Apelul sistem **vfork – aceeasi secventa de apel si valoare returnata - fork**

A fost introdus **ca o forma de optimizare pt a doua situatie de utilizare a lui fork**;

- daca noul proces urmeaza sa execute un alt program nu se mai justifica operatia de copiere a segmentelor programului parintelui: noul proces va utiliz acelasi spatiu de adrese ca si parintele pina cind apeleaza **exec** sau pina se termina.

In plus **vfork** garanteaza ca **fiul are prioritate la executie** fata de parinte pina cind apeleaza **exec** sau **exit**.

Asteptarea terminarii unui proces

Terminarea normala a unui proces de catre el insusi:

Apelurile functiilor: exit( ), _exit( )

void exit(int cod_exit);
void _exit(int cod_exit);

Efect: - cod_exit este o valoare intreaga care se returneaza parintelui pt ca acesta sa poata analiza modul de terminare al fiului.
- un cod_exit este 0 = terminare fara eroare a procesului
- orice alta valoare este utilizata pentru a semnala o eroare.
- _exit a fost definita pentru ca exit definita in C nu lua in considerare toate posibilitatile oferite de programarea concurenta.

Operatii ce au loc la executia functiei exit:

- Ignorarea tuturor semnalelor care se transmit procesului;
- Inchiderea tuturor fisierelor deschise;
- Eliberarea directorului curent
- Starea procesului apelant devine terminat;
- Tuturor proceselor care au procesul apelant ca parinte li se schimba parintele. Procesul init preia toate procesele orfane.
- Trimiterea unui mesaj parintelui pt ca acesta sa poata analiza cum s-a terminat fiul sau.

Problema: ce se intimpla cu starea de terminare a procesului care poate fi obtinuta de procesul parinte la o terminare normala a unui proces fiu.

a) Procesul fiu se termina inaintea procesului parinte;

b) Procesul parinte se termina inaintea procesului fiu.

Actiuni la terminarea unui proces:

- nucleul parcurge lista proceselor inca active in sistem, pt a vedea daca vreunul din ele este **parintele** celui terminat.

- se verifica daca in acel moment mai exista **fii neterminati** ai celui in cauza. In caz afirmativ, **nucleul face pt fiecare fiu modificarile necesare astfel incit in continuare parintele proceselor fii respective sa fie procesul init** => **situatia b) e rezolvata de nucleu**.

Cazul a)

- procesul parinte va fi anuntat printr-un semnal: **SIGCHILD** de terminarea fiului si va trebui sa execute unul din apelurile **wait** pentru a **prelua starea de terminare a fiului.**

-pina in momentul executiei unui **wait**, nucleul pastreaza un minim de informatii despre procesul terminat: PID-ul, starea de terminare, timpul de procesor utilizat (fisiere deschise, memorie, etc. pot fi eliberate) **<=>** procesul e in starea **zombie intre momentul terminarii si cel al executiei unui wait de catre parinte. Acest tip de proces nu are alocata decit o intrare in tabela de procese.**

Procesul zombie poate fi observat cu ajutorul c-zi shell ps care e lansata cu ajutorul functiei sistem

```
#include "hdr.h"

int main (void)
{
        pid_t pid;
        if((pid=fork())==-1
                err_sys("Eroare fork 1");
        else if (pid==0)
                exit(0);
        /*procesul parinte*/
        sleep(3);
        system("ps");
        exit(0);
}
```

```
  PID    TTY   TIME     CMD
 19163 pts/16  00:00:00 bash
 19249 pts/16  00:00:00 exemplul
 19250 pts/16  00:00:00 exemplul <defunct>
 19251 pts/16  00:00:00 ps
```

Procesele **mostenite de init** nu ramin in starea zombie, pt ca init este astfel scris incit sa execute **wait** la **terminarea oricarui fiu**.

OBS daca un proces genereaza multe procese fii si nu executa apeluri wait, la terminarea acestora, este posibil ca sistemul de operare sa intre in criza de resurse cum ar fi intrarile in tabela de procese(eliberarea nu se face decit atunci cind procesul paraseste sistemul - dupa preluarea starii de terminare).

Generarea starii de asteptare a unui proces

Functiile wait() si wait pid()

Un proces ce apeleaza aceste functii poate sa:

- Blocheze daca toti fii sunt in executie
- Receptioneze modul de terminare al fiului care se termina
- Daca nu are fi receptioneaza eroare deoarece este anormal sa astepte

#include <sys/types.h>

#include <sys/wait.h>

pid_t wait(int *statloc);

pid_t waitpid(pid_t *pid*, int **statloc*, int *options*);

Obs Ambele returneaza **PID-ul** unui proces in caz de executie fara eroare, respectiv valorile 0 sau -1 in caz de executie fara eroare.

statloc(status) – este un pointer spre locatia de memorie unde sistemul va depune informatia de stare

Informatia de stare e codificata pe 16 biti si cuprinde codul de exit sau numarul semnalului care a determinat terminarea/blocarea temporara a procesului fiu. Poate fi si NULL, caz in care nu se memoreaza starea ce a determinat terminarea/stoparea procesului fiu.

1. Procesul fiu s-a terminat utilizind exit:

15								8								0
Cod exit al proc fiu asteptat								0	0	0	0	0	0	0	0	

2. Procesul fiu s-a terminat prin receptia unui semnal. X indica daca s-a produs vidaj de memorie sau nu

15								8	7	
0	0	0	0	0	0	0	0	X		Numar semnal receptionat de fiu

3. Daca procesul fiu este doar oprit temporar

15								8								0
Numar semnal receptionat de fiu								0	0	0	0	0	0	0	0	

Obs Ambele returneaza **PID-ul** unui proces in caz de executie fara eroare, respectiv valorile 0 sau -1 in caz de executie fara eroare.

Diferentele semantice intre cele doua apeluri:

- **wait**: asteapta terminarea oricaruia dintre procesele fii ale procesului care executa apelul, pe cind **waitpid** permite precizarea **unor conditii asupra identitatii procesului a carui terminare e asteptata**.

- **wait** duce la blocarea procesului apelant daca in momentul apelului nu exista nici un proces fiu terminat, pe cind **waitpid** are o optiune prin care blocarea poate fi prevenita.

Daca apelul la **wait** se face ca urmare a primirii de catre proces a semnalului **SIGCHLD**, situatia normala este ca revenirea din apel sa se produca imediat (exista un proces zombie, a carui stare este returnata). Altfel procesul se poate bloca pina la terminarea unui proces fiu. **Identitatea pro-**

cesului terminat este returnata ca valoare a functiei de apel.

Efectul waitpid() depinde de valoarea pid in modul urmator:

- daca pid==-1 se asteapta treminarea oricarui proces fiu si deci avem o actiune similara cu cea a functiei **wait**.
- pid>0 se asteapta terminare identificatorului cu id-ul pid.
- **-** pid==0 se asteapta terminarea unui proces cu id-ul de grup identic cu cel al procesului apelant
- pid<-1 se asteapta terminarea unui proces care are id-ul de grup egal cu valoarea absoluta a pid.

