bash. Wprowadzenie

Autorzy: Cameron Newham, Bill Rosenblatt
Tłumaczenie: Daniel Kaczmarek (przedmowa, rozdz. 1–8, 10–12), Małgorzata Czart (rozdz. 9, dod. A–D)
ISBN: 83-246-0047-7
Tytuł oryginału: Learning the Bash Shell: 3rd Edition
Format: B5, stron: 344

Kompletny przewodnik po programowaniu powłoki Uniksa

- Konfigurowanie środowiska tekstowego
- Tworzenie skryptów powłoki
- Administrowanie powłoką bash

Powłoka to pierwszy element systemów uniksowych, z którym spotykają się użytkownicy. Pod nazwą „powłoka" kryje się tekstowy interfejs użytkownika – przez długi czas jedyny sposób komunikacji z systemem. Powłoki to samodzielne narzędzia, odseparowane od właściwego systemu. Z tego właśnie powodu dostępne są różne ich wersje. Dziś, mimo rozpowszechnienia środowisk graficznych, powłoki nadal są wykorzystywane. Dzięki nim można w prosty sposób zrealizować zadania związane z przetwarzaniem plików tekstowych i zawartych w nich danych. Znajomość zagadnień związanych z programowaniem powłoki, szczególnie tej najpopularniejszej – bash, może przydać się każdemu administratorowi Linuksa.

„bash. Wprowadzenie" to książka przedstawiająca tajniki najnowszej wersji powłoki bash (Bourne Again Shell). Przeznaczona jest zarówno dla tych użytkowników systemów uniksowych, którzy wykorzystują powłokę w charakterze interfejsu użytkownika, jak i dla tych, którzy stosują ją w roli narzędzia programistycznego. Niniejsza pozycja opisuje sposób instalowania i konfigurowania powłoki bash, jej zaawansowane mechanizmy, takie jak historia poleceń oraz zagadnienia związane z tworzeniem skryptów powłoki. Zawiera informacje dotyczące pisania programów oraz usuwania z nich błędów. Administratorzy systemów znajdą tu cenne porady związane z zarządzaniem powłoką bash na potrzeby użytkowników systemów.

- Podstawy pracy z powłoką bash
- Edytory emacs i vi
- Dostosowywanie środowiska do własnych potrzeb
- Definiowanie zmiennych powłoki
- Wyrażenia warunkowe
- Operacje wejścia i wyjścia
- Sterowanie działaniem procesów
- Usuwanie błędów ze skryptów
- Administrowanie powłoką bash

Każdy użytkownik Linuksa oraz innych systemów z rodziny Unix znajdzie tu bardzo wartościowe informacje.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Przedmowa **7**

1. Podstawy powłoki bash **15**
Czym jest powłoka? 16
Zakres książki 16
Historia powłok Uniksa 17
Uaktywnianie powłoki bash 19
Interaktywna praca z powłoką 20
Pliki 21
Operacje wejścia-wyjścia 28
Zadania drugoplanowe 31
Znaki specjalne i używanie cudzysłowów 34
Pomoc 40

2. Edytowanie w wierszu poleceń **41**
Włączanie edycji w wierszu poleceń 42
Historia poleceń 42
Tryb edycji emacs 43
Tryb edycji vi 50
Polecenie fc 59
Uzupełnianie historią 62
readline 63
Praca z klawiaturą 67

3. Dostosowywanie środowiska **69**
Pliki .bash_profile, .bash_logout i .bashrc 70
Aliasy 71
Opcje 74
Zmienne powłoki 76
Dostosowywanie i podprocesy 88
Wskazówki co do dostosowywania środowiska 93

4. Podstawy programowania powłoki 95
Skrypty i funkcje powłoki 95
Zmienne powłoki 99
Operatory ciągów znaków 104
Zastępowanie poleceniem 114
Przykłady zaawansowane: pushd i popd 118

5. Sterowanie przebiegiem 123
if/else 124
for 137
case 143
select 146
while i until 149

6. Opcje poleceń i zmienne o określonym typie 151
Opcje poleceń 151
Zmienne o określonym typie 159
Zmienne całkowitoliczbowe i działania arytmetyczne 160
Tablice 170

7. Operacje wejścia-wyjścia i przetwarzanie wiersza poleceń 175
Przekierowania wejścia-wyjścia 175
Operacje wejścia-wyjścia na ciągach znaków 181
Przetwarzanie wiersza poleceń 192

8. Obsługa procesów 209
Identyfikatory procesów i numery zadań 210
Kontrola zadań 210
Sygnały 214
trap 220
Procesy równoległe 225
Podpowłoki 229
Zastępowanie procesem 231

9. Debugowanie skryptów powłoki 233
Podstawowe narzędzia wspomagające debugowanie 233
Debuger dla powłoki bash 241

10. Administrowanie powłoką bash 259
Instalowanie powłoki bash jako powłoki standardowej 259
Dostosowywanie środowiska 261
Mechanizmy zabezpieczania systemu 266

11. Pisanie skryptów powłoki 269
Jak to działa? 269
Początek tworzenia skryptu 271
Potencjalne problemy 273
Kiedy nie używać powłoki bash 274

12. Uruchamianie powłoki bash w systemie 275
Pobieranie powłoki bash 275
Rozpakowywanie archiwum 275
Zawartość archiwum 276
Do kogo się zwrócić? 281

A Powłoki podobne 283
Powłoka Bourne'a 283
Standard powłoki 1003.2 POSIX 285
Powłoka Korn 287
pdksh 289
zsh 289
Klony powłoki i platformy podobne do Uniksa 290

B Listy sumaryczne 293
Uruchamianie 293
Możliwe modyfikacje znaku zachęty 295
Wbudowane polecenia i słowa zastrzeżone 296
Wbudowane zmienne powłoki 298
Operatory testowe 302
Opcje polecenia set 303
Opcje polecenia shopt 305
Przekierowanie wejścia-wyjścia 306
Polecenia edycji trybu emacs 308
Polecenia trybu kontrolnego vi 310

C Ładowalne funkcje wbudowane 313

D Programowalne uzupełnianie 319

Skorowidz 325

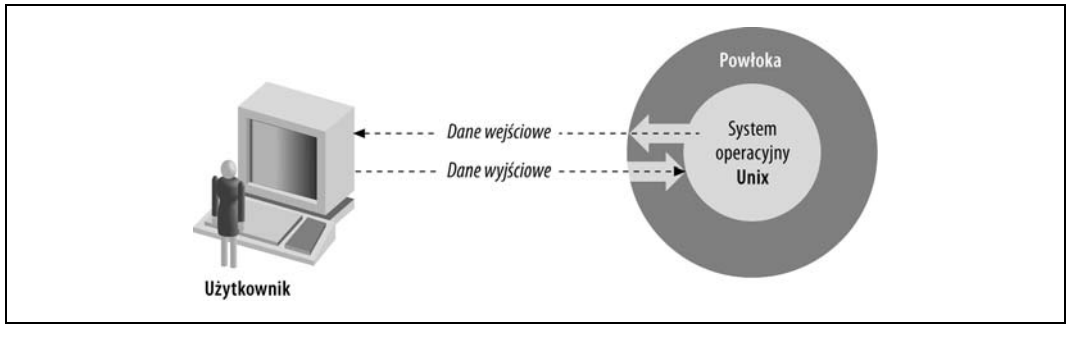
ROZDZIAŁ 1.

Podstawy powłoki bash

Popularność systemu Unix stale wzrasta od chwili jego powstania, to jest od wczesnych lat 70. W tym czasie powstało wiele różnych wersji systemu, noszących takie nazwy jak Ultrix, AIX, Xenix, Unos i Linux. Początkowo Unix był wykorzystywany na minikomputerach i komputerach mainframe, jednak z czasem zaczęto go instalować również na biurkowych stacjach roboczych, a nawet na komputerach osobistych wykorzystywanych w domu i w pracy. Unix nie jest już użytkowany jedynie przez komputerowych zapaleńców na uniwersytetach i w centrach naukowych. Jest także wykorzystywany w biznesie, szkołach i w domach. Z biegiem czasu coraz więcej osób będzie miało styczność z Uniksem.

Czytelnik mógł już używać systemu Unix w szkole, biurze albo w domu do uruchamiania aplikacji, drukowania dokumentów i odczytywania poczty elektronicznej. Mało kto jednak zastanawia się nad mechanizmami uruchamianymi po wpisaniu polecenia i naciśnięciu klawisza *RETURN*.

Po wpisaniu polecenia przetwarzanie odbywa się w różnych warstwach systemu, jednak w niniejszej książce zajmiemy się tylko warstwą najwyższego poziomu, którą jest **powłoka**. Ogólnie mówiąc, powłoka to dowolny interfejs użytkownika do systemu operacyjnego Unix, czyli każdy program, który pobiera dane wejściowe od użytkownika, tłumaczy je na instrukcje zrozumiałe dla systemu operacyjnego, a także przekazuje użytkownikowi dane wyjściowe wygenerowane przez system operacyjny. Na rysunku 1.1 przedstawiono relacje między użytkownikiem, powłoką i systemem operacyjnym.



Rysunek 1.1. Powłoka jest warstwą otaczającą system operacyjny Unix

Istnieje szereg różnych typów interfejsów użytkownika. Powłoka *bash* należy do najbardziej popularnej kategorii: znakowych interfejsów użytkownika. Tego typu interfejsy przyjmują tekstowe wiersze poleceń wpisywanych przez użytkownika i zwykle generują dane wyjściowe również w postaci tekstowej. Do innych kategorii zalicza się coraz bardziej popularne **graficzne interfejsy użytkownika** (GUI), które pozwalają na wyświetlanie określonych elementów graficznych (a nie tylko graficznego obrazu liter) i przyjmują dane wejściowe generowane przez mysz lub inne urządzenia wskazujące, interfejsy w postaci ekranów dotykowych (obecne na przykład w niektórych rodzajach bankomatów) itp.

Czym jest powłoka?

Zadaniem powłoki jest tłumaczenie wierszy poleceń użytkownika na instrukcje systemu operacyjnego. Spójrzmy na poniższe polecenie:

```
sort -n lista > lista.posortowana
```

Polecenie to oznacza: „Posortuj w porządku liczbowym wiersze znajdujące się w pliku *lista* i umieść wyniki w pliku o nazwie *lista.posortowana*". Oto czynności wykonywane przez powłokę w odpowiedzi na to polecenie:

1. Podział wiersza na następujące fragmenty: `sort`, `-n`, `lista`, `>` i `lista.posortowana`. Fragmenty te to tak zwane słowa.
2. Ustalenie znaczenia poszczególnych słów: `sort` jest poleceniem, `-n` i `lista` to argumenty, a `>` i `lista.posortowana` złączone razem stanowią instrukcję wejścia-wyjścia.
3. Ustawienie wejścia-wyjścia zgodnie z zapisem `> lista.posortowana` (przesłanie wyników do pliku *lista.posortowana*) oraz kilka wewnętrznych, standardowych kroków.
4. Odnalezienie pliku z poleceniem `sort` i jego uruchomienie z opcją `-n` (porządek liczbowy) oraz argumentem *lista* (nazwa pliku wejściowego).

Oczywiście, każdy z tych kroków składa się z kilku pomniejszych czynności, a każda z nich zawiera konkretną instrukcję systemu operacyjnego.

Należy pamiętać, że sama powłoka nie jest systemem Unix — jest to jedynie interfejs do systemu. Unix jest jednym z pierwszych systemów operacyjnych, w których interfejs użytkownika jest całkowicie niezależny od samego systemu.

Zakres książki

W niniejszej książce została opisana powłoka *bash*, będąca jedną z najnowszych i najbogatszych spośród najpopularniejszych powłok Uniksa. Powłoki *bash* można używać na dwa sposoby: jako interfejsu użytkownika oraz jako środowiska programistycznego.

Ten i następny rozdział dotyczą interaktywnej pracy z powłoką. Informacje zawarte w obydwu rozdziałach powinny dać wystarczającą podstawę do sprawnego i wydajnego używania powłoki w zakresie najczęściej wykonywanych zadań.

