Systemy operacyjne

Laboratorium 4

Potoki

Jarosław Rudy
Politechnika Wrocławska

30 marca 2017

Laboratorium obejmuje znajomość potoków linuksowych, zwłaszcza w połączeniu z komendami `while` i `read`.

1 Potoki

Do tej pory wykonanie komend odbywało się wyłącznie sekwencyjnie:

```
komenda1
komenda2
```

lub w pojedynczej linii:

```
komenda1; komenda2
```

Oznacza to, że `komenda1` musi się zakończyć zanim `komenda2` się rozpocznie. Ponadto mogliśmy uzależnić uruchomienie `komenda2` od wyniku działania `komenda1`:

```
command1 && command2
command1 || command2
```

Jednak i w tym przypadku `komenda2` nie zacznie się wykonywać, jeśli `komenda1` jeszcze się nie zakończyła. Istnieje jednak możliwość, aby więcej niż jedna komenda pracowała jednocześnie podczas wykonywania skryptu. Jedną z takich możliwości jest użycie potoków, jako że technika ta daje istotne możliwości. Załóżmy, że wywołaliśmy komendę `find`, która drukuje nazwy plików. Teraz chcemy jednak wyświetlić jedynie 10 pierwszych wyników. Można to uzyskać za pomocą komendy `head`, o ile podamy na jej wejście wynik (wyjście) `find`a. Bez użycia potoków musielibyśmy przechować wynik `find`a w pliku tymczasowym i wysłać ten plik na wejście `head` z użyciem przekierowań:

```
find > somefile
head < somefile
```

Znak `<` (`>`) oznacza *przekierowanie* wejścia (wyjścia) komendy. Po ich użyciu na wejście komendy zostanie skierowana zawartość pliku (wyjście zostanie skierowane do danego pliku). Składnia `>` spowoduje wyzerowanie (nadpisanie) pliku. Składnia `>>` dopisze nowe dane na końcu pliku bez jego wyzerowania. Podejście to zadziała, ale ma wady. Komenda `head` musi czekać na zakończenie komendy `find`. Ponadto potrzebne jest utworzenie rzeczywistego pliku, co zajmuje czas (dyski twarde są stosunkowo powolne). Nie mamy też 100-procentowej gwarancji, że plik tymczasowy nie zostanie po drodze usunięty.

Drugim "naiwnym" podejściem jest zapisanie wyniku działania `find`a do zmiennej `bash`owej:

```
var=$(find)        # lub var=`find`
head <<< $var
```

Składnia `<<<` jest podobna do `<`, ale nie wyśle na standardowe wejście `head` zawartości pliku podanego w zmiennej `$var`, tylko wyśle wprost zawartość `$var` na wejście komendy jako tekst. To podejście nie jest jednak dużo lepsze od poprzedniego. Po pierwsze, wszystkie znaki nowej linii "znikną" po zapisaniu tekstu do zmiennej `var`. Po drugie, `find` wciąż musi się zakończyć zanim `head` zacznie pracę. Po trzecie, chociaż to podejście jest szybsze, to i tak *cały* wynik `find`a musi zostać zapisany w zmiennej. Jeśli ten wynik ma wielkość 100 megabajtów, to musimy przechować 100 megabajtów w jednej zmiennej. Jest to możliwe (chociaż skrajnie nieefektywne). A co jeśli wynik `find`a ma wielkość większą niż dostępna pamięć operacyjna?

Istnieje jednak zadziwiająco prosta odpowiedź na wszystkie powyższe problemy. Po prostu "połączmy" komendy `find` i `head` z użyciem potoku:

```
find | head
```

Takie rozwiązanie jest proste, wygodne, łatwe do zapamiętania i wydajne. Ale jak to naprawdę działa?

