
Sygnaly i ich obsluga

1.	Sygnaly i ich obsluga	2
1.1	Informacje wstepne.....	2
1.2	Standardy obslugi sygnalów.....	3
1.3	Wysylanie sygnalów	6
1.4	Maskowanie sygnalów.....	10
1.5	Obsluga sygnalów	12
1.6	Testowanie sygnalów oczekujacych (POSIX)	17
1.7	Oczekiwanie na sygnal (POSIX)	18
1.8	Uwagi o obsludze sygnalów.	19
1.9	Sygnaly a watki.....	20
1.10	Zabezpieczanie operacji blokujacych.....	22

1. Sygnały i ich obsługa

1.1 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:

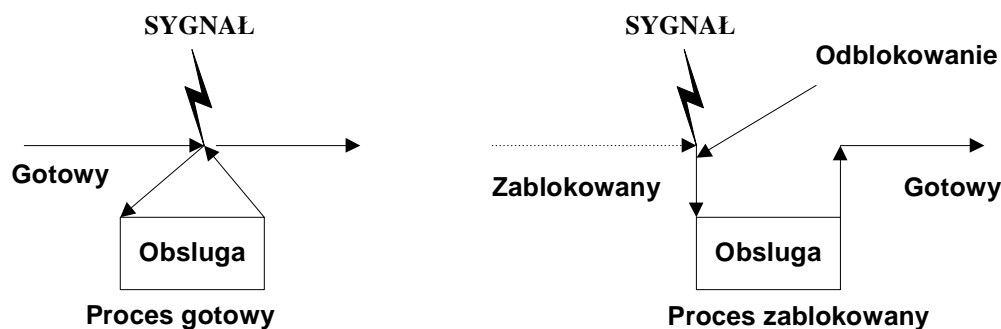
1. System operacyjny, zwykle po wykonaniu nieprawidłowej operacji.
2. Z konsoli operatorskiej poprzez polecenia kill i slay.
3. Z programu aplikacyjnego poprzez funkcje (np. kill, raise, abort, alarm, i inne) oraz timery.

Proces może zareagować na sygnały w sposób następujący:

1. Obsłużyć sygnał czyli wykonać funkcję dostarczoną przez 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.



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

1.2 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

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

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

Sygnały POSIX 1003.1b posiadają dodatkowe rozszerzenia:

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

	Unix	POSIX
Wysyłanie sygnału	<code>kill()</code> , <code>raise()</code>	<code>sigqueue()</code> , <code>pthread_kill()</code>
Instalacja handlera	<code>signal()</code>	<code>sigaction()</code>
Maskowanie sygnału	<code>sigblock()</code> , <code>sigunblock()</code> , <code>sigsetmask()</code>	<code>sigprocmask()</code>
Oczekiwanie na sygnał	<code>pause()</code> , <code>sigpause()</code>	<code>sigsuspend()</code> , <code>sigwait()</code> , <code>sigtimedwait()</code>
Ustawienie alarmu	<code>alarm()</code> , <code>ualarm()</code>	<code>alarm()</code> , <code>ualarm()</code>

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

Tabela 1-2 Zakresy sygnałów

- Sygnały POSIX 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.

