

Introduction to Linux

Linux is an Open-Source (i. e. 'Free') operating system which is virtually indistinguishable in its user-interface and operating characteristics from UNIX. Traditionally, Linux (and UNIX) have had command-line interfaces (like DOS) for humans to interact with. Recently GUIs (Xwindows) and GUI-based command 'shells' (Gnome and KDE) have been added to increase the acceptability of Linux to a wider audience.

Although the Linux kernel source code is free, many companies have created 'distributions' (distros) of Linux which make it easier for someone interested in Linux to get started. These distributions combine the kernel and necessary utilities, compiled, into a package which can easily be installed by anyone, regardless of their prior knowledge of Linux. As a rule they throw in a large number (hundreds, typically) of application and utility software packages as well. Of course, although they are not allowed to charge for the Linux kernel, they can charge for the package with its added value. Typically this is still a great bargain considering the relative ease-of-use and quantity of free software included.

World-wide, Linux is the operating system of choice for Web servers, and those who administer Linux servers generally use a 'command-line interface' in preference to a GUI. Thus we will spend most of this chapter on command-line operations. Typically, this means that you, the user, are faced with a blank screen containing only a 'command prompt'. The command prompt is provided by the current **shell** (see below), and usually looks like one of the following:

```
#_
```

```
$_
```

```
[root@localhost root]#_
```

```
[george@mymachine etc]$_
```

If the '#' symbol appears in the prompt that means that you are logged in as the system administrator, who is called '**root**' in Linux. If the '\$' (or some other symbol) appears in the prompt then you are logged in as an ordinary user. The system administrator, root, has complete access to the system whereas ordinary users have limited access to commands and resources of the system.

The prompts containing brackets appear in some distributions (and shells), and provide additional information to be added to the prompt line in addition to the prompt symbols,

or \$. The first word inside the brackets (root, george) is the name of the user who is currently active on the system. The word following the @ symbol is the name (**hostname**) of the machine you are logged into. The last word is the name of the '**current working directory**'; that is it specifies the default directory in which all commands will be applied if no other directory is explicitly specified in a command. The underscore shown following the final prompt symbol is usually blinking, and indicates where the next character typed by the user will appear on the screen.

It is important to note that commands typed on the command line are **case-sensitive**. Thus, the file names fileone, FileOne, and FILEONE refer to three different files, not the same one. Commands typed on the command line are not recognized unless they are typed in the correct case, usually lowercase. Thus, we say that Linux is '**case-sensitive**',

The Linux File System

File System Organization

In any Operating System, the File System provides a number of important functions to the user, including

- File and Path naming conventions
- Support for directory hierarchies
- Allocating and managing storage for files on hard drives, and
- Methods for accessing files from I/O devices and hard drives.

In Linux, *everything* is treated as a **file**, including directories and I/O devices, although the later (along with some other resources) are considered **special files**. Files are stored in a hierarchical (tree) structure, the top of which is the root of the file system, which is designated by the forward slash, /.²³ When Linux is first installed, the file system is installed with some standard directories already created. The use of these directories is standardized across most Linux distributions. Following are some of the directories that are typically created as 'children' of the root (/) directory when Linux is installed²⁴

root This is the home directory of root, the system administrator. This is the

²³ This *root* should not be confused with the *root user*, or administrator, of the machine into which we are logged. Also, note that this slash is different then the one used to identify the root of a Windows directory (folder) tree, which is the backslash, \

²⁴ This is not a complete list, but contains the most important ones for our purposes.

current working directory when root first logs in.

- etc** The etc directory, and its subdirectories contain all the **configuration** files of the Linux system and also of all application programs which have been installed on the machine. In addition it contains the **startup** scripts used each time the system is booted.
- home** The home directory contains the home directories of all **users** (except root) who are known to the system. Thus, if George is a known user on the system then under the directory *home* will be created another directory, called *george*, where the user, George, can store all of his files. When George first logs in, his current working directory will be his home directory: /home/george.
- bin** 'bin' is short for 'binary'. This directory contains files which are **executable** programs. There are other directories for binary files; bin contains those executables which are the most important in terms of using the system.
- var** 'var' is short for variable. This directory contains variable data; that is, data which may change during the course of running the system. For example, the source directories for web pages or ftp-accessible files are found in subdirectories of /var.
- mnt** This directory is a '**mount** point' for connecting removable or remote file systems. Floppy Disks and CD-ROMs are typically accessed as subdirectories under the /mnt directory as, say, /mnt/floppy and /mnt/cdrom. Mounting file systems will be discussed in detail later on.
- dev** This directory contains files which represent physical devices, such as printers, hard drives, ethernet cards, etc. which may be attached to the system. This directory contains many hundreds of file names, most of which are just placeholders for physical devices which do not necessarily exist.
- sbin** Contains executable files for use only by the system administrator.
- usr** Contains most of the data and binaries which are shared by various programs and users on the system.

