Semnale

Contents

Semnale în Linux

În lumea reală un proces se poate întâlni cu o multitudine de situaii neprevăzute care-i afectează cursul de execuie normal. Dacă procesul nu le poate trata ele sunt trecute mai departe sistemului de operare. Cum sistemul de operare nu poate ti dacă procesul îi poate continua execuia în mod normal fără efecte secundare nedorite este obligat să termine procesul în mod forat. O rezolvare a acestei probleme o reprezintă semnalele.

Semnalele sunt un concept specific sistemelor de operare UNIX. Un semnal este o întrerupere software a unui proces din fluxul de execuie normal. Sistemul de operare le folosete pentru a semnala procesului apariia unor situaii excepionale oferindu-i procesului posibilitatea de a interaciona. Fiecare semnal este asociat cu o clasă de evenimente care pot apărea i care respectă anumite criterii. Procesele pot trata, bloca, ignora sau lăsa sistemul de operare să efectueze aciunea implicită la primirea unui semnal (de obicei aciunea implicită este terminarea procesului).

Mulimea tipurilor de semnale este finită; sistemul de operare ine pentru fiecare proces o tabelă cu aciunile alese de acesta pentru fiecare tip de semnal. La fiecare moment de timp aceste aciuni sunt bine determinate. La pornirea procesului tabela de aciuni este iniializată cu valorile implicite. Modul de tratare al semnalului nu este decis la primirea semnalului de proces, ci se alege în mod automat din tabelă.

În continuare se va folosi noiunea de *semnal* pentru a indica alternativ fie un anumit tip de semnal, fie efectiv obiectele de acest tip.

Semnalele pot fi de asemenea folosite de alte procese pentru a întrerupe anumite procese. Semnalele sunt sincrone/asincrone cu fluxul de execuie al procesului care primete semnalul dacă evenimentul care cauzează trimiterea semnalului este sincron/asincron cu fluxul de execuie al procesului.

Un eveniment este sincron cu fluxul de execuie al procesului dacă apare de fiecare dată la rularea programului, în acelai punct al fluxului de execuie. Exemple în acest sens sunt încercarea de accesare a unei locaii de memorie invalide sau nepermisă, împărire prin 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 dorete să reseteze calculatorul.

Un semnal primit de un proces poate fi generat fie direct de sistemul de operare (în cazul în care acestea raportează diferite erori), fie de un proces (care-i poate trimite i singur semnale); evident semnalul trimis de un proces va trece tot prin sistemul de operare. Daca un proces dorete să ignore un semnal, sistemul de operare nu va mai trimite acel semnal procesului. Dacă un proces specifică că dorete să blocheze un semnal, sistemul de operare nu va mai trimite semnalele de acel tip spre procesul în cauză, i va salva numai primul semnal de acel tip, restul pierzându-se. Când procesul hotărăte că vrea să primească din nou semnale, dacă era vreun semnal în ateptare va fi trimis. 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 destinaie.

În anumite cazuri, există nevoia de a ti în mod sigur că un semnal trimis a ajuns la destinaie i implicit că procesul va răspunde la el (efectuând una din aciunile posibile). Pentru aceste situaii, sistemul de operare oferă un alt mod de a trimite un semnal prin care se garantează fie că semnalul a ajuns la destinaie, fie că această aciune a euat. Acest lucru este realizat prin crearea unei stive de semnale, de o anumită capacitate (ea trebuie să fie finită pentru a nu produce situaii de *overflow*). La trimiterea unui semnal, când cererea ajunge la sistemul de operare acesta verifică dacă stiva este plină. În acest caz cererea euează, altfel semnalul va fi pus în coadă i operaia se termină cu succes. Modul anterior de a trimite semnale este analog cu acesta (stiva are dimensiunea unu) cu excepia faptului că nu se oferă informaii despre ajungerea la destinaie a unui semnal.

Conceptele generării semnalelor

În general, evenimentele care generează semnale se încadrează în trei categorii majore: erori, evenimente externe i cereri explicite.

