

4. Procesy w systemie Linux – interfejs użytkownika

Wstęp

Użytkownicy systemu mają do czynienia ze zbiorem programów systemowych, które umożliwiają wykonywanie typowych działań takich, jak:

- manipulowanie plikami,
- edycja i przetwarzanie plików,
- tworzenie i wykonywanie programów,
- komunikację sieciową,
- informowanie o stanie systemu.

Do programów systemowych zalicza się zarówno tekstowy interpreter poleceń oraz programy tworzące graficzne środowisko pracy użytkownika, jak i przeglądarka WWW, edytor tekstów czy kompilator jakiegoś języka programowania. Ten zestaw programów stanowi rodzaj interfejsu dla użytkowników systemu operacyjnego.

Jedną z podstawowych usług świadczonych przez system na rzecz użytkowników jest wykonywanie programów. Wykonujący się program nosi nazwę procesu. W wykładzie 4 przybliżamy pojęcia procesu, grupy współpracujących procesów oraz sesji procesów korzystających ze wspólnego terminala sterującego. Przedstawiamy metody uruchamiania procesów oraz wpływu na przebieg ich wykonania poprzez zmianę priorytetu i przesyłanie sygnałów. Omawiamy również sterowanie pracami przez interpreter poleceń, umożliwiające wybiórcze wstrzymywanie i wznowianie działających procesów.

4.1. Procesy

Jedną z podstawowych usług każdego systemu operacyjnego jest wykonywanie programów. Program to zbiór instrukcji dla procesora przechowywany na dysku w postaci pliku. Przed uruchomieniem programu jądro systemu musi przydzielić odpowiednie zasoby systemowe i utworzyć środowisko, w którym program będzie wykonany. Takie środowisko wykonania programu określa się mianem **procesu**.

Proces to wykonujący się program.

W odróżnieniu od programu, proces jest obiektem aktywnym. Proces pełni rolę podstawowej jednostki pracy systemu operacyjnego.

Procesy, grupy i sesje

W systemie Linux nowy proces może być utworzony wyłącznie z inicjatywy innego procesu (jedyne wyjątek od tej reguły opisano w dalszej części tego wykładu). Techniczną stroną operacji zajmuje się jądro systemu, ale działa ono jedynie na zlecenie innego procesu. Dlatego powszechnie określa się, że to **proces macierzysty** (rodzic) tworzy **proces potomny** (potomka). Potomek jest tworzony jako kopia rodzica: dziedziczy po nim większość atrybutów oraz otrzymuje kopię jego zasobów systemowych, a niektóre z nich może nawet współdzielić. Od tej chwili obydwa procesy działają niezależnie. Proces macierzysty może nadzorować wykonywanie swoich procesów potomnych. Jądro informuje go o zakończeniu każdego z potomków. Może wtedy odebrać kod wyjścia potomka przechowywany przez jądro. Jeżeli rodzic kończy działanie wcześniej, to osierocone procesy potomne są adoptowane przez proces init.

W celu ułatwienia zarządzania procesy organizowane są przez system w **sesje**. Do każdej sesji przypisany jest terminal sterujący, który procesy wykorzystują do pobierania danych wejściowych i wysyłania wyników. Proces, który utworzył sesję zostaje jej przywódcą (liderem). Zazwyczaj jest to proces interpretera poleceń. Wszyscy jego potomkowie pozostają członkami sesji, dopóki nie zdecydują się zainicjować nowej sesji.

W ramach jednej sesji może powstać wiele **grup** współpracujących procesów. Tworzenie grup umożliwia systemowi utrzymywanie informacji o tym, które procesy współpracują ze sobą. Proces, który założył nową grupę zostaje jej przywódcą, a jego procesy potomne dziedziczą członkostwo w grupie. Procesy mogą dowolnie zmieniać przynależność do grupy w ramach jednej sesji. Wszystkie procesy w danej grupie muszą jednak należeć do tej samej sesji.

Atrybuty procesu

Jądro systemu przechowuje informacje o wszystkich aktywnych procesach. Zapisane są tam atrybuty każdego procesu takie, jak:

1. identyfikator procesu PID,
2. identyfikator procesu macierzystego,
3. identyfikatory grupy procesów i sesji,
4. identyfikator właściciela procesu,
5. stan procesu,
6. priorytet procesu .