Potok oznacza, że `bash` uruchomi jednocześnia DWA procesy (komendy). Oczywiście w przypadku komputera o jednym procesorze komendy te nie będą się wykonywały w pełni równolegle, lecz procesor będzie wykonywał je naprzemiennie po kawałku, ale nie zmienia to faktu, że komenda `head` zacznie się wykonywać zanim komenda `find` się zakończy. Oprócz tego standardowe wyjście komendy `find` zostanie połączone ze standardowym wejściem komendy `head`. Oznacza to, że każda linia tekstu "wyprodukowana" przez `find` będzie widoczna dla `head` tak jak każde inne wejście (z klawiatury czy z pliku) i komenda ta nie musi być świadoma, że jej dane pochodzą z potoku. "Transferem danych" pomiędzy komendami zajmuje się jądro systemu operacyjnego z użyciem własnego buforu. Takie podejście posiada wiele zalet:

1. Transfer danych jest szybki, ponieważ odbywa się bez udziału plików tymczasowych i dysku twardego czy podobnych nośników.

2. Transfer jest "przezroczysty" dla wszystkich użytkowników systemu, więc zagrożenie podejrzenia czy modyfikacji przesyłania danych praktycznie nie istnieje.

3. Jeśli komenda wysyłająca "produkuje" dane wystarczajaco wolno, by komenda odbierająca zdążyła je przetworzyć na bieżąco, to wymagany rozmiar buforu jest bardzo mały. Czasami możliwa jest sytuacja, gdzie w buforze systemu operacyjnego jest jednocześnie tylko kilka bajtów/linii tekstu, nawet jeśli łączny rozmiar "wymienianych" danych wynosi gigajty.

Dla potwierdzenia ostatniego punktu można wykonać następującą komendę:

```
find KAT -print
```

gdzie `KAT` to katalog zawierający dużo plików (np. `~` czyli katalog domowy lub `/`), tak aby wywołanie takiej komendy trwało zauważalny czas (co najmniej sekunda). Następnie wykonujemy:

```
find KAT -print | head
```

Komenda ta wykona się dużo szybciej niż poprzednia, ponieważ `head` domyślnie drukuje tylko 10 pierwszych otrzymanych linii, więc po uzyskaniu 10 nazw plików od `find` po prostu zignoruję resztę i zakończy pracę.

Należy jednak zadać pytanie: co się stanie, jeśli `find` będzie produkował swoje wyjście zbyt wolno i bufor będzie pusty gdy `head` będzie próbowało odczytać kolejne dane? Komenda `head` zostanie wtedy zablokowana do czasu pojawienia się nowych danych w buforze lub odczytania końca pliku. Analogicznie, gdy `find` będzie produkował dane zbyt szybko i przepełni bufor, to zostanie zablokowany przy próbie zapisu do czasu zwolnienia części bufora.

Na tym jednak nie koniec możliwości potoków. Ich praktyczna przydatność bierze się także z tego, że wynik drugiej komendy również można połączyć potokiem z trzecią komendą. Powstaje w ten sposób "łańcuch", który może osiąnąć znaczą długość:

```
command1 | command2 | command3 | command4
```

Taki potok będzie działał o ile wszystkie komendy poprawnie wykorzystują dane na swoim standardowym wejściu/wyjściu. W systemach uniksowych taka praktyka jest często zwana "filtrowaniem", jako że poszczególne komendy w potoku analizują otrzymane dane i zwykle modyfikują je lub obcinają ich część. Przykładowo:

```
cat file1 | sort | grep Ala | tail | tee file2
```

Ten potok spowoduje wysłanie zawartości pliku `file1` (komenda `cat`) w celu posortowania linia po linii (`sort`), następnie usunięte z potoku zostaną linie, które nie zawierają tekstu `"Ala"` (`grep`), potem zostaną usunięte wszystkie linie poza ostatnimi dziesięcioma (`tail`), zaś końcowy wynik trafi zarówno na standardowe wyjście potoku (konsolę) oraz do pliku `file2` (`tee`).