O eroare indică faptul că un program a făcut ceva greit i nu-i poate continua execuia. Însă nu toate tipurile de erori generează semnale - de fapt, cele mai multe nu o fac. De exemplu, deschiderea unui fiier 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(open returnează 0 în caz de succes, ca majoritatea funciilor în Unix). În general, erorile care sunt asociate cu anumite biblioteci sunt raportate returnând o valoare care indică o eroare. Erorile care generează semnale sunt cele care pot apărea oriunde în program, nu doar în apelurile din biblioteci. Ele includ împărirea cu zero i adresele de memorie invalide.

Un eveniment extern este în general legat de I/O i alte procese. Ele includ apariia de noi date de intrare, expirarea unui timer, i terminarea execuiei unui proces copil.

O cerere explicită indică utilizarea unei funcii de bibiliotecă cum ar fi kill pentru a genera un semnal.

Semnalele pot fi generate sincron sau asincron. Un semnal sincron se raportează la o aciune specifică din program, i este trimis(dacă nu este blocat) în timpul acelei aciuni. 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 acelai proces. Pe anumite maini, anumite tipuri de erori hardware (de obicei excepiile în virgulă mobilă) nu sunt raportate complet sincron, i pot ajunge câteva instruciuni mai târziu.

Semnalele asincrone sunt generate de evenimente necontrolabile de către procesul care le primete. Aceste semnale ajung la momente de timp impredictibile în timpul execuiei. Evenimentele externe generează semnale în mod asincron, la fel ca i 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 i primirea semnalelor

Când un semnal este generat, el intră într-o stare de ateptare (pending). În mod normal el rămâne în această stare pentru o perioadă de timp foarte mică i apoi este trimis procesului care era destinaia semnalului. Însă, dacă acel tip de semnal este în momentul de faă blocat, el ar putea rămâne în starea de ateptare în mod indefinit - până când semnalele de acel timp sunt deblocate. Odată deblocat acel tip de semnale, el va fi trimis imediat.

Când semnalul a fost primit, fie imediat sau după o întârziere mare, aciunea specificată pentru acel semnal este executată. Pentru anumite semnale. cum ar fi SIGKILL i SIGSTOP, aciunea este fixată (procesul este terminat), dar pentru majoritatea semnalelor programul poate alege să ignore semnalul, să specifice o funcie de tip handler, sau să accepte aciunea implicită pentru tipul acela de semnal. Programul îi specifică alegerea utilizând funcii precum signal sau sigaction. În timp ce handler-ul rulează, acel tip de semnale este în mod normal blocat (deblocarea se va face printr-o cerere explicită în handlerul care tratează semnalul).

Dacă aciunea 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. Acelai 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 aciune diferită pentru acel tip de semnal i apoi îl deblochează.

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

Când un semnal forează terminarea unui proces, părintele său poate determina cauza terminării sale examinând codul de terminare raportat de funciile wait i waitpid. Informaiile pe care le poate obine includ faptul că terminarea procesului s-a datorat unui semnal, precum i tipul semnalului. Dacă un program pe care îl rulai din linia de comandă este terminat de un semnal, shell-ul printează de obicei nite 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 i un fiier core dump care înregistrează starea procesului în momentul terminării. Putei examina fiierul cu un debugger pentru a afla ce anume a cauzat eroarea.

Dacă generai un semnal care e in mod normal de tip eroare de program prin cerere explicită, i acesta termină

procesul, fiierul este generat ca i 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 i apoi fie operaia va eua (cu errno setat pe EINTR), fie se va restarta operaia. Sistemele System V se comportă ca în primul caz, cele BSD ca în cel de al doilea.

Tipuri standard de semnale

Această seciune prezintă numele pentru diferite tipuri standard de semnale i descrie ce fel de evenimente indică. Fiecare nume de semnal este o macrodefiniie 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 sistem la sistem, dar numele sunt standard. Pentru lista completă de semnale suportate de un sistem se poate rula în linia de comandă `kill -l`

```
$ 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. Semnalul `SIGINT` este transmis la apăsarea combinaiei `CTRL+C`, semnalul `SIGQUIT` în momentul apăsării combinaiei de taste `CTRL+\`, `SIGSEGV` în momentul accesării unei locaii invalide de memorie etc. Semnalul `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 afia un mesaj de descriere a unui semnal este utilizarea funciilor strsignal i psignal. Aceste funcii folosesc un număr de semnal pentru a specifica tipul de semnal care trebuie descris. Mai jos este prezentat un exemplu de folosire a acestor funcii:

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

