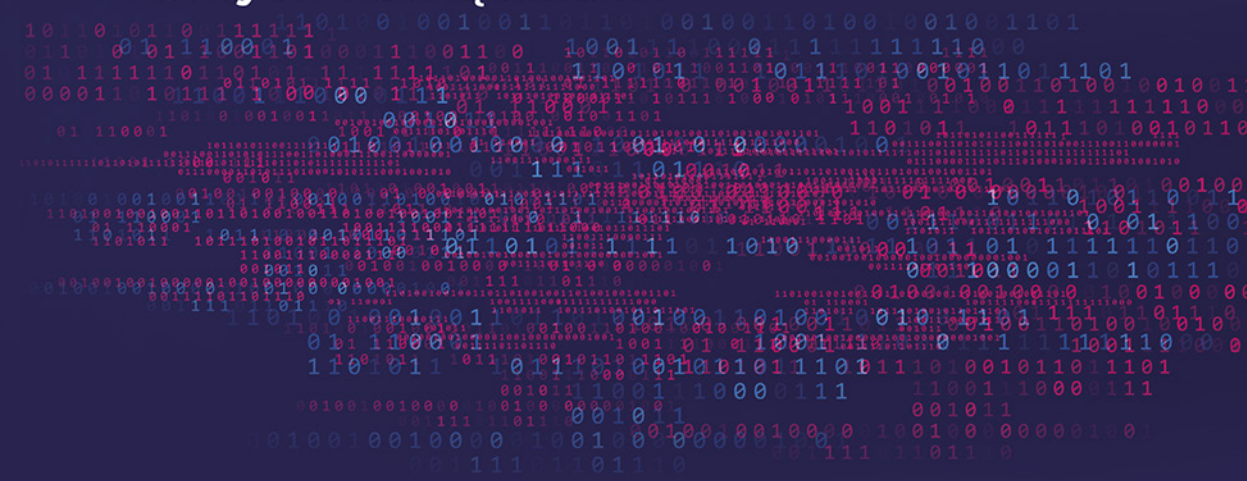
OKIEM EKSPERTA

Linux Podręcznik dewelopera

Rzeczowy przewodnik po wierszu poleceń i innych narzędziach



David Cohen, Christian Sturm

Helion

packt

Tytuł oryginału: The Software Developer's Guide to Linux: A practical, no-nonsense guide to using the Linux command line and utilities as a software developer

Tłumaczenie: Lech Lachowski (rozdz. 1 – 16), Robert Górczyński (wprowadzenie, rozdz. 17)

ISBN: 978-83-289-1754-5

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

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

Printed in Poland.

Spis treści

Rozdział 11
Potoki i przekierowanie

W tym rozdziale pokażmy Ci, jak wykorzystać jedną z najpotężniejszych koncepcji obliczeniowych, jaką są potoki! Potoki mogą być używane do łączenia poleceń, tworząc złożone, dostosowane przepływy, które spełniają określone zadanie. Po lekturze tego rozdziału będziesz w stanie zrozumieć (lub skomponować) coś takiego:

```
history | awk '{print $2}' | sort | uniq -c | sort -rn | head -n 10
```

Powoduje to wypisanie listy 10 najczęściej używanych poleceń powłoki — na naszym komputerze zostały wygenerowane następujące dane wyjściowe:

```
1000 git
 115 ls
 102 go
  83 gpo (alias, który ustawiłem do wysłania lokalnej gałęzi Gita
 ↪do oryginalnej)
  68 make
  65 cd
  59 docker
  42 vagrant
  35 GOOS=linux
  30 echo
```

Aby zrozumieć potoki, musisz najpierw zrozumieć deskryptory plików i przekierowanie operacji we-wy, więc od tego zaczniemy. Niektóre informacje zawarte w tym rozdziale są dość skompilowane, więc nie spiesz się i wypróbuj wszystkie przykłady, aby wszystko dobrze zrozumieć. Czas, który zainwestujesz teraz w naukę tych pojęć, pozwoli Ci zaoszczędzić wiele godzin w całej Twojej karierze programisty.

W tym rozdziale omawiamy następujące zagadnienia:

- deskryptory plików;
- łączenie poleceń za pomocą potoków (|);
- narzędzia CLI;
- praktyczne wzorce potoków;
- sprawdzanie deskryptorów plików.

Deskryptory plików

Zapewne z własnego doświadczenia w inżynierii oprogramowania znasz uchwyty plików (zwane również **deskryptorami plików**). Jeśli nie, zalecamy zapoznanie się z rozdziałem 5. „Wprowadzenie do plików". Krótko mówiąc, jeśli Twój program musi odczytać lub zapisać plik w systemie operacyjnym, otwarcie tego pliku daje do niego „uchwyt" — wskaźnik, czyli inaczej mówiąc, referencję do tego obiektu pliku.

Ponieważ system operacyjny pośredniczy w dostępie do zasobów systemowych, takich jak pliki, śledzi, do których uchwytów (deskryptorów) plików program aktywnie się odwołuje.

Ale nawet jeśli proces nie korzysta z żadnego pliku w systemie operacyjnym, ma otwarte pewne uchwyty plików. W uniksowych systemach operacyjnych każdy proces ma co najmniej trzy deskryptory plików:

- `stdin` — standardowy strumień danych wejściowych lub `fd 0` („zerowy deskryptor pliku");
- `stdout` — standardowy strumień danych wyjściowych lub `fd 1` („pierwszy deskryptor pliku");
- `stderr` — standardowy strumień błędów lub `fd 2` („drugi deskryptor pliku").

Te pierwsze trzy deskryptory plików działają jako standardowe kanały komunikacji między procesami. W rezultacie istnieją one w tej samej kolejności dla każdego procesu tworzonego w systemie. Pierwszy zawsze wskazuje na plik, który będzie używany do odczytu danych wejściowych. Drugi wskazuje na plik, który będzie używany do zapisu danych wyjściowych. A trzeci odwołuje się do pliku, który otrzyma dane wyjściowe na temat błędu.

Opcjonalnie, po tych pierwszych trzech standardowych deskryptorach, może istnieć dowolna liczba innych deskryptorów (uchwytów), w zależności od tego, co robi dany program. Twój proces może mieć następujące elementy:

- pliki, z którymi pracuje;
- gniazda, z których odczytuje lub w których zapisuje (gniazda uniksowe lub TCP zostały napisane dla obsługi sieci);
- urządzenia takie jak klawiatury lub dyski, z których korzysta.

Do czego odwołują się te deskryptory plików?

Teraz wiesz już, do czego służą te deskryptory plików z perspektywy procesów:

- `0` (`STDIN`) — stąd są pobierane dane wejściowe;
- `1` (`STDOUT`) — tutaj umieszczane są normalne dane wyjściowe;
- `2` (`STDERR`) — tutaj umieszczane są dane wyjściowe błędów.

Ale jeśli przyjrzymy się z bliska pojedynczemu procesowi, na które pliki tak naprawdę wskazują te deskryptory plików? Skąd pochodzą dane wejściowe i gdzie zapisywane są dane wyjściowe i błędy?

Jako przykładu użyjmy procesu powłoki Bash: domyślnie pobiera on dane wejściowe (`STDIN`) z terminala (który jest reprezentowany przez plik w systemie plików). Bash wypisuje dane wyjściowe i błędy w tym samym terminalu. Zasadniczo cała sesja powłoki polega na wykonywaniu operacji odczytu i zapisu w jednym pliku. Więcej informacji na ten temat znajdziesz w rozdziale 12. „Automatyzacja zadań za pomocą skryptów powłoki".

