Laborator 04 - Semnale

Materiale ajutătoare

* [lab04-slides.pdf](http://elf.cs.pub.ro/so/res/laboratoare/lab04-slides.pdf)
* [lab04-refcard.pdf](http://elf.cs.pub.ro/so/res/laboratoare/lab04-refcard.pdf)

Nice to read

* TLPI - Chapter 20, Signals: Fundamental Concepts
* TLPI - Chapter 21: Signals: Signal Handlers

Semnale în Linux

În lumea reală, un proces poate cunoaște o multitudine de situații neprevăzute, care-i afectează cursul normal de execuție. Dacă procesul nu le poate trata, ele sunt pasate, mai departe, sistemului de operare. Cum sistemul de operare nu poate ști dacă procesul își poate continua execuția în mod normal, fără efecte secundare nedorite, este obligat să termine procesul în mod forțat. O rezolvare a acestei probleme o reprezintă *semnalele*.

Un semnal este o **întrerupere software**, în fluxul normal de execuție a unui proces.

Semnalele sunt un concept specific sistemelor de operare UNIX. Sistemul de operare le folosește pentru a semnala procesului apariția unor situații excepționale oferindu-i procesului posibilitatea de a reacționa. Fiecare semnal este asociat cu o clasă de evenimente care pot apărea și care respectă anumite criterii.

Procesele pot trata, bloca, ignora sau lăsa sistemul de operare să efectueze acțiunea implicită la primirea unui semnal:

* De obicei acțiunea implicită este **terminarea** procesului.
* Dacă un proces dorește să **ignore** un semnal, sistemul de operare nu va mai trimite acel semnal procesului.
* Dacă un proces specifică faptul că dorește să **blocheze** un semnal, sistemul de operare nu va mai trimite semnalele de acel tip spre procesul în cauză, dar va salva numai primul semnal de acel tip, restul pierzându-se. Când procesul hotărăște că vrea să primească, din nou, semnale de acel tip, dacă există vreun semnal în așteptare, acesta va fi trimis.

Mulțimea tipurilor de semnale este finită; sistemul de operare ține, pentru fiecare proces, o **tabelă cu acțiunile**alese de acesta, pentru fiecare tip de semnal. La fiecare moment de timp aceste acțiuni sunt bine determinate. La pornirea procesului tabela de acțiuni este inițializată cu valorile implicite. Modul de tratare a semnalului nu este decis la primirea semnalului de către proces, ci se alege, în mod automat, din tabelă.

Semnalele sunt sincrone/asincrone cu fluxul de execuție al procesului care primește semnalul dacă evenimentul care cauzează trimiterea semnalului este sincron/asincron cu fluxul de execuție al procesului.

* Un eveniment este **sincron** cu fluxul de execuție al procesului dacă apare de fiecare dată la rularea programului, în același punct al fluxului de execuție. Exemple în acest sens sunt încercarea de accesare a unei locații de memorie nevalide sau nepermise, împărțire la zero etc.
* Un eveniment este **asincron** dacă nu este sincron. Exemple de evenimente asincrone: un semnal trimis de un alt proces (semnalul de terminare unui proces copil), sau o cerere de terminare externă (utilizatorul dorește să reseteze calculatorul).

Un semnal primit de un proces poate fi generat:

* fie direct de **sistemul de operare** - în cazul în care acesta raportează diferite erori;
* fie de un **proces** - care-și poate trimite și singur semnale (semnalul va trece tot prin sistemul de operare).

Dacă două semnale sunt prea apropiate în timp ele se pot confunda într-unul singur. Astfel, în mod normal, nu există niciun mecanism care să garanteze celui care trimite semnalul că acesta a ajuns la destinație.

În anumite cazuri, există nevoia de a ști, în mod sigur, că un semnal trimis a ajuns la destinație și, implicit, că procesul va răspunde la el (efectuând una din acțiunile posibile). Sistemul de operare oferă un alt mod de a trimite un semnal, prin care se **garantează** fie că semnalul a ajuns la destinație, fie că această acțiune a eșuat. Acest lucru este realizat prin crearea unei stive de semnale, de o anumită capacitate (ea trebuie să fie finită, pentru a nu produce situații de overflow). La trimiterea unui semnal, sistemul de operare verifică dacă stiva este plină. În acest caz, cererea eșuează, altfel semnalul este pus în stivă și operația se termină cu succes. Modul clasic de a trimite semnale este analog cu acesta (stiva are dimensiunea 1) cu excepția faptului că nu se oferă informații despre ajungerea la destinație a unui semnal.

Noțiunea de semnal este folosită pentru a indica alternativ fie un anumit tip de semnal, fie efectiv obiectele de acest tip.

Generarea semnalelor

În general, evenimentele care generează semnale se încadrează în trei categorii majore:

* O **eroare** indică faptul că un program a făcut o operație nepermisă și nu-și poate continua execuția. Însă, nu toate tipurile de erori generează semnale (de fapt, cele mai multe nu o fac). De exemplu, deschiderea unui fișier inexistent este o eroare, dar nu generează un semnal; în schimb, apelul de sistem open returnează -1, indicând că apelul s-a terminat cu eroare. În general, erorile asociate cu anumite biblioteci sunt raportate prin întoarcerea unei valori speciale. Erorile care generează semnale sunt cele care pot apărea oriunde în program, nu doar în apelurile din biblioteci. Ele includ împărțirea cu zero și accesarea nevalidă a memoriei.
* Un **eveniment extern** este, în general, legat de I/O și de alte procese. Exemple: apariția de noi date de intrare, expirarea unui timer, terminarea execuției unui proces copil.
* O **cerere explicită** indică utilizarea unui apel de sistem, cum ar fi kill, pentru a genera un semnal.

Semnalele pot fi generate sincron sau asincron:

* Un semnal **sincron** se raportează la o acțiune specifică din program, și este trimis (dacă nu este blocat) în timpul acelei acțiuni. Cele mai multe erori generează semnale în mod sincron. De asemenea, semnalele pot fi generate în mod sincron și prin anumite cereri explicite trimise de un proces lui însuși. Pe anumite mașini, anumite tipuri de erori hardware (de obicei, excepțiile în virgulă mobilă) nu sunt raportate complet sincron, și pot ajunge câteva instrucțiuni mai târziu.
* Semnalele **asincrone** sunt generate de evenimente necontrolabile de către procesul care le primește. Aceste semnale ajung la momente de timp impredictibile. Evenimentele externe generează semnale în mod asincron, la fel ca și cererile explicite trimise de alte procese.