Używanie powłoki niewątpliwie pozwoli na zidentyfikowanie niektórych charakterystycznych cech środowiska, które warto będzie dostosować, oraz zadań, które można zautomatyzować. Kilka sposobów osiągnięcia tych celów zostanie przedstawionych w rozdziale 3.

Rozdział 3. wprowadza również do zagadnień związanych z programowaniem powłoki, które szczegółowo przedstawiane są w rozdziałach od 4. do 6. Zrozumienie prezentowanych zagadnień i nauczenie się programowania powłoki nie wymaga żadnego doświadczenia programistycznego. W rozdziałach 7. i 8. znajdują się szczegółowe informacje na temat operacji wejścia-wyjścia powłoki oraz mechanizmów obsługi procesów, natomiast w rozdziale 9. opisywane są różne techniki debugowania programów powłoki.

Z niniejszej książki można dowiedzieć się bardzo wiele na temat powłoki *bash*, a także na temat narzędzi Uniksa oraz sposobu, w jaki ten system działa. Można nawet zostać doskonałym programistą powłoki i to bez żadnego wcześniejszego doświadczenia programistycznego. Jednocześnie staraliśmy się zanadto nie zagłębiać w szczegóły wewnętrznych mechanizmów Uniksa. Stoimy bowiem na stanowisku, że wcale nie trzeba szczegółowo znać wewnętrznych mechanizmów systemu, by móc efektywnie używać jego powłoki i programować ją. Pominięte zostanie zatem kilka funkcji powłoki, które przeznaczone są wyłącznie dla programistów korzystających z niskopoziomowych mechanizmów systemu.

Historia powłok Uniksa

Niezależność powłoki od systemu operacyjnego spowodowała, że na przestrzeni lat powstało wiele różnych powłok Uniksa. Jednak tylko kilka z nich jest powszechnie używanych.

Pierwszą ważną powłoką była powłoka Bourne (nazwana tak od nazwiska jej twórcy, Stevena Bourne'a), dołączona do pierwszej ogólnodostępnej wersji Uniksa, wersji 7, w 1979 roku. W systemie powłoka Bourne nosi oznaczenie *sh*. Sam Unix przeszedł wiele zmian, lecz powłoka Bourne wciąż cieszy się popularnością i zasadniczo się nie zmienia. Korzystają z niej różne narzędzia i mechanizmy administracyjne Uniksa.

Następną powszechnie używaną powłoką była powłoka C, czyli *csh*, którą napisał Bill Joy z Uniwersytetu Kalifornia w Berkeley. Stanowiła ona część Berkeley Software Distribution (BSD) — wersji systemu Unix opublikowanej kilka lat po pojawieniu się wersji 7.

Powłoka C zawdzięcza swą nazwę podobieństwu udostępnianych poleceń do instrukcji języka programowania C, co bardzo ułatwia jej poznanie programistom korzystającym z systemu Unix. Powłoka ta obsługuje szereg funkcji (na przykład sterowanie zadaniami — więcej na ten temat w rozdziale 8.) charakterystycznych wówczas tylko dla systemu BSD Unix, które jednak są teraz obecne w większości bieżących wersji Uniksa. Posiada także kilka istotnych właściwości (takich jak aliasy — więcej na ten temat w rozdziale 3.) ułatwiających jej stosowanie.

W ostatnich latach popularność zdobyło kilka innych powłok. Najważniejszą z nich jest powłoka Korn. Jest to produkt komercyjny, łączący w sobie najlepsze cechy powłok Bourne oraz C i zawierający wiele rozwiązań autorskich[1]. Powłoka Korn w wielu aspektach przypomina powłokę *bash* — na przykład obydwie oferują cały zbiór możliwości ułatwiających pracę z nimi. Powłoka *bash* jest jednak rozprowadzana na licencji GPL (General Public License). Więcej informacji na temat powłoki Korn znajduje się w dodatku A).

[1] Powłokę Korn można pobrać za darmo, lecz warunki licencji wymagają uiszczenia zapłaty w przypadku wykorzystywania jej w pewnych sytuacjach.

Powłoka Bourne Again

Powłoka Bourne Again (nazwana tak dla upamiętnienia powłoki Steve'a Bourne'a) została stworzona dla celów projektu GNU[2]. Projekt GNU został zainicjowany przez Richarda Stallmana z fundacji Free Software Foundation (FSF), a jego celem jest utworzenie systemu operacyjnego zgodnego z Uniksem i zastępowanie wszystkich komercyjnych narzędzi Uniksa ich otwartymi odpowiednikami. Projekt GNU to jednak nie tylko nowe programy narzędziowe, ale również nowy sposób dystrybucji, tzw. **copyleft**. Oprogramowanie dystrybuowane na zasadzie copyleft może być udostępniane bez ograniczeń, jeśli tylko nie są nakładane żadne ograniczenia co do jego dalszego rozprowadzania (na przykład kod źródłowy musi być ogólnodostępny).

Powłoka *bash*, która została pomyślana jako standardowa powłoka systemu GNU, ujrzała światło dzienne w niedzielę 10 stycznia 1988 roku. Brian Fox napisał pierwotną wersję powłoki *bash* i interfejsu *readline*, po czym rozbudowywał powłokę aż do roku 1993. Na początku 1989 roku dołączył do niego Chet Ramey, którego zadaniem było usunięcie błędów występujących wówczas w powłoce i dołączenie wielu nowych użytecznych udogodnień. Obecnie Chet Ramey jest oficjalnie odpowiedzialny za utrzymanie powłoki *bash* i jej dalszą rozbudowę.

Zgodnie z zasadami GNU wszystkie wersje powłoki *bash* od wersji 0.99 wzwyż są udostępniane za darmo przez FSF. Powłoka *bash* znalazła się we wszystkich ważniejszych odmianach Uniksa i obecnie szybko staje się najpopularniejszą powłoką wywodzącą się od powłoki Bourne. Jednocześnie jest to standardowa powłoka dołączana do Linuksa — szeroko wykorzystywanego darmowego systemu operacyjnego z rodziny Unix — oraz do systemu Mac OS X firmy Apple.

W 1995 roku Chet Ramey rozpoczął pracę nad nową wersją główną, 2.0, która została opublikowana po raz pierwszy 23 grudnia 1996 roku. W powłoce *bash* 2.0 znalazł się cały szereg funkcji nieobecnych we wcześniejszej wersji (oznaczonej numerem 1.14.7), zwiększył się także zakres jej zgodności z różnymi standardami. Powłoka *bash* 3.0 stanowi dalsze ulepszenie poprzednich wersji i dalej rozszerza zakres dostępnych możliwości i zgodność ze standardami.

Niniejsza książka opisuje powłokę *bash* 3.0 i jest zgodna również z wszystkimi wcześniejszymi wersjami. W tekście zostaną wskazane wszystkie mechanizmy obecne w bieżącej wersji, które różnią się od mechanizmów we wcześniejszych wersjach lub w ogóle nie są w nich dostępne.

Cechy powłoki

Wprawdzie to powłoka Bourne wciąż określana jest mianem „standardowej", lecz *bash* zdobywa coraz większą popularność. Oprócz zgodności z powłoką Bourne *bash* oferuje najlepsze rozwiązania zastosowane w powłokach C i Korn, a także kilka rozwiązań charakterystycznych tylko dla niej.

Cechą, która najbardziej przekonuje użytkowników do powłoki *bash*, są tryby edycji w wierszu poleceń. Dzięki możliwości edycji w wierszu poleceń znacznie łatwiej jest się cofnąć i poprawić popełnione błędy albo zmodyfikować polecenia wpisane wcześniej, niż ma to miejsce w przypadku mechanizmu historii powłoki C. Z kolei powłoka Bourne w ogóle nie pozwala na wykonywanie takich działań.

[2] GNU to rekurencyjny akronim wyrażenia „GNU's Not Unix".

Kolejną ważną funkcją powłoki *bash*, opracowaną głównie z myślą o użytkownikach interaktywnych, jest sterowanie zadaniami. Jak zostanie wyjaśnione w rozdziale 8., sterowanie zadaniami pozwala na zatrzymywanie, uruchamianie i wstrzymywanie większej liczby poleceń w tym samym czasie. Właściwość ta została niemal bez zmian zapożyczona z powłoki C.

Pozostałe najważniejsze zalety powłoki *bash* dotyczą przede wszystkim osób przywiązujących wagę do możliwości jej dostosowywania oraz programistów. Powłoka ta udostępnia wiele nowych opcji i zmiennych dostosowujących, a zawarte w niej mechanizmy programistyczne zostały znacznie rozszerzone, między innymi o definiowanie funkcji, dodatkowe struktury sterujące, arytmetykę całkowitoliczbową i zaawansowane sterowanie wejściem-wyjściem.

Uaktywnianie powłoki bash

Być może czytelnik ma od razu możliwość korzystania z powłoki *bash*. Administrator systemu zwykle konfiguruje konto użytkownika w taki sposób, że udostępnia ono powłokę mającą w systemie status „standardowej". Użytkownik może nawet nie zdawać sobie sprawy, że dostępna jest więcej niż jedna powłoka.

Na szczęście bardzo łatwo można ustalić, jaka powłoka jest aktualnie używana. W tym celu należy zalogować się do systemu i w wierszu poleceń wpisać polecenie `echo $SHELL`. W odpowiedzi wyświetlona zostanie odpowiedź `sh`, `csh`, `ksh` lub `bash`, oznaczająca odpowiednio powłoki Bourne, C, Korn i *bash*. (Może się także okazać, że używana jest jeszcze inna powłoka, na przykład *tcsh*).

Jeżeli powłoka *bash* nie jest dostępna, wówczas w celu jej uaktywnienia trzeba najpierw sprawdzić, czy w ogóle jest obecna w systemie. W tym celu wystarczy wpisać `bash`. Jeżeli znak zachęty zmieni się i przyjmie postać jakiejś informacji zakończonej znakiem dolara (na przykład `bash3 $`), będzie to oznaczać, że wszystko jest w porządku. Wystarczy wtedy wpisać polecenie `exit`, aby wrócić do zwykłej powłoki.

Jeśli natomiast wyświetlony zostanie komunikat „not found", może to oznaczać, że powłoka nie występuje w systemie. Wtedy trzeba się zwrócić do administratora systemu albo innego doświadczonego użytkownika. Istnieje jeszcze szansa, że jakaś wersja powłoki *bash* jest zainstalowana w takim miejscu w systemie (katalogu), które nie jest dostępne dla bieżącego użytkownika. Jeśli nie, należy przejść do rozdziału 11. i sprawdzić, w jaki sposób można pobrać powłokę *bash*.

Gdy będzie już pewne, że powłoka *bash* jest dostępna w systemie, można ją wywołać z poziomu dowolnej innej powłoki, tak jak poprzednio wpisując polecenie `bash`. Znacznie lepiej jednak jest zainstalować ją jako **powłokę logowania**, to znaczy jako powłokę udostępnianą automatycznie w momencie zalogowania się. W tym celu może być konieczne samodzielne przeprowadzenie instalacji. Poniżej znajdują się instrukcje, które powinny okazać się skuteczne w niemal wszystkich systemach Unix. Jeżeli coś pójdzie nie tak (na przykład po wpisaniu jakiegoś polecenia w odpowiedzi pojawi się komunikat błędu „not found" lub pusty wiersz), trzeba będzie przerwać proces i zwrócić się do administratora systemu. Można też zajrzeć do rozdziału 12., w którym prezentowany jest nieco bardziej skomplikowany sposób zastępowania bieżącej powłoki.

Najpierw trzeba ustalić, gdzie w systemie znajduje się powłoka, to znaczy w którym katalogu ją zainstalowano. W znalezieniu odpowiedniej lokalizacji pomocne może okazać się polecenie `whereis bash` (dotyczy to zwłaszcza powłoki C); jeśli jednak to rozwiązanie się nie sprawdzi, można wypróbować polecenia `whence bash`, `which bash` albo wpisać bardziej złożone polecenie[3]:

```
grep bash /etc/passwd | awk -F: '{print $7}' | sort -u
```

Zwrócona odpowiedź powinna mieć postać taką jak */bin/bash* albo */usr/local/bin/bash*.

