
Helion
Linux
Wiersz poleceń
i skrypty powłoki
BIBLIA
Wydanie IV
Richard Blum
Christine Bresnahan
WILEY

Tytuł oryginału: Linux® Command Line and Shell Scripting Bible, 4th Edition

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-8322-074-1

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/linwb4
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

Część III. Zaawansowane techniki skryptowe 431

ROZDZIAŁ

11

Podstawy budowy skryptów

W TYM ROZDZIALE:

- **Używanie kilku poleceń na raz**
- **Tworzenie pliku skryptu**
- **Wyświetlanie komunikatów**
- **Zmienne**
- **Przekierowywanie wejścia i wyjścia**
- **Potoki**
- **Wykonywanie działań matematycznych**
- **Wychodzenie ze skryptu**

Znasz już podstawy systemu Linux i wiersza poleceń, więc możesz zacząć pisać kod. W tym rozdziale omawiamy podstawy pisania skryptów powłoki. Musisz je opanować, aby móc zacząć pisać własne arcydzieła sztuki skryptowej.

Używanie kilku poleceń na raz

Do tej pory używaliśmy interfejsu wiersza poleceń (CLI) powłoki do wpisywania poleceń i przeglądania zwracanych przez nie wyników. Podstawę skryptów stanowi możliwość wykonywania wielu poleceń i przetwarzania ich wyników, które mogą być nawet przekazywane od jednego polecenia do innego. Powłoka umożliwia tworzenie łańcuchów poleceń, które są wykonywane w jednym kroku.

Jeśli chcesz wykonać dwa polecenia razem, możesz je wpisać w wierszu poleceń, rozdzielając średnikiem:

```
$ date ; who
pon 01 cze 15:36:09 EST 2020
Christine tty2         2020-06-01 15:26
Samantha  tty3         2020-06-01 15:26
Timothy   tty1         2020-06-01 15:26
user      tty7         2020-06-01 14:03 (:0)
user      pts/0        2020-06-01 15:21 (:0.0)
$
```

Gratulacje, to był Twój pierwszy skrypt! W tym prostym przykładzie użyto dwóch poleceń powłoki Bash. Polecenie `date` zostaje wykonane pierwsze i powoduje wyświetlenie aktualnej daty i godziny. Następnie zostaje wykonane polecenie `who`, które pokazuje, kto jest aktualnie zalogowany w systemie. Przy użyciu tej techniki można połączyć dowolną liczbę poleceń, pod warunkiem że łączna liczba zawartych w nich znaków nie przekroczy 255.

Choć w przypadku niewielkich skryptów ta technika dobrze się sprawdza, jej poważną wadą jest to, że za każdym razem, gdy chcemy wykonać skrypt, całe polecenie musimy wpisywać w wierszu poleceń od nowa. Aby tego nie robić, wszystkie polecenia można zapisać w pliku tekstowym. Potem taki plik można zwyczajnie uruchomić, gdy trzeba wykonać znajdujące się w nim polecenia.

Tworzenie pliku skryptu

Aby zapisać polecenia w pliku tekstowym, należy użyć edytora tekstu (rozdział 10.).

Podczas tworzenia pliku skryptu powłoki w jego pierwszym wierszu należy określić używaną powłokę za pomocą następującego formatu:

```
#!/bin/bash
```

W normalnych skryptach powłoki krzyżyk (`#`) oznacza komentarz. Komentarze znajdujące się w skryptach powłoki nie są przetwarzane przez powłokę. Jednak pierwszy wiersz w takim pliku jest wyjątkowy i w jego przypadku krzyżyk z wykrzyknikiem informuje powłokę, w jakiej powłoce ma zostać wykonany dany skrypt (tak, można używać powłoki Bash i wykonywać skrypty przy użyciu innej powłoki).

Po określeniu powłoki można wpisać polecenia, po jednym na wiersz. W razie potrzeby za pomocą krzyżyka można też wprowadzać komentarze, np.:

```
#!/bin/bash
# ten skrypt wyświetla datę i zalogowanych użytkowników
date
who
```

To wszystko. Ewentualnie można umieścić dwa polecenia w jednym wierszu, rozdzielając je średnikiem, ale w skryptach powłoki polecenia można umieszczać także w osobnych wierszach. Skrypt wykona je w kolejności wpisania w pliku.

Zwróć też uwagę na dodatkowy wiersz zaczynający się od krzyżyka, który jest komentarzem. Wiersze zaczynające się od tego znaku (z wyjątkiem pierwszego wiersza zaczynającego się od znaków `#!`) nie są interpretowane przez powłokę. To świetny sposób na pozostawianie informacji dla samego siebie o tym, co się dzieje w danym miejscu skryptu, aby po paru latach móc sobie to szybko przypomnieć.

Zapisz ten skrypt w pliku o nazwie *test1*, ale zanim będzie można go uruchomić, należy wykonać jeszcze parę czynności.

Jeśli teraz spróbujesz wykonać swój plik, to czeka Cię rozczarowanie:

```
$ test1
bash: test1: command not found
$
```

Przede wszystkim musimy sprawić, aby powłoka Bash znalazła nasz plik skryptu. Może pamiętasz z rozdziału 6., że system Linux znajduje polecenia dzięki zmiennej środowiskowej o nazwie PATH. Wyświetlenie jej zawartości pozwala się dowiedzieć, w czym tkwi nasz problem:

```
$ echo $PATH
/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/bin:/usr/bin
:/bin:/usr/local/sbin:/usr/sbin:/sbin:/home/user/bin $
```

Zmienna PATH określa tylko kilka katalogów, w których mogą się znajdować polecenia. Aby powłoka znalazła skrypt *test1*, musimy wykonać jedną z dwóch czynności:

- dodać katalog zawierający nasz plik skryptu do zmiennej środowiskowej PATH;
- podać bezwzględną lub względną ścieżkę do pliku skryptu w wierszu poleceń.

Niektóre dystrybucje Linuksa mają katalog ***$HOME****/bin* w zmiennej środowiskowej PATH. W ten sposób zostaje utworzone miejsce w katalogu głównym każdego użytkownika, w którym można umieszczać pliki przeznaczone do wykonywania przez powłokę.

W tym przykładzie skorzystamy z drugiej metody, aby poinformować powłokę, gdzie dokładnie znajduje się nasz plik skryptu. Przypomnijmy, że do pliku znajdującego się w bieżącym katalogu można się odnieść za pomocą kropki:

```
$ ./test1
bash: ./test1: Permission denied
$
```

Teraz powłoka znalazła skrypt, ale pojawił się nowy problem. Otrzymaliśmy informację, że nie mamy uprawnień do wykonania tego pliku. Wystarczy zerknąć na jego uprawnienia, aby się dowiedzieć, w czym tkwi problem:

```
$ ls -l test1
-rw-r--r-- 1 user    user            73 Jun 02 15:36 test1
$
```

Kiedy tworzyliśmy plik *test1*, jego domyślne uprawnienia zostały zdeterminowane przez wartość umask, a ponieważ zmienna umask ma wartość 022 (rozdział 7.), system nadał właścicielowi tego pliku uprawnienia tylko do jego odczytu i zapisu.

Następnym krokiem jest więc nadanie właścicielowi uprawnień do wykonywania pliku za pomocą polecenia chmod (rozdział 7.):

```
$ chmod u+x test1
$ ./test1
pon 01 cze 15:38:19 EST 2020
Christine tty2        2020-06-01 15:26
Samantha tty3         2020-06-01 15:26
Timothy  tty1         2020-06-01 15:26
user     tty7         2020-06-01 14:03 (:0)
user     pts/0        2020-06-01 15:21 (:0.0) $
```

Sukces! Wszystkie elementy układanki znalazły się na swoim miejscu, dzięki czemu udało się wykonać nowy plik skryptu powłoki.

Wyświetlanie komunikatów

Większość poleceń powłoki generuje własne wyniki, wyświetlane w konsoli, w której został uruchomiony skrypt. Czasami jednak chcielibyśmy dodać własne wiadomości tekstowe, które pomagają użytkownikowi skryptu zrozumieć, co się dzieje. Służy do tego polecenie echo, które wyświetla proste łańcuchy tekstu wpisane za nim:

```
$ echo To jest test
To jest test
$
```

Zauważ, że domyślnie nie trzeba ujmować tekstu do wyświetlenia w cudzysłów. Czasami jednak, kiedy tekst zawiera cudzysłowy, sprawy mogą się komplikować. Spójrz na poniższy przykład w języku angielskim:

```
$ echo Let's see if this'll work
Lets see if thisll work
$
```

Polecenie echo do oznaczania granic łańcuchów tekstu używa prostych cudzysłowów pojedynczych lub podwójnych. Jeśli jednego z tych dwóch typów użyjesz w swoim tekście, to cały łańcuch musisz ująć w drugim:

```
$ echo "This is a test to see if you're paying attention"
This is a test to see if you're paying attention
$ echo 'Rich says "scripting is easy".'
Rich says "scripting is easy".
$
```

Teraz wszystkie cudzysłowy są prawidłowo wyświetlone w wyniku.

