# Laborator 5 – Semnale

**Ce este un semnal ?**

Un semnal este o **intrerupere software,** in fluxul normal de executie a unui proces.

De multe ori se intampla ca un proces sa nu aiba fluxul de executie ce il asteptam atunci cand am scris programul nostru ce a constituit imaginea pentru el. Acest lucru poate sa fie cauzat atat din cauza programatorului care nu a remarcat anumite "**bug**"-uri in codul scris de el sau din cauza unor evenimente externe precum o defectiune a calculatorului.

A fost nevoie astfel de un mecanism pentru a gestiona astfel de situatii, din acest motiv au fost introduse semnalele.

Gestionarea semnalelor este facuta de catre **kernel-ul** sistemului de operare ce le foloseste pentru a notifica anumite procese atunci cand apare un eveniment in timpul executiei lor normale.

Practic, un semnal este o valoare naturala mai mare ca 0. Numarul de semnale este fix si difera de la un sistem de operare la altul. Fiecare semnal are o intrebuintare specifica astfel ca un proces trebuie de multe ori sa abordeze tratamente diferite in functie de valoarea acestuia. Cand facem referire la un semnal, nu ne vom referi la el prin valoarea sa numerica, ci printr-o constanta simbolica ce este de forma **SIGxxxx.**

**Exemplu : SIGSTOP , SIGCONT , SIGKILL , SIGINT …..etc**

Pentru o descriere detaliata a semnalelor din Linux recomand consultarea paginii de manual prin comanda:

**$ man 7 signal**

Un semnal, poate sa fie trimis unui proces fie de catre **kernel** ( lucru care se intampla cel mai frecvent ) fie de catre un utilizator prin comenzi specifice sau apeluri de sistem oferite de API-ul POSIX.

Cel mai utilizat utilitar pentru a trimite un semnal unu proces este **kill**. Ne amintim ca in sistem un proces este identificat unic prin **PID-ul** sau.

Prin lansarea comenzii:

$ **kill pid_proces**

Se trimite default semnalul **SIGTERM** procesului identificat prin **pid_proces.**

Pentru a trimite un alt semnal se da numarul semnalului sau sufixul din constanta simbolica prin care se identifica ( adica ce urmeaza dupa **"SIG"** ) ca parametru comenzii astfel:

$ **kill -nr pid_proces**

sau

**$ kill -sufix pid_proces**

Daca dorim sa aflam o lista detaliata cu toate semnalele din sistem referite atat prin numarul lor cat si prin constanta simbolica putem folosi comanda **kill** astfel:

$ **kill -l**

**Cateva semnale larg folosite:**

- **SIGINT** se primeste de catre un proces atunci cand se tasteaza combinatia **CTRL + C.** Procesul trebuie sa fie in **forground !**
- **SIGQUIT** se primeste de catre un proces atunci cand se tasteaza combinatia **CTRL + **
- **SIGSTOP** se primeste de catre un proces atunci cand se tasteaza combinatia **CTRL + Z**
- **SIGSEGV** se primeste de catre un proces atunci cand a accesat o zona de memorie ilegala.
- **SIGFPE** atunci cand s-a realizat o eroare aritmetica.
- **SIGCHLD** se primeste atunci cand fiul procesului intra in starea zombie sau este stopat.

**Exercitiu**

Folositi comenzile prezentate mai sus pentru a trimite semnalul **SIGALRM** procesului ce reprezinta **shell-ul** curent de lucru ( **bash-ul** ).

Un proces poate sa trimita un semnal altui proces numai daca procesul destinatar are acelasi propietar real cu procesul expeditor sau daca procesul expeditor il are ca propietar pe utilizatorul privilegiat **root**. Aceasta conventie reprezinta o metoda de securitate a sistemului deoarece un utilizator normal nu poate sa trimita semnale ce ar afecta buna executie a proceselor critice din sistem.

Primirea unui semnal poate avea urmatoarele efecte asupra unui proces:

- Terminarea procesului
- Terminarea procesului cu producerea unui fisier special cu numele **core** ce reprezinta o imagine a memoriei virtuale a procesului la momentul terminari sale anormale
- Nici un efect, executia procesului nu este perturbata
- Suspendarea procesului
- Trezirea procesului, in cazul in care era suspendat, altfel executia procesului nu este perturbata

Pentru fiecare proces, **kernel-ul** mentine o lista cu toate semnalele din sistem si starea in care se afla acestea impreuna cu handler-ul de tratare a fiecarui semnal in parte.