Un tip de semnal dat este fie sincron, fie asincron. De exemplu, semnalele pentru erori sunt, în general, sincrone deoarece erorile generează semnale în mod sincron. Însă, orice tip de semnal poate fi generat sincron sau asincron cu o cerere explicită.

Transmiterea și primirea semnalelor

Când un semnal este generat, el intră într-o stare de așteptare (pending). În mod normal, el rămâne în această stare pentru o perioadă de timp foarte mică și apoi este trimis procesului destinație. Însă, dacă acel tip de semnal este, în momentul de față, blocat, el ar putea rămâne în starea de așteptare nedefinit, până când semnalele de acel tip sunt deblocate. O dată deblocat acel tip de semnal, el va fi trimis imediat.

Când semnalul a fost primit, fie imediat, fie cu întârziere, acțiunea specificată pentru acel semnal este executată. Pentru anumite semnale, cum ar fi SIGKILL și SIGSTOP, acțiunea este **fixată** (procesul este terminat), dar, pentru majoritatea semnalelor, programul poate alege să:

* **ignore** semnalul
* specifice o funcție de tip **handler**
* accepte **acțiunea implicită** pentru acel tip de semnal.

Programul își specifică alegerea utilizând funcții precum [signal](http://linux.die.net/man/2/signal) sau [sigaction](http://linux.die.net/man/2/sigaction). În timp ce handler-ul rulează, acel tip de semnal este în mod normal **blocat** (deblocarea se va face printr-o cerere explicită în handler-ul care tratează semnalul).

Exemplu de folosire signal

În codul de mai jos ne propunem să capturăm semnalele SIGINT și SIGUSR1 și să facem o acțiune în cazul în care le recepționăm. SIGINT e recepționat atât folosind comanda kill -SIGINT <program>, cât și prin trimiterea combinației de taste CTRL+c programului.

*#include <stdio.h>*

*#include <stdlib.h>*

*#include <signal.h>*

*#include <unistd.h>*

pid\_t child1, child2;

int child1\_pid;

void signal\_handler(int signum)

{

switch(signum) {

case SIGINT:

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("CTRL+C received in %d Exiting\n", getpid());

[exit](http://www.opengroup.org/onlinepubs/009695399/functions/exit.html)(EXIT\_SUCCESS);

case SIGUSR1:

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("SIGUSR1 received. Continuing execution\n");

}

}

int main(void)

{

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Process %d started\n", getpid());

*/\* Semnale ca SIGKILL sau SIGSTOP nu pot fi prinse \*/*

if (signal(SIGKILL, signal\_handler) == SIG\_ERR)

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("\nYou shall not catch SIGKILL\n");

if(signal(SIGINT, signal\_handler) == SIG\_ERR) {

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Unable to catch SIGINT");

[exit](http://www.opengroup.org/onlinepubs/009695399/functions/exit.html)(EXIT\_FAILURE);

}

if(signal(SIGUSR1, signal\_handler) == SIG\_ERR) {

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Unable to catch SIGUSR1");

[exit](http://www.opengroup.org/onlinepubs/009695399/functions/exit.html)(EXIT\_FAILURE);

}

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Press CTRL+C to stop us\n");

while(1) {

sleep(1);

}

return 0;

}

Observați că semnalul SIGKILL nu poate fi handle-uit (kill -9 <program> sau kill -SIGKILL <program>).

Dacă acțiunea specificată pentru un tip de semnal este să îl **ignore**, atunci orice semnal de acest tip, care este generat pentru procesul în cauză, este ignorat. Același lucru se întâmplă dacă semnalul este blocat în acel moment. Un semnal neglijat în acest mod nu va fi primit niciodată, nici dacă programul specifică ulterior o acțiune diferită pentru acel tip de semnal și apoi îl deblochează.

Dacă este primit un semnal pentru care nu s-a specificat niciun tip de acțiune, se execută **acțiunea implicită**. Fiecare tip de semnal are propria lui acțiune implicită. Pentru majoritatea semnalelor acțiunea implicită este**terminarea** procesului. Pentru anumite tipuri de procese, care reprezintă evenimente fără consecințe majore, acțiunea implicită este să nu se facă nimic.

Când un semnal forțează terminarea unui proces, părintele procesului poate determina cauza terminării examinând codul de terminare raportat de funcțiile wait și waitpid. Informațiile pe care le poate obține includ faptul că terminarea procesului a fost cauzată de un semnal, precum și tipul semnalului. Dacă un program pe care îl rulați din linia de comandă este terminat de un semnal, shell-ul afișează, de obicei, niște mesaje de eroare.

Semnalele care în mod normal reprezintă erori de program au o proprietate specială: când unul din aceste semnale termină procesul, el scrie și un fișier ***core dump*** care înregistrează starea procesului în momentul terminării. Puteți examina fișierul cu un debugger, pentru a afla ce anume a cauzat eroarea.

Dacă generați un semnal, care reprezintă o eroare de program, printr-o cerere explicită, și acesta termină procesul, fișierul este generat ca și cum semnalul ar fi fost generat de o eroare.

În cazul în care un semnal este trimis procesului, în timp ce acesta execută un apel de sistem**blocant**, procesul va suspenda apelul, va executa handler-ul de tratare a semnalului și apoi fie operația va eșua (cu errno setat pe EINTR), fie se va reporni operația. Sistemele System V se comportă ca în primul caz, cele BSD ca în cel de-al doilea.

Tipuri standard de semnale

Această secțiune prezintă numele pentru diferite tipuri standard de semnale și descrie ce fel de evenimente indică.

Fiecare nume de semnal este o **macrodefiniție** care reprezintă, de fapt, un număr întreg pozitiv (numărul pentru acel tip de semnal).

