Funkcje systemu Unix

Witold Paluszyński
witold@ict.pwr.wroc.pl
http://sequoia.ict.pwr.wroc.pl/~witold/

Niniejszy dokument zawiera materiały do wykładu na temat wykorzystania funkcji systemu Unix w Pascalu. Jest on udostępniony pod warunkiem wykorzystania wyłącznie do własnych, prywatnych potrzeb i może być kopiowany wyłącznie w całości, razem z niniejszą stroną tytułową.

Przekazywanie argumentów do programu

Program wykonujący się w systemie Unix nazywa się *procesem* i posiada pewne *środowisko*, na które składają się **argumenty wywołania programu** i **zmienne środowiskowe**. Tworzenie podprocesów (jednych procesów przez inne) jest podstawowym mechanizmem pracy i programowania Unixa.

Zestaw (wektor) argumentów wywołania procesu stanowi kompletna linijka polecenia wpisanego w interpreterze komend, włącznie z nazwą lub ścieżką do programu, która jest argumentem numer 0. Podział linijki polecenia na argumenty dokonuje interpreter komend. Gdy proces uruchamiany jest programowo, bez udziału interpretera komend, to wektor argumentów wywołania określa wywołanie tworzące proces. Argumenty wywołania traktowane są jako napisy znakowe, choć mogą zawierać np. liczby.

Zmienne środowiskowe są symbolami z przypisanymi im wartościami — napisami znakowymi. Tworzone są dla danego procesu poleceniami interpretera komend (w programie: funkcją setenv). Przy tworzeniu podprocesu cały zestaw zmiennych środowiskowych jest *dziedziczony*, czyli kopiowany. Pozwala to przekazywać informacje do programu na przykład z interpretera komend, który go uruchomił, ale nie w drugą stronę.

Sun Pascal: argumenty wywołania i zmienne środowiskowe

```
PROGRAM sunargs(OUTPUT);

VAR x: INTEGER;
    a: String;

BEGIN
   WRITELN(argc:1, ' argumentow wywolania (argc):');
   FOR x := 0 TO argc-1 DO
   BEGIN
      argv(x,a);
      WRITELN('   argv(',x:1,') = "',a,'"');
   END;
   getenv('PATH',a);
   WRITELN('PATH = ',a);
END.
```


HP Pascal: argumenty wywołania i zmienne środowiskowe

```
PROGRAM hpargs(OUTPUT);

IMPORT arg;
TYPE String80 = String[80];
VAR x: INTEGER;
    a: String80;

FUNCTION getenv(v: String80): String80; EXTERNAL C;

BEGIN
   WRITELN(argc:1, ' argumentow wywolania (argc):');
   FOR x := 0 TO argc-1 DO
   BEGIN
      a := argn(x);
      WRITELN('   argn(',x:1,') = "', a ,'"');
   END;
   a := getenv('PATH');
   WRITELN('PATH = ',a);
END.
```


GNU Pascal: argumenty wywołania i zmienne środowiskowe

```
PROGRAM gpcargs(OUTPUT);

{$X+}
VAR x: INTEGER;
    a: CString;

FUNCTION getenv(v: CString): CString; asmname 'getenv';

BEGIN
   WRITELN(paramcount:1, ' argumentow wywolania (paramcount):');
   FOR x := 0 TO paramcount DO
   BEGIN
      a := paramstr(x);
      WRITELN('   paramstr(',x:1,') = "', a ,'"');
   END;
   a := getenv('PATH');
   WRITELN('PATH = ',a);
END.
```

Uruchamianie podprocesów procedurą SYSTEM

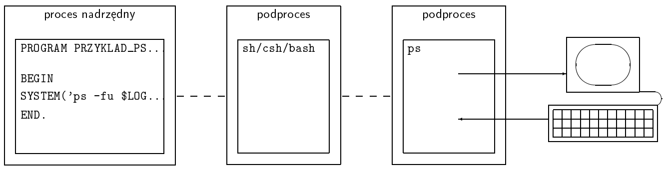
Procedura SYSTEM o następującym nagłówku powoduje utworzenie podprocesu, i wykonanie w nim polecenia systemu operacyjnego podanego jako argument 'command'. Polecenie to może mieć dowolną postać polecenia interpretera komend systemu Unix, a więc może zawierać odwołania do zmiennych interpretera poleceń, mechanizm skierowania strumieni wejścia/wyjścia, znak '&' powodujący uruchomienie polecenia w tle, spójniki logiczne interpretera poleceń, potoki poleceń, itd.

```
PROCEDURE SYSTEM(command : String);   EXTERN;
```

Procedura system tworzy podproces interpretera poleceń, który wykonuje dane polecenie. Jeśli poleceniem jest wywołanie programu, to tworzony jest kolejny proces, będący z kolei podprocesem interpretera poleceń, wykonujący dany program. Program ten ma dostęp do terminala użytkownika, może wyświetlać teksty na ekranie i czytać dane z klawiatury. W tym czasie proces pierwotny, zawierający wywołanie procedury SYSTEM, jest zawieszony, i czeka na zakończenie procesu interpretera poleceń.

