

3 Sygnały

3.1 Opis sygnałów

Najprostszą metodą komunikacji międzyprocesowej w systemie UNIX są **sygnały**. Umożliwiają one *asynchroniczne przerwanie* działania procesu przez inny proces lub jądro, aby przerwany proces mógł zareagować na określone zdarzenie. Można je traktować jako software'owe wersje przerwań sprzętowych. Sygnały mogą pochodzić z różnych źródeł:

- **Od sprzętu** – np. gdy jakiś proces próbuje uzyskać dostęp do adresów spoza własnej przestrzeni adresowej lub kiedy zostanie w nim wykonane dzielenie przez zero.
- **Od jądra** – są to sygnały powiadamiające proces o dostępności urządzeń wejścia-wyjścia, na które proces czekał, np. o dostępności terminala.
- **Od innych procesów** – proces potomny powiadamia swego przodka o tym, że się zakończył.
- **Od użytkowników** – użytkownicy mogą generować sygnały zakończenia, przzerwania lub stopu za pomocą klawiatury (sekwencje klawiszy generujące sygnały można sprawdzić komendą `stty -a`).

Sygnały mają przypisane nazwy zaczynające się od sekwencji `SIG` i są odpowiednio ponumerowane – szczegółowy opis dla danego systemu można znaleźć w `man 7 signal`. Nazwy sygnałów udostępnianych w systemie przechowywane są w tablicy `sys_siglist[]`, która jest indeksowana ich numerami. Aby z niej skorzystać, należy użyć deklaracji:

```
extern const char * const sys_siglist[];
```

Tablica ta w nowszych wersjach systemu Linux uważana jest za przestarzałą (ang. *deprecated*), a zamiast niej zalecane jest korzystanie z funkcji `strsignal`, która dla podanego argumentu `sig` będącego numerem sygnału zwraca jego nazwę w postaci napisu (ang. *string*). Listę ważniejszych sygnałów przedstawia Tablica [2](#).

Pliki włączane	<string.h>,		
Prototyp	char *strsignal(int sig);		
Zwracana wartość	Sukces	Porażka	Czy zmienia <code>errno</code>
	Nazwa sygnału	„unknown signal” lub NULL	Nie

Proces, który otrzymał sygnał może zareagować na trzy sposoby:

1. **Wykonać operację domyślną.** Dla większości sygnałów domyślną reakcją jest zakończenie działania procesu, po uprzednim powiadomieniu o tym procesu macierzystego. Czasem generowany jest plik zrzutu (ang. *core*), tzn. obraz pamięci zajmowanej przez proces.

2. **Zignorować sygnał.** Proces może to zrobić w reakcji na wszystkie sygnały z wyjątkiem dwóch: **SIGSTOP** (zatrzymanie procesu) i **SIGKILL** (bezw warunkowe zakończenie procesu). Dzięki niemożności ignorowania tych dwóch sygnałów system operacyjny jest zawsze w stanie usunąć niepożądane procesy.
3. **Przechwycić sygnał.** Przechwycenie sygnału oznacza wykonanie przez proces specjalnej procedury obsługi – po jej wykonaniu proces może powrócić do swego zasadniczego działania (o ile jest to właściwe w danej sytuacji). Podobnie jak ignorować, przechwytywać można wszystkie sygnały z wyjątkiem: **SIGSTOP** i **SIGKILL**.

Proces potomny dziedziczy po swoim przodku mechanizmy reagowania na wybrane sygnały. Jeżeli jednak potomek uruchomi nowy program przy pomocy funkcji `exec`, to przywrócone zostają domyślne procedury obsługi sygnałów.

Sygnał	Opis	Domyślna akcja
SIGABRT	Wysyłany przez funkcję <code>abort</code>	Zakończenie, zrzut
SIGALRM	Minał czas ustawiony przez funkcję <code>alarm</code>	Zakończenie
SIGBUS	Błąd sprzętowy (szyny)	Zakończenie, zrzut
SIGCHLD	Zakończenie procesu potomnego	Ignorowanie
SIGCONT	Uruchomienie po wstrzymaniu	Ignorowanie
SIGHUP	Zakończenie procesu sterującego terminalem	Zakończenie
SIGFPE	Wyjątek arytmetyczny	Zakończenie, zrzut
SIGILL	Nielegalna instrukcja	Zakończenie, zrzut
SIGINT	Przerwanie z klawiatury [<code>Ctrl-C</code>]	Zakończenie
SIGKILL	Bezw warunkowe zakończenie procesu (nie może być zignorowany ani przechwycony)	Zakończenie
SIGQUIT	Sekwencja wyjścia z klawiatury [<code>Ctrl-\</code>]	Zakończenie, zrzut
SIGPIPE	Proces pisze do potoku, z którego nikt nie czyta	Zakończenie
SIGSEGV	Naruszenie ograniczeń pamięci	Zakończenie, zrzut
SIGSTOP	Zatrzymanie procesu – bez zakończenia (nie może być zignorowany ani przechwycony)	Zatrzymanie procesu
SIGTERM	Żądanie zakończenia	Zakończenie
SIGTSTP	Sekwencja zatrzymania z klawiatury [<code>Ctrl-Z</code>]	Zatrzymanie procesu
SIGTTIN	Proces w tle czyta z terminala sterującego	Zatrzymanie procesu
SIGTTOU	Proces w tle pisze na terminal sterujący	Zatrzymanie procesu
SIGUSR1	Sygnał użytkownika nr 1	Zakończenie
SIGUSR2	Sygnał użytkownika nr 2	Zakończenie

