13

USING THE COMMAND LINE

Some might ask, *Why would I want to use the command line?*

The best answer is that it can save you a lot of time, particularly if you are doing repetitive tasks. For example, sophisticated editing can be performed in vi or Emacs with many fewer keystrokes than when performing the same task using a mouse-based GUI editor. And for repetitive editing, programs such as sed can automate the task for you, as can awk for report generation.

Before we scare you away, you can be a successful user of your KDE environment without ever seeing the command line. If you spend most of your time in an office suite, such as OpenOffice, or running a custom application, you can skip this chapter confidently and get on with your life with KDE.

If, on the other hand, you need the power of the command line, KDE offers a lot of advantages:

- Support for multiple terminal windows on one graphical screen.
- Selection of various screen and character sizes.
- Session management, including the ability to send signals.
- Emulation of different terminal types and support for various character sets.

Terminal Windows

To enter commands you must open a terminal window. This is a program that brings up a text-based window and starts a command interpreter (called a shell). See "The Shell" later in this chapter.

The easiest way to start a terminal window is to click on the terminal icon on the KDE panel.

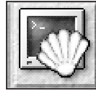

Figure 13.1: Terminal Icon

An alternate way of starting a terminal window is by pressing ALT-F2 to bring up a Run Command dialog box and entering the name of a terminal application, such as Konsole or xterm.

TIP *With the combination of multiple desktops, which you learned about in Chapters 2 and 11, and terminal windows you can have tens or even hundreds of shells running. This could prove to be very useful if you need to monitor many remote systems.*

Konsole

The default terminal application in KDE is Konsole. Another common terminal application is xterm. By default, the icon will bring up Konsole.

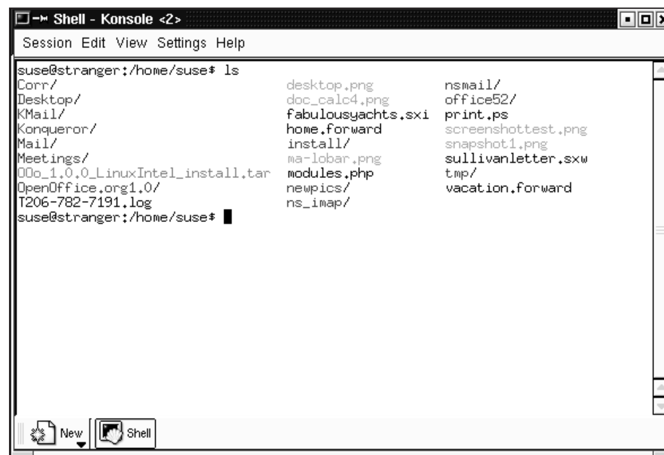

Figure 13.2: Konsole Terminal Window

The Konsole terminal window has menus across the top, a scroll bar on the right, and icons on the bottom. The first icon, labeled New, makes it possible to create additional terminals within this single KDE window. Following the New

icon are the icons for the terminals you create. By default you have only one, but each click on New will create another. You can then click on the terminal icons to switch between them.

TIP *You can use CTRL and the left and right arrow keys to switch between the terminal windows within Konsole.*

The menus allow you to perform various configurations. In the File menu, you can request various types of terminals or exit Konsole completely.

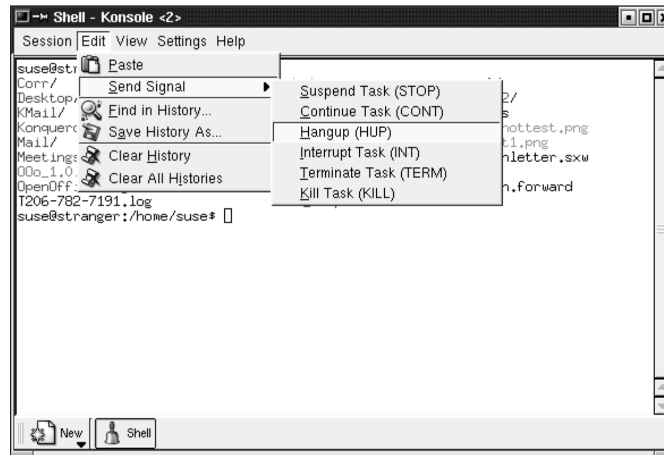

Figure 13.3: Konsole with Sessions and Signal Menus

The Edit menu allows you to send a signal to the current terminal and to rename a terminal session. Send Signal allows you to send an asynchronous event to the program running in your current terminal window. Signals and signal handling are advanced topics, most of which are beyond the scope of this book. A simple example, however, is the hangup signal. If you were connected remotely to a computer system and the connection was dropped (by a call-waiting interrupt on a phone line, for example), the programs you were running would be sent a hangup signal. The programs then have the option of deciding what they want to do with this signal.