Argumentul opt poate lua valorile

- 0
- WNOHANG
- WUNTRACED
- valoarea obtinuta prin sau logic a celor 2 const simbolice

opt = WNOHANG, procesul ce apeleaza waitpid() continua imediat daca nu existe fii care sa se fii executat.

opt = WUNTRACED procesul continua daca exista exista procese oprite sau a caror stari nu au fost raportate

waitpid returneaza 0 daca *opt* = WNOHANG si procesul apelant nu are fii.

Pt **accesul** la starea de terminare sunt definite macrouri in (**<sys/wait.h>**)

Macro	**Descriere**
WIFEXITED (status)	**true** daca s-a returnat starea unui fiu treminat normal. In acest caz se poate executa macro-ul pt a ob codul transmis in status: **WEXITSTATUS** (status)-returneza codul de exit
WIFSIGNALED (status)	**true** daca s-a returnat starea unui fiu terminat datorita unui semnal. In acest caz prin **WTERMSIG** (*status*) se obtine nr semnalului care a cauzat terminarea. In plus prin: **WCOREDUMP** (*status*) se obtine **true** daca s-a generat fisier **core.**
WIFSTOPPED (status)	**true** daca s-a returnat starea unui fiu momentan oprit. In acest caz prin: **WSTOPSIG** (*status*) se obtine numarul semnalului care a cauzat oprirea.

```
pid_t  pid;
int status;
    if ((pid=fork( ) ) <0)          //apelez fork
          printf("eroare fork\n");
    else if (pid==0 )               //fiu
    exit(7);                    // se iasa din procesul fiu si se returneaza 7
                                // parinte
    if (wait (&status) != pid) // asteapta terminarea unicului proces fiu
          printf ("eroare wait");
    if(WIFEXITED(status))   // se afiseaza o descriere a starii de terminare
          printf("terminare normala, exit status =%d\n",
          WEXITSTATUS(status);
La rulare…
$ ./wait
terminare normala, exit status=7
```

Invocarea unui proces - Apelurile **exec**

Reprezinta initierea executiei unui program, altul decit cel din care face parte procesul apelant. Ca urmare a acestei invocari, segmentul de text si date al procesului apelant sunt inlocuite cu continutul noului program, provenind de la un fisier executabil.

Obs – Exista mai multe variante ale acestui apel, **numai una dintre ele fiind in realitate implementata ca apel sistem, iar celelalte sunt doar functii biblioteca.**

#include <unistd.h>
int execl (const char *_pathname_, const char *_arg0, …/ * (char*) */);
int execv (const char *_pathname_, const char *_argv[ ])
int execle (const char *_pathname_, const char *_arg0,…/* (char*)0, char *const envp[ ] */);
int execve (const char *_pathname_, const char *_argv[], const char *_envp[ ]);
int execlp (const char *_filename_, const char *_arg0, …/ * (char*) */);
int execvp (const char *_filename_, const char *_argv[]);

Diferente: **l** Calea de cautare a fisierului executabil (***pathname*** versus ***filename).***
Utilizarea unei cai este aratata prin prezenta literei p spre deosebire de cazul cind se utilizeaza directorul curent.

- daca *filename* contine un caracter (***/***) se considera ca el este un nume cale ca si *pathname*;

- altfel se cauta un **fisier executabil** cu nume in cataloagele specificate de variabila **PATH** .

A-II-a diferenta:

- modul de transmitere al argumentelor spre programul apelat: fie ca lista (**execl, execle, execlp**) fie ca vector (**execv, execve, execvp**)

- in ambele cazuri ultimul argument trebuie sa fie pointerul nul.

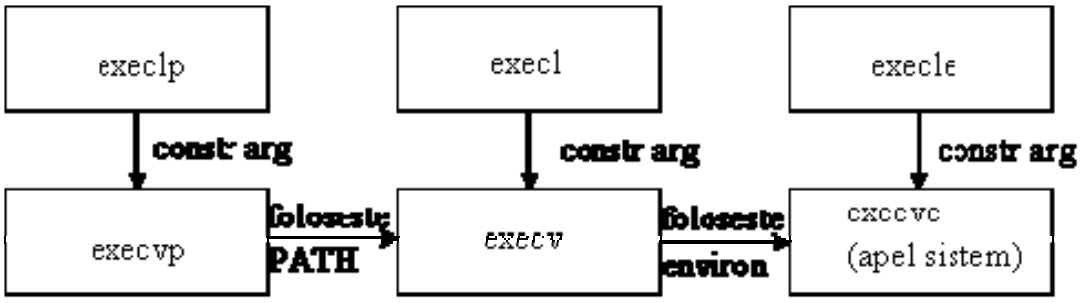
- in cazul vectorilor, acestia trebuie construiti inaintea apelului, elementele fiind pointeri catre siruri de caractere ce reprezinta argumente pentru programul lansat in executie.

A-III-a diferenta: modul in care se transmite lista variabilelor de mediu catre noul program: - **execle** si **execve** permit ca ultim argument al apelului un **tablou de pointeri la variabilele de mediu**, pt celelalte **4** se foloseste implicit variabila **environ** a procesului apelant pt a copia variabilele de mediu in ambianta noului program.

execle si **execve** modofica variabilele de mediu: HOME (calea spre directorul home), PATH(lista de directoare in care se cauta c-zile) , MAIL(fiiserul ce stocheaza mesajele primite prin mail), SHELL(caleaspre fisierul binar ce contine shell-ul).

Relatia intre cele 6 apeluri permite concluzia: e suficient ca **numai execve sa fie implementat efectiv ca apel sistem.**

Relatia intre cele 6 apeluri exec



Desi dupa apelul acestor functii se executa cod apartinand altui program, se considera ca este vorba de acelasi proces.

Majoritatea parametrilor procesului apelant e conservata dupa realizarea apelului si executarea noului program.

Se mostenesc:

- id-ul
- id-ul parintelui
- id-ul grupului de uitlizatori
- directorul radacina
- directorul curent

Principalele actiuni care se realizeaza:

1. Se identifica fisierul ce ar trebui sa contina cod executabil
2. Se verifica ca fisierul respectiv este executabil
3. Se citeste antetul fisierului
4. Se salveaza parametrii de apel ai functiei
5. Se elibereaza zonele atribuite procesului pt a se putea incarca noul cod

Lista argumentelor: calea fisierului executabil, numele fisierului executabil, restul sunt parametrii actuali pt programul ce trebuie lansat in executie.

Alte aspecte ale gestionarii proceselor

Fisiere interpretor sau scripturi = fisierele text declarate executabile si care incep de regula cu o linie de forma:

#! *pathname [argumente optionale]*

Numele de cale este in mod normal un **nume de cale absolut** (ca si la exec)

Recunoasterea acestor fisiere este realizata de nucleu ca parte a prelucrarii apelurilor **exec.**

Daca linia nu e de forma de mai sus nucleul considera ca acolo apare:

#! /bin/sh

Actiunea nucleului: se cauta fisierul cu numele de cale indicat, presupus program executabil, se incarca si se lanseaza in executie.

