Comunicarea interprocese (IPC)

Mecanismele de comunicare intre procese = **elementele fundamentale** ale tratarii proceselor intr-un sistem de operare.

Semnalele UNIX permit doar **anuntarea producerii unor evenimente** si de aceea trebuiesc suplimentate cu **faciliati care sa permita si transmiterea de informatii intre procese.**

Mecanismul de transmitere de date intre procese

- CANALUL (PIPE-ul)- **INITIAL**
- PIPE-urile cu nume (FIFO)
- Semafoarele
- Sirurile de mesaje
- Memoria partajata

PIPE-uri

Definitii. Apelul pipe

Pipe-ul este mecanismul de comunicare unidirectionala intre doua procese

- Analogie intre pipe-uri si fisiere ca purtatoare de date ceea ce explica utilizarea descriptorilor de fisier in legatura cu pipe-urile.

- Procesele care comunica printr-un pipe trebuie sa aibe un grad de rudenie (Un proces care creaza un pipe va apela dupa aceea **fork** iar pipe-ul se va folosi pentru comunicarea intre parinte si fiu).

Procesele care comunica prin PIPE nu pot fi create de utilizatori diferiti ai sistemului.

Apelul sistem pentru crearea unui PIPE:

#include <unistd.h>

int **pipe**(int *filedes [2]*);

Efect: - apelul returneaza: - 0 in caz de succes;

-1 in caz de eroare.

- prin argumentul *filedes* se returneaza 2 descriptori de fisier (e un tablou de intregi):

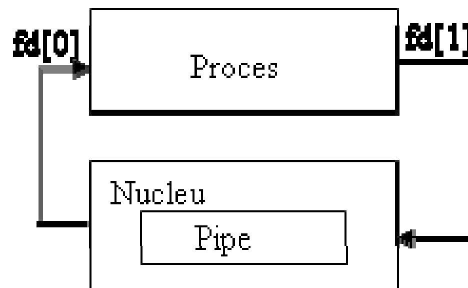
- ***filedes[0]*** **deschis pentru citire**
- ***filedes[1]*** **deschis pentru scriere**

Obs: iesirea lui *filedes[1]* este intrare pentru *filedes[0]* .

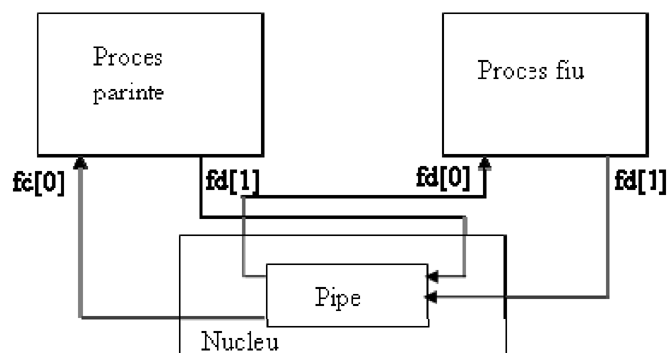
Pipe-ul apare ca o zona de memorie de o anumita dimensiune (de regula 10 kocteti) in spatiul nucleului.

In SVR4 PIPE-UL este biderectional desi pentru portabilitate se considera ca intotdeauna un PIPE este unidirectional.

Imaginea unui pipe in cadrul unui proces



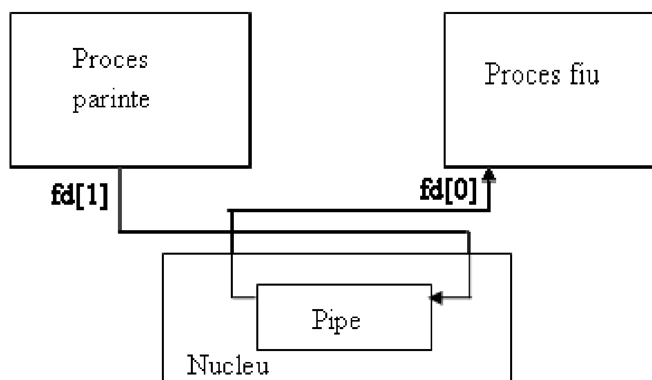
PIPE-ul instrument de comunicare intre doua procese



Majoritatea aplicatiilor care folosesc pipe-uri inchid, in fiecare dintre procese capatul de pipe neutilizat in comunicarea unidirectionala.