Opis niektórych sygnałów

Sygnal	Opis sygnału	Akcja
SIGABRT	Sygnal przerwania procesu (<i>ang. Abort</i>). Sygnal może być wygenerowany poprzez wykonanie funkcji abort w procesie bieżącym. Powoduje że proces przed zakończeniem zapisuje na dysku swój obraz (<i>ang. core dump</i>)	ABRT, DMP
SIGALRM	Sygnal alarmu (<i>ang. Alarm</i>) wskazujący że upłynął zadany czas. Generacja może być spowodowana poprzez wykonanie funkcji alarm lub czasomierze (<i>ang. Timers</i>).	ABRT
SIGBUS	Sygnal wysyłany przez system operacyjny gdy ten stwierdzi błąd magistrali (<i>ang. Bus error</i>).	ABRT
SIGCHLD	Przesyłany do procesu macierzystego gdy proces potomny (<i>ang. Child</i>) kończy się.	IGN
SIGSTOP	Powoduje że proces który otrzymał ten sygnał ulega zablokowaniu do czasu gdy nie otrzyma sygnału SIGCONT	
SIGCONT	Powoduje wznowienie procesu zawieszonoego sygnałem SIGCONT	
SIGFPE	Generowany przez system gdy nastąpił błąd operacji zmiennoprzecinkowej (<i>ang. Floating point exception</i>).	ABRT, DMP
SIGHUP	Generowany gdy następuje zamknięcie terminala (<i>ang. Hangup</i>). Sygnal otrzymują procesy dla których jest to terminal kontrolny.	ABRT
SIGILL	Generowany gdy proces próbuje wykonać nielegalną instrukcję (<i>ang. Illegal instruction</i>).	ABRT
SIGINT	Przerwanie procesu (<i>ang. Interrupt</i>). Sygnal wysyłany do wszystkich procesów związanych z danym terminalem gdy tam naciśnięto Ctrl+Break lub Ctrl+C.	
SIGKILL	Sygnal wysyłany w celu zakończenia procesu. Nie może być przechwycony ani zignorowany.	ABRT
SIGPIPE	Generowany przy próbie zapisu do łącza (<i>ang. Pipe</i>) lub gniazdka gdy proces odbiorcy zakończył się.	ABRT
SIGPOLL	Sygnal generowany przez system gdy na otwarty plik stał się gotowy do zapisu lub odczytu. Lepiej użyć funkcji select	ABRT
SIGQUIT	Próba zakończenia procesu (<i>ang. Quit</i>). Sygnal wysyłany do wszystkich procesów związanych z danym terminalem gdy tam naciśnięto Ctrl+\.	ABRT, DMP
SIGSEGV	Wysyłany przez system gdy proces naruszył mechanizm ochrony pamięci (<i>ang. Segment Violation</i>)	ABRT
SIGTRAP	Pułapka – używany w programach uruchomieniowych	
SIGTERM	Sygnal wysyłany w celu zakończenia procesu. Nie może być przechwycony ani zignorowany.	ABRT
SIGTSTP	Sygnal zatrzymania terminala (CTRL+Z). Może być przechwytywany i ignorowany.	
SIGPWR	Generowany przez system operacyjny gdy ten stwierdzi upadek zasilania (<i>ang. Power Failure</i>) sygnalizowany przez układ dozoru zasilania.	ABRT
SIGUSR1	Sygnal może być wykorzystany przez użytkownika do własnych potrzeb.	ABRT
SIGUSR2	Sygnal może być wykorzystany przez użytkownika do własnych potrzeb.	ABRT

Sygnały

Tabela 1-3 Zestawienie ważniejszych sygnałów

1.3 WYSYŁANIE SYGNAŁÓW

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 spełnione 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|.

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.
- = 0 znaczy że alarm nie był wcześniej ustawiany
- 1 Błąd

Funkcja ualarm

```
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*).

Funkcja `sigqueue` (POSIX)

`sigqueue` – wysłanie sygnału do procesu

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

pid	PID procesu w ramach którego wykonywany jest wątek docelowy
signo	Numer sygnału.
value	Wartość sygnału.

Funkcja powoduje wysłanie sygnału o numerze **signo** i wartości **value** do procesu o identyfikatorze **pid**. Wartość sygnału zawarta jest w polu **sival_int** unii **sigval** będącej trzecim parametrem tej funkcji.

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

Możliwe jest kolejkowanie nie obsłużonych sygnałów gdy ustawiona jest flaga **SA_SIGINFO** (funkcja **sigaction**). Funkcja zwraca 0 gdy sukces a -1 gdy błąd.

```
// Wysyłanie sygnałów
#include <signal.h>
....
int main( void ) {
    int i,pid,kod;
    union sigval sig;
    sigset_t set;
    pid = atoi(argv[1]);
    sig.sival_int = 10;
    while(1) {
        printf("Wysyłam sygnał= %d \n", i);
        //      PID      Numer      Wartość
        sigqueue(pid, SIGUSR1, sig);
        sleep(1);
    }
    return EXIT_SUCCESS;
}
```

Sygnały

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
```

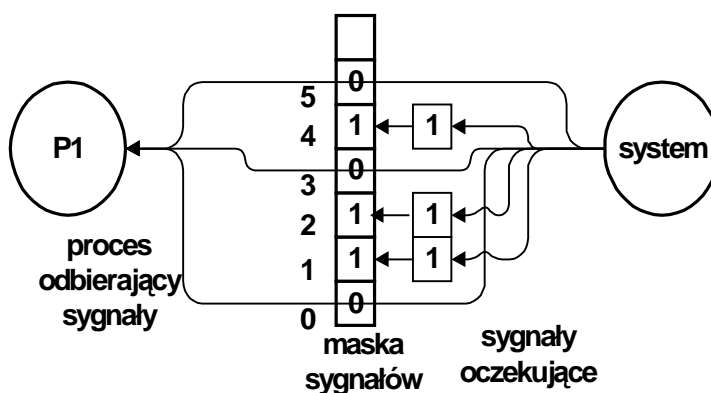
Uwagi:

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|`.

