mgr Maciej Wróbel

Podstawy obsługi Linux: obsługa procesorów i pamięci, obsługa procesów, komunikacja międzyprocesowa. Zarządzanie procesami w systemie Linux.

4 październik 2010

1. Wprowadzenie – Procesy w systemie Linux

1.1. Obsługa procesora i pamięci w Systemie Operacyjnym

Wiele typowych aplikacji przez większość czasu swego działania oczekują na powolne operacje wejścia/wyjścia lub reakcję użytkownika. Powoduje to, że wykorzystanie przez nie procesora jest niewielkie. Ponieważ jednostka obliczeniowa była w historycznych czasach bardzo droga, podjęto próby wykorzystania wolnego czasu procesora. Wprowadzano nadzorujący program, który pozwalał w wolnym czasie procesora uruchamiać inne programy. Rozwój takich nadzorujących programów doprowadził do obecnie funkcjonujących wielozadaniowych systemów operacyjnych.

Działanie programu nadzorującego użycie procesora polega na uruchamianiu kolejnego programu, gdy obecny nie wykorzystuje aktywnie procesora. Implementacja jednak nie jest oczywista. Gdyby wszystkie programy miały w systemie jednakowy status, to awaria jednego z nich powodowałaby awarię całego systemu. Adresowanie pamięci każdego z programów musiałoby uwzględniać istnienie innego działającego oprogramowania. Ponadto dostęp do pamięci musi być tak regulowany, aby programy wzajemnie nie mogły wpływać na swoje działanie.

Rozwiązaniem powyższych problemów przyjętym w systemach Linux jest zastosowanie tzw. maszyny wirtualnej. Aplikacje uruchamiane w systemie nie mają bezpośredniego dostępu do sprzętu, a komunikują się z nim poprzez jądro systemu operacyjnego. Działające aplikacje (nazywane dalej procesami lub zadaniami) wykonywane są w tzw. trybie użytkownika, a jądro w trybie rzeczywistym. Implementacja takiego podziału na poziomie procesora gwarantuje (w zakresie poprawności działania systemu operacyjnego), że aplikacje przestrzeni użytkownika nie mogą na siebie oddziaływać. Adresacja pamięci i przydział zasobów systemowych należy do zadań jądra, a programy można przygotować tak, jakby

były jedynymi działającymi w systemie. O dostępie do czasu procesora decyduje zarządca procesów (lub też planista, organizator, ang. *process sheduler*).

1.2. Pojęcie procesu

Proces jest to wykonujący się egzemplarz programu. Posiada własny segment danych oraz przydzielany przez system czas procesora i zasoby. W systemie Linux posiada własny, unikalny identyfikator (ang. *process ID = PID*), będący dodatnią liczbą całkowitą. Dodatkowo wyróżnia się **wątki**, które są „ubogimi krewnymi" procesów – nie posiadają odrębnego segmentu danych, co z jednej strony ogranicza ich możliwości, a z drugiej skraca czas ich tworzenia oraz umożliwia dostęp do wspólnych obszarów pamięci[1]. W systemach Linux zwykle z procesem związany jest także jego terminal sterujący.

1.3. Procesy w systemie

Procesy w systemie Linux funkcjonują w strukturze hierarchicznej. Każdy proces (oprócz procesu **init**) posiada swój proces nadrzędny, który go wywołał. Procesy tworzone są przy pomocy funkcji jądra systemu **fork**, a ich kod programu ładowany jest przy pomocy jednej z 6-ciu funkcji **exec**[2].

W systemie Linux proces może znajdować się w 4 stanach (wg wykładu w 3):

1. gotowy do działania (lub działający),
2. oczekujący na zdarzenie zewnętrzne,
3. zatrzymany,
4. proces zombi.

Zarządca procesów kolejkuje tylko procesy 1. rodzaju. Procesy 2. rodzaju, po zajściu zdarzenia przechodzą w tryb gotowości do działania. Procesy zatrzymane są pełnoprawnymi procesami znajdującymi się w systemie, jednak nie jest przydzielany im czas procesora. Wreszcie procesy zombie to takie, które zakończyły swoje działania, nie zostały jednak poprawnie zakończone przez swój proces nadrzędny.

2. Zarządzanie procesami

System Linux udostępnia wiele narzędzi, pozwalających monitorować procesy i zarządzać nimi. Wymienić należy graficzne **gnome-system-monitor**, **ksysguard** oraz **gtop**. W trybie tekstowym możemy przeglądać procesy za pomocą poleceń **ps** oraz **pstree**, a zarządzać nimi przy pomocy interaktywnego polecenia **top**.