Exemplu: - daca pipe-ul e folosit pentru a comunica de la parinte la fiu, procesul parinte scrie in PIPE, iar procesul fiu citeste din PIPE.

- **procesul parinte a inchis fd[0];**
- **procesul fiu a inchis fd[1];**



Reguli care se aplica cind unul din capetele PIPE-ului e inchis intr-un proces

1. Daca se citeste dintr-un PIPE al carui capat de scriere a fost inchis, atunci dupa citirea datelor **read** va returna **0** pt a indica sfarsitul fisierului. In realitate pot exista mai multe procese care scriu intr-un PIPE. Sfirsitul de fisier se atinge cind nici un proces nu mai scrie in pipe.

2. Daca se scrie intr-un pipe al carui capat de citire nu mai este deschis de nici un proces, se genereaza semnalul **SIGPIPE**. Daca acest semnal este ignorat sau daca este interceptat, dar se revine din functia de tratare, revine si apelul write care returneaza **-1**, iar **errno** va avea valoarea **EPIPE**.

OBS

Frecvent descriptorii pentru un PIPE se duplica in descriptorii pentru intrarea standard sau iesirea standard. In felul acesta un proces fiu va putea lansa in executie (apelul **exec**) un alt program, care va avea intrarea standard sau iesirea standard in PIPE.

Transmitrerea unui text prin PIPE de la procesul parinte la procesul fiu, unde textul e afisat

```
#include <unistd.h>
 #include <sys/types.h>
int main (void) {
        int fd[2];  - declaram tabloul de intregi
        pid_t pid;
        int p;
        int n;
        char mesaj[50];
        p=pipe(fd);             - s-a creat un PIPE
        if(p<0){
        printf("eroare pipe\n");
        exit(1);                - fortare iesire din program
        }
        pid=fork()              - s-a creat un proces fiu
        if(pid<0){
        printf("eroare fork\n");
        exit(2);                - fortare iesire din program
        }
```

```
    else if (pid>0){            /* parinte */
    close (fd[0]);              parintele inchide capatul nefolosit !!!!!
    write(fd[1],"testare PIPE\n", 13);
    } else {                    /* suntem in procesul fiu*/
    close (fd[1])
    n = read(fd[0], mesaj, 50);          /* fiul citeste mesajul din PIPE */
    printf("mesajul receptionat: %s", mesaj);   /* se afiseaza mesajul citit*/
    }
}
```

Functiile **popen** si **pclose**

Secventa tipica de operatii in aplicatiile care folosesc PIPE-uri

- **Crearea unui PIPE**
- **Apelul fork**
- **Inchiderea capetelor nefolosite ale PIPE-ului**
- **Apelul exec**

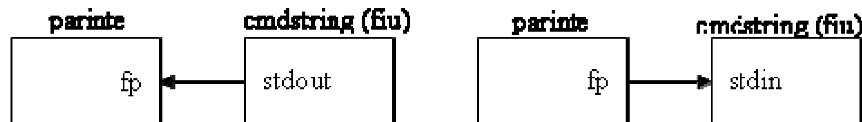
Functiile popen si pclose (biblioteca) standard elibereaza programatorul de o serie de detalii.

```
#incude <stdio.h>
FILE *popen(constchar *cmdstring, const char  *type);
int pclose(FILE *fp);
```

Functia popen apeleaza fork, dupa care apeleaza exec pentru a lansa *cmdstring* si returneaza un pointer de fisier.

Daca argumentul *type* este "r" pointerul de fisier este conectat la iesitrea standard a lui *cmdstring.*

Daca *type* este "w" pointerul de fisier este conectat la intrarea standard a lui *cmdstring.*



Functia **pclose** inchide fluxul standard de introducere/extragere a datelor, asteapta terminarea comenzii lansate prin popen si in final returneaza shell-ului starea de terminare.

