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 cunoaşte o multitudine de situaţii neprevăzute, care-i afectează cursul normal de execuţie. Dacă procesul nu le poate trata, ele sunt pasate, mai departe, sistemului de operare. Cum sistemul de operare nu poate şti dacă procesul îşi poate continua execuţia în mod normal, fără efecte secundare nedorite, este obligat să termine procesul în mod forţat. 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 execuţie a unui proces. Sistemul de operare le foloseşte pentru a semnala procesului apariţia unor situaţii excepţionale oferindu-i procesului posibilitatea de a reacţiona. Fiecare semnal este asociat cu o clasă de evenimente care pot apărea şi care respectă anumite criterii. Procesele pot trata, bloca, ignora sau lăsa sistemul de operare să efectueze acţiunea implicită la primirea unui semnal (de obicei acţiunea implicită este **terminarea** procesului).

- Dacă un proces doreşte să **ignore** un semnal, sistemul de operare nu va mai trimite acel semnal procesului.

- Dacă un proces specifică faptul că doreşte 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 hotărăşte că vrea să primească, din nou, semnale de acel tip, dacă exista vreun semnal în aşteptare, acesta va fi trimis.

Mulţimea tipurilor de semnale este finită; sistemul de operare ţine, pentru fiecare proces, o **tabelă cu acţiunile** alese de acesta, pentru fiecare tip de semnal. La fiecare moment de timp aceste acţiuni sunt bine determinate. La pornirea procesului tabela de acţiuni este iniţializată 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 execuţie al procesului care primeşte semnalul dacă evenimentul care cauzează trimiterea semnalului este sincron/asincron cu fluxul de execuţie al procesului.

- Un eveniment este **sincron** cu fluxul de execuţie al procesului dacă apare de fiecare dată la rularea programului, în acelaşi punct al fluxului de execuţie. Exemple în acest sens sunt încercarea de accesare a unei locaţii de memorie invalide sau nepermise, împărţire 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 doreşte 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-şi poate trimite şi 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 destinaţie.

În anumite cazuri, există nevoia de a şti, în mod sigur, că un semnal trimis a ajuns la destinaţie şi, implicit, că procesul va răspunde la el (efectuând una din acţiunile posibile). Sistemul de operare oferă un alt mod de a trimite un semnal, prin care se **garantează** fie că semnalul a ajuns la destinaţie, fie că această acţiune a eşuat. Acest lucru este realizat prin

crearea unei stive de semnale, de o anumită capacitate (ea trebuie să fie finită, pentru a nu produce situaţii de overflow). La trimiterea unui semnal, sistemul de operare verifică dacă stiva este plină. În acest caz, cererea eşuează, altfel semnalul este pus în stivă şi operaţia se termină cu succes. Modul clasic de a trimite semnale este analog cu acesta (stiva are dimensiunea 1) cu excepţia faptului că nu se oferă informaţii despre ajungerea la destinaţie a unui semnal.

Noţiunea 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 operaţie nepermisă şi nu-şi poate continua execuţia. Însă, nu toate tipurile de erori generează semnale (de fapt, cele mai multe nu o fac). De exemplu, deschiderea unui fişier 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 împărţirea cu zero şi accesarea invalidă a memoriei.

- Un **eveniment extern** este, în general, legat de I/O şi de alte procese. Ele includ apariţia de noi date de intrare, expirarea unui timer şi terminarea execuţiei 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 acţiune specifică din program, şi este trimis (dacă nu este blocat) în timpul acelei acţiuni. 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 acelaşi proces. Pe anumite maşini, anumite tipuri de erori hardware (de obicei excepţiile în virgulă mobilă) nu sunt raportate complet sincron, şi pot ajunge câteva instrucţiuni mai târziu.

- Semnalele **asincrone** sunt generate de evenimente necontrolabile de către procesul care le primeşte. Aceste semnale ajung la momente de timp impredictibile . Evenimentele externe generează semnale în mod asincron, la fel ca şi 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 şi primirea semnalelor

Când un semnal este generat, el intră într-o stare de aşteptare (pending). În mod normal, el rămâne în această stare pentru o perioadă de timp foarte mică şi apoi este trimis procesului destinaţie. Însă, dacă acel tip de semnal este, în momentul de faţă, blocat, el ar putea rămâne în starea de aşteptare nedefinit, până când semnalele de acel timp 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, acţiunea specificată pentru acel semnal este executată. Pentru anumite semnale, cum ar fi `SIGKILL` şi `SIGSTOP`, acţiunea este **fixată** (procesul este terminat), dar, pentru majoritatea semnalelor, programul poate alege să:

- **ignore** semnalul
- specifice o funcţie de tip **handler**
- accepte **acţiunea implicită** pentru acel tip de semnal.

