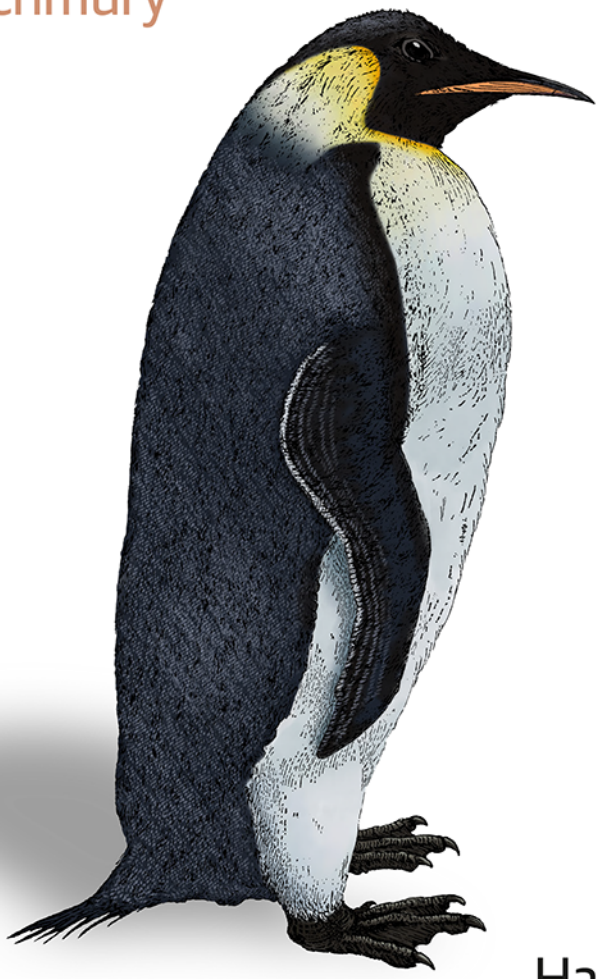
O'REILLY®

Nowoczesny Linux

Przewodnik dla użytkownika natywnej chmury



Michael Hausenblas

Helion

Tytuł oryginału: Learning Modern Linux: A Handbook for the Cloud Native Practitioner

Tłumaczenie: Robert Górczyński

ISBN: 978-83-283-9831-3

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 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/nowlin
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 3.
Powłoki i skrypty

W tym rozdziale skoncentruję się na pracy z Linuksem w terminalu, czyli za pomocą powłoki, która udostępnia interfejs wiersza poleceń (ang. *command-line interface*, CLI). Niezwykle ważna jest umiejętność efektywnego używania powłoki podczas wykonywania codziennych zadań i to będzie motywem przewodnim tego rozdziału.

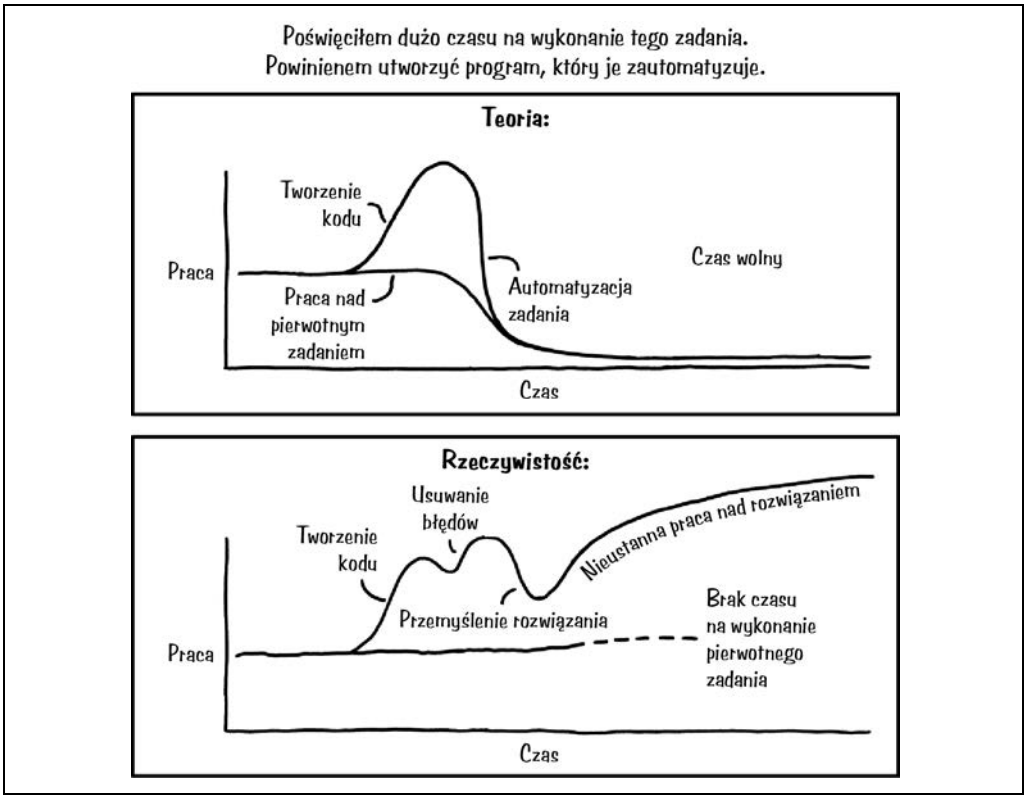
Rozpocznę od wyjaśnienia terminologii, a następnie przejdę do przystępnego i spójnego wprowadzenia do podstaw pracy z powłoką. Dalej pokrótce przedstawię nowoczesne i przyjazne użytkownikowi powłoki, takie jak Fish. Omówię także konfigurację i najczęściej wykonywane zadania w powłoce. Następnie przejdę do tematu efektywnej pracy z interfejsem wiersza poleceń z użyciem terminala, który umożliwi pracę z wieloma sesjami, zarówno lokalnymi, jak i zdalnymi. Natomiast w ostatniej części rozdziału skoncentruję się na automatyzacji zadań w powłoce za pomocą skryptów. Przedstawię przy tym najlepsze praktyki w zakresie tworzenia skryptów w sposób bezpieczny i przenośny, a także pokażę, jak lintować i testować skrypty.

Z perspektywy CLI mamy dwa podstawowe sposoby pracy z Linuksem. Pierwszy, ręczny, polega na tym, że użytkownik siedzi przed terminalem, interaktywnie wydaje polecenia i sprawdza wygenerowane dane wyjściowe. Ten sposób pracy sprawdza się w przypadku większości zadań wykonywanych na co dzień w powłoce, takich jak m.in.:

- wyświetlanie zawartości katalogów i wyszukiwanie plików lub określonych danych w plikach;
- kopiowanie plików między katalogami lub zdalnymi komputerami;
- odczytywanie wiadomości e-mail lub artykułów, a także wysyłanie komunikatów z poziomu terminala.

W dalszej części rozdziału wyjaśnię, jak można wygodnie i efektywnie pracować z wieloma sesjami powłoki jednocześnie.

Drugi sposób pracy z Linuksem to zautomatyzowane przetwarzanie umieszczonych w specjalnego rodzaju pliku serii poleceń, które powłoka interpretuje i po kolei wykonuje. Ten tryb jest zwykle określany mianem *skryptów powłoki* lub po prostu *skryptów*. Zazwyczaj chcemy użyć skryptu zamiast ręcznie powtarzać wykonywanie pewnych zadań. Ponadto skrypty to podstawa dla wielu konfiguracji i instalacji systemów. Skrypty okazują się niezwykle wygodne. Jednakże mogą okazać się również niebezpieczne, jeśli nie będą używane z zachowaniem ostrożności. Dlatego gdy pomyślisz o utworzeniu skryptu, przypomnij sobie komiks XKCD pokazany na rysunku 3.1.



*Rysunek 3.1. Komiks XKCD dotyczący automatyzacji (*https://xkcd.com/1319/*). Autor: Randall Munroe, licencja CC BY-NC 2.5*

Gorąco zachęcam, aby wykorzystać dostępne pod ręką środowisko Linuksa i od razu wypróbowywać przykłady omawiane w tym rozdziale. Czy jesteś w stanie podjąć jakąś interakcję? Jeżeli tak, zacznę od wyjaśnienia terminologii i podstaw pracy z powłoką.

Podstawy

Zanim przejdę do różnych opcji i konfiguracji, chciałbym skoncentrować się na podstawowych pojęciach, takich jak *terminal* i *powłoka*. W tym podrozdziale omówię stosowaną terminologię i pokażę, jak można w powłoce wykonywać codzienne zadania. Ponadto omówię nowoczesne polecenia i zaprezentuję je w działaniu.

Terminal

Rozpocznę od terminala, emulatora terminala lub miękkiego terminala — te wszystkie określenia odwołują się do tego samego: *terminala*, czyli programu zapewniającego tekstowy interfejs użytkownika. Terminal pozwala odczytywać znaki z klawiatury, a następnie wyświetlać je na ekranie. Wiele lat temu były to urządzenia zintegrowane (klawiatura i ekran były ze sobą połączone), natomiast obecnie terminal to po prostu aplikacja.

Poza prostymi danymi wejściowymi i wyjściowymi w postaci tekstu terminal obsługuje również tzw. *sekwencje sterujące* (*https://en.wikipedia.org/wiki/ANSI_escape_code*), nazywane również *kodami sterującymi*, przeznaczone do pracy z kursorem i ekranem oraz potencjalnie do wyświetlania treści w kolorze. Na przykład naciśnięcie klawiszy *Ctrl+H* spowoduje wywołanie sygnału backspace i tym samym usunięcie znaku znajdującego się po lewej stronie kursora.

Zmienna środowiskowa `TERM` podaje nazwę używanego terminala, a jego konfiguracja jest wyświetlana po wydaniu polecenia `infocmp` (dane wyjściowe zostały skrócone):

```
$ infocmp ❶
#       Reconstructed via infocmp from file: /lib/terminfo/s/screen-256color
screen-256color|GNU Screen with 256 colors,
        am, km, mir, msgr, xenl,
        colors#0x100, cols#80, it#8, lines#24, pairs#0x10000,
        acsc=++\,\,--..OO``aaffgghhiijjkkllmmnnooppqqrrssttuuvvwwxxyyzz{{||}}~~,
        bel=^G, blink=\E[5m, bold=\E[1m, cbt=\E[Z, civis=\E[?25l,
        clear=\E[H\E[J, cnorm=\E[34h\E[?25h, cr=\r,
        ...
```

❶ Dane wyjściowe polecenia `infocmp` nie są łatwe do rozszyfrowania. Jeżeli chcesz dowiedzieć się więcej na temat jego możliwości, zapoznaj się z informacjami zamieszczonymi w systemowym podręczniku użytkownika dla polecenia `terminfo` (*https://www.man7.org/linux/man-pages/man5/terminfo.5.html*). Na przykład w moim systemie terminal obsługuje 80 kolumn (`cols#80`) i 24 wiersze (`lines#24`) dla danych wyjściowych oraz 256 kolorów (`colors#0x100` w postaci wartości szesnastkowej).

Przykłady terminali obejmują nie tylko `xterm`, `rxvt` i Gnome Terminal, ale również nowe generacje wykorzystujące procesor graficzny, takie jak Alacritty (*https://github.com/alacritty/alacritty*), kitty (*https://sw.kovidgoyal.net/kitty/*) i warp (*https://www.warp.dev/*).

Do tematu terminali jeszcze powrócę w dalszej części rozdziału.

Powłoka

Powłoka to program uruchomiony w terminalu i działający jako interpreter poleceń. Zapewnia obsługę danych wejściowych i wyjściowych za pomocą strumieni, obsługuje zmienne, ma wbudowane polecenia, radzi sobie z wykonywaniem poleceń i ich stanem oraz zwykle umożliwia pracę w trybie zarówno interaktywnym, jak i za pomocą skryptów (więcej informacji na ten temat znajdziesz w dalszej części rozdziału).

Formalnie powłoka jest zdefiniowana w poleceniu `sh` (*https://www.man7.org/linux/man-pages/man1/sh.1p.html*) i często można natknąć się na określenie powłoka POSIX (*https://drewdevault.com/2018/02/05/Introduction-to-POSIX-shell.html*), co nabierze większego znaczenia w kontekście skryptów i przenośności.

Początkowo mieliśmy do dyspozycji powłokę Bourne `sh` o nazwie pochodzącej od nazwiska jej autora. Obecnie zwykle zastępuje ją powłoka bash (*https://en.wikipedia.org/wiki/Bash_%28Unix_shell%29*) — jej nazwa to akronim od gry słów względem nazwy pierwotnej powłoki, „Bourne Again Shell” — która jest powszechnie używana jako powłoka domyślna.

Jeżeli chcesz sprawdzić, z której powłoki korzystasz, wydaj polecenie `file -h /bin/sh`, a jeśli jego wykonanie zakończy się niepowodzeniem, spróbuj wydać polecenie `echo $0` lub `echo $SHELL`.

W tym podrozdziale przyjąłem założenie, że o ile nie wskażę innej, jest używana powłoka bash.

Istnieje znacznie więcej implementacji `sh`, a także inne warianty, takie jak powłoka Korn (`ksh`) i powłoka C (`csh`), które obecnie nie są zbyt często używane. W dalszej części rozdziału przedstawię wybrane spośród nowoczesnych zamienników powłoki `bash`.

Omawianie podstaw pracy w powłoce rozpoczynam od prezentacji dwóch absolutnie niezbędnych funkcjonalności: strumieni i zmiennych.

