1. Sygnały

1 INFORMACJE WSTĘPNE

Sygnał – mechanizm asynchronicznego powiadamiania procesów o zdarzeniach – zwykle awaryjnych.

Metoda powiadamiania procesów za pomocą sygnałów wywodzi się z systemu UNIX.

Sygnały mogą być generowane przez:

1. System operacyjny, zwykle po wykonaniu nieprawidłowej operacji.
2. Z konsoli operatorskiej poprzez polecenia **`kill`** i **`slay`**.
3. Z programu aplikacyjnego poprzez funkcje (np. **`kill, raise, abort, alarm`**, i inne) oraz timery.

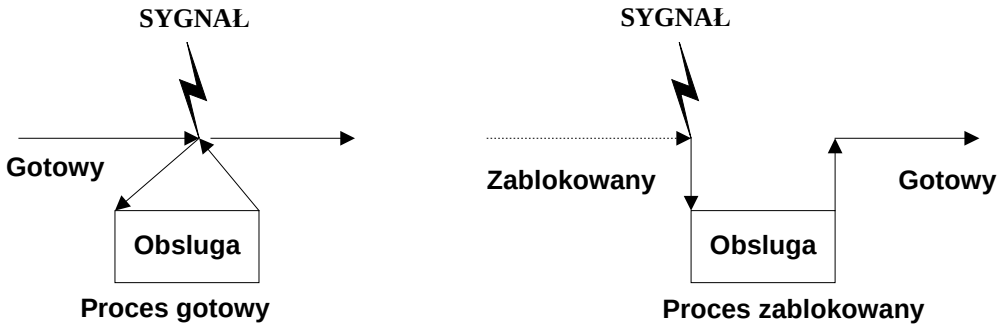
Proces może zareagować na sygnały w sposób następujący:

1. Obsłużyć sygnał czyli wykonać funkcję dostarczoną poprzez programistę.
2. Zignorować sygnał – nie każdy sygnał daje się zignorować.
3. Zablokować sygnał to znaczy odłożyć jego obsługę na później.
4. Zakończyć się po otrzymaniu sygnału.

Reakcja procesu na sygnał w zależności od stanu w jakim znajduje się proces.

1. Gdy proces jest wykonywany lub gotowy to następuje przerwanie sekwencji wykonania i skok do procedury obsługi sygnału.
2. Gdy proces jest zablokowany to następuje jego odblokowanie i wykonanie procedury obsługi tego sygnału.

PDF created with pdfFactory trial version www.pdffactory.com



Rysunek 1-1 Obsługa sygnału dla przypadków gdy proces jest gotowy i zablokowany

2 STANDARDY OBSŁUGI SYGNAŁÓW

Sygnały były już implementowane w pierwszych wersjach UNIX'a. Od tego czasu standard ewoluuje. Omawiane standardy:

1. Standardowa system UNIX
2. POSIX 1003.1
3. System QNX6 Neutrino

Standardowa, pochodząca z systemu Unix, specyfikacja sygnałów zakłada że:

- Sygnały nie niosą oprócz swego numeru żadnej wartości,
- Nie są kolejkowane,
- Nie uwzględniają istnienia wątków
- Mogą być przesyłane tylko w obrębie lokalnego węzła sieci.
- Nie posiadają priorytetów
- Nie są przenoszone poprzez sieć

Sygnały POSIX 1003.1b posiadają dodatkowe rozszerzenia:

- są kolejkowane
- oprócz numeru niosą dane: 8 bitowy kod i 32 bitową wartość
- uwzględniają istnienie wątków
- posiadają priorytety

PDF created with pdfFactory trial version www.pdffactory.com

	Unix	POSIX	Neutrino
Wysyłanie sygnału	**kill(), raise()**	**sigqueue(), pthread_kill()**	**SignalKill()**
Instalacja handlera	**signal()**	**sigaction()**	**SignalAction()**
Maskowanie sygnału	**sigblock(), sigunblock(), sigsetmask()**	**sigprocmask()**	**SignalProcmask()**
Oczekiwanie na sygnał	**pause(), sigpause()**	**sigsuspend(), sigwait(), sigtimedwait()**	**SignalSuspend(), SignalWaitinfo()**
Ustawienie alarmu	**alarm(), ualarm()**	**alarm(), ualarm()**	**TimerAlarm()**