Utilizarea apelului select la lucrul cu PIPE-uri multiple

Operatiile cu PIPE-uri trebuie sa se execute fara blocare;

Solutii:- **apelul fcnt1** pentru a modifica proprietatile operatiilor cu PIPE;

- **apelul sistem select – o solutie mai generala**

```
#include <sys/time.h>
#include <sys/types.h>
#include <sys/unistd.h>
int select (int nfds, fd_set  *readfds, fd_set  *writefds, fd_set
*exceptfds, struct timeval  *timepout);
```

Apare in procese server care controleaza mai multe *pipe*-uri pt comunicarea cu un numar oarecare de procese fii(*clienti*).

OBS

Daca la nici unul dintre pipe-uri nu exista info de transferat procesul server se va bloca(intra in asteptare). Daca exista simultan info la mai multe *pipe*-uri, **serverul trebuie sa selecteze** unul din ele.

Argumente

- ***nfds*** indica nr de descriptori de interes pt server
- ***readfds*, *writefds* si *exceptfds*** reprezinta seturi de descriptori corespunzatori operatiilor de interes in program: citire din *pipe*, scriere, sau situatii de exceptie. **Tipul fd_set** este definit in <sys/types.h> ca o masca de biti cite unul pentru fiecare descriptor posibil. Singurele operatii posibile asupra tipului **fd_set** sunt:

a) Alocarea unei variabile de acest tip;

b) Asignarea unei variabile de acest tip cu valoarea altei variabile de acelasi tip;

c) Uitlizarea uneia din cele 4 macroinstructiuni posibile

```
FD_ZERO(fd_set *fdset); /*pune bitii pe  zero*/
FD_SET(int fd, fd_set *fdset); /*pune pe  unu bitul fd */
FD_CLR(int fd,fd_set *fdset); /*pune pe zero bitul fd*/
FD_ISSET(int fd, fd_set *fdset); /*true daca fd pe unu*/
```

Oricare din cele 3 argumente poate fi pointerul nul daca operatia respectiva nu prezinta interes in program.

- ***timeout*** este un pointer la o structura data in <sys/time.h>. Prin acest argument se exprima timpul cit un proces asteapta la **select**.

Apelul select poate returna -1 sau 0 cu semnificatiile precizate.

Daca insa apelul **returneaza o valoare strict pozitiva**, aceasta reprezinta numarul descriptorilor pregatiti pentru operatie iar in seturile de descriptori bitii corespunzatori acestor descriptori vor fi singurii cu valoarea 1.

FIFO (Pipe-uri cu nume)

Limitarea pipe-urilor: procesele care comunica trebuie sa fie inrudite se elimina prin utilizarea pipe-urilor cu nume (FIFO).

FIFO este un **tip de fisier**, crearea lui facind sa apara un fisier cu numele de cale corespunzator in sistemul de fisiere UNIX.

Apelul sistem pentru crearea unui FIFO:

#include <sys/types.h>

#include <sys/stat.h>

int mkfifo(const char *_pathname_, mode_t _mode_);

Argumentul ***pathname*** este numele FIFO-ului. Al doilea argument, ***mode*** este construit la fel ca si in cazul apelului **open**.

Regulile pentru drepturile de proprietate sunt aceleasi ca si la fisiere.

Dupa ce FIFO a fost creat el poate fi deschis folosind apelul open, apoi toate operatiile cu fisiere (close, read, write, unlink etc) i se pot aplica la fel ca altor fisiere.

Programul care scrie in FIFO

```
#include <stdio.h>
 #include <fcntl.h>
int main (void) {
        char numefifo[20]="canal"                nume de cale pt fifo
        int m, fd, w
        m = mkfifo(numefifo, 0666);
        if(m==-1)
        printf("eroare mkfifo\n");
        else{
        fd=open(numefifo, O_WRONLY | O_NONBLOCK);
        if(fd<0)
        printf("eroare open fifo\n");
        else {
         w=write (fd, "mesaj test\n", 11)        scriem unde am deschis
        if (w==-1)
        printf("eroare write\n");
        }   }
}
```

