Laborator 7 – IPC SYSTEM V

Ce reprezinta SYSTEM V ?

System V reprezinta prima versiune comerciala a sistemului de operare UNIX, aparuta in anul 1983 si dezvoltata la laboratoarele AT&T.

Dupa cum am vazut in laboratorul anterior, prin **IPC (Inter Process Comunication)** intelegem obiecte prin care doua sau mai multe procese pot sa comunice intre ele. Odata cu **System V** au aparut trei mecanisme noi de comunicare, si anume :

- **Cozi de mesaje**
- **Vectori de semafoare**
- **Segmente de memorie partajata**

Deoarece aparitia acestor mecanisme a avut loc odata cu aparitia **System V** acestea poarta numele de **IPC SYSTEM V.**

Generalitati despre IPC SYSTEM V

Desi reprezinta trei mecanisme diferite de comunicare intre procese, acestea au cateva caracteristici comune pe care le vom enunta mai jos.

Fiecare obiect de tip **IPC SYSTEM V** este mentinut de catre kernel pe toata durata instantei sistemului (adica pana inchizi calculatorul) sau pana cand un proces cere explicit inlaturarea acestuia.

Acest lucru inseamna ca obiectele pot sa ramana in sistem chiar daca nu mai exista nici un proces ce le utilizeaza pentru a comunica. Obiectele raman in aceeasi stare in care au fost lasate de ultimul proces ce le-a utilizat.

Fiecare obiect este reprezentat la nivelul sistemului de o cheie **unica.** Un identificator prin care un proces poate sa refere un anumit obiect. Aceasta cheie este o valoare numerica naturala.

In interiorul procesului, manipularea unui obiect se face prin intermediul unui identificator intern, ce prezinta tot o valoare numerica naturala. Daca ar fi sa facem o corespondenta cu ce am invatat pana acum la lucrul cu fisiere, **cheia unica** din sistem poate sa fie vazuta ca numele unui fisier, iar **identificatorul unic** poate sa fie vazut ca descriptorul de fisier prin care manipulam fisierul.

O alta asemanare cu fisierul, o reprezinta faptul ca fiecare obiect are o masca de drepturi asociata acestuia. Aceasta are acelasi format ca cel invatat la fisere (citire, scriere, executie pentru propietar ,grup proprietar sau ceilalti). Nu exista nici un motiv pentru care am avea nevoie de drepturi de executie asupra unui obiect de tip **IPC SYSTEM V** dar pentru a mentine o interfata comuna s-a decis pastrarea acestui tip de drept.

Pentru fiecare instanta a unui asftel de obiect, **kernel-ul** mentine o structura de date asociata acestuia. Aceasta structura difera pentru fiecare tip (dintre cele trei) de **IPC,** dar un camp este mereu prezent in aceasta structura, si anume informatile despre permisiunile si propietarul unui astfel de obiect.

Intr-un program C, astfel de permisiuni sunt reprezentate prin structura :

```
struct ipc_perm {
        key_t        __key;      /* Cheia externa a obiectului */
        uid_t        uid;        /*  UID-ul propietarului */
        gid_t        gid;        /* GID -ul propietarului */
        uid_t        cuid;       /* UID - ul creatorului */
        gid_t        cgid;       /* GID -ul  creatorului */
        unsigned short mode;       /* Permisiunile */
        unsigned short __seq;      /* Numarul de utilizatori ai acestui obiect */
      };
```

O sa vedem mai jos cum arata structura de date pentru fiecare tip de obiect si cum o putem manipula.

Utilitare in linia de comanda

Sistemele de operare din familia UNIX ce au implementat mecanismele de comunicare intre procese de tip **System V** vin cu cateva utilitare in linia de comanda pentru managerierea acestora.

Pentru a avea o lista a tuturor obiectelor existente la un moment dat in sistem putem folosi comanda:

$ ipcs

Aceasta va afisa obiectele pe cele trei categorii (cozi de mesaje, vectori de semafoare, memorie partajata) impreuna cu alte informatii folositoare. La o rulare a acesti comenzi, pe calculator meu, am obtinut urmatorul rezultat:

```
------ Shared Memory Segments --------
key            shmid      owner      perms      bytes      nattch     status
0x00000000 294912     andrei     600        524288     2          dest
0x00000000 622593     andrei     600        524288     2          dest
0x00000000 425986     andrei     600        524288     2          dest
0x00000000 1736707    andrei     600        393216     2          dest
0x00000000 655364     andrei     600        16777216   2
0x00000000 950277     andrei     600        524288     2          dest
0x00000000 1048582    andrei     600        524288     2          dest
0x00000000 1081351    andrei     600        33554432   2          dest
0x00000000 1835016    andrei     600        524288     2          dest
0x00000000 1376265    andrei     600        524288     2          dest
0x00000000 1572874    andrei     600        524288     2          dest
0x00000000 2686987    andrei     600        33554432   2          dest
0x00000000 1933324    andrei     600        524288     2          dest
0x00000000 4325389    andrei     600        2097152    2          dest
0x00000000 2261006    andrei     600        524288     2          dest
0x00000000 2293775    andrei     600        4194304    2          dest
0x00000000 2326544    andrei     600        33554432   2          dest
0x00000000 2588689    andrei     600        524288     2          dest
0x00000000 2621458    andrei     600        4194304    2          dest
0x00000000 2654227    andrei     600        2097152    2          dest
0x00000000 2719764    andrei     600        393216     2          dest
```