Un program nu ar trebui să facă niciodată presupuneri despre codul numeric al unui tip particular de semnal, ci, mai degrabă, să le refere, întotdeauna, prin nume. Acest lucru este din cauza faptului că un număr pentru un tip de semnal poate **varia** de la un sistem la altul, dar numele lor sunt standard. Pentru lista completă de semnale suportate de un sistem se poate rula în linia de comandă:

$ **kill** -l

1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL

5) SIGTRAP 6) SIGABRT 7) SIGBUS 8) SIGFPE

9) SIGKILL 10) SIGUSR1 11) SIGSEGV 12) SIGUSR2

13) SIGPIPE 14) SIGALRM 15) SIGTERM 17) SIGCHLD

18) SIGCONT 19) SIGSTOP 20) SIGTSTP 21) SIGTTIN

22) SIGTTOU 23) SIGURG 24) SIGXCPU 25) SIGXFSZ

26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO

30) SIGPWR 31) SIGSYS 33) SIGRTMIN 34) SIGRTMIN+1

35) SIGRTMIN+2 36) SIGRTMIN+3 37) SIGRTMIN+4 38) SIGRTMIN+5

39) SIGRTMIN+6 40) SIGRTMIN+7 41) SIGRTMIN+8 42) SIGRTMIN+9

43) SIGRTMIN+10 44) SIGRTMIN+11 45) SIGRTMIN+12 46) SIGRTMIN+13

47) SIGRTMIN+14 48) SIGRTMIN+15 49) SIGRTMAX-15 50) SIGRTMAX-14

51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10

55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7 58) SIGRTMAX-6

59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2

63) SIGRTMAX-1 64) SIGRTMAX

Numele de semnale sunt definite în header-ul signal.h. În general, semnalele au roluri predefinite, dar acestea pot fi suprascrise de programator.

Cele mai cunoscute sunt următoarele semnale:

* SIGINT - transmis la apăsarea combinației CTRL+C;
* SIGQUIT - transmis la apăsarea combinației de taste CTRL+\;
* SIGSEGV - transmis în momentul accesării unei locații nevalide de memorie, etc;
* SIGKILL - **nu** poate fi ignorat sau suprascris. Transmiterea acestui semnal are ca efect terminarea procesului, indiferent de context.

Mesaje pentru descrierea semnalelor

Cel mai bun mod de a afișa un mesaj de descriere a unui semnal este utilizarea funcțiilor [strsignal](http://www.kernel.org/doc/man-pages/online/pages/man3/strsignal.3.html) și[psignal](http://www.kernel.org/doc/man-pages/online/pages/man3/psignal.3.html). Aceste funcții folosesc un număr de semnal pentru a specifica tipul de semnal care trebuie descris. Mai jos este prezentat un exemplu de folosire a acestor funcții:

[msg\_signal.c](http://ocw.cs.pub.ro/courses/_export/code/so/laboratoare/laborator-04?codeblock=2)

*#include <stdio.h>*

*#include <stdlib.h>*

*#define \_\_USE\_GNU*

*#include <string.h>*

*#include <signal.h>*

int main(void) {

char \*sig\_p = strsignal(SIGKILL);

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("signal %d is %s\n", SIGKILL, sig\_p);

psignal(SIGKILL, "death and decay");

return 0;

}

Pentru compilare și rulare secvența este:

so@spook$ **gcc** -Wall -g -o msg\_signal msg\_signal.c

so@spook$ ./msg\_signal

signal 9 is Killed

death and decay: Killed

Măști de semnale. Blocarea semnalelor

Pentru a putea efectua operații de blocare/deblocare semnale avem nevoie să știm, la fiecare pas din fluxul de execuție, starea fiecărui semnal. Sistemul de operare are, de asemenea, nevoie de același lucru pentru a putea lua o decizie asupra unui semnal care trebuie trimis unui proces (el are nevoie de acest gen de informație pentru fiecare proces în parte). În acest scop se folosește o mască de semnale proprie fiecărui proces.

O **mască de semnale** are fiecare bit asociat unui tip de semnal.

Masca de biți este folosită de mai multe funcții, printre care și funcția [sigprocmask](http://www.kernel.org/doc/man-pages/online/pages/man2/sigprocmask.2.html), folosită pentru schimbarea măștii de semnale a procesului curent.

int sigprocmask(int how, const sigset\_t \*set, sigset\_t \*oldset);

Tipul de date folosit de sistemele UNIX pentru a reprezenta măștile de semnale este sigset\_t. Variabilele de acest tip sunt *neinițializate*. Operațiile pe acest tip de date sunt:

* de inițializare cu biți de 0;
* de inițializare cu biți de 1;
* de blocare a unui semnal;
* de deblocare a unui semnal;
* de detectare a blocării unui semnal.

Funcțiile următoare sunt folosite pentru a manipula masca de biți. Ele nu decid acțiunea de blocare sau deblocare a unui semnal, ci doar setează semnalul respectiv în masca de biți (pentru adăugare se pune bitul corespunzător semnalului pe 1, iar pentru ștergere pe 0), pentru ca apoi să se folosească sigprocmask pentru a seta acțiunea de blocare/deblocare efectivă. Mai multe detalii despre aceste funcții găsiți [aici](http://linux.die.net/man/3/sigemptyset).

int sigemptyset(sigset\_t \*set);

int sigfillset(sigset\_t \*set);

int sigaddset(sigset\_t \*set, int signo);

int sigdelset(sigset\_t \*set, int signo);

int sigismember(sigset\_t \*set, int signo);

Înainte de a folosi funcțiile sigaddset, sigdelset și sigismember asupra unui sigset\_t, acest tip trebuie inițializat folosind sigemptyset sau sigfillset. Comportamentul este nedefinit în caz contrar.

Secvența de mai jos constituie un caz de utilizare a funcțiilor de lucru cu masca de semnale, în care, la fiecare 5 secunde, se blochează/deblochează semnalul SIGINT:

sigset\_t set;

sigemptyset(&set);

sigaddset(&set, SIGINT);

while (1) {

sleep(5);

sigprocmask(SIG\_BLOCK, &set, NULL);

sleep(5);

sigprocmask(SIG\_UNBLOCK, &set, NULL);

}

O altă valoare pe care o poate lua primul parametru al funcției [sigprocmask](http://www.kernel.org/doc/man-pages/online/pages/man2/sigprocmask.2.html) este SIG\_SETMASK, care specifică pur și simplu că vechea mască (al treilea parametru) e înlocuită cu cel de-al doilea parametru (noua mască). Un exemplu de folosire a acesteia puteți găsi la [această adresă](https://support.sas.com/documentation/onlinedoc/sasc/doc750/html/lr1/zlocking.htm).

Tratarea semnalelor

Tratarea semnalelor se realizează prin asocierea unei funcții (**handler**) unui semnal. Funcția va fi apelată în momentul în care procesul recepționează semnalul respectiv.