Tabela 1-1 Ważniejsze funkcje obsługi sygnałów

Zakresy numerów poszczególnych grup sygnałów:

Zakres sygnałów	Opis
1 ... 40	Sygnały zdefiniowane w specyfikacji POSIX 1003 (także standardowe sygnały systemu Unix)
41 ... 56	16 sygnałów zdefiniowanych w rozszerzeniu specyfikacji POSIX dla systemów czasu rzeczywistego
57 ... 64	8 sygnałów specjalnych systemu QNX6 Neutrino

Tabela 1-2 Zakresy sygnałów

- Sygnały Neutrino oprócz numeru, niosą dodatkowe dane: 8 bitowy kod i 32 bitową wartość. Uwzględniają istnienie wątków.
- Dostarczanie sygnałów do procesów odbywa się zgodnie z priorytetami sygnałów. Sygnał a niższym numerze ma wyższy priorytet.

Uwaga!
Zaimplementowane w systemie QNX6 Neutrino sygnały mogą być dodatkowo przesyłane przez sieć.

PDF created with pdfFactory trial version www.pdffactory.com

3 OPIS NIEKTÓRYCH SYGNAŁÓW

Kategorie sygnałów:

- Zdarzenia generowane przez sprzęt – SIGPWR, SIGBUS
- Sygnał z konsoli do procesu w celu jego zakończenia – SIGINT, SIGQUIT
- Sygnały generowane przez błędy programu – SIGSEGV, SIGILL, SIGFPE
- Sygnały generowane przez system we/wy – SIGPIPE, SIGIO
- Proces potomny się zakończył – SIGCHLD
- Sygnały użytkownika – SIGUSR1, SIGUSR2, SIGSTOP, SIGCONT, SIGKILL
- Użytkownik się wylogował – SIGHUP
- Alarm czasomierza - SIGALRM

Sygnał	Opis sygnału
SIGABRT	Sygnał przerwania procesu (*ang. Abort*). Sygnał może być wygenerowany poprzez wykonanie funkcji abort w procesie bieżącym. Powoduje że proces przed zakończeniem zapisuje na dysku swój obraz (*ang. core dump*
SIGALRM	Sygnał alarmu *(ang. Alarm)* wskazujący że upłynął zadany czas. Generacja może być spowodowana poprzez wykonanie funkcji alarm lub czasomierze (*ang. Timers*).
SIGBUS	Sygnał wysyłany przez system operacyjny gdy ten stwierdzi błąd magistrali (*ang. Bus error*).
SIGCHLD	Przesyłany do procesu macierzystego gdy proces potomny (*ang. Child*) kończy się.
SIGSTOP	Powoduje że proces który otrzymał ten sygnał ulega zablokowaniu do czasu gdy nie otrzyma sygnału SIGCONT
SIGCONT	Powoduje wznowienie procesu zawieszonego sygnałem SIGCONT
SIGFPE	Generowany przez system gdy nastąpił błąd operacji zmiennoprzecinkowej (*ang. Floating point exception).*
SIGHUP	Generowany gdy następuje zamknięcie terminala (*ang. Hangup*). Sygnał otrzymują procesy dla których jest to terminal kontrolny.
SIGILL	Generowany gdy proces próbuje wykonać nielegalną instrukcję (*ang. Illegal instruction*).

PDF created with pdfFactory trial version www.pdffactory.com

SIGINT	Przerwanie procesu (*ang. Interrupt*). Sygnał wysyłany do wszystkich procesów związanych z danym terminalem gdy tam naciśnięto Ctrl+Break lub Ctrl+C.
SIGKILL	Sygnał wysyłany w celu zakończenia procesu. Nie może być przechwycony ani zignorowany.
SIGPIPE	Generowany przy próbie zapisu do łącza (*ang. Pipe*) lub gniazdka gdy proces odbiorcy zakończył się.
SIGPOLL	Sygnał generowany przez system gdy na otwarty plik stał się gotowy do zapisu lub odczytu.
SIGQUIT	Próba zakończenia procesu (*ang. Quit*). Sygnał wysyłany do wszystkich procesów związanych z danym terminalem gdy tam naciśnięto Ctrl+\.
SIGSEGV	Wysyłany przez system gdy proces naruszył mechanizm ochrony pamięci (*ang. Segment Violation)*
SIGTERM	Sygnał wysyłany w celu zakończenia procesu. Nie może być przechwycony ani zignorowany.
SIGPWR	Generowany przez system operacyjny gdy ten stwierdzi upadek zasilania (*ang. Power Failure*) sygnalizowany przez układ dozoru zasilania.
SIGUSR1	Sygnał może być wykorzystany przez użytkownika do własnych potrzeb.
SIGUSR2	Sygnał może być wykorzystany przez użytkownika do własnych potrzeb.