0x00000000 4096021 andrei 600 1639680 2 dest
0x00000000 2850838 andrei 600 393216 2 dest
0x00000000 2883607 andrei 600 1048576 2 dest
0x00000000 3047448 andrei 600 1639680 2 dest
0x00000000 4423705 andrei 600 1639680 2 dest
0x00000000 4554778 andrei 600 524288 2 dest
0x00000000 4587547 andrei 600 4194304 2 dest
0x00000000 4620316 andrei 600 16777216 2 dest

------ Semaphore Arrays --------
key semid owner perms nsems
0x26634b3d 98304 andrei 444 1

------ Message Queues --------
key msqid owner perms used-bytes messages
0x7c987394 0 andrei 555 0 0

Dupa cum se poate observa, exista foarte multe obiecte de tip **Shared Memory Segments (segmente de memorie partajata)** deoarece acestea sunt mai lesne de utilizat si mai rapide decat celelalte doua tipuri de obiecte. Avem un singur obiect de tip vector de semafoare si un singur obiect de tip coada de mesaje. Comanda **ipcs** poate sa fie folosita impreuna cu diverse argumente pentru a obtine o filtrare a informatilor sau pentru a obtine informatii suplimentare.

De exemplu :

$ ipcs -m

va afisa doar segmentele de memorie partajata. Pentru o descriere completa a comenzii puteti consulta pagina de manual.

Pentru a creea un nou obiect se utilizeaza comanda **ipcmk** ce trebuie lansata cu argumentele specifice fiecarui tip de obiect in parte. Astfel, daca dorim un nou obiect de tip coada de mesaje se poate da comanda:

$ ipcmk -Q

Dar daca dorim un obiect de tip vector de semafoare trebuie sa dam comanda:

$ ipcmk -S nsems

Unde **nsems** reprezinta numarul de semafoare din vectorul nostru. Cu optiunea **-p** se pot atribuii alte drepturi decat cele standard obiectului nou creeat.

$ ipcmk -S nsems -p mask

Unde **mask** are aceeasi semnificatie ca la comanda **chmod.** De exemplu, comanda:

$ ipcmk -S 2 -p 666

va creea un vector de semafoare cu doua semafoare asupra caruia vor avea drepturi de scriere si de citire propietarul, grupul propietar si ceilalti.

Pentru a sterge un obiect se foloseste comanda:

$ ipcrm -x identificator_intern

sau

$ ipcrm -X cheie

Unde '**x**' apartine multimii **{ m, q, s}** si '**X**' apartine multimii **{ M, Q, S }.** Pentru fiecare dintre cele trei tipuri. Diferenta dintre cele doua comenzi este ca, in primul caz, se da identificatorul intern al **IPC-ului** iar in al doilea caz se da cheia sa.

Adica, presupunand ca avem un vector de semafoare astfel:

------ Semaphore Arrays --------
key semid owner perms nsems
0x3d5d9b85 131075 andrei 644 1

Cele doua moduri in care putem sa il stergem folosind comanda **ipcrm** sunt:

$ ipcrm -s 131075

sau

$ ipcrm -S 0x3d5d9b85

Putem sa stergem un anumit obiect numai daca suntem propietarii acelui obiect sau **root.**

IPC-urile Private

Dupa cum am mai spus, fiecare obiect din sistem este identificat unic printr-o cheie ce reprezinta o valoarea numerica naturala. Ca rezultat al comenzii **ipcs** aceasta cheie se afla pe prima coloana din tabelul afisat (denumita "**key"**) in baza 16. Totusi, observam ca exista obiecte cu aceeasi cheie:

------ Shared Memory Segments --------
key shmid owner perms bytes nattch status
0x00000000 294912 andrei 600 524288 2 dest
0x00000000 622593 andrei 600 524288 2 dest
0x00000000 425986 andrei 600 524288 2 dest

Valoarea 0 pentru o cheie denota faptul ca acest obiect este un **IPC privat** adica el este disponibil numai procesului creator si al descendentilor acestuia. El este vizibil in lumea exterioara dar nici un alt proces nu poate sa il foloseasca. Mai mult, daca procesele ce l-au folosit nu il sterg explicit, el va ramane in sistem pana cand va fi sters prin comanda **ipcrm** fara sa mai poata fi reutilizat. Deci, daca printr-o greseala de programare se creeaza foarte multe obiecte private fara a fi sterse, acest fapt va conduce la umplerea tabelului de **IPC-uri** ale **kernel-ului** si nu se vor mai putea creea altele noi.

Daca dintr-un program C dorim sa cream un **IPC privat**, avem la dispozitie constanta **IPC_PRIVATE.** Totusi, apare o problema si atunci cand dorim sa cream un **IPC public** pentru ca daca cheia aleasa de noi identifica deja un alt obiect din sistem, atunci s-ar putea sa avem o

problema pe care o vom studia mai jos. Pentru a genera o cheie care are mari sanse sa fie unica, folosim functia :

#include <sys/ipc.h>

key_t ftok(char * pathname , int proj);

ftok (file to key) va intoarce o cheie noua calculata dupa un algoritm intern ce se foloseste de numele si numarul de i-nod al fisierului transmis ca parametru si de intregul **proj.** Tipul de date cheie se reprezinta in C prin **key_t.**

Cozile de mesaje

Primul obiect de tip **IPC SYSTEM V** pe care il vom trata sunt **cozile de mesaje.** Cozile de mesaje reprezinta mecanisme prin care doua sau mai multe procese isi pot transmite informatii intre ele. Prin **mesaj** intelegem un bloc de octeti pe care un proces il plaseaza intr-o structura mentinuta de **kernel** (**coada de mesaje**) pentru a fi citita de catre alt proces. Procesul destinatar va citi din aceasta coada intregul mesaj.

