Clif Flynt, Sarath Lakshman,
Shantanu Tushar

Skrypty powłoki systemu Linux

Receptury

Wydanie III

Helion

Packt›

Tytuł oryginału: Linux Shell Scripting Cookbook – Third Edition

Tłumaczenie: Joanna Zatorska

ISBN: 978-83-283-4031-2

First published in the English language under the title
„Linux Shell Scripting Cookbook - Third Edition – (9781785881985)"

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/sposy3.zip

Printed in Poland.

Spis treści

ROZDZIAŁ 1

Poznanie możliwości powłoki

Ten rozdział zawiera następujące podrozdziały:

- Wyświetlanie w oknie terminalu
- Używanie zmiennych i zmiennych środowiskowych
- Funkcja dołączająca wartość na początku zmiennych środowiskowych
- Wykonywanie obliczeń matematycznych za pomocą powłoki
- Eksperymentowanie z deskryptorami plików i przekierowywaniem
- Tablice zwykłe i tablice asocjacyjne
- Korzystanie z aliasów
- Uzyskiwanie informacji o terminalu
- Uzyskiwanie i ustawianie dat oraz opóźnienia
- Debugowanie skryptu
- Funkcje i argumenty
- Przekazywanie danych wyjściowych do drugiego polecenia
- Odczytywanie n znaków bez naciskania klawisza Enter
- Wykonywanie polecenia aż do osiągnięcia zamierzonego celu
- Separatory pól i iteratory
- Porównania i testy
- Dostosowywanie powłoki za pomocą plików konfiguracyjnych

Wprowadzenie

Pierwsze komputery wczytywały programy zapisane na kartach lub taśmach i generowały jeden raport. Nie istniały systemy operacyjne, monitory graficzne ani nawet interaktywne komunikaty.

W latach 60. XX wieku komputery obsługiwały już interaktywne terminale (zwykle dalekopisy), które służyły do wywoływania poleceń.

Opracowany w laboratorium Bell Labs interaktywny interfejs użytkownika dla zupełnie nowego systemu operacyjnego Unix charakteryzował się unikatową funkcją. Można było na nim wczytywać i wykonywać polecenia zapisane w pliku tekstowym (zwanym skryptem powłoki), a także polecenia wpisane w terminalu.

Ta nowość znacznie przyczyniła się do zwiększenia produktywności. Zamiast wpisywać kilka poleceń w celu wykonania zestawu operacji, programiści uzyskali możliwość zapisania poleceń w pliku i wykonania ich później za pomocą jedynie kilku klawiszy. Skrypty powłoki nie tylko pozwalają zaoszczędzić czas, ale także pozwalają dokumentować wykonywane zadania.

Początkowo Unix wspierał jedną powłokę interaktywną, opracowaną przez Stephena Bourne'a, zwaną powłoką Bourne Shell (sh).

W 1989 roku Brian Fox, pracujący nad projektem GNU, wyselekcjonował funkcje z wielu różnych interfejsów użytkownika i opracował nową powłokę ***Bourne Again Shell*** (**Bash**). Powłoka Bash rozpoznaje wszystkie konstrukcje powłoki Bourne'a, a także zawiera dodatkowe funkcje z powłok csh, ksh i innych.

Ponieważ Linux stał się najpopularniejszą implementacją systemów uniksowych, powłoka Bash stała się w rzeczywistości standardową powłoką w systemach Unix i Linux.

Ta książka koncentruje się na systemie Linux i powłoce Bash. Jednak większość omówionych w niej skryptów można uruchomić zarówno w systemie Linux, jak i Unix, korzystając z powłok Bash, sh, ash, dash, ksh lub innych powłok podobnych do powłoki sh.

Głównym celem tego rozdziału jest zaprezentowanie czytelnikom przeglądu środowiska powłoki i umożliwienie im zapoznania się z podstawowymi funkcjami związanymi z powłoką.

Wyświetlanie w oknie terminalu

Użytkownicy korzystają ze środowiska powłoki za pomocą sesji terminalu. W przypadku systemu opartego na graficznym interfejsie użytkownika jest to okno terminalu. Jeśli system (na przykład system produkcyjny lub sesja ssh) nie posiada interfejsu graficznego, wiersz poleceń będzie dostępny od razu po zalogowaniu się.

Wyświetlanie tekstu w terminalu jest zadaniem, które regularnie wykonuje większość skryptów i narzędzi. Powłoka umożliwia wyświetlanie tekstu za pomocą różnych metod i w różnych formatach.

Wprowadzenie

Polecenia są wpisywane i wykonywane w terminalu powłoki. Po otwarciu okna terminalu dostępny jest wiersz poleceń. Można go skonfigurować na wiele różnych sposobów, ale zwykle ma on następujący format:

```
nazwa_użytkownika@nazwa_hosta$
```

Można go też skonfigurować następująco: `root@nazwa_hosta#` lub w postaci znaku `$` albo `#`.

Znak `$` reprezentuje zwykłych użytkowników, a znak `#` — administratora `root`. W systemie Linux użytkownik `root` ma największe przywileje.

> Odradza się korzystanie z powłoki jako użytkownik `root` (administrator). Jeśli powłoka ma duże przywileje, zwykłe literówki mogą się okazać znacznie większym zagrożeniem. Zawsze należy się logować jako zwykły użytkownik (można to rozpoznać w powłoce po znaku `$` na początku wiersza poleceń), a polecenia wymagające większych uprawnień należy uruchamiać za pomocą narzędzi takich jak `sudo`. Polecenie uruchomione zgodnie ze wzorcem `sudo <polecenie> <argumenty>` zostanie uruchomione z uprawnieniami użytkownika `root`.

Skrypt powłoki zazwyczaj zaczyna się magicznym ciągiem `#!` w następujący sposób:

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

Wiersz magicznego ciągu jest wierszem, w którym ścieżkę interpretera poprzedza ciąg `#!`. W przypadku powłoki Bash ścieżka polecenia interpretera ma postać */bin/bash*. Wiersz rozpoczynający się symbolem `#` jest traktowany przez interpreter powłoki jako komentarz. Magiczny ciąg może się znajdować tylko w pierwszym wierszu skryptu. W ten sposób określamy interpreter służący do odczytania skryptu.

Skrypt może być wykonany na dwa następujące sposoby.

1. Przy użyciu nazwy pliku jako argumentu wiersza polecenia:

   ```
   bash mojSkrypt.sh
   ```

2. Poprzez ustawienie uprawnień wykonywania dla pliku skryptu, aby umożliwić jego uruchomienie:

   ```
   chmod 755 mojSkrypt.hs
   ./mojSkrypt.sh
   ```

Jeśli skrypt jest uruchamiany jako argument wiersza polecenia powłoki `bash`, użycie magicznego ciągu `#!` w skrypcie jest zbędne. Magiczny ciąg ułatwia uruchamianie samodzielnego skryptu, ponieważ znajdująca się w tym samym wierszu ścieżka definiuje interpreter skryptu.

Uprawnienie wykonywania skryptu może być ustawione następująco:

```
$ chmod a+x skrypt.sh
```

Powyższe polecenie nadaje plikowi skryptu uprawnienie wykonywania dla wszystkich użytkowników. Skrypt może być wykonany za pomocą polecenia:

```
$ ./skrypt.sh # ./ Reprezentuje bieżący katalog.
```

Skrypt można też uruchomić następująco:

```
$ /home/path/skrypt.sh # Użyto pełnej ścieżki do skryptu.
```

Jądro systemu wczyta pierwszy wiersz i sprawdzi, czy zawiera on magiczny ciąg `#!/bin/bash`. W dalszej kolejności zidentyfikuje ścieżkę */bin/bash* i wykona skrypt w następujący sposób:

```
$ /bin/bash skrypt.sh
```

Po uruchomieniu interaktywnej powłoki najpierw jest w niej wykonywany zestaw poleceń definiujących różne ustawienia, m.in. opcje wiersza poleceń i kolory. Ten zestaw poleceń jest wczytywany ze skryptu powłoki *~/.bashrc* (lub *~/.bash_profile* w przypadku powłok logowania), który znajduje się w katalogu domowym użytkownika. Powłoka Bash utrzymuje również historię poleceń uruchomionych przez użytkownika. Historia ta jest dostępna w pliku *~/.bash_history*.

> Znak ~ stanowi skrót ścieżki katalogu domowego użytkownika. Zwykle jest to katalog `/home/użytkownik`, gdzie element `użytkownik` odpowiada nazwie użytkownika, lub `/root` w przypadku użytkownika `root`. Powłoka logowania jest uruchamiana podczas logowania się na komputerze. Jednak sesje terminalu uruchamiane podczas logowania się w środowisku graficznym (na przykład GNOME lub KDE) nie są powłokami logowania. Podczas logowania się za pomocą menedżera graficznego, takiego jak GDM lub KDM, pliki `.profile` lub `.bash_profile` mogą zostać zignorowane (zwykle są). Natomiast podczas logowania się w systemie zdalnym przy użyciu `ssh` zostanie wczytany plik `.profile`. W powłoce każde polecenie w sekwencji poleceń jest oddzielone za pomocą średnika lub nowego wiersza. Oto przykład:
> `$ polecenie1 ; polecenie2`
>
> Odpowiada to następującym wierszom:
>
> ```
> $ polecenie1
> $ polecenie2
> ```

Komentarz rozpoczyna się znakiem # i jest kontynuowany do końca wiersza. Wiersze komentarzy najczęściej są używane do zamieszczenia opisu kodu zawartego w pliku lub do wyłączenia wiersza kodu z wykonywania podczas debugowania:

```
# skrypt.sh — wyświetla "Witaj, świecie"
echo "Witaj, świecie"
```

Zajmijmy się teraz podstawowymi recepturami przedstawionymi w tym rozdziale.

Jak to zrobić

Podstawowym poleceniem służącym do wyświetlania w oknie terminalu jest polecenie `echo`.

Domyślnie wstawia ono nowy wiersz na końcu każdego wywołania:

```
$ echo "Witaj w programie Bash"
Witaj w programie Bash
```

Zwykłe użycie tekstu w cudzysłowie w wierszu polecenia `echo` powoduje wyświetlenie tekstu w oknie terminalu. Tekst bez cudzysłowu również zapewnia te same dane wyjściowe:

```
$ echo Witaj w programie Bash
Witaj w programie Bash
```

Inny sposób realizacji tego zadania polega na użyciu apostrofów:

```
$ echo 'Tekst w apostrofach'
```

Choć te metody mogą wyglądać podobnie, każda z nich ma konkretne zastosowanie, a także efekty uboczne. Cudzysłowy umożliwiają interpretację znaków specjalnych znajdujących się w tekście. Apostrofy wyłączają ich interpretację.

Rozważ następujące polecenie:

```
$ echo "Nie można uwzględnić wykrzyknika (!) w cudzysłowie."
```

Spowoduje ono zwrócenie następującego wiersza:

```
bash: !: event not found error
```

Jeśli więc zamierzasz wyświetlić znak specjalny, na przykład wykrzyknik, nie używaj go wewnątrz cudzysłowu, ale skorzystaj z apostrofów. Możesz też poprzedzić wykrzyknik specjalnym znakiem zmiany znaczenia \:

```
$ echo Witaj, świecie!
```

Inny sposób jest taki:

```
$ echo 'Witaj, świecie!'
```

lub:

```
$ echo "Witaj, świecie\!" # Wykrzyknik poprzedzono znakiem \.
```

W przypadku korzystania z polecenia `echo` bez cudzysłowu nie możesz użyć średnika, ponieważ w powłoce Bash pełni on funkcję separatora poleceń. Na przykład:

```
echo witaj;witaj
```

W przypadku powyższego wiersza Bash traktuje ciąg `echo witaj` jako pierwsze polecenie, a kolejne wystąpienie łańcucha `witaj` jako drugie polecenie.

Jeśli użyjemy apostrofów, rozwijanie zmiennych, omówione w kolejnej recepturze, nie będzie działać.

Innym poleceniem służącym do wyświetlania w oknie terminalu jest `printf`. Polecenie to używa tych samych argumentów co polecenie `printf` języka programowania C. Oto przykład:

```
$ printf "Witaj, świecie"
```

Polecenie `printf` pobiera tekst w cudzysłowie lub argumenty oddzielone spacjami. Umożliwia ono zastosowanie sformatowanych łańcuchów. Możesz określić szerokość łańcucha, lewe lub prawe wyrównanie itp. Domyślnie polecenie `printf` nie zapewnia nowego wiersza. W razie potrzeby konieczne jest określenie nowego wiersza w sposób zaprezentowany w następującym skrypcie:

```
#!/bin/bash
#nazwa pliku: printf.sh
printf  "%-5s %-10s %-4s\n" Numer Imię  Wynik
printf  "%-5s %-10s %-4.2f\n" 1 Stefan 80.3456
printf  "%-5s %-10s %-4.2f\n" 2 Janusz 90.9989
printf  "%-5s %-10s %-4.2f\n" 3 Józef 77.564
```

Uzyskamy następujące sformatowane dane wyjściowe:

```
Numer  Imię     Wynik
1      Stefan   80.35
2      Janusz   91.00
3      Józef    77.56
```

Jak to działa

`%s`, `%c`, `%d` i `%f` to znaki zastępcze formatu, określające sposób wyświetlania kolejnych argumentów. Ciąg `%-5s` można opisać jako podstawienie łańcucha z lewym wyrównaniem (reprezentowane przez znak `-`) o szerokości `5`. Jeśli nie zostałby podany znak `-`, łańcuch byłby wyrównany do prawej strony. Szerokość określa liczbę znaków zarezerwowanych dla tej zmiennej. W przypadku łańcucha `Imię` zarezerwowana szerokość wynosi `10`. A zatem dowolne imię będzie uwzględniać zarezerwowaną dla niego szerokość, wynoszącą 10 znaków, a reszta znaków brakujących do dziesięciu zostanie uzupełniona spacjami.