Programul care citeste din FIFO

```
#include <stdio.h>
 #include <fcntl.h>
int main (void) {
        char numefifo[20]="canal"                nume de cale pt fifo
        int m, fd, r
        char text[30]
        m = mkfifo(numefifo, 0666);
        if(m==-1)
        printf("eroare mkfifo\n");
        else{
        fd=open(numefifo, O_RDONLY );
        if(fd<0)
        printf("eroare open fifo\n");
        else {
         r=read (fd, text, 30)           citim din FIFO-ul pe care l-am deschis
        if (r==-1)
        printf("eroare read\n");
        }
        printf("mesajul receptionat: %s", text)
        }
}
```

Modul de comportare al unui FIFO dupa deschidere este afectat de fanionul **O_NONBLOCK**:

1. **O_NONBLOCK nu este specificat** (**cazul normal**) - un open pentru citire se va bloca pina cind un alt proces deschide acelasi FIFO pentru scriere. Analog pt scriere…
2. Daca se specifica **O_NONBLOCK**, deschiderea pentru citire revine imediat dar o deschidere pentru scriere poate returna eroare cu **errno** avind valoarea **ENXIO** daca nu exista un alt proces care a deschis acelasi FIFO pentru citire.

OBS

- Ca si in cazul unui pipe o operatie write la un FIFO pe care nici un proces nu il mai are deschis pentru citire va genera un **SIGPIPE.** Cind ultimul proces care scrie in FIFO il inchide, se va genera un "sfirsit de fisier" pentru procesul care citeste din FIFO.

- In multe aplicatii este normal sa existe un singur proces care citeste dintr-un FIFO dar pot exista mai multe procese care scriu in acel FIFO. Se pune problema ***scrierilor atomice*** daca este important ca ceea ce scriu procesele sa nu se amestece.

Comunicare interprocese se realizeaza prin trei mecanisme de baza (alcatuiesc **structura IPC**):

- transferul de mesaje

- semafoarele

- memoria partajata

Implementarile lor in nucleu ca si interfata de programare au fost gindite cit mai asemanatoare.

Fiecare structura IPC este cunoscuta in nucleu printr-un identificator intreg ne-negativ. Orice operatie asupra structurilor, dupa crearea lor, se face specificind acest identificator obtinut prin procedeul de creare a structurii.

Crearea unei structuri IPC:

- operatii msgget, semget, shmget;

- chei de tipul key_t definit in <sys/types.h> cu reprezentare dependenta de implementare(frecvent un intreg lung). **Aceasta cheie e convertita de nucleu in identificatorul structurii, valorile identificatorilor pt fiecare dintre cele trei mecanisme atribuindu-se crescator pina la o limita superioara definita de sistem.**

Fiecarei structuri IPC ii este asociata o structura de date de tipul ipc_perm:

```
struct ipc_perm{
        uid_t uid;          /* owner effective user id */
        gid_t gid;          /* owner effective group id */
        uid_t cuid;         /* creator effective user id */
        gid_t cgid;         /* creator effective group id */
        mode_t mode;        /* access modes*/
        ulong seq;          /*slot usage sequence number */
        key_t key;          /* key */
};
```

OBS

Cimpul **seq** reprezinta identificatorul structurii IPC si nu poate fi modificat pe parcursul existentei acesteia

Apeluri specifice pentru modificarea celorlalte cimpuri(de catre proprietarul structurii sau de superutilizator):

msgct1, semct1, shmct1

Valorile din cimpul **mode** sunt similare celor de la fisiere dar au si **denumiri simbolice specifice**:

- pentru sirurile de mesaje: **MSG_R, MSG_W**
- pentru semafoare: **SEM_R, SEM_A**
- pentru memoria partajata: **SHM_R, SHM_W**

!!!! Nu exista drepturi de executie iar la semafoare in loc de scriere se utilizeaza denumirea de alterare (modificare).

Modalitati pentru ca un client si un server sa se la intilneasca la aceeasi structura IPC:

- Serverul creaza o noua structura IPC pt care specifica drept valoare a cheii **IPC_PRIVATE** si memoreaza identificatorul returnat intr-un loc(un fisier) unde clientul il poate gasi.