Argumentele optionale: transmit informatii programului interpretor indicat prin *pathname.* Astfel, pt **awk** un fiser interpretor arata de regula:

#!/bin/awk –f

#comenzi awk- interpretor programabil pt interpretarea unei secente ..

Fara optiunea **–f** awk considera ca primul argument din linia de c-da dupa numele fisierului interpretor este cel in care isi gaseste c-zile(si nu chiar in continuarea fisierului interpretor).

Functia execl

#include <unistd.h>

int execl (const char **pathname*, const char **arg0,* const char *arg1* …NULL);

Exemplu de apel

execl(“/bin/time”, “time”,”ps”, NULL);

Functia execv

#include <unistd.h>

int execl (const char **pathname*, const char **argv[ ]*);

pathname= pointer la un sir de carcactere care reprezinta calea fisierului executabil

argv= pointer la un vector ce contine pointerii sirurilor de caractere transmise ca parametrii

Exemplu de apelare

char*argv[3]

argv[0]=time

argv[1]=“ps”

argv[2]=(char*2)0;

execv((“/bin/time”, argv)

Gasirea si modificarea unor atribute ale proceselor

Identificatori asociati proceselor la deschiderea unei sesiuni de lucru

2 care identifica pe cel care a deschis sesiunea:
- Identificatorul utilizatorului real;
- Identificatorul grupului real;

2 care precizeaza posibilitatea de acces la fisiere
- Identificatorul utilizatorului efectiv;
- Identificatorul grupului efectiv;

2 salvati in momentul apelului unei functii din familia exec()
- Identificatorul salvat la modificarea utilizatorului;
- Identificatorul salvat la modificarea grupului.

Procesul nou creat cu ajutorul unei functii **exec** va avea **identificator al user-ului real si unul al userului efectiv, unul al grupului real si unul al grupului efectiv.**

Acestia **depind de procesul initial, de proprietarul fisierului care este lansat in executie si de bitul set-user-ID.**

Daca bitul **set-user-ID** corespunzator fisierului lansat in executie e **setat** atunci identificatorul userului efectiv al noului proces devine proprietarul fisierului care se executa.

Daca bitul **set-group-ID** corespunzator fisierului lansat in executie e **setat** atunci identificatorul grupului efectiv al noului proces devine grupul proprietarului fisierului care se executa.

Identificatorul utilizatorului real si a grupului real al noului proces vor fi cei ai vechiului proces.

Valorile identificatorilor utilizatorului efectiv si a grupului efectiv sunt salvati si sunt denumiti “saved set-user-ID” respectiv “saved set-grup-ID”.

Aflarea identificatorului de proces:

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
```

Aflarea identificatorului procesului parinte:

```
#include <sys/types.h>
#include <unistd.h>
pid_t getppid(void);
```

Aflarea identificatorului utilizatorului real:

```
#include <sys/types.h>
#include <unistd.h>
uid_t getuid(void);
```

Aflarea identificatorului grupului real:

```
#include <sys/types.h>
#include <unistd.h>
gid_t getgid(void);
```

Exemplu:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main (void)
 {
     printf("uid=%d, gid=%d, getuid( ), getgid( ));
     exit(0);
}
```

Aflarea identificatorului utilizatorului efectiv al procesului:

#include <sys/types.h>

#include <unistd.h>

uid_t geteuid(void);

Modificarea identificatorului utilizatorului real si al grupului real al unui proces (Acces la modificare rezervat)

#include <sys/types.h>

#include <unistd.h>

int setuid(uid_t uid);

int setgid(gid_t gid);

-returneaza 0 (succes) si -1(eroare).

Daca procesul are drepturi de superuser, functia **setuid** seteaza ID-ul utilizatorului, efectiv si “saved set-user-ID” la **uid.**

Daca procesul nu are drepturi de superuser dar uid este egal cu identificatorul utilizatorului real sau cu “saved set-user-ID”, functia setuid seteaza identificatorul utilizatorului efectiv la **uid**.

Modificarea identificatorului utilizatorului efectiv si real precum si al grupului efectiv si real al unui proces

Aceste functii au acces la doi dintre cei 3 identif referitori la utilizator sau grup.

#include <sys/types.h>

#include <unistd.h>

int setreuid(uid_t ruid, uid_t euid);

int setregid(gid_t rgid, gid_t egid);

Super-userul poate realiza ce modificari doreste. Daca unul din parametrii are valoarea -1 acea valoare nu se va schimba.

Modificarea identificatorului utilizatorului efectiv al grupului efectiv al unui proces

include <sys/types.h>

#include <unistd.h>

int seteuid(uid_t uid);

int setegid(gid_t gid);

-returneaza 0 (succes) si -1(eroare).

Gestionarea grupurilor de procese

#include <sys/types.h>

#include <unistd.h>

pid_t getpgrp

Valoarea returnata e de tipul identificatorului de proces si este de fapt id-ul unui proces din grup denumit **lider**.

Un proces poate adera la un grup existent sau sa creeze unul nou cu ajutorul functiei setpgid

#include <sys/types.h>

#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);

Efect: procesul cu identificatorul pid devine component al grupului cu identificatorul pgid.

Obs: Se poate utiliza ac functie pt procesul apelant sau pt procesele fii care nu au apelat o functie din familia **exec**.

Daca pid=0atunci se face referinta la procesul apelant.

Daca cele 2 argumente sunt egale atunci procesul cu identificatorul respectiv devine lider de grup.

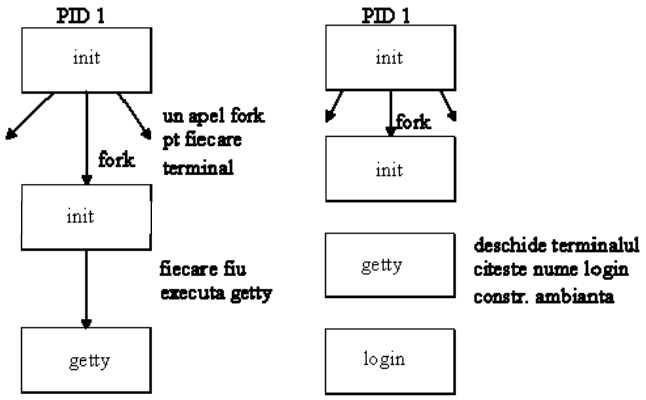
Relatii intre procese. Semnale

Procedura de login – modul in care un utilizator intra in comunicare cu un sistem UNIX.

La terminarea incarcarii partii rezidente a sistemului se creaza procesul cu identificatorul 1**, init,** care va trece sistemul in modul de **operare multiutilizator**.

In acest scop **Init** citeste informatiile din fisierul de configurare **/etc/ttys**, prevazut cu cite o linie pt fiecare terminal al sistemului.