Polecenie echo można umieścić w dowolnym miejscu skryptu, w którym chcemy wyświetlić dodatkowe informacje:

```
$ cat test1
#!/bin/bash
# ten skrypt wyświetla datę i zalogowanych użytkowników
echo Data i godzina:
date
echo "Sprawdźmy, kto jest zalogowany:"
who
$
```

Wynik działania tego skryptu będzie następujący:

```
$ ./test1
Data i godzina:
pon 01 cze 15:41:13 EST 2020
Sprawdźmy, kto jest zalogowany:
Christine tty2       2020-06-01 15:26
Samantha tty3       2020-06-01 15:26
Timothy  tty1       2020-06-01 15:26
user     tty7       2020-06-01 14:03 (:0)
user     pts/0      2020-06-01 15:21 (:0.0)
$
```

W porządku, ale jak wyświetlić łańcuch tekstu w tym samym wierszu co wynik polecenia? W tym celu można użyć parametru -n instrukcji echo. W pierwszej naszej instrukcji echo należałoby wprowadzić taką zmianę:

```
echo -n "Data i godzina: "
```

W tym przypadku łańcuch do wyświetlenia musimy umieścić w cudzysłowie, aby na jego końcu móc dodać spację. Wynik polecenia zaczyna się dokładnie w miejscu zakończenia łańcucha. Teraz wynik całego skryptu wygląda tak:

```
$ ./test1
Data i godzina: pon 01 cze 15:41:13 EST 2020
Sprawdźmy, kto jest zalogowany:
Christine tty2       2020-06-01 15:26
Samantha  tty3       2020-06-01 15:26
Timothy   tty1       2020-06-01 15:26
user      tty7       2020-06-01 14:03 (:0)
user      pts/0      2020-06-01 15:21 (:0.0)
$
```

Doskonale! Polecenie `echo` to niezwykle istotny element w skryptach powłoki, które oddziałują z użytkownikiem. Będziesz go używać bardzo często, szczególnie przy wyświetlaniu wartości zmiennych skryptów. Sprawdźmy więc, jak się to robi.

Zmienne

Wykonywanie pojedynczych poleceń w skryptach powłoki to przydatna, choć ograniczona umiejętność. W poleceniach powłoki często chcemy umieszczać różne dane, które powinny zostać przetworzone. Do tego celu można używać **zmiennych**, które pozwalają na czasowe przechowywanie informacji w skrypcie, aby były dostępne dla poleceń. W tym podrozdziale pokazujemy, jak posługiwać się zmiennymi w skryptach powłoki.

Zmienne środowiskowe

Znasz już jeden rodzaj zmiennych Linuksa — zmienne środowiskowe, które opisaliśmy w rozdziale 6. Dostęp do ich wartości można uzyskać także w skryptach powłoki.

Powłoka przechowuje zmienne środowiskowe zawierające różne informacje na temat systemu, takie jak nazwa systemu, nazwa zalogowanego użytkownika, identyfikator systemowy użytkownika (tzw. UID), domyślny katalog główny użytkownika i ścieżka wyszukiwania używana przez powłokę do znajdowania programów. Listę wszystkich aktywnych zmiennych środowiskowych można uzyskać za pomocą polecenia `set`:

```
$ set
BASH=/bin/bash
...
HOME=/home/Samantha
HOSTNAME=localhost.localdomain
HOSTTYPE=i386
IFS=$' \t\n'
IMSETTINGS_INTEGRATE_DESKTOP=yes
IMSETTINGS_MODULE=none
LANG=en_US.utf8
LESSOPEN='|/usr/bin/lesspipe.sh %s'
LINES=24
LOGNAME=Samantha
...
```

Do zawartości tych zmiennych środowiskowych w skryptach można się dostać przez podanie nazwy poprzedzonej znakiem dolara, jak pokazano w poniższym przykładzie:

```
$ cat test2
#!/bin/bash
# wyświetla informacje o użytkownikach z systemu
echo "Informacje na temat użytkownika $USER"
echo UID: $UID
echo HOME: $HOME
$
```

Zmienne środowiskowe `$USER`, `$UID` i `$HOME` umożliwiają wyświetlenie informacji o zalogowanym użytkowniku. Wyniki powinny wyglądać tak:

```
$ chmod u+x test2
$ ./test2
Informacje na temat użytkownika Samantha
UID: 1001
HOME: /home/Samantha
$ $
```

Zwróć uwagę, że zmienne środowiskowe użyte w poleceniu `echo` w chwili wykonania skryptu zostały zastąpione przez ich obecne wartości. Zauważ też, że choć w pierwszym łańcuchu umieściliśmy zmienną systemową `$USER` w podwójnym cudzysłowie, to skrypt i tak prawidłowo odczytał nasz zamiar. Ta metoda ma jednak pewną wadę. Spójrz na poniższy przykład:

```
$ echo " The cost of the item is $15"
The cost of the item is 5
```

Zdecydowanie nie o to nam chodziło. Kiedy interpreter skryptu znajduje znak dolara w obrębie cudzysłowu, zakłada, że jest to odwołanie do zmiennej. W tym przykładzie skrypt spróbował wyświetlić wartość zmiennej `$1` (która nie jest zdefiniowana), a po niej wyświetlił liczbę 5. Aby wyświetlić znak dolara, należy go poprzedzić wstecznym ukośnikiem:

```
$ echo "The cost of the item is \$15"
The cost of the item is $15
```

Teraz jest lepiej. Ukośnik sprawił, że znak dolara w skrypcie został zinterpretowany jako zwykły znak, a nie oznaczenie zmiennej. W następnej sekcji pokazujemy, jak tworzyć własne zmienne w skryptach.

Można spotkać także zmienne w formacie `${zmienna}`. Klamra jest często używana do oddzielania nazwy zmiennej od znaku dolara.

Zmienne użytkownika

W skryptach powłoki można używać nie tylko zmiennych środowiskowych, ale także własnych zmiennych. Pozwala to na tymczasowe przechowywanie danych do użytku w różnych miejscach skryptu, co czyni go bardziej podobnym do prawdziwego programu komputerowego.

Nazwa zmiennej użytkownika może zawierać maksymalnie 20 znaków — liter, cyfr i znaków podkreślenia. Wielkość liter ma znaczenie, a więc `Var1` różni się od `var1`. Ta zasada często wpędza początkujących programistów w kłopoty.

Wartość zmiennej przypisuje się za pomocą znaku równości. Między nazwą zmiennej, znakiem równości i wartością nie może być ani jednej spacji (kolejna pułapka na początkujących). Oto kilka przykładów przypisania wartości zmiennym użytkownika:

```
var1=10
var2=-57
var3=testowanie
var4="dalsze testowanie"
```

Skrypt powłoki przechowuje wszystkie zmienne jako łańcuchy tekstu i to od poleceń zależy ich interpretacja jako określonego typu danych. Zmienne zdefiniowane w skrypcie powłoki istnieją przez cały czas działania skryptu i zostają usunięte wraz z jego zakończeniem.