```
#define __USE_GNU
#include <string.h>

#include <signal.h>

int main(void)
{
char *sig_p;

        sig_p = strsignal(SIGKILL);

printf("signal %d is %s\n", SIGKILL, sig_p);

psignal(SIGKILL, "death and decay");

return 0;
}
```

Pentru compilare i rulare secvena este:

```
$ gcc -Wall -g -o strsignal_psignal strsignal_psignal.c
$ ./strsignal_psignal
signal 9 is Killed
death and decay: Killed
```

Măti de semnale. Blocarea semnalelor

Pentru a putea efectua operaii de blocare/deblocare semnale avem nevoie să tim la fiecare pas în fluxul de execuie starea fiecărui semnal la acel moment. Sistemul de operare are de asemenea nevoie de acelai lucru pentru a putea face o decizie asupra unui semnal care trebuie trimis unui proces (el are nevoie de acest gen de informaie pentru fiecare proces în parte). În acest scop se folosete o mască de semnale proprie fiecărui proces.

O mască de semnale are fiecare bit asociat unui tip de semnal. Masca de bii este folosită de mai multe funcii, printre care i funcia sigprocmask folosită pentru schimbarea mătii de semnale a procesului curent.

Tipul de date folosit de sistemele UNIX pentru a reprezenta mătile de semnale este `sigset_t`. Variabilele de acest tip sunt neiniializate. Operaiile pe acest tip de date sunt de iniializare cu bii de 0 (toate semnalele neblocate) sau bii de 1 (toate semnalele blocate), de blocare a unui semnal, deblocare i detectare a blocării unui semnal:

```
#include <signal.h>

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);
```

Secvena de mai jos este un caz de utilizare a funciilor 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 funcii (handler) unui semnal. Funcia va fi apelată în momentul în care procesul recepionează mesajul.

În mod tradiional, funcia folosită pentru asocierea de handlere pentru tratarea unui semnal era signal. Pentru a preîntâmpina deficienele acestei funcii, standardul POSIX a definit funcia sigaction pentru asocierea unui handler cu un semnal. sigaction oferă mai mult control, cu preul unui grad de complexitate mai mare.

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

```
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 execuiei 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 semnale;
    sigset_t mask;

    sigemptyset(&mask);
    sigaddset(&mask, SIGUSR2);

    semnale.sa_flags = SA_RESETHAND;   /* restore handler to previous state */
    semnale.sa_mask = mask;
    semnale.sa_handler = usr2_handler;
```

```
    sigaction(SIGUSR2, &semnale, NULL);

    return 0;
}
```

Programatorul poate opta pentru configurarea unui handler propriu sau poate folosi unul predefinit. Se poate folosi `SIG_IGN` pentru ignorarea semnalului sau `SIG_DFL` pentru rularea aciunii implicite (terminarea procesului, ignorarea semnalului etc.)

Structura siginfo_t

În cazul prezenei `SA_SIGINFO` se folosete câmpul `sa_sigaction` al structurii `sigaction` pentru a specifica handler-ul asociat semnalului. Handler-ul folosit primete în acest caz trei parametri i poate fi folosit pentru a transmite o informaie utilă o dată cu procesul. Al treilea argument (de tipul `void *`) este rar utilizat. Al doilea argument, de tipul `siginfo_t` definete o structură conine informaii utile despre contextul apariiei semnalului i alte informaii pe care le poate furniza programatorul. Definiia structurii se găsete în pagina de manual a funciei sigaction.

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

Semnalarea proceselor

Pentru transmiterea unui semnal, se poate folosi funcia kill sau funcia sigqueue. Funcia kill are dezavantajul că nu garantează recepionarea semnalului de procesul destinaie. Dacă este nevoie să se trimitem un semnal unui proces i să se tie sigur că a ajuns se recomandă folosirea funciei sigqueue:

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

Funcia trimite semnalul `signo` cu valoarea specificată 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 trimisa odata 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. In mod evident, nu are sens transmiterea de pointeri dintr-un proces in altul.

Condiiile cerute pentru ca un proces să aibă permisiunea de a trimite un semnal altui proces sunt aceleai ca i în cazul lui `kill`. Dacă semnalul specificat este blocat în acel moment, funcia va iei imediat i dacă flagul SA_SIGINFO este setat i există resurse necesare, semnalul va fi pus în coadă în starea pending (un proces poate avea în coadă de pending maxim SIGQUEUE_MAX semnale). De asemenea, când semnalul este

primit, câmpul `si_code` pasat structurii siginfo va fi setat la `SI_QUEUE`, i `si_value` va fi setat la `value`.

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

Ateptarea unui semnal

În cazul în care se utilizează semnalele pentru comunicare i/sau sincronizare există adeseori nevoie să se atepte ca un anumit tip de semnal să parvină procesului în cauză. Un simplu mod de a realiza acest lucru este o buclă a cărei condiie de ieire 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 funciei sleep:

