Sławomir Wernikowski

Podstawy pracy terminalowej w systemie Linux

Zachodniopomorski
Uniwersytet Technologiczny
w Szczecinie

Sławomir Wernikowski

Podstawy pracy terminalowej w systemie Linux

Szczecin 2025

Fundusze Europejskie Wiedza Edukacja Rozwój
Rzeczpospolita Polska
Unia Europejska Europejski Fundusz Społeczny

Recenzent:
Ireneusz Mrozek

Korekta:
Monika Jagielska

Skład komputerowy:
Sławomir Wernikowski

Wydano za zgodą Rektora Zachodniopomorskiego Uniwersytetu Technologicznego w Szczecinie

Skrypt został przygotowany w ramach Projektu:
„ZUT 2.0 – Nowoczesny Zintegrowany Uniwersytet" współfinansowanego przez Unię Europejską w ramach Europejskiego Funduszu Społecznego.

ISBN 978-83-7663-351-0

Wydawnictwo Uczelniane Zachodniopomorskiego Uniwersytetu Technologicznego w Szczecinie
70-311 Szczecin, al. Piastów 48
tel. 91 449 47 60
e-mail: wydawnictwo@zut.edu.pl

Spis treści

0. Wstęp

0.1. Przeznaczenie i cele skryptu

Niniejszy skrypt w zamyśle przeznaczony jest dla studentów kierunku „Informatyka” i ma stanowić pomoc we wdrożeniu się w regularną pracę konsolową w środowisku systemu operacyjnego Linux, niemniej jednak może być pomocny dla słuchaczy dowolnego kierunku, którzy właśnie rozpoczynają swoją przygodę z Linuksem i nie mają wcześniejszych doświadczeń z pracą w tym systemie. Skrypt może również służyć jako podręczne vademecum w każdej sytuacji, która wymaga bezpośredniej interakcji z terminalem Linuksa.

Zestaw wybranych tematów skupia się na najczęściej spotykanych problemach i zagadnieniach rutynowej pracy użytkownika, pomijając kwestie związane z instalacją systemu, jego konfiguracją, administracją i utrzymaniem. Układ i kolejność przedstawianych treści ma umożliwić „szybki start” użytkownika do pracy z zupełnie nieznanym mu środowiskiem, pozwalając na szybką adaptację i bezbolesne nabycie nowych nawyków.

Każdy rozdział opisuje wybrany fragment systemu Linux, prezentując zwięzłe omówienie podstaw teoretycznych funkcjonowania danego komponentu, a następnie przechodzi przez omówienie typowych narzędzi używanych do wykonania rutynowych zadań i kończy się zestawem ćwiczeń do samodzielnego rozwiązania.

Skrypt skupia się nie tyle na przekazaniu słuchaczowi koniecznego do opanowania zestawu informacji, ile na wyrobieniu w nim nawyku i umiejętności samodzielnego poszukiwania rozwiązań oraz oswojenia go z możliwościami docierania do pomocy oferowanej wprost przez sam system i jego wbudowaną dokumentację.

0.2. Fenomen systemu Linux

Rozważania na temat Linuksa warto rozpocząć od wprowadzenia pewnego porządku w kwestie nazewnicze i ustalenie, co tak naprawdę rozumie się przez daną nazwę. Podchodząc do tego tematu formalnie i skrupulatnie, powinno się pamiętać, że Linux to tak naprawdę jedynie jądro systemu operacyjnego – komponent absolutnie niezbędny i decydujący o strukturze i własnościach całego systemu operacyjnego, jednak niezdolny do samodzielnej pracy i do pełnienia jakichkolwiek funkcji użytkowych. Dopiero obudowanie jądra obszernym zestawem towarzyszącego mu oprogramowania (jest to tak zwany *userspace*) konstytuuje prawdziwy i używalny system operacyjny, pozwalający jego użytkownikom na efektywną i wydajną pracę. Owo programowe otoczenie powstało niezależnie od jądra Linuksa i w przeważającej większości napisane zostało przez społeczność programistów skupionych wokół projektu GNU, dlatego też, aby uhonorować wszystkich twórców systemu, właściwszym jest nazywanie go systemem „GNU/Linux", którą to terminologię stosuje wiele ze współczesnych dystrybucji, a co bywa również używane do odróżnienia od systemów opartych o inne jądra (na przykład GNU/Linux i GNU/Hurd w przypadku dystrybucji Debian).

Powstanie i rozwój systemu GNU/Linux jest związane z dwiema niezwykłymi osobowościami i nie sposób nie wspomnieć o nich choćby zdawkowo. Są to:

- Linus Torvalds, urodzony w 1969 roku, pochodzący z Finlandii programista, który w 1991 roku zakodował i opublikował pierwszą wersję jądra, przeznaczoną dla procesorów Intela rodzin i386 i i486; warto tu dodać, że nazwa *Linux*, będąca dość oczywistą zbitką słów *Linus* i *Unix*, nie pochodzi od Torvaldsa, a została zaproponowana i rozpowszechniona przez Ariego Lemmke, pierwszego popularyzatora nowego systemu, któremu bardzo nie podobała się zaproponowana przez Linusa nazwa *Freax*; dopiero później pojawiły pomysły, by termin Linux tłumaczyć jako rekurencyjny skrót od słów „*Linux is not Unix*" (*Linux nie jest Uniksem*).
- Richard Matthew Stallman, znany także jako *rms*, urodzony w 1953 roku amerykański programista i aktywista Free Software Foundation, inicjator i koordynator rozpoczętego w 1983 roku projektu GNU (co pochodzić ma od rekurencyjnego skrótu *GNU is not Unix*), w wyniku którego powstała obszerna kolekcja oprogramowania stanowiącego kompletny *userspace*.

Tak więc ten system, który potocznie nazywamy Linuksem, oprócz samego Linuxa zawiera coś jeszcze – nieodzowny i nieodłączny komponent napisany i pielęgnowany przez programistów skupionych wokół Richarda Stallmana i przesiąkniętych jego ideami.

Przez trzy dekady swojego rozwoju system GNU/Linux pokonał długą drogę, od zabawki i niewinnego studenckiego eksperymentu do szacownego, poważnego i poważanego systemu operacyjnego, który w drugiej dekadzie XXI wieku możemy spotkać dosłownie wszędzie. I nawet ci, którzy z pełnym przekonaniem twierdzą, że, abstrahując od przyczyn, Linuksa nie używają i nigdy używać nie będą, każdego dnia są jego wiernymi użytkownikami, czy to poprzez fakt korzystania z urządzeń z systemem Android, czy też z powodu prowadzenia samochodu, którego system *infotainment* (lub jak kto woli: *inforozrywki*) jest oparty na Linuksie, czy też w końcu przy okazji odwiedzania witryn internetowych, posadowionych na serwerach pracujących pod kontrolą Linuksa.

Także Microsoft, który spędził wiele czasu i wydał sporo pieniędzy w toku sądowych batalii mających na celu zahamowanie rozwoju Linuksa i zniechęcenie społeczności do jego używania, obecnie przeszedł do polityki harmonijnej współpracy z Linuksem i integracji z jego użytkownikami, zgodnie ze starą zasadą mówiącą, że „jeśli nie możesz go pokonać, to do niego się przyłącz".

Znajomość podstaw Linuksa staje się powoli elementem powszechnej alfabetyzacji informatycznej i inaczej być nie może, bo Linux jest wszędzie i nie zanosi się na to, by miało być inaczej.

Kończąc ten historyczny wątek, warto jeszcze poruszyć ważną kwestię: jak mianowicie traktować Linuksa w polszczyźnie – jak tę nazwę zapisywać i odmieniać. Rada Języka Polskiego sugeruje w tej kwestii dwa podejścia, różniące się traktowaniem obcej polszczyźnie litery *x*. Oto one:

– wersja z *ks*
  M: *Linux*
  D: *Linuksa*
  C: *Linuksowi*
  B: *Linuksa*
  N: *Linuksem*
  Ms: *Linuksie*
  W: *Linuksie!*
– wersja z *x*
  M: *Linux*
  D: *Linuxa*
  C: *Linuxowi*
  B: *Linuxa*

N: *Linuxem*
Ms: *Linuxie*
W: *Linuxie!*

W tekście niniejszym zastosowana została wersja pierwsza, zdaniem autora – przyjaźniejsza duchowi polszczyzny.

0.3. Źródła uzupełniające

1. Strona osobista Linusa Torvaldsa: `https://linuxtorvalds.com/`
2. Strona osobista Richarda Stallmana: `https://stallman.org/`
3. Archiwa jądra Linux: `https://www.kernel.org/`
4. Strona projektu GNU: `https://gnu.org`

1. Terminal i polecenia podstawowe

1.1. Konwencje używane w tekście

Zasadniczy tekst skryptu zapisany jest czcionką o kroju proporcjonalnym. Czcionka o kroju stałym pojawia się w celu wskazania, że sformatowany nią tekst jest *poleceniem* wpisywanym z konsoli bądź *komunikatem* systemu operacyjnego wysyłanym do użytkownika. Elementy poleceń zapisane kursywą są *parametrami*, tzn. ciągami znaków o ściśle przypisanym znaczeniu, które muszą zostać określone przed wydaniem polecenia. Wielokropek (**`...`**) stojący za pewnym parametrem oznacza, że parametr ten może zostać *powtórzony* dowolną liczbę razy. Ujęcie pewnego fragmentu polecenia w *nawiasy kwadratowe* (**`[]`**) oznacza, że jest to element opcjonalny i może zostać pominięty.

Na przykład:

`cat` ***`[nazwa_pliku...]`***

- **`cat`** jest nazwą pewnego polecenia systemu Linux,
- polecenie to można wydać bez podawania parametrów (nawiasy kwadratowe) bądź też z dowolną liczbą parametrów (wielokropek) będących nazwami plików.

Parametry poleceń mogą być *przełącznikami* bądź *argumentami*. Argument to najczęściej **nazwa pliku**, na którym zostanie wykonana wskazana operacja, zaś

przełącznik (*opcja*) jest wskazówką, w jaki szczególny sposób owa operacja ma zostać wykonana. Każde polecenie ma własny unikalny zestaw przełączników, chociaż niektóre z nich, na przykład **-h** (od ang. *help*) najczęściej oznaczające żądanie wypisania krótkiego wyjaśnienia działania polecenia, mogą być wspólne dla wielu różnych poleceń.

Uwaga: polecenie, argumenty i przełączniki zawsze rozdziela się co najmniej jednym białym znakiem (w tej roli najczęściej występuje tu spacja).

Przykłady kompletnej konwersacji z systemem będą prezentowane jako tekst w ramce. W przykładach takich linie rozpoczynające się od znaku dolara (**$**) bądź kratki (**#**) odzwierciedlają tekst wprowadzony przez użytkownika, pozostałe zaś są odpowiedzią systemu.

Na przykład:

```
$ pwd
/home/user
```

Przykład powyższy obrazuje odpowiedź systemu na wydanie przez użytkownika polecenia **pwd**.

Niektóre z przełączników mogą mieć wersję długą (najczęściej pod postacią słowa lub słów języka angielskiego) i krótką (najczęściej pod postacią jednej litery alfabetu łacińskiego lub cyfry). Przełączniki krótkie poprzedza się jednym znakiem minus (**-**), długie – dwoma (**--**). Na przykład polecenie **watch** ma przełącznik **v** (od ang. *version*), który może zostać użyty jako krótki albo długi:

```
watch -v
watch --version
```

W obu przypadkach wynikiem wykonania polecenia będzie ujawnienie wersji używanego programu.

```
$ watch -v
watch from procps-ng 3.3.17
$ watch --version
watch from procps-ng 3.3.17
```

Niektóre z poleceń uruchomione bez parametrów prezentują skrótowy opis przeznaczenia i sposobu uruchomienia, ale nie jest to regułą. Wiele z poleceń reaguje w takiej sytuacji uruchomieniem zachowania domyślnego, które może zdezorientować nieświadomego użytkownika. Na przykład polecenie **cat** w takiej sytuacji zaczyna czekać na wprowadzanie danych z klawiatury, nie sygnalizując tego w żaden jawny sposób, co może wywoływać mylne wrażenie, że konsola po

prostu się zawiesiła. W takich sytuacjach pomocny może być długi przełącznik **--help**, który w większości przypadków powoduje ujawnienie skrótowego opisu przeznaczenia i działania danego polecenia. Warto zauważyć, że nie wszystkie polecenia dopuszczają postać krótką tego przełącznika, czyli **-h** – dzieje się tak na przykład w przypadku wspomnianego już polecenia **cat**).

```
$ cat -h
cat: invalid option -- 'h'
Try 'cat --help' for more information.
$ cat --help
Usage: cat [OPTION]... [FILE]...
Concatenate FILE(s) to standard output.
With no FILE, or when FILE is -, read standard input.
  -A, --show-all           equivalent to -vET
  -b, --number-nonblank    number nonempty output lines, overrides -n
  -e                       equivalent to -vE
  -E, --show-ends          display \$ at end of each line
  -n, --number             number all output lines
  -s, --squeeze-blank      suppress repeated empty output lines
  -t                       equivalent to -vT
  -T, --show-tabs          display TAB characters as \^I
  -u                       (ignored)
  -v, --show-nonprinting   use \^{} and M-notation, except for LFD and TAB
      --help     display this help and exit
      --version  output version information and exit
```

Przełączniki w postaci krótkiej w większości przypadków mogą być podawane w dowolnej kolejności. Wyjątkiem będą takie polecenia, w których kolejność przełączników narzuca kolejność wykonywanych operacji, ale jest to sytuacja dość rzadka i najczęściej wynika z samej natury działania polecenia.

Przełączniki mogą być również łączone ze sobą, co skraca zapis i czas wydawania polecenia. Na przykład wszystkie poniższe przypadki użycia przełączników **-l** i **-a** są równoważne:

```
ls -l -a
ls -a -l
ls -al
ls -la
```

W przypadkach, w których plik będący argumentem pewnego polecenia ma nazwę tożsamą z zapisem jakiegokolwiek przełącznika tego polecenia, pomocny może okazać się argument specjalny zapisywany jako **--**, który powoduje, że wszystkie następujące po nim argumenty traktowane są wyłącznie jako nazwy plików, nawet jeśli ich nazwy zaczynają się od **-** bądź **--**. Konwencja ta warta jest zapamiętania, chociaż takie przypadki zdarzają się raczej rzadko.

Jeśli na przykład w pewnym katalogu istnieje plik o nazwie **`--help`** (co jest absolutnie dopuszczalne), próba wypisania jego zawartości poleceniem:

```
cat --help
```

spali na panewce, ponieważ zamiast spodziewanych danych zobaczymy na terminalu tekst pomocy. Aby wybrnąć z tej opresji, powinniśmy napisać:

```
cat -- --help
```

1.2. Praca z terminalem i wydawanie poleceń

Po poprawnym zalogowaniu użytkownika do sesji terminalowej i wyświetleniu systemowej notki powitalnej, system przechodzi w tryb pracy interaktywnej. Oznacza to, że każdy ciąg znaków wprowadzony przez użytkownika i zakończony klawiszem *Enter* jest traktowany przez system (dokładniej: przez proces interpretera poleceń nazywany *powłoką*, ang. *shell*) jak polecenie z ewentualnymi parametrami, które powłoka ma rozpoznać i spróbować wykonać. Należy pamiętać, że rejestr liter użytych przy pisaniu polecenia oraz jego parametrów, w tym także przełączników, ma krytyczne znaczenie: **`pwd`** jest czymś innym niż **`PWD`**, a przełącznik **`-h`** zapewne wymusza zupełnie inne zachowanie niż **`-H`**. Zasada ta jest jedną z cech silnie odróżniających systemy wywodzące się z Uniksa od systemów rodziny Windows, gdzie rejestr liter nie ma znaczenia ani w nazwach poleceń, ani w ich parametrach, ani nawet w nazwach plików.

Gotowość do wykonywania poleceń jest sygnalizowana przez powłokę wyświetleniem ciągu znaków o zwyczajowej nazwie *prompt*. Słowo to nie doczekało się dobrego polskiego odpowiednika.

Postać prompta jest w pełni konfigurowalna i może wyglądać inaczej w różnych dystrybucjach Linuksa, najczęściej jednak przybiera postać podobną do poniższej:

```
user@host: dir $
```

gdzie:

user nazwa zalogowanego użytkownika;

host nazwa komputera, na którym pracuje powłoka;

dir nazwa katalogu bieżącego (pojęcie to zostanie omówione w dalszej części tekstu);

$ oznacza, że powłoka pracuje na rzecz użytkownika, który nie jest administratorem systemu (tzn. nie jest użytkownikiem *root* – nie mylić z katalogiem **`/root`**!); prompt administratora zwyczajowo kończy się znakiem **`#`** (hash), co w założeniu ma wywoływać u użytkownika wzmożenie jego uwagi, jako że w tej sytuacji ewentualnie popełniony błąd może mieć daleko idące konsekwencje dla stabilności całego systemu.

Długość polecenia nie jest ograniczona i można je pisać, zdając się na domyślne zachowanie terminala, który linię wykraczającą poza prawy margines ekranu samoczynnie przełamie, ale można też przerwać pisanie polecenia w dowolnym momencie, kończąc wpisany fragment znakiem odwrotnego ukośnika (****) i naciskając klawisz *Enter*. System uzna, że polecenie będzie kontynuowane w kolejnej linii i wstrzyma się z jego wykonaniem do chwili skompletowania całości, a zamiast prompta na początku nowej linii zostanie wyświetlony znak **>**.

Pewne kombinacje klawiszy są przez system traktowane specjalnie – najważniejsze z nich ujęte są w poniższym zestawieniu:

Klawisz	Działanie
Alt-F*n* ewentualnie **Ctlr-Alt-F*n***	przełączenie się na wirtualną konsolę o numerze *n* (na przykład **Alt-F2** przełącza na drugą konsolę wirtualną); *uwaga: działa tylko i wyłącznie, gdy terminal pracuje w faktycznej konsoli na fizycznym terminalu, a nie w konsoli emulatora terminala*
↑ ↓	przywołanie poprzedniego/następnego polecenia z historii poleceń
Tab	aktywowanie „podpowiadacza” nazw plików
Ctrl-Z	przerzucenie bieżącego procesu na drugi plan (polecenie **fg** przywróci proces z powrotem na pierwszy plan)
Ctrl-L	wyczyszczenie okna terminala
Ctrl-C	przerwanie i zakończenie bieżącego procesu
Ctrl-D	wysłanie znaku końca pliku (tak zwany znak *EOF*) do bieżącego procesu; ma sens tylko wtedy, gdy proces faktycznie oczekuje na wprowadzenie danych z konsoli; wysłanie tej kombinacji do powłoki systemowej wywołuje efekt natychmiastowego wylogowania się z systemu

1.3. Uzyskiwanie pomocy

W skład systemu Linux zwyczajowo wchodzi złożony zbiór dokumentów tekstowych opisujących różne aspekty systemu i jego narzędzia. Są to tak zwane *manual pages*, w polskim żargonie znane jako *manuale*. Dostęp do dokumentacji możliwy jest dzięki wbudowanej, interaktywnej przeglądarce uruchamianej poleceniem:

```
man nazwa_polecenia
```

gdzie *nazwa_polecenia* to nazwa programu, usługi systemowej, pliku konfiguracyjnego bądź innych obiektów systemowych, dla których chce się uzyskać pomoc, na przykład:

```
man cd
```

pozwala na uzyskanie pomocy dla polecenia **cd**.

Istnieje także strona pomocy dla samej pomocy systemowej, a dostęp do niej można uzyskać poleceniem:

```
man man
```

Wykonanie polecenia nie uda się, gdy wskazany dokument pomocy nie istnieje.

Strony dokumentacji podzielone są na sekcje, które grupują informacje zawarte w pomocy. Nazwy oraz kolejność sekcji podlegają standaryzacji, dzięki czemu ich studiowanie jest znacznie uproszczone.

W przypadku dokumentów opisujących narzędzia systemowe najczęściej spotykane sekcje to:

- NAME: nazwa oraz krótki komentarz lub wyjaśnienie,
- SYNOPSIS: sposoby uruchamiania programu lub polecenia wraz z listą możliwych przełączników,
- DESCRIPTION: pełen opis programu oraz szczegółowy opis możliwych do zastosowania przełączników,
- CONFIGURATION: opis konfiguracji usługi lub programu,
- FILES: opis plików, które wpływają na sposób działania programu lub usługi, na przykład plików konfiguracyjnych,
- SEE ALSO: wskazówki dotyczące podobnych lub powiązanych poleceń,
- BUGS: opisanie słabych punktów programu oraz okoliczności, w których program może dawać niepoprawne wyniki albo też zachować się w nieprzewidywalny sposób.

Całość dokumentacji podzielona jest na rozłączne rozdziały, które zawierają opisy poleceń i programów określonego typu. I tak:

- rozdział 1: „Programy wykonywalne lub polecenia powłoki” (ang. *User commands*) – programy konsolowe, z reguły dostępne dla każdego użytkownika systemu, choć niekoniecznie z takimi samymi funkcjonalnościami, na przykład polecenie **date** wykonywane przez zwykłego użytkownika potrafi jedynie podać bieżący czas na wiele różnych sposobów, a użyte przez administratora umożliwia natychmiastową zmianę ustawień zegara systemowego;
- rozdział 2: „Wywołania systemowe” (ang. *System calls*) – opis funkcji języka C stanowiących interfejs do usług jądra systemu, na przykład **man fork** udostępni opis funkcji jądra służącej do tworzenia nowych procesów;
- rozdział 3: „Wywołania biblioteczne” (ang. *Library calls*) – standardowe funkcje biblioteczne realizujące funkcjonalności implementowane poza jądrem, na przykład **man asin** udostępni opis grupy funkcji obliczających wartość funkcji trygonometrycznej *sin(x)*;

– rozdział 4: „Pliki specjalne” (ang. *Special files*) – opis przeznaczenia i zachowania plików przechowywanych w katalogu **/dev**, które są abstrakcjami urządzeń fizycznych, na przykład polecenie **man 4 hd** przedstawi opis plików **/dev/hd*** funkcjonujących jako obrazy zawartości dysków IDE wykrytych przez jądro w systemie;
– rozdział 5: „Formaty plików i konwencje” (ang. *File formats and configuration files*) – opisy przeznaczenia i wewnętrznej struktury różnorodnych plików tekstowych używanych do konfigurowania właściwości systemu operacyjnego, na przykład **man 5 passwd** przedstawi opis struktury pliku **/etc/passwd** zawierający dane wszystkich kont użytkowników znanych w danej instalacji;
– rozdział 6: „Gry” (ang. *Games*) – opis gier i programów służących rozrywce, na przykład polecenie **man fortune** zaprezentuje opis programu, który po uruchomieniu wypisuje na konsoli losowo wybraną sentencję, cytat bądź aforyzm (teksty te czerpane są z bazy rozbudowywanej od wielu dekad przez kolejne pokolenia użytkowników systemów uniksowych);
– rozdział 7: „Konwencje i rozmaitości” (ang. *Overview, conventions, and miscellaneous*) – opisy różnorodnych konwencji, standardów, protokołów i innych trudno klasyfikowalnych zagadnień, na przykład polecenie **man 7 man** udostępni opis formatu plików wykorzystywanych przez polecenie **man**;
– rozdział 8: „Polecenia do administracji systemem” (ang. *System management commands*) – dokumentacja narzędzi z reguły dostępnych tylko dla administratora systemu, na przykład polecenie **man fdisk** przedstawi instrukcję użycia podstawowego programu używanego do zarządzania partycjami twardych dysków.

Strony pomocy są jednoznacznie identyfikowane za pomocą hasła tematycznego i numeru rozdziału, na przykład **passwd(1)** oznacza, że dla polecenia **passwd** pomoc systemowa znajduje się w rozdziale 1. Jeżeli pewne hasło opisane jest w więcej niż jednym rozdziale, polecenie **man** domyślnie wyświetli to, które znajduje się w rozdziale o niższym numerze. Oznacza to, że polecenie w postaci:

```
man unlink
```

spowoduje wyświetlenie pomocy dla polecenia **unlink** (ten sam efekt dałoby polecenie **man 1 unlink**), natomiast polecenie w postaci:

```
man 2 unlink
```

wyświetli pomoc dla funkcji systemowej o nagłówku:

```
int unlink(const char *pathname);
```

dostępnej poprzez plik nagłówkowy **<unistd.h>**.

Pomoc systemowa wyświetlana jest za pomocą systemowego narzędzia **less**, które obsługuje się z klawiatury za pomocą następujących poleceń klawiszowych:

spacja lub *PgDn* – przejście do następnej strony,
Ctrl-B lub *PgUp* – przejście do poprzedniej strony,
q – zamknięcie i opuszczenie przeglądarki,
/ – wyszukiwanie w tekście w przód; po znaku **/** należy wpisać tekst do wyszukania i nacisnąć *Enter*,
? – wyszukiwanie wstecz (dalsze postępowanie jak wyżej),
n, *N* – przejście do następnego (*n*)/poprzedniego (*N*) wystąpienia poszukiwanego wyrażenia.

W zestawie powyższym szczególnie warte zapamiętania jest polecenie *q*, jako że intuicyjnie nasuwające się użycie klawisza *Esc* nie przynosi żadnego rezultatu.

Przeszukiwanie dokumentów pomocy systemowej w celu odnalezienia tych, które zawierają określone słowa kluczowe, jest możliwe dzięki programom **apropos** oraz **whatis**, które przeglądają wszystkie dokumenty pomocy dostępne w systemie i wyświetlają listę znalezisk, na przykład wydanie któregoś z poniższych poleceń:

```
apropos passwd
whatis passwd
```

doprowadzi do pojawienia się na konsoli listy dokumentów, które nawiązują bądź mogą nawiązywać do słowa *passwd*. Różnica między obydwoma poleceniami polega na głębokości przeprowadzonego przeszukania:

– **apropos** szuka podanych słów w tytułach dokumentów oraz w rozdziałach DESCRIPTION tych dokumentów,
– **whatis** szuka podanego słowa wewnątrz jednowierszowych opisów haseł pomocy (a nie w treści samych dokumentów).

Z tego też powodu wyniki dostarczane przez **apropos** są z reguły znacznie obszerniejsze.

Warto dodać, że oba polecenia akceptują też wieloznaczne definicje szukanych haseł, zapisane przy użyciu znaków specjalnych bądź pod postacią wyrażeń regularnych. Oba te tematy zostaną niebawem omówione szerzej.

Dokładniejsze informacje na temat obu poleceń zawiera manual.

1.4. Drzewo katalogów systemu GNU/Linux

Struktura katalogów systemu Linux tworzy drzewo (z pewnymi wyjątkami, które zostaną omówione w dalszej kolejności). Drzewo to ma korzeń nazywany katalogiem głównym (ang. *root directory*) i oznaczany znakiem ukośnika:

```
/
```

Każdy katalog może zawierać dowolną (w tym również zerową) liczbę plików, w tym katalogów (w systemach wywodzących się z Uniksa katalog jest szczególną formą pliku). Katalog zawarty w innym katalogu nazywa się *podkatalogiem* (ang. *subdirectory*).

W każdym momencie jeden z katalogów wchodzących w skład drzewa katalogów jest *katalogiem bieżącym* (*roboczym*). Katalog roboczy jest zawsze rozpoznawany pod nazwą skróconą zapisywaną jako pojedynczy znak kropki:

```
.
```

W większości przypadków użycie nazwy pliku pozbawionej informacji o jego położeniu w konkretnym katalogu spowoduje, że powłoka uzna, że chodzi o plik w katalogu bieżącym. Pozwala to na znaczne ułatwienie prac związanych z manipulowaniem plikami.

Reguła ta nie dotyczy jednak uruchamiania programów. Na przykład próba uruchomienia w konsoli własnoręcznie napisanego i skompilowanego programu o nazwie **`hello`** spowoduje, że powłoka przeszuka pewien zestaw katalogów domyślnych i jeśli nie znajdzie w nich pliku o podanej nazwie, zasygnalizuje błąd. Stanie się to także wtedy, gdy plik taki będzie znajdował się w katalogu bieżącym. W takim przypadku polecenie uruchomienia programu **`hello`** musi zawierać wskazania katalogu, w którym należy go szukać, i tu właśnie użycie kropki pozwala na uniknięcie żmudnego wprowadzania pełnej ścieżki.

```
$ hello
bash: hello: command not found
$ ./hello
Hello world!
```

Każdy katalog, z wyjątkiem głównego, ma jeden katalog nadrzędny (jest to tak zwany *katalog rodzicielski* lub *nadkatalog*, ang. *parent directory*), oznaczany dwiema kropkami:

```
..
```

Choć brzmi to zdecydowanie paradoksalnie, z punktu widzenia logiki organizacji drzewa katalogów systemów uniksowych, nadkatalogiem katalogu głównego jest sam katalog główny. Przyjęcie takiej zaskakującej fikcji pozwala ujednolicić wiele mechanizmów systemu plików.

Położenie pewnego katalogu względem katalogu głównego zapisuje się jednoznacznie ciągiem nazw katalogów prowadzących od katalogu głównego aż do danego katalogu. Nazwy katalogów rozdziela się znakiem ukośnika (należy zwrócić

uwagę, że żaden katalog nie może tego znaku zawierać w swojej nazwie), a całość tworzy tak zwaną *nazwę kanoniczną*. Na przykład:

```
/home/user/subdir
```

opisuje katalog o nazwie *subdir*, który jest podkatalogiem katalogu *user* będącego podkatalogiem katalogu *home* będącego z kolei podkatalogiem katalogu głównego. Warte uwagi jest to, że brak pierwszego znaku ukośnika może w sposób radykalny zmienić znaczenie takiego zapisu, a nawet uczynić go niepoprawnym, ponieważ w takim przypadku podgałąź drzewa katalogów będzie interpretowana tak, jakby „wyrastała" z katalogu bieżącego, a nie z katalogu *root*!

Po poprawnym logowaniu katalogiem bieżącym użytkownika (o ile nie zdefiniowano tego inaczej) staje się tak zwany *katalog domowy* (ang. *home directory*), przydzielony każdemu z użytkowników systemu w chwili tworzenia jego konta. Katalog domowy danego użytkownika jest dla niego rozpoznawalny pod skróconym oznaczeniem zapisywanym przy pomocy symbolu tyldy:

```
~
```

Poruszanie się po drzewie katalogów (a tym samym zmianę katalogu bieżącego) umożliwia polecenie **cd** (ang. *change directory)*:

```
cd [nazwakatalogu]
```

Poprawne wykonanie polecenia spowoduje zmianę katalogu bieżącego na wskazany w parametrze. Wykonanie polecenia nie uda się, gdy zajdzie któraś z poniższych okoliczności:

- wskazany katalog nie istnieje,
- użytkownik wykonujący polecenie **cd** nie ma uprawnień pozwalających na przemieszczenie się do wskazanego katalogu.

W przypadku pomyślnego wykonania katalog bieżący zostanie zmieniony na katalog wskazany w parametrze. Warty uwagi jest fakt, że postać, w jakiej zapisano nazwę katalogu, oraz to, w jakim katalogu polecenie wydano, ma dramatyczny wpływ na uzyskany w efekcie wynik. Na przykład:

- **cd home**
  przejdź do tego podkatalogu *home*, który znajduje się w katalogu bieżącym,
- **cd /home**
  przejdź do podkatalogu *home* katalogu głównego.

Polecenie **cd** ma kilka interesujących oboczności:

cd .
określa żądanie przejścia do katalogu bieżącego, a więc – mimo formalnej poprawności – skutek wykonania takiego polecenia będzie żaden (katalog bieżący nie ulegnie zmianie),

cd ..
określa żądanie przejścia do katalogu nadrzędnego i – co może wydać się paradoksalne – zostanie pomyślnie wykonane również na poziomie katalogu głównego, chociaż w tym przypadku nie odniesie żadnych skutków,

cd
jest najprostszym i najszybszym sposobem przejścia do własnego katalogu domowego, będąc w tej postaci skrótowym zapisem polecenia **cd** ∼,

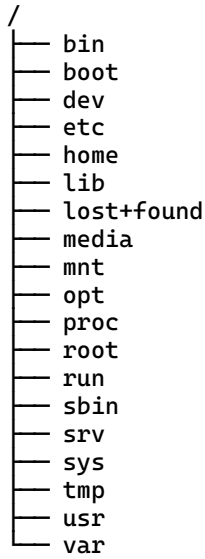
cd -
zmienia katalog bieżący na ten, w którym użytkownik przebywał poprzednio; regułę tę należy brać dosłownie: kolejno wykonywane polecenia w tej postaci będą przemieszczać użytkownika tylko i wyłącznie pomiędzy dwoma ostatnio odwiedzonymi katalogami.

1.5. Standardowa hierarchia katalogów systemu GNU/Linux

Struktura katalogów systemu GNU/Linux jest w znaczącej części zestandaryzowana, zarówno jeśli chodzi o konwencje nazewnicze, jak i przeznaczenie poszczególnych katalogów. Zbiór takich konwencji nosi nazwę FHS (od ang. *Filesystem Hierarchy Standard* – standard hierarchii systemu plików). Różne dystrybucje Linuksa mogą tę konwencję rozszerzać bądź częściowo ignorować, jednak jej zręby wydają się uniwersalne.

