A guide to modern UNIX systems

Daniël de Kok

A guide to modern UNIX systems

by Daniël de Kok

License

DRAFT

DRAFT

Table of Contents

DRAFT

List of Figures

List of Tables

Preface

The last few years UNIX-like operating systems have grown with leaps and bounds, both technically, and in popularity. Since the beginning of the nineties the driving force behind UNIX-like systems has been the Internet, and so-called “Open Source software”. Open source software can be distributed and modified freely, the Internet gave millions computer users the opportunity to download and contribute to UNIX-like operating systems.

Another important trend in UNIX development is the rise of full blown desktop environments like GNOME and KDE. Traditionally, the X Window System was used to do work that required graphical representation, such as previewing graphs. UNIX-like systems can now be used without ever using a shell. This made UNIX-like systems accessible to the mainstream public.

This book aims to provide an introduction to the underlying technology of UNIX, that has a tradition that goes back to the inception of UNIX. Understanding how UNIX works gives a user opportunities to use their systems more effectively. The text- and file-oriented utilities of the UNIX system are often more flexible and faster than their graphical counterparts. But it takes some time to get used to them.

Most material in this book applies to all UNIX-like systems. Unfortunately, there is a lot of variance in how well these systems conform to the *Single UNIX Specification* standard. When I wrote this book I have used three Open Source UNIX-like operating systems as a reference, namely CentOS 4.4, FreeBSD 6.0, and Solaris 10. All the examples in this book have been tested with these operating systems, and differences between these systems are discussed where deemed necessary.

I hope that you will have a lot of fun reading this book!

DRAFT

Chapter 1. Introduction

1.1. A short history of UNIX

UNIX is an operating system that was originally developed in the 1960s by a group of employees of the AT&T Bell Labs, including Ken Thompson and Dennis Ritchie. In 1973 it was decided to rewrite the UNIX assembly code to the C programming language that was also developed by Thompson and Ritchie. C had little assumptions about the underlying hardware, making UNIX a highly portable system. Besides that it made the code more readable for outsiders, accelerating the adoption of UNIX within AT&T, universities and for-profit firms.

Near the end of the 1970s an influential strain of UNIX development outside the Bell Labs started to flourish. The University of California at Berkeley had started producing their own distribution of the UNIX code, that required a UNIX license. This distribution was named after its home town: the Berkeley Software Distribution (BSD). Over the years, BSD contributed significantly to the development of UNIX. For instance, BSD introduced virtual memory, TCP/IP networking and the *Fast Filesystem* (FFS) that is still used in various forms in modern UNIX variants.

AT&T commercialized UNIX during the 1980s, turning it from a research operating system into a product that should bring in cash for AT&T. The commercial efforts resulted in the development of UNIX System III and UNIX System V. Many third parties licensed the commercial UNIX source code and made their own variants, including SCO UNIX, HP-UX and AIX. At the end of the 1980s the UNIX marketplace was dominated by many incompatible UNIX variants that were based on AT&T UNIX System III or V, and the BSD UNIX distribution that still contained to much AT&T code to be distributed freely. During this time Keith Bostic started a project to reimplement code that was AT&T licensed. As a result the University of California in Berkeley was able to release a free operating system that was nearly complete. AT&T did not agree that all their proprietary code was removed and sued BSDi, a company that had started selling a commercial version of the BSD code. The lawsuit was settled in 1994, after Novell bought UNIX from AT&T, in a way that was quite favorable to Berkeley. This was a blessing for two projects that have built a free BSD operating system, NetBSD and FreeBSD, as well as BSDi.

BSD would probably be the dominant free UNIX-like operating system if AT&T did not sue BSDi. But history followed another track: in 1991 a Finnish student, Linus Torvalds, started to work on a UNIX-like kernel as a hobby effort. This was an excellent match for the GNU project, a project started by Richard Stallman, that aimed to reimplement UNIX under a free license. The GNU system was almost complete, but it did not have a kernel. Torvalds did have a kernel, and soon the first GNU based operating systems with a Linux kernel were developed.

1.2. The current state of UNIX

Shortly after the acquisition of UNIX Novell sold it again. But they did not sell the UNIX business as one entity. The source code was sold to the Santa Cruz Operation (SCO), and the UNIX trademark was transferred to The Open Group. A product may carry the UNIX trademark if it complies with the Single UNIX Specification (SUS), and is certified by The Open Group. Some Open Source operating systems are probably almost compliant with one of the (older) SUS standards, but the certification procedure is simply to expensive to be worthwhile. We usually refer to these operating systems as UNIX-like operating systems. The NetBSD website describes the trademark problem pretty well [1]:

> If something looks like a duck, walks like a duck, and quacks like a duck, what is it? The answer, of course, depends on whether or not the name `duck' is a trademark! If it is, then the closest that something can get, without permission of the owner of the trademark, is `duck-like.'

When this book was written there were three dominant strains of UNIX-like operating systems:

- **GNU/Linux:** the GNU operating system with the Linux kernel.

[1] http://www.netbsd.org/Misc/call-it-a-duck.html

- **BSD:** operating systems that are derived from the Berkeley Software Distribution (BSD).
- **Solaris:** a System V derivate that is developed by Sun Microsystems.

1.3. UNIX philosophy

UNIX is one of the longest lasting operating systems that is in active use. Its basic principles did not change over the last thirty years, and made it widely loved. Doug McIlroy summarized the UNIX philosophy in three simple rules:

- Write programs that do one thing and do it well.
- Write programs to work together.
- Write programs to handle text streams, because that is a universal interface.

Even if you don't intend to write UNIX programs, there rules can mean a lot to you as a user of a UNIX-like system. Once you get to know a few simple UNIX commands, and learn how to combine them, you will be able to solve problems easily. Keep this in mind while you learn UNIX, and try get a feeling of how you can reduce complex problems to simple combined operations.

1.4. Free and Open Source software

As I mentioned earlier, both GNU/Linux and various BSD systems are free software. In this context, the word *free* means *freedom*, not gratis. The Free Software Foundation has defined four kinds of freedom that free software should have[2]:

- The freedom to run the program, for any purpose.
- The freedom to study how a program works, and adapt it to your needs.
- The freedom to redistribute copies of the program.
- The freedom improve the program, and release your improvements to the public.

With this list it is fairly easy to find out which software can be called “free software”. However, the term *free software* is not trademarked, and no procedure existed to check the freedom of a license centrally. Besides that, the free software paradigm was and is fairly ideological. This formed a barrier for enterprises to adopt free software.

The *Open Source Initiative* (OSI) was established in 1998 to solve both the image and the licensing problems. The OSI researches new free software licenses, and verifies that they conform to the *Open Source Definition*. After certification, a license is considered an Open Source license, meaning that it protects the freedoms of the user. The OSI also advocates usage and development of Open Source software, mostly on pragmatical grounds.

[2] http://www.gnu.org/philosophy/free-sw.html

Chapter 2. A whirlwind tour through UNIX

This chapter provides a short tour through UNIX. Do not worry if you do not understand all concepts yet. They will be explained in later chapters. The purpose of this chapter is to get you acquainted with UNIX, so that you will know your way around a bit.

2.1. Logging in

UNIX is a multi-user operating system. This means that more than one user can work simultaneously on a UNIX system. Each user has his own user account that is protected by a password. When you start to work on a UNIX system, you will be asked to identify yourself. This identification process is called *logging in*. The login prompt will look comparable to this:

```
CentOS release 4.2 (Final)
Kernel 2.6.9-22.0.2.EL on an i686

mindbender login:
```

You can often deduct information about a particular UNIX system from the login prompt. For instance, in the example listed above you can see that the system is running CentOS on a *i686* machine. The login prompt also displays the first component of the hostname of the system (*mindbender*). Some UNIX systems will also display the device name of the current terminal. You can log in by typing the username, and pressing the return key. The system will then ask your password, which you can enter in the same manner. On most UNIX systems the system does not provide any visual feedback when you type the password. This is done to prevent that an eavesdropper can see the length of your password.

If you use the X Window System, you can open up a shell by starting a terminal emulator like xterm, GNOME Terminal or Konsole.

2.2. Finding your way around

If the login was successful, the shell prompt will be shown. The shell prompt is usually displayed as a dollar sign (*$*) for normal users, and as a hash mark (*#*) for the root user. You can interact with the system by typing commands in the shell. A command is executed when you press the return key. UNIX commands are relatively easy to learn; you will see that they are often abbreviations of what they do.

Like most other operating systems, UNIX uses directories to store and categorize files. All UNIX processes, including the shell have a *working directory*. This is the directory to which relative path names are related. For example, if you specify that a command should open `work/finances/summary.txt`, the command will open `/home/daniel/work/finances/summary.txt` when `/home/daniel` is the current working directory.

UNIX has a simple command to print the current working directory, **ls**. It is simple to use:

```
$ pwd
/home/daniel
```

When you just logged in, and did not change the working directory, the working directory will always be of the form `/home/<username>`. Most users on a UNIX system have their own directory on the system where they can store files, named the *Ihome directory*. The home directory is the working directory after a user logged in.

The current working directory can be changed with the **ls** command. The new working directory can be specified as a parameter. For example:

```
$ cd /
```

This will change the current working directory to `/` this directory is also called the *root directory*, because it is the parent directory of all other directories. You can confirm that the working directory has indeed changed with the **pwd** command:

```
$ pwd
/
```

Without seeing what files and directories are available, browsing the filesystem can be daunting (although we will see later how you can use the shell to complete path and filenames automatically). UNIX provides the **ls** command to list files and directories. If **ls** is used without any arguments it will show the files and directories that are in the current working directory. For example:

```
$ ls
bin             lost+found      sbin
boot            media           selinux
dev             misc            srv
etc             mnt             sys
home            opt             tmp
initrd          proc            usr
lib             root            var
```

You can also add a directory name as a parameter to show the contents of that directory. For instance:

```
$ ls var
account         local           preserve
cache           lock            run
crash           log             spool
db              mail            tmp
empty           named           tux
gdm             nis             www
lib             opt             yp
```

The problem with this output is that there is no visual way to separate files and directories. This can be confusing if you are not familiar with the filesystem yet. The `-F` option of **ls** can assist here; this option adds a visual indication for various different file and directory types. The following table lists the indication characters as they are specified in the SUSv3 standard. Most modern UNIX operating systems adhere to these character conventions.

Table 2.1. Visual indicators for ls -F

Character	Entry type
/	Directory
*	Executable file
|	FIFO
@	Symbolic link

Many GNU/Linux distributions implicitly use this option, as well as coloring for different file types (which can be very disturbing). The following example lists the `/` directory on a CentOS Linux system:

```
$ ls -F
account/        local/          preserve/
cache/          lock/           run/
crash/          log/            spool/
db/             mail@           tmp/
empty/          named/          tux/
gdm/            nis/            www/
lib/            opt/            yp/
```

As you can see, there is one symbolic link (`mail`), all other entries are directories.

We have now looked at the basic commands that are needed to move around in the filesystem tree. If you ended up in a directory other than your home directory while experimenting with these commands, you can go to your home directory by executing **cd** without any parameters.

2.3. Working with files

Traditionally everything in UNIX has been represented as a file. Even hardware devices are represented as special files in the `/dev` directory. Because of its file-centric nature, UNIX has a lot of utilities to deal with files. In this section we will scratch the surface of some widely used file utilities.

Most files in UNIX are text files, but there are also many kinds of binary or special files. The **file** command can help you finding out of what kind a particular file is. The **ls** command that was described earlier is an executable binary file which is stored in the `/bin` directory. This is a good candidate for trying out **file**:

```
$ file /bin/ls
/bin/ls: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
for GNU/Linux 2.2.5, dynamically linked (uses shared libs), stripped
```

The output shows that `/bin/ls` is an executable file. If you use **file** to identify `/dev/null`, it will tell you that it is a character device:

```
$ file /dev/null>
/dev/null: character special (1/3)
```

If you have a text file you can print its contents to the screen with **cat** by providing the text file as a parameter:

```
$ cat hello.txt>
Hello world!
```

cat is a very simple, but versatile utility, that can be used for a lot more than just listing a file. **cat** will be described in more detail later in this book.

If you have a text file with more lines than what will fit on a text screen, or in a terminal emulator window, the text of a file will scroll too fast when you use **cat** to display a file. We can browse the text at our own pace with a so-called *paginator* like **more**. You can use **more** by sending the output of the **cat** command to **more**. Such a connection between two programs is named a *pipe*, and can be made with a vertical bar ("|"):

```
$ cat large_file.txt | more
```

In this case making a pipe is not really necessary, because you can also specify the file as a parameter to **more**. But the use of pipes will prove very useful in your daily shell usage.

You can also use **cat** to create simple text files. Let's use it to create a simple file with book titles in your home directory:

```
$ cat > ~/books.txt << EOF
Alice in wonderland
War and peace
Brave new world
EOF>
```

This command will write every line to the file `~/books.txt`, until it encounters a line that contains the *EOF* character string. The tilde character (~) is a symbol that represents your home directory. You can verify that the file was created with **cat**:

```
$ cat ~/books.txt
Alice in wonderland
War and peace
Brave new world
```

2.4. Organizing files

Now that you can create files, it is a good time to look how you can organize them better, by putting them in directories. Let's create a directory named `books` in your home directory. You can do this with the **mkdir** command:

```
$ mkdir ~/books
```

Let's go to this directory, and move the `books.txt` to this directory. You can use the **mv** command to move files (or directories), by specifying the source file as the first parameter, and the target directory as the second parameter. In this case, you could execute the following two commands:

```
$ cd ~/books
$ mv ../books.txt .
```

The second command may look a bit mysterious, because it uses two special symbols. Two dots (`..`) refer to the parent directory, a single dot (`.`) refers to current directory. So, if the current directory was `/home/daniel/books`, two dots will translate to `/home/daniel`, and a single dot to `/home/daniel/books`. So, the second command is equivalent to: **mv /home/daniel/books.txt /home/daniel/books**. The **mv** command is also useful to rename files. You can do this by using the current filename as the first argument, and the new filename as the second argument. Suppose that you want to rename the `books.txt` to `literature.txt`, you can do this with:

```
$ mv books.txt literature.txt
```

Now that you can move files, you may be curious how you can copy files. UNIX systems provide the **cp** command to copy files. The basic syntax of **cp** is identical to **mv**, you can specify the current file as the first argument, and the destination file or directory as the second argument. You can use the **mkdir** and **cp** commands to make a backup copy of your list of literature:

```
$ mkdir ~/backup
$ cp literature.txt ~/backup
```

Finally, you may want to delete a file or a directory. The **rm** command removes the files that are specified as arguments:

```
$ ls
literature.old.txt  literature.txt  movies.old.txt
$ rm literature.old.txt movies.old.txt
$ ls
literature.txt
```

You can also remove a directory by adding the *`-r`* as an argument, this will recursivily delete the files or directories that are provided as an argument. This means that directories are removed, including all the files and subdirectories that a directory holds. *`-r`* does not have a special meaning for files. In the following example, the `oldbooks` directory is removed from the home directory:

```
$ rm -r ~/oldbooks
```

2.5. Getting help

It will occasionally happen that you are looking for a specific command parameter. UNIX provides some facilities to help you. First of all, many commands on GNU/Linux accept the *`--help`* parameter. This parameter gives a short summary of the parameters that a command accepts. For instance:

```
$ mkdir --help
Usage: mkdir [OPTION] DIRECTORY...
Create the DIRECTORY(ies), if they do not already exist.
```

```
  -Z, --context=CONTEXT (SELinux) set security context to CONTEXT
        Mandatory arguments to long options are mandatory for short options too.
  -m, --mode=MODE   set permission mode (as in chmod), not rwxrwxrwx - umask
  -p, --parents     no error if existing, make parent directories as needed
  -v, --verbose     print a message for each created directory
      --help     display this help and exit
      --version  output version information and exit
```

Sometimes this does not provide enough information. The system also has manual pages for most commands. These manual pages can be read with the **man** command:

```
$ man mkdir
```

2.6. Logging out

After you have finished working with a UNIX system, you should log out to make sure that nobody else can use your account. You can log out by closing the login shell with **exit**.

At this point you should be able to log in to a UNIX system, and finding your way around. Almost all chapters deal with the UNIX shell; you can try the examples in these chapters by logging in to a UNIX system.

Chapter 3. The shell

3.1. Introduction

In this chapter we will look at the traditional working environment of UNIX systems: the shell. The shell is an interpreter that can be used interactively and non-interactively. When the shell is used non-interactively it functions as a simple, but powerful scripting language. The shell constructs for scripting will be described in a later chapter. In this chapter we will look at interactive use of the shell. An interactive shell can be used by a user to start programs.

Before we go any further, we have to warn you that most UNIX systems provide more than just one shell. There are two shell flavors that have become popular over time, the Bourne shell and the C shell. In this book we will describe Bourne shells that conform to the IEEE 1003.1 standard. The Bash (Bourne Again Shell) and ksh (Korn Shell) shells conform well to these standards. So, it is a good idea to use one of these two shells. You can easily see what shell the system is running by executing **echo $SHELL**. This is what a Bash shell may report:

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

If you are using a different shell, you can change your default shell. Before setting a different shell, you have to establish the full path of the shell. You can do this with the **which** command. For example:

```
$ which bash
/bin/bash
$ which ksh
/usr/bin/ksh
```

On this particular system, the full path to the bash shell is `/bin/bash`, and to the ksh shell `/usr/bin/ksh`. With this information, and the **chsh** command you change the default shell. The following example will set the default shell to bash:

```
$ chsh -s /bin/bash
Changing shell for daniel.
Password:
Shell changed.
```

The new shell will be activated after logging out from the current shell (with **logout** or **exit**), or by opening a new X terminal window if you are running X11.