File Specification

Whenever you want to use a command which accesses or references one or more files,

the filename must be specified properly: This **file specification** must identify its location as well as its name (filename). By location, we mean that the directory in which the file can be found must be included as part of the specification.

The location may be specified in an **absolute** fashion by listing all the directories which exist between the root of the file system (/) and the file itself. That is, we specify the **path** to be followed from the root to the file. This is called an **absolute path**. For example, suppose that user Mary has created a subdirectory called *worddocs* in her home directory, in which she has stored a file called *newtext*. Then the absolute path to this file is given as

```
/home/mary/worddocs/newtext
```

Notice that an absolute path always starts with a leading /, the root of the directory tree. When Mary was given an account on this system a home directory, *mary*, was created for her in the *home* directory of the tree.

It is not always necessary to use absolute paths when specifying files. It is usually easier (requires less typing) to specify a file location in terms of its **relative** position with respect to the **current working directory**. The current working directory is, by definition, the directory which is referenced when no path is given, the default directory. For instance, when a user, say Mary, first logs on to the system, they will automatically be 'in' their home directory. Mary will be in the directory */home/mary*. If Mary now creates a new file, say *xmaslist*, (e.g. with the command *vi xmaslist*²⁵) and fails to specify a location for it, then the file will be stored in her home directory. She could have forced it to be stored in her *worddocs* directory by specifying the absolute path of the file as */home/mary/* (e.g. *vi /home/mary/worddocs/xmaslist*). However, quite a bit of typing can be saved in both cases by using a **relative path** in the file specification. *We specify a relative path by dropping that part of the absolute path which specifies the current working directory.*

For example, in the case where Mary issues the command *vi xmaslist*, remember that her current working directory is */home/mary*. Thus, *vi xmaslist* is the same as *vi /home/mary/xmaslist*. Similarly, the command *vi /home/mary/worddocs/xmaslist* could be shortened to *vi worddocs/xmaslist*. Note that there is no leading / in a relative path. In the last example, it would be erroneous to type *vi /worddocs/xmaslist* as this would expect to find a directory called *worddocs* immediately below (as a daughter of) the root directory, /.

One special symbol can be used in the specification of relative paths; this is the 'double

²⁵ The *vi* command invokes a text editor. It will be discussed later in the course.

dot' symbol '..'. The double dot (or 'dotdot') is used to specify the 'parent' of the current working directory. Thus, if the current working directory is Mary's home directory, */home/mary*, then the directory *..* is */home*. Furthermore, the directory *../..* (dotdot slash dotdot) is the parent of the parent of the current directory, in this case, root (*/*). Let's assume that Mary wants to create her *xmaslist* file in John's home directory (*/home/john*). Then she could use the command *vi ../John/xmaslist* to accomplish this. Since *..* stands for */home* (the parent of Mary's current directory, her home directory) then *../john* is equivalent to the absolute path */home/john*.

Naming Conventions

A 'naming convention' specifies rules for naming files and directories. In Linux, the following rules apply

- File and directory names may be up to 256 characters in length
- File and directory names may contain numbers, alphabetic characters, periods, underscores and dashes, but no spaces.²⁶
- File and directory names are case sensitive.

Notice that there is no mention of file extensions in Linux. An extension is a short (usually three-letter) sequence of character following a period at the end of the file name. In windows, for instance, all documents created by MS Word have the extension *.doc* appended to the file, such as in *'mynotes.doc*. In Windows, all programs (i. e. executable files) have the extension *.exe* at the end of the filename (e. g. *explorer.exe*). No such convention is required in Linux, and it is usually not possible to distinguish a data file from a program file by just looking at the file name.