Spre deosebire de **PIPE-uri,** din care se puteau citi mai putini octeti decat au fost scrisi, cozile de mesaje sunt orientate spre mesaj. Adica informatia trebuie sa ajunga in aceeasi forma la destinatie.

Dupa cum ii spune si numele, **coada** de mesaje respecta principiul **FIFO (First In First Out).** Mesajele sunt citite in ordinea introducerii in coada. Citirea este distructiva, adica odata un mesaj citit, acesta iese din coada, deci un mesaj poate sa fie citit de catre un singur proces.

Dupa cum am vorbit mai sus, fiecare proces manipuleaza un obiect de tip **IPC** printr-un identificator intern, obtinut dupa cheia unica asociata unui obiect din sistem. Pentru a obtine acest identificator dintr-un program C vom folosi functia:

#include <sys/types.h>
#include <sys/msg.h>

int msgget(key_t key , int msgflg);

Aceasta functie in caz de succes va intoarce respectivul identificator iar in caz de esec va intoarce -1. Argumentul **key** reprezinta cheia unica a obiectului. Spre exemplu, daca in urma introducerii comenzii :

$ ipcs -q

obtinem urmatorul afisaj :

------ Message Queues --------

key	msqid	owner	perms	used-bytes	messages
0x187525c2	327684	andrei	644	0	0

Atunci parametrul **key** poate avea valoarea **0x187525c2.** Functia **msgget** poate sa fie folosita si pentru a creea un nou obiect de tip **IPC.** Aici intervin flag-urile transmise prin parametrul **msgflg.**

Daca cheia **key** nu refera nici un obiect, atunci flag-ul **IPC_CREAT** trebuie setat pentru a creea un

obiect cu acest identificator. Problema apare atunci cand un obiect cu aceasta cheie exista si noi am specificat acest flag. Atunci obiectul actual va fi distrus si recreeat cea ce inseamna ca datele din el o sa fie pierdute, situatie ce ar putea fi nedorita deoarece poate acest obiect reprezenta deja mijlocul de comunicare intr-o doua procese active. Din acest motiv este bine sa specificam flag-ul **IPC_EXCL** ce va face ca functia **msgget** sa intoarca -1 in cazul in care obiectul cu cheia respectiva exista si flag-ul **IPC_CREAT** a fost si el setat.

In cazul in care un obiect nou este creeat. In parametrul **msgflg** trebuie specificat si drepturile pentru acest **IPC.** Aceste drepturi sunt reprezentate prin aceeasi masca ca la parametrul **mode** din functia :

int open(const char *pathname, int flags, mode_t mode);

Daca parametrul **key** este constanta **IPC_PRIVATE** atunci nu mai este necesar specificarea flag-ului **IPC_CREAT** ci doar masca de drepturi.

Urmatorul program va creea o coada de mesaje cu drepturi de citire si scriere pentru propietar. Cheia o sa o generam prin apel la **ftok()**.

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/msg.h>
#include<sys/stat.h>

int main()
{
      key_t cheia = ftok("/home/andrei/fisier",7);

      int msgqid = msgget(cheia, IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR);

      if ( msgqid != -1 )
     {
        puts("Succes!");
        return 0;
     }
     else
    {
         puts("Nu a mers!");
         return -1;
     }
}
```

Codul echivalent pentru cazul cand doream sa cream un obiect **IPC privat** este:

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/msg.h>
#include<sys/stat.h>

int main()
{
```

```
    int msgqid = msgget( IPC_PRIVATE , S_IRUSR | S_IWUSR);

     if ( msgqid != -1 )
     {
        puts("Succes!");
        return 0;
     }
     else
    {
         puts("Nu a mers!");
         return -1;
     }
}
```

In cazul trimiterii si receptionarii de mesaje, coada de mesaje are anumite propietati comune cu **PIPE-ul**, si anume ca citirea dintr-o coada de mesaje goala si scrierea intr-o coada de mesaje plina sunt implicit operatii blocante. Dar, la fel ca la tuburi, comportamentul acestor operatii se poate modifica astfel incat ele sa nu mai fie blocante. In cazul in care procesul se afla blocat in una dintre aceste operatii si coada de mesaje este stearsa, atunci operatia va esua si procesul va fi trezit.

Formatul unui mesaj

Un mesaj reprezinta un obiect de tipul unei structuri definite de utilizator. Aceasta structura poate sa aiba oricate campuri de orice lungime numai cu conditia ca primul camp sa fie de tip **long.** Prin acest camp utilizatorul specifica tipul mesajului pe care vrea sa il trimita, acest camp o sa fie util atunci cand destinatarul doreste sa citeasca din coada de mesaje.