```
while (!signal_has_arrived) {
sleep(1);
}
```

O astfel de abordare nu ar mai ocupa timp de procesor inutil, dar timpul de răspuns în cazul sosirii unui semnal este destul de mare. O altă soluie a problemei este funcia pause (care blochează fluxul de execuie până când procesul curent este întrerupt de un semnal). Dei această abordare pare foarte simplă, ea introduce adeseori *deadlock-uri* care blochează programul nedefinit. Un exemplu în acest sens este pseudosoluia de mai jos la problema ateptării unui semnal:

```
while (!signal_has_arrived) {
pause();
}
```

Bucla este necesară pentru prevenirea situaiei în care procesul este întrerupt de alte semnale decât cel ateptat. Se poate întâmpla ca semnalul să ajungă după testarea variabilei i inainte de apelul funciei pause. În acest caz procesul se blochează i dacă nu apare un alt semnal care să cauzeze ieirea din pause, el va rămâne blocat nedefinit.

Soluia cea mai bună pentru a atepta un semnal se poate realiza prin utilizarea funciei sigsuspend:

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

Funcia înlocuiete masca de semnale blocate a procesului cu `set` i apoi suspendă procesul până când este primit un semnal care nu este blocat de noua mască. La ieire, funcia restaurează vechea mască de semnale.

În secvena de mai jos, funcia sigsuspend este folosită pentru a întrerupe procesul curent până la recepionarea semnalului `SIGUSR1`. Semnalele `SIGKILL` i `SIGSTOP`, dei 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

Un obiect de tip semnal este ataat unui obiect de tip proces. Dacă procesul nu rulează în acel moment sistemul de operare poate ataa unui proces numai un singur semnal care va rămâne în starea pending. Dacă procesul rula în acel moment semnalul primit este deservit imediat i va întrerupe fluxul normal de execuie, permiându-se primirea fără pierdere a unui semnal de acelai tip. Evident, acelai lucru se întâmplă i cu semnalele trimise cu *sigqueue*, cu deosebirea că numărul de semnale care pot fi pending pentru un proces nu mai este 1, ci SIGQUEUE_MAX.

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

Fluxul de execuie al unui proces este văzut de către sistemul de operare ca o înirure de instruciuni pe care platforma le suportă. De aceea unele operaii din limbajele de programare de nivel înalt nu sunt atomice i indivizibile, fiind nevoie de mai multe instruciuni în cod maină pentru a se efectua respectiva operaie. Un exemplu simplu este atribuirea între variabile