Do zmiennych użytkownika, tak jak do zmiennych systemowych, można się odwoływać za pomocą znaku dolara:

```
$ cat test3
#!/bin/bash
# testowanie zmiennych
days=10
guest="Katie"
echo "$guest zameldowała się $days dni temu."
days=5
guest="Jessica"
echo "$guest zameldowała się $days dni temu."
$
```

Wynik działania tego skryptu jest następujący:

```
$ chmod u+x test3
$ ./test3
Katie zameldowała się 10 dni temu.
Jessica zameldowała się 5 dni temu.
$
```

W miejscu każdego odwołania do zmiennej jest wstawiana aktualnie przypisana jej wartość. Należy pamiętać, że znaku dolara używa się tylko w odwołaniach do wartości zmiennych, natomiast do przypisywania wartości zmiennym znaku tego się nie używa. Poniższy przykład ilustruje, co mamy na myśli:

```
$ cat test4
#!/bin/bash
# przypisanie wartości zmiennej do innej zmiennej

value1=10
value2=$value1
echo Otrzymana wartość to $value2
$
```

W instrukcji przypisania wartości zmiennej `value1` innej zmiennej użycie znaku dolara było konieczne. Wynik wykonania tego kodu jest następujący:

```
$ chmod u+x test4
$ ./test4
Otrzymana wartość to 10
$
```

Jeśli zapomnimy dodać znak dolara i zmiennej `value2` przypiszemy wartość w taki sposób:

```
value2=value1
```

to otrzymamy następujący wynik:

```
$ ./test4
Otrzymana wartość to value1
$
```

Bez znaku dolara powłoka interpretuje nazwę zmiennej jako zwykły tekst, co prawie na pewno jest błędem.

Podmienianie poleceń

Jedną z najbardziej przydatnych cech skryptów powłoki jest możliwość wyciągania informacji z wyników polecenia i przypisywania ich do zmiennych. Po przypisaniu wyników do zmiennej można jej używać w dowolnym miejscu skryptu. Przydaje się to, kiedy przetwarzamy dane w swoich skryptach.

Wynik polecenia można przypisać do zmiennej na dwa sposoby:

- za pomocą odwróconego apostrofu (`),
- za pomocą formatu `$()`.

Zwróć uwagę na odwrócony apostrof — to nie jest zwykły pojedynczy znak cudzysłowu używany w łańcuchach tekstu. Nie jest on zbyt często używany poza skryptami powłoki, przez co wiele osób nawet nie wie o jego obecności na klawiaturze. Musisz jednak się z nim zapoznać, ponieważ w wielu skryptach powłoki odgrywa bardzo ważną rolę. Podpowiedź: na standardowej klawiaturze zazwyczaj znajduje się w tym samym miejscu co tylda (~).

Technika zastępowania poleceń umożliwia przypisanie wyniku polecenia powłoki do zmiennej. Choć nie wydaje się to niczym niesamowitym, jest to jedna z najważniejszych technik w programowaniu skryptowym.

Całe polecenie należy umieścić między odwrotnymi apostrofami:

```
testing=`date`
```

lub należy użyć formatu `$()`:

```
testing=$(date)
```

Powłoka wykonuje polecenia znajdujące się między znakami podmieniania poleceń i przypisuje wynik do zmiennej `testing`. Zwróć uwagę na brak spacji przed i za znakiem równości. Poniżej znajduje się przykład utworzenia zmiennej przy użyciu wyniku zwróconego przez normalne polecenie powłoki:

```
$ cat test5
#!/bin/bash
testing=$(date)
echo "Data i godzina: " $testing
$
```

Zmienna `testing` otrzymuje wynik polecenia `date` i zostaje użyta w poleceniu `echo`, które wyświetla jej zawartość. Wynik wykonania powyższego skryptu wyglądałby tak:

```
$ chmod u+x test5
$ ./test5
Data i godzina: pon 01 cze 15:45:25 EDT 2020
$
```

Ten przykład nie jest jakoś specjalnie fascynujący (równie dobrze polecenie można było umieścić w instrukcji `echo`), ale z wynikiem polecenia zapisanym w zmiennej można zrobić niemal wszystko.

Poniżej znajduje się popularny przykład pokazujący, jak za pomocą podmieniania poleceń pobrać bieżącą datę i użyć jej do utworzenia niepowtarzalnej nazwy pliku w skrypcie:

```
#!/bin/bash
# kopiuje listing zawartości katalogu /usr/bin do pliku dziennika
today=$(date +%y%m%d)
ls /usr/bin -al > log.$today
```

Zmiennej `today` zostaje przypisany wynik sformatowanego polecenia `date`. Ta technika jest często używana w celu pobierania daty do tworzenia nazw plików dziennika. Format `+%y%m%d` nakazuje poleceniu `date` przedstawienie daty w postaci dwucyfrowych oznaczeń roku, miesiąca i dnia:

```
$ date +%y%m%d
200601
$
```

Ten skrypt przypisuje zmiennej wartość, która następnie zostaje wykorzystana przy tworzeniu nazwy pliku. Sam plik zawiera przekierowany wynik (szerzej na ten temat piszemy w punkcie „Przekierowywanie wejścia i wyjścia") listingu zawartości katalogu. Po uruchomieniu tego skryptu w katalogu powinien się pojawić nowy plik:

```
-rw-r--r--    1 user    user        769 01 cze 16:15 log.200601
```

Nazwa pliku dziennika, który pojawił się w katalogu, zawiera wartość zmiennej `$today`. Sam plik zawiera listę zawartości katalogu */usr/bin*. Jeśli ten skrypt zostanie uruchomiony następnego dnia, utworzy plik dziennika o nazwie *log.200602* odpowiadającej nowemu dniowi.

Technika podmieniania poleceń tworzy tzw. podpowłokę w celu wykonania danego polecenia. Podpowłoka to osobna powłoka wygenerowana z powłoki wykonującej skrypt. Z tego powodu wszystkie zmienne utworzone w skrypcie są niedostępne poleceniom wykonywanym w podpowłoce.

Podpowłoki są także tworzone w czasie wykonywania poleceń w wierszu poleceń przy użyciu ścieżki *./*, natomiast nie są tworzone, jeżeli polecenie zostanie wykonane bez podania ścieżki. Jeśli jednak użyjesz wbudowanego polecenia powłoki, to podpowłoka nie zostanie wygenerowana. Zachowaj ostrożność podczas wykonywania skryptów w wierszu poleceń powłoki!

Przekierowywanie wejścia i wyjścia

Czasami wyniki polecenia chcemy nie tylko wyświetlić na monitorze, ale także chcemy je gdzieś zapisać. Powłoka Bash udostępnia kilka operatorów, które umożliwiają **przekierowywanie** wyników polecenia do innego miejsca (np. pliku). Przekierowanie może dotyczyć zarówno wejścia, jak i wyjścia, a także może oznaczać przekazanie pliku na wejście polecenia. W tym podrozdziale opisujemy, co należy zrobić, aby zastosować przekierowanie w swoich skryptach powłoki.

Przekierowywanie wyjścia

Najbardziej podstawowym typem przekierowania jest wysłanie wyników polecenia do pliku. W powłoce Bash służy do tego znak większości (>):

```
polecenie > plikwyjściowy
```

Wszystko, co polecenie wyświetliłoby na monitorze, zostanie zapisane w pliku o podanej nazwie:

```
$ date > test6
$ ls -l test6
-rw-r--r--    1 user    user            29 Jun 01 16:56 test6
$ cat test6
pon 01 cze 16:56:58 EDT 2020
$
```

Operator przekierowania utworzył plik o nazwie *test6* (przy użyciu domyślnych ustawień maski `umask`) i przekierował do niego wynik z polecenia `date`. Gdyby ten plik już istniał, operator zastąpiłby jego zawartość nowymi danymi:

```
$ who > test6
$ cat test6
rich     pts/0       01 cze 16:55
$
```

Teraz plik *test6* zawiera wynik polecenia `who`.

Czasami zamiast nadpisywać zawartość pliku, wolimy dodać do niej wyniki polecenia — np. kiedy tworzymy plik dziennika mający zawierać dokumentację czynności wykonywanych w systemie. W takiej sytuacji, aby dodać nowe dane na końcu, możemy użyć podwójnego znaku większości (>>):

```
$ date >> test6
$ cat test6
rich     pts/0       01 cze 16:55
pon 01 cze 01 17:02:14 EDT 2020
$
```

Plik *test6* nadal zawiera wcześniejsze dane z poprzednio wykonanego polecenia `who` i dodatkowo zawiera nowe wyniki z polecenia `date`.

Przekierowywanie wejścia

Przekierowywanie wejścia to odwrotność przekierowywania wyjścia. Zamiast przekierowywać wyniki polecenia do pliku, pobieramy zawartość pliku i kierujemy ją do polecenia.

Symbolem tej operacji jest znak mniejszości (<):

```
polecenie < plikwejściowy
```

Łatwym sposobem na zapamiętanie tego jest zwrócenie uwagi na fakt, że polecenie zawsze znajduje się na pierwszym miejscu w wierszu, a symbol „wskazuje" kierunek przepływu danych. Znak mniejszości oznacza, że dane płyną z pliku wejściowego do polecenia.

