_sema < 0)) {
    perror ("Eroare la crearea semafoarelor");
    exit (2);
  }

/* Pornesc doua procese: unul care modifica matricea si
   altul care o verifica si o tipareste */
  if (fork ()) {
    while (1)
      verifica_tipareste (tablou);
  }
  else {
    while (1)
      modifica_aleator (tablou);
  }
}
```

Programul ? Sursa `semspread.c`

1.3.3.4 Blocarea fişierelor folosind semafoare

Revenind la problema blocării fişierelor, aceasta se poate face folosind un semafor binar. Programul ? prezintă funcțiile de blocare pentru problema propusă

```
/*
 * Rutine de blocare folosind semafoare
 */

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define SEMKEY 123456L
#define PERMS 0666

static struct sembuf op_lock[2] = {
  0, 0, 0,              /* asteapta ca semaforul 0 sa
                         * devina zero */
  0, 1, 0               /* apoi incrementeaza-l cu 1 */
};

static struct sembuf op_unlock[1] = {
  0, -1, IPC_NOWAIT           /* Decrementeaza semaforul 0 cu
                           * 1 (pune-l la 0) */
};

int       semid = -1;         /* identificatorul semaforului */

my_lock(fd)
  int       fd;
{
  if (semid < 0) {
    if ((semid = semget(SEMKEY, 1, IPC_CREAT | PERMS)) < 0)
      err_sys("Eroare la semget");
  }
  if (semop(semid, &op_lock[0], 2) < 0)
    err_sys("Eroare la blocarea cu semop");
}

my_unlock(fd)
  int       fd;
{

  if (semop(semid, &op_unlock[0], 1) < 0)
    err_sys("Eroare la deblocarea cu semop");
}
```

Programul ? Sursa `lock.semafor.c`

Rezolvarea din programul de mai sus ridică următoarea problemă: dacă procesul se abortează din diverse motive, atunci semaforul rămâne pe 1, deci blocat. Orice proces care doreşte acces la fişier va aştepta la nesfârşit după semafor. Există trei căi de evitare a acestei neplăceri:

- Procesul care este abordat să facă deblocarea înainte de a fi el abortat. Aceasta înseamnă o întrerupere (trap) la semnalul SIGKILL.

- Funcția `my_lock` să fie concepută mai sofisticat, în sensul că dacă după scurgerea unui interval de timp, să zicem câteva minute, nu se deblochează, să se intervină cu semctl.
- A treia soluție este ca în această situație să se deblocheze forțat semaforul. Aceasta este soluția adoptată începând cu Unix System V, pe care o prezentăm mai jos.

Pentru deblocare forțată, orice valoare de semafor va avea opțional asociată o altă valoare, numită *valoare de ajustare*. Regulile de utilizare a ei sunt:

1. Când se inițializează semaforul, valoarea de ajustare se pune zero, iar când semaforul este şters, se şterge şi variabila asociată.
2. Pentru orice operație `semop`, dacă se specifică SEM_UNDO, atunci aceeaşi operație se aplică şi valorii asociate.
3. Când procesul face `exit`, se aplică valoarea de ajustare pentru acest proces.

Programul ? prezintă sursa de blocare în acest caz.

```
/*
 * Rutine de blocare folosind semafoare, care foloseste in plus
 * SEM_UNDO pentru ajustarea la nucleu in caz de iesire
 * prematura.
 */

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define SEMKEY 123456L
#define PERMS 0666

static struct sembuf op_lock[2] = {
  0, 0, 0,              /* asteapta ca semaforul 0 sa
                         * devina zero */
  0, 1, SEM_UNDO        /* apoi incrementeaza-l cu 1 */
};

static struct sembuf op_unlock[1] = {
  0, -1, (IPC_NOWAIT | SEM_UNDO)/* Decrementeaza semaforul 0 cu
                         * 1 (pune-l la 0) */
};

int       semid = -1;         /* identificatorul semaforului */

my_lock(fd)
  int       fd;
{
  if (semid < 0) {
    if ((semid = semget(SEMKEY, 1, IPC_CREAT | PERMS)) < 0)
      err_sys("Eroare la semget");
  }
  if (semop(semid, &op_lock[0], 2) < 0)
    err_sys("Eroare la blocarea cu semop");
}

my_unlock(fd)
  int       fd;
{
```

```
  if (semop(semid, &op_unlock[0], 1) < 0)
    err_sys("Eroare la deblocarea cu semop");
}
```

Programul ? `lock.sem.undo.c`

1.3.3.5 Bibliotecă de emulare a semafoarelor teoretice

Există câteva probleme cu semafoarele, ca de exemplu:

- Crearea unui semafor şi inițializarea lui sunt independente, ceea ce crează probleme.
- Până când un semafor nu este şters efectiv, el rămâne prezent în memorie.

Următorul pachet de funcții uşurează şi simplifică lucrul cu semafoare:

```
/*
 * Se realizeaza o interfata simpla cu apelurile sistem ale
 * semafoarelor, la UNIX System V. Sunt disponibile 7 rutine:
 *
 * id = sem_create(key, initval); # creaza cu val. init. sau
 * deschide
 *
 * id = sem_open(key); # deschide, presupunand ca exista deja
 *
 * sem_wait(id); # operatia P: scade cu 1
 *
 * sem_signal(id); # operatia V creste cu 1
 *
 * sem_op(id, * cantitate); # wait daca cantitate < 0 # signal daca
 * cantitate > 0
 *
 * sem_close(id); # inchide
 *
 * sem_rm(id); # sterge semaforul
 *
 * Vom crea si folosi un set de trei membri pentru fiecare semafor
 * cerut.
 *
 * Primul membru, [0], este valoarea actuala a semaforului.
 *
 * Al doilea membru, [1], este un numarator folosit sa stie cand
 * toate procesele au trecut de semafor. El este initializat cu
 * un numar foarte mare, decrementat la fiecare creare sau
 * deschidere si incrementat la fiecare inchidere. In acest mod
 * se foloseste facilitatea de "ajustare" oferita de SV pentru
 * situatiile de exit accidental sau intentionat si nepotrivit.
 *
 * Al treilea membru, [2], este folosit pentru a bloca variabile si
 * a evita conditiile rele de la sem_create() si sem_close().
 */

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <errno.h>
extern int errno;
#define BIGCOUNT 10000        /* Valoarea initiala a
                          * contorului de procese */
/*
 * Defineste operatiile cu semafoare apelate prin semop()
```

```
 */
static struct sembuf op_lock[2] = {
  2, 0, 0,              /* asteapta ca semaforul [2] sa
                         * devina zero */
  2, 1, SEM_UNDO        /* apoi incrementeaza-l pe [2]
                         * cu 1. UNDO va elibera
                         * blocarea daca procesul exista
                         * inainte de deblocarea
                         * efectiva */
};
static struct sembuf op_endcreate[2] = {
  1, -1, SEM_UNDO,            /* Decrementeaza [1] la iesire.
                         * UNDO ajusteaza contorul de
                         * procese daca procesul exista
                         * inaintea apelului explicit
                         * sem_close() */
  2, -1, SEM_UNDO       /* apoi decrementeaza-l pe [2]
                         * cu 1. */
};
static struct sembuf op_open[2] = {
  1, -1, SEM_UNDO,            /* Decrementeaza [1] la iesire. */
};
static struct sembuf op_close[3] = {
  2, 0, 0,              /* Asteapta pentru [2] sa devina
                         * = 0 */
  2, 1, SEM_UNDO,       /* apoi incrementeaza [2] cu 1
                         * ptr blocare */
  1, 1, SEM_UNDO        /* apoi incrementeaza [1] cu 1
                         * (contor de procese) */
};
static struct sembuf op_unlock[1] = {
  2, -1, SEM_UNDO,            /* Decrementeaza [2] inapoi la 0 */
};
static struct sembuf op_op[1] = {
  0, 99, SEM_UNDO,            /* Decrementeaza sau
                         * incrementeaza [0] cu undo si
                         * exit. 99 este cantitatea
                         * actuala de adaugat sau scazut
                         * (pozitiva sau negativa) */
};
union {
  int       val;
  struct semid_ds *buf;
  ushort   *array;
}         semctl_arg;
/*
 * ******************************************************
 * Creaza un semafor cu o valoare initiala. Daca exista, atunci nu
 * se va mai initializa. Intoarce ID al semaforului, sau -1 la
 * esec
 */
int
sem_create(key, initval)
  key_t     key;
  int       initval;
{
  register int id, semval;
  if (key == IPC_PRIVATE)
    return (-1);
  else if (key == (key_t) - 1)
    return (-1);
iarasi:
  if ((id = semget(key, 3, 0666 | IPC_CREAT)) < 0)
```

```
    return (-1);
  /*
   * Cand se creaza semaforul, se stie ca toate valorile sunt 0.
   * Se face o blocare a semaforului, asteptand ca [2] sa devina
   * 0, apoi il incrementeaza.
   *
   * Apare o problema: intre semget de mai sus si semop de mai jos
   * poate interveni un alt proces. De aceea vom relua
   */
  if (semop(id, &op_lock[0], 2) < 0) {
    if (errno == EINVAL)
      goto iarasi;
    err_sys("Nu se poate bloca");
  }
  /*
   * Intoarce valoarea semaforului
   */
  if ((semval = semctl(id, 1, GETVAL, semctl_arg)) < 0)
    err_sys("Nu pot face GETVAL");
  if (semval == 0) {          /* In loc de a face SETALL, care
                         * va sterge si valoarea de
                         * ajustare, vom initializa [0]
                         * si [1] */
    semctl_arg.val = initval;
    if (semctl(id, 0, SETVAL, semctl_arg) < 0)
      err_sys("Nu pot face SETVAL[0]");
    semctl_arg.val = BIGCOUNT;
    if (semctl(id, 1, SETVAL, semctl_arg) < 0)
      err_sys("Nu pot face SETVAL[1]");
  }
  /*
   * Decrementeaza contorul de procese si elibereaza blocarea
   */
  if (semop(id, &op_endcreate[0], 2) < 0)
    err_sys("Nu pot endcreate");
  return (id);
}
/*
 * ********************************************************
 * Deschide un semafor care exista deja. Functia trebuie folosita
 * in locul lui sem_create daca utilizatorul stie ca semaforul
 * exista deja. De exemplu, un client poate face asa ceva daca
 * stie ca sarcina crearii revine serverului
 */
int
sem_open(key)
  key_t     key;
{
  register int id;
  if (key == IPC_PRIVATE)
    return (-1);
  else if (key == (key_t) - 1)
    return (-1);
  if ((id == semget(key, 3, 0)) < 0)
    return (-1);
  /*
   * Decrementeaza numarul de procese. Aceasta operatie nu
   * trebuie sa se faca blocat
   */
  if (semop(id, &op_open[0], 1) < 0)
    err_sys("Nu pot open");
  return (id);
}
```

```
/*
 * ******************************************************
 * Sterge un semafor. Aceasta sarcina revine de regula serverului,
 * si trebuie sa fie precedata de close de catre toate procesele
 * care-l folosesc
 */
sem_rm(id)
  int       id;
{
  semctl_arg.val = 0;
  if (semctl(id, 0, IPC_RMID, semctl_arg) < 0)
    err_sys("Nu pot IPC_RMID");
}
/*
 * Inchide un semafor.Se va decrementa contorul de procese. Daca
 * este ultimul proces, atunci se va sterge semaforul.
 */
sem_close(id)
  int       id;
{
  register int semval;
  /*
   * Mai intai semop blocheaza semaforul, apoi incrementeaza
   * contorul de procese
   */
  if (semop(id, &op_close[0], 3) < 0)
    err_sys("Nu pot semop");
  /*
   * Dupa blocare, citeste valoarea contorului de procese si
   * vede daca este ultima referire la el. Daca da, atunci ?
   */
  semctl_arg.val = 0;
  if ((semval = semctl(id, 1, GETVAL, semctl_arg)) < 0)
    err_sys("Nu pot GETVAL");
  if (semval > BIGCOUNT)
    err_sys("sem[1]>BIGCOUNT");
  else if (semval == BIGCOUNT)
    sem_rm(id);
  else if (semop(id, &op_unlock[0], 1) < 0)
    err_sys("Nu pot unlock");
}
/*
 * ******************************************************
 * Asteapta pana cand valoarea semaforului este > 0, apoi
 * decrementeaza cu 1. Este vorba de operatia P a lui Dijkstra
 * sau DOWN a lui Tanenbaum
 */
sem_wait(id)
  int       id;
{
  sem_op(id, -1);
}
/*
 * ******************************************************
 * Incrementeaza semaforul cu 1. Este vorba de operatia V a lui
 * Dijkstra sau UP a lui Tanenbaum
 */
sem_signal(id)
  int       id;
{
  sem_op(id, 1);
}
/*
```

```
 * ******************************************************
 * Operatii generale cu semaforul. Incrementeaza sau decrementeaza
 * cu o cantitate specificata, care trebuie sa fie diferita de
 * zero
 */
sem_op(id, cantitate)
  int       id, cantitate;
{
  if ((op_op[0].sem_op = cantitate) == 0)
    err_sys("Cantitate egala cu zero");
  if (semop(id, &op_op[0], 1), 0)
    err_sys("Eroare sem_op");
}
```

Programul ? `bib.semafor.c`

1.3.3.6 Blocarea fişierelor cu semafoare (2)

Reluăm acum blocarea fişierului cu funcţiile din biblioteca de mai sus.

