

# Sygnały

*Sygnały* są przerwaniami programowymi, które dostarczają mechanizmu pozwalającego na obsługę zdarzeń asynchronicznych. Zdarzenia te mogą powstawać poza systemem (na przykład, gdy użytkownik wysyła znak przerwania, uzyskiwany poprzez naciśnięcie sekwencji klawiszy *Ctrl+C*) lub być generowane przez program lub jądro — na przykład, gdy proces wykonuje kod przeprowadzający operację dzielenia przez zero. Proces może również wysyłać sygnały do innego procesu i przez to realizować prymitwną odmianę komunikacji międzyprocesowej (IPC).

Punktem kluczowym całego zagadnienia nie jest jedynie asynchroniczne występowanie zdarzeń (użytkownik może na przykład wysłać sekwencję klawiszy *Ctrl+C* w każdym momencie działania programu), lecz również zdolność programu do asynchronicznej obsługi sygnałów. Funkcje obsługujące sygnały są zarejestrowane w jądrze. Z chwilą nadania sygnałów jądro wywołuje te funkcje w sposób asynchroniczny do pozostałej części programu.

Sygnały od dawna były częścią systemu Unix. Z biegiem czasu ewoluowały w kategoriach niezawodności (dawniej „znikały” w systemie) oraz funkcjonalności (obecnie mogą zawierać pola zdefiniowane przez użytkownika). Na początku w różnych systemach uniksowych przeprowadzono niekompatybilne zmiany w opisach sygnałów. Na szczęście POSIX uratował sytuację i zdefiniował standard ich obsługi, który jest wspierany w systemie Linux i zostanie tutaj omówiony.

Na początku tego rozdziału przedstawiony zostanie przegląd sygnałów oraz analiza poświęcona właściwym i niewłaściwym sposobom ich użycia. Następnie zaprezentowane zostaną różne interfejsy Linuksa, dzięki którym można obsługiwać sygnały i zarządzać nimi.

Sygnałów używa się w większości złożonych aplikacji. Nawet jeśli dana aplikacja jest celowo zaprojektowana tak, aby w swojej warstwie komunikacyjnej nie używała sygnałów (jest to często dobrym pomysłem!), mimo to w niektórych przypadkach wciąż istnieje konieczność współpracy z sygnałami, na przykład podczas obsługi procesu kończenia działania programu.

# Koncepcja sygnałów

Sygnały posiadają określony czas istnienia. Sygnał zostaje zgłoszony (*wysłany* lub *wygenerowany*), a następnie *przechowywany* przez jądro, dopóki nie będzie mogło go dostarczyć. Gdy pojawia się taka możliwość, jądro w odpowiedni sposób *obsługuje* sygnał. W zależności od tego, jakie wymagania posiada proces, jądro może wykonać jedną z poniżej przedstawionych trzech czynności:

## Zignorować sygnał

Nie jest podejmowana żadna akcja. Istnieją dwa sygnały, które nie mogą zostać zignorowane: SIGKILL oraz SIGSTOP. Administrator systemu musi posiadać możliwość usuwania lub zatrzymywania procesów, bo gdyby proces mógł zignorować sygnał SIGKILL (przez co nie można byłoby go usunąć) lub sygnał SIGSTOP (dzięki czemu proces stałby się niemożliwy do zatrzymania), wówczas byłoby to naruszeniem powyższej zasady.

## Przechwycić i obsłużyć sygnał

Jądro wstrzyma wykonywanie aktualnej ścieżki kodu dla danego procesu i wywoła wcześniej zarejestrowaną funkcję. Następnie proces wykona tę funkcję. Gdy tylko zakończy się jej działanie, proces ponownie wróci do poprzedniego miejsca w kodzie, w którym znajdował się w momencie przechwycenia sygnału. Sygnały SIGINT oraz SIGTERM są dwoma powszechnie przechwytywanymi sygnałami. Sygnał SIGINT jest przechwytywany przez procesy, aby obsłużyć wygenerowanie znaku przerwania przez użytkownika — na przykład, możliwe byłoby przechwycenie tego sygnału w terminalu i powrót do standardowego wyświetlanego znaku zachęty w linii poleceń. Sygnał SIGTERM jest przejmowany przez procesy, aby przed zakończeniem programu przeprowadzić niezbędne operacje porządkujące, takie jak odłączenie od sieci lub usunięcie plików tymczasowych. Sygnały SIGKILL i SIGSTOP nie mogą zostać przechwycone.

## Wykonać akcję domyślną

Akcja ta jest zależna od wysyłanego sygnału. Domyślną akcją jest często przerwanie działania procesu. Dotyczy to na przykład sygnału SIGKILL. Wiele sygnałów związanych jest jednak ze specyficznymi zastosowaniami, które interesują programistów tylko w pewnych sytuacjach, dlatego też takie sygnały są domyślnie ignorowane, ponieważ w wielu programach nie istnieje potrzeba, aby się nimi zajmować. Wkrótce zostaną omówione różne sygnały i odpowiednie dla nich akcje domyślne.

Dawniej, gdy sygnał został dostarczony, funkcja, która go obsługiwała, nie posiadała żadnej informacji na temat powodów jego wygenerowania, za wyjątkiem tego, że został zgłoszony określony rodzaj sygnału. Obecnie jądro dostarcza mnóstwo informacji kontekstowej dla programistów, którzy chcą ją wykorzystać. Sygnały, jak będziemy mogli to zauważyc, potrafią nawet przenosić dane zdefiniowane przez użytkownika.

# Identyfikatory sygnałów

Każdy sygnał posiada nazwę symboliczną, która rozpoczyna się od przedrostka SIG. Na przykład SIGINT jest sygnałem, który zostaje wysłany, gdy użytkownik naciska klawisze *Ctrl+C*, SIGABRT pojawia się, gdy proces wywołuje funkcję *abort()*, a SIGKILL zostaje wysłany, gdy proces jest zmuszony do przerwania swojego działania.

Wszystkie te sygnały są zdefiniowane w pliku nagłówkowym dołączanym do pliku `<signal.h>`. Są zwykłymi definicjami preprocesora, które reprezentują dodatnie liczby całkowite — oznacza to, że każdy sygnał związany jest również z liczbowym identyfikatorem. Odwzorowanie nazwy na liczbę całkowitą w przypadku sygnałów jest zależne od implementacji i zmienia się w zależności od rodzaju systemu uniksowego, choć początkowe sygnały są zazwyczaj odwzorowane w taki sam sposób (na przykład, `SIGKILL` to cieszący się złą sławą *sygnał o numerze 9*). Dobry programista będzie zawsze używał nazwy symbolicznej sygnału zamiast jego wartości liczbowej.

Numery sygnałów rozpoczynają się od 1 (jest nim zazwyczaj `SIGHUP`) i rosną w sposób liniowy. W sumie istnieje około 31 sygnałów, lecz w większości programów obsługuje się tylko kilka z nich. Nie istnieje sygnał o wartości 0, lecz jest specjalna wartość nazywana *sygnałem zerowym*. Nie jest on jednak żadnym ważnym sygnałem (nie zasługuje na specjalną nazwę), choć w wyjątkowych przypadkach jest używany przez niektóre funkcje systemowe (na przykład `kill()`).



Listę sygnałów, wspieranych w danym systemie, można wygenerować przez użycie polecenia `kill -l`.

## Sygnały wspierane przez system Linux

Tabela 10.1. przedstawia listę sygnałów, które są wspierane przez system Linux.

Tabela 10.1. Sygnały

Sygnal	Opis	Akcja domyślna
<code>SIGABRT</code>	Wysyłany przez funkcję <code>abort()</code> .	Przerwanie procesu i stworzenie pliku zrzutu systemowego.
<code>SIGALRM</code>	Wysyłany przez funkcję <code>alarm()</code> .	Przerwanie procesu.
<code>SIGBUS</code>	Błąd sprzętu lub wyrównania.	Przerwanie procesu i stworzenie pliku zrzutu systemowego.
<code>SIGCHLD</code>	Proces potomny został zakończony.	Ignorowanie sygnału.
<code>SIGCONT</code>	Zatrzymany proces ponownie rozpoczyna swoje działanie.	Ignorowanie sygnału.
<code>SIGFPE</code>	Wyjątek arytmetyczny.	Przerwanie procesu i stworzenie pliku zrzutu systemowego.
<code>SIGHUP</code>	Sterujący terminal procesu został zamknięty (najprawdopodobniej użytkownik wylogował się z systemu).	Przerwanie procesu.
<code>SIGILL</code>	Proces próbował wykonać niedopuszczalną instrukcję.	Przerwanie procesu i stworzenie pliku zrzutu systemowego.
<code>SIGINT</code>	Użytkownik wysłał znak przerwania ( <code>Ctrl+C</code> ).	Przerwanie procesu.
<code>SIGIO</code>	Zdarzenie asynchronicznej operacji wejścia i wyjścia.	Przerwanie procesu <sup>1</sup> .
<code>SIGKILL</code>	Bezwarkowne przerwanie działania procesu.	Przerwanie procesu.
<code>SIGPIPE</code>	Proces zapisywał do potoku, lecz nie istniały odbiorcy wiadomości.	Przerwanie procesu.
<code>SIGPROF</code>	Upłynął czas dla licznika profilującego.	Przerwanie procesu.

<sup>1</sup> W przypadku innych systemów uniksowych, takich jak BSD, zachowaniem domyślnym jest ignorowanie tego sygnału.

# Koncepcja sygnałów

Sygnały posiadają określony czas istnienia. Sygnał zostaje zgłoszony (*wysłany lub wygenerowany*), a następnie *przechowywany* przez jądro, dopóki nie będzie mogło go dostarczyć. Gdy pojawia się taka możliwość, jądro w odpowiedni sposób *obsługuje* sygnał. W zależności od tego, jakie wymagania posiada proces, jądro może wykonać jedną z poniżej przedstawionych trzech czynności:

## Zignorować sygnał

Nie jest podejmowana żadna akcja. Istnieją dwa sygnały, które nie mogą zostać zignorowane: SIGKILL oraz SIGSTOP. Administrator systemu musi posiadać możliwość usuwania lub zatrzymywania procesów, bo gdyby proces mógł zignorować sygnał SIGKILL (przez co nie można byłoby go usunąć) lub sygnał SIGSTOP (dzięki czemu proces stałby się niemożliwy do zatrzymania), wówczas byłoby to naruszeniem powyższej zasady.

## Przechwycić i obsłużyć sygnał

Jądro wstrzyma wykonywanie aktualnej ścieżki kodu dla danego procesu i wywoła wcześniej zarejestrowaną funkcję. Następnie proces wykona tę funkcję. Gdy tylko zakończy się jej działanie, proces ponownie wróci do poprzedniego miejsca w kodzie, w którym znajdował się w momencie przechwycenia sygnału. Sygnały SIGINT oraz SIGTERM są dwoma powszechnie przechwytywanymi sygnałami. Sygnał SIGINT jest przechwytywany przez procesy, aby obsłużyć wygenerowanie znaku przerwania przez użytkownika — na przykład, możliwe byłoby przechwycenie tego sygnału w terminalu i powrót do standardowego wyświetlanego znaku zachęty w linii poleceń. Sygnał SIGTERM jest przejmowany przez procesy, aby przed zakończeniem programu przeprowadzić niezbędne operacje porządkujące, takie jak odłączenie od sieci lub usunięcie plików tymczasowych. Sygnały SIGKILL i SIGSTOP nie mogą zostać przechwycone.