Każdy działający proces jest jednoznacznie określony przez swój unikalny identyfikator PID. Proces przechowuje też identyfikator procesu, który go utworzył, czyli swojego procesu macierzystego (rodzica), jak również identyfikatory grupy i sesji, do których należy. Identyfikator właściciela decyduje o uprawnieniach procesu. Wszystkie identyfikatory są nieujemnymi liczbami całkowitymi. Proces w systemie Linux może znaleźć się w jednym z pięciu stanów, których listę przedstawiono w tablicy 4.1.

Tablica 4.1 Stany procesu w systemie Linux

Stan	Znaczenie
Działający (TASK_RUNNING)	Proces gotowy do wykonania lub wykonywany przez procesor.
Uśpiony przerywalny (TASK_INTERRUPTIBLE)	Proces uśpiony w oczekiwaniu na jakieś zdarzenie (np. zakończenie operacji wejścia/wyjścia) z możliwością obudzenia sygnałem.
Uśpiony nieprzerywalny (TASK_UNINTERRUPTIBLE)	Proces uśpiony w oczekiwaniu na jakieś zdarzenie bez możliwości obudzenia sygnałem.
Zombie (TASK_ZOMBIE)	Proces został zakończony, ale jądro wciąż przechowuje informacje dla procesu macierzystego.
Zatrzymany (TASK_STOPPED)	Proces zatrzymany w wyniku śledzenia wykonania lub w wyniku odebrania sygnału.

Priorytet procesu decyduje o kolejności przydziału czasu procesora, a w konsekwencji o kolejności wykonania poszczególnych procesów. Przyjmuje wartości całkowite, nieujemne. W systemie Linux przyjęto konwencję, że większa wartość liczbowa oznacza wyższy priorytet.

Informacje o procesach

Polecenie `ps` umożliwia wyświetlenie listy procesów uruchomionych w systemie i obejrzenie ich atrybutów. Oto jego składnia:

`ps [opcje]`

Wywołane bez opcji, polecenie **ps** wyświetla listę procesów z bieżącej sesji (związanych z bieżącym terminalem sterującym). Podawana informacja obejmuje identyfikator procesu, identyfikator terminala sterującego, skumulowany czas wykonywania procesu oraz nazwę programu, który proces wykonuje. Dobierając odpowiednie opcje możemy sterować zarówno wyborem wyświetlanych procesów, jak i zakresem podawanych informacji. Poniżej przedstawiamy znaczenie wybranych opcji. Pełny wykaz można uzyskać wydając polecenie **man ps**.

Opcje wyboru procesów:

- a** - wszystkie procesy związane z jakimś terminalem sterującym,
- e, -A** - wszystkie procesy,
- u ident_użytk.** - wszystkie procesy wskazanego użytkownika,
- t tty** - wszystkie procesy związane z wskazanym przez argument `tty` terminalem sterującym,
- p PID** - proces o podanym numerze PID.

Opcje zakresu informacji:

- f** - format pełny,
- l** - format długi,
- j** - format zorientowany na pracę,
- s** - format zorientowany na sygnały,
- o** - format zdefiniowany przez użytkownika.

Polecenie **ps** wypisuje żądane informacje w układzie kolumnowym. Kategorie tych informacji są przedstawione w tablicy 4.2:

Tablica 4.2. Kategorie informacji o procesie

Nazwa kolumny	Znaczenie
S	stan procesu
PID	identyfikator procesu
PPID	identyfikator procesu macierzystego
PGID	identyfikator grupy procesów
SID	identyfikator sesji
UID	identyfikator lub nazwa właściciela procesu
CMD	polecenie
PRI	priorytet procesu
NI	wartość parametru nice
TTY	terminal sterujący
STIME	czas uruchomienia procesu
TIME	skumulowany czas wykonywania procesu
%CPU	stopień wykorzystania procesora przez proces
%MEM	stopień wykorzystania pamięci przez proces

Przykład.