```
/*
 * Exemplu de blocare folosind operatii simple cu semafoare
 */

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define SEQFILE "secv"
#define MAXBUF  100
#define SEMKEY ((key_t) 23456L)

#include "err_sys.c"
#include "bib.semafor.c"

main()
{
  int       fd, i, n, pid, nrsecv, semid;
  char      buf[MAXBUF];

  pid = getpid();
  if ((fd = open(SEQFILE, 2)) < 0)
    err_sys("Nu poate deschide");

  if ((semid = sem_create(SEMKEY, 1)) < 0)
    err_sys("Nu se poate deschide semaforul");

  for (i = 0; i < 20; i++) {

    sem_wait(semid);          /* Blocheaza fisierul */

    lseek(fd, 0L, 0);         /* Pozitionare la inceput */
    if ((n = read(fd, buf, MAXBUF)) <= 0)
      err_sys("Eroare la citire");
    buf[n] = '\0';            /* Zeroul terminal */

    if ((n = sscanf(buf, "%d\n", &nrsecv)) != 1)
      err_sys("Eroare la sscanf");
    printf("pid = %d, secventa = %d\n", pid, nrsecv);

    nrsecv++;                 /* Incrementeaza secventa */
```

```
    sprintf(buf, "%03d\n", nrsecv);
    n = strlen(buf);
    lseek(fd, 0L, 0);         /* Revenire inainte de scriere */
    if (write(fd, buf, n) != n)
      err_sys("Eroare la scriere");

    sem_signal(semid);        /* Deblocheaza fisierul */
  } sem_close(semid);
}
```

Programul ? `lock.sem.bibl.c`

1.3.3.7 Client / server cu acces exclusiv la memoria partajată

Acum vom prezenta un model de rezolvare a problemelor de tip client / server pe o aceeaşi maşină, folosind un segment de memorie partajată protejat prin semafoare. Concret, clientul trimite serverului un nume de fişier, iar serverul întoarce spre client conținutul acestui fişier sau un mesaj de eroare dacă fişierul nu există.

Programul ? prezintă sursa serverului, iar programul ? prezintă sursa clientului.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#include "err_sys.c"
#include "shm.h"
#include "bib.semafor.c"

int       shmid, clisem, servsem;
Mesg     *mesgptr;

main()
{
  /*
   * Creaza un segment de memorie partajata. Daca se cere, il
   * ataseaza
   */

  if ((shmid = shmget(SHMKEY, sizeof(Mesg), PERMS | IPC_CREAT)) < 0)
    err_sys(" Server:Nu poate obtine memorie partajata ");
  if ((mesgptr = (Mesg *) shmat(shmid, (char *) 0,
                        0)) == (Mesg *) - 1)
    err_sys("Server: Nu poate atasa memorie partajata");
  /*
   * Creaza doua semafoare. Semaforul client va avea valoarea 1
   */

  if ((clisem = sem_create(SHMKEY1, 1)) < 0)
    err_sys("Server: Nu poate crea semafor client");
  if ((servsem = sem_create(SHMKEY2, 0)) < 0)
    err_sys("Server: Nu poate crea semafor server");

  server();

  /*
   * Deataseaza memoria partajata si inchide semafoarele.
   * Procesul client o va folosi ultimul, asa ca el o va sterge
```

```
   */

  if (shmdt(mesgptr) < 0)
    err_sys("Server: Nu pot elibera memoria partajata");

  sem_close(clisem);
  sem_close(servsem);

  exit(0);
}

server()
{
  int       n, filefd;
  char      buf[256];

  /*
   * Asteapta clientul sa scrie in memoria partajata
   */

  sem_wait(servsem);

  mesgptr->mesg_data[mesgptr->mesg_len] = '\0';

  if ((filefd = open(mesgptr->mesg_data, 0)) < 0) {
    /*
     * Eroare, formatul unui mesaj . Se emite un mesaj de eroare
     * la client
     */
    strcat(mesgptr->mesg_data, " Probabil ca nu exista\n");
    mesgptr->mesg_len = strlen(mesgptr->mesg_data);
    sem_signal(clisem);
    sem_wait(servsem);
  } else {
    /*
     * Citeste datele din fisier si le scrie pune in memoria
     * partajata
     */
    while ((n = read(filefd, mesgptr->mesg_data,
                MAXMESGDATA - 1)) > 0) {
      mesgptr->mesg_len = n;
      sem_signal(clisem);
      sem_wait(servsem);
    }
    close(filefd);
    if (n < 0)
      err_sys("server: eroare la citire");
  }
  /*
   * Emite un mesaj de lungime 0 pentru end
   */
  mesgptr->mesg_len = 0;
  sem_signal(clisem);
}
```

Programul ? Sursa `Mpserver.c`

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#include "err_sys.c"
```

```
#include "shm.h"
#include "bib.semafor.c"

int       shmid, clisem, servsem;
Mesg     *mesgptr;

main()
{
  /*
   * Obtine un segment de memorie partajata.
   */

  if ((shmid = shmget(SHMKEY, sizeof(Mesg), 0)) < 0)
    err_sys(" Client:Nu poate obtine memorie partajata ");
  if ((mesgptr = (Mesg *) shmat(shmid, (char *) 0,
                        0)) == (Mesg *) - 1)
    err_sys("Client: Nu poate atasa memorie partajata");
  /*
   * Deschide doua semafoare, create deja de server
   */

  if ((clisem = sem_create(SHMKEY1, 1)) < 0)
    err_sys("Client: Nu poate obtine semafor client");
  if ((servsem = sem_create(SHMKEY2, 0)) < 0)
    err_sys("Client: Nu poate obtine semafor server");

  client();

  /*
   * Deataseaza memoria partajata si inchide semafoarele.
   * Procesul client o va folosi ultimul, asa ca el o va sterge
   */

  if (shmdt(mesgptr) < 0)
    err_sys("Client: Nu pot elibera memoria partajata");

  if (shmctl(shmid, IPC_RMID, (struct shmid_ds *) 0) < 0)
    err_sys("Client: Nu poate sterge memoria partajata");

  sem_close(clisem);
  sem_close(servsem);

  exit(0);
}

client()
{
  int       n;

  /*
   * Citeste numele de fisier de la intrarea stamdard si-l
   * depune in memoria partajata
   */

  sem_wait(clisem);
  if (fgets(mesgptr->mesg_data, MAXMESGDATA, stdin) == NULL)
    err_sys("Eroare la citirea numelui de fisier");
  n = strlen(mesgptr->mesg_data);
  if (mesgptr->mesg_data[n - 1] == '\n')
    n--;
  mesgptr->mesg_len = n;
  sem_signal(servsem);
```

```
  /*
   * Asteapta dupa server sa-l plaseze undeva in memoria
   * partajata
   */

  sem_wait(clisem);
  while ((n = mesgptr->mesg_len) > 0) {
    if (write(1, mesgptr->mesg_data, n) != n)
      err_sys("Eroare la scrierea datelor");

    sem_signal(servsem);
    sem_wait(clisem);
  }
  if (n < 0)
    err_sys("server: eroare la citire");
}
```

Programul ? Sursa `Mpclient.c`

1.3.3.8 Zone tampon multiple

Perechea de programe care urmează implementează mecanismul de acces la fişiere prin intermediul zonelor tampon multiple.

Folosirea de zone tampon multiple face crească viteza de prelucrare a fişierelor. Să considerăm secvența:

```
while ((n = read (fdin, buf, BUFSIZE) > 0) {
      /* prelucrare date */
      write(fdout, buf, n);
};
```

În funcție de numărul de procese care participă la aceste operații şi de numărul de zone tampon folosite, timpul total de lucru se poate scurta simțitor. În fig. ?, ? şi ? prezentăm trei variante de lucru:

- Un singur proces şi o singură zonă tampon pentru citire şi scriere.
- Două procese, un client şi un server, cu o singură zonă tampon.
- Două procese, un client şi un server, şi două zone tampon care-şi schimbă rolurile periodic.

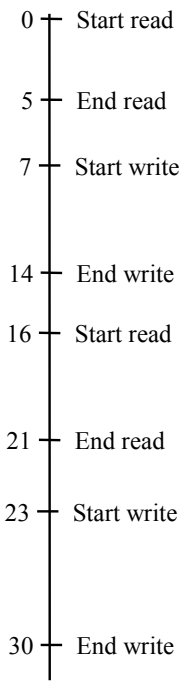

Figura ? Un proces un tampon

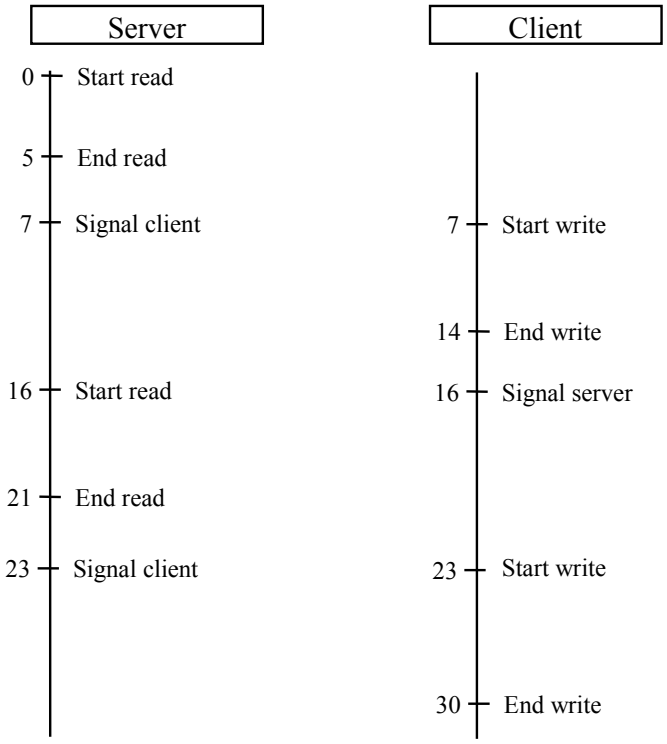

Figura ? Două procese un tampon

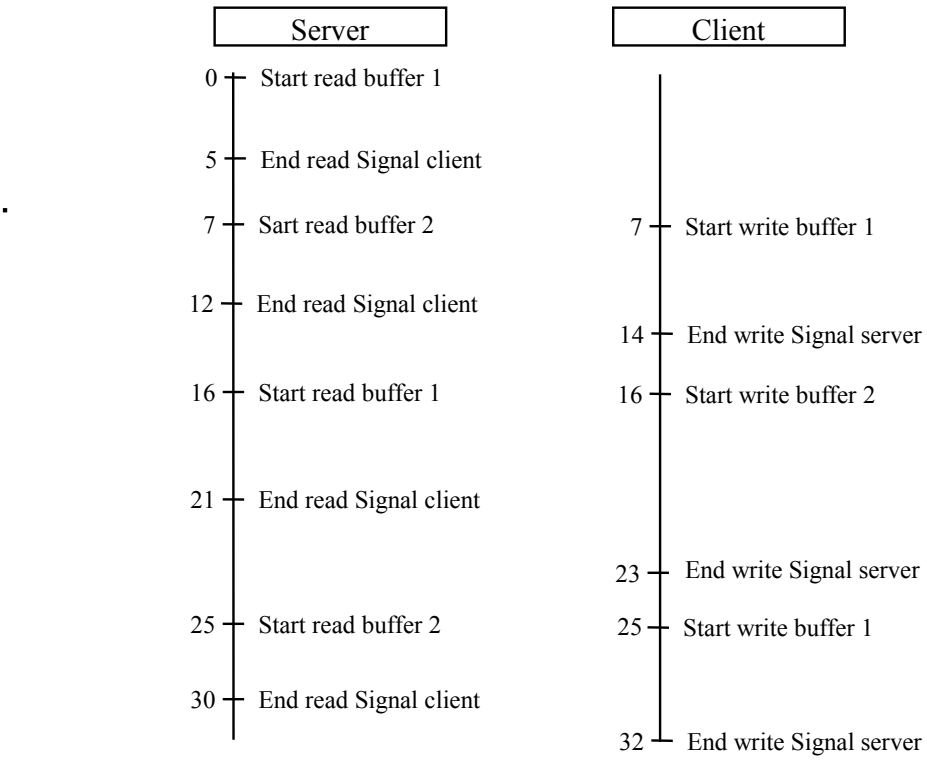

Figura ? Două procese două tampoane

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#include "err_sys.c"
#include "shm.h"
#include "bib.semafor.c"

int       shmid[NBUFF], clisem, servsem;
Mesg     *mesgptr[NBUFF];

main()
{
  register int i;
  /*
   * Creaza segmentele de memorie partajata.
   */

  for (i = 0; i < NBUFF; i++) {
    if ((shmid[i] = shmget(SHMKEY + i, sizeof(Mesg),
                     PERMS | IPC_CREAT)) < 0)
      err_sys(" Server: Nu poate obtine memorie partajata ");
    if ((mesgptr[i] = (Mesg *) shmat(shmid[i], (char *) 0,
                              0)) == (Mesg *) - 1)
      err_sys("Server: Nu poate atasa memorie partajata");
  }
  /*
   * Creaza doua semafoare.
   */
```

```
  if ((clisem = sem_create(SHMKEY1, 1)) < 0)
    err_sys("Server: Nu poate crea semafor client");
  if ((servsem = sem_create(SHMKEY2, 0)) < 0)
    err_sys("Server: Nu poate crea semafor server");

  server();

  /*
   * Deataseaza memoria partajata si inchide semafoarele.
   * Procesul client o va folosi ultimul, asa ca el o va sterge
   */

  for (i = 0; i < NBUFF; i++) {
    if (shmdt(mesgptr[i]) < 0)
      err_sys("Server: Nu pot elibera memoria partajata");
  }
  sem_close(clisem);
  sem_close(servsem);

  exit(0);
}

server()
{
  register int i, n, filefd;
  char      buf[256];

  /*
   * Asteapta clientul sa scrie in memoria partajata
   */

  sem_wait(servsem);

  mesgptr[0]->mesg_data[mesgptr[0]->mesg_len] = '\0';

  if ((filefd = open(mesgptr[0]->mesg_data, 0)) < 0) {
    /*
     * Eroare, formatul unui mesaj . Se emite un mesaj de eroare
     * la client
     */
    strcat(mesgptr[0]->mesg_data, " Probabil ca nu exista\n");
    mesgptr[0]->mesg_len = strlen(mesgptr[0]->mesg_data);
    sem_signal(clisem);
    sem_wait(servsem);
    mesgptr[1]->mesg_len = 0;
    sem_signal(clisem);
  } else {
    for (i = 0; i < NBUFF; i++)
      sem_signal(servsem);
    /*
     * Citeste datele din fisier si le scrie pune in memoria
     * partajata
     */
    for (;;) {
      for (i = 0; i < NBUFF; i++) {
      sem_wait(servsem);
      n = read(filefd, mesgptr[i]->mesg_data,
             MAXMESGDATA - 1);
      if (n < 0)
        err_sys("Server: Eroare read");
      mesgptr[i]->mesg_len = n;
      sem_signal(clisem);
      sem_wait(servsem);
```

```
      if (n == 0)
        goto GATA;
      }
    }
GATA:
    close(filefd);
  }
}
```

