5. Procesy

5.1. Proces

Proces to **uruchomiony program**. Każdemu procesowi obecnemu w systemie towarzyszy zestaw opisujących go atrybutów, na które w przypadku systemów klasy Unix/Linux składają się:

- identyfikator procesu (**PID** – od ang. *process identifier*) – unikalna w danym cyklu życia systemu operacyjnego, dodatnia liczba całkowita, jednoznacznie identyfikująca proces
- identyfikator procesu rodzicielskiego (**PPID** – od ang. *parental PID*) – wartość PID tego procesu, który powołał do życia dany proces (w przeciwieństwie do świata realnego, do wykreowania procesu potomnego potrzebny jest dokładnie jeden rodzic)
- środowisko procesu – zbiór zmiennych środowiskowych, przechowywany w systemie oddzielnie dla każdego procesu; środowisko to powstaje poprzez skopiowanie środowiska procesu rodzicielskiego w chwili tworzenia procesu potomnego; zmiany wykonane przez proces potomny w swoim egzemplarzu środowiska nie są widoczne w procesie macierzystym
- właściciel – użytkownik, identyfikowany przez swój UID, który uruchomił (lub w imieniu którego uruchomiono) dany proces; przechowywanie tej danej

pozwala na wprowadzenie mechanizmów ochrony, dzięki którym nieuprawniony użytkownik nie będzie w stanie wpływać na stan procesów innych niż własne
- priorytet – wartość reprezentująca miejsce zajmowane w hierarchii decydującej o sposobie, w jaki proces jest traktowany przez system; jest liczbą z przedziału od -20 (najwyższy priorytet) do 19 (najniższy priorytet); procesy o wyższym priorytecie mogą być preferowane w dostępie do zasobów systemu, w szczególności do przydziału procesora; nowsze jądra systemu Linux mogą również dopuszczać specjalną wartość priorytetu oznaczaną jaką **rt** (od ang. *real-time*) – procesy o takim priorytecie mają absolutne pierwszeństwo wykonania.

Fakt, iż każdy proces powiązany jest ze swoim rodzicem, powoduje w konsekwencji, że wszystkie obecne w systemie procesy można utożsamiać z węzłami drzewa, którego korzeniem jest proces, który został wystartowany przez jądro jako pierwszy. W starszych wydaniach systemu Linux procesem tym jest *init*, w nowszych – *systemd*, co ma związek z dokonaną gruntowaną zmianą modelu zarządzania uruchamianiem i zamykaniem usług w systemie operacyjnym. Proces uruchomiony jako pierwszy ma zawsze PID równy 1, a jego PPID wynosi 0, co oznacza, że *init* i *systemd* są *de facto* sierotami i tym samym są przodkami wszystkich pozostałych procesów.

Historyczna nazwa procesu *init* jest nieodłącznie powiązana z mechanizmem uruchamiania systemu, nazywanym tradycyjnie *sysvinit* (od ang. *System V Init*, co jest nawiązaniem do Uniksa wersji V, w którym mechanizm ten został zaimplementowany po raz pierwszy). W większości współczesnych dystrybucji Linuksa standard ten jest został zastąpiony przez nowszy i uniwersalniejszy mechanizm pod nazwą *systemd*.

Wiele z procesów jest uruchamianych automatycznie w czasie startu systemu. Zwykle ich właścicielem jest użytkownik *root*, co zapewnia, że procesy żywotne dla funkcjonowania systemu nie zostaną zatrzymane przez zwykłych użytkowników. Od momentu zalogowania się użytkownika do systemu kolejne uruchamiane przez niego procesy będą już należeć do niego i tym samym będzie on mógł decydować o ich losie.