Programul îşi specifică alegerea utilizând funcţii precum http://linux.die.net/man/2/signal 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 handlerul care tratează semnalul).

Dacă acţiunea 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. Acelaşi lucru se întâmplă dacă semnalul este blocat în acel moment. Un semnal neglijat în acest mod nu va fi primit niciodata, nici dacă programul specifică ulterior o acţiune diferită pentru acel tip de semnal şi apoi îl deblochează.

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

Când un semnal forţează terminarea unui proces, părintele său poate determina cauza terminării sale examinând codul de terminare raportat de funcţiile `wait` şi `waitpid`. Informaţiile pe care le poate obţine includ faptul că terminarea procesului s-a datorat unui semnal, precum şi tipul semnalului. Dacă un program pe care îl rulaţi din linia de comandă este terminat de un semnal, shell-ul afişează, de obicei, nişte 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 şi un fişier ***core dump*** care înregistrează starea procesului în momentul terminării. Puteţi examina fişierul cu un debugger, pentru a afla ce anume a cauzat eroarea.

Dacă generaţi un semnal, care are un tip eroare de program, prin cerere explicită, şi acesta termină procesul, fişierul este generat ca şi 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 şi apoi fie operaţia va eşua (cu `errno` setat pe `EINTR`), fie se va reporni operaţia. Sistemele System V se comportă ca în primul caz, cele BSD ca în cel de al doilea.

Tipuri standard de semnale

Această secţiune prezintă numele pentru diferite tipuri standard de semnale şi descrie ce fel de evenimente indică. Fiecare nume de semnal este o **macrodefiniţie** 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.

Mai cunoscute sunt următoarele semnale:

- `SIGINT` - transmis la apăsarea combinaţiei `CTRL+C`;
- `SIGQUIT` - în momentul apăsării combinaţiei de taste `CTRL+\`;
- `SIGSEGV` - în momentul accesării unei locaţii 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 afişa un mesaj de descriere a unui semnal este utilizarea funcţiilor strsignal şi psignal. Aceste funcţii folosesc un număr de semnal pentru a specifica tipul de semnal care trebuie descris. Mai jos este prezentat un exemplu de folosire a acestor funcţii:

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 şi rulare secvenţa este:

```
so@spook$ gcc -Wall -g -o msg_signal msg_signal.c
so@spook$ ./msg_signal
signal 9 is Killed
death and decay: Killed
```

Măşti de semnale. Blocarea semnalelor

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

O **mască de semnale** are fiecare bit asociat unui tip de semnal. Masca de biţi este folosită de mai multe funcţii, printre care şi funcţia sigprocmask, folosită pentru schimbarea măştii de semnale a procesului curent.

Tipul de date folosit de sistemele UNIX pentru a reprezenta măştile de semnale este `sigset_t`. Variabilele de acest tip sunt neiniţializate. Operaţiile pe acest tip de date sunt de iniţializare cu biţi de 0 (toate semnalele neblocate) sau biţi de 1 (toate semnalele blocate), de blocare a unui semnal, deblocare şi 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);
```

Secvenţa de mai jos constituie un caz de utilizare a funcţiilor 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);
}
```

Tratarea semnalelor

Tratarea semnalelor se realizează prin asocierea unei funcţii (**handler**) unui semnal. Funcţia va fi apelată în momentul în care procesul recepţionează mesajul.

În mod tradiţional, funcţia folosită pentru asocierea de handler-e pentru tratarea unui semnal era signal. Pentru a preîntâmpina deficienţele acestei funcţii, standardul POSIX a definit funcţia sigaction pentru asocierea unui handler cu un semnal. `sigaction` oferă mai mult control, cu preţul unui grad de complexitate mai mare.

Componenta importantă a funcţiei `sigaction` este structura omonimă, descrisă tot în pagina de manual a funcţiei:

```
 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 execuţiei 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;
}
```

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 acţiunii implicite (terminarea procesului, ignorarea semnalului etc).