Programul ? Sursa Mbufserver.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#include "err_sys.c"
#include "shm.h"
#include "bib.semafor.c"

int       shmid[NBUFF], clisem, servsem;
Mesg     *mesgptr[NBUFF];

main()
{
  register int i;
  /*
   * Obtine segmentele de memorie partajata.
   */

  for (i = 0; i < NBUFF; i++) {
    if ((shmid[i] = shmget(SHMKEY + i, sizeof(Mesg), 0)) < 0)
      err_sys(" Client:Nu poate obtine memorie partajata ");
    if ((mesgptr[i] = (Mesg *) shmat(shmid[i], (char *) 0,
                                0)) == (Mesg *) - 1)
      err_sys("Client: Nu poate atasa memorie partajata");
  }

  /*
   * Deschide doua semafoare, create deja de server
   */

  if ((clisem = sem_create(SHMKEY1, 1)) < 0)
    err_sys("Clieny: Nu poate obtine semafor client");
  if ((servsem = sem_create(SHMKEY2, 0)) < 0)
    err_sys("Client: Nu poate obtine semafor server");

  client();

  /*
   * Deataseaza memoria partajata si inchide semafoarele.
   * Procesul client o va folosi ultimul, asa ca el o va sterge
   */

  for (i = 0; i < NBUFF; i++) {
    if (shmdt(mesgptr[i]) < 0)
      err_sys("Client: Nu pot elibera memoria partajata");

    if (shmctl(shmid[i], IPC_RMID, (struct shmid_ds *) 0) < 0)
      err_sys("Client: Nu poate sterge memoria partajata");
  }

  sem_close(clisem);
  sem_close(servsem);
```

```
  exit(0);
}

client()
{
  int       i, n;

  /*
   * Citeste numele de fisier de la intrarea stamdard si-l
   * depune in memoria partajata
   */

  sem_wait(clisem);
  if (fgets(mesgptr[0]->mesg_data, MAXMESGDATA, stdin) == NULL)
    err_sys("Eroare la citirea numelui de fisier");
  n = strlen(mesgptr[0]->mesg_data);
  if (mesgptr[0]->mesg_data[n - 1] == '\n')
    n--;
  mesgptr[0]->mesg_len = n;
  sem_signal(servsem);

  /*
   * Asteapta dupa server sa-l plaseze undeva in memoria
   * partajata
   */

  for (;;) {
    for (i = 0; i < NBUFF; i++) {
      sem_wait(clisem);
      if ((n = mesgptr[i]->mesg_len) <= 0)
      goto GATA;
      if (write(1, mesgptr[i]->mesg_data, n) != n)
      err_sys("Eroare la scrierea datelor");
      sem_signal(servsem);
    }
  }
GATA:
  if (n < 0)
    err_sys("server: eroare la citire");
}
```

Programul ? Sursa `Mbufclient.c`

DE SIMULAT FIFO(PIPE) CU SEMAFOARE SI MEMORIE PARTAJATA

2 Concurență la nivel de threaduri Unix

2.1 Relația procese - thread-uri

La nivel conceptual, noțiunile de proces şi thread sunt relativ apropiate. Diferențele între ele ies în evidență atunci când se au în vedere aspectele de implementare ale acestora.

2.1.1 Definirea procesului

Ce este un proces? Un *proces* sau *task*, este un calcul care poate fi executat concurent (în paralel) cu alte calcule. El este o abstractizare a activității procesorului, fiind considerat ca un program în execuție. Existența unui proces este condiționată de existența a trei factori:

- o *procedură* - o *succesiune de instrucțiuni* dintr-un *set predefinit de instrucțiuni*, cu rolul de descriere a unui calcul - descrierea unui algoritm.
- un *procesor* - dispozitiv hardware/software ce recunoaşte şi poate executa setul predefinit de instrucțiuni, şi care este folosit, în acest caz, pentru a executa succesiunea de instrucțiuni specificată în procedură;
- un *mediu* - constituit din partea din resursele sistemului: o parte din memoria internă, un spațiu disc destinat unor fişiere, periferice magnetice, echipamente audio-video etc. - asupra căruia acționează procesorul în conformitate cu secvența de instrucțiuni din procedură.

2.1.2 Reprezentarea în memorie a unui proces

In ceea ce priveşte reprezentarea în memorie a unui proces, indiferent de platforma (sistemul de operare) pe care este operațional, se disting, în esență, următoarele zone:

- Contextul procesului
- Codul programului
- Zona datelor globale
- Zona heap
- Zona stivei

Contextul procesului conține informațiile de localizare în memoria internă şi informațiile de stare a execuției procesului:

- Legături exterioare cu platforma (sistemul de operare): numele procesului, directorul curent în structura de directori, variabilele de mediu etc.;
- Pointeri către începuturile zonelor de cod, date stivă şi heap şi, eventual, lungimile acestor zone;

- Starea curentă a execuției procesului: contorul de program (notat PC -program counter) ce indică în zona cod următoarea instrucțiune mașină de executat, pointerul spre vârful stivei (notat SP - stack pointer);
- Zone de salvare a regiştrilor generali, de stare a sistemului de întreruperi etc.

De exemplu, în [15] se descrie contextul procesului sub sistemul de operare DOS, iar în [13] contextul procesului sub sistemul de operare Unix.

Zona de cod conține instrucțiunile mașină care dirijează funcționarea procesului. De regulă, conținutul acestei zone este stabilit încă din faza de compilare. Programatorul descrie programul într-un limbaj de programare de nivel înalt. Textul sursă al programului este supus procesului de compilare care generează o secvență de instrucțiuni mașină echivalentă cu descrierea din program.

Conținutul acestei zone este folosit de procesor pentru a-şi încărca rând pe rând instrucțiunile de executat. Registrul PC indică, în fiecare moment, locul unde a ajuns execuția.

Zona datelor globale conține constantele şi variabilele vizibile de către toate instrucțiunile programului. Constantele şi o parte dintre variabile primesc valori încă din faza de compilare. Aceste valori inițiale sunt încărcate în locațiile de reprezentare din zona datelor globale în momentul încărcării programului în memorie.

Zona heap - cunoscută şi sub numele de zona variabilelor dinamice - găzduieşte spații de memorare a unor variabile a căror durată de viață este fixată de către programator. *Crearea* (operația *new*) unei astfel de variabile înseamnă rezervarea în heap a unui şir de octeți necesar reprezentării ei şi întoarcerea unui pointer / referințe spre începutul acestui şir. Prin intermediul referinței se poate utiliza în scriere şi/sau citire această variabilă până în momentul *distrugerii* ei (operație *destroy*, *dispose* etc.). Distrugerea înseamnă eliberarea şirului de octeți rezervat la creare pentru reprezentarea variabilei. In urma distrugerii, octeții eliberați sunt plasați în lista de spații libere a zonei heap.

In [13,47,2] sunt descrise mecanismele specifice de gestiune a zonei heap.

Zona stivă In momentul în care programul apelează o procedură sau o funcție, se depun în vârful stivei o serie de informații: parametrii transmişi de programul apelator către procedură sau funcție, adresa de revenire la programul apelator, spațiile de memorie necesare reprezentării variabilelor locale declarate şi utilizate în interiorul procedurii sau funcției etc. După ce procedura sau funcția îşi încheie activitatea, spațiul din vârful stivei ocupat la momentul apelului este eliberat. In cele mai multe cazuri, există o stivă unică pentru fiecare proces. Există însă platforme, DOS este un exemplu [14], care folosesc mai multe stive simultan: una rezervată numai pentru proces, alta (altele) pentru apelurile sistem. Conceptul de thread, pe care-l vom prezenta imediat, induce ca regulă generală existența mai multor spații de stivă.

In figura 2.4 sunt reprezentate două procese active simultan într-un sistem de calcul.

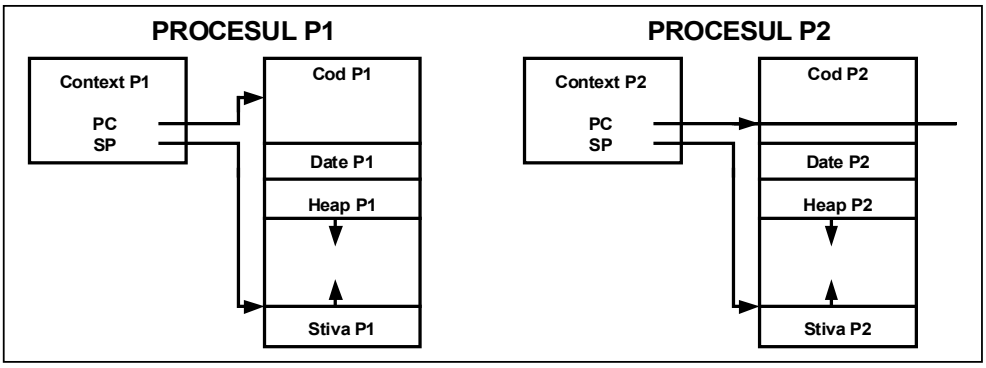

Figura 2.1 Două procese într-un sistem de calcul

2.1.3 Definiția threadului

Conceptul de *thread*, sau *fir de execuție*, a apărut în ultimii 10-15 ani. Proiectanții şi programatorii au "simțit nevoia" să-şi definească entități de calcul independente, dar în cadrul aceluiaşi proces. Astfel, un *thread* se defineşte ca o entitate de execuție din interiorul unui proces, compusă dintr-un context şi o secvență de instrucțiuni de executat.

Deşi noțiunea de thread va fi prezentată pe larg în capitolele următoare, punctăm aici câteva caracteristici de bază ale acestor entități:

- Thread-urile sunt folosite pentru a crea programe formate din unități de procesare concurentă.
- Entitatea thread execută o secvență dată de instrucțiuni, încapsulate în funcția thread-ului.
- Execuția unui thread poate fi întreruptă pentru a permite procesorului să dea controlul unui alt thread.
- Thread-urile sunt tratate independent, fie de procesul însuşi, fie de nucleul sistemului de operare. Componenta sistem (proces sau nucleu) care gestionează thread-urile depinde de modul de implementare a acestora.
- Operațiile de lucru cu thread-uri sunt furnizate cu ajutorul unor librării de programe (C, C++) sau cu ajutorul unor apeluri sistem (în cazul sistemelor de operare: Windows NT, Sun Solaris).

Esența conceptuală a threadului este aceea că execută o procedură sau o funcție, în cadrul aceluiaşi proces, concurent cu alte thread-uri. Contextul şi zonele de date ale procesului sunt utilizate în comun de către toate thread-urile lui.
Esența de reprezentare în memorie a unui thread este faptul că singurul spațiu de memorie ocupat exclusiv este spațiul de stivă. In plus, fiecare thread îşi întreține propriul context, cu elemente comune contextului procesului părinte al threadului.

In figura 2.5 sunt reprezentate trei thread-uri în cadrul aceluiaşi proces.

Există cazuri când se preferă folosirea proceselor în locul thread-urilor. De exemplu, când este nevoie ca entitățile de execuție să aibă identificatori diferiți sau să-şi

gestioneze independent anumite atribute ale fişierelor (directorul curent, numărul maxim de fişiere deschise) [96].

Un program multi-thread poate să obțină o performanță îmbunătățită prin execuția concurentă şi/sau paralelă a thread-urilor. Execuția concurentă a thread-urilor (sau pe scurt, *concurență*) înseamnă că mai multe thread-uri sunt în *progres*, în acelaşi timp. Execuția paralelă a thread-urilor (sau pe scurt, *paralelism*) apare când mai multe thread-uri se execută *simultan* pe mai multe procesoare.

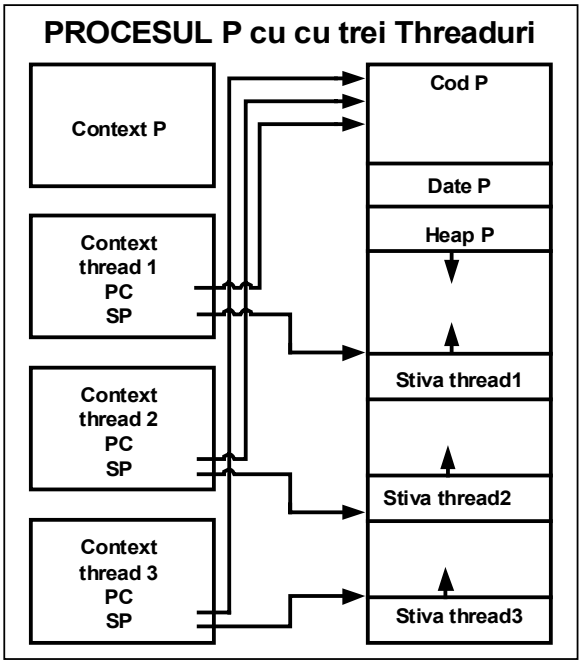

Figura 2.2 Trei thread-uri într-un proces

2.2 Thread-uri pe platforme Unix: Posix şi Solaris

2.2.1 Caracteristici şi comparații Posix şi Solaris

Thread-urile Posix sunt răspândite pe toate platformele Unix, inclusiv Linux, SCO, AIX, Xenix, Solaris, etc. In particular, unele dintre aceste platforme au şi implementări proprii, cu caracteristici mai mult sau mai puțin apropiate de Posix. Dintre acestea, implementarea proprie platformei Solaris este cea mai elaborată şi mai răspândită, motiv pentru care o vom trata în această secțiune, în paralel cu implementarea Posix. Cele două modele au multe similarități la nivel sintactic dar au modalități de implementare diferite

2.2.1.1 Similarități şi facilități specifice

Intre thread-urile Posix şi Solaris există un grad Posix mare de similaritate, atât la nivel *sintactic*, cât şi *funcțional*. Pentru operațiile principale cu entitățile thread există apeluri diferite pentru Posix şi Solaris. Funcțiile thread-urilor Posix au prefixul `pthread_` iar cele Solaris `thr_`. Astfel, o conversie a unui program simplu cu thread-uri de pe una din cele două platforme pe cealaltă, se realizează doar la nivel sintactic, modificând numele funcției şi a unor parametri.
Inainte de a detalia principalele operații cu thread-uri Posix şi Solaris, punctăm câteva diferențe Intre aceste platforme.

Facilități Posix care nu sunt prezente pe Solaris:
- portabilitate totală pe platforme ce suportă Posix
- obiecte purtătoare de atribute
- conceptul de abandon (cancellation)
- politici de securitate

Facilități Solaris care nu sunt prezente pe Posix:
- blocări *reader/writer*
- posibilitatea de a crea thread-uri daemon
- suspendarea şi continuarea unui thread
- setarea nivelului de concurență
- (crearea de noi lwp-uri)

2.2.2 Operații asupra thread-urilor: creare, terminare

Inainte de a descrie operațiile cu thread-uri, trebuie să precizăm că pentru thread-urile Posix trebuie folosit fişierul header `<pthread.h>`, iar pentru Solaris headerele `<sched.h>` şi `<thread.h>`. Compilările în cele două variante se fac cu opțiunile: `-lpthread` (pentru a indica faptul că se foloseşte biblioteca `libpthread.so.0`, în cazul thread-urilor Posix), respectiv opțiunea `-lthread`, care link-editează programul curent cu biblioteca `libthread.so`, în cazul thread-urilor Solaris.

2.2.2.1 Crearea unui thread

Posix API:

