

# Laborator 06 - Semnale

## Materiale ajutătoare

- [lab06-slides.pdf](#)
- [lab06-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.

Semnalele sunt un concept specific sistemelor de operare UNIX. Un semnal este o **întrerupere software**, în fluxul normal de execuție a unui proces. Sistemul de operare le folosește pentru a semnaliza 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 invalide 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 insucces. 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 invalidă a memoriei.
- Un **eveniment extern** este, în general, legat de I/O și de alte procese. Ele includ apariția de noi date de intrare, expirarea unui timer și 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, la fel ca anumite cereri explicite, făcute de către un proces, de a genera un semnal pentru același proces. 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 pentru 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 semnale, el va fi trimis imediat.

Când semnalul a fost primit, fie imediat, fie după o întârziere mare, 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` sau `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).

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 său poate determina cauza terminării sale examinând codul de terminare raportat de funcțiile `wait` și `waitpid`. Informațiile pe care le poate obține includ faptul că terminarea procesului s-a datorat unui 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 are un tip eroare de program, prin cerere explicită, și acesta termină procesul, fișierul este generat ca și cum semnalul ar fi fost generat de o eroare.

**Important:** În cazul în care un semnal este trimis procesului, în timp ce acesta executa un apel de sistem **blocant**, procesul va suspenda apelul, va executa handler-ul de tratare 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 se datorează faptului că un număr pentru un tip de semnal poate **varia** de la un sistem la altul, dar numele 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** - în momentul apăsării combinației de taste `CTRL+\`;
- **SIGSEGV** - în momentul accesării unei locații invalide 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](#) și [psignal](#). 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
```

```
#include <stdio.h>
```

```

#include <stdlib.h>

#define __USE_GNU
#include <string.h>

#include <signal.h>

int main(void) {
    char *sig_p = strsignal(SIGKILL);

    <a href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("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. Mască de biți este folosită de mai multe funcții, printre care și funcția [sigprocmask](#), 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 (toate semnalele neblockate) sau biți de 1 (toate semnalele blockate), de blocare a unui semnal, deblocare și detectare a blocării unui semnal:

```

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);

```

Secvența de mai jos constituie un caz de utilizare a funcțiilor de lucru cu mască 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);
}

```

## Tratarea semnalelor

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

În mod tradițional, funcția folosită pentru asocierea de handler-e pentru tratarea unui semnal era [signal](#). Pentru a preîntâmpina deficiențele acestei funcții, standardul POSIX a definit funcția [sigaction](#) 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 omonimă, descrisă tot 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;
};

```

În cazul în care în câmpul `sa_flags` se precizează flag-ul `SA_SIGINFO`, handler-ul folosit este cel specificat de `sa_sigaction`. Altfel, handler-ul folosit este `sa_handler`. `sa_mask` indică o mască de semnale care ar trebui blocate în timpul execuției handler-ului.

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) {
    ...
}

int main(void) {
    struct sigaction sa;

    memset(&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;
}

```

**Atenție!** 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`

În cazul prezenței `SA_SIGINFO`, 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](#).

```

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` și `si_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 `SIGBUS` completează `si_addr` cu adresa care a provocat eroarea.

## Semnalarea proceselor

Pentru transmiterea unui semnal, se poate folosi funcția [kill](#) sau funcția [sigqueue](#). Funcția [kill](#) 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](#):

```
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 flagul `SA_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

Soluția cea mai bună pentru a aștepta un semnal se poate realiza prin utilizarea funcției [sigsuspend](#):

```
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](#) 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

### 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 funcției `sigaction` (în momentul în care timer-ul expiră se generează un semnal, care la rândul lui generează rularea handler-ului asociat), sau

direct prin intermediul parametrilor funcției [timer\\_settime](#).