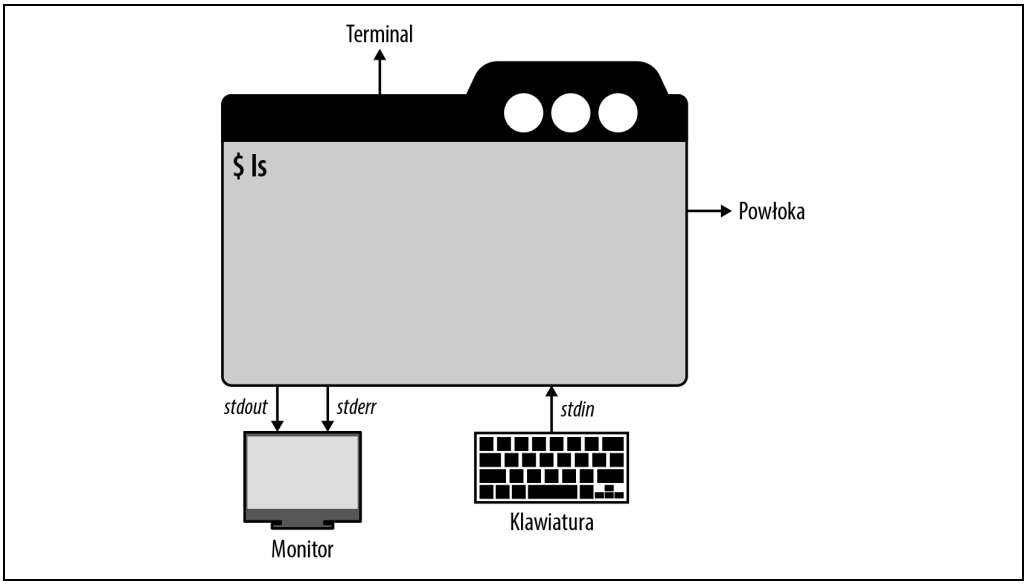
Strumienie

Rozpocznę od strumieni danych wejściowych i strumieni danych wyjściowych, czyli strumieni wejścia-wyjścia. W jaki sposób można dostarczyć programowi dane wejściowe? W jaki sposób kontrolować, gdzie mają się pojawić dane wyjściowe programu, np. w terminalu czy w pliku?

Przede wszystkim powłoka udostępnia każdemu procesowi trzy domyślne deskryptory pliku (ang. *file descriptor*) dla wejścia i wyjścia:

- `stdin` (deskryptor pliku 0),
- `stdout` (deskryptor pliku 1),
- `stderr` (deskryptor pliku 2).

Te deskryptory pliku, pokazane na rysunku 3.2, są domyślnie połączone, odpowiednio, z ekranem i klawiaturą. Innymi słowy o ile nie określisz inaczej, polecenie wprowadzone w powłoce będzie pobierało dane wejściowe (`stdin`) z klawiatury i przekazywało dane wyjściowe (`stdout`) na ekran.



Rysunek 3.2. Domyślne strumienie wejścia-wyjścia w powłoce

W kolejnym fragmencie kodu pokazałem ten domyślny sposób działania:

```
$ cat
To są przykładowe dane wejściowe wprowadzone za pomocą klawiatury i wyświetlone na ekranie^C
```

W tym przykładzie polecenia `cat` widać domyślny sposób działania strumieni. Zwróć uwagę na naciśnięcie klawiszy *Ctrl+C* (ta kombinacja została pokazana jako `^C`) w celu zakończenia działania polecenia.

Jeżeli nie chcesz korzystać z dostarczanych przez powłokę rozwiązań domyślnych — np. nie chcesz przekazywać strumienia `stderr` na ekran i zamiast tego wolisz go przekierować do pliku — wówczas możesz skorzystać z możliwości przekierowania strumieni (*http://teaching.idallen.com/cst8207/12w/notes/270_redirection.txt*).

Do przekierowania strumienia danych wyjściowych procesu służy składnia `$FD>` i `<$FD`, gdzie `$FD` oznacza deskryptor pliku — np. `2>` oznacza przekierowanie strumienia `stderr`. Warto wiedzieć, że składnia `1>` i `>` oznacza to samo, ponieważ strumieniem domyślnym jest `stdout`. Jeżeli chcesz przekierować strumienie `stdout` i `stderr`, musisz użyć składni `&>`. W celu pozbycia się strumienia skorzystaj z urządzenia */dev/null*.

Zobacz teraz, jak to działa w kontekście konkretnego przykładu, w którym za pomocą polecenia `curl` pobierana jest z internetu treść HTML:

```
$ curl https://example.com &> /dev/null ❶

$ curl https://example.com > /tmp/content.txt 2> /tmp/curl-status ❷
$ head -3 /tmp/content.txt
<!doctype html>
<html>
<head>
$ cat /tmp/curl-status
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  1256  100  1256    0     0   3187      0 --:--:-- --:--:-- --:--:--  3195

$ cat > /tmp/interactive-input.txt ❸

$ tr < /tmp/curl-status [A-Z] [a-z] ❹
  % total    % received % xferd  average speed   time    time     time  current
                                 dload  upload   total   spent    left  speed
100  256  100  1256    0     0   3187      0 --:--:-- --:--:-- --:--:--  3195
```

❶ Odrzucenie wszystkich danych wyjściowych przez przekierowanie strumieni `stdout` i `stderr` do */dev/null*.

❷ Przekierowanie do oddzielnych plików strumieni danych wyjściowych i informacji o stanie.

❸ Interaktywne pobieranie danych wejściowych i zapisanie ich w pliku. Naciśnięcie klawiszy *Ctrl+D* kończy przechwytywanie danych wejściowych i powoduje ich zapisanie.

❹ Zmiana na małe wszystkich liter za pomocą polecenia `tr` odczytującego dane wejściowe ze strumienia `stdin`.

Powłoka zwykle potrafi obsłużyć także pewną liczbę znaków specjalnych, np.:

ampersand (&)
: Umieszczenie tego znaku na końcu polecenia powoduje, że zostanie ono wykonane w tle (do tego tematu jeszcze powrócę w dalszej części rozdziału).

ukośnik (\)
: Pozwala kontynuować polecenie w następnym wierszu, co może poprawić czytelność długich poleceń.

potok (|)
: Powoduje połączenie strumienia `stdout` jednego procesu ze strumieniem `stdin` następnego procesu. Umożliwia to przekazywanie danych bez konieczności ich przechowywania w plikach tymczasowych.

Potok i filozofia systemu UNIX

Wprawdzie na pierwszy rzut oka potok (*http://web.cse.ohio-state.edu/~mamrak.1/CIS762/pipes_lab_notes.html*) może nie wydawać się zbyt ekscytującym rozwiązaniem, ale jego możliwości są naprawdę duże. Miałem kiedyś okazję porozmawiać z wynalazcą potoku, Dougiem McIlroyem. Napisałem artykuł *Revisiting the Unix philosophy in 2018* (*https://opensource.com/article/18/11/revisiting-unix-philosophy-2018*), w którym przedstawiłem porównanie systemu UNIX i mikrousług. Ktoś skomentował ten artykuł i ten komentarz skłonił Douga do wysłania mi wiadomości e-mail (całkiem niespodziewanie i musiałem zweryfikować nadawcę, aby uwierzyć, że był nim Doug), w którym wyjaśnił kilka kwestii.

Pora pokazać w działaniu przedstawioną teorię. Załóżmy, że zadanie polega na ustaleniu, ile wierszy zawiera plik HTML. W tym celu plik można pobrać za pomocą polecenia `curl`, a następnie jego zawartość przekazać do narzędzia `wc`:

```
$ curl https://example.com 2> /dev/null | \ ❶
  wc -l ❷
46
```

❶ Polecenie `curl` umożliwia pobranie treści ze wskazanego adresu URL. Tutaj odrzucane są wszelkie informacje o stanie przekazywane przez strumień `stderr`. (W praktyce należy użyć opcji `-s` polecenia `curl`, choć tutaj chciałbym pokazać, jak w praktyce zastosować zdobytą wcześniej wiedzę).

❷ Zawartość strumienia `stdout` polecenia `curl` jest przekazywana do strumienia `stdin` polecenia `wc`, którego opcja `-l` powoduje zliczanie wierszy danych wejściowych.

Po przedstawieniu podstaw dotyczących poleceń, strumieni i przekierowania mogę przejść do następnej kluczowej funkcjonalności powłoki, czyli do obsługi zmiennych.

Zmienne

Pojęciem często pojawiającym się w kontekście powłoki jest *zmienna*. Gdy nie chcesz lub nie możesz na stałe zapisać wartości, rozwiązaniem jest użycie zmiennej do przechowywania i modyfikowania tej wartości. Oto kilka sytuacji, w których zmienne znajdują zastosowanie:

- Zachodzi potrzeba obsługi elementów konfiguracyjnych udostępnianych przez Linuksa — np. katalogi, w których powłoka szuka plików wykonywalnych, zostały wymienione w zmiennej `$PATH`. Jest to rodzaj interfejsu, w którym zmienna może odczytywać i zapisywać dane.
- Chcesz w sposób interaktywny pobrać wartość od użytkownika, np. w kontekście skryptu.
- Chcesz skrócić dane wejściowe przez jednokrotne zdefiniowanie długiej wartości — np. adres URL dla API HTTP. Ten przypadek z grubsza odpowiada wartości *stałej* w języku programowania, ponieważ po zadeklarowaniu zmiennej nie chcemy zmieniać jej wartości.

Rozróżniamy dwa następujące rodzaje zmiennych:

Zmienne środowiskowe

Służą dla potrzeb ustawień na poziomie powłoki i można je wyświetlić za pomocą polecenia `env`.

Zmienne powłoki

Są poprawne w kontekście bieżącego wykonania, można je wyświetlić za pomocą polecenia `set`. Zmienne powłoki nie są dziedziczone przez podprocesy.

W powłoce bash polecenie `export` pozwala utworzyć zmienną środowiskową. Gdy chcesz uzyskać dostęp do wartości zmiennej, musisz przed jej nazwą podać znak $. Do usunięcia zmiennej służy polecenie `unset`.

To była całkiem spora ilość informacji. W kolejnym fragmencie kodu możesz zobaczyć zmienne w działaniu (wykorzystałem powłokę bash):

```
$ set MY_VAR=42 ❶
$ set | grep MY_VAR ❷
_=MY_VAR=42

$ export MY_GLOBAL_VAR="eksperymenty ze zmiennymi" ❸

$ set | grep 'MY_*' ❹
MY_GLOBAL_VAR='eksperymenty ze zmiennymi'
_=MY_VAR=42

$ env | grep 'MY_*' ❺
MY_GLOBAL_VAR=eksperymenty ze zmiennymi

$ bash ❻
$ echo $MY_GLOBAL_VAR ❼
eksperymenty ze zmiennymi

$ set | grep 'MY_*' ❽
MY_GLOBAL_VAR='eksperymenty ze zmiennymi'

$ exit ❾
$ unset $MY_VAR
$ set | grep 'MY_*'
MY_GLOBAL_VAR='eksperymenty ze zmiennymi'
```

❶ Utworzenie zmiennej o nazwie `MY_VAR` i przypisanie jej wartości 42.

❷ Wyświetlenie zmiennych powłoki i odfiltrowanie zmiennej `MY_VAR`. Warto zwrócić uwagę na zapis `_=`, oznaczający, że zmienna nie została wyeksportowana.

❸ Utworzenie nowej zmiennej środowiskowej o nazwie MY_GLOBAL_VAR.

❹ Wyświetlenie zmiennych powłoki i odfiltrowanie tych, które mają nazwy rozpoczynające się od MY_. Zgodnie z oczekiwaniami zostały wyświetlone obie utworzone wcześniej zmienne.

❺ Wyświetlenie zmiennych środowiskowych. Na liście znajduje się również zmienna MY_GLOBAL_VAR.

❻ Utworzenie nowej sesji powłoki — czyli procesu potomnego bieżącej sesji powłoki — która nie będzie dziedziczyła zmiennej MY_VAR.

❼ Uzyskanie dostępu do zmiennej środowiskowej MY_GLOBAL_VAR.

❽ Wyświetlenie zmiennych środowiskowych. Dostępna jest tylko zmienna MY_GLOBAL_VAR, ponieważ obecnie używany jest proces potomny.

❾ Opuszczenie procesu potomnego, usunięcie zmiennej MY_VAR i ponowne wyświetlenie zmiennych powłoki. Zgodnie z oczekiwaniami zmiennej MY_VAR już nie ma.

W tabeli 3.1 wymieniłem najczęściej stosowane zmienne powłoki i środowiskowe. Można je spotkać praktycznie wszędzie, więc ich poznanie i zrozumienie jest ważne. Wartości pozostałych zmiennych można sprawdzać za pomocą polecenia `echo $XXX`, gdzie XXX to nazwa zmiennej.

Tabela 3.1. Najczęściej używane zmienne powłoki i środowiskowe

Zmienna	Typ	Opis
EDITOR	środowiskowa	Ścieżka dostępu do programu domyślnie używanego do edytowania plików
HOME	POSIX	Ścieżka dostępu do katalogu domowego bieżącego użytkownika
HOSTNAME	powłoki bash	Nazwa bieżącego hosta
IFS	POSIX	Lista znaków pozwalających na rozdzielanie pól. Jest używana, gdy skrypt powłoki rozdziela słowa
PATH	POSIX	Zawiera listę katalogów, w których powłoka będzie szukała programów wykonywalnych (pliki binarne lub skrypty)
PS1	środowiskowa	Używany ciąg tekstowy podstawowego znaku zachęty powłoki
PWD	środowiskowa	Pełna ścieżka dostępu do bieżącego katalogu roboczego
OLDPWD	powłoki bash	Pełna ścieżka dostępu do katalogu bieżącego przed wykonaniem polecenia cd
RANDOM	powłoki bash	Losowo wybrana liczba całkowita z przedziału od 0 do 32767
SHELL	środowiskowa	Zawiera nazwę aktualnie używanej powłoki
TERM	środowiskowa	Zawiera nazwę aktualnie używanego terminala
UID	środowiskowa	Unikatowa wartość identyfikatora bieżącego użytkownika (jest to liczba całkowita)
USER	środowiskowa	Nazwa bieżącego użytkownika
_	powłoki bash	Ostatni argument polecenia, które poprzednio zostało wykonane na pierwszym planie
?	powłoki bash	Kod stanu wyjścia (więcej informacji na ten temat znajdziesz w podpunkcie „Kod stanu wyjścia”)
$	powłoki bash	Identyfikator bieżącego procesu (jest to liczba całkowita)
0	powłoki bash	Nazwa bieżącego procesu

Na stronie *https://www.gnu.org/software/bash/manual/html_node/Bash-Variables.html* znajdziesz pełną listę zmiennych związanych z powłoką bash. Zmienne wymienione w tabeli 3.1 okażą się użyteczne ponownie w kontekście tworzenia skryptów powłoki, co omówię w dalszej części rozdziału.