O structura pentru un mesaj poate arata ceva de genul:

```
struct info
{
     long tip_mesaj;
     int an;
     int luna;
     int zi;
}
```

sau

```
struct info
{
     long tip_mesaj;
     char nume[20];
     char prenume[20];
}
```

La general vorbind, o structura ce defineste un mesaj arata ceva de genul:

```
struct nume_structura {
    lont tip_mesaj;
   /* oricate campuri de orice tip * /
}
```

Pentru a trimite un mesaj vom folosi functia:

#include <sys/types.h>
#include <sys/msg.h>

int msgsnd(int msqid , const void * msgp , size_t msgsz , int msgflg);

Aceasta functie returneaza 0 in caz de succes si -1 in caz de esec. Primul argument este identificatorul intern al obiectului obtinut dupa apel la **msgget()** al doilea este adresa obiectului ce dorim sa il transmitem ca mesaj, al treilea argument este dimensiunea in octeti a mesajului, este important ca dimensiunea mesajului sa fie introdusa corect pentru a nu aparea probleme nedorite. Pentru a face ca trimiterea mesajului sa fie o operatie neblocanta putem specifica parametrul **msgflg** ca find constanta **IPC_NOWAIT,** altfel, putem lasa **msgflg** sa fie 0.

Pentru a citi un mesaj dintr-o coada de mesaje vom folosi functia:

#include <sys/types.h>
#include <sys/msg.h>

ssize_t msgrcv(int msqid , void * msgp , size_t maxmsgsz , long msgtyp , int msgflg);

Aceasta functie returneaza in caz de succes numarul de octeti ai mesajului citit si -1 in caz de esec. Primul argument este identificatorul intern al obiectului obtinut dupa apel la **msgget()**, al doilea argument este adresa unde dorim sa fie stocat mesajul, al treilea argument reprezinta numarul maxim de octeti pe care dorim sa ii citim, adica pe care ii putem stoca in **msgp,** daca dimensiuna urmatorului mesaj este mai mare decat **maxmsgz** atunci nici un mesaj nu va fi citit si functia va returna -1. Al patrulea argument reprezinta tipul mesajului pe care dorim sa il citim, in functie de valoare lui **msgtyp** distingem doua situatii:

- **msgtyp > 0** atunci este extras primul mesaj din coada ce are acest tip
- **msgtyp = 0** atunci este extras primul mesaj din coada, indiferent de tipul acestuia.

Tipul mesajului este reprezentat de campul obligatoriu **tip_mesaj** cu care trebuie sa inceapa orice structura ce defineste un mesaj. Daca dorim ca citirea din coada de mesaje sa fie neblocanta, putem seta **msgflg** la **IPC_NOWAIT,** daca nu, il putem pastra 0.

Presupunem ca la comanda **:**

$ ipcs -q

gasim urmatoarea coada:

------ Message Queues --------

key	msqid	owner	perms	used-bytes	messages
0x2b35d488	393220	andrei	644	0	0

Atunci programele C din care se vor creea doua procese ce isi vor transmite un mesaj intre ele sunt:

Expeditorul :

#include<stdio.h>
#include<stdlib.h>

```
#include<sys/types.h>
#include<sys/msg.h>
#include<string.h>

struct message
{
  long mtype;
  int zi;
  int luna;
  int an;
};
int main()
{
  int msgqid = msgget(0x2b35d488 ,0);

  struct message mymessage;

  mymessage.mtype = 3;
  mymessage.zi = 1;
  mymessage.luna = 5;
  mymessage.an = 1994;

  msgsnd(msgqid,&mymessage,sizeof(mymessage),0);

  return 0;
}
```

Destinatarul:

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<sys/msg.h>
#include<string.h>

struct message
{
  long mtype;
  int zi;
  int luna;
  int an;
};
int main()
{
  int msgqid = msgget(0x2b35d488,0);

  struct message mymessage;

  msgrcv(msgqid,&mymessage,sizeof(struct message),3,0);

  printf("%d %d %d\n",mymessage.zi,mymessage.luna,mymessage.an);
```

```
  return 0;
}
```

Cum am spus mai sus, **kernel-ul** mentine o structura de date pentru fiecare obiect de tip **IPC SYSTEM V.** Formatul acestei structuri difera pentru fiecare tip de obiect in parte. Caracteristicile pentru un obiect de tip **cozi de mesaje** sunt retinute intr-o structura de forma :

```
struct msqid_ds
{
        struct ipc_perm msg_perm;     /* Propietarul si drepturile */
        time_t         msg_stime;    /* Data ultimului msgsnd() */
        time_t         msg_rtime;    /* Data ultimului msgrcv() */
        time_t         msg_ctime;    /* Data ultimei modificari asupra cozii */
        unsigned long   __msg_cbytes; /* Numarul curent de octeti din coada */

        msgqnum_t      msg_qnum;     /* Numarul de mesaje din coada */
        msglen_t       msg_qbytes;   /* Numarul maxim de octeti permisi in coada */

        pid_t          msg_lspid;    /* PID-ul ultimului proces ce a facut msgsnd() */
        pid_t          msg_lrpid;    /* PID -ul ultimului proces ce a facut msgrcv() */

  };
```

Pentru a initializa un obiect de tipul acestei structuri cu informatile despre o anumita coada vom folosi functia:

```
#include <sys/types.h>
#include <sys/msg.h>

int msgctl(int msqid , int cmd , struct msqid_ds * buf );
```

Aceasta functie returneaza 0 in caz de succes si -1 in caz de eroare. Primul parametru reprezinta identificatorul intern al cozii de mesaje. Al doilea parametru reprezinta o optiune dintre trei posibile comenzi:

- **IPC_RMID** va sterge obiectul imediat, chiar daca exista procese ce il folosesc. In acest caz ultimul argument nici nu este necesar. Practic, ca sa stergi o coada de message, apelul ar arata ceva de genul:

  ```
  msgctl(msqid,IPC_RMID, NULL);
  ```

- **IPC_STAT** va pune la adresa indicata de **buf** caracteristicile cozii de mesaje dorite
- **IPC_SET** va seta caracteristicile obiectului dupa structura a carei adresa este indicata de **buf.**

Exercitii

1. Scrieti un program ce primeste in linia de comanda cheia unei cozi de mesaje si afiseaza toate informatile disponibile din structura de date asociata ei dupa care o sterge.

2. Scrieti un program care va copia un fisier din directorul curent in alt director. Programul va primi ca argumente in linia de comanda numele fisierului si directorul destinatie. Programul va face

fork() dupa care fiul va face **chdir()** in directorul destinatie unde trebuie copiat fisierul. Tatal si fiul vor comunica printr-o coada de mesaje publica pe care tatal o va creea, folosind pentru generarea cheii functia **ftok()** cu fisierul dat ca parametru. Fisierul va fi scris in directorul destinatie de catre fiu, tatal va trimite fiului continutul fisierului in bucati de cate 50 de octeti prin coada de mesaje. Pentru a nu se confunda mesajele din coada, tipul mesajului va fi **PID-ul** tatalui. Dupa ce aceasta operatie a reusit, si fiul si-a terminat executia, tatal va sterge coada de mesaje.

3. Scrieti un *talk* bazat pe cozi de mesaje. Mesajele vor avea ca tip **PID-ul** procesului destinatar.

Vectori de semafoare

Semafoarele nu reprezinta un mecanism de transmitere a unor informatii concrete intre mai mule procese. Ele reprezinta un mecanism prin care doua sau mai multe procese concurente isi pot sincroniza actiunile. Prin semafoare se pot evita anumite situatii neprevazute atunci cand un grup de procese manipuleaza o resursa comuna, de exemplu un fisier.

Semaforul este o valoare intreaga ne-negativa mentinuta de catre **kernel.** Toate operatile efectuate asupra semaforului sunt mediate de catre acesta. Operatile sunt de doua feluri:

- **aditie –** atunci cand se doreste incrementare valorii semaforului
- **substractie –** atunci cand se doreste decrementarea valorii semaforului

Valoarea semaforului nu are voie sa scada sub 0, astfel ca orice tentativa de substractie asupra semaforului ce ar putea duce la un rezultat negativ va fi blocata de catre kernel.

Semafoarele de tip **IPC SYSTEM V** sunt destul de greoaie de folosit. Nu exista foarte mare utilitate in a avea un **vector de semafoare** deoarece majoritatea aplicatilor folosesc numai unul singur. Mai mult, capacitatea de a putea scadea si aduna la semafor orice valoare aduce probleme, aceste operatii ar trebui sa foloseasca doar valoarea 1 ca al doilea operator. Din aceste considerente semafoarele **POSIX** sunt mult mai puternice, dar ele nu fac obiectul materiei.

Pentru a obtine identificatorul intern al unui vector de semafoare vom folosi functia:

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

int semget(key_t key , int nsems , int semflg);

Aceasta functie intoarce identificatorul respectiv in caz de succes si -1 in caz de esec. Functia este asemanatoare cu functia **msgget()** descrisa in detaliu mai sus. Diferenta este argumentul suplimentar **nsems** ce reprezinta numarul de semafoare din vectorul de semafoare tinta.

Daca dorim sa folosim functia **semget** pentru a creea un nou obiect, abordarea este similara cu cea de la cozile de mesaje.

Pentru a realiza o operatie asupra unui vector de semafoare vom folosi functia:

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

int semop(int semid , struct sembuf * sops , unsigned int nsops);

Aceasta functie intoarce 0 in caz de succes si -1 in caz de esec. Primul argument reprezinta identificatorul intern al semaforului obtinut dupa un apel la **semget()**. Al doilea argument reprezinta un pointer catre un vector de structuri de forma:

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

sem_num reprezinta identificatorul semaforului in cadrul vectorului de semafoare. Ca intr-un vector C, indexarea se face de la 0 la **numarul_de_semafoare – 1 .**

sem_op reprezinta operatia ce se doreste a fi efectuata asupra semaforului si anume:

- sem_op > 0 - incrementare cu valorea respectiva
- sem_op < 0 - decrementare cu valoarea respectiva
- sem_op = 0 - punerea procesului in asteptare pana cand valoarea semaforului va fi 0

sem_flg reprezinta flag-uri pentru operatia respectiva. Daca acest flag este setat la **IPC_NOWAIT** atunci operatile blocante precum decrementarea cu o valoare ce ar duce semaforul la o valoarea negativa vor deveni neblocante.

Al treilea argument al functiei **semop**, reprezinta numarul de operatii din vectorul transmis ca argument secund.

!!! ATENTIE !!! In cazul in care functia returneaza 0, este garantat ca toate operatile au reusit.

La tastarea comenzii :

$ ipcs -s

Obtinem urmatorul semafor:

```
------ Semaphore Arrays --------
key         semid     owner     perms     nsems
0x62e9954b 32768     andrei         600       2
```

Dupa cum se poate vedea, acest vector contine doua semafoare, urmatorul program va incrementa primul semafor cu valoarea 3 si al doilea semafor cu valoarea 1.

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/sem.h>

int main()
{
    int semid = semget(0x62e9954b,2,0);

    if ( semid == -1 )
    {
```

```
    puts("Nu a reusit");
    return -1;
  }

  struct sembuf operatii[2];

  operatii[0].sem_num = 0;
  operatii[0].sem_op = 3;
  operatii[0].sem_flg = 0;

  operatii[1].sem_num = 1;
  operatii[1].sem_op = 1;
  operatii[1].sem_flg = 0;

  int rezultat = semop(semid,operatii,2);

  if ( rezultat == -1 )
  {
    puts("Operatile nu au reusit");
    return -1;
  }
  return 0;
}
```