5.2. Prezentowanie stanu procesów

5.2.1. Polecenie `ps`

Polecenie **ps** (od ang. *process status*) wyprowadza na standardowe wyjście listę procesów obecnych w systemie w momencie pracy polecenia (jest to tak zwana *migawka stanu procesów*). Lista prezentowana jest jako tabela, w kolumnach której prezentowane są różne własności każdego prezentowanego procesu, na przykład:

```
$ ps
  PID TTY          TIME CMD
 1046 pts/1    00:00:00 zsh
 1601 pts/1    00:00:00 ps
```

Ze względu na skomplikowaną historię rozwoju polecenia **ps**, zestaw dopuszczalnych przełączników jest dość pogmatwany. Można uznać, że obecnie implementowane wersje **ps** używają równolegle co najmniej trzech różnych form specyfikowania przełączników – są to:

– *styl UNIX* – przełączniki mogą być grupowane i są poprzedzane łącznikiem, na przykład
ps -ef
– *styl BSD* – przełączniki mogą być grupowane i nie są poprzedzane łącznikiem, na przykład
ps aux
– *styl GNU* – przełączniki nie mogą być grupowane i są poprzedzane podwójnym łącznikiem, na przykład
ps --headers

W dalszej części opisu będziemy posługiwać się wyłącznie przełącznikami w stylu UNIX. Oto najprzydatniejsze z nich:

-e
wyprowadź informację o wszystkich procesach w systemie (domyślnie prezentuje się tylko procesy tego użytkownika, który uruchomił polecenie **ps**)
-p *n*
wyprowadź informację tylko o procesie z PID równym *n*
-l
forma długa, prezentująca bogatszy zestaw informacji o procesach
-u *lista*
wyświetla tylko procesy użytkowników wymienionych na liście *lista*, na przykład: **-u root,user**
-f
pełna postać zestawienia danych o procesach
-F
postać danych pełniejsza niż pełna

Repertuar prezentowany danych zależy od użytych przełączników. Poniżej przedstawiono skrótowe tytuły kolumn w formach używanych przez **ps**:

– **%CPU**

względne użycie czasu procesora obliczane jako procentowy iloraz faktycznego czasu użycia procesora do czasu przebywania procesu w systemie
- **%MEM**
względne użycie pamięci operacyjnej obliczane jako procentowy iloraz rozmiaru pamięci zajętej przez proces do rozmiaru pamięci dostępnej w systemie
- **COMMAND**
linia poleceń (wraz z argumentami), którą uruchomiono dany proces
- **START**
czas uruchomienia procesu
- **TIME**
czas procesora wykorzystany przez proces
- **C**
procentowe wykorzystanie procesora (wartość całkowita z **%CPU**)
- **NI**
wartość *niceness* procesu (pojęcie to omówimy niebawem)
- **F**
różne opcje procesu (od ang. *flags*)
- **PID**
nie wymaga wyjaśnień
- **PPID**
jak wyżej
- **PRI**
priorytet procesu
- **RSS** (od ang. *resident set size*) rozmiar pamięci procesu stale rezydującej w systemie (tzn. nie podlegającej wymianie w ramach pracy pamięci wirtualnej)
- **S**
status procesu
- **SZ**
rozmiar całkowity pamięci użytej przez proces (wraz z pamięcią wirtualną)
- **TTY**
terminal skojarzony z procesem (najczęściej ten, na którym zalogowany jest właściciel procesu)
- **UID**
identyfikator właściciela procesu
- **USER**
nazwa właściciela procesu
- **WCHAN**
zdarzenie, na zakończenie którego czeka proces (jeśli jest w stanie czekania); może to być nazwa usługi systemowej, wewnątrz której czeka proces

5.2.2. Polecenie `pstree`

Polecenie **`pstree`** wyprowadza na swoje standardowe wyjście graficzne zobrazowanie drzewa procesów obecnych w systemie, na przykład:

```
$ pstree
init─┬─atd
     ├─avahi-daemon───avahi-daemon
     ├─colord───{colord}
     ├─colord-sane───2*[{colord-sane}]
     ├─console-kit-dae───64*[{console-kit-dae}]
     ├─cron
     ├─cupsd
     ├─dbus-daemon
     ├─dnsmasq
     ├─6*[getty]
     ├─lighttpd
     ├─master─┬─pickup
     │        └─qmgr
     ├─nmbd
     ├─ntpd
     ├─polkitd───{polkitd}
     ├─rpc.idmapd
     ├─rpc.statd
     ├─rpcbind
     ├─rsyslogd───3*[{rsyslogd}]
     ├─smbd───3*[smbd]
     ├─sshd───sshd───bash───su───bash───pstree
     ├─udevd───2*[udevd]
     └─winbindd───3*[winbindd]
```

Rys. 5..1: Przykładowe drzewo procesów tworzone przez **`pstree`**

Zauważ, że **`pstree`** stara się skomasować obraz drzewa, jeśli w jego strukturze pojawiają się kolejno identyczne gałęzie, co sygnalizowane jest pojawieniem się znacznika **`n*`** (na przykład **`3*[smbd]`**).