W przypadku liczb zmiennoprzecinkowych możliwe jest przekazanie dodatkowych parametrów w celu zaokrąglenia liczb po przecinku.

Na potrzeby wyników sformatowano łańcuch w postaci `%-4.2f`, w którym ciąg `.2` powoduje zaokrąglenie do dwóch miejsc dziesiętnych. Zauważ, że dla każdego wiersza łańcucha formatu użyto symbolu nowego wiersza (`\n`).

To nie wszystko

Zawsze należy zwracać uwagę na to, że flagi poleceń `echo` i `printf` powinny pojawić się przed wszelkimi łańcuchami w poleceniu. W przeciwnym razie program Bash potraktuje flagi jako kolejne łańcuchy.

Eliminowanie nowego wiersza w wyniku polecenia echo

Domyślnie polecenie `echo` na końcu swoich danych wyjściowych umieszcza nowy wiersz. Flaga `-n` umożliwia uniknięcie tego. Polecenie `echo` może też akceptować jako argument sekwencje o zmienionym znaczeniu w łańcuchach umieszczonych w cudzysłowie. W celu zastosowania takich sekwencji użyj polecenia `echo` w formacie `echo -e "łańcuch zawierający sekwencje o zmienionym znaczeniu"`. Oto przykład:

```
echo -e "1\t2\t3"
123
```

Wyświetlanie kolorowych danych wyjściowych

W celu wyświetlenia kolorowych danych wyjściowych w oknie terminalu można użyć sekwencji o zmienionym znaczeniu.

Do reprezentowania każdego koloru używa się odpowiednich kodów. Na przykład: zresetowanie = `0`, czarny = `30`, czerwony = `31`, zielony = `32`, żółty = `33`, niebieski = `34`, purpurowy = `35`, niebieskozielony = `36` i biały = `37`.

Aby wyświetlić kolorowy tekst, wprowadź następujące polecenie:

```
echo -e "\e[1;31m To jest czerwony tekst \e[0m"
```

W poleceniu tym ciąg `\e[1;31` jest łańcuchem o zmienionym znaczeniu, ustawiającym kolor czerwony. Z kolei ciąg `\e[0m` ponownie ustawia kolor czarny. Jeśli chcesz wyświetlić inny kolor, w miejsce wartości `31` wstaw wymagany kod koloru.

W przypadku kolorowego tła powszechnie używa się następujących kodów kolorów: zresetowanie = `0`, czarny = `40`, czerwony = `41`, zielony = `42`, żółty = `43`, niebieski = `44`, purpurowy = `45`, niebieskozielony = `46` i biały = `47`.

Aby wyświetlić kolorowe tło, wprowadź następujące polecenie:

```
echo -e "\e[1;42m Zielone tło \e[0m"
```

Powyższe przykłady obejmują tylko część sekwencji o zmienionym znaczeniu. Więcej można znaleźć w dokumentacji dostępnej za pomocą polecenia `man console_codes`.

Używanie zmiennych i zmiennych środowiskowych

We wszystkich językach programowania zmienne służą do przechowywania danych w celu późniejszego użycia lub modyfikacji. W przeciwieństwie do języków kompilowanych języki skryptowe zwykle nie wymagają zadeklarowania typów zmiennych przed możliwością ich użycia. Typ jest wyznaczany na podstawie sposobu użycia. Wartość zmiennej możemy odczytać, poprzedzając jej nazwę symbolem dolara. W powłoce istnieją specjalne zmienne, które przechowują konfigurację i informacje dotyczące dostępnych drukarek, ścieżek wyszukiwania itd. Są to zmienne środowiskowe.

Wprowadzenie

Nazwy zmiennych określa się przy użyciu sekwencji liter, cyfr, podkreślników, z wyjątkiem białych znaków. Zwykle przestrzega się konwencji nazywania zmiennych środowiskowych za pomocą WIELKICH_LITER, natomiast w przypadku zmiennych używanych w skrypcie stosuje się notację camelCase lub małe_litery.

Zmienne środowiskowe są dostępne we wszystkich aplikacjach i skryptach. Aby w oknie terminalu wyświetlić wszystkie zmienne środowiskowe powiązane z danym procesem terminalu, należy wykonać polecenie `env` lub `printenv`.

```
$> env
PWD=/home/clif/ShellCookBook
HOME=/home/clif
SHELL=/bin/bash
#  ...i wiele więcej wierszy
```

W przypadku każdego procesu wyświetlenie używanych zmiennych środowiskowych umożliwia następujące polecenie:

```
cat /proc/$PID/environ
```

W miejsce łańcucha `PID` wstaw identyfikator odpowiedniego procesu (`PID` jest zawsze wartością całkowitą).

Dla przykładu załóżmy, że działa aplikacja o nazwie `gedit`. W celu uzyskania identyfikatora procesu tej aplikacji wykonaj następujące polecenie `pgrep`:

```
$ pgrep gedit
12501
```

Stosując następujące polecenie, możesz uzyskać zmienne środowiskowe powiązane z procesem:

```
$ cat /proc/12501/environ
GDM_KEYBOARD_LAYOUT=usGNOME_KEYRING_PID=1560USER=slynuxHOME=/home/slynux
```

Zauważ, że dla wygody wiele zmiennych środowiskowych jest usuwanych. Rzeczywiste dane wyjściowe mogą zawierać różne zmienne.

Plik specjalny `/proc/PID/environ` zawiera listę zmiennych środowiskowych i ich wartości. Każda zmienna jest reprezentowana jako para *`nazwa=wartość`*. Pary są oddzielone ciągiem znaków `\0`. Taka reprezentacja jest trudna w interpretacji przez ludzi.

Możesz uzyskać czytelniejszy raport, przekazując wyjście polecenia `cat` do polecenia `tr` i wstawiając ciąg `\n` zamiast ciągu `\0`:

```
$ cat /proc/12501/environ  | tr '\0' '\n'
```

Jak to zrobić

Zmienna może być przypisana w następujący sposób:

```
zmienna=wartość
```

Łańcuch *`zmienna`* to nazwa zmiennej, a łańcuch *`wartość`* określa wartość do przypisania. Jeśli wartość nie zawiera żadnych białych znaków (np. spacji), nie musi być umieszczana w cudzysłowie. W przeciwnym razie konieczne jest zastosowanie dla wartości cudzysłowu lub apostrofów.

Zauważ, że zapisy *`zmienna = wartość`* i *`zmienna=wartość`* są różne. Częstym błędem jest wprowadzanie formatu *`zmienna = wartość`* zamiast *`zmienna=wartość`*. Znak równości bez spacji dotyczy operacji przypisania, natomiast razem ze spacją dotyczy operacji równości.

W celu wyświetlenia zawartości zmiennej użyj symbolu dolara ($), a za nim wstaw nazwę zmiennej w następujący sposób:

```
zmienna="wartość" # przypisanie wartości zmiennej zmienna.
echo $zmienna
```

lub:

```
echo ${zmienna}
```

Dane wyjściowe są następujące:

```
wartość
```

W poleceniach `printf`, `echo` i innych poleceniach powłoki możesz użyć wartości zmiennych, umieszczając je w cudzysłowie.

```
#!/bin/bash
#nazwa pliku: zmienne.sh
fruit=jabłko
count=5
echo "Liczba owoców: $count ${fruit}(ek)."
```

Dane wyjściowe są następujące:

```
Liczba owoców: 5 jabłko(ek).
```

Ponieważ w powłoce spacja służy do rozdzielania słów, musimy skorzystać z nawiasów klamrowych, aby powłoka rozpoznała zmienną o nazwie `fruit`, a nie `fruit(ek)`.

Zmienne środowiskowe dziedziczą po procesach nadrzędnych. Na przykład `HTTP_PROXY` to zmienna środowiskowa, określająca, jaki serwer proxy powinien być użyty dla połączenia internetowego.

Zmienną tę ustawia się następująco:

```
HTTP_PROXY=192.168.0.2:3128
export HTTP_PROXY
```

Polecenie `export` służy do deklarowania jednej lub kilku zmiennych, które będą dziedziczone przez zadania potomne. Po wyeksportowaniu zmiennych dowolna aplikacja uruchomiona z poziomu bieżącego skryptu powłoki otrzyma tę zmienną. Jest wiele standardowych zmiennych środowiskowych utworzonych i wykorzystywanych przez powłokę. Możemy też eksportować własne zmienne.

Przykładowo, zmienna `PATH` zawiera listę folderów, w których powłoka będzie szukała aplikacji. Typowa zmienna `PATH` będzie zawierała:

```
$ echo $PATH
/home/slynux/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/
bin:/usr/games
```

Ścieżki do katalogów są oddzielane od siebie znakiem `:`. Zazwyczaj zmienna `$PATH` jest definiowana w plikach */etc/environment*, */etc/profile* lub *~/.bashrc*.

Gdy konieczne jest dodanie nowej ścieżki do zmiennej środowiskowej `PATH`, użyj następującego polecenia:

```
export PATH="$PATH:/home/user/bin"
```

Możesz również zastosować polecenia:

```
$ PATH="$PATH:/home/user/bin"
$ export PATH
$ echo $PATH
/home/slynux/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/
bin:/usr/games:/home/user/bin
```

W tym przypadku do zmiennej `PATH` dodano ścieżkę */home/user/bin*.

Oto kilka powszechnie znanych zmiennych środowiskowych: `HOME`, `PWD`, `USER`, `UID` i `SHELL`.

Jeśli użyjemy cudzysłowów, zmienne nie zostaną rozwinięte i zostaną wyświetlone w taki sposób, w jaki zostały wpisane. Zatem polecenie `$ echo '$zmienna'` zwróci napis `$zmienna`.

Natomiast w wyniku uruchomienia polecenia `$ echo "$zmienna"` uzyskamy wartość zmiennej `$zmienna`, o ile została zdefiniowana. W przeciwnym razie nie uzyskamy żadnych danych wyjściowych.

To nie wszystko

W powłoce zdefiniowano znacznie więcej funkcji. Poniżej opisano niektóre z nich.

Określanie długości łańcucha

W celu uzyskania długości wartości zmiennej użyj następującego zapisu:

```
length=${#var}
```

Oto przykład:

```
$ var=123456789012345678905
$ echo ${#var}
20
```

`length` określa liczbę znaków w łańcuchu.

Identyfikowanie bieżącej powłoki

Zmienna środowiskowa `SHELL` umożliwia uzyskanie informacji o bieżącej powłoce:

```
echo $SHELL
```

Możesz też użyć zapisu:

```
echo $0
```

Oto przykład:

```
$ echo $SHELL
/bin/bash
```

Taki sam wynik uzyskamy za pomocą polecenia `echo $0`:

```
$ echo $0
/bin/bash
```

Sprawdzenie dotyczące superużytkownika

`UID` to ważna zmienna środowiskowa, która zawiera wartość identyfikatora użytkownika. Może posłużyć do sprawdzenia, czy bieżący skrypt został uruchomiony przez użytkownika `root`, czy przez zwykłego użytkownika. Oto przykład:

```
if [ $UID -ne 0 ]; then
echo Użytkownik inny niż root. Uruchom jako użytkownik root.
else
echo Użytkownik root
fi
```

Warto zauważyć, że znak [pełni funkcję polecenia i należy go oddzielić spacją od reszty napisu. Powyższy skrypt można też zapisać w następujący sposób:

```
if test $UID -ne 0:1
     then
       echo Użytkownik inny niż root. Uruchom jako użytkownik root.
     else
       echo Użytkownik root
   fi
```

W przypadku użytkownika `root` zmienna `UID` ma wartość `0`.

Modyfikowanie łańcucha wiersza poleceń programu Bash (nazwa_użytkownika@nazwa_hosta:~$)

Po otwarciu okna terminalu lub po uruchomieniu powłoki pojawi się łańcuch wiersza poleceń, taki jak `nazwa_użytkownika@nazwa_hosta: /home/$`. Różne dystrybucje systemów GNU/Linux mają nieznacznie różniące się wiersze poleceń i inne kolory. Możliwe jest dostosowanie tekstu wiersza poleceń za pomocą zmiennej środowiskowej `PS1`. Domyślny tekst wiersza poleceń powłoki ustawia się za pomocą wiersza w pliku *~/.bashrc*.

- W następujący sposób możesz wyświetlić wiersz używany do ustawienia zmiennej `PS1`:

  ```
  $ cat ~/.bashrc | grep PS1
  PS1='${debian_chroot:+($debian_chroot)}\u@\h:\w\$ '
  ```

- Aby ustawić niestandardowy łańcuch wiersza poleceń, wprowadź:

  ```
  slynux@localhost: ~$ PS1="PROMPT>" # Zmieniono łańcuch wiersza poleceń.
  PROMPT> Wpisz tutaj polecenia
  ```

- Używając specjalnych sekwencji o zmienionym znaczeniu (np. `\e[1;31`), możesz zastosować kolorowy tekst (przejdź do podrozdziału „Wyświetlanie w oknie terminalu”).

Są również określone znaki specjalne, rozwijane do postaci parametrów systemowych. Na przykład znaki `\u`, `\h` i `\w` są rozwijane odpowiednio do nazwy użytkownika, nazwy hosta i bieżącego katalogu roboczego.