```
a = b
```

Majoritatea platformelor actuale nu permit instruciuni în care ambii operanzi să fie alei din memorie. Deci o implentare standard pentru această operaie 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 atenie suplimentară atunci când un semnal folosete variabile care nu sunt locale funciei, deoarece semnalele pot întrerupe fluxul de execuie 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 execuie care interacionează cu handlerele de semnale. Acest tip este unul din tipurile întregi disponibile, dar care anume este poate varia de la o platformă la alta. Aadar operaiile ce se pot efectua cu acest tip, sunt aceleai 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 funciilor de tip timer folosesc semnale.

Un timer este, de obicei, un întreg a cărui valoare este decrementată pe parcursul trecerii timpului. În momentul în care întregul ajunge la 0, timer-ul expiră. În Linux, rezultatul expirării timer-ului, este, 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 i configurarea handler-ului care va rula. Fiind vorba de semnale, configurarea handler-ului se realizează prin intermediul funciei sigaction. Specificarea unui interval de timeout se realiză folosind funcia alarm sau setitimer.

Apelul alarm este mai simplu, dar oferă granularitate la nivel de secundă. La expirarea timeout-ului, se transmite semnalul SIGALRM.

Apelul setitimer este mai complex, dar oferă granularitate la nivel de microsecundă i permite control mai bun asupra modului de actualizare a întregului. 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 i kernel-space. În funcie de primul argument se vor livra semnalele SIGALRM, SIGVTALRM respectiv SIGPROF.

Mai jos este prezentată o secvenă de cod de folosire a funciilor alarm i setitimer

```
#include <unistd.h>       /* for alarm(2) */
#include <sys/time.h>     /* for setitimer(2) */

#define TIMEOUT     5

    /* setup SIGALRM handler */
    ...

    alarm(TIMEOUT);       /* SIGALRM is delivered every TIMEOUT seconds */
    ....

    struct itimerval timer;

    memset(&timer, 0, sizeof(timer));
    timer.it_value.tv_sec = TIMEOUT;      /* first delivery after TIMEOUT seconds */
    timer.it_interval.tv_sec = TIMEOUT;   /* periodic delivery after TIMEOUT seconds */

    if (setitimer(ITIMER_REAL, &timer, NULL) < 0) {
         /* handle error */
         ...
   }
```

Atenie: Secvena de mai sus are rol academic. Nu este recomandată combinarea apelurilor alarm i setitimer. Standardul POSIX nu specifică modul de interaciune între cele două funcii iar efectele nu sunt deterministe.

Una din formele de utilizare a timerelor este implementarea funciilor de ateptare de tipul sleep sau nanosleep. Avantajul folosirii funciei sleep este simplitatea. Dezavantajele sunt rezoluia scăzută (secunde) i posibila interaciune cu semnale (în special `SIGALRM`). nanosleep are un apel mai complex, dar oferă rezoluie până la ordinul nanosecundelor i este "signal-safe" - nu interacionează cu semnale.

Semnale i 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 execuie asincronă în contextul unui proces/thread. Pot fi folosite pentru rularea unei secvene de cod în combinaie cu un timer.

În seciunile de mai jos vor fi prezentate funciile de interaciune cu evenimentele venite de la tastatură i funciile de lucru cu timere.

Evenimente de la consolă

API-ul Win32 permite configurarea de funcii care să fie apelate în momentul apăsării unei anumite combinaii de taste. Aceste funcii sunt similare cu handler-ele de semnale `SIGINT` sau `SIGQUIT` în Linux. În afara combinaiilor de taste un proces poate primi i semnale de forma `CTRL_CLOSE_EVENT`, `CTRL_LOGOFF_EVENT` sau `CTRL_SHUTDOWN_EVENT`.

Tipul de date PHANDLER_ROUTINE definete rutina apelată în momentul recepionării unui mesaj. Parametrul funciei descrie tipul semnalului primit. Rutina este similară handler-ului de semnal din Linux.

Înregistrarea unei rutine de tratare a semnalului se realizează cu ajutorul apelului SetConsoleCtrlHandler. Spre deosebire de apelul sigaction, funcia înregistrată va fi rulată în momentul generării tuturor semnalelor precizate mai sus.

Exceptând modul "clasic" de transmitere a semnalelor prin intermediul tastaturii, se pot genera semnale folosind apelul GenerateConsoleCtrlEvent.

Mai jos este o secvenă uzuală de înregistrare a unui handler pentru tratarea semnalelor de tastatură:

```
#include <windows.h>

static BOOL CtrlHandler(DWORD eventType)
{
    switch (eventType) {
    case CTRL_C_EVENT:
        [...]
        break;
    case CTRL_BREAK_EVENT:
        [...]
        break;
    default:
        [...]
        break;
    }

    return TRUE;
}

int main(void)
{
    [...]
    if (SetConsoleCtrlHandler((PHANDLER_ROUTINE) CtrlHandler, TRUE) == FALSE) {
        fprintf(stderr, "SetConsoleCtrlHandler failed with %d\n", GetLastError());
        exit(-1);
    }
    [...]