Polecenie **`pstree`** uruchamia się w sposób następujący:

`pstree` ***`[przełączniki...] [pid | użytkownik]`***

gdzie:

– ***`pid`***
  PID procesu, który ma zostać przedstawiony jako korzeń drzewa (domyślnie jest to proces o PID równym 1)

– ***`użytkownik`***
  zaprezentowanie tylko tych procesów, których właścicielem jest wskazany użytkownik

Poniżej przedstawiono niektóre z przełączników polecenia **`pstree`**:

`-a` prezentuje linię poleceń każdego procesu

-c wyłącza domyślne skracanie obrazu drzewa
-l wyprowadza długie wiersze nawet wtedy, gdy zaburza obraz drzewa (przydatne przy wyprowadzanie wyników do pliku)
-n sortuj potomków w kolejności PID (domyślnie sortowanie przebiega według nazw)
-p włącz prezentowanie numerów PID

5.2.3. Polecenie top

Program **top** jest prostym interaktywnym zarządcą procesów, cyklicznie wyprowadzającym na terminal listę danych o obecnych w systemie procesach (domyślnie aktualizacja przeprowadzana jest co 5 sekund). Zachowanie narzędzia może być kontrolowane przy starcie poprzez specyfikowanie pożądanego zestawu przełączników oraz na bieżąco w czasie pracy poprzez polecenia wydawane przez naciskanie wybranych klawiszy. Przykładowy ekran narzędzia top prezentuje się jak poniżej (pokazano tylko kilka początkowych linii zestawienia):

```
Tasks:  87 total,   1 running,  86 sleeping,   0 stopped,   0 zombie
%Cpu(s):  1,0 us,  2,9 sy,  0,0 ni, 95,8 id,  0,0 wa,  0,0 hi,  0,3 si,  0,0 st
KiB Mem:    496600 total,   436192 used,    60408 free,   165372 buffers
KiB Swap:   499996 total,     2992 used,   497004 free,   214788 cached

  PID USER      PR  NI  VIRT  RES  SHR S  %CPU %MEM    TIME+  COMMAND
21108 user      20   0  4636 1324 1028 R   1,9  0,3   0:00.14 top
20889 root      20   0 11436 3828 3068 S   1,0  0,8   0:01.18 sshd
 1 root     20   0  2144  588  556 S   0,0  0,1   0:40.68 init
 2 root     20   0     0    0    0 S   0,0  0,0   0:00.09 kthreadd
 3 root     20   0     0    0    0 S   0,0  0,0   2:57.35 ksoftirqd/0
```

Zauważ, że **top** używa tych samych nagłówków kolumn, co **ps**. Wyjście z programu **top** następuje po naciśnięciu klawisza **q**.

Poniżej przedstawiono niektóre z przełączników narzędzia **top**:

-d *n*
odświeża zestawienie co *n* sekund

-p *pid*
prezentuje tylko procesy o podanych identyfikatorach PID (przełącznik może być użyty maksymalnie 20 razy)

-s
włącza tak zwany *tryb bezpieczny*, w którym wyłączone są polecenia potencjalnie ryzykowne

-c
prezentuje pełne linie poleceń zamiast nazw poleceń
-n *n* odśwież zestawienie *n* razy i zakończ pracę
-b
włącza tryb wsadowy (w trybie tym **top** nie reaguje na polecenie z klawiatury oraz wyprowadza zestawienia w postaci czystego tekstu bez znaków sterowania terminalem, co pozwala umieszczać prezentowane dane wprost w pliku)

Oprócz danych znanych z zestawienia prezentowanego przez **ps**, **top** uwidacznia również między innymi następujące informacje:

- **uptime**
  łączny czas pracy systemu od chwili jego ostatniego włączenia oraz trzy miary obciążenia systemu, określające średnią liczbę procesów gotowych do uruchomienia w ostatniej minucie, ostatnich 5 minutach i w ostatnich 15 minutach
- **processes**
  ogólna liczba procesów w systemie, rozbita na procesy, które:
  - wykonują się (ang. *running*)
  - śpią (ang *sleeping*)
  - są zatrzymane (stopped)
  - są w stanie *zombie* (są to tak zwane *procesy niemartwe*, tzn. procesy, które zakończyły działanie, lecz z różnych przyczyn nie zostały usunięte z systemu)
- **CPU states**
  zobrazowanie zajętości CPU w trybach:
  - użytkownika (us od ang. *user*)
  - systemowym (sy od ang *system*)
  - zadań z ujemną wartością *nice* (ni)
  - bezczynności (id od ang. *idle*)
  - czekania na zdarzenie (wa od ang. wait)
  - obsługi przerwań sprzętowych (hi od ang. *hardware interrupt*)
  - obsługi przerwań programowych (si od ang. software interrupt)
  - czasu skradzionego (st od ang. *stolen*) (parametr ma znaczenie, gdy system pracuje w maszynie wirtualnej, kiedy to może mieć miejsce wywłaszczanie procesu maszyny wirtualnej przez system operacyjny gospodarza – jest to tak zwana *kradzież czasu*)
- **Mem**
  dane o zajętości pamięci:
  - **total** – całkowita pamięć zainstalowana w systemie

  - **used** – pamięć zajęta
  - **free** – dostępna pamięc
  - **buffers** – pamięć używana przez bufory
- **Swap**
  dane opisujące użycie obszaru wymiany, zarządzanego przez systemowy mechanizm pamięci wirtualnej (zakres prezentowanych danych jest niemal identyczny, jak w dziale **Mem**)

W czasie pracy interaktywnej top akceptuje między innymi następujące polecenia wydawane z klawiatury:

- **<spacja>**
  odśwież ekran natychmiast
- **k**
  zabija proces; (wymaga podania numeru PID i numeru sygnału przesłanego do procesu – szczegóły znajdziesz w dalszej części opisu)
- **n**
  zmień liczbę prezentowanych procesów (wymaga podania żądanej liczby procesów)
- **r**
  zmień wartość *niceness* dla procesu (wymaga podania numeru PID i wartości niceness)
- **l**
  włącz/wyłącz informacje o obciążeniu
- **m**
  włącz/wyłącz informacje o pamięci.
- **t**
  włącz/wyłącz informacje o procesach
- **c**
  włącz/wyłącz wyświetlanie linii poleceń
- **A**
  sortuj procesy według czasu przebywania w systemie
- **M**
  sortuj procesy według zużycia pamięci.
- **N**
  sortuj procesy według numerów PID
- **P**
  sortuj procesy według obciążeń CPU
- **T**
  sortuj procesy według czasów użycia CPU
- **W**

zapisz konfigurację do pliku ~/**.toprc**; konfiguracja ta zostanie użyta jako domyślna po następnym uruchomieniu programu **top**

5.2.4. Polecenie `time`

Polecenie **time** pozwala oszacować czas wykonania wskazanego programu, co osiąga się poprzez pomiar trzech niezależnych czasów:

real bezwzględny czas przebywania procesu w systemie, liczony od uruchomienia do zakończenia, określany często angielskim terminem *wall-clock time*

user łączny czas procesora zużyty przez proces i spędzony w trybie użytkownika

sys łączny czas procesora zużyty przez proces i spędzony w trybie systemu

Uwaga: zmierzone czasy są wyprowadzane na stderr, co pozwala na ich łatwe oddzielenie od strumienia stdout monitorowanego procesu.