Utilizarea unui timer presupune mai mulți pași:

- **crearea unui timer** - folosind funcția [timer\\_create](#):

```
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](#):

```
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](#)

```
int timer_delete(timer_t timerid);
```

**Important** - 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](#) sau [nanosleep](#). Avantajul folosirii funcției [sleep](#) este simplitatea. Dezavantajele sunt rezoluția scăzută (secunde) și posibila interacțiune cu semnale (în special SIGALRM). [nanosleep](#) 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](#), [APC-uri](#), [evenimente de consolă](#), [timere](#). **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](#) 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](#)
- **setarea** unui timer - se poate face cu funcția [SetWaitableTimer](#)
- **anularea** - se realizează cu ajutorul funcției [CancelWaitableTimer](#)
- **așteptarea** - folosind funcția [WaitForSingleObject](#)

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(stderr, "CreateWaitableTimer failed (%d)\n", GetLastError());
        exit(-1);
    }

    /* configure to expire in 5 seconds; base unit is 100ns */
    dueTime.QuadPart = -500000000LL;
    if (SetWaitableTimer(timerHandle, &dueTime, 0, NULL, NULL, 0) == FALSE) {
        fprintf(stderr, "SetWaitableTimer failed (%d)\n", GetLastError());
        exit(-1);
    }

    if (WaitForSingleObject(timerHandle, INFINITE) != WAIT_OBJECT_0) {
        fprintf("WaitForSingleObject failed (%d)\n", GetLastError());
        exit(-1);
    }

    <a href="http://www.opengroup.org/onlinepubs/009695399/functions/printf.html">printf</a>("5 seconds timer
    expired\n");