```
int pthread_create(pthread_t *tid, pthread_attr_t *attr,
      void *(*func)(void*), void *arg);
```

Solaris API:

```
int    thr_create(void    *stkaddr,    size_t    stksize,    void
*(*func)(void*),
      void *arg, long flags, thread_t *tid);
```

Prin aceste funcții se creează un thread în procesul curent şi se depune în pointerul `tid` descriptorul threadului. Funcțiile întorc valoarea 0 la succes şi o valoare nenulă (cod de eroare), în caz de eşec.

Execuția noului thread este descrisă de funcția al cărei nume este precizat prin parametrul `func`. Această funcție are un singur argument de tip pointer, transmis prin argumentul `arg`.

Varianta Posix prevede atributele threadului prin argumentul `attr`. Asupra acestuia vom reveni Intr-o secțiune ulterioară. Pe moment vom folosi pentru `attr` valoarea NULL, prin aceasta indicând stabilirea de atribute implicite de către sistem.

`stkaddr` şi `stksize` indică adresa şi lungimea stivei threadului. Valorile NULL şi 0 pentru aceşti parametri cer sistemului să fixeze valori implicite pentru adresa şi dimensiunea stivei.

Parametrul `flags` -reprezintă o combinație de constante legate prin '|', cu valori specifice thread-urilor Solaris:

- THR_SUSPENDED – determină crearea threadului în starea suspendat pentru a permite modificarea unor atribute de planificare înainte de execuția funcției ataşată threadului. Pentru a începe execuția threadului se apelează `thr_continue`.
- THR_BOUND –leagă threadul de un nou lwp creat în acest scop. Threadul va fi planificat doar pe acest lwp.
- THR_DETACHED – creează un nou thread în starea detached. Resursele unui thread detached sunt eliberate imediat la terminarea threadului. Pentru acest thread nu se poate apela `thr_join` şi nici nu se poate obține codul de ieşire.
- THR_INCR_CONC sau, echivalent THR_NEW_lwp – incrementează nivelul de concurență prin adăugarea unui lwp la colecția inițială, indiferent de celelalte flag-uri.
- THR_DAEMON – creează un nou thread cu statut de daemon (de exemplu, pentru a trata evenimente asincrone de I/O). Procesul de bază se termină când ultimul thread non-daemon îşi încheie execuția.
- Dacă sunt precizate atât THR_BOUND cât şi THR_INCR_CONC sunt create două lwp-uri odată cu crearea threadului.

2.2.2.2 Terminarea unui thread

In mod obişnuit, terminarea unui thread are loc atunci când se termină funcția care descrie threadul. Din corpul acestei funcții se poate comanda terminarea threadului prin apelurile funcțiilor `pthread_exit`, respectiv `thr_exit`, cu prototipurile:

Posix API:
```
int pthread_exit(int *status);
```

Solaris API:
```
int thr_exit(int *status);
```

Pointerul `status` se foloseşte atunci când se doreşte ca în urma execuției threadul să întoarcă nişte date rezultat spre procesul părinte al threadului. După cum vom vedea imediat, acesta obține datele printr-un apel de tip `join`. In cele mai multe cazuri, `status` are valoarea NULL.

Un apel `pthread_exit/thr_exit` termină procesul, numai dacă threadul apelant este ultimul thread non-daemon din proces.

La terminarea unui thread, acesta este pus într-o listă *deathrow*. Din motive de performanță, resursele asociate unui thread nu sunt eliberate imediat ce acesta îşi încheie execuția. Un thread special, numit *reaper* parcurge periodic lista deathrow şi dealocă resursele thread-urilor terminate [30].

Posix permite comandarea terminării - abandonului - unui thread de către un alt thread. Trebuie remarcat că, în cazul utilizării abandonului, pot să apară probleme dacă execuția threadului este întreruptă în interiorul unei secțiuni critice. De asemenea, threadul nu va fi abandonat înainte de a dealoca resurse precum segmente de memorie partajată sau descriptori de fişiere.

Pentru a depăsi problemele de mai sus, interfața *cancellation* permite abandonarea execuției threadului, doar în anumite puncte. Aceste puncte, numite *puncte de abandon*, pot fi stabilite prin apeluri specifice.

Abandonarea threadului se poate realiza în 3 moduri:
i) asincron, *ii)* în diferite puncte, în timpul execuției, conform specificațiilor de implementare a acestei facilități sau *iii)* în puncte discrete specificate de aplicație.

Pentru efectuarea abandonului se apelează:

```
int pthread_cancel(pthread_t tid);
```

cu identificatorul threadului ca argument.

Cererea de abandon este tratată în funcție de starea threadului. Această stare se poate seta folosind apelurile: `pthread_setstate()` şi `pthread_setcanceltype()`.

Funcția `pthread_setcancelstate` stabileşte punctul respectiv, din threadul curent, ca punct de abandon, pentru valoarea PTHREAD_CANCEL_ENABLE şi interzice starea de abandon, pentru valoarea PTHREAD_CANCEL_DISABLE.

Funcția `pthread_setcanceltype` poate seta starea threadului la valoarea PTHREAD_CANCEL_DEFERRED sau PTHREAD_CANCEL_ASYNCHRONOUS. Pentru prima valoare, întreruperea se poate produce doar în puncte de abandon, iar pentru cea de a doua valoare, întreruperea se produce imediat.

In rutina threadului, punctele de abandon se pot specifica explicit cu `pthread_testcancel()`.
Funcțiile de aşteptare precum `pthread_join`, `pthread_cond_wait` sau `pthread_cond_timedwait` determină, de asemenea, puncte implicite de

abandon. Variabilele mutex şi funcțiile ataşate acestora nu generează puncte de abandon.

Marcarea pentru ştergere a unui thread se face prin apelul:

```
int pthread_detach(pthread_t tid);
```

Această funcție marchează pentru ştergere structurile interne ale threadului. In acelaşi timp, apelul informează nucleul că după terminare threadul `tid` va fi radiat, iar memoria ocupată de el eliberată. Aceste acțiuni vor fi efectuate abia după ce threadul `tid` îşi termină în mod normal activitatea.

2.2.2.3 Aşteptarea terminării unui thread

Cea mai simplă metodă de sincronizare a thread-urilor este aşteptarea terminării unui thread. Ea se realizează prin apeluri de tip `join`:

Posix API: `int pthread_join(pthread_t tid, void **status);`

Solaris API: `int thr_join(thread_t tid, thread_t *realid, void **status);`

Funcția `pthread_join` suspendă execuția threadului apelant până când threadul cu descriptorul `tid` îşi încheie execuția.

In cazul apelului `thr_join`, dacă `tid` este diferit de NULL, se aşteaptă terminarea threadului `tid`. In caz contrar, se aşteaptă terminarea oricărui thread, descriptorul threadului terminat fiind întors în parametrul `realid`.

Dublul pointer `status` primeşte ca valoare pointerul `status` transmis ca argument al apelului `pthread_exit`, din interiorul threadului. In acest fel, threadul terminat poate transmite apelatorului `join` o serie de date.

Thread-urile pentru care se apelează funcțiile `join`, nu pot fi create în starea *detached*. Resursele unui thread *joinable*, care şi-a încheiat execuția, nu se dealocă decât când pentru acest thread este apelată funcția `join`. Pentru un thread, se poate apela `join` o singură dată.

2.2.2.4 Un prim exemplu

Fără a intra deocamdată în prea multe amănunte, pregătim prezentarea programului `ptAtribLung.c`, (programul 4.1), parametrizat de două constante (deci în patru variante posibile), şi în care introducem utilizarea thread-urilor sub Unix.