În mod tradițional, funcția folosită pentru asocierea de handler-e pentru tratarea unui semnal era [signal](http://www.kernel.org/doc/man-pages/online/pages/man2/signal.2.html). Pentru a preîntâmpina deficiențele acestei funcții, standardul POSIX a definit funcția [sigaction](http://www.kernel.org/doc/man-pages/online/pages/man2/sigaction.2.html) pentru asocierea unui handler cu un semnal. sigaction oferă mai mult control, cu prețul unui grad de complexitate mai mare.

int sigaction(int signum, const struct sigaction \*act, struct sigaction \*oldact);

Componenta importantă a funcției sigaction este structura cu același nume, descrisă în pagina de manual a funcției:

struct sigaction {

void (\*sa\_handler)(int);

void (\*sa\_sigaction)(int, siginfo\_t \*, void \*);

sigset\_t sa\_mask;

int sa\_flags;

};

Dacă în câmpul sa\_flags se precizează flag-ul SA\_SIGINFO, handler-ul folosit este cel specificat desa\_sigaction. Altfel, handler-ul folosit este sa\_handler. Masca de semnale care ar trebui blocate în timpul execuției handler-ului este reprezentată de sa\_mask.

Un exemplu de asociere a unui handler de tratare a unui semnal este prezentat mai jos:

*#include <signal.h>*

...

*/\* SIGUSR2 handler \*/*

static void usr2\_handler(int signum) {

*/\* actions that should be taken when the signal signum is received \*/*

...

}

int main(void) {

struct sigaction sa;

[memset](http://www.opengroup.org/onlinepubs/009695399/functions/memset.html)(&sa, 0, sizeof(sa));

sa.sa\_flags = SA\_RESETHAND; */\* restore handler to previous state \*/*

sa.sa\_handler = usr2\_handler;

sigaction(SIGUSR2, &sa, NULL);

return 0;

}

Se poate opta pentru configurarea unui handler propriu sau se poate folosi unul predefinit. Se poate folosi SIG\_IGN pentru ignorarea semnalului sau SIG\_DFL pentru rularea acțiunii implicite (terminarea procesului, ignorarea semnalului etc).

Structura siginfo\_t

Detalii despre structura siginfo\_t

Dacă flag-ul SA\_SIGINFO este setat, se folosește câmpul sa\_sigaction al structurii sigaction pentru a specifica handler-ul asociat semnalului. Handler-ul folosit primește în acest caz trei parametri și poate fi folosit pentru a transmite o informație utilă, o dată cu procesul. Al treilea argument (de tipul void\*) este rar utilizat. Al doilea argument, de tipul siginfo\_t definește o structură ce conține informații utile despre contextul apariției semnalului și alte informații pe care le poate furniza programatorul. Definiția structurii se găsește în pagina de manual a funcției [sigaction](http://www.kernel.org/doc/man-pages/online/pages/man2/sigaction.2.html).

siginfo\_t {

int si\_signo; */\* Signal number \*/*

int si\_errno; */\* An errno value \*/*

int si\_code; */\* Signal code \*/*

int si\_trapno; */\* Trap number that caused*

*hardware-generated signal*

*(unused on most architectures) \*/*

pid\_t si\_pid; */\* Sending process ID \*/*

uid\_t si\_uid; */\* Real user ID of sending process \*/*

int si\_status; */\* Exit value or signal \*/*

clock\_t si\_utime; */\* User time consumed \*/*

clock\_t si\_stime; */\* System time consumed \*/*

sigval\_t si\_value; */\* Signal value \*/*

int si\_int; */\* POSIX.1b signal \*/*

void \*si\_ptr; */\* POSIX.1b signal \*/*

int si\_overrun; */\* Timer overrun count; POSIX.1b timers \*/*

int si\_timerid; */\* Timer ID; POSIX.1b timers \*/*

void \*si\_addr; */\* Memory location which caused fault \*/*

long si\_band; */\* Band event (was int in*

*glibc 2.3.2 and earlier) \*/*

int si\_fd; */\* File descriptor \*/*

short si\_addr\_lsb; */\* Least significant bit of address*

*(since kernel 2.6.32) \*/*

}

Membrii structurii sunt inițializați numai atunci când valorile lor sunt utile. Membrii si\_signo, si\_errno șisi\_code sunt întotdeauna definiți pentru toate semnalele. Restul structurii poate fi o uniune, așa că ar trebui citite numai câmpurile care au sens pentru semnalul primit. Spre exemplu, apelul de sistem kill, semnalele POSIX.1b și SIGCHLD completează si\_pid și si\_uid, iar SIGILL, SIGFPE, SIGSEGV și SIGBUScompletează si\_addr cu adresa care a provocat eroarea.

Semnalarea proceselor

Pentru transmiterea unui semnal, se poate folosi funcția [kill](http://www.kernel.org/doc/man-pages/online/pages/man2/kill.2.html) sau funcția [sigqueue](http://www.kernel.org/doc/man-pages/online/pages/man2/sigqueue.2.html). Funcția [kill](http://www.kernel.org/doc/man-pages/online/pages/man2/kill.2.html) are dezavantajul că **nu** garantează recepționarea semnalului de procesul destinație. Dacă este nevoie să se trimită un semnal unui proces și să se știe sigur că a ajuns se recomandă folosirea funcției [sigqueue](http://www.kernel.org/doc/man-pages/online/pages/man2/sigqueue.2.html):

int sigqueue(pid\_t pid, int signo, const union sigval value);

Funcția trimite semnalul signo, cu parametrii specificați de value, procesului cu identificatorul pid. Dacă semnalul este zero, se fac verificări pentru cazurile de eroare posibile, dar nu se trimite niciun semnal. Semnalul nul poate fi folosit pentru a verifica faptul că pid-ul este valid.

Valoarea ce poate fi trimisă odată cu semnalul este un union:

union sigval {

int sival\_int;

void \*sival\_ptr;

};

Un parametru trimis astfel apare în câmpul si\_value al structurii siginfo\_t, primite de handler-ul de semnal. În mod evident, **nu** are sens transmiterea de pointeri dintr-un proces în altul.