Wild Cards

It is sometimes useful to be able to specify, as part of some command, say, many files instead of just a single file. One way to accomplish this is with the use of **wild cards**. Wild cards are reserved characters (not allowed to be used as part of file names) which are used to specify multiple characters at once. There are two wild card characters: '*' and '?'. The * matches any set of characters. Thus the following file specification

*abc**

identifies all files which begin with the letters *abc* which exist in the current working directory. The following files would be included, if they existed:

²⁶ There are other punctuation marks allowed in file names, but there are also several that are not allowed. It is safest to limit the use of characters to those listed here. Technically, spaces are allowed but are strongly discouraged.

abc1 abcdefghijklmnop abcdef.345 abcABC.xyz

The file specification */home/mary/worddocs/a*z* calls for all of the files in Mary's worddocs subdirectory which start with the lowercase *a* and end with the lowercase *z*. Note that the file specification *** identifies **all** files in the specified directory. This can be either very useful or very dangerous, such as in the command *del ** which deletes everything in the specified directory!

The other wild card, *?*, is used to match just a single character rather than a set of characters. The file specification

abc?

identifies all files with filenames exactly four characters long, the first three of which are *abc*. This would include files such as

abc1 abcd abc9

if they existed in the specified directory but no files longer than 4 characters.

Special Files

As mentioned earlier, I/O devices are treated as files, and are called special files. All devices attached to a system are identified in the */dev* directory. For instance, the first hard drive on the system (which would be identified as *C:* in Windows) is identified as */dev/hda0* in Linux. The first three characters of the name identify the physical device, and the fourth character identifies the partition on the device. In Windows, a single physical drive with four partitions would assign the character *C:* to the first (primary) partition and the other three (logical) partitions would be given drive letters *D:*, *E:* and *F:*. In Linux those drives are specified as *hda0*, *hda1*, *hda2*, *hda3*, specifying partitions 0, 1, 2, and 3, all on physical device *hda*. A second physical drive would be called *hdb* and the first partition on that drive is *hdb0*. The most important devices for our purposes are

<i>hda</i>	The first hard drive
<i>fd0</i>	The first floppy disk drive
<i>cd0</i>	The first CD ROM drive
<i>eth0</i>	The first Ethernet card plugged in the system.

File Attributes

When a file is created in Linux space is allocated to it on the hard drive and a data structure is created which identifies various attributes of the file and where on the drive

the file can be found. The primary data structure is called an **index node (i-node)** which contains pointers to the physical locations on the disk drive. File names are kept in a directory which includes the i-node number of the i-node for this file. The i-node itself contains a number of attributes belonging to the file as well as the pointers to disk locations containing the file. These attributes include the user id (UID) of the owner of the file, the times of creation and last updated, and the access permissions, among others.

Access Permissions

The access permissions for a file identify who is allowed to access the file, and what they have permission to do with it. Each file belongs to a **user** (usually the person who created the file) and a **group** (which identifies a set of people in addition to the user who may access the file). The permissions specify what access the user has to the file, what access the group has to the file, and what access everyone else (called **other**) has to the file. For each of these three entities there are only three permissions: **read** (the entity has permission to read the file or browse the directory) **write** (the entity has permission to change or delete the file or, in the case of a directory, has permission to add or delete entries in the directory), and **execute** (run the program if it is a file, or make a directory your current directory). The permissions for a file are specified in a nine-bit field, `rwrxrwxrwx`. The first three bits are the permissions for the user, the second three bits are the permissions for the group, and the last three bits are the permissions for other (everyone else). When a permission is allowed, the appropriate letter appears in that bit position; when it is not allowed, an underscore appears there. Within each three bits, the permissions are fixed in the order shown: read first, write second, and execute third.

Consider the following set of permissions for some file:

```
rwxr_xr__
```

This specifies that the user can read, write and execute the file, the group can read and execute (but not write) the file, and everyone else can only read the file.

Links

```
tba
```

Linux Shells

A **shell**, (more accurately, a **command shell**) provides the user interface to the operating system. In old DOS systems, the shell was called `command.com` and provided the command prompt `C:>` for user input. In Windows systems the shell is

called explorer.exe and provides a Graphical User Interface (GUI) for user input. Linux has both command line and GUI shells available. The GUI shells working under Xwindows in linux and are called Gnome and KDE. There are many command line shells, the most popular, currently, being the **bourne again shell (bash)**. Other available shells include the older bourne shell (bsh), the C shell (csh) and the Korn shell (ksh).

Each user who logs onto a linux system can specify the default shell to be used when that user logs on, and can customize the shell to their own preferences.