Funkcja dołączająca wartość na początku zmiennych środowiskowych

Zmienne środowiskowe zwykle zawierają listę ścieżek, w których należy szukać plików wykonywalnych, bibliotek i innych elementów. Przykładem są zmienne `$PATH` i `$LD_LIBRARY_PATH`, których zawartość jest zwykle zbliżona do poniższej:

```
PATH=/usr/bin;/bin
LD_LIBRARY_PATH=/usr/lib;/lib
```

Powłoka przed wykonaniem aplikacji (pliku binarnego lub skryptu) sprawdzi zawartość folderów */usr/bin*, a następnie */bin*.

Podczas budowania i instalowania programu na podstawie kodu źródłowego musimy zwykle dodać niestandardowe ścieżki dla nowych plików wykonywalnych i bibliotek. Możemy na przykład zainstalować aplikację *moja_aplikacja* w folderze */opt/moja_aplikacja*, natomiast pliki binarne zostaną umieszczone w folderze */opt/moja_aplikacja/bin*, a biblioteki — w folderze *opt/moja_aplikacja/lib*.

Jak to zrobić

W tym przykładzie przyjrzymy się, jak dodać nowe ścieżki na początku zmiennej środowiskowej. Pierwszy przykład demonstruje wykorzystanie poznanych już technik, natomiast drugi przykład pokazuje utworzenie funkcji, która upraszcza modyfikowanie zmiennej. Funkcje zostaną omówione w dalszej części tego rozdziału.

```
export PATH=/opt/moja_aplikacja/bin:$PATH
export LD_LIBRARY_PATH=/opt/moja_aplikacja/lib;$LD_LIBRARY_PATH
```

Zmienne `PATH` i `LD_LIBRARY_PATH` powinny teraz zawierać następujące foldery:

```
PATH=/opt/moja_aplikacja/bin:/usr/bin:/bin
LD_LIBRARY_PATH=/opt/moja_aplikacja/lib:/usr/lib;/lib
```

Dodawanie nowej ścieżki możemy sobie ułatwić, definiując funkcję, która będzie umieszczać nowe foldery na początku zmiennej w pliku *.bashr*.

```
prepend() { [ -d "$2" ] && eval $1=\"$2':'\$$1\" && export $1; }
```

Z powyższej funkcji można korzystać następująco:

```
prepend PATH /opt/moja_aplikacja/bin
prepend LD_LIBRARY_PATH /opt/moja_aplikacja/lib
```

Jak to działa

Funkcja `prepend()` najpierw sprawdza, czy folder przekazany w drugim parametrze istnieje. Jeśli tak, wyrażenie `eval` przypisze do zmiennej o nazwie podanej w pierwszym parametrze wartość drugiego parametru, po której umieści znak `:` (separator ścieżki), a następnie oryginalną wartość zmiennej.

Jeśli zmienna, na początku której chcemy dodać wartość, jest pusta, na jej końcu znajdzie się znak `:`. Aby to poprawić, zmodyfikujmy funkcję w taki sposób:

```
prepend() { [ -d "$2" ] && eval $1=\"$2\$\{$1:+':'\$$1\}\" && export $1 ; }
```

Skoro już rozważamy tego typu funkcje, możemy wprowadzić rozwijanie parametrów powłoki w następującej postaci:

```
${parametr:+wyrażenie}
```

Powyższy zapis zostanie rozwinięty do wyrażenia, jeśli jest podany parametr, który nie ma wartości `null`.

Po wprowadzeniu w definicji funkcji powyższej zmiany dołączyliśmy znak `:` i oryginalną wartość zmiennej, tylko wtedy gdy zmienna miała już wartość w momencie wywołania funkcji.

Wykonywanie obliczeń matematycznych za pomocą powłoki

Środowisko powłoki Bash umożliwia wykonywanie podstawowych operacji arytmetycznych przy użyciu polecenia `let` oraz operatorów `(( ))` i `[]`. W przypadku przeprowadzania zaawansowanych operacji pomocne są również dwa programy narzędziowe: `expr` i `bc`.

Jak to zrobić

1. Wartość liczbowa może być skojarzona jako zwykłe przypisanie zmiennej przechowywane w postaci łańcucha. Jednakże do przetwarzania tych łańcuchów jako liczb są używane odpowiednie metody:

```
#!/bin/bash
no1=4;
no2=5;
```

2. Polecenie `let` może posłużyć do bezpośredniego wykonania podstawowych operacji. Podczas korzystania z tego polecenia używa się nazw zmiennych bez prefiksu `$`. Oto przykład:

```
let result=no1+no2
echo $result
```

Polecenie let można używać również następująco:

- Operacja inkrementacji:

```
$ let no1++
```

- Operacja dekrementacji:

```
$ let no1--
```

- Zapisy skrócone:

```
let no+=6
let no-=6
```

Są to skróty poleceń odpowiednio `let no=no+6` i `let no=no-6`.

- Inne metody.

Operator [] może być użyty podobnie jak w przypadku polecenia `let` w następujący sposób:

```
result=$[ no1 + no2 ]
```

Dozwolone jest użycie prefiksu $ wewnątrz operatora []. Oto przykład:

```
result=$[ $no1 + 5 ]
```

Możliwe jest też zastosowanie operatora `(( ))`. Nazwa zmiennej umieszczona przed prefiksem $ jest używana w przypadku skorzystania z operatora `(( ))`. Oto przykład:

```
result=$(( no1 + 50 ))
```

Do wykonania podstawowych operacji może być też użyte polecenie `expr`.

```
result=`expr 3 + 4`
result=$(expr $no1 + 5)
```

Wszystkie powyższe metody nie obsługują liczb zmiennoprzecinkowych, a jedynie liczby całkowite.

3. Precyzyjny kalkulator `bc` to zaawansowany program narzędziowy do obliczeń matematycznych. Oferuje szeroki zestaw opcji. Umożliwia wykonywanie operacji zmiennoprzecinkowych i w następujący sposób korzysta z funkcji zaawansowanych:

```
echo "4 * 0.56" | bc
2.24
no=54;
result=`echo "$no * 1.5" | bc`
echo $result
81.0
```

Poleceniu `bc` mogą być przekazane dodatkowe parametry przy użyciu prefiksów operacji ze średnikami w roli separatorów.

- **Określanie dokładności dziesiętnej:** w następującym przykładzie parametr `scale=2` określa liczbę miejsc dziesiętnych jako `2`. A zatem dane wyjściowe polecenia `bc` będą zawierać liczbę z dwoma miejscami dziesiętnymi:

```
echo "scale=2;22/7" | bc
3.14
```

- **Konwersja podstawy za pomocą polecenia bc:** możliwa jest konwersja jednej podstawy systemu liczbowego na inną. Skonwertujmy system dziesiętny na dwójkowy oraz dwójkowy na dziesiętny:

```
#!/bin/bash
# opis: konwersja systemu liczbowego
no=100
echo "obase=2;$no" | bc
1100100
no=1100100
echo "obase=10;ibase=2;$no" | bc
100
```

- Obliczanie kwadratów i pierwiastków kwadratowych może być przeprowadzone w następujący sposób:

```
echo "sqrt(100)" | bc #pierwiastek kwadratowy
echo "10^10" | bc #kwadrat
```

Eksperymentowanie z deskryptorami plików i przekierowywaniem

Deskryptory plików są liczbami całkowitymi powiązanymi z danymi wejściowymi i wyjściowymi. Najpowszechniejsze deskryptory plików to: `stdin`, `stdout` i `stderr`. Możliwe jest przekierowanie zawartości jednego strumienia do drugiego. W przedstawionej poniżej recepturze zaprezentowano przykłady metod przetwarzania i przekierowywania za pomocą deskryptorów plików.

Wprowadzenie

Podczas pisania skryptów często są używane: standardowe wejście (`stdin`), standardowe wyjście (`stdout`) i standardowy błąd (`stderr`). Skrypt może przekierować dane wyjściowe do pliku za pomocą znaku większości. Tekst zwrócony przez polecenie może być tekstem danych wyjściowych lub tekstem komunikatu o błędzie. Domyślnie wyświetlane są zarówno standardowe wyjście (`stdout`), jak i komunikaty błędów (`stderr`). Wspomniane dwa strumienie można wyodrębnić, korzystając z deskryptorów plików określających poszczególne strumienie.

Deskryptory plików są liczbami całkowitymi powiązanymi z otwartym plikiem lub strumieniem danych. Deskryptory plików `0`, `1` i `2` są zarezerwowane w następujący sposób:

- `0` — `stdin`,
- `1` — `stdout`,
- `2` — `stderr`.

Jak to zrobić

1. Tekst można dołączyć do pliku za pomocą znaku większości:

```
$ echo "To jest przykładowy tekst 1" > temp.txt
```

   Polecenie zapisze wyświetlony tekst w pliku *temp.txt*. Jeśli ten plik już istnieje, w ramach operacji przed zapisem zostanie usunięta istniejąca zawartość pliku.

2. Skorzystaj z dwóch znaków większości, aby dołączyć tekst do pliku:

```
$ echo "To jest przykładowy tekst 2" >> temp.txt
```

3. Wyświetl zawartość pliku przy użyciu polecenia `cat`:

```
$ cat temp.txt
To jest przykładowy tekst 1
To jest przykładowy tekst 2
```

Wyjaśnijmy przeznaczenie standardowego błędu, a także to, jak może on zostać przekierowany. Komunikaty `stderr` wyświetlają się, gdy polecenia zwracają komunikat o błędzie. Rozważ następujący przykład:

```
$ ls +
ls: cannot access +: No such file or directory
```

Znak + jest niepoprawnym argumentem, dlatego został zwrócony błąd.

> **Polecenia zakończone powodzeniem i niepowodzeniem**
>
> Gdy polecenie zakończy działanie po wystąpieniu błędu, zwróci status wyjścia różny od zera. Polecenie zwraca wartość zero, gdy pomyślnie zakończy działanie. Zwrócony status może być odczytany ze specjalnej zmiennej `$?` (w celu wyświetlenia statusu wyjścia uruchom polecenie `echo $?` bezpośrednio po ciągu wykonywanego polecenia).

Następujące polecenie wyświetla na ekranie tekst standardowego błędu `stderr`, zamiast umieszczać go w pliku (a ponieważ polecenie nie zwraca danych w strumieniu `stdout`, plik *out.txt* będzie pusty):

```
$ ls + > out.txt
ls: cannot access +: No such file or directory
```

W poniższym poleceniu przekierowujemy `stderr` do pliku *out.txt*, korzystając z zapisu `2>` (dwa większe niż).

```
$ ls + 2> out.txt # Działa.
```

Można też przekierować standardowy błąd `stderr` do jednego pliku, a standardowe wyjście `stdout` do drugiego pliku.

```
$ cmd 2>stderr.txt 1>stdout.txt
```

Możliwe jest również przekierowanie standardowego błędu `stderr` oraz standardowego wyjścia `stdout` do jednego pliku przez przekształcenie standardowego błędu `stderr` w standardowe wyjście `stdout` za pomocą następującej (preferowanej) metody:

```
$ cmd 2>&1 allOutput.txt
```

lub metody alternatywnej:

```
$ cmd &> output.txt
```

Jeśli w danych wyświetlonych w oknie terminalu nie powinno być szczegółów dotyczących standardowego błędu `stderr`, musisz przekierować dane wyjściowe `stderr` do pliku */dev/null*. W efekcie dane zostaną całkowicie usunięte. Dla przykładu rozważmy trzy pliki: *a1*, *a2* i *a3*. W przypadku pliku *a1* dla użytkownika nie ustawiono uprawnień odczytu, zapisu i wykonywania. Gdy okaże się konieczne wyświetlenie zawartości plików o nazwie rozpoczynającej się od litery *a*, możesz użyć polecenia `cat`. Utwórz pliki testowe w następujący sposób:

```
$ echo A1 > a1
$ echo A2 > a2
$ echo A3 > a3
$ chmod 000 a1  # Odebrano wszystkie uprawnienia.
```

W przypadku wyświetlenia zawartości plików przy użyciu znaków wieloznacznych (`a*`) zostanie wygenerowany komunikat o błędzie dla pliku *a1*, ponieważ nie ma on odpowiedniego uprawnienia odczytu.

```
$ cat a*
cat: a1: Permission denied
A2
A3
```

W tym przypadku komunikat `cat: a1: Permission denied` stanowi dane standardowego błędu `stderr`. Dane te możesz przekierować do pliku, natomiast standardowe wyjście wyświetli się w oknie terminalu. Rozważ następujące wiersze:

```
$ cat a* 2> err.txt # Standardowe wyjście stderr przekierowano do pliku err.txt.
A2
A3

$ cat err.txt
cat: a1: Permission denied
```

Niektóre polecenia generują dane wyjściowe potrzebne do dalszej analizy lub takie, które chcemy zapisać na później. Strumień standardowego wyjścia `stdout` można przekierować do pliku lub przekazać za pomocą potoku do innego polecenia. Prawdopodobnie myślisz, że nie ma sposobu, aby jednocześnie mieć ciastko i zjeść ciastko.