Kod stanu wyjścia

Powłoka informuje o zakończeniu wykonywania polecenia za pomocą tzw. *kodu stanu*. Ogólnie rzecz biorąc, oczekuje się, że po zakończeniu działania polecenie Linuksa zwróci kod stanu. Działanie procesu może zakończyć się w zwykły sposób (wszystko w porządku) lub w nagły (coś poszło nie tak). Kod stanu `0` oznacza, że wykonywanie polecenia zakończyło się sukcesem i bez żadnych błędów. Z kolei wartość niezerowa z przedziału od 1 do 255 wskazuje na niepowodzenie. Aby wyświetlić kod stanu wyjścia, należy wydać polecenie `echo $?`.

Trzeba zachować ostrożność podczas obsługi kodu stanu w potoku, ponieważ powłoka udostępnia jedynie ostatni dostępny kod stanu wyjścia. Z tym ograniczeniem można sobie poradzić za pomocą zmiennej `$PIPESTATUS` (*https://www.shellscript.sh/tips/pipestatus/*).

Polecenia wbudowane

Powłoka jest dostarczana razem z pewną liczbą wbudowanych poleceń, np. `yes`, `echo`, `cat` i `read` (w zależności od dystrybucji Linuksa niektóre z tych poleceń mogą nie być wbudowane w powłokę i będą znajdowały się w katalogu */usr/bin*). Do wyświetlenia listy poleceń wbudowanych w powłokę można wykorzystać polecenie `help`. Jednak pamiętaj, że wszystko pozostałe to programy zewnętrzne dla powłoki i można je znaleźć w katalogu */usr/bin* (polecenia użytkownika) lub */usr/sbin* (polecenia administratora).

Jak sprawdzić, gdzie znajduje się plik wykonywalny? Można to zrobić na dwa sposoby:

```
$ which ls
/usr/bin/ls

$ type ls
ls is aliased to `ls --color=auto'
```

Jeden z korektorów merytorycznych książki słusznie wskazał, że `which` to program zewnętrzny nienależący do standardu POSIX i tym samym nie zawsze będzie dostępny. Ponadto zasugerował użycie składni *`polecenie -v`* zamiast `which` w celu pobrania ścieżki dostępu do programu i/lub aliasu bądź funkcji powłoki. Więcej informacji na ten temat znajdziesz pod adresem *https://github.com/koalaman/shellcheck/wiki/SC2230*.

Kontrola zadań

Funkcjonalnością obsługiwaną przez większość powłok jest *kontrola zadań* (*https://www.digitalocean.com/community/tutorials/how-to-use-bash-s-job-control-to-manage-foreground-and-background-processes*). Domyślnie po wydaniu polecenia przejmuje ono kontrolę nad ekranem i klawiaturą, co zwykle określa się mianem *działania na pierwszym planie*. Co zrobić w sytuacji, gdy

nie chcesz uruchamiać czegoś w trybie interaktywnym lub jeśli w ogóle nie są używane dane wejściowe pochodzące ze strumienia `stdin`? Witaj w świecie kontroli zadań i zadań wykonywanych w tle — aby uruchomić proces działający w tle, należy umieścić znak `&` na końcu albo przenieść proces do działania w tle przez naciśnięcie klawiszy *Ctrl+Z*.

Tutaj zobaczysz to w działaniu:

```
$ watch -n 5 "ls" & ❶

$ jobs ❷
Job     Group   CPU     State   Command
1       3021    0%      stopped watch -n 5 "ls" &

$ fg ❸
Every 5.0s: ls                                          Sat Aug 28 11:34:32 2021
Dockerfile
app.yaml
example.json
main.go
script.sh
test
```

❶ Umieszczenie znaku `&` na końcu polecenia powoduje uruchomienie danego polecenia w tle.

❷ Wyświetlenie wszystkich zadań.

❸ Wydanie polecenia `fg` powoduje przeniesienie procesu do działania na pierwszym planie. Jeżeli chcesz opuścić polecenie `watch`, naciśnij klawisze *Ctrl+C*.

Gdy proces ma kontynuować działanie w tle, nawet po zamknięciu powłoki, wówczas trzeba go uruchomić za pomocą polecenia `nohup`. Co więcej, w przypadku procesu nieuruchomionego za pomocą `nohup` można już po fakcie użyć polecenia `disown` i osiągnąć ten sam efekt. Natomiast jeśli chcesz się pozbyć działającego procesu, możesz skorzystać z polecenia `kill` razem z odpowiednim poziomem siły (zajrzyj do punktu „Sygnały” w rozdziale 9.).

Zamiast kontroli zadań zachęcam do użycia multipleksera terminala, co dokładnie wyjaśnię w podrozdziale „Multiplekser terminala”. Program takiego typu zajmuje się większością najczęściej spotykanych przypadków (zamknięcie powłoki, wiele uruchomionych programów wymagających koordynacji itd.), a także umożliwia pracę ze zdalnymi systemami.

W następnym punkcie omówię nowoczesne zamienniki dla najczęściej używanych poleceń podstawowych, które istnieją od zawsze.

Nowoczesne polecenia

Istnieje dość duża liczba poleceń, które są używane nieustannie w trakcie codziennej pracy. To m.in. polecenia przeznaczone do poruszania się po strukturze katalogów (`cd`), wyświetlania zawartości katalogu (`ls`), wyszukiwania plików (`find`) oraz wyświetlania ich treści (`cat`, `less`). Biorąc pod uwagę to, że te polecenia są używane bardzo często, chcesz z nich korzystać w najbardziej efektywny sposób i każdy wpisywany znak ma znaczenie.

Dla najczęściej używanych poleceń istnieją nowoczesne odpowiedniki. Część z nich to bezpośrednie odpowiedniki, podczas gdy inne zapewniają rozszerzoną funkcjonalność. Wszystkie oferują rozsądne wartości domyślne dla najczęściej wykonywanych operacji oraz bogate dane wyjściowe, które przeważnie są łatwiejsze do zrozumienia i wymagają wpisania mniejszej liczby znaków, aby wykonać dane zadanie. To zmniejsza tarcia podczas pracy w powłoce oraz powoduje, że staje się ona przyjemniejsza i płynniejsza. Jeżeli chcesz dowiedzieć się więcej o nowoczesnych narzędziach, zajrzyj do dodatku B. W tym kontekście trzeba zachować ostrożność, zwłaszcza podczas korzystania z takiego rozwiązania w środowisku firmowym. Nie mam żadnego udziału w tworzeniu tych narzędzi i polecam je, ponieważ okazały się użyteczne w mojej pracy. Podczas instalowania i stosowania dowolnego z tych narzędzi należy pamiętać, aby użyć wersji przeznaczonej dla danej dystrybucji systemu Linux.

Wyświetlanie zawartości katalogu za pomocą polecenia exa

Gdy chcesz poznać zawartość katalogu, używasz polecenia `ls` bądź jednego z jego wariantów z parametrami. Na przykład w powłoce bash zwykle korzystam z aliasu `l` będącego odpowiednikiem polecenia `ls -GAhltr`. Istnieje jednak lepszy sposób: `exa` (*https://the.exa.website/*), czyli nowoczesny zamiennik polecenia `ls`, utworzony w języku Rust oraz zapewniający obsługę systemu kontroli wersji Git i wyświetlania zawartości w postaci drzewa. W tym kontekście jak sądzisz, jakie jest najczęściej używane polecenie po wyświetleniu zawartości katalogu? Z mojego doświadczenia wynika, że jest to polecenie przeznaczone do usunięcia zawartości ekranu. Do tego celu użytkownicy często korzystają z polecenia `clear`. To oznacza konieczność wpisania pięciu znaków i naciśnięcia klawisza *Enter*. Ten sam efekt można uzyskać znacznie szybciej — wystarczy nacisnąć klawisze *Ctrl+L*.

Wyświetlanie zawartości pliku za pomocą polecenia bat

Załóżmy, że po wyświetleniu zawartości katalogu chcesz sprawdzić zawartość jednego ze znajdujących się w nim plików. Do tego celu można użyć polecenia `cat`, prawda? Istnieje jednak lepsze rozwiązanie, które warto wypróbować: `bat` (*https://github.com/sharkdp/bat*). Polecenie `bat`, jak pokazałem na rysunku 3.3, umożliwia kolorowanie składni, wyświetla znaki niedrukowalne, obsługuje system kontroli wersji Git oraz ma zintegrowaną możliwość stronicowania treści (do wyświetlania zawartości pliku, która jest dłuższa niż jednorazowo mieszcząca się na ekranie).

Wyszukiwanie treści w pliku za pomocą polecenia rg

Do wyszukiwania czegoś w treści pliku tradycyjnie używa się polecenia `grep`. Dla tego polecenia również istnieje nowoczesny odpowiednik, `rg` (*https://github.com/BurntSushi/ripgrep*), który charakteryzuje się dużą wydajnością działania i potężnymi możliwościami.

W kolejnym fragmencie kodu pokazałem porównanie polecenia `rg` z poleceniami `find` i `grep`. Celem jest tutaj znalezienie plików w formacie YAML zawierających ciąg tekstowy `sample`:

```
$ find . -type f -name "*.yaml" -exec grep "sample" '{}' \; -print ❶
      app: sample
        app: sample
./app.yaml
```

```
$ rg -t "yaml" sample ❷
app.yaml
9:      app: sample
14:        app: sample
```

❶ Użycie poleceń `find` i `grep` w celu znalezienia ciągu tekstowego w plikach YAML.

❷ Użycie polecenia `rg` do tego samego zadania.

Jeżeli porównasz te polecenia i wygenerowane przez nie wyniki, zobaczysz, że `rg` nie tylko jest łatwiejsze w użyciu, ale również jego dane wyjściowe zawierają więcej informacji (zapewniają kontekst, w tym przypadku jest to numer wiersza).

Rysunek 3.3. Polecenie bat wyświetlające plik zawierający kod w języku Go (na górze) i kod w formacie YAML (na dole)

Przetwarzanie danych JSON za pomocą jq

W tym punkcie omówię polecenie dodatkowe, `jq`. To nie jest rzeczywisty zamiennik dla istniejącego polecenia, a raczej specjalizowane narzędzie dla formatu JSON, czyli popularnego formatu danych tekstowych. Znajduje ono zastosowanie w API HTTP oraz w plikach konfiguracyjnych.

Dlatego w celu pobierania określonych wartości należy korzystać z polecenia `jq` (*https://stedolan.github.io/jq/*) zamiast z `awk` i `sed`. Na przykład dzięki użyciu generatora JSON (*https://json-generator.com/*) do przygotowania losowo wybranych danych otrzymałem plik w formacie JSON o nazwie *example.json* i wielkości 2,4 kB, o zawartości podobnej do tej (pokazałem jedynie początek pliku):

```
[
  {
    "_id": "612297a64a057a3fa3a56fcf",
    "latitude": -25.750679,
    "longitude": 130.044327,
    "friends": [
      {
        "id": 0,
        "name": "Tara Holland"
      },
      {
        "id": 1,
        "name": "Giles Glover"
      },
      {
        "id": 2,
        "name": "Pennington Shannon"
      }
    ],
    "favoriteFruit": "strawberry"
  },
...
```

Załóżmy, że zadanie polega na znalezieniu wszystkich „najbliższych" przyjaciół — tzn. elementu 0 w tablicy `friends` — spośród osób, których ulubionym owocem jest truskawka (ang. *strawberry*). W przypadku polecenia `jq` powinno mieć ono następującą postać:

```
$ jq 'select(.[].favoriteFruit=="strawberry") | .[].friends[0].name' example.json
"Tara Holland"
"Christy Mullins"
"Snider Thornton"
"Jana Clay"
"Wilma King"
```

To było nawet całkiem zabawne, prawda? Jeżeli chcesz dowiedzieć się nieco więcej o nowoczesnych poleceniach i innych kandydatach do zastąpienia klasycznych poleceń, zajrzyj do repozytorium nowoczesnego systemu UNIX (*https://github.com/ibraheemdev/modern-unix*), w którym znajdziesz pewne sugestie. Skoncentruję się teraz na najczęściej wykonywanych zadaniach, ale innych niż poruszanie się po hierarchii plików i wyświetlanie ich zawartości. Wyjaśnię także, jak można je wykonywać.

Najczęściej wykonywane zadania

Istnieją zadania, które są wykonywane bardzo często, stąd różne sztuczki pozwalające przyśpieszyć pracę w powłoce. Omówię teraz wybrane z tych zadań i wyjaśnię, jak można je wykonywać znacznie efektywniej.

Skracanie często używanych poleceń

Podstawową zasadą w interfejsach jest to, że najczęściej używane polecenia powinny wymagać jak najmniejszego wysiłku ze strony użytkownika — tzw. wystarczające ma być wpisanie minimalnej liczby znaków. Teraz pokażę zastosowanie tej idei w powłoce. Na przykład setki razy dziennie przeglądam zmiany wprowadzone w repozytoriach. Dlatego zamiast wydawać polecenie `git diff --color-moved` wpisuję tylko pojedynczą literę `d`. W zależności od powłoki istnieją różne sposoby na otrzymanie takiego efektu. W powłoce bash nosi to nazwę *aliasu* (*https://ss64.com/bash/alias.html*), natomiast w powłoce Fish (nieco więcej na jej temat dowiesz się z dalszej części rozdziału) dostępne są tzw. *skróty* (*https://fishshell.com/docs/current/cmds/abbr.html*).

Nawigacja

Po wpisaniu polecenia w powłoce istnieje wiele operacji, które możemy chcieć przeprowadzić, takich jak przejście do określonego miejsca w wierszu (np. przeniesienie kursora na początek wiersza) lub modyfikacja wiersza (np. usunięcie wszystkiego, co znajduje się po lewej stronie kursora). W tabeli 3.2 wymieniłem najczęściej stosowane skróty powłoki.