Zadania

1. Zapoznaj się z dokumentacją poleceń *ps, pstree, top.*
2. Uruchom program *gnome-system-monitor* i zapoznaj się z jego działaniem.

[1] Więcej o wątkach na poświęconych im zajęciach
[2] Więcej o funkcjach fork i exec na osobnych zajęciach

3. Wypróbuj działanie poleceń *ps aux*, *ps -Aj*, *ps -f*, *pstree*.
4. Wypróbuj działanie polecenia **top**.

2.1. Uruchamianie procesów

Gdy uruchamiamy program, zwykle tworzymy proces. W przypadku uruchamiania w terminalu standardowo proces stanie się procesem działającym w pierwszym planie (ang. *foreground*). Proces taki blokuje terminal do czasu zakończenia swojego działania. Proces może zostać uruchomiony w tle (ang. *background*) poprzez wywołanie go z operatorem &:

```
nazwa_programu parametry &
```

pod warunkiem, że nic nie będzie wyświetlał w terminalu, gdyż wtedy może zostać zatrzymany przez system operacyjny jako proces z niedziałającym terminalem (problem ten można rozwiązać np. przekierowując wyście programu do pliku). Program taki będzie kontrolowany przez bieżący terminal (więc zakończenie pracy terminala spowoduje zakończenie działania procesu), pozostawiając terminal interaktywnym. Listę zadań działających w tle można sprawdzić poleceniem **jobs**. Poleceniem **fg** można przenieść na pierwszy plan polecenie z tła, zaś **bg** przenieść zadanie w tło. Zadania wyświetlone poleceniem *jobs* są ponumerowane i w poleceniach *bg* i *fg* możemy odnościć się jako %1, %2 itd. Brak argumentu lub argument %% oznacza ostatnie zadanie na liście (czyli najpóźniej uruchomione).

Częstym problemem w przypadku długo działających procesów jest to, że procesy działają tylko tak długo, jak ich sterujący terminal (po zakończeniu pracy terminala wysyłany jest do nich sygnał SIGHUP, który domyślnie kończy ich działanie). Aby tego uniknąć można skorzystać z polecenia **nohup** lub narzędzia **screen**, a w przypadku własnej aplikacji zapewnić właściwą obsługę sygnału SIGHUP.

Zadania

1. Uruchom z terminala długo działający proces. Sygnałem Ctrl+Z wstrzymaj jego działanie.
2. Uruchom polecenie **jobs**.
3. Wznów wykonywanie procesu poleceniem **bg**.
4. Wznów wykonywanie procesu w pierwszym planie.
5. Zapoznaj się z dokumentacją poleceń screen i nohup.

2.2. Usuwanie procesów

System operacyjny Linux umożliwia wyłączenie dowolnego procesu. Dokonuje się to przez zwykle wysłanie sygnałów SIGINT (w przypadku przerwania z klawiatury) lub SIGTERM. Domyślnym działaniem podejmowanym przez aplikację jest zakończenie pracy, jednak programista może je zmienić (na przykład po to, aby umożliwić użytkownikowi zapisanie danych). Sygnał SIGKILL jest sygnałem którego proces nie może zignorować i zawsze kończy się usunięciem procesu z systemu. Sygnały wysyłamy poleceniem **kill**:

```
kill -sygnal PID
```

gdzie *sygnal* jest numerem sygnału (SIGINT=2, SIGTERM=15, SIGKILL=9), a PID jest identyfikatorem procesu. Podobne do *kill* działanie ma polecenie **killall**, które pozwala wysłać sygnał do poleceń spełniających zadane kryteria oraz **pkill**, które wysyła sygnały do procesów o podanej nazwie.

Zadania

1. Wypróbuj działanie poleceń *kill* i *killall* na wybranych przez siebie procesach.
2. Wypróbuj działanie *kill -9* na wybranych procesach.
3. Zapoznaj się z dokumentacją poleceń *kill*, *pkill* i *killall*.