You can use the Rename session menu item to change the names of your terminal windows. By default, they are labeled as Terminal or Shell and are numbered sequentially. Changing the name can make it a lot easier to remember which window does what.

TIP *Some distributions include a command-line command, titlebar, or xttitle, that allows you to change any terminal window title; simply type* `titlebar` *or* `xttitle` *followed by a space and whatever you would like to name the terminal.*

The Settings menu allows you to configure the terminal window.

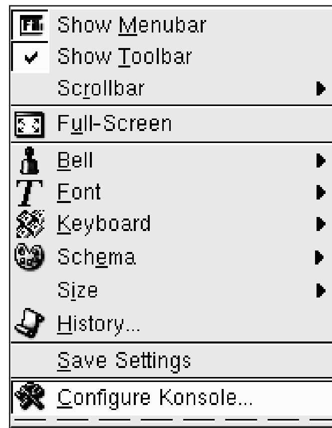

Figure 13.4: Konsole with Settings Menu

The Schema configuration options let you set a combination of foreground and background colors, sizes, and fonts. Size allows you to change the number of columns and lines in the window, and font allows you to change the size of the characters within the window.

The Show options allow you to turn on and off the various types of ancillary information around the outside of the window itself. You can turn the bottom menubar, the top toolbar, and the frame border on and off. Don't panic if you turn off the toolbar. You can also call up the Settings menu by right-clicking your mouse while the cursor is within the terminal window.

These settings are only for the current session within Konsole. If you want them to affect future invocations, use the Save Settings button.

The Help menu gives information about Konsole or, by selecting Contents, lets you search the online manual.

xterm

A second type of terminal window is xterm. It can be started by entering `xterm` in a Run Command box. You should know about xterm only because there may be cases where Konsole is not recognized by a remote system. This can happen if the remote system was, for example, a mainframe running something other than Linux. Generally, Konsole is a better choice and has more capabilities.

The Shell

The shell is the program that interprets the commands you enter and invokes the appropriate programs to perform the requested tasks. In other words, the shell performs some basic functions that make it easier for you to communicate with programs. These functions include:

- Command-line editing: enables you to backspace and edit within the line you are entering.
- Job control: allows you to control (start, stop, and terminate) multiple programs from one shell invocation.
- Control of input and output files: normally a program receives input from your keyboard and sends output to your screen. The shell gives you the ability to change the location of these data files to other programs or files on your computer system.
- Command history: lets you call up, edit, and execute previously entered commands.

Shell Choices

Linux systems come with a choice of shells. Each has its advantages and disadvantages. For example, some programmers find the editing features of csh (called the C shell) to be very appealing. Some see the compact size of ash as an important consideration, and for some, the standardized nature of ksh (called the Korn shell) eases their transition between Linux systems and other UNIX platforms.

The most popular shell among Linux users is Bash, which stands for Bourne Again Shell. The name comes from UNIX history. The first shell that offered its own programming language—that is, the ability to make control decisions within a saved set of commands based on conditions that could be tested for—was written by Steven Bourne of, at the time, AT&T. Bash took these capabilities and has expanded them over the years.

Learning all there is to know about a shell is easily the subject for a whole book. For the purposes of this chapter, we will use Bash as our shell. Most of the information presented, however, applies to virtually all available shells.

TIP *The shell tcsh, an enhanced clone of csh from UC Berkeley, is available for those who are familiar with csh and want to use its advanced history and scripting features.*

Shell Prompt

The character string that the shell displays when it is ready to accept your command input is called the shell prompt. This prompt can be configured to be as simple as a single character or it can be generated dynamically each time.

The default shell prompt is typically a combination of useful information about your current environment. As I write this, my prompt reads

```
fyl@nicaragua:~/KDE/Command >
```

This string offers three pieces of dynamic information:

- fyl is my user name.
- nicaragua is the name of this computer.
- ~/KDE/Command is the name of my current directory. Note that ~ is shorthand for my home directory.

Occasionally you may see a single > character when you are expecting either your usual shell prompt or the prompt from an applications program. The > character is the default secondary shell prompt and means the shell is expecting more input from you. Most commonly, this situation occurs when you have typed one quotation mark and the shell is expecting you to type a matching quotation mark. The quickest way out of this problem is to press CTRL-C. This will end the execution attempt and return you to the shell prompt.

TIP *The shell prompt can be changed by setting the* `PS1` *environment variable.*

Control Characters

The shell interprets various control characters allowing you to edit the lines you enter and perform other local tasks. Signify that you have completed entering a command line by pressing the ENTER key. Prior to pressing ENTER, the two basic editing keys are BACKSPACE and CTRL-U. Pressing the BACKSPACE key erases the most recently entered character. Entering CTRL-U deletes all the characters you have entered on the current line, allowing you to start over.