Pt fiecare terminal care permite **login init** va apela un **fork**, urmat de un **exec** pt programul getty…



a) Procese create de init b) Starea de apelare la login

OBS: Toate procesele din fig au privilegii de superutilizator (UID real si efectiv egal cu zero) si ambianta vida (nici o variabila de mediu nu este inca definita)
Programul **getty** apeleaza **open** pt terminal in modul **read/write**(daca termi-nalul este in retea se poate produce o asteptare in **open** pina la stabilirea legaturii)
Dupa revenirea din **open** descriptorii de fisier 0,1si 2 sunt conectati la terminal, iar in continuare **getty** afiseaza textul **login: si asteapta ca utilizatorul sa intoduca textul de login.**

Grupuri de procese, sesiuni, terminale de control

Un grup de procese este o colectie de unul sau mai multe procese, cu un **identificator de grup de procese unic**. Fiecare grup de procese poate avea un **lider de grup de procese**, desemnat prin faptul ca **PID-ul sau este egal cu identificatorul grupului de procese.**

Un lider de grup de procese poate sa creeze un **grup de procese**, apoi sa creeze procese in cadrul grupului, dupa care sa se termine.

Grupul de procese isi va continua existenta cit timp va mai exista cel putin un proces in grup.

Parasirea grupului se poate face fie prin treminarea procesului fie cind procesul se ataseaza altui grup.

Apelul sistem setpgid

Scop: Ataseaza **un proces unui grup de procese** sau **creaza un nou grup de procese.**

Sintaxa:

#include <sys/types.h>

#include <unistd>

int setpgid(pid_t *pid* , pid_t *pgid);

Efect: - identificatorul de grup de procese pentru procesul ***pid*** devine ***pgid***.

- daca cele **2 argumente sunt egale**, procesul specificat de ***pid*** devine **lider de grup**.

- apelul returneaza 0 in caz de succes si -1 in caz de eroare.

OBS: - un proces poate sa modifice identificatorul de grup de procese doar pt el insusi sau pentru unul din fii sai; daca unul din fii a apelat **exec**, parintele nu ii mai poate schimba identificatorul de grup de procese.

- daca pid = 0 se foloseste PID-ul procesului apelant

- daca pgid=0 se foloseste ca valoare primul argument *pid.*

Apelul sistem getpgrp

Scop: **permite unui proces sa-si afle propriul identificator de grup.**

Sintaxa:

#include <sys/types.h>

#include <unistd>

pid_t getpgrp(void);

Valoarea returnata este identificatorul de grup de procese al procesului apelant.

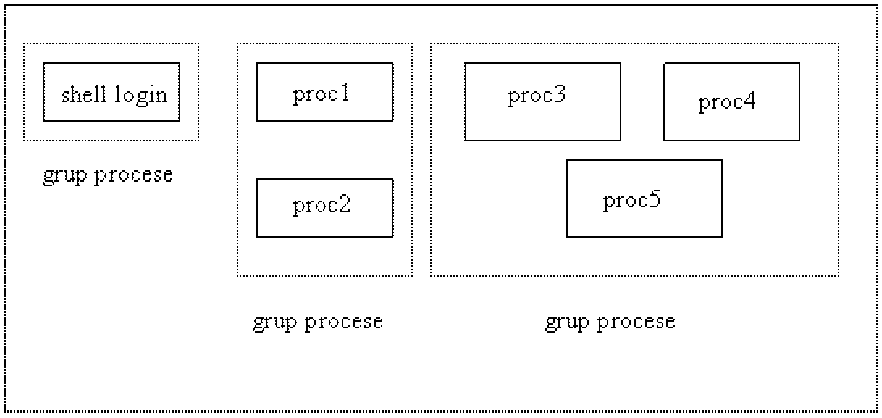
O sesiune este o colectie de unul sau mai multe grupuri de procese.

Exemplu:

proc1 | proc2 &

proc3 | proc4 | proc5

Creaza o sesiune formata din 3 grupuri de procese:



Apelul sistem **setsid**

Scop: permite unui proces sa stabileasca o noua sesiune.

#include <sys/types.h>
#include <unistd>
pid_t setsid(void);

Efect: valoarea returnata de apel este un **identificator de grup de procese** in caz de succes, **respectiv -1** in caz de eroare.

Daca **procesul apelant nu e lider de grup**, se creaza o **noua sesiune** si au loc urmatoarele:

- procesul devine **lider de sesiune** pt sesiunea nou creata si este momentan singurul proces din sesiune;
- procesul devine **lider de grup** al unui nou grup de procese, **PID**-ul sau devenind **identificatorul grupului de procese**;
- procesul nu mai are terminal de control(daca a avut terminal inainte de apelul setsid, asocierea e terminata).

Apelul **setsid** returneaza eroare daca procesul apelant este **deja lider de grup**.

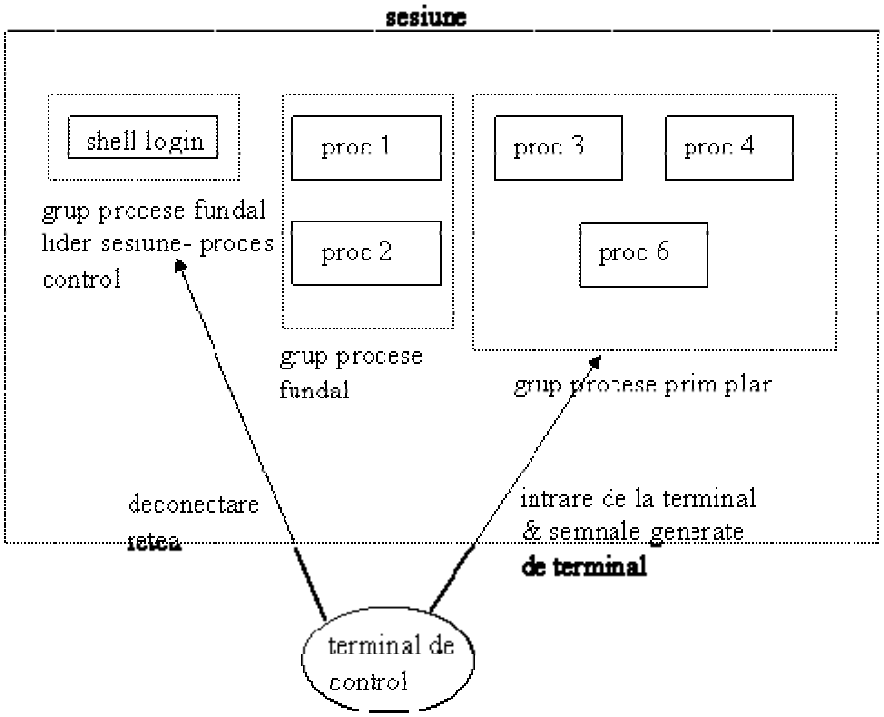
Pt a evita: procesul apeleaza **fork si apoi se termina asigurind astfel ca procesul fiu nu este lider de grup. Procesul fiu poate apela acum setsid si devine lider de sesiune.**

Principalele caracteristici ale sesiunilor si grupurilor de procese

- O **sesiune** poate avea **doar un terminal de control** (de la care s-a efectuat procedura de **login**);
- Liderul de sesiune care stabileste legatura cu terminalul de control se numeste **proces de control***;*
- Grupurile de procese dintr-o sesiune se pot imparti astfel: **un singur grup** de procese **in prim plan (foreground**) si **unul sau mai multe grupuri de procese in fundal (background)**;
- Daca o sesiune are terminal de control, ea va avea un grup de procese in prim plan iar celelalte in fundal;
- Actionarea tastei de intrerupere (DELETE sau Control-C sau quit) determina trimiterea unui semnal de intrerupere, respectiv de renuntare la toate procesele din grupul in prim plan;
- Daca se intrerupe **legatura in retea**, liderul de sesiune primeste un semnal de **hangup**.