Aby ustawić powłokę *bash* jako powłokę logowania, należy wpisać polecenie `chsh` *`nazwa_powłoki`*, gdzie *`nazwa_powłoki`* powinna zostać zastąpiona odpowiedzią wygenerowaną przez polecenie `whereis` (albo inne, które okazało się skuteczne). Na przykład:

```
% chsh /usr/local/bin/bash
```

W odpowiedzi zwrócony zostanie komunikat o błędzie informujący, że powłoka jest nieprawidłowa, albo użytkownik zostanie poproszony o podanie hasła[4]. Należy wówczas wpisać hasło, po czym wylogować się i zalogować ponownie, by móc zacząć korzystać z *bash*.

Interaktywna praca z powłoką

Interaktywna praca z powłoką odbywa się w trakcie sesji logowania, rozpoczynającej się w chwili zalogowania i kończącej w momencie wpisania polecenia `exit` lub `logout` albo naciśnięcia *CTRL+D*[5]. W trakcie sesji logowania wpisuje się **wiersze poleceń** przeznaczonych dla powłoki. Są to wiersze tekstu zakończone naciśnięciem klawisza *RETURN*, wpisywane na terminalu lub stacji roboczej.

Domyślnie symbol zachęty prezentowany przez powłokę dla kolejnych poleceń ma postać informacji zakończonej znakiem dolara. Jak się jednak przekonamy w trakcie lektury rozdziału 3., cały symbol zachęty można zmienić.

Polecenia, argumenty i opcje

Wiersze poleceń powłoki składają się z jednego lub więcej słów oddzielonych spacjami lub znakami tabulacji. Pierwsze słowo w wierszu to **polecenie**, a pozostałe (jeżeli występują) stanowią **argumenty** (nazywane również **parametrami**) tego polecenia, będące nazwami elementów, na które polecenie ma działać.

Na przykład wiersz polecenia `lp mójplik` składa się z polecenia `lp` (drukowanie pliku) i pojedynczego argumentu `mójplik`. Polecenie `lp` traktuje `mójplik` jako nazwę pliku, którego zawartość ma być wydrukowana. Często argumentami są właśnie nazwy pliku, jednak nie zawsze. Na przykład w wierszu polecenia `mail cam` program *mail* potraktuje argument `cam` jako nazwę użytkownika, do którego ma zostać wysłany komunikat.

---

[3] Należy się upewnić, że w poleceniu tym zostanie zastosowany odpowiedni apostrof: ' zamiast `.

[4] Z uwagi na bezpieczeństwo systemu tylko niektóre programy można instalować jako powłoki logowania.

[5] Powłokę można skonfigurować w taki sposób, aby ignorowała pojedyncze naciśnięcie klawiszy *CTRL+D*, nie kończąc sesji. Zalecamy stosowanie takiego rozwiązania, ponieważ *CTRL+D* często jest naciskany przypadkowo. Więcej informacji na ten temat znajduje się w rozdziale 3., w punkcie dotyczącym opcji.

Opcja to specjalny typ argumentu, który stanowi dla polecenia informację, jaka czynność ma zostać wykonana. Opcje zwykle składają się z myślnika i litery. Mówimy „zwykle", ponieważ jest to raczej konwencja, a nie reguła, od której nie ma odstępstwa. Polecenie `lp -h mójplik` zawiera opcję `-h`, która zakazuje poleceniu `lp` drukowania „strony tytułowej" przed wydrukowaniem zawartości pliku.

Czasami opcje przyjmują własne argumenty. Na przykład polecenie `lp -d lp1 -h mójplik` zawiera dwie opcje i jeden argument. Pierwszą opcją jest `-d lp1`, co oznacza „prześlij dane wynikowe do urządzenia `lp1`". Druga opcja oraz argument są takie same jak w poprzednim przykładzie.

Pliki

Argumentami poleceń nie zawsze są pliki, lecz to właśnie pliki są najważniejszym elementem każdego systemu Unix. Plik może zawierać dowolny rodzaj informacji, poza tym pliki występują w różnych odmianach. Najważniejsze są następujące trzy rodzaje plików:

Zwykłe pliki
: Nazywane również plikami tekstowymi, ponieważ zawierają czytelne znaki. Na przykład niniejsza książka powstała z kilku zwykłych plików zawierających tekst oraz czytelne dla człowieka instrukcje formatujące, przeznaczone dla edytora tekstu *troff*.

Pliki wykonywalne
: Nazywane również programami, są wywołane w postaci poleceń. Niektóre z nich są nieczytelne dla człowieka, inne natomiast — skrypty powłoki, którymi będziemy zajmować się w dalszej części książki — to specjalne pliki tekstowe. Sama powłoka jest nieczytelnym dla człowieka plikiem wykonywalnym o nazwie *bash*.

Katalogi
: Są one czymś w rodzaju folderów zawierających pliki, w tym również inne katalogi (zwane **podkatalogami**).

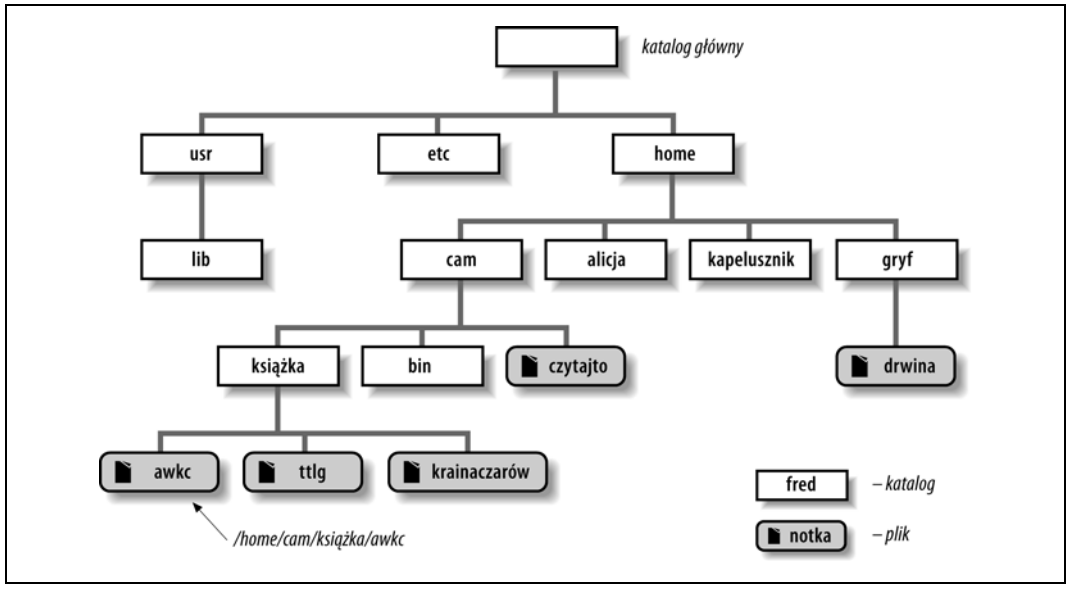
Katalogi

W niniejszym punkcie zostaną opisane najważniejsze zagadnienia dotyczące katalogów. Katalogi mogą zawierać inne katalogi, co prowadzi do powstania hierarchicznej struktury wszystkich plików systemu Unix, zwanej również **drzewem**.

Na rysunku 1.2 przedstawiono fragment standardowego drzewa katalogów. Prostokąty symbolizują katalogi, a elementy owalne to zwykłe pliki.

Szczyt drzewa to katalog główny, **root**, który nie ma nazwy w systemie[6]. Wszystkie pliki można zidentyfikować za pomocą nazwy określającej ich lokalizację względem katalogu głównego. Nazwy takie powstają poprzez wskazanie nazw wszystkich katalogów (kolejno, począwszy od katalogu głównego), oddzielonych od siebie znakiem ukośnika (`/`), i są zakończone nazwą samego pliku. Tak skonstruowana nazwa to tak zwana **pełna** (albo **bezwzględna**) **ścieżka dostępu**.

[6] Większość samouczków Uniksa podaje, że nazwą katalogu *root* jest `/`. Pozostaniemy jednak przy naszym stwierdzeniu o braku nazwy, ponieważ jest ono pod względem logicznym bardziej spójne z innymi konwencjami nazywania plików w Uniksie.



Rysunek 1.2. Drzewo katalogów i plików

Załóżmy na przykład, że istnieje plik *awkc* znajdujący się w katalogu *książka*, ten zaś znajduje się w katalogu *cam* należącym do katalogu *home*, z kolei ten znajduje się w katalogu głównym. Pełną ścieżką dostępu do tego pliku będzie zatem */home/cam/książka/awkc*.

Katalog roboczy

Oczywiście, konieczność wskazywania za każdym razem pełnych ścieżek dostępu do pliku byłaby irytująca. Na szczęście istnieje tak zwany katalog roboczy (nazywany czasami katalogiem bieżącym) — katalog, w którym „jest się" w danym momencie. Jeżeli ścieżka dostępu nie będzie rozpoczynać się znakiem ukośnika, wówczas lokalizacja pliku zostanie ustalona względem katalogu roboczego. Tak skonstruowane nazwy to tak zwane **względne** ścieżki dostępu, wykorzystywane znacznie częściej niż pełne ścieżki dostępu.

Po zalogowaniu się do systemu katalogiem roboczym staje się specjalny katalog zwany katalogiem **domowym** (lub **logowania**). Administratorzy systemu często konfigurują go w taki sposób, aby nazwa katalogu domowego każdego użytkownika była taka sama jak nazwa tego użytkownika i aby wszystkie katalogi domowe znajdowały się we wspólnym katalogu będącym podkatalogiem katalogu głównego.

Na przykład typowym katalogiem domowym jest */home/cam*. Jeżeli będzie on jednocześnie katalogiem roboczym i wpisane zostanie polecenie `lp notes`, wówczas system poszuka pliku *notes* w katalogu */home/cam*. Jeżeli w tym katalogu domowym znajduje się katalog o nazwie *kapelusznik* zawierający plik o nazwie *herbatka*, wówczas zawartość pliku można wydrukować poleceniem `lp kapelusznik/herbatka`.

Zapis ze znakiem tyldy

Nietrudno się domyślić, że katalogi domowe często występują w ścieżkach dostępu. Wprawdzie większość systemów jest tak zorganizowana, aby wszystkie katalogi domowe miały wspólnego rodzica (na przykład */home* albo */users*), lecz nie można za każdym razem przyjmować takiego założenia ani nawet nie powinna istnieć konieczność wskazywania bezwzględnej ścieżki dostępu do czyjegoś katalogu domowego.

Z tego powodu powłoka *bash* pozwala na stosowanie skrótowej nazwy katalogów domowych — w tym celu wystarczy poprzedzić nazwę użytkownika znakiem tyldy (~). Można więc na przykład odwoływać się do pliku *książka* znajdującego się w katalogu domowym użytkownika o nazwie `alicja`, stosując zapis *~alicja/książka*. Jest to bezwzględna ścieżka dostępu, dlatego nie ma znaczenia rzeczywista nazwa katalogu domowego. Jeżeli katalog domowy użytkownika *alicja* zawiera podkatalog o nazwie *przygoda* i to właśnie w nim znajduje się plik, wówczas jako ścieżkę dostępu można podać *~alicja/przygoda/bajka*.

Jeszcze większą wygodę zapewnia to, że tylda sama odwołuje się do katalogu domowego użytkownika. Do pliku o nazwie *notes* znajdującego się w katalogu domowym można się odwołać, wpisując *~/notes* (zwróćmy uwagę na różnicę między tym zapisem a zapisem *~notes*, który zostałby zinterpretowany przez powłokę jako katalog domowy użytkownika o nazwie `notes`). Jeśli notes znajduje się w podkatalogu *przygoda*, wówczas można go wywoływać jako *~/przygoda/notes*. Zapis taki jest przydatny w sytuacji, gdy katalog roboczy nie znajduje się w drzewie katalogu domowego, to znaczy gdy jest nim na przykład jakiś katalog systemowy, na przykład */tmp*.