3.2. Executing commands

An interactive shell is used to start programs by executing commands. There are two kinds of commands that a shell can start:

- *Built-in commands:* built-in commands are integrated in the shell. Commonly used built-in commands are: **cd**, **fg**, **bg**, and **jobs**.

- *External commands:* external commands are programs that are not part of the shell program, and are separately stored on the filesystem. Commonly used external commands are: **ls**, **cat**, **rm**, and **mkdir**.

As you have seen in the examples from Chapter 2, *A whirlwind tour through UNIX*, all commands are executed with the same syntax:

```
commandname [argument1 argument2 ... argumentn]
```

The number of arguments is arbitrary, and are always passed to the command. The command can decide what it does with these arguments.

All built-in commands can always be executed, because they are part of the shell. External commands can be executed by name when the program is in the search path of the shell. Otherwise, you will have to specify the path to the program. The search path of the shell is stored in a variable named *PATH*. A variable is a named piece of memory, of which the contents can be changed. We can see the contents of the PATH variable in the following manner:

```
$ echo $PATH
/usr/kerberos/bin:/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/home/daniel/bin
```

The directory paths in the *PATH* variable are separated with the colon (*:*) character. You can use the **which** command to check whether a given command is in the current shell path. You can do this by providing the command as an argument to **which**. For example:

```
$ which pwd
/bin/pwd
$ which sysstat
/usr/bin/which: no sysstat in (/usr/kerberos/bin:/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:
```

If a program is not in the path, you can still run it by entering its absolute or relative path.

3.3. Moving around

It is often necessary to jump through various parts of a line, and to alter it, when you are editing larger commands. Both **bash** and **ksh** have keyboard shortcuts for doing common operations. There are two shell modes, in which the shortcut keys differ. These modes correspond with two popular editors for UNIX in their behavior. These editors are vi and EMACS. In this book we will only cover the EMACS-like keystrokes. You can check in which mode a shell is running by printing the *SHELLOPTS* variable. In the first example the shell is used in *emacs* mode, in the second example the *vi* mode is used. You identify the mode by looking for the *emacs* or *vi* strings in the contents of the variable.

```
$ echo $SHELLOPTS
braceexpand:emacs:hashall:histexpand:history:interactive-comments:monitor
```

```
$ echo $SHELLOPTS
braceexpand:hashall:histexpand:history:interactive-comments:monitor:vi
```

If your shell is currently using the *vi* mode, you can switch to the *emacs* mode by setting the *emacs* option:

```
$ set -o emacs
```

With the *emacs* editing mode enabled, you can start using shortcuts. We will look at three kinds of shortcuts: character editing shortcuts, word editing shortcuts, and line editing shortcuts. Later in this chapter, we will also have a look at some shortcuts that are used to retrieve entries from the command history.

Character editing

The first group of shortcuts have characters as their logic unit, meaning that they allow command line editing operations on characters. Table 3.1, “Moving by character” provides an overview of the shortcuts that are used to move through a line by character.

Table 3.1. Moving by character

Keys	Description
Ctrl-b	Move a character backwards.
Ctrl-f	Move a character forward.

These shortcuts are simple, and don't do anything unexpected. Suppose that you have typed the following line:

```
find ~/music -name '*.ogg' - -print
```

The cursor will be at the end. You can now move to the start of the line by holding *Ctrl-b*:

```
find ~/music - -name '*.ogg' -print
```

Likewise, you can go back again to the end by holding *Ctrl-f*. There is an error in this line, since there is one erroneous dash. To remove this dash, you can use one of the character deletion shortcuts.

Table 3.2. Deleting characters

Keys	Description
Ctrl-h	Delete a character before the cursor. This has the same effect as using the Backspace key on most personal computers.
Ctrl-d	Delete the character the cursor is on.

You can delete the dash in two manners. The first way is to move the cursor to the dash:

```
find ~/music - -name '*.ogg' -print
```

and then press *Ctrl-d* twice. This will delete the dash character, and the space that follows the dash:

```
find ~/music -name '*.ogg' -print
```

The other approach is to position the cursor on the space after the dash:

```
find ~/music -  -name '*.ogg' -print
```

and then press *Ctrl-h* twice to delete the two preceding characters, namely the dash and the space before the dash. The result will be the same, except that the cursor will not move:

```
find ~/music -name '*.ogg' -print
```

One of the nice features of most modern shells is that you can transpose (swap) characters. This is handy if you make a typing error in which two characters are swapped. Table 3.3, “Swapping characters” lists the shortcut for transposing characters.

Table 3.3. Swapping characters

Keys	Description
Ctrl-t	Swap (transpose) the characters the cursor is on, and the character before the cursor. This is handy for quickly correcting typing errors.

Suppose that you have typed the following command:

```
cat myreport.ttx
```

The extension contains a typing error if you intended to **cat** `myreport.txt`. This can be corrected with the character transpose shortcut. First move to the second character of the pair of characters that are in the wrong order:

```
cat myreport.ttx
```

You can then press *Ctrl-t*. The characters will be swapped, and the cursor will be put behind the swapped characters:

```
cat myreport.txt
```

Word editing

It if often tedious to move at character level. Fortunately the Korn and Bash shells can also move through lines at a word level. Words are sequences of characters that are separated by a special character, such as a space. Table 3.4, “Moving by word” summarizes the shortcuts that can be used to navigate through a line by word.

Table 3.4. Moving by word

Keys	Description
Esc b	Move back to the start of the current or previous word.
Esc f	Move forward to the last character of the current or next word.

As you can see the letters in these shortcuts are equal to those of moving forward and backwards by character. The movement logic is a bit curious. Moving forward puts the cursor to the end of the current word, not to the first character of the next word as you may have predicted. Let's look at a quick example. In the beginning the cursor is on the first character of the line.

```
find ~/music -name '*.ogg' -print
```

Pressing *Esc f* will move the cursor behind the last character of the first word, which is *find* in this case:

```
find ~/music -name '*.ogg' -print
```

Going forward once more will put the cursor behind *~/music*:

```
find ~/music -name '*.ogg' -print
```

Backwards movement puts the cursor on the first character of the current word, or on the first character of the previous word if the cursor is currently on the first character of a word. So, moving back one word in the previous example will put the cursor on the first letter of “music”:

```
find ~/music -name '*.ogg' -print
```

Deleting words works equal to moving by word, but the characters that are encountered are deleted. Table 3.5, “Deleting words” lists the shortcuts that are used to delete words.

Table 3.5. Deleting words

Keys	Description
Alt-d	Delete the word, starting at the current cursor position.
Alt-Backspace	Delete every character from the current cursor position to the first character of a wordt that is encountered. position

Finally, there are some shortcuts that are useful to manipulate words. These shortcuts are listed in Table 3.6, “Modifying words”.

Table 3.6. Modifying words

Keys	Description
Alt-t	Swap (transpose) the current word with the previous word.
Alt-u	Make the word uppercase, starting at the current cursor position.
Alt-l	Make the word lowercase, starting at the current cursor position.
Alt-c	Capitalize the current word character or the next word character that is encountered.

Transposition swaps words. If normal words are used, it's behavior is predictable. For instance, if we have the following line with the cursor on “two”

```
one two three
```

Word transposition will swap “two” and “one”:

```
two one three
```

But if there are any non-word characters, the shell will swap the word with the previous word while preserving the order of non-word characters. This is very handy for editing arguments to commands. Suppose that you made an error, and mixed up the file extension you want to look for, and the *print* parameter:

```
find ~/music -name '*.print' -ogg
```

You can fix this by putting the cursor on the second faulty word, in this case “ogg”, and transposing the two words. This will give the result that we want:

```
find ~/music -name '*.ogg' -print
```

Finally, there are some shortcuts that change the capitalization of words. The *Alt-u* shortcut makes all characters uppercase, starting at the current cursor position till the end of the word. So, if we have the lowercase name “alice”, uppercasing the name with the cursor on “i” gives “alICE”. *Alt-l* has the same behavior, but changes letters to lowercase. So, using *Alt-l* on “alICE” with the cursor on “I” will change the string to “alice”. *Alt-c* changes just the character the cursor is on, or the next word character that is encountered, to uppercase. For instance, pressing *Alt-c* with the cursor on “a” in “alice” will yield “Alice”.

Line editing

The highest level we can edit is the line itself. Table 3.7, “Moving through lines” lists the two movement shortcuts.

Table 3.7. Moving through lines

Keys	Description
Ctrl-a	Move to the beginning of the current line.
Ctrl-e	Move to the end of the current line.

Suppose that the cursor is somewhere halfway a line:

```
find ~/music -name '*.ogg' - -print
```

Pressing Ctrl-e once will move the cursor to the end of the line:

```
find ~/music -name '*.ogg' - -print
```

Pressing Ctrl-a will move the cursor to the beginning of the line:

```
find ~/music - -name '*.ogg' -print
```

You can also delete characters by line level. The shortcuts are listed in Table 3.8, “Deleting lines”. These shortcuts work like movement, but deletes all characters that are encountered. Ctrl-k will delete the character the cursor is on, but Ctrl-x Backspace will not. Moving to the beginning of the line with Ctrl-a, followed by Ctrl-k, is a fast trick to remove a line completely.

Table 3.8. Deleting lines

Keys	Description
Ctrl-k	Delete all characters in the line, starting at the cursor position.
Ctrl-x Backspace	Delete all characters in the line up till the current cursor position.

3.4. Command history

Both the **bash** and **ksh** shells keep a history of the commands that you have executed during the current session. Using commands from the history can save a lot of typing, even if the command needs slight alteration.

You can show the command history with the built-in **history** command:

```
$ history
    1  pwd
    2  uname -a
    3  history
```

You can execute an history item with the exclamation mark character (!), and the number of the item. For example, to re-run **uname -a**, you could execute:

```
$ !1
```

Table 3.9. Shell history browsing

Keys	Description
Ctrl-p/Up	Move one step back in the command history.
Ctrl-n/Down	Move one step forward in the command history.
Ctrl-r	Reverse search the command history.
Ctrl-s	Search forward in the command history.

Reading the history list everytime you want to pick something from the history is a bit tedious. Fortunately, there are some quick shortcuts that make life a bit easier. The most important shortcuts are listed in Table 3.9, “Shell history browsing”. The most basic movements are moving backward and forward. Pressing the *Ctrl-p* shortcut will show the previous command that was executed. Pressing *Ctrl-p* the command that was executed before the previous command, et cetera. *Ctrl-n* will do the reverse, and will show the next command in the history.

The two search shortcuts help you to browse through the history quickly. *Ctrl-r* searches backwards, thus starting at the commands that were executed last, working up to the oldest command that is still in the history. *Ctrl-s* searches forward, starting at the oldest commands in the history. Once you have pressed one of both shortcuts, you can start typing a part of the command that you are searching. As you can see, the search is incremental, meaning that the result will automatically be updated when you add characters. Once you have found the command that you were looking for, you can press *Enter* to execute the command, or *Esc* to go back to the command line to edit the command.

3.5. Completion

One of the purposes of the shell is to provide functionality that helps you avoiding doing tedious work. One of the nicest features that speeds up working with the shell is completion. Completion tries to guess what file, command, or variable you are referring to when you call the completion functionality. This works threefold:

- If the word being completed is a command, the shell will look up commands that start with the characters that you have typed in the directories that are specified in the *PATH* variable.
- If the word being completed is a file or directory name, the shell will search for files that start with the characters that you have typed in the current directory, or in an other directory if you use an absolute path.
- If the work being completed is a variable name (meaning that it starts with the *$* sign), the shell will look up variables that start with the characters that you have typed.

How the completion is done, is dependend on what you try to complete. You can call the shell completion by pressing the *Tab* key after typing some characters. If only one possible completion was found, the shell will fill in this completion. If there is more than one possible completion, the shell will fill in the completion up till the point that the completions are equal. For instance, if there is a directory named `resources`, and one named `responses`, typing *r<Tab>* will fill in *res*. Pressing *Tab* two times more will show all possible completions.

Let's look at some examples of filename completion. Suppose that we have three files in the current directory: `questions.txt`, `resources.txt`, and `responses.txt`. Now, suppose that you want to have a look at the contents of `questions.txt`. This can easily be done without typing the whole filename, by typing

```
$ less q
```

Pressing *Tab* will complete the filename, and fills in the filename:

```
$ less questions.txt
```

That saved a lot of typing! Now, what happens if we want to view `responses.txt`. At first, you type

```
$ less r
```

And press *Tab*. Since there are two file names that start with the "r" character, the shell will fill in the shared characters of both filenames:

```
$ less res
```

To see what possible completions there are, we can press *Tab* twice:

```
$ less res
resources.txt  responses.txt
```

We can continue the completion by typing the next character, in this case "p", and pressing tab:

```
$ less resp<tab>
$ responses.txt
```

Completion can be used in the same manner for commands and variable names. UNIX commands are often very short, so completion is often not useful for entering commands. Notable exceptions are many programs that use a graphical user interface. For instance, if we want to complete to **gnome-system-monitor**, we could do this in the following steps:

```
$ gno<tab>
```

Which, completes to *gnome*. It is usually a good idea to start off with three letters, because there are many program names that start with a *g* or *gn*. Most programs that are part of the GNOME desktop environment have their command names start with *gnome-*, so it is a good idea to provide a bit more context, and attempt to complete the command again:

```
$ gnome-s<tab><tab>
gnome-search-tool         gnome-session-remove      gnome-sound-properties
```

```
gnome-session             gnome-session-save        gnome-sound-recorder
gnome-session-properties  gnome-smproxy             gnome-system-monitor
```

We are now down to a handful of choices. Adding *y* is enough to complete the command:

```
$ gnome-sy<tab>
$ gnome-system-monitor
```

As you can see in these examples, it may take some time to get used to filename completion. But once you get a feeling for it, it will save you from many keystrokes and potential typing errors.

3.6. Aliases

Shell users often use certain commands frequently. If these commands are tedious to type every time, you can make aliases for these commands. For instance, I often want to have an overview of the Ogg files that are in my home directory. This can be done with:

```
$ find ~ -name '*.ogg'
```

Now, wouldn't it be simpler if this can just be done with a simple command, say **oggs**. This is where the alias functionality of the shell comes in. An alias can be added with the **alias** command:

```
$ alias oggs="find ~ -name '*.ogg'"
```

Now, let's give this newly created alias a try:

```
$ oggs
/home/daniel/Music/song1.ogg
```

Right, **alias** did its work. It is good to know that some UNIX systems have predefined aliases. You can see these predefined, and your own aliases with the *-p* parameter of **alias**:

```
$ alias -p
alias l.='ls -d .* --color=tty'
alias ll='ls -l --color=tty'
alias ls='ls --color=tty'
alias mc='. /usr/share/mc/bin/mc-wrapper.sh'
alias oggs='find ~ -name '\''*.ogg'\'''
alias vi='vim'
alias which='alias | /usr/bin/which --tty-only --read-alias --show-dot
--show-tilde'
```

As you can see, there are some handy macros for **vi**, as well as some plumbing to make certain commands work differently (e.g. executing **vi** will launch the **vim** editor).

3.7. Executing shell commands upon shell invocation

It is often handy to execute some commands when the shell is invoked. For instance, to set up aliases that you would like to use, or to modify the *PATH* variable. The files that are executed depends on the shell that you use, but both **bash** and **ksh** will execute `/etc/profile` and `~/.profile` when a login shell is started. Other shells that are executed as a child of the login shell usually inherit settings that were made in a login shell. **bash** also tries to execute `~/.bashrc` if the shell is not a login shell. But Bash-specific initialization files should be used with care, because they go against the principle of portability.

The initialization files are “shell scripts”. Shell scripts are files in which you can stash commands, and effectively allows you to do simple programming. For now, it is enough to know that you can just add commands, like the alias command that we have used above, or to set the default editing mode, as explained in Section 3.3, “Moving around”.

Chapter 4. Filesystem

Now that you have seen how to move around in the shell, it is time for more exiting stuff, the UNIX filesystem.

4.1. Some theory

Before we move on to look at practical filesystem operations, we are going to look at a more theoretical overview of how filesystems on UNIX-like systems work. The various UNIX implementations and clones provide many different filesystems, but all these filesystems use virtually the same semantics. These semantics are provided through the *Virtual Filesystem* (VFS) layer), giving a generic layer for disk and network filesystems.

inodes, directories and data

The filesystem consists of two types of elements: data and meta-data. The metadata describes the actual data blocks that are on the disk. Modern UNIX filesystems use information nodes (inodes) to provide store metadata. Most filesystems store the following data in their inodes:

Table 4.1. Common inode fields

Field	Description
mode	The file permissions.
uid	The user ID of the owner of the file.
gid	The group ID of the group of the file.
size	Size of the file in bytes.
ctime	File creation time.
mtime	Time of the last file modification.
links_count	The number of links pointing to this inode.
i_block	Pointers to data blocks

If you are not a UNIX afficiendo, these names will probably sound bogus to you, but we will clear them up in the following sections. At any rate, you can probably deduct the relation between inodes and data from this table, and specifically the *i_block* field: every inode has pointers to the data blocks that the inode provides information for. Together, the inode and data blocks are the actual file on the filesystem.

You may wonder by now where the names of files (and directories) reside, since there is no file name field in the inode. Actually, the names of the files are separated from the inode and data blocks, which allows you to do groovy stuff, like giving the same file more than one name. The filenames are stored in so-called directory entries. These entries specify a filename and the inode of the file. Since directories are also represented by inodes, a directory structure can also be constructed in this manner.

We can simply show how this all works by illustrating what the kernel does when we execute the command **cat /home/daniel/note.txt**

1. The system reads the inode of the `/` directory, checks if the user is allowed to access this inode, and reads the data block to find the inode number of the `home` directory.

