2) There would be more than one way. You can enclose the entire filename in single or double quotes, or you could escape the spaces with a \ immediately preceding the space character.

Check Your Progress 9

1) Try Yourself.

2) Use ls -l.

3) While you could certainly store information in the filename itself, that would not offer the same convenience as storing the information in the file. This is because you could not edit the filename conveniently, or find patterns in it or use any file manipulation commands on it.

4) Try Yourself

5) Try Yourself

Check Your Progress 10

1) Try Yourself.

2) Linux complains that there is no such file or directory and you remain wherever you were.

3) Yes, there is no restriction on this or on another directory of the same name under a directory. It is possible because the absolute pathnames of the two will not be the same.

Check Your Progress 11

1) Try Yourself

2) Try Yourself

2.6 FURTHER READINGS

There are a host of resources available for further reading on the subject of Red Hat Linux version 9.0.

1) http://www.redhat.com/docs/manuals/linux

2) http://www.linux.org gives among other information, a list of good books on Red Hat Linux.

3) Consider joining a good linux mailing list, e.g.

UNIT 3 LINUX UTILITIES AND EDITOR

Structure **Page Nos.**

3.0 INTRODUCTION

In this unit, you will start to delve deeper into Linux and learn some more useful commands. You will also see how to combine commands together to perform useful tasks for which there might not be any single command available. You will learn to write simple programs in the bash shell that will let you write your own Linux commands. You will also see how to use the graphical user interface of Linux to perform many tasks without issuing any commands on the command line. Finally, you will look at the editor, vi, available in Linux for you to edit text files.

Because of the large amount of material to be covered, we will have to be brief. However, we shall try to illustrate important concepts with realistic examples. You will then need to practice whatever you learn on a Linux computer so that you understand the variations and nuances of the commands.

3.1 OBJECTIVES

After studying this unit, you should be able to:

- use some simple and important Linux commands;
- understand the concept of standard input, standard output and standard error;
- be able to use filters and pipes to connect commands together;
- write simple shell scripts to produce your own commands;
- use the graphical user interface to perform tasks without needing the command line, and
- use the programmer's editor iv,

3.2 SOME USEFUL COMMANDS

In the last unit, you have seen how to use some basic Linux commands like ls, cd and pwd. We will now look at some more useful commands that are commonly required. Linux has a large number of commands some of which are useful for system administrators, some for software developers and so on. Here we will consider only general-purpose commands that any category of user needs. Also remember that we will look at only a few most commonly used options of the commands. For a full description you should refer to the documentation for the command.

File Manipulation Commands

cat

You have so far learnt how to see directory listings that tell you the contents of a directory in Linux. Usually there will be several files and perhaps some other directories listed. But how do you see the contents of a file? This is done with the cat command.

[kumarr@linux kumarr] `cat first_file`

prints out the contents of `first_file` on the screen.

Remember that Linux is not concerned with what kind of file `first_file` is. If it is an executable file or a file produced by using some editor that does formatting, then the output will most likely not be intelligible to a human reader. In any case, the file will probably be printed out so fast that you will not be able to see anything but the last screenful. So it is actually useful only if you want to see what is there in a small text file. There are other forms of the command that you will study a bit later in this unit.

For now, try the following command

[kumarr@linux kumarr] `cat first_file second_file`

You will find that the contents of both the files are printed on the screen one after the other without any kind of gap in between. Actually `cat` stands for concatenate, that is, join the files together. You can give any number of arguments to the cat command and it will print them out on the screen in the sequence you have specified. You will see later how to use this facility to advantage.

tail

Sometimes you might only be interested in something at the end of a text file. You could use `cat` and see what lies at the end, but if the file is a big one, you could be waiting for quite a while before the gobbledygook on the screen stops and you can make sense of what you see. For such a situation, you can use the `tail` command. It shows what is in the last part of a file.