Zmiana katalogów roboczych

Aby zmienić katalog roboczy, należy użyć polecenia `cd`. Jeżeli użytkownik zapomni nazwy aktualnego katalogu roboczego, system wyświetli go po wpisaniu polecenia `pwd`.

Argumentem polecenia `cd` jest nazwa katalogu, który ma stać się katalogiem roboczym. Może to być katalog względny wobec katalogu bieżącego, może zawierać tyldę albo być określony ścieżką bezwzględną (rozpoczynającą się od znaku ukośnika). Jeżeli argument zostanie pominięty, `cd` katalogiem roboczym ustanowi katalog domowy (to znaczy jego działanie będzie takie samo jak działanie polecenia `cd ~`).

W tabeli 1.1 przedstawiono przykładowe polecenia `cd`. W przypadku każdego z nich przyjęto założenie, że katalogiem roboczym jest */home/cam* i że struktura katalogu jest taka sama jak na rysunku 1.2.

Tabela 1.1. Przykładowe polecenia cd

Polecenie	Nowy katalog roboczy
`cd książka`	*/home/cam/książka*
`cd książka/krainaczarów`	*/home/cam/książka/krainaczarów*
`cd ~/książka/krainaczarów`	*/home/cam/książka/krainaczarów*
`cd /usr/lib`	*/usr/lib/*
`cd ..`	*/home*
`cd ../gryf`	*/home/gryf*
`cd ~gryf`	*/home/gryf*

Pierwsze cztery polecenia mają prostą konstrukcję. W dalszych dwóch użyto specjalnego katalogu o nazwie .. (dwie kropki), który oznacza rodzica bieżącego katalogu. Istnieje on dla każdego katalogu, a zapis ten jest standardowym sposobem przechodzenia do katalogu znajdującego się w hierarchii katalogów nad katalogiem bieżącym, czyli do tak zwanego katalogu rodzica[7].

Kolejną ciekawą odmianą polecenia `cd` powłoki *bash* jest `cd -`, powodujące przejście do katalogu, który był katalogiem bieżącym przed przejściem do obecnego katalogu bieżącego. Na przykład, jeżeli najpierw katalogiem bieżącym będzie */usr/lib*, następnie wpisane zostanie polecenie `cd` bez argumentu, powodujące przejście do katalogu domowego, po czym zostanie wydane polecenie `cd -`, wówczas użytkownik zostanie przeniesiony z powrotem do katalogu */usr/lib*.

Rozwijanie symbolami wieloznacznymi nazw plików i ścieżek dostępu

Czasami istnieje potrzeba uruchomienia polecenia jednocześnie na więcej niż jednym pliku. Najprostszym przykładem takiego polecenia jest `ls`, prezentujące listę informacji na temat plików. W najprostszej postaci — to znaczy bez opcji i argumentów — polecenie to wypisuje nazwy wszystkich plików znajdujących się w katalogu roboczym oprócz specjalnych plików ukrytych, których nazwy zaczynają się znakiem kropki (.).

Jeżeli w poleceniu `ls` podane zostaną argumenty w postaci nazw plików, wówczas zostaną wyświetlone informacje o wskazanych plikach. Takie działanie nie ma jednak większego sensu, jeśli bowiem w katalogu bieżącym znajdują się pliki *księżna* i *królowa* i zostanie wpisane polecenie `ls księżna królowa`, wówczas system po prostu zwróci nazwy tych plików.

Tak naprawdę `ls` stosuje się częściej wraz z opcjami wskazującymi, jakie informacje mają być wyświetlone. Na przykład opcja `-l` (`long`) nakazuje poleceniu `ls` wypisanie właściciela pliku, jego rozmiaru, czasu ostatniej modyfikacji oraz innych informacji, zaś opcja `-a` (`all`) powoduje wypisanie również wcześniej wspomnianych plików ukrytych. Czasami jednak trzeba sprawdzić istnienie określonej grupy plików, gdy nie zna się nazw ich wszystkich — na przykład, jeśli używany jest edytor tekstu, konieczne może być sprawdzenie, które pliki kończą się literami *.txt*.

Nazwy plików są w Uniksie na tyle istotne, że powłoka udostępnia wbudowany mechanizm pozwalający zdefiniować wzorzec zbioru nazw plików bez konieczności podawania ich konkretnych nazw. W tym celu w nazwach plików stosuje się specjalne znaki, tak zwane symbole **wieloznaczne**, służące do definiowania wzorców nazw plików. W tabeli 1.2 wymieniono podstawowe symbole wieloznaczne.

Symbol wieloznaczny `?` oznacza dowolny pojedynczy znak, dlatego jeśli katalog zawiera pliki *program.c*, *program.log* i *program.o*, wówczas wyrażenie `program.?` będzie pasowało do *program.c* i *program.o*, ale już nie do *program.log*.

[7] Każdy katalog zawiera również specjalny katalog . (jedna kropka), oznaczający po prostu „ten katalog". Zatem polecenie `cd .` nie da tak naprawdę żadnego efektu. Katalogi . i .. są w istocie specjalnymi plikami ukrytymi obecnymi w każdym katalogu, które wskazują odpowiednio na sam katalog oraz katalog rodzica. Katalog główny jest jednocześnie swoim własnym rodzicem.

Tabela 1.2. Podstawowe symbole wieloznaczne

Symbol wieloznaczny	Oznacza
`?`	`Dowolny pojedynczy znak.`
`*`	`Dowolny ciąg znaków.`
`[`*`zbiór`*`]`	Dowolny znak w *`zbiorze`*.
`[!`*`zbiór`*`]`	Dowolny znak, który **nie znajduje się** w *`zbiorze`*.

Znak gwiazdki (*) daje większe możliwości i dlatego jest używany znacznie częściej. Symbolizuje on dowolny ciąg znaków. Wyrażenie `program.*` będzie pasować do wszystkich trzech plików wspomnianych w poprzednim akapicie. Użytkownicy edytora tekstu mogą więc wykorzystać wyrażenie `*.txt` do odnalezienia plików wejściowych[8].

Tabela 1.3 ilustruje działanie symbolu gwiazdki. Załóżmy, że w katalogu roboczym znajdują się pliki o nazwach *robert*, *daniel*, *dawid*, *ed*, *franek* i *fred*.

Tabela 1.3. Stosowanie symbolu wieloznacznego `*`

Wyrażenie	Pasuje do
`fr*`	*franek fred*
`*ed`	*ed fred*
`r*`	*robert*
`*e*`	*robert daniel ed franek fred*
`*r*`	*robert franek fred*
`*`	*robert daniel dawid ed franek fred*
`d*l`	*daniel*
`g*`	*g**

Zauważmy, że symbol * pasuje również do znaku nieistniejącego: wyrażenia `*ed` i `*e*` pasują do nazwy pliku *ed*. Zwróćmy również uwagę na ostatni przykład, ilustrujący reakcję powłoki, gdy nie zostaną odnalezione żadne pasujące nazwy: zwracane jest wówczas podane wyrażenie w niezmienionej postaci.

Ostatni symbol wieloznaczny to **zbiór**. Zbiór jest listą znaków (np. abc), zakresem (np. a – z) albo połączeniem obu. Jeżeli częścią listy ma być myślnik, trzeba go wpisać jako pierwszy lub ostatni znak zbioru. Tabela 1.4 powinna rozjaśnić działanie tego symbolu wieloznacznego.

Pozostając przy początkowym przykładzie stosowania symboli wieloznacznych — wyrażenia `program.[co]` i `program.[a-z]` będą pasować do nazw plików *program.c* i *program.o*, ale już nie do *program.log*.

[8] Użytkownicy systemów MS-DOS i VAX/VMS powinni zwrócić uwagę, że znak kropki (`.`) w nazwach plików Uniksa nie jest **niczym specjalnym** (nie licząc kropki występującej na początku, która powoduje „ukrycie" pliku) — jest to po prostu jeden z dostępnych znaków. Na przykład polecenie `ls *` spowoduje wyświetlenie wszystkich plików znajdujących się w bieżącym katalogu, nie trzeba więc stosować zapisu `*.*`, jak w innych systemach. Tak naprawdę `ls *.*` nie wyświetli wszystkich plików, a jedynie te, których nazwa zawiera w środku co najmniej jeden znak kropki.

Tabela 1.4. Stosowanie symbolu wieloznacznego w postaci zbioru

Wyrażenie	Oznacza
`[abc]`	a, b lub c.
`[.,;]`	Kropka, przecinek lub średnik.
`[-_]`	Myślnik lub podkreślenie.
`[a-c]`	a, b lub c.
`[a-z]`	Wszystkie małe litery.
`[!0-9]`	Wszystkie znaki, które nie są cyframi.
`[0-9!]`	Wszystkie cyfry i znak wykrzyknika.
`[a-zA-Z]`	Wszystkie małe i wielkie litery.
`[a-zA-Z0-9_-]`	Wszystkie litery, wszystkie cyfry, podkreślenie i myślnik.

Znak wykrzyknika znajdujący się za otwierającym nawiasem pozwala „zanegować" zbiór. Na przykład zapis `[!.;]` pasuje do każdego znaku oprócz kropki i średnika, a `[!a-zA-Z]` pasuje do wszystkich znaków, które nie są literami. Aby znaleźć dopasowanie do samego znaku !, należy umieścić go po pierwszym znaku zbioru albo poprzedzić lewym ukośnikiem, na przykład `[\!]`.

Bardzo użytecznym rozwiązaniem są zakresy, lecz nie należy czynić zbyt daleko idących założeń w odniesieniu do znaków wchodzących w skład zakresu. Bezpiecznie jest używać zakresu wielkich liter, małych liter, cyfr oraz ich dowolnych podzakresów (na przykład `[f-q]`, `[2-6]`). Nie należy natomiast używać zakresów dla znaków interpunkcyjnych albo liter o różnej wielkości. Na przykład nie można zakładać, że w zakresach `[a-Z]` i `[A-z]` znajdują się wyłącznie litery i nic więcej. Problem polega na tym, że tego rodzaju zakresy nie są przenośne między różnymi typami komputerów[9].

Proces dopasowywania wyrażeń zawierających symbole wieloznaczne to tak zwane **rozwijanie symbolami wieloznacznymi** albo **rozwijanie i uogólnianie**. Jest to tylko jeden z kilku kroków wykonywanych przez powłokę w trakcie odczytywania i przetwarzania wiersza poleceń. Kolejnym krokiem (wspomnianym wcześniej) jest rozwijanie znakiem tyldy, w którym tyldy są zastępowane nazwami katalogów domowych. Inne kroki poznamy w dalszych rozdziałach, a szczegółowy opis tego mechanizmu znalazł się w rozdziale 7.

Należy jednak pamiętać, że wykonywane polecenia wykorzystują jedynie wyniki rozwijania symbolami wieloznacznymi. Innymi słowy, widzą one jedynie listę argumentów, natomiast nie ma dla nich znaczenia, w jaki sposób te argumenty powstały. Na przykład, jeśli zostanie wpisane polecenie `ls fr*` i będą istnieć pliki wymienione na poprzedniej stronie, wówczas powłoka rozwinie wiersz poleceń do postaci `ls fred franek` i wywoła polecenie `ls` z argumentami `fred` i `franek`. Jeśli wpisane zostanie polecenie `ls g*`, wówczas polecenie `ls` zostanie wykonane (ponieważ nie zostaną znalezione pliki o pasującej nazwie) z niezmienionym ciągiem znaków `g*` i zwróci komunikat o błędzie `g*: Nie ma takiego pliku ani katalogu`[10].

[9] Mówiąc precyzyjnie, zakresy zależą od schematu kodowania znaków używanego przez komputer (zwykle jest to ASCII, lecz na przykład komputery mainframe firmy IBM używają schematu EBCDIC) oraz zestawu znaków wyznaczanego przez bieżące ustawienia regionalne (zakresy w językach innych niż angielski mogą powodować generowanie innych wyników).