The shell has a number of built-in commands that the user may use which are only available when that shell is active. Other commands are external to the shell and are available to all shells.

Shells interpret and run commands as typed by the user at the command prompt. In addition, the shell supports a number of **environment variables**, and also supports **shell programming**. Shell programs are **scripts** written using the shell's own programming language and may be used to create new commands or automate complex recurring procedures.

Many of the commands listed in the Command Summary below are bash internal commands, but not all. We need not be concerned with which are which.

Linux Commands

When working at the command line (or terminal) commands typed at the prompt all have a common syntax:

commandname [*options*]²⁷ [*arguments*]

The *commandname* is simply the command itself, and specifies what task is to be performed. For instance, the command **ls** (for 'list') is used to list the contents of a directory.

The *arguments* specify upon what object(s), if any, the specified action should be performed. In the case of **ls**, if no arguments are specified, the contents of the current working directory are displayed. On the other hand, the command **ls /home/john** specifies that the contents of john's home directory be displayed.

²⁷ The square brackets indicate parts of a command which may be optional; that is, not all commands take either arguments or options, or both. Those commands that may take arguments and/or options may not require them at all times.

The *options* (sometimes called *switches*) are used to modify the way the command behaves. **ls** without any options lists only non-hidden files, and lists only the file names. However, the command **ls -a** lists all files, including hidden and system files which are not shown by the **ls** command when it is used without the **-a** option. Note that options are usually preceded by a hyphen ('-'), sometimes by a double hyphen ('--')²⁸. In some cases, the option itself may take a parameter. Consider the following command:

```
ping -c 5 msmc.edu
```

The **ping** command is used to test a connection to a remote host on a network. The argument in this command is **msmc.edu** and the option **-c** stands for 'count', which is the number of times we want **msmc.edu** to be pinged²⁹. The parameter **5** is part of the **-c** option and specifies that we want to ping the remote host just 5 times.

Piping

In Linux, the piping mechanism allows several commands to be used together to create more complex operations. It is traditional in Linux (Unix) to only implement very simple commands in the operating system kernel or in the shell, and to give the user the ability to combine these 'simple' commands into complex operations.

Under normal circumstances, when a command executes it sends the results of its operation to the **standard output** device, usually the **display**. Similarly, a command may get its input data from the **standard input** device, usually the **keyboard**.

The pipe symbol '|', is used to create a pathway, or pipe, from the output of one command to the input of another; that is, output from one command, which normally goes to the standard output device, will be used as input to another command, which would normally get its input from the standard input device. For example, the command

```
ls | sort
```

takes the output of the *list* command, which is a list of the files in the current working directory, and sends them as input to the *sort* command, which will sort the list into alphanumerical order before displaying them on the standard output device. Here's another example:

²⁸ A single hyphen is used when the option itself is a single character. A double hyphen indicates that the option name contains multiple characters.

²⁹ Without the count, **ping** will send ping signals to **msmc.edu** continuously until interrupted by the user pressing **ctrl-C** at the keyboard.

```
ls | grep bad | sort | more
```

grep is a command that filters its input data, and only outputs lines which contain the specified string ('bad' in this case). More is a command (see below) which takes very long output (output which is larger than a single display screen can hold) and displays a screenful at a time, waiting for the user to hit a key on the keyboard before displaying the next screenful of data. Thus, the above command line expression creates a directory listing consisting only of those files containing the string 'bad', sorted into alphabetical order, and displayed, if the resulting list is very long, a screenful of data at a time on the output device..

Redirection

There are times when output that would normally go to the standard output device is more useful if it could be saved to a file for future study, or sent to some other location. Similarly, there are times when the input data for a command would more conveniently be supplied by the contents of a data file rather than received from the standard input device. The redirection operators supply this capability.

The output redirection character is the 'greater than' symbol, >. Using this symbol, the command

```
ls > directorylisting
```

would send the output of the ls command to the file called 'directorylisting' instead of displaying it on the monitor. If the file does not exist, it will be created. If it does exist, the contents will be erased and replaced by the new listing.

```
ls >> directorylisting
```

(note the double >) will append the new directory listing to the end of the current contents of the *directorylisting* file.

The input redirection symbol is the 'less than' symbol, <.

```
sort < directorylisting
```

uses the contents of the file *directorylisting* as input to the sort command and sends the sorted data to the standard output device (display). The input and output redirection symbols can be used in the same command expression, such as

```
sort < directorylisting > sortedlisting
```

which uses the contents of the file *directorylisting* as input to the sort command and sends the sorted data to the file *sortedlisting*.