Jest sprytny sposób na przekierowanie danych do pliku, a także utworzenie kopii tych danych w postaci standardowego wejścia `stdin` dla następnego zestawu poleceń. Umożliwia to polecenie `tee`, które odczytuje strumień `stdin` i przekierowuje dane wejściowe do strumienia `stdout` oraz do jednego lub kilku plików.

```
polecenie | tee PLIK1 PLIK2 | innePolecenie
```

W poniższym wierszu polecenia dane standardowego wejścia `stdin` są odbierane przez polecenie `tee`. Zapisuje ono kopię standardowego wyjścia `stdout` w pliku *out.txt* i wysyła kolejną kopię dla następnego polecenia jako standardowe wejście `stdin`. Polecenie `cat -n` wstawia numer dla każdego wiersza odebranego ze standardowego wejścia `stdin` i zapisuje wiersz w standardowym wyjściu `stdout`.

```
$ cat a* | tee out.txt | cat -n
cat: a1: Permission denied
     1 A2
     2 A3
```

Sprawdź zawartość pliku *out.txt* w następujący sposób:

```
$ cat out.txt
A2
A3
```

> Zauważ, że wiersz `cat: a1: Permission denied` nie pojawia się, ponieważ został przekierowany do strumienia `stderr`. Polecenie `tee` może odczytywać wyłącznie dane ze standardowego wejścia `stdin`.

Choć domyślnie polecenie `tee` nadpisuje plik, może być użyte z dołączoną opcją `-a`. Oto przykład:

```
$ cat a* | tee –a out.txt | cat –n
```

Polecenia z argumentami są wyświetlane jako `polecenie PLIK1 PLIK2` lub po prostu `polecenie PLIK`.

Aby przekazać dwie kopie danych wejściowych do standardowego wyjścia `stdout`, zastosuj znak `-` jako argument nazwy pliku polecenia:

```
$ polecenie1 | polecenie2 | polecenie -
```

Oto przykład:

```
$ echo kto to jest | tee -
kto to jest
kto to jest
```

W celu zastosowania standardowego wejścia `stdin` możesz również skorzystać z */dev/stdin* jako nazwy pliku danych wyjściowych.

W podobny sposób użyj plików */dev/stderr* i */dev/stdout* odpowiednio dla standardowego błędu i standardowego wyjścia. Są to specjalne pliki urządzeń, które odpowiadają deskryptorom `stdin`, `stderr` i `stdout`.

Jak to działa

Operatory przekierowania (> i >>) wysyłają dane wyjściowe do pliku zamiast do terminalu. Operatory > i >> różnią się. Choć oba przekierowują tekst do pliku, pierwszy operator (>) czyści plik, a następnie dokonuje zapisu, natomiast drugi operator (>>) dodaje dane wyjściowe na końcu istniejącej zawartości pliku.

Domyślnie przekierowanie dotyczy standardowego wyjścia. Aby został pobrany konkretny deskryptor pliku, musisz przed operatorem podać numer deskryptora.

Operator > jest równoznaczny ciągowi 1>. Podobnie jest w przypadku operatora >> (odpowiednik ciągu 1>>).

W przypadku wystąpienia błędów dane wyjściowe standardowego błędu `stderr` są zrzucane w pliku */dev/null*. Plik *./dev/null* jest specjalnym plikiem urządzenia, którego wszystkie dane są usuwane. Urządzenie to często jest nazywane czarną dziurą, ponieważ wszystkie trafiające tam dane zostaną utracone na zawsze.

To nie wszystko

Polecenie odczytujące standardowe wejście `stdin` może odbierać dane na wiele sposobów. Ponadto możliwe jest określenie własnych deskryptorów plików przy użyciu polecenia `cat` i potoków. Oto przykład:

```
$ cat plik | polecenie
$ polecenie1 | polecenie2
```

Przekierowywanie z pliku do polecenia

Używając znaku mniejszości (<), w następujący sposób możesz odczytać dane z pliku jako standardowe wejście `stdin`:

```
$ polecenie < plik
```

Przekierowywanie bloku tekstowego zawartego w skrypcie

Możemy przekierować tekst ze skryptu do pliku. Aby umieścić komunikat w nagłówku automatycznie generowanego pliku, można wykonać następującą operację:

```
#!/bin/bash
cat <<EOF>log.txt
To jest wygenerowany plik, którego nie należy edytować. Zmiany zostaną nadpisane.
EOF
```

Wiersze znajdujące się między wierszami `cat <<EOF >log.txt` i `EOF` zostaną wyświetlone jako dane standardowego wejścia `stdin`. Wyświetl zawartość pliku *log.txt* w następujący sposób:

```
$ cat log.txt
To jest wygenerowany plik, którego nie należy edytować. Zmiany zostaną nadpisane.
```

Niestandardowe deskryptory plików

Deskryptor pliku to abstrakcyjny wskaźnik dotyczący operacji dostępu do pliku. Każda taka operacja jest powiązana ze specjalną liczbą, nazywaną deskryptorem pliku. Liczby `0`, `1` i `2` są zarezerwowanymi liczbami deskryptorów `stdin`, `stdout` i `stderr`.

Za pomocą polecenia `exec` możesz utworzyć własne, niestandardowe deskryptory plików. Jeśli jesteś już zaznajomiony z obsługą plików w dowolnym innym języku programowania, być może zauważyłeś tryby otwierania plików. Zwykle są używane następujące trzy tryby:

- tryb odczytu,
- tryb zapisu z dołączaniem,
- tryb zapisu z usuwaniem.

Znak `<` jest operatorem używanym do odczytu z pliku do standardowego wejścia `stdin`. Z kolei znak `>` to operator służący do zapisu w pliku z operacją usuwania (dane są zapisywane w pliku docelowym po usunięciu jego zawartości). Operator `>>` umożliwia zapis w pliku z operacją dołączania (dane są dołączane do istniejącej zawartości pliku docelowego, która nie przepada). Deskryptory plików mogą być tworzone przy użyciu jednego z trzech trybów.

W następujący sposób utwórz deskryptor pliku w celu odczytania pliku:

```
$ exec 3<input.txt # otwarty do odczytu przy użyciu deskryptora o wartości 3
```

Deskryptor może być użyty w następujący sposób:

```
$ echo To jest wiersz testowy > input.txt
$ exec 3<input.txt
```

Możesz teraz zastosować deskryptor pliku `3` z poleceniami. Oto przykładowe polecenie:

```
$ cat <&3
To jest wiersz testowy
```

Jeśli wymagana jest druga operacja odczytu, nie możesz ponownie zastosować deskryptora pliku `3`. W celu umożliwienia drugiego odczytu lub odczytu z innego pliku konieczne jest utworzenie nowego deskryptora (na przykład `4`) za pomocą polecenia `exec`.

W następujący sposób utwórz deskryptor pliku na potrzeby zapisu (tryb z usuwaniem):

```
$ exec 4>output.txt # otwarty do zapisu
```

Oto przykład:

```
$ exec 4>output.txt
$ echo nowy wiersz >&4
$ cat output.txt
nowy wiersz
```

W następujący sposób utwórz deskryptor pliku na potrzeby zapisu (tryb z dołączaniem):

```
$ exec 5>>input.txt
```

Oto przykład:

```
$ exec 5>>input.txt
$ echo dołączony wiersz >&5
$ cat input.txt
nowy wiersz
dołączony wiersz
```

Tablice zwykłe i tablice asocjacyjne

Tablice służą w skryptach do przechowywania zbioru danych jako osobnych elementów przy użyciu indeksów. Program Bash obsługuje tablice zarówno zwykłe, jak i asocjacyjne. W przypadku zwykłych tablic w roli indeksu możliwe jest użycie tylko liczb całkowitych. Z kolei tablice asocjacyjne pozwalają na zastosowanie łańcucha jako ich indeksu. Zwykłych tablic można używać w przypadku danych uporządkowanych numerycznie, na przykład będących zbiorem kolejnych iteracji. Z kolei tablic asocjacyjnych można używać w przypadku danych uporządkowanych według napisów, na przykład nazw hostów. W kolejnej recepturze dowiemy się, jak korzystać z obydwu typów tablic.

Wprowadzenie

Tablice asocjacyjne są obsługiwane w programie Bash, począwszy od wersji 4.0.

Jak to zrobić

Tablica może być zdefiniowana na wiele sposobów.

1. Zdefiniuj tablicę za pomocą listy wartości podanych w wierszu:

```
array_var=(test1 test2 test3 test4)
# Wartości będą przechowywane w kolejnych położeniach, począwszy od indeksu 0.
```

Tablicę można zdefiniować również jako zestaw par indeks-wartość — w następujący sposób:

```
array_var[0]="test1"
array_var[1]="test2"
array_var[2]="test3"
array_var[3]="test4"
array_var[4]="test5"
array_var[5]="test6"
```

2. Wyświetl zawartość tablicy dla danego indeksu:

```
$ echo ${array_var[0]}
test1
index=5
$ echo ${array_var[$index]}
test6
```

3. Wyświetl wszystkie wartości tablicy jako listę:

```
$ echo ${array_var[*]}
test1 test2 test3 test4 test5 test6
Możesz też wykonać poniższe polecenie:
$ echo ${array_var[@]}
test1 test2 test3 test4 test5 test6
```

4. Wyświetl długość tablicy (liczbę elementów tablicy) w następujący sposób:

```
$ echo ${#array_var[*]}
6
```

To nie wszystko

Tablice asocjacyjne pojawiły się w programie Bash począwszy od wersji 4.0. Jeżeli jako indeksów używamy łańcuchów (na przykład nazw witryn, nazw użytkowników czy nieuporządkowanych liczb), wówczas łatwiej przetwarza się tablice asocjacyjne niż tablice z indeksami liczbowymi.

Definiowanie tablic asocjacyjnych

W tablicy asocjacyjnej jako jej indeks możesz użyć dowolnych danych tekstowych. Na początku jest wymagana instrukcja do zadeklarowania nazwy zmiennej jako tablicy asocjacyjnej.

```
$ declare -A ass_array
```

Po zadeklarowaniu elementy mogą być dodane do tablicy asocjacyjnej przy użyciu dwóch metod w następujący sposób:

- Przy użyciu metody wstawianej listy par indeks-wartość:

```
$ ass_array=([indeks1]=wartość1 [indeks2]=wartość2)
```

- Inny sposób to skorzystanie z osobnych przypisań par indeks-wartość:

```
$ ass_array[indeks1]=wartość1
$ ass_array[indeks2]=wartość2
```

Dla przykładu rozważ przypisanie cen owoców za pomocą tablicy asocjacyjnej:

```
$ declare -A fruits_value
$ fruits_value=([jabłko]='100 złotych' [pomarańcza]='150 złotych')
```

Wyświetl zawartość tablicy w następujący sposób:

```
$ echo "Jabłko kosztuje ${fruits_value[apple]}"
Jabłko kosztuje 100 złotych
```

Wyszczególnianie indeksów tablicy

Tablice korzystają z indeksów do indeksowania każdego z elementów. Tablice zwykłe i asocjacyjne różnią się pod względem typu indeksu.

W następujący sposób możesz uzyskać listę indeksów tablicy:

```
$ echo ${!array_var[*]}
```

Możesz też użyć polecenia:

```
$ echo ${!array_var[@]}
```

Dla poprzedniego przykładu tablicy `fruits_value` uwzględnij następujące polecenie:

```
$ echo ${!fruits_value[*]}
pomarańcza jabłko
```

Sprawdzi się to również w przypadku zwykłych tablic.

Korzystanie z aliasów

Alias to skrót eliminujący konieczność wpisywania długiej sekwencji poleceń. W kolejnej recepturze dowiemy się, jak utworzyć aliasy za pomocą polecenia `alias`.

Jak to zrobić

Z aliasami związane są następujące operacje:

1. Tworzenie aliasu:

   ```
   $ alias nowe_polecenie='sekwencja poleceń'
   ```

   Aby utworzyć skrót dla polecenia instalowania `apt-get install`, wykonaj następujące polecenie:

   ```
   $ alias install='sudo apt-get install'
   ```

   Po zdefiniowaniu aliasu można użyć polecenia `install` zamiast `sudo apt-get install`.

2. Polecenie `alias` ma działanie tymczasowe. Alias istnieje tylko do momentu zamknięcia bieżącego okna terminalu. Aby udostępnić skrót wszystkim powłokom, dodaj powyższe polecenie do pliku *~/.bashrc*. Polecenia w pliku *~/.bashrc* są zawsze wykonywane w momencie wywołania nowego procesu powłoki:

   ```
   $ echo 'alias polecenie="sekwencja poleceń"' >> ~/.bashrc
   ```

3. Aby usunąć alias, usuń jego wpis z pliku *~/.bashrc* (o ile istnieje) lub użyj polecenia `unalias`. Aby usunąć alias o nazwie `example`, można też użyć polecenia `alias example=`.

4. Możesz określić alias dla polecenia `rm`, aby za jego pomocą usunąć oryginał i zachować kopię w katalogu kopii zapasowych:

```
alias rm='cp $@ ~/backup && rm $@'
```

Jeśli utworzysz alias dla już istniejącego polecenia, zostanie ono zastąpione przez nowe polecenie, do którego odwołuje się alias.

To nie wszystko

Jeśli korzystasz z konta użytkownika uprzywilejowanego, tworzenie aliasu może stanowić zagrożenie z punktu widzenia zabezpieczeń. Aby uniknąć takiego zagrożenia, należy skorzystać z techniki pomijania aliasów.