[10] Różni się to od działania symboli wieloznacznych w powłoce C, która od razu wyświetli komunikat o błędzie i w ogóle nie wykona polecenia.

Rozważmy przykład, który powinien rozjaśnić omawiane zagadnienie. Załóżmy, że jesteśmy programistami C. Oznacza to, że pracujemy na plikach, których nazwy kończą się znakami *.c* (są to programy, zwane również plikami źródłowymi), *.h* (pliki nagłówków programów) i *.o* (pliki kodu obiektowego, nieczytelne dla człowieka), a także mamy do czynienia z innymi plikami. Powiedzmy, że chcemy uzyskać listę wszystkich plików źródłowych, kodu obiektowego i nagłówków znajdujących się w katalogu roboczym. Służy do tego polecenie `ls *.[cho]`. Powłoka rozwinie symbol `*.[cho]` do nazw wszystkich plików, których nazwy kończą się znakiem kropki i literą `c`, `h` lub `o`, po czym przekaże tak ustaloną listę jako argumenty polecenia `ls`. Inaczej mówiąc, polecenie `ls` zostanie wykonane z nazwami plików tak, jakby wszystkie te nazwy zostały wpisane ręcznie jedna po drugiej. Zwróćmy uwagę, że w ogóle nie musieliśmy znać rzeczywistych nazw tych plików! Całą pracę wykonał za nas symbol wieloznaczny.

Analizowany dotąd przykład z symbolami wieloznacznymi jest tak naprawdę jedynie częścią bardziej ogólnego zagadnienia pod tytułem **rozwijanie ścieżek dostępu**. Symbole wieloznaczne można stosować w katalogu roboczym, mogą one także stanowić część ścieżki dostępu. Na przykład, aby uzyskać listę wszystkich plików znajdujących się w katalogach */usr* i */usr2*, można wpisać polecenie `ls /usr*`. Jeśli będą nas interesować jedynie pliki z tych katalogów, których nazwa rozpoczyna się na literę b lub e, wówczas ich listę zwróci polecenie `ls /usr*/[be]*`.

Rozwijanie nawiasami klamrowymi

Zagadnieniem ściśle związanym z rozwijaniem ścieżek dostępu jest rozwijanie nawiasami klamrowymi. Symbole wieloznaczne w ścieżkach dostępu zostaną rozwinięte do nazw plików i katalogów, które istnieją, natomiast rozwijanie nawiasami klamrowymi prowadzi do utworzenia określonego ciągu znaków w konkretnej postaci, to znaczy zawierającego opcjonalny **wstęp**, następnie umieszczone w nawiasach klamrowych ciągi znaków oddzielone od siebie znakiem przecinka i opcjonalne **zakończenie**. Jeżeli na przykład zostanie wpisane polecenie `echo b{ra,ere,ru}k`, wówczas wyświetlone zostaną słowa `brak`, `berek` i `bruk`. Każdy ciąg znaków znajdujący się w nawiasach klamrowych jest łączony ze wstępem `b` oraz zakończeniem `k`. Co ważne, nie są to wcale nazwy plików — wygenerowane w ten sposób ciągi znaków nie zależą od nazw istniejących plików. Nawiasy klamrowe można dodatkowo zagnieżdżać, jak na przykład w wyrażeniu `b{ar{a,e},ru}k`. Zostanie ono rozwinięte do słów `barak`, `barek` i `bruk`.

Nieco inny rodzaj rozwijania nawiasami klamrowymi można zastosować w celu utworzenia sekwencji liter lub liczb. Jeżeli na przykład wpisane zostanie polecenie `echo {2..5}`, wyrażenie zostanie rozwinięte do ciągu `2 3 4 5`. Z kolei polecenie `echo {d..h}` wygeneruje ciąg `d e f g h`[11].

Rozwijanie nawiasami klamrowymi można stosować łącznie z rozwijaniem symbolami wieloznacznymi. W przykładzie z poprzedniego punktu, w którym była wyświetlana lista plików źródłowych, nagłówków i kodu obiektowego znajdujących się w katalogu roboczym, można było użyć polecenia `ls *.{c,h,o}`[12].

[11] Taka postać rozwijania nawiasami klamrowymi nie jest dostępna w wersjach powłoki *bash* niższych niż 3.0.

[12] Różni się to nieco od rozwijania nawiasami klamrowymi w powłoce C. Powłoka *bash* wymaga, aby w nawiasie obecny był co najmniej jeden znak przecinka, który nie będzie objęty cudzysłowami. W przeciwnym razie całe słowo pozostanie niezmienione, to znaczy wyrażenie `b{a}rk` po rozwinięciu nadal będzie miało postać `b{a}rk`.

Operacje wejścia-wyjścia

Informatyka — a tak naprawdę każda dziedzina nauki — przeżywa największy rozkwit wówczas, gdy ktoś (nie instytucja) wpadnie na bardzo prosty pomysł, który będzie miał jednak rozległe skutki. Na krótkiej liście takich pomysłów niewątpliwie znalazłby się schemat operacji wejścia-wyjścia systemu Unix, a także klasyczne już wynalazki takie jak język LISP, relacyjny model danych i programowanie zorientowane obiektowo.

Schemat wejścia-wyjścia systemu Unix opiera się na dwóch zaskakująco prostych pomysłach. Po pierwsze, plikowe wejście-wyjście w Uniksie ma postać sekwencji znaków o odpowiedniej długości (ilości bajtów). Schematy wejścia-wyjścia w starszych systemach są bardziej skomplikowane (mają na przykład postać „bloków", „rekordów", „kart" itp.). Po drugie, wszystkie elementy systemu, które generują lub przyjmują dane, są traktowane jak pliki. Dotyczy to również urządzeń sprzętowych, takich jak napędy dyskowe i terminale. W starszych systemach każde takie urządzenie jest traktowane odmiennie. Obydwa pomysły znacznie ułatwiły życie programistom systemowym.

Standardowe operacje wejścia-wyjścia

Konwencja stanowi, że każdy program Uniksa zawiera pojedynczy mechanizm przyjmowania danych wejściowych zwany **standardowym wejściem**, pojedynczy mechanizm generowania danych wyjściowych zwany **standardowym wyjściem** oraz pojedynczy mechanizm generowania komunikatów o błędzie zwany **standardowym wyjściem błędów**. Oczywiście każdy program może mieć także inne źródła danych wejściowych i wyjściowych, o czym przekonamy się w trakcie lektury rozdziału 7.

Standardowe operacje wejścia-wyjścia stanowiły pierwszy tego rodzaju schemat zaprojektowany z myślą o użytkownikach interaktywnych, korzystających z terminali, czyli rozwiązania odmiennego od stylu wsadowego polegającego zwykle na używaniu kart perforowanych. Powłoka Uniksa zazwyczaj udostępnia interfejs użytkownika, dlatego w naturalny sposób standardowe operacje wejścia-wyjścia skonstruowano tak, aby pasowały do sposobu działania powłoki.

Zasadniczo wszystkie powłoki wykonują operacje wejścia-wyjścia w taki sam sposób. Każdy wywoływany program ma trzy standardowe kanały wejścia-wyjścia skonfigurowane do pracy z terminalem lub stacją roboczą, a zatem standardowym wejściem jest klawiatura, a standardowym wyjściem oraz wyjściem błędów jest ekran lub okno. Na przykład narzędzie *mail* wyświetla wiadomości na standardowym wyjściu, a jeżeli zostanie użyte do wysyłania wiadomości do innych użytkowników, wówczas narzędzie pobierze dane wejściowe ze standardowego wejścia. Oznacza to, że otrzymane wiadomości przegląda się na ekranie, a nowe wiadomości wpisuje przy użyciu klawiatury.

W razie potrzeby można przekierować wejście i wyjście tak, aby pochodziły z pliku albo były do niego wysyłane. Jeżeli na przykład trzeba wysłać zawartość istniejącego pliku w postaci wiadomości pocztowej, wówczas należy przekierować standardowe wejście programu *mail* w taki sposób, aby odczytywał on dane z tego pliku, a nie z klawiatury.

Programy można również łączyć w tak zwane **potoki**, w których standardowe wyjście jednego programu stanowi jednocześnie bezpośrednie standardowe wejście dla innego programu. Na przykład wyjście programu *mail* można skierować jako wejście do programu *lp*, dzięki czemu wiadomości będą od razu drukowane, a nie wyświetlane na ekranie.

Dzięki temu rozwiązaniu narzędzi Uniksa można używać w roli elementów składowych większych programów. Wiele programów narzędziowych Uniksa tworzono właśnie z myślą o ich wykorzystaniu w taki sposób — każde z nich wykonuje określoną operację przetwarzania tekstu wejściowego. Wprawdzie niniejsza książka nie jest podręcznikiem z dziedziny programów narzędziowych Uniksa, lecz mają one zasadnicze znaczenie dla efektywnego wykorzystywania powłoki. Najbardziej znane narzędzia przetwarzania potokowego wymieniono w tabeli 1.5.

Tabela 1.5. Popularne narzędzia Uniksa przetwarzające dane

Narzędzie	Cel
cat	Kopiowanie wejścia do wyjścia.
grep	Wyszukiwanie ciągów znaków w danych wejściowych.
sort	Sortowanie wierszy danych wejściowych.
cut	Odczytywanie kolumn z danych wejściowych.
sed	Wykonywanie operacji edycyjnych na danych wejściowych.
tr	Zamienianie znaków znajdujących się w danych wejściowych na inne znaki.

Czytelnicy, którzy już używali niektórych z wymienionych narzędzi, być może zauważyli, że ich argumentami są nazwy plików wejściowych, a dane wyjściowe są przesyłane do standardowego wyjścia. Nie wszyscy jednak wiedzą, że w razie pominięcia argumentu wszystkie te narzędzia (tak samo zresztą jak większość innych programów narzędziowych Uniksa) przyjmą dane wejściowe pochodzące ze standardowego wejścia[13].

Na przykład podstawowym programem narzędziowym jest *cat*, który po prostu kopiuje dane wejściowe do wyjścia. Jeżeli zostanie wpisane polecenie `cat` z nazwą pliku jako argumentem, na ekranie zostanie wyświetlona zawartość tego pliku. Jeżeli natomiast polecenie zostanie wywołane bez argumentów, program wykorzysta standardowe wejście i skopiuje je do standardowego wyjścia. Zobaczmy, jak to działa: *cat* będzie czekać na wpisanie wiersza tekstu przez użytkownika; gdy naciśnięty zostanie klawisz *RETURN*, *cat* zwróci użytkownikowi wpisany tekst. Aby zatrzymać ten proces, na początku wiersza trzeba nacisnąć klawisze *CTRL+D*. Ich naciśnięcie spowoduje, że na ekranie pojawi się symbol `^D`. Oto efekt działania programu *cat*:

```
$ cat
Oto wiersz tekstu.
Oto wiersz tekstu.
Oto kolejny wiersz tekstu.
Oto kolejny wiersz tekstu.
^D
$
```

Przekierowywanie wejścia-wyjścia

Nazwa *cat* to skrót słowa „catenate", oznaczającego łączenie. Argumentami tego polecenia może być jedna lub więcej nazw plików, które mają zostać skopiowane do standardowego wyjścia. Na razie jednak załóżmy, że *cat* i inne programy narzędziowe nie akceptują argumentów w postaci nazw plików, a jedynie standardowe wejście. Jak już wcześniej wspomniano,

[13] Jeżeli w przypadku pominięcia nazwy pliku jako argumentu któreś z narzędzi Uniksa nie zaakceptuje standardowego wejścia, można spróbować zastosować jako argument znak myślnika (`-`). W niektórych systemach Unix standardowe wejście ma postać pliku, zatem można spróbować wskazać plik */dev/stdin* jako argument oznaczający plik wejściowy.

powłoka umożliwia przekierowywanie standardowego wejścia tak, aby pochodziło z pliku. Służy do tego zapis `polecenie < nazwa_pliku` — dzięki niemu polecenie pobierze standardowe wejście z pliku, a nie z terminala.