## Wykonać akcję domyślną

Akcja ta jest zależna od wysyłanego sygnału. Domyślną akcją jest często przerwanie działania procesu. Dotyczy to na przykład sygnału SIGKILL. Wiele sygnałów związanych jest jednak ze specyficznymi zastosowaniami, które interesują programistów tylko w pewnych sytuacjach, dlatego też takie sygnały są domyślnie ignorowane, ponieważ w wielu programach nie istnieje potrzeba, aby się nimi zajmować. Wkrótce zostaną omówione różne sygnały i odpowiednie dla nich akcje domyślne.

Dawniej, gdy sygnał został dostarczony, funkcja, która go obsługiwała, nie posiadała żadnej informacji na temat powodów jego wygenerowania, za wyjątkiem tego, że został zgłoszony określony rodzaj sygnału. Obecnie jądro dostarcza mnóstwo informacji kontekstowej dla programistów, którzy chcą ją wykorzystać. Sygnały, jak będziemy mogli to zauważyc, potrafią nawet przenosić dane zdefiniowane przez użytkownika.

# Identyfikatory sygnałów

Każdy sygnał posiada nazwę symboliczną, która rozpoczyna się od przedrostka SIG. Na przykład SIGINT jest sygnałem, który zostaje wysłany, gdy użytkownik naciśka klawisze *Ctrl+C*, SIGABRT pojawia się, gdy proces wywołuje funkcję *abort()*, a SIGKILL zostaje wysłany, gdy proces jest zmuszony do przerwania swojego działania.

Wszystkie te sygnały są zdefiniowane w pliku nagłówkowym dołączanym do pliku `<signal.h>`. Są zwykłymi definicjami preprocesora, które reprezentują dodatnie liczby całkowite — oznacza to, że każdy sygnał związany jest również z liczbowym identyfikatorem. Odwzorowanie nazwy na liczbę całkowitą w przypadku sygnałów jest zależne od implementacji i zmienia się w zależności od rodzaju systemu uniksowego, choć początkowe sygnały są zazwyczaj odwzorowane w taki sam sposób (na przykład, `SIGKILL` to cieszący się złą sławą *sygnał o numerze 9*). Dobry programista będzie zawsze używał nazwy symbolicznej sygnału zamiast jego wartości liczbowej.

Numery sygnałów rozpoczynają się od 1 (jest nim zazwyczaj `SIGHUP`) i rosną w sposób liniowy. W sumie istnieje około 31 sygnałów, lecz w większości programów obsługuje się tylko kilka z nich. Nie istnieje sygnał o wartości 0, lecz jest specjalna wartość nazywana *sygnałem zerowym*. Nie jest on jednak żadnym ważnym sygnałem (nie zasługuje na specjalną nazwę), choć w wyjątkowych przypadkach jest używany przez niektóre funkcje systemowe (na przykład `kill()`).



Listę sygnałów, wspieranych w danym systemie, można wygenerować przez użycie polecenia `kill -l`.

## Sygnały wspierane przez system Linux

Tabela 10.1. przedstawia listę sygnałów, które są wspierane przez system Linux.

Tabela 10.1. Sygnały

Sygnal	Opis	Akcja domyślna
<code>SIGABRT</code>	Wysyłany przez funkcję <code>abort()</code> .	Przerwanie procesu i stworzenie pliku zrzutu systemowego.
<code>SIGALRM</code>	Wysyłany przez funkcję <code>alarm()</code> .	Przerwanie procesu.
<code>SIGBUS</code>	Błąd sprzętu lub wyrównania.	Przerwanie procesu i stworzenie pliku zrzutu systemowego.
<code>SIGCHLD</code>	Proces potomny został zakończony.	Ignorowanie sygnału.
<code>SIGCONT</code>	Zatrzymany proces ponownie rozpoczyna swoje działanie.	Ignorowanie sygnału.
<code>SIGFPE</code>	Wyjątek arytmetyczny.	Przerwanie procesu i stworzenie pliku zrzutu systemowego.
<code>SIGHUP</code>	Sterujący terminal procesu został zamknięty (najprawdopodobniej użytkownik wylogował się z systemu).	Przerwanie procesu.
<code>SIGILL</code>	Proces próbował wykonać niedopuszczalną instrukcję.	Przerwanie procesu i stworzenie pliku zrzutu systemowego.
<code>SIGINT</code>	Użytkownik wysłał znak przerwania ( <code>Ctrl+C</code> ).	Przerwanie procesu.
<code>SIGIO</code>	Zdarzenie asynchronicznej operacji wejścia i wyjścia.	Przerwanie procesu <sup>1</sup> .
<code>SIGKILL</code>	Bezwarkowne przerwanie działania procesu.	Przerwanie procesu.
<code>SIGPIPE</code>	Proces zapisywał do potoku, lecz nie istniał odbiorcy wiadomości.	Przerwanie procesu.
<code>SIGPROF</code>	Upłynął czas dla licznika profilującego.	Przerwanie procesu.

<sup>1</sup> W przypadku innych systemów uniksowych, takich jak BSD, zachowaniem domyślnym jest ignorowanie tego sygnału.

Tabela 10.1. Sygnały (ciąg dalszy)

Sygnal	Opis	Akcja domyślna
SIGPWR	Awaria zasilania.	Przerwanie procesu.
SIGQUIT	Użytkownik wysłał znak wyjścia ( <i>Ctrl+`</i> ).	Przerwanie procesu i stworzenie pliku zrzutu systemowego.
SIGSEGV	Błąd dostępu do pamięci.	Przerwanie procesu i stworzenie pliku zrzutu systemowego.
SIGSTKFLT	Błąd stosu koprocesora.	Przerwanie procesu <sup>2</sup> .
SIGSTOP	Zatrzymanie wykonania procesu.	Zatrzymanie procesu.
SIGSYS	Proces próbował wykonać błędą funkcję systemową.	Przerwanie procesu i stworzenie pliku zrzutu systemowego.
SIGTERM	Przerwanie wykonywania procesu, możliwe do przechwycenia.	Przerwanie procesu.
SIGTRAP	Napotkano punkt wstrzymania.	Przerwanie procesu i stworzenie pliku zrzutu systemowego.
SIGTSTP	Użytkownik wysłał znak zatrzymania ( <i>Ctrl+Z</i> ).	Zatrzymanie procesu.
SIGTTIN	Proces drugorzędny czytał z terminala sterującego.	Zatrzymanie procesu.
SIGTTOU	Proces drugorzędny pisał do terminala sterującego.	Zatrzymanie procesu.
SIGURG	Pilna operacja wejścia i wyjścia oczekuje na obsługę.	Ignorowanie sygnału.
SIGUSR1	Sygnal zdefiniowany dla procesu.	Przerwanie procesu.
SIGUSR2	Sygnal zdefiniowany dla procesu.	Przerwanie procesu.
SIGVTALRM	Sygnal wygenerowany przez funkcję <code>setitimer()</code> podczas jej wywołania, z ustawionym znacznikiem <code>ITIMER_VIRTUAL</code> .	Przerwanie procesu.
SIGWINCH	Rozmiar okna dla terminala sterującego uległ zmianie.	Ignorowanie sygnału.
SIGXCPU	Przekroczenie ograniczeń dla zasobów procesora.	Przerwanie procesu i stworzenie pliku zrzutu systemowego.
SIGXFSZ	Przekroczenie ograniczeń dla zasobów plikowych.	Przerwanie procesu i stworzenie pliku zrzutu systemowego.

Istnieją też inne sygnały, lecz w systemie Linux są one zdefiniowane jako równoważniki istniejących wartości: `SIGINFO` jest zdefiniowany jako `SIGPWR`<sup>3</sup>, `SIGIOT` jako `SIGABRT`, a `SIGPOLL` oraz `SIGLOST` jako `SIGIO`.

W powyższej tabeli przedstawiono skróconą definicję sygnałów, teraz dokładnie je scharakteryzujemy:

#### SIGABRT

Sygnal ten zostaje wysłany przez funkcję `abort()` do procesu, który ją wywołuje. Proces kończy swoje działanie oraz generuje plik zrzutu systemowego. Funkcje weryfikacji warunków, takie jak `assert()`, wywołują w Linuksie `abort()`, gdy warunek nie jest spełniony.

#### SIGALRM

Sygnal ten jest wysyłany przez funkcje `alarm()` oraz `setitimer()` (z aktywnym znacznikiem `ITIMER_REAL`) do procesu, który je wywołał, gdy upłynie termin ważności dla alarmu. Te i podobne im funkcje omówione zostaną w rozdziale 11.

<sup>2</sup> Jądro Linuksa nie generuje już tego sygnału; pozostał on jedynie w celu zachowania wstępnej kompatybilności.

<sup>3</sup> Sygnal ten jest zdefiniowany wyłącznie w architekturze Alpha. W przypadku innych architektur maszynowych `SIGPWR` nie istnieje.

## SIGBUS

Jądro zgłasza ten sygnał, gdy wykonanie procesu powoduje powstanie błędu sprzętowego, innego niż błąd ochrony pamięci, który generuje SIGSEGV. W tradycyjnych systemach uniksowych sygnał ten reprezentował różne błędy, niemożliwe do naprawienia, takie jak dostęp do pamięci niewyrównanej. Jądro Linuksa naprawia jednak większość z nich w sposób automatyczny i nie generuje SIGBUS. Jądro nie zgłasza tego sygnału, gdy proces w sposób niepoprawny próbuje uzyskać dostęp do rejonu pamięci stworzonego za pomocą funkcji mmap() (analiza odwzorowań w pamięci przeprowadzona jest w rozdziale 9.). Dopóki ten sygnał nie będzie przechwycony, jądro zakończy działanie procesu i wygeneruje plik zrzutu systemowego.

## SIGCHLD

Gdy proces kończy swoje działanie lub zatrzymuje się, jądro wysyła ten sygnał do jego procesu rodzicielskiego. Ponieważ SIGCHLD jest domyślnie ignorowany, procesy muszą go jawnie przechwycić i obsłużyć, jeśli są zainteresowane tym, co dzieje się z ich potomkami. Procedura obsługi tego sygnału wywołuje zazwyczaj funkcję wait(), omówioną w rozdziale 5., aby ustalić identyfikator procesu oraz kod powrotu dla potomka.

## SIGCONT

Sygnał ten zostaje wysłany przez jądro do procesu, który był zatrzymany, a obecnie ponownie rozpoczyna swoje wykonywanie. SIGCONT jest domyślnie ignorowany, lecz może zostać przechwycony przez procesy, jeśli wymagana jest jakaś akcja po ponownym rozpoczęciu ich wykonywania. Sygnał ten jest powszechnie używany przez terminali lub edytory, które wymagają uaktualnienia zawartości ekranu.

## SIGFPE

Wbrew swojej nazwie sygnał ten reprezentuje powstanie dowolnego wyjątku arytmetycznego, niekoniecznie związanego z operacjami zmiennoprzecinkowymi. Rodzajami wyjątków są przepelnilenia, niedomiary lub dzielenie przez zero. Domyślną akcją jest przerwanie procesu i wygenerowanie pliku zrzutu systemowego, lecz procesy mogą przechwycić i obsłużyć ten sygnał, jeśli zaistnieje taka potrzeba. Należy zauważać, że zachowanie procesu oraz wynik operacji, która spowodowała powstanie problemu, będą niezdefiniowane, jeśli proces postanowi kontynuować swoje działanie.