OBS Cheia **IPC_PRIVATE** garanteaza ca nucleul creaza o noua structura IPC, dar dezavantajul acestui procedeu este acela ca se foloseste un fisier ca intermediar intre server si client.

- Clientul si serverul stabilesc o cheie comuna, definita, de exemplu, intr-un fisier antet, serverul urmind sa creeze o structura IPC cu aceasta cheie.

Problema!!! Poate sa existe o structura cu acea cheie si in acest caz serverul trebiuie sa trateze eroarea, stergand, de exemplu, structura existenta si incercind din nou eroarea.

- Clientul si serverul stabilesc de comun acord un nume de cale si un identificator de proiect (o valoare intre 0 si 255) si apeleaza functia ftok pt a converti aceste doua valori intr-o cheie care se foloseste ca in cazul precedent.

Toate cele trei functii **get** pt structurile IPC au 2 argumente:

- ***cheia***
- un ***fanion***

Se creaza o structura IPC noua daca:

- ***cheie*** are valoarea **IPC_PRIVATE**
- ***cheie*** nu este momentan asociata cu o structura IPC de tipul corespunzator, iar in *fanion* este specificat bitul IPC_CREAT.

Accesul la o structura existenta:

- valoarea folosita pentru ***cheie*** este egala cu cea **specificata la crearea structurii**. **Nu este posibil sa se foloseasca IPC_PRIVATE.**

Probleme generale asociate structurilor IPC

- structurile IPC sunt vizibile la nivelul intregului sistem si nu au numarator de referinte analog numaratorului de legaturi din i-nodurile de fisiere. O structura IPC ramine in sistem pina cind este explicit stearsa.

-IPC- urile nu folosesc descriptori de fisier. De aceea pt modificarea proprietatilor lor e nevoie sa se **introduca un numar destul de mare de apeluri sistem.**

- nu pot fi folosite functiile pt multiplexarea operatiilor de introducere/extragere, **select** si **poll** ceea ce face dificila utilizarea simultana a mai multor asemenea structuri. De expl: nu se poate scrie un server care sa astepte mesaje in 2 siruri de mesaje fara a apela la o forma de asteptare activa ("busy waiting").

Avantajele structurilor IPC

- sunt **fiabile**;
- sunt **orientate pe inregistrari**, spre deosebire de *pipe*-uri, de exemplu care sunt orintate pe **flux de octeti**, raminind ca aplicatiile sa convina asupra unei divizari logice a fluxului.
- pot fi prelucrate si in alt aordine decit strict "primul venit, primul servit"

Siruri de mesaje

Un sir de mesaje este o lista inlantuita de mesaje memorata in nucleu si prevazuta cu **un identificator de sir**.

Crearea sau deschiderea unui sir se realizeaza cu apelul sistem **msgget**.

Mesajele se adauga la sfirsitul unui sir cu **msgsnd** si contin:

- un cimp care specifica tipul
- un cimp care specifica lungimea
- un cimp ce reprezinta datele pr-zise ale mesajului.

Extragerea mesajelor din sir se face cu **msgrcv** si nu trebuie efectuata in ordinea introducerii. Se poate folosi pt a stabili ordinea extragerii valoarea din campul care indica tipul mesajului.

Fiecare sir are asociata o structura msquid_ds care defineste starea curenta a sirului.

```
struct msqid_ds {
    struct ipc_perm msg_perm;
    struct msg *msg_first;   /*ptr. la primul mesaj din sir*/
    struct msg *msg_last;    /*ptr. la ultimul mesaj din sir*/
    ulong msg_cbytes;        /*nr. octeti in sir*/
    ulong msg_qnum;          /*nr. mesaje in sir*/
    ulong msg_qbytes;        /*nr. maxim, admis de octeti*/
    pid_t msg_lspid;         /*pid-ul ultimului msgsnd() */
    pid_t msg_lrpid;         /*pid-ul ultimului msgrcv() */
    time_t msg_stime;        /* timpul ult. msgsnd() */
    time_t msg_rtime;        /* timpul ult. msgrcv() */
    time_t msg_ctime;        /* timpul ultimei modoficari */
};
```

Apelul msgget

Sintaxa:

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

Campuri care se initializeaza in msgqid_ds la crearea unui nou sir:

- structura ipc_perm: cimpul mode al structurii primeste valorile din cimpul *flag* al apelului;

-msg_qnum, msg_qlspid, msg_lrpid, msg_stime, msg_rtime primesc valoarea 0;

- msg_ctime primeste ca valoare timpul curent al sistemului;

- msg_qbytes primeste ca valoare o limita specificata de sistem.