Tabela 3.2. Skróty stosowane podczas nawigacji w powłoce i edytowania poleceń powłoki

Opis	Skrót	Uwagi
Przeniesienie kursora na początek wiersza	*Ctrl+A*	-
Przeniesienie kursora na koniec wiersza	*Ctrl+E*	-
Przeniesienie kursora do przodu o jeden znak	*Ctrl+F*	-
Przeniesienie kursora do tyłu o jeden znak	*Ctrl+B*	-
Przeniesienie kursora do przodu o jedno słowo	*Alt+F*	Tylko z lewym klawiszem *Alt*
Przeniesienie kursora do tyłu o jedno słowo	*Alt+B*	-
Usunięcie bieżącego znaku	*Ctrl+D*	-
Usunięcie znaku po lewej stronie kursora	*Ctrl+H*	-
Usunięcie słowa po lewej stronie kursora	*Ctrl+W*	-
Usunięcie wszystkiego po prawej stronie kursora	*Ctrl+K*	-
Usunięcie wszystkiego po lewej stronie kursora	*Ctrl+U*	-
Usunięcie zawartości ekranu	*Ctrl+L*	-
Przerwanie polecenia	*Ctrl+C*	-
Cofnięcie operacji	*Ctrl+_*	Tylko powłoka bash
Historia wyszukiwania	*Ctrl+R*	Wybrane powłoki
Przerwanie wyszukiwania	*Ctrl+G*	Wybrane powłoki

Nie wszystkie skróty wymienione w tabeli 3.2 będą dostępne w każdej powłoce. Ponadto pewne akcje, np. zarządzanie historią, mogą być zaimplementowane odmiennie w poszczególnych powłokach. Warto również dodać, że te skróty bazują na skrótach używanych w edytorze Emacs. Jeżeli preferujesz edytor vi, w pliku *.bashrc* możesz umieścić polecenie `set -o vi` i tym samym edycję poleceń przeprowadzać z użyciem skrótów stosowanych w edytorze vi. Tabelę 3.2 wykorzystaj jako punkt wyjścia, sprawdź, które z tych skrótów są obsługiwane w Twojej powłoce, i zobacz, jak można je skonfigurować do własnych potrzeb.

Zarządzanie zawartością pliku

Aby dodać pojedynczy wiersz tekstu, nie zawsze trzeba uruchamiać edytor, taki jak `vi`. Czasami to jest nawet niemożliwe — np. w kontekście tworzenia skryptu powłoki (więcej informacji na ten temat znajdziesz w dalszej części rozdziału).

W jaki sposób można przeprowadzać operacje na danych tekstowych? Zapoznaj się z kilkoma przykładami:

```
$ echo "Wiersz pierwszy" > /tmp/something ❶

$ cat /tmp/something ❷
Wiersz pierwszy

$ echo "Wiersz drugi" >> /tmp/something && \ ❸
  cat /tmp/something
Wiersz pierwszy
Wiersz drugi

$ sed 's/line/LINE/' /tmp/something ❹
Wiersz pierwszy
Wiersz drugi

$ cat << 'EOF' > /tmp/another ❺
Wiersz pierwszy
Wiersz drugi
Wiersz trzeci
EOF

$ diff -y /tmp/something /tmp/another ❻
Wiersz pierwszy                                         Wiersz pierwszy
Wiersz drugi                                            Wiersz drugi
                                                      > Wiersz trzeci
```

❶ Utworzenie pliku przez przekierowanie danych wyjściowych polecenia `echo`.

❷ Wyświetlenie zawartości pliku.

❸ Dołączenie wiersza tekstu do pliku za pomocą operatora `>>`, a następnie wyświetlenie zawartości pliku.

❹ Zastąpienie zawartości pliku za pomocą polecenia `sed` i przekazanie danych wyjściowych do strumienia `stdout`.

❺ Utworzenie pliku za pomocą składni *here document* (*https://tldp.org/LDP/abs/html/here-docs.html*).

❻ Wyświetlenie różnic między utworzonymi plikami.

W ten sposób omówiłem podstawowe techniki modyfikowania zawartości pliku. W następnym podpunkcie zajmę się zaawansowanymi rozwiązaniami dotyczącymi wyświetlania zawartości pliku.

Wyświetlanie zawartości dużych plików

W przypadku dużych plików, czyli składających się z większej liczby wierszy niż możliwa do wyświetlenia na ekranie, można skorzystać z narzędzi stronicowania, takich jak w poleceniu `less` lub `bat` (to drugie ma wbudowany mechanizm stronicowania). Dzięki stronicowaniu program

dzieli dane wyjściowe na strony mieszczące się na ekranie oraz wyświetla interfejs powalający poruszać się po stronach (przejście do następnej lub poprzedniej strony itd.).

Innym sposobem pomocnym w pracy z dużymi plikami jest wyświetlenie tylko pewnego fragmentu pliku, np. kilku pierwszych wierszy. Dwa użyteczne polecenia przeznaczone do tego celu to `head` i `tail`.

Oto przykład pokazujący, jak wyświetlić początek pliku:

```
$ for i in {1..100} ; do echo $i >> /tmp/longfile ; done ❶

$ head -5 /tmp/longfile ❷
1
2
3
4
5
```

❶ Utworzenie większego pliku (w tym przypadku składa się ze stu wierszy).

❷ Wyświetlenie tylko pierwszych pięciu wierszy nowego pliku.

Jeżeli chcesz na bieżąco wyświetlać nowe wiersze nieustannie zwiększającego się pliku, możesz wydać następujące polecenie:

```
$ sudo tail -f /var/log/Xorg.0.log ❶
[ 36065.898] (II) event14 - ALPS01:00 0911:5288 Mouse: device is a pointer
[ 36065.900] (II) event15 - ALPS01:00 0911:5288 Touchpad: device is a touchpad
[ 36065.901] (II) event4  - Intel HID events: is tagged by udev as: Keyboard
[ 36065.901] (II) event4  - Intel HID events: device is a keyboard
...
```

❶ Wyświetlanie końcówki pliku za pomocą polecenia `tail` i jego opcji `-f`, oznaczającej automatyczne uaktualnienie danych wyjściowych.

W kolejnym podpunkcie wyjaśnię, jak w powłoce pracować z wartościami daty i godziny.

Obsługa daty i godziny

Polecenie `date` może zapewnić niezwykle użyteczny sposób na generowanie unikatowych nazw plików. Pozawala wygenerować datę w różnych formatach, w tym znacznik czasu systemu UNIX (*https://en.wikipedia.org/wiki/Unix_time*), a także konwertować wartości daty między różnymi formatami.

```
$ date +%s ❶
1629582883

$ date -d @1629742883 '+%m/%d/%Y:%H:%M:%S' ❷
08/21/2021:21:54:43
```

❶ Utworzenie znacznika czasu systemu UNIX.

❷ Konwersja znacznika czasu systemu UNIX na datę w formacie czytelnym dla człowieka.

Czas epoki systemu UNIX

Czas epoki systemu UNIX (lub prościej czas uniksowy) to liczba sekund, które upłynęły od północy 1 stycznia 1970 roku. W takim przypadku każdy dzień składa się z 86400 sekund.

Jeżeli pracujesz z oprogramowaniem przechowującym czas uniksowy jako 32-bitową liczbę całkowitą ze znakiem, zachowaj ostrożność, ponieważ dla dnia 19 stycznia 2038 roku ta wartość będzie większa niż obsługiwana przez 32-bitowy typ liczb całkowitych. Jest to tzw. problem roku 2038 (*https://en.wikipedia.org/wiki/Year_2038_problem*).

Dla bardziej zaawansowanych operacji można wykorzystać dostępne w internecie konwertery, np. *https://www.epochconverter.com/*, które zapewniają dokładność do mikrosekund i milisekund.

Na tym kończę krótkie wprowadzenie do podstaw pracy z powłoką. W tym momencie doskonale już wiesz, czym są terminal i powłoka, a także jak z nich korzystać podczas wykonywania prostych zadań, takich jak poruszanie się po systemie plików, wyszukiwanie plików itd. Tematem następnego podrozdziału są powłoki przyjazne użytkownikowi.

Powłoki przyjazne użytkownikowi

Wprawdzie powłoka bash (*https://en.wikipedia.org/wiki/Bash_%28Unix_shell%29*) jest prawdopodobnie najczęściej używana, ale niekoniecznie zalicza się do najbardziej przyjaznych użytkownikowi. Istnieje od późnych lat osiemdziesiątych ubiegłego stulecia i to czasami widać. Dostępnych jest wiele nowoczesnych i przyjaznych powłok, gorąco zachęcam do zapoznania się z nimi i używania ich zamiast powłoki bash.

Zacznę od dokładnego omówienia jednej z tego rodzaju powłok, o nazwie Fish, a następnie pokrótce przedstawię inne. Dzięki temu poznasz szeroką gamę dostępnych powłok, z których możesz wybrać odpowiednią dla siebie. Na końcu podrozdziału zamieszczę krótkie podsumowanie i wnioski.

Powłoka Fish

Powłoka Fish (*https://fishshell.com/*) jest przedstawiana jako sprytnie działająca i przyjazna użytkownikowi. W tym punkcie zacznę od omówienia podstaw pracy z nią, a następnie przejdę do zagadnień związanych z konfiguracją powłoki Fish.

Podstaw pracy z powłoką Fish

W zakresie danych wejściowych podczas codziennych zadań nie zauważysz zbyt dużych różnic względem powłoki bash. Większość skrótów wymienionych w tabeli 3.2 będzie działała. Jednak istnieją dwa obszary, na których powłoka Fish różni się od powłoki bash i jest od niej znacznie wygodniejsza.

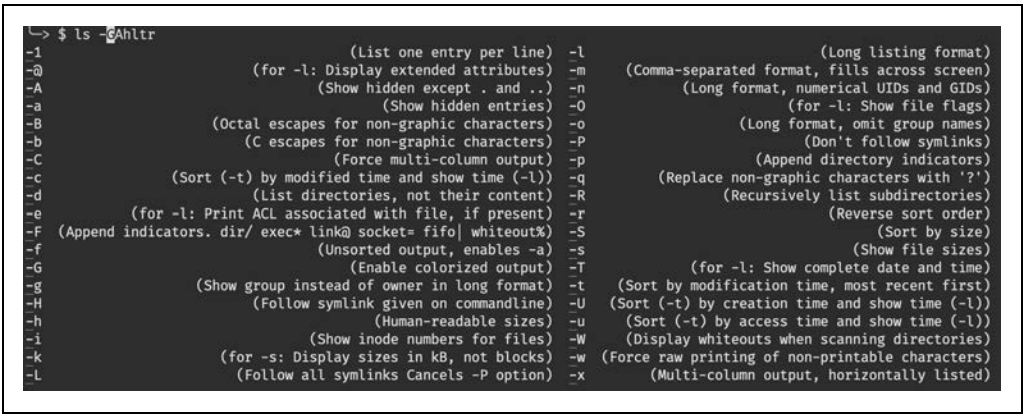
Brak wyraźnego zarządzania historią

Wystarczy zacząć pisać, a wyświetlą się dopasowane wcześniejsze polecenia. Klawisze strzałek w górę i w dół pozwalają wybrać odpowiednie (zobacz rysunek 3.4).

Rysunek 3.4. Obsługa historii w powłoce Fish w działaniu

Dla wielu poleceń są dostępne automatyczne sugestie

To możesz zobaczyć na rysunku 3.5. Po naciśnięciu klawisza *Tab* powłoka spróbuje uzupełnić polecenie, argument lub ścieżkę dostępu, podpowiadając wizualnie, np. przez kolorowanie danych wyjściowych na czerwono, gdy polecenie nie zostało rozpoznane.



Rysunek 3.5. Obsługa automatycznych sugestii w działaniu

W tabeli 3.3 wymieniłem wybrane z najczęściej używanych poleceń powłoki Fish. W tym kontekście warto zwrócić uwagę na obsługę zmiennych środowiskowych.

Tabela 3.3. Wybrane z najczęściej używanych poleceń powłoki Fish

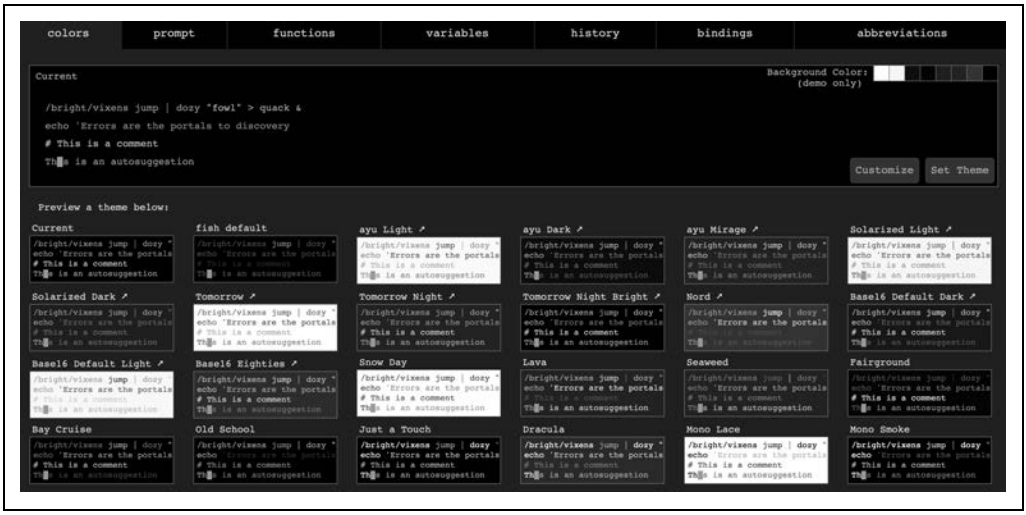
Zadanie	Polecenie
Wyeksportowanie zmiennej środowiskowej `KEY` z wartością `VAL`	`set -x KEY VAL`
Usunięcie zmiennej środowiskowej `KEY`	`set -e KEY`
Użycie zmiennej środowiskowej `KEY` osadzonej w poleceniu cmd	`env KEY=VAL cmd`
Zmiana na 1 długości ścieżki dostępu	`set -g fish_prompt_pwd_dir_length 1`
Zarządzanie skrótami	`abbr`
Zarządzanie funkcjami	`functions i funcd`

W przeciwieństwie do innych powłok kod stanu wyjścia ostatniego polecenia powłoka Fish przechowuje w zmiennej `$status` zamiast `$?`.

Jeżeli migrujesz z powłoki bash, zapoznaj się z dokumentem Fish FAQ (*https://fishshell.com/docs/current/faq.html*), w którym omówiono wiele ważnych kwestii związanych z migracją.