Poniżej znajduje się lista komend przydatnych w pracy z potokami:

1. `sort`. Sortuje dane linia po linii. Brzmi to dosyć prosto, ale komenda ta dysponuje znacznymi możliwościami oferując różne tryby sortowania (numeryczne, alfabetyczne itp.) oraz pozwala na sortowanie względem różnych kluczy (kolumn). Komenda może rozpocząć wypisywania wyjścia dopiero jak pozna całe swoje wejście (może spowalniać potok).

2. `tac`. Odwrotność `cat` – łączy i listuje podane pliki, ale każdy plik jest listowany od końca (odwrócona kolejność linii).

3. `rev`. Listuje podane pliki, odwracając kolejność *znaków* w każdej linii.

4. `uniq`. Domyślnie usuwa powtarzające się linie (zostawia jedynie linie unikalne), ale z łatwością można go zmusić do zachowania odwrotnego. Uwaga! `uniq` działa bardzo naiwnie i po prostu usuwa wszystkie *następujące po sobie* identycznie linie poza pierwszą. Oznacza to, że do poprawnej pracy `uniq` musi otrzymać dane *posortowane*! Przy użyciu odpowiednich opcji komenda `sort` może również działać jak `uniq`.

5. `head`. Domyślnie wypisuje 10 pierwszych linii wejścia i ignoruje resztę. Bardziej ogólnie można ją zmusić do wypisywania N pierwszych linii wejścia lub np. N pierwszych znaków każdej linii.

6. `tail`. Analogicznie do `head`, ale zajmuje się końcowymi liniami/znakami pliku, a nie początkowymi.

7. `grep`. Rozbudowana komenda używana do filtrowania wejścia w poszukiwaniu linii pasujących do wzorca (zwykle danego wyrażeniem regularnym tzw. regexem). Na laboratorium z potoków dozwolone jest jedynie proste użycie tej komendy (bez regexów).

8. `tee`. Użyteczna komenda nazwana od litery "T", która działa jak trójnik. Nie zmienia swojego standardowego wejścia, ale podwaja je i wysyła do dwóch miejsc: 1) na swoje standardowe wyjście (co pozwala kontynuować potok) oraz do wskazanego pliku.

9. `tr`. Komenda służąca do zastępowania pewnych znaków na wejściu innymi znakami zgodnie z podaną definicją.

2 Pętla `while`

`While` jest konstrukcją składniową `bash`a, jednocześnie podobną jak i odmienną od konstrukcji `for`. Zacznijmy od założenia, że chcemy przetworzyć dane wypisane przez komendę `find` w pętli. Teoretycznie możemy do tego wykorzystać pętlę `for` w następujący sposób:

```
for var in `find`
do
   (...)
done
```

Podejście to nie jest jednak pozbawione wad. Pętla NIE zacznie pracy dopóki wywołanie `find` się nie zakończy i nie będzie można skonstruować listy dla pętli. Ponadto, lista dla pętli `for` dzieli wyrazy względem każdego białego znaku, co oznacza, że wyjście `find`a postaci:

```
plik1 dane
plik2 dane
```

spowoduje 4 iteracje pętli, a to nie jest zwykle to co nas interesuje w tym przypadku. Dobrym pomysłem jest wykorzystanie potoku, ale pętla `for` nie jest do tego przystosowana. Wywołanie:

```
find | for var in somelist
```

nie ma sensu, gdyż pętla `for` nie analizuje swojego wejścia.

Rozwiązaniem jest użycie pętli `while`, ale aby je zastosować należy wpierw zrozumieć samą ideę tej pętli. Składnia `while` wygląda następująco:

```
while warunek
do
   (...)
done
```

W przeciwieństwie do pętli `for`, `while` będzie kontynuować pracę dopóki kod powrotu komendy `warunek` będzie równy prawdziwe (tzn. 0). W wielu zastosowaniach rolę warunek dla `while` pełni komenda `test` (`[`):

```
var=10
while [ $var -gt 0 ]
do
   echo $var
   var=$[var-1]
done
```

Taki warunek nie pomoże jednak w naszym przypadku. Do użycia `while` wraz z potokiem potrzebna nam jest jeszcze jedna komenda: `read`.