Aborting Commands

Sometimes a command may take a very long time to complete execution. In fact, there are some commands which may never complete execution. For instance, the command

```
ping msmc.edu
```

will send ping commands to msmc without end. In other cases, non-terminating commands may be the result of user error, or programming bugs. In any case, a way is needed to terminate, or abort, such operations and programs. In Linux this is accomplished by holding down the control key and hitting the c key. This key combination is called ctrl-c, and is a standard way of aborting any operation or program.

Command Summary

Logging On and Off

When the system is initially booted or you open a new terminal window, no command is required for logging in; you simply type your userid at the **login:** prompt. Subsequently, however you may want to login with a different userid. You can use the **su** command for this.

su	With no arguments, you temporarily become logged in as root.
su <i>userid</i>	Temporarily become the user identified by userid. You will be prompted for userid's password.

su can take additional options and arguments which are beyond our needs at this time.

shutdown -h NOW The shutdown command with the -h parameter halts the system. The shutdown command takes a time parameter which gives you the option of delaying shutdown for a specified amount of time. Using NOW instead of a time causes the command to take affect immediately. Only root can execute the shutdown command.

shutdown -r NOW This command restarts (reboots) the system.

Adding Users

The following commands add an account to the Linux system for a new user. Only root

can execute these commands.

useradd *username* Creates a new account for user *username*. A new home directory, */home/username*, is created, and a new group, also called *username* by default, is created of which *username* is a member (the only member).

A new entry is created in **/etc/passwd** containing all account information, except the password. An entry is also made in **/etc/group**

passwd *username* Creates a password for *username*.

The password is encrypted and stored in **/etc/shadow**

Getting Help

Help is available with the online manual, which is accessed with the **man** command.

man *commandname*

man -k *keywords*

This will open up a document containing detailed syntax and descriptions of all arguments and options associated with the specified command. To exit the man page, type 'q'.

The k switch allows you to search for related information using keywords

info *commandname*

Similar to *man*, but may provide more historical and usage notes than are in the man pages.

help *commandname*

This provides help for commands which are built into the shell.

***commandname* -h**

***commandname* -help**

***commandname* -?**

Many commands provide their own help which is accessible with the indicated options.

find *file specification*

Use the *find* command to locate files matching the criteria in the file specification. Wild cards may be used.

Example: **find /etc/she*** lists all files & directories in the */etc* directory whose names start with the letters 'she'.

find file specification -d

find file specification -f The d switch limits the output to only those names which are directories; the f switch limits the output to only those names which are files.

slocate text string

Use *slocate* to find all files with filenames containing the given text string. This command uses an index database rather than searching the file system directly. See **updatedb**.

updatedb

Run this command (usually once a day) to create/update the index database used by **slocate**.

File System Commands

ls List the contents of the current working directory.
ls *directoryname* List the contents of the specified directory
ls -a List the contents, including all hidden and system files
ls -l List the contents, showing file attributes for each file

The -l option provides a listing which looks like the following:

```
_rwxr_xr__ 2    mary networkos  4523 Dec 15  12:05  filename
```

The first character is the file type. If it is blank (an underscore) than the file is an ordinary file (data or program). Other characters which may appear as the first character include

d	Indicates that the file is a directory
l	The file is a link
b	A block special file
c	A character special file
p	A named pipe

The next nine characters are the permissions as previously described. The remaining fields are

2	The number of links to this file.
mary	The name of the owner of this file.
networkos	The name of the group with access to this file
4523	The size of the file in bytes
Dec 15 12:05	The date and time when the file was last modified
filename	The name of the file