Pomijanie aliasów

Mając na uwadze prostotę tworzenia aliasu dla dowolnego ważnego polecenia, aliasów nie powinni uruchamiać użytkownicy zalogowani jako użytkownicy uprzywilejowani. Możesz zignorować wszelkie aktualnie zdefiniowane aliasy przez zastosowanie znaku zmiany znaczenia dla polecenia, które ma zostać uruchomione. Oto przykład:

```
$ \polecenie
```

Znak \ powoduje, że polecenie jest uruchamiane bez uwzględniania jakichkolwiek zmian określonych za pomocą aliasu. W przypadku wykonywania poleceń uprzywilejowanych w niezaufanym środowisku dobrą praktyką z punktu widzenia bezpieczeństwa jest ignorowanie aliasów przez poprzedzenie polecenia znakiem \. Atakujący może zdefiniować alias dla uprzywilejowanego polecenia przy użyciu własnego polecenia niestandardowego, aby wykraść ważne informacje przekazywane poleceniu przez użytkownika.

Wyświetlanie listy aliasów

Polecenie `alias` umożliwia wyświetlenie aliasów zdefiniowanych w powłoce:

```
$ aliasalias lc='ls -color=auto'
alias ll='ls -l'
alias vi='vim'
```

Uzyskiwanie informacji o terminalu

Podczas pisania skryptów powłoki wiersza poleceń często konieczne będzie intensywne przetwarzanie informacji dotyczących bieżącego okna terminalu, takich jak: liczba kolumn i wierszy, pozycje kursora, maskowane pola haseł itp. Ta receptura ułatwia opanowanie procesu gromadzenia ustawień terminalu i przetwarzanie ich.

Wprowadzenie

`tput` i `stty` to programy narzędziowe, które mogą posłużyć do wykonywania operacji przetwarzania związanych z terminalem.

Jak to zrobić

Oto kilka możliwości polecenia `tput`:

- W następujący sposób uzyskaj liczbę kolumn i wierszy w oknie terminalu:

```
tput cols
tput lines
```

- Aby wyświetlić nazwę bieżącego okna terminalu, użyj następującego polecenia:

```
tput longname
```

- Aby umieścić kursor w pozycji `100,100`, wprowadź polecenie:

```
tput cup 100 100
```

- Ustaw kolor tła terminalu w następujący sposób:

```
tput setb n
```

  Argument `n` może być wartością z przedziału od `0` do `7`.

- Aby ustawić kolor pierwszoplanowy dla tekstu, należy wpisać:

```
tput setf n
```

  Argument `n` może być wartością z przedziału od `0` do `7`.

> Niektóre polecenia, włącznie z popularnym poleceniem `color ls`, mogą zresetować kolor pierwszoplanowy i kolor tła.

- Aby pogrubić tekst, wykonaj polecenie:

```
tput bold
```

- Podkreślanie rozpocznij i zakończ przy użyciu następujących poleceń:

```
tput smul
tput rmul
```

- Aby usunąć dane od pozycji kursora do końca wiersza, użyj polecenia:

```
tput ed
```

- Podczas wpisywania hasła nie powinny być wyświetlane jego znaki. W następującym przykładzie pokazano, jak to osiągnąć za pomocą polecenia `stty`:

```
#!/bin/sh
#nazwa pliku: password.sh
echo -e "Wprowadź hasło: "
#wyłączenie polecenia echo przed odczytem hasła
stty -echo
read password
#ponowne włączenie polecenia echo
stty echo
echo
echo Hasło wczytano.
```

Opcje `-echo` i `echo` powodują odpowiednio wyłączenie i włączenie prezentowania danych wyjściowych w oknie terminalu.

Uzyskiwanie i ustawianie dat oraz opóźnienia

Opóźnienia są często stosowane do zapewnienia czasu oczekiwania (np. 1 sekunda) w trakcie wykonywania programu lub do monitorowania zadań co kilka sekund (lub kilka miesięcy). Zastosowania powiązane z czasem i datami wymagają zrozumienia tego, jak w programie reprezentuje się i przetwarza czas i daty. W tej recepturze wyjaśniono, jak pracować z datami i opóźnieniami czasu.

Wprowadzenie

Daty mogą być wyświetlane w różnych formatach. W systemach uniksowych daty są przechowywane jako wartość całkowita, określająca liczbę sekund, jaka upłynęła od 1 stycznia 1970 roku. i godziny 00:00:00 UTC (uniwersalny czas koordynowany). Ten czas określa się mianem czasu **epoki** lub **czasu uniksowego**.

Możliwe jest ustawienie daty systemowej z poziomu wiersza poleceń. Dowiedz się, jak odczytywać daty i ustawiać je.

Jak to zrobić

Możesz odczytywać dane zapisane w różnym formacie, a także ustawić datę.

1. Datę możesz odczytać w następujący sposób:

```
$ date
wto, 25 paź 2011, 12:19:10 CEST
```

2. Czas epoki możesz zaś wyświetlić w następujący sposób:

```
$ date +%s
1290047248
```

   Czas epoki może zostać określony poleceniem `date` na podstawie danego łańcucha sformatowanej daty. Daty mogą być użyte w wielu formatach dat jako dane wejściowe. Zwykle nie musisz przejmować się użytym formatem łańcucha daty, jeśli datę uzyskujesz z dziennika systemowego lub danych wyjściowych wygenerowanych przez dowolną standardową aplikację. Łańcuch daty możesz przekształcić w czas epoki w następujący sposób:

```
$ date --date "Wed mar 15 08:09:16 EDT 2017" +%s
1489579718
```

   Opcja `--date` służy do udostępniania łańcucha daty jako danych wejściowych. Jednakże do wyświetlenia danych wyjściowych możesz użyć dowolnych opcji formatowania daty. Przekazanie daty wejściowej z łańcucha może być wykorzystane do znalezienia dnia tygodnia dla danej daty.

```
$ date --date "Jan 20 2001" +%A
sobota
```

   Łańcuchy formatów dat wyszczególniono w tabeli znajdującej się w punkcie „Jak to zrobić”.

3. Aby wyświetlić datę w żądanym formacie, należy użyć kombinacji łańcuchów formatu z wstawionym na jej początku znakiem + jako argumentem polecenia `date`. Oto przykład:

```
$ date "+%d %B %Y"
25 październik 2011
```

4. W następujący sposób możesz ustawić datę i godzinę:

```
# date -s "Łańcuch sformatowanej daty"
# date -s "21 June 2009 11:01:22"
```

W systemie połączonym z siecią datę i czas możesz ustawić za pomocą polecenia `ntpdate`. Oto przykład:

```
/usr/sbin/ntpdate -s time-b.nist.gov
```

5. Kod optymalizujemy, mierząc czas jego wykonania. Taką informację możesz wyświetlić w następujący sposób:

```
#!/bin/bash
#nazwa pliku: time_take.sh
start=$(date +%s)
polecenia;
instrukcje;
end=$(date +%s)
difference=$(( end - start))
echo Czas wykonania poleceń wynosi: $difference (w sekundach).
```

Polecenie `date` umożliwia uzyskanie czasu z dokładnością do sekundy. Lepszy wynik pomiaru czasu wykonania skryptu można uzyskać za pomocą polecenia `time`:

```
time nazwaPoleceniaLubSkryptu
```

Jak to działa

Epoka uniksowa jest definiowana jako liczba sekund, jaka upłynęła od północy 1 stycznia 1970 roku. **uniwersalnego czasu koordynowanego** (UTC), nie licząc sekund przestępnych. Czas epoki przydaje się, gdy musisz obliczyć różnicę między dwiema datami lub godzinami. W tym celu należy przekształcić dwa łańcuchy daty w czas epoki, a następnie wyznaczyć różnicę między dwiema wartościami czasu epoki. W poniższym przykładzie obliczymy liczbę sekund, jaka upłynęła między dwiema datami:

```
secs1=`date -d "Jan 2 1970"
secs2=`date -d "Jan 3 1970"
echo "Między 2 a 3 stycznia upłynęło  `expr $secs2 - $secs1` sekund."
Między 2 a 3 stycznia upłynęło 86400 sekund.
```

Czas określony liczbą sekund, jaka upłynęła od północy 1 stycznia 1970 roku, jest trudny do zrozumienia przez ludzi. Polecenie `date` może zwrócić dane wyjściowe w czytelnym formacie.

Tabela na następnej stronie prezentuje opcje formatowania, które można wykorzystać razem z poleceniem `date`.

To nie wszystko

Określanie interwałów czasowych odgrywa kluczową rolę w przypadku pisania skryptów monitorujących wykonywanych w ramach pętli. Dowiedz się, jak wygenerować opóźnienia czasowe.

Element daty	Format
Dzień tygodnia	%a (np. sob) %A (np. sobota)
Miesiąc	%b (np. lis) %B (np. listopad)
Dzień	%d (np. 31)
Data w formacie (mm/dd/rr)	%D (np. 10/18/10)
Rok	%y (np. 10) %Y (np. 2010)
Godzina	%I lub %H (np. 08)
Minuta	%M (np. 33)
Sekunda	%S (np. 10)
Nanosekunda	%N (np. 695208515)
Uniksowy czas epoki (w sekundach)	%s (np. 1290049486)

Generowanie opóźnień w skrypcie

Aby opóźnić wykonanie czegoś w skrypcie o określoną liczbę sekund, użyj polecenia `sleep`. Przykładowy skrypt odlicza od 0 do 40, korzystając z poleceń `tput` i `sleep`:

```
#!/bin/bash
#nazwa pliku: sleep.sh
echo Liczba:
tput sc

#powtarzanie przez 40 sekund
for count in `seq 0 40`
do
     tput rc
     tput ed
     echo -n $count
     sleep 1
done
```

W powyższym przykładzie zmienna przyjmuje kolejno wartości z listy liczb generowanej przez polecenie `seq`. Instrukcja `tput sc` służy do przechowywania pozycji kursora. Dla każdego wykonania pętli w terminalu jest zapisywana nowa wartość licznika przez odtworzenie pozycji kursora dla liczby za pomocą instrukcji `tput rc`. Instrukcja `tput ed` usuwa tekst od bieżącej pozycji kursora do końca wiersza. Po wyczyszczeniu wiersza skrypt za pomocą polecenia `echo` wyświetla nową wartość. Jednosekundowe opóźnienie zapewniono w pętli za pomocą instrukcji `sleep`.

Debugowanie skryptu

Zwykle debugowanie zajmuje więcej czasu niż tworzenie kodu. Jest to jedna z krytycznych funkcji każdego języka programowania, która powinna zostać zaimplementowana w celu uzyskania informacji zwrotnej po wystąpieniu czegoś nieoczekiwanego. Informacje procesu debugowania mogą być wykorzystane do określenia przyczyny awarii programu lub zadziałania w nieoczekiwany sposób. Program Bash zapewnia określone opcje debugowania, które powinny być znane każdemu programiście. W poniższej recepturze dowiemy się, jak z nich skorzystać.

Jak to zrobić

Proces debugowania będzie łatwiejszy, jeśli skorzystamy z wbudowanych narzędzi powłoki Bash lub zadbamy o odpowiednią zawartość pisanych skryptów. Oto kilka przykładów:

1. W następujący sposób dodaj opcję `-x` w celu włączenia śledzenia skryptu powłoki przez proces debugowania:

   ```
   $ bash -x skrypt.sh
   ```

   Uruchomienie skryptu z flagą `-x` spowoduje wyświetlenie każdego wiersza źródłowego z bieżącym statusem.

> Możesz też skorzystać z polecenia `sh –x skrypt`.

2. Zdebuguj tylko fragment skryptu, korzystając z poleceń `set -x` i `set +x`. Na przykład:

   ```
   #!/bin/bash
   #nazwa pliku: debug.sh
   for i in {1..6}
   do
       set -x
       echo $i
       set +x
   done
   echo "Skrypt wykonano"
   ```

   Po wykonaniu tego skryptu zostaną wyświetlone jedynie informacje debugowania powiązane z instrukcją `echo $i`, ponieważ debugowanie ograniczono do sekcji tej instrukcji przy użyciu flag `-x` i `+x`.

   Użyta w powyższym skrypcie konstrukcja `{start..end}` służy do iterowania od wartości `start` do wartości `end`. Zastępuje ona polecenie `seq`, użyte w poprzednim przykładzie, i jest od niego nieco szybsza.

3. Powyższe metody debugowania działają dzięki wbudowanym elementom powłoki Bash. Jednakże zawsze generują one informacje debugowania w takim samym formacie. W wielu przypadkach niezbędne jest zaprezentowanie informacji debugowania w formacie niestandardowym. Aby włączyć lub wyłączyć debugowanie i generowanie komunikatów we własnym stylu debugowania, należy zdefiniować zmienną środowiskową `_DEBUG`.

   Przyjrzyj się następującemu przykładowemu kodowi:

```
#!/bin/bash
function DEBUG()
{
    [ "$_DEBUG" == "on" ] && $@ || :
}
for i in {1..10}
do
    DEBUG echo "I wynosi $i"
done
```

   Powyższy skrypt możesz uruchomić z wartością `on` ustawioną dla zmiennej środowiskowej `_DEBUG` w następujący sposób:

```
$ _DEBUG=on ./skrypt.sh
```

   Zmienna ta jest umieszczana przed każdą instrukcją, w przypadku której mają zostać wyświetlone informacje debugowania. Jeśli ciąg `_DEBUG=on` nie będzie przekazany skryptowi, informacje debugowania nie zostaną zaprezentowane. W programie Bash polecenie `:` nakazuje powłoce, aby nie wykonywała żadnej operacji.

Jak to działa

Flaga `-x` kieruje każdy wiersz wykonywanego skryptu do standardowego wyjścia. Jednakże wymagane może być obserwowanie tylko niektórych fragmentów wierszy źródłowych. W takiej sytuacji możesz skorzystać z wbudowanego polecenia `set`, dzięki czemu w skrypcie można będzie włączać i wyłączać wyświetlanie informacji debugowania. Oto argumenty polecenia `set`:

- `set -x`: wyświetla argumenty i polecenia w momencie ich wykonywania;
- `set +x`: wyłącza debugowanie;
- `set -v`: wyświetla dane wejściowe w momencie ich odczytywania;
- `set +v`: wyłącza wyświetlanie danych wejściowych.