3 Read

Rola komendy `read` jest prosta: pobiera ona pojedynczą linię wejścia i rozdziela ją pomiędzy podane zmienne. Przykładowo:

```
read var1 var2 var3
```

odczyta (i usunie!) JEDNĄ linię ze swojego standardowego wejścia i spróbuje podzielić tą linię na 3 części względem (domyślnie) białych znaków: spacji i tabulatora. Jeśli teraz na wejście `read` wyślemy tekst `"jakis przykladowy tekst"`, to w rezultacie nastąpi przypisanie:

```
var1=jakis
var2=przykladowy
var3=tekst
```

Zdaje się to proste (pamiętamy, że wykorzystana linia znika z wejścia `read`). Powstaje jednak pytanie co stanie się gdy liczba "słów" w linii będzie różna od liczby podanych zmiennych (szczególnie istotne, gdy `find` będzie produkował linie o różnej liczbie słów dla różnych przypadków)? Jeśli podamy za dużo zmiennych, to nadmiarowe zmienne nie będą miały przypisanej wartości. Jeśli więc wyślemy w powyższym przykładzie tekst `"jakis przykladowy"`, to zmienna `var3` będzie po prostu pusta. Z drugiej strony, jeśli podamy za mało zmiennych, to wszystkie nadmiarowe słowa rozdzielanej linii zostaną umieszczone w ostatniej zmiennej. Jeśli więc podamy na nasz `read` tekst `"jakis przykladowy nieco dluzszy text"`, to nastąpi przypisanie:

```
var1=jakis
var2=przykladowy
var3=nieco dluzszy tekst
```

Należy też zauważyć, że kod powrotu komendy `read` przyjmuje wartość 0, chyba że wystąpił błąd lub koniec pliku.

4 While + read

Możemy teraz użyć obu opisanych komend by przetworzyć wyjście `find`a w potoku:

```
find | while read jakies zmienne
do
   (...)
done
```

Jak to działa? Dzięki zastosowaniu potoku wyjście `find`a zostaje skierowane na komendę `while`. Komenda ta jest jednak złożoną komendą, na którą może składać się wiele innych komend "wewnątrz" pętli. Trik polega na tym, że wszystkie komendy "wewnątrz" `while` współdzielą z nią standardowe wejście i wyjście, dotyczy to więc także `read`. Oznacza to, że `read` będzie odczytywał (i "usuwał") wejście `find`a, które następnie zostanie rozdzielone na podane zmienne, które można z kolei wykorzystać we wnętrzy pętli. Co więcej, gdy `read` przeczyta wszystkie linie, to "odnajdzie" koniec pliku i zmieni swój kod powrotu na niezerowy, co zakończy pętlę. Należy jednak cały czas pamiętać, że wejście `while` jest współdzielone przez wszystkie komendy w obrębie pętli, a odczytane wejście znika, więc trzeba zadbać, by odpowiednie komendy odczytywały odpowiedni fragment wejścia. Sama pętla `while` i wiele komend nie będzie w ogóle ruszało tego wejścia, ale należy zachować ostrożność. Dotyczy to sytuacji, gdy w pętli znajduje się więcej niż jeden `read`, a już szczególnie gdy różne komendy `read` umieszczone są pod różnymi warunkami i ciężko przewidzieć, które linie wejścia zostaną "wyłapane" przez który `read`. Przykład takiej sytuacji:

```
find | while read var1 var2
do
   read var3 var4
   if [ $var2 = $var4 ]; then
      read var5
   fi
done
```