Jeżeli na przykład istnieje plik o nazwie *plik*, który zawiera jakiś tekst, wówczas polecenie `cat < plik` wyświetli zawartość tego pliku na terminalu. Z kolei polecenie `sort < plik` posortuje wiersze znajdujące się w tym pliku i wyświetli wyniki sortowania na terminalu (pamiętajmy jednak: cały czas zakładamy, że programy narzędziowe nie akceptują nazw plików jako argumentów).

Analogicznie `polecenie > nazwa_pliku` spowoduje, że standardowe wyjście polecenia zostanie przekierowane do pliku o wskazanej nazwie. Klasycznym przykładem jest polecenie `date > teraz`: polecenie `date` wyświetli na standardowym wyjściu bieżącą datę i godzinę. Wspomniane polecenie spowoduje zapisanie ich w pliku o nazwie *teraz*.

Przekierowywanie wejścia i wyjścia można łączyć. Na przykład polecenie `cp` standardowo służy do kopiowania plików, jeśli jednak z jakiegoś powodu polecenie to nie istnieje lub uległo uszkodzeniu, można w jego miejsce zastosować polecenie `cat`. Robi się to następująco:

```
$ cat < plik1 > plik2
```

Wynik tego polecenia będzie podobny do wyniku polecenia `cp plik1 plik2`.

Potoki

Istnieje również możliwość przekierowywania wyjścia polecenia do standardowego wejścia innego polecenia zamiast do pliku. Służy do tego konstrukcja zwana potok (*pipe*), oznaczana znakiem `|`. Wiersz polecenia zawierający dwa lub więcej poleceń połączonych znakiem `|` jest nazywany przetwarzaniem potokowym.

Potoku często używa się z poleceniem `more`, które działa podobnie jak polecenie `cat`, z tą różnicą, że dane wynikowe są wyświetlane ekran po ekranie, a wyświetlanie kolejnych ekranów jest wstrzymywane do czasu naciśnięcia przez użytkownika klawisza spacji (następny ekran), klawisza *RETURN* (następny wiersz) lub wpisania innych poleceń. Jeżeli bieżący katalog zawiera znaczną liczbę plików i trzeba uzyskać szczegółowe informacje na ich temat, polecenie `ls -l | more` spowoduje wyświetlenie pożądanych informacji w podziale na ekrany.

Potoki mogą przybierać bardzo złożoną postać, mogą być również łączone z przekierowywaniem wejścia-wyjścia. Aby wyświetlić posortowaną zawartość pliku *plik* w podziale na kolejne ekrany, należy wpisać polecenie `sort < plik | more`. Natomiast aby wydrukować tę zawartość zamiast prezentować ją na ekranie, należy wpisać `sort < plik | lp`.

Spójrzmy na bardziej skomplikowany przykład. W pliku */etc/passwd* przechowywane są informacje na temat kont użytkowników systemu Unix. Każdy wiersz tego pliku zawiera nazwę użytkownika, jego numer identyfikacyjny, zaszyfrowane hasło, katalog domowy, powłokę logowania i inne dane. Pierwsze pole każdego wiersza to nazwa użytkownika, a poszczególne pola są oddzielane znakiem dwukropka (`:`). Przykładowy wiersz może mieć następującą postać:

```
cam:LM1c7GhNesD4GhF3iEHrH4FFeCKB/:501:100:Cameron Newham:/home/cam:/bin/bash
```

Aby uzyskać posortowaną listę wszystkich użytkowników systemu, należy wpisać polecenie:

```
$ cut -d: -f1 < /etc/passwd | sort
```

(Tak naprawdę znak < można pominąć, ponieważ `cut` akceptuje argumenty w postaci nazw plików wejściowych). Polecenie `cut` odczyta z wejścia pierwsze pole (`-f1`), przy czym pola te są oddzielane dwukropkiem (`-d:`). Przedstawiony powyżej potok wyświetli listę podobną do poniższej:

```
adm
bin
cam
daemon
davidqc
ftp
games
gonzo
...
```

Jeżeli lista ma zostać wysłana bezpośrednio na drukarkę zamiast na ekran, można rozszerzyć potok w następujący sposób:

```
$ cut -d: -f1 < /etc/passwd | sort | lp
```

Powyższy przykład powinien wyjaśnić, w jaki sposób przekierowania wejścia-wyjścia i potoki wpisują się w filozofię elementów składowych w systemach Unix. Stosowany zapis jest niezwykle zwięzły i daje ogromne możliwości. Równie ważne jest to, że mechanizm potoków eliminuje konieczność posługiwania się plikami tymczasowymi, w których zapisywano by dane wynikowe poleceń, zanim zostaną przekazane do innych poleceń.

Na przykład, aby uzyskać taki sam efekt jak w wyniku wywołania powyższego przetwarzania potokowego, w innych systemach operacyjnych trzeba wykonać trzy polecenia (zakładając przy tym, że w systemach tych dostępne są podobne programy narzędziowe). W systemie VAX/VMS firmy DEC mogą one mieć postać:

```
$ cut [etc]passwd /d=":" /f=1 /out=temp1
$ sort temp1 /out=temp2
$ print temp2
$ delete temp1 temp2
```

Po jakimś czasie użytkownicy są w stanie bez problemu pisać w jednym wierszu rozbudowane potoki wykonujące operacje, których wykonanie w innych systemach operacyjnych byłoby możliwe dopiero po użyciu kilku poleceń (i plików tymczasowych).

Zadania drugoplanowe

Potoki są tak naprawdę specjalnym przypadkiem bardziej ogólnego mechanizmu, który polega na wykonywaniu więcej niż jednej czynności naraz. Jest to rozwiązanie, które nie jest dostępne w wielu innych komercyjnych systemach operacyjnych ze względu na ograniczenia nakładane na użytkowników[14]. Unix natomiast został opracowany w laboratorium naukowym i początkowo miał pracować jedynie na jego potrzeby, dlatego nie ogranicza się w nim zbytnio dostępu użytkowników do zasobów, dążąc do utrzymania prostoty zamiast zmuszać do tworzenia skomplikowanych rozwiązań.

[14] Obecnie wszystkie popularne systemy operacyjne posiadają taką możliwość — *przyp. tłum.*

„Wykonywanie więcej niż jednej czynności naraz" oznacza uruchamianie więcej niż jednego programu w tym samym momencie. Tak właśnie dzieje się, gdy używany jest potok. Z tym samym efektem mamy do czynienia, jednocześnie logując się do systemu Unix kilka razy pod rząd bez wcześniejszego wylogowania. (Gdyby taką operację wykonać w systemie VM/CMS firmy IBM, zostałby wyświetlony komunikat „already logged in").

Powłoka pozwala również wykonywać więcej niż jedno polecenie naraz w trakcie pojedynczej sesji logowania. Gdy wpisywane jest polecenie i naciskany klawisz *RETURN*, zwykle powłoka przekazuje sterowanie terminalem temu poleceniu do momentu zakończenia jego wykonywania; wówczas nie można wpisywać kolejnych poleceń, póki nie zakończy się wykonywanie wcześniejszego. Jeżeli natomiast trzeba uruchomić polecenie, które nie wymaga danych wejściowych od użytkownika, i chcemy, aby w trakcie działania tego polecenia wykonywane były również inne czynności, należy zakończyć polecenie znakiem ampersanda (`&`).

Jest to tak zwane uruchamianie polecenia w tle. Polecenie uruchomione w taki sposób określane jest jako zadanie drugoplanowe, natomiast zadanie uruchomione w sposób standardowy nazywane jest zadaniem pierwszoplanowym. Gdy uruchomione zostanie zadanie drugoplanowe, symbol zachęty systemu operacyjnego pojawi się od razu na ekranie i możliwe będzie wpisywanie kolejnych poleceń.

Najbardziej oczywiste jest uruchamianie jako zadań drugoplanowych programów działających przez dłuższy czas, na przykład wykonywanie operacji `sort` czy `uncompress` na dużych plikach. Załóżmy na przykład, że właśnie otrzymaliśmy bardzo duży plik skompresowany, który został załadowany do katalogu z taśmy magnetycznej[15]. Przyjmijmy, że plikiem tym jest *gcc.tar.Z* — skompresowany plik archiwum zawierający ponad 10 MB plików z kodem źródłowym.

Należy wpisać polecenie `uncompress gcc.tar &` (ostatni fragment, `.Z`, można pominąć). System rozpocznie wówczas realizację zadania drugoplanowego, które „na miejscu" rozpakuje dane i utworzy plik *gcc.tar*. Zaraz po wpisaniu wspomnianego polecenia na ekranie pojawi się wiersz podobny do przedstawionego poniżej:

```
[1] 175
```

Po nim nastąpi znak zachęty powłoki, oznaczający możliwość wpisywania następnych poleceń. Na podstawie wyświetlonych numerów można odwoływać się do zadania drugoplanowego (szczegółowe informacje na ten temat znajdują się w rozdziale 8.).

Zadania drugoplanowe można sprawdzać poleceniem `jobs`. Dla każdego zadania drugoplanowego `jobs` wyświetla wiersz podobny do powyższego, wskazując dodatkowo stan zadania:

```
[1]+ Running uncompress gcc.tar &
```

Gdy wykonywanie zadania zakończy się, zaraz przed znakiem zachęty pojawi się komunikat podobny do poniższego:

```
[1]+ Done uncompress gcc.tar
```

Treść komunikatu będzie inna, jeśli zadanie drugoplanowe zakończy się błędem. Więcej na ten temat również w rozdziale 8.

[15] Pliki skompresowane są tworzone przez narzędzie *compress* (narzędzie to kompresuje pliki, aby zajmowały mniej miejsca na dysku). Pliki takie noszą nazwy w postaci *nazwa.Z*, gdzie *nazwa* oznacza nazwę oryginalnego, nieskompresowanego pliku.

Drugoplanowe operacje wejścia-wyjścia

Zadania uruchamiane w tle nie powinny wykonywać operacji wejścia-wyjścia na terminalu. Zastanówmy się nad tym przez chwilę, a od razu będzie wiadomo, dlaczego tak jest.

Zadanie drugoplanowe z definicji nie ma kontroli nad terminalem. Oznacza to między innymi, że na dane wejściowe z klawiatury oczekuje jedynie zadanie pierwszoplanowe (albo sama powłoka, jeżeli takiego zadania nie ma). Jeżeli zadanie drugoplanowe wymaga wpisania danych wejściowych z klawiatury, często jego wykonywanie po prostu zostaje zawieszone aż do momentu podjęcia jakichś kroków (co zostało opisane w rozdziale 8.).

Jeżeli zadanie drugoplanowe wysyła dane wyjściowe na ekran, dane te po prostu pojawią się na ekranie. Jeżeli jednocześnie uruchomione jest zadanie pierwszoplanowe, które także generuje dane wyjściowe, wówczas wyniki obydwu zadań będą się przeplatać w sposób losowy i często irytujący dla użytkownika.

Jeżeli w tle ma zostać uruchomione zadanie oczekujące standardowego wejścia lub generujące standardowe wyjście, zwykle wykonuje się przekierowanie wejścia-wyjścia w taki sposób, aby dane pochodziły z pliku lub były do niego wysyłane. Programy, które tworzą krótkie, jednowierszowe komunikaty (ostrzeżenia, komunikaty o zakończeniu pracy itd.), są wyjątkiem od tej reguły, ponieważ zwykle użytkownik nie ma nic przeciwko temu, aby taki komunikat znalazł się między pozostałymi danymi wyjściowymi.

Na przykład narzędzie *diff* analizuje dwa pliki, których nazwy stanowią argumenty polecenia, a następnie w standardowym wyjściu prezentuje podsumowanie różnic między nimi. Jeżeli pliki okażą się identyczne, *diff* nie zwróci żadnych wyników. Zwykle jednak polecenie `diff` uruchamia się wówczas, gdy istnieje konieczność odnalezienia wierszy różniących się od siebie.