Once you have pressed the ENTER key, control is turned over to the application. For example, if you type in `ls -l` and press ENTER, the shell turns over control to the ls program. Thus, the BACKSPACE and line-delete character sequences can no longer be interpreted by the shell. If you want to terminate the execution of a program, you can press CTRL-C. This will send an interrupt to the program.

If you want to suspend the execution of a program, press CTRL-Z. This suspends the program and returns you to the shell prompt. Entering the `fg` command (followed by ENTER) will resume the suspended program.

You can also enter `bg` (followed by ENTER) to resume the suspended program in the background. This means the program will be restarted, but you will receive a shell prompt. This is useful when you want to run a time-consuming report program and redirect both its input and output streams to files.

Finally, you can use CTRL-S to pause the output of a running command. CTRL-Q resumes the output. Note that program output may scroll by too fast to make effective use of these control sequences. Generally, you will be better off using a pager program such as more or less, both of which allow you to view the output a screenful at a time. For example, if the output of the ls -l command is more than a screenful, you can enter `ls -l | more`.

TIP *Some programs require that you send them an End-of-File (EOF) character to signify that you have completed your input. To send this character, press CTRL-D.*

Pipes and Redirection

By default, the shell establishes three connections with the terminal window. These connections are called standard input, standard output, and standard error. In UNIX documentation you will commonly see them abbreviated as stdin, stdout, and stderr.

The stdin connection is connected to the keyboard. Programs read your keyboard input by reading from stdin. The stdout connection is connected to your screen and is the way that programs send their output back to you. The stderr connection is also connected to your screen and is the way programs send output about problems and abnormalities.

There are two output streams (stdout and stderr) so that it is possible to send error information to your screen, even though the normal output of the command is being sent somewhere else (to a file or possibly to another program).

The tools to connect these streams to other places are called pipes and redirection. A pipe allows you to connect the output of one program to the input of another program, that is, connect stdout of one program to stdin of another program.

For example, say you have a program called bigreport that produces a great deal of output, and you want to view it a screenful at a time. We previously discussed the more program that allows viewing by the screenful.

Thus, you want to run the program bigreport and send its output (stdout) to the input (stdin) of more. The vertical bar character (|) is used to represent the pipe, so the command to perform this task looks like this:

```
bigreport | more
```

The preceding example shows spaces around the pipe character. These are allowed but not necessary. The following command would work also:

```
bigreport|more
```

Of course, you probably don't have a program called bigreport available to you. So, to practice, enter a command that will produce a lot of output. Try the following command:

```
ls /bin | more
```

You should see output like this:

```
Shell - Konsole <4>
Session Edit View Settings Help
suse@manyroads:/home/suse$ ls /bin | more
arch*
ash*
ash.static*
awk@
basename*
bash*
cat*
chgrp*
chmod*
chown*
chvt*
cp*
cpio*
csh@
date*
dd*
deallocvt*
df*
dmesg*
dnsdomainname@
domainname@
echo*
--More--
New Shell
```

Figure 13.5: Stdout Piped to more

Note that the word More appears at the bottom of the screen in reverse video. This tells you that you are still in the more program rather than at the shell prompt. Pressing the space bar will display the next screenful. Pressing `Q` will terminate more and return you to the shell prompt.

The more program and its more capable cousin less are very handy. To find out about additional capabilities, enter ? at the more prompt.

With pipes we connect two programs together. With redirection we can tell a program to read either input from a file or write output to a file. To redirect stdout to a new file, use the > character. Going back to the previous example, we could get the ls command to write its output to the file ls.output by entering the following command:

```
ls /bin > ls.output
```

As with pipes, the spaces around the redirection character are optional.

Now that we have this information in a file, the next trick is actually reading it. All we need to do is use the more command again and redirect its input (stdin) from the file we created. The input redirection operator is <, so we enter:

```
more < ls.output
```

6594 LITW Chapter 13 9/12/02 3:33 PM Page 275

The operation of more is virtually identical to the way it was before—that is, the shell handled piping, and here it handles redirection. The more program itself knows nothing of the actual source of the data. Contrast this with `more ls.output` where more itself opens the file.

Besides these two forms of redirection, there are others. The following table lists other forms of redirection.

Character	What It Does
>	Redirects output (stdout) of the command to the specified file.
>>	Similar to > but will append to a file if it already exists.
2>	Redirects the error output (stderr) of the command to the specified file.
<	Redirects the input of the command (stdin) from the specified file.
|	Joins two commands. Output (stdout) of the command on the left is sent to the input (stdin) of the command on the right.

Command-Line Format

The previous section introduced some basic command-line formats. Now it's time to see how command lines really work.