## SIGHUP

Sygnał ten zostaje wysłany przez jądro do lidera sesji, gdy terminal tej sesji zostaje odłączony. Jądro wysyła ten sygnał również do każdego procesu z pierwszoplanowej grupy procesów, gdy lider sesji kończy swoje działanie. Domyślną akcją jest przerwanie działania procesu i ma to sens — sygnał informuje, że użytkownik się wylogował. Procesy demonów „przesłaniają” ten sygnał mechanizmem, który umożliwia ponowne załadowanie ich plików konfiguracyjnych. Wysłanie sygnału SIGHUP na przykład do serwera Apache spowoduje, że odczytany zostanie ponownie plik *httpd.conf*. Użycie do tego celu sygnału SIGHUP jest powszechnie stosowaną konwencją, lecz nie jest wymagane. To bezpieczny sposób, ponieważ demony nie posiadają terminali sterujących i dlatego też nigdy nie powinny odebrać takiego sygnału.

## SIGILL

Sygnał ten zostaje wysłany przez jądro, gdy proces przystępuje do wykonania niepoprawnej instrukcji maszynowej. Domyślną akcją jest przerwanie działania procesu i wygenerowanie pliku zrzutu systemowego. Procesy mogą próbować przechwycić oraz obsłużyć sygnał SIGILL, lecz ich zachowanie będzie wówczas niezdefiniowane.

## SIGINT

Sygnal ten zostaje wysłany do wszystkich procesów należących do pierwszoplanowej grupy procesów, gdy użytkownik generuje znak przerwania (zwykle *Ctrl+C*). Domyślnym zachowaniem jest przerwanie działania procesu, lecz sygnal ten może zostać przez niego przechwycony i obsłużony. Zazwyczaj dzieje się tak, aby wykonać porządkowanie przed zakończeniem działania procesu.

## SIGIO

Sygnal ten zostaje wysłany, gdy wygenerowane jest asynchroniczne zdarzenie operacji wejścia i wyjścia w stylu BSD. Ten sposób wykonywania operacji wejścia i wyjścia jest rzadko używany w systemie Linux (w rozdziale 4. omówione zostały zaawansowane techniki wykonywania operacji wejścia i wyjścia, które są charakterystyczne dla Linuksa).

## SIGKILL

Sygnal ten zostaje wysłany z funkcji systemowej `kill()`; dostarcza administratorom systemu sprawdzonego sposobu pozwalającego na bezwarunkowe przerwanie działania procesu. Sygnal ten nie może zostać przechwycony ani zignorowany, a jego wynikiem jest zawsze zakończenie procesu.

## SIGPIPE

Jeśli proces wykonuje operację zapisu do potoku, lecz odbiorca danych zakończył swoje działanie, wówczas sygnal ten zostaje zgłoszony przez jądro. Domyślną akcją jest przerwanie działania procesu, lecz sygnal ten może zostać przez niego przechwycony i obsłużony.

## SIGPROF

Sygnal ten zostaje wygenerowany przez funkcję `setitimer()`, użytą ze znacznikiem `ITIMER_PROF`, gdy upłynął czas dla licznika profilującego. Domyślną akcją jest przerwanie działania procesu.

## SIGPWR

Ten sygnal jest zależny od systemu. W przypadku Linuksa reprezentuje on niski stan naładowania urządzenia podtrzymującego napięcie (takiego jak zasilacz awaryjny). Demon śledzący stan zasilacza awaryjnego wysyła ten sygnal do procesu `init`, który odpowiada na niego poprzez wykonanie operacji porządkowania i zamknięcia systemu operacyjnego — zakładając, że nastąpi to przed całkowitym zanikiem zasilania!

## SIGQUIT

Sygnal ten zostaje wysłany do wszystkich procesów należących do pierwszoplanowej grupy procesów, gdy użytkownik generuje znak wyjścia z terminala (zwykle *Ctrl+\*). Domyślną akcją jest przerwanie działania procesu i wygenerowanie pliku zrzutu systemowego.

## SIGSEGV

Sygnal ten, którego nazwa oznacza *błąd segmentacji*, wysyłany jest do procesu, który pragnął uzyskać dostęp do niepoprawnego adresu pamięci. Dotyczy to również dostępu do nieodwzorowanej pamięci, czytania z pamięci, która nie posiada uprawnienia do odczytu, uruchamiania kodu w pamięci pozbawionej uprawnienia do wykonywania lub zapisywania do pamięci, która nie posiada uprawnienia do zapisu. Sygnal ten może zostać przechwycony i obsłużony przez dany proces, lecz domyślną akcją jest przerwanie jego działania i wygenerowanie pliku zrzutu systemowego.

## SIGSTOP

Sygnal ten jest wysyłany przez funkcję `kill()`. Zatrzymuje on bezwarunkowo proces i nie może zostać przechwycony ani zignorowany.

## SIGSYS

Sygnal ten zostaje wysłany przez jądro do procesu, gdy próbuje on wywołać niepoprawną funkcję systemową. Może się to zdarzyć, gdy plik binarny zostanie stworzony w nowszej wersji systemu operacyjnego (z nowszymi wersjami funkcji systemowych), lecz będzie uruchomiony w maszynie, która wyposażona jest w starszą wersję systemu. Poprawnie stworzone pliki binarne, które wywołują funkcję systemowe poprzez bibliotekę *glibc*, nie powinny nigdy otrzymać tego sygnału. Błędna funkcja systemowa powinna zwrócić `-1` oraz ustawić zmienną `errno` na `ENOSYS`.

## SIGTERM

Sygnal ten jest wysyłany wyłącznie przez funkcję `kill()`; pozwala użytkownikowi na poprawne zakończenie działania procesu (jest to akcja domyślna). Sygnal ten może zostać przechwycony przez dany proces, dzięki czemu możliwe będzie wykonanie operacji porządkowania, lecz niepoprawnym działaniem jest przechwycenie sygnału, a następnie przeprowadzenie niewłaściwego zakończenia procesu.

## SIGTRAP

Sygnal ten zostaje wysłany przez jądro do procesu, gdy napotyka on punkt wstrzymania. Zasadniczo SIGTRAP остaje przechwycony przez programy wspomagające uruchamianie, a inne procesy go ignorują.

## SIGTSTP

Sygnal ten zostaje wysłany do wszystkich procesów należących do pierwszoplanowej grupy procesów, gdy użytkownik generuje znak zatrzymania (zwykle *Ctrl+Z*).

## SIGTTIN

Sygnal ten zostaje wysłany do procesu działającego w tle, gdy zamierza on czytać z terminala sterującego. Domyślną akcją jest zatrzymanie tego procesu.

## SIGTTOUT

Sygnal ten zostaje wysłany do procesu, działającego w tle, gdy zamierza on pisać do terminala sterującego. Domyślną akcją jest zatrzymanie tego procesu.

## SIGURG

Sygnal ten zostaje wysłany do procesu, gdy w gnieździe pojawiły się dane poza kolejką. Analiza danych poza kolejką nie jest uwzględniona w tej książce.

## SIGUSR1 oraz SIGUSR2

Sygnały te są udostępnione dla zastosowań zdefiniowanych przez użytkownika: nie znajdują one nigdy zgłoszone przez jądro. Procesy mogą używać sygnałów SIGUSR1 oraz SIGUSR2 dla dowolnych celów. Powszechnym sposobem ich użycia jest wysłanie instrukcji do demona, aby zmienić jego sposób działania. Domyślną akcją jest przerwanie działania procesu.

## SIGVTALRM

Sygnal ten zostaje wysłany przez funkcję `setitimer()`, gdy upłynął czas dla licznika stworzonego przy użyciu znacznika `ITIMER_VIRTUAL`. Liczniki omówione są w rozdziale 11.

#### SIGWINCH

Sygnal ten zostaje wysłany do wszystkich procesów należących do pierwszoplanowej grupy procesów, gdy rozmiar okna dla ich terminala zostanie zmieniony. Procesy domyślnie ignorują ten sygnał, lecz może zostać on przechwycony i obsłużony, jeśli rozmiar okna terminala jest dla nich przydatnym parametrem. Dobrym przykładem programu, który przechwytuje ten sygnał, jest *top* — wystarczy zmienić rozmiar okna podczas jego działania i obserwować, jak się zachowuje.

#### SIGXCPU

Sygnal ten zostaje wysłany przez jądro, gdy proces przekroczy swoje miękkie ograniczenie czasu procesora. Jądro będzie wysyłać ten sygnał co sekundę, dopóki proces nie zakończy swojego działania lub nie przekroczy trwałego ograniczenia czasu procesora. Gdy tylko zostanie ono przekroczone, jądro wyśle procesowi sygnał SIGKILL.

#### SIGXFSZ

Sygnal ten zostaje wysłany przez jądro, gdy proces przekroczy swoje miękkie ograniczenie rozmiaru pliku. Domyślnym zachowaniem jest przerwanie działania procesu, lecz jeśli sygnał ten zostanie przechwycony i zignorowany, wówczas funkcje systemowe, których wywołanie powodowałoby przekroczenie ograniczenia rozmiaru pliku, zwrócią -1 oraz ustawią zmienną errno na EFBIG.

## Podstawowe zarządzanie sygnałami

Po zaprezentowaniu sygnałów możemy obecnie zająć się ich obsługą w Twoich programach. Najprostszym i jednocześnie najstarszym interfejsem służącym do zarządzania sygnałami jest funkcja `signal()`. To bardzo prosta funkcja, opisana w standardzie ISO C89 definiującym najmniejszą wspólną część zagadnienia zarządzania sygnałami. Linux oferuje znacznie więcej możliwości kontroli sygnałów dzięki udostępnianiu innych interfejsów, które zostaną omówione w dalszej części rozdziału. Ponieważ funkcja `signal()` jest najbardziej podstawowa, a dzięki obecności w ISO C, również najczęściej używana, dlatego też zostanie zaprezentowana jako pierwsza:

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal (int signo, sighandler_t handler);
```

Poprawne wywołanie funkcji `signal()` instaluje w miejscu aktualnie istniejącej procedury obsługi odebranego sygnału `signo` nową procedurę, wskazywaną przez parametr `handler`. Parametr `signo` jest jedną z nazw sygnałów, takich jak SIGINT lub SIGUSR1, omówionych w poprzednim podrozdziale. Jak już napisano, proces nie może przechwycić sygnałów SIGKILL oraz SIGSTOP, dlatego też definiowanie dla nich procedury obsługi nie ma sensu.

Funkcja `handler` musi zwracać `void` — jest to logiczne, ponieważ (w przeciwieństwie do zwykłych funkcji) nie istnieje dla niej standardowe miejsce w programie, do którego mogłyby wracać. Handler posiada jeden parametr — liczbę całkowitą, określającą identyfikator sygnału (na przykład SIGUSR2), który jest właśnie obsługiwany. Pozwala to na obsługę wielu sygnałów dla pojedynczej funkcji. Prototyp funkcji wygląda następująco:

```
void my_handler (int signo);
```

Linux zamienia go na definicję typu `sighandler_t`. W innych systemach uniksowych używane są bezpośrednio wskaźniki do funkcji, a niektóre systemy definiują własne typy danych, które mogą posiadać inne nazwy niż `sighandler_t`. W programach, dla których ważna jest przenośność, nie powinno się bezpośrednio używać tego typu.

Gdy sygnał zostaje zgłoszony do procesu, który zarejestrował jego procedurę obsługi, wówczas jądro zawiesza wykonywanie zwykłego strumienia instrukcji programu, a zamiast tego wywołuje powyższą procedurę. Jest do niej przekazana wartość informującą o rodzaju sygnału, równa parametrowi `signo`, który pierwotnie został przekazany do funkcji `signal()`.

Funkcji `signal()` można również użyć, aby poinformować jądro, by ignorowało dany sygnał dla aktualnego procesu lub przywróciło dla tego sygnału domyślne zachowanie. Można to uzyskać poprzez użycie specjalnych wartości przekazanych w parametrze `handler`:

#### `SIG_DFL`

Dla sygnału `signo` zostanie przywrócone domyślne zachowanie. Na przykład, w przypadku sygnału `SIGPIPE` proces zostanie zakończony.

#### `SIG_IGN`

Sygnał `signo` zostanie zignorowany.

Funkcja `signal()` zwraca kod reprezentujący poprzednie zachowanie sygnału, którym może być wskaźnik do procedury obsługi, `SIG_DFL` lub `SIG_IGN`. W przypadku błędu funkcja zwraca `SIG_ERR`, lecz nie ustawia zmiennej `errno`.

## Oczekiwanie na dowolny sygnał

Funkcja `pause()`, zdefiniowana w standardzie POSIX i przydatna do celów demonstracyjnych oraz wspomagających uruchamianie programów, pozwala na przełączenie procesu w tryb uśpienia. Proces pozostanie w tym stanie, dopóki nie odbierze sygnału, który zostanie przez niego obsłużony lub spowoduje zakończenie jego działania:

```
#include <unistd.h>

int pause (void);
```

Funkcja `pause()` kończy swoje działanie jedynie wtedy, gdy może przechwycić odebrany sygnał, co oznacza, że zostanie on obsłużony. W tym przypadku funkcja zwraca `-1` i ustawia zmienną `errno` na wartość `EINTR`. Jeśli jądro zgłasza sygnał ignorowany, proces nie zostaje uaktywniony.

Funkcja `pause()` jest jedną z najprostszych funkcji w jądrze Linuksa. Wykonuje tylko dwa działania. Po pierwsze, przełącza proces w tryb uśpienia możliwy do przerwania. Po drugie, wywołuje funkcję `schedule()`, aby uaktywnić zarządcę procesów Linuksa, który powinien znaleźć inny proces do uruchomienia. Ponieważ proces w rzeczywistości na nic nie czeka, nie zostanie uaktywniony przez jądro, dopóki nie odbierze jakiegoś sygnału. Cała czynność zajmuje jedynie dwie linie kodu w języku C<sup>4</sup>.

<sup>4</sup> Dlatego też funkcja `pause()` jest drugą pod względem prostoty funkcją systemową w Linuksie. Najprostszymi funkcjami są `getpid()` oraz `getgid()`, z których każda zajmuje tylko jedną linię kodu źródłowego.

## Przykłady

Poniżej przedstawione zostaną dwa proste przykłady. Pierwszy z nich rejestruje procedurę obsługi dla sygnału SIGINT, która po prostu wyprowadza wiadomość oraz przerywa działanie programu (co zresztą jest domyślnym zachowaniem tego sygnału):

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

/* procedura obsługi sygnału SIGINT */
static void sigint_handler (int signo)
{
/*
 * Technicznie rzecz ujmując, funkcja printf() nie powinna być używana
 * w procedurze obsługi sygnału, lecz nie należy
 * się tym przesadnie przejmować. W podrozdziale "Współużywalność"
 * wyjaśnimy, dlaczego tak jest.
 */
    printf ("Przechwycono sygnał SIGINT!\n");
    exit (EXIT_SUCCESS);
}

int main (void)
{
/*
 * Zarejestruj funkcję sigint_handler jako procedurę
 * obsługi sygnału SIGINT.
 */
    if (signal (SIGINT, sigint_handler) == SIG_ERR)
    {
        fprintf (stderr, "Nie można obsłużyć sygnału SIGINT!\n");
        exit (EXIT_FAILURE);
    }

    for (;;)
        pause ( );
}

return 0;
```

W kolejnym przykładzie przedstawiono zarejestrowanie tej samej procedury dla sygnałów SIGTERM oraz SIGINT. Oprócz tego dla sygnału SIGPROF zostaje przywrócone domyślne zachowanie (którym jest przerwanie działania procesu), natomiast sygnał SIGHUP zostaje zignorowany (choć mógłby w przeciwnym razie również przerwać proces):

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

/* procedura obsługi sygnałów SIGINT oraz SIGTERM */
static void signal_handler (int signo)
{
    if (signo == SIGINT)
        printf ("Przechwycono sygnał SIGINT!\n");
    else if (signo == SIGTERM)
        printf ("Przechwycono sygnał SIGTERM!\n");
    else
{
```

```

/* to nie powinno się nigdy zdarzyć */
fprintf (stderr, "Nieoczekiwany sygnał!\n");
exit (EXIT_FAILURE);
}

exit (EXIT_SUCCESS);
}

int main (void)
{
/*
* Zarejestruj funkcję signal_handler jako procedurę
* obsługi sygnału SIGINT.
*/
if (signal (SIGINT, signal_handler) == SIG_ERR)
{
    fprintf (stderr, "Nie można obsłużyć sygnału SIGINT!\n");
    exit (EXIT_FAILURE);
}

/*
* Zarejestruj funkcję signal_handler jako procedurę
* obsługi sygnału SIGTERM.
*/
if (signal (SIGTERM, signal_handler) == SIG_ERR)
{
    fprintf (stderr, "Nie można obsłużyć sygnału SIGTERM!\n");
    exit (EXIT_FAILURE);
}

/* Przywróć domyślne zachowanie dla sygnału SIGPROF. */
if (signal (SIGPROF, SIG_DFL) == SIG_ERR)
{
    fprintf (stderr, "Nie można przywrócić domyślnego zachowania dla sygnału
SIGPROF!\n");
    exit (EXIT_FAILURE);
}

/* Zignoruj sygnał SIGHUP. */
if (signal (SIGHUP, SIG_IGN) == SIG_ERR)
{
    fprintf (stderr, "Nie można zignorować sygnału SIGHUP!\n");
    exit (EXIT_FAILURE);
}

for (;;)
    pause ( );

return 0;
}

```

## Uruchomienie i dziedziczenie

Podczas rozwidlania proces potomny dziedziczy od swojego rodzica akcje związane z sygnałami. Oznacza to, że dla każdego sygnału zdefiniowanego dla rodzica następuje skopiowanie do potomka zarejestrowanych akcji (zignorowanie, działanie domyślne, obsługa). Sygnały oczekujące *nie* są dziedziczone. Jest to sensowne, ponieważ takiego typu sygnały zostały wysłane do określonego identyfikatora procesu, lecz zdecydowanie nie do procesu potomnego.

Gdy proces jest tworzony za pomocą rodziny funkcji systemowych `exec`, wszystkim sygnalom zostają przypisane domyślne akcje, chyba że proces rodzicielski je ignoruje; w tym przydzieku nowo utworzony proces również będzie ignorować te same sygnały. Inaczej mówiąc, przypisane domyślne zachowanie, które będzie brane pod uwagę już po wykonaniu tej funkcji. Pozostałe sygnały się nie zmieniają. Ma to sens, gdyż nowo uruchomiony proces nie wspólnie dzieli przestrzeni adresowej ze swoim rodzicem, dlatego też mogą nie istnieć zarejestrowane dla niego procedury obsługi sygnałów. Sygnały oczekujące nie są dziedziczone. W tabeli 10.2 podsumowano metody dziedziczenia.

Tabela 10.2. Sposoby dziedziczenia sygnałów

Działanie sygnału	Dla rozwidleń	Dla funkcji exec
Zignorowanie	Dziedziczenie	Dziedziczenie
Działanie domyślne	Dziedziczenie	Dziedziczenie
Obsługa	Dziedziczenie	Brak dziedziczenia
Sygnały oczekujące	Brak dziedziczenia	Dziedziczenie

To zachowanie, dotyczące uruchamiania procesu, posiada jedno znaczące zastosowanie: gdy powłoka systemowa uruchamia „w tle” jakiś proces (lub gdy proces drugoplanowy uruchamia inny proces), powinien on wówczas ignorować znaki przerwania i wyjścia. W związku z tym zanim powłoka uruchomi proces drugoplanowy, powinna ustawić sygnały SIGINT oraz SIGQUIT na wartość SIG\_IGN. Stąd też często spotykanym rozwiązaniem w programach obsługujących powyższe sygnały, jest sprawdzanie, czy nie są one ignorowane. Na przykład:

```
/* obsłuz sygnał SIGINT tylko wtedy, gdy nie jest on ignorowany */
if (signal (SIGINT, SIG_IGN) != SIG_IGN)
{
    if (signal (SIGINT, sigint_handler) == SIG_ERR)
        fprintf (stderr, "Obsługa sygnału SIGINT nie powiodła się!\n");

/* obsłuz sygnał SIGQUIT tylko wtedy, gdy nie jest on ignorowany */
if (signal (SIGQUIT, SIG_IGN) != SIG_IGN)
{
    if (signal (SIGQUIT, sigquit_handler) == SIG_ERR)
        fprintf (stderr, "Obsługa sygnału SIGQUIT nie powiodła się!\n");
```

Istniejące wymaganie, które nakazuje, aby w celu sprawdzenia sposobu działania sygnału najpierw ustawić jego zachowanie, uwydatnia słabość interfejsu `signal()`. W dalszej części rozdziału przedstawiona zostanie funkcja nieposiadająca takiej wady.

## Odwzorowanie numerów sygnałów na łańcuchy znakowe

W przedstawionych do tej pory przykładach nazwy sygnałów były na stałe wpisane w kodzie programu. Czasem jednak bardziej wygodną (lub nawet wymaganą) operacją jest przekształcenie numeru sygnału na łańcuch znakowy, reprezentujący jego nazwę. Istnieje kilka metod, aby to uzyskać. Jedną z nich jest odczytywanie łańcucha znaków ze statycznie zdefiniowanej listy:

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

Struktura danych `sys_siglist[]` jest tablicą łańcuchów znakowych, przechowującą nazwy sygnałów udostępniane w systemie. Jest indeksowana numerem sygnału.

Alternatywnym rozwiązaniem jest użycie interfejsu `psignal()`, zdefiniowanego w systemie BSD, który jest na tyle popularny, że jest również wspierany przez Linuksa:

```
#include <signal.h>

void psignal (int signo, const char *msg);
```

Wywołanie funkcji `psignal()` wyprowadza na standardowe wyjście błędów `stderr` łańcuch znaków, dostarczony do niej jako parametr `msg`, po którym występuje znak dwukropka, spacja, a wreszcie sama nazwa sygnału z jego numerem podanym w parametrze `signo`. Jeśli parametr `msg` nie zostanie użyty, będzie wyświetlona tylko sama nazwa sygnału. Jeśli parametr `signo` będzie niepoprawny, wówczas informacja o tym znajdzie się w wyprowadzonym tekście.

Lepszym interfejsem jest funkcja `strsignal()`. Nie jest ona zdefiniowana w standardzie, lecz mimo to wspierana przez Linux i wiele innych, nielinuksowych systemów operacyjnych:

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

char *strsignal (int signo);
```

Wywołanie funkcji `strsignal()` zwraca wskaźnik do tekstu, opisującego sygnał reprezentowany przez parametr `signo`. Jeśli wartość `signo` jest niepoprawna, zwrócony opis informuje o tym (w przypadku niektórych systemów uniksowych funkcja zwraca `NULL`). Zwrócony łańcuch znaków pozostaje poprawny jedynie do następnego wywołania funkcji `strsignal()`, gdyż nie obsługuje ona bezpiecznie wątków.