Konfiguracja

W celu skonfigurowania powłoki Fish (*https://fishshell.com/docs/current/index.html#configuration*) należy wydać polecenie `fish_config` (w zależności od dystrybucji konieczne może być dodanie podpolecenia `browse`). To spowoduje uruchomienie przez powłokę Fish serwera dostępnego pod adresem *http://localhost:8000* i automatyczne otworzenie w przeglądarce WWW eleganckiego interfejsu użytkownika (zobacz rysunek 3.6). Wyświetlona strona pozwala przeglądać i modyfikować ustawienia.



Rysunek 3.6. Konfiguracja powłoki Fish za pomocą przeglądarki WWW

Aby podczas nawigacji za pomocą powłoki przejść między trybami skrótów klawiaturowych `vi` i Emacs (domyślny), należy wydać polecenie `fish_vi_key_bindings` (tryb `vi`) lub `fish_default_key_bindings` (tryb Emacs, domyślny). Wszystkie zmiany zostają wprowadzone natychmiast we wszystkich aktywnych sesjach powłoki.

Spójrz teraz na moją konfigurację powłoki Fish. Ta konfiguracja jest dość krótka, zawartość mojego pliku *config.fish* przedstawia się następująco:

```
set -x FZF_DEFAULT_OPTS "-m --bind='ctrl-o:execute(nvim {})+abort'"
set -x FZF_DEFAULT_COMMAND 'rg --files'
set -x EDITOR nvim
set -x KUBE_EDITOR nvim
set -ga fish_user_paths /usr/local/bin
```

Z kolei znak zachęty zdefiniowałem w pliku *fish_prompt.fish* za pomocą następującego kodu:

```
function fish_prompt
    set -l retc red
    test $status = 0; and set retc blue

    set -q __fish_git_prompt_showupstream
    or set -g __fish_git_prompt_showupstream auto

    function _nim_prompt_wrapper
        set retc $argv[1]
        set field_name $argv[2]
        set field_value $argv[3]

        set_color normal
        set_color $retc
        echo -n '—'
        set_color -o blue
        echo -n '['
        set_color normal
        test -n $field_name
        and echo -n $field_name:
        set_color $retc
        echo -n $field_value
        set_color -o blue
        echo -n ']'
    end

    set_color $retc
    echo -n '┬—'
    set_color -o blue
    echo -n [
    set_color normal
    set_color c07933
    echo -n (prompt_pwd)
    set_color -o blue
    echo -n ']'

     # Zmienne środowiskowe
    set -q VIRTUAL_ENV_DISABLE_PROMPT
    or set -g VIRTUAL_ENV_DISABLE_PROMPT true
    set -q VIRTUAL_ENV
    and _nim_prompt_wrapper $retc V (basename "$VIRTUAL_ENV")

    # Git
    set prompt_git (fish_git_prompt | string trim -c ' ()')
    test -n "$prompt_git"
    and _nim_prompt_wrapper $retc G $prompt_git

    # Nowy wiersz
    echo

    # Zadania wykonywane w tle
    set_color normal
    for job in (jobs)
        set_color $retc
        echo -n '| '
        set_color brown
        echo $job
```

```
    end
    set_color blue
    echo -n '└─> '
        set_color -o blue
    echo -n '$ '
    set_color normal
end
```

Ta definicja powoduje utworzenie znaku zachęty pokazanego na rysunku 3.7. Zwróć uwagę na różnicę między katalogiem zawierającym repozytorium Git a katalogiem bez takiego repozytorium. Informacje dotyczące Gita ułatwiają pracę. Zauważ także, że godzina jest wyświetlona po prawej stronie okna powłoki.

Rysunek 3.7. Znak zachęty w mojej powłoce Fish

Zdefiniowane przeze mnie skróty — w innych powłokach są to tzw. *aliasy* — przedstawiają się następująco:

```
$ abbr
abbr -a -U -- :q exit
abbr -a -U -- cat bat
abbr -a -U -- d 'git diff --color-moved'
abbr -a -U -- g git
abbr -a -U -- grep 'grep --color=auto'
abbr -a -U -- k kubectl
abbr -a -U -- l 'exa --long --all --git'
abbr -a -U -- ll 'ls -GAhltr'
abbr -a -U -- m make
abbr -a -U -- p 'git push'
abbr -a -U -- pu 'git pull'
abbr -a -U -- s 'git status'
abbr -a -U -- stat 'stat -x'
abbr -a -U -- vi nvim
abbr -a -U -- wget 'wget -c'
```

Aby dodać nowy skrót, należy użyć polecenia `abbr --add`. Skróty okazują się użyteczne w przypadku prostych poleceń niepobierających argumentów. Co można zrobić w sytuacji, gdy zachodzi potrzeba skrócenia znacznie bardziej skomplikowanej konstrukcji? Załóżmy, że chcesz skrócić sekwencję zawierającą polecenie `git` i pobierającą argument. Skorzystaj z funkcji w powłoce Fish.

W kolejnym fragmencie kodu zamieściłem przykładową funkcję zdefiniowaną w pliku *c.fish*. Polecenie `functions` pozwala wyświetlić dostępne zdefiniowane funkcje, polecenie `function` zaś umożliwia utworzenie nowej funkcji. W omawianym przykładzie jest to funkcja `c`, której kod przedstawia się następująco:

```
function c
    git add --all
    git commit -m "$argv"
end
```

Na tym kończę omawianie powłoki Fish. Przedstawiłem podstawy pracy z nią i jej konfiguracji. W kolejnych punktach omówię inne nowoczesne powłoki.

Powłoka Z

Powłoka Z (*https://zsh.sourceforge.io/Doc/*), w skrócie `zsh` (ang. *z-shell*), to pochodna powłoki Bourne'a. Zawiera oferujący potężne możliwości system uzupełnień (*http://bewatermyfriend.org/p/2012/003/*) oraz obsługuje rozbudowane motywy. Dzięki dodatkowi Oh My Zsh (*https://ohmyz.sh/*) tę powłokę można skonfigurować i stosować w sposób pokazany podczas omawiania powłoki Fish, a jednocześnie zachować zgodność z powłoką bash.

zsh używa pięciu plików startowych, wymienionych w kolejnym fragmencie kodu. (Jeżeli zmienna `$ZDOTDIR` nie została zdefiniowana, zamiast niej zostanie użyta zmienna `$HOME`):

```
$ZDOTDIR/.zshenv ❶
$ZDOTDIR/.zprofile ❷
$ZDOTDIR/.zshrc ❸
$ZDOTDIR/.zlogin ❹
$ZDOTDIR/.zlogout ❺
```

❶ Ten plik jest używany w trakcie każdego wywołania powłoki. Powinien zawierać polecenia definiujące ścieżkę wyszukiwania i inne ważne zmienne środowiskowe. Natomiast nie powinien zawierać poleceń, które generują dane wyjściowe lub przyjmują założenie, że powłoka jest dołączona do `tty`.

❷ Ten plik jest alternatywą *.zlogin* przygotowaną dla fanów powłoki `ksh` (obie powłoki nie są przystosowane do jednoczesnego używania). Plik jest podobny do *.zlogin*, ale przetwarzany jeszcze przed plikiem *.zshrc*.

❸ Ten plik jest używany w powłoce interaktywnej. Powinien zawierać polecenia definiujące aliasy, funkcje, opcje, skróty klawiszowe itd.

❹ Ten plik jest używany w powłoce logowania. Powinien zawierać polecenia przeznaczone do wykonywania jedynie w powłoce logowania. Zwróć uwagę na to, że plik *.zlogin* nie jest miejscem dla definicji aliasów, opcji, ustawień zmiennych środowiskowych itd.

❺ Ten plik jest używany, gdy powłoka kończy działanie.

Więcej informacji dotyczących wtyczek zsh znajdziesz w repozytorium *awesome-zsh-plugins* w serwisie GitHub (*https://github.com/unixorn/awesome-zsh-plugins*). Jeżeli chcesz poznać powłokę `zsh`, przeczytaj dokument *An Introduction to the Z Shell* (*https://www.ee.ryerson.ca/guides/zsh-intro.pdf*) przygotowany przez Paula Falstada i Basa de Bakkera.

Inne nowoczesne powłoki

Poza Fish i zsh istnieje jeszcze wiele innych interesujących powłok, które jednak nie zawsze są zgodne z powłoką bash. Przyglądając się im, zadaj sobie pytanie, pod jakim kątem została opracowana dana powłoka (praca interaktywna czy skrypty) oraz czy wokół niej istnieje aktywna społeczność.

Oto kilka przykładów nowoczesnych powłok dla systemu Linux, na które się natknąłem i którym warto się przyjrzeć nieco dokładniej:

Oil (*https://www.oilshell.org/*)

Przeznaczona dla użytkowników Pythona i JavaScriptu. Innymi słowy mniej skoncentrowana na użyciu interaktywnym, a bardziej na wykorzystaniu jej w skryptach.

murex (*https://murex.rocks/*)

Powłoka zgodna ze standardem POSIX oferująca wiele interesujących funkcjonalności, np. zintegrowany framework testowania, typowane potoki i programowanie sterowane testami.

Nushell (*https://www.nushell.sh/*)

Eksperymentalny nowy paradygmat oferujący m.in. tabelaryczne dane wyjściowe i potężny język zapytań. Więcej informacji na temat tej powłoki znajdziesz pod adresem *https://www.nushell.sh/book/*.

PowerShell (*https://github.com/powershell/powershell*)

Dostępna dla wielu platform powłoka, która na początku była jedynie wersją powłoki Windows PowerShell. Oferuje inną semantykę i interakcje niż znane z powłok zgodnych ze standardem POSIX.

Opracowano naprawdę wiele powłok. Przejrzyj je i sprawdź, która Ci najlepiej odpowiada. Spróbuj wykroczyć poza powłokę bash i zoptymalizuj wybraną pod kątem własnych potrzeb.

Która powłoka jest dla mnie najlepsza?

W tym momencie każda nowoczesna powłoka — inna niż bash — wydaje się dobrym wyborem z perspektywy użytkownika. Płynne automatyczne uzupełnianie, łatwa konfiguracja oraz sprytnie działające środowiska nie są luksusem w 2022 roku. Biorąc pod uwagę ilość czasu zwykle spędzanego w powłoce, warto wypróbować niektóre z nich i wybrać tę, którą lubisz najbardziej. Ja używam powłoki Fish, natomiast wielu moim współpracownikom w zupełności wystarcza `zsh`.

Pewne kwestie mogą utrudniać odejście od powłoki bash — oto kilka z nich:

- Pracujesz ze zdalnymi systemami i/lub nie możesz zainstalować własnej powłoki.
- Pozostajesz przy powłoce bash w celu zachowania zgodności i/lub nie chcesz się uczyć od zera pracy w nowym narzędziu. Pozbycie się pewnych nawyków może być naprawdę trudne.
- W niemalże wszystkich poleceniach (niejawnie) przyjmuje się założenie, że jest używana powłoka bash. Na przykład polecenie typu `export FOO=BAR` jest ściśle związane z powłoką bash.

Okazuje się, że dla większości użytkowników te problemy są w zasadzie nieistotne. Wprawdzie tymczasowo być może musisz korzystać z powłoki bash w zdalnym systemie, ale przez większość czasu pracujesz w środowisku, nad którym masz kontrolę. Poznanie powłoki nie jest prostym zadaniem, ale na dłuższą metę ten wysiłek się opłaci.

W następnym podrozdziale skoncentruję się na kolejnym sposobie, w jaki można poprawić produktywność podczas pracy w powłoce: omówię multiplekser.

Multiplekser terminala

O terminalu wspomniałem już na początku rozdziału. Teraz nieco bardziej zagłębię się w ten temat i wyjaśnię, jak poprawić sposób pracy z terminalem, bazując na koncepcji, która jest prosta, a zarazem ma potężne możliwości: multipleksowanie.

Pomyśl o tym w następujący sposób: zwykle pracujesz nad różnymi rzeczami, które mogą być ze sobą grupowane. Na przykład możesz pracować nad projektem open source, postem bloga lub dokumentacją, uzyskiwać zdalny dostęp do serwera, korzystać z API HTTP do przetestowania czegoś itd. Każde z tych zadań będzie wymagało co najmniej jednego okna terminala. Bardzo często chcesz lub musisz wykonywać potencjalnie niezależne zadania jednocześnie w dwóch oknach.

- Używasz polecenia `watch` w celu nieustannego sprawdzania zawartości katalogu i jednocześnie edytowania pliku.
- Uruchamiasz proces serwera (np. serwer WWW lub aplikacji) i pozwalasz mu działać w tle (zobacz punkt „Kontrola zadań”), aby monitorować dzienniki zdarzeń.
- Chcesz edytować plik za pomocą edytora vi i jednocześnie używasz polecenia `git` do sprawdzenia stanu repozytorium i przekazania wprowadzonych zmian.
- Masz maszynę wirtualną uruchomioną w chmurze publicznej, chcesz się do niej dostać za pomocą SSH i jednocześnie zachować możliwość lokalnego zarządzania plikami.

Wszystkie te przykłady potraktuj jako powiązane ze sobą logicznie zadania. Zwróć uwagę, że czas ich wykonywania może być zróżnicowany, od krótkiego (kilka minut) do długiego (dni i tygodnie). Grupowanie poszczególnych zadań jest określane mianem *sesji*.

Oto kilka wyzwań pojawiających się, gdy chcesz przeprowadzić grupowanie zadań:

- Koniecznych jest kilka okien. Jednym z rozwiązań jest uruchomienie kilku terminali lub, jeśli pozwala na to interfejs użytkownika, wielu egzemplarzy (kart) terminala.
- Wszystkie okna i ścieżki chcesz pozostawić, nawet jeśli zamkniesz terminal lub połączenie zostanie zakończone przez zdalny system.
- Chcesz mieć możliwość przybliżania i oddalania w celu koncentracji na określonych zadaniach i jednocześnie chcesz zachować podgląd wszystkich sesji oraz możliwość poruszania się między nimi.

Aby umożliwić wykonywanie tych zadań, opracowano ideę pracy z wieloma oknami terminala (oraz sesjami i grupami okien) — innymi słowy następuje multipleksowanie operacji wejścia-wyjścia terminala.