Operatiile uzuale asupra sirurilor se fac cu apelurile msgsnd si msgrcv cu prototipurile:

```
struct mymsg {
        long mtype;             /* tipul mesajului */
        char mtext [512];       /* datele, de lungime nbytes */
}
```

Pentru msgsnd argumentul *flag* poate avea valoarea **IPC_NOWAIT**, cu efect similar fanionului de ne-blocare de la operatiile de introducere /extragere: daca sirul e plin msgsnd revine imediat cu eroare, iar errno primeste valoarea EAGAIN.

In absenta **IPC_NOWAIT , apelul** msgsnd se blocheaza pina:

- se elibereaza spatiu pentru mesaj;
- sirul este sters din sistem;
- se intercepteaza un semnal si se revine din functia de tratare a semnalului.

Apelul msgsnd mai poate esua si daca drepturile de acces ale apelantului la sirul identificat prin mqid nu permit scrierea. Valoarea lui errno este in acest caz EACCES.

Pentru un apel **msgrv,**

- daca lungimea mesajului primit este mai mare decit *nbyte*s si in *flag* figureaza **MSG_NOERROR**, mesajul va fi trunchiat fara a se anunta utilizatorul.

- daca in flag nu apare acasta eroare si lungimea mesajului este prea mare, mesajul ramine in sir, iar errno primeste valoarea **E2BIG**.

Argumentele apelului msgrcv

- ***type****:* permite specificarea tipului de mesaj dorit dupa urmatoarea conventie:

type == 0	Se returneaza primul mesaj din sir.
type > 0	Se returneaza primul mesaj din sir care are tipul egal cu *type*
type < 0	Se returneaza primul mesaj din sir al carui tip este cea mai mica valoare situata sub sau egala cu valoarea absoluta a lui *type*. **Se poate construi astfel un sir cu prioritati, valoarea cea mai mica pentru *type* insemnind prioritate maxima.**

- *flag*: daca se specifica **IPC_NOWAIT** se returneaza eroare cu errno egal cu ENOMSG atunci cind sirul nu contine un mesaj de tipul specificat. In absenta IPC_NOWAIT, apelul se blocheaza pina cind:

a) Este disponibil un measj de tipul dorit;

b) Sirul este sters din sistem;

c) Se intercepteaza un semnal si se revine din functia de tratare a semnalului.

OBS Si apelul msgrcv poate esua daca apelantul nu are drepturi de acces corespunzatoare la sirul identificat prin *msqid*.

Apelul msgct1

Permite obtinerea unor informatii de stare despre un sir, modificarea unor limite asociate sirului sau stergerea unui sir din sistem.

Sintaxa apelului:

```
#include <sys/types.h>
 #include <sys/ipc.h>
 #include <sysmsg.h>
int msgct1(int msqid, int cmd, struct msqid_ds *buf);
```

Argumente:

- *cmd* specifica ce comanda se va executa asupra sirului identificat de *msqid*:

IPC_STAT

Obtine structura msqid_ds pentru un sir si o memoreza in structura in care indica *buf*.

IPC_SET

Specifica prin structura spre care indica *buf* valori pentru urmatoarele cimpuri din structura msquid_ds asociata sirului msqid: msg_perm.uid, msg_perm.gid, msg_perm.mode, si msg_bytes. Aceasta c-da poate fi executata numai de catre un proces pentru care UID-ul efectiv este egal cu msg_perm.cuid sau cu msg_perm.uid, respectiv de catre un proces cu privilegii de superutilizator.

IPC_RMID

Srerge sirul de mesaje din sistem pierzindu-se astfel si eventualele date care se gasesc in sir la momentul respectiv. Orice alt proces care inca utilizeaza sirul va primi eroarea EIDRM la urmatoarea incercare de acces la sir Pt efectuarea c-zii se cer drepturi de acces corespunzatoare.