[kumarr@linux kumarr] `tail first_file`

This prints out the last 10 lines of first_file. What if you need to see something that is in the 14th line from the end? You can say

[kumarr@linux kumarr] `tail -14 first_file`

You can use any other number instead of 14 that you need. So to see just the last three lines, say

[kumarr@linux kumarr] `tail -3 first_file`

Like `cat`, you can give multiple file names as arguments; the last part of each is printed out with a header showing the name of the file that follows. Some other useful options are `+n` to start printing from the nth line of the file instead of the beginning, unlike cat. You can also use `-cn` to print the last few bytes instead of lines, with n being the number of bytes you want to see.

cmp

This command lets you compare two files and determine whether their contents are the same or different. If they are the same, the command says nothing. Otherwise it prints out the first byte for both files where there is a difference between the two. There is also the -l option to print out all differences.

```
[kumarr@linux kumarr] cmp first_file second_file
```

Remember that if one file is a part of the other, they are still considered to be different files.

diff

This command compares two files line by line and reports the differences between them. Unlike the mechanical comparison of the cmp command, `diff` makes a much more intelligent comparison. It indicates the differences between the files in three ways a, d and c. These stand for lines which have been added, deleted and changed between the two files. The symbol "<"refers to the first file and ">" to the second file.

```
kumarr@linux kumarr] diff first_file second_file
```

There is also the -b option to diff that ignores white space, and the -B option that ignores blank lines.

wc

This command counts the number of characters, words and lines in a text file. You can give the command any number of arguments, whereupon it performs the count for each file and gives the total on the last line. It takes the -c option to report only characters, the -w option to report only words and the -l option to report only lines.

```
[kumarr@linux kumarr] wc first_file
```

You can also combine options together, such as `-cl` to report characters as well as lines.

sort

This is a useful command that lets you print out the contents of a file in sorted fashion. You can choose the delimiter that separates fields and the default is the space character. It produces output on the screen by default but you can place the output into a file using the `-f` option.

```
[kumarr@linux kumarr] sort unsorted_file
```

There are several options to sort on fields, to use numeric order, to choose the collating sequence and so on.

There are many other useful commands that manipulate text files. The `tr` command can be used to translate or change characters in a file. The `split` command breaks up large text files into a number of files with a fixed number of lines each. The cut command can be used to produce vertical sections of a text file. To get formatted output on the screen, rather than the plain dump of the file that `cat` gives, you can use the `pr` command.

☞ **Check Your Progress 1**

1) Try using the `cat` command on a directory. What do you see?

.....

.....

2) What happens if you try to `cat` a non-existent file?

.....

.....

3) The `cmp` command normally reports files as different if they differ in any way. Suppose you have two files that differ at the beginning but their last portions are the same. How can you ascertain using cmp that two files are the same after a certain offset?

.....

.....

4) Are you able to use `wc` on a binary file? Are the results meaningful?

.....

.....

3.3 PERMISSION MODES AND STANDARD FILES

We will now see what file permissions are and how to change the permission modes of files and directories. So far we have had occasion to look at various commands, many of which have to do with files of various types. The permissions of files can make a great deal of difference to the way the commands behave. What are these?

You have already seen in Unit 2 that the long form of the ls command, the one with the -l option, tells you about the permissions of the file.

[kumarr@linux kumarr] `ls -l`

```
total 32
-rw-rw-r--    1 kumarr   users     4 Oct 15 01:18 abc
-rw-rw-r--    1 kumarr   users     4 Oct 16 17:31 abcd
-rw-rw-r--    1 kumarr   users     5 Oct 15 01:17 abc\d
-rw-rw-r--    1 kumarr   users    5 Oct 15 01:17 abc d e f
drwxr-xr-x    2 kumarr    users    4096 Aug 21 20:33 _files
drwxrwxr-x    3 kumarr    users    4096 Oct  9 18:01 ignou
-rw-rw-r--    1 kumarr   users     27 Oct 11 22:51 xx
-rw-rw-r--    1 kumarr   users     75 Nov  2 22:13 xx.c
```