Structura siginfo_t

În cazul prezenţei `SA_SIGINFO`, se foloseşte câmpul `sa_sigaction` al structurii `sigaction` pentru a specifica handler-ul asociat semnalului. Handler-ul folosit primeşte în acest caz trei parametri şi poate fi folosit pentru a transmite o informaţie utilă, o dată cu procesul. Al treilea argument (de tipul `void*`) este rar utilizat. Al doilea argument, de tipul `siginfo_t` defineşte o structură ce conţine informaţii utile despre contextul apariţiei semnalului şi alte informaţii pe care le poate furniza programatorul. Definiţia structurii se găseşte în pagina de manual a funcţiei sigaction.

Membrii structurii sunt iniţializaţi numai atunci când valorile lor sunt utile. Membrii `si_signo`, `si_errno` şi `si_code` sunt întotdeauna definiţi pentru toate semnalele. Restul structurii poate fi o uniune, aşa că ar trebui citite numai câmpurile care au sens pentru semnalul primit. Spre exemplu, apelul de sistem `kill`, semnalele POSIX.1b şi `SIGCHLD` completează `si_pid` şi `si_uid`, iar `SIGILL`, `SIGFPE`, `SIGSEGV` şi `SIGBUS` completează `si_addr` cu adresa care a provocat eroarea.

Semnalarea proceselor

Pentru transmiterea unui semnal, se poate folosi funcţia kill sau funcţia sigqueue. Funcţia kill are dezavantajul că **nu** garantează recepţionarea semnalului de procesul destinaţie. Dacă este nevoie să se trimită un semnal unui proces şi să se ştie sigur că a ajuns se recomandă folosirea funcţiei sigqueue:

```
int sigqueue(pid_t pid, int signo, const union sigval value);
```

Funcţia trimite semnalul `signo`, cu parametrii specificaţi 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.

Condiţiile cerute pentru ca un proces să aibă permisiunea de a trimite un semnal altui proces sunt aceleaşi ca şi în cazul lui `kill`. Dacă semnalul specificat este blocat în acel moment, funcţia va ieşi imediat şi dacă flagul `SA_SIGINFO` este setat şi 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, şi si_value va fi setat la value.

Dacă flagul SA_SIGINFO nu este setat, atunci signo, dar nu în mod necesar şi value, vor fi trimise, cel puţin o dată, procesului care trebuie să primească semnalul.

Aşteptarea unui semnal

Soluţia cea mai bună pentru a aştepta un semnal se poate realiza prin utilizarea funcţiei sigsuspend:

```
int sigsuspend(const sigset_t *set);
```

Funcţia înlocuieşte masca de semnale blocate a procesului, cu set, şi suspendă procesul până când este primit un semnal care nu este blocat de noua mască. La ieşire, funcţia restaurează vechea mască de semnale.

În secvenţa de mai jos, funcţia sigsuspend este folosită pentru a întrerupe procesul curent până la recepţionarea semnalului SIGINT. Semnalele SIGKILL şi SIGSTOP, deşi 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 funcţiilor 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 şi configurarea handler-ului care va rula. Configurarea handler-ului se poate realiza atât prin intermediul funcţiei sigaction (în momentul in care timerul expiră se generază un semnal, care la rândul lui generează rularea handlerului asociat), sau direct prin intermediul parametrilor funcţiei timer_settime.

Utilizarea unui timer presupune mai mulţi paşi:

- **crearea unui timer** - folosind funcţia timer_create:

```
 int timer_create(clockid_t clockid, struct sigevent *evp, timer_t *timerid)
```

Timerul create se indentifică prin timerid. Prin intermediul structurii sigevent se seteaza modul în care va interacţiona timerul cu procesul/threadul 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 a procesului sau timpul de rulare a procesului în user-space şi kernel-space. La timeout timerul va livra semnalul salvat în sev.sigev_signo

- **armarea unui timer** - folosind funcţia timer_settime:

```
int timer_settime(timer_t timerid, int flags,
                        const struct itimerspec *new_value,
```

```
                        struct itimerspec * old_value);
```