Semafoare

In principiu un semafor este un numarator folosit pentru a controla accesul la o resursa partajata intre mai multe procese.

Pentru ca un proces sa poata accesa acea resursa trebuie sa efectueze urmatoarele operatii:

1. **Testeaza** valoarea semaforului corespunzator

2. Daca **valoarea semaforului e pozitiva**, **procesul va utiliza resursa**. Valoarea semaforului e **decrementata** (cu 1) pentru a indica utilizarea unei unitati din acea resursa.

3. Daca **valoarea semaforului e zero**, procesul intra in **asteptare** pina cind semaforul redevine strict pozitiv. In acest moment, procesul se reintoarce la pasul 1.

Cind un proces termina operatia care utilizeaza resursa comuna controlata de un anumit semafor, procesul **incrementeaza** valoarea acelui semafor . Daca exista procese in asteptare la semafor acestea vor fi **reactivate**.

OBS Pentru o implementare corecta a semafoarelor este necesar ca testarea valorii semaforului impreuna cu decrementarea acestuia sa se efectueze ca operatie atomica, motiv pentru care semafoarele sunt in mod normal implementate in nucleu.

Caracteristici privind implementarea semafoarelor in SVR4:

1. Semafoarele nu exista individual ci numai ca seturi numarul de semafoare dintr-un set fiind definit la crearea setului.
2. Crearea unui semafor (prin apelul semget) este distincta de initializarea semaforului (prin semctl) ceea ce face ca initializarea si crearea sa nu poata fi implementate impreuna ca operatie atomica.
3. Ca si celelalte structuri IPC semafoarele ramin in sistem si dupa terminarea proceselor care le utilizeaza deci este necesara o modalitate de tratare a situatiilor in care un program se termina fara a elibera semafoarele alocate.

Structura pastrata in nucleu pt fiecare set de semafoare este:

```
struct sem_id_ds{
    struct ipc_perm sem_perm;
    struct sem      *sem_base;   /* ptr la primul sem. din set */
    ushort          sem_nsems;   /* nr. de sem. din set */
    time_t          sem_otime;   /* timpul ultimei oper. */
    time_t          sem_ctime;   /* timpul ultimei modificari*/
```

Cimpul **sem_base** nu este direct accesibil programelor, el indicind spre o zona de memorie din nucleu care contine **sem_nsem** elemente, fiecare element declarat astfel:

```
struct sem{
        struct ipc_perm sem_perm;
        ushort    semval;        /* val. semafor, mereu >=0  */
        pid_t     sempid;        /* pid pentru ultima operatie */
        ushort    semncnt;       /*nr. procese ast. semval > crtval */
        ushort    semzcnt;       /* nr. procese ast. semval =0 */
}
```

Apelul sistem **semget**

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

Parametrul *nsems* reprezinta nr de semafoare din set valoarea sa fiind necesara la crearea setului de semafoare. Daca semget se refera la un set de semafoare existent se poate utiliza pt nsems valoarea 0.

Apelul sistem semct1

```
 int semct1(int  semid, int semnum, int  cmd, union semun arg);
```

Paramentrul *semnum* este folosit pt a indica un anumit semafor din set, la unele dintre functiile redate prin *cmd.*

Parametrul *arg* este o uniune care are urmatoarea definitie:

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

Prin argumentul *cmd* pot fi specificate 10 functii de executat asupra setului identificat de *semid*:

IPC_STAT Citeste structura semid_ds a setului si o memoreaza in structura spre care indica *arg.buf*.

IPC_SET Atribuie valori gasite in arg.buf urmatoarelor cimpuri din structura asociata setului: se_perm.uid, sem_perm.gid si sem_perm.mode. Procesul care executa apelul trebuie sa aibe drepturi de acces corespunzatoare.

IPC_RMID Srege setul de semafoare di sistem. Stergerea este imediata ca si la siruri, orice proces care incearca sa foloseasca un semafor din set primind eroarea EIDRM. Se cer drepturi coresp…

GETVAL Returneaza valoarea lui semval pentru membrul semnum.