Polecenie `diff`, podobnie jak `sort` i `compress`, może potrzebować wiele czasu na wykonanie zadania, zwłaszcza w przypadku dużych plików. Załóżmy, że mamy dwa duże pliki o nazwach *wojnaipokój.txt* i *wojnaipokój.txt.old*. Polecenie `diff wojnaipokój.txt wojnaipokój.txt.old`[16] może na przykład ujawnić, że autor zdecydował się zmienić w całym pliku imię „Iwan" na „Aleksander", wykonał więc setki zmian, a przez to również zbiór wygenerowanych danych wynikowych będzie obszerny.

Jeżeli zostanie wpisane polecenie `diff wojnaipokój.txt wojnaipokój.txt.old &`, system będzie co chwila zwracał kolejne dane wynikowe, co będzie trudne do zatrzymania nawet przy wykorzystaniu technik przedstawionych w rozdziale 7. Natomiast polecenie:

```
$ diff wojnaipokój.txt wojnaipokój.txt.old > txtdiff &
```

spowoduje, że zidentyfikowane różnice zostaną zapisane w pliku *txtdiff* i będzie można je przeanalizować później.

[16] Aby oszczędzić sobie pisania, można by wpisać skrótowe polecenie `diff wojnaipokój*`, o ile tylko nie istnieją inne pliki o nazwie rozpoczynającej się w ten sposób. Należy pamiętać, że `diff` rozpoznaje argumenty dopiero po rozwinięciu przez powłokę symboli wieloznacznych. Wielu użytkowników o tym zapomina.

Zadania drugoplanowe i ich priorytety

Dzięki zadaniom drugoplanowym można zaoszczędzić czas potrzebny na wpisywanie poleceń. Trzeba jednak pamiętać, że takie zadania zużywają znaczne ilości zasobów takich jak pamięć i czas procesora (CPU). To, że kilka zadań jest wykonywanych jednocześnie, wcale nie oznacza, że będą one trwać krócej, niż gdyby uruchamiano je jedno po drugim. W rzeczywistości wydajność zwykle jest nieco niższa.

Każdemu zadaniu w systemie przypisywany jest **priorytet**, czyli numer wskazujący systemowi operacyjnemu, jaki stopień ważności należy nadać zadaniu w sytuacji wyczerpywania się zasobów (im wyższy numer, tym niższy priorytet). Polecenia wpisywane w powłoce — bez względu na to, czy będą to zadania pierwszoplanowe, czy drugoplanowe — zwykle mają taki sam priorytet. Administrator systemu ma możliwość uruchamiania poleceń z wyższym priorytetem niż zwykli użytkownicy.

Zwróćmy uwagę, że uruchamianie w systemie znacznej liczby zadań drugoplanowych dla wielu użytkowników może spowodować zużycie większej ilości zasobów, niż przysługują danemu użytkownikowi. Należy w takiej sytuacji zastanowić się, czy krótszy czas wykonania zadania jest naprawdę ważniejszy, niż przestrzeganie zasad współużytkowania systemu.

A jeśli mówimy już o zasadach współużytkowania systemu, Unix udostępnia polecenie, które pozwala na obniżenie priorytetu dowolnego zadania. Jest to polecenie `nice`. Jeśli zostanie wpisane `nice` *`polecenie`* — gdzie *`polecenie`* może być nawet skomplikowanym wierszem poleceń powłoki z potokami, przekierowaniami i tak dalej — wówczas zostanie ono uruchomione z niższym priorytetem[17]. Aby odpowiednio obniżyć priorytet, można przekazać poleceniu `nice` argument w postaci liczby. Więcej informacji można znaleźć w podręczniku na stronach poświęconych `nice`[18].

Znaki specjalne i używanie cudzysłowów

Znaki `<`, `>`, `|` i `&` to cztery przykłady **znaków specjalnych**, które dla powłoki mają określone znaczenie. Symbole wieloznaczne przedstawione we wcześniejszej części rozdziału (`*`, `?` i `[...]`) to również znaki specjalne.

W tabeli 1.6 opisano znaczenie wszystkich znaków specjalnych, obowiązujące jedynie w wierszach poleceń powłoki. Inne znaki mają specjalne znaczenie w określonych sytuacjach (o których więcej powiemy w rozdziałach 3. i 4.), takich jak stosowanie wyrażeń regularnych i operatorów przetwarzania ciągów znaków.

Stosowanie cudzysłowów

Czasami trzeba zastosować znaki specjalne w sposób dosłowny, to znaczy bez ich specjalnego znaczenia. W tym celu stosuje się **cudzysłowy**. Jeżeli ciąg znaków zostanie objęty pojedynczymi **apostrofami**, wówczas wszystkie znaki objęte apostrofami stracą swe specjalne znaczenie.

[17] Dłuższe polecenia występujące za poleceniem `nice` powinno się wyróżniać cudzysłowem lub apostrofami.

[18] Administratorzy systemu zalogowani jako użytkownicy `root` mogą też poleceniem `nice` podwyższać priorytety zadań.

Tabela 1.6. Znaki specjalne

Znak	Znaczenie	Patrz rozdział
~	Katalog domowy.	1
`	Zastąpienie poleceniem (przestarzałe).	4
#	Komentarz.	4
$	Wyrażenia i zmienne.	3
&	Zadanie drugoplanowe.	1
*	Symbol wieloznaczny dla ciągu znaków.	1
(	Uruchomienie podpowłoki.	8
)	Zatrzymanie podpowłoki.	8
\	Umieszczenie następnego znaku w cudzysłowie.	1
|	Potok.	1
[	Początek symbolu wieloznacznego w postaci zbioru znaków.	1
]	Koniec symbolu wieloznacznego w postaci zbioru znaków.	1
{	Początek bloku poleceń.	7
}	Koniec bloku poleceń.	7
;	Separator poleceń powłoki.	3
'	Silny cudzysłów.	1
"	Słaby cudzysłów.	1
<	Przekierowanie wejścia.	1
>	Przekierowanie wyjścia.	1
/	Separator katalogów ścieżki dostępu.	1
?	Symbol wieloznaczny dla pojedynczego znaku.	1
!	Logiczne zaprzeczenie.	5

Z najbardziej oczywistą sytuacją, gdy ciąg znaków trzeba objąć cudzysłowami, mamy do czynienia w przypadku polecenia `echo`, które pobiera argumenty i przekazuje je do standardowego wyjścia. W jakim celu? Jak przekonamy się w trakcie lektury kolejnych rozdziałów, w trakcie przetwarzania wiersza poleceń powłoka wykonuje stosunkowo dużo czynności, z których większość dotyczy znaków specjalnych wymienionych w tabeli 1.6. Polecenie `echo` to jedno z narzędzi, dzięki którym wyniki takiego przetwarzania można udostępnić na standardowym wyjściu.

A gdyby chcieć wyświetlić ciąg znaków `2 * 3 > 5 to nierówność fałszywa`? Załóżmy, że wpisywane jest następujące polecenie:

```
$ echo 2 * 3 > 5 to nierówność fałszywa.
```

W odpowiedzi pojawiłby się znak zachęty, tak jakby nic się nie stało! Ale tak naprawdę powstałby nowy plik o nazwie *5*, zawierający cyfrę `2`, nazwy wszystkich plików w bieżącym katalogu, a także ciąg znaków `3 to nierówność fałszywa`. Warto się upewnić, że rozumiemy przyczynę takiego stanu rzeczy[19].

[19] Dowodzi to również elastyczności przekierowań wejścia-wyjścia, które mogą znajdować się w dowolnym miejscu wiersza poleceń — nawet takim, w którym na pierwszy rzut oka nie mają żadnego sensu.

Jeżeli natomiast wpisane zostanie polecenie:

```
$ echo '2 * 3 > 5 to nierówność fałszywa.'
```

jego wynikiem będzie ciąg znaków w niezmienionej wersji. Nie trzeba jednak obejmować apostrofami całego wiersza, a jedynie tę jego część, która zawiera znaki specjalne (albo dla pewności znaki, które mogą sprawiać wrażenie specjalnych). Polecenie:

```
$ echo '2 * 3 > 5' to nierówność fałszywa.
```

zwróci identyczny wynik.

Zwróćmy uwagę, że w tabeli cudzysłów (`"`) określono mianem słabego cudzysłowu. Ciąg znaków ujęty w cudzysłów podlega **niektórym** czynnościom wykonywanym przez powłokę w trakcie przetwarzania wiersza poleceń, ale nie wszystkim. (Inaczej mówiąc, tylko niektóre znaki są traktowane jako znaki specjalne). W dalszych rozdziałach pokażemy, dlaczego stosowanie cudzysłowów jest czasami lepszym rozwiązaniem. W rozdziale 7. znajdują się szczegółowe informacje na temat obowiązujących w powłoce zasad traktowania cudzysłowów oraz innych aspektów przetwarzania wiersza poleceń. Na razie jednak pozostańmy przy samych apostrofach.

Unieważnianie lewym ukośnikiem

Kolejnym sposobem na zmianę znaczenia znaku jest poprzedzenie go znakiem lewego ukośnika (`\`). Jest to tak zwane **unieważnianie znaku lewym ukośnikiem**. W większości przypadków poprzedzenie znaku lewym ukośnikiem jest równoznaczne z ujęciem go w cudzysłów. Na przykład polecenie:

```
$ echo 2 \* 3 \> 5 to nierówność fałszywa.
```

zwróci taki sam wynik, jak gdyby ciąg znaków objęto apostrofami. Aby wykorzystać w ciągu znaków lewy ukośnik, wystarczy objąć go apostrofami (`'\'`) albo — co jest jeszcze łatwiejsze — poprzedzić lewym ukośnikiem (`\\`).

Zajmijmy się jakimś bardziej praktycznym przykładem obejmowania znaków specjalnych cudzysłowami. Niektóre polecenia Uniksa wymagają podania argumentów, które często zawierają znaki specjalne. Trzeba je zatem unieważnić, aby powłoka nie przetworzyła ich w pierwszej kolejności. Najczęściej używanym tego rodzaju poleceniem jest `find`, które wyszukuje pliki w całych drzewach katalogów.

Aby móc użyć polecenia `find`, trzeba wskazać katalog główny drzewa, które ma być przeszukiwane, oraz podać argumenty opisujące charakterystykę poszukiwanego pliku (lub plików). Na przykład polecenie `find . -name` *`ciąg`* spowoduje wyszukanie w drzewie katalogów, którego katalogiem głównym jest katalog bieżący, plików o nazwach pasujących do ciągu znaków. (Inne argumenty pozwalają na przeszukiwanie pod względem rozmiaru pliku, jego właściciela, uprawnień, daty ostatniego dostępu i tak dalej).

W ciągu znaków można stosować symbole wieloznaczne, trzeba jednak umieścić je w cudzysłowach, aby polecenie `find` mogło dopasować je do nazw plików występujących w poszczególnych przeszukiwanych katalogach. Polecenie `find -name '*.c'` wyszuka w katalogu bieżącym i jego podkatalogach wszystkie pliki, których nazwy kończą się znakami *.c*.

Unieważnianie cudzysłowów

Wykorzystując lewy ukośnik, można również umieszczać znaki cudzysłowu w ciągu znaków objętym cudzysłowami. Na przykład polecenie:

```
$ echo \"2 \* 3 \> 5\" to nierówność fałszywa.
```

zwróci następujący wynik:

```
"2 * 3 > 5" to nierówność fałszywa.
```

Mechanizm ten nie zadziała jednak w przypadku apostrofów znajdujących się wewnątrz wyrażeń objętych apostrofami. Na przykład polecenie `echo 'Herbatka u Harry'ego'` wcale nie zwróci ciągu znaków `Herbatka u Harry'ego`. Ograniczenie to można obejść na kilka sposobów. Najpierw spróbujmy wyeliminować apostrofy:

```
$ echo Herbatka u Harry\'ego
```

Jeżeli żadne pozostałe znaki nie są znakami specjalnymi (a tak jest w tym przypadku), rozwiązanie to zadziała. W przeciwnym razie można wykonać następujące polecenie:

```
$ echo 'Herbatka u Harry'\''ego'
```

W tym przypadku zapis `'\''` (czyli apostrof, lewy ukośnik, apostrof, apostrof) zadziała tak, jakby apostrof znajdował się w ciągu znaków objętym apostrofami. Dlaczego? Otóż pierwszy znak `'` w ciągu `'\''` kończy pierwszy ciąg objęty apostrofami (`'Herbatka u Harry`), `\'` dodaje znak apostrofu, a następny apostrof rozpoczyna kolejny ciąg znaków objęty apostrofami, zawierający litery `ego`. Zrozumienie tej zasady pozwoli uniknąć kłopotów związanych ze zrozumieniem innych skomplikowanych zagadnień dotyczących składni powłoki.