    return 0;
}
```

Waitable Timer Objects

Un obiect de tipul waitable timer este un obiect de sincronizare a cărui starea este configurată ca 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 funciei SetWaitableTimer setează un nou timp de ateptare;
- timer cu resetare automată - un timer a cărui stare rămâne signaled până când un thread efectuează o operaie de ateptare 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. Operaiile care se pot efectua cu un timer sunt:

- crearea unui timer: se realizează prin intermediul funciei CreateWaitableTimer
- setarea unui timer: cu ajutorul funciei SetWaitableTimer
- anularea: cu ajutorul funciei CancelWaitableTimer
- ateptarea: folosind WaitForSingleObject

În secvena de cod de mai jos, se folosete un timer pentru afiarea 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);
    }

    printf("5 seconds timer expired\n");

    return 0;
}
```

Quiz

Pentru autoevaluare răspundei la întrebările din acest quiz.

Exerciii

Prezentare

Pentru a urmări mai uor noiunile expuse la începutul laboratorului folosii această prezentare. odp|pdf

Exerciii de laborator

Folosii arhiva de sarcini a laboratorului.

Linux

- **Atenie**: La folosirea sigaction, vei iniializa, în general, câmpul `sa_flags` al structurii `struct sigaction` la 0. Iniializarea la `SA_RESETHAND` restaurează comportamentul implicit (ignorare, terminarea procesului etc.)

1. **(2 puncte)** Intrai în directorul `1-nohup`.
   - ♦ Realizai un program denumit `mynohup` care simulează comanda nohup.
   - ♦ Programul primete 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 întiinat de închiderea terminalului la care era conectat.
   - ♦ Analizai coninutul fiierului `mynohup.c`.
     Hint:
     - ◊ Pentru testare procesul trebuie rulat în background `./mynohup command args &`
     - ◊ Va trebui să ignorai semnalul `SIGHUP` livrat de shell procesului în momentul încheierii sesiunii curente.
   - ♦ Dacă fiierul standard de ieire era legat la un terminal acesta trebuie redirectat în fiierul `NOHUP_OUT_FILE`.
     Hint:
     - ◊ Folosii apelul isatty.
   - ♦ Pentru testare rulai `./mynohup sleep 100 &`.
     - ◊ După rulare închidei sesiunea de shell curentă.
     - ◊ Dintr-o altă consolă rulai `ps -ef | grep sleep`. Cine este noul părinte al procesului?
     - ◊ Consultai seciunea Tratarea semnalelor i seciunile Înlocuirea imaginii unui proces i Redirectări în Linux din laboratoarele precedente.

2. **(1.5 puncte)** Intrai în directorul `2-zombie`.
   - ♦ Urmării coninutul fiierelor `mkzombie.c` i `nozombie.c`.
   - ♦ Realizai două programe denumite `mkzombie` i `nozombie`.
   - ♦ Fiecare program va crea câte un proces copil nou care se va termina imediat după rulare apelând exit.
   - ♦ `mkzombie` va atepta `TIMEOUT` secunde i va iei.
     - ◊ Din altă consolă rulai `ps -eF | grep zombie`.
     - ◊ Observai faptul că procesul copil, dei nu mai rulează, apare în lista de procese ca `<defunct>` i are un pid (unic în sistem la acel moment).
     - ◊ Observai că după moartea procesului părinte dispare i procesul zombie.
   - ♦ `nozombie` va atepta `TIMEOUT` secunde i va iei.
     - ◊ Implementai, fără a folosi funciile de ateptare de tipul wait, `nozombie` astfel încât procesul său copil să nu treacă în starea de zombie.

       Hints:

       · Folosii semnalul `SIGCHLD`. Informaii găsii în sigaction(2) i wait(2).