Wydanie polecenia `ps -fu root` spowoduje wyświetlenie wszystkich procesów użytkownika **root**, czyli większości procesów systemowych (rys. 4.1).

```

xterm
[zj@venus zj]$ ps -fu root
UID          PID  PPID  C  STIME TTY          TIME CMD
root          1    0  0 May31 ?        00:00:08 init [3]
root          2    1  0 May31 ?        00:00:00 [kflushd]
root          3    1  0 May31 ?        00:00:10 [kupdate]
root          4    1  0 May31 ?        00:00:00 [kpiod]
root          5    1  0 May31 ?        00:00:12 [kswapd]
root          6    1  0 May31 ?        00:00:00 [mdrecoveryd]
root        329    1  0 May31 ?        00:00:07 syslogd -m 0
root        339    1  0 May31 ?        00:00:00 [klogd]
root        370    1  0 May31 ?        00:00:00 [lockd]
root        371   370  0 May31 ?        00:00:00 [rpciod]
root        467    1  0 May31 ?        00:00:01 inetd
root        482    1  0 May31 ?        00:00:00 [xinetd]
root        547    1  0 May31 ?        00:00:00 sendmail: accepting connections
root        563    1  0 May31 ?        00:00:00 crond
root        627    1  0 May31 ?        00:01:19 /usr/local/sbin/sshd
root        635    1  0 May31 tty2      00:00:00 [mingetty]
root        636    1  0 May31 tty3      00:00:00 [mingetty]
root        637    1  0 May31 tty4      00:00:00 [mingetty]
root        638    1  0 May31 tty5      00:00:00 [mingetty]
root        641    1  0 May31 tty6      00:00:00 [mingetty]
root       13848    1  0 Jun07 tty1      00:00:00 [mingetty]
root       12469   627  0 14:29 ?        00:00:02 /usr/local/sbin/sshd
root       12550   467  0 14:53 ?        00:00:00 in.telnetd: giewont.imio.pw.edu.
root       12551 12550  0 14:53 pts/1    00:00:00 login -- zj
root       12571 12552  0 14:54 pts/1    00:00:00 xterm
[zj@venus zj]$

```

Rys 4.1 Przykładowy wynik działania polecenia **ps -fu root**

Listę aktywnych procesów można również uzyskać przy pomocy polecenia **ps tree**. Wyświetla ono informacje w postaci drzewa uwzględniając powiązania rodzinne między procesami (rys.4.2).

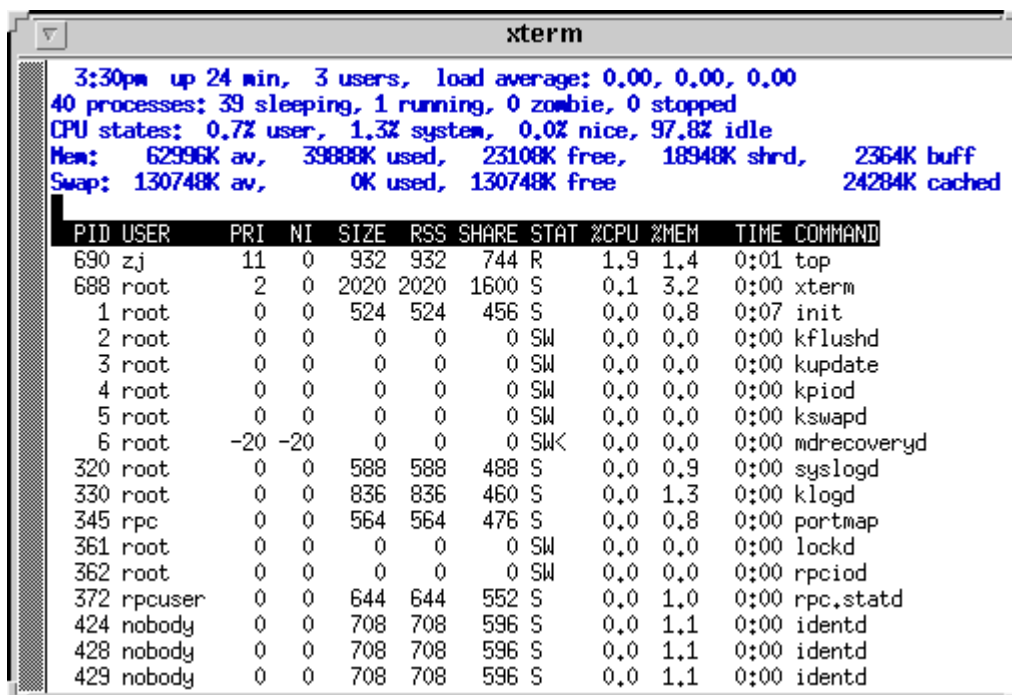
```

xterm
[zj@venus zj]$ pstree
init--+-atd
      |-crond
      |-echous
      |-etcp_s---2*[etcp_s]
      |-identd---identd---3*[identd]
      |-inetd--4*[in.ftpd]
      |   `--in.telnetd---login---bash---xterm---bash---pstree
      |-kflushd
      |-klogd
      |-kpiod
      |-kswapd
      |-kupdate
      |-lockd---rpciod
      |-lpd
      |-mdrecoveryd
      |-6*[mingetty]
      |-portmap
      |-rpc.statd
      |-sendmail
      |-serv_tcp
      |-server
      |-sshd---sshd---bash
      |-syslogd
      |-user_echo
      |-xfs
      `--xinetd
[zj@venus zj]$ pstree zj
bash---xterm---bash---pstree
[zj@venus zj]$