Poniżej znajduje się przykład użycia przekierowania wejścia z poleceniem `wc`:

```
$ wc < test6
      2      11      60
$
```

Polecenie wc zwraca dane statystyczne dotyczące tekstu. Domyślnie uwzględnia trzy wartości:

- liczbę wierszy tekstu,
- liczbę słów,
- liczbę bajtów.

Przekierowując plik tekstowy do polecenia wc, można szybko się dowiedzieć, ile zawiera wierszy i słów oraz ile zajmuje bajtów. Z przykładu wynika, że plik *test6* zawiera 2 wiersze i 11 słów i zajmuje 60 bajtów.

Istnieje jeszcze jedna metoda przekierowywania wejścia, która nazywa się **śródliniowym przekierowywaniem wejścia** (ang. *inline input redirection*). Za jej pomocą dane wejściowe do przekierowania można określić w wierszu poleceń zamiast w pliku. W pierwszej chwili może się to wydać dziwne, ale metoda ta ma kilka zastosowań (takich jak opisane w podrozdziale „Wykonywanie działań matematycznych").

Symbol śródliniowego przekierowania wejścia to podwójny znak mniejszości (<<). Oprócz niego należy określić marker tekstowy oznaczający początek i koniec danych używanych na wejściu. Może to być dowolny łańcuch, aby na początku i na końcu danych był taki sam:

```
polecenie << marker
dane
marker
```

Podczas korzystania ze śródliniowego przekierowania wejścia w wierszu poleceń powłoka będzie prosić o dane przy użyciu dodatkowego wiersza poleceń zdefiniowanego w zmiennej środowiskowej PS2 (rozdział 6.). Tak to wygląda w praktyce:

```
$ wc << EOF
> test łańcuch 1
> test łańcuch 2
> test łańcuch 3
> EOF
       3       9        45
$
```

Dodatkowy wiersz poleceń prosi o podawanie kolejnych porcji danych, aż użytkownik wpisze marker oznaczający koniec. Polecenie wc policzy wiersze, słowa i bajty wprowadzonych danych.

Potoki

Czasami trzeba wysłać dane wyjściowe jednego polecenia na wejście innego polecenia. Można to zrobić za pomocą przekierowywania, choć jest to dość niezgrabne rozwiązanie:

```
$ rpm -qa > rpm.list
$ sort < rpm.list
abattis-cantarell-fonts-0.0.25-1.el7.noarch
abrt-2.1.11-52.el7.centos.x86_64
abrt-addon-ccpp-2.1.11-52.el7.centos.x86_64
abrt-addon-kerneloops-2.1.11-52.el7.centos.x86_64
abrt-addon-pstoreoops-2.1.11-52.el7.centos.x86_64
abrt-addon-python-2.1.11-52.el7.centos.x86_64
abrt-addon-vmcore-2.1.11-52.el7.centos.x86_64
abrt-addon-xorg-2.1.11-52.el7.centos.x86_64
abrt-cli-2.1.11-52.el7.centos.x86_64
abrt-console-notification-2.1.11-52.el7.centos.x86_64
...
```

Polecenie `rpm` zarządza pakietami oprogramowania zainstalowanymi w systemach zawierających narzędzie Red Hat Package Management (RPM), a więc takich jak np. CentOS. Po dodaniu parametrów `-qa` zwraca listę zainstalowanych pakietów, ale nie sortuje jej w żadnej konkretnej kolejności. Jeśli szukasz konkretnego pakietu lub grupy pakietów, to w wynikach polecenia `rpm` może być Ci trudno je znaleźć.

Za pomocą standardowego przekierowania wyników przekierowano wyniki polecenia `rpm` do pliku o nazwie *rpm.list*. Gdy polecenie zakończyło działanie, w pliku *rpm.list* znalazła się lista wszystkich zainstalowanych pakietów oprogramowania w systemie. Następnie przekierowanie wejścia zostało użyte w celu wysłania zawartości pliku *rpm.list* do polecenia `sort` w celu alfabetycznego posortowania nazw pakietów.

Udało się, ale to też był niezbyt zgrabny sposób na uzyskanie informacji. Zamiast przekierowywać wyniki polecenia do pliku, można przekierować je do innego polecenia. W tym procesie wykorzystuje się tzw. **potok**.

Symbol potoku, podobnie jak znak podmieniania poleceń (`), poza skryptami powłoki jest rzadko używany. Składa się z dwóch pionowych kresek ustawionych jedna nad drugą, ale w druku zazwyczaj wygląda jak jedna pionowa kreska (|). Na standardowej klawiaturze ten znak zazwyczaj znajduje się na tym samym klawiszu co ukośnik wsteczny (\). Potok umieszcza się między poleceniami, aby przekierować wyniki jednego do drugiego:

```
polecenie1 | polecenie2
```

Nie wyobrażaj sobie potoku jako wykonywania dwóch poleceń obok siebie. System Linux wykonuje je jednocześnie, łącząc je wewnętrznie. Kiedy pierwsze polecenie zwróci wynik, zostaje on natychmiast wysłany do drugiego polecenia. Do przesyłania danych nie są używane żadne pośrednie pliki ani bufory.

Teraz za pomocą potoku możemy bez problemu przekazać wyniki polecenia `rpm` bezpośrednio do polecenia `sort`, aby uzyskać posortowaną listę:

```
$ rpm -qa | sort
abattis-cantarell-fonts-0.0.25-1.el7.noarch
abrt-2.1.11-52.el7.centos.x86_64
abrt-addon-ccpp-2.1.11-52.el7.centos.x86_64
abrt-addon-kerneloops-2.1.11-52.el7.centos.x86_64
abrt-addon-pstoreoops-2.1.11-52.el7.centos.x86_64
abrt-addon-python-2.1.11-52.el7.centos.x86_64
abrt-addon-vmcore-2.1.11-52.el7.centos.x86_64
abrt-addon-xorg-2.1.11-52.el7.centos.x86_64
abrt-cli-2.1.11-52.el7.centos.x86_64
abrt-console-notification-2.1.11-52.el7.centos.x86_64
...
```

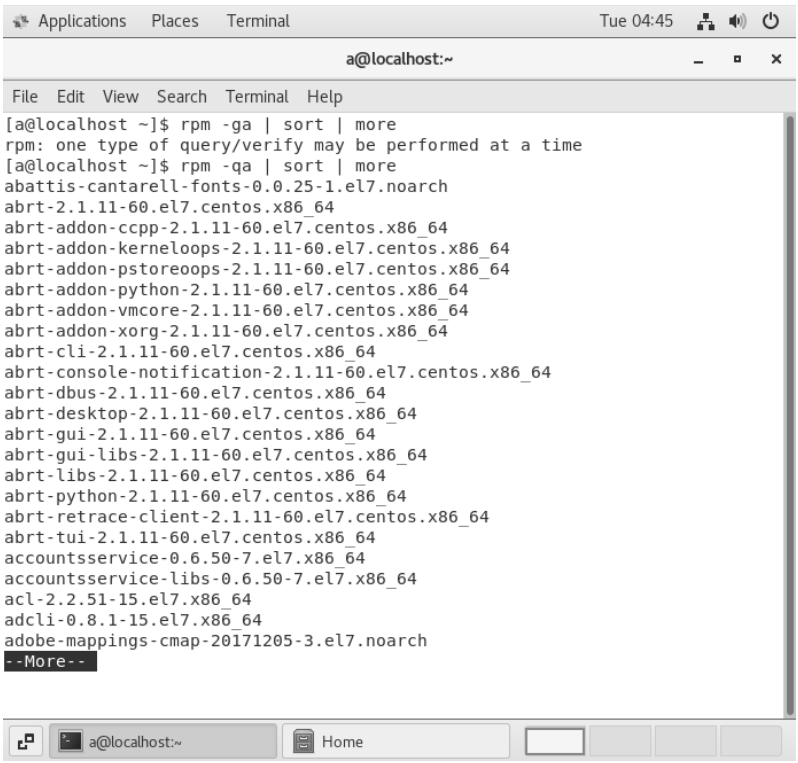
Jeśli nie umiesz czytać z niezwykłą prędkością, to pewnie nie udało Ci się wiele zobaczyć w wynikach wygenerowanych przez to polecenie. Potok działa na bieżąco, co oznacza, że gdy tylko polecenie `rpm` wygeneruje dane, polecenie `sort` natychmiast bierze się za ich obróbkę. Zanim polecenie `rpm` zakończy zwracanie danych, polecenie `sort` ma je już posortowane i rozpoczyna wyświetlanie ich na ekranie monitora.