Najczęściej pierwszy poziom podkatalogów prezentuje się jak poniżej:
Przeznaczenie wybranych katalogów prezentuje poniższa lista:

- **/bin** – katalog zawierający podstawowe programy, dostępne dla wszystkich użytkowników i niezbędne do rutynowego użytkowania systemu,
- **/boot** – katalog zawierający pliki niezbędne do poprawnego wystartowania systemu operacyjnego,
- **/dev** – katalog zawierający tak zwane *pliki specjalne*, reprezentujące dostępne w systemie urządzenia; można uznać, że każdy plik w tym katalogu jest abstrakcją pewnego urządzenia fizycznego rozpoznanego przez system; pewna operacja wykonana na takim pliku będzie faktycznie wykonana na stowarzyszonym z plikiem urządzeniu; na przykład odczyt 512 bajtów z pliku



Rys. 1.1. Zawartość katalogu głównego według standardu FHS

/dev/sda może *de facto* oznaczać odczytanie pierwszego sektora urządzenia dyskowego dostępnego w bieżącej konfiguracji komputera,

– **/etc** – katalog z lokalnymi plikami konfiguracyjnymi systemu; to tu umieszczony jest na przykład plik **passwd** przechowujący informacje o znanych systemowi kontach użytkowników,
– **/home** – katalog zawierający podkatalogi domowe użytkowników systemu; zwyczajowo nazwą katalogu domowego użytkownika *xyz* będzie **/home/xyz**; w instalacjach o dużej liczbie kont użytkowników może zostać wykorzystana konwencja z dodatkowym pośrednim poziomem katalogów i wtedy nazwa przybierze postać **/home/x/xyz**,
– **/lib** – katalog przechowujący biblioteki i moduły, wykorzystywane przez pracujące procesy i jądro systemu,
– **/lost+found** – katalog zakładany automatycznie w chwili tworzenia systemu plików i przeznaczony do umieszczania w nim plików zagubionych w wyniku awarii, a odnalezionych przez narzędzia naprawcze bez możliwości ustalenia, w jakim katalogu pliki te były przechowywane oryginalnie; jest to swego rodzaju „biuro rzeczy znalezionych”,
– **/media** – katalog, którego podkatalogi reprezentują urządzenia automatycznie i dynamicznie podłączone do systemu, na przykład pendrajwy; operację dołączenia nowego urządzenia do drzewa katalogów nazywa się *montowa-*

niem (ang. *mount*), a odłączenie urządzenia – *odmontowaniem* (ang. *unmount*),
- **/mnt** – katalog, którego podkatalogi reprezentują urządzenia statycznie dołączone do drzewa katalogów (najczęściej w chwili startu systemu operacyjnego),
- **/opt** – katalog zawierający oprogramowanie opcjonalne, najczęściej instalowane przez użytkownika i niestanowiące standardowego wyposażenia systemu operacyjnego,
- **/proc** – wirtualny system plików, który dostarcza między innymi informacji o procesach obecnych w systemie oraz o rozpoznanej przez jądro konfiguracji sprzętowej; „wirtualność" plików w tym katalogu oznacza, że pliki te nie są przechowywane fizycznie na nośniku, ale są generowane „w locie" na konkretne żądanie,
- **/root** – zwyczajowo katalog domowy użytkownika *root*, czyli administratora systemu,
- **/run** – katalog, którego zadaniem jest przechowywanie plików tymczasowych tworzonych przez pracujące procesy, a które to pliki muszą być chronione przed nieuprawnionym usunięciem lub zmodyfikowaniem, czego nie zapewnia starszy koncepcyjnie mechanizm udostępniany przez katalog **/tmp**,
- **/sbin** – pełni rolę podobną do katalogu **/bin**, z tym, że przechowuje programy i narzędzia przeznaczone do użytku przez administratora systemu,
- **/srv** – katalog przechowujący dane potrzebne do poprawnej pracy serwerów różnych usług systemu operacyjnego,
- **/sys** – katalog zawierający wirtualny system plików o działaniu podobnym do katalogu **/proc**, z tym, że przeznaczony do bezpośredniej komunikacji z usługami jądra systemu, badania ich aktualnego stanu bądź wpływania na ich stan,
- **/tmp** – katalog przeznaczony do przechowywania plików tymczasowych,
- **/usr** – katalog zawierający zestaw oprogramowania użytkowego dostępnego dla użytkowników oraz pliki umożliwiające pisanie własnych programów, w tym pliki nagłówkowe i i biblioteki,
- **/var** – katalog ten zawiera pliki, które często zmieniają swoją zawartość i rozmiar, na przykład kroniki (tak zwane *logi*), pliki tymczasowe różnych procesów i tym podobne.

1.6. Manipulowanie katalogami

Jeżeli uprawnienia użytkownika na to pozwalają, może on tworzyć nowe katalogi i usuwać katalogi już istniejące. Używa się do tego celu poleceń **mkdir** i **rmdir**.

– **mkdir *[przełącznik...] nazwakatalogu...***
(od ang. *make directory*) – powoduje utworzenie katalogu o wskazanej nazwie; wykonanie polecenia nie uda się, gdy zajdzie jedna z poniższych okoliczności:
  - taki katalog już istnieje,
  - posiadane uprawnienia nie pozwalają na tworzenie plików (katalogów) we wskazanym miejscu drzewa katalogów.

  Oto kilka przykładów:
  - utworzenie katalogu *xyz* w katalogu bieżącym:
    mkdir xyz
  - utworzenie katalogu *xyz* w bezpośrednim nadkatalogu:
    mkdir ../xyz
  - utworzenie katalogu *xyz* w katalogu głównym (z reguły możliwe do wykonania tylko w przypadku posiadania uprawnień administracyjnych):
    mkdir /xyz

– **rmdir *[przełącznik...] nazwakatalogu...***
(od ang. *remove directory*) – usuwa katalog o wskazanej nazwie, przy czym usuwany katalog musi być pusty; wykonanie polecenia nie uda się, gdy ma miejsce jedna z poniższych sytuacji:
  - katalog o takiej nazwie nie istnieje;
  - katalog nie jest pusty (zawiera niezerową liczbę plików);
  - użytkownik nie dysponuje odpowiednimi uprawnieniami.

  Oto kilka przykładów:
  - usunięcie katalogu *xyz* z własnego katalogu domowego:
    rmdir ~xyz
  - usunięcie podkatalogu *xyz* z katalogu bieżącego:
    rmdir xyz

  Uwaga: dla poleceń **rmdir** *i* **mkdir** *dostępny jest przełącznik* **-p** *(od ang.* path*), który pozwala odpowiednio usuwać i tworzyć więcej niż jeden katalog jednocześnie, o ile tylko tworzą one spójną hierarchię, na przykład jednoczesne usunięcie katalogów* ghi*,* def *oraz* abc*, które wspólnie tworzyły hierarchię, można przeprowadzić jednym poleceniem zamiast trzema:*
  rmdir -p abc/def/ghi

Poruszanie się po drzewie katalogów i orientowanie się w ich zawartości ułatwiają poniższe polecenia:

– **pwd**
Polecenie **pwd** (od ang. *print working directory*) wypisuje na konsolę kanoniczną (tzn. zaczepioną w katalogu głównym) nazwę bieżącego katalogu;

– **pushd *katalog...***
Polecenie **pushd** powoduje odłożenie nazw wskazanych katalogów (należy podać co najmniej jedną) na szczyt tak zwanego *stosu katalogów* (ang. *directory stack*); stos katalogów przechowywany jest do końca sesji, po czym jest niszczony; dodatkowo polecenie to wypisuje aktualny stan stosu; odłożenie na stos nazwy katalogu bieżącego można wykonać na przykład tak:

```
pushd .
```

– **popd**
Polecenie **popd** powoduje zdjęcie ze szczytu stosu katalogów znajdującej się tam ewentualnie nazwy (jeśli stos jest pusty, wykonanie polecenia sprowadza się do wyświetlenia komunikatu o błędzie), a następnie wykonuje polecenie **cd** dla tej właśnie nazwy; oba powyższe polecenia pozwalają na wygodne zapamiętywanie dowolnego ciągu położeń w drzewie katalogów i szybki powrót do wcześniej zajmowanych miejsc; *uwaga: **pushd** i **popd** są realizowane wewnętrznie przez powłokę, a nie przez zewnętrzny program, i, co za tym idzie, nie mają swojej strony pomocy; dokładny opis można uzyskać, gdy wykona się polecenie:*

```
man bash
```

i wyszuka stosowny fragment dotyczący współdziałania obu poleceń.
– **ls *[przełącznik...] [nazwa_katalogu...]***
Polecenie **ls** (od ang. *list*) wyświetla informację o zawartości katalogu bądź katalogów, na przykład:
  - **ls**
    wyświetla zwięzłe zestawienie zawartości katalogu bieżącego (tylko nazwy i tylko pliki/katalogi widoczne);
  - **ls -a**
    wyświetla zwięzłą informację o zawartości katalogu bieżącego z uwzględnieniem wszystkich plików (od ang. *all*), to znaczy także tych, których nazwa zaczyna się od znaku „.” (umownie są to pliki *ukryte*);
  - **ls -al**
    wyświetla informację o wszystkich plikach w katalogu bieżącym z wykorzystaniem tak zwanego *długiego formatu* (od ang. *long*), czyli podaje typ każdego obiektu w katalogu (pierwszy znak linii: **d** – katalog, znak **-** – plik zwykły, **l** – dowiązanie), prawa dostępu, liczbę dowiązań, nazwę właściciela, nazwę grupy właścicielskiej, rozmiar (w bajtach), datę ostatniej modyfikacji oraz samą nazwę pliku/katalogu;

- **`ls -al ~`**
  jak wyżej, przy czym wyświetlana jest zawartość katalogu domowego;
- **`ls -al /etc`**
  jak wyżej, ale wyświetlana jest zawartość katalogu **`/etc`**.

1.7. Manipulowanie plikami

Plik to kontener do składowania danych. Jest przechowywany w systemie plików i opatrzony szeregiem atrybutów takich jak nazwa, rozmiar, prawa dostępu i tym podobnych. W systemach GNU/Linux większość obiektów systemowych (na przykład urządzeń) jest na poziomie usług systemu prezentowana tak, jakby były plikami, nawet wtedy, gdy ich fizyczna reprezentacja nie ma z plikiem nic wspólnego. Pozwala to na zachowanie spójnego i przenośnego sposobu dostępu i obsługi do wielu heterogenicznych zasobów, na przykład komunikacja procesów poprzez sieć odbywa się tak, jakby transmisja danych była realizowana poprzez zapisy do plików i odczyty z nich.

W systemach wywodzących się z Uniksa nazwy plików nie mają formalnego podziału na nazwę i rozszerzenie (całość nazwy pliku traktowana jest jak jeden spójny ciąg znaków), jednakże mimo to tradycyjne rozumienie pojęcia *rozszerzenie* daje się pomyślnie stosować.

Możliwe jest używanie w nazwach plików znaków specjalnych (na przykład: $, %, :, #), ale nie jest to zalecane.

W szczególności (co stanowi przedmiot bardzo złośliwego, klasycznego uniksowego żartu) można wręcz nadać plikowi nazwę następującą:

```
*
```

Bezrefleksyjna próba usunięcia takiego pliku poleceniem **`rm`** o poniższej postaci:

```
rm *
```

spowoduje kataklizm w katalogu bieżącym, bowiem w takim przypadku gwiazdka nie oznacza samej siebie, ale staje się znakiem wieloznacznym, pasującym do nazwy dowolnego pliku.

Poprawna i mniej niszczycielska postać takiego polecenia prezentuje się następująco:

```
rm "*"
```

Nazwy plików mogą również zawierać spacje. Powoduje to pewne komplikacje przy manipulowaniu takimi plikami. Na przykład pliku o nazwie „tajny plik” nie da się wyświetlić poleceniem **`cat`** wydanym w sposób następujący:

```
cat tajny plik
```

Dzieje się tak dlatego, że w takim przypadku spacja nie jest brana za część nazwy pliku, lecz za separator rozdzielający parametry polecenia, co w tym przypadku oznacza, że polecenie **cat** uruchomiono z dwiema nazwami plików, a nie z jedną.

W przypadkach, gdy nazwa pliku zawiera znaki niealfanumeryczne, może być konieczne albo ujęcie nazwy w cudzysłów, albo poprzedzenie znaków specjalnych znakiem odwróconego ukośnika (\). Dotyczy to w szczególności sytuacji, gdy znaków **** i **"** używa się wewnątrz nazw plików. Oba poniższe polecenia są poprawne i równoważne:

```
cat "tajny plik"
cat tajny\ plik
```

Podstawowe operacje manipulacji plikami można realizować z wykorzystaniem następujących poleceń:

– **cp** ***[przełącznik...] nazwa_pliku... nowa_nazwa_lub_katalog***
kopiowanie pliku określonego przez pierwszy i ewentualne kolejne argumenty pod nazwę lub do katalogu określonego ostatnim argumentem, na przykład:

- **cp abc.txt xyz.txt**
kopiuje plik *abc.txt* pod nową nazwę *xyz.txt* w katalogu bieżącym,
- **cp /tmp/abc.txt ~**
kopiuje plik *abc.txt* z katalogu */tmp* do katalogu domowego użytkownika,
- **cp abc.txt ~/xyz.txt**
kopiuje plik *abc.txt* z katalogu bieżącego pod nową nazwę *xyz.txt* w katalogu domowym użytkownika,
- **cp abc.txt xyz.txt ~/documents**
kopiuje pliki *abc.txt* i *xyz.txt* z katalogu bieżącego do podkatalogu *documents* w katalogu domowym użytkownika; przypadek ten można uogólnić następującą regułą: jeśli do polecenia **cp** podano więcej niż dwie nazwy plików, to wszystkie nazwy oprócz ostatniej są nazwami obiektów do skopiowania, a nazwa ostatnia musi być nazwą istniejącego katalogu, do którego mają trafić kopiowane obiekty.

Przydatnym przełącznikiem polecenia **cp** jest **-r** (od ang. *recursive*), który służy do kopiowania całych struktur katalogów. *Uwaga: domyślnie polecenie*

***cp** w żaden sposób nie ostrzega przed utratą (nadpisaniem) istniejących plików.*

– **rm *[przełącznik...] nazwa_pliku...***
usuwanie plików podanych jako argumenty wywołania, na przykład:
  - **rm abc.txt xyz.txt**
    usuwa pliki *abc.txt* i *xyz.txt* w katalogu bieżącym,
  - **rm /tmp/abc.txt**
    usuwa plik *abc.txt* z katalogu */tmp*.

*Uwaga: domyślnie polecenie **rm** w żaden sposób nie ostrzega przed utratą istniejących plików. Ponadto w czasie pracy w konsoli nie działa żaden mechanizm ochronny, znany ze środowisk graficznych pod nazwą kosza. Usunięcie pliku/katalogu przy pomocy polecenia **rm** jest w większości przypadków nieodwracalne, dlatego wskazana jest daleko idąca ostrożność.*
Przydatnym przełącznikiem polecenia **rm** jest przełącznik **-r**, który służy do usuwania całych struktur katalogów wraz z zawartością, co pozwala unikać niewygód związanych z użyciem polecenia **rmdir**. Opcji tej używa się często w połączeniu z przełącznikiem **-f** (od ang. *force*), który powoduje, że operacja usuwania przeprowadzana jest bez interakcji z użytkownikiem i bez sygnalizowania błędów.

– **mv *[przełącznik...] nazwa_pliku... nowa_nazwa***
zmienia nazwę pliku określonego pierwszym argumentem wywołania na nazwę określoną ostatnim argumentem wywołania, jednak jeśli ostatni argument jest katalogiem, to wówczas plik zostanie przeniesiony do tego katalogu, na przykład:
  - **mv abc.txt xyz.txt**
    zmiana nazwy pliku *abc.txt* na *xyz.txt* w katalogu bieżącym, chyba że *xyz.txt* jest nazwą katalogu – w takim przypadku plik *abc.txt* zostanie przeniesiony do tego katalogu bez zmiany nazwy;
  - **mv /tmp/abc.txt ~**
    przeniesienie pliku *abc.txt* z katalogu */tmp* do katalogu domowego użytkownika.

*Uwaga: domyślnie polecenie **mv** w żaden sposób nie ostrzega przed utratą (nadpisaniem) istniejących plików. Ponadto wartym zapamiętania jest fakt, że w odróżnieniu od poleceń **cp** i **rm** polecenie **mv** nie używa opcji **-r**.*

– **touch *[przełącznik...] nazwa_pliku...***
modyfikuje informacje na temat czasów modyfikacji i dostępu pliku, ale pozwala także na utworzenie nowego pliku, na przykład:

```
touch abc.txt
```

utworzy nowy (pusty) plik *abc.txt* w katalogu bieżącym, o ile taki plik

nie istniał już wcześniej; w przeciwnym przypadku **touch** ograniczy się do aktualizacji jego metadanych odnoszących się do czasów modyfikacji i dostępu;

– **cat *[nazwa_pliku...]***
polecenie **cat** (od ang. *concatenate*) zostało wymyślone jako uniwersalny „sklejacz", pozwalający połączyć ze sobą (skonkatenować) zawartość dowolnej liczby plików, jednak na razie zostanie użyty do zupełnie innego celu, a mianowicie do wypisywania zawartości pliku na terminal; wypisywanie plików tekstowych zwykle przebiega bez zakłóceń, jednak próba potraktowania w ten sam sposób pliku binarnego (na przykład wykonywalnego albo zawierającego mapę bitową) może doprowadzić terminal do zachowań niecodziennych (na przykład zaśmiecenie ekranu dziwnymi znaczkami, kakofonia pisków, a na koniec niemożność napisania czegokolwiek, co byłoby widoczne na ekranie). Efekt taki ma źródło w fakcie, że pewne sekwencje znaków są przez terminal traktowane jako znaki sterujące i w niesprzyjających okolicznościach może to spowodować kompletną dezorganizację jego ustawień. W takich sytuacjach należy zachować zimną krew i nawet na ślepo (nie widząc efektów na ekranie) wpisać polecenie:

```
reset
```

po czym nacisnąć klawisz *Enter*. Terminal zostanie przywrócony do stanu stabilnego.

– **less *[nazwa_pliku...]***
prosty *page*r (czyli narzędzie do dzielenia pliku tekstowego na strony i wyświetlania go w sposób interaktywny); klawisze sterujące wyświetlaniem zostały opisane w punkcie poświęconym poleceniu **man**.

Polecenia dotyczące plików i katalogów można także wydawać z wykorzystaniem tak zwanych *wzorców uogólniających*, znanych także jako *znaki wieloznaczne* (ang. *wildcards*), które tworzy się z zastosowaniem następujących operatorów:

***** – zastępuje dowolny ciąg znaków (także pusty),
? – zastępuje dokładnie jeden dowolny znak,
[<znaki>] – zastępuje dokładnie jeden znak z podanego zbioru, na przykład **[xyz]** zastępuje dowolny z podanych trzech znaków **x**, **y** lub **z**,
[^<znaki>] – znak *caret* (**^**) użyty jako pierwszy znak zbioru oznacza dopełnienie tego zbioru, czyli na przykład **[^xyz]** oznacza jeden dowolny znak nie będący ani literą **x**, ani **y**, ani **z**.

Oto przykładowe polecenia z wykorzystaniem wzorców uogólniających:

- **`cp ./*.txt ~`**
  kopiowanie wszystkich plików z rozszerzeniem **`.txt`** z katalogu bieżącego do katalogu domowego użytkownika,
- **`rm ./[0-9]*`**
  usunięcie tych plików z katalogu domowego, których nazwa rozpoczyna się od cyfry dziesiętnej.

1.8. Inne przydatne polecenia

– **`id`**
Polecenie **`id`** wydane bez parametrów spowoduje wyświetlenie zestawu identyfikatorów przypisanych użytkownikowi wykonującemu to polecenie. Są to kolejno:
  - identyfikator użytkownika (*uid*),
  - identyfikator grupy macierzystej (podstawowej) użytkownika (*gid*),
  - lista identyfikatorów wszystkich grup, do których należy użytkownik (*groups*).

```
$ id
uid=1000(user) gid=1000(user) groups=1000(user),6(disk),18(audio)
```

– **`who`**
Polecenie **`who`** spowoduje wyświetlenie listy wszystkich zalogowanych użytkowników (dokładniej: wszystkich sesji terminalowych wszystkich zalogowanych użytkowników); podobną funkcję realizują polecenia **`w`** i **`finger`**.

```
$ who
user     pts/0        2022-01-01 13:00 (192.168.0.100)
$ w
14:00:00 up 100 days, 10:20,  2 users,  load average: 0,10, 0,12, 0,12
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
user     pts/0    192.168.0.100    13:27    0.00s  0.17s  0.04s w
$ finger
Login     Name       Tty      Idle  Login Time   Office     Office Phone
user      user       pts/0          Jan 01 13:27 (192.168.0.100)
```

– **`exit`**
Polecenie **`exit`** spowoduje zakończenie bieżącej sesji i wylogowanie użytkownika.

– **`watch polecenie [parametr...]`**
Polecenie **`watch`** pozwala wykonywać dowolne inne polecenie w sposób ciągły, nadając mu pozory interaktywności; domyślne działanie polecenia **`watch`** można opisać następująco:

1. Wyczyść ekran terminala.
2. Wykonaj wskazane polecenie.
3. Odczekaj 2 sekundy.
4. Idź do punktu 1.

Kombinacja klawiszy *Ctrl-C* przerywa powyższą pętlę i kończy wykonanie polecenia **watch**. Wydanie polecenia w postaci:

```
watch ls -al
```

umożliwi bieżący (z dokładnością do dwóch sekund) podgląd zmian w zawartości bieżącego katalogu.

1.9. Źródła uzupełniające

1. Cytowanie znaków i łańcuchów w powłoce Bash: https://www.gnu.org/software/bash/manual/bash.html#Quoting
2. Manual polecenia **apropos**: https://man7.org/linux/man-pages/man1/apropos.1.html
3. Manual powłoki Bash: https://www.man7.org/linux/man-pages/man1/bash.1.html
4. Manual polecenia **cat**: https://man7.org/linux/man-pages/man1/cat.1.html
5. Manual polecenia **cd**: https://www.man7.org/linux/man-pages/man1/cd.1p.html
6. Manual polecenia **cp**: https://man7.org/linux/man-pages/man1/cp.1.html
7. Manual polecenia **finger**: https://man.openbsd.org/finger
8. Manual polecenia **id**: https://man7.org/linux/man-pages/man1/id.1.html
9. Manual polecenia **ls**: https://man7.org/linux/man-pages/man1/ls.1.html
10. Manual polecenia **man**: https://man7.org/linux/man-pages/man1/man.1.html
11. Manual polecenia **mkdir**: https://man7.org/linux/man-pages/man1/mkdir.1.html
12. Manual polecenia **mv**: https://www.man7.org/linux/man-pages/man1/mv.1.html
13. Manual polecenia **pwd**: https://www.man7.org/linux/man-pages/man1/pwd.1.html
14. Manual polecenia **reset**: https://www.man7.org/linux/man-pages/man1/reset.1.html
15. Manual polecenia **rm**: https://www.man7.org/linux/man-pages/man1/rm.1.html
16. Manual polecenia **rmdir**: https://www.man7.org/linux/man-pages/man1/

rmdir.1.html

17. Manual polecenia **touch**: https://man7.org/linux/man-pages/man1/touch.1.html
18. Manual funkcji systemowej **unlink()**: https://man7.org/linux/man-pages/man2/unlink.2.html
19. Manual polecenia **unlink**: https://man7.org/linux/man-pages/man1/unlink.1.html
20. Manual polecenia **w**: https://man7.org/linux/man-pages/man1/w.1.html
21. Manual polecenia **watch**: https://man7.org/linux/man-pages/man1/watch.1.html
22. Manual polecenia **whatis**: https://man7.org/linux/man-pages/man1/whatis.1.html
23. Manual polecenia **who**: https://man7.org/linux/man-pages/man1/who.1.html
24. Standard FHS 3.0: https://refspecs.linuxfoundation.org/FHS_3.0/fhs/index.html
25. Znaki wieloznaczne w interpretacji powłoki Bash: https://www.gnu.org/software/bash/manual/bash.html#Pattern-Matching

1.10. Zadania do samodzielnego wykonania

1. Zbadaj szczegółowo działanie „podpowiadacza" nazw plików, który uaktywnia się, gdy podczas pisania polecenia użyjesz klawisza *Tab*. W tym celu przy użyciu polecenia **touch** załóż w dowolnym pustym katalogu pliki o poniższych nazwach:

   aaa
   aab
   aac
   aba
   abb
   abc

   a następnie, będąc w tymże katalogu, napisz w linii poleceń:

   ```
   cat a
   ```

   i naciśnij dwukrotnie klawisz *Tab*. Teraz dopisz literę **a** lub **b** i ponownie naciśnij klawisz *Tab*. Uważnie obserwuj zachowanie konsoli.
2. Przetestuj zachowanie opisane w poniższym fragmencie strony pomocy powłoki *Bash* (pisownia zgodna z oryginałem – przyp. aut.):
   Uzupełnianie complete (TAB):
   Usiłuje przeprowadzić uzupełnianie tekstu przed punktem. Bash próbuje uzupełniania traktując tekst kolejno: jako zmienną (jeżeli tekst zaczyna się od $), nazwę

użytkownika (jeśli tekst zaczyna się od ~), nazwę hosta (jeśli tekst zaczyna się od @) lub polecenie (łącznie z aliasami i funkcjami). Jeżeli żadne z powyższych nie daje dopasowania, to próbowane jest uzupełnianie nazw plików.

3. Sprawdź, jaki efekt wywołują następujące postaci polecenia **cd**:

```
cd .
cd ..
cd
cd -
```

4. Zbadaj zachowanie poleceń **pushd** i **popd**.
5. Polecenie **history** powoduje wypisanie historii dotychczas wydanych poleceń. Każde z poleceń w historii opatrzone jest numerem. Wydanie polecenia:

```
!n
```

(gdzie *n* jest numerem jednego z poleceń w historii) powoduje ponowienie wykonania tego polecenia. Sprawdź, czy po wydaniu takiego polecenia w historii zmieni się numeracja.

6. Użyj dokumentu pomocy polecenia **ls**, aby odpowiedzieć na poniższe pytania:
   - Jak wypisać nazwy plików/katalogów w jednej kolumnie?
   - Jak posortować listę plików według ich rozmiaru?
7. Polecenie w postaci:

```
ls katalog
```

spowoduje wypisanie zawartości katalogu *katalog*. Jak nakłonić polecenie **ls**, aby zamiast tego podało tylko informacje o katalogu *katalog* bez wypisywania jego zawartości?

8. Czy możliwe jest istnienie w pewnym katalogu pliku i katalogu o takiej samej nazwie?
9. Załóż w swoim katalogu domowym następujące poddrzewo katalogów:

```
~
└── dira
    ├── diraa
    │   ├── diraaa
    │   └── diraab
    └── dirab
        ├── diraaa
        └── diraab
```

W każdym podkatalogu stwórz poleceniem **touch** jeden plik nazwany tak, jak ciąg znaków umieszczony po słowie *dir* w nazwie katalogu (na przykład w katalogu *dira* plik *a*). Użyj tej struktury do przetrenowania następujących działań:

   - skopiowania pliku z *dira/diraa/diraaa* do *dira/dirab/diraab*;
   - przemianowania katalogu *dira/dirab/diraab* na *fake*;
   - przeniesienia całej gałęzi zaczynającej się na katalogu *dira/dirab* do katalogu *dira/diraa*.
10. Uwaga, bądź ostrożny – to niebezpieczne!
    Jaki efekt wywołuje użycie polecenia **`rm -rf *`** ?
11. Usuń całe drzewo katalogów z zadania 9. wraz z zawartością.
12. Sprawdź zachowanie poleceń **`cp`** i **`mv`** w sytuacji, gdy wywołano je z dwoma argumentami i drugi argument określa:
    - istniejący plik;
    - istniejący katalog;
    - nieistniejący plik/katalog.
13. Sprawdź zachowanie poleceń **`cp`** i **`mv`** w sytuacji, gdy wywołano je z trzema argumentami, z których dwa pierwsze są nazwami istniejących plików, a trzeci:
    - jest nazwą istniejącego pliku;
    - jest nazwą istniejącego katalogu;
    - jest nazwą nieistniejącego pliku/katalogu.
14. Jaki efekt wywołuje użycie opcji **`-h`** polecenia **`ls`**?
15. Polecenie **`df`** powoduje wypisanie informacji o wolnym miejscu na dyskach dostępnych w systemie. Użyj polecenia **`watch`**, aby co jedną sekundę wyświetlać bieżące dane o dostępnej pamięci dyskowej.
16. Co odróżnia od siebie dane prezentowane przez polecenia **`w`**, **`who`** i **`finger`**?
17. Zbadaj, co zawiera i do jakich celów używany jest plik ~**`/.bash_history`**? Jaki efekt wywoła usunięcie tego pliku?
18. Co zawiera plik **`/proc/cpuinfo`**?
19. Jaki efekt wywołuje opcja **`-v`** dodana do poleceń **`cp`**, **`mv`** i **`rm`**?
20. Spróbuj użyć polecenia **`cat`** do wypisania na ekranie zawartości pliku **`/proc/kcore`**. Wyjaśnij uzyskany efekt, wiedząc, że plik ten (a dokładniej rzecz biorąc pseudoplik) odwzorowuje zawartość całej pamięci operacyjnej komputera i tym samym daje dostęp do przestrzeni adresowej wszystkich obecnych w systemie procesów.

2. Prawa dostępu plików i dowiązania

2.1. Prawa dostępu plików

W systemach klasy UNIX dostęp do plików i katalogów (oraz innych obiektów, które obsługiwane są tak, jakby były plikami) kontrolowany jest przez tak zwane **prawa dostępu** (ang. *access rights* ewentualnie *file permissions*), które regulują zasady, na jakich użytkownicy mogą korzystać z tych zasobów. Wyróżnia się trzy podstawowe rodzaje dostępów:

- odczyt – oznaczane jako **r** (od ang. *read*),
- zapis – oznaczane jako **w** (od ang. *write*),
- wykonanie – oznaczane jako **x** (od ang. *execute*).

Każdy z powyższych typów dostępów jest dopuszczany bądź wykluczany niezależnie dla:

- użytkownika (ang. *user*), który jest właścicielem pliku lub katalogu (domyślnie właścicielem jest ten użytkownik, który utworzył dany plik lub katalog),
- użytkowników, którzy należą do tej samej grupy (ang. *group*), do której przynależy plik lub katalog (jest to tak zwana *grupa właścicielska*),
- pozostałych (ang. *other*) użytkowników.

Interpretacja praw dostępu w przypadku zwykłych plików (tzn. plików niebędących katalogami) jest następująca:

– **r** – umożliwia **odczytanie** zawartości pliku, na przykład w celu wyświetlenia jego zawartości w konsoli albo skopiowania pliku pod inną nazwę,
– **w** – umożliwia **zapis** do pliku oraz jego **usunięcie**, uprawnieniem tym jest na przykład regulowana możliwość edytowania zawartości pliku, ale już nie przemianowanie go,
– **x** – umożliwia **uruchomienie** pliku, gdy zawiera on kod wykonywalny, niezależnie od tego, czy zawartość jest kodem binarnym, czy też tekstem skryptu.

Znaczenie praw zmienia się jednak radykalnie, kiedy odnoszą się do katalogu:
– **r** – umożliwia odczytanie metadanych plików znajdujących się w katalogu (co pozwala na przykład skutecznie wykonać polecenie `ls` w tym katalogu),
– **x** – umożliwia uczynienie katalogu katalogiem bieżącym, czyli daje możność wykonania takiego polecenia `cd`, w wyniku którego przemieścimy się do wnętrza takiego katalogu,
– **w** – umożliwia tworzenie, usuwanie i przemianowywanie plików znajdujących się w katalogu, ale tylko wtedy, gdy jednocześnie posiada się do tego katalogu uprawnienie **x**; bez uprawnienia **x** uprawnienie **w** jest pozbawione znaczenia.

Podstawowym sposobem zasięgania informacji o prawach plików i katalogów jest użycie polecenia `ls` z przełącznikiem `-l` – oto przykład wyjścia wyprowadzanego przez to polecenie użytego w katalogu, w którym istnieje już plik *abc.txt*:

```
$ ls -l
drwx------ 15 adam students 4096 lip  6 13:27 ./
drwxr-xr-x 54 adam students 4096 lip  6 11:20 ../
-rwxr--r-x  2 adam students 4096 cze 23 13:32 abc.txt
```

Pierwszy znak maski praw wyświetlanej przez polecenie `ls -l` używany jest do uwidocznienia rodzaju pliku, i tak:
`-` (minus) sygnalizuje brak szczególnych własności prezentowanego pliku, a więc jest to **plik zwykły**;
`d` oznacza, że prezentowany plik to **katalog** (od ang. *directory*).

Informacja o prawach wyświetlana jest na kolejnych dziewięciu znakach według następującego schematu:

user			*group*			*other*		
r	w	x	r	w	x	r	w	x

Wystąpienie na wskazanej pozycji jednego z powyższych znaków oznacza istnienie danego prawa, znak „–” oznacza jego brak.