```

Rys 4.2 Przykładowy wynik działania polecenia **ps tree**

Polecenie **top** szereguje wyświetlane procesy według stopnia wykorzystania zasobów systemowych, na bieżąco aktualizując informacje (rys. 4.3).



PID	USER	PRI	NI	SIZE	RSS	SHARE	STAT	%CPU	%MEM	TIME	COMMAND
690	zj	11	0	932	932	744	R	1.9	1.4	0:01	top
688	root	2	0	2020	2020	1600	S	0.1	3.2	0:00	xterm
1	root	0	0	524	524	456	S	0.0	0.8	0:07	init
2	root	0	0	0	0	0	SW	0.0	0.0	0:00	kflushd
3	root	0	0	0	0	0	SW	0.0	0.0	0:00	kupdate
4	root	0	0	0	0	0	SW	0.0	0.0	0:00	kpiod
5	root	0	0	0	0	0	SW	0.0	0.0	0:00	kswaped
6	root	-20	-20	0	0	0	SW<	0.0	0.0	0:00	mdrecoveryd
320	root	0	0	588	588	488	S	0.0	0.9	0:00	syslogd
330	root	0	0	836	836	460	S	0.0	1.3	0:00	klogd
345	rpc	0	0	564	564	476	S	0.0	0.8	0:00	portmap
361	root	0	0	0	0	0	SW	0.0	0.0	0:00	lockd
362	root	0	0	0	0	0	SW	0.0	0.0	0:00	rpciod
372	rpcuser	0	0	644	644	552	S	0.0	1.0	0:00	rpc.statd
424	nobody	0	0	708	708	596	S	0.0	1.1	0:00	identd
428	nobody	0	0	708	708	596	S	0.0	1.1	0:00	identd
429	nobody	0	0	708	708	596	S	0.0	1.1	0:00	identd

Rys. 4.3 Przykładowy wynik działania polecenia **top**

Przegląd procesów systemowych

W chwili startu systemu Linux tworzony jest proces **INIT_TASK** o numerze PID=0. Jest to jedyny proces tworzony przez jądro systemu, a nie przez inny proces. Jego jedynym zadaniem jest utworzenie procesu **init** o numerze PID=1. Proces **INIT_TASK** nigdy nie pojawia się na liście wyświetlanych procesów.

W innych systemach uniksowych proces o PID=0 otrzymuje również inne zadanie i jest uwzględniany na liście aktywnych procesów pod nazwą **sched** lub **swapper**.

Proces **init** uruchamia pozostałe procesy systemowe zgodnie z zawartością odpowiednich plików konfiguracyjnych (m.in. pliku `/etc/inittab`). Procesy utworzone przez **init**a pełnią rolę serwerów różnych usług systemowych i noszą nazwę demonów, np.:

- **syslogd** - demon przechowywania komunikatów systemowych,
- **klogd** - demon przechowywania komunikatów jądra,
- **lockd** - demon zajmowania plików i rekordów,
- **crond** - demon zegarowy,
- **inetd** - demon internetowy, który na żądanie uruchamia poszczególne usługi sieciowe, czyli uruchamia odpowiednie demony,
- **sendmail** - demon poczty elektronicznej.

Demony nie mają terminala sterującego i przez większość czasu pozostają uśpione w oczekiwaniu na wystąpienie żądania udostępnienia danej usługi.