2.3. Priorytety procesów

Użytkownik w systemie Linux nie ma bezpośredniego wpływu na priorytety działania programów. Może jednak określić tzw. liczbę **nice**, która wskazuje systemowi operacyjnemu jak ważne jest dane zadanie. Liczba ta przyjmuje wartości z przedziału $[-20, 19]$. Im mniejsza wartość liczby *nice* tym wyższy priorytet będzie miało zadanie wykonywane. Liczbę *nice* ustala się przy pomocy polecenia **renice** dla działającego procesu. Uruchomić polecenie z zadaną liczbą *nice* można przy pomocy polecenia **nice**.

```
renice zmiana_nice PIDprocesu
nice wartosc_nice nazwalpolecenia
```

na przykład polecenie

```
renice +5 1
```

będzie usiłowało zwiększyć wartość *nice* dla procesu *init*, a

```
nice -5 find -name '*'
```

będzie próbowało uruchomić polecenie find z nice=-5.

Zadania

1. Zapoznaj się z dokumentacją poleceń **nice** i **renice**.
2. Spróbuj zmienić priorytet procesu init oraz innych wybranych przez siebie procesów.

2.4. Status zakończenia procesu

Każdy program kończąc swoje działania zwraca programowi kontrolującemu niewielką wartość całkowitą, będącą statusem jego zakończenia (zwane też kodem wyjścia) (ang. *exit code*). Przyjęła się konwencja, że poprawnie wykonany program zwraca 0, a w razie błędu inną wartość. Nie należy na tej konwencji ślepo polegać, ponieważ programista pisząc program może dowolnie zmienić wartość kodu wyjścia.

3. Komunikacja międzyprocesowa i interakcja z procesami

3.1. Kody wyjścia

Najprostszym sposobem na przekazanie przez proces o swoim działaniu informacji do innych procesów jest ustawienie określonego kodu wyjścia. Jego wartość w języku C określa funkcja return w części głównej programu. Oczywiście ilość informacji w ten sposób przekazanej jest bardzo ograniczona (jedna liczba całkowita) i przekazana zostaje dopiero po zakończeniu działania programu.

Standardowa powłoka **sh** przypisuje zmiennej $? wartość kodu wyjścia ostatnio wykonanego programu. Sprawdzając jej wartość możemy więc dowiedzieć się jaki był status ostatnio zakończonego programu. Powłoka ta (i jej pochodne) implementuje także dwa operatory: && oraz ||. Pozwalają one wykonać program w zależności od kodu wyjścia jego poprzednika. Używa się ich następująco:

```
(wyrazenie1) && (wyrazenie2)
(wyrazenie1) || (wyrazenie2)
```

gdzie *(wyrazenie?)* to pojedyncza nazwa programu lub łańcuch instrukcji połączonych operatorami && lub ||[3]. Operator && powoduje wykonanie instrukcji (wyrazenie2) wtedy, gdy (wyrazenie1) miało kod wyjścia 0. Operator || pozwoli wykonać (wyrazenie2) jeżeli kod wyścia (wyrazenie1) był różny od zera.

Zadania

1. Wykonaj wybrane polecenie i sprawdź jego kod wyjścia.
2. Wykonaj polecenia `rm /proc/cpuinfo &&echo 'usunalem'`
   oraz `rm /proc/cpuinfo ||echo 'nie usunalem'`. Wyjaśnij otrzymane rezultaty.
3. Wykonaj polecenie `echo 1&&echo 2&&echo 3||echo 4 &&echo 5`. Wyjaśnij otrzymany rezultat.

3.2. Sygnały

Wykorzystanie kodów wyjścia pozwala powiązać zakończenie jednego procesu z uruchomieniem drugiego. Nie można natomiast wpłynąć na działanie już istniejącego procesu. Najprostszym mechanizmem, który udostępnia system operacyjny są wspomniane wcześniej (przy omawianiu poleceń kill) sygnały.

Najczęstszym zastosowaniem sygnałów jest przerywanie pracy programu (stąd nazwa programu wysyłającego sygnały – *kill*). Ponieważ jednak pisząc program możemy zaimplementować w nim dowolną obsługę sygnałów, więc przekazywanie mu sygnału może być sposobem na komunikację z tym procesem. Podejście to jest ograniczone tym, że sygnały są wyłącznie liczbami całkowitymi, a co ważniejsze, obsługiwane są w sposób asynchroniczny. Powoduje to, że zaburzona zostaje kolejność wykonywania programu, a funkcja obsługująca

[3] W dalszej części poznamy jeszcze operator |

sygnał musi spełniać określone kryteria. Zaletą jest to, że mamy pewność dostarczenia sygnału do aplikacji.

Zadania

1. zapoznaj się ze stronami 2 i 7 podręcznika signal (polecenia **man 2 signal** oraz **man 7 signal**).