Zatem dla pliku *abc.txt* określone są następujące prawa:
– właściciel ma komplet praw do swojego pliku (*rwx*),
– dla członków grupy *students* dostępny jest tylko odczyt (*r–*),
– pozostałym użytkownikom zezwala się na odczyt i wykonanie (*r-x*).

Prawami dostępu można także operować z zastosowaniem notacji numerycznej, w której każde z uprawnień traktowane jest jako **bit** (istnienie prawa jest *jedynką*, brak prawa *zerem*), a uzyskaną w ten sposób wartość binarną prezentuje się jako liczbę **ósemkową**. Skoro więc komplet praw opisywany jest dziewięcioma bitami, równoważna postać ósemkowa składać się będzie z trzech cyfr. Każda z nich uwidacznia prawa jednego rodzaju użytkowników (u, g, o) i tworzona jest jako suma arytmetyczna następujących wartości:

- 4 – uprawnienie **r**,
- 2 – uprawnienie **w**,
- 1 – uprawnienie **x**.

Wynika stąd, że prawa pliku *abc.txt* zapisuje się w tej notacji jako **`745`**, gdyż:

- user: **`rwx`** $\Rightarrow 4 + 2 + 1 = 7$,
- group: **`r--`** $\Rightarrow 4 + 0 + 0 = 4$,
- others: **`r-x`** $\Rightarrow 4 + 0 + 1 = 5$.

Należy pamiętać, że wpływ na to, jak i przez kogo może być używany dany plik, ma również to, do kogo plik należy i do jakiej grupy właścicielskiej jest przypisany. System Linux udostępnia zestaw poleceń umożliwiający efektywne manipulowanie każdą z tych danych. Są to:

– **`chmod`** (ang. *change mode*) do zmiany praw dostępu do pliku,
– **`chown`** (ang. *change owner*) do zmiany właściciela pliku (chociaż należy zaznaczyć, że polecenie to pozwala również zmieniać grupę pliku),
– **`chgrp`** (ang. *change group*) do zmiany grupy właścicielskiej pliku.

2.1.1. Polecenie `chmod`

`chmod` ***`[przełączniki...] uprawnienia nazwa_pliku...`***

Polecenie **`chmod`** zmienia prawa dostępu w sposób wskazany pierwszym argumentem wywołania na rzecz plików lub katalogów wskazanych drugim i kolejnymi argumentami wywołania.

Uwaga: co oczywiste, zmiana praw pliku nie zawsze będzie możliwa – na przykład nie można w ten sposób przydzielić sobie uprawnień do cudzego pliku.

Specyfikacja uprawnień, jakie plik ma nabyć po wykonaniu polecenia **`chmod`**, możliwa jest na kilka sposobów, używanych zależnie od okoliczności. Pierwszy z nich posługuje się czytelnym (na pewno czytelniejszym od zapisu ósemkowego)

formalizmem pozwalającym określić nadawane bądź odbierane prawa, a który można zapisać w skrócie jako:

Kto-Jak-Co

Kto wskazuje, jakiej kategorii użytkowników dotyczy operacja:

- **u** – właściciel,
- **g** – grupa,
- **o** – pozostali,
- **a** – wszyscy.

Litery te mogą być łączone w dowolne zestawy, choć nie wszystkie możliwe zestawy mają sens praktyczny. Możliwe jest również użycie pustego zestawu, w takim przypadku uznaje się, że jest on tożsamy z **a**.

Jak opisuje operację, która ma zostać wykonana:

- **+** – dodanie uprawnień do już istniejących;
- **–** – odjęcie uprawnień od już istniejących;
- **=** – przypisanie uprawnień bez względu na to, jakie istniały wcześniej.

Co specyfikuje uprawnienia, które są brane pod uwagę w czasie wykonywania operacji, i używa się tutaj niepustego złożenia znaków **r**, **w** i **x**. Zestaw *kto-jak-co* może wystąpić więcej niż raz, ale w takim przypadku kolejne zestawy muszą być rozdzielone przecinkami, bez wtrąconych białych znaków. Pojawienie się w tym miejscu białego znaku będzie oznaczać, że kolejny ciąg znaków widocznych jest już nazwą pliku, co z pewnością nie będzie zgodne z intencjami wydającego polecenie.

Oto przykładowe zlecenia z wykorzystaniem powyższej składni polecenia **`chmod`**:

- **`chmod u+w plik.txt`**
  dodaje prawo zapisu dla właściciela do pliku *plik.txt*;
- **`chmod go-x skrypt`**
  usuwa prawo wykonywania pliku *skrypt* przez użytkowników z grupy właścicielskiej i innych;
- **`chmod a=r fb`**
  ustawia prawa dostępu do pliku **`fb`** na „tylko do odczytu” dla wszystkich użytkowników;
- **`chmod u+x,g-x wirus1 wirus2`**
  nadaje prawo wykonywania plików *wirus1* i *wirus2* przez właściciela i odbiera to prawo grupie.

Dygresja: warto zauważyć, że prawo wykonania jest tak naprawdę czymś więcej niż tylko zezwoleniem bądź jego brakiem. Można powiedzieć, że nadanie tego prawa de facto *czyni plik wykonywalnym, ponieważ w odróżnieniu od systemów klasy Windows informacja o tym, czy plik jest wykonywalny,* ***nie pochodzi z jego nazwy****. Oznacza to także, że każdy plik (w tym bitmapa) może być z punktu widzenia systemu rozpoznawany jako wykonywalny. Sens takiego działania jest raczej dyskusyjny, ale jest to możliwe.*

Polecenie **chmod** umożliwia także określanie praw dostępu wprost w postaci ósemkowej, jednak w tym przypadku nie ma możliwości modyfikowania pojedynczych uprawnień i zawsze należy podać komplet praw pliku/katalogu. Na przykład:

- **chmod 777 file***
  nadaje plikom, których nazwy zaczynają się od liter *file*, wszystkie możliwe uprawnienia wszystkim kategoriom użytkowników,
- **chmod 742 you**
  ustawia prawa odczytu, zapisu i wykonywania właścicielowi, prawo odczytu użytkownikom z tej samej grupy oraz prawo zapisu innym użytkownikom do pliku *you*.

Poniżej zaprezentowano zapis kilku operacji zmiany praw pliku wraz z uwidocznieniem uzyskanych efektów. Analizę tej sekwencji poleceń pozostawia się czytelnikowi.

wykonane polecenie	stan uprawnień po wykonaniu polecenia
touch plik	**644 -rw-r-r-**
chmod +x plik	**755 -rwxr-xr-x**
chmod g+w,o-x plik	**774 -rwxrwxr-**
chmod 755 plik	**755 -rwxr-xr-x**
chmod 640 plik	**640 -rw-r---**

2.1.2. Polecenie chown

chown *[przełączniki...] właściciel[:grupa] plik...*

Polecenie **chown** umożliwia zmianę właściciela lub grupy pliku.

Uwaga: nie każda zmiana właściciela jest dopuszczalna. Nie można w ten sposób przywłaszczyć sobie pliku należącego do użytkownika root*. Co więcej, nie można również w ten sposób uczynić użytkownika* root *właścicielem swojego pliku. Z powyższych powodów pełnia mocy tego polecenia ujawnia się dopiero wtedy, gdy jest używane przez administratora, któremu zezwala się na dowolne modyfikowanie własności plików.*

2.1.3. Polecenie `chgrp`

`chgrp` ***`[przełączniki...] grupa plik...`***

Zmienia grupę, do której należy wskazany plik lub katalog. Obowiązują te same zastrzeżenia, jak przy poleceniu **`chown`**.

2.2. Użytkownicy i grupy – reprezentacja w systemie

W systemach klasy UNIX informacja o użytkownikach jest tradycyjnie przechowywana w plikach tekstowych o znormalizowanej strukturze, nazwie i lokalizacji. Jeżeli konkretna instalacja tego wymaga, możliwe jest również stosowanie zamiast tego innych rozwiązań, na przykład w postaci systemów katalogowych lub baz danych, jednak nie są to rozwiązania typowe i raczej niespotykane w instalacjach desktopowych.

Pliki te to:

– **`/etc/passwd`** wraz z **`/etc/shadow`**
– **`/etc/groups`**

Pliki te mają charakterystyczną dla systemów uniksowych strukturę wierszową, gdzie każdy z wierszy jest podzielony na pola przy pomocy separatora, którym jest znak **`:`** (dwukropek).

2.2.1. Plik `/etc/passwd`

Plik ten przechowuje informacje o:

– nazwie każdego, znanego systemowi, użytkownika (przy czym warto zaznaczyć, że istnienie pewnego konta użytkownika nie oznacza, że użytkownik ten jest człowiekiem, bowiem konta zakłada się również dla wybranych usług systemu, jeśli ich funkcjonowanie z uprawnieniami użytkownika *root* mogłoby zagrażać integralności instalacji,
– identyfikatorze użytkownika (UID, od ang. *user ID*),
– identyfikatorze grupy macierzystej tego użytkownika (GID, od ang. *group ID*),
– położeniu katalogu domowego użytkownika,
– informacji osobistych (numer telefonu, numer pokoju itd.),
– nazwy programu powłoki, który ma zostać uruchomiony po zalogowaniu użytkownika (podanie w tym miejscu nazwy programu **`/sbin/nologin`** jest najprostszą metodą karnego zablokowania możliwości faktycznego zalogowania się, ponieważ **`nologin`** kończy pracę natychmiast z kodem zakończenia sugerującym wystąpienie błędu; tego samego mechanizmu używa się do uniemożliwienia zalogowania się na konto przypisane do usług systemu.

Historycznie plik ten używany był również do przechowywania informacji o hasłach użytkowników, co jest o tyle intrygujące, że domyślne prawa tego pliku to **644** (właścicielem jest użytkownik *root*). Oznacza to, że zawarte w pliku informacje są dostępne dla wszystkich (*sic!*). Oczywiście w pliku tym przechowywano nie same hasła, a tak zwane *hasze* haseł (aby uczynić historię krótszą, można powiedzieć tylko, że są to wyniki przekształcenia tekstu w tekst przy pomocy pewnej nieróżnowartościowej i nieodwracalnej funkcji; funkcja taka jest też znana publicznie). W momencie logowania system obliczał hasz z wprowadzonego hasła i porównywał go z haszem pamiętanym w **/etc/passwd**. Niestety bardzo szybko okazało się, że wiara w niemożność odtworzenia oryginalnego hasła z hasza jest co najmniej naiwna, bowiem praktyka pokazała, że hasło to jest zbędne – wystarczy odnalezienie tak zwanej *kolizji*, tzn. ciągu znaków, który daje w wyniku taki sam hasz, a to otwiera na oścież podwoje systemu.

Narzucający się niemal natychmiast pomysł, aby po prostu zmienić prawa pliku **/etc/passwd** na przykład na **600** jest spóźniony – zbyt wiele narzędzi i usług zakłada, że plik ten jest publicznie odczytywalny. W tej sytuacji zdecydowano się na inne rozwiązanie: w polu przeznaczonym na hasz hasła wpisuje się znak „x” (który nie ma prawa wystąpić w pojedynkę w prawdziwym haszu hasła), a same hasze przechowuje się w pliku **/etc/shadow**. Plik ten dla odmiany ma prawa **640** i nie da się go odczytać bez posiadania odpowiednio wysokich uprawnień.

Poniżej przytoczono przykładowy obraz zapisu w pliku **/etc/passwd**, dotyczący hipotetycznego użytkownika o nazwie **user**.

```
user:x:1001:1002:Waldemar Kiepski:/home/user:/bin/bash
```

Powyższy zapis odczytuje się następująco:
- w systemie istnieje konto użytkownika o loginie **user**,
- hasz hasła tego użytkownika przechowywany jest jest w pliku **/etc/shadow**,
- UID tego użytkownika to **1001**,
- GID grupy macierzystej tego użytkownika to **1002**,
- użytkownik ów zwie się naprawdę **Waldemar Kiepski**,
- jego katalog domowy to **/home/user**,
- w momencie logowania tego użytkownika należy dla niego uruchomić powłokę **/bin/bash**.

Z kolei wpis do pliku **/etc/shadow** odnoszący się do tego samego użytkownika może wyglądać jak poniżej (dla celów publikacyjnych skrócono znacząco zapis hasza hasła):

```
user:$6$GYVr5axz$BlCIPnhv3jkwJLICmrHALQP/:18762:0:99999:7:::
```

Powyższy wpis interpretuje się następująco:

- wpis dotyczy użytkownika **user**,
- hasz hasła tego użytkownika to **6GYVr5axz$BlCIPnhv3jkwJLICmrHALQP/**,
- ostatnia zmiana hasła tego użytkownika dokonała się w dniu odległym o **18762** doby od 1 stycznia 1970 roku (czyli 15 maja 2021 roku),
- użytkownikowi wolno będzie zmienić hasło po upływie **0** dni (czyli może je zmienić, kiedy chce),
- hasło użytkownika wygaśnie po upływie **99999** dni (czyli praktycznie nigdy),
- użytkownik zostanie uprzedzony o konieczności zmiany hasła na **7** dni przed jego wygaśnięciem,
- pozostałe parametry dotyczące czasu wygaśnięcia konta nie zostały podane, a więc konto będzie aktywne aż do momentu jego usunięcia.

2.2.2. Plik `/etc/group`

Plik ten przeznaczony jest do przechowywania informacji o grupach użytkowników znanych w systemie. Dla każdej grupy pamięta się:

- jej nazwę,
- jej identyfikator (GID, od ang. *group ID*),
- listę nazw użytkowników będących członkami grupy.

Standardowe prawa tego pliku to **644**.

Nie ma potrzeby rozwijania tego wątku, ale warto dodać, że w ogólnym przypadku grupy mogą mieć własnych administratorów oraz własne hasła. Ze względu na niewielkie rozpowszechnienie wykorzystania tego mechanizmu nie będzie on dokładniej omawiany i zostanie ograniczony do wzmianki, że problemy z przechowywaniem haszy haseł były tu identyczne, jak w przypadku pliku */etc/passwd* i identycznie też je rozwiązano – wrażliwe dane przeniesiono do pliku **/etc/gshadow**.

Poniżej przytoczono przykładowy obraz zapisu w pliku **/etc/group** dotyczący grupy o nazwie **power**, zwyczajowo służącej do nadania określonym użytkownikom prawa do posługiwania się narzędziami do zarządzania zasilaniem.

```
power:x:87:user,mickey,ola
```

Powyższy zapis odczytuje się następująco:

- w systemie istnieje grupa o nazwie **power**,
- hasz hasła tej grupy jest przechowywany w pliku **/etc/gshadow**,
- GID tej grupy to **87**,
- do grupy tej należą użytkownicy **user**, **mickey** i **ola**.

2.2.3. Zmiana własnego hasła

Zmianę własnego hasła wykonuje się z użyciem polecenia **`passwd`**. Zmianę hasła użytkownika innego niż uruchamiający polecenie może wykonać tylko użytkownik *root*.

Polecenie uruchamia się bez argumentów, sama zmiana hasła przebiega konwersacyjnie, a przebieg dialogu silnie zależy od stopnia zabezpieczeń wprowadzonych w konkretnej instalacji. Jeden z możliwych scenariuszy zaprezentowano poniżej (znakami X zamaskowano wprowadzane hasła – normalnie są one niewidoczne):

```
user@host $ passwd
Changing password for user.
Current password: XXXXXXXXXXXXXX

You can now choose the new password or passphrase.

A valid password should be a mix of upper and lower case letters, digits, and
other characters.  You can use a password containing at least 7 characters
from all of these classes, or a password containing at least 8 characters
from just 3 of these 4 classes.
An upper case letter that begins the password and a digit that ends it do not
count towards the number of character classes used.

A passphrase should be of at least 3 words, 11 to 72 characters long, and
contain enough different characters.

Alternatively, if no one else can see your terminal now, you can pick this as
your password: "lofty6Trough&Bogus".

Enter new password: XXXXXXXXXXX
Re-type new password: XXXXXXXXXXXX
passwd: password updated successfully
```

2.3. i-węzły i dowiązania

W systemach klasy UNIX metadane plików przechowywane są w strukturach, które nazywa się i-węzłami (ang. *i-node*, od *index-node*). Każdy taki i-węzeł przechowuje między innymi następujące metadane:

- prawa dostępu,
- czas ostatniej zmiany zawartości lub metadanych pliku (ang. *ctime – last change time*) – sytuacja taka może zajść na przykład na skutek zmiany nazwy albo praw pliku, ale także po zapisie do pliku,
- czas ostatniej modyfikacji zawartości pliku (ang. *mtime – last modification time*) – na przykład na skutek dopisania nowych danych na koniec pliku,
- czas ostatniego dostępu do pliku (ang. *atime – last access time*) – za dostęp taki uznaje się również odczyt z pliku,

– rozmiar pliku (w bajtach),
– licznik dowiązań (to zagadnienie zostanie omówione wkrótce),
– inne atrybuty, w tym zależne od konkretnej wersji systemu operacyjnego.

Pomiędzy wymienionymi powyżej trzema czasami zachodzi następująca zależność:

– *atime* może się zmienić jako jedyny spośród tej trójki (na przykład na skutek odczytu z pliku),
– zmiana *ctime* automatycznie pociąga za sobą zmianę *atime* (na przykład po zmianie nazwy pliku),
– zmiana *mtime* wywołuje jednoczesną zmianę *atime* i *mtime* (na przykład po zapisie do pliku).

Warto dodać, że każdy z powyższych czasów reprezentowany jest w systemie plików jako dana całkowita bez znaku, obrazująca liczbę sekund od północy czasu uniwersalnego 1 stycznia 1970 roku, co umownie uznaje się za początek tak zwanej *epoki Uniksa* (ang. *Unix Epoch*).

Należy jednak zwrócić uwagę na fakt, że konsekwentne i pełne aktualizowanie *atime* każdego pliku w systemie może być niezwykle kosztowne, jako że wymaga ogromnej liczby zapisów do wszystkich zaangażowanych i-węzłów. Z tego też powodu konkretna instalacja może minimalizować aktualizację *atime* tylko do wybranych sytuacji, na przykład do operacji zamknięcia pliku otwartego wcześniej do czytania.

Licznik dowiązań określa, **ile razy** dany plik jest dostępny w systemie plików (w tym w różnych katalogach i pod różnymi nazwami). Licznik ten umożliwia realizację tak zwanych *dowiązań do plików* (ang. *links*), które to dowiązania są dodatkowymi nazwami (aliasami) nadanymi plikowi i umożliwiają dostęp do oryginału z poziomu różnych katalogów albo w celu wybrania jednej spośród wielu dostępnych wersji plików.

Istnieją dwa rodzaje dowiązań:

– dowiązania **twarde** (ang. *hard links*), realizowane fizycznie na poziomie organizacji i-węzłów wewnątrz systemu plików w sposób niewidoczny dla użytkownika,
– dowiązania **miękkie**, zwane również **symbolicznymi** (ang. *soft* lub *symbolic links*), realizowane poprzez pliki specjalnego przeznaczenia, reprezentujące dowiązanie w sposób logiczny.

Podstawowa różnica w działaniu obu rodzajów dowiązań polega na tym, że dowiązania twarde mogą funkcjonować tylko w obrębie tego samego systemu plików, który dla potrzeb niniejszych rozważań zostały utożsamione z partycją. Dowiązania symboliczne mogą natomiast przekraczać granice partycji i odnosić się także do plików umieszczonych na innych urządzeniach fizycznych.

Informacja o dowiązaniu dostępna jest dzięki omówionemu już poleceniu **ls -l**. Wszystkie dowiązania można przetwarzać dokładnie tak samo jak odpowiadające im pliki i katalogi, w szczególności mogą być także usunięte poleceniem **rm** (warto dostrzec, że w ten sam sposób, a nie poleceniem **rmdir**, usuwa się również dowiązanie do katalogu).

Fakt, że pewien plik jest dowiązaniem symbolicznym, najłatwiej zarejestrować poprzez analizę wyjścia z polecenia **ls** z przełącznikiem **-l**, na przykład:

```
$ ln -s plik symboliczne
$ ls -l
lrwxrwxrwx 1 user   user   4 01-01 12:00 symboliczne -> plik
-rw-r--r-- 1 user   user   0 01-01 11:00 plik
```

Warto zauważyć, że:

- dowiązanie symboliczne o nazwie **symboliczne** ma literę **l** (od ang. *link*) na pierwszym znaku maski praw, i to niezależnie od tego, czy prowadzi do pliku, czy do katalogu,
- dowiązanie takie ma komplet możliwych praw, co jednak nie ma znaczenia w przypadku korzystania z niego, bowiem dowiązanie symboliczne ma faktycznie dokładnie takie same prawa, jak plik/katalog, do którego prowadzi,
- za nazwą dowiązania symbolicznego, a po strzałce, prezentowana jest nazwa pliku/katalogu, do którego dowiązanie prowadzi,
- rozmiar dowiązania symbolicznego jest równy długości nazwy pliku/katalogu, do którego dowiązanie prowadzi (z ewentualną poprzedzającą ścieżką) i nie ma nic wspólnego z faktycznym rozmiarem dowiązanego obiektu,
- należy zaznaczyć, że dowiązanie symboliczne może mieć inny znacznik czasu niż plik, do którego prowadzi.

Dla odmiany dowiązanie twarde jest nieodróżnialne od pliku/katalogu, co potwierdza, że ten twór jest w istocie aliasem nazwy pliku reprezentowanym na poziomie systemu plików.

```
$ ln plik twarde
$ ls -l
-rw-r--r-- 1 user   user   0 01-01 11:00 plik
-rw-r--r-- 1 user   user   0 01-01 11:00 twarde
```

Jak widać, dowiązanie twarde ma taki sam znacznik czasu co plik, do którego prowadzi, a ponadto:

- dowiązanie twarde o nazwie **twarde** nie ma żadnego specjalnego wyróżnika na pierwszym znaku maski praw (dowiązanie takie może prowadzić wyłącznie do pliku);

– dowiązanie takie ma dokładnie takie same prawa jak plik docelowy;
– rozmiar dowiązania twardego jest równy rozmiarowi pliku docelowego.

Tworzenie dowiązań jest możliwe dzięki poleceniu **ln** (ang. *link*), którego najczęściej stosowana postać prezentuje się jak poniżej:

ln *[przełącznik...] plik_faktyczny nazwa_dowiązania*

Argument *plik_faktyczny* musi wskazywać na istniejący plik (lub katalog w przypadku dowiązań symbolicznych), do którego tworzone jest dowiązanie, a drugim argumentem jest nazwa, pod którą zostanie utworzone nowe dowiązanie. Utworzenie dowiązania symbolicznego wymaga zastosowania przełącznika **-s**.

Oto kilka przykładów wywołania zlecenia utworzenia dowiązań:

ln ./abc/plik.txt plik1.txt
tworzy w katalogu bieżącym dowiązanie **twarde** do pliku *plik.txt* w katalogu *./abc* pod nazwą *plik1.txt*,

ln -s abc/plik.txt /plik1.txt
tworzy w katalogu domowym użytkownika dowiązanie **symboliczne** do pliku *plik.txt* w katalogu *abc* pod nazwą *plik1.txt*.

Inne rzadziej stosowane formy polecenia **ln** to:

ln *[przełącznik...]* plik_faktyczny
tworzy w katalogu bieżącym dowiązanie o nazwie takiej, jaką nosi *plik_faktyczny*; z oczywistych przyczyn *plik_faktyczny* musi znajdować się poza katalogiem bieżącym,

ln *[przełącznik...] plik_faktyczny... katalog*
we wskazanym *katalogu* dla każdego z *plików_faktycznych* tworzy dowiązanie o takiej samej nazwie jak *plik_faktyczny*,

ln *[przełącznik...] -t katalog plik_faktyczny...*
działanie jak w poprzednim przypadku.

*Dygresja: Ciekawą cechą dowiązań jest to, że jeśli pewien plik używany jest poprzez któreś ze swoich dowiązań, to traktowany jest tak, jakby nosił nazwę dowiązania, a nie nazwę pliku faktycznego. W szczególności w programach napisanych w języku C parametr **argv[0]** będzie zawierał nazwę użytą, aby uruchomić dany program, a nie nazwę pliku, w którym znajduje się kod wykonywalny. Oznacza to, że ten sam kod może wypełniać różne funkcje w zależności od nazwy dowiązania, jakie spowodowało jego uruchomienie. Ta cecha dowiązań wykorzystywana jest na przykład przez program o charakterystycznej nazwie **busybox**, dołączany standardowo do praktycznie wszystkich dystrybucji Linuksa, który potrafi zachowywać się jak polecenia **cp**, **mv**, **rm** i tym podobne, o ile tylko zostanie uruchomiony taką właśnie nazwą. Takie rozwiązanie pozwala oszczędzić miejsce na nośniku w przypadku zastosowania go w rozwiązaniach typu live, dystrybuowanych na mediach o ograniczonej pojemności (na przykład w plikach ISO przeznaczonych do*

bezpośredniego zabootowania systemu) bądź też w systemach wbudowanych, gdzie każdy bajt nośnika jest na wagę złota.

2.4. Polecenie `date`

Polecenie **date** pozwala każdemu użytkownikowi odczytać bieżący czas systemowy, a użytkownikowi o odpowiednio wysokich uprawnieniach umożliwia również tenże czas ustawić. Z tego też powodu omówione zostaną wyłącznie te formy polecenia **date**, które mogą być użyte przez wszystkich:

date ***[przełączniki...] [+FORMAT]***

- użyte bez argumentów wyświetli bieżący czas systemowy w formacie zgodnym z ustawioną „narodowością" systemu i z użyciem nazw miesięcy oraz dni tygodnia w stosownym języku naturalnym,
- przełącznik **-R** spowoduje wyświetlenie czasu w formacie RFC 2822 (używanym na przykład w poczcie elektronicznej),
- przełącznik **-u** spowoduje wyświetlenie czasu przekształconego do czasu uniwersalnego (UTC),
- przełącznik **-r** z parametrem będącym nazwą pliku poda czas ostatniej modyfikacji tego pliku,
- parametr **FORMAT**, jeśli zostanie użyty, będzie ciągiem znaków formatujących podobnych w swojej postaci do formaterów używanych przez funkcję **printf()** z języka C.

Poniżej zaprezentowano kilka najważniejszych specyfikatorów używanych przez polecenie **date**:

%a lokalny skrót nazwy dnia tygodnia (na przykład **nie**);
%A lokalna pełna nazwa dnia tygodnia (na przykład **niedziela**);
%b lokalny skrót nazwy miesiąca (na przykład **sty**);
%B lokalna pełna nazwa miesiąca (na przykład **styczeń**);
%c lokalna data i czas (na przykład **czw mar 3 23:05:25 2005**);
%C stulecie, jak **%Y**, tylko z obciętymi ostatnimi dwiema cyframi;
%d dzień miesiąca (na przykład **01**);
%D data, to samo co **%m/%d/%y**;
%e dzień miesiąca uzupełniony spacjami, jak **%_d**;
%F pełna data; to samo co **%Y-%m-%d**;
%g ostatnie dwie cyfry roku numeru tygodnia ISO (patrz **%G**);
%H godzina (**00..23**);
%I godzina (**01..12**);
%j dzień roku (**001..366**);
%k godzina, ewentualnie uzupełniona spacją z lewej (**0..23**); to samo co **%_H**;
%l godzina uzupełniona spacją (**_1..12**); to samo co **%_I**;

%m miesiąc (**01..12**);
%M minuta (**00..59**);
%n znak nowego wiersza;
%N nanosekundy (**000000000..999999999**);
%p lokalny odpowiednik **AM** lub **PM**; w wielu językach (w tym w polskim) pusty;
%P jak **%p**, lecz małymi literami;
%r czas w formacie 12-godzinnym (na przykład **11:11:04 PM**);
%R godzina i minuty w formacie 24-godzinnym; to samo co **%H:%M**;
%s liczba sekund od północy 1 stycznia 1970 UTC;
%S sekunda (**00..60**);
%t tabulator;
%T czas; to samo co **%H:%M:%S**;
%u dzień tygodnia (**1..7**); 1 to poniedziałek;
%U numer tygodnia w roku, przy założeniu, że tydzień zaczyna się od niedzieli (**00..53**);
%V numer tygodnia według normy ISO 8601:2004, przy założeniu, że tydzień zaczyna się od poniedziałku (**01..53**); ze względu na powszechność stosowania tej konwencji (jest ona honorowana także w Polsce), poniżej zaprezentowano ją w pełnym brzmieniu:
  – tydzień trwa 7 dni;
  – pierwszym dniem tygodnia jest poniedziałek;
  – pierwszym tygodniem danego roku jest tydzień zawierający pierwszy czwartek tegoż roku.
%w dzień tygodnia (**0..6**); 0 oznacza niedzielę;
%W numer tygodnia w roku, przy założeniu, że tydzień zaczyna się od poniedziałku (**00..53**);
%x lokalna reprezentacja daty (na przykład **19.03.2012**);
%X lokalna reprezentacja czasu (na przykład **23:13:48**);
%y dwie ostatnie cyfry roku (**00..99**);
%Y rok.

Domyślnie **date** dopełnia pola numeryczne zerami. Następujące opcjonalne modyfikatory mogą być umieszczone po **%**:

- (łącznik) bez wypełniania pola;
_ (podkreślenie) wypełnianie spacjami;
0 (zero) wypełnianie zerami.

Ponadto można również wpływać na rejestr liter używanych do prezentowania tekstu:

^ używaj wielkich liter;
zamieniaj wielkość liter.

2.5. Inne przydatne polecenia

2.5.1. Polecenie `stat`

Polecenie **`stat`** służy do prezentacji kompletu metadanych pliku/katalogu i w najprostszym wariancie jest uruchamiane w sposób następujący:

```
stat plik...
```

Poniżej przedstawiono postać wyjścia tego polecenia w przypadku, gdy jego argumentem jest zwykły plik:

```
$ stat plik
Plik: plik
  rozmiar: 905688     bloków: 1776       bloki I/O: 4096   plik zwykły
Urządzenie: 10308h/66312d inody: 8085747     dowiązań: 1
Dostęp: (0755/-rwxr-xr-x)  Uid: ( 1000/  user)   Gid: ( 1000/  user)
Dostęp:      2022-09-21 14:18:55.676384812 +0200
Modyfikacja: 2022-09-21 14:18:55.677384811 +0200
Zmiana:      2022-09-21 14:18:55.677384811 +0200
Utworzenie:  2022-09-21 14:18:55.676384812 +0200
```

Jak widać, zestawienie prezentuje następujące metadane pliku:

- nazwę pliku (**`plik`**),
- rozmiar pliku w bajtach (**`905688`**),
- liczbę bloków alokacji przydzielonych do pliku (**`1776`**),
- rozmiar bloku alokacji w bajtach (**`4096`**),
- identyfikator urządzenia, na którym rezyduje plik, szesnastkowo (**`10308h`**) i dziesiętnie (**`66312d`**); warto nadmienić, bez głębszej interpretacji znaczenia tej danej, że jeśli dwa pliki mają ten sam numer urządzenia, to rezydują na tej samej partycji,
- numer pierwszego i-węzła pliku (**`8085747`**),
- liczbę dowiązań twardych prowadzących do pliku; paradoksalnie, wartość **`1`** oznacza, że plik nie ma dodatkowych dowiązań,
- prawa pliku w postaci ósemkowej (**`0755`**) i znakowej (**`-rwxr-xr-x`**),
- UID (**`1000`**) i nazwę (**`user`**) właściciela pliku,
- GID (**`1000`**) i nazwę (**`user`**) grupy właścicielskiej pliku,
- cztery znaczniki czasu informujące kolejno o ostatnim dostępie do pliku, ostatniej modyfikacji zawartości, zmianie atrybutów i utworzeniu.

Postać informacji prezentowanej przez **`stat`** może zostać sformatowana w inny sposób, jeśli użytkownik wykorzysta przełącznik **`-c`** lub jego długą formę **`--format`**, a następnie poda ciąg znaków formatujących wybranych z poniższego zestawu (zaprezentowano tylko te warianty, które mają odbicie w standardowej postaci wyjścia polecenia **`stat`**):

`%a` prawa pliku w postaci ósemkowej,

%A prawa pliku w postaci znakowej,
%b liczba przydzielonych bloków,
%B rozmiar bloku alokacji w bajtach,
%d numer urządzenia dziesiętnie,
%D numer urządzenia szesnastkowo,
%F rodzaj pliku w postaci tekstowej (na przykład **plik zwykły**),
%g GID grupy właścicielskiej,
%G nazwa grupy właścicielskiej,
%h liczba twardych dowiązań prowadzących do pliku,
%i numer pierwszego i-węzła pliku,
%n nazwa pliku,
%s rozmiar pliku w bajtach,
%u UID właściciela,
%U nazwa właściciela,
%w czas utworzenia pliku (postać czytelna dla ludzi),
%x czas ostatniego dostępu do pliku (postać czytelna dla ludzi),
%y czas ostatniej modyfikacji zawartości pliku (postać czytelna dla ludzi),
%z czas ostatniej modyfikacji atrybutów pliku (postać czytelna dla ludzi).