Poprawne sformułowanie komend `while` i `read` to jednak nie koniec. Równie ważna jest poprawna definicja komendy `find`, a ściślej określenie formatu danych wyjściowych jaki `find` ma stosować, by `while/read` mogły je przetworzyć. Przykładowo, jeśli `find` wysyła dane w formacie:

```
znacznik1 jakiesdane
znacznik2 jakiesdane
```

to wystarczy prosta konstrukcja pętli typu:

```
find | while read var1 var2
do
   if [ var1 = znacznik1 ]; then
       # jakas akcja
   else
       # inna akcja
   fi
done
```

W tym przykładzie "znacznik1" oraz "znacznik2" są jakimś tekstem, który posłuży konstrukcji `while/read` do zorientowania się z jakim przypadkiem mamy do czynienia w danej linii. Ten sam przykład można rozwiązać inaczej. Jeśli zmusimy `find` (poprzez `-printf`) do formatowania wyjścia w taki sposób:

```
znacznik1
jakiesdane
znacznik2
jakiesdane
```

to pętlę należy do tego dostosować:

```
find | while read var1
do
   read var2
   if [ var1 = znacznik1 ]; then
       # jakas akcja
   else
       # inna akcja
   fi
done
```

Dobre zaprojektowanie pętli jest tym bardziej istotne, jeśli `find` wypisuje różną liczbę słów dla różnych przypadków. Należy wtedy tak rozpisać `read` (`ready`), aby pomieścić każdy przypadek i używać tych zmiennych właściwych dla danego przypadku.

Pozostają jeszcze do omówienia pewne kwestie praktyczne wynikające z zastosowania `while` jako części potoku. Po pierwsze, pętla ta jest traktowana jako zwyczajna komenda i nie musi kończyć potoku:

```
find | while read (...)
do
   (...)
done | kontynuacja | potoku
```

Możliwe jest połączenie ze sobą w potoku dwóch pętli `while`, aczkolwiek podczas laboratorium nie jest to niezbędne i zwykle oznacza błąd.

Drugi problem jest bardziej złożony. Rozważmy poniższy skrypt:

```
var="przed"
find | while read jakies zmienne
do
   var="po"
done
echo $var
```

Każda komenda "dziedziczy" wartości zmiennych po komendzie, która ją wywołała. Ponadto `while`, będąc komendą złożoną, może zmieniać wartości zmiennych jak pokazano powyżej. Proste. Jednakże, jeśli uruchomimy powyższy skrypt, to okaże się, że ostatnia komenda `echo` wypisze tekst "przed" zamiast "po". Dlaczego?

"Winowajcą" tej sytuacji jest potok, gdyż obie jego strony (w tym przypadku `find` i `while`) wykonują się jednocześnie (współbieżnie). Różnica polega na tym, że komendy te wykonują się w ramach osobnych "skryptów". Co więcej, komenda `echo` należy do tego samego "skryptu" co `find` (powiedzmy, że jest to skrypt A), zaś `while` należy do drugiego skryptu (B). Komenda `echo` odziedziczy więc zmiany po swoim poprzedniku, czyli `find`, a nie `while`.

My chcemy jednak, by `echo` było świadome zmian dokonanych przez `while`. Jak tego dokonać? Należy po prostu umieścić komendy `while` i `echo` w tym samym "skrypcie". Co prawda "składnikiem" potoku może być tylko pojedyncza komenda, ale istnieje sposób, by "umieścić" ciąg komend w osobnym podskrypcie. Służą do tego nawiasy okrągłe `"("` oraz `")"`:

```
var="przed"
find | ( while read jakies zmienne
do
   var="po"
done
echo $var )
```

Po tej zmianie `find` nie jest połączony bezpośrednie z `while`, a z całym podskryptem, na który składa się `while` i `echo`, a skrypt zadziała teraz zgodnie z oczekiwaniem, wypisując na ekran tekst "po". To specjalne znaczenie nawiasów okrągłych jest powodem, dla którego na poprzednim laboratorium należało je poprzedzać odwrotnym ukośnikiem (lub otaczać apostrofami), by nie zostały zinterpretowane przez `bash` jako próba utworzenia podskryptu.