- Command lines are made up of words.
- Words are separated by whitespace, which is defined as one or more spaces or tabs, or a combination thereof.
- The first word of the line is the command itself. This is the program you are asking the shell to run. It could be a binary program, an alias, or a script to be interpreted by the shell, or one of the many other interpreted languages, such as awk or Python.
- The remaining words on the command line are called arguments.
- A special argument, called an option, begins with a hyphen (-). Long options, a special kind of option, begin with two hyphens (--). Generally, options are used to modify the way a command works.
- Multiple commands can be entered on one line by separating them with a semicolon (;).
- You can tell the shell to run a command in the background and thus not tie up your terminal screen by following it with an ampersand (&).

Examples

Display a long (detailed) listing of all the files in the directory /bin paged through more:

```
ls -l /bin | more
```

Print the date on the terminal, and then write a long list of all files in the directory /bin to the file /tmp/testme. Note that the redirection applies only to the ls command, not to the date command:

```
date; ls -l /bin > /tmp/testme
```

Command-Line Editing and Shell History

The shell keeps a record of your recent commands and allows you to retrieve them, edit them, and re-execute them. Depending on the mode in which your shell is set, the history access and editing will work much like either the Emacs editor or the vi editor. That is, all the text-editing features of either vi or Emacs that make sense to use on a single line are available within the shell. To select vi editor mode, enter `set -o vi` at your shell prompt. To select Emacs editing mode, enter `set -o emacs` at your shell prompt.

You can make a permanent change to your editing mode by adding the appropriate set command to the end of your .bashrc file in your home directory. Note that the change will not take effect until your next login.

If you are not familiar with either Emacs or vi, you will most likely do better in the Emacs mode. In this mode, the up and down arrow keys can be used to move through the history list, and diligent use of the left and right arrows, as well as the BACKSPACE and DELETE keys, allow you to edit the command. Pressing ENTER executes the resulting command.

If you are in vi mode, press ESC (ESCAPE), and then any vi command that works on a single line can be used. Use the up and down arrows, or J and K, to move within the lines of the history list.

Command Completion

Besides providing command history, Bash will *guess* the name you mean if you are entering the name of a file. Once you have entered enough of a filename to make it unique, press TAB. If it is in fact unique, Bash will fill in the rest of the name. If it is not, nothing happens. At that point you can, however, press TAB again, and the shell will list all the possible matches and wait for you to continue entering characters.

Shell Expansions

A special shorthand called shell expansions is available to allow you to select multiple filenames on command lines. This expansion capability is one of the most powerful capabilities of the shell that is useful on the command line. Properly used, it can save you many keystrokes and much time.

? Matches any single character.

* Matches any number (zero or more) of any characters.

~ Shorthand for the name of a home directory. ~*name* is replaced with the pathname of the home directory of user *name*. If name is omitted, ~ is replaced by the pathname of your home directory.

[ ] Encloses a set of characters for a single-character match. For example, [aQz] matches a, Q, or z. A hyphen (-) can be used to specify a range. For example, [am-pZ] matches a, m, n, o, p, or Z. A leading ^ negates the match string. For example, [^a-z] matches anything except a lowercase letter.

$ Introduces a shell variable. The variable is replaced with its value as saved by the shell. For example, $HOME is replaced with the pathname of your home directory.

\ Turns off the special meaning of the following character. For example, \? matches a ? instead of any single character, and \[matches a [instead of introducing a set of matched characters.

TIP *Don't confuse shell expansions with regular expressions. They have some similarities, but they have different capabilities available in different places.*

Quoting

Besides the use of backslash (\) to protect a character from shell expansion or interpretation, there are other quoting characters. Of course, a backslash can be used to turn off their special meaning.

" Disables the special meaning of all characters except $ in the enclosed string. This is commonly used when you want to treat multiple words as a single argument.

' Disables the special meaning of all characters in the enclosed string.

` Executes the enclosed string as a command and replaces it with the output of the executed command.

Examples

The echo command displays its arguments back to the screen. In this example, echo displays the string entered, but $HOME will be replaced with the value of the HOME environment variable, which is the pathname of your home directory.

```
echo "My home directory is $HOME"
```

In this example, $HOME is displayed literally instead of being replaced with the value of the environment variable.

```
echo 'My home directory is $HOME'
```

The date command returns the current date and time as a 26-character string. In this example, the backquoted reference is replaced by the result of executing the date command before the line is printed:

```
echo "The current time is `date`"
```

Conventions

The remainder of this chapter uses certain conventions in its examples as a metalanguage to help you understand the commands.