Teraz pokrótce przedstawię pierwotną implementację multipleksera terminala, która jest dostępna w postaci polecenia `screen`. Następnie nieco dokładniej omówię powszechnie używaną implementację o nazwie `tmux` i zakończę podrozdział prezentacją innych opcji w tym zakresie.

screen

`screen` (*https://www.gnu.org/software/screen/*) to wciąż używana pierwotna wersja multipleksera terminala. O ile nie pracujesz w zdalnym środowisku, w którym żadne inne rozwiązanie w tym

zakresie nie jest dostępne i/lub nie możesz zainstalować innego multipleksera, prawdopodobnie nie będziesz korzystać z polecenia `screen`. Przede wszystkim to narzędzie nie jest już aktywnie rozwijane, nie jest zbyt elastyczne oraz brakuje w nim wielu funkcji, które znajdziemy w nowoczesnych multiplekserach terminala.

tmux

`tmux` (*https://github.com/tmux/tmux*) to elastyczny i rozbudowany multiplekser terminala, który można dostosować do własnych potrzeb. Jak pokazałem na rysunku 3.8, mamy trzy podstawowe elementy pozwalające prowadzić interakcję z multiplekserem tmux. W kolejności od najbardziej ogólnego do najbardziej szczegółowego są to:

Sesja

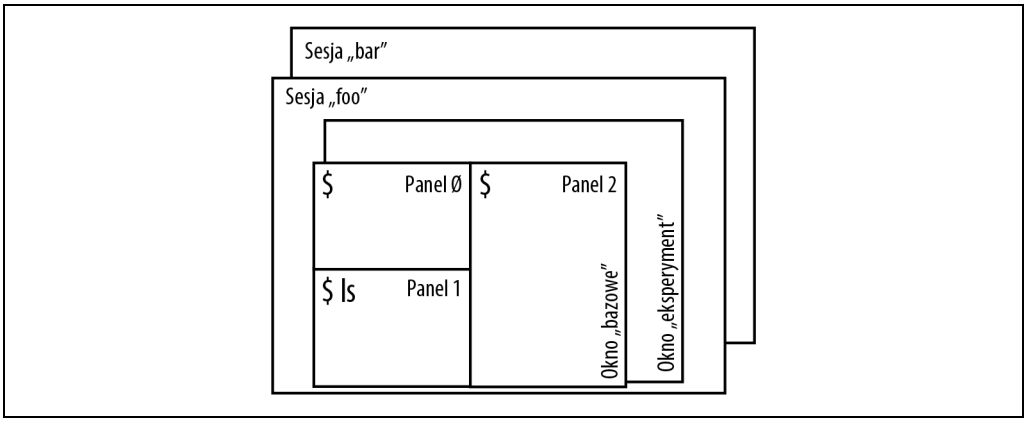
Jednostka logiczna, którą można traktować jako środowisko robocze przeznaczone dla konkretnego zadania, takiego jak „praca nad projektem X” lub „tworzenie posta na blog Y”. Jest to kontener dla wszystkich pozostałych jednostek.

Okno

Okno można potraktować jako kartę w przeglądarce WW. Należy do sesji. Jego użycie jest opcjonalne i często istnieje tylko jedno okno dla danej sesji.

Panel

To prawdziwy koń roboczy, praktycznie pojedynczy działający egzemplarz powłoki. Panel jest częścią okna. Bardzo łatwo można go dzielić poziomo lub pionowo, a także rozwijać i zwijać (mam tutaj na myśli przybliżanie i oddalanie) oraz zamykać wedle potrzeb.



Rysunek 3.8. Elementy tworzące multiplekser terminala tmux: sesje, okna i panele

Podobnie jak polecenie `screen` także `tmux` umożliwia dołączenie do sesji lub pracy w trybie odłączonym. Załóżmy, że praca rozpoczyna się od początku. Zaczynamy więc od uruchomienia sesji o nazwie `test`:

```
$ tmux new -s test
```

W tym przykładzie tmux działa jako serwer, natomiast użytkownik pracuje w powłoce skonfigurowanej w tmux, która działa jako klient. Model klient – serwer pozwala tworzyć sesje, wchodzić do nich, opuszczać je i usuwać, a także korzystać z uruchomionych w nich powłok bez konieczności zastanawiania się nad procesami, które w nich działają lub nie działają.

tmux używa *Ctrl+B* jako domyślnego skrótu klawiszowego, nazywanego także *prefiksem* lub *wyzwalaczem*. Dlatego aby na przykład wyświetlić listę wszystkich okien, należy nacisnąć klawisze *Ctrl+B*, a później *W*. Ewentualnie w celu rozwinięcia bieżącego (aktywnego) panelu należy nacisnąć klawisze *Ctrl+B*, a potem *Z*.

Domyślnym wyzwalaczem w multiplekserze tmux jest *Ctrl+B*. Aby ułatwić sobie pracę, zmapowałem wyzwalacz na nieużywany klawisz i dzięki temu wyzwalacz staje się dostępny po naciśnięciu pojedynczego klawisza. Zacząłem od zmapowania w tmux wyzwalacza na klawisz *Home*, a następnie zmapowałem *Home* na *Caps Lock* przez zmianę w pliku */usr/share/X11/xkb/symbols/pc* mapowania na `key <CAPS> { [ Home ] };`.

Takie podwójne mapowanie było rozwiązaniem, którego potrzebowałem. W zależności od docelowego klawisza lub terminala być może nie trzeba będzie tego robić. Mimo wszystko gorąco zachęcam do mapowania wyzwalacza *Ctrl+B* na nieużywany klawisz, który będzie łatwo dostępny — będziesz bowiem z niego korzystać wielokrotnie w ciągu dnia.

Dowolne polecenie wymienione w tabeli 3.4 można wykorzystać do zarządzania dalszymi sesjami, oknami i panelami. Naciśnięcie klawiszy *Ctrl+B+D* spowoduje odłączenie sesji, co w praktyce oznacza, że multiplekser tmux będzie działał w tle.

Tabela 3.4. Polecenia używane podczas pracy w multiplekserze tmux

Cel	Zadanie	Polecenie
Sesja	Utworzenie nowej sesji	`:new -s NAZWA`
Sesja	Zmiana nazwy sesji	*wyzwalacz*+$
Sesja	Wyświetlenie wszystkich sesji	*wyzwalacz*+s
Sesja	Zamknięcie sesji	*wyzwalacz*
Okno	Utworzenie nowego okna	*wyzwalacz*+c
Okno	Zmiana nazwy okna	*wyzwalacz*+,
Okno	Przejście do okna	*wyzwalacz*+1...9
Okno	Wyświetlenie wszystkich okien	*wyzwalacz*+w
Okno	Zamknięcie okien	*wyzwalacz*+&
Panel	Podział poziomy panelu	*wyzwalacz*+"
Panel	Podział pionowy panelu	*wyzwalacz*+%
Panel	Przełączanie panelu	*wyzwalacz*+z
Panel	Zamknięcie panelu	*wyzwalacz*+x

Po uruchomieniu nowego egzemplarza terminala lub powiedzmy sesji SSH z poziomu zdalnego komputera do Twojego możesz dołączyć do istniejącej sesji. Pokażę to teraz na przykładzie utworzonej wcześniej sesji o nazwie `test`:

```
$ tmux attach -t test ❶
```

❶ Dołączenie do istniejącej sesji o nazwie `test`. Jeżeli chcesz odłączyć się od sesji w poprzednim terminalu, musisz użyć parametru `-d`.

W tabeli 3.4 wymieniłem najczęściej używane polecenia multipleksera `tmux`, pogrupowane według omawianych jednostek, od zasięgu najszerszego (sesji) do najwęższego (panel).

Skoro już wiesz, jak wyglądają podstawy pracy z multiplekserem `tmux`, mogę przejść do omówienia jego konfiguracji i dostosowania do własnych potrzeb. Mój plik konfiguracyjny *.tmux.conf* przedstawia się następująco:

```
unbind C-b ❶
set -g prefix Home
bind Home send-prefix
bind r source-file ~/.tmux.conf \; display "tmux config reloaded :)" ❷
bind \\ split-window -h -c "#{pane_current_path}" ❸
bind - split-window -v -c "#{pane_current_path}"
bind X confirm-before kill-session ❹
set -s escape-time 1 ❺
set-option -g mouse on ❻
set -g default-terminal "screen-256color" ❼
set-option -g status-position top ❽
set -g status-bg colour103
set -g status-fg colour215
set -g status-right-length 120
set -g status-left-length 50
set -g window-status-style fg=colour215
set -g pane-active-border-style fg=colour215
set -g @plugin 'tmux-plugins/tmux-resurrect' ❾
set -g @plugin 'tmux-plugins/tmux-continuum'
set -g @continuum-restore 'on'
run '~/.tmux/plugins/tpm/tpm'
```

❶ Ten wiersz i dwa kolejne powodują zmianę wyzwalacza (będzie nim klawisz *Home*).

❷ Ponowne wczytanie konfiguracji, jak po naciśnięciu klawiszy *wyzwalacz+R*.

❸ Ten wiersz i następny ponownie definiują podział panelu. Katalog bieżący istniejącego panelu pozostanie.

❹ Dodanie skrótów dla nowych i zamykanych sesji.

❺ Żadnych opóźnień.

❻ Włączenie możliwości dokonywania wyboru myszą.

❼ Zdefiniowanie domyślnego trybu terminala na 256-kolorowy.

❽ Ustawienia motywu (następne sześć wierszy).

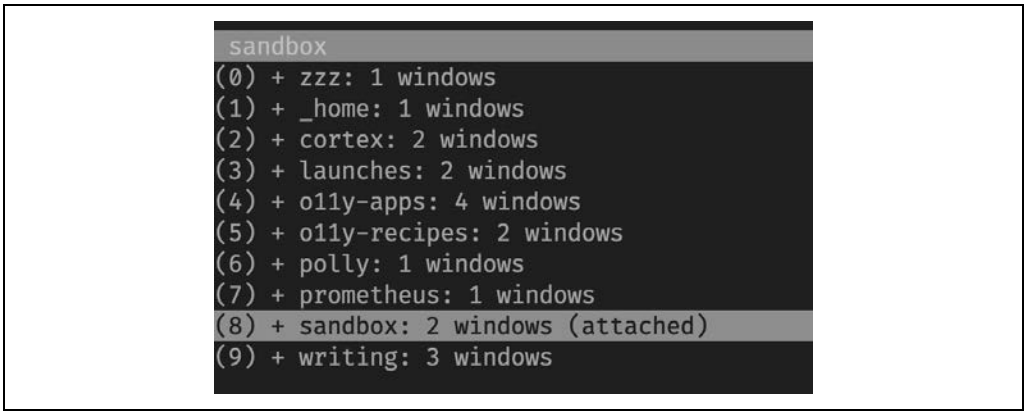
❾ Od tego miejsca do końca: zarządzanie wtyczkami.

Najpierw należy zainstalować `tpm` (*https://github.com/tmux-plugins/tpm*), czyli menedżer wtyczek `tmux`, a następnie zdefiniować skrót *wyzwalacz+I* dla wtyczek. Oto wtyczki, których używam:

`tmux-resurrect` (*https://github.com/tmux-plugins/tmux-resurrect*)
: Umożliwia przywracanie sesji za pomocą *Ctrl+S* (zapisanie) i *Ctrl+R* (przywrócenie).

`tmux-continuum` (*https://github.com/tmux-plugins/tmux-continuum*)
: Automatycznie zapisuje i przywraca sesję (w odstępach 15-minutowych).

Na rysunku 3.9 pokazałem terminal Alacritty z uruchomionym multiplekserem `tmux`. Możesz zobaczyć sesje razem ze skrótami, od 0 do 9.



Rysunek 3.9. Przykład egzemplarza tmux w działaniu — pokazuje dostępne sesje

Wprawdzie `tmux` to bez wątpienia doskonały wybór, ale istnieją jeszcze inne opcje, których użycie warto rozważyć.

Inne multipleksery

Oto wybrane multipleksery, z którymi warto się zapoznać i które warto wypróbować:

tmuxinator (*https://github.com/tmuxinator/tmuxinator*)
: Metanarzędzie umożliwiające zarządzanie sesjami `tmux`.

Byobu (*https://www.byobu.org/*)
: Rodzaj opakowania dla `screen` i `tmux`. To narzędzie jest szczególnie interesujące, jeśli używasz dystrybucji opartej na Ubuntu lub Debianie.

Zellij (*https://zellij.dev/about/*)
: Utworzona w języku programowania Rust tzw. przestrzeń robocza terminala. Wykracza poza możliwości `tmux`, oferuje silnik układu oraz potężny system wtyczek.

dvtm (*https://github.com/martanne/dvtm*)
: Przynosi do terminala koncepcję zarządzania kafelkowanymi oknami. To narzędzie ma potężne możliwości, choć podobnie jak `tmux` nie jest łatwo je opanować.

3mux (*https://github.com/aaronjanse/3mux*)

Utworzony w języku programowania Go prosty multiplekser terminala. Łatwy w użyciu, choć nie ma tak potężnych możliwości jak `tmux`.

Po tym krótkim omówieniu dostępnych multiplekserów warto powiedzieć nieco na temat wyboru jednego z nich.

Połączenie wszystkiego w całość — terminal, mux i powłoka

Jako terminala używam Alacritty. Jest szybki i, co najważniejsze, do jego konfiguracji korzystam z pliku w formacie YAML, który przechowuję w systemie kontroli wersji Git. Dzięki temu jestem w stanie skonfigurować system docelowy w ciągu zaledwie sekund. Ten plik konfiguracyjny nosi nazwę *alacritty.yml* i definiuje wszystkie ustawienia dla terminala, od kolorów, poprzez skróty klawiszowe, po wielkość czcionki.