PRZYKŁAD — następujące wywołanie procedury SYSTEM, daje taki sam efekt, jak wywołanie polecenia "ps -fu $LOGNAME" na terminalu użytkownika (zmienna środowiskowa $LOGNAME zawiera nazwę login użytkownika systemu):

```
PROGRAM PRZYKLAD_PS1(INPUT,OUTPUT);
BEGIN
  SYSTEM('ps -fu $LOGNAME');
END.
```



W tym przykładzie wszystkie procesy mają dostęp do strumieni INPUT i OUTPUT, ale w danym momencie wykonywany jest tylko jeden proces, a pozostałe są zamrożone.

Uruchamianie podprocesów funkcją POPEN

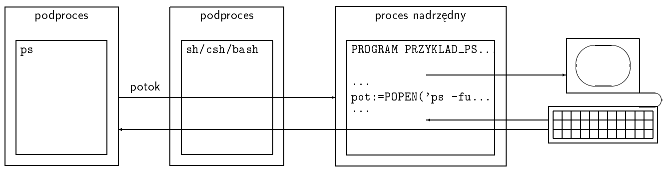
Funkcja POPEN umożliwia utworzenie poprocesu wykonującego zadane polecenie za pośrednictwem interpretera poleceń, podobnie jak procedura SYSTEM. Jednak funkcja POPEN kończy pracę od razu po utworzeniu podprocesu, który zostaje uruchomiony równolegle z procesem pierwotnym, i oba te procesy mogą komunikować się przez potok (jednokierunkowy strumień danych).

```
FUNCTION POPEN(command, mode : String): CHANNEL;   EXTERN;
```

Drugi argument mode ma wartość 'r' lub 'w' i określa, czy potok ma być skierowany do procesu macierzystego, czy od niego. Potok zostaje przez funkcję popen przypisany do standardowego wyjścia utworzonego procesu (mode='r') lub do jego standardowego wejścia (mode='w'). Wartość funkcji POPEN jest specjalnego typu CHANNEL. Zmienna tego typu określa otwarte łącze komunikacyjne, przez który można przesyłać dane specjalnie zdefiniowanymi procedurami.

PRZYKŁAD — dane wyświetlane przez program ps na jego standardowym wyjściu są czytane przez program z potoku i wyświetlane na jego wyjściu. Wszystkie procesy mają równoczesny dostęp do klawiatury, jednak nie mogą równocześnie z niej czytać, bo wynik byłby trudny do przewidzenia.

```
PROGRAM PRZYKLAD_PS2(INPUT,OUTPUT);
VAR pot: CHANNEL; buf: String;
BEGIN
  pot := POPEN('ps -fu $LOGNAME', 'r');
  WHILE NOT PEOF(pot)
  BEGIN
    PREADS(pot, buf);
    WRITELN(buf);
  END;
END.
```



Wywołanie POPEN('...','r'); ma sens jeśli program wywołany przez tę funkcję pisze coś na swoim wyjściu, co nasz pierwotny program chciałby czytać, a raczej nic nie czyta z klawiatury, chyba że program pierwotny sam nic nie czyta z klawiatury, i dzięki temu w taki albo inny sposób nie ma możliwości wystąpienia kolizji przy czytaniu z klawiatury.

Drugie możliwe wywołanie funkcji POPEN('...','w'); ma sens w sytuacji dualnej, a więc wtedy, gdy tworzony proces czyta dane ze swojego wejścia. Te dane będzie mu dostarczał program macierzysty przez potok. W tej sytuacji wszystkie procesy będą miały równoczesny dostęp do ekranu użytkownika, i jeśli chciałyby jednocześnie pisać na swoim wyjściu to dane te pojawią się na ekranie wymieszane w trudny do przewidzenia sposób.

Jeżeli jeden z komunikujących się procesów działa szybciej, i wysyła dane do potoku zanim drugi proces będzie gotów do ich odczytywania, albo jeśli spróbuje czytać dane z potoku zanim drugi proces cokolwiek do potoku zapisze, to operacja ta wykona się mimo wszystko poprawnie. Szybszy proces zostanie wstrzymany przez system (przestawiony w tzw. stan uśpienia), i wznowiony gdy potok będzie gotów do realizacji żądanej operacji wejścia/wyjścia. Taka synchronizacja pracy przez operacje I/O odbywa się automatycznie i w sposób niewidzialny dla procesów.

PRZYKŁAD — proces pierwotny uruchamia w podprocesie program mail do wysłania poczty, po czym czyta treść listu z klawiatury i przekazuje ją do podprocesu. W chwili zamknięcia potoku nastąpi wysłanie poczty.

```
PROGRAM PRZYKLAD_MAIL(INPUT,OUTPUT);
VAR pot: CHANNEL; buf: String;
BEGIN
  pot := POPEN('mail adresat@o2.pl', 'w');
  WHILE NOT EOF(INPUT)
  BEGIN
    READLN(buf);
    PWRITESLN(pot, buf);
  END;
  PCLOSE(pot);
END.
```