`Monospaced font`	Monospaced font is used to indicate examples.
[ ]	Square brackets are used to show optional information that may be included within a command and shouldn't actually be typed. Don't confuse this use with the character-matching operation.
Boldface	Boldface identifies the proper syntax for typing particular commands—in other words, the way the commands should be set up and typed.
italic	Parameters to be replaced are shown in italic. The word used is intended to be descriptive. The most common example is *file* to represent a filename. Plurals (for example, *files*) are used to indicate where multiple instances are permitted.
option	The option keyword can be replaced with any of the available command options. Using options indicates that more than one may be used.

File Hierarchy

You learned about files back in Chapter 3, where you could move around the file hierarchy by pointing and clicking. When using the shell, you need to know how to move around the hierarchy because you don't get any graphical clues.

TIP *Although Linux allows any character (including control characters) in filenames, it is best to stick to letters, numbers, and some safe, special characters, such as _, ., and ,.*

Naming the Pieces

By convention, the slash character (/) is used to separate the levels in the file hierarchy. The top level of the whole file hierarchy, usually referred to as the root, is identified by /, and all other files are located relative to it.

The location of a file within the hierarchy is called its pathname. Pathnames can be specified relative to your current directory—that is, where you are now—or relative to the root directory. A pathname starting with / signifies that it is relative to the root and is called a full pathname. If a pathname does not start with a /, it is relative to your current directory and is called a relative pathname.

Each directory in the hierarchy has two special files in it named . (dot) and .. (dot dot): . is a synonym for the current directory, and .. is a synonym for the parent of the current directory. They are always there, so you can use them as shorthand to help you move around the hierarchy.

Examples

My home directory is called fyl and lives under the directory home, which is off the root of the hierarchy. The full pathname of my home directory is `/home/fyl`.

If I am in my home directory, as described in the preceding example, and your home directory is called jill, also located under home, then `/home/jill` is the full pathname to your home directory. `../jill` is the relative pathname to your home directory.

Moving About

To move about the hierarchy and know where you are, there are two commands you need to know: cd and pwd.

The simplest, pwd, tells you the name of your current directory. It takes no arguments.

You can use cd in three ways. First, just entering `cd` with no arguments will move you to your home directory. You can, of course, check the results by entering `pwd`.

If you know the location of a directory relative to the root, you can follow cd with the full pathname of the desired directory. For example, many of the programs you will use from the command line are located in the directory /usr/bin. Entering `cd /usr/bin` will change /usr/bin to your current directory. Similarly, entering `cd /var/spool` will make the system spool directory your current directory.

File Manipulation Utilities

Now that you can move around the filesystem and know what commands look like, it's time to try a few.

ls—Listing Directory Contents

You have already seen this command in earlier examples. It produces a list of names and, if so requested, other information, about the specified files. The most basic case is `ls`, which lists all files in your current directory whose name does not start with a dot (.).

The general form of the command is: **ls** [*options*] [*files*]

Common Options

`-a` Includes files whose names start with a dot. (Linux treats any file whose name starts with a dot as *hidden*, meaning that the names are not displayed by default.)

`-l` Includes file size, ownership, and permission information.

`-R` Recursive—includes subdirectories.

`-t` Sorts by last modification time.

`-u` Sorts by time of last access instead of last modify.

Examples

Lists all files in your current directory sorted by last modify time

```
ls -lt
```

Lists files in /usr/bin whose name starts with gif; make it a long list:

```
ls -l /usr/bin/gif*
```

cp—Copy Files

The cp command copies one or more files. In the most basic form, you specify a source filename and a destination filename:

```
cp [options] source destination
```

If the final argument is a directory, then all specified files are copied to that directory under their same name:

```
cp [options] files directory
```

Common Options

-a	Archives by preserving the file attributes and including subdirectories but not following symbolic links.
--help	Displays a help message.
-p	If possible, preserves file attributes such as owner, modify time, and access permissions.
-R	Recursively includes subdirectories.
-v	Explains what is being done.

Examples

Make a copy of harry in your current directory. The copy is given the new name chest in the directory tmp.

```
cp harry /tmp/chest
```

Copy all files whose names start with a or b in /tmp to your current directory, preserving file attributes:

```
cp -p /tmp/[ab]* .
```

Copy all files in the directory /home/bill/secret and any subdirectories into /tmp/savebill.

```
cp -R /home/bill/secret/* /tmp/savebill
```

mv—Rename or Move Files

The mv command moves or renames files. You need not concern yourself with the distinction, as mv does what is necessary. Much like the cp command, there are two formats.

In the most basic form, you just specify a source filename and a destination filename:

```
mv [options] source destination
```

Common Options

`--help`	Displays a help message.
`-i`	Prompts before overwriting any files (interactive).
`-u`	Moves only older or brand-new nondirectories.
`-v`	Explains what is being done.