1.4 MASKOWANIE SYGNAŁÓW

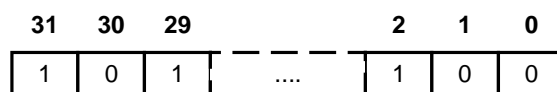
W kodzie programu mogą istnieć sekcje które nie powinny być przerywane sygnałami. Stąd w systemie istnieją funkcje służące do blokowania sygnałów.

- 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`.



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`:

Sygnały

```
int sigaddset(sigset_t *set, int signo)
```

Usunięcie 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)
```

<code>how</code>	<code>SIG_SETMASK</code> – blokowanie sygnałów ze zbioru <code>set</code> <code>SIG_UNBLOCK</code> – odblokowanie sygnałów ze zbioru <code>set</code> <code>SIG_BLOCK</code> – zablokowane sygnały są unią bieżąco zablokowanych sygnałów i wyspecyfikowanych w zbiorze <code>set</code>
<code>set</code>	Zbiór sygnałów
<code>oset</code>	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

1.5 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_DEF Domyślna reakcja na sygnał - zakończenie procesu lub zignorowanie sygnału.

```
#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

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)
```

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 domyślna
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`

```
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`:

<code>si_signo</code>	Numer sygnału
<code>si_code</code>	Kod sygnału
<code>si_value</code>	Wartość sygnału

Sygnały

```
// Wysyłanie i odbior sygnałów POSIX
// Sygnały kolejgowane
#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 = getpid();
        for(i=0;i<10;i++) {
            printf("Wysylam: sygnał= %d \n", i);
            // Wartość sygnału
            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**

Sygnały


```
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

1.6 TESTOWANIE SYGNAŁÓW OCZEKUJĄCYCH (POSIX)

```
int sigpending( sigset_t *set );
```

Funkcja zwraca zbiór sygnałów dostarczonych do procesu ale zablokowanych

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

void main() {
    sigset_t set, oset, pset;

    sigemptyset( &set );
    sigaddset( &set, SIGINT );
    // Ustawienie maski
    sigprocmask( SIG_BLOCK, &set, &oset );
    printf( "Old set was %8.8ld.\n", oset );
    // Sprawdzanie sygnałów oczekujących
    sigpending( &pset );
    printf( "Pending set is %8.8ld.\n", pset );

    kill( getpid(), SIGINT );

    sigpending( &pset );
    printf( "Pending set is %8.8ld.\n", pset );

    sigprocmask( SIG_UNBLOCK, &set, &oset );
    /* The program terminates with a SIGINT */
}
```

1.7 OCZEKIWANIE NA SYGNAŁ (POSIX)

```
int sigsuspend(sigset_t *sigmask)
```

sigmask Nowa maska sygnałów

Funkcja tymczasowo zastępuje bieżącą maskę sygnałów przez **sigmask** i oczekuje na sygnał. Następuje zablokowanie procesu. Gdy sygnał jest obsługiwany odblokowanie procesu następuje po zakończeniu obsługi sygnału i maska sygnału zostaje przywrócona do poprzedniej.

```
sigset_t    set;

void main() {
    sigemptyset( &set );
    sigaddset( &set, SIGINT );

    printf("Program suspended and immune to breaks.\n" );
    printf( "A SIGALRM will terminate the program"
           " in 10 seconds.\n" );
    alarm( 10 );
    sigsuspend( &set );
}
```

1.8 UWAGI O OBSŁUDZE SYGNAŁÓW.

1. Blokada sygnałów

Podczas obsługi sygnału dostarczanie innych sygnałów jest zablokowane.

2. 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.

3. Kolejkowanie sygnałów

Sygnały UNIX nie są kolejkowane. Sygnały POSIX mogą być kolejkowane.

1.9 SYGNAŁY A WĄTKI

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

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.

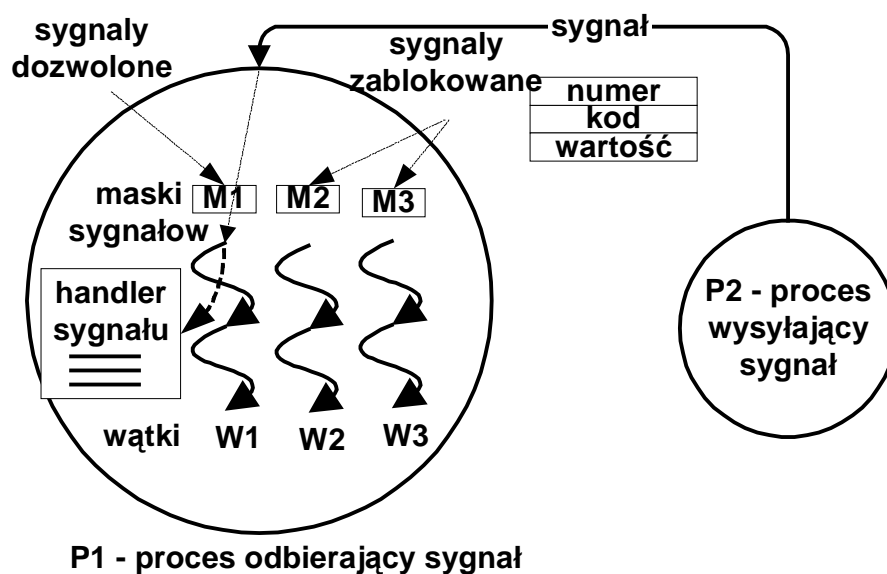
Zasada obsługi sygnałów w środowiskach wielowątkowych

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

Ustawianie maski sygnałów dla wątku