Przyjrzyjmy się dokładnie temu rodzajowi przekierowania wejścia i wyjścia.

Przekierowywanie wejścia i wyjścia (praca z deskryptorami plików dla zabawy i potencjalnych korzyści)

Ta wiedza przydaje się dość często podczas wykonywania rzeczywistych zadań programistycznych: gdy chcesz uniknąć wpisywania dużej ilości danych wejściowych i zamiast tego pobierać je z pliku, gdy chcesz rejestrować dane wyjściowe programu i w wielu innych sytuacjach. Podczas tworzenia procesu można kontrolować, gdzie wskazują jego trzy standardowe deskryptory plików. Można dzięki temu wiele osiągnąć.

Przekierowywanie danych wejściowych — <

Symbol < (mniejsze niż) pozwala kontrolować, skąd proces pobiera dane wejściowe. Z pewnością jesteś na przykład przyzwyczajony do podawania powłoce Bash danych wejściowych za pomocą klawiatury, po jednym poleceniu. Zamiast tego spróbujmy podać tej powłoce dane wejściowe z pliku!

Załóżmy, że mamy plik *commands.txt* z następującą zawartością (używamy tutaj `cat` do wypisania zawartości przykładowego pliku):

```
# cat commands.txt
pwd
echo "witajcie, przyjaciele"
echo $SHELL
cd /tmp
pwd
```

Jeśli chodzi o Bash, są to prawidłowe polecenia powłoki, zamierzamy więc uruchomić nowy proces powłoki i użyć tego pliku jako standardowych danych wejściowych:

```
# bash < commands.txt
/tmp/gopsinspect
witajcie, przyjaciele
/bin/bash
/tmp
```

Zamiast monitować nas o wprowadzenie danych i czekać, aż je podamy, Bash odczytuje i wykonuje po jednej linii: odczytuje dane wejściowe z pliku, dopóki nie natknie się na znak nowej linii (`\n`), i wykonuje polecenie tak, jakbyśmy wcisnęli przycisk *Return*.

W tym przykładzie standardowe dane wyjściowe wracają do naszego terminala, gdzie możemy je odczytać. Spróbujmy to teraz zmienić.

Przekierowywanie danych wyjściowych — >

Chcemy przekierować STDOUT (deskryptor pliku 1) do pliku zamiast do terminala, rejestrując dane wyjściowe każdego polecenia zamiast wypisywać je w terminalu w czasie rzeczywistym:

```
# bash < commands.txt > output.log
```

Zauważ, że w terminalu nie ma teraz żadnych widocznych danych wyjściowych, ponieważ znak > przekierował je do pliku *output.log*. Użyj polecenia `cat`, aby wypisać ten plik dziennika i potwierdzić, że zawiera on oczekiwane dane wyjściowe:

```
# cat output.log
/tmp/gopsinspect
witajcie, przyjaciele
/bin/bash
/tmp
```

Co ciekawe, zauważysz, że ponieważ deskryptor pliku 1 jest standardowym wyjściem, zapis > jest równoważny z zapisem 1>. Rzadko spotkasz się z zapisem 1, gdyż przyjmuje się założenie, że standardowe wyjście jest przekierowywane. Innymi słowy poniższe zapisy są równoważne:

```
date > mydate.log
równoważne z napisaniem
date 1> mydate.log
```

Użycie znaków >>, aby dołączać dane wyjściowe bez nadpisywania

W poprzednim przykładzie utworzyliśmy plik dziennika, przekierowując wyjście polecenia za pomocą znaku >. Jeśli uruchomisz ten przykład kilka razy, zauważysz, że plik dziennika w ogóle się nie rozrasta. Za każdym razem gdy przekierujesz dane wyjściowe do pliku za pomocą > *nazwa_pliku*, wszystko w tym pliku zostanie nadpisane.

Aby tego uniknąć — tak jak w przypadku długotrwałego pliku dziennika, który zbiera dane wyjściowe z więcej niż jednego procesu lub polecenia — użyj znaków >> (dołącz). Spowoduje to dołączanie danych do pliku wyjściowego, zamiast nadpisywania za każdym razem całej jego zawartości.

Skrypty powłoki Bash omówimy szczegółowo dalej, a na razie przedstawimy krótki skrypt, który zapisuje znacznik bieżącego czasu w pliku dziennika raz na sekundę:

```
while true; do
    date >> /tmp/date.log
    sleep 1
done
```

W tym przykładowym skrypcie tworzymy nieskończoną pętlę (`while true; do [...]`), która uruchamia polecenie `date`. Przekierowuje ona dane wyjściowe tego polecenia do pliku */tmp/date.log* za pomocą znaków >>, które dołączają dane wyjściowe do pliku (znak > za każdym razem nadpisałby plik). Następnie skrypt śpi przez sekundę i zaczyna się od początku.

Uruchomienie polecenia `date` jeden raz generuje następujące dane wyjściowe:

```
→ ~ date
Sat Jan 6 16:39:37 EST 2024
```

Z drugiej strony uruchomienie tego skryptu nie robi na początku nic widocznego, dane wyjściowe są bowiem przekierowywane do pliku. Oto jak to wygląda, kiedy wklejamy ten mały skrypt do terminala, pozwalamy mu działać przez chwilę, przerywamy jego działanie za pomocą *Ctrl+C*, a następnie wypisujemy zawartość utworzonego pliku:

```
→ ~ while true; do
  date >> /tmp/date.log
  sleep 1
done
^C%
→ ~ cat /tmp/date.log
Sat Jan 6 16:44:01 EST 2024
Sat Jan 6 16:44:02 EST 2024
Sat Jan 6 16:44:03 EST 2024
[ ... ]
Sat Jan 6 16:44:08 EST 2024
```

Tego rodzaju prostego przekierowania wyjściowego będziesz używać we wszystkich codziennych sytuacjach, takich jak tworzenie pliku dziennika ad hoc dla szybkiego debugowania skryptu .

Przekierowywanie błędów za pomocą 2>

Wiele programów wiersza poleceń, które mają dużo oczekiwanych wyników, generuje również sporadyczne błędy — wyobraź sobie polecenie `find`, które napotyka sporadyczne błędy „odmowy uprawnień” dla katalogów, do których nie możesz zajrzeć.

Chociaż tego rodzaju błędy są rzadkie i oczekiwane, nie chcesz, aby były zmieszane ze wszystkim innym, wpływając negatywnie na Twoją wydajność. Staje się to szczególnie ważne, gdy nie używasz narzędzi wiersza poleceń interaktywnie, lecz raczej piszesz krótkie skrypty lub większe programy, które przetwarzają wyniki uruchamianych poleceń.

Pokazaliśmy już, jak przekierowywać standardowe dane wejściowe (`fd 0`) i standardowe dane wyjściowe (`fd 1`). Przyjrzyjmy się teraz, jak przekierować standardowy strumień błędów (`fd 2`) za pomocą składni `2>` (deskryptora `2` dla przekierowania pliku):

```
find /etc/ -name php.ini > /tmp/phpinis.log 2>/dev/null
```

To polecenie wyszukuje wszystkie pliki o nazwie *php.ini* w drzewie katalogu */etc*. Znalezione pliki (`STDOUT` polecenia `find`) są zapisywane w pliku */tmp/phpinis.log*, a wszelkie napotkane błędy są ignorowane przez wysłanie ich do specjalnego pliku o nazwie */dev/null*.