2. The system reads the inode of the `home` directory, checks if the user is allowed to access this inode, and reads the data block to find the inode number of the `daniel` directory.

3. The system reads the inode of the `home` directory, checks if the user is allowed to access this inode, and reads the data block to find the inode number of the `note.txt` file.

4. The system reads the inode of the `notice.txt` file, checks if the user is allowed to access this inode, and returns the data blocks to **cat** through the *read()* system call.

File permissions

As I have described earlier, UNIX is a multi-user system. This means that each user has his/her own files (that are usually located in the home directory). Besides that users can be members of a group, which may give the user additional privileges.

As you have seen in the inode field table, every file has a owner and a group. Traditional UNIX access control gives read, write, or executable permissions to the file owner, file group, and other users. These permissions are stored in the *mode* field of the inode. The mode field represents the file permissions as a four digit octal number. The first digit represents stores some special options, the second digit stores the owner permissions, the third the group permissions, and the fourth the permissions for other users. The permissions are established by digit by using or adding one of the number in Table 4.2, “Meaning of numbers in the mode octet”

Table 4.2. Meaning of numbers in the mode octet

Number	Meaning
1	Execute (x)
2	Write (w)
4	Read (r)

Now, suppose that a file has mode *0644*, this means that the file is readable and writable by the owner (*6*), and readable by the file group (*4*) and others (*4*).

Most users do not want to deal with octal numbers, so that is why many utilities can also deal with an alphabetic representation of file permissions. The letters that are listed in Table 4.2, “Meaning of numbers in the mode octet” between parentheses are used in this notation. In the following example information about a file with *0644* permissions is printed. The numbers are replaced by three rwx triplets (the first character can list special mode options).

```
$ ls -l note.txt
-rw-r--r--  1 daniel daniel 5 Aug 28 19:39 note.txt
```

Over the years these traditional UNIX permissions have proven not to be sufficient in some cases. The POSIX 1003.1e specification aimed to extend the UNIX access control model with *Access Control Lists* (ACLs). Unfortunately this effort stalled, though some systems (like GNU/Linux) have implemented ACLs[1]. Access control lists follow the same semantics as normal file permissions, but give you the opportunity to add *rwx* triplets for additional users and groups.

The following example shows the access control list of a file. As you can see, the permissions look like normal UNIX permissions (the access rights for the user, group, and others are specified). But there is also an additional entry for the user *joe*.

[1] At the time of writing, ACLs were supported on ext2, ext3, and XFS filesystems

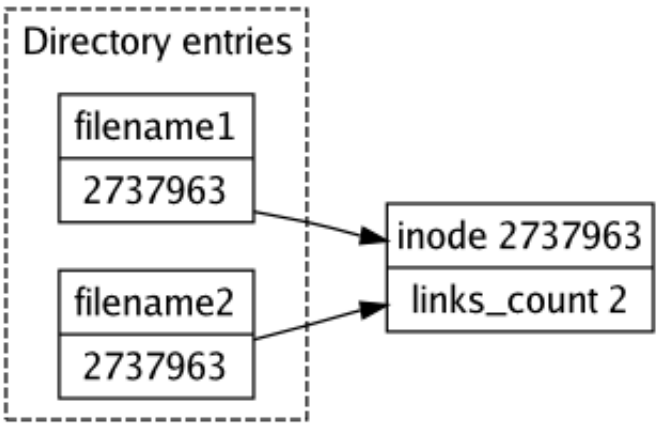
```
user::rwx
user:joe:r--
group::---
mask::r--
other::---
```

To make matters even more complex (and sophisticated), some GNU/Linux systems add more fine-grained access control through mandatory Access Control Frameworks (MAC) like SELinux and AppArmor. But these access control frameworks are beyond the scope of this book.

Links

A directory entry that points to an inode is named a *hard link*. Most files are only linked once, but nothing holds you from linking a file twice. This will increase the *links_count* field of the inode. This is a nice way for the system to see which inodes and data blocks are free to use. If links_count is set to zero, the inode is not referred to anymore, and can be reclaimed.

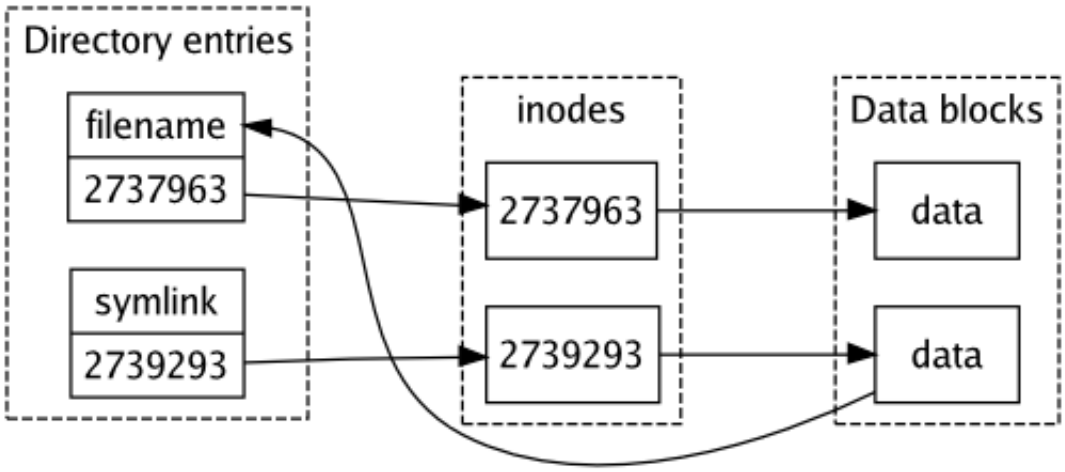
Figure 4.1. The structure of a hard link



Hard links have two limitations. First of all, hard links can not interlink between filessystems, since they point to inodes. Every filesystem has its own inodes and corresponding inode numbers. Besides that, most systems do not allow you to create hard links to directories (as defined in the Single UNIX Specification version 3). Allowing creation of hard links to directories could produce directory loops, potentially leading deadlocks and filesystem inconsistencies. In addition to that, most implementations of **rm** and **rmdir** do not know how to deal with such extra directory hard links.

Symbolic links do not have these limitations, because they point to file names, rather than inodes. When the symbolic link is used, the operating system will follow the path to that link. Symbolic links can also refer to a file that does not exist, since it just contains a name. Such links are called dangling links.

Figure 4.2. The structure of a symbolic link



Note

If you ever get into system administration, it is good to be aware of the security implications of hard links. If the `/home` directory is on the same filesystem as any system binaries, a user will be able to create hard links to binaries. In the case that a vulnerable program is upgraded, the link in the user's home directory will keep pointing to the old program binary, effectively giving the user continuing access to a vulnerable binary.

For this reason it is a good idea to put any directories that users can write to on different filesystems. In practice, this means that it is a good idea to put at least `/home` and `/tmp` on separate filesystems.

4.2. Analyzing files

Before jumping going some more adventurous venues, we will start doing the occasional stuff with files. This section will expand a little upon whirlwind chapter, so most of it will seem familiar.

Listing files

One of the most common things that you will want to do is to list all or certain files. The **ls** command serves this purpose very well. Using **ls** without any arguments will show the contents of the actual directory:

```
$ ls
dns.txt  network-hosts.txt  papers
```

If you use a GNU/Linux distribution, you may also see some fancy coloring based on the type of file. The standard output is handy to skim through the contents of a directory, but if you want more information, you can use the `-l` parameter. This provides a so-called long listing for each file:

```
$ ls -l
total 36
-rw-rw-r--  1 daniel daniel 12235 Sep  4 15:56 dns.txt
```

```
-rw-rw-r--  1 daniel daniel  7295 Sep  4 15:56 network-hosts.txt
drwxrwxr-x  2 daniel daniel  4096 Sep  4 15:55 papers
```

This gives a lot more information about the three directory entries that we have found with **ls**. The first column shows the file permissions. The line that shows the `papers` entry starts with a “d”, meaning that this entry represents a directory. The second column shows the number of hard links pointing to the inode that a directory entry points to. If this is higher than 1, there is some other filename for the same file. Directory entries usually have at least two hard links, namely the link in the parent directory and the link in the directory itself (each directory has a `.` entry, which refers to the directory itself). The third and the fourth columns list the file owner and group respectively. The fifth column contains the file size in bytes. The sixth column the last modification time and date of the file. And finally, the last column shows the name of this entry.

Files that start with a period (.) will not be shown by most applications, including **ls**. You can list these files too, by adding the *-a* option to **ls**:

```
$ ls -la
total 60
drwxrwxr-x   3 daniel daniel  4096 Sep 11 10:01 .
drwx------  88 daniel daniel  4096 Sep 11 10:01 ..
-rw-rw-r--   1 daniel daniel 12235 Sep  4 15:56 dns.txt
-rw-rw-r--   1 daniel daniel  7295 Sep  4 15:56 network-hosts.txt
drwxrwxr-x   2 daniel daniel  4096 Sep  4 15:55 papers
-rw-rw-r--   1 daniel daniel     5 Sep 11 10:01 .settings
```

As you can see, three more entries have appeared. First of all, the `.settings` file is now shown. Besides that you can see two additional directory entries, `.` and `..`. These represent the current directory and the parent directory respectively.

Earlier in this chapter (the section called “inodes, directories and data”) we talked about inodes. The inode number that a directory entry points to can be shown with the *-i* parameter. Suppose that I have created a hard link to the inode that points to the same inode as `dns.txt`, they should have the same inode number. The following **ls** output shows that this is true:

```
$ ls -i dns*
3162388 dns-newhardlink.txt
3162388 dns.txt
```

Determining the type of a file

Sometimes you will need some help to determine the type of a file. This is where the **file** utility becomes handy. Suppose that I find a file named `HelloWorld.class` somewhere on my disk. I suppose that this is a file that holds Java bytecode, but we can use **file** to check this:

```
$ file HelloWorld.class
HelloWorld.class: compiled Java class data, version 49.0
```

That is definitely Java bytecode. **file** is quite smart, and handles most things you throw at it. For instance, you could ask it to provide information about a device node:

```
$ file /dev/zero
/dev/zero: character special (1/5)
```

Or a symbolic link:

```
$ file /usr/X11R6/bin/X
/usr/X11R6/bin/X: symbolic link to `Xorg'
```

If you are rather interested in the file `/usr/X11R6/bin/X` links to, you can use the `-L` option of **file**:

```
$ file -L /usr/X11R6/bin/X
/usr/X11R6/bin/X: setuid writable, executable, regular file, no read permission
```

You may wonder why **file** can determine the file type relatively easy. Most files start of with a so-called *magic number*, this is a unique number that tells programs that can read the file what kind of file it is. The **file** program uses a file which describes many file types and their magic numbers. For instance, the magic file on my system contains the following lines for Java compiled class files:

```
# Java ByteCode
# From Larry Schwimmer (schwim@cs.stanford.edu)
0       belong          0xcafebabe      compiled Java class data,
>6      beshort x       version %d.
>4      beshort x       \b%d
```

This entry says that if a file starts with a long (32-bit) hexadecimal magic number *0xcafebabe*[2], it is a file that holds "compiled Java class data". The short that follows determines the class file format version.

File integrity

While we will look at more advanced file integrity checking later, we will have a short look at the **cksum** utility. **cksum** can calculate a cyclic redundancy check (CRC) for an input file. This is a mathematically sound method for calculating a unique number for a file. You can use this number to check whether a file is unchanged (for example, after downloading a file from a server). You can specify the file to calculate a CRC for as a parameter to **cksum**, and **cksum** will print the CRC, the file size in bytes, and the file name:

```
$ cksum myfile
1817811752 22638 myfile
```

Some systems also provide utilities for calculating checksums based on one-way hashes (for instance MD5 or SHA-1), but these utilities are not standardized in the The Single UNIX Specification version 3, and not consistent across different implementations.

[2] Yeah, you can be creative with magic numbers too!

Viewing files

Since most files on UNIX systems are usually text files, they are easy to view from a character-based terminal or terminal emulator. The most primitive way of looking at the contents of a file is by using **cat**. **cat** reads files that were specified as a parameter line by line, and will write the lines to the standard output. So, you can write the contents of the file `note.txt` to the terminal with **cat note.txt**. While some systems and most terminal emulators provide support for scrolling, this is not a practical way to view large files. You can pipe the output of **cat** to the **more** paginator:

```
$ cat note.txt | more
```

or let **more** read the file directly:

```
$ more note.txt
```

The **more** paginator lets you scroll forward and backward through a file. Table 4.3, “more command keys” provides an overview of the most important keys that are used to control **more**

Table 4.3. more command keys

Key	Description
j	Scroll forward one line.
k	Scroll backwards one line.
f	Scroll forward one screen full of text.
b	Scroll backwards one screen full of text.
q	Quit more.
g	Jump to the beginning of the file.
G	Jump to the end of the file.
/*pattern*	Search for the regular expression *pattern*.
n	Search for the next match of the previously specified regular expression.
m*letter*	Mark the current position in the file with *letter*.
'*letter*	Jump to the mark *letter*

The command keys that can be quantized can be prefixed by a number. For instance *11j* scrolls forward eleven lines, and *3n* searches the third match of the previously specified regular expression.

Some UNIX-like systems (most notably GNU/Linux), provide an alternative to **more**, named “less” (as in less is more). We will not go into *less* here, but its basic use is identical to **more**.

File and directory sizes

The **ls -l** output that we have seen earlier provides information about the size of a file. While this usually provides enough information about the size of files, you might want to gather information about collections of files or directories. This is where the du (1) command comes in. By default, du (1) prints the file size per directory. For example:

```
$ du ~/qconcord
72      /home/daniel/qconcord/src
24      /home/daniel/qconcord/ui
132     /home/daniel/qconcord
```

The size units may differ per operating system. The Single UNIX Specification version 3 requires that by default file sizes should be printed as 512 byte units. But various implementations, like GNU du use 1024 byte units. You can explicitly specify that du (1) should use 1024 byte units by adding the *-k* flag:

```
$ du -k ~/qconcord
72      /home/daniel/qconcord/src
24      /home/daniel/qconcord/ui
132     /home/daniel/qconcord
```

If you would also like to see per-file disk usage, you can add the *-a* flag:

```
$ du -k -a ~/qconcord
8       /home/daniel/qconcord/ChangeLog
8       /home/daniel/qconcord/src/concordanceform.h
8       /home/daniel/qconcord/src/textfile.cpp
12      /home/daniel/qconcord/src/concordancemainwindow.cpp
12      /home/daniel/qconcord/src/concordanceform.cpp
8       /home/daniel/qconcord/src/concordancemainwindow.h
8       /home/daniel/qconcord/src/main.cpp
8       /home/daniel/qconcord/src/textfile.h
72      /home/daniel/qconcord/src
12      /home/daniel/qconcord/Makefile
16      /home/daniel/qconcord/ui/concordanceformbase.ui
24      /home/daniel/qconcord/ui
8       /home/daniel/qconcord/qconcord.pro
132     /home/daniel/qconcord
```

You can also use the name of a file or a wildcard as a parameter. But this will not print the sizes of files in subdirectories, unless *-a* is used:

```
$ du -k -a ~/qconcord/*
8       /home/daniel/qconcord/ChangeLog
12      /home/daniel/qconcord/Makefile
8       /home/daniel/qconcord/qconcord.pro
8       /home/daniel/qconcord/src/concordanceform.h
8       /home/daniel/qconcord/src/textfile.cpp
12      /home/daniel/qconcord/src/concordancemainwindow.cpp
```

```
12      /home/daniel/qconcord/src/concordanceform.cpp
8       /home/daniel/qconcord/src/concordancemainwindow.h
8       /home/daniel/qconcord/src/main.cpp
8       /home/daniel/qconcord/src/textfile.h
72      /home/daniel/qconcord/src
16      /home/daniel/qconcord/ui/concordanceformbase.ui
24      /home/daniel/qconcord/ui
```

If you want to see the total sum of the disk usage of the files and subdirectories that a directory holds, use the `-s` flag:

```
$ du -k -s ~/qconcord
132     /home/daniel/qconcord
```

4.3. Working with directories

After having a bird's eye view of directories in the section called “inodes, directories and data”, we will have a look at some directory-related commands.

Listing directories

The ls (1) command that we have looked at in the section called “Listing files” can also be used to list directories in various ways. As we have seen, the default ls (1) output includes directories, and directories can be identified using the first output column of a long listing:

```
$ ls -l
total 36
-rw-rw-r--  1 daniel daniel 12235 Sep  4 15:56 dns.txt
-rw-rw-r--  1 daniel daniel  7295 Sep  4 15:56 network-hosts.txt
drwxrwxr-x  2 daniel daniel  4096 Sep  4 15:55 papers
```

If a directory name, of if wildcards are specified, ls (1) will list the contents of the directory, of the directories that match the wildcard. For example, if there is a directory `papers`, **ls paper*** will list the contents of this directory `paper`. This is often annoying if you would just like to see the matches, and not the contents of the matching directories. The `-d` avoid that this recursion happens:

```
$ ls -ld paper*
drwxrwxr-x  2 daniel daniel  4096 Sep  4 15:55 papers
```

You can also recursively list the contents of a directory, and its subdirectory with the `-R` parameter:

```
$ ls -R
.:
dns.txt  network-hosts.txt  papers

./papers:
cs  phil
```

```
./papers/cs:
entr.pdf

./papers/phil:
logics.pdf
```

Creating and removing directories

UNIX provides the mkdir (1) command to create directories. If a relative path is specified, the directory is created in the current active directory. The basic syntax is very simple: *mkdir <name>*, for example:

```
$ mkdir mydir
```

By default, mkdir (1) only creates one directory level. So, if you use mkdir (1) to create `mydir/mysubdir`, mkdir (1) will fail if `mydir` does not exist already. If you would like to create both directories at one, use the *-p* parameter:

```
$ mkdir -p mydir/mysubdir
```

rmdir (1) removes a directory. Its behavior is comparable to mkdir (1) . **rmdir mydir/mysubdir** removes `rmdir mydir/mysubdir`, while **rmdir -p mydir/mysubdir** removes `mydir/mysubdir` and then `mydir`.