Polecenie time uruchamia się w sposób następujący:

```
time [przełączniki...] polecenie [argument_polecenia...]
```

Na przykład w poniższy sposób można oszacować obciążenie procesora i systemu operacyjnego przez wykonanie kompilacji wybranego kodu źródłowego.

```
$ time gcc prog.c
real    0m0.706s
user    0m0.540s
sys     0m0.150s
```

5.3. Usuwanie procesów

5.3.1. Sygnały

Właściciel procesu lub użytkownik root ma możliwość usunięcia procesu z systemu. Zadanie to wykonuje polecenie o sugestywnej nazwie **kill**, chociaż należy zaznaczyć, że rola tego narzędzia nie ogranicza się tylko do działań eksterminacyjnych. Jego podstawowym zadaniem jest wysyłanie do procesu/procesów tak zwanych sygnałów (ang. *signals*), uruchamiających asynchronicznie krótkie sekwencje kodu, przeznaczone do reagowania na przeróżne sytuacje mające znaczenie dla życia procesu.

Na potrzeby niniejszego opracowania zadowolimy się stwierdzeniem, że każdy z pracujących procesów może zostać poinformowany wyspecjalizowanym sygnałem

o zaistnieniu jednego z predefiniowanych zdarzeń. Odebranie takiej informacji powinno być skojarzone z jej obsłużeniem.

Trzy z sygnałów mają ścisły związek z wymuszonym zakończeniem działania procesu – są to:

- **SIGTERM** (sygnał nr 15)
  proces, do którego wysłano sygnał SIGTERM, powinien obsłużyć go kończąc pracę, jednak sygnał ten może zostać przez proces zignorowany, co z kolei może sprawić, że będzie bezskuteczny; obsłużenie tego sygnału daje procesowi szansę zakończenia działania w sposób kontrolowany i bezpieczny
- **SIGINT** (sygnał nr 2)
  sygnał, który wysyłany jest do procesu w reakcji na użycie kombinacji klawiszy Ctrl-C; może zostać zignorowany, a jego obsłużenie pozwala na kontrolowane zakończenie pracy procesu
- **SIGKILL** (sygnał nr 9)
  proces, do którego wysłano sygnał *SIGKILL*, bezzwłocznie kończy pracę; zignorowanie tego sygnału nie jest możliwe, ale nie jest też możliwe wykonanie czynności gwarantujących bezpieczne zakończenie

5.3.2. Polecenie `kill`

Polecenie to wydaje się w następujący sposób:

```
kill [ -nazwa_lub_numer_sygnału ] [ przełącznik... ] PID_procesu...
```

Użycie polecenia **kill** powoduje wysłanie sygnału o podanym numerze bądź o podanej nazwie do procesów o podanych numerach PID. Pominięcie specyfikacji sygnału spowoduje wysłanie sygnału *SIGTERM*.

Obie poniższe formy polecenia **kill** są równoważne:

```
kill -9 2222
kill -SIGKILL 2222
```

Pracujący proces można również przerwać używając kombinacji klawiszy Ctrl-C, jednak w pewnych wyjątkowych sytuacjach (na przykład wtedy, gdy działanie procesu jest zakłócone przez błędy w kodzie) *SIGTERM* i *SIGINT* mogą być bezskuteczne i wtedy nieodzowne staje się użycie sygnału *SIGKILL*.

5.3.3. Polecenie `killall`

Ponieważ posługiwanie się numerami PID bywa kłopotliwe i ze względu na łatwość popełnienia błędu, potencjalnie niebezpieczne, w pewnych sytuacjach korzystniejsze jest używanie nazw programów (**kill** przed wysłaniem sygnału nie pyta użytkownika o potwierdzenie swojej decyzji, wskutek czego omyłkowe skierowa-

nie sygnału do niewłaściwego procesu może spowodować prawdziwą katastrofę, szczególnie, gdy ma się uprawnienia użytkownika *root*).

Zatrzymanie wszystkich procesów pochodzących z programu o podanej nazwie powoduje polecenie **killall**, wydawane w sposób następujący:

killall *[przełącznik...] nazwa_programu...*

spowoduje wysłanie sygnału SIGKILL do wszystkich procesów powstałych wskutek uruchomienia polecenia **grep**.