Wykorzystanie tablicy `sys_siglist[]` jest zwykle najlepszym rozwiązaniem. Używając tej metody, można ponownie napisać kod dla wcześniej przedstawionej procedury obsługi sygnałów:

```
static void signal_handler (int signo)
{
    printf ("Przechwycono sygnał %s\n", sys_siglist[signo]);
```

## Wysyłanie sygnału

Funkcja systemowa `kill()`, będąca podstawą popularnego narzędzia `kill`, wysyła sygnał z jednego procesu do drugiego:

```
#include <sys/types.h>
#include <signal.h>

int kill (pid_t pid, int signo);
```

W przypadku jej normalnego użycia (gdy wartość `pid` jest większa od zera), funkcja `kill()` wysyła sygnał `signo` do procesu, identyfikowanego przez parametr `pid`.

Jeśli parametr `pid` jest równy zeru, wówczas sygnał `signo` zostanie wysłany do wszystkich procesów z grupy procesów, do której należy proces wywołujący.

Jeśli parametr `pid` jest równy `-1`, wówczas sygnał `signo` zostanie wysłany do procesów, do których procesowi wywołującemu wolno go wysłać, wyłączając samego siebie i proces `init`. Uprawnienia, określające zasady dostarczania sygnałów, zostaną omówione w następnym podrozdziale.

Jeśli parametr `pid` jest mniejszy od `-1`, wówczas sygnał `signo` zostanie wysłany do grupy procesów o identyfikatorze `-pid`.

W przypadku sukcesu funkcja `kill()` zwraca `0`. Wywołanie funkcji kończy się sukcesem, gdy zostanie wysłany przynajmniej jeden sygnał. W przypadku błędu (gdy nie zostanie wysłany żaden sygnał) funkcja zwraca `-1` oraz odpowiednio ustawia zmienną `errno` na jedną z poniższych wartości:

`EINVAL`

Sygnał, reprezentowany przez parametr `signo`, jest nieprawidłowy.

`EPERM`

Proces wywołujący nie posiada odpowiednich uprawnień, aby wysłać sygnał do dowolnego z żądanych procesów.

`ESRCH`

Proces lub grupa procesów, określona w parametrze `pid`, nie istnieje lub (w przypadku procesu) jest procesem zombie.

## Uprawnienia

Proces musi posiadać właściwe uprawnienia, aby wysłać sygnał do innego procesu. Proces posiadający uprawnienie `CAP_KILL` (zazwyczaj proces administratora) może wysłać sygnał do dowolnego procesu. Jeśli proces wysyłający nie posiada tego uprawnienia, wówczas jego efektywny lub rzeczywisty identyfikator użytkownika musi być równy rzeczywistemu lub zapisanemu identyfikatorowi użytkownika procesu odbierającego. Podsumowując, użytkownik może wysłać sygnał jedynie do takiego procesu, którego jest właścicielem.



W systemach uniksowych (włącznie z Linuksem) zdefiniowany jest wyjątek `SIGCONT`: proces może wysłać ten sygnał do dowolnego procesu w tej samej sesji. Identyfikatory użytkownika dla tych procesów nie muszą być wówczas sobie równe.

Jeśli wartość `signo` jest równa zeru — wcześniej wspomnianemu sygnałowi zerowemu — wówczas funkcja nie wysyła sygnału, lecz wciąż przeprowadza kontrolę błędów. Jest to użyteczne w przypadku, gdy należy sprawdzić, czy proces posiada odpowiednie uprawnienia, aby wysłać lub przetworzyć sygnał.

## Przykłady

Oto, w jaki sposób należy wysłać sygnał `SIGHUP` do procesu o identyfikatorze `1722`:

```
int ret;  
  
ret = kill (1722, SIGHUP);  
if (ret)  
    perror ("kill");
```

Powyższy fragment kodu wykonuje praktycznie taką samą operację jak wywołanie programu użytkowego `kill`:

```
$ kill -HUP 1722
```

Aby sprawdzić, czy istnieją wystarczające uprawnienia pozwalające na wysłanie sygnału do procesu o identyfikatorze `1722`, a jednocześnie nie generując podczas tej operacji żadnego sygnału, należy wykonać poniższy kod:

```
int ret;  
ret = kill (1722, 0);  
if (ret)  
    ; /* nie ma uprawnień */  
else  
    ; /* są uprawnienia */
```

## Wysyłanie sygnału do samego siebie

Wykonanie funkcji `raise()` jest prostą metodą pozwalającą na to, aby proces wysłał sygnał do samego siebie:

```
#include <signal.h>  
  
int raise (int signo);
```

Weźmy pod uwagę następujące wywołanie funkcji:

```
raise(signo);
```

Jest ono równe poniższemu kodowi:

```
kill (getpid ( ), signo);
```

W przypadku sukcesu funkcja zwraca wartość 0, natomiast w przypadku błędu zwraca wartość niezerową. Nie ustawia zmiennej `errno`.

## Wysyłanie sygnału do całej grupy procesów

Inna przydatna funkcja pozwala na łatwe wysłanie sygnału do wszystkich procesów z danej grupy, w przypadku, gdy zanegowanie wartości identyfikatora grupy procesów i użycie `kill()` wydaje się być zbyt skomplikowaną operacją:

```
#include <signal.h>  
  
int killpg (int pgrp, int signo);
```

Weźmy pod uwagę następujące wywołanie powyższej funkcji:

```
killpg (pgrp, signo);
```

Jest ono równe kodowi:

```
kill (-pgrp, signo);
```

Powyższe wywołania działają identycznie nawet w przypadku, gdy parametr `pgrp` wynosi zero, co powoduje, że funkcja `killpg()` wysyła sygnał `signo` do grupy procesów, w której znajduje się proces wywołujący.

W przypadku sukcesu funkcja zwraca wartość 0. W przypadku błędu zwraca -1 oraz odpowiednio ustawia zmienną `errno` na jedną z poniższych wartości:

EINVAL

Sygnał, przekazany w parametrze `signo`, jest nieprawidłowy.

EPERM

Proces wywołujący nie posiada odpowiednich uprawnień, aby wysłać sygnał do wskazanych procesów.

ESRCH

Grupa procesów, określona w parametrze `pgrp`, nie istnieje.

# Współużywalność

Gdy jądro zgłasza sygnał, proces może w tym momencie wykonywać dowolny kod. Na przykład, mógłby znajdować się on w trakcie wykonywania ważnej operacji, która w razie przerwania mogłaby pozostawić go w niespójnym stanie — powiedzmy, struktura danych została tylko w połowie uaktualniona lub obliczenie wykonałoby się tylko częściowo. Proces mógłby nawet w tym czasie obsługiwać inny sygnał.

Procedury obsługi sygnałów nie posiadają informacji, jaki kod jest wykonywany przez proces, gdy pojawia się sygnał; mogą one zostać uruchomione w trakcie wykonywania dowolnej operacji. Dlatego też jest bardzo ważne, aby dowolna procedura obsługi sygnału, która zostaje zarejestrowana przez proces, przeprowadzała swoje działania i używała zmiennych w rozważny sposób. Podczas tworzenia procedur obsługi sygnałów należy zadbać, aby nie czynić żadnych założeń związanych z tym, w jakim miejscu kodu może znajdować się proces podczas jego przerwania. Szczególna ostrożność jest wskazana podczas modyfikowania zmiennych globalnych (to znaczy współdzielonych). Zasadniczo dobrym pomysłem jest założenie, że procedura obsługi zdarzeń nie może modyfikować takich zmiennych. W kolejnym podrozdziale omówiony zostanie sposób, pozwalający na tymczasowe zablokowanie mechanizmu dostarczania sygnałów, będący rozwiązaniem, by bezpiecznie modyfikować dane współdzielone przez procedurę obsługi oraz pozostałą część procesu.

W jaki sposób powyżej wspominany problem związany jest z funkcjami systemowymi i innymi funkcjami bibliotecznymi? Co się stanie, gdy proces jest w trakcie zapisywania do pliku lub przydzielania pamięci, a procedura obsługi sygnału zacznie zapisywać do tego samego pliku lub wywoła funkcję `malloc()`? Jaki będzie skutek, gdy proces jest w trakcie wykonywania funkcji, takiej jak `strsignal()`, która używa bufora statycznego, i nastąpi dostarczenie sygnału?

Niektóre funkcje nie są w oczywisty sposób współużywalne. Jeśli program jest w trakcie wykonywania funkcji, która nie jest współużywalna, a zostanie wygenerowany sygnał i procedura jego obsługi rozpoczęcie wykonywanie tej samej funkcji, wówczas może wkrąść się chaos. *Funkcja współużywalna* jest funkcją, którą można bezpiecznie wywołać z niej samej (lub wspólnie z innego wątku tego samego procesu). Aby funkcja była współużywalna, nie może modyfikować danych statycznych, może natomiast modyfikować jedynie takie dane, które zostały przydzielone na stosie lub dostarczone przez program wywołujący. Funkcja ta nie może również wywoływać żadnych innych funkcji, które nie są współużywalne.

## Funkcje, dla których współużywalność jest zagwarantowana

Podczas tworzenia procedury obsługi sygnału należy założyć, że przerwany proces może być w trakcie wykonywania funkcji, która nie jest współużywalna (lub w trakcie wykonywania innego fragmentu kodu). Dlatego też procedury obsługi sygnałów mogą używać jedynie takich funkcji, które są współużywalne.

W wielu standardach zdefiniowano listy funkcji *bezpiecznych dla sygnałów*, czyli współużywalnych, a więc nadających się do zastosowania w procedurach obsługi. Najbardziej znane z nich, POSIX.1-2003 oraz Single UNIX Specification, definiują listy funkcji, dla których współużywalność jest zagwarantowana we wszystkich zgodnych platformach. W tabeli 10.2. przedstawiono listę tych funkcji.

Istnieje dużo więcej funkcji, które są bezpieczne, lecz w Linuksie oraz innych systemach, zgodnych ze standardem POSIX, współużywalność jest gwarantowana tylko dla powyżej przedstawionych. Przedstawiono je w tabeli 10.3.

Tabela 10.3. Funkcje, które mogą być bezpiecznie współużywalne podczas obsługi sygnałów

abort()	accept()	access()
aio_error()	aio_return()	aio_suspend()
alarm()	bind()	cfgetispeed()
cfgetospeed()	cfsetispeed()	cfsetospeed()
chdir()	chmod()	chown()
clock_gettime()	close()	connect()
creat()	dup()	dup2()
execle()	execve()	Exit()
_exit()	fchmod()	fchown()
fcntl()	fdatasync()	fork()
fpathconf()	fstat()	fsync()
ftruncate()	getegid()	geteuid()
getgid()	getgroups()	getpeername()
getpgid()	getpid()	getppid()
getsockname()	getsockopt()	getuid()
kill()	link()	listen()
lseek()	lstat()	mkdir()
mkfifo()	open()	pathconf()
pause()	pipe()	poll()
posix_trace_event()	pselect()	raise()
read()	readlink()	recv()
recvfrom()	recvmsg()	rename()
rmdir()	select()	sem_post()
send()	sendmsg()	sendto()
setgid()	setpgid()	setsid()
setsockopt()	setuid()	shutdown()
sigaction()	sigaddset()	sigdelset()
sigemptyset()	sigfillset()	sigismember()
signal()	sigpause()	sigpending()
sigprocmask()	sigqueue()	sigset()
sigsuspend()	sleep()	socket()
socketpair()	stat()	symlink()
Sysconf()	tcdrain()	tcflow()
tcflush()	tcgetattr()	tcgetpgrp()
tcsendbreak()	tcsetattr()	tcsetpgrp()
time()	timer_getoverrun()	timer_gettime()
timer_settime()	times()	umask()
uname()	unlink()	utime()
wait()	waitpid()	write()

Istnieje znacznie więcej funkcji, które są bezpieczne, lecz w Linuksie oraz innych systemach zgodnych ze standardem POSIX współprzywalność jest gwarantowana tylko dla powyżej przedstawionych.