Condițiile cerute pentru ca un proces să aibă permisiunea de a trimite un semnal altui proces sunt aceleași ca și în cazul lui kill. Dacă semnalul specificat este blocat în acel moment, funcția va ieși imediat și dacă flagulSA\_SIGINFO este setat și există resurse necesare, semnalul va fi pus în coadă în starea pending (un proces poate avea în coadă maxim SIGQUEUE\_MAX semnale). De asemenea, când semnalul este primit, câmpul si\_code, pasat structurii siginfo, va fi setat la SI\_QUEUE, și si\_value va fi setat la value.

Dacă flagul SA\_SIGINFO nu este setat, atunci signo, dar nu în mod necesar și value, vor fi trimise, cel puțin o dată, procesului care trebuie să primească semnalul.

Așteptarea unui semnal

Abordări pentru așteptarea unui semnal

În cazul în care se utilizează semnalele pentru comunicare și/sau sincronizare, există, deseori, nevoie să se aștepte ca un anumit tip de semnal să-i sosească procesului în cauză. Un mod simplu de a realiza acest lucru este o buclă, a cărei condiție de ieșire ar fi setarea corespunzătoare a unei variabile (variabila trebuie să fie de tipul sig\_atomic\_t). De exemplu:

while (!signal\_has\_arrived);

Principalul dezavantaj al abordării de mai sus (de tip *busy-waiting*) este timpul de procesor pe care procesul considerat îl pierde în mod inutil. O variantă ar fi folosirea funcției [sleep](http://www.kernel.org/doc/man-pages/online/pages/man3/sleep.3.html):

while (!signal\_has\_arrived) {

sleep(1);

}

O astfel de abordare nu ar mai ocupa timp inutil de procesor, dar timpul de răspuns în cazul sosirii unui semnal este destul de mare. O altă soluție a problemei este funcția [pause](http://www.kernel.org/doc/man-pages/online/pages/man2/pause.2.html) (care blochează fluxul de execuție până când procesul curent este întrerupt de un semnal). Deși această abordare pare foarte simplă, ea introduce, adeseori, deadlock-uri, care blochează programul nedefinit. Un exemplu în acest sens este pseudosoluția de mai jos, la problema așteptării unui semnal:

while (!signal\_has\_arrived) {

pause();

}

Bucla este necesară pentru prevenirea situației în care procesul este întrerupt de alte semnale decât cel așteptat. Se poate întâmpla ca semnalul să ajungă **după** testarea variabilei și **înainte** de apelul funcției pause. În acest caz, procesul se blochează și, dacă nu apare un alt semnal care să cauzeze ieșirea din[pause](http://www.kernel.org/doc/man-pages/online/pages/man2/pause.2.html), el va rămâne blocat nedefinit.

Soluția cea mai bună pentru a aștepta un semnal se poate realiza prin utilizarea funcției [sigsuspend](http://www.kernel.org/doc/man-pages/online/pages/man2/sigsuspend.2.html):

int sigsuspend(const sigset\_t \*set);

Funcția înlocuiește masca de semnale blocate a procesului, cu set, și suspendă procesul până când este primit un semnal care nu este blocat de noua mască. La ieșire, funcția restaurează vechea mască de semnale.

În secvența de mai jos, funcția [sigsuspend](http://www.kernel.org/doc/man-pages/online/pages/man2/sigsuspend.2.html) este folosită pentru a întrerupe procesul curent până la recepționarea semnalului SIGINT. Semnalele SIGKILL și SIGSTOP, deși prezente în masca de semnale, nu vor fi blocate:

sigset\_t set;

*/\* block all signals except SIGINT \*/*

sigfillset(&set);

sigdelset(&set, SIGINT);

*/\* wait for SIGINT \*/*

sigsuspend(&set);

Considerente privind utilizarea unui handler de semnal

Detalii despre utilizarea unui handler de semnal

Un obiect de tip semnal este atașat unui obiect de tip proces. Dacă procesul nu rulează în acel moment, sistemul de operare poate atașa unui proces numai un singur semnal care va rămâne în starea pending. Dacă procesul rula în acel moment, semnalul primit este deservit imediat și va întrerupe fluxul normal de execuție, permițându-se primirea, fără pierdere, a unui semnal de același tip.

Pentru că numărul de semnale de un anumit tip care poate fi primit de un proces într-un anumit timp este limitat și pentru a evita pierderea de semnale, un handler trebuie să se execute cât mai **repede**.

Fluxul de execuție al unui proces este văzut de către sistemul de operare ca o înșiruire de instrucțiuni pe care platforma le suportă. Unele operații din limbajele de programare de nivel înalt **nu** sunt atomice și indivizibile, fiind nevoie de mai multe instrucțiuni în cod mașină pentru a se efectua respectiva operație. Un exemplu simplu este atribuirea între variabile:

a = b;

Majoritatea platformelor actuale nu permit instrucțiuni în care ambii operanzi să fie în memorie. Pe astfel de platforme, o implementare standard pentru această operație ar fi încărcarea valorii lui b într-un registru, după care ar urma încărcarea la adresa lui a a valorii salvate în registru :

load registru\_1, b

store a, registru\_1

De aceea, este nevoie de atenție suplimentară atunci când un semnal folosește variabile care nu sunt locale funcției, deoarece semnalele pot întrerupe fluxul de execuție în orice punct al său, lăsând astfel unele variabile într-o stare **inconsistentă**. Pentru a fi siguri că o variabilă nu are valori inconsistente se recomandă folosirea tipului sig\_atomic\_t pentru variabilele din fluxul de execuție care interacționează cu handlerele de semnale. Acest tip este unul din tipurile întregi disponibile, putând varia de la o platformă la alta. Așadar operațiile ce se pot efectua cu acest tip, sunt aceleași cu cele ale unui întreg.

Timere în Linux

În Linux, folosirea timer-elor este legată de folosirea semnalelor. Acest lucru se întâmplă întrucât cea mai mare parte a funcțiilor de tip timer folosesc semnale.

Un **timer** este, de obicei, un întreg a cărui valoare este decrementată în timp. În momentul în care întregul ajunge la 0, timer-ul expiră. În Linux, expirarea timer-ului are drept rezultat, în general, transmiterea unui semnal. Definirea unui “timer handler” (rutină apelată în momentul expirării timer-ului) este, astfel, echivalentă cu definirea unui handler pentru semnalul asociat.

Înregistrarea unui timer, în Linux, înseamnă specificarea unui interval după care un timer expiră și configurarea handler-ului care va rula. Configurarea handler-ului se poate realiza atât prin intermediul funcțieisigaction (în momentul in care timer-ul expiră se generază un semnal, care la rândul lui generează rularea handler-ului asociat), sau direct prin intermediul parametrilor funcției [timer\_create](http://www.kernel.org/doc/man-pages/online/pages/man2/timer_create.2.html).