Większość ustawień ma zastosowanie od razu, podczas gdy inne dopiero po zapisaniu pliku. Jedna z opcji konfiguracyjnych, `shell`, definiuje integrację używanego multipleksera (tutaj `tmux`) ze stosowaną powłoką (tutaj `fish`). W omawianym przykładzie ta konfiguracja przedstawia się następująco:

```
...
shell:
  program: /usr/local/bin/fish
  args:
  - -l
  - -i
  - -c
  - "tmux new-session -A -s zzz"
...
```

W tym fragmencie kodu skonfigurowałem Alacritty do użycia powłoki Fish jako powłoki domyślnej. Ponadto po uruchomieniu terminala następuje automatyczne dołączenie do określonej sesji. W połączeniu z wtyczką `tmux-continuum` to zapewnia mi spokój. Nawet po wyłączeniu komputera i jego ponownym uruchomieniu następuje przywrócenie terminala z wszystkimi sesjami, oknami i panelami w (prawie) dokładnie takim stanie (poza zmiennymi powłoki), w jakim znajdowały się przed awarią systemu.

Który multiplekser wybrać?

Inaczej niż w przypadku powłok dla użytkowników mam konkretne preferencje w zakresie multipleksera terminala: `tmux`. Powody są różnorodne: oprogramowanie to jest dopracowane, stabilne, rozbudowane (wiele dostępnych wtyczek) i elastyczne. Korzysta z niego sporo osób, więc w internecie znajdziesz wiele zasobów na jego temat, a ponadto łatwiej uzyskać pomoc w razie problemów. Wprawdzie inne multipleksery są ekscytujące, ale pozostają względnie nowe bądź też, jak w przypadku `screen`, nie są już aktywnie rozwijane.

Mam nadzieję, że zachęciłem Cię do używania multipleksera w celu poprawienia wrażeń podczas pracy w terminalu, przyśpieszenia wykonywania zadań oraz ogólnie ułatwienia sposobu pracy.

W następnym podrozdziale zajmę się ostatnim tematem, czyli automatyzacją zadań za pomocą skryptów powłoki.

Skrypty

We wcześniejszej części tego rozdziału skoncentrowałem się na ręcznym, interaktywnym użyciu powłoki. Gdy nieustannie wykonujesz w powłoce pewne zadania, należy rozważyć ich automatyzację. Świetnie nadają się do tego skrypty.

W tym podrozdziale przedstawię krótkie wprowadzenie do tworzenia skryptów w powłoce bash. Wybrałem ją z dwóch powodów:

- Większość skryptów została utworzona dla powłoki bash, stąd istnieje wiele przykładów i wskazówek pomocnych w tworzeniu skryptów w tej powłoce.
- Jest wysokie prawdopodobieństwo, że w systemie docelowym znajduje się powłoka bash. To oznacza większą potencjalną bazę użytkowników niż w przypadku stosowania alternatywnej powłoki (potencjalnie mającej znacznie większe możliwości, choć jednocześnie nie tak powszechnie używanej).

Zanim przystąpię do omówienia tematu, chciałbym przedstawić pewien kontekst. Istnieje wiele skryptów powłoki składających się z tysięcy wierszy kodu (*https://github.com/oilshell/oil/wiki/The-Biggest-Shell-Programs-in-the-World).* Nie zachęcam do tworzenia takich skryptów, a wręcz przeciwnie — jeżeli zauważysz, że tworzysz długi skrypt, zadaj sobie pytanie, czy lepszym rozwiązaniem nie będzie użycie do tego zadania innego języka skryptowego, takiego jak Python lub Ruby.

Zrobimy krok wstecz i opracujemy mały, choć jednocześnie użyteczny skrypt powłoki. W trakcie tej pracy pokażę zastosowanie najlepszych praktyk. Załóżmy, że celem jest zautomatyzowanie zadania wyświetlenia na ekranie pojedynczego zdania, w którym dla danego użytkownika serwisu GitHub będzie podana data jego dołączenia do serwisu, a także pełne imię i nazwisko. Wygenerowane przez skrypt dane wyjściowe będą miały następującą postać:

```
XXXX XXXXX dołączył(a) do serwisu GitHub w YYYY roku
```

Jak można takie zadanie automatyzować za pomocą skryptu? Rozpocznę od omówienia podstaw, następnie przejdę do przenośności, a później wyjaśnię sposób działania „logiki biznesowej" tworzonego skryptu.

Podstawy tworzenia skryptów powłoki

Dobrą wiadomością jest to, że używając interaktywnie powłoki, znasz już większość niezbędnych pojęć i technik. Poza zmiennymi, strumieniami, przekierowaniami i najczęściej używanymi poleceniami istnieje jeszcze kilka aspektów, które trzeba poznać w kontekście skryptów powłoki. Omówię je w kolejnych podpunktach.

Zaawansowane typy danych

Wprawdzie powłoki zwykle traktują wszystko jako ciągi tekstowe (jeżeli chcesz wykonywać bardziej złożone zadania, raczej nie należy do tego używać skryptu powłoki), ale zapewniają również obsługę bardziej zaawansowanych typów danych, np. tablic.

Oto przykład tablicy w działaniu:

```
os=('Linux' 'macOS' 'Windows') ❶
echo "${os[0]}" ❷
numberofos="${#os[@]}" ❸
```

❶ Zdefiniowanie tablicy zawierającej trzy elementy.

❷ Uzyskanie dostępu do pierwszego elementu tablicy, co spowoduje wyświetlenie ciągu tekstowego `Linux`.

❸ Pobranie wielkości tablicy, w efekcie wartością zmiennej `numberofos` będzie 3.

Kontrola nad sposobem działania

Kontrola nad sposobem działania pozwala tworzyć odgałęzienia (`if`) lub powtórzenia (`for` i `while`) w skrypcie, aby wykonywanie skryptu zależało od pewnego warunku.

Spójrz na przykłady tego typu kontroli w działaniu:

```
for afile in /tmp/* ; do ❶
  echo "$afile"
done

for i in {1..10}; do ❷
    echo "$i"
done

while true; do
  ...
done ❸
```

❶ Prosta pętla przeprowadza iterację przez katalog i wyświetla nazwy poszczególnych plików.

❷ Pętla zakresu.

❸ Pętla wykonywana w nieskończoność. Aby ją zakończyć, należy nacisnąć klawisze *Ctrl+C*.

Funkcje

Funkcje pozwalają tworzyć bardziej modularne skrypty wielokrotnego użycia. W kodzie funkcję trzeba zdefiniować, zanim zostanie wywołana, ponieważ powłoka interpretuje skrypt od początku do końca.

Oto przykład prostej funkcji powłoki:

```
sayhi() { ❶
    echo "Cześć, $1, mam nadzieję, że masz się dobrze!"
}

sayhi "Michał" ❷
```

❶ Definicja funkcji. Parametry są przekazywane niejawnie za pomocą `$n`.

❷ Wywołanie funkcji, które spowoduje wygenerowanie danych wyjściowych „Cześć, Michał, mam nadzieję, że masz się dobrze!”.

Zaawansowane operacje wejścia-wyjścia

Polecenie `read` pozwala odczytywać dane wejściowe użytkownika ze strumienia `stdin`, który można wykorzystać do pobierania danych wejściowych podczas działania programu. Takie dane można pobrać na przykład z menu opcji. Co więcej, zamiast polecenia `echo` warto rozważyć użycie `printf`, które zapewnia dokładniejszą kontrolę nad danymi wyjściowymi, w tym także nad kolorami. Ponadto `printf` jest bardziej przenośne niż `echo`.

Oto przykład użycia w działaniu zaawansowanych operacji wejścia-wyjścia:

```
read name ❶
printf "Witaj, %s" "$name" ❷
```

❶ Pobranie wartości z danych wejściowych użytkownika.

❷ Wyświetlenie wartości odczytanej w poprzednim kroku.

Istnieją jeszcze inne, bardziej zaawansowane koncepcje, takie jak sygnały i pułapki (*https://linuxconfig.org/how-to-modify-scripts-behavior-on-signals-using-bash-traps*). W tym rozdziale zamierzam jedynie pokrótce omówić temat tworzenia skryptów w powłoce, więc odsyłam Cię na doskonałą stronę *https://devhints.io/bash*, na której znajdziesz wyczerpujące informacje o różnych konstrukcjach. Jeżeli poważnie myślisz o tworzeniu skryptów powłoki, zachęcam do sięgnięcia po książkę *Bash. Receptury* (autorzy: Carl Albing, JP Vossen, Cameron Newham). Znajdziesz w niej wiele doskonałych przykładów, które możesz wykorzystać jako punkt wyjścia dla własnych skryptów.

Tworzenie przenośnych skryptów powłoki bash

W tym punkcie zobaczysz, jak odbywa się tworzenie przenośnych skryptów powłoki bash. Co w tym kontekście oznacza słowo „przenośny” i dlaczego jest ono ważne?

Na początku rozdziału wyjaśniłem znaczenie terminu „POSIX”, więc będę się na tym opierał. Gdy używam słowa „przenośny”, mam na myśli brak zbyt wielu założeń — jawnych lub niejawnych — odnośnie do środowiska, w którym będzie wykonywany skrypt. Jeżeli skrypt jest przenośny, będzie działał w wielu różnych systemach (powłoki, dystrybucje Linuksa itd.).

Pamiętaj, że jeśli nawet ograniczysz się do danego typu powłoki (w omawianym przykładzie jest to bash), nie wszystkie funkcje będą działały tak samo w każdej wersji powłoki. Ostatecznie to i tak sprowadza się do wielu różnych środowisk, w których można przetestować skrypt.

Wykonywanie przenośnych skryptów

W jaki sposób są wykonywane skrypty? Trzeba zacząć od tego, że skrypt to po prostu plik zwykłego tekstu. Rozszerzenie nie ma żadnego znaczenia, choć bardzo często konwencją jest używanie rozszerzenia *.sh*. Dwa ważne aspekty powodują, że plik zwykłego tekstu staje się wykonywalny i może zostać uruchomiony przez powłokę:

- W pierwszym wierszu musi być zadeklarowany interpreter, za pomocą konstrukcji nazywanej *shebang* (*https://linuxize.com/post/bash-shebang/*) lub *hashbang*, zapisanej w postaci `#!` (przeanalizuj pierwszy wiersz omawianego tutaj przykładu).

- Skrypt musi mieć uprawnienia umożliwiające jego wykonanie. W tym celu należy użyć polecenia `chmod +x` lub, jeszcze lepiej, `chmod 750`. Pozostaje to w zgodzie z regułą najmniejszych uprawnień, ponieważ pozwala uruchamiać skrypt jedynie użytkownikowi i powiązanej z nim grupie. Do tego tematu jeszcze powrócę w następnym rozdziale.

Omówiwszy podstawy, przejdę do konkretnego szablonu skryptu powłoki, który można wykorzystać jako punkt wyjścia podczas tworzenia własnych skryptów.

Szablon skryptu

Szablon przenośnego skryptu powłoki, który można wykorzystać jako punkt wyjścia, przedstawia się następująco:

```
#!/usr/bin/env bash ❶
set -o errexit ❷
set -o nounset ❸
set -o pipefail ❹

firstargument="${1:-somedefaultvalue}" ❺

echo "$firstargument"
```

❶ Konstrukcja shebang (*https://en.wikipedia.org/wiki/Shebang_(Unix)*) informuje program, że ten skrypt ma być zinterpretowany przez powłokę bash.

❷ Zdefiniowanie zatrzymania skryptu w przypadku wystąpienia błędów.

❸ Zdefiniowanie, że brak deklaracji zmiennej ma być traktowany jako błąd. (Zmniejsza to niebezpieczeństwo cichego i zakończonego niepowodzeniem działania skryptu).

❹ Zdefiniowanie, że jeśli jedna część potoku ulegnie awarii, cały potok powinien być uznany za uszkodzony. Pomoże to uniknąć cichych niepowodzeń.

❺ Przykład zawierającego wartość domyślną parametru polecenia powłoki.

Ten szablon wykorzystamy później podczas implementacji skryptu pobierającego informacje o użytkowniku serwisu GitHub.

Dobre praktyki

Użyłem wyrażenia „*dobre* praktyki" zamiast „*najlepsze* praktyki", ponieważ należy w danej sytuacji określić, co trzeba zrobić i jak daleko można się posunąć. Istnieje różnica między skryptami tworzonymi dla siebie i skryptami przeznaczonymi dla tysięcy użytkowników. Mimo to ogólne dobre praktyki tworzenia skryptów powłoki przedstawiają się następująco:

Szybka i głośna awaria

Należy unikać cichych awarii. Awaria powinna następować szybko. Pomagają w tym polecenia takie jak `errexit` i `pipefail`. Ponieważ powłoka bash domyślnie ma tendencję do cichych awarii, jak najszybsze jej zgłoszenie zawsze będzie dobrym rozwiązaniem.

Ochrona danych wrażliwych

W skrypcie nie wolno umieszczać żadnych danych wrażliwych, takich jak hasła. Tego rodzaju informacje powinny być dostarczane w trakcie wykonywania skryptu, np. w postaci danych wejściowych użytkownika, bądź za pomocą API. Pamiętaj również, że dane wyjściowe polecenia `ps` ujawniają m.in. parametry, co stanowi kolejny potencjalny sposób na wyciek danych wrażliwych.

Weryfikacja danych wejściowych

Gdy tylko jest to możliwe, należy dostarczać dane wejściowe zmiennym oraz weryfikować dane wejściowe pochodzące od użytkownika i z innych źródeł. To dotyczy na przykład parametrów startowych, zarówno dostarczonych, jak i interaktywnie pobranych za pomocą polecenia `read`, aby uniknąć sytuacji, w której niewinnie wyglądające polecenie `rm -rf "$PROJECTHOME/"*` usunie zawartość zmiennej, ponieważ zmienna nie została zdefiniowana.

Sprawdzanie zależności

Nie należy zakładać, iż określone polecenie lub narzędzie jest dostępne, chyba że jest elementem wbudowanym albo masz pewność, że istnieje w środowisku docelowym. Nawet jeśli komputer zawiera polecenie `curl`, wcale nie oznacza to, iż będzie ono dostępne w środowisku docelowym. O ile to możliwe, warto przygotować rozwiązanie awaryjne — np. jeżeli nie ma polecenia `curl`, należy użyć `wget`.

Obsługa błędów

Gdy działanie skryptu prowadzi do awarii (nie chodzi o „czy", ale raczej „kiedy" i „gdzie"), użytkownik powinien mieć możliwość podjęcia działania. Na przykład zamiast lakonicznego `Błąd 123` należy wskazać przyczynę błędu i sposób jego usunięcia, np. `Próba zapisu w /projekt/xyz/ zakończyła się niepowodzeniem, katalog jest tylko do odczytu`.