The first column of the first field has a "d" in it if the file is a directory. So `ignou` and `_files` are both directories. For ordinary files, the first column of the first field is a

hyphen. The next 9 columns specify the file permissions. The user community in Linux is divided into three categories – owner, group and others. The owner is the person who first creates the file. Several users at an installation can be made part of a user group. Such a facility is useful in keeping working on the same project or department categoriesed together. All group members form the second category of users. Finally, the rest of the community is lumped under others, which are users who are neither the owner nor part of the same group. In the example shown above, the owner of the files and directories is kumarr and he belongs to the group users. Other users might also belong to the same group, though it is quite possible for kumarr to be the only one who is part of the group.

Having studied this characterisation of Linux users, you can now begin to understand the permission modes. Every file has three possible modes of access – read, write and execute, represented in the directory listing by r, w and x respectively. A file to which you have read access can be read by you, which means you can look at its contents by using any method like the `cat` command. If you have write access to a file, you can alter its contents. Execute permission is relevant in the case of executable files like those that we have seen in `/bin`, or for shell scripts, that we will study later in this unit. You can run or execute a file only if you have execute permission on it. Execute permission does not mean anything in the case of text files, nor actually for any file that is not executable, except for directories where this permission has another meaning.

Now you know enough to understand the permission information. The nine columns are divided into three parts – owner, group and others – of three columns each. The permissions are specified in the order read, write and execute. If a permission is available, the corresponding letter is shown, while the absence of a permission is indicated by a hyphen. So `rwx` means the category concerned can read the file, can write to it or alter its contents and can execute the program which the file contains. Similarly the permission `r-x` means that the file can be read or executed but not altered or written to, because write permission is absent. `r--` means the file can only be read, and `--x` means it can only be executed. `---` means that there are no permissions available and the file cannot be accessed at all. However, you can still see an ordinary file like this listed in a directory listing taken in the usual way. So you see that the permission columns in the listing shown above mean that for directories the owner has read, write and execute permission, whereas other members of his group have read and execute permission but no write permission. Similarly others, that is, users who are not part of the owner's group, also have read and execute but no write permission. Thus if you want to have all the permissions on a file, while denying group members write permission and allowing others only execute permission, the permission modes should be `rwxr-xr-x`.

With this knowledge, you should again look at the Linux system files and study the owners and permission bits for each. You will find that all users have executed permission on the files containing system commands. This is obviously necessary otherwise you could not use those commands. If possible, ask your system administrator to remove your execute permission on the ls command for a short while and then try to see your directory listing. If you are at a large installation where this kind of thing might not be possible, you can either wait until you know more about Linux to be able to see the effects of the absence of permissions, or ask somebody to make a copy of a system command in your directory, remove execute permissions on it and try to run it from your directory. Here you must take care not to run the system version of that command.

So far we have talked only about ordinary files and what file permissions mean in their case. But directories are also files, as you have seen earlier. Do the permissions all have the same meaning where directories are concerned? If so, what is it like to execute a directory? Let us delve a bit into this and find out some answers.

First of all you must understand that directories are special kinds of files, which contain information about other files, the permission bits have somewhat different meanings than what a hasty guess would suggest. Before we even look at what read permission for a directory means, try running the cat command on a directory and see what output you get. Linux will report an error and tell you that you were trying to cat a directory.

```
[kumarr@linux kumarr] cat ignou
cat: ignou: Is a directory
```

Actually a directory contains information on the files it contains, such as their names. Knowing this, it should be easy to deduce what read permission for a directory could mean. If you can read a directory you can see what files are in it, which means you can do an ls on the directory. In the absence of read permission you will not be able to look at the directory listing for that directory.

What about write permission? If you have write permission on a file you can change its contents. In the same way, having write permission on a directory allows you to change its contents. What does that mean? Creating, renaming or removing files would mean altering the contents of that directory. So it follows that having write permission on a directory enables you to create, rename or delete files in that directory.