Reprezentare grafica a acestor caracteristici:

Relatia terminalului de control cu sesiunea si grupurile de procese



Obs: exista posibilitatea ca un program sa comunice cu terminalul de control indiferent de faptul ca intrarea si/sau iesirea sa standard sunt redirectate.

Programul trebuie sa deschida fisierul special**: /dev/tty** care pt nucleu este sinonim cu terminalul de control. Daca programul nu are terminal de control **operatia de deschidere** va esua.

Functii biblioteca prin care se comunica nucleului care este grupul de procese in prim plan, respectiv un proces poate fixa identitatea grupului de procese in prim plan;

#include <sys/types.h>
#include <unistd>
pid_t tcgetpgrp(int *filedes*);
int tcpsetpgrp (int *filedes,* pid_t *pgrpid*);

Efect: - **tcgetpgrp**- returneaza identificatorul grupului de procese in prim plan asociat cu terminalul deschis la *filedes*

- daca un proces are terminal de control, el poate apela **tcsetpgrp** pt a fixa **pgrpid ca identificator al grupului de procese in prim plan**. Valoarea lui filedes corespunde terminalului deschis la *filedes*.

OBS cele 2 functii nu se apeleaza direct din programele de aplicatie ci se folosesc de catre interpretoarele de comenzi prevazute cu facilitati pt controlul joburilor

Controlul joburilor

Facilitate care permite ca:

- de la un singur terminal sa fie pornite mai multe joburi (grupuri de procese)

- sa se specifice care joburi au drept de acces la terminal si care sunt reluate in fundal.

Necesita 3 forme suport:

- **Un interpretor de c-zi (shell) adecvat;**
- **Un driver de terminal adecvat in nucleu;**
- **Suport pt semnale specifice controlului joburilor.**

OBS

Suportul necesar in shell este ca atunci cind se porneste un job in fundal, sa **se atribuie un identificator de job si sa se afiseze acest identificator impreuna cu cel putin un identificator de proces**.

Exemple

$ ls –al | grep ^d >ddd &

[1] 1179

$ make >Make.out &

[2] 1182

$ // aici trebuie ca utilizatorul sa actioneze tasta RETURN

[1] + Done ls –al | grep ^d >ddd &

[2] + Done $ make >Make.out &

La **terminarea joburilor** interpretorul de c-zi afiseaza mesaje de terminare, afisarea nu se produce decit imediat inainte ca interpretorul sa-si afiseze promptul de aceea a fost necesar sa se **tasteze RETURN.**

OBS

- Interactiunea apare pentru ca trebuie sa existe un caracter care afecteaza jobul din prim plan, tasta de suspendare, de regula **Control-Z**. Tastarea acestui caracter face ca drivul terminalului sa trimita semnalul **SIGTSTP** tuturor proceselor din grupul in prim plan.

Driverul terminalului trebuie sa poata recunoaste **trei caractere speciale care genereaza semnale pentru grupul de porcese in prim plan.**

- **Caracterul de intrerupere (de regula DELETE sau Control-C) care genereaza SIGINIT;**
- **Caracterul de renuntare (Control \), care genereaza SIGQUIT;**
- **Caracterul de suspendare (Control-Z) care genereaza SIGTSTP;**

Mai exista o situatie in care intervine drive-ul terminalului trebuie sa intervina: cind un job din fundal incearca sa citeasca de la terminal. In aceasta situatie, drive-ul terminalului trebuie sa detecteze acest lucru si sa trimita jobului respectiv semnalul SIGTTIN care duce la oprirea job-ului;

$ cat >temp & // pornit in fundal fara redirectarea intrarii (cat asteapta date de la tastatura

[1] 2045

$ //se tasteaza RETURN

[1] + Stopped (tty input)

Obs: job-ul este doar oprit nu terminat. El poate fi adus in prim plan prin **fg** si treminat.

Comunicare de la un job in fundal spre terminal poate fi permisa sau interzisa cu ajutorul **stty:**

$ cat temp & // citeste din fisierul temp

[1] 2051

hello world // continutul lui temp apare imediat

$ // aici tasta RETURN

[1] + Done cat temp &

$ stty tostop //comunicarea cu terminalul interzisa

$ cat temp &

[1] 2055

$ // tasta RETURN

[1] + Stopped (tty output) cat temp &

Semnale

Se definesc ca **intreruperi software** pe care un proces le primeste pentru a fi informat despre aparitia unui eveniment.

Pot proveni de la nucleu, de la alt proces sau de la utilizator.

Constituie o modalitate de tratare a unor **evenimente asincrone** si sunt un **mecanism elementar de comunicare intre procese**.

Semnalele poarta extrem de putina informatie fiind reprezentate printr-un identificator, un numar intreg pozitiv, asociat (in fisierul <signal.h>) cu constante simbolice care incep cu caracterele SIG, urmate de inca 2-6 litere pt a identifica natura semnalului.

Conditii care duc la generarea unui semnal:

- **Actionarea anumitor taste ale terminalului ;**
- **Aparitia unor exceptii hardware: referiri invalide la memorie. Acestea conduc la interventia nucleului care va genera un semnal adecvat spre procesul in executie in momentul producerii exceptiei;**
- **Apelul sistem kill** permite unui proces sa trimita un semnal spre un alt proces sau grupe de procese, asupra carora trebuie sa aibe drept de proprietate (sau sa fie root);
- **C-da kill** permite ca un utilizator sa trimita semnale spre procese asupra carora are drept de proprietate. Este de fapt o interfata ca apelul sistem kill si se foloseste pt a termina procese din fundal care au fost detectate cu comportare anormala;
- Aparitia unor conditii software care trebuie semnalate de catre nucleu procesului curent cum ar fi: incercarea de a scrie intr-un pipe dupa ce procesul care citeste din acel pipe s-a terminat (semnalul SIGPIPE), sau expirarea unui interval de timp solicitat de proces (semnalul SIGALRM).
- Terminarea unui proces cind acesta executa functia de sistem **exit**.