Wskazówka

/dev/null to specjalny obiekt podobny do pliku, który zwraca zera, gdy próbujesz go odczytać, i ignoruje wszystko, co zostało w nim zapisane — jest używany jako coś w rodzaju wysypiska śmieci dla danych wyjściowych, które inżynierowie chcą wyciszyć lub zignorować. Przekonasz się, że jest on dość często wykorzystywany w skryptach.

Skoro znasz już przekierowanie wejścia i wyjścia, przyjrzyjmy się potokom, które łączą obie te koncepcje: przekierowują wyjście jednego polecenia do wejścia drugiego.

Łączenie poleceń za pomocą potoków (|)

Nauczyłeś się przekierowywać każdy z trzech standardowych deskryptorów plików do różnych lokalizacji i przekonałeś się, dlaczego często jest to przydatne. Ale co, jeśli zamiast przekierowywać wejścia i wyjścia między plikami plików, chcesz połączyć ze sobą *wiele programów*?

W wierszu poleceń można użyć znaku potoku (|), aby połączyć wyjście jednego programu z wejściem drugiego. Jest to niezwykle potężny paradygmat, który jest często używany w systemach Unix i Linux do tworzenia niestandardowych poleceń sortowania, filtrowania i przetwarzania:

```
echo -e "jakiś tekst \n znaleziono skarb \n więcej jakiegoś tekstu" | grep skarb
```

Jeśli wkleisz to do swojej powłoki, wypisany zostanie tekst `znaleziono skarb`. Oto co się stało:

1. Zostało uruchomione pierwsze polecenie `echo` i wygenerowało dane wyjściowe, które widzisz między podwójnymi cudzysłowami (znaki nowej linii sprawiają, że jest to 3-liniowy łańcuch znaków).
2. Znak potoku przesyłał strumieniowo te dane wyjściowe (deskryptor pliku `1`) do wejścia następnego polecenia (deskryptor pliku `0`) `grep`. Dane wejściowe `grep` zostały teraz podłączone do wyjścia z poprzedniego polecenia.
3. Następnie polecenie `grep` przejrzało każdą oddzieloną nową linię i znalazło dopasowanie do `skarb` w drugiej linii. `grep` wypisało tę drugą linię do standardowego wyjścia.

Polecenia z wieloma potokami

Oto dość ekstremalny przykład, który pokazaliśmy na początku rozdziału:

```
history | awk '{print $2}' | sort | uniq -c | sort -rn | head -n 10 >
↪/tmp/top10commands
```

Każdy potok w tym złożonym poleceniu pobiera wynik poprzedniego polecenia (`STDOUT`) i używa go jako wejścia (`STDIN`) dla następnego polecenia.

Przekazywanie za pomocą potoku wyjścia jednego polecenia do wejścia drugiego umożliwia tego rodzaju przepływy pracy, filtrując i sortując strumienie danych między tymi poleceniami bez konieczności pisania niestandardowego oprogramowania. To, że nie ma programu o nazwie `top10commands`, nie oznacza, iż nie można szybko poskładać takiego polecenia z istniejących standardowych poleceń, takich jak te.

Odczytywanie (i budowanie) złożonych poleceń wielopotokowych

Bez względu na to, jak skomplikowane lub magiczne będą się wydawać niektóre z połączonych potokami poleceń, wszystkie zostały zbudowane w ten sam sposób: po jednym poleceniu w danym momencie. Niezależnie od tego, czy próbujesz odczytywać tak złożoną serię poleceń, czy tworzyć własne, proces jest taki sam:

1. Weź pierwsze polecenie i upewnij się, że rozumiesz, co robi na podstawowym poziomie. Przejrzyj stronę podręcznika lub inną dokumentację, jeśli nie jesteś zaznajomiony z tym poleceniem.
2. Uruchom polecenie i sprawdź jego dane wyjściowe.
3. Dodaj potok i następujące po nim polecenie.
4. Powtarzaj czynności od *kroku 1.* do momentu, aż zrozumiesz działanie wszystkich poleceń.

Zobaczysz, że gdy zastosujesz ten proces, będziesz w stanie opanować nawet najbardziej przerażające powłokowo-potokowe zawiłości. Zawsze pamiętaj, że masz do czynienia tylko ze strumieniem danych, który przepływa przez potoki od polecenia do polecenia, a po drodze jest kształtowany, modyfikowany, filtrowany, przekierowywany i przekształcany.

Omówimy to szczegółowo w rozdziale 12. „Automatyzacja zadań za pomocą skryptów powłoki", ale postaraj się szanować innych programistów, którzy muszą czytać Twój kod: ogranicz swoje instrukcje do dwóch lub trzech potoków i używaj dobrze nazwanych zmiennych do przechowywania pośrednich wyników w celu ułatwienia odczytu, jeśli pozwalają na to ograniczenia pamięci.

Skoro wiesz już, jak podstawowe deskryptory plików są udostępniane jako łatwe w użyciu przekierowania wejścia i wyjścia, przyjrzyjmy się kilku prawdziwym przykładom użytecznych kombinacji programów, które opierają się na tej komponowalności wbudowanej w system Unix.

Narzędzia CLI, które należy znać

Zanim przejdziemy do dziwacznych kombinacji, które pokazaliśmy na początku rozdziału, przyjrzyjmy się kilku z najczęściej używanych narzędzi Uniksa, które służą do filtrowania, sortowania i sklejania strumieni danych, które będziesz tworzyć w wierszu poleceń.

cut

Narzędzie `cut` pobiera separator (`-d`) i na jego podstawie dzieli dane wyjściowe, podobnie jak `String.Split()` lub `String.Fields()` w wielu językach programowania. Następnie należy za pomocą opcji `-f`, na przykład `f1` dla pierwszego pola, wybrać, które pole (element listy) ma generować dane wyjściowe.

Jeśli podasz poleceniu `cut` więcej niż jedną linię wejścia, powtórzy ona tę samą operację na wszystkich liniach:

```
echo "this is a space-delimited line" | cut -d " " -f4
space-delimited
```

Możesz również zobaczyć, jak działa użycie różnych separatorów w poleceniu `cut` — w poniższym przykładzie zamiast spacji wykorzystamy znak łącznika:

```
→ ~ echo "this is a space-delimited line" | cut -d "-" -f1
this is a space
→ ~ echo "this is a space-delimited line" | cut -d "-" -f2
delimited line
```

Widać, że zmienia to również liczbę dostępnych pól — w tym przypadku są dwa pola, ponieważ w tekście jest tylko jeden łącznik. Próba wypisania czwartego pola za pomocą `f4`, jak w poprzednim przykładzie, daje po prostu pustą linię.

Aby na komputerze z systemem macOS uzyskać przyjazne nazwy dla wszystkich użytkowników root, można użyć następujących opcji:

```
# grep root /etc/passwd | cut -d ":" -f5
System Administrator
System Services
CVMS Root
```

sort

Polecenie `sort` przeprowadza sortowanie poszczególnych linii alfabetycznie lub numerycznie.

Sortowanie odwrotne za pomocą opcji `-r` jest często przydatne w przypadku danych liczbowych (`-n`). Często będziesz używać razem opcji `-rn` (zobacz punkt „»Top X« z licznikiem” w podrozdziale „Praktyczne wzorce potoków”).