Poniżej zaprezentowano wzór użycia sekwencji formatującej zastosowanej do tego samego pliku, co w poprzednim przykładzie:

```
$ stat -c '%n: %s bajtów, %b bloków' plik
plik: 905688 bajtów, 1776 bloków
```

2.5.2. Polecenie `sync`

Polecenie **sync** użyte bez argumentów wymusza opróżnienie wszystkich buforów zapisu i przeniesienie ich zawartości do systemów plików oraz zaktualizowanie informacji pamiętanych w i-węzłach. Używane jest, gdy użytkownik chce mieć pewność, że wszystkie zainicjowane wcześniej operacje zapisu (na przykład szczególnie kierowane na powolne urządzenia wymienne takie jak pendrajwy albo karty SD) zostały faktycznie zakończone.

2.5.3. Polecenie `df`

df ***[przełącznik...] [plik...]***

Powoduje wyświetlenie informacji o dostępnej wolnej przestrzeni w systemie plików. Jeśli nie podano argumentu/argumentów *plik...*, prezentowana jest informacja o wszystkich systemach plików znanych w danym momencie systemowi operacyjnemu. Jeśli argument/argumenty podano, prezentowana jest tylko infor-

macja o systemie bądź systemach plików, na których rezydują pliki o podanych nazwach.

Na sposób prezentowania tej informacji mają wpływ następujące przełączniki:

`--total`
wyświetla podsumowanie całości,

`-h`
wyświetla rozmiary w formacie czytelnym dla ludzi,

`-H`
podobnie, ale używa potęg 1000 zamiast 1024,

`--no-sync`
nie wywołuje **`sync`** przed pozyskaniem informacji (zachowanie domyślne), co powoduje, że prezentowane informacje mogą odbiegać od stanu faktycznego, ale ich uzyskanie przebiega natychmiast (w pewnych przypadkach i przy korzystaniu z powolnych urządzeń pełne wykonanie synchronizacji może trwać nawet kilkadziesiąt sekund),

`--sync`
wywołuje **`sync`** przed pozyskaniem informacji o wolnym nośniku.

2.5.4. Polecenie `du`

`du` ***`[przełącznik...] [plik...]`***

W pewnym sensie jest to dopełnienie polecenia **`df`**, ponieważ powoduje wyświetlenie informacji o przestrzeni **zajętej** przez pliki lub katalogi. W obliczeniach uwzględniany jest nie tylko rozmiar pliku, ale również fakt, że plik może zajmować na nośniku więcej bądź mniej miejsca, niż wymaga tego jego zawartość. Jest to konsekwencja faktu, że nośnik dla pliku przydzielany jest kwantami o stałym rozmiarze i ostatni z przydzielonych kwantów nie musi być zajęty w całości przez dane. Inną okolicznością, która musi być tu wzięta pod uwagę, jest fakt istnienia tak zwanych *plików rzadkich* (ang. *sparse files*), dopuszczalnych w wielu współczesnych systemach plików. Plik rzadki to plik, wypełniony danymi tylko częściowo, przy czym puste (niewypełnione danymi) partie pliku zastępuje się krótkimi metadanymi, dzięki czemu taki plik zajmuje znacznie mniej miejsca na nośniku.

Wynika z tego, że w pełni możliwe (choć nie równie prawdopodobne) są wszystkie trzy poniższe sytuacje:

– zajętość nośnika jest równa rozmiarowi pliku,
– zajętość nośnika jest większa od rozmiaru pliku (ostatni kwant przydziału nie został zapełniony),
– zajętość nośnika jest mniejsza od rozmiaru w pliku (plik jest rzadki).

Warto zwrócić uwagę na fakt, że z punktu widzenia polecenia **du** rozmiar pliku wykazywany na przykład przez polecenia **ls** albo **stat** jest tak zwanym *rozmiarem pozornym* (ang. *apparent size*).

Na sposób, w jaki **du** prezentuje informacje, mają wpływ następujące przełączniki:

-a
wypisuje podsumowania dla wszystkich plików, nie tylko katalogów,
--apparent-size
wyświetla rozmiar pozorny zamiast faktycznego użycia dysku,
-Bs
przelicza rozmiary według jednostki *s*; dopuszczalne postaci *s* to:
  – liczba całkowita,
  – liczba całkowita z przyrostkiem **K** (*kibi*), **M** (*mebi*), **G** (*gibi*), **T** (*tebi*), **P** (*pebi*), **E** (*exbi*), **Z** (*zebi*), **Y** (*yobi*) czyli z użyciem jednostek wynikających z potęg liczby 1024 albo **KB**, **MB** itd., czyli przy pomocy jednostek branych z potęgi liczby 1000,
  – sam przyrostek.
-b
równoważne **--apparent-size -B1**,
-c
wyświetla podsumowanie całości,
-D
rozwija tylko dowiązania symboliczne podane jako argumenty,
-h
wyświetla rozmiary w formacie czytelnym dla ludzi (na przykład **1 KiB**, **234 MiB**, **2 GiB**),
--si
jak **-h**, lecz używa potęg 1000 zamiast 1024,
-k
jak **-B1K**,
-m
jak **-B1M**,
-L

rozwija wszystkie dowiązania symboliczne,

-P

nie podąża za żadnymi dowiązaniami symbolicznymi (domyślnie),

-S

nie uwzględnia rozmiarów podkatalogów,

-s

wyświetla tylko podsumowanie dla każdego podanego argumentu,

--exclude=*pattern*

pomija pliki pasujące do wzorca podanego w *pattern*.

Poniżej zaprezentowano wszystkie trzy możliwości wzajemnej relacji rozmiaru faktycznego i pozornego pewnego pliku. Przyjęto założenie, że poniższy eksperyment przeprowadzany jest na systemie plików, w którym rozmiar alokacji wynosi 4096 bajtów (4 KiB).

W pierwszym kroku zostanie utworzony plik o nazwie **plik**, którego faktyczna zawartość będzie miała rozmiar 16 bajtów. Czynność ta zostanie wykonana z wykorzystaniem polecenia **echo** i mechanizmu przekierowania, omówionych dokładniej w kolejnych rozdziałach.

```
$ echo -n '0123456789ABCDEF' > plik
$ ls -l plik
-rwxr-xr-x 1 user user 16 01-01 12:00 plik
$ du -h plik
4,0K plik
$ u --apparent-size -h plik
16 plik
```

Uzyskany wynik ilustruje sytuację, w której rozmiar danych w pliku (rozmiar pozorny) równy 16 bajtom jest mniejszy od zajętości nośnika (4 KiB).

W drugim kroku plik zostanie wypełniony danymi o rozmiarze równym rozmiarowi bloku alokacji, do czego zostanie użyte zaawansowane narzędzie o nazwie **dd**. Nie wnikając głębiej w możliwości tego niezwykle uniwersalnego programu, należy wyjaśnić, że zostanie on wykorzystany po to, aby z pliku **/dev/zero** – który jest niewyczerpanym źródłem bajtów o wartości zero – odczytać dane o rozmiarze 4096 bajtów i zapisać je w pliku **plik**.

```
$ dd if=/dev/zero of=plik bs=1 count=4096
4096+0 przeczytanych rekordów
4096+0 zapisanych rekordów
skopiowane 4096 bajtów (4,1 kB, 4,0 KiB), 0,0061595 s, 665 kB/s
$ ls -l plik
-rwxr-xr-x 1 user user 4096 01-01 12:00 plik
$ du -h plik
4,0K plik
$ du --apparent-size -h plik
4,0K plik
```

W tym przypadku rozmiar danych w pliku (rozmiar pozorny) równy 4096 bajtom jest dokładnie równy zajętości nośnika (4 KiB).

Do wywołania trzeciej możliwości zostanie wykorzystane polecenie **`truncate`**, które zdolne jest zmienić rozmiar danych w pliku bez zmiany liczby przydzielonych do niego bloków. Mimo zwodniczej nazwy polecenie potrafi nie tylko plik obciąć, ale również rozszerzyć. W tym konkretnym przypadku zostanie wykorzystane do ustalenia rozmiaru przechowywanych w pliku danych na 8192 bajty. Tym samym plik **`plik`** stanie się plikiem rzadkim.

```
$ truncate --size=8192 plik
$ ls -l plik
-rwxr-xr-x 1 user user 8192 01-01 12:00 plik
$ du -h plik
4,0K plik
$ du --apparent-size -h plik
8,0K plik
```

Uzyskany tutaj wynik ilustruje paradoksalną na pierwszy rzut oka sytuację, w której rozmiar danych w pliku (rozmiar pozorny) równy 8192 bajtom jest dwukrotnie większy od zajętości nośnika (nadal równej 4 KiB).

2.5.5. Polecenie `file`

`file` ***`[przełącznik...] plik...`***

To pożyteczne narzędzie istnieje w systemach klasy UNIX głównie z powodów doktrynalnych. Otóż filozofia tego systemu nigdy nie przewidywała, aby w nazwie pliku miała się znajdować jakakolwiek informacja o tym, co plik zawiera (co od zarania dziejów było standardem w systemach klasy Windows). Rozwój Internetu zmienił to nastawienie i w chwili obecnej używanie identyfikujących plik przyrostków jest nie tylko na porządku dziennym, ale ponadto zostało ujęte w normach międzynarodowych, jednak zawartość pewnych plików nadal może być zagadką.

Na przykład pliki, które oznaczone są jako **x** (wykonywalne), mogą mieć bardzo różną zawartość: być programami skompilowanymi, w slangu określane mianem *binarek*, ale też zwykłymi plikami tekstowymi zawierającymi skrypty w przeróżnych językach interpretowanych (Bash, awk, tcl, Python, Perl itd.). Można oczywiście próbować oglądać takie pliki od środka i wydawać osąd na podstawie wzrokowej analizy ich zawartości, ale nie zawsze jest to wygodne. Ponadto może się również zdarzyć, że na skutek nieszczęśliwego wypadku plik może stracić oryginalną nazwę i warto wówczas posiadać narzędzie, które samo przeanalizuje jego zawartość i postawi hipotezę odnośnie do tego, czym ten plik jest w istocie.

Polecenie **file** wykonuje tę czynność na podstawie analizy zawartości specjalnej bazy danych nazywanej *magic numbers*. Jest to zbiór prostych heurystyk pozwalających sprawdzić, czy dany plik ma (lub nie) pewne ewidentne cechy (na przykład zbiór użytych znaków, obecność charakterystycznych sygnatur i tym podobne). Same heurystyki zapisywane są w plikach tekstowych umieszczanych zwyczajowo w katalogu **/usr/share/misc/magic**, a ich lektura dostarcza bardzo ciekawych informacji na temat tego, jak dalece wiarygodne (bądź niewiarygodne) są diagnozy stawiane przez to narzędzie.

W najprostszym przypadku wystarczy wskazanie nazwy pliku, który interesuje użytkownika, a **file** odpowie stosowną informacją, na przykład:

```
$ file win/explorer.exe
win/explorer.exe: PE32+ executable (GUI) x86-64, for MS Windows
```

co w tym konkretnym wypadku oznacza, że diagnozowany plik jest według heurystyk narzędzia **file** 64-bitowym plikiem wykonywalnym pochodzącym z systemu operacyjnego MS Windows, a przeznaczonym do uruchomienia na platformie Intel x86.

2.5.6. Polecenie `sleep`

sleep *liczba[przyrostek]*

Polecenie **sleep** wywołuje efekt bezczynności trwający przez czas określany argumentem. Przydatne bywa głównie w skryptach, ale może być użyteczne także w pracy konsolowej na przykład w celu wykonania kolejnego polecenia dopiero po upływie pewnego czasu. Dopuszczalne przyrostki to:

- **s** sekundy (domyślne),
- **m** minuty,
- **h** godziny,
- **d** dni.

Warto pamiętać, że wykonanie polecenia **sleep** może zostać w dowolnym momencie przerwane przez kombinację klawiszy *Ctrl-C*.

2.5.7. Polecenie `diff`

`diff` *[przełącznik...] plik1 plik2*

Polecenie **`diff`** służy do porównania zawartości wskazanych plików – w najprostszym przypadku porównuje zawartość dwóch plików, *plik1* i *plik2*. Jeśli *plik1* jest katalogiem, a *plik2* nie, **`diff`** porównuje ten plik z katalogu *plik1*, którego nazwa jest taka sama jak *plik2* i odwrotnie. Jeśli *plik1* i *plik2* są katalogami, **`diff`** porównuje pliki o zgodnych nazwach istniejące w obu katalogach, w kolejności alfabetycznej. To porównanie nie jest ponawiane w podkatalogach, chyba że podano opcję **`-r`**. Jeśli porównywane pliki się nie różnią, **`diff`** nie odzywa się ani słowem.

Informacje wyprowadzane przez **`diff`** mogą być bezpośrednio użyte przez polecenie **`patch`**, co umożliwia oszczędne dystrybuowanie przyrostowych poprawek zamiast kompletnych, obszernych plików źródłowych. Mechanizm ten wykorzystuje się na przykład przy publikowaniu nieznacznych poprawek jądra Linuksa. Zapoznanie się z możliwościami narzędzia **`patch`** pozostawia się czytelnikowi.

Należy podkreślić, że **`diff`** został stworzony do wyszukiwania różnic w plikach tekstowych, a postać dostarczanych przez niego informacji ułatwia na przykład identyfikowanie miejsc, w których zmodyfikowano kod źródłowy. Z tego też powodu **`diff`** nie nadaje się do porównywania plików binarnych (na przykład plików graficznych). Do tego celu służy polecenie **`cmp`**.

Najczęściej używane przełączniki **`diff`** to:

- **`-i`** ignorowanie zmian w wielkości liter, wielkie i małe litery są uznawane za równe;
- **`-w`** ignorowanie wszystkich odstępów przy porównywaniu plików;
- **`-b`** ignorowanie zmian w ilości odstępów;
- **`-B`** ignorowanie zmian, które jedynie dodają lub usuwają puste linie;
- **`-q`** poinformowanie jedynie o tym, czy pliki się różnią, bez podawania szczegółów na temat różnic;
- **`-r`** rekurencyjne porównanie wszystkich podkatalogów, jeśli porównywane są katalogi;
- **`-N`** jeśli podczas porównywania katalogów plik istnieje jedynie w jednym z katalogów, będzie traktowany tak, jakby był obecny w drugim katalogu, ale jako plik pusty;
- **`-P`** jeśli podczas porównywania katalogów plik istnieje jedynie w drugim katalogu, będzie traktowany tak, jakby był obecny w pierwszym katalogu, ale jako plik pusty;

-s poinformowanie, jeśli oba pliki są identyczne (normalnie w takiej sytuacji **diff** nie odzywa się ani słowem);

-x w ignorowanie, podczas porównywania katalogów, tych plików i podkatalogów, których nazwy pasują do wzorca *w*.

Informacje wyprowadzane z polecenia **diff** prezentowane są w dość pokrętny sposób, dlatego zostaną zilustrowane prostymi przykładami, przy czym nie sposób nie zauważyć, że bardzo wskazane jest, aby czytelnik kontynuował te eksperymenty na bardziej złożonych przypadkach.

W dalszej części rozważań przyjęto założenie, że pierwszy z porównywanych plików będzie nazywany plikiem *lewym*, a drugi plikiem *prawym*. Terminologia ta pozwala łatwo zidentyfikować znaczenie zwrotów strzałek, jakie pojawiają się na wyjściu polecenia **diff**.

W tym miejscu warto zauważyć, że informacje pojawiające się na wyjściu polecenia **diff** są nie tyle opisem wykrytych różnic, ile specyficzną receptą na to, co należy zrobić, aby oba porównywane pliki stały się identyczne. Sztywna i sformalizowana postać tej recepty pozwala na, w miarę łatwe, zaimplementowanie automatu, który takie ujednolicenie może przeprowadzać samodzielnie, z równą łatwością doprowadzając plik prawy do zgodności z plikiem lewym albo odwrotnie.

W ogólnym przypadku, jeśli wykryto różnice w porównywanych plikach, wyjście z polecenia **diff** zawiera jedno- lub wielokrotne wystąpienia bloku tekstu o następującej strukturze:

```
OPIS-RÓŻNICY
< LINIA-PLIKU-LEWEGO
< LINIA-PLIKU-LEWEGO...
---
> LINIA-PLIKU-PRAWEGO
> LINIA-PLIKU-PRAWEGO...
```

Element *OPIS-RÓŻNICY* może mieć jedną z trzech możliwych postaci:

LaP (gdzie **a** pochodzi od angielskiego słowa *addition* – dodanie) i ma następujące znaczenie:
*dodaj linie z przedziału opisanego przez **P** z pliku prawego za linią o numerze **L** w pliku lewym*; na przykład **4a8,11** oznacza, że w celu ujednolicenia obu plików należy za czwartą linią w pliku lewym dodać cztery linie (od ósmej do jedenastej) z pliku prawego,

LcP (gdzie **c** pochodzi od angielskiego słowa *change* – zmiana) i ma następujące znaczenie:
*wymień linie z przedziału opisanego przez **L** w pliku lewym na linie z prze-*

*działu **R** z pliku prawego*; na przykład **2,3c6,8** oznacza, że w celu ujednolicenia obu plików należy dwie linie w pliku lewym (drugą i trzecią) wymienić na trzy linie z pliku prawego (szóstą, siódmą i ósmą),

LdP (gdzie **d** pochodzi od angielskiego słowa *deletion* – usunięcie) i ma następujące znaczenie:
*usuń z pliku lewego wszystkie linie z przedziału opisanego przez **L**, podczas gdy **R** opisuje miejsce w pliku prawym, gdzie linie te mogłyby się pojawić w celu uzyskania efektu identyczności*; na przykład **2,3d4** oznacza, że w celu ujednolicenia obu plików należy albo usunąć dwie linie w pliku lewym (drugą i trzecią), albo dodać je za linią czwartą w pliku prawym (oczywiście mimo faktu, że obie alternatywne operacje doprowadzają do identyczności plików, efekt będzie diametralnie inny).

Poniżej prezentowane są wszystkie linie z obu plików, których dotyczy wcześniejszy opis, przy czym zwrot grotów strzałek wskazuje pochodzenie linii, a wiersz złożony z trzech łączników ułatwia odróżnienie tekstu pochodzącego z różnych plików.

2.5.8. Znacznik a (dodanie)

W tym przypadku porównaniu podlegać będą pliki o nazwach *lewy* oraz *prawy* i o następującej zawartości:
lewy:

```
111
333
```

prawy:

```
111
222
333
```

Uruchomienie polecenia **diff** spowoduje tu następujący skutek:

```
$ diff lewy prawy
1a2
> 222
```

Uzyskany wynik ma następujące znaczenie:

– różnica dotyczy linii, która jest linią numer 2 w pliku prawym, a którą należy **dodać** za linią numer 1 pliku lewego (*1a2*);

– poniżej zacytowano linię, która występuje w pliku prawym, a jest nieobecna w pliku lewym;
– warto tu zauważyć, że znacznik **a** jest *de facto* złożeniem diagnoz podawanych przez znaczniki **d** i **c**, pozwala jednak na znaczne skrócenie rozmiaru prezentowanych wyników.

2.5.9. Znacznik c (wymiana)

W tym przykładzie użyto plików o następującej zawartości:
lewy:

```
11
```

prawy:

```
111
```

Uruchomienie polecenia **diff** spowoduje następujący rezultat:

```
$ diff lewy prawy
1c1
< 11
---
> 111
```

Otrzymany wynik należy zinterpretować następująco:
– wykryto różnicę położoną w linii 1 pliku lewego odpowiadającą linii 1 pliku prawego (*1c1*);
– poniżej zacytowano obie różniące się linie, wskazując, z jakich plików pochodzą oraz rozdzielając je znacznikiem **---**, co stanowi sugestię, że wymiana dowolnej z prezentowanych linii na drugą z nich sprawi, że pliki będą identyczne.

2.5.10. Znacznik d (usunięcie)

Założono, że oba pliki mają teraz poniższą zawartość:
lewy:

```
111
222
```

prawy:

```
111
```

Odpowiedź polecenia **diff** prezentuje się jak poniżej:

```
$ diff lewy prawy
2d1
< 222
```

Interpretacja takiego wyniku jest następująca:

- różnica dotyczy linii, która jest linią numer 2 w pliku lewym i która powinna się znaleźć **za** linią numer 1 w pliku prawym, aby pliki były identyczne; alternatywnie linię tę należy usunąć z pliku lewego (*2d1*);
- poniżej zacytowano linię, której usunięcie, z punktu widzenia polecenia **diff**, sprawi, że pliki nie będą się różnić.

2.5.11. Polecenie cmp

cmp *[przełącznik...] plik1 plik2*

cmp porównuje zawartość dwóch plików dowolnego typu i wypisuje na konsolę wynik tego porównania (albo nie). Domyślnie **cmp** zachowuje pełne godności milczenie, gdy pliki są identyczne, a w przeciwnym przypadku **cmp** oznajmia numer bajtu i linii, w których wykryto pierwszą różnicę. *Uwaga: bajty i linie są numerowane od **jeden**.*

Najprzydatniejsze przełączniki to:

-c wypisuje różniące się znaki;

-i *n* ignoruje wszelkie różnice w *n* początkowych bajtach każdego z plików; traktuje pliki zawierające mniej niż *n* bajtów tak, jakby były puste;

-l dla każdej różnicy wypisuje numer bajtu (dziesiętnie) i wartości różniących się bajtów (ósemkowo).

Do demonstracji działania polecenia **cmp** zostały wykorzystane dwa spreparowane pliki różniące się zawartością dokładnie dwóch bajtów. Utworzono je sposobem zastosowanym wcześniej przy prezentacji polecenia **du**.

```
$ echo -n '0123456789' > plik1
$ echo -n '012x4567x9' > plik2
```

Powyższy zestaw danych testowych posłużył do przedstawienia czterech poniższych wariantów użycia polecenia **cmp** oraz uzyskanych rezultatów:

```
$ cmp plik1 plik2
plik1 plik2 różnią się: bajt 4, linia 1
$ cmp -l plik1 plik2
 4  63 170
 9  70 170
$ cmp -c plik1 plik2
plik1 plik2 różnią się: bajt 4, linia 1 zawiera  63 3 170 x
$ cmp -cl plik1 plik2
 4  63 3    170 x
 9  70 8    170 x
```

2.6. Źródła uzupełniające

1. Manual pliku **/etc/group**: `https://man7.org/linux/man-pages/man5/group.5.html`
2. Manual pliku **/etc/gshadow**: `https://man7.org/linux/man-pages/man5/gshadow.5.html`
3. Manual pliku **/etc/passwd**: `https://man7.org/linux/man-pages/man5/passwd.5.html`
4. Manual pliku **/etc/shadow**: `https://man7.org/linux/man-pages/man5/shadow.5.html`
5. Manual metadanych zagregowanych w i-węzłach: `https://man7.org/linux/man-pages/man7/inode.7.html`
6. Manual polecenia **chgrp**: `https://man7.org/linux/man-pages/man1/chgrp.1.html`
7. Manual polecenia **chmod**: `https://man7.org/linux/man-pages/man1/chmod.1.html`
8. Manual polecenia **chown**: `https://man7.org/linux/man-pages/man1/chown.1.html`
9. Manual polecenia **cmp**: `https://man7.org/linux/man-pages/man1/cmp.1.html`
10. Manual polecenia **date**: `https://man7.org/linux/man-pages/man1/date.1.html`
11. Manual polecenia **df**: `https://man7.org/linux/man-pages/man1/df.1.html`
12. Manual polecenia **diff**: `https://man7.org/linux/man-pages/man1/diff.1.html`
13. Manual polecenia **du**: `https://man7.org/linux/man-pages/man1/du.1.html`
14. Manual polecenia **echo**: `https://man7.org/linux/man-pages/man1/echo.1.html`

15. Manual polecenia **file**: https://www.man7.org/linux/man-pages/man1/file.1.html
16. Manual polecenia **ln**: https://man7.org/linux/man-pages/man1/ln.1.html
17. Manual polecenia **passwd**: https://man7.org/linux/man-pages/man1/passwd.1.html
18. Manual polecenia **stat**: https://man7.org/linux/man-pages/man1/stat.1.html
19. Manual polecenia **sync**: https://man7.org/linux/man-pages/man1/sync.1.html
20. Manual polecenia **truncate**: https://man7.org/linux/man-pages/man1/truncate.1.html

2.7. Zadania do samodzielnego wykonania

1. Czy kolejność zestawów *kto-jak-co*, wymienionych w jednym poleceniu **chmod**, ma znaczenie? Podaj przykład, który uzasadnia twoją odpowiedź.
2. Oznacz jako **x** plik, który nie jest plikiem wykonywalnym, i wykonaj go. Jak zareagował system operacyjny?
3. Zapoznaj się z zawartością pliku **/etc/passwd**. Czy są tam użytkownicy, który używają powłoki innej niż **/bin/bash**?
4. Spróbuj zapoznać się z zawartością pliku **/etc/shadow**. Co się stało i dlaczego?
5. Zapoznaj się z zawartością pliku **/etc/group**. Czy są tam grupy, do których należy więcej niż jeden użytkownik?
6. Odszukaj w systemie operacyjnym plik wykonywalny **busybox** i stwórz do niego link symboliczny o nazwie **cp**. Uruchom link symboliczny (upewnij się, że uruchamiasz **busyboksa**, a nie oryginalne polecenie **cp**). Sprawdź, czy możesz po swoim dowiązaniu oczekiwać zachowania zgodnego z **cp**. Jakie jeszcze polecenia potrafi emulować **busybox**?
7. Czy przy pomocy polecenia **ln** można dowiązać się do któregoś ze swoich nadkatalogów i tym samym stworzyć zapętloną strukturę katalogów?
8. Stwórz plik, a następnie dowiązanie symboliczne, które do niego prowadzi. Usuń plik. Jak teraz zachowuje się dowiązanie?
9. Stwórz plik, dowiązanie do niego, a następnie dowiązanie do tego dowiązania. Zbadaj właściwości tego tworu. Spróbuj rozbudować łańcuch dowiązań.
10. Użyj polecenia **chmod** w celu zmiany praw dowiązania symbolicznego. Co się stało?
11. Użyj polecenia **stat** do zaprezentowania informacji o pliku w sposób maksymalnie zbliżony do wyjścia z polecenia **ls -l**.

12. W systemie operacyjnym istnieje para programów nazywających się **`true`** i **`false`**. Zbadaj ich działanie i zapoznaj się z ich opisem w manualu. Jakiego skutku można oczekiwać, gdy w pliku **`/etc/passwd`** pewien użytkownik będzie miał jako swoją powłokę wpisany jeden z tych programów? Czy obecnie w twoim systemie jest taki użytkownik?
13. Przeanalizuj sposób, w jaki program **`diff`** prezentuje odnalezione różnice między plikami. Zaplanuj i przeprowadź stosowny eksperyment, posługując się kolejno plikami identycznymi, różniącymi się nieznaczne, różniącymi się znacznie i kompletnie innymi. Upewnij się, że potrafisz przewidzieć, co powie ci **`diff`**, jeśli znasz dobrze zawartość obu plików.
14. Jeśli czasy w metadanych pliku byłyby reprezentowane przy pomocy liczb 32-bitowych, to kiedy skończyłaby się ich pojemność?
15. Czy polecenie **`file`** jest w stanie odróżnić kod źródłowy w C od kodu źródłowego w C++? Przeprowadź eksperyment.
16. Przy pomocy poleceń **`watch`** i **`date`** skonstruuj prosty, odświeżany co sekundę zegarek, który prezentuje informację o czasie w sposób przedstawiony poniżej (ostatnia linia prezentuje sekundy):

```
22 październik
2014
13:44
00
```

17. Użyj polecenia **`date`** do wypisania daty w formie zgodnej z normą ISO 8601, czyli na przykład **`2024-10-30`**.
18. Który tydzień roku mamy obecnie według ustaleń normy ISO 8601?
19. Co to jest *rozmiar pozorny* prezentowany przez polecenie **`du`**?
20. Co pojawi się na ekranie, jeśli użyjesz **`du`** z przełącznikiem **`-B512`**?
21. Sprawdź, ile przestrzeni dyskowej zajmuje twój katalog domowy.
22. Wydaj z konsoli polecenie, które zasygnalizuje ci upływ jednej minuty.

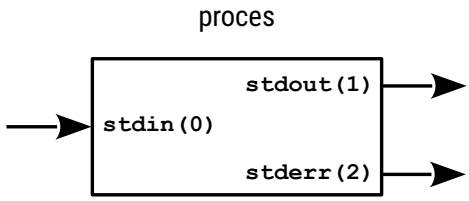
3. Strumienie i potoki

3.1. Strumienie

Strumień (ang. *stream*) jest abstrakcyjnym łączem znakowym (tzn. łączem zdolnym transmitować pojedyncze znaki/bajty, w odróżnieniu od łącz blokowych, które transmitują w jednym cyklu większe porcje danych, tak jak czynią to na przykład urządzenia dyskowe albo sieciowe), służącym do komunikacji pomiędzy pracującym procesem a światem zewnętrznym. W czasie pracy procesu strumień jest kojarzony z pewnym wybranym zasobem fizycznym, na przykład urządzeniem, plikiem czy też połączeniem sieciowym. Strumień identyfikowany jest przez nazwę oraz nieujemną liczbę całkowitą, która pełni rolę tak zwanego *deskryptora pliku* (ang. *file descriptor*) bądź według innej terminologii – uchwytu pliku (ang. *file handle*). Dana ta jest udostępniana procesowi w momencie otwarcia strumienia (na przykład z użyciem funkcji **`open()`**, która zwraca deskryptor poprzez swój wynik), a następnie proces używa jej do reprezentowania plików obsługiwanych przez funkcje systemowe takie jak na przykład **`read()`**, **`write()`** i **`close()`**.

Każdy proces w chwili uruchomienia otrzymuje od systemu trzy wstępnie otwarte strumienie reprezentujące podstawowe źródła i ujścia danych. Są to:

– **stdin** (od ang. *standard input*, standardowe wejście, deskryptor 0) – strumień domyślnie powiązany z podstawowym urządzeniem wejściowym (klawiatura terminala),
– **stdout** (ang. *standard output*, standardowe wyjście, deskryptor 1) – strumień domyślnie powiązany z podstawowym urządzeniem wyjściowym (ekran terminala),
– **stderr** (ang. *standard error output*, standardowe wyjście diagnostyczne, deskryptor 2) – strumień domyślnie powiązany z podstawowym urządzeniem wyjściowym (ekran terminala).



Rys. 3.1. Współpraca procesu ze strumieniami standardowymi

Strumienie domyślnie wykorzystywane są w sposób następujący:
– dane wprowadzane przez użytkownika przez klawiaturę trafiają do strumienia 0,
– dane wyprowadzane przez proces trafiają poprzez strumień 1 na ekran bądź inne urządzenie wizualizujące,
– dane diagnostyczne (na przykład komunikaty o błędach) trafiają przez strumień 2 na ekran bądź inne urządzenie wizualizujące.

Zasadniczo użytkownik nie jest w stanie rozpoznać, które z prezentowanych na ekranie danych dotarły do niego przez strumień 1 (*stdout*), a które przez strumień 2 (*stderr*), ale istniejące pomiędzy nimi rozróżnienie pozwala odseparować te informacje od siebie, o ile użytkownik sobie tego zażyczy.

3.2. Przekierowania

Przekierowaniem (ang. *redirection*) nazywa się działanie, w wyniku którego pewien strumień danych zostaje skojarzony z innym urządzeniem (lub plikiem) niż przypisanym domyślnie przez system. Przekierowanie określone w linii poleceń obowiązuje od momentu uruchomienia procesu aż do jego zakończenia. Warto zauważyć, że proces ma możliwość wykrycia faktu przekierowania wykorzystywanego przez niego strumienia i może zmodyfikować swoje działanie zależnie od sytuacji. Na

przykład polecenie **ls** może kolorować nazwy plików, gdy są one wyświetlane na ekranie, i nie robić tego, gdy wynik działania narzędzia jest kierowany do pliku.

Przy tej okazji warto wspomnieć też o ciekawym urządzeniu, które reprezentowane jest w systemie jako plik o nazwie **/dev/null**, a które zachowuje się jak czarna dziura. Oznacza to, że wszelkie dane kierowane do tego urządzenia (pliku) przepadają w nim bez najmniejszego śladu, a w przypadku próby odczytu plik ten zachowuje się tak, jakby był pusty, tj. natychmiast sygnalizuje sytuację końca danych. Z tego też powodu plik **/dev/null** jest często wykorzystywany jako śmietnik dla danych, które nie są nikomu potrzebne, lub jako wzorzec pustego pliku.

Do definiowania przekierowań używa się następującego zbioru operatorów:

> przekierowanie standardowego wyjścia; w wyniku działania tego operatora dane wyprowadzane przez proces na strumień 1 zostaną **skierowane** do pliku/urządzenia wymienionego jako prawy argument operatora; jeżeli plik określony jako ujście danych już istnieje, jego zawartość zostanie usunięta i zastąpiona danymi wprowadzonymi przez proces; takiej operacji nie towarzyszy żadne ostrzeżenie, więc wskazana jest ostrożność; na przykład:

```
ls -l > ls-l.txt
```

spowoduje, że w pliku *ls-l.txt* zostanie umieszczona długa lista plików znajdujących się w bieżącym katalogu;

2> przekierowanie wyjścia diagnostycznego; w wyniku działania tego operatora dane wyprowadzane przez proces na strumień 2 zostaną **skierowane** do pliku/urządzenia wymienionego jako prawy argument operatora zgodnie z zachowaniem opisanym wcześniej dla operatora **>**; na przykład:

```
ls -l hidden_dir 2> error.txt
```

spowoduje, że w pliku *error.txt* zostanie umieszczona treść komunikatu o błędzie, który pojawi się, jeśli katalog *hidden_dir* jest niedostępny.