Procesy związane z sesją użytkownika

W otwarciu **tekstowej sesji** użytkownika zaangażowane są tylko 3 procesy:

- **getty**,
- **login**,
- **bash** (lub inny interpreter poleceń, np. **sh**, **tcsh**) .

Proces **getty** to proces obsługi terminala. Określa on parametry początkowe linii terminala i oczekuje na zgłoszenie użytkownika po czym uruchamia proces **login**.

Proces **login** to proces autoryzacji użytkowników. Jego rola polega na pobraniu nazwy i hasła użytkownika, dokonania autoryzacji a po jej pomyślnym zakończeniu uruchomieniu procesu interpretera poleceń np. **sh**, **csch**, **bash**, **tcsh**, **zsh**.

Sesja graficzna wymaga dodatkowo uruchomienia wielu innych procesów, a w szczególności serwera systemu X Window (np. XFree86), menedżera okien (np. **fvwm2**) oraz najczęściej systemu aktywnego pulpitu (np. KDE lub GNOME).

4.2. Uruchamianie procesów

Użytkownik może uruchomić nowy proces na 2 sposoby:

- wydając odpowiednie polecenie powłoce,
- wywołując program w środowisku graficznym (przez wybranie odpowiedniej pozycji z menu lub przyciśnięcie ikony programu).

Uruchamianie procesów przez polecenia powłoki

Wywołując program w powłoce można uruchomić proces na **pierwszym planie** (ang. *foreground*) lub w **tle**, czyli na drugim planie (ang. *background*).

Proces pierwszoplanowy jest uprzywilejowany w dostępie do terminala sterującego, może pobierać dane z terminala i wypisywać wyniki. Do momentu zakończenia procesu nie można wydawać powłoce żadnych poleceń. Proces pierwszoplanowy uruchamia się przez podanie w powłoce nazwy polecenia z argumentami:

```
polecenie arg1 arg2 arg3 ...
```

Przykład

```
ps -fu root
```

Proces drugoplanowy nie może komunikować się z terminalem. Sterowanie powraca do procesu powłoki, dzięki czemu można wydawać kolejne polecenia. W celu uruchomienia procesu w tle należy zakończyć polecenie znakiem **&**:

```
polecenie arg1 ... &
```

Przykład

```
firefox &
```

Interpreter poleceń umożliwia także jednoczesne uruchomienie grupy współpracujących procesów, tworzących **potok**:

```
polecenie1 | polecenie2 | polecenie3 ...
```

Przykład

```
ps -ef | more
```

Każdy z tych procesów przesyła swoje dane wyjściowe do następnego procesu w potoku zamiast do terminala sterującego. W podanym przykładzie lista procesów wyprodukowana przez **ps** zostaje przesłana do procesu **more**, który wyświetla ją strona po stronie.

W przypadku przerwania połączenia z terminalem sterującym (w wyniku zakończenia sesji użytkownika lub zerwania łączności fizycznej) system usiłuje zakończyć wszystkie procesy z sesji związanej z terminalem, wysyłając sygnał SIGHUP (patrz punkt 4.3). Zapobiega to pozostawianiu niepotrzebnych procesów w systemie po wylogowaniu użytkownika. Istnieje jednak możliwość świadomego pozostawienia procesu działającego po zakończeniu sesji w celu wykonania jakichś operacji (np. długotrwałych obliczeń, zdalnego kopiowania dużych plików):

```
nohup polecenie arg ... [&]
```


Polecenie **nohup** uruchamia proces w taki sposób że ignoruje on sygnał SIGHUP oraz kieruje wyjście procesu do pliku **nohup.out** zamiast na terminal. Takie procesy uruchamiane są zwykle w tle.

Zmiana priorytetu procesu

Użytkownik nie może ustawić bezpośrednio wartości **priorytetu** procesu. Bieżąca wartość priorytetu obliczana jest na podstawie wartości parametru **nice** oraz czasu wykorzystywania procesora. Wartość parametru **nice** określa stopień "uprzejmości" procesu wobec innych procesów i przyjmuje wartość początkową 0. Zwiększanie tej wartości powoduje obniżanie priorytetu.

Zwykły użytkownik może jedynie **obniżyć** priorytet własnych procesów, **zwiększając** wartości parametru **nice**.