TIP *Some Linux distributions include a shell alias that sets the* `-i` *option by default. Type* `alias` *at the shell prompt to see your current list of aliases.*

Examples

Move the file harry in your current directory to /tmp, and name the new file chest in the directory tmp:

```
mv harry /tmp/chest
```

Move all files whose names start with a or b in /tmp to your current directory, prompting before overwriting any files:

```
mv -i /tmp/[ab]* .
```

rm—Remove Files

The rm command deletes one or more files. It can also be used to force the removal of directories.

```
rm [options] files
```

Common Options

`-f`	Forces removal without prompting.
`--help`	Displays a help message.
`-i`	Prompts before any removal (interactive).
`-r`	Recusively removes the contents of directories.
`-R`	Recursively includes subdirectories.
`-v`	Explains what is being done.

Examples

Remove the file harry from your current directory:

```
rm harry
```

Remove all files whose names start with a or b in /tmp:

```
rm /tmp/[ab]*
```

Remove the directory called Garbage that is in your current directory, including all of its files and subdirectories. In this case, the -R option can be either upper- or lowercase:

```
rm -r Garbage
```

TIP *Sometimes you need to remove a file whose name starts with a hyphen. To do this, preceed the name with ./, which doesn't change the meaning of the file location, or use the full pathname.*

mkdir—Create a Directory

The mkdir command creates directories. This is the same as making a new folder with Konqueror:

```
mkdir [options] dirnames
```

Common Options

`--help`	Displays a help message.
`-m` *mode*	Sets directory permissions to *mode* masked by your umask value. Uses the symbolic values (e.g., `a=rwx`) for all permissions for everyone or octal numeric values (`777`). See the "How Permissions Work" section in this chapter.
`-p`	Creates any missing parent directories for each argument. Mode is set to umask modified by u+wx. Arguments corresponding to existing directories are ignored.
`-v`	Explains what is being done.

Examples

Make a directory named Harry in your current directory and a directory named Chest in /tmp:

```
mkdir Harry /tmp/Chest
```

Make a directory named Harry in your current directory and force the directory permissions to everything for user, read, and execute (search) for group, and nothing for others. (See the "How Permissions Work" section in this chapter.)

```
mkdir -m u=rwx,g=rx,o-rwx Harry
```

Repeat the preceding directions, only using a numeric mode specification:

```
mkdir -m 750 Harry
```

Create the directory Chest as a subdirectory of the directory Harry in your current directory. Also, create Harry if it doesn't exist:

```
mkdir -p Harry/Chest
```

rmdir—Remove a Directory

The rmdir command is the opposite of the mkdir command. It removes empty directories.

```
rmdir [options] directories
```

Common Options

`--help`	Displays a help message.
`-p`	Removes a directory and then tries to remove each component of the pathname.
`-v`	Explains what is being done.

Examples

Remove the directory named Harry in your current directory and a directory named Chest in /tmp:

```
rmdir Harry /tmp/Chest
```

Remove the directory Chest as a subdirectory of the directory Harry in your current directory. Then, if Harry is empty, delete it as well:

```
rmdir -p Harry/Chest
```

Text Editors

This section introduces the most common editors used with Linux on the command line, but an entire book could be written on each editor. In the case of vi and Emacs, that has already happened.

Don't confuse these text editors with word processors. These are programs designed to let you enter and edit text. Their text-formatting capabilities are limited to setting line-length limits. Traditionally, these editors were used in conjunction with text-processing programs, such as troff, to produce typeset documents.

Emacs

Emacs is a project of the Free Software Foundation. Many people see it as a work environment rather than only an editor. It includes its own built-in programming language, so you can customize its operation.

Joe

Less popular than Emacs or vi, Joe has a lot to offer the beginner. Besides its easy-to-use default mode with on-screen menu, it can emulate the basics of Emacs and vi.

vi

vi is the most popular, by far, of the available editors. There are many flavors of vi, including nvi, vim, vile, elvis, and stevie. All share what is called a moded system, where you enter text in one mode and edit in another. The users and developers of vi pride themselves on minimizing the number of keystrokes required to complete a task.

lpr—Send File to a Printer

The lpr command can be used to spool a file for printing. This is similar to the menu choices in GUI applications and the drag-and-drop capability in KDE.

```
lpr [options] [files]
```

If no *files* are specified, then lpr reads from standard input.

Common Options

`-m [user]`	Sends email to *user* (you as default) when the job has printed.
`-P ptr`	Sends the file to the printer named *ptr* instead of your default ($PRINTER).
`-r`	Removes the file after printing.

File Attributes and Permissions

Associated with each file and directory are two ownership fields and a set of permissions. Ownership and permissions are used to restrict access to files.

How Permissions Work

Each file has an owner, which is the user ID associated with the file. It also belongs to a group, which by default is set to the default group of the user who created the file. For example, your system might be set up so that everyone in a department is in the same group. To share a file with a member of another group, you could change the group owner to the owner of the other group.