If a subdirectory that want to remove contains directory entries, rmdir (1) will fail. If you would like to remove a directory, including all its contents, use the rm (1) command instead.

4.4. Managing files and directories

Copying

Files and directories can be copied with the cp (1) command. In its most basic syntax the source and the target file are specified. The following example will make a copy of `file1` named `file2`:

```
$ cp file1 file2
```

It is not surprising that relative and absolute paths do also work:

```
$ cp file1 somedir/file2
$ cp file1 /home/joe/design_documents/file2
```

You can also specify a directory as the second parameter. If this is the case, cp (1) will make a copy of the file in that directory, giving it the same file name as the original file. If there is more than one parameter, the last parameter will be used as the target directory. For instance

```
$ cp file1 file2 somedir
```

will copy both `file1` and `file2` to the directory `somedir`. You can not copy multiple files to one file. You will have to use cat (1) instead:

```
$ cat file1 file2 > combined_file
```

You can also use cp (1) to copy directories, by adding the *-R*. This will recursively copy a directory and all its subdirectories. If the target directory exists, the source directory or directories will be placed under the target directory. If the target directory does not exist, it will be created if there is only one source directory.

```
$ cp -r mytree tree_copy
$ mkdir trees
$ cp -r mytree trees
```

After executing these commands, there are two copies of the directory `mytree`, `tree_copy` and `trees/mytree`. Trying to copy two directories to a nonexistent target directory will fail:

```
$ cp -R mytree mytree2 newdir
usage: cp [-R [-H | -L | -P]] [-f | -i] [-pv] src target
       cp [-R [-H | -L | -P]] [-f | -i] [-pv] src1 ... srcN directory
```

Note

Traditionally, the *-r* has been available on many UNIX systems to recursively copy directories. However, the behavior of this parameter can be implementation-dependent, and the Single UNIX Specification version 3 states that it may be removed in future versions of the standard.

When you are copying files recursively, it is a good idea to specify the behavior of what cp (1) should do when a symbolic link is encountered explicitly, because the Single UNIX Specification version 3 does not specify how they should be handled by default. If *-P* is used, symbolic links will not be followed, effectively copying the link itself. If *-H* is used, symbolic links specified as a parameter to cp (1) may be followed, depending on the type and content of the file. If *-L* is used, symbolic links that were specified as a parameter to cp (1) and symbolic links that were encountered while copying recursively may be followed, depending on the content of the file.

If you want to preserve the ownership, SGID/SUID bits, and the modification and access times of a file, you can use the *-p* flag. This will try to preserve these properties in the file or directory copy. Good implementations of cp (1) provide some additional protection as well - if the target file already exists, it may not be overwritten if the relevant metadata could not be preserved.

Moving files and directories

The UNIX command for moving files, mv (1) , can move or rename files or directories. What actually happens depends on the location of the files or directories. If the source and destination files or directories are on the same filesystem, mv (1) usually just creates new hard links, effectively renaming the files or directories. If both are on different filesystems, the files are actually copied, and the source files or directories are unlinked.

The syntax of mv (1) is comparable to cp (1) . The most basic syntax renames `file1` to `file2`:

```
$ mv file1 file2
```

The same syntax can be used for two directories as well, which will rename the directory given as the first parameter to the second parameter.

When the last parameter is an existing directory, the file or directory that is specified as the first parameter, is copied to that directory. In this case you can specify multiple files or directories as well. For instance:

```
$ targetdir
$ mv file1 directory1 targetdir
```

This creates the directory `targetdir`, and moves `file1` and `directory1` to this directory.

Removing files and directories

Files and directories can be removed with the rm (1) command. This command unlinks files and directories. If there are no other links to a file, its inode and disk blocks can be reclaimed for new files. Files can be removed by providing the files that should be removed as a parameter to rm (1) . If the file is not writable, rm (1) will ask for confirmation. For instance, to remove `file1` and `file2`, you can execute:

```
$ rm file1 file2
```

If you have to remove a large number of files that require a confirmation before they can be deleted, or if you want to use rm (1) to remove files from a script that will not be run on a terminal, add the `-f` parameter to override the use of prompts. Files that are not writable, are deleted with the `-f` flag if the file ownership allows this. This parameter will also suppress printing of errors to *stderr* if a file that should be removed was not found.

Directories can be removed recursively as well with the `-r` parameter. rm (1) will traverse the directory structure, unlinking and removing directories as they are encountered. The same semantics are used as when normal files are removed, as far as the `-f` flag is concerned. To give a short example, you can recursively remove all files and directories in the `notes` directory with:

```
$ rm -r notes
```

Since rm (1) command uses the unlink (2) function, data blocks are not rewritten to an uninitialized state. The information in data blocks is only overwritten when they are reallocated and used at a later time. To remove files including their data blocks securely, some systems provide a shred (1) command that overwrites data blocks with random data. But this is not effective on many modern (journaling) filesystems, because they don't write data in place.

The unlink (1) commands provides a one on one implementation of the unlink (2) function. It is of relatively little use, because it can not remove directories.

4.5. Permissions

We touched the subject of file and directory permissions in the section called “File permissions”. In this section, we will look at the chown (1) and chmod (1) commands, that are used to set the file ownership and permissions respectively. After that, we are going to look at a modern extension to permissions named Access Control Lists (ACLs).

Changing the file ownership

As we have seen earlier, every file has an owner (user) ID and a group ID stored in the inode. The chown (1) command can be used to set these fields. This can be done by the numeric IDs, or their names. For instance, to change the owner of the file `note.txt` to *john*, and its group to *staff*, the following command is used:

```
$ chown john:staff note.txt
```

You can also omit either components, to only set one of both fields. If you want to set the user name, you can also omit the colon. So, the command above can be split up in two steps:

```
$ chown john note.txt
$ chown :staff note.txt
```

If you want to change the owner of a directory, and all the files or directories it holds, you can add the `-R` to chown (1) :

```
$ chown -R john:staff notes
```

If user and group names were specified, rather than IDs, the names are converted by chown (1) . This conversion usually relies on the system-wide password database. If you are operating on a filesystem that uses another password database (e.g. if you mount a root filesystem from another system for recovery), it is often useful to change file ownership by the user or group ID. In this manner, you can keep the relevant user/group name to ID mappings in tact. So, changing the ownership of `note` to UID 1000 and GUID 1000 is done in the following (predictable) manner:

```
$ chown 1000:1000 note.txt
```

Changing the file permission bits

After reading the introduction to filesystem permissions in the section called “File permissions”, changing the permission bits that are stored in the inode is fairly easy with the chmod (1) command. chmod (1) accepts both numeric and symbolic representations of permissions. Representing the permissions of a file numerically is very handy, because it allows setting all relevant permissions tersely. For instance:

```
$ chmod 0644 note.txt
```

Make `note.txt` readable and writable for the owner of the file, and readable for the file group and others.

Symbolic permissions work with addition or subtraction of rights, and allow for relative changes of file permissions. The syntax for symbolic permissions is:

```
[ugo][-+][rwxst]
```

The first component specifies the user classes to which the permission change applies (user, group or other). Multiple characters of this component can be combined. The second component takes away rights (-), or adds rights (+). The

third component is the access specifier (read, write, execute, set UID/GID on execution, sticky). Multiple components can be specified for this component too. Let's look at some examples to clear this up:

```
ug+rw        # Give read/write rights to the file user and group
chmod go-x   # Take away execute rights from the file group and others.
chmod ugo-wx # Disallow all user classes to write to the file and to
             # execute the file.
```

These commands can be used in the following manner with chmod:

```
$ chown ug+rw note.txt
$ chown go-x script1.sh
$ chown ugo-x script2.sh
```

Permissions of files and directories can be changed recursively with the `-R`. The following command makes the directory `notes` world-readable, including its contents:

```
$ chmod -R ugo+r notes
```

Extra care should be taken with directories, because the *x* flag has a special meaning in a directory context. Users that have execute rights on directories can access a directory. User that don't have execute rights on directories can not. Because of this particular behavior, it is often easier to change the permissions of a directory structure and its files with help of the find (1) command .

There are a few extra permission bits that can be set that have a special meaning. The SUID and SGID are the most interesting bits of these extra bits. These bits change the active user ID or group ID to that of the owner or group of the file when the file is executed. The **su(1)** command is a good example of a file that usually has the SUID bit set:

```
$ ls -l /bin/su
-rwsr-xr-x  1 root root 60772 Aug 13 12:26 /bin/su
```

This means that the **su** command runs as the user *root* when it is executed. The SUID bit can be set with the *s* modifier. For instance, if the SUID bit was not set on `/bin/su` this could be done with:

```
$ chmod u+s /bin/su
```

Note

Please be aware that the SUID and SGID bits have security implications. If a program with these bits set contain a bug, it may be exploited to get privileges of the file owner or group. For this reason it is good manner to keep the number of files with the SUID and SGID bits set to an absolute minimum.

The sticky bit is also interesting when it comes to directory. It disallows users to rename of unlink files that they do not own, in directories that they do have write access to. This is usually used on world-writeable directories, like the temporary directory (`/tmp`) on many UNIX systems. The sticky tag can be set with the *t* modifier:

```
$ chmod g+t /tmp
```

File creation mask

The question that remains is what initial permissions are used when a file is created. This depends on two factors: the mode flag that was passed to the *open(2)* system call, that is used to create a file, and the active file creation mask. The file creation mask can be represented as an octal number. The effective permissions for creating the file are determined as *mode & ~mask*. Or, if represented in an octal fashion, you can substract the digits of the mask from the mode. For instance, if a file is created with permissions *0666* (readable and writable by the file user, file group, and others), and the effective file creation mask is *0022*, the effective file permission will be *0644*. Let's look at anothere example. Suppose that files are still created with *0666* permissions, and you are more paranoid, and want to take away all read and write permissions for the file group and others. This means you have to set the file creation mask to *0066*, because substracting *0066* from *0666* yields *0600*

The effective file creation mask can be queried and set with the **umask** command, that is normally a built-in shell command. The effective mask can be printed by running **umask** without any parameters:

```
$ umask
0002
```

The mask can be set by giving the octal mask number as a parameter. For instance:

```
$ umask 0066
```

We can verify that this works by creating an empty file:

```
$ touch test
$ ls -l test
-rw-------  1 daniel daniel 0 Oct 24 00:10 test2
```

Access Control Lists

Access Control lists (ACLs) are an extension to traditional UNIX file permissions, that allow for more fine-grained access control. Most systems that support filesystem ACLs implement them as they were specified in the POSIX.1e and POSIX.2c draft specifications. Notable UNIX and UNIX-like systems that implement ACLs according to this draft are FreeBSD, Solaris, and Linux.

As we have seen in the section called “File permissions” access control lists allows you to use read, write and execute triplets for additional users or groups. In contrast to the traditional file permissions, additional access control lists are note stored directly in the node, but in extended attributes that are associated with files. Two thing to be aware of when you use access control lists is that not all systems support them, and not all programs support them.

Reading access control lists

On most systems that support ACLs, **ls** uses a visual indicator to show that there are ACLs associated with a file. For example:

```
$ ls -l index.html
-rw-r-----+ 1 daniel daniel 3254 2006-10-31 17:11 index.html
```

As you can see, the permissions column shows an additional plus (+) sign. The permission bits do not quite act like you expect them to be. We will get to that in a minute.

The ACLs for a file can be queried with the **getfacl** command:

```
$ getfacl index.html
# file: index.html
# owner: daniel
# group: daniel
user::rw-
group::---
group:www-data:r--
mask::r--
other::---
```

Most lines can be interpreted very easily: the file user has read/write permissions, the file group no permissions, users of the group *www-data* have read permissions, and other users have no permissions. But why does the group entry list no permissions for the file group, while **ls** does? The secret is that if there is a *mask* entry, **ls** displays the value of the mask, rather than the file group permissions.

The *mask* entry is used to restrict all list entries with the exception of that of the file user, and that for other users. It is best to memorize the following rules for interpreting ACLs:

- The *user::* entry permissions correspond with the permissions of the file owner.
- The *group::* entry permissions correspond with the permissions of the file group, unless there is a *mask::* entry. If there is a *mask::* entry, the permissions of the group correspond to the group entry with the the mask entry as the maximum of allowed permissions (meaning that the group restrictions can be more restrictive, but not more permissive).
- The permissions of other users and groups correspond to their *user:* and *group:* entries, with the value of *mask::* as their maximum permissions.

The second and third rules can clearly be observed if there us a user or group that has more rights than the mask for the file:

```
$ getfacl links.html
# file: links.html
# owner: daniel
# group: daniel
user::rw-
group::rw-                      #effective:r--
group:www-data:rw-              #effective:r--
mask::r--
other::---
```

Although read and write permissions are specified for the file and *www-data* groups, both groups will effectively only have read permission, because this is the maximal permission that the mask allows.

Another aspect to pay attention to is the handling of ACLs on directories. Access control lists can be added to directories to govern access, but directories can also have *default ACLs* which specify the initial ACLs for files and directories created under that directory.

Suppose that the directory `reports` has the following ACL:

```
$ getfacl reports
# file: reports
# owner: daniel
# group: daniel
user::rwx
group::r-x
group:www-data:r-x
mask::r-x
other::---
default:user::rwx
default:group::r-x
default:group:www-data:r-x
default:mask::r-x
default:other::---
```

New files that are created in the `reports` directory get a ACL based on the entries that have *default:* as a prefix. For example:

```
$ touch reports/test
$ getfacl reports/test
# file: reports/test
# owner: daniel
# group: daniel
user::rw-
group::r-x                      #effective:r--
group:www-data:r-x              #effective:r--
mask::r--
other::---
```

As you can see, the default ACL was copied. The execute bit is removed from the mask, because the new file was not created with execute permissions.

Creating access control lists

The ACL for a file or directory can be changed with the **setfacl** program. Unfortunately, the usage of this program highly depends on the system that is being used. To add to that confusion, at least one important flag (`-d`) has a different meanings on different systems. One can only hope that this command will get standardized.

Due to these differences, we will only have a look at the **setfacl** implementations of Solaris, FreeBSD, and GNU/Linux. Table 4.4, “System-specific **setfacl** flags” contains an overview of all important flags.

Table 4.4. System-specific setfacl flags

Operation	FreeBSD	Linux	Solaris
Set entries, removing all old entries	Not supported	`--set`	`-s`
Modify entries	-m	`-m`	`-m`
Modify default ACL entries	`-d`	`-d`	Use *default:* prefix
Delete entry	`-x`	`-x`	`-d`
Remove all ACL entries (except for the three required entries.	`-b`	`-b`	Not supported
Recalculate mask	Always recalculated, unless `-n` is used, or an mask entry expicitly specified.	Always recalculated, unless `-n` is used, or an mask entry expicitly specified.	`-r`
Use ACL specification from a file	`-M` (modify) or `-X` (delete)	`-M` (modify), `-X` (delete), or `--restore`	-f
Recursive modification of ACLs	Not supported	`-R`	Not supported

As we have seen in the previous section, entries can be specified for users and groups, by using the following syntax: *user/group:name:permissions*. Permissions can be specified as a triplet by using the letters *r* (read), *w* (write), or *x* (execute). A dash (-) should be used for permissions that you do not want to give to the user or group, since Solaris requires this. If you want to disallow access completely, you can use the --- triplet.

The specification for other users, and the mask differs per system. Solaris requires one colon between *other*/*mask* and the permissions, for instance: *other:r-x*. FreeBSD requires the use of two colons, for example: *other::r-x*. On GNU/Linux, both syntaxes are allowed.