W poleceniu można użyć nieograniczonej liczby potoków, tzn. można dowolnie długo przekazywać wyniki kolejnych poleceń do następnych, aby uzyskać dokładnie to, czego się potrzebuje.

Jako że w tym przypadku wyniki polecenia `sort` bardzo szybko przelatują przed oczami, można użyć jednego z poleceń paginacji tekstu (np. `less` lub `more`), aby wymusić zatrzymywanie wyników na każdym ekranie danych:

```
$ rpm -qa | sort | more
```

W tej sekwencji najpierw zostaje wykonane polecenie `rpm`, którego wyniki zostają przekazane do polecenia `sort`, którego wyniki z kolei zostaną przesłane do polecenia `more` w celu ich wyświetlenia po jednym ekranie na raz. Teraz możesz zatrzymywać wyświetlanie danych, aby je spokojnie przejrzeć, jak pokazano na rysunku 11.1.



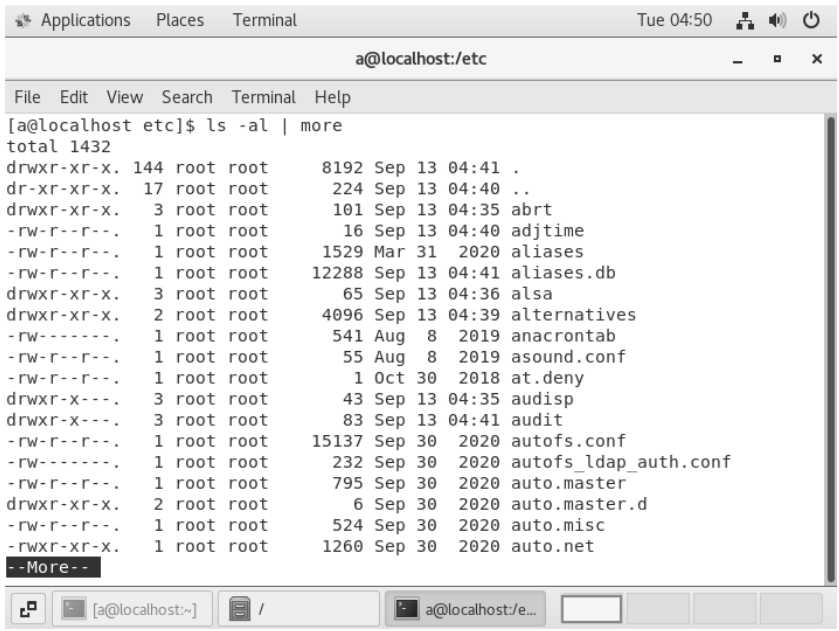
RYSUNEK 11.1. Przesłanie danych do polecenia more za pomocą potoku

Aby było ciekawiej, wraz z potokiem można użyć przekierowania, by zapisać wyniki do pliku:

```
$ rpm -qa | sort > rpm.list
$ more rpm.list
abrt-1.1.14-1.fc14.i686
abrt-addon-ccpp-1.1.14-1.fc14.i686
abrt-addon-kerneloops-1.1.14-1.fc14.i686
abrt-addon-python-1.1.14-1.fc14.i686
abrt-desktop-1.1.14-1.fc14.i686
abrt-gui-1.1.14-1.fc14.i686
abrt-libs-1.1.14-1.fc14.i686
abrt-plugin-bugzilla-1.1.14-1.fc14.i686
abrt-plugin-logger-1.1.14-1.fc14.i686
abrt-plugin-runapp-1.1.14-1.fc14.i686
acl-2.2.49-8.fc14.i686
...
```

Zgodnie z oczekiwaniami teraz dane w pliku *rpm.list* są posortowane!

Zdecydowanie najpopularniejszym sposobem wykorzystania potoków jest przesyłanie wyników poleceń generujących duże ilości danych do polecenia `more`. W szczególności dotyczy to polecenia `ls`, jak pokazano na rysunku 11.2.



RYSUNEK 11.2. Użycie polecenia more w połączeniu z poleceniem ls

Polecenie `ls -l` generuje długą listę wszystkich plików znajdujących się w katalogu. W przypadku katalogów zawierających dużo plików ta lista naprawdę może być bardzo długa. Przez przekazanie wyników do polecenia `more` wymusiliśmy zatrzymywanie prezentacji danych po każdym ekranie.

Wykonywanie działań matematycznych

Kolejną niezbędną cechą każdego języka programowania jest możliwość wykonywania działań na liczbach. Niestety w skryptach powłoki wykonywanie działań matematycznych jest dość toporne i może być realizowane na dwa sposoby.

Polecenie expr

Pierwotnie powłoka Bourne zawierała specjalne polecenie służące do obliczania równań matematycznych. Miało ono nazwę `expr` i umożliwiało wykonywanie obliczeń w wierszu poleceń, ale nie sprawiało dobrego wrażenia:

```
$ expr 1 + 5
6
```

Polecenie `expr` rozpoznaje kilka różnych operatorów matematycznych i łańcuchowych, które wymieniono w tabeli 11.1.

TABELA 11.1. Operatory polecenia expr

Operator	Opis
ARG1 | *ARG2*	Zwraca *ARG1*, jeśli żaden argument nie jest `null` ani 0. W przeciwnym razie zwraca *ARG2*
ARG1 & *ARG2*	Zwraca *ARG1*, jeśli żaden z argumentów nie jest `null` ani 0. W przeciwnym razie zwraca 0
ARG1 < *ARG2*	Zwraca 1, jeśli *ARG1* jest mniejszy od *ARG2*. W przeciwnym razie zwraca 0
ARG1 <= *ARG2*	Zwraca 1, jeśli *ARG1* jest nie większy od *ARG2*. W przeciwnym razie zwraca 0
ARG1 = *ARG2*	Zwraca 1, jeśli *ARG1* jest równy *ARG2*. W przeciwnym razie zwraca 0
ARG1 != *ARG2*	Zwraca 1, jeśli *ARG1* nie jest równy *ARG2*. W przeciwnym razie zwraca 0
ARG1 >= *ARG2*	Zwraca 1, jeśli *ARG1* jest nie mniejszy niż *ARG2*. W przeciwnym razie zwraca 0
ARG1 > *ARG2*	Zwraca 1, jeśli *ARG1* jest większy od *ARG2*. W przeciwnym razie zwraca 0
ARG1 + *ARG2*	Zwraca sumę arytmetyczną *ARG1* i *ARG2*
ARG1 - *ARG2*	Zwraca różnicę arytmetyczną *ARG1* i *ARG2*
ARG1 * *ARG2*	Zwraca iloczyn arytmetyczny *ARG1* i *ARG2*
ARG1 / *ARG2*	Zwraca wynik dzielenia arytmetycznego *ARG1* przez *ARG2*
ARG1 % *ARG2*	Zwraca resztę z dzielenia arytmetycznego *ARG1* przez *ARG2*
ŁAŃCUCH : *REGEXP*	Zwraca dopasowanie wzorca, jeśli *REGEXP* pasuje do wzorca w ciągu *ŁAŃCUCH*
`match` *ŁAŃCUCH REGEXP*	Zwraca dopasowanie wzorca, jeśli *REGEXP* pasuje do wzorca w ciągu *ŁAŃCUCH*
`substr` *ŁAŃCUCH POZ DŁUGOŚĆ*	Zwraca podłańcuch ciągu *ŁAŃCUCH* o długości *DŁUGOŚĆ*, zaczynający się na pozycji *POZ* (zaczyna liczenie od 1)
`index` *ŁAŃCUCH ZNAKI*	Zwraca pozycję w ciągu znaków *ŁAŃCUCH*, na której znaleziono *ZNAKI*. W przeciwnym razie zwraca 0
`length` *ŁAŃCUCH*	Zwraca liczbę określającą długość ciągu *ŁAŃCUCH*
+ *TOKEN*	Interpretuje *TOKEN* jako łańcuch nawet, jeśli jest to słowo kluczowe
(*WYRAŻENIE*)	Zwraca wartość wyrażenia *WYRAŻENIE*

Standardowe operatory dobrze działają w poleceniu `expr`, ale sprawiają problemy w skryptach i wierszu poleceń. Wiele z nich (np. gwiazdka) ma inne znaczenie w powłoce. Ich obecność w poleceniu `expr` sprawia, że otrzymujemy dziwne wyniki:

```
$ expr 5 * 2
expr: syntax error
$
```

Aby rozwiązać ten problem, należy za pomocą znaku modyfikacji (ukośnika wstecznego) oznaczyć wszystkie znaki, które mogłyby zostać źle zinterpretowane przez powłokę:

```
$ expr 5 \* 2
10
$
```

Zaczyna się robić nieelegancko! Posługiwanie się poleceniem `expr` w skryptach wcale nie wygląda lepiej:

```
$ cat test6
#!/bin/bash
# przykład użycia polecenia expr
var1=10
var2=20
```

```
var3=$(expr $var2 / $var1)
echo Wynik wynosi $var3
```

Aby przypisać wynik równania matematycznego do zmiennej, należy użyć techniki podstawiania poleceń, która pozwala pobrać wynik wygenerowany przez wyrażenie expr:

```
$ chmod u+x test6
$ ./test6
Wynik wynosi 2
$
```

Na szczęście powłoka Bash ma pewne ulepszenie, ułatwiające przetwarzanie operatorów matematycznych, które opisujemy poniżej.