Utilizarea unui timer presupune mai mulți pași:

* **crearea unui timer** - folosind funcția [timer\_create](http://www.kernel.org/doc/man-pages/online/pages/man2/timer_create.2.html):

int timer\_create(clockid\_t clockid, struct sigevent \*evp, timer\_t \*timerid)

Timer-ul creat se identifică prin timerid. Prin intermediul structurii sigevent se setează modul în care va interacționa timer-ul cu procesul/thread-ul care l-a lansat. Exemplu de folosire:

timer\_t timerid;

struct sigevent sev;

sev.sigev\_notify = SIGEV\_SIGNAL; */\* notification method \*/*

sev.sigev\_signo = SIGRTMIN; */\* Timer expiration signal \*/*

sev.sigev\_value.sival\_ptr = &timerid;

timer\_create(CLOCK\_REALTIME, &sev, &timerid);

Prin intermediul primului argument se poate măsura timpul real al sistemului, timpul de rulare al procesului sau timpul de rulare al procesului în user-space și kernel-space. La timeout timer-ul va livra semnalul salvat în sev.sigev\_signo

* **armarea unui timer** - folosind funcția [timer\_settime](http://www.kernel.org/doc/man-pages/online/pages/man2/timer_settime.2.html):

int timer\_settime(timer\_t timerid, int flags,

const struct itimerspec \*new\_value,

struct itimerspec \* old\_value);

Armarea timer-ului presupune completarea structurii itimerspec în care specifică timpul de pornire al timer-ului, cât și intervalul de expirare al timeout-ului (intervalele sunt măsurate în secunde și nanosecunde). Exemplu de folosire:

its.it\_value.tv\_sec = freq\_nanosecs / 1000000000; */\* Initial expiration in secs\*/*

its.it\_value.tv\_nsec = freq\_nanosecs % 1000000000;*/\* Initial expiration in nsecs\*/*

its.it\_interval.tv\_sec = its.it\_value.tv\_sec; */\* Timer interval in secs \*/*

its.it\_interval.tv\_nsec = its.it\_value.tv\_nsec; */\* Timer interval in nanosecs \*/*

timer\_settime(timerid, 0, &its, NULL);

* **ștergerea unui timer** - folosind funcția [timer\_delete](http://www.kernel.org/doc/man-pages/online/pages/man2/timer_delete.2.html)

int timer\_delete(timer\_t timerid);

Pentru a folosi funcțiile de mai sus programul trebuie compilat cu -lrt

Una dintre formele de utilizare a timer-elor este implementarea funcțiilor de așteptare de tipul [sleep](http://www.kernel.org/doc/man-pages/online/pages/man3/sleep.3.html) sau[nanosleep](http://www.kernel.org/doc/man-pages/online/pages/man2/nanosleep.2.html). Avantajul folosirii funcției [sleep](http://www.kernel.org/doc/man-pages/online/pages/man3/sleep.3.html) este simplitatea. Dezavantajele sunt rezoluția scăzută (secunde) și posibila interacțiune cu semnale (în special SIGALRM). [nanosleep](http://www.kernel.org/doc/man-pages/online/pages/man2/nanosleep.2.html) are un apel mai complex, dar oferă rezoluție până la ordinul nanosecundelor și este *signal-safe* (nu interacționează cu semnale).

Timere sub Windows

În Windows există mai multe mecanisme de notificare a proceselor: [evenimente](http://msdn.microsoft.com/en-us/library/ms682655(VS.85).aspx), [APC-uri](http://msdn.microsoft.com/en-us/library/ms681951(VS.85).aspx), [evenimente de consolă](http://msdn.microsoft.com/en-us/library/ms682073%28VS.85%29.aspx), [timere](http://msdn.microsoft.com/en-us/library/ms687012(VS.85).aspx). **Evenimentele** sunt folosite pentru sincronizarea între thread-uri/procese. **APC**-urile sunt mecanisme de execuție asincronă în contextul unui proces/thread. Pot fi folosite pentru rularea unei secvențe de cod în combinație cu un timer.

Waitable Timer Objects

Un obiect de tipul **waitable timer** este un obiect de sincronizare a cărui stare este semnalizată (***signaled***) atunci când timpul specificat se scurge. Există două tipuri de obiecte waitable timer ce pot fi create:

* timer cu **resetare manuală**: un timer a cărui stare rămâne semnalizată până când un nou apel al funcției [SetWaitableTimer](http://msdn.microsoft.com/en-us/library/ms686289(VS.85).aspx) setează un nou timp de așteptare;
* timer cu **resetare automată**: un timer a cărui stare rămâne *signaled* până când un thread efectuează o operație de așteptare pe acel obiect.

Oricare dintre cele două tipuri de timer poate fi configurat ca un timer periodic. Un timer periodic este reactivat de fiecare dată când perioada de timp specificată expiră, până când timer-ul este setat din nou sau anulat. Operațiile care se pot efectua cu un timer sunt:

* **crearea** unui timer - se realizează prin intermediul funcției [CreateWaitableTimer](http://msdn.microsoft.com/en-us/library/ms682492(VS.85).aspx)
* **setarea** unui timer - se poate face cu funcția [SetWaitableTimer](http://msdn.microsoft.com/en-us/library/ms686289(VS.85).aspx)
* **anularea** - se realizează cu ajutorul funcției [CancelWaitableTimer](http://msdn.microsoft.com/en-us/library/ms681985.aspx)
* **așteptarea** - folosind funcția [WaitForSingleObject](http://msdn.microsoft.com/en-us/library/ms687032.aspx)

În secvența de cod de mai jos, se folosește un timer pentru afișarea unui mesaj după 5 secunde:

*#define \_WIN32\_WINNT 0x0500*

*#include <windows.h>*

...

int main(void) {

HANDLE timerHandle;

LARGE\_INTEGER dueTime;

timerHandle = CreateWaitableTimer(NULL, FALSE, NULL);

if (timerHandle == NULL) {

[fprintf](http://www.opengroup.org/onlinepubs/009695399/functions/fprintf.html)(stderr, "CreateWaitableTimer failed (%d)\n", GetLastError());

[exit](http://www.opengroup.org/onlinepubs/009695399/functions/exit.html)(-1);