To nie wszystko

Możliwe jest też użycie innych wygodnych metod debugowania skryptów. W tym celu możesz wykorzystać magiczny ciąg `#!` w bardziej zaawansowany sposób.

Użycie magicznego ciągu #!

Postać wiersza magicznego ciągu `#!` może zostać zmieniona z `#!/bin/bash` na `#!/bin/bash –xv`, przez co włączy się debugowanie bez stosowania żadnych dodatkowych flag (oprócz oczywiście flagi `-xv`).

Śledzenie przebiegu wykonywania skryptu może być trudne w przypadku domyślnego strumienia wyjścia, w którym każdy wiersz jest poprzedzony znakiem +. Aby wyświetlić rzeczywiste numery wierszy, ustaw zmienną środowiskową `PS4` na `'$LINENO:'`, jak w poniższym przykładzie:

```
PS4='$LINENO: '
```

Dane wyjściowe wygenerowane podczas debugowania mogą być obszerne. Jeśli używamy polecenia `-x` lub `set -x`, dane wyjściowe debugowania są przekazywane do strumienia `stderr`. Można je jednak przekierować do pliku za pomocą następującego polecenia:

```
sh -x testScript.sh 2> debugout.txt
```

Powłoka Bash począwszy od wersji 4.0 wspiera wyświetlanie wyników debugowania przy użyciu numerowanych strumieni:

```
exec 6> /tmp/debugout.txt
BASH_XTRACEFD=6
```

Funkcje i argumenty

Na pierwszy rzut oka funkcje i aliasy wydają się podobne, jednak różnią się nieco działaniem. Główna różnica polega na tym, że argumenty funkcji można wykorzystać w dowolnym miejscu definicji funkcji, natomiast w aliasie argumenty są po prostu dołączane na końcu polecenia.

Jak to zrobić

Funkcję definiuje się za pomocą polecenia `function`, nazwy funkcji, nawiasów otwierających i zamykających oraz ciała funkcji zawartego między nawiasami klamrowymi.

1. Funkcja może być zdefiniowana w następujący sposób:

```
function nazwa_funkcji()
{
    instrukcje;
}
```

lub:

```
nazwa_funkcji()
{
    instrukcje;
}
```

Definicja funkcji może mieć nawet taką postać (w przypadku prostych funkcji):

```
nazwa_funkcji() { instrukcje; }
```

2. Funkcja może być wywołana przy użyciu jej nazwy:

```
$ nazwa_funkcji ; # Funkcja zostanie wykonana.
```

3. Argumenty przekazywane funkcjom są dostępne według kolejności. `$1` odpowiada pierwszemu argumentowi, `$2` drugiemu itd.:

```
nazwa_funkcji arg1 arg2 ; # przekazanie argumentów
```

Poniżej zaprezentowano sposób definiowania funkcji `nazwa_funkcji`. Wewnątrz tej funkcji uwzględniono różne metody uzyskiwania dostępu do jej argumentów.

```
nazwa_funkcji()
{
    echo $1, $2; # uzyskiwanie dostępu do argumentów arg1 i arg2
    echo "$@"; # jednoczesne wyświetlenie wszystkich argumentów jako listy
    echo "$*"; # Podobnie jak w przypadku $@, lecz argumenty są pobierane jako jedna całość.
    return 0; # Zwraca wartość.
}
```

Argumenty przekazane do skryptów mogą być użyte za pomocą instrukcji `$0` (reprezentuje nazwę skryptu). Oto lista argumentów:

- `$1` identyfikuje pierwszy argument;
- `$2` identyfikuje drugi argument;
- `$n` identyfikuje n-ty argument;
- `"$@"` jest rozwijane do postaci `"$1" "$2" "$3"` itd;
- `"$*"` jest rozwijane do postaci `"$1c$2c$3"`, gdzie `c` to pierwszy znak separatora IFS (ang. *Internal Field Separator*).
- Ciągu `"$@"` używa się najczęściej. Ciąg `"$*"` jest rzadko stosowany, ponieważ przekazuje wszystkie argumenty jako pojedynczy łańcuch.

Porównywanie aliasu do funkcji

- Poniżej zdefiniowany jest alias służący do wyświetlania podzbioru plików poprzez przekazanie danych wyjściowych z polecenia `ls` do programu `grep`. Argument znajduje się na końcu polecenia, dlatego fragment `lsg txt` zostanie rozwinięty do postaci `ls | grep txt`:

```
$> alias lsg='ls | grep'
$> lsg txt
    plik1.txt
    plik2.txt
    plik3.txt
```

- Jeśli chcemy w ten sposób uzyskać adres IP urządzenia */sbin/ifconfig*, możemy wykonać następujące polecenia:

```
$> alias wontWork='/sbin/ifconfig | grep'
$> wontWork eth0
eth0  Link  encap:Ethernet  HWaddr 00:11::22::33::44:55
```

- Polecenie `grep` znalazło łańcuch `eth0` zamiast adresu IP. Jeśli użyjemy funkcji zamiast aliasu, możemy przekazać argument do polecenia `ifconfig`, zamiast dołączać go do polecenia `grep`:

```
$> function getIP() { /sbin/ifconfig $1 | grep 'inet ';  }
$> getIP eth0
inet addr:192.168.1.2 Bcast:192.168.255.255 Mask:255.255.0.0
```

To nie wszystko

Poznaj więcej wskazówek dotyczących funkcji programu Bash.

Funkcja rekurencyjna

Funkcje w programie Bash obsługują też rekurencję (czyli wywoływanie funkcji przez samą siebie). Oto przykład:

```
F() { echo $1; F witaj; sleep 1; }
```

Fork-bomba

Funkcja rekurencyjna wywołuje samą siebie. Funkcje rekurencyjne muszą mieć warunek wyjścia, w przeciwnym razie będą wywoływane aż do wyczerpania zasobów systemu, a w konsekwencji — do awarii.

Funkcja :(){ :|:& };: bez końca inicjuje procesy, co kończy się atakiem DoS (ang. *Denial of Service*).

Przed znakiem & znajduje się wywołanie funkcji mające na celu umieszczenie podprocesu w tle. Jest to niebezpieczny kod, ponieważ nieustannie tworzy procesy za pomocą funkcji `fork`. Z tego powodu jest określany mianem fork-bomby.

Możesz mieć problem z interpretacją powyższego kodu. Na stronie Wikipedii (*http://pl.wikipedia.org/wiki/Fork-bomba*) zamieszczono więcej szczegółów, a także interpretację kodu fork-bomby.

Aby zapobiec atakowi DoS, w pliku konfiguracyjnym */etc/security/limits.conf* ogranicz maksymalną liczbę procesów, które mogą być wywoływane, przypisując tę wartość do zmiennej `nproc`.

Poniższy kod ograniczy do 100 liczbę procesów, jakie może uruchomić użytkownik:

```
hard nproc 100
```

Eksportowanie funkcji

Funkcja może być wyeksportowana, podobnie jak zmienne środowiskowe, za pomocą polecenia `export`, w taki sposób, że zasięg funkcji może być rozszerzony o podprocesy:

```
export -f nazwa_funkcji
$> function getIP() { /sbin/ifconfig $1 | grep 'inet '; }
$> echo "getIP eth0" >test.sh
$> sh test.sh
    sh: getIP: No such file or directory
$> export -f getIP
$> sh test.sh
    inet addr: 192.168.1.2 Bcast: 192.168.255.255 Mask:255.255.0.0
```

Odczytywanie wartości zwracanej (statusu) polecenia

Ciąg $? zapewni wartość zwracaną polecenia *nazwa_polecenia*.

```
nazwa_polecenia;
echo $?;
```

Wartość zwracana jest nazywana **statusem wyjścia**. Może ona być użyta do analizowania tego, czy działanie polecenia zakończyło się pomyślnie, czy też nie. Jeśli polecenie zakończyło się pomyślnie, status wyjścia będzie równy zero. W przeciwnym razie będzie to wartość różna od zera.

W następujący sposób możesz sprawdzić, czy działanie polecenia zakończyło się powodzeniem, czy nie:

```
#!/bin/bash
#nazwa pliku: success_test.sh
#Sprawdzenie argumentów wiersza poleceń, np. success_test.sh 'ls |grep txt'.
eval $@
if [ $? –eq 0 ];
then
    echo "Polecenie $CMD zakończyło się pomyślnie"
else
    echo "Polecenie $CMD zakończyło się niepomyślnie"
fi
```

Przekazywanie argumentów poleceniom

Argumenty mogą być przekazywane większości poleceń w różnych formatach. Załóżmy, że dostępne są opcje `-p` i `-v`, a `-k LICZBA` to kolejna opcja, która pobiera liczbę. Ponadto polecenie pobiera nazwę pliku jako argument. Polecenie może być wykonane na wiele sposobów, np.:

- $ polecenie -p -v -k 1 plik
- $ polecenie -pv -k 1 plik
- $ polecenie -vpk 1 plik
- $ polecenie plik -pvk 1

W skrypcie argumenty wiersza poleceń można odczytać według kolejności ich wystąpienia w wierszu poleceń. Pierwszemu argumentowi odpowiada zmienna $1, drugiemu — $2 itd.

Poniższy skrypt wyświetli pierwsze trzy argumenty wiersza poleceń:

```
echo $1 $2 $3
```

Częściej spotyka się sytuacje, w których iterujemy po kolei po argumentach polecenia. Polecenie shift przesuwa w lewo każdy argument po kolei, dzięki czemu w skrypcie można odczytać wartość każdego argumentu za pomocą zmiennej $1. Poniższy skrypt wyświetli wartości wszystkich argumentów wiersza poleceń:

```
$ cat showArgs.sh
for i in `seq 1 $#`
do
echo $i ma wartość $1
shift
done
$ sh showArgs.sh a b c
1 ma wartość a
2 ma wartość b
3 ma wartość c
```

Przekazywanie danych wyjściowych do drugiego polecenia

Jedną z najbardziej przydatnych cech powłok w systemie Unix jest łatwość łączenia wielu poleceń w celu uzyskania danych wyjściowych. Dane wyjściowe jednego polecenia mogą pojawić się na wejściu innego polecenia, które przekazuje swoje dane wyjściowe do kolejnego polecenia itd. Dane wyjściowe takiej kombinacji poleceń mogą być odczytane ze zmiennej. W tej recepturze pokazano, jak połączyć wiele poleceń, a także w jaki sposób można odczytać ich dane wyjściowe.

Wprowadzenie

Dane wejściowe są zwykle przekazywane poleceniu za pośrednictwem standardowego wejścia stdin lub argumentów. Dane wyjściowe mają postać standardowego błędu (stderr) lub standardowego wyjścia (stdout). W przypadku łączenia wielu poleceń zazwyczaj na potrzeby danych wejściowych i wyjściowych są używane deskryptory odpowiednio stdin i stdout.

W tym kontekście polecenia są nazywane filtrami. Poszczególne filtry są łączone za pomocą potoków. Operatorem potoku jest znak |. Oto następujący przykład:

```
$ polecenie1 | polecenie2 | polecenie3
```

W tym przypadku połączono trzy polecenia. Dane wyjściowe polecenia *polecenie1* trafiają do polecenia *polecenie2*, a jego dane wyjściowe są przekazywane poleceniu *polecenie3*. Ostateczne dane wyjściowe (zwrócone przez polecenie *polecenie3*) zostaną wyświetlone lub skierowane do pliku.

Jak to zrobić

Potoków można używać w metodzie podpowłoki do łączenia danych wyjściowych wielu poleceń.

1. Zacznij od połączenia dwóch poleceń:

   ```
   $ ls | cat -n > out.txt
   ```

   Dane wyjściowe polecenia `ls` (listing bieżącego katalogu) są przekazywane poleceniu `cat -n`, które do odebranych danych wstawia numery wierszy za pośrednictwem standardowego wejścia `stdin`. A zatem dane wyjściowe polecenia są kierowane do pliku *out.txt*.

2. Dane wyjściowe sekwencji poleceń połączonych możesz odczytać w następujący sposób:

   ```
   cmd_output=$(POLECENIA)
   ```

   Jest to określane mianem **metody podpowłoki**. Oto przykład:

   ```
   cmd_output=$(ls | cat -n)
   echo $cmd_output
   ```

   Inna metoda, oparta na znakach `, może też być użyta do przechowywania danych wyjściowych poleceń:

   ```
   cmd_output=`POLECENIA`
   ```

   Oto przykład:

   ```
   cmd_output=`ls | cat -n`
   echo $cmd_output
   ```

Ukośnik różni się od znaku apostrofu. Ukośnik znajduje się na klawiszu ze znakiem ~.

To nie wszystko

Jest wiele sposobów grupowania poleceń.

Wywoływanie osobnego procesu z podpowłoką

Podpowłoki są osobnymi procesami. Podpowłoka może zostać następująco zdefiniowana przy użyciu operatorów `( )`:

- Polecenie `pwd` wyświetla ścieżkę katalogu roboczego.
- Polecenie `cd` zmienia bieżący katalog na katalog o podanej ścieżce.

```
$> pwd
/
$> (cd /bin; ls)
awk bash cat...
$> pwd
/
```

Gdy określone polecenia są wykonywane w podpowłoce, w bieżącej powłoce nie zachodzi żadna zmiana. Zmiany są ograniczone do podpowłoki. Jeśli na przykład w bieżącym katalogu podpowłoki wprowadzono zmianę za pomocą polecenia `cd`, zmiana ta nie zostanie odzwierciedlona w środowisku głównej powłoki.