```
//#define ATRIBLUNG 1
//#define MUTEX 1
#include <stdio.h>
#include <pthread.h>
```

```
typedef struct { char *s;  int nr;  int pas;  int sec;}argument;
int p = 0;
pthread_mutex_t mutp = PTHREAD_MUTEX_INITIALIZER;

void f (argument * a) {
   int i, x;
   for (i = 0; i < a->nr; i++) {
   #ifdef MUTEX
   pthread_mutex_lock (&mutp);
   #endif
      x = p;
      if (a->sec > 0)
         sleep (random () % a->sec);
      printf ("\n%s i=%d pas=%d", a->s, i, a->pas);
      x += a->pas;
   #ifdef ATRIBLUNG
   p = x;
   #else
      p += a->pas;
   #endif
   #ifdef MUTEX
   pthread_mutex_unlock (&mutp);
   #endif
   }
}

main () {
   argument x = { "x:", 20, -1, 2 },
   argument y = { "y:", 10,  2, 3 };
   pthread_t th1, th2;

   pthread_create ((pthread_t *) & th1, NULL, (void *) f,
                   (void *) &x);
   pthread_create ((pthread_t *) & th2, NULL, (void *) f,
                   (void *) &y);
   pthread_join (th1, NULL);
   pthread_join (th2, NULL);
   printf ("\np: %d\n", p);
}
```

Programul 2.1 Sursa `primPThread.c`

Este vorba de crearea şi lansarea în execuție (folosind funcția `pthread_create`) a două thread-uri, ambele având aceeaşi acțiune, descrisă de funcția `f`, doar cu doi parametri diferiți. După ce se aşteaptă terminarea activității lor (folosind funcția `pthread_join`), se tipăreşte valoarea variabilei globale `p`. Rezultatele pentru cele patru variante sunt prezentate în tabelul următor. Pentru compilare, trebuie să fie specificată pe lângă sursă şi biblioteca `libpthread.so.0`. (sau, pe scurt, se specifică numele `pthread`, la opțiunea `-l`).

Tabelul din figura 4.6 ilustrează funcționarea programului în cele patru variante. In această secțiune ne vor interesa doar primele două coloane.

In continuare, o să explicăm, pe rând, elementele introduse de către acest program, făcând astfel primii paşi în lucrul cu thread-uri. Pentru o mai bună înțelegere, este bine ca din program să se "rețină" din sursă numai liniile ce rămân în urma parametrizării.

	ATRIBLUNG	MUTEX	ATRIBLUNG MUTEX
x: i=0 pas=-1	x: i=0 pas=-1	x: i=0 pas=-1	x: i=0 pas=-1
y: i=0 pas=2	y: i=0 pas=2	y: i=0 pas=2	y: i=0 pas=2
x: i=1 pas=-1	x: i=1 pas=-1	x: i=1 pas=-1	x: i=1 pas=-1
y: i=1 pas=2	y: i=1 pas=2	y: i=1 pas=2	y: i=1 pas=2
x: i=2 pas=-1	x: i=2 pas=-1	x: i=2 pas=-1	x: i=2 pas=-1
y: i=2 pas=2	x: i=3 pas=-1	y: i=2 pas=2	y: i=2 pas=2
x: i=3 pas=-1	x: i=4 pas=-1	x: i=3 pas=-1	x: i=3 pas=-1
y: i=3 pas=2	y: i=2 pas=2	y: i=3 pas=2	y: i=3 pas=2
x: i=4 pas=-1	x: i=5 pas=-1	x: i=4 pas=-1	x: i=4 pas=-1
y: i=4 pas=2	x: i=6 pas=-1	y: i=4 pas=2	y: i=4 pas=2
x: i=5 pas=-1	y: i=3 pas=2	x: i=5 pas=-1	x: i=5 pas=-1
x: i=6 pas=-1	x: i=7 pas=-1	y: i=5 pas=2	y: i=5 pas=2
y: i=5 pas=2	x: i=8 pas=-1	x: i=6 pas=-1	x: i=6 pas=-1
x: i=7 pas=-1	y: i=4 pas=2	y: i=6 pas=2	y: i=6 pas=2
x: i=8 pas=-1	x: i=9 pas=-1	x: i=7 pas=-1	x: i=7 pas=-1
x: i=9 pas=-1	x: i=10 pas=-1	y: i=7 pas=2	y: i=7 pas=2
x: i=10 pas=-1	x: i=11 pas=-1	x: i=8 pas=-1	x: i=8 pas=-1
x: i=11 pas=-1	x: i=12 pas=-1	y: i=8 pas=2	y: i=8 pas=2
x: i=12 pas=-1	y: i=5 pas=2	x: i=9 pas=-1	x: i=9 pas=-1
x: i=13 pas=-1	x: i=13 pas=-1	y: i=9 pas=2	y: i=9 pas=2
x: i=14 pas=-1	x: i=14 pas=-1	x: i=10 pas=-1	x: i=10 pas=-1
y: i=6 pas=2	x: i=15 pas=-1	x: i=11 pas=-1	x: i=11 pas=-1
y: i=7 pas=2	y: i=6 pas=2	x: i=12 pas=-1	x: i=12 pas=-1
x: i=15 pas=-1	x: i=16 pas=-1	x: i=13 pas=-1	x: i=13 pas=-1
x: i=16 pas=-1	x: i=17 pas=-1	x: i=14 pas=-1	x: i=14 pas=-1
x: i=17 pas=-1	x: i=18 pas=-1	x: i=15 pas=-1	x: i=15 pas=-1
x: i=18 pas=-1	x: i=19 pas=-1	x: i=16 pas=-1	x: i=16 pas=-1
y: i=8 pas=2	y: i=7 pas=2	x: i=17 pas=-1	x: i=17 pas=-1
x: i=19 pas=-1	y: i=8 pas=2	x: i=18 pas=-1	x: i=18 pas=-1
y: i=9 pas=2	y: i=9 pas=2	x: i=19 pas=-1	x: i=19 pas=-1
p: 0	p: 20	p: 0	p: 0

Figura 2.3 Comportări ale programului 4.1

Vom începe cu cazul în care nici una dintre constante nu este definită. In spiritul celor de mai sus, variabila `mutp` nu este practic utilizată, vom reveni asupra ei în secțiunile următoare. De asemenea, se vede că variabila `p` este modificată direct cu parametrul de intrare, fără a mai folosi ca şi intermediar variabila `x`.

Din conținutul programului se observă că funcția `f` primeşte ca argument o variabilă incluzând în structura ei: numele variabilei, numărul de iterații al funcției `f` şi pasul de incrementare al variabilei `p`. Se vede că cele două thread-uri primesc şi cedează relativ aleator controlul. Variabila `x` indică 20 iterații cu pasul –1, iar `y` 10 iterații cu pasul 2.

In absența constantei `ATRIBLUNG`, incrementarea lui `p` se face printr-o singură instrucțiune, `p+=a->pas`. Este extrem de puțin probabil ca cele două thread-uri să-şi treacă controlul de la unul la altul exact în timpul execuției acestei instrucțiuni. In consecință, variabila `p` rămâne în final cu valoarea 0.

Prezența constantei `ATRIBLUNG` are menirea să "încurce" lucrurile. Se vede că incrementarea variabilei `p` se face prin intermediul variabilei locale `x`. Astfel, după ce reține în `x` valoarea lui `p`, threadul stă în aşteptare un număr aleator de secunde şi abia după aceea creşte `x` cu valoarea `a->pas` şi apoi atribuie lui `p` noua valoare. In

mod natural, în timpul aşteptării celălalt thread devine activ şi îşi citeşte şi el aceeaşi valoare pentru `p` şi prelucrarea continuă la fel. Bineînțeles, aceste incrementări întrețesute provoacă o mărire globală incorectă a lui `p`, motiv pentru care `p` final rămâne cu valoarea 20. (De fapt, acest scenariu concret de aşteptări are drept consecință faptul că efectul global coincide cu efectul celui de-al doilea thread.)

2.2.3 Instrumente standard de sincronizare

Instrumentele (obiectele) de sincronizare specifice thread-urilor sunt, aşa cum am arătat în 2.5: variabilele mutex, variabilele condiționale, semafoarele şi blocările cititor/scriitor (reader/writer). Fiecare variabilă de sincronizare are asociată o coadă de thread-uri care aşteaptă - *sunt blocate*- la variabila respectivă.

Cu ajutorul unor primitive ce verifică dacă variabilele de sincronizare sunt disponibile, fiecare thread blocat va fi "trezit" la un moment dat, va fi şters din coada de aşteptare şi controlul lor va fi cedat componentei de planificare. Trebuie remacat faptul că thread-urile sunt repornite într-o ordine arbitrară, neexistând nici o relație între ordinea în care au fost blocate şi elimnarea lor din coada de aşteptare.

O observație importantă! In situația în care un obiect de sincronizare este blocat de un thread, iar acest thread îşi încheie fără a debloca obiectul, acesta - obiectul de sincronizare - va rămâne blocat! Este deci posibil ca thread-urile blocate la obiectul respectiv vor intra în impas.

In continuare descriem primitivele de lucru cu aceste obiecte (variabile, entități) de sincronizare pe platforme Unix, atât sub Posix, cât şi sub Solaris.

2.2.3.1 Operații cu variabile mutex

Inițializarea unei variabile mutex se poate face static sau dinamic, astfel:
Posix, inițializare statică:

```
pthread_mutex_t numeVariabilaMutex = PTHREAD_MUTEX_INITIALIZER;
```

Posix, inițializare dinamică:

```
int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t
*mutexattr);
```

Solaris, inițializare statică:

```
mutex_t numeVariabilaMutex = 0;
```

Solaris, inițializare dinamică:

```
int mutex_init(mutex_t *mutex, int type, void *arg);
```

Deci, inițializarea statică presupune atribuirea unei valori standard variabilei mutex. Inițializarea dinamică se face apelând o funcție de tip `init`, având ca prim argument un pointer la variabila mutex.

Apelul `pthread_mutex_init` inițializează variabila mutex cu atribute specificate prin parametrul `mutexattr`. Semnificația acestei variabile o vom

prezenta într-o secțiune ulterioară. Pe moment acest argument are valoarea NULL, ceea ce semnifică fixarea de atribute implicite.

Parametrul `type` din apelul `mutex_init` indică domeniul de vizibilitate al variabilei mutex. Dacă are valoarea USYNC_PROCESS, atunci ea poate fi accesată din mai multe procese. Dacă are valoarea USYNC_THREAD, atunci variabila este accesibilă doar din procesul curent. Argumentul `arg` este rezervat pentru dezvoltări ulterioare, deci singura valoare permisă este NULL.

Distrugerea unei variabile mutex înseamnă eliminarea acesteia şi eliberarea resurselor ocupate de ea. In prealabil, variabila mutex trebuie să fie deblocată. Apelurile de distrugere sunt:
Posix:
```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```
Solaris:
```
int mutex_destroy(mutex_t *mutex);
```

Blocarea unei variabile mutex:
Posix:
```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock (pthread_mutex_t *mutex);
```
Solaris:
```
int mutex_lock(mutex_t *mutex);
int mutex_trylock(mutex_t *mutex);
```

In apelurile `lock`, dacă variabila `mutex` nu este blocată de alt thread, atunci ea va deveni proprietatea threadului apelant şi funcția returnează imediat. Dacă este deja blocată de un alt thread, atunci funcția intră în aşteptare până când variabila `mutex` va fi eliberată.

In apelurile `trylock`, funcțiile returnează imediat, indiferent dacă variabila `mutex` este sau nu blocată de alt thread. Dacă variabila este liberă, atunci ea va deveni proprietatea threadului. Dacă este blocată de un alt thread (sau de threadul curent), funcția returnează imediat cu codul de eroare EBUSY.