Nawiasy

Powłoka Bash ma polecenie expr, by zachować zgodność z powłoką Bourne, ale oprócz tego umożliwia obliczanie wartości równań matematycznych w znacznie łatwiejszy sposób. W powłoce Bash, kiedy przypisujemy wartość matematyczną zmiennej, równanie możemy umieścić w kwadratowym nawiasie poprzedzonym znakiem dolara ($[*operacja*]):

```
$ var1=$[1 + 5]
$ echo $var1
6
$ var2=$[$var1 * 2]
$ echo $var2
12
$
```

Nawiasy znacznie ułatwiają wykonywanie działań matematycznych w porównaniu z poleceniem expr. Ta technika działa także w skryptach powłoki:

```
$ cat test7
#!/bin/bash
var1=100
var2=50
var3=45
var4=$[$var1 * ($var2 - $var3)]
echo Ostateczny wynik to $var4
$
```

Ten skrypt zwróci następujący wynik:

```
$ chmod u+x test7
$ ./test7
Ostateczny wynik to 500
$
```

Ponadto zwróć uwagę, że w metodzie obliczania wartości równań z użyciem prostokątnych nawiasów nie istnieje problem dotyczący znaku mnożenia ani błędnej interpretacji jakiegokolwiek innego znaku. Powłoka wie, że to nie jest symbol wieloznaczny, ponieważ znajduje się on w prostokątnym nawiasie.

Wykonywanie działań matematycznych w skryptach powłoki Bash ma jedno ważne ograniczenie. Spójrz na poniższy przykład:

```
$ cat test8
#!/bin/bash
var1=100
```

```
var2=45
var3=$[$var1 / $var2]
echo Ostateczny wynik to $var3
$
```

Teraz wykonaj ten skrypt i zobacz, co się stanie:

```
$ chmod u+x test8
$ ./test8
Ostateczny wynik to 2
$
```

Operatory matematyczne powłoki Bash obsługują tylko arytmetykę całkowitoliczbową. To ogromne ograniczenie, jeśli zamierzasz wykonywać jakiekolwiek rzeczywiste obliczenia matematyczne.

Powłoka z (zsh) obsługuje pełną arytmetykę zmiennoprzecinkową. Jeśli chcesz wykonywać obliczenia zmiennoprzecinkowe w swoich skryptach powłoki, możesz zainteresować się tą powłoką (jej opis znajduje się w rozdziale 23.).

Rozwiązanie zmiennoprzecinkowe

Ograniczenie powłoki Bash do wykonywania tylko działań arytmetyki całkowitoliczbowej można zlikwidować na kilka sposobów. Najpopularniejszy polega na użyciu wbudowanego kalkulatora Bash o nazwie `bc`.

Podstawy kalkulatora bc

Kalkulator Bash to język programowania, który umożliwia wpisywanie wyrażeń zmiennoprzecinkowych w wierszu poleceń, aby je następnie zinterpretować, obliczyć i zwrócić wynik. Rozpoznaje następujące elementy:

- liczby (zarówno całkowite, jak i zmiennoprzecinkowe);
- zmienne (zarówno proste, jak i tablice);
- komentarze (wiersze zaczynające się od krzyżyka i komentarze w stylu języka C — /* */);
- wyrażenia;
- instrukcje programistyczne (takie jak instrukcje `if-then`);
- funkcje.

Dostęp do kalkulatora Bash można uzyskać przez wpisanie polecenia `bc` w wierszu poleceń:

```
$ bc
bc 1.06.95
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006 Free Software
Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type 'warranty'.
12 * 5.4
64.8
3.156 * (3 + 5)
25.248
quit
$
```

Przykład zaczyna się od wyrażenia `12 * 5.4`. Kalkulator Bash zwraca wynik. Każde kolejne polecenie wpisane do kalkulatora zostaje obliczone i następuje wyświetlenie wyniku. Aby wyjść z kalkulatora Bash, należy wpisać `quit`.

Arytmetyką zmiennoprzecinkową steruje wbudowana zmienna o nazwie `scale`. Jej wartość należy ustawić na liczbę miejsc dziesiętnych, jaką chcemy otrzymać w wyniku. Jeśli tego nie zrobimy, możemy otrzymać inny wynik, niż się spodziewaliśmy:

```
$ bc -q
3.44 / 5
0
scale=4
3.44 / 5
.6880
quit
$
```

Domyślną wartością zmiennej `scale` jest 0, a więc jeśli jej nie zmienimy, kalkulator będzie zwracał wyniki z zerem miejsc dziesiętnych. Po ustawieniu jej wartości na 4 kalkulator zwraca wyniki z dokładnością do czterech miejsc po przecinku. Parametr wiersza poleceń `-q` wyłącza długi tekst powitalny kalkulatora Bash.

Kalkulator Bash oprócz zwykłych liczb rozpoznaje także zmienne:

```
$ bc -q
var1=10
var1 * 4
40
var2 = var1 / 5
print var2
2
quit
$
```

Kiedy zmienna jest zdefiniowana, można jej używać wszędzie w obrębie sesji kalkulatora Bash. Instrukcja `print` umożliwia drukowanie zmiennych i liczb.

Zastosowanie kalkulatora bc w skryptach

Teraz pewnie się zastanawiasz, w jaki sposób kalkulator Bash może Ci się przydać przy wykonywaniu zmiennoprzecinkowych działań arytmetycznych w skryptach powłoki. Pamiętasz swojego dobrego przyjaciela odwrotny apostrof? Tak, za pomocą znaku podstawiania poleceń można wykonywać polecenia `bc`, a ich wyniki można zapisywać w zmiennej! Podstawowy format tej operacji wygląda tak:

```
zmienna=$(echo "opcje; wyrażenie" | bc)
```

Pierwsza część, *`opcje`*, umożliwia ustawienie wartości zmiennych. Jeśli ustawiasz wartości więcej niż jednej zmiennej, rozdziel je średnikami. Parametr *`wyrażenie`* reprezentuje wyrażenie matematyczne, które ma zostać obliczone przez `bc`. Oto krótki przykład zastosowania tej techniki w skrypcie:

```
$ cat test9
#!/bin/bash
var1=$(echo " scale=4; 3.44 / 5" | bc)
echo Odpowiedź to $var1
$
```

W tym przykładzie ustawiliśmy zmienną `scale` na cztery miejsca dziesiętne, a następnie podaliśmy konkretne wyrażenie do obliczenia. Ten skrypt zwróci następujący wynik:

```
$ chmod u+x test9
$ ./test9
Odpowiedź to .6880
$
```

Świetnie! W wyrażeniach możemy używać nie tylko liczb, ale również zmiennych zdefiniowanych w skrypcie powłoki:

```
$ cat test10
#!/bin/bash
var1=100
var2=45
var3=$(echo "scale=4; $var1 / $var2" | bc)
echo Odpowiedź w tym przypadku to $var3
$
```

Ten skrypt definiuje dwie zmienne, które zostały użyte w wyrażeniu wysłanym do polecenia `bc`. Pamiętaj, aby za pomocą znaku dolara zaznaczać, że masz na myśli wartości zmiennych, a nie same zmienne. Wynik tego skryptu jest następujący:

```
$ ./test10
Odpowiedź w tym przypadku to 2.2222
$
```

Oczywiście kiedy w zmiennej zostanie zapisana wartość, można jej użyć w kolejnym działaniu:

```
$ cat test11
#!/bin/bash
var1=20
var2=3.14159
var3=$(echo "scale=4; $var1 * $var1" | bc)
var4=$(echo "scale=4; $var3 * $var2" | bc)
echo Ostateczny wynik to $var4
$
```

Ta metoda dobrze się sprawdza w przypadku wykonywania krótkich obliczeń, ale czasami działania są bardziej skomplikowane. Jeśli chcemy wykonać więcej niż tylko parę operacji, to lista wielu wyrażeń w jednym wierszu może być mało przejrzysta.