Armarea timerului presupune completarea structurii `itimerspec` în care specifică timpul de pornire al timerului, cât şi intervalul de expirare al timeout-ului (intervalele sunt măsurate în secunde şi 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);
```

- **ştergerea unui timer** - folosind funcţia timer_delete

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

Important - pentru a folosi funcţiile de mai sus programul trebuie compilat cu `-lrt`

Una din formele de utilizare a timerelor este implementarea funcţiilor de aşteptare de tipul sleep sau nanosleep. Avantajul folosirii funcţiei sleep este simplitatea. Dezavantajele sunt rezoluţia scăzută (secunde) şi posibila interacţiune cu semnale (în special `SIGALRM`). nanosleep are un apel mai complex, dar oferă rezoluţie până la ordinul nanosecundelor şi este *signal-safe* (nu interacţionează 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 execuţie asincronă în contextul unui proces/thread. Pot fi folosite pentru rularea unei secvenţe de cod în combinaţie 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 funcţiei SetWaitableTimer setează un nou timp de aşteptare;
- timer cu **resetare automată**: un timer a cărui stare rămâne *signaled* până când un thread efectuează o operaţie de aşteptare pe acel obiect.

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

- **crearea** unui timer - se realizează prin intermediul funcţiei CreateWaitableTimer
- **setarea** unui timer - se poate face cu funcţia SetWaitableTimer
- **anularea** - se realizează cu ajutorul funcţiei CancelWaitableTimer
- **aşteptarea** - folosind funcţia WaitForSingleObject

În secvenţa de cod de mai jos, se foloseşte un timer pentru afişarea 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 = -50000000LL;
    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;
}
```

Exerciţii

În rezolvarea laboratorului folosiţi arhiva de sarcini lab06-tasks.zip

Observaţii: Pentru a vă ajuta la implementarea exerciţiilor din laborator, în directorul `utils` din arhivă există un fişier `utils.h` cu funcţii utile.

Linux

Atenţie: La folosirea `sigaction`, veţi iniţializa, în general, câmpul `sa_flags` al structurii `struct sigaction` la `0`.

1. (**1 punct**) Intraţi în directorul `1-hitme/`:
   - Analizaţi conţinutul fişierului `hitme.c`, compilaţi şi rulaţi programul.
   - Într-o altă consolă trimiteţi programului `hitme` semnale cu valori cuprinse între 20 şi 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)
   - Folosiţi comanda `kill -l` pentru a lista toate semnalele disponibile. Ce valoare are semnalul `SIGKILL`?
   - Încercaţi să trimiteţi acelaşi semnal de două ori şi explicaţi comportamentul. Ce ar trebui schimbat ca să se poată trimite acelaşi semnal de două ori.
   - **Hints:**
     - Analizaţi cu atenţie `signals.sa_flags`.
     - Puteţi reveni la secţiunea Tratarea semnalelor sau să investigaţi structura sigaction
2. (**1 punct**) Normal signals vs Real-Time signals
   - Intraţi în directorul `2-signals` şi urmăriţi conţinutul fişierului `signals.c`.
   - Programul numără de câte ori se apelează handlerul de semnal în cazul trimiterii semnalelor `SIGINT` şi `SIGRTMIN`
   - Pentru cazul semnalelor `normale`:
     - Porniţi într-o consolă programul `signals`:./signals
     - Într-o altă consolă rulaţi scriptul `send_normal.sh`:./send_normal.sh
   - Pentru semnalele `real-time`:
     - Porniţi într-o consolă programul `signals`:./signals
     - Într-o altă consolă rulaţi scriptul `send_rt.sh`:./send_rt.sh
   - Pentru a închide executabilul `signals` trimiteţi semnalul `SIGQUIT` (revedeţi secţiunea Tipuri standard de semnale)
   - De unde apare diferenţa ?
3. (**1 punct**) Intraţi în directorul `3-askexit`.
   - Programul face busy waiting, afişând la consolă numere consecutive.
   - Trebuie să completaţi programul pentru a intercepta semnalele generate de `CTRL+\`, `CTRL+C` şi `SIGUSR1` (folosiţi comanda `kill`). Handler-ul asociat cu fiecare din semnale va fi `ask_handler`.
   - Pentru fiecare semnal primit va întreba utilizatorul dacă doreşte să încheie execuţia sau nu.
   - Testaţi funcţionalitatea programului.
   - Observaţi vreo problemă în folosirea funcţiilor `printf`, `scanf` în handlere de semnale?
   - **Hints**
     - Consultaţi secţiunea Tratarea semnalelor.
4. (**2 puncte**) Intraţi în directorul `4-nohup`.
   - Realizaţi un program, denumit `mynohup`, care simulează comanda nohup.
   - Programul primeşte, 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 înştiinţat de închiderea terminalului la care era conectat.
     - Va trebui să ignoraţi semnalul `SIGHUP`, livrat de shell procesului, în momentul încheierii sesiunii curente.
   - Dacă fişierul standard de ieşire era legat la un terminal acesta trebuie redirectat în fişierul `NOHUP_OUT_FILE`.
     - Folosiţi apelul isatty.