Un semnal se poate afla in urmatoarele stari relativ la un proces:

- **pending** , adica asteapta sa fie luat in considerare de catre proces
- **blocat,** adica nu va avea efect imediat asupra procesului cand ajunge la acesta, in schimb, va trece in starea **pending** pana cand procesul il deblocheaza manual.

Atunci cand un semnal este tratat, bit-ul sau de pending devine 0. Pentru mai multe semnale de acelasi tip exista un singur bit de **pending**, astfel, daca la un proces ajung mai multe semnale de un anumit tip in timp ce acest tip este blocat de catre proces, bit-ul de pending se va schimba o singura data si nu vom putea afla cate semnale de acel tip au ajuns intradevar la proces.

Deobicei nu se poate prezice ordinea in care semnalele vor ajunge la un proces sau timpul exact cand un semnal va ajunge la un proces.

**API-ul POSIX**

Declaratia tuturor functilor de care este nevoie pentru lucrul cu semnale se afla in header-ul **signal.h.**

Numarul de semnale disponibile pe o anumita platforma este referit prin consta simbolica **NSIG**. Astfel, semnalele disponibile au valoare intre 1 si **NSIG.**

Pentru a trimite un semnal catre un anumit proces se foloseste functia :

**int kill ( int pid , int sig )**

Primul parametru reprezinta pid-ul procesului catre care vrea sa se trimita semnalul iar al doilea reprezinta semnalul. Este indicat sa se foloseasca o constanta simbolica pentru referirea unui semnal si nu o valoare numerica. Pentru o descriere mai detaliata a acestei functi **RECOMAND !!!!** citirea paginii de manual cu comanda :

**$ man 2 kill**

Daca dorim ca un proces sa isi trimita singur un anumit semnal vom folosi functia:

**in raise ( int sig )**

Ambele functii intorc 0 in caz ca apelul s-a soldat cu succes si -1 in caz contrar iar **errno** este setat. Succes inseamna ca procesele destinatare au primit semnalul.

**Exercitii**

1. Scrieti un program care primeste in linia de comanda pid-urile unor procese si o valoare de semnal si va trimite semnalul catre toate procesele din lista. Se vor afisa pid-urile proceselor impreuna cu un mesaj care arata daca semnalul s-a trimis cu succes fiecarui proces in parte.

2. Scrieti un program care isi va trimite singur semnalul SIGSEGV.

**Multimi de semnale. Blocarea si deblocarea semnalelor**

In header-ul **signal.h** avem definit un tip de date numit **sigset_t.** Acest tip de date exprima o multime de semnale. Adica se respecta proprietatile de multime invatate in clasa a 5-a. Pentru a prelucra aceasta multime avem la dispozitie anumite functii.

**int sigemptyset( sigset_t * set)**
**int sigfillset( sigset_t *set )**

Ambele intorc 0 in caz de succes si -1 in caz de eroare. Prima functie atribuie lui **set** statutul de multime vida, iar a doua face din **set** o multime ce contine toate semnale posibile, adica de la 1 la **NSIG.**

**int  sigaddset( sigset_t * set, int sig )**
**int  sigdelset( sigset_t *set, int sig )**

Ambele intorc 0 in caz de succes si -1 in caz de eroare. Prima functie adauga in multimea **set** semnalul cu valoarea **sig.** A doua functie inlatura din multimea **set** semnalul cu valoarea **sig.**

**int  sigismember(const sigset_t *set, int sig )**

Intoarce 1 in caz ca semnalul **sig** apartine multimii **set** si 0 in caz contrar.

**int  sigisemptyset( const sigset_t *set )**

Intoarce 1 in caz ca multimea **set** este vida si 0 in caz contrat.

**ATENTIE !!!** Atunci cand declaram un obiect de tipul **sigset_t** nu trebuie sa plecam de la prezumtia ca exprima o multime vida de semnale, deoarece standard-ul nu specifica asta. Este important sa folosim functia **sigemptyset** pentru a fi siguri ca manipulam o multime vida.

Prin "**masca de semnale blocate**" ne referim la multimea de semnale ce un proces le are blocate. Adica primirea lor nu produce o intrerupere imediata a fluxului de executie, acestea sunt trecute in starea **pending** pana cand procesul le deblocheaza manual dupa care sunt trimise catre acesta.

Semnalele **SIGKILL, SIGSTOP** si **SIGCONT** nu pot sa fie blocate de catre un proces. Primul produce o terminare imediata a procesului, al doilea il pune in starea **STOPED** iar al treilea il trezeste din aceasta stare.