Tabela 1-3 Zestawienie ważniejszych sygnałów

PDF created with pdfFactory trial version www.pdffactory.com

4 WYSYŁANIE SYGNAŁÓW Z PROGRAMU

Funkcja kill

kill- wysłanie sygnału do procesu	
int kill(pid_t pid, int sig)	
pid	PID procesu do którego wysyłany jest sygnał
sig	Numer sygnału.

Funkcja powoduje wysłanie sygnału **sig** do procesu **PID**.
Funkcja zwraca 0 gdy sukces, -1 gdy błąd.

Aby proces bieżący mógł wysłać sygnał do innego procesu muszą być spełniony jeden z warunków:

1. Efektywny identyfikator użytkownika EUID (*ang. Effective User ID*) procesu wysyłającego sygnał i procesu docelowego muszą być zgodne.
2. Rzeczywisty identyfikator użytkownika UID (*ang. User ID*) procesu wysyłającego sygnał i procesu docelowego muszą być zgodne.
3. Proces wysyłający sygnał ma prawa administratora (*ang. root*).

Specjalne znaczenie parametru pid:

1. Gdy pid = 0 to sygnał będzie wysyłany do wszystkich procesów należących do tej samej grupy co nadawca.
2. Gdy pid < 0 to sygnał będzie wysyłany do wszystkich procesów należących do grupy o numerze id = |pid|.

PDF created with pdfFactory trial version www.pdffactory.com

Funkcja alarm

Funkcja **alarm** posiada następujący prototyp:

```
int alarm(int seconds)
```

seconds Liczba sekund do wysłania sygnału SIGALRM. Gdy 0 poprzednio ustawiony alarm jest kasowany.

Funkcja **alarm** powoduje wygenerowanie sygnału SIGALRM po upływie liczby sekund wyspecyfikowanej jako parametr. Sygnał wysyłany jest do procesu który funkcję wywołał.

Funkcja zwraca:
> 0 to wynik jest liczbą sekund pozostałych do wysłania sygnału przy wcześniejszym ustawieniu.
= 0 znaczy że alarm nie był wcześniej ustawiany
- 1 Błąd

Funkcja ualarm

Umożliwia generowanie sygnałów z mikrosekundową rozdzielczością (alarm z sekundową).

```
int ualarm(int usecs, int interval);
```

seconds Liczba mikrosekund do wysłania sygnału SIGALRM. Gdy 0 poprzednio ustawiony alarm jest kasowany.
interval Gdy > 0 jest to okres repetycji sygnału w mikrosekundach

Funkcja zwraca:
> 0 liczba mikrosekund pozostałych do wysłania sygnału.
= 0 znaczy że alarm nie był wcześniej ustawiany
- 1 Błąd

Uwaga

Do generowania sygnałów używa się także czasomierzy (ang. *timer*).

PDF created with pdfFactory trial version www.pdffactory.com

5 WYSYŁANIE SYGNAŁU Z KONSOLI

Do wysłania sygnału z konsoli użyć można polecenia **`kill`** lub **`slay`**.

Polecenie kill

Polecenie **`kill`** ma postać:

```
kill  [-nazwa_sygnału | -numer_sygnału] pid
```

`pid`	PID procesu do którego wysyłany jest sygnał
`numer_sygnału`	Numeryczne określenie sygnału
`nazwa_sygnału`	Symboliczne określenie sygnału – może być uzyskane przez polecenie: kill –l

Przykład:

```
kill – SGUSR1  211
```

Uwagi:

1. Gdy **`pid = 0`** to sygnał będzie wysyłany do wszystkich procesów należących do tej samej grupy co użytkownik.
2. Gdy **`pid < 0`** to sygnał będzie wysyłany do wszystkich procesów należących do grupy o numerze **`id = |pid|.`**