Najczęściej używane przełączniki to:

-I
nazwy programów brane są bez uwzględniania różnic w wielkości liter

-y *time*
wysyła sygnał tylko do procesów młodszych niż *time*; time podawany jest jako liczba rzeczywista z przyrostkiem:
- **s** sekundy
- **m** minuty
- **h** godziny
- **d** dni
- **w** tygodnie
- **M** miesiące
- **y** lata

Dygresja: autor niniejszego tekstu powątpiewa w to, aby ostatni z podanych przyrostków został kiedykolwiek sensownie użyty, ale składa wyrazy uznania dla wyobraźni autorów polecenia **killall**

-o *time*
wysyła sygnał tylko do procesów starszych niż *time*

-i
pytanie o potwierdzenie przed wysłaniem sygnału do każdego z żądanych procesów

-r *name*
potraktowanie nazwy name jako wyrażenia regularnego

`-s sig`
wysłanie sygnału *sig* zamiast domyślnego *SIGTERM*

`-u user`
wysłanie sygnału tylko do procesów użytkownika *user*

5.4. Priorytety procesów

5.4.1. Niceness

Bieżąca wartość priorytetu procesu w systemach klasy Unix/Linux jest często określana angielskim terminem *niceness*, co na język polski można spróbować przetłumaczyć jako *uprzejmość*. Domyślnie nowo utworzony proces dziedziczy wartość *niceness* po swoim rodzicu i w większości sytuacji jest to zero. W systemie istnieją dwa standardowe narzędzia pozwalające manipulować tą wartością, jednak użytkownik inny niż *root* jest w tym działaniu bardzo ograniczony – pozwala mu się jedynie na **pogarszanie** priorytetu swoich procesów (a więc może on jedynie **powiększać** wartość *niceness*). Użytkownik *root* może zmieniać *niceness* w dowolny sposób.

Taka niecodzienna nomenklatura powstała jako efekt spostrzeżenia, że proces o gorszym priorytecie jest **uprzejmiejszy** (ang. *nicer*) dla systemu operacyjnego.

5.4.2. Polecenie `nice`

Polecenie **`nice`** służy do uruchomienia procesu z priorytetem innym niż domyślny i ma następującą postać:

```
nice -n niceness polecenie [ argument... ]
```

Użycie **`nice`** spowoduje uruchomienie polecenia *polecenie* z wartością *niceness* podaną za przełącznikiem **`-n`**. Dla zwykłych użytkowników dozwolone jest jedynie użycie wartości dodatnich, na przykład:

```
nice -n 10 kompilacja_cyber_punka
```

spowoduje uruchomienie (zapewne bardzo czasochłonnego) skryptu kompilującego złożony projekt z wartością *niceness* równą 10. Taki sposób wystartowania procesu gwarantuje, że nie będzie obciążał on nadmiernie zasobów systemu, choć zostanie to najprawdopodobniej okupione wydłużeniem czasu jego pracy.

5.4.3. Polecenie `renice`

Procesowi już uruchomionemu wartość *niceness* może zostać zmieniona poprzez wydanie polecenia **`renice`** i tak jak poprzednio, użytkownik inny niż *root* może to uczynić tylko wtedy, gdy godzi się pogorszyć priorytet swojego zadania (podwyższyć wartość niceness). Dokonuje się tego poleceniem o poniższej postaci (wymieniono najprostszą z form):

```
renice niceness pid...
```

przy czym **`niceness`** traktuje się tu nie jako przyrost, a jako nową wartość, czyli:

```
renice +10 3442
```

będzie próbować ustawić *niceness* procesu o PID równym 3442 na 10, co zwykłemu użytkownikowi uda się tylko wtedy, gdy dotychczasowa wartość *niceness* była mniejsza lub równa żądanej.

5.5. Manipulowanie procesami

Programy uruchamiane za pośrednictwem sesji terminalowej tworzą tak zwane *procesy pierwszoplanowe* (ang. *foreground processes*). Generalną zasadą jest, że powłoka czeka na zakończenie takiego procesu i będzie gotowa na przyjęcie kolejnych poleceń dopiero wtedy, gdy poprzednio uruchomiony proces zostanie usunięty z systemu.

Innym sposobem uruchomienia programu jest uczynienie z niego tak zwanego *procesu drugoplanowego*, nazywanego również *procesem pracującym w tle* (ang. *background process*). Proces drugoplanowy zaczyna pracować równolegle z powłoką, a powłoka nie czeka na jego zakończenie.