Structura de date pe care kernel-ul o mentine pentru un obiect de tip vector de semafoare este reprezentata prin structura:

```
 struct semid_ds
{
        struct ipc_perm sem_perm;  /* Permisiunile si propietarul */
        time_t          sem_otime; /* Data ultimei operatii */
        time_t          sem_ctime; /* Data ultimei modificari */
        unsigned long   sem_nsems; /* Numarul de semafoare din vector */
};
```

Functia pe care trebuie sa o folosim pentru a initializa un obiect de tipul acestei structuri cu caracteristicile unui anumit vector de semafoare este:

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

int semctl(int semid , int semnum , int cmd , ... /* union semun arg */);
```

Functia este in unele masuri similara cu **msgctl** dar difera in anumite perspective. Nu voi explica utilizarea functiei in detaliu aici deoarece este foarte bine explicata in pagina de manual (**man semctl**)**,** in cartea domnului Baranga la capitolul dedicat vectorilor de semafoare, in **The Linux Programming Interface** pagina 969. Vom vorbi desprea ea la laborator si ma puteti contacta oricand aveti nevoie.

!!! ATENTIE !!! La test o sa o consider stiuta

In continuare vom prezenta un program ce face **fork()** si utilizeaza semafoare pentru a sincroniza actiunile tata-fiu.

Urmatorul program va face **fork().** Tatal va scrie de 500 de ori litera 'a' la standard output iar fiul de 500 de ori litera 'b'.

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

int main()
{
   if ( fork() )
   {
      int i;
      for ( i = 0 ; i < 500; i++ )
        write(STDOUT_FILENO,"a",1);
      return 0;
   }
   else
   {
      int i;
      for ( i = 0 ; i < 500 ; i++ )
        write(STDOUT_FILENO,"b",1);
      return 0;
   }
}
```

La o rulare a acestui program vom vedea ca literele 'a' si 'b' apar amestecate. Acest lucru se datoreaza faptului ca nucleul sistemului aloca o cuanta de timp celor doua procese astfel incat sa fie echitabil cu fiecare. Vom folosi un semafor pentru a ne asigura ca tatal va afisa toate literele inainte ca fiul sa afiseze ceva. Semaforul va fi un vector de semafoare privat ce va contine un singur semafor.

Codul programului nostru devine astfel:

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

int main()
{

   int semid = semget(IPC_PRIVATE, 1 , S_IRUSR | S_IWUSR);

   if ( fork() )
```

```
    {
       struct sembuf operatie_de_eliberare;
       operatie_de_eliberare.sem_num = 0;
       operatie_de_eliberare.sem_op = 1;
       operatie_de_eliberare.sem_flg = 0;

       int i;
       for ( i = 0 ; i < 500; i++ )
          write(STDOUT_FILENO,"a",1);

       semop(semid,&operatie_de_eliberare,1);
      /* Incrementeaza semaforul si trezeste fiul */

       return 0;
    }
    else
    {
       struct sembuf operatie_de_asteptare;

       operatie_de_asteptare.sem_num = 0;
       operatie_de_asteptare.sem_op = -1;
       operatie_de_asteptare.sem_flg = 0;
       semop(semid,&operatie_de_asteptare,1);
       /* Procesul incearca sa decrementeze un semafor aflat pe 0, deci va intra in starea adormit */

       int i;
       for ( i = 0 ; i < 500 ; i++ )
          write(STDOUT_FILENO,"b",1);
       return 0;
    }
}
```

Acum vom sti sigur ca la oricate rulari ale programului se vor afisa 500 de litere 'a' si 500 de litere 'b'

Urmatorul program exemplifica cum putem face o operatie neblocanta asupra unui semafor:

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/sem.h>
#include<errno.h>

extern int errno;

int main()
{
   int semid = semget( 0x2c2c8caf, 1,0);

   if ( semid == -1 )
     return -1;
```

```
    struct sembuf operatie;

    operatie.sem_num = 0;
    operatie.sem_op = -1;
    operatie.sem_flg = IPC_NOWAIT ;
    int rezultat;

    while ( 1 )
    {
       rezultat = semop( semid, &operatie, 1);

      if ( rezultat == 0)
         break;
       if ( errno == EAGAIN )
       {
            puts("Operatie nepermisa");
            sleep(2);
       }
    }

     puts("Operatie efectuata!");
     return 0;
}
```

Segmente de memorie partajata

Ultimul mecanism de tip **IPC SYSTEM V** despre care o sa vorbim sunt segmentele de memorie partajata.

Segmentele de memorie partajata reprezinta un mecanism foarte rapid de comunicare intre procese deoarece aceasta comunicare nu mai este mediata de catre **kernel.** Dupa cum am vazut in capitolul dedicat proceselor, sistemele de operare moderne implementeaza mecanismul de **memorie virtuala.** Acest lucru face ca procesele sa nu refere direct adrese fizice din memoria **RAM** ci adrese virtuale ce sunt mai apoi translatate de catre **MMU (Memory Management Unit)** la adrese fizice.