Kontynuacja wierszy

Kolejnym zagadnieniem jest sposób, w jaki można kontynuować tekst polecenia, które na terminalu lub stacji wykracza poza jeden wiersz. Odpowiedź jest w zasadzie prosta: wystarczy ująć w cudzysłów klawisz *RETURN*. W końcu *RETURN* to jeszcze jeden zwykły znak.

Taki cel można osiągnąć na dwa sposoby: kończąc wiersz lewym ukośnikiem lub nie wpisując kończącego znaku apostrofu (czyli ująć klawisz *RETURN* w cudzysłów). Jeżeli zastosowany zostanie lewy ukośnik, wówczas między nim a końcem wiersza nie może się już nic znajdować — nawet spacja czy znak tabulacji.

Gdy zastosowany zostanie lewy ukośnik lub znak apostrofu, będzie to informacja dla powłoki, że należy zignorować specjalne znaczenie klawisza *RETURN*. Po jego naciśnięciu powłoka rozpozna, że tekst polecenia nie został jeszcze zakończony (bo przecież „prawdziwy" klawisz *RETURN* nie został jeszcze naciśnięty). Wyświetlony zostanie zatem drugi znak zachęty, domyślnie mający postać >, i powłoka dalej będzie oczekiwać na zakończenie wiersza. Wiersz można kontynuować nieograniczoną ilość razy.

Gdybyśmy na przykład chcieli wyświetlić w powłoce pierwsze zdanie z *Ogniem i mieczem* Henryka Sienkiewicza, można by wpisać następujące polecenie:

```
$ echo Rok 1647 był to dziwny rok, w którym \
> rozmaite znaki na niebie i ziemi zwiastowały \
> jakoweś klęski i nadzwyczajne zdarzenia.
```

Taki sam efekt można osiągnąć tak:

```
$ echo 'Rok 1647 był to dziwny rok, w którym
> rozmaite znaki na niebie i ziemi zwiastowały
> jakoweś klęski i nadzwyczajne zdarzenia.'
```

Klawisze kontrolne

Klawisze kontrolne, czyli kombinacje klawisza *CONTROL* (albo *CTRL*) oraz drugiego klawisza, to kolejny rodzaj znaków specjalnych. Zwykle po ich naciśnięciu na ekranie nie pojawiają się żadne znaki, natomiast system operacyjny interpretuje niektóre z nich jako polecenia specjalne. Jeden z takich klawiszy już znamy: *RETURN* jest tożsamy z klawiszem *CTRL+M* (można samemu się o tym przekonać). Często używa się również klawiszy *BACKSPACE* lub *DEL* w celu usunięcia błędów literowych popełnionych w wierszu poleceń.

Tak naprawdę działanie wielu klawiszy kontrolnych nie dotyczy w żaden sposób użytkownika. Warto jednak je poznać na zapas, także na wypadek niezamierzonego naciśnięcia któregoś z nich.

Chyba najbardziej kłopotliwym aspektem korzystania z klawiszy kontrolnych jest to, że ich działanie może być odmienne w różnych systemach. Standardowe działanie takich klawiszy przedstawiono w tabeli 1.7, zawierającej listę klawiszy kontrolnych obsługiwanych przez wszystkie najważniejsze współczesne wersje Uniksa. Zwróćmy uwagę, że klawisze *DEL* i *CTRL+?* mają takie samo działanie.

Dzięki poleceniu `stty` można sprawdzić aktualne ustawienia oraz zmienić je w razie potrzeby (więcej informacji na ten temat znajduje się w rozdziale 8.). Jeżeli używana wersja Uniksa wywodzi się z systemu BSD (na przykład jest to Unos albo OS X), można wpisać polecenie `stty all` i wyświetlić w ten sposób ustawienia klawiszy kontrolnych. Na ekranie pojawią się informacje podobne do poniższych:

```
erase  kill   werase rprnt  flush  lnext  susp   intr   quit   stop   eof
^?     ^U     ^W     ^R     ^O     ^V     ^Z/^Y  ^C     ^\     ^S/^Q  ^D
```

Tabela 1.7. Klawisze kontrolne

Klawisz kontrolny	Nazwa stty	Opis funkcji
CTRL+C	`intr`	Zatrzymanie bieżącego polecenia.
CTRL+D	`eof`	Koniec danych wejściowych.
CTRL+	`quit`	Zatrzymanie bieżącego polecenia, jeżeli *CTRL+C* nie działa.
CTRL+S	`stop`	Przerwanie wyświetlania danych wynikowych na ekranie.
CTRL+Q		Ponowne uruchomienie wyświetlania danych wynikowych na ekranie.
DEL albo *CTRL+?*	`erase`	Usunięcie ostatniego znaku.
CTRL+U	`kill`	Usuniecie całego wiersza poleceń.
CTRL+Z	`susp`	Zawieszenie bieżącego polecenia (więcej o tym w rozdziale 8.).

Zapis `^X` oznacza klawisz *CTRL+X*. Jeżeli używany system wywodzi się z System III albo System V (należą do nich AIX, HP/UX, SCO, Linux i Xenix), należy wpisać polecenie `stty -a`.

Komunikat wynikowy będzie zawierał następujące informacje:

```
intr = ^C; quit = ^|; erase = DEL; kill = ^u; eof = ^d; eol = ^';
swtch = ^`; susp = ^z; dsusp <undef>;
```

Najczęściej używanym klawiszem kontrolnym jest *CTRL+C*, czasami nazywany **klawiszem przerwania**. Zatrzymuje on (a przynajmniej próbuje zatrzymać) aktualnie wykonywane polecenie. Klawisza tego używa się na przykład w sytuacji, gdy wpisane polecenie okazuje się działać zbyt długo, w poleceniu wpisano złe argumenty, użytkownik zrezygnował z dalszego wykonywania polecenia itp.

Czasami klawisz *CTRL+C* nie działa. W takim przypadku, jeżeli potrzeba zatrzymania polecenia jest naprawdę paląca, należy nacisnąć *CTRL+*. Nigdy jednak nie powinno się od razu beztrosko stosować *CTRL+* — zawsze trzeba najpierw spróbować użyć *CTRL+C*! Szczegółowe informacje, dlaczego należy tak postępować, znajdują się w rozdziale 8. Na razie wystarczy wspomnieć, że po naciśnięciu *CTRL+C* działające polecenie ma większe szanse po sobie „posprzątać", tak aby pliki i inne używane przez polecenie zasoby z powrotem przybrały odpowiednie stany.

Przykład działania *CTRL+D* już przedstawiono. Gdy zostanie uruchomione polecenie przyjmujące dane wejściowe pochodzące z klawiatury, naciśnięcie *CTRL+D* będzie stanowiło dla procesu informację, że nastąpił koniec danych wynikowych — tak samo, jakby proces odczytywał dane z pliku i dotarł do jego końca. Klawisz ten często stosuje się na przykład w trakcie pracy z programem narzędziowym *mail*: gdy wpisywana jest wiadomość, w celu jej zakończenia naciska się *CTRL+D*. W ten sposób *mail* otrzymuje informację, że wiadomość jest już skończona i gotowa do wysłania. Większość programów narzędziowych, które pobierają dane ze standardowego wejścia, interpretuje klawisz *CTRL+D* jako znak końca danych wejściowych, choć wiele z nich rozpoznaje również polecenia takie jak `q`, `quit`, `exit` i im podobne.

Klawisze *CTRL+S* i *CTRL+Q* to tak zwane znaki sterowania przepływem. Stanowią one przestarzały sposób zatrzymywania i restartowania przepływu danych wyjściowych, generowanych przez jedno urządzenie, do innego (na przykład z komputera do terminala), przydatny w czasach, gdy takie operacje trwały dosyć długo. Obecnie, w czasach szybkich połączeń sieciowych, właściwie są bezużyteczne. Tak naprawdę *CTRL+S* i *CTRL+Q* sprawiają tylko kłopot. Warto o nich wiedzieć jedynie to, że jeżeli generowanie danych wynikowych na ekranie nagle zatrzymało się, prawdopodobnie użytkownik przez przypadek nacisnął klawisz *CTRL+S*. Należy wówczas nacisnąć *CTRL+Q*, aby ponownie uruchomić tę czynność. Trzeba jednak pamiętać, że wówczas zostaną uwzględnione wszystkie klawisze naciśnięte w międzyczasie.

Ostatnia grupa znaków kontrolnych to podstawowe narzędzia edycji w wierszu poleceń. Klawisz *DEL* działa jak klawisz *BACKSPACE* (tak naprawdę w niektórych systemach usuwanie odbywa się za pomocą klawisza *BACKSPACE* albo *CTRL+H*, a nie *DEL*), a *CTRL+U* usuwa cały wiersz poleceń i pozwala użytkownikowi na wpisanie go na nowo. Również te klawisze nie są już powszechnie używane[20]. Następny rozdział dotyczy trybów edycji powłoki *bash*, które stanowią jeden z najbardziej przydatnych mechanizmów i mają znacznie bogatsze możliwości niż ograniczone w swych funkcjach klawisze kontrolne, opisywane w tym punkcie.

[20] Dlaczego nadal funkcjonują takie przestarzałe klawisze kontrolne? Nie mają one nic wspólnego z samą powłoką, są natomiast rozpoznawane przez **sterownik tty** — starą wewnętrzną część systemu operacyjnego, kontrolującą dane wejściowe i wyjściowe przesyłane do (z) terminala.

Pomoc

Rozwiązaniem charakterystycznym jedynie dla powłoki *bash* jest system pomocy online. Polecenie `help` wyświetla informacje na temat poleceń dostępnych w powłoce *bash*. Jeżeli zostanie wpisane samo polecenie `help`, zwrócona zostanie lista wbudowanych poleceń powłoki wraz z dostępnymi dla nich opcjami.

Jeżeli polecenie `help` zostanie wywołane wraz z nazwą polecenia powłoki, wyświetlony zostanie szczegółowy opis podanego polecenia:

```
$ help cd
cd: cd [-L | -P] [dir]
 Change the current directory to DIR. The variable $HOME is the
 default DIR. The variable $CDPATH defines the search path for
 the directory contatining DIR. Alternative directory names in
 CDPATH are separated by a colon (:). A null directory name is
 the same as the current directory, i.e. `.'. If DIR begins with
 a slash (/), then $CDPATH is not used. If the directory is not
 found, and the shell option `cdable_vars' is set, then try the
 word as a variable name. If that variable has a value, then cd
 to the value of that variable. The -P option says to use the
 physical directory structure instead of following symbolic links;
 the -L option forces symbolic links to be followed.
```

Polecenie `help` można wywoływać jedynie z częścią nazwy polecenia — wówczas zostaną wyświetlone szczegółowe informacje na temat wszystkich poleceń pasujących do podanej części nazwy. Na przykład wpisanie `help re` spowoduje wyświetlenie opisu poleceń `read`, `readonly` i `return`. Fragment nazwy może zawierać symbole wieloznaczne. W takiej sytuacji trzeba będzie objąć część nazwy apostrofami, aby nie została rozwinięta do nazwy pliku. Efekt identyczny jak w poprzednim przykładzie da polecenie `help 're*'`, natomiast `help 're??'` spowoduje wyświetlenie opisu jedynie polecenia `read`.

Czasami polecenie `help` zwróci tyle informacji, że nie będą się one mieścić na jednym ekranie i ekran zostanie przewinięty. Można wówczas użyć polecenia `more`, aby wymusić wyświetlanie tylko jednego ekranu naraz — należy zatem posłużyć się poleceniem `help` *`polecenie`* `| more`.