Modifying ACL entries

The simplest operation is to modify an ACL entry. This will create a new entry if the entry does not exist yet. Entries can be modified on Solaris, FreeBSD and GNU/Linux with the `-m`. For instance, suppose that we want to give the group *friend* read and write access to the file `report.txt`. This can be done one all three systems with:

```
$ setfacl -m group:friends:rw- report.txt
```

If we look at the resulting ACL, the difference in the default behavior of FreeBSD and GNU/Linux becomes apparent. Both FreeBSD and GNU/Linux recalculate the mask entry, setting it to the union of all group entries, and additional user entries:

```
$ getfacl report.txt
# file: report.txt
# owner: daniel
# group: daniel
user::rw-
group::r--
group:friends:rw-
mask::rw-
other::r--
```

While Solaris just creates the mask entry (based on the file group permissions), but does not touch it otherwise:

```
$ getfacl report.txt
# file: report.txt
# owner: daniel
# group: other
user::rw-
group::r--              #effective:r--
group:friends:rw-               #effective:r--
mask::r--
other::r--
```

The default mask is only recalculated if the *-r* flag is used:

```
$ setfacl -r -m group:friends:rw- report.txt
$ getfacl report.txt
# file: report.txt
# owner: daniel
# group: other
user::rw-
group::r--              #effective:r--
group:friends:rw-               #effective:rw-
mask::rw-
other::r--
```

On all three systems, you can combine multiple ACL entries by separating them with a comma character. For instance:

```
$ setfacl -m group:friends:rw-,group:foes:--- report.txt
```

Removing ACL entries

An entry can be removed with the *-x* flag on FreeBSD and GNU/Linux:

```
$ setfacl -x group:friends: report.txt
```

On Solaris, the *-d* flag is used instead:

On Solaris and GNU/Linux the leading colon (:) can be omitted. FreeBSD's **setfacl** requires the use of the colon.

```
$ setfacl -d group:friends: report.txt
```

Making a new ACL

Both Solaris and GNU/Linux provide a flag to set new permissions for a file, clearing all existing entries, except for the three required entries. These flags are `-s` and `--set` respectively. On both systems, it is required that the file user, group and other entries are also specified. Solaris also requires that a mask entry is specified. For instance, on GNU/Linux, you could use:

```
$ setfacl --set user::rw-,group::r--,other:---,group:friends:rwx report.txt
```

FreeBSD does not provide such option, but it can be emulated by combining the `-b` flag, which clears all entries except for the three required entries, and the `-m` flag. This also works on GNU/Linux. For instance:

```
$ setfacl -b -m group:friends:rw- report.txt
```

Setting a default ACL

As we have seen in the section called “Access Control Lists”, directories can have default ACL entries that specify what permissions should be used for files and directories that are created below that directory. Both FreeBSD and GNU/Linux use the `-d` flag to operate on default entries:

```
$ setfacl -d -m group:friends:rwx reports
$ getfacl reports
# file: reports
# owner: daniel
# group: daniel
user::rwx
group::r-x
other::r-x
default:user::rwx
default:group::r-x
default:group:friends:rwx
default:mask::rwx
default:other::r-x
```

Default entries are set on Solaris by adding the *default:* prefix. Default entries for the *user*, *group*, and *other* are not automatically generated, so you will have to specify them explicitly. For instance:

```
$ setfacl -m default:user::rwx,default:group::rwx,default:other:rwx,default:mask:rwx,default:
```

Using an ACL from a reference file

All three systems provide options to use a file as the input for ACL entries. An input file follows the same syntax as specifying entries as a parameter to **setfacl**, but the entries are separated by newlines, rather than by commas. This is very useful, because you can use the ACL for an existing file as a reference:

```
$ getfacl report.txt > ref
```

Both FreeBSD and GNU/Linux provide the `-M` to modify the ACL for a file by reading the entries from a file. So, if we have a file named `report2.txt`, we could modify the ACL for this file with the entries from `ref` with:

```
$ setfacl -M ref report2.txt
```

If you would like to start with a clean ACL, and add the entries from `ref`, you can add the `-b` flag that we encountered earlier:

```
$ setfacl -b -M ref report2.txt
```

Of course, it is not necessary to use this interim file. We can directly pipe the output from **getfacl** to **setfacl**, by using the symbolic name for the standard input (-), rather than the name of a file:

```
$ getfacl report.txt | setfacl -b -M - report2.txt
```

On GNU/Linux and FreeBSD also provides the `-X` flag to remove permissions from a file. This follows the same syntax as the `-x` flag, with commas replaced by newlines.

setfacl on Solaris provides the `-f` flag to read from a file. This flag is comparable with the `-s` flag, it requires that the user, group, mask, and other entries are included.

The `-f` parameter can be used to read both from a file or from the standard input. The following example uses the file `ref` and the output of **getfacl** respectively:

```
$ setfacl -f ref report2.txt
$ getfacl report.txt | setfacl -f - test
```

4.6. Finding files

find

The **find** command is without doubt the most comprehensive utility to find files on UNIX systems. Besides that it works in a simple and predictable way: **find** will traverse the directory tree or trees that are specified as a parameter to **find**. Besides that a user can specify an expression that will be evaluated for each file and directory. The name of a file or directory will be printed if the expression evaluates to *true*. The first argument that starts with a dash (-), exclamation mark (*!*, or an opening parenthesis (*(*, signifies the start of the expression. The expression can consist of various operands. To wrap it up, the syntax of **find** is: *find paths expression*.

The simplest use of **find** is to use no expression. Since this matches every directory and subdirectory entry, all files and directories will be printed. For instance:

```
$ find .
.
./economic
./economic/report.txt
./economic/report2.txt
./technical
./technical/report2.txt
./technical/report.txt
```

You can also specify multiple directories:

```
$ find economic technical
economic
economic/report.txt
economic/report2.txt
technical
technical/report2.txt
technical/report.txt
```

Operands that limit by object name or type

One common scenario for finding files or directories is to look them up by name. The *-name* operand can be used to match objects that have a certain name, or match a particular wildcard. For instance, using the operand *-name 'report.txt'* will only be true for files or directories with the name `report.txt`. For example:

```
$ find economic technical -name 'report.txt'
economic/report.txt
technical/report.txt
```

The same thing holds for wildcards:

```
$ find economic technical -name '*2.txt'
economic/report2.txt
technical/report2.txt
```

Note

When using **find** you will want to pass the wildcard to **find**, rather than letting the shell expand it. So, make sure that patterns are either quoted, or that wildcards are escaped.

It is also possible to evaluate the type of the object with the *-type c* operand, where *c* specifies the type to be matched. Table 4.5, “Parameters for the '-type' operand” lists the various object types that can be used.

Table 4.5. Parameters for the '-type' operand

Parameter	Meaning
b	Block device file
c	Character device file
d	Directory
f	Regular file
l	Symbolic link
p	FIFO
s	Socket

So, for instance, if you would like to match directories, you could use the *d* parameter to *-type* operand:

```
$ find . -type d
.
./economic
./technical
```

We will look at forming a complex expression at the end of this section about **find**, but at this moment it is handy to know that you can make a boolean 'and' expression by specifying multiple operands. For instance *operand1 operand2* is true if both *operand1* and *operand2* are true for the object that is being evaluated. So, you could combine the *-name* and *-type* operands to find all directories that start with *eco*:

```
$ find . -name 'eco*' -type d
./economic
```

Operands that limit by object ownership or permissions

Besides matching objects by their name or type, you can also match them by their active permissions or the object ownership. This is often useful to find files that have incorrect permissions or ownership.

The owner (user) or group of an object can be matched with respectively the *-user username* and *-group groupname* variants. The name of a user or group will be interpreted as a user ID or group ID of the name is decimal, and could not be found on the system with getpwnam(3) or getgrnam(3). So, if you would like to match all objects of which *joe* is the owner, you can use *-user joe* as an operand:

```
$ find . -user joe
./secret/report.txt
```

Or to find all objects with the group *friend* as the file group:

```
$ find . -group friends
./secret/report.txt
```

The operand for checking file permissions *-perm* is less trivial. Like the **chmod** command this operator can work with octal and symbolic permission notations. We will start with looking at the octal notation. If an octal number is specified

as a parameter to the *-perm* operand, it will match all objects that have exactly that permissions. For instance, *-perm 0600* will match all objects that are only readable and writable by the user, and have no additional flags set:

```
$ find . -perm 0600
./secret/report.txt
```

If a dash is added as a prefix to a number, it will match every object that has at least the bits set that are specified in the octal number. A useful example is to find all files which have at least writable bits set for *other* users with *-perm -0002*. This can help you to find device nodes or other objects with insecure permissions.

```
$ find /dev -perm -0002
/dev/null
/dev/zero
/dev/ctty
/dev/random
/dev/fd/0
/dev/fd/1
/dev/fd/2
/dev/psm0
/dev/bpsm0
/dev/ptyp0
```

Note

Some device nodes have to be world-writable for a UNIX system to function correctly. For instance, the `/dev/null` device is always writable.

The symbolic notation of *-perm* parameters uses the same notation as the **chmod** command. Symbolic permissions are built with a file mode where all bits are cleared, so it is never necessary to use a dash to take away rights. This also prevents ambiguity that could arise with the dash prefix. Like the octal syntax, prefixing the permission with a dash will match objects that have at least the specified permission bits set. The use of symbolic names is quite predictable - the following two commands repeat the previous examples with symbolic permissions:

```
$ find . -perm u+rw
./secret/report.txt
```

```
$ find /dev -perm -o+w
/dev/null
/dev/zero
/dev/ctty
/dev/random
/dev/fd/0
/dev/fd/1
/dev/fd/2
/dev/psm0
/dev/bpsm0
/dev/ptyp0
```

Operands that limit by object creation time

There are three operands that operate on time intervals. The syntax of the operand is *operand n*, where *n* is the time in days. All three operators calculate a time delta in seconds that is divided by the the number of seconds in a day (86400), discarding the remainder. So, if the delta is one day, *operand 1* will match for the object. FreeBSD deviates from the Single UNIX Specification in this respect, because it rounds times to the next full day, this can be an unwelcome trap in scripts. The three operands are:

- *-atime n* - this operand evaluates to true if the initialization time of **find** minus the last access time of the object equals to *n*.
- *-ctime n* - this operand evaluates to true if the initialization time of **find** minus the time of the latest change in the file status information equals to *n*.
- *-mtime n* - this operand evaluates to true if the initialization time of **find** minus the latest file change time equals to *n*.

So, these operands match if the latest access, change, modification respectively was *n* days ago. To give an example, the following command shows all objects in `/etc` that have been modified one day ago:

```
$ find /etc -mtime 1
/etc
/etc/group
/etc/master.passwd
/etc/spwd.db
/etc/passwd
/etc/pwd.db
```

The plus or minus sign can be used as modifiers for the meaning of *n*. *+n* means more than *n* days, *-n* means less than *n* days. So, to find all files in `/etc` that were modified less than two days ago, you could execute:

```
$ find /etc -mtime -2
/etc
/etc/network/run
/etc/network/run/ifstate
/etc/resolv.conf
/etc/default
/etc/default/locale
[...]
```

Another useful time-based operand is the *-newer reffile* operand. This matches all files that were modified later that the file with filename `reffile`. The followin example shows how you could use this to list all files that have later modification times than `economic/report2.txt`:

```
$ find . -newer economic/report2.txt
.
./technical
./technical/report2.txt
./technical/report.txt
./secret
./secret/report.txt
```

Operands that affect tree traversal

Some operands affect the manner in which the **find** command traverses the tree. The first of these operands is the *-xdev* operand. *-xdev* prevents that **find** decends into directories that have a different device ID, effectively avoiding traversal of other filesystems. The directory to which the filesystem is mounted, is printed, because this operand always returns *true*. A nice example is a system where `/usr` is mounted on a different filesystem than `/`. For instance, if we search for directories with the name *bin*, this may yield the following result:

```
$ find / -name 'bin' -type d
/usr/bin
/bin
```

But if we add *-xdev* `/usr/bin` is not found, because it is on a different filesystem (and device):

```
$ find / -name 'bin' -type d -xdev
/bin
```

The *-depth* operand modifies the order in which directories are evaluated. With *-depth* the contents of a directory are evaluated first, and then the directory itself. This can be witnessed in the following example:

```
$ find . -depth
./economic/report.txt
./economic/report2.txt
./economic
./technical/report2.txt
./technical/report.txt
./technical
.
```

As you can see in the output, files in the *./economic* directory is evaluated before `.`, and `./economic/report.txt` before `./economic`. *-depth* always evaluates to *true*.

Finally, the *-prune* operand causes find not to decend into a directory that is being evaluated. *-prune* is discarded if the *-depth* operand is also used. *-depth* always evaluates to *true*.

Operands that execute external utilities

find becomes a very powerful tool when it is combined with external utilities. This can be done with the *-exec* operand. There are two syntaxes for the *-exec* operand. The first syntax is *-exec utility arguments ;*. The command *utility* will be executed with the arguments that were specified for each object that is being evaluated. If any of the arguments is *{}*, these braces will be replaced by the file being evaluated. This is very handy, especially when we consider that, if we use no additional expression syntax, operands will be evaluated from left to right. Let's look at an example:

```
$ find . -perm 0666 -exec chmod 0644 {} \;
```

The first operand returns true for files that have their permissions set to *0666*. The second operand executes *chmod 0644 filename* for each file that is being evaluated. If you were wondering why this command is not executed for every file, that is a good question. Like many other interpreters of expressions, **find** uses "short-circuiting". Because no other operator was specified, the logical *and* operator is automatically is assumed between both operands. If the first operand evaluates to *false*, it makes no sense to evaluate any further operands, because the complete expression will always evaluate to false. So, the *-exec* operand will only be evaluated if the first operand is true. Another particularity is that the semi-colon that closes the *-exec* is escaped, to prevent that the shell parses it.

A nice thing about the *-exec* operator is that it evaluates to *true* if the command terminated sucessfully. So, you could also use the *-exec* command to add additional conditions that are not represented by **find** operands. For instance, the following command prints all objects ending with *.txt* that contain the string *gross income*:

```
$ find . -name '*.txt' -exec grep -q 'gross income' {} \; -print
./economic/report2.txt
```

The **grep** command will be covered lateron. But for the moment, it is enough to know that it can be used to match text patterns. The *-print* operand prints the current object path. It is always used implicitly, except when the *-exec* or *-ok* operands are used.

The second syntax of the *-exec* operand is *-exec utility arguments {} +*. This gathers a set of all matched object for which the expression is true, and provides this set of files as an argument to the utility that was specified. The first example of the *-exec* operand can also be written as:

```
$ find . -perm 0666 -exec chmod 0644 {} +
```

This will execute the **chmod** command only once, with all files for which the expression is true as its arguments. This operand always returns *true*.

If a command executed by find returns a non-zero value (meaning that the execution of the command was not succesful), **find** should also return a non-zero value.

Operators for building complex expressions

find provides some operators that can be combined to make more complex expressions:

Operators

(expr)	Evaluates to *true* if *expr* evaluates to *true*.
expr1 [-a] expr2	Evaluates to *true* if both *expr1* and *expr2* are true. If *-a* is omitted, this operator is implicitly assumed.
	find will use short-circuiting when this operator is evaluated: *expr2* will not be evaluated when *expr1* evaluates to *false*
expr1 -o expr2	Evaluates to *true* if either or both *expr1* and *expr2* are true.
	find will use short-circuiting when this operator is evaluated: *expr2* will not be evaluated when *expr1* evaluates to *true*
! expr	Negates *expr*. So, if *expr* evaluates to true, this expression will evaluate to *false* and vise versa.

Since both the parentheses and exclamation mark characters are interpreted by most shells, they should usually be escaped.

The following example shows some operators in action. This command executes **chmod** for all files that either have their permissions set to *0666* or *0664*.

```
$ find . \( -perm 0666 -o -perm 0664 \) -exec chmod 0644 {} \;
```

which

The **which** command is not part of the Single UNIX Specification version 3, but it is provided by most sysmtems. **which** locates a command that is in the user's path (as set by the PATH environment variable), printing its full path. Providing the name of a command as its parameter will show the full path:

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

You can also query the paths of multiple commands:

```
$ which ls cat
/bin/ls
/bin/cat
```

The handling of commands that could not be found is implementation-dependent. **which** on GNU/Linux and FreeBSD returns a non-zero return value. Solaris' **which** prints an error message, but always returns zero.

whereis

Both Linux and FreeBSD provide the **whereis**. This command searches binaries, manual pages and sources of a command in some predefined places. For instance, the following command shows the path of the **ls** and the ls(1) manual page:

```
$ whereis ls
ls: /bin/ls /usr/share/man/man1/ls.1.gz
```

locate

Some systems, like FreeBSD and GNU/Linux provide a **locate** that searches through a file database that can be generated periodically with the **updatedb** command. Since it uses a prebuilt database of the filesystem, it is a lot faster than **command**, especially when directory entry information has not been cached yet. Though, the **locate**/**updatedb** combo has some downsides:

- New files are not part of the database until the next **updatedb** invocation.
- **locate** has no conception of permissions, so users may locate files that are normally hidden to them.
- A newer implementation, named *slocate* deals with permissions, but requires elevated privileges.

With filesystems becoming faster, and by applying common sense when formulating **find** queries, **locate** does not really seem worth the hassle. Of course, your mileage may vary. That said, the basic usage of **locate** is *locate filename*. For example:

```
$ locate locate
/usr/bin/locate
/usr/lib/locate
/usr/lib/locate/bigram
/usr/lib/locate/code
/usr/lib/locate/frcode
[...]
```

4.7. Compression and archiving

It is often practical to condense a set of files and directories in one file - it is easier to send, distribute or store somewhere. Such files are called *archives*. For historical reasons, most UNIX-like systems provide at least three archiving tools: **tar**, **cpio**, and **pax**. Of these, **tar** and **cpio** have their own archive formats (and variations thereof). Fortunately, the newer **pax** program can deal with both cpio and tar archives, and some historical variations thereof.

In contrast to some other systems, the archiving tools of UNIX systems follow the "Write programs that do one thing and do it well."-philosophy, and let external programs handle compression of archives. The traditional UNIX compression program, that is also described in the Single UNIX Specification version 3 is **compress**. Unfortunately, this program uses the LZW compression algorithm, which was patented from 1983 to 2003[3]. This prompted the GNU project to implement a file compressor that was patent-free. This program is named **gzip**, and uses the LZ77 compression algorithm and Huffman coding. **gzip** has become the dominant compression program. Many GNU/Linux distributions do not even install **compress** by default, even now that the patent has expired. More recently, the **bzip2** program was introduced, which uses a block-sorting compressor. **bzip2** normally gives better compression rathes than **gzip** at the cost of time.

This section will describe the **compress**, **gzip**, and **bzip2** file compression programs, and the **tar** and **pax** archivers.

File compression

compress

compress is the compression utility specified by the Single UNIX Specification version 3. The author of this book would recommend you to use the the widely used **gzip** or **bzip2** tools instead.