Stosowanie cudzysłowu w przypadku podpowłoki w celu zachowania odstępów i znaku nowego wiersza

Załóżmy, że dane wyjściowe polecenia są wczytywane do zmiennej przy użyciu podpowłoki lub metody opartej na znakach `` ` ``. Aby zachować odstępy i znak nowego wiersza (`\n`), zawsze umieszczaj dane wyjściowe w cudzysłowie. Oto przykład:

```
$ cat text.txt
1
2
3

$ out=$(cat text.txt)
$ echo $out
1 2 3 # brak w ciągu 1, 2, 3 odstępów uzyskiwanych za pomocą znaków \n

$ out="$(cat text.txt)"
$ echo $out
1
2
3
```

Odczytywanie n znaków bez naciskania klawisza Enter

Program Bash zawiera polecenie `read`, które może posłużyć do odczytania tekstu ze standardowego wejścia lub tekstu wprowadzonego za pomocą klawiatury. Choć jest ono używane do interaktywnego odczytywania danych wpisanych przez użytkownika, oferuje znacznie więcej możliwości. Większość obsługujących dane wejściowe bibliotek obecnych w dowolnym języku programowania wczytuje dane wprowadzane za pomocą klawiatury. Wtedy zakończenie wpisywania łańcucha jest sygnalizowane przez wciśnięcie klawisza *Enter*. Bywają jednak krytyczne sytuacje, w których klawisz *Enter* nie może zostać wciśnięty, i wówczas operacja zakończenia jest realizowana na podstawie liczby znaków lub pojedynczego znaku. Na przykład

w grze piłka jest przemieszczana w górę po naciśnięciu klawisza +. Naciśnięcie klawisza +, a następnie, w celu potwierdzenia tej czynności, każdorazowo klawisza *Enter* nie jest efektywne.

Przyjrzyjmy się nowej recepturze, która ilustruje, jak wykonać wspomniane zadanie za pomocą polecenia `read`, bez konieczności naciśnięcia klawisza *Enter*.

Jak to zrobić

Różne opcje polecenia `read` umożliwiają uzyskanie różnych rezultatów, zgodnie z poniższym opisem:

1. W następującym wierszu polecenia *n* znaków zostanie z wejścia wczytanych do zmiennej *`nazwa_zmiennej`*:

   ```
   read -n liczba_znaków nazwa_zmiennej
   ```

   Oto przykład:

   ```
   $ read -n 2 var
   $ echo $var
   ```

2. Wczytanie hasła w trybie bez wyświetlania na ekranie:

   ```
   read -s var
   ```

3. Wyświetlenie komunikatu za pomocą polecenia `read`:

   ```
   read -p "Wprowadź dane wejściowe:"  var
   ```

4. Wczytanie danych wejściowych po upływie czasu oczekiwania:

   ```
   read -t czas_oczekiwania var
   ```

   Oto przykład:

   ```
   $ read -t 2 var
   # wczytanie do zmiennej var łańcucha wpisanego w ciągu 2 sekund
   ```

5. Aby zakończyć wiersz danych wejściowych, należy użyć znaku separatora:

   ```
   read -d znak_separatora
   ```

   Oto przykład:

   ```
   $ read -d ":" var
   Witaj:# Dla zmiennej var ustawiono łańcuch Witaj.
   ```

Wykonywanie polecenia aż do osiągnięcia zamierzonego celu

Czasem polecenie zakończy się sukcesem tylko po spełnieniu pewnych warunków. Przykładowo, plik można pobrać dopiero po jego utworzeniu. W podobnych sytuacjach przydaje się możliwość powtarzalnego wykonywania polecenia aż do osiągnięcia zamierzonego celu.

Jak to zrobić

Funkcję należy zdefiniować następująco:

```
repeat()
{
    while true
    do
        $@ && return
    done
}
```

Inny sposób polega na dodaniu poniższego kodu do pliku `rc` powłoki:

```
repeat() { while true; do $@ && return; done }
```

Jak to działa

W funkcji `repeat` zdefiniowano nieskończoną pętlę `while`, w której jest wywoływane polecenie przekazane do funkcji w postaci parametru (dostępne poprzez zmienną `$@`). Jeśli polecenie powiedzie się, pętla zostanie przerwana.

To nie wszystko

Poznaliśmy podstawowy sposób wielokrotnego uruchamiania poleceń aż do ich pomyślnego zakończenia. Możemy go nieco usprawnić.

Szybszy sposób

W najnowszych systemach wartość `true` jest zaimplementowana jako wartość binarna w katalogu */bin*. Oznacza to, że podczas każdego wykonania pętli `while` powłoka musi uruchomić proces. Aby tego uniknąć, możemy użyć wbudowanego w pętlę polecenia `:`, które zawsze zwraca kod wyjścia 0:

```
repeat() { while :; do $@ && return; done }
```

Chociaż ten sposób jest mniej czytelny, to jednak jest szybszy od poprzedniego.

Dodawanie opóźnienia

Załóżmy, że za pomocą funkcji `repeat()` chcemy pobrać z internetu plik, który nie jest jeszcze dostępny, ale kiedyś będzie. Oto przykład:

```
repeat wget -c http://www.example.com/software-0.1.tar.gz
```

Powyższy skrypt będzie generował ogromny ruch na serwerze *www.example.com*, co może sprawiać problemy (nie tylko obsłudze serwera, ale być może również Tobie, jeśli serwer umieści Cię na czarnej liście adresów IP przeznaczonej dla agresorów). Aby tego uniknąć, zmodyfikujmy funkcję i dodajmy opóźnienie:

```
repeat() { while :; do $@ && return; sleep 30; done }
```

Dzięki powyższej zmianie polecenie będzie wykonywane co 30 sekund.

Separatory pól i iteratory

Wewnętrzny separator pól IFS to ważne zagadnienie w przypadku skryptów powłoki. Separator jest bardzo przydatny podczas przetwarzania danych tekstowych.

Separator **IFS** ma specjalne zastosowanie. Jest on zmienną środowiskową przechowującą znaki rozdzielające. Jest to domyślny łańcuch separatorów używany przez działające środowisko powłoki.

Wyobraź sobie sytuację, w której konieczna jest iteracja słów w łańcuchu lub **wartości rozdzielonych przecinkami** (format CSV — ang. *Comma Separated Values*). W pierwszym przypadku zostanie użyty separator `IFS=" "`, a w drugim `IFS=","`.

Wprowadzenie

Rozważ przypadek danych formatu CSV:

```
data="imię,płeć,numer,lokalizacja"
# Do odczytania każdej pozycji w zmiennej może być użyty separator IFS.
oldIFS=$IFS
IFS=, # separator IFS ma teraz postać ,
for item in $data;
do
    echo Pozycja: $item
done

IFS=$oldIFS
```

Dane wyjściowe wyglądają następująco:

```
Pozycja: imię
Pozycja: płeć
Pozycja: numer
Pozycja: lokalizacja
```

Domyślną wartością zmiennej IFS jest element odstępu (znak nowego wiersza, znak tabulacji lub spacja).

Jeśli dla zmiennej IFS ustawiono łańcuch `","`, powłoka interpretuje przecinek jako znak separatora. Oznacza to, że podczas iteracji zmienna `$item` pobiera podłańcuchy oddzielone przecinkami jako swoją wartość.

Jeśli dla zmiennej IFS nie ustawiono łańcucha `","`, pełne dane zostaną wyświetlone jako pojedynczy łańcuch.

Jak to zrobić

Przeanalizujmy kolejny przykład użycia zmiennej IFS, biorąc pod uwagę plik */etc/passwd*. W tym pliku każdy wiersz zawiera pozycje oddzielone znakiem `:`. Każdy wiersz odpowiada atrybutowi związanemu z użytkownikiem.

Rozważ następujące dane wejściowe: `root:x:0:0:root:/root:/bin/bash`. Ostatnia pozycja w każdym wierszu określa domyślną powłokę użytkownika.

Aby wyświetlić użytkowników oraz ich domyślne powłoki, możesz użyć następującego kodu ze zmienną IFS:

```
#!/bin/bash
#opis: przykład użycia zmiennej IFS
line="root:x:0:0:root:/root:/bin/bash"
oldIFS=$IFS;
IFS=":"
count=0
for item in $line;
do
    [ $count -eq 0 ]  && user=$item;
    [ $count -eq 6 ]  && shell=$item;
    let count++
done;
IFS=$oldIFS
echo Powłoka użytkownika $user to $shell;
```

Dane wyjściowe mają następującą postać:

```
Powłoka użytkownika root to /bin/bash
```

Pętle szczególnie przydają się w przypadku przeprowadzania iteracji dla sekwencji wartości. Program Bash zapewnia wiele typów pętli.

- **Pętla `for` w przypadku listy:**

  ```
  for var in list;
  do
      polecenia; # Użyj zmiennej $var.
  done
  ```

  Zmienna `list` może być łańcuchem lub sekwencją wartości.

Za pomocą polecenia `echo` z łatwością możesz wygenerować różne sekwencje.

```
echo {1..50} ;# Generowanie listy liczb z zakresu od 1 do 50.

echo {a..z} {A..Z} ;# Tworzenie listy zawierającej małe i wielkie litery.
```

Łącząc takie listy, możesz scalać różne dane.

W przypadku następującego kodu w każdej iteracji zmienna `i` będzie przechowywać znak z zakresu od `a` do `z`:

```
for i in {a..z}; do działania; done;
```

- **Przeprowadzanie iteracji zakresu liczb**:

```
for((i=0;i<10;i++))
{
    polecenia; # Użyj zmiennej $i.
}
```

- **Wykonywanie pętli, aż zostanie spełniony warunek:**

Pętla `while` jest wykonywana, jeśli warunek ma wartość `true`, natomiast pętla `until` jest wykonywana, aż zostanie spełniony warunek:

```
while warunek
do
    polecenia;
done
```

W celu uzyskania pętli nieskończonej jako warunku użyj wartości `true`.

- **Pętla `until`:**

W programie Bash jest dostępna specjalna pętla `until`. Jest ona wykonywana do momentu, w którym dany warunek stanie się prawdziwy. Oto przykład:

```
x=0;
until [ $x -eq 9 ]; # [ $x -eq 9 ] jest warunkiem.
do
    let x++; echo $x;
done
```

Porównania i testy

Kontrola przepływu w programie jest obsługiwana przez instrukcje porównujące i testujące. Program Bash oferuje też kilka opcji służących do przeprowadzania testów. Instrukcje `if`, `if else` oraz operatory logiczne mogą być użyte do wykonywania testów. Określone operatory porównania mogą posłużyć do porównywania elementów danych. Dostępne jest również polecenie `test`, które umożliwia przeprowadzanie testów.

Jak to zrobić

Poznaj kilka metod służących do porównywania danych i wykonywania testów.

- Warunek instrukcji `if`:

```
if warunek;
then
    polecenia;
fi
```

- Instrukcje `else if` i `else`:

```
if warunek;
then
    polecenia;
elif warunek; then
    polecenia
else
    polecenia
fi
```

W przypadku instrukcji `if` i `else` możliwe jest też zagnieżdżanie. Warunki instrukcji `if` mogą mieć znaczną długość. Aby je skrócić, w następujący sposób możesz użyć operatorów logicznych:

```
[ warunek ] && działanie; # Działanie jest wykonywane, gdy warunek jest prawdziwy.
[ warunek ] || działanie; # Działanie jest wykonywane, gdy warunek jest nieprawdziwy.
```

Znaki `&&` i `||` reprezentują operacje logiczne odpowiednio AND i OR. Jest to rozwiązanie bardzo przydatne podczas pisania skryptów programu Bash.

W przypadku porównań matematycznych warunki zwykle umieszcza się w nawiasach kwadratowych `[]`. Zauważ, że między tymi nawiasami i argumentami znajduje się spacja. Jeśli nie wstawiono spacji, zostanie wygenerowany błąd.

```
[ $var -eq 0 ] lub [ $var -eq 0 ]
```

Operacje porównań matematycznych dla zmiennych lub wartości mogą być wykonane w następujący sposób:

```
[ $var -eq 0 ]  # Zwraca wartość true, gdy zmienna $var jest równa 0.
[ $var -ne 0 ] # Zwraca wartość true, gdy zmienna $var nie jest równa 0.
```

Inne ważne operatory to:

- `-gt`: większe niż,
- `-lt`: mniejsze niż,
- `-ge`: większe niż lub równe,
- `-le`: mniejsze niż lub równe.