Deblocarea unei variabile `mutex` se realizează prin:
Posix:
```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```
Solaris:
```
int mutex_unlock(mutex_t *mutex);
```

Se presupune că variabila `mutex` a fost blocată de către threadul care apelează funcția de deblocare.

Este momentul să analizăm rezultatele din coloanele 3 şi 4 ale tabelului 4.6. In ambele variante accesul la variabila `p` este exclusiv, fiind protejat de variabila `mutp`. Se observă că `p` final are valoarea corectă. De asemenea, indiferent de faptul că `ATRIBLUNG` este sau nu definită (ceea ce diferențiază cele două cazuri), se observă o mare regularitate în succesiunea la control a thread-urilor.

Intrebarea naturală care se pune este “ce se întamplă dacă mutex-ul este deja blocat de threadul curent?”. Răspunsul diferă de la platformă la platformă. Programul 4.2

prezintă o situație bizară, cititorul poate să-l testeze, dar să nu-l utilizeze în aplicații!:)

```
#include <synch.h>
#include <thread.h>
mutex_t mut;
thread_t t;
void* f(void* a) {
   mutex_lock(&mut);
   printf("lin 1\n");
   mutex_lock(&mut); // 1
   printf("lin 2\n");
   mutex_unlock(&mut);
   mutex_lock(&mut);
}
main() {
   mutex_init(&mut,USYNC_THREAD,NULL);
   thr_create(NULL,0,f,NULL,THR_NEW_lwp,&t);
   thr_join(t,NULL,NULL);
}
```

Programul 2.2 Un (contra)exemplu: sursa `dublaBlocareMutex.c`

Punctăm ca observație faptul că, pe Solaris, execuția acestui program produce impas în punctul `// 1`.

2.2.3.2 Operații cu variabile condiționale

Orice variabilă condițională aşteaptă un anumit eveniment. Ea are asociată o variabilă mutex şi un predicat. Predicatul conține condiția care trebuie să fie îndeplinită pentru a apărea evenimentul, iar variabila mutex asociată are rolul de a proteja acest predicat. Scenariul de aşteptare a evenimentului pentru care există variabila condițională este:

```
Blochează variabila mutex asociată
Câttimp (predicatul este fals)
   Aşteaptă la variabila condițională
Execută eventuale acțiuni
Deblochează variabila mutex
```

Este de remarcat faptul că pe durata aşteptării la variabila condițională, sistemul eliberează variabila mutex asociată. In momentul în care se semnalizează îndeplinirea condiției, înainte de ieşirea threadului din aşteptare, i se asociază din nou variabila mutex. Prin aceasta se permit în fapt două lucruri: (1) să se aştepte la condiție, (2) să se actualizeze predicatul şi să se semnalizeze apariția evenimentului.

Scenariul de semnalare - *notificare* - a apariției evenimentului este:

```
Blochează variabila mutex asociată
Fixează predicatul la true
Semnalizează apariția evenimentului
      la variabila condițională
      pentru a trezi thread-urile ce aşteaptă
Deblochează variabila mutex.
```

Inițializarea unei variabile conditionale se poate face static sau dinamic, astfel:

Posix, inițializare statică:
```
pthread_cond_t numeVariabilaCond = PTHREAD_COND_INITIALIZER;
```
Posix, inițializare dinamică:
```
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t
*condattr);
```
Solaris, inițializare statică:

```
cond_t numeVariabilaCond = DEFAULTCV;
```
Solaris, inițializare dinamică:
```
int cond_init(cond_t *cond, int type, void *arg);
```

Deci, inițializarea statică presupune atribuirea unei valori standard variabilei condiționale. Inițializarea dinamică se face apelând o funcție de tip `init`, având ca prim argument un pointer la variabila condițională.

Apelul `pthread_cond_init` inițializează variabila condițională cu atribute specificate prin parametrul `condattr`. Semnificația acestei variabile o vom prezenta într-o secțiune ulterioară. Pe moment acest argument are valoarea NULL, ceea ce semnifică fixarea de atribute implicite.

Parametrul `type` din apelul `cond_init` indică domeniul de vizibilitate al variabilei condiționale. Dacă are valoarea USYNC_PROCESS, atunci ea poate fi accesată din mai multe procese. Dacă are valoarea USYNC_THREAD, atunci variabila este accesibilă doar din procesul curent. Argumentul `arg` este rezervat pentru dezvoltări ulterioare, deci singura valoare permisă este NULL.

Distrugerea unei variabile condiționale înseamnă eliminarea acesteia şi eliberarea resurselor ocupate de ea. In prealabil, variabila mutex trebuie să fie deblocată. Apelurile de distrugere sunt:
Posix:
```
int pthread_cond_destroy(pthread_cond_t *cond);
```
Solaris:
```
int cond_destroy(cond_t *cond);
```

Operația de aşteptare
Posix:
```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t
*mutex,
      struct timespec*timeout);
```
Solaris:
```
int cond_timedwait(cond_t *cond, mutex_t *mutex, timestruc_t
*timeout);
```

Inainte de apelul funcțiilor de aşteptare (funcții de tip `wait`), se cere blocarea variabilei `mutex`, asociată variabilei condiționale `cond`. După apelul unei funcții de tip `wait`, se eliberează variabila `mutex` şi se suspendă execuția threadului până când condiția aşteptată este îndeplinită, moment în care variabila `mutex` este blocată din nou. Parametrul `timeout` furnizează un interval maxim de aşteptare. Dacă condiția nu apare (evenimentul nu se produce) în intervalul specificat, aceste funcții returnează un cod de eroare.

Operația de notificare

Posix:
```
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```
Solaris:
```
int cond_signal(cond_t *cond);
int cond_broadcast(cond_t *cond);
```

Funcțiile de tip `signal` anunță îndeplinirea condiției după care se așteaptă la `cond`. Dacă nici un thread nu se afla în așteptare, atunci nu se întâmplă nimic. Dacă sunt mai multe thread-uri interesate, numai unul singur dintre acestea își va relua execuția. Alegerea threadului care va fi "trezit" depinde de implementare, de priorități și de politica de planificare. In cazul Solaris, thread-urile legate sunt prioritare celor multiplexate pe lwp-uri.

Funcțiile de tip `broadcast` repornesc toate thread-urile care așteaptă la `cond`.

In continuare vom prezenta un exemplu simplu, programul 4.3 `ptVarCond.c`. El descrie trei thread-uri. Două dintre ele, ambele descrise de funcția `inccontor`, incrementează de câte 7 ori varibila `contor`. Al treilea thread, descris de funcția `watchcontor`, așteaptă evenimentul ca variabila `contor` să ajungă la valoarea 12 și semnalizează acest lucru.

```
#include <pthread.h>
   int contor = 0;
   pthread_mutex_t mutcontor = PTHREAD_MUTEX_INITIALIZER;
   pthread_cond_t condcontor = PTHREAD_COND_INITIALIZER;
   int thid[3] = { 0, 1, 2 };

   void incContor (int *id) {
      int i;  printf ("\nSTART incContor %d\n", *id);
      for (i = 0; i < 7; i++) {
         sleep(random() % 3);
         pthread_mutex_lock (&mutcontor);
         contor++;
         printf("\n incContor: thread %d contor vechi %d contor nou
%d",
               *id, contor - 1, contor);
         if (contor == 12)
            pthread_cond_signal (&condcontor);
         pthread_mutex_unlock (&mutcontor);
      }
      printf ("\nSTOP incContor %d\n", *id);
   }

   void verifContor (int *id) {
      printf ("\nSTART verifContor \n");
      pthread_mutex_lock (&mutcontor);
      while (contor <= 12) {
         pthread_cond_wait (&condcontor, &mutcontor);
         printf  ("\n  verifContor:  thread  %d  contor  %d",  *id,
contor);
         break;
      }
      pthread_mutex_unlock (&mutcontor);
      printf ("\nSTOP verifContor \n");
   }

   main () {
```

```
      pthread_t th[3];
      int i;
   //creaaza cele 3 thread-uri
      pthread_create ((pthread_t *) & th[0], NULL,
                      (void *) verifContor, &thid[0]);
      pthread_create ((pthread_t *) & th[1], NULL,
                      (void *) incContor, &thid[1]);
      pthread_create ((pthread_t *) & th[2], NULL,
                      (void *) incContor, &thid[2]);
   //asteapta terminarea thread-urilor
      for (i = 0; i < 3; i++)
         pthread_join (th[i], NULL);
   }
```

Programul 2.3 Sursa `ptVarCond.c`

Rezultatul execuției programului este următorul:

```
START verifContor

START incContor 1

START incContor 2

 incContor: thread 2 contor vechi 0 contor nou 1
 incContor: thread 2 contor vechi 1 contor nou 2
 incContor: thread 1 contor vechi 2 contor nou 3
 incContor: thread 2 contor vechi 3 contor nou 4
 incContor: thread 1 contor vechi 4 contor nou 5
 incContor: thread 2 contor vechi 5 contor nou 6
 incContor: thread 2 contor vechi 6 contor nou 7
 incContor: thread 2 contor vechi 7 contor nou 8
 incContor: thread 1 contor vechi 8 contor nou 9
 incContor: thread 2 contor vechi 9 contor nou 10
STOP incContor 2

 incContor: thread 1 contor vechi 10 contor nou 11
 incContor: thread 1 contor vechi 11 contor nou 12
 verifContor: thread 0 contor 12
STOP verifContor

 incContor: thread 1 contor vechi 12 contor nou 13
 incContor: thread 1 contor vechi 13 contor nou 14
STOP incContor 1
```

2.2.3.3 Operații cu semafoare

Spre deosebire de utilizarea semafoarelor Unix la nivel de proces descrise în 3.4, semafoarele thread sunt mai simplu de utilizat, însă mai potrivite în interiorul aceluiaşi proces.

Inițializare

Posix:

```
int sem_init(sem_t *sem, int type, int v0);
```

Solaris:

```
int sema_init(sema_t *sem, int v0, int type, void *arg);
```

Se inițializează semaforul `sem` cu valoarea inițială `v0`. Parametrul `type` din apelul Posix este 0 dacă semaforul este o resursă locală procesului şi diferit de 0, dacă semaforul poate fi partajat de mai multe procese. în cazul thread-urilor Linux, valoarea este întotdeauna 0.

Acelaşi `type` din apelul Solaris are valorile posibile: USYNC_THREAD pentru resursă locală sau USYNC_PROCESS pentru partajarea între procese. Parametrul `arg` are obligatoriu valoarea NULL.

Se observă că pe Solaris este posibilă şi inițializarea statică a semaforului, prin atribuirea valorii inițiale - obligatoriu - 0. In acest caz, semaforul este de tipul USYNC_THREAD.

Distrugerea semaforului
Posix:
```
int sem_destroy(sem_t * sem);
```
Solaris:
```
int sema_destroy(sema_t *sem);
```

Folosind acest apel, sunt eliberate resursele ocupate de semaforul `sem`. Dacă semaforul nu are asociate resurse sistem, funcțiile `destroy` nu fac altceva decât să verifice dacă există thread-uri care aşteaptă la semafor. In acest caz funcția returnează eroarea EBUSY. In caz de semafor invalid, întoarce eroarea EINVAL.

Incrementarea valorii semaforului (echivalentul operației V - vezi 2.5.1)
Posix:
```
int sem_post(sem_t * sem);
```
Solaris:
```
int sema_post(sema_t *sem);
```

Funcțiile `sem_post`, respectiv `sema_post` incrementează cu 1 valoarea semaforului `sem`. Dintre thread-urile blocate, planificatorul scoate unul şi-l reporneşte. Alegerea threadului restartat depinde de parametrii de planificare.

Decrementarea valorii semaforului (echivalentul operației P - vezi 2.5.1)

Posix:
```
int sem_wait(sem_t * sem);
int sem_trywait(sem_t * sem);
```
Solaris:
```
int sem_wait(sema_t *sem);
int sem_trywait(sema_t *sem);
```

Funcțiile `sem_wait/sema_wait` suspendă execuția threadului curent până când valoarea semaforului `sem` devine mai mare decât 0, după care decrementează atomic valoarea respectivă. Funcțiile `sem_trywait/sema_trywait` sunt variantele fără blocare ale funcțiilor `sem_wait/ sema_wait`. Dacă semaforul nu are valoarea 0, valoarea acestuia este decrementată, altfel, funcțiile se termină cu codul de eroare EAGAIN.

In varianta Posix, există apelul:

```
int sem_getvalue(sem_t * sem, int *sval);
```

care depune valoarea curentă a semaforului `sem` în locația indicată de pointerul `sval`.

2.2.3.4 Blocare de tip cititor / scriitor (reader / writer)

Aceste obiecte (implementate atât în cazul thread-urilor Posix, cât şi Solaris) sunt folosite pentru a permite mai multor thread-uri să acceseze, la un moment dat, o resursă partajabilă: fie în citire de către oricâte thread-uri, fie numai de un singur thread care să o modifice.