If **compress** is used without any parameters, it will read and compress data from *stdin*, and send the compressed output to *stdout*. The following command will compress the sentence "Hello world.", and stores the result in `hello.Z`:

```
$ echo "Hello world." | compress > hello.Z
```

The **uncompress** command does the exact opposite of compress, and decompresses its input data. For example, you can decompress the file that was just created:

```
$ uncompress < hello.Z
```

[3] http://en.wikipedia.org/wiki/Lzw

```
Hello world.
```

To compress a file, rather than data read from *stdin*, add the name of the file as an argument. **compress** stores the compressed file in a file named after the original file, with the suffix *.Z* appended. The original file is removed afterwards. The following example compresses `alice_wonderland.txt`, and stores the result as `alice_wonderland.txt.Z`:

```
$ ls -l
total 164
-rw-r--r-- 1 daniel daniel 163218 2005-03-05 20:21 alice_wonderland.txt
$ compress alice_wonderland.txt
$ ls -l
total 72
-rw-r--r-- 1 daniel daniel 68263 2005-03-05 20:21 alice_wonderland.txt.Z
```

As you can see, the original permissions, ownership, and modification and access time are retained.

In a similar fashion, the file can be uncompressed with **uncompress**. **uncompress** creates an uncomressed file with the *.Z* suffix removed, and removes the compressed file:

```
$ uncompress alice_wonderland.txt.Z
$ ls -l
total 164
-rw-r--r-- 1 daniel daniel 163218 2005-03-05 20:21 alice_wonderland.txt
```

And again, the original file properties (permissions, ownership, access/modification times) are retained.

If you would like to have information about the compression rate, you can add the `-v` to **compress** or **uncompress**:

```
$ compress -v alice_wonderland.txt
alice_wonderland.txt:  -- replaced with alice_wonderland.txt.Z Compression: 58.17%
$ uncompress -v alice_wonderland.txt.Z
alice_wonderland.txt.Z:  58.2% -- replaced with alice_wonderland.txt
```

Both **compress** and **uncompress** can write output to *stdout*, rather than a file. When the input data is read from *stdin*, this is done automatically. Otherwise, the `-c` option has to be added to do this. For example, we could read the compressed version of Alice in Wonderland with:

```
$ uncompress -c alice_wonderland.txt.Z | more
```

If the target file exists during compression or uncompression, and the process is running in the foreground, confirmation from the user will be asked to overwrite the file. You can avoid this, and force overwriting the target file by adding the `-f` option to **compress** or **uncompress**. When **compress** or **uncompress** is not running in the foreground, and `-f` is not used, it should abort exit with a non-zero return value if the target file already exists.

Chapter 5. Text processing

Text manipulation is one of the things that UNIX excels at, because it forms the heart of the UNIX philosophy, as described in Section 1.3, “UNIX philosophy”. Most UNIX commands are simple programs that read data from the standard input, performs some operation on the data, and sends the result to the program's standard output. These programs basically act as an filters, that can be connected as a pipeline. This allows the user to put the UNIX tools to uses that the writers never envisioned. In later chapters we will see how you can build simple filters yourself.

This chapter describes some simple, but important, UNIX commands that can be used to manipulate text. After that, we will dive into regular expressions, a sublanguage that can be used to match text patterns.

5.1. Simple text manipulation

Repeating what is said

The most simple text filter is the **cat**, it does nothing else than sending the data from stdin to stdout:

```
$ echo "hello world" | cat
hello world
```

Another useful feature is that you can let it send the contents of a file to the standard output:

```
$ cat file.txt
Hello, this is the content of file.txt
```

cat really lives up to its name when multiple files are added as arguments. This will concatenate the files, in the sense that it will send the contents of all files to the standard output, in the same order as they were specified as an argument. The following screen snippet demonstrates this:

```
$ cat file.txt file1.txt file2.txt
Hello, this is the content of file.txt
Hello, this is the content of file1.txt
Hello, this is the content of file2.txt
```

Text statistics

The **wc** command provides statistics about a text file or text stream. Without any parameters, it will print the number of lines, the number of words, and the number of bytes respectively. A word is delimited by one white space character, or a sequence of whitespace characters.

The following example shows the number of lines, words, and bytes in the canonical “Hello world!” example:

```
$ echo "Hello world!" | wc
      1       2      13
```

If you would like to print just one of these components, you can use one of the `-l` (lines), `-w` (words), or `-c` (bytes) parameters. For instance, adding just the `-l` parameter will show the number of lines in a file:

```
$ wc -l /usr/share/dict/words
  235882 /usr/share/dict/words
```

Or, you can print additional fields by adding a parameter:

```
$ wc -lc /usr/share/dict/words
 235882 2493082 /usr/share/dict/words
```

Please note that, no matter the order in which the options were specified, the output order will always be the same (lines, words, bytes).

Since `-c` prints the number bytes, this parameter may not represent the number of characters that a text holds, because the character set in use maybe be wider than one byte. To this end, the `-m` parameter has been added which prints the number of characters in a text, independent of the character set. `-c` and `-m` are substitutes, and can never be used at the same time.

The statistics that **wc** provides are more useful than they may seem on the surface. For example, the `-l` parameter is often used as a counter for the output of a command. This is convenient, because many commands seperate logical units by a newline. Suppose that you would like to count the number of files in your home directory having a filename ending with `.txt`. You could do this by combining **find** to find the relevant files and **wc** to count the number of occurences:

```
$ find ~ -name '*.txt' -type f | wc -l
```

Manipulating characters

The **tr** command can be used to do common character operations, like swapping characters, deleting characters, and squeezing character sequences. Depending on the operation, one or two sets of characters should be specified. Besides normal characters, there are some special character sequences that can be used:

\character	This notation is used to specify characters that need escaping, most notably \n (newline), \t (horizontal tab), and \\ (backslash).
character1-character2	Implicitly insert all characters from *character1* to *character2*. This notation should be used with care, because it does not always give the expected result. For instance, the sequence *a-d* may yield *abcd* for the POSIX locale (language setting), but this may not be true for other locales.
[:class:]	Match a predefined class of characters. All possible classes are shown in Table 5.1, “tr character classes”.
[character*]	Repeat *character* until the second set is as long as the first set of characters. This notation can only be used in the second set.
[character*n]	Repeat *character n* times.

Table 5.1. tr character classes

Class	Meaning
[:alnum:]	All letters and numbers.
[:alpha:]	Letters.
[:blank:]	Horizontal whitespace (e.g. spaces and tabs).
[:cntrl:]	Control characters.
[:digit:]	All digits (0-9).
[:graph:]	All printable characters, except whitespace.
[:lower:]	Lowercase letters.
[:print:]	All printable characters, including horizontal whitespace, but excluding vertical whitespace.
[:punct:]	Punctuation characters.
[:space:]	All whitespace.
[:upper:]	Uppercase letters.
[:xdigit:]	Hexadecimal digits (0-9, a-f).

Swapping characters

The default operation of **tr** is to swap (translate) characters. This means that the *n*-th character in the first set is replaced with the *n*-th character in the second set. For example, you can replace all *e*'s with *i*'s and *o*'s with *a*'s with one **tr** operation:

```
$ echo 'Hello world!' | tr 'eo' 'ia'
Hilla warld!
```

The behavior of **tr** is different per system when the second set is not as large as the first set. Solaris will not touch characters that were specified in the first set that have no matching character in the second set, while GNU/Linux and FreeBSD implicitly repeat the last character. So, if you want to use **tr** in a system-independent manner, explicitly define what character should be repeated. For instance

```
$ echo 'Hello world!' | tr 'eaiou' '[@*]'
H@ll@ w@rld!
```

Produces the same output on Solaris, FreeBSD, and GNU/Linux. Another particularity is the use of the repetition syntax in the middle of the set. Suppose that set 1 is *abcdef*, and set 2 *@[-*]!*. GNU/Linux will replace *a* with *@*, *b*, *c*, *d*, and *e* with -, and *f* with *!*. FreeBSD and Solaris follow the standard thoroughly, and replace *a* with *@*, and the rest of the set characters with -. So, a more correct notation would be the more explicit *@[-*4]!*, which gives the same results on all three systems:

```
$ echo 'abcdef' | tr 'abcdef' '@[-*4]!'
@----!
```

Squeezing character sequences

When the `-s` parameter is used, **tr** will squeeze all characters that are in the second set. This means that a sequence of the same characters will be reduced to one character. Let's squeeze the character "*e*":

```
$ echo "Let's squeeze this." | tr -s 'e'
Let's squeze this.
```

We can combine this with translation to show a useful example of **tr** in action. Suppose that we would like to mark al vowels with the *at* sign (@), with consecutive vowels represented by one *at* sign. This can easily be done by piping two **tr** commands:

```
$ echo "eenie meenie minie moe" | tr 'aeiou' '[@*]' | tr -s '@'
@n@ m@n@ m@n@ m@
```

Deleting characters

Finally, **tr** can be used to delete characters. If the `-d` parameter is used, all characters from the first set are removed:

```
$ echo 'Hello world!' | tr -d 'lr'
Heo wod!
```

Cutting and pasting text columns

The **cut** command is provided by UNIX systems to “cut” one or more columns from a file or stream, printing it to the standard output. It is often useful to selectively pick some information from a text. **cut** provides three approaches to cutting information from files:

1. By byte.
2. By character, which is not the same as cutting by byte on systems that use a character set that is wider than eight bits.
3. By field, that is delimited by a character.

In all three approaches, you can specify the element to choose by its number starting at *1*. You can specify a range by using a dash (-). So, *M*-*N* means the Mth to the Nth element. Leaving M out (-*N*) selects all elements from the first element to the Nth element. Leaving N out (*M*-) selects the Mth element to the last element. Multiple elements or ranges can be combined by separating them by commas (,). So, for instance, *1*,*3*- selects the first element and the third to the last element.

Data can be cut by field with the `-f fields` parameter. By default, the horizontal tab is used a separator. Let's have a look at **cut** in action with a tiny Dutch to English dictionary:

```
$ cat dictionary
appel   apple
```

```
banaan  banana
peer    pear
```

We can get all English words by selecting the first field:

```
$ cut -f 2 dictionary
apple
banana
pear
```

That was quite easy. Now let's do the same thing with a file that has a colon as the field separator. We can easily try this by converting the dictionary with the **tr** command that we have seen earlier, replacing all tabs with colons:

```
$ tr '\t' ':' < dictionary > dictionary-new
$ cat dictionary-new
appel:apple
banaan:banana
peer:pear
```

If we use the same command as in the previous example, we do not get the correct output:

```
$ cut -f 2 dictionary-new
appel:apple
banaan:banana
peer:pear
```

What happens here is that the delimiter could not be found. If a line does not contain the delimiter that is being used, the default behavior of **cut** is to print the complete line. You can prevent this with the `-s` parameter.

To use a different delimiter than the horizontal tab, add the `-d delimter_char` parameter to set the delimiting character. So, in this case of our `dictionary-new` file, we will ask **cut** to use the colon as a delimiter:

```
$ cut -d ':' -f 2 dictionary-new
apple
banana
pear
```

If a field that was specified does not exist in a line, that particular field is not printed.

The `-b bytes` and `-c characters` respectively select bytes and characters from the text. On older systems a character used to be a byte wide. But newer systems can provide character sets that are wider than one byte. So, if you want to be sure to grab complete characters, use the `-c` parameter. An entertaining example of seeing the `-c` parameter in action is to find the ten most common sets of the first three characters of a word. Most UNIX systems provide a list of words that are separated by a new line. We can use **cut** to get the first three characters of the words in the word list, add `uniq` to count identical three character sequences, and use **sort** to sort them reverse-numerically (**sort** is described in the section called “Sorting text”). Finally, we will use **head** to get the ten most frequent sequences:

```
$ cut -c 1-4 /usr/share/dict/words | uniq -c | sort -nr | head
    254 inte
    206 comp
    169 cons
    161 cont
    150 over
    125 tran
    111 comm
    100 disc
     99 conf
     96 reco
```

Having concluded with that nice piece of UNIX commands in action, we will move on to the **paste** command, which combines files in columns in a single text stream.

Usage of **paste** is very simple, it will combine all files given as an argument, separated by a tab. With the list of English and Dutch words, we can generate a tiny dictionary:

```
$ paste dictionary-en dictionary-nl
apple   appel
banana  banaan
pear    peer
```

You can also combine more than two files:

```
$ paste dictionary-en dictionary-nl dictionary-de
apple   appel   Apfel
banana  banaan  Banane
pear    peer    Birne
```

If one of the files is longer, the column order is maintained, and empty entries are used to fill up the entries of the shorter files.

You can use another delimiter by adding the *-d delimiter* parameter. For example, we can make a colon-separated dictionary:

```
$ paste -d ':' dictionary-en dictionary-nl
apple:appel
banana:banaan
pear:peer
```

Normally, **paste** combines files as different columns. You can also let **paste** use the lines of each file as columns, and put the columns of each file on a separate line. This is done with the *-s* parameter:

```
$ paste -s dictionary-en dictionary-nl dictionary-de
apple   banana  pear
```

```
appel   banaan  peer
Apfel   Banane  Birne
```

Sorting text

UNIX offers the **sort** command to sort text. **sort** can also check whether a file is in sorted order, and merge two sorted files. **sort** can sort in dictionary and numerical orders. The default sort order is the dictionary order. This means that text lines are compared character by character, sorted as specified in the current collating sequence (which is specified through the LC_COLLATE environment variable). This has a catch when you are sorting numbers, for instance, if you have the numbers 1 to 10 on different lines, the sequence will be 1, 10, 2, 3, etc. This is caused by the per-character interpretation of the dictionary sort. If you want to sort lines by number, use the numerical sort.

If no additional parameters are specified, **sort** sorts the input lines in dictionary order. For instance:

```
$ cat << EOF | sort
orange
apple
banana
EOF
apple
banana
orange
```

As you can see, the input is correctly ordered. Sometimes there are two identical lines. You can merge identical lines by adding the *-u* parameter. The two samples listed below illustrate this.

```
$ cat << EOF | sort
orange
apple
banana
banana
EOF
apple
banana
banana
orange
$ cat << EOF | sort -u
orange
apple
banana
banana
EOF
apple
banana
orange
```

There are some additional parameters that can be helpful to modify the results a bit:

- The *-f* parameter makes the sort case-insensitive.

- If `-d` is added, only blanks and alphanumeric characters are used to determine the order.
- The `-i` parameter makes **sort** ignore non-printable characters.

You can sort files numerically by adding the `-n` parameter. This parameter stops reading the input line when a non-numeric character was found. The leading minus sign, decimal point, thousands separator, radix character (that separates an exponential from a normal number), and blanks can be used as a part of a number. These characters are interpreted where applicable.

The following example shows numerical sort in action, by piping the output of **du** to **sort**. This works because **du** specifies the size of each file as the first field.

```
$ du -a /bin | sort -n
0       /bin/kernelversion
0       /bin/ksh
0       /bin/lsmod.modutils
0       /bin/lspci
0       /bin/mt
0       /bin/netcat
[...]
```

In this case, the output is probably not useful if you want to read the output in a paginator, because the smallest files are listed first. This is where the `-r` parameter becomes handy. This reverses the sort order.

```
$ du -a /bin | sort -nr
4692    /bin
1036    /bin/ksh93
668     /bin/bash
416     /bin/busybox
236     /bin/tar
156     /bin/ip
[...]
```

The `-r` parameter also works with dictionary sorts.

Quite often, files use a layout with multiple columns, and you may want to sort a file by a different column than the first column. For instance, consider the following score file named `score.txt`:

```
John:US:4
Herman:NL:3
Klaus:DE:5
Heinz:DE:3
```

Suppose that we would like to sort the entries in this file by the two-letter country name. **sort** allows us to sort a file by a column with the `-k col1[,col2]` parameter. Where *col1* up to *col2* are used as fields for sorting the input. If *col2* is not specified, all fields up till the end of the line are used. So, if you want to use just one column, use `-k col1,col1`. You can also specify the the starting character within a column by adding a period (.) and a character index. For instance, `-k 2.3,4.2` means that the second column starting from the third character, the third column, and the fourth column up to (and including) the second character.

There is yet another particularity when it comes to sorting by columns: by default, **sort** uses a blank as the column separator. If you use a different separator character, you will have to use the `-t char` parameter, that is used to specify the field separator.

With the `-t` and `-k` parameters combined, we can sort the scores file by country code:

```
$ sort -t ':' -k 2,2 scores.txt
Heinz:DE:3
Klaus:DE:5
Herman:NL:3
John:US:4
```

So, how can we sort the file by the score? Obviously, we have to ask sort to use the third column. But sort uses a dictionary sort by default[1]. You could use the `-n`, but **sort** also allows a more sophisticated approach. You can append the one or more of the *n*, *r>*, *f*, *d*, *i*, or *b* to the column specifier. These letters represent the **sort** parameters with the same name. If you add just the starting column, append it to that column, otherwise, add it to the ending column.

The following command sorts the file by score:

```
$ sort -t ':' -k 3n /home/daniel/scores.txt
Heinz:DE:3
Herman:NL:3
John:US:4
Klaus:DE:5
```

It is good to follow this approach, rather than using the parameter variants, because **sort** allows you to use more than one `-k` parameter. And, adding these flags to the column specification, will allow you to sort by different columns in different ways. For example using **sort** with the `-k 3,3n -k 2,2` parameters will sort all lines numerically by the third column. If some lines have identical numbers in the third column, these lines can be sorted further with a dictionary sort of the second column.

If you want to check whether a file is already sorted, you can use the `-c` parameter. If the file was in a sorted order, sort will return the value *0*, otherwise *1*. We can check this by echoing the value of the *?* variable, which holds the return value of the last executed command.

```
$ sort -c scores.txt ; echo $?
1
$ sort scores.txt | sort -c ; echo $?
0
```

The second command shows that this actually works, by piping the output of the sort of `scores.txt` to **sort**.

Finally, you can merge two sorted files with the `-m` parameter, keeping the correct sort order. This is faster than concatenating both files, and resorting them.