       Hints:
     - ◊ Consultai seciunea Tratarea semnalelor i Crearea unui proces.
3. **(1.5 puncte)** Intrai în directorul `3-askexit`.
   - ♦ Urmării coninutul fiierului `askexit.c`.
   - ♦ Realizai un program denumit `askexit` care face busy waiting afiând la consolă numere consecutive.
   - ♦ Programul va intercepta semnalele generate de `CTRL+\`, `CTRL+C` i `SIGUSR1` (folosii comanda `kill`).
   - ♦ Pentru fiecare semnal primit va întreba utilizatorul dacă dorete să încheie execuia sau nu.

     Hints:
     - ◊ Dei funciile `printf`, `scanf`, etc. nu trebuie folosite în handlere de semnale, le putei folosi în cadrul acestui exerciiu pentru a nu complica inutil problema.
     - ◊ Consultai seciunea Tratarea semnalelor.
4. **(1.5 puncte)** Intrai în directorul `4-timer`.
   - ♦ Urmării coninutul fiierului `mytimer.c`.
   - ♦ Folosii setitimer pentru a afia timpul curent la fiecare `TIMEOUT` secunde.

     Hints:
     - ◊ Completai funciile din fiierul sursă.
     - ◊ Folosii ctime i time pentru afiarea timpului curent.
       · ctime adaugă un caracter new-line (`\n`) la sfâritul irului întors.
     - ◊ Folosii sigsupend pentru a atepta semnale.
       · Obinei, folosind sigprocmask, masca procesului curent i transmitei-o ca argument către sigsupend.
     - ◊ Urmării seciunile Timere în Linux i Ateptarea unui semnal

Windows

Folosii header-ul `../util/util.h` i funcia de tratare a unei erori (`doError`).

1. **(1 puncte)** Intrai în directorul `1-ignore`.
   - ♦ Urmării coninutul fiierului `myignore.c`.
   - ♦ Configurai procesul curent să ignore închiderea consolei (semnalul `CTRL_CLOSE_EVENT`).
   - ♦ Rulai programul i închidei consola. Ce mesaj primii?

     Hints:
     - ◊ Consultai seciunea Evenimente de la consolă.

2. **(1.5 puncte)** Intrai în directorul `2-askexit`.
    - ♦ Urmării coninutul fiierului `askexit.c`.
    - ♦ Realizai un program denumit care face face sleep la nivel de secundă i afiează pe ecran numere în ordine crescătoare.
    - ♦ Programul va intercepta semnalele generate de CTRL+C, CTRL+Break.
    - ♦ Pentru fiecare semnal primit va întreba utilizatorul dacă dorete să încheie execuia sau nu.
        Hints:
        - ◊ Consultai seciunea Evenimente de la consolă.
3. **(2 puncte)** Intrai în directorul `3-timer`.
    - ♦ Urmării coninutul fiierului `mytimer.c`.
    - ♦ Realizai un program care afiează data curentă la fiecare `TIMEOUT` secunde.
        Hints:
        - ◊ Folosii componenta `QuadPart` a tipului `LARGE_INTEGER` pentru specificarea timeoutului. Timeoutul este negativ i expiră la atingerea valorii 0.
        - ◊ Al treilea argument al SetWaitableTimerObject este timpul (în milisecunde) după care se va livra primul semnal de timer.
    - ♦ Folosii un handler APC pentru tratarea timer-ului i afiarea mesajului:
        Hints:
        - ◊ Folosii exemplul de utilizare a timer-elor cu APC-uri din documentaia MSDN.
        - ◊ Ignorai warning-urile de compilare.

        Hints:
        - ◊ Completai funciile din fiierul sursă.
        - ◊ Folosii ctime i time pentru afiarea timpului curent.
            - · ctime adaugă un caracter new-line (`\n`) la sfâritul irului întors.
        - ◊ Folosii SleepEx i argumentele `INFINITE` pentru a atepta nedefinit i `TRUE` pentru a fora intrarea procesului într-o stare alertabilă (care să declaneze rularea APC-ului).
        - ◊ Urmării seciunea Waitable Timer Objects.

Soluii

- Soluii exerciii laborator 6

Resurse utile

Note

1. ^ pagina gnu despre semnale aici
2. ^ mai multe informaii despre threaduri i semnale aici