Transmiterea semnalelor

- De catre utilizator prin apasarea unor taste

  CTRL-C- SIGINT si CTRL-Z SIGSTP;
- Printr-o comanda de la linia de comanda

  kill -<nume_signal> <PID>

- Prin program

```
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>
pid_t my_pid = getpid ( );
kill (my_pid, SIGSTOP);
```

Actiunea pe care un proces o poate asocia unui semnal:

- ignorarea semnalului: se poate aplica majoritatii semnalelor cu exceptia **SIGKILL** si **SIGSTOP**.

- interceptarea semnalului: la aparitia semnalului nucleul apeleaza o functie definita de programator pentru tratarea semnalelor de tipul respectiv. In functie se pot introduce orice actiuni doreste programatorul: de exempl, daca o aplicatie creaza fisiere temporare se poate intercepta **SIGTERM** si sa se scrie o functie de tratare care sa stearga fisierele temporare.

- aplicarea actiunii implicite: pt fiecare semnal este definita in nucleu o **actiune implicita**, in majoritatea cazurilor terminarea sau oprirea procesului cu sau fara vidaj de memorie. Exceptie fac semnalele **SIGCHLD, SIGCONT, SIGINFO, SIGPWR, SIGURG, si SIGWINCH,** care sunt ignorate.

Pe sistemele pe care se lucreaza la laborator sunt definite un numar de 61 de semnale

1) SIGHUP	17) SIGCHLD	34) SIGRTMIN	49) SIGRTMIN+15
2) SIGINT	18) SIGCONT	35) SIGRTMIN+1	50) SIGRTMAX-14
3) SIGQUIT	19) SIGSTOP	36) SIGRTMIN+2	51) SIGRTMAX-13
4) SIGILL	20) SIGTSTP	37) SIGRTMIN+3	52) SIGRTMAX-12
5) SIGTRAP	21) SIGTTIN	38) SIGRTMIN+4	53) SIGRTMAX-11
6) SIGABRT	22) SIGTTOU	39) SIGRTMIN+5	54) SIGRTMAX-10
7) SIGBUS	23) SIGURG	40) SIGRTMIN+6	55) SIGRTMAX-9
8) SIGFPE	24) SIGXCPU	41) SIGRTMIN+7	56) SIGRTMAX-8
9) SIGKILL	25) SIGXFSZ	42) SIGRTMIN+8	57) SIGRTMAX-7
10) SIGUSR1	26) SIGVTALRM	43) SIGRTMIN+9	58) SIGRTMAX-6
11) SIGSEGV	27) SIGPROF	44) SIGRTMIN+10	59) SIGRTMAX-5
12) SIGUSR2	28) SIGWINCH	45) SIGRTMIN+11	60) SIGRTMAX-4
13) SIGPIPE	29) SIGIO	46) SIGRTMIN+12	61) SIGRTMAX-3
14) SIGALRM	30) SIGPWR	47) SIGRTMIN+13	62) SIGRTMAX-2
15) SIGTERM	31) SIGSYS	48) SIGRTMIN+14	63) SIGRTMAX-1
			64) SIGRTMAX

SIGABRT	Se genereaza la apelarea functiei abort() ; procesul este terminat anormal
SIGALARM	Se genereaza datorita expirarii ceasului de alarma al procesului
SIGBUS	Se genereaza datorita aparitiei unei erori de hardware.
SIGCLD	Se genereaza spre procesul parinte si anunta terminarea procesului fiu. Este in mod implicit ignorat.
SIGCONT	Se genereaza pentru a determinat un proces oprit sa continue
SIGEMT	Se genereaza cand componente hardware de control detecteaza o eroare datorata implementarii
SIGFPE	Se genereaza cand apar erori relative la reprezentarea in virgula flotanta (divizare prin zero)
SIGHUP	Se genereaza la deconectarea unui terminal de control (tastatura)
SIGILL	Se genereaza cand componentele hardware detecteaza o instructiune ilegala
SIGINFO	Se genereaza de catre driverul terminal la apasarea tastelor Ctrl-T si determina afisarea informatiilor de stare despre procesele care se executa.
SIGINT	Se genereaza la apasarea tastelor Ctrl-C si este primit de toate procesele asociate cu terminalul de control.
SIGIO	Se genereaza pentru a semnala un eveniment I/O asincron
SIGIOT	Se genereaza cand apare o eroare la nivel hardware
SIGKILL	Se genereaza pentru a termina orice proces – pentru cazuri de urgenta
SIGPIPE	Se genereaza spre un proces care scrie intr-un pipe (sau socket) pentru care nu exista procese cititoare.
SIGPOOL	Se genereaza la aparitia unui eveniment in timpul operatiilor I/O
SIGPROF	Se genereaza la expirarea timpului unui ceas pozitionat prin functia setitimer
SIGQUIT	Se genereaza la apasarea tastelor Ctrl-\ pentru a termina un proces.
SIGSERGV	Se genereaza cand un proces realizeaza o referire la o data din afara spatiului de adresare.
SIGSTOP	Se genereaza pentru a pune in asteptare un proces care poate eventual reporni.
SIGSYS	Se genereaza in cazul folosirii de argumente gresite la apelul functiilor sistem
SIGTERM	Se genereaza de comanda kill.
SIGTRAP	Se genereaza dupa executia fiecarei instructiuni in cazul in care programul este executat in modul trace
SIGTSTP	Se genereaza la apasarea tastelor Ctrl-Z
SIGTTIN	Se genereaza cand un proces incearca sa citeasca de la terminalul sau de control.
SIGTTOU	Se genereaza cand un proces incearca sa scrie la terminalul sau de control.
SIGURG	Se genereaza la aparitia unei urgente – receptia gresita a datelor in timpul comunicarii in retea.
SIGVALARM	Se genereaza la expirarea timpului unui ceas virtual pozitionat prin functia setitimer

SIGABRT- generat de nucleu pt apelul sistem **abort** si produce terminarea anormala a procesului destinatar.

SIGALRM – este generat la expirarea unui interval de timp anterior specificat de proces printr-un **apel alarm** sau **setitimer**.

SIGCHLD - este trimis de nucleu la terminarea unui proces catre procesul parinte. Deoarece tratarea implicita in acest caz este ignorarea, procesul parinte trebuie sa specifice ca doreste interceptarea semnalului pt a putea fi informat de schimbarile de stare ale proceselor fii. Actiunea uzuala in rutinele de tratare va fi apelul uneia din variantele de **wait** pt a **prelua PID-ul** si **starea de terminare a procesului fiu.**

SIGHUP- 1. Trimis liderului de sesiune asociat cu un terminal de control daca se detecteaza o deconectare in interfata terminalului

2. La terminarea liderului de sesiune fiind atunci trimis tuturor proceselor din prim-plan.

3. Pt a notifica procesele demon sa-si reciteasca fisierele de configurare

SIGPIPE – se genereaza la incercarea de scriere intr-o conducta daca procesul care citeste din conducta s-a terminat.

TRATAREA SEMNALELOR

Semnalizarea aparitiei semnalelor se face prin **setarea unui bit** in cimpul semnalelor existent in tabela proceselor.

Fiind considerate intreruperi, tratarea lor va urma procedura clasica de tratare a intreruperilor.