Beginners often find it a bit difficult to grasp this point though it is not really hard to understand. You should remember that having write permission on an ordinary file allows you to change the contents of the file but does not allow you to delete or rename the file. That is possible only if you have write permission on the directory containing the file. Later we will see how this could lead to a security lapse in some situations.

Lastly, coming to execute permission you would agree that there is no way you could execute a directory corresponding to the normal sense that the operation refers to for an ordinary file. For a directory this permission bit determines whether you can cd to the directory or can copy files from that directory (this only if you have read permission on the directory as well). This permission is often called search permission. To be able to cd to any directory you must have search permission on every component of the absolute pathname of the directory. If search permission is absent on any component, all files and directories on that component and below it become inaccessible.

Apart from these permissions there are some other permission modes that you will come across. We will take these up in the unit on system administration. For now it will be sufficient to know that some other permission modes exist so that you do not get taken aback if you find characters other than r, w or x in the permission modes of a directory listing.

Changing Permission Modes

We will now see how to change the permission modes of files and directories. The action of a command can differ greatly depending on permissions. For example, the `cat` command will usually type a file on the terminal but will refuse to do so if the user does not have permission to read the file.

How do we change the permission modes of a file? The command to do so is chmod. It can be used only by the user who created the file or by the superuser. The owner is the user who created the file first, by any means such as by using a text editor or by copying an existing file.

Note that having all permissions on a file does not amount to owning it. If you can read somebody else's file, it does not mean that you can prevent others from doing the same, but you can establish such protection for a file of your own. Also do not get confused between the original file and a copy you might have created, having only read access to the source file. You can change your copy in any fashion you wish, but you will not be able to alter the original.

There are two forms in which you can use the chmod command. Let us look at the absolute method first, as it is slightly easier to understand and use. In this the permission mode desired for the files is given to the command in an octal notation that we will explain shortly. The mode of the file then gets changed to what was asked irrespective of the permissions before the command was run. The form of the command is thus

```
[kumarr@linux kumarr] chmode mode file
```

where `filename` is a list of one or more files whose permission modes are to be set to mode. If the permission bits are mode to start with there is no effect on the file or files after running the chmod command.

You know that file permissions are specified by 9 columns, for example `rwxr-xr-x` or `rw-r--r--`. In the absolute method the presence of a permission is indicated by a 1 and its absence by a 0. The resulting 9 bit binary number is then converted to octal. This octal number is what has to be specified as the mode for the chmod command.

You know that a binary number can be easily converted to octal by making groups of 3 bits starting from the right. Now convert each group into octal as if it were a single number. The resulting string of octal digits is the number in octal. Thus

```
rwxr-xr-x
```

can be written in binary as `111101101` after replacing each permission by a 1 and each hyphen by a 0. This binary number can be written as `111 101 101` after grouping the bits in threes. The octal form of the number is thus 751. So to convert a file to this mode say

```
[kumarr@linux kumarr] chmod 755 progfile
```

This will give `progfile` the permissions `rwxr-xr-x`. Likewise `rw-r--r--` in octal is 644. So you can provide these permissions to your file `motd` by saying

```
[kumarr@linux kumarr] chmod 644 motd
```

Instead of one file you can set the permissions on several files at the same time (all to the same value) by listing the files after the mode. So

```
[kumarr@linux kumarr] chmod 600 motd passwd
```

will make their permissions `rw-------`. Thus you can read these files or change them whereas nobody else (except root) can even read them. So

```
[kumarr@linux kumarr] chmod 0 passwd
```

will mean nobody has any permissions on the file passwd and even you will not be able to read a file of your own with such a set of permissions. However, you can change the permissions any time since you own your file. Also, the super user can change the permissions of any file.