Ten problem można rozwiązać. Polecenie `bc` rozpoznaje przekierowywanie wejścia, dzięki czemu można do niego skierować zawartość pliku do przetworzenia. Jednak to też nie jest idealne rozwiązanie, ponieważ wszystkie wyrażenia trzeba zapisać w pliku.

Najlepszą metodą jest użycie śródliniowego przekierowania wejścia, które umożliwia przekierowanie danych bezpośrednio z wiersza poleceń. W skrypcie powłoki wynik przypisujemy do zmiennej:

```
zmienna=$(bc << EOF
opcje
instrukcje
wyrażenia
EOF
)
```

Łańcuch `EOF` oznacza początek i koniec śródliniowego przekierowania danych. Pamiętaj, że do przypisania wyniku polecenia `bc` do zmiennej potrzebne są znaki podmieniania polecenia.

Teraz wszystkie poszczególne elementy kalkulatora Bash można umieścić w osobnych wierszach w pliku skryptu. Oto przykład zastosowania tej techniki w skrypcie:

```
$ cat test12
#!/bin/bash

var1=10.46
var2=43.67
var3=33.2
var4=71

var5=$(bc << EOF
scale = 4
a1 = ( $var1 * $var2)
b1 = ($var3 * $var4)
a1 + b1
EOF
)

echo Ostateczna odpowiedź w tym całym zamieszaniu to $var5
$
```

Umieszczenie każdej opcji i każdego wyrażenia w osobnym wierszu w skrypcie sprawia, że tekst jest znacznie bardziej czytelny. Łańcuch `EOF` oznacza początek i koniec danych, które mają zostać przekierowane do polecenia `bc`. Oczywiście należy użyć znaków podmieniania poleceń, aby wskazać polecenie, które ma zostać przypisane do zmiennej.

Ponadto w tym przykładzie zauważ, że w kalkulatorze Bash można przypisywać zmienne. Należy pamiętać, że wszystkie zmienne utworzone w tym kalkulatorze są dostępne tylko w nim i nie można ich używać w skrypcie powłoki.

Wychodzenie ze skryptu

W naszych dotychczasowych przykładowych skryptach zakończenie pracy następowało bardzo raptownie. Po zakończeniu wykonywania ostatniego polecenia po prostu kończyliśmy skrypt. Można to jednak zrobić znacznie bardziej elegancko.

Każde polecenie, które jest wykonywane w powłoce, po zakończeniu pracy zwraca do powłoki **kod stanu zakończenia**. Jest to liczba z przedziału od `0` do `255`, którą polecenie przekazuje powłoce, gdy skończy działanie. Wartość tę można przechwycić i wykorzystać w swoich skryptach.

Sprawdzanie stanu wyjścia

W systemie Linux można używać specjalnej zmiennej `$?`, która przechowuje stan wyjścia ostatnio wykonanego polecenia. Należy jej użyć bezpośrednio po poleceniu, które nas interesuje, ponieważ jej wartość zmienia się za każdym razem, gdy w powłoce zostaje wykonane jakieś polecenie:

```
$ date
pon 01 cze 16:01:30 EDT 2020
$ echo $?
0
$
```

Standardowo kod stanu wyjścia polecenia, które zostało pomyślnie wykonane, to 0. Jeśli wykonywanie polecenia zakończy się błędem, to jego stan wyjścia jest oznaczony całkowitą liczbą dodatnią:

```
$ asdfg
-bash: asdfg: command not found
$ echo $?
127
$
```

Nieprawidłowe polecenie zwraca kod stanu wyjścia 127. Kody stanu wyjścia z błędem w Linuksie nie są ustandaryzowane, ale można się kierować paroma wytycznymi, które przedstawiono w tabeli 11.2.

TABELA 11.2. Kody stanu wyjścia Linuksa

Kod	Opis
0	Polecenie zostało wykonane pomyślnie
1	Ogólny nieznany błąd
2	Nieprawidłowe użycie polecenia powłoki
126	Nie można wykonać polecenia
127	Nie znaleziono polecenia
128	Nieprawidłowy argument wyjścia
128+x	Błąd krytyczny z sygnałem Linuksa **x**
130	Polecenie zakończono przez naciśnięcie klawiszy *Ctrl+C*
255	Stan wyjścia poza zakresem

Kod stanu wyjścia 126 oznacza, że użytkownik nie miał odpowiednich uprawnień do wykonania polecenia:

```
$ ./myprog.c
-bash: ./myprog.c: Permission denied
$ echo $?
126
$
```

Innym często spotykanym błędem jest przekazanie do polecenia nieprawidłowego parametru:

```
$ date %t
date: invalid date '%t'
$ echo $?
1
$
```

Powoduje to wygenerowanie ogólnego stanu kodu wyjścia 1, który oznacza, że w poleceniu wystąpił nieznany błąd.

Polecenie exit

Domyślnie skrypt powłoki zwraca kod wyjścia odpowiadający stanowi ostatniego wykonanego w nim polecenia:

```
$ ./test6
Wynik wynosi 2
$ echo $?
0
$
```

Użytkownik może to zmienić, aby zwracać własny kod stanu wyjścia. Służy do tego polecenie `exit`, umożliwiające określenie stanu wyjścia po zakończeniu wykonywania skryptu:

```
$ cat test13
#!/bin/bash
# test stanu wyjścia
var1=10
var2=30
var3=$[ $var1 + var2 ]
echo Odpowiedź to $var3
exit 5
$
```

Gdy sprawdzisz stan wyjścia skryptu, otrzymasz wartość podaną jako parametr polecenia `exit`:

```
$ chmod u+x test13
$ ./test13
Odpowiedź to 40
$ echo $?
5
$
```

W poleceniu `exit` można także używać zmiennych:

```
$ cat test14
#!/bin/bash
# test stanu wyjścia
var1=10
var2=30
var3=$[ $var1 + var2 ]
exit $var3
$
```

Kiedy wykonasz to polecenie, wygeneruje ono następujący kod stanu wyjścia:

```
$ chmod u+x test14
$ ./test14
$ echo $?
40
$
```

Z tej możliwości należy jednak korzystać ostrożnie, ponieważ maksymalna wartość kodu stanu wyjścia wynosi 255. Spójrz, co się dzieje w tym przykładzie:

```
$ cat test14b
#!/bin/bash
# test stanu wyjścia
var1=10
var2=30
var3=$[ $var1 * var2 ]
echo Wartość wynosi $var3
exit $var3
$
```

Wynik wykonania tego skryptu będzie następujący:

```
$ ./test14b
Wartość wynosi 300
$ echo $?
44
$
```

Kod stanu wyjścia został zredukowany do wartości mieszczącej się w przedziale od 0 do 255. W tym celu powłoka stosuje arytmetykę modulo, czyli pobiera resztę z dzielenia. Otrzymana w wyniku liczba jest resztą z dzielenia podanej liczby przez 246. W tym przypadku podzielono 300 (wynik) przez 246, co dało resztę 44 i ta wartość została zwrócona jako kod stanu wyjścia.

W następnym rozdziale pokazujemy, jak za pomocą instrukcji `if-then` sprawdzać stan błędu zwrócony przez polecenie, aby się dowiedzieć, czy jego wykonanie się powiodło.

Praktyczny przykład

Znasz już podstawy tworzenia skryptów powłoki, więc możemy spróbować napisać jakiś skrypt o praktycznym zastosowaniu. Utworzymy program, który będzie obliczał liczbę dni dzielącą dwie daty. Użytkownik będzie mógł podać datę w dowolnym formacie rozpoznawanym przez linuksowe polecenie `date`. Najpierw zapiszemy obie daty w zmiennych:

```
$date1="01/01/2020"
$date2="01/01/2021"
```

Arytmetyka na datach jest trudna, ponieważ wymaga wiedzy, które miesiące mają 28, 30 i 31 dni i które lata to lata przestępne. Na szczęście samo polecenie `date` może nam trochę pomóc.