Pentru a modifica masca de semnale blocate a unui proces folosim functia:

**int sigprocmask( int how, const sigset_t *newset, sigset_t *oldset )**

Acesta functie intoarce 0 in caz de succes si -1 in caz de eroare. In **oldset** se va stoca multimea de semnale blocate de dinaintea acestui apel, al treilea parametru poate sa fie si **NULL** in cazul in care nu ne intereseaza valoare acestui parametru. Primul argument desemneaza modul in care noua multime de semnale va fi setata. Astfel, daca parametrul **how** are valoarea:

- **SIG_SETMASK** noua multime de semnale blocate devine **newset**
- **SIG_BLOCK** noua multime de semnale blocate devine multimea obtinuta prin reuniunea dintre **newset** si **oldset**
- **SIG_UNBLOCK** noua multime de semnale blocate devine multimea obtinuta prin diferenta dintre **oldset** si **newset**. Adica: **oldset \ newset !**

Pentru a afla multimea de semnale blocate si aflate in pending putem folosi functia:

**int sigpending ( sigset_t *set )**

Intoarce 0 in caz de succes si -1 in caz de eroare. Daca functia se termina cu succes atunci va pune in multimea **set** rezultatul.

**Exercitii**

**1**. Setati ca masca de semnale blocate  multimea formata din semnalele **SIGINT** si **SIGQUIT.**

Faceti in asa fel in cat programul sa intre intr-un loop infinit. Incercati sa tastati caracterele speciale ce duc la aparitia acestor semnale. Ce observati ?

**2.** Setati ca masca de semnale blocate multimea formata din semnalul **SIGSEGV.** Apelati o functie ce va produce un **stack overflow.** Ce observati ?

**Handle pentru tratarea semnalelor**

Fiecare semnal are asociat un handle de tratare al acestuia care poate produce asupra procesului unul din efectele enumerate mai sus. Pentru fiecare semnal exista un handle implicit ( sau **default** ) dar care poate sa fie suprascris daca dorim sa schimbam comportamentul procesului la primirea semnalului.

Pentru semnalele **SIGKILL, SIGSTOP** si **SIGCONT** nu se poate suprascrie handle-ul. Efectul implicit al acestor semnale a fost enuntat mai sus.

Un handle de tratare a unui semnal este defapt o functie care trebuie sa aiba urmatoarea declaratie :

**void nume_handle( int semnal )**

Aceasta va fi apelata de catre kernel atunci cand un semnal neboclat este receptionat. **semnal** este semnalul ce a cauzat lansarea acestei functii, semnalul receptionat.

Pentru a instala un nou handle de tratare a unui semnal se foloseste functia:

**typedef void (*sighandler_t)(int);**

**sighandler_t signal(int signum, sighandler_t handler);**

Primul parametru este identificatorul semnalului a carui handle vrem sa il suprascriem, al doilea parametru este un pointer catre noul handle. Functia va intoarce un pointer catre vechiul handle, acela ce era instalat inaintea apelarii acestei functii.

Exista doua constante, **SIG_DFL** si **SIG_IGN** in header-ul **signal.h**. Prima reprezinta handle-ul implicit a unui semnal, iar al doilea reprezinta un handle ce va ignora semnalul.

Aceste constante pot sa fie date ca argument functiei **signal.**

Atunci cand un semnal neblocat este receptionat sau cand un semnal aflat in pending este deblocat, se produce o **intrerupere software** in fluxul normal de executie a procesului. Rareori sunt cazurile cand putem sa prezicem locul exact unde aceasta intrerupere se va produce ( adica la ce instructiune ). Cert este ca atunci cand se produce, executia normala a procesului este intrerupta si se apeleaza handle-ul de tratare a semnalului dupa care, daca handle-ul nu intrerupe executia procesului ( prin apel la functi **exit()** sau **abort()** ), se revine la executia normala.

Apar doua intrebari pe care trebuie sa ni le punem:

1. Ce se intampla daca in timpul executiei unui handle se primeste un alt semnal neblocat ? Executia handle-ului se va intrerupe si ea pentru a lansa handle-ul pentru semnalul nou primit ? Sau semnalul se va bloca pana cand handle-ul curent isi termina exectutia ?