Let us now look at the symbolic method of telling `chmod` the mode. Here the permission types are, as always, `r`, `w` and `x`. In addition, there is a set of characters which specify the target of the actions. The target can be `u` (users), `g` (group), `o` (others) or `a` (all of these). The actions are + to add a permission, = to set it absolutely and – to remove a permission. So you can say (there must be no spaces in the mode argument)

```
[kumarr@linux kumarr] chmod a+x progfile
```

to allow everybody to run progfile, irrespective of the earlier execute permissions on it. However, this is different from saying

```
[kumarr@linux kumarr] chmod 111 progfile
```

because this would remove read or write permission for everybody, whereas in the earlier case, those permissions would have been left untouched. If the owner had read, write and execute permission, he would retain it. If the group earlier had read and execute permission it would continue to have that privilege. If others had no permission, they would acquire execute permission. One can remove read and write permissions for others by saying

```
[kumarr@linux kumarr] chmod o-rw progfile
```

One can specify absolute permissions by saying

```
[kumarr@linux kumarr] chmod u=rwx,g=rx,o=x progfile
```

Here the different target categories are given different permissions on `progfile` by separating them with commas. No spaces should be present in the argument, otherwise only the first part will be taken as the desired mode. The portion after the space will be treated as a filename, which has to be assigned those permissions.

You might find the numerical way easier to use. However, if you do not want to alter some permission bits, there is no straightforward alternative to using the symbolic mode. For example, if you want to deny others any permission on a file but do not want to alter your own or your group's permissions, you can say

```
[kumarr@linux kumarr] chmod o-rwx progfile
```

But to achieve the same result using the absolute method you would have to first determine the existing permissions. Suppose the value is 644. You will now have to say

```
[kumarr@linux kumarr] chmod 640 progfile
```

If the initial permissions were 666, you need to say

```
[kumarr@linux kumarr] chmod 660 progfile
```

instead. If using the symbolic method, you would not need to worry. The command you need to give remains the same in both cases since the o action leaves the u and g permissions intact.

☞ Check Your Progress 2

1) Try executing a directory on which you do not have search permission, and another on which you do have it. What happens?

.....

.....

1) Can you look at the listing of a directory if you do not have search permission on it? Why?

.....

.....

3) Can you remove files form a directory if you lack permission on them?

.....

.....

3.4 PIPES, FILTERS AND REDIRECTION

By now you have seen quite a few Linux commands, and you must have observed that many commands produce or can produce output on the terminal screen. Likewise many commands can take input from the keyboard. Actually, these commands have been written to accept input from a standard input file and to produce output in a standard output file. Usually these files are set to the keyboard and the terminal screen respectively. Let us look at this in somewhat more detail by studying some examples.

Standard Output

If you make a list of commands you have learnt so far you will find that many of them produce some output. For instance let us say

```
[kumarr@linux kumarr] cal
```

which prints the calendar for the current month and year on the screen. In practice there are very few commands designed to produce output on the screen specifically. The programs are written to produce output on what is called the standard output, and Linux sets the standard output to be the screen by default. That is how the output happens to appear on the terminal.

The shell, which interprets all your commands and passes them onto the Linux kernel for execution, has a facility to alter the standard output. In other words, you can define a file, rather than the screen, to be your standard output. (Actually the terminal screen is also a file as far as Linux is concerned.) To do this you need to say

```
[kumarr@linux kumarr] cal > calfile
```

There can be zero or more spaces before and after the sign. This sign indicates that the standard output of the command preceding it should go to the file specified to its right rather than to the terminal screen. This is called redirecting the standard output. In the current case the calendar for the current month will be placed in the file calfile. You can verify this by

```
[kumarr@linux kumarr] cat calfile
```

although you could have redirected this output as well.

```
[kumarr@linux kumarr] cat calfile > catfile
```

Is that not a way of copying calfile to catfile? Note that the file to which output gets redirected gets overwritten if it already exists. You can verify this easily by

```
[kumarr@linux kumarr] ls -l > calfile
```

and now examining the contents of `calfile`.