>> przekierowanie standardowego wyjścia; w wyniku działania tego operatora dane wyprowadzane przez proces na strumień 1 zostaną **dopisane** do pliku wymienionego jak prawy argument operatora; jeżeli plik określony jako ujście danych już istnieje, jego dotychczasowa zawartość pozostanie nienaruszona; jeśli nie istnieje, to zostanie utworzony; na przykład:

```
ls -l .. >> ls-l.txt
```

spowoduje, że do pliku *ls-l.txt* zostanie dopisana długa lista plików znajdujących się w katalogu nadrzędnym.

2>> przekierowanie wyjścia diagnostycznego; w wyniku działania tego operatora dane wyprowadzane przez proces na strumień 2 zostaną **dopisane** do pliku wymienionego jako prawy argument operatora zgodnie z zachowaniem opisanym dla operatora **>>**; na przykład:

```
ls /root 2>> error.txt
```

spowoduje, że do pliku *error.txt* zostanie dopisana treść nieuniknionego komunikatu o błędzie, który pojawi się, jeśli nie jesteś użytkownikiem *root*.

< przekierowanie standardowego wejścia; w wyniku działania tego operatora dane z pliku o nazwie podanej jako prawy argument zostaną **wprowadzone** do procesu na jego strumień 0 dokładnie w taki sposób, jakby zostały odczytane z klawiatury; znane już czytelnikowi polecenie **less** może wyświetlać dane z pliku o wskazanej nazwie albo też (jeśli nie podano żadnych nazw plików) odczytywać je ze swojego standardowego wejścia; z tego powodu oba podane poniżej postaci są równoważne:

```
less dane.txt
less <dane.txt
```

<< wielowierszowe przekierowanie standardowego wejścia; w wyniku działania tego operatora wszystkie kolejne wiersze wprowadzane z klawiatury zostaną **wprowadzone** do procesu na jego strumień 0, a działanie to zostanie zakończone z chwilą napotkania na wejściu wiersza zawierającego taki sam ciąg znaków, jak podany w prawym argumencie operatora; na przykład:

```
$ cat <<STOP
Wlazł kotek na płotek
I mruga
STOP
```

przydatność tej konstrukcji w zwykłej pracy konsolowej jest znikoma, jednak może ona oddać nieocenione usługi w przypadku użycia jej wewnątrz skryptu.

Wiele z poleceń wyprowadza wyniki swojego działania na standardowe wyjście oraz równolegle dodatkowe informacje o błędach i/lub ostrzeżeniach na standardowe wyjście diagnostyczne. Ponieważ operatory dotyczące różnych strumieni można ze sobą dowolnie łączyć w jednej linii poleceń, tym samym istnieje możliwość rozdzielenia tych strumieni, na przykład w taki sposób:

```
cat in1 in2 2> err
```

Powyższe polecenie spowoduje wyświetlenie zawartości plików *in1* i *in2* oraz zapisanie informacji o błędach do pliku *err*.

W celu zignorowania danych, które z jakichkolwiek względów nie są do niczego potrzebne, można je przekierować do pliku **/dev/null**, na przykład:

```
cat in1 in2 2> /dev/null
```

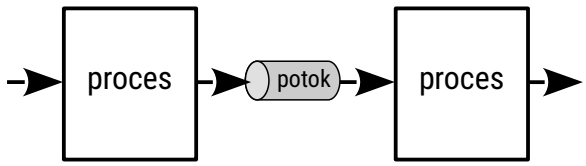
W przypadku, gdy standardowy strumień wyjściowy został już przekierowany, a życzymy sobie, aby dane ze strumienia diagnostycznego trafiły w to samo miejsce, można użyć zapisu **2>&1**, na przykład:

```
cat in1 in2 in3 in4 in5 >out 2>&1
```

3.3. Potoki

Potok (ang. *pipeline* lub w skrócie *pipe*) to łańcuch współpracujących procesów skonstruowany w taki sposób, aby każdy z nich (oprócz pierwszego) czytał dane z wyjścia procesu poprzedniego oraz aby każdy z nich (oprócz ostatniego) pisał swoje wyniki na wejście procesu następnego. Dane przesyłane pomiędzy procesami są w przezroczysty sposób buforowane przez system operacyjny, sprawiając wrażenie, że każdy z procesów pracuje absolutnie autonomicznie w takt napływających danych.

Można stwierdzić, że potok jest specyficzną formą przekierowania, w której źródłem/ujściem danych nie jest plik/urządzenie, ale inny pracujący **proces**. Operatorem, który spina dwa procesy w potok, jest znak **|** (ang. *bar*, chociaż spotyka się również określenie *pipe*). Jego lewy argument jest nadajnikiem danych, a prawy odbiornikiem. Warto zauważyć, że w potokach, w których uczestniczą więcej niż dwa procesy, wszystkie oprócz pierwszego i ostatniego są zarówno odbiornikami, jak i nadajnikami.



Rys. 3.2. Współpraca procesów poprzez potok

Oto kilka prostych przykładów:

`ls -al | less`
wyniki produkowane przez polecenie **`ls`** są tu buforowane przez proces **`less`** i prezentowane na ekranie w wygodny sposób, który umożliwia użytkownikowi ich swobodne przeglądanie i przeszukiwanie (w tym miejscu warto być może przypomnieć, że wyjście z przeglądarki **`less`** następuje po naciśnięciu klawisza *q*);

`who | less`
jak wyżej, ale w odniesieniu do listy zalogowanych użytkowników;

`who | sort | less`
jak wyżej, ale przed wyświetleniem na konsoli lista zostanie posortowana alfabetycznie według nazw użytkowników.

Nie każdy proces jest zdolny do pracy w potoku. Te programy, które nie wykorzystują standardowego wejścia jako źródła przetwarzanych danych oraz standardowego wyjścia do wyprowadzania wyników, nie nadają się do tego celu (na przykład edytory). Oznacza to, że najczęściej w potokach wykorzystuje się tylko te narzędzia, które zostały do tego specjalnie stworzone. Są to tak zwane *filtry*.

3.4. Filtry

Filtrem nazywa się program, którego podstawowym zadaniem jest wpływanie na postać danych odbieranych ze standardowego wejścia (niekiedy również z pliku, którego nazwę filtr dostaje w linii poleceń) i przekazywanie ich w zmienionej formie na standardowe wyjście (niekiedy również do pliku, którego nazwę filtr dostaje w linii poleceń). Pełnię swoich możliwości filtr ujawnia, gdy zostanie umiejętnie zastosowany jako element potoku. Poniżej przedstawiono krótki opis najczęściej używanych i najbardziej przydatnych filtrów stosowanych standardowo w systemie Linux.

Dla celów dalszych rozważań założono istnienie pliku *studenci.txt*, który mógłby być częścią listy ocen z pewnej trudnej wejściówki, a który zawiera następujące cztery wiersze tekstu:

```
Waldemar Zaskoczony 3
Roksana Pewna 3,5
Euzebia Nadzwyczajna 05
Waldemar Zaskoczony 3
```

Założono również, że każdy prezentowany potok rozpoczyna się od polecenia **cat**, którego zadaniem będzie jedynie „karmienie" potoku danymi, natomiast ewentualne przekształcenia zawartości pliku odbywać się będą w kolejnych ogniwach.

cat (od ang. *concatenate*)
zasadniczo nie wpływa na treść przesyłanych w nim danych, chociaż można sprawić, że będzie je subtelnie modyfikował zgodnie z podanymi mu przełącznikami, na przykład:

- **-b** numeruje niepuste linie;
- **-n** numeruje wszystkie linie;
- **-s** zastępuje wiele następujących po sobie pustych linii jedną;
- **-v** prezentuje znaki niedrukowalne w czytelny i bezpieczny sposób, niezakłócający pracy terminala.

Poniższy przykład ilustruje użycie polecenia **cat** do ponumerowania wszystkich linii tekstu z potoku:

```
$ cat studenci.txt | cat -n
     1 Waldemar Zaskoczony 3
     2 Roksana Pewna 3,5
     3 Euzebia Nadzwyczajna 05
     4 Waldemar Zaskoczony 3
```

head
wyprowadza na standardowe wyjście początkową porcję danych pojawiających się na standardowym wejściu (domyślnie jest to 10 pierwszych linii); może też odczytywać dane z plików o nazwach podanych jako argumenty; jego zachowanie mogą modyfikować następujące przełączniki:

-c *x* przesyła *x* pierwszych znaków;
-n *x* przesyła *x* pierwszych linii.

Poniższy przykład ilustruje użycie polecenia **head** do przesłania jedynie pierwszych dwóch linii tekstu z potoku:

```
$ cat studenci.txt | head -n 2
Waldemar Zaskoczony 3
Roksana Pewna 3,5
```

tail
wyprowadza na standardowe wyjście końcową porcję danych pojawiających się na standardowym wejściu (domyślnie 10 ostatnich linii); może odczytywać dane z plików o nazwach podanych jako argumenty; jego zachowanie mogą modyfikować następujące przełączniki:
-c *x* przesyła *x* ostatnich znaków;
-n *x* przesyła *x* ostatnich linii.

Poniższy przykład ilustruje użycie polecenia **tail** do wyprowadzenia jedynie ostatniej linii z potoku:

```
$ cat studenci.txt | head -n 2 | tail -1
Roksana Pewna 3,5
```

sort
służy do sortowania danych wejściowych, które domyślnie porządkowane są leksykograficznie (słownikowo); sortowanie danych odbywa się wierszami; najważniejsze przełączniki to:

- **-n** wymusza sortowanie numeryczne (próbuje traktować klucze sortowania jak liczby, nie jak tekst);
- **-b** ignoruje spacje na początkach linii;
- **-d** wymusza tak zwany *tryb książki telefonicznej*, to znaczy ignoruje w czasie sortowania znaki, które nie są literami albo cyframi;
- **-f** ignoruje wielkość liter (utożsamia litery małe z wielkimi);
- **-t *x*** zmienia domyślny separator kolumn (tzn. umownych pól wewnątrz wierszy) na znak *x* (domyślnie separatorami są spacje i tabulacje);
- **-r** wymusza odwrotny porządek sortowania;
- **-i** ignoruje znaki niedrukowalne;
- **-k *k*** wskazuje położenie klucza **-k *k***, według którego będzie odbywać się sortowanie; w najprostszym przypadku **sort** zakłada, że każda z linii składa się z ponumerowanych od 1 pól/kolumn (ciągów znaków rozdzielanych domyślnie białymi znakami), a znaki wewnątrz każdego pola z osobna również są numerowane i także od 1; najprostsza postać definicji klucza to:
  col_beg.char_beg,col_end.char_end
  gdzie **col_beg** to numer pola zawierającego początek klucza, a **char_beg** to numer znaku w polu, od którego zaczyna się klucz; analogicznie **col_end** i **char_end** określają położenie końca klucza i jeśli są pominięte wraz z poprzedzającym przecinkiem, to

zakłada się, że klucz kończy się na ostatnim znaku linii, na przykład:

```
sort -k 2.1
```

pomija w sortowaniu pierwszą kolumnę.

Poniżej podano kilka elementarnych przykładów prezentujących różne zachowania filtra **sort**. Oto jak działa posortowanie linii z potoku bez wskazania klucza:

```
$ cat studenci.txt | sort
Euzebia Nadzwyczajna 05
Roksana Pewna 3,5
Waldemar Zaskoczony 3
Waldemar Zaskoczony 3
```

Posortowanie linii z potoku alfabetycznie według nazwiska:

```
$ cat studenci.txt | sort -k 2
Euzebia Nadzwyczajna 05
Roksana Pewna 3,5
Waldemar Zaskoczony 3
Waldemar Zaskoczony 3
```

Posortowanie linii z potoku alfabetycznie według oceny:

```
$ cat studenci.txt | sort -k 3
Euzebia Nadzwyczajna 05
Waldemar Zaskoczony 3
Waldemar Zaskoczony 3
Roksana Pewna 3,5
```

Posortowanie linii z potoku numerycznie według oceny:

```
$ cat studenci.txt | sort -k 3 -n
Waldemar Zaskoczony 3
Waldemar Zaskoczony 3
Roksana Pewna 3,5
Euzebia Nadzwyczajna 05
```

uniq (od ang. *unique*)
usuwa **powtarzające się, sąsiednie linie**, wczytywane z wejścia albo z plików o nazwach podanych w argumentach; używany z następującymi przełącznikami:

- **-d** wyprowadza tylko powtarzające się linie (czyli jest to odwrócenie zachowania domyślnego);
- **-u** wyprowadza tylko linie unikalne;
- **-c** poprzedza wyprowadzane linie licznikami powtórzeń.

Ze względu na sposób jego działania użycie filtra **uniq** ma sens tylko i wyłącznie wtedy, gdy napływające do niego wiersze tekstu są już wstępnie posortowane – tylko w takim przypadku może mieć miejsce faktyczne usuwanie bądź zliczanie powtórzeń. Dlatego też filtr ten najczęściej pojawia się w potokach dopiero po filtrze **sort**.
Poniższy przykład ilustruje użycie filtra **uniq** do odnalezienia w potoku powielonej linii:

```
$ cat studenci.txt | sort | uniq -d
Waldemar Zaskoczony 3
```

Policzenie liczby wystąpień każdej linii w posortowanym potoku:

```
$ cat studenci.txt | sort | uniq -c
      1 Euzebia Nadzwyczajna 05
      1 Roksana Pewna 3,5
      2 Waldemar Zaskoczony 3
```

wc (od ang. *word counter*)
wyprowadza na wyjście wartości liczników podających, ile w danych wejściowych było linii, słów i znaków; przełączniki służą do wybrania podzbioru prezentowanych wartości, i tak:

- **-l** wyprowadź tylko licznik linii;
- **-w** wyprowadź tylko licznik słów;
- **-c** wyprowadź tylko licznik znaków.

Zebranie statystyk przetwarzanego potoku ilustruje poniższy przykład:

```
$ cat studenci.txt | wc
      4      12      86
```

tr (od ang. *translate*)
filtr **tr** służy do zamiany wybranych znaków na inne bądź też do ich usunięcia z potoku i używa następującej składni do wyspecyfikowania argumentów:
tr [przełącznik...] zbiór1 [zbiór2]
efektem działania filtra jest zamiana znaków ze wskazanego zbioru *zbiór1* na znaki ze zbioru *zbiór2*; wymaga podania co najmniej jednego zbioru (w takim przypadku uznaje się, że drugi zbiór jest pusty, co oznacza żądanie usuwania znaków wymienionych w pierwszym zbiorze); w pełnej wersji **tr** używa złożonej składni do zapisywania uogólnionych postaci zbiorów, w przykładzie natomiast poprzestano na postaci prostszej, w której składowymi zbiorów są pojedyncze znaki; najczęściej stosowane przełączniki to:

- **-c** traktuje pierwszy zbiór tak, jakby był swoim własnym dopełnieniem (czyli wszystkimi znakami różnymi od podanych *explicite*);
- **-s** usuwa powtarzające się, sąsiednie znaki.

Zamianę przecinków na kropki zrealizuje poniższa postać użycia filtra:

```
$ cat studenci.txt | tr , .
Waldemar Zaskoczony 3
Roksana Pewna 3.5
Euzebia Nadzwyczajna 05
Waldemar Zaskoczony 3
```

cut
wycina wskazane fragmenty wierszy podawanych na wejście i w sposób zależny od przełączników wysyła na wyjście; najczęściej używa się następujących przełączników:

- **-c** określa początkowe i końcowe kolumny wycinanych fragmentów, na przykład **-c 1-10** wybiera pierwsze 10 znaków z każdego wiersza;
- **-f** określa numery wycinanych pól, na przykład **-f1,3-5,10** wyświetla pola 1, 3, 4, 5 i 10;
- **-d *c*** ustawia znak ***c*** (ewentualnie ciągi znaków) użyte w pliku do separowania pól (separatorem domyślnym jest tabulator).
  Uwaga: przełączniki **-c** *i* **-f** *wykluczają się wzajemnie i nie mogą być użyte więcej niż raz; co więcej, przełączników tych nie można użyć do zmiany kolejności znaków/pól, na przykład* **-f3,2** *wywoła taki skutek jak* **-f2,3***; do zamiany kolejności pól najczęściej używa się bardziej uniwersalnego narzędzia o nazwie* **awk** *(nazwa pochodzi od nazwisk twórców pierwszej wersji tego programu, Alfreda V. Aho, Petera Weinbergera i Briana Kernighana, i odczytuje się ją jako jedno słowo*

awk, *wymawiane jak pierwsza sylaba w słowie* awkward*; rozpoznanie możliwości* **awk** *pozostawia się czytelnikowi).*

Wybranie z potoku wyłącznie nazwisk i ocen ilustruje poniższy przykład:

```
$ cat studenci.txt| cut -d ' ' -f2,3
Zaskoczony 3
Pewna 3,5
Nadzwyczajna 05
Zaskoczony 3
```

grep

Pochodzenie nazwy tego niezwykle elastycznego narzędzia nie jest do końca jasne. Być może jest to skrót od angielskich słów *global regular expression print*, ale nie jest też wykluczone, że *grep* wziął swoją nazwę od nieużywanego już interaktywnie, prymitywnego wierszowego edytora **ex** (zdarza się jeszcze, że używa się go jako komponentu skryptów do wykonania automatycznie złożonych modyfikacji plików). Edytor **ex** do wyszukania łańcucha w tekście używał polecenia o następującej postaci:

```
:g/wyrażenie regularne/p
```

a ponieważ w angielszczyźnie wyrażenia regularne są często określane mianem *re* (od słów *regular expression*), to w efekcie otrzymujemy wyglądający już bardzo znajomo ciąg znaków **g/re/p**. Podstawowa składnia uruchomienia polecenia **grep** prezentuje się następująco:

```
grep [przełącznik...] wyrażenie
```

Użyty jako filtr **grep** wyszukuje w strumieniu danych odbieranym ze standardowego wyjścia ciągu znaków opisanego argumentem **wyrażenie** i reaguje stosownie do użytych przełączników; w najprostszym przypadku wyprowadza na swoje standardowe wyjście tylko te wiersze, które zawierają poszukiwany ciąg.

Uwaga: argument **wyrażenie** *może być wyrażeniem regularnym, jednak ten wariant zostanie omówiony w następnym rozdziale; na razie uznano, że wyrażenie jest łańcuchem znaków do wyszukania; należy mieć jednak na uwadze, że pewne znaki mogą być tutaj traktowane specjalnie (są to:* **.** *(kropka),* **^** *(caret),* **$** *(dolar),* ***** *(gwiazdka),* **[]** *(nawiasy kwadratowe),* **<>** *(nawiasy ostre),* **** *(backslash); jeśli zachodzi konieczność wyszukania ich jako zwykłych znaków, należy je poprzedzać (zacytować) znakiem* **** *(backslash); ponadto* **grep** *potrafi również przeszukiwać pliki o nazwach wymienionych w argumentach, ale ten wariant zostanie omówiony w kolejnym rozdziale.*

Najczęściej używane przełączniki to:

- **-v** wyprowadza linie niezawierające szukanego wzorca (jest to odwrócenie zachowania domyślnego);
- **-c** wyprowadza liczbę odnalezionych wyrażeń;
- **-i** ignoruje wielkość liter przy wyszukiwaniu (utożsamia litery małe z wielkimi);
- **-n** wyprowadza numery linii zawierających wyszukiwany wzorzec.

Wybranie z potoku ocen pana Zaskoczonego zrealizuje następujący potok:

```
$ cat studenci.txt| grep -i zaskoczony | cut -d ' ' -f3
3
3
```

3.5. Źródła uzupełniające

1. Manual polecenia **cat**: `https://man7.org/linux/man-pages/man1/cat.1.html`
2. Manual polecenia **cut**: `https://man7.org/linux/man-pages/man1/cut.1.html`
3. Manual polecenia **grep**: `https://man7.org/linux/man-pages/man1/grep.1.html`
4. Manual polecenia **head**: `https://man7.org/linux/man-pages/man1/head.1.html`
5. Manual polecenia **less**: `https://man7.org/linux/man-pages/man1/less.1.html`
6. Manual polecenia **sort**: `https://man7.org/linux/man-pages/man1/sort.1.html`
7. Manual polecenia **tail**: `https://man7.org/linux/man-pages/man1/tail.1.html`
8. Manual polecenia **tr**: `https://man7.org/linux/man-pages/man1/tr.1.html`
9. Manual polecenia **uniq**: `https://man7.org/linux/man-pages/man1/uniq.1.html`
10. Manual polecenia **wc**: `https://man7.org/linux/man-pages/man1/wc.1.html`
11. Opis mechanizmu potoków w powłoce Bash: `https://www.gnu.org/software/bash/manual/bash.html#Pipelines`
12. Opis mechanizmu przekierowań w powłoce Bash: `https://www.gnu.org/software/bash/manual/bash.html#Redirections`

3.6. Zadania do samodzielnego wykonania

1. Program **cat** uruchomiony bez argumentów kopiuje na ekran znaki wprowadzane z klawiatury. Sprawdź, czy **cat** wykonuje to kopiowanie znak po znaku, czy linia po linii.
2. Jak zakończyć działanie programu **cat** w powyższej sytuacji? Podaj dwa dostępne sposoby. Który z nich jest lepszy?
3. Jak **cat** prezentuje znaki niedrukowalne?
4. Jak użyć programu **cat**, aby wystąpił jako zamiennik programu **cp** w poniższym kontekście?
   ```
   cp plik1 katalog2/plik2
   ```
5. Jak przy pomocy programu **cat** uzyskać efekt identyczny z tym, który uzyskamy przy użyciu programu **touch** z nazwą nieistniejącego pliku?
6. Zbadaj, na który ze strumieni trafia komunikat emitowany przez polecenie **cd**, jeśli użyto w nim nazwy nieistniejącego katalogu. Jak się tego dowiedzieć?
7. Wiele poleceń systemu Linux uznaje, że jeśli w linii poleceń jako nazwa pliku wystąpi znak **-** (łącznik), to reprezentuje on standardowe wejście. Tak zachowuje się też polecenie **cat**. Załóż, że **a** i **b** są nazwami istniejących, niepustych plików i sprawdź, jaki efekt wywoła polecenie w następującej postaci:
   ```
   cat a - b > c
   ```
8. Jak ukryć komunikat o błędzie produkowany przez poniższe polecenie?
   ```
   cat /etc/shadow
   ```
9. Dlaczego nie istnieje operator **2<**?
10. Czy istnieje operator **>&1**?
11. Filtry **head** i **tail** potrafią posługiwać się kilobajtami i megabajtami jako licznikami początkowych/końcowych znaków; sprawdź w manualu, jak osiąga się ten efekt.
12. Przeprowadź eksperyment pokazujący, jak polecenie w postaci
    tail -f nazwa_pliku prezentuje w czasie rzeczywistym fakt przyrastania danych w pliku (*wskazówka: posłuż się więcej niż jednym oknem terminala*).
13. Wyjaśnij, co to znaczy, że domyślne sortowanie stosowane przez filtr **sort** jest *leksykograficzne*. W jakich sytuacjach sortowanie takie będzie nieprzydatne?
14. Skonstruuj eksperyment ilustrujący, na czym polega sortowanie w trybie numerycznym (opcja **-n** polecenia **sort**).
15. Czym jest *słowo* w sensie używanym przez filtr **wc**?
16. Posortuj plik **/etc/passwd** według identyfikatorów UID. Zapisz wynik do pliku.
17. Ile jest różnych (niepowtarzalnych) identyfikatorów GID w pliku **/etc/passwd**?
18. Sporządź własną kopię pliku **/etc/group**, w którym wszystkie znaki przestankowe zostaną zamienione na spacje.

19. Do czego służy nieomówiony w tekście filtr o nazwie **`tee`**?
20. Do czego służy nieomówiony w tekście filtr o nazwie **`pv`**?
21. Pewien plik zawiera listę studentów. Każda linia składa się kolejno z nazwiska, imienia i oceny, rozdzielonych znakiem spacji. Sporządź nową wersję tego pliku tak, aby w każdej linii znalazły się najpierw ocena, a potem nazwisko (*wskazówka: użyj nieomówiony w instrukcji filtr* **`paste`**).
22. Otrzymany nowy plik posortuj według nazwisk.
23. Pewien plik zawiera numery PESEL, po jednym w linii. Jak wykryjesz powtarzające się numery? Jak je policzysz?
24. Pewien zestaw pięciu plików tekstowych zawiera listy obecności sporządzane na pięciu kolejnych wykładach. Jak wyszukać studentów, którzy byli na wszystkich wykładach?

4. Wyszukiwanie informacji

4.1. Wyrażenia regularne

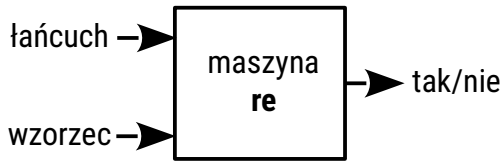
Wyrażeniem regularnym (ang. *regular expression*, często skracane do *regex*, *regexp* albo *re*) nazywa się ciąg znaków (napis) tworzący wzorzec wyszukiwania (często również wzorzec zamiany) i nakładany na inny ciąg w celu stwierdzenia istnienia lub braku zgodności między nimi. Pomysł wyrażeń regularnych pochodzi od amerykańskiego matematyka Stephena Kleene'a (1909–1994), który w 1950 roku pracował nad przetwarzaniem i rozpoznawaniem języka naturalnego.

Każdy znak wyrażenia regularnego jest albo tak zwanym *metaznakiem* (któremu przypisuje się specjalne znaczenie) albo *literałem* (to znaczy znakiem, który oznacza sam siebie).

Semantykę wyrażeń regularnych definiuje się najczęściej przy pomocy tak zwanej *maszyny wyrażeń regularnych* będącej deterministycznym automatem skończonym. Maszyna próbuje odszukać w przekazanym jej łańcuchu taką sekwencję znaków, która dopasowuje się do podanego jej wzorca. Wynik próby dopasowania zwracany jest jako rezultat działania maszyny.

Współcześnie wyrażenia regularne są powszechnie używane do wyszukiwania napisów i manipulowania nimi, przy czym podejście do ich stosowania bywa bardzo różnorodne. W niektórych językach programowania wyrażenia regularne

są dostępne wprost i stanowią część definicji języka (na przykład Perl, AWK), w innych dostępne są za pośrednictwem funkcji bibliotecznych (na przykład język C i biblioteka *regex*) bądź specjalizowanych klas (na przykład Java i klasy *Pattern* i *Matcher*) czy też modułów (na przykład Python z modułem *re*).



Rys. 4.1. Środowisko działania maszyny wyrażeń regularnych

W dalszej części tekstu zostanie opisany podzbiór metaznaków języka wyrażeń regularnych oraz reguły ich dopasowywania w postaci akceptowanej przez większość narzędzi w systemach uniksowych. W większości przypadków zaprezentowane zostaną także przykłady wzorców i napisów wraz z wynikiem dopasowania.

4.2. Reguły dopasowania wzorca

1. Dopasowanie wzorca do łańcucha odbywa się zawsze od lewej do prawej.
2. Jeżeli pewien znak wzorca nie jest metaznakiem, to dopasowuje się do napisu literalnie, to znaczy dopasowanie wystąpi tylko wtedy, gdy na odpowiednim miejscu napisu znajdować się będzie dokładnie taki sam znak (ewentualnie z dokładnością do wielkości litery); w szczególności wszystkie cyfry i litery dopasowują się zawsze literalnie:

napis	*wzorzec*	*dopasowanie?*
`Ala ma kota`	`kota`	tak
`Ala ma kota`	`koty`	nie

3. Znaki literalne można zapisywać za pomocą ich kodów ASCII wyrażonych szesnastkowo z przedrostkiem `\x` lub ósemkowo z przedrostkiem `\o` (kod ASCII litery `A` to 41 szesnastkowo i 101 ósemkowo):

napis	*wzorzec*	*dopasowanie?*
`Ala ma kota`	`\x41`	tak
`Ala ma kota`	`\o101`	tak

4. Metaznakami są znaki z następującego zbioru:
   - `.` (kropka),
   - `^` (*caret*),
   - `$` (dolar),
   - `*` (gwiazdka),
   - `[]` (nawiasy kwadratowe),
   - `<>` (nawiasy ostre),
   - `()` (nawiasy okrągłe),
   - `\` (*backslash*).
5. Jeżeli za znakiem `\` stoi jeden ze znaków wymienionych poniżej, to element taki dopasowuje się do jednego ze znaków sterujących:
   - `\n` znak końca linii (*LF*),
   - `\r` znak końca linii (*CR*),
   - `\t` tabulator (*HT*),
   - `\e` tak zwany *znak ucieczki* (ang. *escape char* – w ASCII jest to znak o kodzie 27 i używany jest, między innymi, do sterowania terminalem),
   - `\f` znak przejścia do nowej strony (*FF*).
6. Jeżeli za znakiem `\` znajduje się jeden ze znaków wymienionych poniżej, to element taki dopasowuje się do klasy znaków (warto zauważyć, że litera wielka neguje znaczenie litery małej):
   - `\s` tak zwany *biały znak* (ang. *whitespace*),
   - `\S` znak niebędący białym znakiem,
   - `\w` znak w słowie (tak zwany *znak słowotwórczy* – litera, cyfra, podkreślenie),
   - `\W` znak, który nie jest znakiem w słowie,
   - `\b` miejsce w napisie na granicy słowa,
   - `\B` miejsce w napisie pomiędzy znakami słowa.

napis	*wzorzec*	*dopasowanie?*
`Ala ma kota`	`\s\S\S\s`	tak
`Ala nie ma kota`	`e\b`	tak
`Ala nie ma kota`	`\be`	nie

7. Metaznak `.` (kropka) dopasowuje się do dowolnego (jednego!) znaku w napisie:

napis	*wzorzec*	*dopasowanie?*
`Ala ma kota`	`kot.`	tak
`Ala ma koty`	`kot.`	tak

8. Ujęcie listy znaków w nawiasy kwadratowe `[]` definiuje tak zwany *zbiór znaków*; zbiór znaków dopasowuje się do pewnego znaku w napisie, gdy znak ten jest elementem zbioru:

napis	*wzorzec*	*dopasowanie?*
`Ala ma trzy koty`	`kot[uy]`	tak
`Ala idzie z kotem`	`kot[uy]`	nie
`Ala mówi kotu`	`kot[uy]`	tak

9. Jeżeli pierwszym znakiem zbioru jest `^`, to zbiór rozumiany jest jako swoje dopełnienie, tzn. traktowany jest tak, jakby składał się ze wszystkich tych znaków, których nie wymieniono:

napis	*wzorzec*	*dopasowanie?*
`Ala ma trzy koty`	`kot[^uy]`	nie
`Ala idzie z kotem`	`kot[^uy]`	tak
`Ala mówi kotu`	`kot[^uy]`	nie

10. Zbiór może być definiowany tak zwanym *zakresem*, podawanym jako krańce dolny i górny rozdzielone znakiem łącznika (`-`); zakres rozumiany jest jako wszystkie znaki, które w danym alfabecie leżą pomiędzy wskazanymi krańcami wraz z tymi krańcami; jeżeli system operacyjny został prawidłowo zlokalizowany, zakres powinien obejmować sobą również ewentualnie istniejące pomiędzy krańcami znaki narodowe:

napis	*wzorzec*	*dopasowanie?*
`Ala ma 12 kotów`	`1[1-5]`	tak
`Ala ma 16 kotów`	`1[1-5]`	nie
`Ala ma 10 kotów`	`1[1-5]`	nie

11. Jeden zbiór może zawierać dowolnie wiele zakresów, a zakresy mogą być rozdzielane pojedynczymi elementami:

napis	*wzorzec*	*dopasowanie?*
`x = 0xff;`	`0x[0-9a-f]`	tak
`x = 0xff;`	`0x[0-9A-F]`	nie
`x = 0xff;`	`[0-9abc-f]`	tak

12. Jeżeli wewnątrz zbioru umieszczony jest element w postaci **`[:klasa:]`**, to element taki dopasowuje się do odpowiedniego znaku w napisie wtedy, gdy znak ten należy do wskazanej klasy; definiuje się następujące klasy i ich nazwy:

nazwa	*klasa*	*zbiór równoważny*
`[:alnum:]`	litera (mała lub wielka) lub cyfra dziesiętna	`[a-zA-Z0-9]`
`[:alpha:]`	litera (mała lub wielka)	`[a-zA-Z]`
`[:ascii:]`	znak kodu ASCII	`[\x00-\x7F]`
`[:blank:]`	spacja lub tabulator	`[ \t]`
`[:cntrl:]`	znak sterujący	`[\x00-\x1F\x7F]`
`[:digit:]`	cyfra dziesiętna	`[0-9]`
`[:lower:]`	mała litera	`[a-z]`
`[:graph:]`	znak widoczny (każdy oprócz białych i sterujących)	`[\x21-\x7E]`
`[:print:]`	znak drukowalny (każdy oprócz sterujących)	`[\x20-\x7E]`
`[:punct:]`	znak przestankowy	`[!"#$%&'()*+ ,\-./:;<=>?@ []^_‘{|}~]`
`[:space:]`	biały znak	`[ \t\r\n\f]`
`[:upper:]`	wielka litera	`[A-Z]`
`[:xdigit:]`	cyfra szesnastkowa	`[0-9a-fA-F]`

Na przykład:

napis	*wzorzec*	*dopasowanie?*
`Ola`	`[[:upper:]]la`	tak
`bla`	`[[:upper:]]la`	nie
`dla`	`[[:lower:]]la`	tak

13. Metaznak `^` dopasowuje się nie do znaku, ale do miejsca, które leży przed pierwszym znakiem napisu:

napis	*wzorzec*	*dopasowanie?*
`Ala ma kota`	`^Ala`	tak
`Ala ma kota`	`^kota`	nie

14. Metaznak `$` dopasowuje się nie do znaku, ale do miejsca, które leży za ostatnim znakiem napisu:

napis	*wzorzec*	*dopasowanie?*
`Ala ma kota`	`Ala$`	nie
`Ala ma kota`	`kota$`	tak

15. Metaznaki `\<` i `\>` dopasowują się do miejsc, które leżą odpowiednio przed pierwszym i za ostatnim znakiem słowa (tzn. na początkowej i końcowej granicy słowa):

napis	*wzorzec*	*dopasowanie?*
`To jest rzeka`	`\<rzeka\>`	tak
`On narzekał`	`\<rzeka\>`	nie

16. Metaznak `?` oznacza, że poprzedni element wzorca występuje w napisie opcjonalnie (dokładnie zero razy lub raz):

napis	*wzorzec*	*dopasowanie?*
`To jest kot`	`koty?`	tak
`To są koty`	`koty?`	tak

17. Metaznak `*` oznacza, że poprzedni element wzorca może wystąpić w napisie dowolną liczbę razy (w tym zero razy):

napis	*wzorzec*	*dopasowanie?*
`Oto__kot`	`Oto_*kot`	tak
`Otokot`	`Oto_*kot`	tak

18. Metaznak **+** oznacza, że poprzedni element wzorca występuje w napisie co najmniej raz:

napis	*wzorzec*	*dopasowanie?*
Oto__kot	**Oto_+kot**	tak
Otokot	**Oto_+kot**	nie

19. Element w postaci **{n}**, gdzie *n* jest liczbą naturalną, oznacza, że poprzedni element wzorca występuje w napisie dokładnie *n* razy:

napis	*wzorzec*	*dopasowanie?*
12345	**[[:digit:]]{5}**	tak
1234	**[[:digit:]]{5}**	nie

20. Element w postaci **{n,m}**, gdzie *n* i *m* są liczbami naturalnymi oraz $n \leq m$, oznacza, że poprzedni element wzorca występuje w napisie co najmniej *n* razy i nie więcej niż *m* razy:

napis	*wzorzec*	*dopasowanie?*
1	**[[:digit:]]{2,3}**	nie
1234	**[[:digit:]]{2,3}**	tak

21. Element w postaci **{n,}**, gdzie *n* jest liczbą naturalną, oznacza, że poprzedni element wzorca występuje w napisie co najmniej *n* razy:

napis	*wzorzec*	*dopasowanie?*
1	**[[:digit:]]{2,}**	nie
123	**[[:digit:]]{2,}**	tak

22. Metaznak **|** jest operatorem alternatywy i oznacza, że w napisie ma wystąpić jeden z elementów stojących na lewo i prawo od tego operatora:

napis	*wzorzec*	*dopasowanie?*
Ala	**O|Ala**	tak
Ula	**O|Ala**	nie

23. Metaznaki **(** i **)** służą do grupowania elementów wzorca i pełnią rolę podobną do tej, w jakiej używa się ich w wyrażeniach algebraicznych:

napis	*wzorzec*	*dopasowanie?*
tato ma kota	**(ma){2}**	nie
mama ma kota	**(ma){2}**	tak

4.3. Polecenie grep

Niektóre cechy polecenia **grep** zaprezentowano w poprzednim rozdziale, kiedy używano go w charakterze filtra. Teraz przedstawione zostaną jego kolejne, bardziej zaawansowane możliwości. Należy tu zaznaczyć, że historycznie **grep** występował w kilku odmianach różniących się sposobem działania – były to:

- **grep** – wersja podstawowa;
- **egrep** – wersja akceptująca wyrażenia regularne;
- **fgrep** – wersja akceptująca wzorce wyszukiwania podawane jako kolejne wiersze.