- intreruperea programului in executie;
- salvarea starii programului intrerupt;
- recunoasterea semnalului receptionat;
- executarea procedurii corespunzatoare;
- eliminarea semnalului;
- reluarea procesului intrerupt

Tratarea semnalelor se face de o functie numita **handler** care poate fi **predefinita** in sistem sau **definita de catre programator** (un fisier executabil).

Daca la executia handlerului apare un apel de functie **exec** ( ), codul procesului (identic cu cel al procesului parinte) este inlocuit cu noul executabil, semnalele urmind a fi tratate de nucleu.

Semnalele pot fi ignorate cu exceptia lui **SIGKILL** sau **tratate de catre nucleu.**

Apelul sistem signal. Semnale nefiabile

Scop: de a stabili modul de tratare al unui semnal (**Specifica functia destinata tratarii unui anumit tip de semnale - pe care procesul le intercepteaza)**

#include <signal.h>

void (*signal (int *signo*, void*(*func*) (int))) (int);

signo – numarul sau simbolul (const simbolica) semnalului caruia I se ataseaza handler-ul.

func – al-II-lea arg este adresa functiei de tratare(numele handlerului) care trebuie sa fie o functie cu un argument de tip intreg(de regula nr semnalului) si sa nu returneze valoare.

Explicarea declaratiei: - se declara un pointer * **la o functie care are ca intrare un int si returneaza un void**

- argumentele functiei signal ( ) care returneaza un pointer sunt un intreg si un alt pointer la o functie care returneaza void si are ca intrare un numar intreg int.

- numele handlerului, func este utilizat ca pointer spre handler.

Rezulta

- functia signal se apeleaza cu un intreg si cu un nume de functie (care e chiar rutina ce va trata semnalul sig).

Functia returneaza adresa rutinei anterioare de tratare a semnalului respectiv sau o constanta denumita SIG_ERR in caz de eroare.

Exista trei tipuri de actiuni care sunt executate in momentul in care a fost generat un semnal

- intr-un mod predefinit de nucleu, mod indicat prin constanta simbolica SIG_DFL. Ac trebuie plasata ca cel de-al II-lea parametru actual al functiei signal ( ). Ea e un pointer la o functie(cum se cere in declaratia lui signal) si este definita ca:

  #define SIG_DFL (void(*)()) 0

- poate fi ignorat, situatie semnalata prin constanta simbolica SIG_IGN. Aceasta trebuie definita ca:

  #define SIG_IGN (void(*)()) 1

- utilizatorul poate defini o functie care sa permita tratarea semnaluljui in modul dorit de acesta. In acest caz functiei semnal i se transmite chiar numele functiei definite de utilizator.

In caz de eroare signal returneaza const simbolica SIG_ERR definita ca

#define SIG_ERR (void(*)())-1

Exemplu programul instaleaza aceeasi functie de tratare pentru SIGUSR1 si SIGUSR2. Functia de tratare este declarata static astfel incit adresa ei e fixa.

```
#include <signal.h>
static void sigusr(int); // declararea functiei de tratare
int main(void){
    if (signal(SIGUSR1, sigusr) == SIG_ERR){     // se apeleaza fct signal pt
                                  tratarea semnalului de tip SIGUSR1-
        printf("eroare instalare handler SIGUSR1\n");
        exit(1);
    }
    if (signal(SIGUSR2, sigusr) == SIG_ERR){
        printf("eroare instalare handler SIGUSR2\n");
        exit(2);
    }
    while (1) pause( ); // bucla infinita care asteapta
```

```
static void sigusr(int sig){
    if (sig == SIGUSR1)
        printf ("Received SIGUSR1\n");
    else if (sig == SIGUSR2)
         printf ("Received SIGUSR2\n");
    else  {
        printf ("Received signal %d\n", sig); // afiseaza numarul
    semnalului
        exit(3);
    }
    return;
}
```

Testarea programului

$ sig1 & // se lanseaza programul in fundal dupa care I se trimit semnale prin c-da kill

[1] 1268 // pid-ul procesului din fundal

$ kill –USR1 1268

Received SIGUR1

$ kill –USR2 1268

Received SIGUR2

$ kill –INT 1268

$ // aici trebuie actionata tasta Return

[1]+ Interrupt sig1 // cind programul primeste semnalul SIGINT executia sa se termina pt ca actiunea implicita pt SIGINT este terminarea progr

Apeluri sistem intreruptibile. Functii reentrante

Apeluri sistem lente - cele care se pot bloca un timp nedefinit:

- citiri din fisiere care pot bloca apelantul un timp nedefinit, daca nu exista date: conducte, terminale, dispozitive decomunicare in retea;
- scrieri in aceleasi fisiere daca datele nu pot fi acceptate imediat;
- deschiderea unor fisiere unde blocarea poate apare pina cind satisfacute anumite conditii (deschiderea unui terminal la distanta, care asteapta pina la conectarea modemului);
- apeluri **pause** si **wait;**
- anumite operatii ioctl;
- unele apeluri pt comunicarea interprocese;

Se excepteaza din categoria apelurilor sistem lente toate cele care implica operatii de introducere/extragere cu discul.

Unele implementari UNIX ofera posibilitatea de reluare automata a apelurilor intrerupte. Cind un semnal este interceptat **=>** procesul paraseste executia secventei sale normale de instructiuni si trece la executia instructiunilor din functia de tratare a semnalului.

Pt rezolvarea acestui gen de probleme se specifica in UNIX **ca majoritatea apelurilor sistem sunt reentrante = pot fi apelate din nou inainte ca un apel anterior sa se termine, fara a produce perturbari ale rezultatelor.**

Nu sunt reentrante:

- apelurile care folosesc **strucruri de date statice**;
- majoritatea functiilor din biblioteca standard de introducere/extragere.

OBS: si apelurile reentrante pot returna erori si exista o singura variabila **errno** per proces.

Regula: inainte de efectuarea unui apel sistem in functiile de tratare a semnalelor trebuie salvata valoarea lui errno, iar la revenire valoarea se restaureaza.

Semnale fiabile

Terminologia folosita:

- Un semnal este **generat sau trimis** unui proces atunci cind se produce evenimentul care cauzeaza aparitia semnalului. In acel moment **nucleul pozitioneaza un fanion in tabela de procese**.
- Un semnal este **livrat** unui proces atunci cind se declanseaza actiunea asociata semnalului.

Intre momentul generarii si cel al livrarii semnalul este in asteptare (pending).

Este posibil ca un proces sa blocheze livrarea unui semnal.

Nucleul determina ce se intimpla cu un semnal blocat atunci cind semnalul este livrat, nu la generarea sa => este posibil ca dispozitia procesului pentru semnalul in cauza sa se schimbe intre momentul generarii si livrare.

Daca in timp ce un semnal este blocat se produc mai multe generari ale semnalului respectiv, este posibil ca toate aparitiile semnalului sa fie inregistrate si livrate pe rind la deblocare.