Avand aceste lucruri in vedere putem privi mecanismul de memorie partajata ca pe actiunea de mapare a adreselor virtuale a mai multor procese la aceeasi adresa fizica.

Pentru a obtine identificatorul intern al unui obiect de acest tip vom folosi functia:

#include <sys/types.h>
#include <sys/shm.h>

int shmget(key_t key , size_t size , int shmflg);

In caz de succes functia va intoarce acest identificator iar in caz de esec va intoarce -1. Functia se aseamana cu celelalte doua functii studiate anterior (**msgget()** si **semget()**). Al doilea parametru reprezinta numarul de octeti al zonei de memorie. In cazul in care se deschide un obiect deja existent, acest parametru nu trebuie sa depaseasca dimensiuna declarata la crearea sa.

Pentru a crea un nou **IPC** se procedeaza la fel ca la cozile de mesaje si la vectorii de semafoare.

Odata ce am obtinut identificatorul intern al memoriei partajate trebuie sa ne **atasam** la ea. Prin **atasare** intelegem un apel sistem facut **kernel-ul** prin care cerem sa mapeze o adresa virtuala a procesului la adresa fizica reprezentata de **IPC.**

Acest lucru se face prin functia:

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

void *shmat(int shmid , const void * shmaddr , int shmflg );
```

Functia va intoarce in caz de succes adresa virtuala unde a fost mapat obiectul sau **NULL** in caz de esec. Primul argument reprezinta identificatorul intern obtinut prin apel la **shmget()**. Al doilea argument il reprezinta adresa virtuala unde am vrea noi ca sa fie atasat obiectul. Nu este bine totusi sa furnizam noi acest parametru deoarece nu avem o vedere asa clara asupra intreg spatiului de adresare virtual al procesului. Astfel, al doilea parametru este bine sa fie transmis ca **NULL** pentru a lasa la latitudinea nucleului zona de adresare. Ultimul paramatru reprezinta diferite flag-uri asupra memoriei. Din motive pe care nu le vom explica aici, este bine ca flag-ul sa aiba valoarea **SHM_RND**. Putem specifica prin disjunctie si flag-ul **SHM_RDONLY** ce va marca zona de memorie ca read-only. Acest lucru inseamna ca orice tentativa de a scrie la aceasta adresa va genera un **SIGSEGV.**

Pentru a ne **detasa** folosim functia:

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

int shmdt(const void * shmaddr );
```

Functia va returna 0 in caz de succes si -1 in caz de esec. Pointer-ul transmis ca parametru trebuie sa fie obtinut in prealabil prin apel la **shmat().**

Structura de date pe care kernel-ul o mentine pentru un obiect de tip segment de memorie partajata este reprezentata prin structura:

```
 struct shmid_ds
{
        struct ipc_perm shm_perm;    /* Propietarul si permisiunile */
        size_t         shm_segsz;   /* Marimea segmentului in octeti*/
        time_t          shm_atime;   /* Data ultimei atasari */
        time_t          shm_dtime;   /* Data ultimei detasari */
        time_t          shm_ctime;   /* Data ultimei modificari asupra memoriei */
        pid_t          shm_cpid;    /* PID-ul procesului creator */
        pid_t          shm_lpid;    /* PID ultimului proces care a facut shmat sau shmdt */
        shmatt_t        shm_nattch;  /* Numarul de atasari curente */
 };
```

Pentru a initializa un obiect de tipul acestei structuri cu informatile despre un anumit segment de memorie vom folosi functia:

#include <sys/types.h>
#include <sys/shm.h>

int shmctl(int shmid , int cmd , struct shmid_ds * buf);

Aceasta functie returneaza 0 in caz de succes si -1 in caz de eroare. Primul parametru reprezinta identificatorul intern al **IPC-ului.** Al doilea parametru reprezinta o optiune dintre trei posibile comenzi:

- **IPC_RMID** va sterge obiectul imediat doar daca nu exista procese atasate la el, daca exista procese, atunci acesta va fi marcat pentru stergere si nici un alt proces nu se va mai putea atasa la el. In acest caz ultimul argument nici nu este necesar. Practic, ca sa stergi un segment de memorie partajat, apelul ar arata ceva de genul:

  shmctl(shmid, IPC_RMID, NULL);

- **IPC_STAT** va pune la adresa indicata de **buf** caracteristicile obiectului dorit
- **IPC_SET** va seta caracteristicile obiectului dupa structura a carei adresa este indicata de **buf.**

Problema producator-consumator

Sa presupunem urmatoarea problema:

Un program care face **fork()**. Fiul si tatal comunica printr-un segment de memorie partajata de un octet. Tatal scrie fiecare litera din alfabetul englez (de la 'a' la 'z') in zona respectiva. Fiul citeste 26 de litere (cate are alfabetul englez) din zona de memorie partajata si le afiseaza pe fiecare in parte. Programul corespunzator ar arata asa:

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/shm.h>
#include<sys/wait.h>
#include<sys/stat.h>

int main()
{
  int shmid = shmget( IPC_PRIVATE, 1, S_IRUSR | S_IWUSR );

  if ( fork() )
  {
      char *p = shmat( shmid, NULL, SHM_RND );
      char c;

      for ( c = 'a' ; c <= 'z'; c++)
          *p = c;
      wait(NULL);
      shmdt(p);
      shmctl(shmid, IPC_RMID, NULL );
      return 0;
```