Polecenie slay

Polecenie slay umożliwia wysłanie sygnału do procesu bez znajomości jego PID. Jako parametr podaje się nazwę procesu.

```
slay  [-numer_sygnału] nazwa
```

`nazwa`	nazwa procesu do którego wysyłany jest sygnał
`numer_sygnału`	Numeryczne określenie sygnału – domyślnie SIGTERM

Przykład:

```
slay mój_proces
```

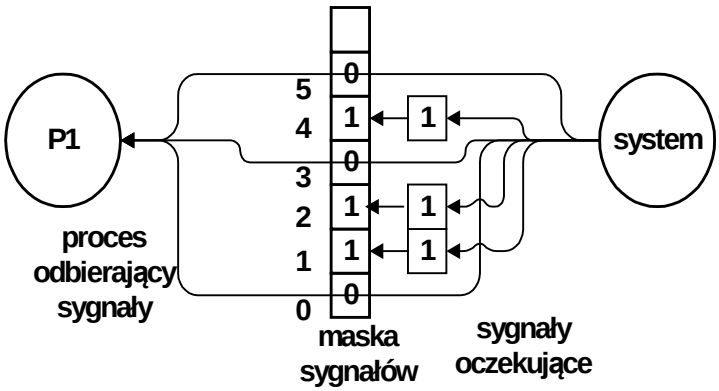
W Linuksie stosowane jest podobne polecenie **`pkill.`**

PDF created with pdfFactory trial version www.pdffactory.com

6 MASKOWANIE SYGNAŁÓW

W kodzie programu mogą istnieć sekcje które nie powinny być przerywane sygnałami. Stąd w systemie istnieją funkcje służące do blokowania sygnałów.

- Zablokowany sygnał jest pamiętany - może on być obsłużony gdy zostanie odblokowany.
- Standardowo tylko jeden nie obsłużony sygnał może być pamiętany ale sygnały mogą być kolejkowane gdy ustawiona jest flaga **`SA_SIGINFO`** (funkcja **`sigaction()`**)



Rys. 1-1 Maska sygnałów blokuje dostarczanie sygnałów do procesu P1

Maskowanie sygnałów wymaga operowania na zbiorach. W pliku **`<signal.h>`** zdefiniowany został typ **`sigset_t`**.

31	30	29		2	1	0
1	0	1	....	1	0	0

Maska sygnałów

Funkcje operujące na zbiorach sygnałów:

Inicjowanie pustego zbiorów sygnałów set:
`int sigemptyset(sigset_t *set)`

Inicjowanie pełnego zbiorów sygnałów set:
`int sigfillset(sigset_t *set)`

Dodanie nowego sygnału signo do zbioru **`set:`**

PDF created with pdfFactory trial version www.pdffactory.com

int sigaddset(sigset_t *set, int signo)

Usuniecie sygnału **signo** ze zbioru **set:**
int sigdelset(sigset_t *set, int signo)

Testowanie czy sygnał **signo** należy do zbioru **set:**
int sigismember(sigset_t *set, int signo)

Funkcja sigprocmask (POSIX)

W sekcjach krytycznych programu sygnały można zablokować.

Ustawianie i testowanie maski sygnałów - **sigprocmask**

int sigprocmask(int how, sigset_t *set, sigset_t *oset)	
how	**SIG_SETMASK** – blokowanie sygnałów ze zbioru **set** **SIG_UNBLOCK** – odblokowanie sygnałów ze zbioru **set**
set	Zbiór sygnałów
oset	Poprzedni zbiór sygnałów

```
....
sigset_t set1;
sigfillset(&set1);
sigprocmask(SIG_SETMASK,&set1,NULL);
 Sekcja Krytyczna
sigprocmask(SIG_UNBLOCK,&set1,NULL);
....
```

Przykład 1-1 Blokowanie sygnałów w sekcji krytycznej

Zablokowane sygnały pozostają jako oczekujące.

PDF created with pdfFactory trial version www.pdffactory.com

7 OBSŁUGA SYGNAŁÓW

Ustalenie reakcji procesu na sygnał odbywa się za pomocą funkcji **`signal (UNIX)`**. Ma ona następujący prototyp:

```
void(*signal(int sig, void(*func)(int)))(int)
```

`sig` Numer lub symbol sygnału który ma być obsłużony
`func` Nazwa funkcji która ma być wykonana gdy proces odbierze sygnał **`sig`**.