Flaga `-h` może być bardzo przydatna do sortowania na podstawie czytelnych dla człowieka danych wyjściowych wielu innych poleceń pokazanych w poniższym listingu:

```
# du -h | sort -rh
1.6M    .
1.3M    ./.git
1.2M    ./.git/objects
 60K     ./.git/hooks
 28K     ./.git/objects/d8
```

uniq

To polecenie usuwa zduplikowane linie. W celu oczekiwanego działania wymaga posortowanych danych, w przeciwnym razie sprawdza tylko, czy każda linia nie jest duplikatem poprzedniej:

```
# cat /tmp/uniq
one
two
one
one
one
seven
one
```

Domyślne zachowanie nie jest prawdopodobnie tym, czego byśmy oczekiwali:

```
# uniq /tmp/uniq
one
two
one
seven
one
```

`uniq` pomija wystąpienia, gdy znajdują się jedno po drugim, ale pozostawia je, kiedy są oddzielone innym tekstem. Teraz to samo z posortowanymi danymi:

```
# sort /tmp/uniq | uniq
one
seven
two
```

Zliczanie

Polecenie `uniq` ma również użyteczną opcję zliczania, którą można włączyć za pomocą argumentu `-c`. Warto tutaj powtórzyć to samo zastrzeżenie co przy posortowanych danych wejściowych — na przykład dla pliku o następującej zawartości:

```
arch
alpine
arch
arch
```

Po uruchomieniu przez `uniq -c` wygenerowany zostanie następujący wynik:

```
$ uniq -c /tmp/sort1.txt
1 arch
1 alpine
2 arch
```

Nie tego oczekiwałaby większość użytkowników: w tym pliku są trzy wystąpienia `arch`, ale `uniq` pokazuje dwa oddzielne liczniki dla tego samego słowa. Aby uzyskać oczekiwane zachowanie (polecenie `uniq` powinno zwrócić wyjście, które nie zawiera żadnych zduplikowanych linii), dane wejściowe muszą być posortowane.

Dla początkujących użytkowników jest to irytujące, ale całkowicie zgodne z filozofią Uniksa: narzędzia powinny być niewielkie i ostre i nie powinny powielać wzajemnie swoich funkcjonalności. Jeśli piszesz narzędzie sortowania, powinno ono tylko sortować, a jeśli piszesz narzędzie ujednolicające, może ono opierać się na sortowaniu, które jest wykonywane przez inne narzędzia, aby zapewnić ekstremalnie konserwatywne (i spójne) wykorzystanie pamięci.

Tutaj przed użyciem `uniq` najpierw sortujemy, co daje nam oczekiwane dane wyjściowe:

```
$ sort /tmp/sort1.txt | uniq -c
   1 alpine
   3 arch
```

Zwróć uwagę, że to sortowanie odbywa się w kolejności rosnącej, a nie tego chcielibyśmy dla listy poleceń top X, którą pokazaliśmy na początku rozdziału. Aby rozwiązać ten problem, przeprowadzamy sortowanie „liczbowe odwrotne" (`-rn`) tej numerowanej listy (ponieważ każda linia zaczyna się teraz od liczby, a dzięki `uniq -c` jest to łatwe do zrobienia). Oto przykład tego w działaniu dla pliku z wieloma duplikatami:

```
$ sort /tmp/sortme.txt | uniq -c | sort -rn
   6 ubuntu
   4 alpine
   3 gentoo
   2 yellow dog
   2 arch
   1 suse
   1 mandrake
```

wc

Za pomocą tego polecenia można policzyć wejściowe słowa, linie, znaki i bajty. Można również liczyć słowa rozdzielane spacjami za pomocą opcji `-w`:

```
# echo "foo bar baz" | wc -w
       3
```

Liczenie linii jest niezwykle powszechne w następującym formacie:

```
# wc -l < /etc/passwd
     123
```

head

Polecenie `head` zwraca pierwsze linie strumienia lub pliku — domyślnie 10 linii. Liczbę linii możesz określić za pomocą opcji `-n`:

```
# head -n 2 /etc/passwd
##
# User Database
```

tail

Jest to przeciwieństwo polecenia `head`: zwraca linie z końca pliku lub strumienia. Przyjmuje flagę `-n` podobnie jak `head`.

Polecenie `tail` może być również używane interaktywnie do śledzenia pliku dziennika, nawet jeśli do tego pliku są strumieniowane (zapisywane) nowe dane. Bardzo często będziesz je napotykać podczas rozwiązywania problemów:

```
tail -f /var/log/ngnix/access.log
```

tee

Czasami jedna kopia danych ze standardowego wejścia nie wystarczy. Polecenie `tee` kopiuje standardowe wejście do standardowego wyjścia, zapisując jednocześnie kopię w pliku. Bardzo lubimy stosować `tee` w dwóch konkretnych przypadkach.

Pierwszy to debugowanie i rejestrowanie: kiedy uruchamiamy skrypty lub programy, które generują dane wyjściowe, polecenie `tee` może być używane do wyświetlania wyjścia na ekranie i rejestrowania go w pliku w celu późniejszej analizy. Wykorzystujemy tutaj polecenie `echo`, ale prawdopodobnie przed pierwszym potokiem wywołasz tu swój własny program:

```
# echo "Hello" | tee /tmp/greetings.txt
Hello

# cat /tmp/greetings.txt
Hello
```

Drugi przypadek użycia, w którym `tee` jest przydatne, to kopiowanie danych z potoków, takich jak te, które uczymy się konstruować w tym rozdziale. Możesz użyć `tee`, aby przechwycić ten przepływ w dowolnym punkcie potoku i zapisać lub sprawdzić wyniki pośrednie bez zakłócania działania potoku.

Oto wcześniejszy przykład „10 najlepszych poleceń", ale z poleceniem `tee` wstawionym przed ograniczeniem wyników do 10. Zapisuje to pełne wyniki w pliku tymczasowym przed ich skróceniem:

```
history | awk '{print $2}' | sort | uniq -c | sort -rn | tee
/tmp/all_commands.txt | head -n 10
```

Jeśli chcesz teraz zobaczyć wszystkie polecenia, a nie tylko najlepsze 10, możesz po prostu użyć narzędzia `cut` lub `less`, aby sprawdzić plik */tmp/all_commands.txt*.

awk

Polecenie `awk` jest często stosowane jedynie do kolumn danych, ale w rzeczywistości jest to cały język.

Można na przykład pobrać drugą kolumnę z każdej linii w następujący sposób:

```
# echo "two columns" | awk '{print $2}'
columns
```

sed

Polecenie `sed` jest edytorem strumieni z mnóstwem opcji. Najczęściej jest używane do zastępowania znaków w strumieniach lub plikach.

Wyobraź sobie, że masz plik o nazwie *somefile.txt*:

```
# cat /tmp/sensitive.txt
Nopasswords
not_a_password_either
sillypasswordtimes
password
ok this works
```

Jeśli chcesz zredagować *tylko* wiersz, który zwiera `password`, wykonaj następujące polecenie:

```
sed 's/^password$/REDACTED/' /tmp/sensitive.txt
nopasswords
not_a_password_either
sillypasswordtimes
REDACTED
ok this works
```

W tym przykładzie użyliśmy pliku zamiast strumienia wejściowego pochodzącego z innego polecenia. Domyślnie nie spowoduje to modyfikacji oryginalnego pliku. Jeśli *chcesz* zmodyfikować plik wejściowy, użyj opcji `-i` (ang. *in-place*), czyli modyfikacji w miejscu.