There are a total of nine basic file permissions, divided into three groups. Those groups are associated with the file owner, the group the file belongs in, and everyone else.

Within those three groups, permission to read the file, write to the file, and use the file as an executable program are available. In addition, there are similar permissions associated with each directory.

Note that when you create a file, you are the owner. A normal user cannot change that ownership. If such a change is necessary, your system administrator must log on as the root user to make the change.

You can think of the permissions as a three-digit octal number. The leftmost digit represents the user's permissions. The next one represents the group's permissions, and the last digit represents the permissions of everyone else.

Within each digit, a 4 represents file-read permission, a 2 represents file-write permission, and a 1 represents file-execute permission or the ability to search for directories. It is only necessary to add up these values for each digit in order to set the permissions. For example, 754 represents read, write, and execute permission for the file owner (7=4+2+1), read and execute permission for the group (5=4+1), and only read permission for everyone else.

As an alternative, the permissions can be specified symbolically using a combination of letters (u, g, and o to represent user, group, and other, respectively, and r, w, and x to represent read, write, and execute permission, respectively). The letter a is used to represent the combination of u, g, and o.

These letters are combined using punctuation to specify the permissions. Using an equal sign (=) sets the permissions to the value specified, a minus sign (-) indicates permissions to be removed, and a plus sign (+) represents permissions to be added.

If more than one of the ownership/punctuation/permissions strings is required (for example, if you want to add group write and remove read for other), you combine the sets with a comma (,). If all this sounds harder than the numeric specification, it is. But it is there if you wish to use it.

To use the example 754 from the preceding numeric explanation, you could specify these same permissions as `u=rwx,g=rx,o=r`.

chmod—Change File Permissions

The chmod command is used to change file (and directory) permissions. Specify the desired permissions symbolically or numerically in octal:

```
chmod [options] mode files
```

Common Options

`-c`	Like -v but only reports if a change is made.
`-f`	Suppresses error messages
`--help`	Displays a help message.
`-R`	Changes file and directory permissions recursively.
`-v`	Explains what is being done.

Examples

Change the permissions on the file harry in your current directory to read and write for the owner, read for the group, and nothing for everyone else:

```
chmod 640 harry
```

Change the permissions on the directory Harry and anything it contains to read and execute for owner only:

```
chmod -R 500 Harry
```

TIP *There are three additional permission bits for each file and directory that may be of interest to the advanced user. Type* `man chmod` *for details.*

chgrp—Change File Group

The chgrp command is used to change the group to which a file (or directory) belongs:

```
chgrp [options] group files
```

Common Options

`-c`	Like -v but only reports if a change is made.
`--help`	Displays a help message.
`-R`	Changes file and directory permissions recursively.
`-v`	Explains what is being done.

Examples

Change the group on harry in your current directory to buenas:

```
chgrp buenas harry
```

Change the group ownership on the directory Harry and anything it contains to buenas:

```
chgrp -R buenas Harry
```

TIP *The chown command can change both the owner and group of a file, but only the superuser is permitted to change file ownership.*

Information

file—Guess File Type

Sometimes you need an educated guess as to what is contained in a file. The file command will do this for you. It uses a combination of innate intelligence and a *magic number file* to come up with a reasonable guess.

```
file files
```

Examples

Guess the type of the file harry in your current directory:

```
file harry
```

Guess the type of all files in /tmp:

```
file /tmp/*
```

man—Online Documentation

If you are looking for the description of a command or a command that works with something in particular, the man command can help.

Examples

To display documentation on a specific command, use:

```
man [section] command
```

To search for all commands related to a keyword, use:

```
man -k keyword
```

Examples

Display information about the more command, defaulting the section number:

```
man more
```

Display information about the more command, specifying the section number. Note that all user commands are in section 1, system calls in section 2, library functions in section 3, and so on. You will almost always want section 1:

```
man 1 more
```

List a synopsis of all commands related to the keyword permissions:

```
man -k permissions
```

Sorting and Searching

Linux comes with a whole host of utilities for finding things and manipulating data. What you see here is a very small subset. For serious data manipulation and reporting, look into awk, Perl, and Python.

Using pipes to string these commands together and connect them to pager commands, such as more, is very common. This is where the toolkit aspect of Linux comes into play.

grep—Searching for Strings in Files

The grep command is used to locate a data pattern within a file. It gets its strange name from the command you would need to type into the original UNIX editor in order to perform this task.

In its most basic form, grep only displays lines that match the specified character string. However, to take full advantage of grep you need to understand regular expressions, a special pattern-specification language.

```
grep [options] string files
```

Common Options

- `-c` Counts the number of matched lines rather than actually printing them out.
- `-i` Ignores the case of the letters in the match string.
- `-l` Prefixes each output line with the line number within the input file.
- `-v` Inverts match, only displaying lines that do not match.