[1]Of course, that will not really matter in this case, because we don't use numbers higher than 9, and virtually all character sets have numbers in a numerical order).

```
# sort -m scores-sorted.txt scores-sorted2.txt
```

Differences between files

Since text streams, and text files are very important in UNIX, it is often useful to show the differences between two text files. The main utilities for working with file differences are **diff** and **patch**. **diff** shows the differences between files. The output of **diff** can be processed by **patch** to apply the changes between two files to a file. “diffs” are also form the base of version/source management systems. The following sections describe **diff** and **patch**. To have some material to work with, the following two C source files are used to demonstrate these commands. These files are named `hello.c` and `hello2.c` respectively.

```
#include <stdio.h>

void usage(char *programName);

int main(int argc, char *argv[]) {
  if (argc == 1) {
    usage(argv[0]);
    return 1;
  }

  printf("Hello %s!\n", argv[1]);

  return 0;
}

void usage(char *programName) {
  printf("Usage: %s name\n", programName);
}
```

```
#include <stdio.h>
#include <time.h>

void usage(char *programName);

int main(int argc, char *argv[]) {
  if (argc == 1) {
    usage(argv[0]);
    return 1;
  }

  printf("Hello %s!\n", argv[1]);

  time_t curTime = time(NULL);
  printf("The date is %s\n", asctime(localtime(&curTime)));


  return 0;
```

```
}

void usage(char *programName) {
  printf("Usage: %s name\n", programName);
}
```

Listing differences between files

Suppose that you received the program `hello.c` from a friend, and you modified it to give the user the current date and time. You could just send your friend the updated program. But if a file grows larger, the can become uncomfortable, because the changes are harder to spot. Besides that, your friend may have also received modified program sources from other persons. This is a typical situation where **diff** becomes handy. **diff** shows the differences between two files. It most basic syntax is **diff file file2**, which shows the differences between `file` and `file2`. Let's try this with the our source files:

```
$ diff hello.c hello2.c
1a2 ❶
> #include <time.h> ❷
12a14,17
>   time_t curTime = time(NULL);
>   printf("The date is %s\n", asctime(localtime(&curTime)));
>
```

The additions from `hello2.c` are visible in this output, but the format may look a bit strange. Actually, these are commands that can be interpreted by the **ed** line editor. We will look at a more comfortable output format after touching the surface of the default output format.

Two different elements can be distilled from this output:

❶ This is an ed command that specified that text should be appended (`a`) after line `2`.
❷ This is the actual text to be appended after the second line. The “>” sign is used to mark lines that are added.

The same elements are used to add the second block of text. What about lines that are removed? We can easily see how they are represented by swapping the two parameters to **diff**, showing the differences between `hello2.c` and `hello.c`:

```
$ diff hello2.c hello.c
2d1 ❶
< #include <time.h> ❷
14,16d12
<   time_t curTime = time(NULL);
<   printf("The date is %s\n", asctime(localtime(&curTime)));
<
```

The following elements can be distinguished:

❶ This is the ed delete command (`d`), stating that line 2 should be deleted. The second delete command uses a range (line 14 to 17).
❷ The text that is going to be removed is preceded by the “<” sign.

That's enough of the ed-style output. Although, it is not part of the Single UNIX Specification Version 3 standards, almost all UNIX systems support so-called unified diffs. Unified diffs are very readable, and provide context by default. **diff** can provide unified output with the *-u* flag:

```
$ diff -u hello.c hello2.c
--- hello.c     2006-11-26 20:28:55.000000000 +0100 ❶
+++ hello2.c    2006-11-26 21:27:52.000000000 +0100 ❷
@@ -1,4 +1,5 @@ ❸
 #include <stdio.h> ❹
+#include <time.h> ❺

 void usage(char *programName);

@@ -10,6 +11,9 @@

   printf("Hello %s!\n", argv[1]);

+  time_t curTime = time(NULL);
+  printf("The date is %s\n", asctime(localtime(&curTime)));
+
   return 0;
 }
```

The following elements can be found in the output

❶ The name of the original file, and the timestamp of the last modification time.

❷ The name of the changed file, and the timestamp of the last modification time.

❸ This pair of numbers show the location and size of the chunk that the text below affects in the original file and the modified file. So, in this case the numbers mean that in the affected chunk in the original file starts at line 1, and is four lines long. In the modified file the affected chunk starts at line 1, and is five lines long. Different chunks in diff output are started by this header.

❹ A line that is not preceded by a minus (-) or plus (+) sign is unchanged. Unmodified lines are included because they give contextual information, and to avoid that too many chunks are made. If there are only a few unmodified lines between changes, **diff** will choose to make only one chunk, rather than two chunks.

❺ A line that is preceded by a plus sign (+) is an addition to the modified file, compared to the original file.

As with the ed-style **diff** format, we can see some removals by swapping the file names:

```
$ diff -u hello2.c hello.c

--- hello2.c    2006-11-26 21:27:52.000000000 +0100
+++ hello.c     2006-11-26 20:28:55.000000000 +0100
@@ -1,5 +1,4 @@
 #include <stdio.h>
-#include <time.h>

 void usage(char *programName);

@@ -11,9 +10,6 @@
```

```
   printf("Hello %s!\n", argv[1]);

-  time_t curTime = time(NULL);
-  printf("The date is %s\n", asctime(localtime(&curTime)));
-
   return 0;
 }
```

As you can see from this output, lines that are removed from the modified file, in contrast to the original file are preceded by the minus (-) sign.

When you are working on larger sets of files, it's often useful to compare whole directories. For instance, if you have the original version of a program source in a directory named `hello.orig`, and the modified version in a directory named `hello`, you can use the `-r` parameter to recursively compare both directories. For instance:

```
$ diff -ru hello.orig hello
diff -ru hello.orig/hello.c hello/hello.c

--- hello.orig/hello.c  2006-12-04 17:37:14.000000000 +0100
+++ hello/hello.c       2006-12-04 17:37:48.000000000 +0100
@@ -1,4 +1,5 @@
 #include <stdio.h>
+#include <time.h>

 void usage(char *programName);

@@ -10,6 +11,9 @@

   printf("Hello %s!\n", argv[1]);

+  time_t curTime = time(NULL);
+  printf("The date is %s\n", asctime(localtime(&curTime)));
+
   return 0;
 }
```

It should be noted that this will only compare files that are available in both directories. The GNU version of diff, that is used by FreeBSD and GNU/Linux provides the `-N` parameter. This parameters treats files that exist in only one of both directories as if it were an empty file. So for instance, if we have added a file named `Makefile` to the `hello` directory, using the `-N` parameter will give the following output:

```
$ diff -ruN hello.orig hello

diff -ruN hello.orig/hello.c hello/hello.c
--- hello.orig/hello.c  2006-12-04 17:37:14.000000000 +0100
+++ hello/hello.c       2006-12-04 17:37:48.000000000 +0100
@@ -1,4 +1,5 @@
```

```
 #include <stdio.h>
+#include <time.h>

 void usage(char *programName);

@@ -10,6 +11,9 @@

   printf("Hello %s!\n", argv[1]);

+  time_t curTime = time(NULL);
+  printf("The date is %s\n", asctime(localtime(&curTime)));
+
   return 0;
 }

diff -ruN hello.orig/Makefile hello/Makefile
--- hello.orig/Makefile 1970-01-01 01:00:00.000000000 +0100
+++ hello/Makefile      2006-12-04 17:39:44.000000000 +0100
@@ -0,0 +1,2 @@
+hello: hello.c
+       gcc -Wall -o $@ $<
```

As you can see the chunk indicator says that the chunk in the original file starts at line 0, and is 0 lines long.

UNIX users often exchange the output of **diff**, usually called “diffs” or “patches”. The next section will show you how you can handle diffs. But you are now able to create them yourself, by redirecting the output of diff to a file. For example:

```
$ diff -u hello.c hello2.c > hello_add_date.diff
```

If you have multiple diffs, you can easily combine them to one diff, by concatenating the diffs:

```
$ cat diff1 diff2 diff3 > combined_diff
```

But make sure that they were created from the same directory if you want to use the **patch** utility that is covered in the next section.

Modifying files with diff output

Suppose that somebody would send you the output of **diff** for a file that you have created. It would be tedious to manually incorporate all the changes that were made. Fortunately, the **patch** can do this for you. **patch** accepts diffs on the standard input, and will try to change the original file, according to the differences that are registered in the diff. So, for instance, if we have the `hello.c` file, and the patch that we produced previously based on the changes between `hello.c` and `hello2.c`, we can patch `hello.c` to become equal to its counterpart:

```
$ patch < hello_add_date.diff
patching file hello.c
```

If you have `hello2.c`, you can check whether the files are identical now:

```
$ diff -u hello.c hello2.c
```

There is no output, so this is the case. One of the nice features of **patch** is that it can revert the changes made through a diff, by using the *-R* parameter:

```
$ patch -R < hello_add_date.diff
```

In these examples, the original file is patched. Sometimes you may want to want to apply the patch to a file with a different name. You can do this by providing the name of a file as the last argument:

```
$ patch helloworld.c < hello_add_date.diff
patching file helloworld.c
```

You can also use **patch** with diffs that were generated with the *-r* parameter, but you have to take a bit of care. Suppose that the header of a particular file in the diff is as follows:

```
--------------------------
|diff -ruN hello.orig/hello.c hello/hello.c
|--- hello.orig/hello.c 2006-12-04 17:37:14.000000000 +0100
|+++ hello/hello.c      2006-12-04 17:37:48.000000000 +0100
--------------------------
```

If you process this diff with **patch**, it will attempt to change `hello.c`. So, the directory that holds this file has to be the active directory. You can use the full pathname with the *-p n*, where *n* is the number of pathname components that should be stripped. A value of *0* will use the path as it is specified in the patch, *1* will strip the first pathname component, etc. In this example, stripping the first component will result in patching of `hello.c`. According to the Single UNIX Specification version 3 standard, the path that is preceded by --- should be used to construct the file that should be patched. The GNU version of patch does not follow the standard here. So, it is best to strip off to the point where both directory names are equal (this is usually the top directory of the tree being changed). In most cases where relative paths are used this can be done by using *-p 1*. For instance:

```
$ cd hello.orig
$ patch -p 1 < ../hello.diff
```

Or, you can use the *-d* parameter to specify in which directory the change has to be applied:

```
$ patch -p 1 -d hello.orig < hello.diff
patching file hello.c
patching file Makefile
```

If you want to keep a backup when you are changing a file, you can use the *-b* parameter of **patch**. This will make a copy of every affected file named `filename.orig`, before actually changing the file:

```
$ patch -b < hello_add_date.diff
$ ls -l hello.c*
-rw-r--r-- 1 daniel daniel 382 2006-12-04 21:41 hello.c
-rw-r--r-- 1 daniel daniel 272 2006-12-04 21:12 hello.c.orig
```

Sometimes a file can not be patched. For instance, if it has already been patched, it has changed to much to apply the patch cleanly, or if the file does not exist at all. In this case, the chunks that could not be saved are stored in a file with the name `filename.rej`, where *filename* is the file that **patch** tried to modify.

5.2. Regular expressions

Introduction

In daily life, you will often want to some text that matches to a certain pattern, rather than a literal string. Many UNIX utilities implement a language for matching text patterns, *regular expressions* (regexps). Over time the regular expression language has grown, there are now basically three regular expression syntaxes:

- Traditional UNIX regular expressions.
- POSIX extended regular expressions.
- Perl-compatible regular expressions (PCRE).

POSIX regexps are mostly a superset of traditional UNIX regexps, and PCREs a superset of POSIX regexps. The syntax that an application supports differs per application, but almost all applications support at least POSIX regexps.

Each syntactical unit in a regexp expresses one of the following things:

- **A character**: this is the basis of every regular expression, a character or a set of characters to be matched. For instance, the letter *p* or the the sign ,.
- **Quantification**: a quantifier specifies how many times the preceding character or set of characters should be matched.
- **Alternation**: alternation is used to match “a or b” in which *a* and *b* can be a character or a regexp.
- **Grouping**: this is used to group subexpressions, so that quantification or alternation can be applied to the group.

Traditional UNIX regexps

This section describes traditional UNIX regexps. Because of a lack of standardisation, the exact syntax may differ a bit per utility. Usually, the manual page of a command provides more detailed information about the supported basic or traditional regular expressions. It is a good idea to learn traditional regexps, but to use POSIX regexps for your own scripts.

Matching characters

Characters are matched by themselves. If a specific character is used as a syntactic character for regexps, you can match that character by adding a backslash. For instance, \+ matches the plus character.

A period (.) matches any character, for instance, the regexp *b.g* matches *bag*, *big*, and *blg*, but not *bit*.

The period character, often provides too much freedom. You can use square brackets (*[]*) to specify characters which can be matched. For instance, the regexp *b[aei]g* matches *bag*, *beg*, and *big*, but nothing else. You can also match any character but the characters in a set by using the square brackets, and using the caret (^) as the first character. For instance, *b[^aei]g* matches any three character string that starts with *b* and ends with *g*, with the exception of *bag*, *beg*, and *big*. It is also possible to match a range of characters with a dash (-). For example, *a[0-9]* matches a followed by a single number character.

Two special characters, the caret (^) and the dollar sign (*$*), respectively match the start and end of a line. This is very handy for parsing files. For instance, you can match all lines that start with a hash (*#*) with the regexp ^#.

Quantification

The simplest quantification sign that traditional regular expressions support is the (Kleene) star (*). This matches zero or arbitrary instances of the preceding character. For instance, *ba** matches *b*, *babaa*, etc. You should be aware that a single character folowed by a star without any context matches every string, because *c** also matches a string that has zero *c* characters.

More specific repetitions can be specified with backslash-escaped curly braces. *\{x,y\}* matches the preceding character at least *x* times, but not more than *y* times. So, *ba\{1,3\}* matches *ba*, *baa*, and *baaa*.

Grouping

Backslash-escaped parentheses group various characters together, so that you can apply quantification or alternation to a group of characters. For instance, *\(ab\)\{1,3\}* matches *ab*, *abab*, and *ababab*.

Alternation

A backslash-escaped pipe vertical bar (\|) allows you to match either of two expressions. This is not useful for single characters, because *a\|b* is equivalent to *[ab]*, but it is very useful in conjunction with grouping. Suppose that you would like an expression that matches *apple* and *pear*, but nothing else. This can be done easily with the vertical bar: *(apple)|(pear)*.

POSIX extended regular expressions

POSIX regular expressions build upon traditional regular expressions, adding some other useful primitives. Another comforting difference is that grouping parenthesises, quantification accolades, and the alternation sign (|) are not backslash-escaped. If they are escaped, they will match the literal characters instead, thus resulting in the opposite behavior of traditional regular expressions. Most people find POSIX extended regular expressions much more comfortable, making them more widely used.

Matching characters

Normal character matching has not changed compared to the traditional regular expressions described in the section called “Matching characters”

Quantification

Besides the Kleene star (*), that matches the preceding character or group zero or more times, POSIX extended regular expressions add two new simple quantification primitives. The plus sign (+) matches the preceding character or group one or more times. For example, *a+*, matches *a* (or any string with more consecutive *a*'s), but does not match zero *a*'s. The questions mark character (*?*) matches the preceding character zero or one time. So, *ba?d* matches *bd* and *bad*, but not *baad* or *bed*.

Curly braces are used for repetition, like traditional regular expressions. Though the backslash should be omitted. To match *ba* and *baa*, one should use *ba{1,2}* rather than *ba\{1,2\}*.

Grouping

Grouping is done in the same manner as traditional regular expressions, leaving out the escape-backslashes before the parenthesises. For example, *(ab){1,3}* matches *ab*, *abab*, and *abababab*.

Alternation

Alternation is done in the same manner as with traditional regular expressions, leaving out the escape-backslashes before the vertical bar. So, *(apple)|(pear)* matches *apple* and *pear*.

5.3. grep

Basic grep usage

We have now arrived at one of the most important utilties of the UNIX System, and the first occasion to try and use regular expressions. The **grep** command is used to search a text stream or a file for a pattern. This pattern is a regular expression, and can either be a basic regular expression or a POSIX extended regular expression (when the `-E` parameter is used). By default, **grep** will write the lines that were matched to the standard output. In the most basic syntax, you can specify a regular expression as an argument, and **grep** will search matches in the text from the standard input. This is a nice manner to practice a bit with regular expressions.

```
$ grep '^\(ab\)\{2,3\}$'
ab
abab
abab
ababab
ababab
abababab
```

The example listed above shows a basic regular expression in action, that matches a line solely consisting of two or three times the *ab* string. You can do the same thing with POSIX extended regular expressions, by adding the `-E` (for extended) parameter:

```
$ grep -E '^(ab){2,3}$'
ab
abab
abab
ababab
ababab
abababab
```

Since the default behavior of **grep** is to read from the standard input, you can add it to a pipeline to get the interesting parts of the output of the preceding commands in the pipeline. For instance, if you would like to search for the string *2006* in the third column in a file, you could combine the **cut** and **grep** command:

```
$ cut -f 3 | grep '2006'
```

grepping files

Naturally, **grep** can also directly read a file, rather than the standard input. As usual, this is done by adding the files to be read as the last arguments. The following example will print all lines from the `/etc/passwd` file that start with the string *daniel:*.

```
$ grep "^daniel" /etc/passwd
daniel:*:1001:1001:Daniel de Kok:/home/daniel:/bin/sh
```

A non-standard extension to **grep** on FreeBSD and GNU/Linux also provides a parameter to recursively traverse a directory structure, trying to find matches in each file that was encountered during the traversal. This parameter, *-r* often proves to be very handy. Though, it is better to combine **find** with the *-exec* operand in scripts that have to be portable.

```
$ grep -r 'somepattern' somedir
```

is the non-portable functional equivalent of

```
$ find /somedir -type f -exec grep 'somepattern' {} \; -print
```

Pattern behavior

grep can also print all lines that do not match the pattern that was used. This is done by adding the *-v* parameter:

```
$ grep -Ev '^(ab){2,3}$'
ab
ab
abab
ababab
abababab
abababab
```

If you want to use the pattern in a case-insensitive manner, you can add the *-i* parameter. For example:

```
$ grep -i "a"
a
a
A
A
```

You can also match a string literally with the *-F* parameter:

```
$ grep -F 'aa*'
a
aa*
aa*
```

Using multiple patterns

As we have seen, you can use the alternation character (|) to match either of two or more subpatterns. If two patterns that you would like to match differ a lot, it is often more comfortable to make two separate patterns. **grep** allows you to use more than one pattern by separating patterns with a newline character. So, for example, if you would like to print lines that match either the *a* or *b* pattern, this can be done easily by starting a new line:

```
$ grep 'a
b'
a
a
b
b
c
```

This works, because quotes are used, and the shell passes quoted parameters literally. Though, it must be admitted that this is not quite pretty. **grep** accepts one or more `-e pattern` parameters, giving the opportunity to specify more than one parameter on one line. The **grep** invocation in the previous example could be rewritten as:

```
$ grep -e 'a' -e 'b'
```

Chapter 6. Process management

6.1. Theory

Processes

A running instance of a program is called a *process*. Each process has its own protected memory, named the *process address space*. This address space consists of two areas: the *text area* and the *data area*. The text area contains the actual program code, and tells the system what to do. The data area stores constant and runtime data of a process. Since there are many processes on a system, and only one or a few processors, the operating system kernel divides processor time between processes. This process is called *time-sharing*.