Oczywiście, praca procesu w tle ma sens tylko wtedy, gdy dla nie jest wymagana interakcja za strony użytkownika. W przeciwnym przypadku strumienie danych przyjmowane i wysyłane przez powłokę będą się mieszać z danymi emitowanymi i pobieranymi przez proces drugoplanowy. W wielu przypadkach problem tej natury może zostać rozwiązany przez przekierowania i/lub potoki zastosowane przy uruchomieniu programu pracującego w tle.

Program zostanie skierowany do uruchomienia w tle, jeśli za ostatnim parametrem jego polecenia będzie umieszczony znak **`&`** (ampersand):

```
polecenie &
```

Aktualnie pracujący proces pierwszoplanowy można wstrzymać (ale nie zatrzymać!) za pomocą kombinacji klawiszy *Ctrl–Z*. Proces wstrzymany rezyduje w systemie, ale nie jest aktywnie wykonywany (nie przydziela mu się procesora). Taki proces można ponownie skierować do wykonywania w tle poleceniem:

```
bg
```

Przywrócenie takiego procesu z powrotem na pierwszy plan wykona polecenie:

```
fg
```

Będzie to jednak możliwe tylko wtedy, gdy pomiędzy tymi poleceniami nie został uruchomiony w tle inny proces. Listę wszystkich kontrolowanych zadań wyprowadza na strumień *stdout* polecenie o nazwie

```
jobs
```

Jeśli wstrzymaniu poddano więcej niż jedno zadanie, jako argumentów poleceń **bg** i **fg** można używać unikalnych identyfikatorów zadań (ang. *job ids*), podawanych przez **jobs** (*uwaga! nie mają one nic wspólnego z numerami PID!*) i poprzedzonych znakiem % (procent).

Na przykład:

```
fg %1
```

wyprowadzi ze stanu wstrzymania zadanie o identyfikatorze 1.

5.6. Kod zakończenia procesu

Każdy kończący pracę proces przekazuje do systemu tak zwany *kod zakończenia* (ang. *completion code*). W programach napisanych w języku C jest to wartość zwrócona przez funkcję **main()** lub wartość przekazana jako argument do funkcji kończących pracę procesu (na przykład **exit()**). Kod zakończenia jest ośmiobitową liczbą całkowitą bez znaku i przyjęto, że wartość 0 oznacza bezbłędne wykonanie i zakończenie procesu. Wartości różne od 0 mogą oznaczać błędy lub wystąpienie sytuacji nietypowych (na przykład kompilator może w ten sposób sygnalizować, że proces translacji przebiegł z ostrzeżeniami i/lub błędami). Kod zakończenia ostatnio wykonywanego programu jest umieszczany w zmiennej środowiskowej o nazwie **$?** i może być ujawniony poleceniem:

```
echo $?
```

lub użyty w skrypcie powłoki w celu zdiagnozowania sytuacji powstałej po uruchomieniu ostatniego programu. Zauważ, że standardowym elementem wielu stron podręcznikowych jest opisanie kodów zakończenia zwracanych przez polecenia wraz ze wskazaniem sposobu ich interpretacji (sekcja o nazwie *EXIT STATUS*).

Status zakończenia pewnego procesu można również wykorzystać do warunkowego uruchomiania kolejnych poleceń. Na przykład zapis taki jak poniżej:

```
cmd1 && cmd2
```

oznacza, że **cmd2** należy wykonać tylko wtedy, gdy **cmd1** zakończyło się **sukcesem** (zwróciło zerowy kod powrotu).

Zapis o postaci:

```
cmd1 || cmd2
```

oznacza, że **cmd2** ma być wykonane tylko wtedy, gdy **cmd1** zakończyło się **niepowodzeniem** (zwróciło niezerowy kod powrotu). Semantyka ta może być wytłumaczona następującym spostrzeżeniem:

(true && x) = x
(false || x) = x

co jednoznacznie określa, kiedy obliczenie prawego argumentu operacji logicznej jest niezbędne.