Polecenie **nice** umożliwia uruchomienie nowego procesu z obniżonym priorytetem:

```
nice [-zmiana] polecenie [arg ...]
```

Argument **zmiana** przyjmuje wartości z przedziału **<-20, 19>** i określa zmianę wartości parametru **nice** dla procesu. Zwykły użytkownik może użyć wartości argumentu **zmiana** z przedziału **<0, 19>**, pełen zakres wartości dostępny jest tylko dla administratora.

Możliwa jest również zmiana priorytetu działającego już procesu za pomocą polecenia **renice**:

```
renice wartość [[-p] pid ...] [[-g] pgrp ...] [[-u] użytkownik ...]
```

Polecenie **renice** ustawia nową wartość parametru **nice** z przedziału **<-20, 19>** dla procesów specyfikowanych przez:

- identyfikator **pid**,
- identyfikator grupy procesów **pgrp**,
- nazwę użytkownika będącego właścicielem procesów.

Podobnie jak w przypadku polecenia **nice**, zwykły użytkownik przy pomocy **renice** może jedynie obniżyć priorytet własnych procesów.

4.3. Sygnały

Sygnał stanowi asynchroniczną informację dla procesu. Umożliwia asynchroniczne przerwanie działania procesu w celu poinformowania go o określonym zdarzeniu. Po obsłużeniu sygnału proces wznowia działanie od miejsca przerwania.

Dostarczaniem sygnału do procesu zawsze zajmuje się jądro systemu, ale informacja może być wysłana przez samo jądro, przez inny (a nawet ten sam) proces lub przez użytkownika. Zastosowanie sygnałów może być bardzo szerokie, na przykład:

- jądro może poinformować proces o wykonaniu niedozwolonej operacji, zakończeniu jednego z procesów potomnych lub zajściu innego zdarzenia, na które proces oczekuje,
- proces współpracujący może poinformować o zakończeniu obliczeń a sam proces może ustawić sobie czas pobudki,
- użytkownik może spowodować wstrzymanie lub zakończenie niewłaściwie działającego procesu.

Przegląd sygnałów

W systemie Linux zdefiniowane są 32 sygnały. Każdy sygnał ma unikalną nazwę i numer, przy czym numery niektórych sygnałów mogą się zmieniać w zależności od implementacji. W tabelicy 4.3 zebrano informacje o najczęściej używanych sygnałach.

Tablica 4.3. Sygnały w systemie Linux

Nazwa sygnału	Numer sygnału	Znaczenie	Domyślna reakcja procesu
SIGHUP	1	Zerwanie połączenia z terminalem sterującym	Zakończenie procesu
SIGINT	2	Przerwanie	Zakończenie procesu
SIGQUIT	3	Zakończenie procesu	Zakończenie procesu i zrzut obrazu pamięci do pliku core
SIGILL	4	Nielegalna instrukcja w kodzie	Zakończenie procesu
SIGKILL	9	Zabicie procesu	Bezwzględne zakończenie procesu
SIGSEGV	11	Przekroczenie dopuszczalnego adresu pamięci	Zakończenie procesu
SIGALRM	14	Alarm programowy	Ignorowanie
SIGTERM	15	Przerwanie programowe	Zakończenie procesu
SIGCONT	<i>zależny od implementacji</i>	Wznowienie wykonywania procesu	Wznowienie wykonywania procesu
SIGSTOP	<i>zależny od implementacji</i>	Bezwzględne zatrzymanie procesu	Bezwzględne wstrzymanie wykonywania procesu

SIGTSTP	<i>zależny od implementacji</i>	Zatrzymanie procesu w wyniku wprowadzenie z terminala sekwencji Ctrl-z	Wstrzymanie wykonywania procesu
SIGCHLD	<i>zależny od implementacji</i>	Informacja o zakończeniu (śmierci) procesu potomnego	Ignorowanie
SIGUSR1	<i>zależny od implementacji</i>	Sygnał definiowany przez użytkownika	Niezdefiniowana
SIGUSR2	<i>zależny od implementacji</i>	Sygnał definiowany przez użytkownika	Niezdefiniowana

Wysyłanie sygnałów

Użytkownik może wysłać sygnał do procesu posługując się poleceniem **kill**:

```
kill [-sygnał] pid
```

gdzie:

sygnał - to numer (np. 9) lub nazwa sygnału (np. KILL),

pid - to identyfikator procesu (PID).

W przypadku pominięcia argumentu **sygnał**, domyślnie wysyłany jest sygnał **SIGTERM**.