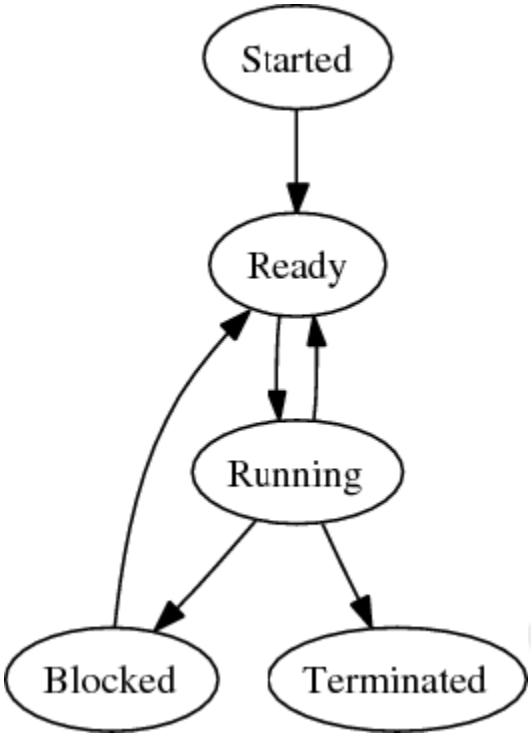
Table 6.1. The structure of a process

Field	Description
pid	The numeric process identifier
ppid	The process identifier of the parent process
euid	The effective user ID of the process.
ruid	The real user ID of the process
egid	The group ID of the process
rgid	The real group ID of the process
fd	Pointer to the list of open file descriptors
vmspace	Pointer to the process address space.

Table 6.1, “The structure of a process” lists the most important fields of information that a kernel stores about a process. Each process can be identified uniquely by its *PID* (process identifier), which is an unsigned number. As we will see later, a user can easily retrieve the PID of a process. Each process is associated with a *UID* (user ID) and *GID* (group ID) on the system. Each process has a *real UID*, which is the UID as which the process was started, and the *effective UID*, which is the UID as which the process operates. Normally, the effective UID is equal to the real UID, but some programs ask the system to change its effective UID. The effective UID determines is used for access control. This means that if a user named joe starts a command, say less, less can only open files that joe has read rights for. In parallel, a process also has an *real GID* and an *effective GID*.

Many processes open files, the handle used to operate on a file is called a *file descriptor*. The kernel manages a list of open file descriptors for each process. The *fd* field contains a pointer to the list of open files. The *vmspace* field points to the process address space of the process.

Figure 6.1. Process states



Not every process is in need of CPU time at a given moment. For instance, some processes maybe waiting for some *I/O* (Input/Output) operation to complete or may be terminated. Not taking subtleties in account, processes are normally *started*, *running*, *ready* (to run), *blocked* (waiting for I/O), or *terminated*. Figure 6.1, “Process states” shows the lifecycle of a process. A process that is terminated, but for which the process table entry is not reclaimed, is often called a *zombie process*. Zombie processes are useful to let the parent process read the exit status of the process, or reserve the process table entry temporarily.

Creating new processes

New processes are created with the *fork()* system call. This system call copies the process address space and process information of the caller, and gives the new process, named the child process, a different PID. The child process will continue execution at the same point as the parent, but will get a different return value from the *fork()* system call. Based on this return value the code of the parent and child can decide how to continue executing. The following piece of C code shows a *fork()* call in action:

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main() {
  pid_t pid = fork();
  if (pid == 0)
    printf("Hi, I am the child!\n");
  else
    printf("Hi, I am the parent, the child PID is %d!\n", pid);

  return 0;
```

```
}
```

. This little program calls *fork()*, storing the return value of *fork()* in the variable *pid*. *fork()* returns the value *0* to the child, and the PID of the child to the parent. Since this is the case, we can use a simple conditional structure to check the value of the *pid* variable, and print an appropriate message.

You may wonder how it is possible to start new programs, since the *fork()* call duplicates an existing process. That is a good questions, since with *fork()* alone, it is not possible to execute new programs. UNIX kernels also provide a set of system calls, starting with *exec*, that load a new program image in the current process. We saw at the start of this chapter that a process is a running program -- a process was constructed in memory from the program image that is stored on a storage medium. So, the *exec* family of system calls gives a running process the facilities to replace its contents with a program stored on some medium. In itself, this is not wildly useful, because every time the an *exec* call is done, the original calling code (or program) is removed from the process. This can be witnessed in the following C program:

```
#include <stdio.h>
#include <unistd.h>

int main() {
  execve("/bin/ls", NULL, NULL);

  /* This will never be printed, unless execve() fails. */
  printf("Hello world!\n");

  return 0;
}
```

This program executes **ls** with the *execve()* call. The message printed with *printf()* will never be shown, because the running program image is replaced with that of **ls**. Though, a combination of *fork()* and the *exec* functions are very powerful. A process can fork itself, and let the child “sacrifice” itself to run another program. The following program demonstrates this pattern:

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main() {
  pid_t pid = fork();
  if (pid == 0)
    execve("/bin/ls", NULL, NULL);

  printf("Hello world!");

  return 0;
}
```

This program forks itself first. The program image of the child process will be replaced with **ls**, while the parent process prints the “Hello world!” message to the screen and returns.

This procedure is followed by many programs, including the shell, when a command is executed from the shell prompt. In fact all processes on a UNIX system are directly or indirectly derrived from the *init* process, which is the first program that is started on a UNIX system.

Threads

Although forks are very useful to build some parallelism[1], they can be to expensive for some purposes. Copying the whole process takes some time, and there is cost involved if processes want to share data. This is solved by offering a more lightweight alternative, namely allowing more than one thread of execution. Each thread of execution is executed separately, but the process data is shared between the threads.

Writing good *multithreaded* programs requires good knowledge of data sharing and locking. Since all data is shared, uncareful programming can lead to bugs like race conditions.

6.2. Analyzing running processes

Listing running processes

UNIX systems provide the **ps** command to show a list of running processes. Unfortunately, this command is an example of the pains of the lack of standardization. The BSD and System V variants of **ps** have their own sets of options. GNU/Linux implements both the System V and BSD-style parameters, as well as some (GNU-style) long options. Of the systems covered in this book, FreeBSD uses BSD-style options, Solaris SysV options, and on GNU/Linux options preceded by a dash are interpreted as System V options and options without a dash as GNU options.

Fortunately, the designers of the Single UNIX Specification have attempted to standardize the **ps** options. The writers of the standard attempted to make the standard conflict with existing options as little as possible. This section will describe **ps** according to the SUSv3 standard, while also describing exceptions to the standard. This section will describe **ps** according to the standard, noting exceptions where they apply.

If **ps** is used without any parameters, it shows all processes owned by the user that invokes **ps** and that are attached to the same controlling terminal. For example:

```
$ ps
  PID TTY          TIME CMD
 8844 pts/5    00:00:00 bash
 8862 pts/5    00:00:00 ps
```

A lot of useful information can be distilled from this output. As you can see, two processes are listed: the shell that we used to call **ps** (*bash*), and the **ps** command itself. In this case there are four information fields. *PID* is the process ID of a process, *TTY* the controlling terminal, *TIME* the amount of CPU time the proces has used, and **CMD** the command or program of which a copy is running. The fields that are shown by default may vary a bit per system, but usually at least these fields are shown, with somewhat varying field labels.

Sometime you may want to have a broader view of processes that are running. Adding the `-a` option shows all processes that are associated with terminals. For instance:

```
$ ps -a
  PID TTY          TIME CMD
 7487 pts/1    00:00:00 less
 8556 pts/4    00:00:10 emacs-x
11324 pts/3    00:00:00 ps
```

[1] For instance, a web server could fork multiple child processes to handle requests

As you can see, processes with different controlling terminals are shown. Though, in contrast to the plain **ps** output, only processes that control the terminal at the moment are shown. For instance, the shell that was used to call **ps** is not shown.

You can also print all processes that are running, including processes that are not associated with a terminal, by using the *-A* option:

```
$ ps -A | head -n 10
  PID TTY          TIME CMD
    1 ?        00:00:01 init
    2 ?        00:00:00 migration/0
    3 ?        00:00:00 ksoftirqd/0
    4 ?        00:00:00 watchdog/0
    5 ?        00:00:00 migration/1
    6 ?        00:00:00 ksoftirqd/1
    7 ?        00:00:00 watchdog/1
    8 ?        00:00:00 events/0
    9 ?        00:00:00 events/1
```

You can print all processes with a certain user ID, with the *-U* option. This option accepts a user name as a parameter, or multiple user names that are separated by a comma. The following command shows all processes that have *xfs* or *rpc* as their user ID:

```
$ ps -U xfs,rpc
  PID TTY          TIME CMD
 2409 ?        00:00:00 portmap
 2784 ?        00:00:00 xfs
```

Likewise, you can also print processes with a particular group ID, with the *-G* option:

```
$ ps -G messagebus,haldaemon
  PID TTY          TIME CMD
 8233 ?        00:00:00 dbus-daemon
11312 ?        00:00:00 hald
11320 ?        00:00:00 hald-addon-keyb
11323 ?        00:00:00 hald-addon-acpi
```

If you would like to have a list for a physical or pseudo-terminal, you can use the *-t* option:

```
$ ps -t tty2
  PID TTY          TIME CMD
 2655 tty2     00:00:00 getty
```

6.3. Managing processes

Sending singals to a process

Signals are a crude, but effective form of inter-process communication (IPC). A signal is basically a number that is delivered to a process that has a special meaning. For all signals there are default signal handlers. Processes can install their own signal handlers, or choose to ignore signals. Some signals (normally SIGKILL and SIGSTOP) can not be ignored. All signals have convenient symbolic names.

Only a few signals are normally interesting for interactive use on UNIX(-like) systems. These are (followed by their number):

- *SIGKILL (9):* forcefully kill a process.
- *SIGTERM (15):* request a process to terminate. Since this is a request, a program could ignore it, in contrast to *SIGKILL*.
- *SIGHUP (1):* Traditionally, this has signalled a terminal hangup. But nowadays some daemons (e.g. *inetd*) reread their configuration when this signal is sent.

The **kill** command is used to send a signal to a process. By default, **kill** sends the *SIGTERM* signal. To send this signal, the process ID of the process that you would like to send this signal to should be added as a parameter. For instance:

```
$ kill 15631
```

To send another signal, you can use one of two options: *-signalnumber* or *-signalname*. So, the following commands both send the *SIGKILL* signal to the process with process ID *15631*:

```
$ kill -9 15631
```

```
$ kill -SIGKILL 15631
```

Being nice to others

In an act of altruism you can be nice to other users of computer resources. If you plan to run a CPU-time intensive process, but do not want that to interfere with work of other users on the system (or other processes), you can assign some grade of 'niceness' to a process. Practically, this means that you will be able to influence the scheduling priority of a process. Nicer processes get a lower scheduling priority. The normal niceness of a process is *0*, and can be changed by executing a program with the **nice** command. The *-n [niceness]* option can be used to specify the niceness:

```
$ nice -n 20 cputimewaster
```

The maximum number for niceness is implementation-dependent. If a program was started with **nice**, but no niceness was specified, the niceness will be set to *10*. In case you were wondering: yes, you can also be rude, but this right is restricted to the *root* user. You can boost the priority of a process by specifying a negative value as the niceness.

You can also modify the niceness of a running processes with the **renice** command. This can be done for specific process IDs (`-p PIDs`), users (`-u user/uid`), and effective groups (`-g group/gid`). The `-n` option is used specify how much the niceness should be increased.

These are some examples that respectively increase the niceness of some PIDs, users, and groups:

```
$ renice -n 10 -p 3108 4022
$ renice -n 10 -u daniel
$ renice -n 10 -g mysql
```

The niceness of a process can only be increased. And, of course, no user except for *root* can affect the niceness of processes of other users.

GNU/Linux systems do not follow the standard here, and do not accept the *-n* option to **renice**. Instead its **renice** command expects the absolute niceness (not an incremental step up) as the first option. For instance, to set the niceness of a process with PID *3108* to *14*, you could use the following command:

```
$ renice 14 -p 3108
```

FreeBSD also accepts this syntax of **renice**. With this syntax, no one but the *root* user can set the niceness of a process lower than the current niceness.

6.4. Job control

It is often useful to group processes to allow operations on a set of processes, for instance to distribute a signal to all processes in a group rather than a single process. Not too suprisingly, these sets of processes are called *program groups* in UNIX. After a fork, a child process is automatically a member of the process group of the parent. Though, new process groups can be created by making one process a process group leader, and adding other processes to the group. The process group ID is the PID of the process group leader.

Virtually all modern UNIX shells give processes that are created through the invocation of a command their own process group. All processes in a pipeline are normally added to one process group. If the following commands that create a pipepine are executed

```
cat | tr -s ' ' | egrep 'foob.r'
```

the shell roughly performs the following steps:

1. Three child processes are forked.
2. The first process in the pipeline is put in a process group with its own PID as the process group ID, making it the process leader. The other processes in the pipeline are added to the process group.
3. The file descriptors of the processes in the pipeline are reconfigured to form a pipeline.
4. The programs in the pipeline are executed.

The shell uses process groups to implement *job control*. A shell can run multiple jobs in the background, there can be multiple stopped job, and one job can be in the foreground. A foreground job is wired to the terminal for its standard input (meaning that it is the job the gets user input).

Stopping and continuing a job

A job that is in the foreground (thus, a job that potentially accepts userinput from the terminal) can be stopped by pressing *Ctrl-z* (pressing both the 'Ctrl' and 'z' keys simultaneously). This will stop the job, and will handle control over the terminal back to the shell. Let's try this with the **sleep** command, which waits for the number of seconds provided as an argument:

```
$ sleep 3600
Ctrl-z
[1]+  Stopped                 sleep 3600
```

The process group, which we will refer to as a job has been stopped now, meaning the the **sleep** has stopped counting - it's execution is completely stopped. You can retrieve a list of jobs with the **jobs** command:

```
$ jobs
[1]+  Stopped                 sleep 3600
```

This shows the job number (*1*), its state, and the command that was used to start this job. Let's run another program, stop that too, and have another look at the job listing.

```
$ cat
Ctrl-z
[2]+  Stopped                 cat
$ jobs
[1]-  Stopped                 sleep 3600
[2]+  Stopped                 cat
```

As expected, the second job is also stopped, and was assigned job number *2*. The plus sign (+) following the first job has changed to a minus (-) sign, while the second job is now marked by a plus sign. The plus sign is used to indicate the *current job*. The **bg** and **fg** commands that we will look at shortly, will operate on the current job if no job was specified as a parameter.

Usually, when you are working with jobs, you will want to move jobs to the foreground again. This is done with the **fg** command. Executing **fg** without a parameter will put the current job in the foreground. Most shells will print the command that is moved to the foreground to give an indication of what process was moved to the foreground:

```
$ fg
cat
```

Of course, it's not always useful to put the current job in the foreground. You can put another job in the foreground by adding the job number preceded by the percentage sign (%) as an argument to **fg**:

```
$ fg %1
sleep 3600
```

Switching jobs my stopping them and putting them in the foreground is often very useful when the shell is used interactively. For example, suppose that you are editing a file with a text editor, and would like to execute some other command and then continue editing. You could stop the editor with *Ctrl-z*, execute a command, and put the editor in the foreground again with **fg**.

Background jobs

Besides running in the foreground, jobs can also run in the background. This means that they are running, but input from the terminal is not redirected to background processes. Most shells do configure background jobs to direct output to the terminal of the shell where they were started.

A process that is stopped can be continued in the background with the **bg** command:

```
$ sleep 3600

[1]+  Stopped                 sleep 3600
$ bg
[1]+ sleep 3600 &
$
```

You can see that the job is indeed running with **jobs**:

```
$ jobs
[1]+  Running                 sleep 3600 &
```

Like **fg**, you can also move another job than the current job to the background by specifying its job number:

```
$ bg %1
[1]+ sleep 3600 &
```

You can also run put a job directly in the background when it is started, by adding an trailing ampersand (&) to a command or pipeline. For instance:

```
$ sleep 3600 &
[1] 5078
```

Bibliography

Books

[SUSv3] Copyright © 2001-2004 The IEEE and The Open Group. *The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition* .