Skoro zapoznałeś się z już z potokami i niektórymi z najczęściej wykorzystywanych narzędzi wiersza poleceń, zestawimy razem te elementy konstrukcyjne i pokażemy kilka praktycznych wzorców, których możesz używać, aby ułatwić sobie codzienną pracę z wierszem poleceń.

Praktyczne wzorce potoków

Jak wspomnieliśmy wcześniej, dłuższe polecenia wielopotokowe buduje się iteracyjnie — po jednym. Istnieje jednak kilka przydatnych wzorców, które są często wielokrotnie wykorzystywane.

„Top X" z licznikiem

Ten wzorzec sortuje dane wejściowe w porządku malejącym według liczby wystąpień. Widziałeś to w pierwotnym przykładzie z tego rozdziału, który wyświetlał najczęściej używane polecenia powłoki z pliku historii Basha.

Oto ten wzorzec:

```
jakieś_wejście | sort | uniq -c | sort -rn | head -n 3
```

Możemy zauważyć następujące szczegóły dotyczące tego wzorca:

- Dane wejściowe są sortowane alfabetycznie, a następnie przekazywane do polecenia `uniq -c`, które do działania wymaga posortowania danych wejściowych.
- Narzędzie `uniq -c` eliminuje duplikaty, ale dodaje licznik (`-c`), wskazujący, ile duplikatów znaleziono dla każdego wpisu.
- Polecenie `sort` jest uruchamiane ponownie, tym razem jako sortowanie odwrotne (`-r` i `-n`), które sortuje unikatowe zliczenia z wejścia i przekazuje linie posortowane w odwrotnej kolejności (od najwyższej liczby).
- Polecenie `head` przyjmuje ten ranking i przycina go do trzech linii (`-n 3`), dając trzy pierwsze łańcuchy znaków z oryginalnych danych wejściowych wraz częstotliwością ich występowania.

Ten wzorzec może się przydać, gdy trzeba poznać najczęstsze przeglądarki użytkowników wyświetlające Twoją stronę, adresy IP najgorszych hakerów, którzy próbują sondować i wykorzystać Twoją stronę, lub w każdej innej sytuacji, w której użyteczna jest posortowana, rankingowa lista.

curl | bash

Wzorzec `curl | bash` jest popularnym skrótem stosowanym w Linuksie do pobierania i wykonywania skryptów bezpośrednio z internetu. Metoda ta łączy w sobie dwa potężne narzędzia wiersza poleceń: `curl`, który pobiera zawartość z adresu URL, i `bash`, czyli interpreter powłoki, wykonujący pobrany skrypt. Ten wzorzec pozwala zaoszczędzić mnóstwo czasu i umożliwia programistom szybkie wdrażanie aplikacji lub uruchamianie skryptów bez ich ręcznego pobierania, a następnie wykonywania.

Zainstalujmy na przykład blokujący reklamy serwer DNS Pi-hole, korzystając z tego wzorca:

```
curl -sSL https://install.pi-hole.net | bash
```

Przeanalizujmy to krok po kroku:

1. `curl -sSL https://install.pi-hole.net` — powoduje pobranie skryptu instalacji Pi-hole, który jest hostowany pod podanym adresem URL. Przekazujemy dwie opcje:
   - `-sS` — tryb cichy daje nieprzetworzoną odpowiedź z serwera, ale pokazuje błędy w przypadku ich wystąpienia;
   - `-L` — stosowanie przekierowań.
2. `|` — symbol potoku przekazuje wyjście z poprzedniego polecenia (`curl`) jako wejście do następnego polecenia (`bash`).
3. `bash` — wykonuje skrypt pobrany przez `curl`.

Jest to niezwykle przydatny wzorzec do automatyzacji takich rzeczy jak wdrożenia kodu albo instalacja lub konfiguracja lokalnego środowiska. Należy jednak zachować szczególną ostrożność, aby skrypt, który pobierasz i wykonujesz, nie był złośliwy. Ślepe uruchamianie skryptów z internetu jest bardzo złą praktyką.

Względy bezpieczeństwa dotyczące curl | sudo | bash

Za każdym razem gdy pozwalasz zewnętrznej stronie, aby uruchamiała kod na Twoim komputerze, przehandlowujesz dla wygody pewne względy bezpieczeństwa. Pod tym względem używanie `curl | sudo | bash` do instalowania czegoś za pośrednictwem skryptu hostowanego na zaufanym serwerze nie różni się zbytnio od korzystania z menedżera pakietów. Większość menedżerów pakietów (z wyjątkiem `nix`) nie ma również szczególnie imponującego projektu dotyczącego zabezpieczeń, ale zasadniczo zapewnia rozsądny zestaw funkcji bezpieczeństwa. Gdy stosujesz `curl | sudo | bash` do instalowania jakiegoś skryptu, rezygnujesz z tych wszystkich funkcjonalności bezpieczeństwa:

- Nie ma takiego pakietu, którego sumę kontrolną można sprawdzić i który można kryptograficznie podpisać, aby mieć pewność, że mamy poprawną i oficjalną wersję.
- Nie ma ograniczeń — ani egzekwowania — z których serwerów pobierane są pakiety, i nie wiemy, jak bezpieczne są te serwery: nie masz sposobu na zidentyfikowanie zainfekowanego serwera, na którym znajdują się złośliwe skrypty instalacyjne.
- Same skrypty są tylko kodem uruchamianym z uprawnieniami roota na Twoim komputerze, więc mogą zrobić wszystko, co *Ty* możesz zrobić na swoim komputerze, to zaś może wyjść na dobre lub na złe. Szczerze mówiąc, wiele popularnych menedżerów pakietów również ma ten problem.

Z tych wszystkich powodów zwróć uwagę na nasze ostrzeżenie, aby rozdzielić polecenie `curl` na osobny krok i przed uruchomieniem `sudo bash` w celu wykonania pobranego skryptu instalacyjnego najpierw go przeczytać. Oto najważniejsze rzeczy, na które należy zwrócić uwagę:

- Upewnij się, że serwer lub domena, z których pobierasz skrypt, są godne zaufania; powinna to być strona renomowanego programisty lub zaufana zewnętrzna platforma hostingowa kodu.
- Upewnij się, że do pobierania za pomocą `curl` używasz szyfrowanego protokołu HTTPS (tzn. adres URL powinien zaczynać się od *https://*).
- Przeczytaj uważnie skrypt, aby zobaczyć, które polecenia uruchamia i skąd pobiera dodatkowy kod lub pliki wykonywalne. Jeśli pobiera dodatkowe skrypty lub pliki wykonywalne, przyjrzyj się również im.

Ustaliliśmy, iż `curl | sudo | bash` nie jest szczególnie bezpieczną metodą instalacji oprogramowania. Przestrzeganie tych wytycznych może pomóc Ci zachować większe bezpieczeństwo, jeśli — jak większość z nas — pewnego dnia ulegniesz pokusie i zastosujesz tę metodę instalacji dla konkretnego oprogramowania (na przykład za pomocą `homebrew` w systemie macOS).

Przyjrzyjmy się teraz innemu typowemu wzorcowi: filtrowaniu i wyszukiwaniu przy użyciu narzędzia `grep`.