Zwykły użytkownik może wysyłać sygnały tylko do swoich procesów, natomiast administrator może wysyłać dowolne sygnały do wszystkich procesów.

W celu wysłania niektórych sygnałów do grupy procesów pierwszoplanowych można posłużyć się specjalnymi sekwencjami klawiszy, zdefiniowanymi w powłoce. Zwyczajowe ustawienia przedstawiono w tablicy 4.4.

Tablica 4.4. Sekwencje znaków pozwalające na wysłanie sygnałów

Sekwencja	Wysyłany sygnał
Ctrl-c	SIGINT
Ctrl-\	SIGQUIT
Ctrl-z	SIGTSTP

Obsługa sygnałów w procesie

Po otrzymaniu sygnału proces musi zareagować w sposób zdefiniowany w kodzie programu. Istnieją 3 sposoby obsługi sygnałów:

- domyślna reakcja ze strony jądra systemu (patrz Tabl. 4.3),
- ignorowanie sygnału,
- przechwycenie sygnału i wykorzystanie własnej funkcji obsługi zdefiniowanej w programie.

Każdy nowy proces przejmuje (dziedziczy) ustawienia sposobu obsługi sygnałów od swojego procesu macierzystego. Te ustawienia mogą być potem zdefiniowane w kodzie programu.

Możliwe jest również blokowanie przez proces odbierania poszczególnych sygnałów. W takim przypadku jądro przechowuje nadesłane do procesu sygnały do momentu odblokowania ich odbioru.

Sygnały SIGKILL i SIGSTOP muszą być obsługiwane w sposób domyślny, a więc nie mogą być blokowane, ignorowane ani przechwytywane. Zapewnia to możliwość wstrzymania bądź zakończenia niewłaściwie działającego procesu.

Polecenie **trap** umożliwia ustawienie obsługi poszczególnych sygnałów przez powłokę. Ustawienia te **nie zostaną** przekazane nowym procesom uruchamianym przez powłokę.

```
trap [polecenie] [sygnał]
```

gdzie:

sygnał - to numer (np. 9) lub nazwa sygnału (np. KILL),

polecenie - to nazwa polecenia (z argumentami), które zostanie wykonane po odbiorze podanego sygnału.

Polecenie **trap** rejestruje wskazane w argumencie polecenie do obsługi podanego sygnału. Podanie pustego napisu zamiast argumentu **polecenie** spowoduje ustawienie ignorowania sygnału. Pominięcie argumentu **polecenie** przywróci domyślny sposób obsługi. Pominięcie obydwu argumentów spowoduje wypisanie wszystkie poleceń przypisanych do sygnałów.

4.4. Sterowanie pracami

Pojęcie sterowanie pracami oznacza możliwość selektywnego wstrzymywania i wznowiania procesów na pierwszym planie lub w tle.

Odpowiednich mechanizmów dostarcza tu powłoka oraz program emulacji terminala. Powłoka łączy pracę z każdym procesem lub grupą procesów tworzących potok. W ramach jednej sesji może istnieć jedna praca pierwszoplanowa i wiele prac drugoplanowych lub wstrzymanych.

Listę wszystkich prac wraz z ich numerami można uzyskać poleceniem **jobs**:

```
jobs [-l]
```

Polecenie wyświetla numery i nazwy prac. Opcja **-l** umożliwia dodatkowo uzyskanie numerów PID procesów.

Wstrzymanie wykonywania procesu następuje w wyniku odebrania odpowiedniego sygnału. Procesy pierwszoplanowe można zatrzymać wysyłając do nich sygnał SIGTSTP poprzez sekwencję znaków **Ctrl-z**. Do procesów drugoplanowych możemy wysłać sygnał poleceniem **kill**:

```
kill [-sygnał] %praca
```

gdzie:

praca - to numer pracy lub ciąg znaków identyfikujący nazwę pracy.

Wznawianie procesów na pierwszym planie i w tle realizują odpowiednio polecenia **fg** i **bg** (nazwy pochodzą od angielskich słów *foreground* i *background*):

```
fg [%praca]
```

```
bg [%praca]
```

Wywołanie poleceń bez argumentu powoduje wznowienie bieżącej pracy tzn. takiej, która jako ostatnia została wstrzymana lub uruchomiona w tle. Dostępne są też argumenty **+** i **?** oznaczające odpowiednio bieżącą i poprzednią pracę.