2. Dupa prima receptie a unui semnal a carui handle a fost suprascris, se revine la handle-ul implicit ?

Din nefericire, standard-ul POSIX nu reglementeaza acest lucru referitor la functia **signal,** comportarea unui proces la primirea unui semnal a carui handle a fost suprascris prin aceastra functie difera de la implementare la implementare.

Pentru a avea un control deplin asupra unui handle de semnal este necesar sa folosim functia:

**int sigaction ( int signal, const struct sigaction* action, struct sigaction* oldaction )**

Functia returneaza 0 in caz de succes si -1 in caz de eroare, primul argument reprezinta semnalul caruia vrem sa ii schimbam comportamentul, al doilea argument este o structura ce specifica noul comportament al semnalului iar al treilea argument este o structura ce specifica vechiul comportament al semnalului.

Structura sigaction este de forma:

```
struct sigaction
{
    void (*sa_handler)(int);  /* pointer catre noul handle
    sigset_t sa_mask;  /* multimea de semnale blocate in timpul executie handle-ului aceasta /*
                       /*  va fi adaugata la masca de semnale blocate curenta */
    int sa_flags;  /* flag-uri ce controleaza invocarea handle-ului

}
```

Campul **sa_handler** poate sa aiba si una dintre valorile **SIG_IGN** sau **SIG_DFL** dar atunci celelalte doua campuri nu mai sunt luate in considerare.

Campul **sa_flags** este construit prin disjunctie pe biti cu urmatoarele constante simbolice ( acest camp poate sa aiba si valoarea 0, dar e bine sa specificam asta manual, nu sa presupunem ) :

- **SA_NOCLDSTOP** daca argumentul **signal** din functia **sigaction** are valoarea **SIGCHLD** atunci nu se va primi acest semnal in cazul in care unul din fi intra in starea stopat.
- **SA_NOCLDWAIT** daca argumentul **signal** din functia **sigaction** are valoarea **SIGCHLD** atunci fii care se termina nu vor intra in starea zombie.
- **SA_NODEFER** semnalul curent nu se va adauga la masca de semnale blocate pe timpul executiei handle-ului.
- **SA_RESETHAND** handle-ul instalat va fi valabil doar pentru prima receptie a semnalului, handle-ul va fi resetat la **SIG_DFL** chiar inainte de executia handle-ului curent.

Parametrii **action** si **oldaction** pot sa fie NULL. Daca se doreste doar modificarea comportamentului actual fara sa ne intereseze cel vechi, **oldaction** poate sa fie NULL. Daca se doreste doar aflarea comportamentului actual **action** poate sa fie NULL.

**Exercitii**

**1.** Setati un handle pentru semnalul **SIGSEGV.** In cadrul acestui handle afisati ceva la ecran pentru a sti cand acest handle a fost apelat, dupa care apelati functia **exit()** cu un parametru de esec. Faceti

ca procesul sa faca un **segmentation fault** prin dereferentierea pointer-ului NULL.

**2**. Setati un handle pentru semnalul **SIGFPE.** In cadrul acestui handle afisati ceva la ecran pentru a sti cand acest handle a fost apelat, dupa care apelati functia **exit()** cu un parametru de esec. Faceti ca procesul sa faca o impartire prin 0.

**3**. Setati un handle pentru semnalul **SIGFPE** si unul pentru **SIGSEGV.** In cadrul acestor handle afisati ceva la ecran pentru a sti cand aceste handle au fost apelate, dupa care apelati functia **exit()** cu un parametru de esec. Faceti ca procesul sa faca o dereferentiere de pointer NULL dupa care, in handle de tratare pentru semnalul **SIGSEGV**, sa se faca o impartire prin 0. Ce observati ?

**4.** Functia **:**

**unsigned int alarm(unsigned int seconds);**

seteaza un cronometru pentru **seconds** secunde dupa care revine din cadrul apel. Dupa acest timp procesul ce a apelat aceasta functie va primi semnalul **SIGALRM.**
Scriet un program care armeaza ceasul ("alarm") pentru 2 secunde, apoi intra intr-un ciclu infinit. La primirea lui SIGALRM nu se va termina procesul ci se va iesi din ciclu (se va continua programul dupa ciclul respectiv).

**5.** Setati un handle pentru semnalul **SIGSEGV.** In cadrul acestui handle afisati ceva la ecran pentru a sti cand acest handle a fost apelat, dupa care functia va returna . Faceti ca procesul sa faca un **segmentation fault** prin dereferentierea pointer-ului NULL. Ce observati de data asta ?