Filtrowanie i wyszukiwanie za pomocą narzędzia grep

Kiedy uruchamiasz polecenia, które generują dużo danych wyjściowych, najlepszą praktyką jest filtrowanie tych danych w celu uzyskania pożądanych informacji. Najpopularniejszym służącym do tego narzędziem jest `grep` i możesz potraktować je jako wysoce konfigurowalną funkcję wyszukiwania tekstu lub dopasowywania łańcuchów znaków. Oto przykład, jak może wyglądać filtrowanie.

Wyobraź sobie, że musisz znaleźć katalog roboczy procesu Linuksa. W tym celu można zastosować narzędzie `lsof`:

```
→ ~ lsof -p 3243 | grep cwd
vagrant 3243 dcohen cwd   DIR                1,4      192
51689680 /Users/dcohen/code/my_vagrant_testenv
```

Oto krótki opis tego, co się tutaj dzieje:

1. Przy użyciu `lsof` otrzymujemy listę otwartych uchwytów plików dla określonego procesu (`PID 3243`).
2. Następnie przekazujemy wyniki (`|`) do narzędzia `grep` i używamy go do wyszukiwania wyników dla łańcucha znaków `cwd`. Jest tylko jedna linia wyników, która zawiera łańcuch znaków `cwd`, więc jest to jedyna linia wypisywana przez `grep` w terminalu.

Ten wzorzec jest przydatny zawsze, gdy masz *dużo* danych wejściowych, ale potrzebujesz tylko podzbioru tych danych, które można zidentyfikować za pomocą określonego łańcucha znaków. Polecenie `grep` działa na liniach tekstu wejściowego, więc jest niezwykle pomocne przy zbieraniu różnych danych. Mogą to być:

- dzienniki zawierające obserwowany adres IP;
- wystąpienia nazwy użytkownika w strumieniu danych przekazywanych za pomocą potoku;
- linie pasujące do wzorca (`grep` może wykorzystywać wyrażenia regularne i przyjmować wzorce łańcuchów znaków oprócz literałów łańcuchów znaków wyszukiwania).

Polecenie `grep` to duże i potężne narzędzie, z którego będziesz korzystać prawie każdego dnia. Więcej informacji na temat `grep` znajdziesz na stronie podręcznika, gdy wpiszesz `man grep`.

Pokazaliśmy już narzędzie `grep` używane dla plików (na przykład `grep searchstring hello.txt`), ale jest to również nieoceniony składnik filtrujący w poleceniach potokowych. Przyjrzyjmy się teraz praktycznemu przykładowi.

grep i tail do monitorowania dzienników

Kiedy przeglądasz dzienniki produkcyjne, aby dowiedzieć się, co jest nie tak, często chcesz zobaczyć tylko dzienniki zawierające określone słowa kluczowe lub łańcuchy znaków wyszukiwania. W tym celu uruchom coś takiego:

```
tail -f /var/log/webapp/too_many_logs.log | grep "twójRegexWyszukiwania"
```

Ten wzorzec stale monitoruje plik dziennika pod kątem nowych wpisów, które pasują do *twójRegexWyszukiwania*, dzięki czemu możesz zobaczyć tylko dzienniki potrzebne do wykonania aktualnego zadania.

find i xargs do wykonywania operacji na grupach plików

`xargs` jest potężnym narzędziem, które umożliwia iterację (czyli wykonywanie pętli `for`) wewnątrz pojedynczego polecenia. Domyślnie `xargs` pobiera każdy otrzymywany fragment danych wejściowych (rozdzielony spacją, tabulatorem, nową linią i końcem pliku) i wykonuje określony program, używając tego fragmentu jako wejścia. Na przykład jeśli chcesz wyszukać określoną zawartość *tylko* w plikach zwróconych przez określone zapytanie `find`, możesz uruchomić następujące polecenie:

```
find . -type f -name "*\.txt" | xargs grep "termin_wyszukiwania"
```

Polecenie to wyszukuje wszystkie pliki, których nazwy kończą się na `.txt`, a następnie używa `xargs` do zastosowania polecenia `grep` indywidualnie do każdego pliku. Ten wzorzec jest przydatny do wyszukiwania lub modyfikowania wielu plików jednocześnie. Musimy Cię uprzedzić, że `xargs` jest potężnym — i *dużym* — programem, zdolnym do wykonywania wielu rzeczy (w tym interpolacji łańcucha znaków do wykonywanego przez Ciebie polecenia). Nie możemy opisać tutaj tego wszystkiego, dlatego zapoznaj się ze stroną podręcznika i przeszukaj internet, aby znaleźć przykłady, jeśli jesteś w sytuacji, w której tego rodzaju funkcjonalność może Ci pomóc.

sort, uniq i odwrotne sortowanie liczbowe do przeprowadzania analizy danych

Jest to użyteczny wzorzec, który pokazaliśmy na początku rozdziału, gdzie używaliśmy go do filtrowania obszernej historii poleceń, aby uzyskać listę „najpopularniejszych X poleceń uruchamianych w tym systemie". Podstawowy wzór jest następujący:

```
(strumień wejściowy) | sort | uniq -c | sort -rn
```

Ten przydatny do analizy danych wzorzec sortuje dane ze strumienia wejściowego, deduplikuje je podczas liczenia unikatowych wystąpień, a następnie wykonuje odwrotne sortowanie liczbowe, aby zwrócić zdeduplikowane dane, z najczęściej używanymi liniami na początku listy.

Często przycina się go za pomocą | `head -n $LICZBA`, aby uzyskać tylko określoną liczbę wyników:

```
history | awk '{print $2}' | sort | uniq -c | sort -rn | head -n 10
```

Używamy tutaj `history`, aby pobrać całą historię poleceń powłoki. Daje nam to wiele linii:

```
12 brew install --cask emacs
```

Interesuje nas tylko najczęściej używane polecenie (w tym przypadku `brew`), korzystamy więc z `awk` w celu pobrania drugiej kolumny.

Następnie sortujemy, aby duplikaty tego samego polecenia występowały obok siebie w strumieniu.

Potem usuwamy te duplikaty za pomocą `uniq`, dodając do każdego pozostałego wystąpienia licznik wystąpień. Teraz sortujemy ponownie — tym razem wykorzystując `-rn` do odwróconego sortowania liczbowego, co daje nam efekt „top X". Na koniec wybieramy pierwsze 10 linii za pomocą polecenia `head`.

Powoduje to wypisanie wspomnianej listy 10 najczęściej używanych poleceń powłoki — na naszym komputerze wygląda to tak:

```
1000 git
 115 ls
 102 go
  83 gpo (alias, który ustawiłem do wysłania lokalnej gałęzi Gita do oryginalnej)
  68 make
  65 cd
  59 docker
  42 vagrant
  35 GOOS=linux
  30 echo
```

awk i sort do przeformatowywania danych i przetwarzania opartego na polach

`awk` to coś więcej niż program — to język przetwarzania strumieniowego. Jeśli pracujesz ze strumieniami danych w systemie Unix, to spędzenie kilku dni na nauce podstaw może zaoszczędzić tygodnie podczas pracy. Dobrym początkiem jest jednak użycie składni `$#` w celu odwołania się do kolumn rozdzielanych znakami niedrukowalnymi w każdej linii strumienia danych.