Obecnie wszystkie te zadania wykonuje jedno narzędzie (**grep**) uruchamiane z odpowiednimi przełącznikami (na przykład **-E** włącza tryb **egrep**), jednak dla zachowania zgodności ze starymi konwencjami zwykle w systemie istnieją również pozostałe warianty.

Polecenie **grep**, oprócz pracy w potoku, kiedy to analizuje przepływający przez nie strumień znaków, potrafi przeglądać wskazane w argumentach pliki i dokonywać w nich przeszukań z wykorzystaniem wyrażeń regularnych. W trybie domyślnym **grep** wyprowadza na swoje standardowe wyjście wszystkie linie plików wejściowych, które albo zawierają poszukiwany ciąg znaków, albo dają się dopasować do wskazanego wyrażenia regularnego. W takim wariancie **grep** uruchamiany jest w następujący sposób:

grep ***[przełącznik...] wzorzec [plik...]***
gdy wzorzec jest prostym ciągiem znaków;

grep -E ***[przełącznik...] wzorzec [plik...]***
gdy wzorzec jest wyrażeniem regularnym.

Najwygodniejszym sposobem zapisu wzorca jest ujęcie go w apostrofy bądź cudzysłów. W przeciwnym przypadku należy się liczyć z tym, że niektóre z metaznaków wyrażeń regularnych mogą być fałszywie interpretowane przez powłokę i wtedy konieczne będzie ich cytowanie za pomocą znaku `\` (niezależnie od cytowania wymuszanego w wyrażeniach regularnych).

Oprócz przełączników opisanych w poprzednim rozdziale akceptowane są również wymienione poniżej:

`-f` *`plik`*
pobierz wzorce nie z linii poleceń, ale z pliku o wskazanej nazwie;

`-x`
wybieraj tylko te linie, które w całości dopasowują się do wzorca;

`-L`
zamiast domyślnej postaci wyjścia wygeneruj listę nazw plików, z których nie wyprowadzono by żadnej pasującej linii;

`-l`
zamiast domyślnej postaci wyjścia wygeneruj listę nazw plików, z których wyprowadzono by pasujące linie;

`-m` *`num`*
przerwij czytanie pliku wejściowego po znalezieniu *num* dopasowań;

`-o`
wypisuj tylko te części linii, dla których znaleziono dopasowanie (normalnie wypisywane są całe linie);

`-s`
wyłącz komunikaty o błędach w dostępie do przeglądanych plików;

`-b`
przed każdą prezentowaną linią wyprowadź licznik bajtów przedstawiający odległość linii od początku pliku;

`-H`
przed każdą linią wyprowadź nazwę zawierającego ją pliku;

`-h`
nie wyprowadzaj nazw plików (zachowanie domyślne, gdy przeszukiwany jest tylko jeden plik);

`-n`
przed każdą prezentowaną linią wyprowadź licznik linii;

`-a`
przetwarzaj pliki binarne tak, jakby zawierały tekst;

--exclude=*wzorzec_nazwy*
pomijaj pliki, których nazwy pasują do wzorca *wzorzec_nazwy* (*uwaga: to nie jest wyrażenie regularne, ale prosta maska nazwy pliku ze znakami* **?** *i* ∗ *w tradycyjnym znaczeniu*);
--exclude-dir=*wzorzec_nazwy*
pomijaj katalogi, których nazwy pasują do wzorca *wzorzec_nazwy* (*uwaga jak powyżej*);
-r
przetwarzaj katalogi rekursywnie; honoruj linki symboliczne tylko wtedy, gdy zostały jawnie wymienione w linii poleceń;
-R
przetwarzaj katalogi rekursywnie; honoruj wszystkie linki symboliczne;
-w
szukaj tylko linii, w których wzorzec dopasowuje się do całych słów;
-v
wyprowadzaj tylko te linie, które **nie pasują** do wzorca.

4.4. Przykłady użycia narzędzia grep

W zaprezentowanych poniżej przykładach założono istnienie plików o nazwach **test1** i **test2** oraz o następującej zawartości:

```
SAME WIELKIE LITERY
same małe litery
Linia Wielbłądzia Czyli Kapitaliki


powyżej są 2 puste linie
ostatnia linia
```

Poniżej podano kilka przykładów użycia polecenia **grep**, prezentując kompletne polecenia i dane wyprowadzane na standardowe wyjście w wyniku działania tego polecenia:

– **grep "litery" test1**

```
same małe litery
```

– **grep "li" test1**

```
same małe litery
Linia Wielbłądzia Czyli Kapitaliki
powyżej są 2 puste linie
ostatnia linia
```

– **grep "litery" test*

```
test1:same małe litery
test2:same małe litery
```

– **grep -i "litery" test1**

```
SAME WIELKIE LITERY
same małe litery
```

– **grep -E "lini." test1**

```
powyżej są 2 puste linie
ostatnia linia
```

– **grep -E "[[:digit:]]" test1**

```
powyżej są 2 puste linie
```

– **grep -En "^$" test1**

```
4:
5:
```

– **grep -v -E "^$" test1**

```
SAME WIELKIE LITERY
same małe litery
Linia Wielbłądzia Czyli Kapitaliki
powyżej są 2 puste linie
ostatnia linia
```

– **grep -c -v -E "^$" test1**

```
5
```

– **grep -l -i "li" test***

```
test1
test2
```

– **grep -b -i "li" test***

```
test1:0:SAME WIELKIE LITERY
test1:20:same małe litery
test1:38:Linia Wielbłądzia Czyli Kapitaliki
test1:77:powyżej są 2 puste linie
test1:104:ostatnia linia
test2:0:SAME WIELKIE LITERY
test2:20:same małe litery
test2:38:Linia Wielbłądzia Czyli Kapitaliki
test2:77:powyżej są 2 puste linie
test2:104:ostatnia linia
```

4.5. Polecenie find

Polecenie **find** służy do wyszukiwania plików, których metadane spełniają określone kryteria. Składnia polecenia prezentuje się następująco:

find [-H] [-L] [-P] ***[katalog...] [wyrażenie]***

Polecenie **find** będzie szukać plików o cechach określonych przez *wyrażenie*, a proces szukania będzie kolejno rozpoczynany w każdym z wymienionych *katalogów*.

Pierwsze trzy opcje, z których jednocześnie może wystąpić co najwyżej jedna (jeśli została podana, musi poprzedzać jakiekolwiek inne elementy linii poleceń), determinują sposób, w jaki **find** traktuje dowiązania symboliczne. Są to:

- **-H** ignoruj dowiązania symboliczne (zachowanie domyślne);
- **-L** honoruj dowiązania symboliczne, traktując je tak, jakby były obiektami, na które wskazują;
- **-P** honoruj dowiązanie symboliczne tylko wtedy, gdy zostały jawnie wymienione w linii poleceń.

Wyrażenie jest kombinacją warunków domyślnie połączonych spójnikiem „i”. Polecenie **find** umożliwia określenie innego sposobu wiązania warunków za pomocą nawiasów okrągłych i jawnych operatorów logicznych. Po znalezieniu pliku spełniającego podane warunki **find** wypisuje jego nazwę na swoje standardowe

wyjście. Dodatkowo w takiej sytuacji **find** może wykonać pewną akcję (na przykład uruchomić wskazane polecenie). Argumenty numeryczne będące składnikami warunków zapisuje się w sposób następujący:

+*n* większe od *n*;
-*n* mniejsze od *n*;
n równe *n*.

Warunki, które odwołują się do metadanych pliku określających czas, mają przedrostek:

a gdy chodzi o *atime* (czas dostępu);
c gdy chodzi o *ctime* (czas utworzenia);
m gdy chodzi o *mtime* (czas modyfikacji).

Poniżej zaprezentowano wybrane warunki sprawdzane przez polecenie **find**:

-min *n* (gdzie *min* to jedno z: **amin**, **cmin**, **mmin**)
czas *min* jest odległy o *n* minut od bieżącego;
-newer *nazwa_pliku* (gdzie *newer* to jedno z: **anewer**, **cnewer**, **mnewer**)
szukany jest plik nowszy niż plik o nazwie *nazwa_pliku*;
-time *n* (gdzie *time* to jedno z: **atime**, **ctime**, **mtime**)
czas *time* jest odległy od bieżącego o $n \cdot 24$ godziny (stosuje się zaokrąglenie w górę, które oznacza, że na przykład **-atime +1** opisuje plik, który był użyty przed dwoma dniami albo wcześniej);
-empty
szukany plik/katalog ma być pusty;
-executable
szukany plik ma być wykonywalny lub katalog ma być „przeglądalny”;
-gid *n*
plik ma należeć do grupy o numerze *n*;
-group *g*
plik ma należeć do grupy o nazwie *g*;
-name *nazwa*
nazwa odnalezionego pliku ma pasować do wzorca *nazwa* (wielkość liter **ma znaczenie**);
-iname *nazwa*
nazwa odnalezionego pliku ma pasować do wzorca *nazwa* (wielkość liter **nie ma znaczenia**);
-perm *prawa_pliku*
odnaleziony plik ma mieć prawa jak określone ósemkowo w *prawa_pliku*;
-readable
szukany plik ma być odczytywalny;

-size *n*[cwbkMG]
odnaleziony plik ma mieć rozmiar n jednostek, gdzie:
- **b** blok 512-bajtowy;
- **c** bajt;
- **w** słowo (2 bajty);
- **k** kibibajt;
- **M** mebibaft;
- **G** gibibajt.

-type *t*
odnaleziony plik ma być:
- **d** katalogiem;
- **f** zwykłym plikiem;
- **l** dowiązaniem symbolicznym.

-uid *n*
plik ma należeć do użytkownika o numerze *n*;
-user *u*
plik ma należeć do użytkownika o nazwie *u*;
-writable
plik ma być zapisywalny.

W tym miejscu nieodzownym staje się uzupełnienie powyższego opisu o pewien szczegół dotyczący operatora **-size** użytego w połączeniu z rozmiarem poprzedzonym znakiem minus, a więc sprawdzającego, czy rozmiar pliku jest mniejszy niż żądany. Otóż z powodów znanych tylko i wyłącznie autorom polecenia **find** narzędzie to w czasie pracy zaokrągla rozmiary badanych plików w górę do takiej jednostki, która została użyta w operatorze. W efekcie rezultaty wyszukiwania w sposób absolutnie zaskakujący i kompletnie absurdalny (*sic!*) odbiegają od spodziewanych.

Założono, że zawartość pewnego katalogu prezentuje się następująco:

```
$ ls -l
-rw-r--r-- 1 user user    0 09-21 15:37 0
-rw-r--r-- 1 user user 1024 09-21 15:38 1024
-rw-r--r-- 1 user user    8 09-21 15:38 8
```

Dla ułatwienia analizy przykładu przyjęto, że nazwy plików są tożsame z ich rozmiarem w bajtach. W tak przygotowanym środowisku podjęto następnie próbę znalezienia plików mniejszych od 1 KiB.

```
$ find -size -1k
./0
```

Jak widać, **find** odnalazł plik **0** (poprawnie), ale nie odnalazł pliku **8**. Czemu? Wyjaśnienie tego faktu brzmi schizofrenicznie, ale jest prawdziwe:

– do wyszukiwania użyto jednostki **k**, więc rozmiary badanych plików będą zokrąglane do 1KiB,
– rozmiar wyrażony jako 0 bajtów i zokrąglony do 1KiB to nadal 0, więc plik zostaje uwzględniony w wynikach,
– rozmiar wyrażony jako 8 bajtów i zokrąglony do 1KiB to 1KiB, a ponieważ 1KiB nie jest mniejszy od 1Kib, to plik zostaje pominięty w wynikach (*sic!*).

Oznacza to, że w pełni wiarygodne wyniki poszukiwania plików mniejszych niż pewna wartość są osiągalne tylko i wyłącznie wtedy, gdy użytkownik użyje jednostki **c** (bajty), kiedy opisane wcześniej szaleńcze „zaokrąglanie" nie jest w stanie przynieść szkód. Po takim działaniu uzyskano następujący rezultat:

```
$ find -size -1024c
./0
./8
```

I dopiero teraz **find** prezentuje oczekiwany i poprawny wynik.

Dodatkowo definiuje się pewną liczbę pseudowarunków, których zadaniem nie jest badanie cech pliku, ale wykonanie specyficznej akcji w chwili, gdy plik/katalog zostanie odnaleziony. Poniżej podano dwa z nich:

-exec *polecenie* \;
wykonuje dowolne *polecenie* w miejscu odnalezienia pliku; traktowany jest jako spełniony, gdy polecenie kończy się poprawnie (zwraca kod powrotu 0); w *poleceniu* można używać symbolu **{}**, oznaczającego nazwę odnalezionego pliku/katalogu;

-print
traktowany tak, jakby zawsze był spełniony; powoduje wypisanie nazwy odnalezionego pliku do standardowego wyjścia; jest używany, gdy na skutek użycia innych warunków (na przykład **-exec**) domyślne wypisywanie nazw jest wyłączone.

Powyższe warunki można dowolnie wiązać poniższymi operatorami (wymienionymi w kolejności malejących priorytetów):

(*wyrażenie*)
nawiasy wymuszają zmianę wiązania operatorów w tradycyjny, matematyczny sposób; ponieważ powłoka może używać ich do swoich celów, warto przygotować się na konieczność użycia cytowania **\(** i **\)**;

`! wyrażenie`
negacja – tu również może być konieczne użycie cytowania;

`-not wyrażenie`
jak wyżej, ale wygodniej, bo bez cytowania;

`wyrażenie1 wyrażenie2`
domyślne złączenie wyrażeń operatorem „i”;

`wyrażenie1 -a wyrażenie2`
jawne złączenie wyrażeń operatorem „i” (**`-a`** jak *and*);

`wyrażenie1 -o wyrażenie2`
jawne złączenie wyrażeń operatorem „lub” (**`-o`** jak *or*).

Przykłady użycia polecenia **`find`**:

`find ~-name abc.txt`
wyszukuje wszystkie pliki/katalogi o nazwie *abc.txt*, które mogą znajdować się w katalogu domowym użytkownika oraz w jego podkatalogach;

`find ~/temp -name "*.txt"`
wyszukuje wszystkie pliki/katalogi o nazwie z przyrostkiem **`.txt`**, które znajdują się w podkatalogu **`temp`** katalogu domowego użytkownika oraz ewentualnych podkatalogach tego katalogu;

`find ~ -iname "*.txt" -type f -size +10M`
wyszukuje wszystkie pliki zwykłe w katalogu domowym użytkownika (i jego podkatalogach), których nazwy mają przyrostek **`.txt`** (wielkość liter bez znaczenia) oraz rozmiar większy niż 10 mebibajtów;

`find /tmp -type f -atime +2 -exec rm {} \; -print`
wyszukuje w katalogu **`/tmp`** oraz jego podkatalogach wszystkie pliki zwykłe, których nie używano w żaden sposób w ciągu ostatnich 48 godzin; wszystkie odnalezione pliki zostaną usunięte; dodatkowy przełącznik **`-print`** spowoduje, że zostaną wyświetlone nazwy odnalezionych plików, pomimo wykonania na nich dodatkowej operacji (tutaj **`rm`**).

4.6. Źródła uzupełniające

1. Manual polecenia **`grep`**: `https://man7.org/linux/man-pages/man1/grep.1.html`
2. Opis implementacji wyrażeń regularnych w języku Perl: `https://perldoc.perl.org/perlre`

3. Opis implementacji wyrażeń regularnych w języku Python: `https://docs.python.org/3/howto/regex.html`
4. Opis implementacji wyrażeń regularnych w bibliotece Gnulib: `https://www.gnu.org/software/gnulib/manual/html_node/Regular-expressions.html`
5. Manual polecenia **`find`**: `https://man7.org/linux/man-pages/man1/find.1.html`

4.7. Zadania do samodzielnego wykonania

Wskazówka: do testowania swoich wyrażeń regularnych możesz wykorzystać taki oto prosty potok:

```
echo napis | grep -E 'wzorzec'
```

Dzięki niemu będziesz mógł na bieżąco obserwować zachowanie twoich wyrażeń i sposób działania polecenia ***`grep`***.

1. W jaki sposób można umieścić znaki **`[`** i **`]`** wewnątrz zbioru?
2. Czy twój system operacyjny jest zlokalizowany w sposób, który sprawia, że litera „ą” jest elementem zbioru **`[a-c]`**?
3. Jakie znaki w napisie mogą dopasować się do klasy **`[:cntrl:]`**?
4. Korzystając z podręcznika systemowego, zbadaj sposób użycia narzędzia **`fgrep`**.
5. Od jakiej wartości startują liczniki prezentowane przez przełączniki **`-n`** i **`-b`** polecenia **`grep`**?
6. Zbadaj działanie przełączników **`-A`** i **`-B`** polecenia **`grep`**.
7. Skonstruuj wyrażenia regularne sprawdzające, czy plik zawiera:
   - numery PESEL;
   - polskie kody pocztowe;
   - adresy IP;
   - adresy e-mail;
   - numer konta bankowego w standardzie IBAN ze spacjami rozdzielającymi lub bez nich;
   - datę w formacie ISO 8601;
   - puste linie;
   - linie niezawierające cyfr;
   - linie potencjalnie zawierające imiona i nazwiska (dwa słowa obok siebie, rozdzielone białymi znakami, każde zaczynające się wielką literą);
   - znaki końca linii w standardzie systemu MS Windows;
   - linie zaczynające się od liczb;
   - linie niekończące się liczbami;
   - linie zawierające dokładnie jedną liczbę.
8. Zbadaj działanie warunków **`-maxdepth`** i **`-mindepth`** polecenia **`find`**.

9. Skonstruuj linię poleceń uruchamiającą narzędzie **`find`** i odszukującą pliki o następujących cechach:
   - nieużywane od 2 lat;
   - mniejsze od 1KiB i większe od 100B;
   - większe od 1KiB lub mniejsze od 100B;
   - takie, których właścicielem jest ktoś inny niż ty;
   - wykonywalne w systemie MS Windows;
   - większe niż 100MiB i użyte w ciągu ostatniego miesiąca;
   - uruchamiane w ciągu ostatniego tygodnia.
10. Powiąż ze sobą polecenia **`find`** i **`grep`** używając warunku **`-exec`** i symbolu **`{}`** w celu znalezienia plików:
    - których nazwy kończą się przyrostkiem **`txt`** i które zawierają adresy e-mail;
    - które są mniejsze niż 10KB i które sprawiają wrażenie, że zawierają kod źródłowy w języku C/C++ (*wskazówka: co zwykle zawierają pierwsze linie pliku źródłowego w tych językach?*).

5. Procesy

5.1. Proces

Proces to **uruchomiony program**. Każdemu procesowi obecnemu w systemie towarzyszy zestaw opisujących go atrybutów. W przypadku systemów klasy Unix/Linux są to:

- identyfikator procesu (**PID** – od ang. *process identifier*) – unikalna w danym cyklu życia systemu operacyjnego dodatnia liczba całkowita, jednoznacznie identyfikująca proces,
- identyfikator procesu rodzicielskiego (**PPID** – od ang. *parental PID*) – wartość PID tego procesu, który powołał do życia dany proces (w przeciwieństwie do świata realnego do wykreowania procesu potomnego potrzebny jest tylko jeden rodzic),
- środowisko procesu – zbiór zmiennych środowiskowych przechowywany w systemie oddzielnie dla każdego procesu; środowisko to powstaje poprzez skopiowanie środowiska procesu rodzicielskiego w chwili tworzenia procesu potomnego; zmiany wykonane przez proces potomny w swoim egzemplarzu środowiska nie są widoczne w procesie macierzystym,
- właściciel – użytkownik, identyfikowany przez swój UID, który uruchomił (lub w którego imieniu uruchomiono) dany proces; przechowywanie tej danej

pozwala na wprowadzenie mechanizmów ochrony, dzięki którym nieuprawniony użytkownik nie będzie w stanie wpływać na stan procesów innych niż własne,
- priorytet – wartość reprezentująca miejsce zajmowane w hierarchii decydującej o sposobie, w jaki proces jest traktowany przez system; jest liczbą z przedziału od –20 (najwyższy priorytet) do 19 (najniższy priorytet); procesy o wyższym priorytecie mogą być preferowane w dostępie do zasobów systemu, w szczególności do przydziału procesora; nowsze jądra systemu Linux mogą również dopuszczać specjalną wartość priorytetu oznaczaną jaką **rt** (od ang. *real-time*) – procesy o takim priorytecie mają absolutne pierwszeństwo wykonania.

Fakt, iż każdy proces powiązany jest ze swoim rodzicem, powoduje w konsekwencji, że wszystkie obecne w systemie procesy można utożsamiać z węzłami drzewa, zaś korzeniem tego drzewa jest proces, który został wystartowany przez jądro jako pierwszy. W starszych wydaniach systemu Linux procesem tym jest *init*, w nowszych – *systemd*, co ma związek z dokonaną gruntowną zmianą modelu zarządzania uruchamianiem i zamykaniem usług w systemie operacyjnym. Proces uruchomiony jako pierwszy ma zawsze PID równy 1, a jego PPID wynosi 0, co oznacza, że *init* i *systemd* są *de facto* sierotami i tym samym są przodkami wszystkich pozostałych procesów.

Historyczna nazwa procesu *init* jest nieodłącznie powiązana z mechanizmem uruchamiania systemu, nazywanym tradycyjnie *sysvinit* (od ang. *System V Init*, co jest nawiązaniem do Uniksa wersji V, w którym mechanizm ten został zaimplementowany po raz pierwszy). W większości współczesnych dystrybucji Linuksa standard ten został zastąpiony przez nowszy i uniwersalniejszy mechanizm pod nazwą *systemd*.

Wiele z procesów jest uruchamianych automatycznie w czasie startu systemu. Zwykle ich właścicielem jest użytkownik *root*, co zapewnia, że procesy żywotne dla funkcjonowania systemu nie zostaną zatrzymane przez zwykłych użytkowników. Od momentu zalogowania się użytkownika do systemu kolejne uruchamiane przez niego procesy będą już należeć do niego i tym samym będzie on mógł decydować o ich losie.

5.2. Prezentowanie stanu procesów

5.2.1. Polecenie ps

Polecenie **ps** (od ang. *process status*) wyprowadza na standardowe wyjście listę procesów obecnych w systemie w momencie pracy polecenia (jest to tak zwana *migawka stanu procesów*). Lista prezentowana jest jako tabela, w której kolumnach prezentowane są różne właściwości każdego procesu, na przykład:

```
$ ps
  PID TTY          TIME CMD
 1046 pts/1    00:00:00 zsh
 1601 pts/1    00:00:00 ps
```

Ze względu na skomplikowaną historię rozwoju polecenia **ps** zestaw dopuszczalnych przełączników jest dość pogmatwany. Można uznać, że obecnie implementowane wersje **ps** używają równolegle co najmniej trzech różnych form specyfikowania przełączników – są to:

– *styl UNIX* – przełączniki mogą być grupowane i są poprzedzane łącznikiem, na przykład:
ps -ef
– *styl BSD* – przełączniki mogą być grupowane i nie są poprzedzane łącznikiem, na przykład:
ps aux
– *styl GNU* – przełączniki nie mogą być grupowane i są poprzedzane podwójnym łącznikiem, na przykład:
ps --headers

W dalszej części opisu posłużono się wyłącznie przełącznikami w stylu UNIX. Oto najprzydatniejsze z nich:

-e
wyprowadź informację o wszystkich procesach w systemie (domyślnie prezentuje się tylko procesy tego użytkownika, który uruchomił polecenie **ps**),
-p *n*
wyprowadź informację tylko o procesie z PID równym *n*,
-l
forma długa, prezentująca bogatszy zestaw informacji o procesach,
-u *lista*
wyświetla tylko procesy użytkowników wymienionych na liście *lista*, na przykład: **-u root,user**,
-f
pełna postać zestawienia danych o procesach,
-F
postać danych obszerniejsza niż uzyskiwana z **-f**.

Repertuar prezentowany danych zależy od użytych przełączników. Poniżej przedstawiono skrótowe tytuły kolumn w formach używanych przez **ps**:

– **%CPU**
względne użycie czasu procesora obliczane jako procentowy iloraz faktycznego czasu użycia procesora do czasu przebywania procesu w systemie,

- **%MEM**
  względne użycie pamięci operacyjnej obliczane jako procentowy iloraz rozmiaru pamięci zajętej przez proces do rozmiaru pamięci dostępnej w systemie,
- **COMMAND**
  linia poleceń (wraz z argumentami), którą uruchomiono dany proces,
- **START**
  czas uruchomienia procesu,
- **TIME**
  czas procesora wykorzystany przez proces,
- **C**
  procentowe wykorzystanie procesora (wartość całkowita z **%CPU**),
- **NI**
  wartość *niceness* procesu (pojęcie to zostanie omówione niebawem),
- **F**
  różne opcje procesu (od ang. *flags*),
- **PID**
  nie wymaga wyjaśnień,
- **PPID**
  jak wyżej,
- **PRI**
  priorytet procesu,
- **RSS**
  (od ang. *resident set size*) rozmiar pamięci procesu stale rezydującej w systemie (tzn. niepodlegającej wymianie w ramach pracy pamięci wirtualnej),
- **S**
  status procesu,
- **SZ**
  rozmiar całkowity pamięci użytej przez proces (wraz z pamięcią wirtualną),
- **TTY**
  terminal skojarzony z procesem (najczęściej ten, na którym zalogowany jest właściciel procesu),
- **UID**
  identyfikator właściciela procesu,
- **USER**
  nazwa właściciela procesu,
- **WCHAN**
  zdarzenie, na którego zakończenie czeka proces (jeśli jest w stanie czekania); może to być nazwa usługi systemowej, wewnątrz której czeka proces.

5.2.2. Polecenie `pstree`

Polecenie **`pstree`** wyprowadza na swoje standardowe wyjście graficzne zobrazowanie drzewa procesów obecnych w systemie, na przykład:

```
$ pstree
init─┬─atd
     ├─avahi-daemon───avahi-daemon
     ├─colord───{colord}
     ├─colord-sane───2*[{colord-sane}]
     ├─console-kit-dae───64*[{console-kit-dae}]
     ├─cron
     ├─cupsd
     ├─dbus-daemon
     ├─dnsmasq
     ├─6*[getty]
     ├─lighttpd
     ├─master─┬─pickup
     │        └─qmgr
     ├─nmbd
     ├─ntpd
     ├─polkitd───{polkitd}
     ├─rpc.idmapd
     ├─rpc.statd
     ├─rpcbind
     ├─rsyslogd───3*[{rsyslogd}]
     ├─smbd───3*[smbd]
     ├─sshd───sshd───bash───su───bash───pstree
     ├─udevd───2*[udevd]
     └─winbindd───3*[winbindd]
```

Rys. 5.1. Przykładowe drzewo procesów tworzone przez **`pstree`**

Należy zauważyć, że **`pstree`** stara się skomasować obraz drzewa, jeśli w jego strukturze pojawiają się kolejno identyczne gałęzie, co sygnalizowane jest pojawieniem się znacznika **`n*`** (na przykład **`3*[smbd]`**).

Polecenie **`pstree`** uruchamia się w sposób następujący:

`pstree` ***`[przełączniki...] [pid | użytkownik]`***

gdzie:

– ***`pid`***
  PID procesu, który ma zostać przedstawiony jako korzeń drzewa (domyślnie jest to proces o PID równym 1);

– ***`użytkownik`***
  zaprezentowanie tylko tych procesów, których właścicielem jest wskazany użytkownik.

Poniżej przedstawiono niektóre z przełączników polecenia **pstree**:

-a prezentuje linię poleceń każdego procesu,

-c wyłącza domyślne skracanie obrazu drzewa,

-l wyprowadza długie wiersze nawet wtedy, gdy zaburza obraz drzewa (przydatne przy wyprowadzaniu wyników do pliku),

-n sortuje potomków w kolejności PID (domyślnie sortowanie przebiega według nazw),

-p włącza prezentowanie numerów PID.

5.2.3. Polecenie top

Program **top** jest prostym interaktywnym zarządcą procesów, cyklicznie wyprowadzającym na terminal listę danych o obecnych w systemie procesach (domyślnie aktualizacja przeprowadzana jest co 5 sekund). Zachowanie narzędzia może być kontrolowane przy starcie poprzez specyfikowanie pożądanego zestawu przełączników oraz na bieżąco w czasie pracy poprzez polecenia wydawane przez naciskanie wybranych klawiszy. Przykładowy ekran narzędzia top prezentuje się jak poniżej (pokazano tylko kilka początkowych linii zestawienia):

```
Tasks:  87 total,   1 running,  86 sleeping,   0 stopped,   0 zombie
%Cpu(s):  1,0 us,  2,9 sy,  0,0 ni, 95,8 id,  0,0 wa,  0,0 hi,  0,3 si,  0,0 st
KiB Mem:    496600 total,   436192 used,    60408 free,   165372 buffers
KiB Swap:   499996 total,     2992 used,   497004 free,   214788 cached

  PID USER      PR  NI  VIRT  RES  SHR S  %CPU %MEM    TIME+  COMMAND
21108 user      20   0  4636 1324 1028 R   1,9  0,3   0:00.14 top
20889 root      20   0 11436 3828 3068 S   1,0  0,8   0:01.18 sshd
 1 root     20   0  2144  588  556 S   0,0  0,1   0:40.68 init
 2 root     20   0     0    0    0 S   0,0  0,0   0:00.09 kthreadd
 3 root     20   0     0    0    0 S   0,0  0,0   2:57.35 ksoftirqd/0
```

Warty pokreślenia jest fakt, że **top** używa tych samych nagłówków kolumn, co **ps**. Wyjście z programu **top** następuje po naciśnięciu klawisza *q*.