Polecenie `date` pozwala na określenie konkretnej daty za pomocą opcji `-d` (w dowolnym formacie), a następnie wyświetlenie jej w dowolnym innym zdefiniowanym przez nas formacie. Aby wykonać nasze obliczenia, skorzystamy z funkcji Linuksa o nazwie **czas epoki**.
Jest to liczba całkowita oznaczająca liczbę sekund, jaka upłynęła od północy 1 stycznia 1970 r. (stary uniksowy standard). Aby sprawdzić czas epoki dla 1 stycznia 2020 r., należy wykonać następujące polecenie:

```
$ date -d "01/01/2020" +%s
1577854800
$
```

Tą metodą sprawdzimy czas epoki dla obu dat, a następnie odejmiemy od siebie te dwie wartości, aby otrzymać liczbę dzielących je sekund. Potem wystarczy podzielić otrzymaną liczbę przez liczbę sekund w dniu (60 sekund w minucie, 60 minut w godzinie i 24 godziny w dobie), aby otrzymać różnicę między dwiema interesującymi nas datami w dniach.

Za pomocą techniki podmieniania poleceń prześlemy wynik polecenia `date` do zmiennej:

```
$time1=$(date -d "$date1" +%s)
```

Po otrzymaniu czasu epoki dla obu dat wystarczy użyć niedawno poznanego polecenia `expr` do obliczenia różnicy (moglibyśmy też użyć narzędzia `bc`, ale do pracy z liczbami całkowitymi wystarczy `expr`).

Po połączeniu wszystkich kawałków układanki otrzymujemy następujący skrypt:

```
$ cat mydate.sh
#!/bin/bash
# obliczanie liczby dni dzielących dwie daty
date1="01/01/2020"
date2="01/01/2021"

time1=$(date -d "$date1" +%s)
time2=$(date -d "$date2" +%s)
```

```
diff=$(expr $time2 - $time1)
secondsinday=$(expr 24 \* 60 \* 60)
days=$(expr $diff / $secondsinday)

echo "Różnica między $date2 i $date1 w dniach wynosi: $days"
$
```

Jeszcze tylko przypiszemy odpowiednie uprawnienia do pliku i możemy uruchomić skrypt:

```
$ chmod u+x mydate.sh
$ ./mydate.sh
Różnica między 01/01/2021 i 01/01/2020 w dniach wynosi: 366
$
```

Teraz w zmiennych możesz zapisać dowolne inne daty (w niemal każdym formacie, jaki Cię interesuje), aby otrzymać prawidłowy wynik.

Podsumowanie

W skryptach powłoki Bash można używać łańcuchów poleceń. Najprostszym sposobem na utworzenie skryptu jest wpisanie kilku poleceń w jednym wierszu i rozdzielenie ich średnikami. Powłoka wykona je po kolei i wyświetli wynik każdego z nich na ekranie monitora.

Skrypty powłoki składające się z wielu poleceń można także zapisywać w plikach. Plik skryptu powłoki musi określać powłokę, w której ma być wykonywany. Określenie to należy umieścić w pierwszym wierszu, zaczynającym się od znaków `#!`, po których następuje pełna ścieżka do powłoki.

W skrypcie powłoki można odwoływać się do wartości zmiennych środowiskowych za pomocą ich nazw poprzedzonych znakiem dolara. Ponadto można definiować własne zmienne do użycia w skrypcie, a także można przypisywać im wartości, a nawet wyniki poleceń, za pomocą znaku odwrotnego apostrofu lub formatu `$()`. Wartości zmiennej można użyć w skrypcie przez postawienie znaku dolara przed jej nazwą.

Powłoka Bash umożliwia przekierowywanie zarówno wejścia, jak i wyjścia polecenia. Użytkownik może przekierować wynik dowolnego polecenia z monitora do pliku za pomocą symbolu większości, po którym powinna się znajdować nazwa pliku przeznaczonego do przechowywania danych. Dane wynikowe można dodać do istniejącego pliku przez użycie dwóch znaków większości. Znak mniejszości z kolei służy do przekierowywania wejścia poleceń. Użytkownik może przekierować dane z pliku do polecenia.

Polecenie potoku Linuksa (symbol dwóch pionowych kresek) umożliwia przekierowanie wyjścia polecenia bezpośrednio na wejście innego polecenia. System Linux wykonuje oba polecenia jednocześnie, wysyłając wyniki pierwszego na wejście drugiego bez użycia jakichkolwiek plików pośrednich.

Powłoka Bash umożliwia wykonywanie działań matematycznych w skryptach na dwa sposoby. Polecenie `expr` pozwala na proste wykonywanie działań całkowitoliczbowych. Ponadto powłoka Bash umożliwia wykonywanie podstawowych obliczeń matematycznych przez umieszczenie równań w nawiasach prostokątnych poprzedzonych znakiem dolara. Aby wykonać obliczenia arytmetyki zmiennoprzecinkowej, należy użyć polecenia `bc`, przekierowując wejście z danych śródliniowych i zapisując wyniki w zmiennej użytkownika.

W ostatniej części rozdziału omówiliśmy kody stanu wyjścia skryptów powłoki. Każde polecenie działające w powłoce zwraca kod stanu wyjścia. Jest to liczba całkowita z przedziału od 0 do 255, która określa, czy dane polecenie zostało wykonane poprawnie, a jeśli nie, to co może być tego przyczyną. Kod stanu wyjścia 0 oznacza pomyślne wykonanie polecenia. Za pomocą polecenia `exit` można zadeklarować kod stanu wyjścia po zakończeniu wykonywania skryptu.

W skryptach, które dotychczas pokazaliśmy, polecenia były wykonywane jedno po drugim w kolejności wpisania. W następnym rozdziale poznasz techniki modyfikacji logiki wykonywania poleceń skryptu.

Skorowidz

F

L

Ł

M

N

O

P

R

S

Z

Wiersz poleceń: oto pełny potencjał Linuksa!

Odkąd systemy linuksowe mają atrakcyjne interfejsy graficzne, wielu administratorów wykonuje swoje zadania za pomocą myszy. Wiersz poleceń jest o wiele trudniejszym interfejsem, ten sposób pracy jednak pozwala na wykorzystanie pełnego potencjału Linuksa i uzyskanie dostępu do funkcji, które w inny sposób byłyby nieosiągalne. Umiejętność pisania skryptów wiersza poleceń i powłoki Linuksa wciąż jest niezwykle ważna dla każdego administratora, któremu zależy na efektywnym działaniu systemu.

Ta książka, podobnie jak inne z serii „Biblia", zawiera zarówno niezbędne teoretyczne informacje, jak i mnóstwo praktycznych wskazówek i instrukcji, dzięki czemu jest znakomitą pomocą w nauce pisania skryptów dla Linuksa. Pokazano tu, kiedy efektywniej jest używać interfejsu graficznego, a kiedy lepiej sięgnąć do wiersza poleceń. Opisano elementy systemu Linux i przedstawiono podstawowe informacje o powłoce, a także o pracy w wierszu poleceń. Zaprezentowano techniki przetwarzania danych użytkownika w skryptach, pracę z systemami plików, wreszcie — sposoby instalacji i aktualizacji oprogramowania. Sporo miejsca poświęcono pracy z wyrażeniami regularnymi i zaawansowanym metodom manipulacji danymi, pokazano też, jak zmodyfikować skrypty powłoki, aby działały w innych powłokach Linuksa.

Najciekawsze zagadnienia:
- tworzenie praktycznych skryptów i narzędzi skryptowych i zarządzanie nimi
- alternatywne powłoki, takie jak dash i zsh
- edytory gawk i sed
- podstawowe i rozszerzone wyrażenia regularne
- tworzenie skryptów powłoki dla pulpitów graficznych w środowiskach KDE i GNOME
- pisanie podstawowych i zaawansowanych funkcji

Richard Blum od ponad 30 lat administruje systemami komputerowymi i angażuje się w różnego rodzaju przedsięwzięcia non profit. Jest autorem popularnych książek na temat Linuksa. Lubi grać na fortepianie i gitarze basowej.

Christine Bresnahan jest adiunktką w Ivy Tech Community College, gdzie prowadzi zajęcia certyfikacyjne z Linuksa i uczy programowania w języku Python. Jest też autorką książek i materiałów dydaktycznych. Jej pasja to ogród i piesze wycieczki.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-8322-074-1
9 788383 220741
Cena: 149,00 zł
WILEY