Inițializare

Posix:

```
int pthread_rwlock_init(pthread_rwlock_t *rwlock,
      pthread_rwlockattr_t *rwlockattr);
```

Solaris:

```
int rwlock_init(rwlock_t *rwlock, int type, void *arg);
```

Se inițializează obiectul `rwlock`. Parametrul `rwlockattr` din apelul Posix este purtătorul de atribute al obiectului `rwlock`. Parametrul `type` din apelul Solaris are valorile posibile: USYNC_THREAD pentru resursă locală sau USYNC_PROCESS pentru partajarea între procese. Parametrul `arg` are obligatoriu valoarea NULL.

Distrugerea obiectului reader/writer

Posix:

```
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

Solaris:

```
int rwlock_destroy(rwlock_t *rwlock);
```

Sunt eliberate resursele ocupate de obiectul `rwlock`.

Operația de blocare pentru citire presupune incrementarea numărului de cititori, dacă nici un scriitor nu a blocat sau nu aşteaptă la obiectul reader/writer. In caz contrar, funcțiile `lock` intră în aşteptare până când obiectul devine disponibil, iar funcțiile `trylock` întorc imediat cu cod de eroare. Apelurile pentru această operație sunt:

Posix:

```
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
```

Solaris:

```
int rw_rdlock(rwlock_t *rwlock);
int rw_tryrdlock(rwlock_t *rwlock);
```

Operația de blocare pentru scriere are rolul de a obține obiectul reader/writer dacă nici un thread nu l-a blocat în citire sau scriere. In caz contrar, fie se aşteaptă eliberarea obiectului In cazul funcțiilor `wrlock`, fie întoarce imediat cu cod de eroare în cazul funcțiilor `wrtrylock`. Apelurile pentru această operație sunt:

Posix:

```
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```

Solaris:

```
int rw_wrlock(rwlock_t *rwlock);
int rw_trywrlock(rwlock_t *rwlock);
```

2.2.4 Exemple de sincronizări

Ca exemplu de folosire a acestor mecanisme, vom rezolva o problemă generală de sincronizare: `m` thread-uri accesează `n` resurse (`m>n`) în mai multe moduri, care conduc în final la rezultate echivalente.
Pentru a trata această situație, rezolvăm o problema concretă "`nrTr` intră într-o gară prin `nrLin` linii, `nrTr>nrLin`", folosind diverse tehnici de sincronizare: semafoare, variabile mutex, etc.

1. Implementare sub Unix, folosind semafoare Posix, programul 4.4.

```
#include <semaphore.h>
#include <pthread.h>
#include <stdlib.h>
#define nrLin 5
#define nrTr 13
   pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
   sem_t sem;
   int poz[nrTr];
   pthread_t tid[nrTr];

//afisare trenuri care intra in gara
   void afisare() {
      int i;
      pthread_mutex_lock(&mutex);
      printf("Trenuri care intra in gara:");
      for (i=0;i<nrTr;i++)
         if (poz[i]==1)
            printf(" %d",i);
      printf("\n");
      pthread_mutex_unlock(&mutex);
   }

//rutina unui thread
   void* trece(char* sind){
      int sl,ind;
      ind=atoi((char*)sind);
      sem_wait(&sem);
      poz[ind]=1;
      afisare();
      sl=1+(int) (3.0*rand()/(RAND_MAX+1.0));
      sleep(sl);
      poz[ind]=2;
      sem_post(&sem);
      free(sind);
   }

//main
   main(int argc, char* argv[]) {
      char* sind;
      int i;
      sem_init(&sem,0,nrLin);
      for (i=0;i<nrTr;i++) {
         sind=(char*) malloc(5*sizeof(char));
```

```
         sprintf(sind,"%d",i);
         pthread_create(&tid[i],NULL,trece,sind);
      }
      for (i=0;i<nrTr;i++)
         pthread_join(tid[i],NULL);
   }
```

Programul 2.4 Sursa `trenuriSemPosix.c`

2. Implementare sub Unix folosind variabile mutex şi variabile condiționale, programul 4.5.

```
#include <stdlib.h>
#include <pthread.h>
#include <errno.h>
#define nrLin 5
#define nrTr 13
   pthread_mutex_t semm[nrLin];
   pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
   pthread_mutex_t mutexc=PTHREAD_MUTEX_INITIALIZER;
   pthread_cond_t condElm=PTHREAD_COND_INITIALIZER;
   int poz[nrTr];
   pthread_t tid[nrTr];

   int semmutex_lock() {
      int i;
      while (1) {
         for (i=0;i<nrLin;i++)
            if (pthread_mutex_trylock(&semm[i])!=EBUSY)
               return i;
         pthread_mutex_lock(&mutexc);
         pthread_cond_wait(&condElm,&mutexc);
         pthread_mutex_unlock(&mutexc);
      }
   }

   int semmutex_unlock(int i) {
      pthread_mutex_unlock(&semm[i]);
      pthread_cond_signal(&condElm);
   }

//afisare trenuri care intra in gara
   void afisare() {
      int i;
      pthread_mutex_lock(&mutex);
      printf("Trenuri care intra in gara:");
      for (i=0;i<nrTr;i++)
         if (poz[i]==1)
            printf(" %d",i);
      printf("\n");
      pthread_mutex_unlock(&mutex);
   }

//rutina unui thread
   void* trece(char* sind){
      int sl,ind,indm;
      ind=atoi((char*)sind);
      indm=semmutex_lock();
      poz[ind]=1;
      afisare();
      sl=1+(int) (3.0*rand()/(RAND_MAX+1.0));
```

```
      sleep(sl);
      poz[ind]=2;
      semmutex_unlock(indm);
      free(sind);
   }

//main
   main(int argc, char* argv[]) {
      char* sind;
      int i;
      for (i=0;i<nrLin;i++)
      //semm[i]=PTHREAD_MUTEX_INITIALIZER;
         pthread_mutex_init(&semm[i],NULL);
      for (i=0;i<nrTr;i++) {
         sind=(char*) malloc(5*sizeof(char));
         sprintf(sind,"%d",i);
         pthread_create(&(tid[i]),NULL,trece,sind);
      }
      for (i=0;i<nrTr;i++)
         pthread_join(tid[i],NULL);
   }
```

Programul 2.5 Sursa `trenuriMutexCond.c`

Programele 4.4 şi 4.5 conduc la rezultate de execuție similare.
Exemplu rezultat execuție pentru nrLin=5 şi nrTr=13.

```
Trenuri care intra in gara: 0
Trenuri care intra in gara: 0 1
Trenuri care intra in gara: 0 1 2
Trenuri care intra in gara: 0 1 2 3
Trenuri care intra in gara: 0 1 2 3 4
Trenuri care intra in gara: 0 2 3 4 5
Trenuri care intra in gara: 5 8
Trenuri care intra in gara: 5 7 8
Trenuri care intra in gara: 5 6 7 8
Trenuri care intra in gara: 5 6 7 8 9
Trenuri care intra in gara: 6 7 8 9 10
Trenuri care intra in gara: 7 8 9 10 11
Trenuri care intra in gara: 7 10 11 12
```

Se observă că la un moment dat intră în gară maximum 5 trenuri (câte linii sunt), iar execuția programului se încheie după ce toate cele 13 trenuri au intrat în gară.

2.2.5 Obiecte purtătoare de atribute Posix

In secțiunea legată de instrumentele standard de sincronizare (4.3.3), am amânat prezentarea unui anumit parametru şi am promis că vom reveni asupra lui. Este vorba de:

- apelul `pthread_create` (4.3.2.1), argumentul pointer la tipul `pthead_attr_t` pe care l-am numit `attr`;
- apelul `pthread_mutex_init` (4.3.3.1), argumentul pointer la tipul `pthead_mutexattr_t` pe care l-am numit `mutexattr`;
- apelul `pthread_cond_init` (4.3.3.2), argumentul pointer la tipul `pthead_condattr_t` pe care l-am numit `condattr`;

- apelul `pthread_rwlock_init` (4.3.3.4), argumentul pointer la tipul `pthead_rwlockattr_t` pe care l-am numit `rwlockattr`;

In exemplele de până acum am folosit pentru aceste argumente valoarea NULL, lăsând sistemul să fixeze valori implicite.

O variabilă având unul dintre tipurile enumerate mai sus este numită, după caz, *obiect purtător de atribute: thread, mutex, cond, rwlock*. Un astfel de obiect păstrează o serie de caracteristici, fixate de către programator. Aceste caracteristici trebuie transmise în momentul creării sau inițializării threadului sau obiectului de sincronizare respectiv. In succesiune logică, crearea unui astfel de obiect precede crearea threadului sau obiectului de sincronizare care va purta aceste atribute.

Numai modelul Posix utilizează acest mecanism. Modelul Solaris include aceste atribute printre parametrii de creare a threadului, respectiv de inițializare a obiectului de sincronizare respectiv.

Obiectele purtătoare de atribute au avantajul că Imbunătățesc gradul de portabilitate a codului. Apelul de creare a unui thread / inițializarea unui obiect de sincronizare rămâne acelaşi, indiferent de modul cum este implementat obiectul purtător de atribute. Un alt avantaj este faptul ca un obiect purtător de atribute se inițializează simplu, o singură dată şi poate fi folosit la crearea mai multor thread-uri. La terminarea thread-urilor, trebuie eliberată memoria alocată pentru obiectele purtătoare de atribute.

Asupra unui obiect purtător de atribute, indiferent care dintre cele trei de mai sus, se pot efectua operațiile:

- Inițializare (operație `init`)
- Distrugere (operație `destroy`)
- Setarea valorilor unor atribute (operație `set`)
- Obținerea valorilor unor atribute (operație `get`)

2.2.5.1 Inițializarea şi distrugerea unui obiect atribut

Mecanismul de inițializare este foarte asemănător la aceste tipuri de obiecte. Mai întâi este necesară, după caz, declararea unei variabile (vom folosi aceleaşi nume ca în prototipurile din 4.3.2):

```
pthread_attr_t                attr;
pthread_mutexattr_t           mutexattr;
pthread_condattr_t            condattr;
pthread_rwlockattr_t    rwlockattr;
```

Apoi se transmite adresa acestei variabile prin apelul sistem corespunzător:

```
int pthread_attr_init(pthread_attr_t &attr);
int pthread_mutexattr_init(pthread_mutexattr_t &mutexattr);
int pthread_condattr_init(pthread_condattr_t &condattr);
int pthread_rwlockattr_init(pthread_rwlockattr_t &rwlockattr);
```

Distrugerea se face, după caz, folosind unul dintre apelurile sistem:

```
int pthread_attr_destroy(pthread_attr_t &attr);
int pthread_mutexattr_destroy(pthread_mutexattr_t &mutexattr);
int pthread_condattr_destroy(pthread_condattr_t &condattr);
int pthread_rwlockattr_destroy(pthread_rwlockattr_t &rwlockattr);
```

2.2.5.2 Gestiunea obiectelor purtătoare de atribute thread

Unui thread i se pot fixa, printre altele, o serie de atribute privind politica de planificare, atribute de moştenire, componenta care gestionează threadul, priorități şi caracteristici ale stivei proprii. In apelurile sistem care urmează, vom nota cu `attr` referința la un obiect purtător de atribut, iar prin `p` parametrul specific atributului.

Pentru fixarea politicii de planificare este folosit apelul sistem:

```
int pthread_attr_setpolicy(pthread_condattr_t *attr, int p);
int pthread_attr_getpolicy(pthread_condattr_t *attr, int *p);
```

Politica `p`, atribuită obiectului referit prin `attr`, este specificată (`set`) prin una din următoarele constante:

- `SCHED_FIFO` dacă se doreşte planificarea primul venit – primul servit
- `SCHED_RR` dacă se doreşte planificarea "Round-Robin" (servire circulară a fiecăruia câte o cuantă de timp).
- `SCHED_OTHERS` dacă se doreşte o anumită politică specială (nu ne ocupăm de ea).

Pentru a se cunoaşte politica fixată (`get`) dintr-un obiect atribut `attr`, aceasta se depune în întregul indicat de pointerul `p`.

Fixarea moştenirii politicii de planificare se face prin:

```
int pthread_attr_setinheritsched(pthread_attr_t *attr, int p);
int pthread_attr_getinheritsched(pthread_attr_t *attr, int *p);
```

La `set`, `p` poate avea valorile:

- `PTHREAD_INHERIT_SCHED` politica de planificare este moştenită de la procesul creator.
- `PTHREAD_EXPLICIT_SCHED` politica trebuie specificată explicit.

Valoarea tipului de moştenire fixat se obține prin `get` în `p`.

Fixarea domeniului de vizibilitate a threadului: Componenta care gestionează thread-urile nou create poate fi ori procesul curent, ori nucleul sistemului. Pentru a specifica unul dintre ele se utilizează apelul sistem:

```
int pthread_attr_setscope(pthread_attr_t *attr, int p);
int pthread_attr_getscope(pthread_attr_t *attr, int *p);
```

La `set`, `p` poate avea una dintre valorile:

- `PTHREAD_SCOPE_PROCESS` - thread user, local procesului.
- `PTHREAD_SCOPE_SYSTEM` - thread nucleu.