Przyjrzyjmy się przykładowi podania strumienia danych w następujący sposób:

```
Foo bar baz
Some data is nice
```

Gdy narzędzie `awk` widzi znak `$1`, interpretuje to jako „pierwszą kolumnę" lub w tym przypadku `Foo` w linii `1` i `some` w linii `2`. `$2` to druga kolumna (`bar`, `data`) itd. Jest to niezwykle powszechna funkcja do wykorzystania podczas pracy z danymi, które są zbyt skomplikowane dla prostych poleceń `cut`:

```
cat file.txt | awk '{print $2, $1}'
```

Wygeneruje to następujące dane wyjściowe:

```
bar Foo
data Some
```

W tym przypadku dla każdego pliku wypisuje kolumnę `2` przed kolumną `1` i ignoruje wszystkie pozostałe dane z każdej linii. Jest to często wykorzystywane do przeformatowywania i porządkowania danych na podstawie określonych pól.

sed i tee do edytowania i tworzenia kopii zapasowych

Polecenie `sed` (ang. *Stream EDitor*), czyli edytor strumienia, jest stosowane, gdy chcemy przekształcić strumień danych. Robisz to kilkanaście razy dziennie w edytorze tekstu, kiedy wyszukujesz i zastępujesz jakiś symbol. To polecenie jest zasadniczo wersją wiersza polecenia tej funkcjonalności: przekształca wszystkie wystąpienia `old` z pliku *file.txt* w nowy łańcuch znaków i zapisuje wynikowy strumień w nowym pliku *file.txt.changed*. Robi to bez wprowadzania zmian w oryginalnym pliku *file.txt*:

```
sed 's/old/new/g' file.txt | tee file.txt.changed
```

Chociaż edycja zawartości pliku jest łatwą demonstracją tej koncepcji, `sed` jest niezwykle przydatne do przekształcania danych strumieniowych, gdy przepływają z wyjścia jednego polecenia do wejścia drugiego:

```
(strumień wejściowy) | sed 's/old/new/g' | (następne polecenie)
```

ps, grep, awk, xargs i kill do zarządzania procesami

Chociaż `pgrep` jest dobrym narzędziem do wysyłania sygnałów do wszystkich procesów, których nazwa pasuje do wzorca, czasami jest po prostu niedostępny w systemie. Używając poniższego zestawu połączonych potokami poleceń, możesz łączyć podobne funkcjonalności (i uzyskać znacznie bardziej szczegółowe informacje docelowe, a nie tylko nazwy):

```
ps aux | grep "nazwa_procesu" | awk '{print $2}' | xargs kill
```

Polecenie `ps` rozpoczyna od listy uruchomionych procesów, które `greo` filtruje, aby uzyskać tylko te, które zawierają szukany wzorzec. Narzędzie `awk` pobiera drugą kolumnę (identyfikator procesu) dla każdej pasującej linii, a następnie podaje wszystkie dopasowane linie poleceniu `xargs` (naszej quasi pętli `for`), która wykonuje `kill` dla każdego PID. Powoduje to wysłanie sygnału `SIGTERM` do każdego dopasowanego procesu i (miejmy nadzieję) zatrzymuje go.

tar i gzip do tworzenia kopii zapasowych i kompresowania

Chociaż wiele narzędzi ma flagi, które pozwalają zrobić obie rzeczy, łączenie archiwizacji i kompresji jest kolejnym przypadkiem użycia, który ma sens. Daje to dodatkową elastyczność dodawania kolejnych poleceń do łańcucha. Jeśli chcesz na przykład dodać szyfrowanie, wystarczy tylko jedno dodatkowe polecenie w potoku:

```
tar cvf - /path/to/directory | gzip > backup.tar.gz
```

Powoduje to utworzenie skompresowanego archiwum katalogu, powszechnie używanego do tworzenia kopii zapasowych i przechowywania plików. Tego rodzaju wzorca używają też większe polecenia:

```
ssh user@mysql-server "mysqldump --add-drop-table database_name | gzip -9c" |
↪gzip -d | mysql
```

Jest to szczególnie fajny przykład, który umożliwia zalogowanie się do serwera bazy danych za pomocą SSH, zrzuca bazę danych, kompresuje ten strumień danych, przesyła go z powrotem do lokalnej maszyny przez SSH, ponownie go dekompresuje, a ostatecznie zrzuca do lokalnego serwera MySQL.

Twoim celem nie musi być pisanie tak złożonych poleceń (lub podobnie skomplikowanych, które pokazaliśmy w tym rozdziale), ale jeśli będziesz potrafić w mgnieniu oka złożyć coś takiego, może to wydostać Cię z niektórych skrajnych tarapatów programistycznych. Mamy nadzieję, że w tym podrozdziale wykazaliśmy, iż zrozumienie podstaw przekierowywania wejścia i wyjścia, które umożliwiają systemy Unix — za pośrednictwem <, >, >>, | i ogólnie deskryptorów plików — to w zasadzie supermoce. Używaj ich mądrze.

Zagadnienia zaawansowane: sprawdzanie deskryptorów plików

W systemie Linux można łatwo *zobaczyć*, gdzie wskazują deskryptory plików procesu. W tym celu będziemy używać odrobinę magicznego wirtualnego systemu plików */proc*.

Procfs (ang. *proc virtual filesystem*), czyli wirtualny system plików `proc`, to wyłącznie linuksowa warstwa abstrakcji, która reprezentuje stan jądra i procesu w postaci plików. Dane w tych plikach pochodzą bezpośrednio z jądra systemu operacyjnego i istnieją tylko podczas ich odczytywania. Samo wylistowanie katalogu */proc* pokaże Ci wiele plików — oto niektóre z ważniejszych, zaczerpnięte ze strony wiki Arch Linux:

- */proc/cpuinfo* — informacja o CPU
- */proc/meminfo* — informacja o pamięci fizycznej
- */proc/vmstats* — informacja o pamięci wirtualnej
- */proc/mounts* — informacja o montowaniach
- */proc/filesystems* — informacja o systemach plików, które zostały skompilowane w jądrze i których moduły jądra są aktualnie załadowane
- */proc/uptime* — czas działania systemu
- */proc/cmdline* — wiersz poleceń jądra

Jeśli chodzi o deskryptory plików, najciekawsze dla nas jest coś, czego nie widać na powyższej liście: */proc* zawiera katalog dla każdego procesu uruchomionego na komputerze; katalog taki ma w nazwie **ID procesu** (**PID**).

W katalogu */proc* procesu deskryptory plików tego procesu są reprezentowane jako dowiązania symboliczne w katalogu o nazwie `fd`. Kiedy wyświetlisz długą listę dla tego katalogu */proc/$PID/fd*, zobaczysz, że `l` jest pierwszym znakiem na długiej liście, który oznacza specjalny plik `link`, jak zapewne pamiętasz z rozdziału 5. „Wprowadzenie do plików".

Praktycznie rzecz biorąc, `/proc/1/` to katalog `init` *proc* procesu, a deskryptory plików init można wyświetlić, wyświetlając długą listę dla `/proc/1/fd`.

Przyjrzyjmy się deskryptorom plików dla interaktywnego procesu powłoki Bash, działającej na naszym komputerze, której `ps aux | grep bash` mówi, że mamy `PID 9`:

```
root@server:/# ls -alh /proc/9/fd
total 0
dr-x------ 2 root root 0 Sep   1 19:16 .
dr-xr-xr-x 9 root root 0 Sep   1 19:16 ..
lrwx------ 1 root root 64 Sep  1 19:16 0 -> /dev/pts/1
lrwx------ 1 root root 64 Sep  1 19:16 1 -> /dev/pts/1
lrwx------ 1 root root 64 Sep  1 19:16 2 -> /dev/pts/1
lrwx------ 1 root root 64 Sep  5 00:46 255 -> /dev/pts/1
```

Zauważysz, że jest to interaktywna sesja powłoki: jej standardowe wejście pochodzi z wirtualnego terminala (`/dev/pts/1`), a standardowe strumienie błędów i wyjścia wracają do tego samego terminala. To się zgadza.