Examples

Display all lines that contain the string `cool` in the file harry in your current directory:

```
grep cool harry
```

Display all the lines that do not contain the string `real cool` in the file harry in your current directory and page the output through more:

```
grep -v "real cool" harry | more
```

Display all the lines in a long list of the files in your current directory that contain the string `Oct` followed by a space. Page the output through more:

```
ls -l | grep "Oct " | more
```

find—Locating a File

The find command has a very complicated syntax and a lot of options but, because of this, is extremely powerful:

```
find [paths] expression
```

The *paths* portion tells find where to start searching. It then continues recursively under the specified starting point or points.

The *expression* part is a series of conditions used to specify the file you want to find, plus, if you wish, the action to be taken to each match. The default action is simply to print the name of the file. By default, all the specified conditions must be met. They are matched in order from left to right. For “or” rather than “and” conditions, use an `-o` connecting expression.

Common Options

`-name` *filename*	The *filename* must match. Note that the shell wildcard characters can be used in the match, but the match string must be quoted to prevent the shell, rather than find, from doing the expansion.
`-group` *group*	File belongs to group *group*.
`-name` *filename*	Like -iname but case-insensitive.
`-newer` *file*	Desired file was modified more recently than *file*.
`-v`	Explains what is being done.

Examples

Look for the file harry, starting in your current directory:

```
find . -name harry
```

Repeat the preceding example, but ignore the case and match any file whose name starts with harry:

```
find . -iname "harry*"
```

Look for any file whose name matches harry or chest in your current directory and the home directory of bill:

```
find . ~bill -name harry -o -name chest
```

locate—Locating a File

Although find will always find a file, it can take a long time. The locate command uses a database that is created regularly (usually daily) to try to find a matching file.

It is sometimes desirable to combine locate with grep to narrow down the match.

```
locate [options] pattern
```

Common Options

`-i` Ignores the case of the match.

`--help` Displays a help message.

Examples

Locate any file whose pathname contains the string `picture`:

```
locate picture
```

Locate any file whose pathname contains both `picture` and `Secret`:

```
locate picture | grep Secret
```

System-Related Commands

Here is just a brief introduction to some system-related commands. While you saw much of this functionality with the KDE-specific graphical commands, there may be an advantage to using the command line with some of these.

Think about what you have learned about grep, for example. You could use grep to select specific lines from the output of the ps command.

ps—Process Status

Each active program in Linux is called a process. When you enter a command, you start a process. If you enter multiple commands, connected by pipes, you are starting two or more processes.

In addition, the system has many daemons. These are programs that take care of the system and perform background processes for you. A good example is the print spooler—the program that monitors the print queues and interacts with the printer.

By default, ps shows only processes that are owned by you.

```
ps [options]
```

Common Options

- a Shows all processes associated with a terminal.
- T Shows all processes associated with this terminal.
- l Shows a long display format.
- f Shows the ASCII art process hierarchy.

TIP *There is an amazing array of options available to format and sort the output. See the man page for details.*

Examples

Display all your processes:

```
ps
```

Display all processes associated with a terminal in the long format:

```
ps al
```

Display a process hierarchy tree for all terminal-related processes.

```
ps af
```

top—Ongoing Process Status

If you want to see an ongoing display of current processes, top is the answer. Processes are sorted with the most active at the top. There are many options and interactive commands, but 99 percent of all runs are just top alone. Terminate top by pressing Q.

```
top
```

df—Display Free Disk Space

This command displays the location, mount point, size, amount used, and percentage used of all mounted filesystems of nonzero size. Many options exist, but the default is almost always what you need.

```
df
```

du—Display Disk Space Usage

This command estimates file usage recursing downward from the specified location or locations.

```
du [options] files
```

Common Options

-a	Writes counts for all files, not just directories.
-h	Prints sizes in human-readable format.
--help	Displays a help message.
-s	Prints summaries for each argument.

Examples

Show usage of all files under your current directory:

```
du
```

Show summary usage information for /usr/bin and /bin:

```
du -s /usr/bin /bin
```

Conclusion

This chapter has only scratched the surface as far as the capabilities of the command line. Hundreds of commands are available at the shell prompt and a lot more capabilities are within the command line of the shell. In addition, the shell offers a built-in programming language that makes it possible to make decisions and execute commands selectively.

Complete books have been written on shell programming. If you would like to know more, we suggest you take a trip to a good technical book store and find a book that is a good fit for you.

Also, SSC offers two reference cards, one on Bash and the other on the Korn shell as part of its *Linux Library* series. You can check them out, as well as other SSC books and reference cards at http://store.linuxjournal.com.