Poniżej przedstawiono niektóre z przełączników narzędzia **top**:

-d *n*

odświeża zestawienie co *n* sekund;

-p *pid*

prezentuje tylko procesy o podanych identyfikatorach PID (przełącznik może być użyty maksymalnie 20 razy);

-s

włącza tak zwany *tryb bezpieczny*, w którym wyłączone są polecenia potencjalnie ryzykowne;

-c

prezentuje pełne linie poleceń zamiast ich nazw;

`-n n`
odświeża zestawienie *n* razy i kończy pracę;
`-b`
włącza tryb wsadowy (w trybie tym **`top`** nie reaguje na polecenie z klawiatury oraz wyprowadza zestawienia w postaci czystego tekstu bez znaków sterowania terminalem, co pozwala umieszczać prezentowane dane wprost w pliku).

Oprócz danych znanych z zestawienia prezentowanego przez **`ps top`** uwidacznia również między innymi następujące informacje:

– **`uptime`**
łączny czas pracy systemu od chwili jego ostatniego włączenia oraz trzy miary obciążenia systemu określające średnią liczbę procesów gotowych do uruchomienia w ostatniej minucie, ostatnich 5 minutach i w ostatnich 15 minutach;
– **`processes`**
ogólna liczba procesów w systemie rozbita na te procesy, które:
  • wykonują się (ang. *running*),
  • śpią (ang. *sleeping*),
  • są zatrzymane (ang. stopped),
  • są w stanie *zombie* (są to tak zwane *procesy niemartwe*, tzn. procesy, które zakończyły działanie, lecz z różnych przyczyn nie zostały usunięte z systemu).
– **`CPU states`**
zobrazowanie zajętości CPU w trybie:
  • użytkownika (us od ang. *user*),
  • systemowym (sy od ang. *system*),
  • zadań z ujemną wartością *nice* (ni),
  • bezczynności (id od ang. *idle*),
  • czekania na zdarzenie (wa od ang. *wait*),
  • obsługi przerwań sprzętowych (hi od ang. *hardware interrupt*),
  • obsługi przerwań programowych (si od ang. *software interrupt*),
  • czasu skradzionego (st od ang. *stolen* – parametr ma znaczenie, gdy system pracuje w maszynie wirtualnej, kiedy to może mieć miejsce wywłaszczanie procesu maszyny wirtualnej przez system operacyjny gospodarza (jest to tak zwana *kradzież czasu*).
– **`Mem`**
dane o zajętości pamięci:
  • **`total`** – całkowita pamięć zainstalowana w systemie,
  • **`used`** – pamięć zajęta,

  - **free** – dostępna pamięć,
  - **buffers** – pamięć używana przez bufory.

– **Swap**
  dane opisujące użycie obszaru wymiany zarządzanego przez systemowy mechanizm pamięci wirtualnej (zakres prezentowanych danych jest niemal identyczny, jak w dziale **Mem**).

W czasie pracy interaktywnej **top** akceptuje między innymi następujące polecenia wydawane z klawiatury:

– **<spacja>**
  odśwież ekran natychmiast,
– **k**
  zabij proces (wymaga podania numeru PID i numeru sygnału przesłanego do procesu – szczegóły znajdują się w dalszej części opisu),
– **n**
  zmień liczbę prezentowanych procesów (wymaga podania żądanej liczby procesów),
– **r**
  zmień wartość *niceness* dla procesu (wymaga podania numeru PID i wartości *niceness*),
– **l**
  włącz/wyłącz informacje o obciążeniu,
– **m**
  włącz/wyłącz informacje o pamięci,
– **t**
  włącz/wyłącz informacje o procesach,
– **c**
  włącz/wyłącz wyświetlanie linii poleceń,
– **A**
  sortuj procesy według czasu przebywania w systemie,
– **M**
  sortuj procesy według zużycia pamięci,
– **N**
  sortuj procesy według numerów PID,
– **P**
  sortuj procesy według obciążeń CPU,
– **T**
  sortuj procesy według czasów użycia CPU,

– **W**
zapisz konfigurację do pliku ∼**/.toprc**; konfiguracja ta zostanie użyta jako domyślna po następnym uruchomieniu programu **top**.

5.2.4. Polecenie `time`

Polecenie **time** pozwala oszacować czas wykonania wskazanego programu, co osiąga się poprzez pomiar trzech niezależnych czasów:

real bezwzględny czas przebywania procesu w systemie, liczony od uruchomienia do zakończenia i określany często angielskim terminem *wall-clock time*,

user łączny czas procesora zużyty przez proces i spędzony w trybie użytkownika,

sys łączny czas procesora zużyty przez proces i spędzony w trybie systemu.

Uwaga: zmierzone czasy są wyprowadzane na **stderr***, co pozwala na ich łatwe oddzielenie od strumienia* **stdout** *monitorowanego procesu.*

Polecenie time uruchamia się w sposób następujący:

```
time [przełączniki...] polecenie [argument_polecenia...]
```

Na przykład w poniższy sposób można oszacować obciążenie procesora i systemu operacyjnego przez wykonanie kompilacji wybranego kodu źródłowego:

```
$ time gcc prog.c
real    0m0.706s
user    0m0.540s
sys     0m0.150s
```

5.3. Usuwanie procesów

5.3.1. Sygnały

Właściciel procesu lub użytkownik *root* ma możliwość usunięcia procesu z systemu. Zadanie to wykonuje polecenie o sugestywnej nazwie **kill**, chociaż należy zaznaczyć, że rola tego narzędzia nie ogranicza się tylko do działań eksterminacyjnych. Jego podstawowym zadaniem jest wysyłanie do procesu/procesów tak zwanych *sygnałów* (ang. *signals*), uruchamiających asynchronicznie krótkie sekwencje kodu, przeznaczone do reagowania na przeróżne sytuacje, które mają znaczenie dla życia procesu.

Na potrzeby niniejszego opracowania poprzestano na stwierdzeniu, że każdy z pracujących procesów może zostać poinformowany wyspecjalizowanym sygnałem

o zaistnieniu jednego z predefiniowanych zdarzeń. Odebranie takiej informacji powinno być powiązane z jej obsłużeniem.

Trzy z sygnałów mają ścisły związek z wymuszonym zakończeniem działania procesu – są to:

- **SIGTERM** (sygnał nr 15) – proces, do którego wysłano sygnał *SIGTERM*, powinien obsłużyć go na końcu pracy, jednak sygnał ten może zostać przez proces zignorowany, co z kolei może sprawić, że będzie on bezskuteczny; obsłużenie tego sygnału daje procesowi szansę zakończenia działania w sposób kontrolowany i bezpieczny;
- **SIGINT** (sygnał nr 2) – sygnał, który wysyłany jest do procesu w reakcji na użycie kombinacji klawiszy *Ctrl-C*; może zostać zignorowany, a jego obsłużenie pozwala na kontrolowane zakończenie pracy procesu;
- **SIGKILL** (sygnał nr 9) – proces, do którego wysłano sygnał *SIGKILL*, bezzwłocznie kończy pracę; zignorowanie tego sygnału nie jest możliwe, ale nie jest też możliwe wykonanie czynności gwarantujących bezpieczne zakończenie.

5.3.2. Polecenie `kill`

Polecenie to wydaje się w następujący sposób:

```
kill [ -nazwa_lub_numer_sygnału ] [ przełącznik... ] PID...
```

Użycie polecenia **kill** powoduje wysłanie sygnału o podanym numerze bądź o podanej nazwie do procesów o wymienionych numerach PID. Pominięcie specyfikacji sygnału spowoduje wysłanie sygnału *SIGTERM*.

Obie poniższe formy polecenia **kill** są równoważne:

```
kill -9 2222
kill -SIGKILL 2222
```

Pracujący proces można również przerwać z użyciem kombinacji klawiszy *Ctrl-C*, jednak w pewnych wyjątkowych sytuacjach (na przykład wtedy, gdy działanie procesu jest zakłócone przez błędy w kodzie) *SIGTERM* i *SIGINT* mogą być bezskuteczne i wtedy nieodzowne staje się użycie sygnału *SIGKILL*.

5.3.3. Polecenie `killall`

Ponieważ posługiwanie się numerami PID bywa kłopotliwe i, ze względu na łatwość popełnienia błędu, potencjalnie niebezpieczne, w pewnych sytuacjach korzystniejsze jest używanie nazw programów (**kill** przed wysłaniem sygnału nie pyta użytkownika o potwierdzenie swojej decyzji, wskutek czego omyłkowe skierowanie sygnału do niewłaściwego procesu może spowodować prawdziwą katastrofę, szczególnie, gdy ma się uprawnienia użytkownika *root*).

Zatrzymanie wszystkich procesów pochodzących z programu o podanej nazwie powoduje polecenie **killall**; wydawane w sposób następujący:

killall *[przełącznik...] nazwa_programu...*

spowoduje wysłanie sygnału *SIGKILL* do wszystkich procesów powstałych wskutek uruchomienia polecenia **grep**.

Najczęściej używane przełączniki to:

-I
nazwy programów brane są bez uwzględniania różnic w wielkości liter;

-y *time*
wysyła sygnał tylko do procesów młodszych niż *time*; time podawany jest jako liczba rzeczywista z przyrostkiem:

- **s** sekundy,
- **m** minuty,
- **h** godziny,
- **d** dni,
- **w** tygodnie,
- **M** miesiące,
- **y** lata.

Dygresja: autor niniejszego tekstu powątpiewa w to, aby ostatni z podanych przyrostków został kiedykolwiek sensownie użyty, ale składa wyrazy uznania dla wyobraźni autorów polecenia **killall**.

-o *time*
wysłanie sygnału tylko do procesów starszych niż *time*;

-i
pytanie o potwierdzenie przed wysłaniem sygnału do każdego z żądanych procesów;

-r *name*
potraktowanie nazwy *name* jako wyrażenia regularnego;

-s *sig*
wysłanie sygnału *sig* zamiast domyślnego *SIGTERM*;

-u *user*
wysłanie sygnału tylko do procesów użytkownika *user*.

5.4. Priorytety procesów

5.4.1. *Niceness*

Bieżąca wartość priorytetu procesu w systemach klasy Unix/Linux jest często określana angielskim terminem *niceness*, co na język polski można spróbować przetłumaczyć jako *uprzejmość*. Domyślnie nowo utworzony proces dziedziczy

wartość *niceness* po swoim rodzicu i w większości sytuacji jest to zero. W systemie istnieją dwa standardowe narzędzia pozwalające manipulować tą wartością, jednak użytkownik inny niż *root* jest w tym działaniu bardzo ograniczony – pozwala mu się jedynie na **pogarszanie** priorytetu swoich procesów (a więc może on jedynie **powiększać** wartość *niceness*). Użytkownik *root* może zmieniać *niceness* w dowolny sposób.

Taka niecodzienna nomenklatura powstała jako efekt spostrzeżenia, że proces o gorszym priorytecie jest **uprzejmiejszy** (ang. *nicer*) dla systemu operacyjnego.

5.4.2. Polecenie `nice`

Polecenie **`nice`** służy do uruchomienia procesu z priorytetem innym niż domyślny i ma następującą postać:

```
nice -n niceness polecenie [ argument... ]
```

Użycie **`nice`** spowoduje uruchomienie polecenia *polecenie* z wartością *niceness* podaną za przełącznikiem **`-n`**. Dla zwykłych użytkowników dozwolone jest jedynie użycie wartości dodatnich, na przykład:

```
nice -n 10 kompilacja_cyber_punka
```

spowoduje uruchomienie (zapewne bardzo czasochłonnego) skryptu kompilującego złożony projekt z wartością *niceness* równą 10. Taki sposób wystartowania procesu gwarantuje, że nie będzie on obciążał nadmiernie zasobów systemu, choć zostanie to najprawdopodobniej okupione wydłużeniem czasu jego pracy.

5.4.3. Polecenie `renice`

Wartość *niceness* procesu już uruchomionego może zostać zmieniona poprzez wydanie polecenia **`renice`** i, tak jak poprzednio, użytkownik inny niż *root* może to uczynić tylko wtedy, gdy godzi się na pogorszenie priorytetu swojego zadania (podwyższenie wartości *niceness*). Dokonuje się tego poleceniem o poniższej postaci (wymieniono najprostszą z form):

```
renice niceness pid...
```

przy czym *niceness* traktuje się tu nie jako przyrost, ale jako nową wartość, czyli:

```
renice +10 3442
```

będzie próbować ustawić *niceness* procesu o PID równym 3442 na 10, co zwykłemu użytkownikowi uda się tylko wtedy, gdy dotychczasowa wartość *niceness* była mniejsza od żądanej lub jej równa.

5.5. Manipulowanie procesami

Programy uruchamiane za pośrednictwem sesji terminalowej tworzą tak zwane *procesy pierwszoplanowe* (ang. *foreground processes*). Generalną zasadą jest, że powłoka czeka na zakończenie takiego procesu i będzie gotowa na przyjęcie kolejnych poleceń dopiero wtedy, gdy poprzednio uruchomiony proces zostanie usunięty z systemu.

Innym sposobem uruchomienia programu jest uczynienie z niego tak zwanego *procesu drugoplanowego*, nazywanego również *procesem pracującym w tle* (ang. *background process*). Proces drugoplanowy zaczyna pracować równolegle z powłoką, a powłoka nie czeka na jego zakończenie.

Oczywiście praca procesu w tle ma sens tylko wtedy, gdy nie jest wymagana interakcja ze strony użytkownika. W przeciwnym przypadku strumienie danych przyjmowane i wysyłane przez powłokę będą się mieszać z danymi emitowanymi i pobieranymi przez proces drugoplanowy. W wielu przypadkach problem tej natury może zostać rozwiązany przez przekierowania i/lub potoki zastosowane przy uruchomieniu programu pracującego w tle.

Program zostanie skierowany do uruchomienia w tle, jeśli za ostatnim parametrem jego polecenia będzie umieszczony znak **&** (ampersand):

```
polecenie &
```

Aktualnie pracujący proces pierwszoplanowy można wstrzymać (ale nie zatrzymać!) za pomocą kombinacji klawiszy *Ctrl-Z*. Proces wstrzymany rezyduje w systemie, ale nie jest aktywnie wykonywany (nie przydziela mu się procesora). Taki proces można ponownie skierować do wykonywania w tle poleceniem:

```
bg
```

Przywrócenie takiego procesu z powrotem na pierwszy plan wykona polecenie:

```
fg
```

Będzie to jednak możliwe tylko wtedy, gdy pomiędzy tymi poleceniami nie zostanie uruchomiony w tle inny proces. Listę wszystkich kontrolowanych zadań wyprowadza na strumień *stdout* polecenie o nazwie:

```
jobs
```

Jeśli wstrzymaniu poddano więcej niż jedno zadanie, jako argumentów poleceń **bg** i **fg** można używać unikalnych identyfikatorów zadań (ang. *job ids*) podawanych przez **jobs** (*uwaga: nie mają one nic wspólnego z numerami PID!*) i poprzedzonych znakiem % (procent).

Przykładowo

```
fg %1
```

wyprowadzi ze stanu wstrzymania zadanie o identyfikatorze 1.

5.6. Kod zakończenia procesu

Każdy kończący pracę proces przekazuje do systemu tak zwany *kod zakończenia* (ang. *completion code*). W programach napisanych w języku C jest to wartość zwrócona przez funkcję **`main()`** lub wartość przekazana jako argument do funkcji kończących pracę procesu (na przykład **`exit()`**). Kod zakończenia jest ośmiobitową liczbą całkowitą bez znaku i przyjęto, że wartość 0 oznacza bezbłędne wykonanie i zakończenie procesu. Wartości różne od zera mogą oznaczać błędy lub wystąpienie sytuacji nietypowych (na przykład kompilator może w ten sposób sygnalizować, że proces translacji przebiegł z ostrzeżeniami i/lub błędami). Kod zakończenia ostatnio wykonywanego programu jest umieszczany w zmiennej środowiskowej o nazwie **`$?`** i może być ujawniony poleceniem:

```
echo $?
```

lub użyty w skrypcie powłoki w celu zdiagnozowania sytuacji powstałej po uruchomieniu ostatniego programu. Warto zauważyć, że standardowym elementem wielu stron podręcznikowych jest opisanie kodów zakończenia zwracanych przez polecenia wraz ze wskazaniem sposobu ich interpretacji (sekcja o nazwie *EXIT STATUS*).

Status zakończenia pewnego procesu można również wykorzystać do warunkowego uruchomienia kolejnych poleceń. Na przykład zapis taki jak poniżej:

```
cmd1 && cmd2
```

oznacza, że **`cmd2`** należy wykonać tylko wtedy, gdy **`cmd1`** zakończyło się **sukcesem** (zwróciło zerowy kod powrotu).

Zapis o postaci:

```
cmd1 || cmd2
```

oznacza, że **`cmd2`** ma być wykonane tylko wtedy, gdy **`cmd1`** zakończyło się **niepowodzeniem** (zwróciło niezerowy kod powrotu). Semantyka ta może być wytłumaczona następującym spostrzeżeniem:

(true && x) = x
(false || x) = x

co jednoznacznie określa, kiedy obliczenie prawego argumentu operacji logicznej jest niezbędne.

Jedno polecenie powłoki może spowodować kolejne uruchomienie kilku procesów, o ile nazwy programów zostaną rozdzielone średnikami, na przykład tak:

```
cmd1; cmd2; cmd3
```

Sekwencja taka zostanie wprowadzona w tło, jeśli lista programów będzie ujęta w nawiasy i zakończona znakiem **&**:

```
(cmd1; cmd2; cmd3) &
```

5.7. Źródła uzupełniające

1. Manual polecenia **kill**: `https://man7.org/linux/man-pages/man1/kill.1.html`
2. Manual polecenia **killall**: `https://man7.org/linux/man-pages/man1/killall.1.html`
3. Manual polecenia **nice**: `https://man7.org/linux/man-pages/man1/nice.1.html`
4. Manual polecenia **ps**: `https://man7.org/linux/man-pages/man1/ps.1.html`
5. Manual polecenia **pstree**: `https://man7.org/linux/man-pages/man1/pstree.1.html`
6. Manual polecenia **renice**: `https://man7.org/linux/man-pages/man1/renice.1.html`
7. Manual polecenia **time**: `https://man7.org/linux/man-pages/man1/time.1.html`
8. Manual polecenia **top**: `https://man7.org/linux/man-pages/man1/top.1.html`
9. Opis zarządzania zadaniami w powłoce Bash: `https://www.gnu.org/software/bash/manual/bash.html#Job-Control-Basics`
10. Opis mechanizmu sygnałów w powłoce Bash: `https://www.gnu.org/software/bash/manual/bash.html#Signals`

5.8. Zadania do samodzielnego wykonania

1. Ile procesów prezentuje **ps** uruchomiony z przełącznikami **-er**? Dlaczego tylko tyle?
2. Co powoduje uruchomienie polecenia w stylu BSD o postaci: **ps aux**?
3. Do czego służy polecenie **uptime**? Jaki jest **uptime** komputera, na którym obecnie pracujesz?
4. Do czego służy polecenie **free**?
5. Do czego służy interaktywne polecenie **0** programu **top**?

6. Jakie jest domyślne sortowanie procesów w programie **top**?
7. Jaki efekt powoduje przełącznik **-l** polecenia **kill**?
8. Spróbuj zatrzymać ten ze swoich procesów, który ma najniższą wartość PID. Co się stało?
9. Co oznacza kod powrotu 1 zwrócony przez polecenie **grep**?
10. Czy polecenie **kill** może popełnić samobójstwo, to znaczy wysłać sygnał SIGKILL do siebie samego?
11. Czy polecenie **killall** może popełnić samobójstwo?
12. Ile pamięci RAM ma komputer, na którym obecnie pracujesz?
13. Skonstruuj prosty potok zliczający procesy wskazanego użytkownika.
14. Znajdź ścieżkę pokrewieństwa wiodącą od procesu *init/systemd* do procesu przeglądarki, w której oglądasz niniejszy dokument (wypisz PIDy wszystkich procesów na tej ścieżce).
15. Przeprowadź eksperyment sprawdzający, czy zmiana priorytetu na najgorszy może wpłynąć na czas wykonania złożonego polecenia. Użyj na przykład czasochłonnych wariantów poleceń **find** lub **grep** albo skomplikowanych potoków. Sprawdź czas wykonania z domyślnym i najgorszym możliwym *niceness*.
16. Gdzie w Polsce znajduje się miejscowość o nazwie Renice?
17. Rzetelnie przećwicz użycie poleceń **jobs**, **fg** i **bg**. W tym celu uruchom kilka procesów pracujących interaktywnie (na przykład **top**, **watch** albo edytor tekstu) i naucz się biegle przemieszczać między nimi oraz przełączać je pomiędzy pracą w tle a na pierwszym planie. Obserwuj wyniki podawane przez **top** w czasie twoich działań.
18. Zapoznaj się z listą sygnałów na stronie podręcznikowej **signal(7)**. Uruchom sesję edytora **vi** i wysyłaj do niego poleceniem **kill** kilka różnych sygnałów (użyj co najmniej **SIGHUP**, **SIGINT**, **SIGTERM**, **SIGQUIT**, **SIGKILL**, **SIGSTOP** i **SIGCONT**). Zanotuj zachowanie edytora w reakcji na przesłane sygnały.
19. Do czego służy polecenie **sleep**? Jak **top** uwidacznia proces wykonujący **sleep**?

6. Skrypty powłoki

6.1. Interpreter poleceń

Interpreter poleceń, nazywany inaczej także *powłoką systemową* (ang. *system shell*), jest pośrednikiem pomiędzy użytkownikiem a funkcjami systemu operacyjnego. Powłoka systemowa pobiera polecenia użytkownika wraz z argumentami i po sprawdzeniu ich poprawności próbuje je wykonać, używając do tego celu różnych usług świadczonych przez jądro systemu operacyjnego. Współcześnie dostępnych jest wiele alternatywnych powłok, z których najpopularniejsze wydają się następujące:

- ***bash*** (angielski akronim wzięty ze słów *Bourne Again Shell*, co wbrew pierwszemu skojarzeniu nie ma żadnego związku z postacią głównego protagonisty cyklu powieści sensacyjnych Roberta Ludluma, a stanowi nawiązanie do historycznej już powłoki napisanej w 1979 roku przez inżyniera Bell Laboratories Stephena Bourne'a, z przeznaczeniem do wykorzystania w systemie Unix wersji siódmej); obecnie jest to najpopularniejsza powłoka w systemach linuksowych, instalowana domyślnie w przeważającej większości z nich, a jej język skryptowy stał się *de facto* standardem publikacyjnym;
- ***zsh*** (ang. *Z Shell*, od inicjałów i loginu profesora Universytety Yale Zhong Shao, który to ciąg znaków wydał się autorowi pierwszej wersji powłoki, Paulowi Falstadowi, również pracownikowi tego uniwersytetu, bardzo dobrą

nazwą dla jego dzieła); *zsh* znacząco rozszerza wiele funkcjonalności powłoki *bash* i czyni pracę szybszą i wygodniejszą; ciekawostka: to właśnie *zsh* jest wzorcem, na którym implementowano powłokę systemu macOS, oraz domyślną powłoką dystrybucji *Kali Linux*;
- ***csh*** (ang. *C Shell*) – stabilna i szacowna powłoka wywodząca się z Uniwersytetu Berkeley i napisana w 1978 roku przez Billa Joya z przeznaczeniem dla systemu BSD Unix drugiej wersji; od powłoki *bash* odróżnia ją szczególnie składnia instrukcji upodobniona do tej znanej z języka C (stąd nazwa);
- ***tcsh*** (ang. *T-csh*, gdzie *T* jest pierwszą literą słowa *Tenex* – nazwy systemu operacyjnego napisanego w 1969 roku dla komputera DEC PDP-10, który to system był inspiracją dla autora powłoki Kena Greera, pracownika Uniwersytetu Carnegie Mellon); pierwsza edycja powłoki *tcsh* została wydana w 1981 roku z przeznaczeniem dla systemu BSD Unix wersji trzeciej; jest znaczącym udoskonaleniem powłoki *csh* i jest z nią wstecznie kompatybilna.

W dalszej części rozważań posłużono się wyłącznie konwencjami pochodzącymi z powłoki *bash*.

6.2. Zmienne środowiskowe

Zmienne środowiskowe (ang. *environment variables*) to wygodny i uniwersalny sposób konfigurowania i parametryzowania powłok systemowych oraz uruchamianych przez nie programów. Wszystkie znane w danym momencie zmienne środowiskowe tworzą tak zwane *środowisko* (ang. *environment*) wykonania procesu. Środowisko to jest kopiowane do wszystkich nowych procesów, a więc modyfikacje zmiennych wykonane w powłoce są widoczne we wszystkich programach uruchomionych przy użyciu tej powłoki po wykonaniu modyfikacji. Każdy użytkownik może definiować dowolną liczbę własnych zmiennych oraz przypisywać im dowolne wartości, chociaż należy pamiętać, że wartością zmiennych jest zawsze ciąg znaków (*string*), nawet jeśli reprezentuje liczbę.

Aby zdefiniować zmienną środowiskową, należy zastosować operator przypisania (znak „=”) w następujący sposób:

```
ZMIENNA=wartość
```

W tym wypadku ciąg znaków `ZMIENNA` to nazwa zmiennej, a ciąg znaków składających się na łańcuch `wartość` to przypisywana jej wartość. Należy zwrócić uwagę, że pomiędzy nazwą zmiennej, operatorem przypisania a wartością **nie mogą wystąpić żadne białe znaki**. Jeżeli wartość przypisywana do zmiennej ma zawierać białe znaki, to należy je zacytować w sposób identyczny jak w przypadku nazw plików.

Takie właśnie zachowanie powłoki spowodowane jest bardzo specyficznym

podejściem interpretera do analizy kodu źródłowego – mianowicie nie wykonuje on żadnej z analiz tradycyjnie przypisywanych interpreterom języków takich jak Perl czy Python, w tym nie przeprowadza analizy leksykalnej pozwalającej wyodrębnić leksemy z tekstu. Dla powłoki każda analizowana przez nią linia jest **poleceniem do wykonania**. Wstawienie do powyższego zapisu spacji, które zdaniem nieświadomego programisty pozwolą mu upiększyć tekst, na przykład w poniższy sposób

```
ZMIENNA = wartość
```

spowoduje, że powłoka będzie próbować uruchomić program o nazwie **ZMIENNA**, przekazując mu dwa argumenty: **=** oraz **wartość**, co oczywiście skończy się niepowodzeniem.

Odwołanie się do wartości zmiennej jest możliwe dzięki operatorowi **$**, który w tym kontekście staje się czymś na kształt *operatora wyłuskania*. Poprzedzenie nazwy zmiennej znakiem **$** sprawi, że powłoka zastąpi nazwę zmiennej jej bieżącą tekstową wartością. Oznacza to także, że użycie nieistniejącej zmiennej w połączeniu z **$** nie wywoła błędu, a nazwę taką powłoka zamieni na pusty łańcuch.

Wyprowadzenie wartości dowolnej zmiennej na strumień *stdout* jest możliwe poprzez wykorzystanie polecenia **echo**, co ilustruje poniższy przykład (błąd w trzecim z poleceń **echo** jest celowy i służy demonstracji zachowania powłoki w momencie odwołania do nieznanej jej zmiennej środowiskowej):

```
$ SYSTEM=Linux
$ echo $SYSTEM
Linux
$ echo SYSTEM
SYSTEM
$ echo SSYTEM

$
```

Polecenie powłoki o nazwie **set** pozwala wyświetlić wartości wszystkich znanych powłoce zmiennych środowiskowych, a samo zestawienie posortowane jest według kolejności alfabetycznej nazw zmiennych (w poniższym przykładzie uwidoczniono tylko 10 pierwszych wierszy wyemitowanych przez polecenie – całe wyjście może zajmować nawet kilkaset wierszy, co daje obraz tego, jak intensywnie powłoka używa tego mechanizmu).

```
$ set
'!'=0
'#'=0
'$'=16423
'*'=(  )
-=0569BJXZims
0=zsh
'?'=0
@=(  )
ANT_HOME=/usr/share/ant
ARGC=0
CDPATH=''
```

Polecenie w postaci

unset *zmienna*

usuwa zmienną środowiskową o wskazanej nazwie, na przykład:

```
$ VAR=123
$ echo $VAR
123
$ unset VAR
$ echo $VAR

$
```

Jak wspomniano, środowisko wykonania procesu jest kopiowane do jego procesów potomnych, jednak nie wszystkie zmienne powłoki muszą podlegać temu procesowi. Te zmienne, które są przekazywane do środowiska potomków, nazywa się zmiennymi *eksportowanymi* (ang. *export variables*), a zmienne, które takiemu przekazaniu nie podlegają, nazywa się zmiennymi *lokalnymi* (ang. *local variables*).

Z reguły nowo tworzone zmienne są zmiennymi lokalnymi i niezbędne jest jawne wskazanie, że mają być zmiennymi eksportowanymi. Proces taki realizowany jest poleceniem:

export *lista_zmiennych*

Oto przykład utworzenia zmiennej i jej wyeksportowania:

```
$ COUNTER=1
$ export COUNTER
```

Powyższe dwa zlecenia można także zrealizować w jednym kroku:

```
$ export COUNTER=1
```

Domyślnie powłoka definiuje pokaźną liczbę zmiennych środowiskowych, które można wykorzystywać do rozpoznania bieżącego stanu systemu operacyjnego oraz otoczenia, w jakim wykonuje się skrypt.

Oto niektóre z takich zmiennych:

HOME
kompletna ścieżka prowadząca do katalogu domowego użytkownika, na przykład:

```
$ echo $HOME
/home/slawek
```

USER
nazwa użytkownika, który odwołuje się do wartości zmiennej, na przykład:

```
$ echo $USER
slawek
```

PATH
lista katalogów przeszukiwanych przez powłokę w celu odnalezienia programu, którego uruchomienia żąda użytkownik; nazwy katalogów są rozdzielane znakiem dwukropka (`:`), na przykład:

```
$ echo $PATH
/usr/local/sbin:/usr/sbin:/usr/bin:/sbin:/bin
```

PS1
postać *prompta*, jaki powłoka wyświetla użytkownikowi, zapisana przy użyciu specyficznej składni, na przykład:

```
$ echo $PS1
\u@\h:\w\$
```

SHELL
pełna ścieżka prowadząca do domyślnej powłoki użytkownika;
uwaga: powłoka używana w danym momencie może być inna niż domyślna, na przykład:

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

6.3. Skrypty i ich argumenty

Skrypty powłoki to pliki tekstowe, które zawierają ciągi poleceń dla powłoki systemowej. Współczesne powłoki pozwalają na to, aby oprócz poleceń skrypty zawierały także typowe konstrukcje programistyczne takie jak instrukcje warunkowe czy pętle, operacje wejścia/wyjścia, które umożliwiają interaktywną pracę skryptu czy też opracowywanie argumentów wywołania skryptów. W ten sposób skrypty pozwalają kodować i wykonywać bardzo złożone czynności przy użyciu wyłącznie środków udostępnianych przez powłokę i narzędzia systemu, bez konieczności uciekania się do korzystania z języków programowania i ich środowisk.

Skrypty mogą być *parametryzowane* argumentami ich wywołania – oznacza to, że podczas uruchamiania skryptu można przekazać do niego dowolną liczbę różnorakich danych. Do argumentów wywołania można się odwoływać w skryptach za pomocą tak zwanych *parametrów pozycyjnych* (ang. *positional parameters*), oznaczanych cyframi dziesiętnymi od **1** do **9**. Każdy parametr pozycyjny przechowuje tekst przekazany za pośrednictwem odpowiadającego mu argumentu wywołania – pierwszy argument dostępny jest w parametrze **$1**, drugi w parametrze **$2** itd. Oto przykład trywialnego skryptu, który wyświetla wartości pierwszych trzech argumentów jego wywołania (użyte w nim polecenie **echo**, służące do wyprowadzania danych na *stdout*, zostanie omówione niebawem):

```
1 #!/bin/bash
2 # pierwszy skrypt
3 echo argument nr 1: $1
4 echo argument nr 2: $2
5 echo argument nr 3: $3
```

W celu uruchomienia skryptu należy powyższy tekst umieścić w pliku o dowolnej nazwie (zaleca się jednak, aby pliki skryptów miały rozszerzenie **.sh**). Przedstawiony kod należy zinterpretować w następujący sposób:

- pierwsza linia zawiera element składniowy nazywany zwyczajowo *hash-bang* (**#!**), (chociaż spotyka się również określenie *shebang*) jednoznacznie wskazujący, który konkretnie interpreter poleceń ma zostać wykorzystany do wykonania zawartości pliku – w tym przypadku jest to powłoka *bash*;
- druga linia ilustruje sposób zapisu komentarza: jest nim tekst rozpoczynający się od znaku **#** aż do końca zawierającego go linii;
- kolejne trzy linie wyświetlają wartości pierwszych trzech argumentów wywołania skryptu przy wykorzystaniu polecenia **echo**.