Rzućmy okiem na edytor tekstu taki jak vim, który zachowuje się podobnie do terminala — wejście i wyjście odbywa się za pośrednictwem terminala. Istnieje jednak dodatkowa komplikacja, która polega na tym, że edytory tekstu zazwyczaj mają otwarty co najmniej jeden plik do zapisania. Jak to wygląda?

W tym przykładzie uruchamiam edytor tekstu vim i edytuję plik w katalogu */tmp*. Znajdźmy identyfikator procesu dla vim, skoro wiemy, do którego katalogu */proc* zajrzeć:

```
root@server:/# ps aux | grep vim
root       453  0.0  0.1   17232   9216 pts/1     S+     15:57     0:00 vim
/tmp/hello.txt
root       458  0.0  0.0   2884    1536 pts/0     S+     15:58     0:00 grep
--color=auto vim
```

I oto jest: proces `453`. Nie daj się zwieść poleceniu `grep`, które zawiera również `vim` w swoich argumentach. Skoro mamy PID, przyjrzyjmy się deskryptorom plików vima:

```
root@server:/# ls -l /proc/453/fd
total 0
lrwx------ 1 root root 64 Jan  7 15:58 0 -> /dev/pts/1
lrwx------ 1 root root 64 Jan  7 15:58 1 -> /dev/pts/1
lrwx------ 1 root root 64 Jan  7 15:58 2 -> /dev/pts/1
lrwx------ 1 root root 64 Jan  7 15:58 3 -> /tmp/.hello.txt.swp
```

Widzimy, że stdin (`0`), stdout (`1`) i stderr (`2`) wskazują na urządzenie terminala, podobnie jak na powłokę. Widzimy również, że edytor ma otwarty plik, z deskryptorem pliku `3`, dowiązanym do pliku, który edytuje vim. Gdy proces otwiera dodatkowe pliki, tworzone są nowe deskryptory plików i można je tutaj wyświetlić.

Poza tym, że jest to interesujące samo w sobie, może się przydać, gdy programy zachowują się nieprawidłowo z powodu błędów lub gdy próbujesz prześledzić, co robi potencjalnie szkodliwy program. Narzędzie `procfs` jest dość interesujące i przydatne, jeżeli poświęcisz trochę czasu na jego naukę: na początek wpisz po prostu **`man proc`** lub przeczytaj stronę wiki Arch Linux *https://wiki.archlinux.org/title/Procfs*, aby uzyskać łagodniejsze wprowadzenie do tego zagadnienia.

Podsumowanie

W tym rozdziale zebraliśmy wszystkie omówione wcześniej praktyczne umiejętności i teorię, aby odblokować jedną z najpotężniejszych funkcjonalności systemów Unix i Linux: przesyłanie danych przez wiele poleceń za pomocą potoków i przekierowywania wejścia i wyjścia.

Zaczęliśmy od pokazania, w jaki sposób system operacyjny udostępnia podstawowe elementy, takie jak deskryptory plików, a potem przedstawiliśmy praktyczne zastosowania przekierowywania danych wejściowych i wyjściowych. Następnie omówiliśmy potoki, które są prawdopodobnie jedną z najbardziej przydatnych funkcji Linuksa i innych uniksowych systemów operacyjnych. Po omówieniu niezbędnej teorii i pokazaniu kilku użytecznych przykładów przyjrzeliśmy się najpopularniejszym narzędziom pomocniczym, których programiści używają do okrajania i przycinania strumieni danych gromadzonych za pomocą potoków. Na koniec przedstawiliśmy kilka najczęstszych i najbardziej przydatnych wzorców i kombinacji programów używanych w prawdziwym świecie.

Materiał omówiony w tym rozdziale jest podstawą większości zaawansowanego użycia wiersza poleceń, z którym będziesz mieć styczność w codziennej pracy. Znasz już podstawowe koncepcje, narzędzia i wzorce, które napotkasz w środowisku naturalnym. Dzięki temu łatwiej Ci będzie zacząć budować niestandardowe polecenia stosowane w rozwoju oprogramowania, rozwiązywaniu problemów i automatyzacji zadań.

Aby rozwijać swoje umiejętności, wykorzystaj w codziennej pracy to, czego dowiedziałeś się podczas lektury tego rozdziału. Użyj go jako odniesienia dla wzorców do wypróbowywania i uczenia się nowych narzędzi i poleceń, które możesz dodawać do własnych receptur i stosować do filtrowania danych lub operowania nimi w wierszu poleceń. Już niedługo poczujesz się jak magik.


PROGRAM
PARTNERSKI
GRUPY HELION
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ
Zmień swoją stronę WWW w działający bankomat!
Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl
GRUPA
Helion

Linux bez tajemnic — Twoja przewaga w programowaniu!

To zaskakujące, ale wielu inżynierów oprogramowania wciąż nie czuje się dobrze w pracy z systemami uniksowymi. A przecież są one wszechobecne: od środowiska pracy (macOS), przez procesy tworzenia oprogramowania (kontenery Dockera), po narzędzia kompilacji i automatyzacji (potoki ciągłej integracji, GitHub), a także środowiska produkcyjne (serwery Linuksa, kontenery). Owszem, opanowanie pracy z Linuksem wymaga nieco wysiłku i czasu, szybko się jednak przekonasz, jak wiele przynosi korzyści!

Dzięki tej niezwykle praktycznej książce, napisanej z myślą o inżynierach oprogramowania, a nie administratorach Linuksa, zdobędziesz umiejętności, z których natychmiast skorzystasz w codziennych zadaniach programisty. Informacje teoretyczne ograniczono do niezbędnego minimum pozwalającego zrozumieć zasady pracy z wierszem poleceń. W ten sposób szybko nauczysz się sprawnie i wygodnie działać w środowisku uniksowym. Dowiesz się także, jak można zastosować te umiejętności w różnych kontekstach, takich jak tworzenie obrazów Dockera i praca z nimi, automatyzacja zadań za pomocą skryptów czy rozwiązywanie problemów w środowiskach produkcyjnych. Efekt? Zaoszczędzisz czas i staniesz się mistrzem wiersza poleceń!

W książce:

- działanie Linuxa i jego powłoki
- najużyteczniejsze sztuczki i narzędzia
- tworzenie potężnych narzędzi dostosowanych do konkretnych potrzeb
- efektywne metody pracy z Dockerem, SSH i wierszem poleceń
- wygodne wyszukiwanie danych w dziennikach zdarzeń
- radzenie sobie z typowymi sytuacjami w środowisku uniksowym, sprawiającymi trudności innym programistom

David Cohen od ponad 15 lat zajmuje się szeroko pojętą inżynierią oprogramowania i niezawodności systemów. Prowadzi w serwisie YouTube kanał tutoriaLinux, na którym uczy podstaw pracy z systemem Linux i podejścia DevOps.

Christian Sturm jest konsultantem oprogramowania i architektury systemów. Uczestniczy w wielu projektach open source.

Helion

helion.pl

HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-289-1754-5

9 788328 917545

Cena: 79,00 zł

packt