There is another operator, which appends output to the file specified rather than overwriting it. This is achieved by

```
[kumarr@linux kumarr] cal 06 1994 > > calfile
```

Now calfile will contain the calendars for the current month as well as for June 1994. Compare this with

```
[kumarr@linux kumarr] cal 06 1994 > calfile
```

which leaves only the calendar for June 1994 in `calfile`.

Thus the >> sign is safer to use because it never destroys any data, but this operation will keep adding to the file, and it can sometimes be difficult to make out what part of the output was produced by your last command and which portion is the outcome of previous redirections or was simply the original content of the file.

Standard Input

Just as many commands produce output on the screen, some commands take input from the keyboard although most take input from files. Look at an aspect of the cat command you have not studied so far.

```
[kumarr@linux kumarr] cat
```

The result of this command is deafening silence. The uninitiated might wait several minutes before aborting the command, thinking there is something wrong because the system does not appear to be doing anything at all. The truth is that `cat` can take its input both from the standard input as well as from a file. However, the output is always produced on the standard output. If any filenames are specified they are used as the input but if none is mentioned the input is taken from the standard input. There are also some commands that take input only from the standard input.

In the present case no filename has been specified and `cat` is waiting for input from the standard input, the keyboard here. So if you type something cat writes it out to the standard output and the effect is that of echoing your input.

```
A foolish consistency is the hobgoblin of little minds – Emerson
A foolish consistency is the hobgoblin of little minds – Emerson
```

If you want to put an end to your misery you can terminate your input file by saying ^d, thereby causing cat to finish and present you with your prompt.

To redirect standard input, say

```
[kumarr@linux kumarr] cat < catfilesrc
```

whereupon cat will print the contents on the screen. This is just the same as

```
[kumarr@linux kumarr] cat catfilesrc
```

because cat can take its input from a file as well. So to copy this file to `catfiletarget`, you can say

```
[kumarr@linux kumarr] cat < catfilesrc > catfiletarget
```

or

```
[kumarr@linux kumarr] cat catfilesrc > catfiletarget
```

Thus you can redirect both standard input and standard output in the same command. Some commands do not take input from the standard input. In such cases redirection of the input is not possible, as with the `ls` or `who` commands.

Remember that redirection is a facility provided by the shell, not by the command. The command being run does not know or care what its standard input and output are connected to, and it continues to use them. So the command has to be designed to take input from the standard input if redirection of input is to be possible. Thus you cannot say

```
[kumarr@linux kumarr] cp < cpsrcfile
```

because cp does not take its input from the standard input. Similarly, output redirection is not possible unless the command is designed to write to its standard output.

Standard Error

So far we have seen the effect of redirecting the output of some commands that completed successfully. Let us look at this a bit more closely. For example, if there is no command like gah, say

```
[kumarr@linux kumarr] gah > gahfile
```

If you do so you will find that you get a protest message from Linux on the terminal but that gahfile is empty. Similarly

```
[kumarr@linux kumarr] ls -l gah > lsfile
```

produces a message on the terminal but nothing in lsfile. Why does the redirection fail? After all the command did produce output.

The reason is that there is a third standard file in Linux, called the standard error. Linux utilities and programs are usually designed to provide error messages in case there is something wrong and the program is not able to proceed as expected. Such messages are often referred to as diagnostic output because they can help the user diagnose the reason for failure. This kind of output is usually written to the standard error file. Usually the standard error is also connected to the terminal by default, but like the standard input or output, the standard error can also be redirected. To do this in the bash shell, say

```
[kumarr@linux kumarr] gah 2> gahfile
```

This will place the standard error in gahfile. How does one place both the standard output and standard error in the same file? For this, say

```
[kumarr@linux kumarr] ls -l gahfile yy > lsfile 2>&1
```

To redirect the standard error and standard output to different files, say

```
[kumarr@linux kumarr] ls -l gahfile yy > lsfile
2>lserrfile
```

Filters

A filter is a command which can take its input from the standard input and can produce output on the standard output. Having the capability to read from or write to files is not a disqualification. So ls is not a filter because it does not read from the standard input but cat is one because it can do so (although it can read from a file as well) and also writes to the standard output.