## Zbiory sygnałów

Niektóre funkcje, przedstawione w dalszej części rozdziału, operują na zbiorach sygnałów, takich jak grupy sygnałów blokowanych przez proces lub grupy sygnałów oczekujących na obsłuszenie. Takie zbiory sygnałów mogą być przetwarzane za pomocą operacji zbioru sygnałów:

```
#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 (const sigset_t *set, int signo);
```

Funkcja `sigemptyset()` inicjalizuje zbiór sygnałów, podany w parametrze `set`, oznaczając go jako pusty (wszystkie sygnały są wykluczone ze zbioru). Funkcja `sigfillset()` inicjalizuje zbiór sygnałów, podany w parametrze `set`, oznaczając go jako pełny (wszystkie sygnały należą do zbioru). Obie funkcje zwracają zero. Powinny być wywołane dla danego zbioru sygnałów przed jego późniejszym użyciem.

Funkcja `sigaddset()` dodaje sygnał `signo` do zbioru sygnałów przekazanego w parametrze `set`, natomiast funkcja `sigdelset()` usuwa z niego sygnał `signo`. Obie funkcje zwracają zero w przypadku sukcesu, natomiast `-1` w przypadku niepowodzenia, podczas którego ustawiają zmienną `errno` na wartość kodu błędu `EINVAL`, oznaczającego, że `signo` jest niepoprawnym identyfikatorem sygnału.

Funkcja `sigismember()` zwraca `1`, jeśli `signo` znajduje się w zbiorze sygnałów podanym w parametrze `set`. Zwraca `0`, jeśli `signo` nie znajduje się w tym zbiorze lub `-1` w przypadku błędu, podczas którego zmienna `errno` zostaje ponownie ustawiona na wartość `EINVAL`, oznaczającą, że identyfikator `signo` jest niepoprawny.

## Inne funkcje obsługujące zbiory sygnałów

Poprzednio przedstawione funkcje są zdefiniowane w standardzie POSIX i można je znaleźć w każdym nowoczesnym systemie uniksowym. Linux udostępnia również kilka niestandardowych funkcji:

```
#define _GNU_SOURCE
#define <signal.h>

int sigisemptyset (sigset_t *set);

int sigorset (sigset_t *dest, sigset_t *left, sigset_t *right);

int sigandset (sigset_t *dest, sigset_t *left, sigset_t *right);
```

Funkcja `sigisemptyset()` zwraca 1, gdy zbiór sygnałów, reprezentowany przez parametr `set`, jest pusty. W przeciwnym przypadku zwraca 0.

Funkcja `sigorset()` umieszcza unię (sumę binarną) zbiorów sygnałów `left` i `right` w zbiorze `dest`. Funkcja `sigandset()` umieszcza przecięcie (iloczyn binarny) zbiorów sygnałów `left` i `right` w zbiorze `dest`. Obie funkcje zwracają 0 w przypadku sukcesu, a -1 w przypadku niepowodzenia i ustawiają wówczas zmienną `errno` na wartość `EINVAL`.

Funkcje te są użyteczne, lecz nie powinno się ich używać w programach, które wymagają pełnej zgodności ze standardem POSIX.

## Blokowanie sygnałów

We wcześniejszej części tego rozdziału omówiono mechanizm współużywalności oraz problemy, jakie mogą powstawać podczas asynchronicznego działania procedur obsługi sygnałów. Pokazano, że funkcje, które nie są współużywalne, nie mogą być wywoływane z procedury obsługi sygnału.

Co dzieje się w sytuacji, gdy program wymaga współdzielenia danych między procedurą obsługi sygnału a jakimś innym dowolnym fragmentem kodu? Jak wygląda sytuacja w przypadku, gdy istnieją pewne obszary wykonania programu, które nie mogą zostać niczym przerwane, wliczając w to przerwanie spowodowane przez procedury obsługi sygnałów? Takie miejsca w programie zwane są *rejonami krytycznymi* i mogą być one chronione poprzez tymczasowe wstrzymanie dostarczania sygnałów. Sygnały takie to sygnały *zablokowane*. Dowolny sygnał, który zostaje zgłoszony podczas tego stanu, nie będzie obsłużony, dopóki sygnały nie zostaną ponownie odblokowane. Proces może blokować dowolną liczbę sygnałów; zbiory sygnałów blokowanych przez proces zwane są jego *maską sygnałową*.

W standardzie POSIX zdefiniowano, a w systemie Linux zaimplementowano funkcję, która zarządza maską sygnałową procesu:

```
#include <signal.h>

int sigprocmask (int how, const sigset_t *set, sigset_t *oldset);
```

Zachowanie funkcji `sigprocmask()` zależy od wartości parametru `how`, który może być równy jednemu z poniżej podanych znaczników:

`SIG_SETMASK`

Maska sygnałowa procesu wywołującego została zmieniona na wartość `set`.

`SIG_BLOCK`

Sygnały, zdefiniowane w zbiorze sygnałów `set`, zostaną dodane do maski sygnałowej procesu wywołującego. Finalna maska sygnałowa będzie wynikiem operacji unii (binarnej sumy) aktualnej wartości tej maski oraz zbioru `set`.

`SIG_UNBLOCK`

Sygnały, zdefiniowane w zbiorze sygnałów `set`, zostaną usunięte z maski sygnałowej procesu wywołującego. Finalna maska sygnałowa będzie wynikiem operacji przecięcia (binarnego iloczynu) aktualnej wartości tej maski oraz negacji (binarnego odwrócenia bitów) zbioru `set`.

Jeśli parametr oldset nie jest równy NULL, wówczas funkcja umieszcza w nim poprzednią wartość zbioru sygnałów.

Jeśli parametr set jest równy NULL, wówczas funkcja ignoruje parametr how i nie zmienia maski sygnałowej, lecz w dalszym ciągu umieszcza ją w oldset. Przekazanie wartości zerowej w parametrze set jest sposobem na odczytanie aktualnej maski sygnałowej.

W przypadku sukcesu, funkcja zwraca wartość 0. W przypadku błędu, zwraca -1 oraz odpowiednio ustawia zmienną errno na wartość EINVAL, oznaczającą, że parametr how jest niepoprawny, lub na wartość EFAULT, wskazującą, że parametry set lub oldset są błędnymi wskaźnikami.

Blokowanie sygnałów SIGKILL bądź SIGSTOP nie jest dozwolone. Funkcja sigprocmask() w sposób przezroczysty ignoruje próby ich dodania do maski sygnałowej.

## Odzyskiwanie oczekujących sygnałów

Gdy jądro zgłasza sygnał blokowany, nie zostaje on dostarczony. Sygnały takie zwane są *oczekującymi*. Gdy sygnał oczekujący będzie odblokowany, jądro przekaże go do procesu w celu obsłużenia.

POSIX definiuje funkcję, która odzyskuje zbiór oczekujących sygnałów:

```
#include <signal.h>

int sigpending (sigset_t *set);
```

Poprawne wywołanie funkcji sigpending() umieszcza zbiór oczekujących sygnałów w parametrze set oraz zwraca 0. W przypadku błędu, funkcja zwraca -1 oraz ustawia zmienną errno na wartość EFAULT, oznaczającą, że set jest niepoprawnym wskaźnikiem.

## Oczekiwanie na zbiór sygnałów

Trzecia funkcja, zdefiniowana w standardzie POSIX, pozwala procesowi na chwilową zmianę jego maski sygnałowej, a następnie oczekивание, dopóki nie zostanie zgłoszony sygnał, który przerwie działanie tego procesu lub zostanie przez niego obsłużony:

```
#include <signal.h>

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

Jeśli sygnał przerywa działanie procesu, wykonanie funkcji sigsuspend() nie zakończy się. Kiedy sygnał zostanie zgłoszony i obsłużony, wówczas funkcja sigsuspend() zwróci wartość -1, gdy procedura obsługi sygnału się zakończy, jednocześnie ustawiając zmienną errno na wartość EINTR. Jeśli set będzie niepoprawnym wskaźnikiem, wtedy zmienna errno zostanie ustawiona na EFAULT.

Standardowym scenariuszem użycia funkcji sigsuspend() jest odzyskiwanie sygnałów, które mogły nadejść i zostać zablokowane podczas wykonywania sekcji krytycznej w trakcie działania programu. Proces najpierw używa funkcji sigprocmask(), aby zablokować zbiór sygnałów, zapisując poprzednią maskę do parametru oldset. Po wyjściu z rejonu krytycznego proces wywołuje sigsuspend(), przekazując wartość oldset do parametru set.

# Zaawansowane zarządzanie sygnałami

Funkcja `signal()`, która została omówiona na początku tego rozdziału, jest bardzo prosta. Ponieważ jest częścią standardowej biblioteki języka C, dlatego też musi odpowiadać minimalnym wymaganiom dotyczącym możliwości systemu operacyjnego, w którym działa. Powoduje to, że zakres jej działania ograniczony jest jedynie do obsługi najmniejszej wspólnej części zagadnienia zarządzania sygnałami. Jako alternatywę dla niej, standard POSIX definiuje funkcję systemową `sigaction()`, która udostępnia dużo więcej możliwości zarządzania sygnałami. Między innymi można użyć jej do blokowania odbioru niektórych sygnałów, gdy procedura obsługi jest wciąż aktywna, a także do uzyskiwania szerokiego zakresu informacji dotyczących systemu oraz stanu procesu w momencie, gdy został zgłoszony sygnał:

```
#include <signal.h>

int sigaction (int signo, const struct sigaction *act, struct sigaction *oldact);
```

Wywołanie funkcji `sigaction()` zmienia zachowanie sygnału identyfikowanego przez parametr `signo`, który może posiadać dowolną wartość, za wyjątkiem `SIGKILL` oraz `SIGSTOP`. Jeśli parametr `act` nie jest równy `NULL`, wówczas funkcja systemowa zmienia aktualne zachowanie sygnału, który zostaje w nim przekazany. Jeśli parametr `oldact` nie jest równy `NULL`, wówczas funkcja przechowuje w nim poprzednie (lub aktualne, jeśli wartość `act` jest równa `NULL`) zachowanie dla danego sygnału.

Struktura `sigaction` pozwala na dokładną kontrolę zachowania sygnałów. Jest zdefiniowana w pliku nagłówkowym `<sys signal.h>`, dołączonym do pliku `<signal.h>`:

```
struct sigaction
{
    void (*sa_handler)(int); /* procedura obsługi sygnału lub rodzaj akcji */
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask; /* sygnały do zablokowania */
    int sa_flags; /* znaczniki */
    void (*sa_restorer)(void); /* pole przestarzałe i niekompatybilne z POSIX */
}
```

Pole `sa_handler` określa rodzaj akcji, która musi zostać podjęta po otrzymaniu sygnału. Podobnie jak w przypadku funkcji `signal()` pole to może być równe `SIG_DFL`, co oznacza akcję domyślną, `SIG_IGN`, informujące w tym przypadku jądro, aby ignorował sygnał dla procesu lub może być również wskaźnikiem do funkcji obsługi sygnału. Funkcja ta posiada taki sam prototyp jak procedura obsługi sygnału instalowana przez `signal()`:

```
void my_handler (int signo);
```

Jeśli pole `sa_flags` będzie równe znacznikowi `SA_SIGINFO`, wówczas definicję funkcji obsługi sygnału określać będzie pole `sa_sigaction`, a nie `sa_handler`. Prototyp tej funkcji jest trochę inny:

```
void my_handler (int signo, siginfo_t *si, void *ucontext);
```

Funkcja otrzymuje numer sygnału w swoim pierwszym parametrze, a strukturę `siginfo_t` w drugim, natomiast struktura `ucontext_t` (zrzutowana jako wskaźnik do `void`) zostaje przekazana w trzecim parametrze. Funkcja nie posiada wartości powrotnej. Struktura `siginfo_t` dostarcza dużo informacji do procedury obsługi sygnału; zostanie ona wkrótce omówiona.

Należy zauważać, że w przypadku niektórych architektur maszynowych (i prawdopodobnie innych systemów uniksowych) pola `sa_handler` oraz `sa_sigaction` są uni i nie należy przypisywać im jednocześnie wartości.

Pole `sa_mask` dostarcza zestawu sygnałów, które powinny być zablokowane przez system w czasie wykonywania procedury obsługi. Pozwala to programistom na uzyskiwanie prawidłowej ochrony przed współużywalnością pomiędzy różnymi procedurami obsługi sygnałów. Aktualnie obsługiwany sygnał będzie zablokowany, dopóki w polu `sa_flags` nie pojawi się znacznik `SA_NODEFER`. Nie można blokować sygnałów `SIGKILL` oraz `SIGSTOP`; funkcja w sposób niewidoczny będzie je ignorować w polu `sa_mask`.

Pole `sa_flags` jest maską bitową zera, jednego lub większej liczby znaczników, które zmieniają obsługę sygnału podanego w parametrze `signo`. Do tej pory omówiono już dwa znaczniki: `SA_SIGINFO` oraz `SA_NODEFER`; inne wartości pola `sa_flags` składają się również z poniższych znaczników:

#### `SA_NOCLDSTOP`

Jeśli parametr `signo` jest równy `SIGCHLD`, wówczas znacznik ten informuje system, aby nie dostarczał powiadomienia, gdy proces potomny zostaje zatrzymany lub ponownie uruchomiony.

#### `SA_NOCLDWAIT`

Jeśli parametr `signo` jest równy `SIGCHLD`, wówczas znacznik ten uaktywnia automatyczne przechwytywanie potomków: procesy potomne nie zostają zamienione na procesy zombie po zakończeniu ich działania, a proces rodzicielski nie musi (nie może) wywoływać dla nich funkcji `wait()`. Szczegółowe informacje dotyczące procesów potomnych, zombie i funkcji `wait()` znajdują się w rozdziale 5.

#### `SA_NOMASK`

Znacznik ten jest przestarzałym, niezdefiniowanym przez POSIX równoważnikiem znacznika `SA_NODEFER` (omówionego już w tym podrozdziale). Zamiast tego znacznika można podać `SA_NODEFER`, lecz należy być przygotowanym na to, że w starszym kodzie zamiast niego pojawi się `SA_NOMASK`.

#### `SA_ONESHOT`

Znacznik ten jest przestarzałym, niezdefiniowanym przez POSIX równoważnikiem znacznika `SA_RESETHAND` (który zostanie wkrótce omówiony). Zamiast tego znacznika można podać `SA_RESETHAND`, lecz należy być przygotowanym na to, że w starszym kodzie zamiast niego pojawi się `SA_ONESHOT`.

#### `SA_ONSTACK`

Znacznik ten informuje system, aby wywołał daną procedurę obsługi, używając w tym celu alternatywnego stosu sygnałowego udostępnionego przez funkcję `signalstack()`. Jeśli alternatywny stos nie zostanie dostarczony, będzie użyty domyślny — oznacza to, że system zachowa się tak, jak gdyby nie użyto znacznika `SA_ONSTACK`. Alternatywne stose sygnałowe stosuje się rzadko, chociaż są one użyteczne dla pewnych aplikacji Pthreads posiadających stos o mniejszym rozmiarze, który może zostać przepełniony podczas użycia procedur obsługi sygnałów.

#### `SA_RESTART`

Znacznik ten uaktywnia proces ponownego uruchamiania funkcji systemowych, przerwanych przez sygnały, który jest charakterystyczny dla systemu BSD.

#### **SA\_RESETHAND**

Znacznik ten uaktywnia tryb „jednorazowego działania”. Gdy tylko procedura obsługi zakończy swoje działanie, wówczas zachowaniu danego sygnału zostaną przywrócone wartości domyślne.

Pole `sa_restorer` jest przestarzałe i nie jest już używane w systemie Linux. Nie jest również zdefiniowane w standardzie POSIX. Należy przyjąć, że pole to po prostu nie istnieje i w ogóle go nie używać.

Funkcja `sigaction()` zwraca 0 w przypadku sukcesu. W przypadku błędu zwraca -1 oraz odpowiednio ustawia zmienną `errno` na jedną z poniższych wartości:

#### **EFAULT**

Parametry `act` lub `oldact` są niepoprawnymi wskaźnikami.

#### **EINVAL**

Parametr `signo` jest niepoprawnym sygnałem lub jest równy wartościami `SIGKILL` albo `SIGSTOP`.

## **Struktura `siginfo_t`**

Struktura `siginfo_t` jest również zdefiniowana w pliku nagłówkowym `<sys/signal.h>` i wygląda następująco:

```
typedef struct siginfo_t
{
    int si_signo;          /* numer sygnału */
    int si_errno;          /* wartość błędu */
    int si_code;           /* kod sygnału */
    pid_t si_pid;          /* PID dla procesu wysyłającego */
    uid_t si_uid;          /* rzeczywisty UID dla procesu wysyłającego */
    int si_status;         /* kod wyjścia lub sygnał */
    clock_t si_utime;     /* złyty czas użytkownika */
    clock_t si_stime;     /* złyty czas systemowy */
    sigval_t si_value;    /* wartość pola użytkownika dla sygnału */
    int si_int;            /* sygnał POSIX.1b */
    void *si_ptr;          /* sygnał POSIX.1b */
    void *si_addr;         /* obszar pamięci, który spowodował błąd */
    int si_band;           /* zdarzenie poza kolejką */
    int si_fd;             /* deskryptor pliku */
};
```

Struktura ta zawiera dużo informacji, które zostają przekazane do procedury obsługi sygnału (jeśli zamiast pola `sa_sighandler` zostanie użyte pole `sa_sigaction`). Uwzględniając stan nowoczesnej techniki komputerowej, uważa się, że model sygnałów systemu Unix nie pozwala na prostą realizację komunikacji międzyprocesowej (IPC). Być może problemem jest to, że nie potrzebnie używa się funkcji `signal()`, kiedy należałoby stosować raczej funkcję `sigaction()`, uruchamianą z ustawionym znacznikiem `SA_SIGINFO`. Struktura `sigaction_t` pozwala na użycie większej funkcjonalności podczas użycia sygnałów.

Znajduje się w niej wiele użytecznych danych, włącznie z informacjami dotyczącymi procesu, który wysłał sygnał wraz z powodem jego wygenerowania. Oto dokładny opis każdego z pól struktury:

### `si_signo`

Pole to określa numer danego sygnału. Tę samą informację dostarcza pierwszy parametr procedury obsługi sygnału (jego użycie zapobiega odwoływaniu się do wartości dostępnej przez wskaźnik, co ma miejsce w przypadku użycia struktury `siginfo_t`).

### `si_errno`

Jeśli wartość ta jest różna od zera, oznacza kod błędu związany z sygnałem. Pole to jest poprawne dla wszystkich sygnałów.

### `si_code`

Zawiera wyjaśnienie, dlaczego i skąd proces otrzymał dany sygnał (na przykład z funkcji `kill()`). Dopuszczalne wartości tego pola zostaną omówione w następnym podrozdziale. Jest ono poprawne dla wszystkich sygnałów.

### `si_pid`

W przypadku sygnału `SIGCHLD` pole to określa identyfikator procesu, który został przerwany.

### `si_uid`

W przypadku sygnału `SIGCHLD` pole to określa właścielski identyfikator użytkownika dla procesu, który został przerwany.

### `si_status`

W przypadku sygnału `SIGCHLD` pole to określa kod wyjścia dla procesu, który został przerwany.

### `si_utime`

W przypadku sygnału `SIGCHLD` pole to określa czas użytkownika zużyty przez proces, który został przerwany.

### `si_stime`

W przypadku sygnału `SIGCHLD` pole to określa czas systemowy zużyty przez proces, który został przerwany.

### `si_value`

Unia pól `si_int` oraz `si_ptr`.

### `si_int`

Dla sygnałów wysłanych za pomocą funkcji `sigqueue()` (szczegóły w podrozdziale „Wyślanie sygnału z wykorzystaniem pola użytkowego”, znajdującym się w dalszej części rozdziału) pole to oznacza dostarczoną wartość pola użytkowego, przedstawioną w postaci liczby całkowitej.

### `si_ptr`

Dla sygnałów wysłanych za pomocą funkcji `sigqueue()` (szczegóły w podrozdziale „Wyślanie sygnału z wykorzystaniem pola użytkowego”, znajdującym się w dalszej części rozdziału) pole to oznacza dostarczoną wartość pola użytkowego, przedstawioną w postaci wskaźnika `void`.

### `si_addr`

W przypadku sygnałów `SIGBUS`, `SIGFPE`, `SIGILL`, `SIGSEGV` oraz `SIGTRAP` ten wskaźnik `void` zawiera adres powstania błędu. Na przykład, w przypadku `SIGSEGV` pole zawiera adres, w którym została naruszona ochrona pamięci (stąd też często jest on równy `NULL!`).

### `si_band`

W przypadku sygnału `SIGPOLL` pole to zawiera informacje priorytetowe oraz dane poza kolejką dla deskryptora pliku, podanego w polu `si_fd`.

### si\_fd

W przypadku sygnału SIGPOLL pole to jest deskryptorem pliku, dla którego zostały zakończone operacje.

Pola si\_value, si\_int oraz si\_ptr dotyczą złożonej problematyki, ponieważ mogą zostać użyte przez dany proces, aby przesyłać dowolne dane do innego procesu. Dlatego też można ich użyć, aby wysłać zarówno zwykłą liczbę całkowitą, jak również wskaźnik do struktury danych (należy zwrócić uwagę na to, że przesyłanie wskaźnika nie przyda się, jeśli procesy nie współdzielą przestrzeni adresowej między sobą). Pola te zostaną omówione w dalszym podrozdziale „Wysyłanie sygnału z wykorzystaniem pola użytkowego”.

POSIX zapewnia, że tylko pierwsze trzy pola będą poprawne dla wszystkich sygnałów. Inne pola powinny być używane jedynie wówczas, gdy należy obsłużyć odpowiedni sygnał. Na przykład, pole si\_fd należy używać jedynie wówczas, gdy obsługiwany sygnałem jest SIGPOLL.

## Wspaniały świat pola si\_code

Pole si\_code opisuje przyczynę wygenerowania sygnału. Dla sygnałów wysyłanych przez użytkownika pole informuje, w jaki sposób zostały one wysłane. Dla sygnałów wysyłanych przez jądro, wskazuje, dlaczego zostały one wysłane.

Poniżej przedstawione zostaną możliwe wartości pola si\_code, które są poprawne dla wszystkich sygnałów. Oznaczają, w jaki sposób lub z jakiego powodu został wysłany dany sygnał:

### SI\_ASYNCIO

Sygnał został wysłany z powodu zakończenia asynchronicznych operacji wejścia i wyjścia (szczegóły w rozdziale 5.).

### SI\_KERNEL

Sygnał został zgłoszony przez jądro.

### SI\_MESGQ

Sygnał został wysłany z powodu zmiany stanu kolejki wiadomości POSIX (nieuwzględnionej w tej książce).

### SI\_QUEUE

Sygnał został wysłany przez funkcję `sigqueue()` (opis w następnym podrozdziale).

### SI\_TIMER

Sygnał został wysłany w wyniku upłynięcia czasu dla licznika POSIX (szczegóły w rozdziale 11.).

### SI\_TKILL

Sygnał został wysłany przez funkcje `tkill()` lub `tgkill()`. Są one używane w bibliotekach wątkowych i nie zostały uwzględnione w niniejszej publikacji.

### SI\_SIGIO

Sygnał został wysłany z powodu umieszczenia sygnału SIGIO w kolejce.

### SI\_USER

Sygnał został wysłany przez funkcje `kill()` lub `raise()`.

Poniżej zostaną przedstawione możliwe wartości pola si\_code, poprawne jedynie dla sygnału SIGBUS. Oznaczają one rodzaj błędu sprzętowego, który wystąpił:

#### **BUSADRALN**

Proces spowodował powstanie błędu wyrównania (szczegółowe informacje na temat wyrównania znajdują się w rozdziale 9.).

#### **BUSADRERR**

Proces próbował uzyskać dostęp do niepoprawnego adresu fizycznego.

#### **BUSOBJERR**

Proces spowodował powstanie innego rodzaju błędu sprzętowego.

W przypadku sygnału SIGCHLD następujące wartości opisują przyczynę wysłania sygnału od procesu potomnego do rodzica:

#### **CLD\_CONTINUED**

Proces potomny był zatrzymany, lecz został ponownie uruchomiony.

#### **CLD\_DUMPED**

Proces potomny zakończył się w niepoprawny sposób.

#### **CLD\_EXITED**

Proces potomny zakończył się w poprawny sposób poprzez wykonanie funkcji `exit()`.

#### **CLD\_KILLED**

Proces potomny został usunięty.

#### **CLD\_STOPPED**

Proces potomny został zatrzymany.

#### **CLD\_TRAPPED**

Proces potomny natrafił na pułapkę.

Poniżej zostaną przedstawione wartości pola `si_code`, poprawne jedynie dla sygnału SIGFPE. Opisują rodzaj błędu arytmetycznego, który wystąpił:

#### **FPE\_FLTDIV**

Proces wykonał operację zmennoprzecinkową, która zakończyła się dzieleniem przez zero.

#### **FPE\_FLTOVF**

Proces wykonał operację zmennoprzecinkową, która zakończyła się przepelnieniem.

#### **FPE\_FLTINV**

Proces wykonał błędna operację zmennoprzecinkową.

#### **FPE\_FLTRES**

Proces wykonał operację zmennoprzecinkową, która spowodowała powstanie niedokładnego lub błędного wyniku.

#### **FPE\_FLTSUB**

Proces wykonał operację zmennoprzecinkową, która używała indeksu spoza zakresu.

#### **FPE\_FLTUND**

Proces wykonał operację zmennoprzecinkową, która zakończyła się niedomiarem.

#### **FPE\_INTDIV**

Proces wykonał operację całkowitoliczbową, która zakończyła się dzieleniem przez zero.

#### **FPE\_INTOVF**

Proces wykonał operację całkowitoliczbową, która zakończyła się przepelnieniem.

Poniżej zostaną przedstawione możliwe wartości pola `si_code`, które są poprawne jedynie dla sygnału SIGILL. Opisują przyczynę wykonania niepoprawnej instrukcji:

`ILL_ILLADR`

Proces zamierzał użyć niepoprawnego trybu adresowania.

`ILL_ILOPC`

Proces zamierzał wykonać niepoprawny kod operacji.

`ILL_ILOPN`

Proces zamierzał użyć niepoprawnego argumentu.

`ILL_PRVOPC`

Proces zamierzał wykonać uprzywilejowany kod operacji.

`ILL_PTVREG`

Proces miał być wykonany w uprzywilejowanym rejestrze.

`ILL_ILLTRP`

Proces zamierzał wkroczyć do niepoprawnej pułapki.

W przypadku wszystkich powyższych wartości pole `si_addr` zawiera adres powstania błędu.

Poniżej zostaną przedstawione możliwe wartości pola `si_code`, które są poprawne dla sygnału SIGPOLL. Opisują zdarzenie związane z operacjami wejścia i wyjścia, które spowodowało wygenerowanie sygnału:

`POLL_ERR`

Wystąpił błąd operacji wejścia i wyjścia.

`POLL_HUP`

Urządzenie zawiesiło się lub gniazdo zostało odłączone.

`POLL_IN`

Plik posiada dane dostępne do odczytu.

`POLL_MSG`

Wiadomość jest dostępna.

`POLL_OUT`

Można wykonać operację zapisu dla pliku.

`POLL_PRI`

Plik posiada dane o wysokim poziomie priorytetu, które mogą zostać odczytane.

Poniżej zostaną przedstawione możliwe wartości pola `si_code`, które są poprawne dla sygnału SIGSEGV. Opisują dwa rodzaje niewłaściwego dostępu do pamięci:

`SEGV_ACCERR`

Proces próbował w nieprawidłowy sposób uzyskać dostęp do poprawnego obszaru w pamięci — oznacza to, że naruszył uprawnienia dostępu do pamięci.

`SEGV_MAPERR`

Proces próbował uzyskać dostęp do niepoprawnego obszaru pamięci.

W przypadku powyższych dwóch wartości pole `si_addr` zawiera adres powstania błędu.

Poniżej zostaną przedstawione dwie możliwe wartości pola `si_code`, które są poprawne dla sygnału SIGTRAP. Opisują rodzaj pułapki:

#### TRAP\_BRKPT

Proces natrafił na punkt wstrzymania.

#### TRAP\_TRACE

Proces natrafił na pułapkę śledzenia.

Należy zwrócić uwagę na to, że `si_code` jest polem zawierającym wartość, a nie polem bitowym.

## Wysyłanie sygnału z wykorzystaniem pola użytkowego

Jak zostało pokazane w poprzednim podrozdziale, procedury obsługi sygnałów, zarejestrowane przy użyciu znacznika `SA_SIGINFO`, otrzymują parametr o typie `siginfo_t`. Struktura ta zawiera element zwany `si_value`, który jest opcjonalnym polem użytkowym, przekazywanym między źródłem sygnału a jego odbiorcą.

Funkcja `sigqueue()`, zdefiniowana w standardzie POSIX, pozwala procesowi na wysłanie sygnału z wykorzystaniem tego pola użytkowego:

```
#include <signal.h>

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

Funkcja `sigqueue()` działa podobnie jak funkcja `kill()`. W przypadku sukcesu sygnał, który identyfikowany jest przez parametr `signo`, zostaje umieszczony w kolejce dla procesu lub grupy procesów reprezentowanej przez `pid`. Następnie funkcja zwraca 0. Pole użytkowe dla sygnału reprezentowane jest przez parametr `value`, który jest unią liczby całkowitej oraz wskaźnika `void`:

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

W przypadku błędu funkcja zwraca -1 oraz odpowiednio ustawia zmienną `errno` na jedną z poniższych wartości:

#### EAGAIN

Proces wywołujący osiągnął granicę kolejkowania sygnałów.

#### EINVAL

Sygnał, reprezentowany przez parametr `signo`, jest nieprawidłowy.

#### EPERM

Proces wywołujący nie posiada odpowiednich uprawnień, aby wysłać sygnał do dowolnego żądanego procesu. Uprawnienia, niezbędne dla wysłania sygnału, są takie same jak w przypadku funkcji `kill()` (szczegóły w podrozdziale „Wysyłanie sygnału”, znajdującym się we wcześniejszej części rozdziału).

#### ESRCH

Proces lub grupa procesów reprezentowana przez parametr `pid` nie istnieje lub (w przypadku procesu) jest procesem zombie.

Podobnie jak ma to miejsce w przypadku funkcji `kill()`, w parametrze `signo` można przekazać wartość sygnału zerowego (0), aby przetestować uprawnienia.

## Przykład wykorzystania pola użytkowego

Poniższy przykład wysyła do procesu o identyfikatorze 1722 sygnał SIGUSR2, który zawiera pole użytkowe o typie całkowitoliczbowym, równym wartości 404:

```
sigval value;
int ret;

value.sival_int = 404;

ret = sigqueue (1722, SIGUSR2, value);
if (ret)
    perror ("sigqueue");
```

Jeśli proces o identyfikatorze 1722 obsłuży sygnał SIGUSR2 za pomocą procedury używającej znacznika SA\_SIGINFO, wówczas parametr signo zostanie ustawiony na SIGUSR2, pole si->si\_int będzie posiadać wartość 404, natomiast pole si->si\_code będzie równe SI\_QUEUE.

## Ułomność systemu Unix?

Sygnały posiadają złą opinię wśród wielu programistów Uniksa. Są starym rozwiązaniem, nieodpowiednim dla zaimplementowania wymiany informacji między jądrem a użytkownikiem oraz realizują — w najlepszym przypadku — prymitywną formę komunikacji międzyprocesowej (IPC). W świecie wielowątkowych programów i pętli zdarzeń są po prostu anachronizmem. W systemie, który w sposób tak imponujący przetrwał wiele lat i potrafił zachować pierwotne paradygmaty programowania (zdefiniowane podczas jego „narodzin”), sygnały są rzadkim przypadkiem pomyłki.

Mimo tych wad są one mniej lub bardziej potrzebne. Sygnały to jedynie narzędzia umożliwiające odbieranie wielu powiadomień (takich jak powiadomienie o wykonaniu niepoprawnego kodu operacji) z jądra. Za pomocą sygnałów system Unix (a także Linux) przerywa działanie procesów i zarządza relacjami między rodzicami i potomkami. Dlatego też programiści muszą je rozumieć i ich używać.

Jednym z podstawowych powodów unikania użycia sygnałów jest to, że nie istnieje prosty sposób na napisanie poprawnej procedury obsługi, która bezpiecznie obsługuje sytuacje związane ze współużywalnością. Jeśli jednak procedury obsługi sygnałów będą proste i użyte zostaną w nich jedynie takie funkcje, których listę zawiera tabela 10.3. (jeśli w ogóle jakieś będą użyte!), wówczas powinny działać bezpiecznie.

Inną, słabą stroną sygnałów jest fakt, że wielu programistów wciąż używa funkcji `signal()` i `kill()`, zamiast stosować `sigaction()` oraz `sigqueue()`, by nimi zarządzać. Jak pokazano w ostatnich dwóch podrozdziałach, sygnały udostępniają dużo więcej możliwości oraz informacji, gdy użyte zostaną dla nich procedury obsługi wykorzystujące znacznik `SA_SIGINFO`. Pomimo że osobiście nie jestem miłośnikiem sygnałów, trzeba przyznać, że unikanie niepewnych metod oraz wykorzystywanie zaawansowanych interfejsów obsługi sygnałów pozwala na złagodzenie niedogodności związanych z ich użyciem.