Dokumentacja

Bloki główne skryptów należy dokumentować w kodzie (za pomocą składni `# Komentarz`). Dla zachowania czytelności dobrze jest, aby długość wiersza nie przekraczała 80 znaków.

Wersjonowanie

Rozważ wersjonowanie skryptów za pomocą systemu kontroli wersji Git.

Testowanie

Przeprowadzaj lintowanie i *testowanie* skryptów. Ponieważ jest to ważna praktyka, omówię ją dokładniej w następnym punkcie.

Czas przejść do tematu zapewnienia większego bezpieczeństwa skryptom przez ich lintowanie w trakcie tworzenia i testowanie przed udostępnieniem użytkownikom.

Lintowanie i testowanie skryptów

Podczas tworzenia skryptu trzeba go sprawdzać i lintować, aby mieć pewność, że polecenia i instrukcje są stosowane poprawnie. W tym zakresie istnieje elegancki sposób, pokazany na rysunku 3.10, w postaci internetowego narzędzia ShellCheck (*https://blog.davidjeddy.com/2018/11/27/using-shellcheck-to-lint-your-bash-sh-scripts/*). Można je pobrać, zainstalować i stosować lokalnie albo

skorzystać z wersji online dostępnej pod adresem *https://www.shellcheck.net/*. Rozważ także sformatowanie skryptu za pomocą narzędzia `shfmt` (*https://github.com/mvdan/sh*), automatycznie usuwającego błędy, które jeśli pozostaną w kodzie, zostaną zgłoszone przez ShellCheck.

Rysunek 3.10. Dostępne w internecie narzędzie ShellCheck

Ponadto przed przekazaniem skryptu do repozytorium warto rozważyć jego przetestowanie za pomocą `bats` (*https://github.com/sstephenson/bats*), czyli *Bash Automated Testing System*. To narzędzie pozwala zdefiniować plik testowy jako skrypt powłoki stosujący składnię specjalną zapewniającą obsługę testów. Każdy zbiór testu to po prostu funkcja powłoki bash razem z opisem. Skrypty te są zwykle wywoływane jako część potoku ciągłej integracji, np. akcji GitHub.

Nadeszła pora na praktyczne zastosowaniu dobrych praktyk, lintowania i testowania podczas tworzenia skryptów powłoki. W następnym punkcie pokażę implementację skryptu, o którym wspomniałem we wcześniejszej części rozdziału.

Kompletny przykład — skrypt dostarczający informacje o użytkowniku serwisu GitHub

W tym przykładzie wszystkie przedstawione wcześniej wskazówki i narzędzia wykorzystam podczas tworzenia skryptu, który pobiera nazwę użytkownika serwisu GitHub, a następnie wyświetla jego pełne imię i nazwisko oraz rok dołączenia do serwisu.

Dzięki temu pokażę przykład implementacji wykorzystującej dobre praktyki. Przedstawiony tutaj kod należy umieścić w pliku *gh-user-info.sh*, który powinien mieć uprawnienie wykonywalności:

```
#!/usr/bin/env bash

set -o errexit
set -o errtrace
set -o nounset
set -o pipefail

### Parametr powłoki
targetuser="${1:-mhausenblas}" ❶

### Sprawdzenie, czy zależności zostały spełnione
if ! [ -x "$(command -v jq)" ]
then
  echo "Polecenie jq jest niedostępne" >&2
  exit 1
fi

### Kod główny skryptu
githubapi="https://api.github.com/users/"
tmpuserdump="/tmp/ghuserdump_$targetuser.json"

result=$(curl -s $githubapi$targetuser) ❷
echo $result > $tmpuserdump

name=$(jq .name $tmpuserdump -r) ❸
created_at=$(jq .created_at $tmpuserdump -r)

joinyear=$(echo $created_at | cut -f1 -d"-") ❹
echo $name dołączył(a) do serwisu GitHub w $joinyear roku ❺
```

❶ Dostarczenie wartości domyślnej do użycia, jeśli użytkownik żadnej nie dostarczył.

❷ Uzyskanie za pomocą polecenia `curl` dostępu do API GitHub (*https://docs.github.com/en/rest*) w celu pobrania pliku JSON zawierającego informacje o użytkowniku. Te dane są przechowywane w pliku tymczasowym, utworzonym w następnym wierszu kodu.

❸ Pobranie niezbędnych pól za pomocą polecenia `jq`. Zwróć uwagę, że pole `created_at` ma wartość w postaci `"2009-02-07T16:07:32Z"`.

❹ Za pomocą polecenia `cut` następuje wyodrębnienie roku z pola `created_at` w pliku JSON.

❺ Przygotowanie komunikatu danych wyjściowych i wyświetlenie go na ekranie.

Oto przykład uruchomienia skryptu i wykorzystania zdefiniowanych wartości domyślnych:

```
$ ./gh-user-info.sh
Michael Hausenblas dołączył(a) do serwisu GitHub w 2009 roku
```

Gratulacje, masz już wszystko, co jest potrzebne do używania powłoki, zarówno w trybie interaktywnym, jak i w skryptach. Zanim przejdę do podsumowania rozdziału, poświęć chwilę na przeanalizowanie kwestii związanych ze sposobem działania utworzonego przed chwilą skryptu *gh-user-info.sh*:

- Co się stanie, jeśli dane JSON zwrócone przez API GitHuba będą nieprawidłowe? Co po napotkaniu błędu 500 HTTP? Być może dodanie komunikatu z tekstem w stylu `Spróbuj ponownie` jest bardziej przydatne, gdy użytkownik nie może nic innego zrobić.
- Do działania skrypt potrzebuje połączenia sieciowego, w przeciwnym razie polecenie `curl` zakończy działanie niepowodzeniem. Co można zrobić w przypadku braku połączenia sieciowego? Użytkownika trzeba poinformować o tym i zasugerować mu sprawdzenie stanu sieci.
- Rozważ usprawnienia w zakresie sprawdzania zależności — np. przyjęliśmy niejawne założenie o dostępności polecenia `curl`. Czy można dodać operację sprawdzenia pozwalającą upewnić się o dostępności tego pliku binarnego, a jeśli go nie ma, powodującą użycie polecenia `wget`?
- Rozważ dodanie pewnych informacji o działaniu programu. Jeżeli skrypt będzie wywołany z parametrem `-h` lub `--help`, powinien wówczas wyświetlić konkretny przykład użycia i opcje, za pomocą których użytkownik może wpływać na sposób jego działania. (Idealnie byłoby podanie także użytych wartości domyślnych opcji).

Mimo że ten skrypt przedstawia się dobrze i działa w większości przypadków, zawsze można go usprawnić, aby stał się bardziej niezawodny i dostarczał użytkownikowi użyteczne komunikaty błędów. W takim kontekście warto rozważyć użycie frameworków, takich jak `bashing` (*https://github.com/xsc/bashing*), `rerun` (*https://github.com/rerun/rerun*) lub `rr` (*https://taarr.com/*), i tym samym poprawić modularność skryptu.

Podsumowanie

W tym rozdziale skoncentrowałem się na pracy z Linuksem w terminalu, czyli tekstowym interfejsie użytkownika. Omówiłem terminologię związaną z powłoką, wyjaśniłem podstawy pracy z powłoką, pokazałem najczęściej wykonywane zadania, a także wskazałem możliwości zwiększenia produktywności w powłoce dzięki użyciu nowoczesnych wersji wybranych poleceń (np. `exa` zamiast `ls`).

Następnie omówiłem nowoczesne i przyjazne użytkownikowi powłoki, zwłaszcza Fish, a także sposoby ich konfiguracji i pracy z nimi. Przedstawiłem także multiplekser terminala na podstawie `tmux` oraz przykład praktyczny użycia multipleksera, umożliwiający pracę z sesjami lokalnymi i zdalnymi. Użycie nowoczesnej powłoki i multipleksera może znacznie poprawić efektywność. Gorąco zachęcam do ich zastosowania.

W ostatniej części rozdziału omówiłem automatyzację zadań przez tworzenie bezpiecznych i przenośnych skryptów powłoki, a także ich lintowanie i testowanie. Pamiętaj, że w praktyce te powłoki to interpretery poleceń i podobnie jak w przypadku każdego języka programowania nabycie biegłości wymaga praktyki. Masz już podstawową wiedzę z zakresu użycia Linuksa w powłoce i możesz pracować z większością systemów z rodziny Linux, np. mogą to być systemy osadzone lub maszyny wirtualne w chmurze. W każdym razie znajdziesz sposób na uruchomienie terminala i rozpoczęcie wydawania poleceń, interaktywnie bądź poprzez skrypty.

Jeżeli chcesz dowiedzieć się więcej o zagadnieniach omówionych w tym rozdziale, następujące zasoby powinny zapewnić doskonały punkt wyjścia:

Terminal

- Artykuł *Anatomy of a Terminal Emulator* (*https://poor.dev/blog/terminal-anatomy/*)
- Artykuł *The TTY demystified* (*http://www.linusakesson.net/programming/tty/*)
- Artykuł *The terminal, the console and the shell — what are they?* (*https://www.unixsheikh.com/articles/the-terminal-the-console-and-the-shell-what-are-they.html*)
- Artykuł *What is a TTY on Linux? (and How to Use the tty Command) (https://www.howtogeek.com/428174/what-is-a-tty-on-linux-and-how-to-use-the-tty-command/*)
- Artykuł *Your terminal is not a terminal: An Introduction to Streams* (*https://lucasfcosta.com/2019/04/07/streams-introduction.html*)

Powłoka

- Artykuł *Unix Shells: Bash, Fish, Ksh, Tcsh, Zsh* (*https://hyperpolyglot.org/unix-shells*)
- Artykuł *Comparison of command shells* (*https://en.wikipedia.org/wiki/Comparison_of_command_shells*)
- Wątek *Bash vs Zsh* (*https://www.reddit.com/r/linux/comments/1csl7c/bash_vs_zsh/*)
- Artykuł *Ghost in the Shell — Part 7 — ZSH Setup* (*https://vermaden.wordpress.com/2021/09/19/ghost-in-the-shell-part-7-zsh-setup/*)

Multiplekser terminala

- Artykuł *A tmux Crash Course* (*https://thoughtbot.com/blog/a-tmux-crash-course*)
- Artykuł *A Quick and Easy Guide to tmux* (*https://www.hamvocke.com/blog/a-quick-and-easy-guide-to-tmux/*)
- Artykuł *How to Use tmux on Linux (and Why It's Better Than Screen)* (*https://www.howtogeek.com/671422/how-to-use-tmux-on-linux-and-why-its-better-than-screen/*)
- Książka *The Tao of tmux* (*https://leanpub.com/the-tao-of-tmux/read*)
- Książka *tmux 2 Productive Mouse-Free Development* (*https://pragprog.com/titles/bhtmux2/tmux-2/*)
- Witryna *Tmux Cheat Sheet & Quick Reference* (*https://tmuxcheatsheet.com/*)

Skrypt powłoki

- Artykuł *Shell Style Guide* (*https://google.github.io/styleguide/shellguide.html*)
- Artykuł *Bash Style Guide* (*https://github.com/bahamas10/bash-style-guide*)
- Artykuł *Bash best practices* (*https://bertvv.github.io/cheat-sheets/Bash.html*)
- Strona *Bash scripting cheatsheet* (*https://devhints.io/bash*)
- Artykuł *Writing Bash Scripts that are not only Bash: Checking for Bashisms and testing with Dash* (*https://dev.to/bowmanjd/writing-bash-scripts-that-are-not-only-bash-checking-for-bashisms-and-testing-with-dash-1bli*)

Po opanowaniu podstaw pracy w powłoce możesz zapoznać się z tematem kontroli dostępu w systemie Linux.

Sprawdź, co dziś może Ci zaoferować nowoczesny Linux!

Linux cieszy się dużą popularnością wśród administratorów i użytkowników. Znakomicie sprawdza się nawet na bardzo skromnym sprzęcie. Działa na komputerach Raspberry Pi, na maszynach wirtualnych i na komputerach marsjańskich łazików. Niezależnie od tego systemy linuksowe są cały czas rozwijane i dostosowywane do najnowszych trendów i technologii systemów informatycznych. Bez względu na to, czy używasz Linuksa do programowania, do tworzenia złożonych projektów, czy też do pracy biurowej, bardzo zyskasz na dokładniejszym poznaniu jego możliwości.

To książka przeznaczona dla użytkowników komputerów pracujących pod kontrolą Linuksa. Znalazły się w niej tak ważne kwestie, jak omówienie komponentów o krytycznym znaczeniu i mechanizmów kontroli dostępu czy wyjaśnienie systemu plików w Linuksie. Umieszczono tu również liczne wskazówki i ćwiczenia, dzięki którym nauczysz się obsługiwać nowoczesne terminale i powłoki systemu Linux, a także zarządzać obciążeniami. Ponadto dowiesz się, jak uruchamiać aplikacje Linuksa za pomocą kontenerów, i poznasz systemd, nowoczesne systemy plików i niemodyfikowalne dystrybucje, takie jak Flatcar i Bottlerocket. Opisano tutaj również bardziej zaawansowane narzędzia, takie jak połączenia typu P2P i mechanizmy synchronizacji chmury. Oto prosta droga, by szybko zacząć korzystać z przebogatych możliwości nowoczesnego Linuksa!

Najciekawsze zagadnienia:

- Linux jako nowoczesne środowisko pracy
- najważniejsze komponenty Linuksa
- mechanizmy kontroli dostępu
- stos sieciowy Linuksa i związane z nim narzędzia
- mechanizmy obserwacji systemu a zarządzanie obciążeniami
- komunikacja międzyprocesowa, maszyny wirtualne i zapewnianie bezpieczeństwa

Michael Hausenblas jest liderem inżynierów w zespole Amazon Web Services (AWS). Zdobył duże doświadczenie w zakresie inżynierii danych i orkiestracji kontenerów, od Mesos po Kubernetes. Jest też propagatorem standaryzacji W3C i IETF. Obecnie tworzy kod przede wszystkim w języku Go. Wcześniej przez dekadę zajmował się badaniami aplikacyjnymi.

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 978-83-283-9831-3
9 788328 398313
Cena: 69,00 zł