}

*/\* configure to expire in 5 seconds; base unit is 100ns \*/*

dueTime.QuadPart = -50000000LL;

if (SetWaitableTimer(timerHandle, &dueTime, 0, NULL, NULL, 0) == FALSE) {

[fprintf](http://www.opengroup.org/onlinepubs/009695399/functions/fprintf.html)(stderr, "SetWaitableTimer failed (%d)\n", GetLastError());

[exit](http://www.opengroup.org/onlinepubs/009695399/functions/exit.html)(-1);

}

if (WaitForSingleObject(timerHandle, INFINITE) != WAIT\_OBJECT\_0) {

[fprintf](http://www.opengroup.org/onlinepubs/009695399/functions/fprintf.html)("WaitForSingleObject failed (%d)\n", GetLastError());

[exit](http://www.opengroup.org/onlinepubs/009695399/functions/exit.html)(-1);

}

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("5 seconds timer expired\n");

return 0;

}

Exerciții

În rezolvarea laboratorului folosiți arhiva de sarcini [lab04-tasks.zip](http://elf.cs.pub.ro/so/res/laboratoare/lab04-tasks.zip).

Pentru a vă ajuta la implementarea exercițiilor din laborator, în directorul utils din arhivă există un fișier utils.h cu funcții utile.

Exercițiul -1 - GSOC (0p)

Google Summer of Code este un program de vară în care studenții (indiferent de anul de studiu) sunt implicați în proiecte Open Source pentru a își dezvolta skill-urile de programare, fiind răsplătiți cu o bursă în valoare totală de 5500$ [gsoc](https://developers.google.com/open-source/gsoc).

UPB se află în top ca număr de studenți acceptați; în fiecare an fiind undeva la aprox. 30-40 de studenți acceptați. Vă încurajăm să aplicați! Există și un grup de fb cu foști participanți unde puteti să îi contactați pentru sfaturi [facebook page](https://www.facebook.com/groups/240794072931431/)

Exercițiul 0 - Joc interactiv (2p)

* Detalii desfășurare [joc](http://ocw.cs.pub.ro/courses/so/meta/notare#joc_interactiv).

Linux (7p)

La folosirea sigaction, veți inițializa, în general, câmpul sa\_flags al structurii struct sigaction la0.

Exercițiul 1 - hitme (1p)

Intrați în directorul 1-hitme/ și analizați conținutul fișierului hitme.c. Compilați și rulați programul.

Folosiți comanda kill -l pentru a lista toate semnalele disponibile. Ce valoare are semnalul SIGKILL? Într-o altă consolă trimiteți programului hitme semnale cu valori cuprinse între 20 și 25 astfel:

**kill** -20 $(**pidof** hitme)

**kill** -21 $(**pidof** hitme)

**kill** -22 $(**pidof** hitme)

**kill** -23 $(**pidof** hitme)

**kill** -24 $(**pidof** hitme)

**kill** -25 $(**pidof** hitme)

Încercați să trimiteți același semnal de două ori și explicați comportamentul. Ce ar trebui schimbat ca să se poată trimite același semnal de două ori?

* Analizați cu atenție ce este setat pe signals.sa\_flags.
* Puteți reveni la secțiunea [Tratarea semnalelor](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-04#tratarea_semnalelor) sau investigați structura [sigaction](http://www.kernel.org/doc/man-pages/online/pages/man2/sigaction.2.html)

Observați că eliminarea flagului SA\_RESETHAND nu mai reface handlerul la valoarea implicită după primirea primului semnal.

Exercițiul 2 - Normal signals vs Real-Time signals (1p)

Intrați în directorul 2-signals și urmăriți conținutul fișierului signals.c. Programul numără de câte ori se apelează handlerul de semnal în cazul trimiterii semnalelor SIGINT și SIGRTMIN

Porniți într-o consolă programul signals:

./signals

Pentru cazul semnalelor normale, într-o altă consolă rulați scriptul send\_normal.sh:

./send\_normal.sh

Pentru semnalele real-time, într-o altă consolă rulați scriptul send\_rt.sh:

./send\_rt.sh

Pentru a închide executabilul signals este trimis semnalul SIGQUIT. De unde apare diferența? Citiți din pagina de manual man 7 signal secțiunea “Real-time signals” și revedeți secțiunea [Tipuri standard de semnale](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-04#tipuri_standard_de_semnale).

Diferența între numărul semnalelor primite se datorează faptului că semnalele cu indecșii întreSIGRTMIN și SIGRTMAX sunt semnale real time, prin urmare se garantează că ele ajung la destinație. Vezi [link](http://www.linuxprogrammingblog.com/all-about-linux-signals?page=9).

Exercițiul 3 - askexit (1p)

Intrați în directorul 3-askexit și urmăriți codul sursă. Programul face busy waiting, afișând la consolă numere consecutive.

Trebuie să completați programul pentru a intercepta semnalele generate de CTRL+\, CTRL+C și SIGUSR1 (folosiți comanda kill). Handler-ul asociat cu fiecare din semnale va fi ask\_handler. Pentru fiecare semnal primit, utilizatorul va fi întrebat dacă dorește să încheie execuția sau nu.

Testați funcționalitatea programului.

Observați că folosirea funcțiilor printf, scanf în handlere de semnale poate fi problematică deoarece aceste funcții nu sunt signal-safe.

* Consultați secțiunea [Tratarea semnalelor](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-04#tratarea_semnalelor).

Exercițiul 4 - nohup (2p)

Intrați în directorul 4-nohup și realizați un program, denumit mynohup, care simulează comanda [nohup](http://linux.die.net/man/1/nohup). Programul primește, ca prim parametru, numele unei comenzi de executat. Restul parametrilor reprezintă argumentele cu care trebuie invocată comanda respectivă; lista de argumente poate fi nulă.

Programul executat de mynohup trebuie să nu fie înștiințat de închiderea terminalului la care era conectat. Va trebui să ignorați semnalul SIGHUP, livrat de shell procesului, în momentul încheierii sesiunii curente.

* Revedeți secțiunea despre [Tratarea semnalelor](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-04#tratarea_semnalelor).

Dacă fișierul standard de ieșire era legat la un terminal acesta trebuie redirectat într-un fișier definit prin macro-ul NOHUP\_OUT\_FILE.

* Folosiți apelul [isatty](http://www.kernel.org/doc/man-pages/online/pages/man3/isatty.3.html).