```
int pthread_sigmask(int how, const sigset_t
*set, sigset_t *oset)
```

how	SIG_SETMASK – blokowanie sygnałów ze zbioru set SIG_UNBLOCK – odblokowanie sygnałów ze zbioru set SIG_BLOCK – blokowane sygnały są unią bieżąco zablokowanych sygnałów i wyspecyfikowanych w zbiorze set
set	Zbiór sygnałów
oset	Poprzedni zbiór sygnałów



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.

1.10 ZABEZPIECZANIE OPERACJI BLOKUJĄCYCH

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

Przeterminowanie operacji (*ang. timeout*)

Zabezpieczenie powodujące 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

alarm(useconds_t usec, useconds_t interval)

usec	Liczba mikrosekund do wysłania sygnału SIGALRM . 0 kasuje alarm
-------------	------------------------------------------------------------------------

interval	Czas repetycji alarmu (0 gdy brak)
-----------------	------------------------------------

Sygnały

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 `errno` aby sprawdzić czy operacja blokująca została przerwana sygnałem.
6. Odwołać alarm poprzez wykonanie funkcji `alarm(0)`.

```
// Proces odbierający komunikaty - wysyła udp_cli
// Na odbiór komunikatów narzucony timeout
#include <arpa/inet.h>
#include <netinet/in.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <signal.h>
#include <errno.h>

#define BUFLen 80
#define KROKI 100
#define PORT 9950

typedef struct {
    int typ;
    char buf[BUFLen];
} msgt;

int sigcnt = 0;

void blad(char *s) {
    perror(s);
    exit(1);
}

void sighandler(int signum) {
    sigcnt++;
}
```

```
int main(void) {
    struct sockaddr_in adr_moj, adr_cli;
    int s, i, res, slen=sizeof(adr_cli),snd, rec,
    blen=sizeof(msgt);
    char buf[BUFLLEN];
    msgt msg;
    struct sigaction act;
    sigset_t set;

    gethostname(buf,sizeof(buf));
    printf("Host: %s\n",buf);

    // Instalacja handlera sygnalu
    sigemptyset( &set );
    sigaddset( &set, SIGALRM );
    act.sa_flags = 0;
    act.sa_mask = set;
    act.sa_handler = &sighandler;
    sigcnt = 0;

    sigaction( SIGALRM, &act, NULL );

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

    // Odbior komunikatow -----
    for (i=0; i<KROKI; i++) {
        // Ustawienie alarmu na 2 sek
        res = alarm(2);
        rec = recvfrom(s, &msg, blen, 0, &adr_cli, &slen);
        if(rec < 0) {
            // blad("recvfrom()");
            if(errno = EINTR) printf("timeout %d\n",sigcnt);
            else printf("err: %d\n",errno);
        } else {
            // skasowanie alarmu
            alarm(0);
            printf("Odebr kom z %s:%d res %d typ: %d %s\n",
                inet_ntoa(adr_cli.sin_addr),
                ntohs(adr_cli.sin_port), rec,msg.typ,msg.buf);
        }
    }
}
```

Sygnaly


```
        // Odpowiedz -----
        sprintf(msg.buf,"Odpowiedz %d",i);
        snd = sendto(s, &msg, blen, 0, &adr_cli, slen);
        if(snd < 0) blad("sendto()");
    }
}
close(s);
return 0;
}
```

Przykład 1-5 Implementacja przeterminowania operacji blokującej