Fiecare proces are intre info pastrate in tabela de procese o masca de semnale care precizeaza tipurile de semnale momentan blocate.

S-au introdus noi apeluri sistem care sa opereze asupra acestei structuri de date.

Apeluri sistem folosite pentru a trimite un semnal catre un proces.

- Nu sunt specifice semnalelor fiabile; au existat si inainte.

Apelul **kill –** permite trimiterea unui semnal catre un proces sau grup de procese.

Apelul **raise –** se foloseste numai ca un proces sa trimita un semnal catre el insusi.

Sintaza apelurilor:

#include <sys/types.h>

#include <signal.h>

int kill(pid_t *pid,* int *signo*);

int raise (int *signo*);

Argumentul *pid* de la apelul kill este interpretat in functie de valoarea sa, astfel:

- *pid* >0 Semnalul este trimis procesului identificat prin *pid*;

- *pid*==0 Semnalul este trimis tuturor proceselor din acelasi grup de procese ca si apelantul si pentru care apelantul are drept de trimitere. Se utilizeaza pt a elimina procesele din background fara a avea nevoie de identificatorii lor;

- *pid* <0 & pid!= -1. Semnalul este trimis tuturor proceselor cu identificatorul de grup egal cu valoarea absoluta a lui *pid* si pt care apelantul are drept de trimitere;

- *pid* == -1 In acest caz comportarea este dependenta de implementare. Trimis de root este utilizat la oprirea sistemului

Superutilizatorul poate trimite semnale oricarui proces.

Pt ceilalti utilizatori regula de baza:

- UID-ul **real** sau **efectiv** al **emitatorului** trebuie sa **egal** cu **UID-u**l **real** sau efectiv al **destinatarului**.

- semnalul de continuare **SIGCONT** poate fi trimis oricarui alt proces membru al aceleiasi sesiuni.

- semnalul cu numarul 0 (semnalul nul) nu exista, dar apelul **kill** permite folosirea valorii 0 pentru *signo* cu scopul de a verifica daca un anumit proces mai exista. Daca procesul identificat prin *pid* nu mai exista, apelul returneaza -1, iar valoarea lui errno devine ESRCH.

Apelurile sistem **alarm** si **pause**

#include <sys/types.h>

unsigned int alarm(unsigned int *seconds*);

int pause (void);

alarm() – permite setarea unui timer, iar terminarea timpului setat permite transmiterea semnalului SIGALARM.

#include <unistd.h>

unsigned int alarm(unsigned int sec);

sec- reprezinta nr de secunde dupa care e generat semnalul SIGALARM de nucleu.

Apelul returneaza 0 sau nr de secunde ramase de la cererea anterioara de activare a semnalului SIGALARM. Exista un singur orologiu de alarma intr-un proces.

pause() – suspenda procesul apelant pina la primirea unui semnal.

#include <unistd.h>

int pause(void);

Daca procesul care a apelat functia pause() primeste un semnal pe care il ignora sau nu-l trateaza atunci procesul se termina.

Manipularea unui set de semnale

Fiecare proces poate masca anumite semnale cu ajutorul unei masti ce-i este specifica.

Pentru a putea manipula cu usurinta semnalele existente s-a definit un set de semnale ce vor fi blocate si care por fi manipulate impreuna.

Tipul de date ce reprezinta acest set se numeste sigset_t (set de semnale). Definitia exacta a tipului nu este esentiala in posibilitatea de a manipula acest set de semnale mascate. Au fost definite 4 operatii elementare:

- Vidarea setului – adica initializarea astfel incit nici un semnal sa nu fie prezent in set si deci nici un semnal sa nu fie blocat;
- Umplerea setului - adica initializarea setului astfel incit toate semnale sunt prezente in set si deci toate semnalele sunt blocate;
- Adaugarea unui semnal la set;
- Stergerea unui semnal din setul respectiv;

Apelurile sistem corespunzatoare acestor operatii:

int sigemptyset(sig_set *set);

int sigfillset(sigset_t *set);

int sigaddset(sigset_t *set, int semnal);

int sigdelset(sigset_t *set, int semnal);

int sigismember(sigset_t *set, int semnal);

Masca utilizata pentru mascarea semnalelor este asimilata de fapt unui set de semnale. Pentru a putea inspecta sau modifica masca atunci cind nu se cunoaste numele setului corespunzator a fost definita functia sigprocmask() dupa cum urmeaza:

#include <signal.h>

int sigprocmask(int mod, const sigset_t *set, sigset_t *oset);

oset – daca nu e NULL va contine dupa executia functiei masca anterioara executiei

set - daca nu e NULL masca se va modifica in functie de valoarea arg **mod**

mod = SIG_BLOCK – noua masca de semnale pentru proces este reuniunea dintre masca de semnale curenta si setul de semnale referit de set SIG_BLOCK adica blocheaza si acest set de semnale

mod = SIG_UNBLOCK - noua masca este intersectia dintre masca de semnale curente si complementul setului de semnale referit de set. – adica semnalele indicate de set sunt deblocate

mod = SIG_SETMASK vechea masca este inlocuita cu cea reprezentata de set.

Gestionarea detaliata a actiunii semnalelor: Functia **sigaction()**

Este versiunea ameliorata a functiei signal(), studiata anterior.

#include <signal.h>

int sigaction(int sig, const struct sigaction *act, const struct sigaction *oact);

sig – semnalul al carui efect este inspectat sau modificat

act - daca nu este NULL va determina modificarea actiunii la aparitia semnalului sig, noua actiune fiind un cimp al structurii act.

oact - daca nu este NULL, va indica structura ce reprezinta starea anterioara a sig.

```
struct sigaction {
        void (*sa_handler)(); //adresa rutinei de tratare;
        sigset_t sa_mask; // semnale suplimentare deblocat pe durata
                                executiei hadler-ului
        int_sa_flags; // optiuni pentru semnale
}
```

Obs- actiunea sigaction include oarecum actiunea signal pentru schimbarea handler-ului si sigprocmask() pentru modificarea mastii semnalelor

sa_flags poate lua diferite valori.

Obtinerea setului de semnale blocate de la livrare si aflate momentan in asteptare

Functia **sigpending()**

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

Setul de semnale dorit va fi returnat prin argumentul set, iar functia returneaza 0 in caz de succes si -1 in caz de eroare.

Suspendarea unui proces

Functia **sleep**

```
#include <signal.h>
unsigned int sleep(unsigned int sec);
```

Suspendarea poate sa dureze pina la scurgerea celor **sec** secunde precizate la apelul functiei sau pina la executia handlerului unui semnal captat si revenirea din aceea rutina.

Terminarea anormala a unui program

Pt a termina un program se utilizeaza **abort() a carei executie va genera semnalul SIGABRT spre procesul apelant. Deci practic procesul isi trimite semnalul SIGABRT.**

```
#include <signal.h>
void abort(void);
```

Pt a avea efect trebuie este necesar ca **SIGABRT sa nu fie blocat pentru procesul respectiv. In consecinta el trebuie deblocat.**