Możliwe są trzy typy akcji podejmowanych w reakcji na nadejście sygnału:

1. Zignorowanie sygnału
2. Wykonanie akcji domyślnej - działanie określone przez OS – zwykle zakończenie procesu.
3. Wykonanie funkcji dostarczonej przez programistę.

Nie jest możliwe obsłużenie sygnałów:

- SIGSTOP
- SIGKILL

Funkcja obsługi sygnału powinna być zdefiniowana w programie. Funkcja zwraca wskaźnik na poprzednią funkcję obsługi sygnału. Istnieją dwie pierwotnie zdefiniowane funkcje obsługi sygnałów:

`SIG_IGN` Funkcja powodująca zignorowanie sygnału.
`SIG_DFL` Domyślna reakcja na sygnał - zakończenie procesu lub zignorowanie sygnału.

PDF created with pdfFactory trial version www.pdffactory.com

```
#include <signal.h>
#include <stdlib.h>
#include <setjmp.h>
int sigcnt = 0;
int sig = 0;

void sighandler(int signum) { // Procedura obsługi
sygnału
  sigcnt++;
  sig = signum;
}

main() {
  int i;  i = 0;
  printf("Start programu \n");
  signal(SIGINT, sighandler);
  do {
     printf(" %d %d %d  \n",i,sigcnt,sig);
     sleep(1);
     i++;
   } while(1);
}
```

Przykład 1-2 Program obsługujący sygnał SIGINT

```
#include <stdio.h>
#include <signal.h>
int time_out;

void time_sig(int signo) {
   time_out = 1;
}

main() {
 char passwd[16];
 signal(SIGALRM,time_sig);
 for(i=0;i<5;i++) {
     time_out = 0;
     printf("Podaj haslo:\n");
     alarm(5);
     gets(passwd);
     alarm(0);
     if(  time_out == 0) break;
 }
 ....
}
```

Przykład 1-3 Wykorzystanie sygnału do ustalenia limitu czasowego na podanie hasła

PDF created with pdfFactory trial version www.pdffactory.com

Funkcja `pause (UNIX).`

Funkcja **`pause`** powoduje zablokowanie procesu aż do chwili nadejścia sygnału. Aby proces się nie zakończył sygnał musi być obsługiwany. Prototyp funkcji **`pause`** jest następujący:

```
int pause(void)
```


PDF created with pdfFactory trial version www.pdffactory.com

8 ODPORNY INTERFEJS SYGNAŁOWY (POSIX)

Funkcje **sigaction** pozwala na lepsze kontrolowanie obsługi sygnału niż poprzedni interfejs.

Funkcja sigaction:

sigaction – definiowanie reakcji procesu na sygnał	
int sigaction(int signo, struct sigaction *act, struct sigaction *oldact)	
act	Definicja działania które ma być podjęte gdy przyjdzie sygnał.
oldact	Definicja poprzedniej akcji lub NULL

Funkcja **sigaction** definiuje sposób obsługi sygnału **signo**.

```
struct sigaction {
     void (*sa_handler)(int) ;
     void(*sa_sigaction)(int signo,siginfo_t *info,
          void *inne)
     sigset_t sa_mask;    // Sygnały blok. podczas obsługi
     int sa_flags;        // Flagi modyfikacji działania
}
```

Pole **sa_mask** definiuje sygnały blokowane podczas obsługi danego sygnału. Będą one zgłoszone później. Pole **sa_handler** może mieć jedną z wartości:

1. SIG_IGN – zignorowanie sygnału
2. SIG_DFL – obsługa domyslna
3. Adres handlera obsługi sygnału **void handler(int signo)**
4. Adres handlera obsługi sygnału **void handler(int signo, siginfo_t *info, void *)**

Postać funkcji obsługi zależy od flagi SA_SIGINFO ustawianej w zmiennej **sa_flags** struktury typu **sigaction**.