Tablica 2: *Lista ważniejszych sygnałów.*

3.2 Wysyłanie sygnałów

Do wysyłania sygnałów do procesów i ich grup służy funkcja systemowa `kill`. Parametr `pid`

Pliki włączane	<sys/types.h>, <signal.h>		
Prototyp	<code>int kill(pid_t pid, int sig);</code>		
Zwracana wartość	Sukces	Porażka	Czy zmienia <code>errno</code>
	0	-1	Tak

określa proces lub grupę procesów, do których zostanie wysłany sygnał – jego znaczenie objaśnia poniższa tabelka.

Wartość <code>pid</code>	Jakie procesy odbierają sygnał
> 0	Proces o PID = <code>pid</code>
= 0	Procesy należące do samej grupy co proces wysyłający sygnał
< -1	Procesy należące do grupy o PGID = <code>-pid</code>

Parametr `sig` oznacza numer wysyłanego sygnału (można używać nazw symbolicznych). Jeżeli `sig = 0`, to funkcja `kill` nie wysyła sygnału, ale wykonuje test błędów, np. sprawdza czy proces istnieje – jeżeli nie, to `errno = ESRCH`.

Z poziomu powłoki sygnały można wysyłać za pomocą polecenia:

`kill -sig pid`

Znaczenie parametrów `sig` i `pid` jest takie jak powyżej, przy czym zamiast numeru sygnału można użyć jego nazwy symbolicznej (można nawet pominąć człon `SIG`). Listę nazw wszystkich sygnałów można uzyskać poleceniem: `kill -l`, a nazwę konkretnego sygnału o numerze `sig` poleceniem: `kill -l sig`.

Sygnał `SIGALRM` można wysłać posługując się funkcją systemową `alarm`. Funkcja ta generuje sygnał kiedy minie ustalona liczba sekund przekazana przez parametr `sec`. Jeżeli `sec = 0`, to czasomierz zostanie wyzerowany.

Pliki włączane	<unistd.h>		
Prototyp	<code>unsigned alarm(unsigned sec);</code>		
Zwracana wartość	Sukces	Porażka	Czy zmienia <code>errno</code>
	Liczba pozostałych sekund		Nie

3.3 Obsługa sygnałów

Pliki włączane	<signal.h>		
Prototyp	<code>void (*signal(int sig, void (*handler)(int)))(int);</code>		
Zwracana wartość	Sukces	Porażka	Czy zmienia <code>errno</code>
	Poprzednia dyspozycja sygnału	<code>SIG_ERR (-1)</code>	Tak

Do modyfikowania sposobu, w jaki proces zareaguje na sygnał można użyć funkcji `signal`. Prototyp tej funkcji na pierwszy rzut oka wywołuje często niemałą konsternację. Łatwiej go zrozumieć posługując się pomocniczą definicją typu `sighandler_t` będącego wskaźnikiem do funkcji:

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int sig, sighandler_t handler);
```

Pierwszym parametrem funkcji `signal` jest numer sygnału, który ma być obsługiwany – za wyjątkiem `SIGKILL` i `SIGSTOP`. Drugim parametrem natomiast jest wskaźnik do funkcji, która ma być wywołana w chwili przybycia sygnału. Funkcja ta może być określona stałymi `SIG_DFL`, `SIG_IGN` lub zdefiniowana przez użytkownika; `SIG_DFL` oznacza domyślną obsługę sygnału, natomiast `SIG_IGN` – ignorowanie sygnału. Funkcja do obsługi sygnału ma jeden parametr typu `int`, do którego zostanie automatycznie wstawiony numer sygnału.

Pliki włączane	<unistd.h>		
Prototyp	int pause(void);		
Zwracana wartość	Sukces	Porażka	Czy zmienia <code>errno</code>
	-1, jeśli sygnał nie powoduje zakończenia procesu	Nie zwraca nic	Tak

Aby spowodować oczekiwanie procesu na pojawienie się sygnału można posłużyć się funkcją biblioteczną `pause`. Funkcja ta zawiesza proces do czasu odebrania sygnału, który nie został zignorowany. Funkcja `pause` wraca tylko w przypadku przechwycenia sygnału i powrotu funkcji obsługi sygnału; zwraca wtedy wartość `-1` i ustawia zmienną `errno` na `EINTR`.

Przykładowe wywołania funkcji `signal` powodujące ignorowanie sygnału `SIGQUIT` oraz włączenie własnej obsługi sygnału `SIGINT` przy pomocy funkcji `my_sighandler` mogą wyglądać następująco:

```
void my_sighandler(int);
...
if (signal(SIGQUIT,SIG_IGN) == SIG_ERR){
    perror("Funkcja signal ma problem z SIGQUIT");
    exit(EXIT_FAILURE);
}
...
if (signal(SIGINT,my_sighandler) == SIG_ERR){
    perror("Funkcja signal ma problem z SIGINT");
    exit(EXIT_FAILURE);
}
...
```

Oczywiście funkcja `my_sighandler` musi zostać zdefiniowana przez użytkownika, zgodnie z jego życzeniem obsługi sygnału `SIGINT`.

Funkcja `signal` występuje we wszystkich wersjach systemu UNIX, ale niestety nie jest niezawodna (może nie obsłużyć poprawnie wielu sygnałów, które następują w krótkim czasie po sobie). Dlatego w standardzie POSIX wprowadzono dodatkową funkcję do obsługi sygnałów o nazwie `sigaction`, która spełnia wymogi niezawodności. Również do wysyłania sygnałów wprowadzono bardziej wyrafinowany odpowiednik funkcji `kill` o nazwie `sigqueue`. Obie te funkcje umożliwiają zaawansowane i szczegółowe zarządzanie sygnałami, i powinno się ich używać w profesjonalnych zastosowaniach. Ich opisy można znaleźć w podręczniku systemowym `man`. My jednak dla celów dydaktycznych ograniczymy się do funkcji `signal` i `kill`, które są znacznie prostsze w użyciu.

ĆWICZENIE 3: WYSYŁANIE I OBSŁUGA SYGNAŁÓW

- (a) Napisać program do obsługi sygnałów z możliwościami: (1) wykonania operacji domyślnej, (2) ignorowania oraz (3) przechwycenia i własnej obsługi sygnału. Do oczekiwania na sygnał użyć funkcji `pause`. Uruchomić program i wysyłać do niego sygnały przy pomocy sekwencji klawiszy oraz przy pomocy polecenia `kill` z poziomu powłoki.
- (b) Uruchomić powyższy program poprzez funkcję `execlp` w procesie potomnym innego procesu (z użyciem funkcji `fork`) i wysyłać do niego sygnały poprzez funkcję systemową `kill` z procesu macierzystego. → UWAGA: Przed wysłaniem sygnału sprawdzić, czy proces istnieje (patrz podrozdział 3.2). Proces macierzysty powinien poczekać na zakończenie swojego potomka przy pomocy funkcji `wait`, a następnie wypisać jego PID i status zakończenia zwracane przez tę funkcję. W przypadku zakończenia procesu potomnego przez sygnał powinien wypisać dodatkowo numer tego sygnału zawarty w statusie zakończenia, a także jego nazwę (użyć funkcji `strsignal`). Do wyłuskania numeru sygnału ze statusu zakończenia użyć makr opisanych w podrozdziale 2.4.
- (c) W procesie macierzystym utworzyć proces potomny i sprawić, aby stał się liderem nowej grupy procesów (funkcja `setpgid`), a następnie poprzez funkcję `execlp` uruchomić w nim program tworzący kilka procesów potomnych, które (poprzez `execlp`) wykonują program do obsługi sygnałów z punktu (a). Z pierwszego procesu macierzystego wysyłać sygnały do całej grupy procesów potomnych po uprzednim sprawdzeniu jej istnienia. Identyfikator tej grupy procesów uzyskać przy pomocy funkcji `getpgid`. Proces będący liderem grupy procesów niech ignoruje sygnały, a na końcu czeka na zakończenie wszystkich swoich procesów potomnych i wypisuje ich identyfikatory PID oraz status zakończenia (zwracane przez `wait`), a w przypadku zakończenia tych procesów przez sygnał – również numer i nazwę odpowiedniego sygnału. Pierwszy proces macierzysty również powinien poczekać na zakończenie swojego bezpośredniego potomka i wypisać jego PID oraz status zakończenia zwracane przez odpowiednią funkcję `wait`.

Numer sygnału oraz opcję jego obsługi we wszystkich powyższych programach przekazywać za pomocą argumentów wywołania programu – sprawdzać ich liczbę i wypisywać odpowiedni komunikat w przypadku błędnego uruchomienia (ze wskazówką poprawnego uruchomienia).