Obținerea valorii curente se face prin `get`, depunând valoarea în întregul punctat de `p`.

Fixarea statutului unui thread în momentul terminării acțiunii lui se face folosind:

```
int pthread_attr_setdetachstate(pthread_attr_t *attr, int p);
int pthread_attr_getdetachstate(pthread_attr_t *attr, int *p);
```

La `set`, parametrul `p` indică acest statut prin una dintre valorile:

- `PTHREAD_CREATE_DETACHED` - threadul este distrus la terminare.
- `PTHREAD_CREATE_JOINABLE` - threadul este păstrat după terminare.

Statutul unui thread se obține în variabila indicată de `p` prin metota `get`.

Parametrii stivei unui thread pot fi manevrați prin apelurile:

```
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *p);
int pthread_attr_setstacksize(pthread_attr_t *attr, int p);
int pthread_attr_getstackaddr(pthread_attr_t *attr, void **p);
int pthread_attr_getstacksize(pthread_attr_t *attr, int *p);
```

Este vorba de fixarea adresei stivei şi a alungimii acesteia, respectiv de obținerea acestor valori.

Fixarea unei priorități o vom prezenta într-o secțiune destinată special planificării thread-urilor.

2.2.5.3 Gestiunea obiectelor purtătoare de atribute mutex

Fixarea protocolului de acces la mutex:

```
int pthread_mutexattr_setprotocol(pthread_attr_t *mutexattr, int p);
int  pthread_mutexattr_getprotocol(pthread_attr_t  *mutexattr,  int
*p);
```

Pentru `set`, parametrul `p` poate lua valorile:

- `PTHREAD_PRIO_INHERIT`, dacă prioritatea este moştenită de la threadul creator.
- `PTHREAD_PRIO_NONE`, dacă nu se foloseşte nici un protocol de prioritate.
- `PTHREAD_PRIO_PROTECT`, dacă se foloseşte un protocol explicit.

Obținerea valorii setate se face prin metoda `get`, care depune valoarea în `p`.

Fixarea domeniului de utilizare:

```
int  pthread_mutexattr_setpshared(pthread_attr_t  *mutexattr,
int p);
int  pthread_mutexattr_getpshared(pthread_attr_t  *mutexattr,
int *p);
```

La set, parametrul `p` poate lua valorile:

- PTHREAD_PROCESS_PRIVATE, dacă se foloseşte numai în procesul curent.
- PTHREAD_PROCESS_SHARED, dacă se foloseşte şi în alte procese.

Valoarea de partajare se obține prin `get`, care depune valoarea în `p`.

2.2.5.4 Gestiunea obiectelor purtătoare de atribute pentru variabile condiționale

O variabilă condițională se poate folosi nu numai în procesul curent, ci şi în alte procese. Această modalitate de partajare este gestionată prin apelurile:

```
int pthread_condattr_setpshared(pthread_attr_t *condattr, int
p);
int pthread_condattr_getpshared(pthread_attr_t *condattr, int
*p);
```

La set, parametrul `p` poate lua valorile:

- PTHREAD_PROCESS_PRIVATE, dacă variabila condițională se va folosi numai în procesul curent.
- PTHREAD_PROCESS_SHARED, dacă ea se va folosi şi în alte procese.

Valoarea de partajare se obține în `p`, prin metoda `get`.

2.2.5.5 Purtătoare de atribute pentru obiecte partajabile reader / writer

Gestiunea obiectelor purtătoare de atribute include operațiile obişnuite: inițializare, distrugere, setare / obținere atribute. Aceste atribute sunt: partajarea obiectului inter-procese şi alte alte caracterisitci specifice reader/writer. Prototipurile funcțiilor API corespunzătoare se găsesc în fişierul antet `pthread.h`, dar, din pacate, manualele Unix nu includ documentații specifice acestor apeluri.

2.2.6 Planificarea thread-urilor sub Unix

Există, în principiu, trei politici de planificare a thread-urilor, desemnate prin trei constante specifice:

- SCHED_OTHER, sau echivalent SCHED_TS politică implicită time-sharring, non real-time.
- SCHED_RR (round-robin) planificare circulară, preemptivă, politică real-time: sistemul de operare întrerupe execuția la cuante egale de timp şi dă controlul altui thread.
- SCHED_FIFO (first-in-first-out) planificare cooperativă, threadul în execuție decide cedarea controlului spre următorul thread.

Pentru fixarea politicilor real-time este nevoie ca procesul să aibă privilegiile de superuser [17, 110].

Atât sub Posix, incluzând aici Linux cât şi pe Solaris există funcții specifice pentru modificarea priorității thread-urilor după crearea acestora.

2.2.6.1 Gestiunea priorităților sub Posix

Modelul Posix foloseşte în acest scop obiectele purtătoare de atribute ale threadului, despre care am vorbit în 4.3.5. Este vorba de o funcție de tip `set` care fixează prioritatea şi de o funcție de tip `get` care obține valoarea acestei priorități:

```
int pthread_attr_setschedparam(pthread_attr_t *attr,
      struct sched_param *p);
int pthread_attr_getschedparam(pthread_attr_t *attr,
      struct sched_param *p);
```

Structura `sched_param` este:

```
struct sched_param {
      int sched_priority;
}
```

Câmpul `sched_priority` conține valoarea curentă a priorității.

Pentru thread-uri sunt fixate 32 nivele de priorități. Valorile concrete ale numerelor de prioritate nu sunt nişte numere prefixate, ci depind de implementare. Pentru a le putea manevra, utilizatorul trebuie să obțină mai întâi valorile extreme, după care să stabilească el, în mod proporțional, cele 32 de valori ale priorităților. Valorile extreme se obțin prin apelurile:

```
sched_get_priority_max(SCHED_FIFO);
sched_get_priority_min(SCHED_FIFO);
sched_get_priority_max(SCHED_RR);
sched_get_priority_min(SCHED_RR);
```

2.2.6.2 Gestiunea priorităților sub Solaris

Prioritatea implicită pentru thread-urile multiplexate (libere), este 63. Priorităţile thread-urilor legate sunt stabilite de sistem.

Pentru modificarea/obținerea priorității unui thread se pot folosi şi funcțiile:

```
int thr_setprio(thread_t tid, int prio);
int thr_getprio(thread_t tid, int *prio);
void thr_get_rr_interval(timestruc_t *rr_time);
```

Ultima funcție se foloseşte doar pentru politica de planificare round-robin, care furnizează în structura punctată de `rr_time` intervalul de timp în milisecunde şi nanosecunde de aşteptare pentru fiecare thread înainte de a fi planificat.

2.2.7 Problema producătorilor şi a consumatorilor

Prezentăm, în programul 4.11, soluția problemei producătorilor şi consumatorilor, problemă enunțată în capitolul 1.4. Această soluție este implementată folosind thread-uri Posix.

```
#include <pthread.h>
#define MAX 10
#define MAX_NTHR 20
#include <stdlib.h>

   int recip[MAX];
   int art=0;
   pthread_t tid[MAX_NTHR];
   pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
   pthread_cond_t condPut=PTHREAD_COND_INITIALIZER;
   pthread_cond_t condGet=PTHREAD_COND_INITIALIZER;
   int P=5, C=5;
   int p[5],c[5];
   int pozPut,pozGet;

//afiseaza starea curenta a producatorilor si a consumatorilor
   void scrie() {
      int i;
      for (i=0;i<P;i++)
         printf("P%d_%d ",i,p[i]);
      for (i=0;i<C;i++)
         printf("C%d_%d ",i,c[i]);
      printf("B: ");
      for (i=0;i<MAX;i++)
         if (recip[i]!=0)
            printf("%d ",recip[i]);
      printf("\n");
   }

//verifica daca buferul este plin
   int plin() {
   //pthread_mutex_lock(&mutex);
      if (recip[pozPut]!=0)
         return 1;
      else
         return 0;
   //pthread_mutex_unlock(&mutex);
   }

//verifica daca buferul este gol
   int gol() {
   //pthread_mutex_lock(&mutex);
      if (recip[pozGet]==0)
         return 1;
      else
         return 0;
   //pthread_mutex_unlock(&mutex);
   }

//pune un articol in buffer
   put(int art,char sind[]) {
      int i=atoi(sind);
      pthread_mutex_lock(&mutex);
      while(plin()) {
         p[i]=-art;
         pthread_cond_wait(&condPut,&mutex);
      }
      recip[pozPut]=art;
      pozPut=(pozPut+1)%MAX;
      p[i]=art;
      scrie();
      p[i]=0;
      pthread_mutex_unlock(&mutex);
```

```
      pthread_cond_signal(&condGet);
   }

//extrage un articol din buffer
   get (char sind[]) {
      int i=atoi(sind);
      pthread_mutex_lock(&mutex);
      while(gol()) {
         c[i]=-1;
         pthread_cond_wait(&condGet,&mutex);
      }
      c[i]=recip[pozGet];
      recip[pozGet]=0;
      pozGet=(pozGet+1)%MAX;
      scrie();
      c[i]=0;
      pthread_mutex_unlock(&mutex);
      pthread_cond_signal(&condPut);
   }

//rutina thread-urilor producator
   void* produc(void* sind) {
      int sl;
      while (1) {
         art++;
         put(art,sind);
         sl=1+(int) (3.0*rand()/(RAND_MAX+1.0));
         sleep(sl);
      }
   }

//rutina thread-urilor consumator
   void* consum (void* sind) {
      int sl;
      while (1) {
         get(sind);
         sl=1+(int) (3.0*rand()/(RAND_MAX+1.0));
         sleep(sl);
      }
   }

//functia principala "main"
   main() {
      int i;
      char *sind;
      srand(0);
      pozPut=0;pozGet=0;
      for (i=0;i<MAX;i++) recip[i]=0;
      for (i=0;i<P;i++) {
         sprintf(sind,"%d",i);
         pthread_create(&tid[i],NULL,produc,sind);
      }
      for (i=0;i<C;i++) {
         sprintf(sind,"%d",i);
         pthread_create(&tid[i+P],NULL,consum,sind);
      }
      for (i=0;i<P+C;i++)
         pthread_join(tid[i],NULL);
   }
```

Programul 2.6 Sursa `ProducatorConsumator.c`

O porțiune din rezultatul execuției este:

```
- - - - - - - - - - - -
P0_0 P1_1 P2_0 P3_0 P4_0 C0_0 C1_0 C2_0 C3_0 C4_0 B: 1
P0_0 P1_0 P2_2 P3_0 P4_0 C0_0 C1_0 C2_0 C3_0 C4_0 B: 1 2
P0_0 P1_0 P2_0 P3_3 P4_0 C0_0 C1_0 C2_0 C3_0 C4_0 B: 1 2 3
P0_0 P1_0 P2_0 P3_0 P4_4 C0_0 C1_0 C2_0 C3_0 C4_0 B: 1 2 3 4
P0_5 P1_0 P2_0 P3_0 P4_0 C0_0 C1_0 C2_0 C3_0 C4_0 B: 1 2 3 4 5
P0_0 P1_0 P2_0 P3_0 P4_0 C0_0 C1_1 C2_0 C3_0 C4_0 B: 2 3 4 5
P0_0 P1_0 P2_0 P3_0 P4_0 C0_0 C1_0 C2_2 C3_0 C4_0 B: 3 4 5
P0_0 P1_0 P2_0 P3_0 P4_0 C0_0 C1_0 C2_0 C3_3 C4_0 B: 4 5
P0_0 P1_0 P2_0 P3_0 P4_0 C0_0 C1_0 C2_0 C3_0 C4_4 B: 5
P0_0 P1_0 P2_0 P3_0 P4_0 C0_0 C1_0 C2_0 C3_0 C4_5 B:
P0_0 P1_0 P2_0 P3_0 P4_6 C0_0 C1_0 C2_0 C3_0 C4_-1 B: 6
P0_0 P1_0 P2_0 P3_0 P4_0 C0_0 C1_0 C2_0 C3_0 C4_6 B:
P0_0 P1_0 P2_0 P3_0 P4_7 C0_0 C1_0 C2_0 C3_0 C4_-1 B: 7
P0_0 P1_0 P2_0 P3_0 P4_8 C0_0 C1_0 C2_0 C3_0 C4_-1 B: 7 8
P0_0 P1_0 P2_0 P3_0 P4_9 C0_0 C1_0 C2_0 C3_0 C4_-1 B: 7 8 9
P0_0 P1_0 P2_0 P3_0 P4_0 C0_0 C1_0 C2_0 C3_0 C4_7 B: 8 9
- - - - - - - - - - - -
```