Flaga wyzerowana - funkcja przyjmuje formę **sa_handler**,
Flaga ustawiona - funkcja przyjmuje formę **sa_sigaction**

PDF created with pdfFactory trial version www.pdffactory.com

```
typedef struct siginfo {
        int si_signo;
        int si_code;
        union sigval si_value;
        ...
} siginfo_t;

typedef union sigval {
        int     sival_int;
        void    *sival_ptr;
} sigval_t;
```

Pola struktury **siginfo**:

si_signo	Numer sygnału
si_code	Kod sygnału
si_value	Wartość sygnału

PDF created with pdfFactory trial version www.pdffactory.com

```
// Wysylanie i odbior sygnalow  POSIX
// Sygnaly kolejkowane
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

static int count, numer, kod, value = 0;

void handler( int signo, siginfo_t *info, void * o )   {
    count++;
    numer = signo;
    kod   = info->si_code;
    value = info->si_value.sival_int;
}

int main( int argc, char * argv[] )   {
  int i,pid,kod,wart;
  union sigval sig;
  sigset_t set;
  struct sigaction act;
  sigemptyset( &set );
  sigaddset( &set, SIGUSR1 );
  act.sa_flags = SA_SIGINFO;
  act.sa_mask = set;
  act.sa_handler = &handler;

  sigaction( SIGUSR1, &act, NULL );

  if((pid = fork()) == 0) { // Potomny
     pid = getppid();
     for(i=0;i<10;i++) {
         printf("Wysylam: sygnal= %d \n", i);
         // Wartość sygnalu
         sig.sival_int = i;
         sigqueue(pid, SIGUSR1, sig);
         sleep(1);
     }
     exit(0);
  }
  // Macierzysty -----------
  while(1) {
    printf("Odbior: licz= %d num= %d kod= %d val= %d\n",
        count,numer,kod,value);
    sleep(1);
    if(value == 9) break;
  }
  return EXIT_SUCCESS;
}
```

Przykład 1-4 Obsługi sygnałów przy pomocy funkcji **`sigaction`**

PDF created with pdfFactory trial version www.pdffactory.com

```
Wysylam: sygnal= 0
Odbior: licznik= 1 numer= 16 kod= -1  val= 0
Wysylam: sygnal= 1
Odbior: licznik= 2 numer= 16 kod= -1  val= 1
...
```

Wyniki poprzedniego przykładu

PDF created with pdfFactory trial version www.pdffactory.com

9 UWAGI O OBSŁUDZE SYGNAŁÓW.

1. Blokada sygnałów
Podczas obsługi sygnału dostarczanie innych sygnałów jest zablokowane.

2. Sygnały i komunikaty.
Gdy proces jest zablokowany na funkcji **`MsgSend`** lub **`MsgReceive`** reakcja na sygnał jest następująca:

- Proces jest odblokowywany
- Sygnał jest obsługiwany
- Funkcja **`MsgSend`** lub **`MsgReceive`** kończy się błędem: kod powrotu –1 i zmienna **`errno`** = EINTR.

3. Sygnały i funkcje systemowe
W większości przypadków w czasie wykonania funkcji systemowych sygnały są zablokowane. Wyjątek stanowią:

- Funkcje **`read, write, open`** w odniesieniu do terminali.
- Funkcje **`wait, pause, sigsuspend`**

Funkcje te będą przerywane przez sygnał. Możliwe jest ustawienie flagi SA_RESTART aby przerwane funkcje kontynuować.

4. Kolejkowanie sygnałów
Sygnały UNIX nie są kolejkowane. Sygnały POSIX mogą być kolejkowane.

PDF created with pdfFactory trial version www.pdffactory.com

10 SYGNAŁY A WĄTKI

Specyfikacja sygnałów POSIX definiuje ich działanie tylko dla procesów jednowątkowych.
Sygnały mogą być kierowane do procesów i do wątków.