**Asteptarea primirii unui semnal**

Chiar daca receptionarea unui semnal sugereaza deobicei aparitia unei situatii exceptionale in executia unui proces, sunt momente cand dorim intentionat sa suspendam procesul pana cand un semnal apare. Cand acesta va fi receptionat, se va executa handle-ul corespunzator semnalului dupa care procesul isi continua executia.

Pentru a pune un proces in starea **SLEEPING** pana la aparitia unui semnal folosim functia:

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

Aceasta functie returneaza intotdeauna valoarea -1 si seteaza **errno** la EINTR.

Procesul va fi trezit din aceasta stare de catre semnale care nu sunt blocate si care nu au setat ca handle **SIG_IGN.**

**ATENTIE !!!** Handle-ul **SIG_IGN** nu este acelasi lucru cu un handle scris de noi ce are corpul functiei gol.
**Adica :**

```
void handle ( int signal )
{
    /* corpul functiei este gol */
}
```

O functie mai puternica pentru asteptarea unui semnal este:

**int sigsuspend ( const sigset_t* mask );**

Aceasta functie returneaza intotdeauna valoarea -1 si seteaza **errno** la EINTR. Functia pune procesul apelant in starea **SLEEPING** si seteaza masca de semnale blocate a procesului la multimea indicata prin **mask.** Dupa ce un semnal ce nu este blocat ( adica nu se afla in multimea **mask** ) este primit, handle-ul asociat lui este executat dupa care **sigsuspend** reseteaza masca de semnale la cea existenta de dinainte de apel.

Practic, un apel la **sigsuspend** este echivalent cu un apel **atomic** al urmatoarelor functii:

**sigprocmask(SIG_SETMASK, &mask, &prevMask);**

**pause();**

**sigprocmask(SIG_SETMASK, &prevMask, NULL);**

**ATENTIE !!! Apel atomic !!! Adica in programele scrise de voi nu este acelasi lucru daca apelati sigsuspend sau blocul continand cele 3 apeluri. Atomicitatea operatilor o poate asigura numai kernel-ul sistemului de operare !!!**

**Exercitii**

**1.** Semnalele **SIGUSR1** si **SIGUSR2** nu sunt niciodata generate de catre kernel ! Aceste semnale sunt rezervate pentru programatorii de aplicatii. Prin default, aceste semnale duc la terminarea procesului destinatar. Scrieti doua programe ! Primul program va trimite un numar de 2000 de semnale **SIGUSR1** celui de al doilea iar al doilea va numara cate a primit, dupa ce a terminat de trimis semnalele, programul unu va trimite un semnal **SIGINT** celui de al doilea. La primirea acestui semnal, programul doi va afisa cate a primit dupa care va face **exit().**

**2.** Semnalul **SIGCHLD** se primeste de catre un proces atunci cand fiul sau intra in starea zombie, este stopat sau este continuat dupa ce s-a aflat intr-o stare stopat. Prin default, acest semnal are setat handle-ul **SIG_IGN**. Scrieti un program care citeste un vector de numere de la tastatura dupa care face **fork()** fiul va face suma numerelor din acest vector si o va returna la tata prin apel la **exit().** Tatal va trece in starea sleeping pana cand fiul se termina, dupa care va afisa la ecran suma numerelor din vector intoarsa de fiu. Atentie ! Tatal va ignora semnalele **SIGCHLD** produse de faptul ca fiu daca intra in starea stopat ( vezi **sigaction** ). In timpul asteptarii tatal va bloca celelalte semnale posibile.

**Tema**

Talk bazat doar pe semnale: un acelasi program este lansat de la 2 terminale diferite de acelasi utilizator,obtinand 2 procese diferite. Fiecare proces citeste de la tastatura PID-ul celuilalt. Fiecare proces are un tabel care asociaza cate un semnal != SIGKILL, SIGCONT, SIGSTOP celor 26 litere, blank-ului si capului de linie. Fiecare proces citeste intr-un ciclu cate o linie de la tastatura, apoi o parcurge si trimite celuilalt proces semnalul asociat fiecarui caracter din ea (inclusiv blank-urile si capul de linie). De asemenea, fiecare proces, la primirea unui asemenea semnal, va afla caracterul corespunzator si-l va scrie pe ecran. Se va asigura protectia la pierderea unor semnale si se va

asigura ca semnalele trimise de un proces sa fie primite de celalalt in aceeasi ordine (de exemplu un proces nu va emite semnalul corespunzator unui caracter decat daca a primit confirmarea ca celalalt proces a tratat semnalul pentru caracterul precedent).