Aby uruchomić taki skrypt wprost z linii poleceń, należy zawierającemu go plikowi nadać prawo wykonywania (**x**).

Poniżej przedstawiono przykładowe wywołanie zaprezentowanego powyżej skryptu oraz wynik jego działania (przyjęto, że plik ze skryptem nazywa się **skrypt1.sh**):

```
./skrypt1.sh abc xyz 12345
argument nr 1: abc
argument nr 2: xyz
argument nr 3: 12345
```

6.4. Wyprowadzanie danych na *stdout*

Podstawowym środkiem umożliwiającym skryptowi powłoki wyprowadzanie danych na strumienie wyjściowe jest polecenie **echo** o następującej składni:

echo ***opcja... łańcuch...***

Polecenie **echo** spowodowuje przesłanie do *stdout* wszystkich *łańcuchów* podanych w argumentach.

Możliwe do użycia opcje to:

- **-n** nie przechodź do nowego wiersza po wyprowadzeniu tekstu (domyślnie każde uruchomienie polecenie **echo** wyprowadza do *stdout* osobny wiersz tekstu);
- **-e** włącz rozpoznawanie i honorowanie sekwencji sterujących, rozpoczynających się znakiem ****;
- **-E** wyłącz rozpoznawanie i honorowanie sekwencji sterujących (zachowanie domyślne).

Jeśli **echo** pracuje w trybie honorowania sekwencji sterujących, rozpoznawane są następujące z nich:

- **** znak ****;
- **\a** znak kodu ASCII o nazwie *BEL* (alarm), prowokujący terminal do wydania krótkiego dźwięku ostrzegawczego;
- **\b** znak kodu ASCII o nazwie *BS*, powodujący cofnięcie kursora o jeden znak;
- **\c** przerwanie wyprowadzania na *stdout* bieżącego argumentu i przejście do następnego;
- **\e** znak kodu ASCII o nazwie *ESC*, rozpoczynający sekwencje sterowania terminalem;
- **\f** znak kodu ASCII o nazwie *FF*, wymuszający na drukarce przejście do nowej strony; na terminalu bezużyteczny, wyprowadza jedynie jedną pustą linię;
- **\n** znak kodu ASCII o nazwie *LF*, wymuszający przejście do nowego wiersza;

\r znak kodu ASCII o nazwie *CR*, wymuszający powrót kursora do początku bieżącego wiersza;

\t znak kodu ASCII o nazwie *HT*, wymuszający przejście kursora do kolejnej pozycji tabulatora;

\0NNN znak kodu ASCII o kodzie wyrażonym ósemkowo jako *NNN* (od 1 do 3 cyfr);

\xHH znak kodu ASCII o kodzie wyrażonym szesnastkowo jako *HH* (od 1 do 2 cyfr).

6.5. Wprowadzanie danych z *stdin*

Jeśli skrypt wymaga interakcji z użytkownikiem, to niezbędne staje się pobieranie wartości przekazywanych przez użytkownika. Służy do tego polecenie:

```
read [ -r ] zmienna...
```

Opcja **-r** – jeśli zostanie użyta, informuje polecenie **read**, że znaki odwróconego ukośnika podawane przez użytkownika jako wartości zmiennych mają być traktowane dosłownie, a nie jako znaki rozpoczynające cytowanie.

Argumentami polecenia **read** są nazwy zmiennych środowiskowych, które przyjmą wartości odczytane ze standardowego wejścia (czytanie wejścia trwa aż do napotkania znaku nowej linii). Jeśli podano więcej niż jedną zmienną, to są one inicjowane w ten sposób, że pierwsze słowo (ciąg znaków ograniczony białymi znakami) trafia do pierwszej zmiennej, drugie do drugiej itd. Działanie **read** można przetestować, wykonując następującą sekwencję:

```
$ read X Y
uzytkownik adam
$ echo $X
uzytkownik
$ echo $Y
adam
```

6.6. Śledzenie wykonania skryptu

Skrypty mogą być także wykonywane w trybie debugowania, na przykład w celu testowania poprawności działania warunków, pętli i tym podobnych. Aby zrealizować wykonanie skryptu z włączonym mechanizmem śledzenia, należy zastosować przełącznik **-x** wywołania interpretatora poleceń. Można to zrealizować na dwa sposoby:

– dopisać tenże przełącznik w pierwszej linii skryptu, na przykład w taki sposób:

```
#!/bin/bash -x
```

– uruchomić skrypt, wywołując go poprzez jawne wskazanie interpretatora z przełącznikiem i nazwą skryptu jako argumentem; przy przyjęciu, że skrypt umieszczony jest w pliku o nazwie *skrypt.sh*, uruchomienie miałoby następującą postać:

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

Załóżmy, że w pliku o nazwie *dword.sh*, posiadającym ustawiony atrybut **x**, umieszczony jest następujący kod skryptu:

```
1 #! /bin/bash
2
3 echo "Podaj słowo:"
4 read VAR
5 VAR=$VAR$VAR
6 echo $VAR
```

Zadaniem skryptu jest zdublowanie ciągu znaków wprowadzonych na strumień *stdin* przez użytkownika. Wykonanie skryptu może w takim przypadku przebiegać następująco:

```
$ ./dword.sh
Podaj słowo:
WORD
WORDWORD
$
```

Wykonanie tego samego skryptu w trybie śledzenia wygląda jak poniżej:

```
$ /bin/bash -x dword.sh
+ echo 'Podaj słowo:'
Podaj słowo:
+ read VAR
WORD
+ VAR=WORDWORD
+ echo WORDWORD
WORDWORD
$
```

Jak widać, w tym przypadku wykonanie każdej linii kodu źródłowego jest poprzedzone wyprowadzeniem na *stdout* jej treści poprzedzonej znakiem **+**.

6.7. Zmienne predefiniowane

Wykonujący się skrypt ma dostęp do licznych zmiennych predefiniowanych, które pozwalają efektywnie rozpoznawać kontekst wykonania kodu. Najczęściej używane zebrano w poniższej tabeli:

Nazwa	*Zawartość*
$0	Nazwa, pod jaką uruchomiono skrypt
$*n*	Wartość argumentu *n* (*1..9*)
$@	Wszystkie argumenty skryptu złożone w wektor
$*	Wszystkie argumenty skryptu złożone w łańcuch znaków
$#	Liczba argumentów skryptu
$?	Kod powrotu z ostatnio wykonanego polecenia
$$	PID aktualnie wykonywanej powłoki (tym samym PID skryptu)

6.8. Łańcuchy znaków

Skrypty powłoki Bash rozpoznają trzy rodzaje łańcuchów znaków:

– łańcuchy ujęte w cudzysłów (**"**), na przykład **"tekst"**: łańcuchy tego rodzaju są przeglądane przez powłokę i jeśli w ich treści znajdują się odwołania do zmiennych, zostaną one zastąpione wartościami tych zmiennych, na przykład:

```
$ VAR=123
$ echo "VAR=$VAR"
VAR=123
```

– łańcuchy ujęte w apostrofy (**'**), na przykład **'tekst'**: łańcuchy tego rodzaju są traktowane dosłownie i nie są poddawane żadnej dodatkowej obróbce, na przykład:

```
$ VAR=123
$ echo 'VAR=$VAR'
VAR=$VAR
```

– łańcuchy ujęte w akcenty (odwrócone apostrofy, ang. *backtick*), na przykład **`tekst`**: łańcuchy tego rodzaju są traktowane jako polecenia powłoki, a ich faktyczną wartością jest całe wyjście, jakie użyte polecenie wyprowadziło na swoje *stdout*, na przykład:

```
$ VAR=‘date -I‘
$ echo $VAR
2022-04-01
```

6.9. Instrukcja warunkowa

Składnia instrukcji warunkowej w powłoce Bash jest następująca:

```
if warunek
then
   polecenie
   :
else
   polecenie
   :
fi
```

Warto zauważyć, że:

- gałąź rozpoczynająca się od **else** jest opcjonalna,
- wcięcia nie są konieczne, ale zaleca się ich stosowanie dla poprawienia czytelności kodu,
- możliwe jest zastąpienie przejścia do nowego wiersza znakiem średnika, co pozwala na zwięzłe zapisywanie prostszych partii kodu (przykład takiego zapisu zostanie przedstawiony niebawem).

Warunek może być dowolnym poleceniem i w takim przypadku uznaje się, że jest **spełniony** (prawdziwy), kiedy polecenie to zwróciło kod powrotu **zero**. Zwrócenie niezerowego kodu powrotu powoduje, że warunek uznaje się za niespełniony (fałszywy). Ponieważ takie zachowanie może być sprzeczne z intuicjami programisty nienawykłego do programowania skryptów powłoki, wskazana jest pewna ostrożność.

Jeśli *warunek* ma przybrać postać wyrażenia zbliżonego do składni używanej w tradycyjnych językach programowania, należy posłużyć się poleceniem **test** używającym znaków **[** i **]** do wyróżnienia wyrażenia, którego wartość ma zostać wyznaczona. Na przykład:

```
1 if [ $1 = 1 ]
2 then
3     echo "eq"
4 else
5     echo "ne"
6 fi
```

W tym miejscu warto zauważyć, że znak **[** nie reprezentuje tu leksemu (co zostałoby uznane za normalne w przypadku klasycznego języka programowania), ale jest **poleceniem** (*sic!*) uruchamianym z pliku **/usr/bin/[**. Polecenie to analizuje swoje argumenty aż do momentu napotkania wśród nich znaku **]**, następnie próbuje złożyć z nich wyrażenie logiczne i, jeśli jego obliczenie jest możliwe, kończy pracę z kodem powrotu odpowiadającym prawdziwości warunku. Dlatego też po znaku **[** i przed znakiem **]** konieczne jest umieszczenie co najmniej jednego znaku spacji – niespełnienie tego warunku uniemożliwi powłoce prawidłowe wyodrębnienie warunku spośród otaczającego go tekstu.

Taki sam efekt można uzyskać jeśli zapisze się kod w zwięźlejszy sposób, na przykład tak:

```
if [ $1 = 1 ]; then echo "eq"; else echo "ne"; fi
```

Pierwszy ze sposobów jest zalecany w treści skryptów, drugi upraszcza wydawanie poleceń z konsoli.

Najczęściej polecenia **test** używa się do wykonania następujących sprawdzeń:

Warunek	*Opis*
str1 = str2	Weryfikacja równości dwóch łańcuchów znaków
str1 != str2	Weryfikacja nierówności dwóch łańcuchów znaków
-z str	Weryfikacja, czy łańcuch znaków ma zerową długość
-n str	Weryfikacja, czy łańcuch znaków ma niezerową długość
num1 -eq num2	Weryfikacja równości dwóch liczb
num1 -ne num2	Weryfikacja nierówności dwóch liczb
num1 -gt num2	Weryfikacja, czy liczba **num1** jest większa od **num2**
num1 -ge num2	Weryfikacja, czy liczba **num1** jest większa lub równa **num2**
num1 -lt num2	Weryfikacja, czy liczba **num1** jest mniejsza od **num2**
num1 -le num2	Weryfikacja, czy liczba **num1** jest mniejsza lub równa **num2**
-e file	Weryfikacja, czy plik **file** istnieje
-f file	Weryfikacja, czy plik **file** jest plikiem zwykłym
-d file	Weryfikacja, czy plik **file** jest katalogiem
-L file	Weryfikacja, czy plik **file** jest dowiązaniem symbolicznym
-r file	Weryfikacja, czy użytkownik ma prawo odczytu pliku **file**
-w file	Weryfikacja, czy użytkownik ma prawo zapisu pliku **file**
-x file	Weryfikacja, czy użytkownik ma prawo wykonania pliku **file**
cond1 -a cond2	Iloczyn logiczny (*and*) warunków **cond1** i **cond2**
cond1 -o cond2	Suma logiczna (*or*) warunków **cond1** i **cond2**
! cond	Negacja warunku **cond**

Pełny zestaw możliwych do wykonania sprawdzeń znajduje się w manualu do polecenia **test**.

Warto zauważyć, że rozważne użycie znaków cudzysłowu ma fundamentalne znaczenie przy kodowaniu warunków z użyciem polecenia **test**. Poniższa sekwencja (mimo pozornej poprawności wynikającej z nieodpowiedzialnego poddania się nawykom nabytym podczas programowania w innych językach) może wywołać błąd, gdy zmienna **VAR** nie ma nadanej wartości:

```
if [ -n $VAR ]
then
    echo podano parametr
fi
```

W takim przypadku interpreter będzie próbował wykonać kod, który po zastąpieniu zmiennych ich wartościami będzie prezentował się następująco:

```
if [ -n ]
then
    echo podano parametr
fi
```

i spowoduje błąd wykonania już w pierwszej linii.

Użycie cudzysłowu uczyni powyższy kod w pełni poprawnym:

```
if [ -n "$VAR" ]
then
    echo podano parametr
fi
```

6.10. Pętle

Skrypty powłoki mogą także zawierać pętle – podstawowe z nich to warianty pętli **for** oraz pętla **while**.

6.10.1. Pętla `for` iterująca po liście

Pętla taka wykonywana jest z góry określoną liczbę razy, a jej zadanie to przejrzenie zawartości pewnej listy. Jej składnia ogólna prezentuje się następująco:

```
for zmienna in lista
do
   polecenie
   :
done
```

albo (w wersji zwięzłej):

```
for zmienna in lista; do polecenie; ...; done
```

Wykonanie pętli powoduje przypisywanie zmiennej **zmienna** kolejnych wartości wymienionych na liście **lista**; liczba iteracji jest zatem zależna od długości podanej listy.

Samą listę można skonstruować na kilka sposobów:

– podać ją literalnie jak zbiór słów rozdzielonych białymi znakami, na przykład tak:

```
#! /bin/bash

for zm in ala ma kota
do
  echo $zm
done
```

Powyższy skrypt produkuje następujące wyjście:

```
ala
ma
kota
```

– podać ją jako wieloznaczną nazwę pliku – w takim przypadku powłoka skonstruuje listę, wstawiwszy do niej wszystkie pasujące do wzorca nazwy plików, na przykład tak:

```
#! /bin/bash

for zm in /s*
do
  echo $zm
done
```

Powyższy skrypt produkuje następujące wyjście:

```
/sbin
/srv
/sys
```

– użyć w charakterze listy wyjścia z dowolnego programu uruchomionego przez powłokę – w takim przypadku każde słowo odnalezione w strumieniu zostanie użyte jako kolejna wartość zmiennej, na przykład:

```
#! /bin/bash

for zm in `wc /etc/passwd`
do
  echo $zm
done
```

Przykładowe wyjście z wykonania powyższego skryptu może wyglądać następująco:

```
65
270
4580
/etc/passwd
```

6.10.2. Pętla `for` ze zmienną licznikową

W dawniejszych implementacjach powłoki Bash zakładano, że jeśli użytkownik zażyczy sobie iterowania po kolejnych wartościach liczb z podanego przedziału, to posłuży się w tym celu poleceniem **`seq`**, produkującym na swoim wyjściu sekwencje wartości liczbowych z zakresu definiowanego argumentami polecenia. Możliwe warianty użycia polecenia **`seq`** prezentują się następująco:

```
seq OSTATNIA
seq PIERWSZA OSTATNIA
seq PIERWSZA PRZYROST OSTATNIA
```

Pominięcie którejś z wartości *PIERWSZA* i *PRZYROST* spowoduje przyjęcie dla nich wartości 1. Poniżej podano kilka przykładów użycia tego polecenia:

```
$ seq 2
1
2
$ seq 2 3
2
3
$ seq 1 2 5
1
3
5
$ seq 5 -2 1
5
3
1
```

Jeśli dysponuje się takim oto *generatorem* sekwencji, można zaprząc go do współpracy z pętlą **for**, na przykład w poniższy sposób:

```
#! /bin/bash

for n in `seq 3`
do
  mkdir dir$n
done
```

Kod powyższy spowoduje utworzenie trzech katalogów o nazwach **dir1**, **dir2** i **dir3**.

Ponieważ zapis taki jest w oczywisty sposób niewygodny i nieczytelny, pętla **for** ewoluowała i na pewnym etapie rozwoju powłoki przyjęła taką oto postać:

```
#! /bin/bash

for $n in {1..3}
do
  echo $n
done
```

Kod powyższy spowoduje wypisanie na *stdout* trzech linii tekstu, zawierających kolejno liczby **1**, **2** i **3**.

Na dalszym etapie dodano również możliwość użycia przyrostu różnego od 1, co zapisuje się w nieco nieczytelny sposób:

```
for $n in {1..5..2}
do
  echo $n
done
```

W tym przypadku na strumieniu *stdout* pojawią się kolejno liczby **1**, **3** i **5**.

Współcześnie używane wersje powłoki Bash umożliwiają na szczęście zapis iterowanej pętli **for** w sposób niemalże identyczny z tym, jaki implementują języki programowania wywodzące się z języka C. Oto przykład:

```
#! /bin/bash

for((n=1; n<=3; n++))
do
  rmdir dir$n
done
```

Należy zwrócić uwagę na użycie podwojonych nawiasów okrągłych – jest to niezbędny element takiej pętli. Użycie pojedynczych nawiasów wywoła błąd.

W tym wariancie możliwe jest – dokładnie jak w języku C – zapisanie pętli nieskończonej:

```
#! /bin/bash

for(( ; ; ))
do
  echo "Pomocy!"
  sleep 1
done
```

6.10.3. Pętla `while`

Analogicznie do klasycznych języków programowania pętla **`while`** pozwala organizować cykliczne wykonanie partii kodu, dla których liczba iteracji nie jest znana z góry. Jej składnia dla skryptów powłoki Bash jest następująca:

```
while warunek
do
   polecenie
   :
done
```

Warunek może być dowolnym poleceniem i najczęściej jest konstruowany – tak jak w przypadku instrukcji warunkowej – z zastosowaniem programu **`test`**.

Przykładem zastosowania pętli **`while`** może być skrypt wypisujący na ekranie wartości wszystkich argumentów jego wywołania (niezależnie od ich liczby). Pętla wykorzystuje polecenie **`shift`**, które powoduje przesunięcie argumentów w lewo (tzn. argument nr 2 staje się argumentem nr 1, nr 3 – nr 2 itd.). W ten sposób można przezwyciężyć przeszkodę wynikającą z narzuconego przez powłokę Bash ograniczenia nie zezwalającego na użycie zmiennych **`$n`** dla *n* większego od 9.

Oto skrypt:

```
while [ -n "$1" ]
do
  echo $1
  shift
done
```

lub w zwięzłym, równoważnym zapisie:

```
while [ -n "$1" ]; do echo $1; shift; done
```

Warunkiem wykonania pętli jest tu sprawdzenie, czy pierwszy argument wywołania skryptu (reprezentowany zmienną **`$1`**) ma niezerową długość – jeśli skrypt został uruchomiony bez żadnych argumentów, to pętla nie zostanie wykonana ani razu. Jeśli natomiast skrypt został wykonany z argumentami, to w pierwszym wykonaniu pętli zostanie wyświetlona wartość pierwszego argumentu, a następnie nastąpi przesunięcie argumentów w lewo poleceniem **`shift`** (drugi argument stanie

się pierwszym, trzeci drugim itd.). Pętla zakończy się, jeśli zostaną wyświetlone i przesunięte wszystkie argumenty (zmienna **$1** będzie miała wówczas zerową długość).

Należy zauważyć, że otoczenie nazwy zmiennej **$1** cudzysłowem jest w tym przypadku koniecznością – jego pominięcie spowoduje błąd, gdy zmienna ta nie będzie miała przypisanej wartości, co w przypadku powłoki Bash oznacza, że odwołanie do niej zostanie zastąpione pustym łańcuchem znaków.

6.10.4. Sterowanie pętlą

Do przerywania wykonania pętli, podobnie jak w języku C, służy polecenie **break** – oto przykład jego zastosowania:

```
#!/bin/bash
for FILE in *.tmp
do
  if [ ! -f $FILE ]
  then
    echo "$FILE nie jest plikiem!"
    break
  fi
  rm -v $FILE
done
```

W powyższym przykładzie pętla **for** zostanie przerwana, jeśli nazwa przechowywana w zmiennej **FILE** nie będzie identyfikowała pliku zwykłego. Możliwe jest również posłużenie się poleceniem **continue**, które (podobnie jak w języku C) rozpocznie kolejny obrót pętli z pominięciem jej wnętrza.

Zachowanie polecenia **continue** można prześledzić na poniższym przykładzie:

```
#! /bin/bash

for FILE in *
do
  if [ ! -L "$FILE" ]
  then
    continue
  fi
  echo $FILE
done
```

Powyższy kod powoduje wypisanie na strumień *stdout* nazw wszystkich dowiązań symbolicznych, jakie istnieją w katalogu bieżącym (w celu wykrycia dowiązań użyto przeznaczonego do tego celu operatora **-L**).

W powyższym przykładzie otoczenie cudzysłowem odwołania do zmiennej **FILE** jest niezbędne w celu uniknięcia błędów, jakie nieuchronnie by wystąpiły, gdyby nazwa jakiegokolwiek z badanych plików zawierała spację.

6.11. Instrukcja wyboru

Instrukcja **case** pozwala na dokonanie wyboru wielowariantowego i jest bliskim kuzynem znanej z języka C/C++ instrukcji **switch**.

Składnia instrukcji **case** prezentuje się następująco:

```
case zmienna in
"wzorzec_1") polecenie_1 ;;
"wzorzec_2") polecenie_2 ;;
:
"wzorzec_n") polecenie_n ;;
*) polecenie_domyślne
esac
```

Wartość zmiennej **zmienna** porównywana jest tu z kolejno podanymi wzorcami. Jeśli wzorzec ma taką samą wartość jak zmienna, wówczas wykonywane są polecenia przypisane do tego wzorca. Jeśli nie uda się znaleźć dopasowania, może zostać wykonane polecenie domyślne, oznaczone symbolem * (gwiazdka). Polecenie domyślne warto umieszczać w instrukcji **case zawsze**, co może zabezpieczać przed błędami popełnionymi przez użytkownika w konstrukcji poprzedzających wzorców.

Poniższy przykład ilustruje użycie instrukcji **case** do weryfikacji poprawności odpowiedzi udzielonej przez użytkownika:

```
1  #!/bin/bash
2
3  echo "Czy mogę usunąć twój katalog domowy? (tak/nie):"
4  read odp
5  case "$odp" in
6    "tak") echo "Dziękuję"
7           echo "rm -rf /home/yourdir" ;;
8    "nie") echo "Jak sobie życzysz."
9           echo "Pora na CSa";;
10   *) echo "Zła odpowiedź"
11 esac
```

6.12. Wektory

Powłoka Bash umożliwia posługiwanie się wektorami, które – podobnie jak w języku C – indeksowane są począwszy od zera. Istotną różnicą jest jednak fakt, że odwołanie się do nieistniejącego elementu wektora **nie wywołuje błędu**, ale jedynie udostępnia w wyniku pusty łańcuch.

Analogicznie do zwyczajów panujących w języku Python indeksowanie wektora indeksami ujemnymi pozwala uzyskać dostęp do elementów od końca wektora, czyli indeks –1 udostępnia ostatni element wektora, –2 – przedostatni itd.

Składnia umożliwiającą utworzenie wektora i manipulowanie nim zostanie przedstawiona na przykładach.

Utworzenie wektora może zostać wykonane poprzez przypisanie do niego listy (warto zauważyć, że separatorami elementów listy są białe znaki, więc jeśli któryś element listy zawiera na przykład spację, powinien zostać ujęty w cudzysłów bądź apostrofy):

```
wektor=(file1.txt "file 2.bin" 128)
```

Ten sam efekt można również osiągnąć po przypisaniu po kolei elementów wektora:

```
wektor[0]=file1.txt
wektor[1]="file 2.bin"
wektor[2]=128
```

Dostęp do wartości elementu wektora o wybranym indeksie wymaga użycia specyficznej składni o poniższej postaci:

```
${wektor[indeks]}
```

Poniższa sekwencja instrukcji spowoduje zamianę wartości drugiego i trzeciego elementu wektora:

```
zm=${wektor[1]}
wektor[1]=${wektor[2]}
wektor[2]=$zm
```

Użycie indeksu podanego jako * (gwiazdka) spowoduje udostępnienie łańcucha powstałego z konkatenacji wszystkich elementów listy, na przykład:

```
$ echo ${wektor[*]}
file1.txt 128 file 2.bin
```

Liczbę elementów w wektorze udostępnia niecodzienny operator **#**, a niecodzienność ta wynika z faktu, że **#** użyty w innych kontekstach oznacza otwarcie komentarza.

```
$ echo ${#wektor[*]}
3
```

Iterowanie po elementach tablicy można wykonać z użyciem pętli **for**:

```
$ for((i=0;i<3;i++)); do echo ${wektor[$i]}; done
file1.txt
128
file 2.bin
```

6.13. Operacje arytmetyczne

Skrypty powłoki mogą wykonywać operacje arytmetyczne, jednakże z powodów historycznych w tej części składni panuje pewien nieporządek.

Najstarszym środkiem umożliwiającym wykonanie prostej operacji arytmetycznej jest specjalnie do tego przeznaczone polecenie **let** o następującej składni:

let ***wyrażenie...***

Uwaga: zaleca się ujęcie całego wyrażenia w cudzysłów, co pozwala na uniknięcie kłopotów z białymi znakami, jakie ewentualnie mogą pojawić się w jego treści.

Zestaw operatorów akceptowanych przez polecenie **let** jest niemal w całości zgodny z możliwościami oferowanymi przez język C i obejmuje:

=	przypisanie,
x++, **++x**	post- i preinkrementacja zmiennej,
x--, **--x**	post- i predekrementacja zmiennej,
!	negacja logiczna,
~	negacja bitowa,
**	potęgowanie (jak w języku Python),
+, **-**	dodawanie i odejmowanie oraz zachowanie i zmiana znaku,
*, /, %	mnożenie, dzielenie, reszta z dzielenia,
<<, **>>**	przesuwanie bitowe,
<, **>**	porównania ostre,
<=, **=>**	porównania nieostre,
==, **!=**	równość i nierówność,
&, **|**, ~	operacje bitowe: AND, OR, XOR,
&&, **||**	operacje logiczne: AND, OR,
w1?w2:w3	operator trójargumentowy (jak w języku C).

Cenną własnością składni wyrażeń polecenia **let** jest możliwość używania nazw zmiennych **wprost**, bez konieczności poprzedzania ich znakiem **$**.

Poniższy przykład ilustruje obliczanie silni dla podanego argumentu (kod jest oczywiście skrajnie naiwny i służy tylko do celów demonstracji działania polecenia **let**):

```
#! /bin/bash

echo -n "Podaj n:"
read n
let "s=1"
for((i = 2; i <= n; i++))
do
  let "s=s*i"
done
echo "$n! = $s"
```

W trakcie rozwoju powłoki Bash udostępniono użytkownikom bardziej zaawansowany sposób konstruowania wyrażeń, który pozwala ich używać wprost w kodzie, bez konieczności wcześniejszego wykonania obliczeń i przypisania ich wyników do zmiennych. Techniki te sprowadzają się do umieszczenia wyrażenia, o składni zgodnej z tą używaną przez polecenie **let**, wewnątrz specjalnie do tego przeznaczonych nawiasów składniowych o jednej z poniższych postaci:

$[*wyrażenie*]
$((*wyrażenie*))

Poniżej przedstawiono kod z poprzedniego przykładu przepisany w tej konwencji:

```
#! /bin/bash

echo -n "Podaj n:"
read n
s=1
for((i = 2; i <= n; i++))
do
  s=$((s*i))
done
echo "$n! = $s"
```

6.14. Źródła uzupełniające

1. Manual polecenia **echo**: `https://man7.org/linux/man-pages/man1/echo.1.html`
2. Manual polecenia **export**: `https://man7.org/linux/man-pages/man1/export.1p.html`
3. Manual polecenia **read**: `https://man7.org/linux/man-pages/man1/read.1p.html`
4. Manual polecenia **set**: `https://man7.org/linux/man-pages/man1/set.1p.html`

5. Manual polecenia **seq**: `https://man7.org/linux/man-pages/man1/seq.1.html`
6. Manual polecenia **test**: `https://man7.org/linux/man-pages/man1/test.1.html`
7. Manual polecenia **top**: `https://man7.org/linux/man-pages/man1/top.1.html`
8. Manual polecenia **unset**: `https://man7.org/linux/man-pages/man1/unset.1p.html`
9. Spis słów zarezerwowanych dla powłoki Bash: `https://www.gnu.org/software/bash/manual/bash.html#Reserved-Word-Index`
10. Spis zmiennych wbudowanych w powłokę Bash: `https://www.gnu.org/software/bash/manual/bash.html#Bash-Variables`
11. Strona domowa projektu GNU Bash: `https://www.gnu.org/software/bash/`
12. Strona domowa powłoki zsh: `https://zsh.sourceforge.io/`

6.15. Zadania do samodzielnego wykonania

Napisz skrypt, który:

1. Po uruchomieniu będzie czekał na wprowadzenie działania w postaci **A @ B**, gdzie **A** i **B** to liczby, a **@** to jeden z operatorów **+**, **-**, **/** lub *****, po czym będzie wyświetlał wynik wprowadzonego działania.
2. Będzie przyjmował dowolną liczbę argumentów wywołania będących liczbami całkowitymi, sumował je i na końcu wyświetlał wynik.
3. Będzie przyjmował dowolną liczbę argumentów wywołania i wyświetlał je w odwrotnej kolejności (*wskazówka: wykorzystaj polecenie* **tac**).
4. Będzie przyjmował dowolną liczbę argumentów liczbowych i wyświetlał je posortowane malejąco.
5. Wyświetli najkrótszą i najdłuższą nazwę użytkownika w systemie (*wskazówka: nazwa użytkownika jest pierwszym polem w pliku* **/etc/passwd**).
6. Porówna zawartość pliku o nazwie podanej jako pierwszy argument wywołania z plikami, których nazwy znajdują się w kolejnych liniach pliku o nazwie podanej jako drugi parametr wywołania (porównanie wykonaj za pomocą polecenia **cmp**).
7. Będzie przyjmował dowolną liczbę argumentów będących nazwami katalogów i wyświetlał dla każdego katalogu jego nazwę oraz liczbę znajdujących się w nim plików nieodczytywalnych (bez prawa **r**).
8. W katalogu podanym jako jego argument wywołania zmieni nazwy wszystkich plików i katalogów, tak aby litery małe zostały zamienione na wielkie, a wielkie na małe (czyli na przykład z **RazDwa.txt** na **rAZdWA.TXT**).
9. Zamieni rozszerzenia nazw plików we wskazanym katalogu z podanych na

inne (skrypt ma przyjmować trzy argumenty: katalog, stare rozszerzenie, nowe rozszerzenie); należy uwzględnić możliwość pojawienia się w nazwie pliku kropek.

10. Sprawdzi, czy wybrany użytkownik (login podany jako argument wywołania skryptu) jest aktualnie zalogowany w systemie; w przypadku niepowodzenia skrypt powinien zaczekać na zalogowanie użytkownika i wtedy poinformować o ewentualnym powodzeniu.
11. Jako argumenty wywołania (dowolna liczba argumentów) będzie przyjmował loginy użytkowników systemu; skrypt ma sprawdzić, czy któryś z podanych użytkowników jest aktualnie zalogowany w systemie więcej niż raz; dla każdego takiego użytkownika system ma wyświetlić jego login oraz liczbę logowań.
12. Przyjmuje jako argument nazwę programu (na przykład **`nano`**); skrypt powinien monitorować system, wyświetlając co sekundę informację o tym, którzy użytkownicy używają w danej chwili programu o podanej jako argument nazwie.
13. Pobiera jako argumenty wywołania loginy użytkowników (może być podany więcej niż jeden login) i sprawdza, ile instancji edytora **`nano`** poszczególni użytkownicy mają uruchomione; w jednej linii mają zostać wyświetlone login i liczba uruchomionych edytorów.
14. Szyfruje plik za pomocą szyfru Cezara (przesunięcie liter alfabetu o stałą wartość); jako argument należy podać przesunięcie (liczba od 0 do 26) oraz nazwę pliku do zaszyfrowania; wynik powinien być zapisany do nowego pliku o takiej samej nazwie jak plik oryginalny, jednak z dodanym rozszerzeniem **`.cezar`** (użyj polecenia **`tr`**).
15. Będzie obliczał *n-ty* element ciągu Fibonacciego (*n* podawane jako argument wywołania skryptu); skrypt ma działać **rekurencyjnie** i wywoływać sam siebie, a na ekranie powinien pojawić się jedynie ostateczny wynik (bez wyników pośrednich).
16. Jest iteracyjną wersją skryptu z poprzedniego zadania.

Skorowidz

M

O

P

S

T

W

Z