Przeniesienie działającego procesu z pierwszego planu do tła można zrealizować w następujący sposób:

- zatrzymać proces sekwencją **Ctrl-z**,
- wznowić proces w tle poleceniem **bg**.

Działanie odwrotne wymaga jedynie użycia polecenia **fg**.

Przykład.

Poniżej, na Rys. 4.4, pokazano przykład sterowania pracami. Na wstępie użytkownik uruchomił w pierwszym planie program **netscape**. Następnie wysłał do tego procesu sygnał **SIGTSTP** naciskając na klawiaturze kombinację **Ctrl-z**. Wykonywanie procesu zostało zatrzymane, co potwierdzone zostało przez komunikat wypisany na terminalu. W dalszej kolejności użytkownik sprawdził przy pomocy polecenia **jobs -l** jakie prace istnieją aktualnie w tej powłoce, po czym wznowił wykonywanie procesu **netscape** w tle (poleceniem **bg**) i sprawdził, czy proces rzeczywiście się wykonuje (poleceniem **jobs -l**). Potem uruchomił w tle dwa nowe procesy: nowy terminal (poleceniem **xterm &**) oraz zegar (poleceniem **xclock &**). Na koniec użytkownik ponownie wypisał listę prac w powłoce.

```
xterm
[zj@venus zj]$ netscape
[1]+  Stopped                  netscape
[zj@venus zj]$
[zj@venus zj]$ jobs -l
[1]+  761 Stopped              netscape
[zj@venus zj]$
[zj@venus zj]$ bg
[1]+  netscape &
[zj@venus zj]$
[zj@venus zj]$ jobs -l
[1]+  761 Running              netscape &
[zj@venus zj]$ xterm &
[2] 779
[zj@venus zj]$ xclock &
[3] 781
[zj@venus zj]$
[zj@venus zj]$ jobs -l
[1]  761 Running              netscape &
[2]-  779 Running              xterm &
[3]+  781 Running              xclock &
[zj@venus zj]$
```

Rys.4.4 Przykład sterowania pracami

Bibliografia

- [1] Glass G., Ables K.: Linux dla programistów i użytkowników, Wydawnictwo Helion 2007 (rozdziały: 5.9-5.11, 5.18)
- [2] Johnson M.K., Troan E.W.: Oprogramowanie użytkowe w systemie Linux, WNT 2000 (rozdziały: 9.1, 9.2, 9.6, 13.1, 14.1),

Słownik

Termin	Objaśnienie
grupa procesów	zbiór procesów współpracujących ze sobą w ramach jednej sesji
potok	grupa współpracujących procesów, z których każdy przesyła swoje dane wyjściowe do następnego procesu w potoku zamiast do terminala sterującego
praca	utworzony przez powłokę proces lub grupa procesów połączonych w potok
proces	wykonujący się program, podstawowa jednostka pracy systemu operacyjnego
sesja	zbiór procesów korzystających z tego samego terminala sterującego
sygnał	asynchroniczna informacja dla procesu

Zadania do wykładu 4

Zadanie 1

Obejrzyj listę wszystkich procesów w systemie przy pomocy polecenia **ps**. Zidentyfikuj własne procesy i narysuj drzewo dziedziczenia od procesu **init** do tych procesów. Zwróć uwagę na stan poszczególnych procesów.

Czy struktura procesów jaką otrzymałeś zgadza się z wynikiem działania polecenia **pstree**?

Zadanie 2

Zmień domyślną obsługę sygnału SIGINT w bieżącej powłoce.

Zadanie 3

Uruchom kilka prac (np. `sleep n`, gdzie `n` - liczba sekund) i sprawdź działanie poleceń `jobs`, `fg`, `bg` wykorzystując różne argumenty wywołania.

Czy już potrafisz przenieść proces z pierwszego planu do tła i z powrotem ?

Zadanie 4

Uruchomić w tle proces (np. `sleep 100`) z obniżonym priorytetem. Następnie zmienić priorytet tego procesu.

Czy priorytet procesu można zmieniać dowolnie?

Zadanie 5

Sprawdź reakcję procesów działających na pierwszym planie i w tle na wybrane sygnały. Wykorzystaj polecenie `kill` oraz sekwencje sterujące `Ctrl`-znak.