```
  }
  else
  {
      char *p = shmat( shmid, NULL, SHM_RND );
      int i;
      for (  i = 1 ; i <= 26 ; i++ )
         write(STDOUT_FILENO,p,1);
      puts("");
      shmdt(p);
      return 0;
  }
}
```

Apare insa o problema daca am dori ca alfabetul sa fie corect afisat de catre fiu la standard output. Modul in care **kernel-ul** va executa cele doua procese nu depinde de noi ci de algoritmul intern de **scheduling.** Prim urmare, tatal poate sa scrie peste o litera pe care fiul nu a apucat sa o citeasca sau fiul sa citeasca de doua ori aceeasi litera. Apare astfel o problema ce in literatura de specialitate poarta numele de "**producator – consumator".**

In acest scenariu, avem doua entitati pe care le numim generic **producator** si **consumator.** Este important ca cele doua sa aiba un comportament bine definit pentru a nu exista probleme de sincronizare. Pentru a impune acest comportament, este necesar sa folosim un mecanism de sincronizare.

In modelul rezolvat de mine, vom utiliza doua semafoare, un semafor pentru fiu si un semafor pentru tata. Fiul va astepta la primul semafor pana cand tatal va anunta ca a inteprins o scriere, dupa care acesta se va trezi si va citi. Tatal, dupa ce a scris, va astepta la un semafor pana cand fiul va termina de citit si el va putea sa isi continue treaba. Programul de mai sus modificat astfel incat sa foloseasca semaforul ca mecanism de sincronizare este:

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

int main()
{
  int shmid = shmget( IPC_PRIVATE, 1, S_IRUSR | S_IWUSR );
  int semafor_fiu = semget( IPC_PRIVATE, 1, S_IRUSR | S_IWUSR );
  int semafor_tata = semget( IPC_PRIVATE, 1, S_IRUSR | S_IWUSR );

  semctl(semafor_fiu,0,SETVAL,0);
  semctl(semafor_tata,0,SETVAL,0);
  /*Ne asiguram ca cele doua semafoare sunt pe 0 */

  if ( fork() )
  {
     char *p = shmat( shmid, NULL, SHM_RND );
     char c;
```

```
    /* Ne definim cele doua operatii asupra semafoarelor */
    struct sembuf trezire_fiu;
    trezire_fiu.sem_num = 0;
    trezire_fiu.sem_op = 1;
    trezire_fiu.sem_flg = 0;

    struct sembuf asteptare_fiu;
    asteptare_fiu.sem_num = 0;
    asteptare_fiu.sem_op = -1;
    asteptare_fiu.sem_flg = 0;

    for ( c = 'a' ; c <= 'z'; c++)
    {
        *p = c;
        semop(semafor_fiu,&trezire_fiu,1);/*Trezim fiu */
        semop(semafor_tata,&asteptare_fiu,1); /* Punem tatal in asteptare*/
    }
    wait(NULL);
    /*asteptam fiul sa se termine */
    shmdt(p);
    shmctl(shmid, IPC_RMID, NULL );
    semctl(semafor_fiu, 0 ,IPC_RMID, 0);
    semctl(semafor_tata, 0 ,IPC_RMID, 0);
    /*stergem resursele */
    return 0;
}
else
{
   char *p = shmat( shmid, NULL, SHM_RND );
   int i;

  /*Ne definim cele doua operatii asupra semafoarelor */
   struct sembuf trezire_tata;
   trezire_tata.sem_num = 0;
   trezire_tata.sem_op = 1;
   trezire_tata.sem_flg = 0;

   struct sembuf asteptare_tata;

   asteptare_tata.sem_num = 0;
   asteptare_tata.sem_op = -1;
   asteptare_tata.sem_flg = 0;

   for (  i = 1 ; i <= 26 ; i++ )
   {
      semop(semafor_fiu,&asteptare_tata,1);/*Punem fiul in asteptare*/
      write(STDOUT_FILENO,p,1);
      semop(semafor_tata,&trezire_tata,1);/*Trezim tatal */
   }
   puts("");
   shmdt(p);
```

```
    return 0;
  }
}
```

Acum suntem siguri ca indiferent de cate ori vom rula programul comportamentul este unul bine definit.

Exercitii

1. Scrieti un program care foloseste semafoare pentru a sincroniza tatal si fiul dupa apel la **fork()** in loc de functia **wait()**

2. Scrieti un program care primeste ca argument in linia de comanda cheia pentru un obiect de tip segment de memorie partajata si afiseaza toata informatile din structura de date din kernel asociata ei.

3. Aceeasi problema ca la **2** numai ca acum cheia este a unui vector de semafoare.

4. Scrieti un program care lanseaza 5 procese fiu. Fiecare proces va fi atasat la o zona de memorie partajata si doua semafoare. Tatal va intra intr-un ciclu infinit ce citeste informatii de la tastatura. Fii vor astepta la semafor, care initial este pe 0. Tatal va citi de la utilizator cate un string ce reprezinta calea catre un fisier dupa care va scrie aceasta cale in zona de memorie partajata si va ridica semaforul cu 1. Fiul va citi din zona de memorie calea si va numara cate cuvinte contine acel fisier, dupa care le va afisa la standard output. Tatal va astepta la un semafor pentru a fi sigur ca cel putin un fiu a citit calea catre fisierul dorit dupa care isi va continua executia. Dupa ce ce un fiu isi termina treaba revine la semafor pentru a astepta noi comenzi. Deci nici un proces nu se termina decat explicit (de exemplu cand primesc un semnal)!