- Testare:
  - Rulaţi ./mynohup sleep 100 &
  - După rulare închideţi sesiunea de shell curentă.
    - Folosiţi comenzile `exit`, `logout` sau combinaţia de taste `CTRL+D`.
    - **Nu** închideţi fereastra terminalului în care rulează shell-ul (folosind mediul grafic).
  - Dintr-o altă consolă rulaţi ps -ef | grep sleep Cine este noul părinte al procesului?
- Consultaţi secţiunea Tratarea semnalelor şi secţiunile Înlocuirea imaginii unui proces şi Redirectări din laboratoarele precedente.

5. (**2 puncte**) Intraţi în directorul `5-zombie`.
   - Urmăriţi conţinutul fişierelor `mkzombie.c` şi `nozombie.c`.
   - Fiecare program va crea câte un proces copil nou, care doar va apela `exit`.
   - `mkzombie` va aştepta `TIMEOUT` secunde şi va ieşi.
     - Din altă consolă rulaţi ps -eF | grep zombie
     - Observaţi faptul că procesul copil, deşi nu mai rulează, apare în lista de procese ca şi are un pid (unic în sistem la acel moment).
     - Observaţi că, după moartea procesului părinte, dispare şi procesul zombie.
   - `nozombie` va aştepta `TIMEOUT` secunde şi va ieşi.
     - Implementaţi `nozombie` fără a folosi funcţiile de aşteptare de tipul `wait`, astfel încât procesul copil să nu treacă în starea de zombie.
       - Folosiţi semnalul `SIGCHLD`. Informaţii găsiţi în sigaction(2) şi 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.
   - Consultaţi secţiunile Tratarea semnalelor şi Crearea unui proces.
6. (**3 puncte**) Intraţi în directorul `6-timer`.
   - Urmăriţi conţinutul fişierului `mytimer.c`.
   - Exerciţiul urmăreşte afişarea timpului curent la fiecare `TIMEOUT` secunde. Pentru a nu consuma inutil timpul de procesor, se va suspenda procesul curent până la apariţia semnalului generat de timer.
   - **Hints**:
     - Urmăriţi secţiunile cu `TODO` din fişierul sursă.
     - Folosiţi ctime şi time pentru afişarea timpului curent.
       - Puteţi vedea un exemplu de folosire aici
     - Folosiţi timer_create şi timer_settime pentru a crea şi arma timerul. Semnalul generat va fi `SIGALRM`.
       - Urmăriţi secţiunea Timere în Linux
     - Folosiţi sigsupend pentru a aştepta declanşarea semnalului `SIGALRM`.
       - Obţineţi, folosind sigprocmask, masca procesului curent şi transmiteţi-o ca argument către sigsupend.
     - Urmăriţi secţiunile Timere în Linux şi Aşteptarea unui semnal

BONUS - Linux

1. (**1 so karma**) Ramâneţi în directorul `6-timer`.
   - Modificaţi sursa de la exerciţiul 6 astfel încât să configuraţi funcţia de handler direct din parametrii funcţiei timer_create()
   - Hint:
     - Urmariţi conţinutul structurii `sigevent`
     - Un exemplu găsiţi aici

BONUS - Windows

1. (**2 so karma**) Intraţi în directorul `3-timer`.
   - Urmăriţi conţinutul fişierului `mytimer.c`.
   - Realizaţi un program care afişează data curentă la fiecare `TIMEOUT` secunde.
   - **Hints**:
     - Folosiţi componenta `QuadPart` a tipului `LARGE_INTEGER` pentru specificarea timeout-ului. Timeout-ul este **negativ** şi expiră la atingerea valorii `0`.
     - Al treilea argument al SetWaitableTimerObject este timpul (în milisecunde) după care se va livra primul semnal de timer.
   - Folosiţi un handler APC pentru tratarea timer-ului şi afişarea mesajului:
   - **Hints**:
     - Folosiţi exemplul de utilizare a timer-elor cu APC-uri din documentaţia MSDN.
     - Ignoraţi warning-urile de compilare.

- Completaţi funcţiile din fişierul sursă.
- Folosiţi ctime şi time pentru afişarea timpului curent.
  - ctime adaugă un caracter new-line (`\n`) la sfârşitul şirului întors.
- Folosiţi SleepEx şi argumentele `INFINITE`, pentru a aştepta nedefinit, şi `TRUE`, pentru a forţa intrarea procesului într-o stare **alertabilă** (care să declanşeze rularea APC-ului).
- Urmăriţi secţiunea Waitable Timer Objects.

Soluţii

Soluţii exerciţii laborator 6

Resurse utile

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