3.3. Standardowe wejście/wyjście

Przy uruchamianiu każdego procesu w Linux posiada on 3 ważne otwarte uchwyty plików: standardowe wejście **stdin** (odpowiada mu nr 0), standardowe wyjście **stdout** (nr 1) i standardowe wyjście błędów **stderr** (nr 2). Uchwyty te domyślnie powiązane są z terminalem tak, że wprowadzane z klawiatury znaki trafiają do procesu przez stdin, program wyświetla komunikaty przez stdout a błędy przez stderr. Domyślne przypisanie tych uchwytów można w powłokach wywodzących się z *sh* zmienić przy pomocy operatorów $<$, $>$, $n>$ oraz |. Ich działanie jest następujące:

nazwaprogramu $<$ **nazwapliku** zamiast czytać z klawiatury, program będzie czytał z pliku.

nazwaprogramu $>$ **nazwapliku** zamiast wyświetlać na terminalu, program będzie zapisywał do pliku.

nazwaprogramu $2>$ **nazwapliku** program nie wyświetli błędów na terminalu, a zapisze je pliku.

(wyrazenie1) | **(wyrazenie2)** standardowe wyjście ostatniego programu wyrażenia 1 przekazane zostanie jako standardowe wejście pierwszego programu wyrażenia 2.

Powyższe operatory (szczególnie |) powodują, że powłoki sh mają bardzo duże możliwości. Standardowo każda dystrybucja Linux posiada wiele tzw. filtrów, które przekształcają w zadany sposób dane „wchodzące" standardowym wejściem i przekazują je standardowym wyjściem. Łącząc je przy pomocy powyższych operatorów można osiągnąć zadziwiające rezultaty. Przykładowo:

```
tr -c a-zA-Z '\n' < Readme1.txt  | sed '/^$/d' | sort | uniq -i -c
```

Powoduje wyświetlenie listy wszystkich słów znajdujących się w pliku Readme1.txt wraz z ilością ich wystąpień. Natomiast:

```
cat ~/.bash_history | tr "\|\;" "\n" | sed -e "s/^ //g" \
| cut -d " " -f 1 | sort | uniq -c | sort -n | tail -n 15
```

wyświetli 15 najczęściej używanych poleceń. Więcej o filtrach na zajęciach poświęconych programowaniu w Bash.

Zadania

1. Porównaj działanie poleceń `ls /etc` oraz `ls /etc |grep '^a'`. Zapoznaj się z podręcznikiem polecenia grep.

2. Zapoznaj się z podręcznikami poleceń **sort, tail, uniq, tr, join, cut**.
3. Zapoznaj się z podręcznikiem info coreutils (polecenie *info coreutils*).
4. Wyszukaj w zasobach Internetu ciekawe przykłady jednolinijkowych programów w Bash (ang. *Bash oneliners*).

---

4. Sprawozdanie

Sprawozdanie powinno zawierać:

1. Dyskusję, jakie zagrożenia wynikają z możliwości uzyskania przez proces dostępu do pamięci innego procesu.
2. Opis działania zarządcy procesów (ang. *process sheduler*).
3. Opis wybranych 5-ciu poleceń dotyczących zarządzania procesami (z części 2 instrukcji).
4. Przykład użycia operatorów || i &&.
5. Po trzy przykłady wykorzystania operatorów |, > i <.
6. Dwa ciekawe przykłady jednolinijkowych programów w Bash wraz z opisem działania.
7. Opis wybranych 5-ciu poleceń z podręcznika coreutils (innych niż w pkt. 3).

W wersji elektronicznej instrukcji podane niżej linki są hiperłączami.

Literatura

[1] Materiały MiMUW (j. polski) na `http://wazniak.mimuw.edu.pl/index.php?title=Systemy_operacyjne` dotyczące użytkowania systemu uniksopodobnego.

[2] Każdy podstawowy podręcznik systemu Linux.

[3] *Linux: administracja*. M. Carling, Stephen Degler, James Dennis. Wrocław : Wydaw. Robomatic, 2000.

[4] Materiały (w j. angielskim) dotyczące certyfikatu LPI udostępniane przez IBM na `http://www.ibm.com/developerworks/linux/library/l-lpic1-v3-map/`

[5] Materiały (także w j. polskim) dotyczące obsługi Linux udostępniane przez społeczność Gentoo na `http://www.gentoo.org/doc/pl/articles/`.

[6] Materiały na Oopweb dotyczące administracji Linux (w j. angielskim): `http://oopweb.com/OS/Documents/SAG/VolumeFrames.html`.