Operator `-a` jest logicznym operatorem `AND`, natomiast operator `-o` jest logicznym operatorem `OR`. W następujący sposób można łączyć ze sobą wiele warunków testowych:

```
[ $var1 -ne 0 -a $var2 -gt 2 ]  # użycie operatora AND -a
[ $var -ne 0 -o var2 -gt 2 ] # OR -o
```

Testy związane z systemem plików

Przy użyciu różnych flag warunkowych możliwe jest testowanie różnych atrybutów powiązanych z systemem plików:

- `[ -f $file_var ]`: zwraca wartość `true`, jeśli dana zmienna przechowuje zwykłą ścieżkę do pliku lub jego nazwę;
- `[ -x $var ]`: zwraca wartość `true`, jeśli dana zmienna przechowuje ścieżkę do pliku lub nazwę pliku wykonywalnego;
- `[ -d $var ]`: zwraca wartość `true`, jeśli dana zmienna przechowuje ścieżkę katalogu lub jego nazwę;
- `[ -e $var ]`: zwraca wartość `true`, jeśli dana zmienna przechowuje istniejący plik;
- `[ -c $var ]`: zwraca wartość `true`, jeśli dana zmienna przechowuje ścieżkę do pliku urządzenia znakowego;
- `[ -b $var ]`: zwraca wartość `true`, jeśli dana zmienna przechowuje ścieżkę do pliku urządzenia blokowego;
- `[ -w $var ]`: zwraca wartość `true`, jeśli dana zmienna przechowuje ścieżkę do pliku zapisywalnego;
- `[ -r $var ]`: zwraca wartość `true`, jeśli dana zmienna przechowuje ścieżkę do pliku możliwego do odczytania;
- `[ -L $var ]`: zwraca wartość `true`, jeśli dana zmienna przechowuje ścieżkę dowiązania symbolicznego.

Oto przykład zastosowania flagi `-e`:

```
fpath="/etc/passwd"
if [ -e $fpath ]; then
    echo Plik istnieje;
else
    echo Plik nie istnieje;
fi
```

Porównanie łańcuchów

W przypadku porównywania łańcuchów najlepiej skorzystać z podwójnych nawiasów kwadratowych, ponieważ użycie pojedynczych nawiasów może czasem powodować błędy.

> Zwróć uwagę na podwójne nawiasy kwadratowe w rozszerzeniach powłoki Bash. W skryptach, które mają zostać uruchomione w powłoce `ash` lub `dash` (ze względu na większą wydajność), nie można korzystać z podwójnych nawiasów kwadratowych.

Aby stwierdzić, czy dwa łańcuchy są identyczne:

- `[[ $str1 = $str2 ]]`: zwraca wartość `true`, gdy łańcuch `str1` jest równy łańcuchowi `str2`. Oznacza to, że zawartość łańcuchów `str1` i `str2` jest jednakowa.
- `[[ $str1 == $str2 ]]`: alternatywna metoda sprawdzania, czy łańcuchy są identyczne.

W następujący sposób możesz sprawdzić, czy dwa łańcuchy różnią się:

- `[[ $str1 != $str2 ]]`: zwraca wartość `true`, gdy łańcuchy `str1` i `str2` różnią się.

Możesz określić krótszy lub dłuższy łańcuch z uwzględnieniem kolejności alfabetycznej:

Łańcuchy są porównywane według kolejności alfabetycznej poprzez porównywanie wartości ASCII znaków. Na przykład znak *A* jest reprezentowany przez kod ASCII 0x41, natomiast litera *a* jest reprezentowana przez kod 0x61. Dlatego *A* jest mniejsze od *a*, natomiast wartość ciągu *AAa* jest mniejsza niż wartość ciągu *Aaa*.

- `[[ $str1 > $str2 ]]`: zwraca wartość `true`, gdy łańcuch `str1` jest dłuższy od łańcucha `str2` z zachowaniem kolejności alfabetycznej;
- `[[ $str1 < $str2 ]]`: zwraca wartość `true`, gdy łańcuch `str1` jest krótszy od łańcucha `str2` z zachowaniem kolejności alfabetycznej.

Zauważ, że przed znakiem = i po nim znajduje się spacja. Jeśli nie wstawiono spacji, znak = nie spełnia funkcji porównawczej, lecz staje się elementem instrukcji przypisania.

W następujący sposób możesz sprawdzić, czy łańcuch jest pusty:

- `[[ -z $str1 ]]`: zwraca wartość `true`, jeśli łańcuch `str1` przechowuje pusty łańcuch;
- `[[ -n $str1 ]]`: zwraca wartość `true`, jeśli łańcuch `str1` przechowuje niepusty łańcuch.

Za pomocą operatorów logicznych `&&` i `||` łatwiej jest połączyć wiele warunków:

```
if [[ -n $str1 ]] && [[ -z $str2 ]] ;
    then
        polecenia;
    fi
```

Oto przykład:

```
str1="Niepusty "
str2=""
if [[ -n $str1 ]] && [[ -z $str2 ]];
then
    echo Łańcuch str1 nie jest pusty, a łańcuch str2 jest pusty.
fi
```

Dane wyjściowe mają następującą postać:

```
Łańcuch str1 nie jest pusty, a łańcuch str2 jest pusty.
```

Polecenie `test` może być używane do sprawdzania warunków. Pozwala ono uniknąć konieczności użycia wielu nawiasów i zwiększa czytelność kodu. Dla polecenia `test` można bowiem zastosować te same warunki testowania zawarte w nawiasach `[]`.

> Zauważmy, że polecenie `test` jest programem zewnętrznym, dlatego przed użyciem należy pobrać jego kopię. Natomiast polecenie `[` jest wewnętrzną funkcją programu Bash, dzięki czemu jest wydajniejsze. Polecenie `test` jest spójne z powłoką Bourne'a, `ash`, `dash` i innymi.

Oto przykład:

```
if  [ $var -eq 0 ]; then echo "Prawda"; fi
# może być zapisane jako:
if  test $var -eq 0 ; then echo "Prawda"; fi
```

Dostosowywanie powłoki za pomocą plików konfiguracyjnych

Większość poleceń wpisywanych w wierszu poleceń można umieścić w specjalnym pliku, który zostanie wczytany podczas logowania się lub uruchamiania nowej sesji powłoki. Często korzysta się z tej możliwości w celu dostosowania swojej powłoki. W ten sposób można zdefiniować funkcje, aliasy i zmienne środowiskowe.

W pliku konfiguracyjnym zwykle umieszcza się następujące polecenia:

```
# Definiowanie kolorów polecenia ls
LS_COLORS='no=00:di=01;46:ln=00;36:pi=40;33:so=00;35:bd=40;33;01'
export LS_COLORS
# Mój główny znak zachęty
PS1=Witaj $USER'; export PS1
# Aplikacje, które instaluję poza zwykłymi ścieżkami dystrybucyjnymi
PATH=$PATH:/opt/MySpecialApplication/bin; export PATH
# Skrót często używanych poleceń
function lc () {/bin/ls -C $* ; }
```

Za pomocą którego pliku powinienem dostosować powłokę?

W systemach Linux i Unix mamy do dyspozycji kilka plików, które mogą posłużyć do dostosowania powłoki. Można je podzielić na trzy grupy. Do pierwszej grupy należą pliki wczytywane podczas logowania, do drugiej — wczytywane w czasie uruchamiania interaktywnej powłoki, a do trzeciej — pliki wczytywane, gdy w powłoce uruchamiamy przetwarzanie pliku skryptu.

Jak to zrobić

Poniższe pliki są wczytywane podczas logowania użytkownika w powłoce:

```
/etc/profile, $HOME/.profile, $HOME/.bash_login, $HOME/.bash_profile /
```

> Zauważ, że pliki */etc/profile*, *$HOME/.profile* i *$HOME/.bash_profile* mogą nie zostać wczytane, jeśli zalogujesz się poprzez graficzny menedżer logowania. Otóż menedżer graficzny nie uruchamia powłoki. Powłoka zostanie utworzona po otwarciu okna terminalu, ale nie będzie to powłoka logowania.

Jeżeli w powłoce jest zdefiniowany plik *.bash_profile* lub *.bash_login*, wówczas plik *.profile* nie zostanie wczytany.

Poniższe pliki zostaną wczytane przez interaktywną powłokę, na przykład przez sesję terminalu X11 lub podczas uruchamiania jednego polecenia przez `ssh`, na przykład: `ssh 192.168.1.1 ls /tmp`.

```
/etc/bash.bashrc $HOME/.bashrc
```

Uruchom następujący skrypt:

```
$> cat myscript.sh
#!/bin/bash
echo "Skrypt uruchomiony"
```

Żaden z tych plików nie zostanie wczytany, chyba że została zdefiniowana zmienna środowiskowa `BASH_ENV`:

```
$> export BASH_ENV=~/.bashrc
$> ./myscript.sh
```

Za pomocą `ssh` uruchom jedno polecenie, na przykład następujące:

```
ssh 192.168.1.100 ls /tmp
```

W ten sposób zostanie uruchomiona powłoka Bash, która wczyta pliki */etc/bash.bashrc* i *$HOME/.bashrc*, ale pominie pliki */etc/profile* i *.profile*.

W następujący sposób uruchom sesję logowania `ssh`:

```
ssh 192.168.1.100
```

W powyższy sposób utworzysz nową powłokę logowania programu Bash, która wczyta poniższe pliki:

```
/etc/profile
/etc/bash.bashrc
$HOME/.profile lub .bashrc_profile
```

Ostrzeżenie: Ten plik zostanie też wczytany przez inne powłoki, na przykład przez tradycyjną powłokę Bourne'a, `ash`, `dash` i `ksh`. Wspomniane powłoki nie wspierają tablic liniowych (listy) i asocjacyjnych. Dlatego należy unikać ich używania w plikach */etc/profile* i *$HOME/.profile*.

Za pomocą poniższych plików możesz zdefiniować obiekty, które nie będą eksportowane, na przykład aliasy potrzebne wszystkim użytkownikom. Rozważ taki oto przykład:

```
alias l "ls -l"
/etc/bash.bashrc /etc/bashrc
```

W poniższych plikach można przechowywać ustawienia osobiste. Możliwość taka przydaje się w przypadku ustawiania ścieżek, które muszą być dziedziczone przez inne instancje powłoki Bash. Możemy na przykład umieścić w nich następujące ustawienia:

```
CLASSPATH=$CLASSPATH:$HOME/MyJavaProject; export CLASSPATH
$HOME/.bash_login $HOME/.bash_profile $HOME/.profile
```

Jeśli dostępne są pliki *.bash_login* lub *.bash_profile*, plik *.profile* nie zostanie wczytany. Plik *.profile* może zostać odczytany przez inne powłoki.

W plikach tych można przechowywać własne wartości, które muszą zostać zdefiniowane podczas tworzenia każdej powłoki. Jeśli chcesz udostępnić aliasy i funkcje sesji terminalu X11, skorzystaj z poniższych plików:

```
$HOME/.bashrc, /etc/bash.bashrc
```

Wyeksportowane zmienne i funkcje są propagowane do powłok zależnych. Nie dotyczy to jednak aliasów. Aby skorzystać z aliasów w skrypcie powłoki, do zmiennej `BASH_ENV` należy przypisać pliki *.bashrc* lub *.profile*, w których są zdefiniowane aliasy.

Poniższy plik zostanie wczytany, gdy użytkownik wyloguje się z sesji:

```
$HOME/.bash_logout
```

Na przykład jeśli użytkownik zaloguje się zdalnie, powinien oczyścić ekran podczas wylogowywania się:

```
$> cat ~/.bash_logout
# Wyczyść ekran po zdalnym logowaniu/wylogowaniu.
clear
```

Skorowidz

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

V

W

Z

Ż

Skrypty powłoki — najlepsze wsparcie admina!

Obecnie systemy uniksowe wyposaża się w intuicyjne GUI, a pojawiające się dystrybucje Linuksa stają się coraz łatwiejsze w obsłudze i administracji. Wciąż jednak jednym z najważniejszych narzędzi administratora i użytkownika systemu uniksowego pozostaje opracowana w zeszłym stuleciu powłoka Bourne'a, czyli *bash.* Umiejętność pisania i używania skryptów powłoki jest bezcenna: można w ten sposób automatyzować monotonne zadania, monitorować stan i działanie systemu, włączając w to identyfikację problematycznych procesów. Łatwiejsze też stają się operacje na plikach, optymalizacje wydajności czy dostosowanie systemu do specyficznych potrzeb.

Niniejsza książka przyda się zarówno użytkownikom, jak i administratorom systemów uniksowych. Znalazły się tu receptury dotyczące prostych czynności, takich jak wyszukiwanie plików, a także złożonych zadań administracyjnych, w tym monitorowania i dostosowywania systemu, obsługi sieci, bezpieczeństwa i korzystania z chmury. Nie zabrakło receptur ułatwiających rozwiązywanie złożonych problemów, takich jak tworzenie kopii zapasowych, kontrola wersji, śledzenie pakietów oraz korzystanie z kontenerów, maszyn wirtualnych i chmury. Zawarto tu również receptury przydatne dla programistów, którzy nauczą się analizować aplikacje systemowe i korzystać z takich narzędzi, jak git i fossil.

W tej książce:

- pisanie i debugowanie skryptów oraz konfigurowanie powłoki
- sterowanie pracą skryptu i praca z plikami
- rozwiązywanie problemów z aplikacjami internetowymi
- kopie zapasowe, monitorowanie systemu i inne zadania administracyjne
- analiza sieci i rozwiązywanie problemów z bezpieczeństwem

Clif Flynt ma kilkadziesiąt lat doświadczenia jako programista i administrator systemów Linux i Unix. Jego aplikacje były wykorzystywane m.in. przez marynarkę wojenną USA. W wolnym czasie gra na gitarze i bawi się z kotami swojej żony.
Sarath Lakshman jest programistą w firmie Zynga w Indiach. Entuzjasta systemu GNU/Linux, jest szeroko znany jako jeden z twórców dystrybucji SLYNUX. Jego pasją są skalowalne systemy rozproszone.
Shantanu Tushar jest programistą i uczestniczy w projektach związanych z oprogramowaniem KDE. Pracuje też nad projektami: Calligra, Gluon i Plasma.

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 978-83-283-4031-2
9 788328 340312
INFORMATYKA W NAJLEPSZYM WYDANIU
Cena: 89,00 zł
Packt>