Pentru testare, rulați

./mynohup **sleep** 120 &

După rulare închideți sesiunea de shell curentă: fie trimițând un semnal SIGHUP, fie folosind iconița 'X' din partea dreaptă a ferestrei.

Dintr-o altă consolă rulați respectiv

**ps** -ef | **grep** **sleep**

Cine este noul părinte al procesului?

* Consultați secțiunea [Tratarea semnalelor](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-04#tratarea_semnalelor) și secțiunile [Înlocuirea imaginii unui proces](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-03#inlocuirea_imaginii_unui_proces) și [Redirectări](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-02#redirectari) din laboratoarele precedente.

Folosirea comenzii exit, fie combinația de taste Ctrl-d nu va trimite un semnal SIGHUP procesului de sleep; puteți testa utilizând sleep 120 &, inchideți shell-ul curent utilizând una din cele 2 metode, iar după verificați că procesul înca rulează.

Exercițiul 5 - zombie (2p)

Intrați în directorul 5-zombie și urmăriți conținutul fișierelor mkzombie.c și nozombie.c. Fiecare program va crea câte un proces copil nou, care doar va apela exit.

Implementați mkzombie fără a aștepta copilul creat să se termine. Procesul părinte va aștepta TIMEOUT secunde și va ieși (urmăriți *TODO*-urile).

Din altă consolă rulați:

**ps** -eF | **grep** zombie

Observați faptul că procesul copil, deși nu mai rulează, apare în lista de procese ca <defunct> și are un pid (unic în sistem la acel moment). De asemenea, observați că, după moartea procesului părinte, dispare și procesul zombie.

Implementați nozombie fără a folosi funcțiile de așteptare de tipul wait, astfel încât procesul copil să nu treacă în starea de zombie. nozombie va aștepta TIMEOUT secunde și va ieși. Folosiți semnalul SIGCHLD (informații găsiți în [sigaction(2)](http://linux.die.net/man/2/sigaction) și [wait(2)](http://linux.die.net/man/3/wait)). Consultați, de asemenea, secțiunile [Tratarea semnalelor](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-04#tratarea_semnalelor) și [Crearea unui proces](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-03#crearea_unui_proces).

**Pe scurt**: dacă părintele ignoră în mod explicit semnalul SIGCHLD prin setarea handler-ului la SIG\_IGN (în loc să ignore semnalul în mod implicit) informația despre exit status al copiilor va fi aruncată iar copiii nu vor deveni procese zombie.

Windows (2p)

Exercițiul 1 - timer (2p)

Intrați în directorul 1-timer și urmăriți conținutul fișierului mytimer.c. Realizați un program care afișează data curentă la fiecare TIMEOUT secunde.

Folosiți componenta QuadPart a tipului LARGE\_INTEGER pentru specificarea timeout-ului. Timeout-ul este **negativ**și expiră la atingerea valorii 0. Al treilea argument al [SetWaitableTimerObject](http://msdn.microsoft.com/en-us/library/ms686289(VS.85).aspx) este timpul (în milisecunde) după care se va livra primul semnal de timer.

Folosiți un handler APC pentru tratarea timer-ului și afișarea mesajului.

* folosiți exemplul de utilizare a timer-elor cu APC-uri din [documentația MSDN](http://msdn.microsoft.com/en-us/library/ms686898(VS.85).aspx).
* ignorați warning-urile de compilare.
* completați funcțiile din fișierul sursă.
* folosiți [ctime](http://www.kernel.org/doc/man-pages/online/pages/man3/ctime.3.html) și [time](http://www.kernel.org/doc/man-pages/online/pages/man2/time.2.html) pentru afișarea timpului curent.
  + ctime adaugă un caracter new-line (\n) la sfârșitul șirului întors.
* folosiți [SleepEx](http://msdn.microsoft.com/en-us/library/ms686307(VS.85).aspx) și argumentele INFINITE, pentru a aștepta nedefinit, și TRUE, pentru a forța intrarea procesului într-o stare **alertabilă** (care să declanșeze rularea APC-ului).
  + urmăriți secțiunea [Waitable Timer Objects](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-04#waitable_timer_objects).

BONUS - Linux

2 so karma - timer

Intrați în directorul 6-timer și urmăriți conținutul fișierului mytimer.c.

Exercițiul urmărește afișarea timpului curent la fiecare TIMEOUT secunde. Pentru a nu consuma inutil timpul de procesor, se va suspenda procesul curent până la apariția semnalului generat de timer.

Urmăriți secțiunile cu TODO din fișierul sursă. Folosiți [ctime](http://linux.die.net/man/3/ctime) și [time](http://linux.die.net/man/2/time) pentru afișarea timpului curent. Puteți vedea un exemplu de folosire [aici](http://www.cplusplus.com/reference/clibrary/ctime/ctime/)

Folosiți [timer\_create](http://www.kernel.org/doc/man-pages/online/pages/man2/timer_create.2.html) și [timer\_settime](http://www.kernel.org/doc/man-pages/online/pages/man2/timer_settime.2.html) pentru a crea și arma timer-ul. **Semnalul generat de timer va fi SIGALRM**. Urmăriți secțiunea [Timere în Linux](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-04#timere_in_linux)

Folosiți [sigsuspend](http://www.kernel.org/doc/man-pages/online/pages/man2/sigsuspend.2.html) pentru a aștepta declanșarea semnalului generat de timer. Obțineți, folosind[sigprocmask](http://www.kernel.org/doc/man-pages/online/pages/man2/sigprocmask.2.html), masca procesului curent și transmiteți-o ca argument către [sigsuspend](http://www.kernel.org/doc/man-pages/online/pages/man2/sigsuspend.2.html). Urmăriți secțiunile [Timere în Linux](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-04#timere_in_linux) și [Așteptarea unui semnal](http://ocw.cs.pub.ro/courses/so/laboratoare/laborator-04#asteptarea_unui_semnal)

1 so karma - timer

Ramâneți în directorul 6-timer. Modificați sursa de la exercițiul 7 astfel încât să configurați funcția de handler direct din parametrii funcției [timer\_create()](http://www.kernel.org/doc/man-pages/online/pages/man2/timer_create.2.html). Urmăriți conținutul structurii sigevent. Un exemplu găsiți [aici](http://nicku.org/ossi/lab/processes/programming-posix-threads/sigev_thread.c).