    return 0;
}
```

## Exerciții

În rezolvarea laboratorului folosiți arhiva de sarcini [lab06-tasks.zip](#)

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

## Linux

**Atenție:** La folosirea `sigaction`, veți inițializa, în general, câmpul `sa_flags` al structurii `struct sigaction` la 0.

1. (1 punct) Intrați în directorul `1-hitme/`:

- Analizați conținutul fișierului `hitme.c`, compilați și rulați programul.
- 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)`
- Folosiți comanda `kill -l` pentru a lista toate semnalele disponibile. Ce valoare are semnalul `SIGKILL`?
- 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.
- **Hints:**
  - Analizați cu atenție `signals.sa_flags`.
  - Puteți reveni la secțiunea [Tratarea semnalelor](#) sau să investigați structura [sigaction](#)

2. (1 punct) Normal signals vs Real-Time signals

- Intrați în directorul `2-signals` și urmăriți conținutul fișierului `signals.c`.
- Programul numără de câte ori se apelează handlerul de semnal în cazul trimerii 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 ?
- **Hint**
  - Revedeți secțiunea [Tipuri standard de semnale](#)
- 3. (1 punct) Intrați în directorul 3-askexit.
  - 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 va întreba utilizatorul dacă dorește să încheie execuția sau nu.
  - Testați funcționalitatea programului.
  - Observați vreo problemă în folosirea funcțiilor `printf`, `scanf` în handler de semnale?
  - **Hint**
    - Consultați secțiunea [Tratarea semnalelor](#).
- 4. (2 puncte) Intrați în directorul 4-nohup.
  - Realizați un program, denumit `mynohup`, care simulează comanda [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.
    - **Hint:**
      - Revedeți secțiunea despre [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](#).
  - Testare:
    - Rulați `./mynohup ls -R / &`
    - După rulare închideți sesiunea de shell curentă.
      - Folosiți comenzile `exit`, `logout` sau combinația de taste CTRL+D.
      - **Nu** închideți fereastra terminalului în care rulează shell-ul (folosind mediul grafic).
    - Dintr-o altă consolă rulați `ps -ef | grep ls`
      - Cine este noul părinte al procesului?
  - Consultați secțiunea [Tratarea semnalelor](#) și secțiunile [Înlocuirea imaginii unui proces](#) și [Redirecțări](#) din laboratoarele precedente.
- 5. (2 puncte) Intrați în directorul 5-zombie.
  - Urmăriți conținutul fișierelor `mkzombie.c` și `nozombie.c`.
  - Fiecare program va crea câte un proces copil nou, care doar va apela `exit`.
  - `mkzombie` va aștepta TIMEOUT secunde și va ieși.
    - Din altă consolă rulați `ps -eF | grep zombie`
    - Observați faptul că procesul copil, deși nu mai rulează, apare în lista de procese ca și are un pid (unic în sistem la acel moment).
    - Observați că, după moartea procesului părinte, dispăre și procesul zombie.
  - `nozombie` va aștepta TIMEOUT secunde și va ieși.
    - Implementați `nozombie` fără a folosi funcțiile de așteptare de tipul `wait`, astfel încât procesul copil să nu treacă în starea de zombie.
      - Folosiți semnalul SIGCHLD. Informații găsiți în [sigaction\(2\)](#) și [wait\(2\)](#).
      - **Pe scurt:** if the parent explicitly ignores SIGCHLD by setting its handler to SIG\_IGN (rather than simply ignoring the signal by default) all child exit status information will be discarded and no zombie processes will be left.
  - Consultați secțiunile [Tratarea semnalelor](#) și [Crearea unui proces](#).
- 6. (3 puncte) Intrați în directorul 6-timer.
  - 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.
  - **Hints:**
    - Urmăriți secțiunile cu TODO din fișierul sursă.
    - Folosiți [ctime](#) și [time](#) pentru afișarea timpului curent.
      - Puteți vedea un exemplu de folosire [aici](#)
    - Folosiți [timer\\_create](#) și [timer\\_settime](#) pentru a crea și arma timer-ul. Semnalul generat va fi SIGALRM.
      - Urmăriți secțiunea [Timere în Linux](#)
    - Folosiți [sigsuspend](#) pentru a aștepta declanșarea semnalului SIGALRM.
      - Obțineți, folosind [sigprocmask](#), masca procesului curent și transmiteți-o ca argument către [sigsuspend](#).
    - Urmăriți secțiunile [Timere în Linux](#) și [Așteptarea unui semnal](#)

## BONUS - Linux

1. **(1 so karma)** Ramâneți în directorul `6-timer`.
  - Modificați sursa de la exercițiul 6 astfel încât să configurați funcția de handler direct din parametrii funcției `timer_create()`
  - Hint:
    - Urmăriți conținutul structurii `sigevent`
    - Un exemplu găsiți [aici](#)

## BONUS - Windows

1. **(2 so karma)** Intrați în directorul `3-timer`.
  - Urmăriți conținutul fișierului `mytimer.c`.
  - Realizați un program care afișează data curentă la fiecare `TIMEOUT` secunde.
  - **Hints:**
    - 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](#) este timpul (în milisecunde) după care se va livra primul semnal de timer.
  - Folosiți un handler APC pentru tratarea timer-ului și afișarea mesajului:
  - **Hints:**
    - Folosiți exemplul de utilizare a timer-elor cu APC-uri din [documentația MSDN](#).
    - Ignorați warning-urile de compilare.
    - Completați funcțiile din fișierul sursă.
    - Folosiți [ctime](#) și [time](#) pentru afișarea timpului curent.
      - `ctime` adaugă un caracter new-line (`\n`) la sfârșitul șirului întors.
    - Folosiți [SleepEx](#) 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](#).

## Soluții

[Soluții exerciții laborator 6](#)

## Resurse utile

- TLPI - Chapter 22: Signals - Advanced features
- TLPI - Chapter 23: Timers and Sleeping
- WSP - Chapter 14: Asynchronous IO - Waitable Timers

From:

<http://elf.cs.pub.ro/so/wiki/> - **Sisteme de Operare**

Permanent link:

<http://elf.cs.pub.ro/so/wiki/laboratoare/laborator-06>

Last update: 2012/03/25 21:33