Jedno polecenie powłoki może spowodować kolejne uruchomienie kilku procesów, o ile nazwy programów zostaną rozdzielone średnikami, na przykład tak:

```
cmd1; cmd2; cmd3
```

Sekwencja taka zostanie wprowadzona w tło, jeśli lista programów zostanie ujęta w nawiasy i zakończona znakiem **&**:

```
(cmd1; cmd2; cmd3) &
```

5.7. Źródła uzupełniające

1. Manual polecenia **kill**: https://man7.org/linux/man-pages/man1/kill.1.html
2. Manual polecenia **killall**: https://man7.org/linux/man-pages/man1/killall.1.html
3. Manual polecenia **nice**: https://man7.org/linux/man-pages/man1/nice.1.html
4. Manual polecenia **ps**: https://man7.org/linux/man-pages/man1/ps.1.html
5. Manual polecenia **pstree**: https://man7.org/linux/man-pages/man1/pstree.1.html
6. Manual polecenia **renice**: https://man7.org/linux/man-pages/man1/renice.1.html
7. Manual polecenia **time**: https://man7.org/linux/man-pages/man1/time.1.html
8. Manual polecenia **top**: https://man7.org/linux/man-pages/man1/top.1.html
9. Opis zarządzania zadaniami w powłoce Bash: https://www.gnu.org/software/bash/manual/bash.html#Job-Control-Basics
10. Opis mechanizmu sygnałów w powłoce Bash: https://www.gnu.org/software/bash/manual/bash.html#Signals

5.8. Zadania do samodzielnego wykonania

1. Ile procesów prezentuje **ps** uruchomiony z przełącznikami **-er**? Dlaczego tylko tyle?
2. Co powoduje uruchomienie polecenia w stylu BSD o postaci:
   ps aux
3. Do czego służy polecenie **uptime**? Jaki jest **uptime** komputera, na którym obecnie pracujesz?
4. Do czego służy polecenie **free**?
5. Do czego służy interaktywne polecenie **O** programu **top**?
6. Jakie jest domyślne sortowanie procesów w programie **top**?
7. Jaki efekt powoduje przełącznik **-l** polecenia **kill**?
8. Spróbuj zatrzymać ten ze swoich procesów, który ma najniższą wartość PID. Co się stało?
9. Co oznacza kod powrotu 1 zwrócony przez polecenie **grep**?
10. Czy polecenie **kill** może popełnić samobójstwo (to znaczy, wysłać sygnał SIGKILL do siebie samego)?

11. Czy polecenie killall może popełnić samobójstwo?
12. Ile pamięci RAM ma komputer, na którym obecnie pracujesz?
13. Skonstruuj prosty potok zliczający procesy wskazanego użytkownika.
14. Znajdź ścieżkę pokrewieństwa wiodącą od procesu *init/systemd* do procesu przeglądarki, w której oglądasz niniejszy dokument (wypisz PIDy wszystkich procesów na tej ścieżce).
15. Przeprowadź eksperyment sprawdzający, czy zmiana priorytetu na najgorszy może wpłynąć na czas wykonania złożonego polecenia. Użyj na przykład czasochłonnych wariantów poleceń **`find`** lub **`grep`** albo skomplikowanych potoków. Sprawdź czas wykonania z domyślnym i najgorszym możliwym *niceness*.
16. Gdzie w Polsce znajduje się miejscowość o nazwie Renice?
17. Rzetelnie przećwicz użycie poleceń **`jobs`**, **`fg`** i **`bg`**. W tym celu uruchom kilka procesów pracujących interaktywnie (na przykład **`top`**, **`watch`** albo edytor tekstu) i naucz się biegle przemieszczać między nimi oraz przełączać je pomiędzy pracą w tle i na pierwszym planie. Obserwuj wyniki podawane przez **`top`** w czasie twoich działań.
18. Zapoznaj się z listą sygnałów na stronie podręcznikowej **`signal(7)`**. Uruchom sesję edytora **`vi`** i wysyłaj do niego poleceniem **`kill`** kilka różnych sygnałów (użyj co najmniej **`SIGHUP`**, **`SIGINT`**, **`SIGTERM`**, **`SIGQUIT`**, **`SIGKILL`**, **`SIGSTOP`**, **`SIGCONT`**). Zanotuj zachowanie edytora w reakcji na przesłane sygnały.
19. Do czego służy polecenie **`sleep`**? Jak **`top`** uwidacznia proces wykonujący **`sleep`**?