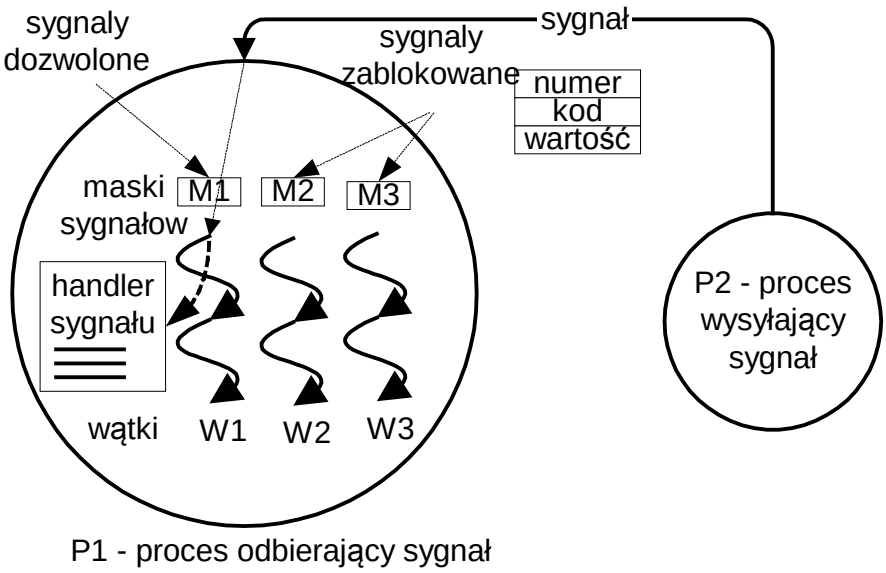
Zachowanie się sygnałów w środowisku procesów wielowątkowych zdefiniowane jest regułami:

1. Sygnały obsługiwane są na poziomie procesu. Znaczy to że gdy wątek zignoruje lub obsłuży sygnał, fakt ten wpływa na inne wątki tego procesu.
2. Maskowanie sygnałów zachodzi na poziomie wątków.
3. Jeżeli sygnał skierowany jest do określonego wątku to będzie on do tego wątku dostarczony.
4. Jeżeli sygnał skierowany jest do procesu to będzie dostarczony do pierwszego wątku który nie blokuje danego sygnału.

Zasada obsługi sygnałów w środowiskach wielowątkowych

Standardową strategią obsługi sygnałów w środowisku procesów wielowątkowych jest zamaskowanie sygnałów we wszystkich watkach z wyjątkiem jednego. Ten właśnie wątek będzie obsługiwał sygnały.

PDF created with pdfFactory trial version www.pdffactory.com



Rysunek 1-2 Obsługa sygnału dla procesów wielowątkowych

Sygnał dostarczany jest do procesu P1. W ramach tego procesu wykonują się wątki W1, W2 i W3. Maski M2 i M3 blokują dochodzący sygnał a maska M1 go nie blokuje. Sygnał dochodzi do wątku W1 w ramach którego wykonywany jest handler.

PDF created with pdfFactory trial version www.pdffactory.com

11 ZABEZPIECZANIE OPERACJI BLOKUJĄCYCH

☞ O ile działanie aplikacji uzależnione jest od działania zewnętrznych względem niej procesów to powinna być ona zabezpieczona przez błędnym działaniem tych procesów.

Przeterminowanie operacji (*ang. timeout*)

Zabezpieczenie powodującej zablokowanie procesu operacji polega na ustanowieniu limitu czasowego na wykonanie takiej operacji. Gdy w zadanym okresie operacja nie zakończy się sama, ulega ona przeterminowaniu i jest przerywana.

Do przerywania powodujących zablokowanie operacji mogą być użyte sygnały.
Do generowania sygnałów po zadanym czasie wykorzystuje się funkcje:

- **`alarm`**,
- **`ualarm`**,
- timery

`alarm` - wysłanie sygnału alarmu do procesu	
`int alarm(int seconds)`	
`seconds`	Liczba sekund do wysłania sygnału **`SIGALRM`**. Gdy parametr ustawiony jest na 0 poprzednio ustawiony alarm jest kasowany.

`ualarm` - wysłanie sygnału alarmu do procesu	
`ualarm(useconds_t usec, useconds_t interval)`	
`usec`	Liczba mikrosekund do wysłania sygnału **`SIGALRM`**. 0 kasuje alarm
`interval`	Czas repetycji alarmu (0 gdy brak)

PDF created with pdfFactory trial version www.pdffactory.com

Aby zabezpieczyć operację blokującą sygnałem należy:

1. Napisać procedurę obsługi sygnału SIGALRM.
2. Zainstalować procedurę na obsługę sygnału SIGALRM.
3. Przed wykonaniem operacji blokującej wykonać funkcję **`alarm(T), ualarm(T,interval)`** lub nastawić timer
4. Wykonać operację blokującą.
5. Testować kod powrotu funkcji realizującej operację blokującą i zmienną erno aby sprawdzić czy operacja blokująca została przerwana sygnałem.
6. Odwołać alarm poprzez wykonanie funkcji **`alarm(0)`**.