ls -al	Options may combined following a single hyphen, rather than typing ls -a -l
ls ~	List the contents of the user's home directory ³⁰ .
ls ..	List the contents of the parent directory of the current working directory.
cd [directory]	Make the specified directory the current working directory (change directory). If no directory is specified the new directory will be the users home directory.
cd ~	Make the user's home directory the current working directory
cd /	Make the directory root the current working directory
cd ..	Make the parent of the current working directory the new current working directory
mkdir directoryname	Create a new subdirectory as a child of the current working <i>directory</i>
mkdir /path/directoryname	Create a new subdirectory as a child of the specified path.
rmdir directoryname	Remove the specified directory. Note that the directory must be empty
rm directoryname	Remove the specified directory and all its contents and subdirectories
cp /path1/file1 /path2/	Makes a copy of file1 at the new location defined by path2. If path1 specifies the current working directory it is usually not specified. E. g. cp file1 /path2/
cp /path1/file1 /path2/file2	Makes a copy of file1 at the new location, and gives it a new name.
Multiple files can be copied by using wild cards in the file specifications. E. g.	
cp * /path/	copies all files in the current working directory to the specified directory.
mv /path1/file1 /path2/	Moves a file. This copies the file to the new locatin and deletes it from the original location
pwd	Print working directory; displays the name of the current working directory
chmod arguments filename	Change the permissions of <i>filename</i> as specified by the <i>arguments</i> .

³⁰ The tilde (~) can be used in many commands as a shorthand for the user's home directory.
nos text 2.wpd
NTC Feb 22, 2006

Recall that the permissions consist of 9 bits, three each for the *user*, the *group* and *other* for the file. The arguments for the `chmod` command take a couple of different formats.

In the first format, the arguments take the form of three decimal digits, each of which can take the values of 0-7. Consider for example the three bits corresponding to the user. If the user has the `r` bit on, we consider a binary 1 to be in that position; if the `r` bit is blank (`_`) then we consider there to be a zero bit in that position. Then all possible combinations of `r`, `w`, and `x` can be represented by the binary combinations 000, 001, thru 111, which are specified by their decimal equivalents 0, 1, thru 7. More explicitly, here are the decimal representations of all possible combinations of `r`, `w`, and `x`:

<u>Permissions</u>	<u>Binary</u>	<u>Decimal</u>
<code>___</code>	000	0
<code>__x</code>	001	1
<code>_w_</code>	010	2
<code>_wx</code>	011	3
<code>r__</code>	100	4
<code>r_x</code>	101	5
<code>rw_</code>	110	6
<code>rwx</code>	111	7

Thus the command **`chmod 700 filename`** says that the permissions for `filename` should be changed to `rwx_____`. Similarly, the command **`chmod 641 filename`** changes the permissions for the file to `rw_r__x`

The drawback with this form of the `chmod` command is that all permissions must be specified. It is not usable if you just want to change one permission, (say, give others read permission) and you don't know what the other permission are, or want a faster way to make the change without finding out what all the permissions are and calculating the appropriate decimal arguments. To solve this problem there is an alternative form for the arguments which allows specification of whose access should be changed and which bits should be affected. In this form of the command, the user's permissions are called **`u`**, the group's permissions **`g`**, and others **`o`**. *All* are specified with the letter **`a`**. We specify whether particular permissions are to be turned on or off using **`+`**, **`-`** and **`=`** signs. Thus,

`chmod u+w, g-rx filename`

specifies that the write bit should be turned on for users, and the read and execute bits should be turned off for the group.

`chmod a+r filename`

specifies that the read bit should be turned on for all: the user, group, and other.

chmod o=r filename

specifies that other should be given read permission but not write or execute permission.

Displaying Text**cat filename**

The cat (for catenate) can be used to display the contents of a file on the output display device.

```
cat myfile.doc
```

will display the contents of the file, myfile.doc on the output device.

cat

If no filename is specified data is taken from the standard input device, the keyboard. This form of the command echoes keyboard input to the screen. Redirection can be used to redirect this keyboard input to a file, and provides a quick and dirty way to create a file from keyboard input. The command is terminated by typing the 'end of file' character, ctrl-d on a new line.

echo 'text string'

The echo command causes the text string to be displayed on the standard output device, This may not seem to useful, but used in conjunction with the redirection operator, it provides a convenient way to put text into a file:

```
echo 'This is some text' > textfile
```

Note that the quote marks are required, especially if there are spaces in the string.

more

The *more* command is also used to display a file, but it only outputs a screenful of data to the display at a time. So, while cat is useful for very short files, longer files are generally displayed using more, as in

```
more mydata.doc31
```

The file is scrolled a page at a time by pressing the space bar, or a line at a time using the enter key.

³¹ Note that this is equivalent to **cat mydata.doc | more**.

The command is aborted by typing the letter q (for 'quit').

less

The *less* command

```
less mydata.doc
```

provides the same functionality as the *more* command, but with many more features. One important feature is the ability to scroll both backwards and forwards in the file using the page up and page down keys, while *more* only allows paging forward.