SETVAL Atribuie valoarea lui semval pentru membrul semnum. Valoarea care se atribuie e preluata din arg.val.

GETPID Returneaza valoarea lui sempid pentru membrul semnum.

GETNCNT Returneaza valoarea lui semncnt pentru membrul semnum.

GETZCNT Returneaza valoarea lui semzcnt pentru membrul semnum.

GETALL Citeste valorile tuturor semafoarelor din set. Valorile sunt memorate in tabloul spre care indica *arg.array*.

SETALL Atribuie valori tuturor semafoarelor din set, preluind valorile din tabloul spre care indica *arg.array*.

Pt efectuarea unor operatii atomice asupra unui set de semafoare se foloseste **apelul** semop, cu prototipul:

int semop(int *semid,* struct sembuf *semoparray[ ]*, sixe_t *nop*s);

Argumentul *semoparray* este un pointer la un tablou de operatii asupra semafoarelor, sembuf fiind definit astfel:

```
struct sembuf{
        ushort    sem_num;                /* nr semafor din set */
        short     sem_op;                 /*operatia (<0,0, sau >0) */
        short     sem_flg;                /* IPC_NOWAIT, SEM_UNDO  */
};
```

Argumentul nops arata nr de operatii (elemente) din tabloul *semoparray*.

Operatia specificata prin sem_op se interpreteaza in felul urmator:

1. Daca sem_op este pozitiv inseamna c aprocesul rteurneaza resurse. Valoarea lui sem_op se adauga la valoarea semaforului.
2. Daca sem_op este negativ inseamna ca procesul solicita resursele controlate de semafor:

   - daca valoarea semaforului este mai mare sau cel putin egala cu valoarea absoluta a lui sem_op, atunci cantitatea de resurse solicitate se scade din valoarea semaforului (rezultatul va fi mai mare decit zero)

   - daca valoarea semaforului nu satisface conditia de dinainte se pot petrece urmatoarele:

- Daca este specificat IPC_NOWAIT, atunci apelul revine cu eroarea EAGAIN.
- Daca nu e specificat IPC_NOWAIT, atunci valoarea lui semnsnt pentru acest semafor este incrementata (pt ca procesul urmeaza sa intre in asteptare), iar procesul e suspendat pina cind va fi satisfacuta una din urmatoarele conditii:

  - valoarea semaforului devine mai mare sau egala cu valoarea absoluta a lui **sem_op**. Valoarea lui semncnt este decrementata iar valoarea absoluta a lui **sem_op** se scade din valoarea semaforului;

  -semaforul este sters din sistem: in acest caz functia va returna eroare cu **errno** egal cu ERMID;

  - Procesul intercepteaza un semnal si se revine din functia de tratare a semnalului. In acest caz **semncnt** se decrementeaza(procesul iese din asteptare), iar functia returneaza eroare, cu errno egala cu EINTR.

3. Daca sem_op este zero se asteapta ca valoarea semaforului sa devina zero. Daca semaforul este zero in momentul apelului se revine imediat. Daca insa valoarea este diferita de zero, atunci:

- Daca e specificat IPC_NOWAIT, se revine cu EAGAIN.
- Daca nu e specificat IPC_NOWAIT, valoarea semzcnteste incrementata, iar procesul apelant este suspendat pana cand apare una din urmatoarele conditii:

  - valoarea semaforului devine zero; se decrementeaza atunci **semzcnt**, pentru ca procesul iese din asteptare;

  - semaforul este sters din sistem: functia returneaza eroarea ERMID;

  - procesul intercepteaza un semnal si revine din functia de tratare a semnalului. In acest caz **semzcnt** se decrementeaza (procesul iese din asteptare), iar functia returneaza eroare cu **errno** egala cu **EINTR**.

Daca se specifica fanionul **SEM_UNDO** pt o operatie asupra unui semafor si se aloca resurse (adica **sem_op** este negativ) atunci nucleul inregistreaza ce resurse se aloca, iar la terminarea normala sau fortata a procesului , nucleul ***ajusteaza*** corespunzator valoarea resurselor.