PDF created with pdfFactory trial version www.pdffactory.com

12 UZYSKIWANIE PRZETERMINOWANIA W PROGRAMOWANIU GNIAZDEK

Możliwość 1
Wykorzystać opcję gniazdka SO_RCVTIMEO – funkcja **setsockopt**

```
struct timeval tv;
tv.tv_sec = 0;
tv.tv_usec = 100000;

// Ustawianie opcji gniazdka
if (setsockopt(rcv_sock, SOL_SOCKET,
SO_RCVTIMEO,&tv,sizeof(tv)) < 0) {
    perror("Error");
}

// Odbior komunikatu ----------
rec = recvfrom(s,(char *) &msg, blen, 0,&adr_cli,
      &slen);
if(rec < 0)  perror("recfrom");
```

Przykład 1-5 Wykorzystanie opcji SO_RCVTIMEO gniazdka do uzyskania przeterminowania odbioru komunikatu

PDF created with pdfFactory trial version www.pdffactory.com

Możliwość 2
Wykorzystać funkcję select

```
int timeout_recvfrom (int sock, char *buf, int *length,
struct sockaddr_in *connection, int timeoutinseconds)
{
   fd_set socks;
   struct timeval t;
   int res;
   FD_ZERO(&socks);
   FD_SET(sock, &socks);
   t.tv_sec = timeoutinseconds;
   res = select(sock + 1,&socks, NULL, NULL, &t);
   if(res > 0) {
     res = recvfrom(sock,buf,*length,0,
     (struct sockaddr * connection, length);
     return res;
   } else
     return 0;
}
```

Przykład 1-6 Wykorzystanie funkcji select do uzyskania przeterminowania odbioru komunikatu

PDF created with pdfFactory trial version www.pdffactory.com

Możliwość 3
Wykorzystanie sygnałów i funkcji alarm

```
#define KROKI 10
#define PORT 9950
...
int sigcnt = 0;

void sighandler(int signum) {
   sigcnt++;
   printf("alarm %d\n",sigcnt);
}

int main(void) {
   struct sockaddr_in adr_moj, adr_cli;
   int s, i,res, slen=sizeof(adr_cli),snd, rec;
   int blen=sizeof(msgt);
   msgt msg;

   s=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
   if(s < 0) blad("socket");
   printf("Gniazdko %d utworzone\n",s);

   // Ustalenie adresu IP serwera
   memset((char *) &adr_moj, 0, sizeof(adr_moj));
   adr_moj.sin_family = AF_INET;
   adr_moj.sin_port = htons(PORT);
   adr_moj.sin_addr.s_addr = htonl(INADDR_ANY);
   if (bind(s, &adr_moj, sizeof(adr_moj))==-1)
       blad("bind");

   // Instalacja handlera sygnalu
   signal(SIGALRM,sighandler);
```


PDF created with pdfFactory trial version www.pdffactory.com

```
// Odbior komunikatow ------------
   for (i=0; i<KROKI; i++) {
      res = alarm(1);
      rec = recvfrom(s,(char *) &msg, blen, 0,
            &adr_cli, &slen);
      if(rec < 0) {
            perror("recfrom");
            if(errno == 4) { // timeout
            // Tutaj mozna umiescic rozne akcje
            // okresowe
               printf("Timeout: %d\n",sigcnt);
            }
            sleep(1);
            continue;
      }
      // Odpowiedz -----
      sprintf(msg.buf,"Odpowiedz %d",i);
      snd = sendto(s, &msg, blen, 0,
            &adr_cli, slen);
   }
   close(s);
   return 0;
}
```

Przykład 1-7 Szkic serwera UDP z przeterminowaniem uzyskanym przy wykorzystaniu sygnałów

Metoda ta jest skuteczna także dla innych przypadków:

- Łączy nienazwanych
- Łączy nazwanych
- Kolejek POSIX
- Gniazdek
- Komunikatów QNX
- Plikowych operacji we/wy

PDF created with pdfFactory trial version www.pdffactory.com