You can think of a filter as a "device" placed between the standard input and the standard output which filters the standard input before placing it on the standard output. In the case of cat there is no filtering action at all, but a command like `grep` does perform some weeding action on its output.

The standard output of a command can serve as the standard input of another. Several commands can be chained together like this. Such an arrangement is called a pipeline. Pipelines are one of the big strengths of Linux, because they often enable us to group several existing commands quickly to perform a task for which there is no command directly available.

A major design goal of Linux was to have an operating system which allowed easy sharing of data and programs, and allowed people to build on the work of others instead of having to do things from scratch. The facility of pipelining helps meet this goal because you can piece together commands written by different people to achieve your objective rather than wasting your time on doing things which have already been done. Let us take a simple example.

Suppose you want to find out how many of the files in a directory are directories rather than ordinary files. It would have been wonderful if there had been an option to ls which did this job, but since that is not the case we will have to try something else. One-way is to look at the directory listing with ls -p and count lines, which end in /. Such a visual method is tedious and prone to error, especially if there are many files in the directory. So let us try to make Linux do this for us. How about the following?

```
[kumarr@linux kumarr] ls -1 -p > tmp

[kumarr@linux kumarr] grep -c '/$' tmp
```

We first get the listing in a temporary file tmp and then count the number of occurrences of / at the end of a line in tmp using the `grep` command. The result will be available on the standard output. While this method will work it has a few disadvantages. One is that it is slow because an intermediate file has to be created. Secondly we cannot start the `grep` command before the `ls` finishes. Also if we run many commands like this we will be left with temporary files, which we will have to meticulously delete lest they clutter up our directory listing and otherwise waste disk space. So we can use a pipeline like this

```
[kumarr@linux kumarr] ls -1 -p | grep -c '/$'
```

The '|' symbol is the pipe character. It means that the standard outpu of ls -p is passed to grep. The act of connecting the standard output of a command to the standard input of another is also referred to as piping the output of the first command to the second. Here no temporary files need to be created or cleaned up by the user as Linux itself takes care of the details. Also the speed improves because the subsequent commands can start as soon as some data is available to them.

A command like ls which does not take its input from the standard input can only be the first command in a pipeline. Similarly a command which does not write to the standard output can only be the last command in a pipeline. Also, it is the user's responsibility to see that each command receives input in a form which it can meaningfully transform, otherwise the results will be gibberish. Thus do not pass data files other than text files to grep because grep works only with text files, with lines delimited by the newline character.

☞ Check Your Progress 3

1) Which of the commands you have learnt so far are filters?

.....

2) How will you count the number of all files in a directory?

.....

3) Look up the fee command. How do you think it can be useful?

.....

.....

3.5 SHELL SCRIPTS

The shell in Linux is a wonderful entity that serves us in various ways. It is started up automatically every time you login to the system. The shell sets up your environment when you start off on the machine. It is the shell that lets you run different commands without having to type the full pathname to them, even when they do not exist in the current directory. The shell expands wildcard characters, thus saving you laborious typing. It gives you the ability to run previously run commands without having to run the full command again. It is the shell that does input, output and error redirection.

You can use the shell as a programming language. It has all the usual language constructs like sequencing, looping, decisions, variables, functions and parameters. Here we will take a very brief look at bash, the shell commonly used in Linux. A shell itself is a program and there can be many different shells available. Linux also has a shell called tcsh that has a C like syntax and is an enhanced version of the Unix C shell. Bash is the Bourne again shell and is compatible with the Unix Bourne shell. To discuss the capabilities and features of bash to some extent would require an entire block in itself. What we will try to do here is introduce some basic features and refer you to the documentation or to other books on the subject for more detail. To become comfortable with shell programming, you will need to practice a lot, just as you would need to do for any other programming language.

After writing some shell programs you will realise that some Linux commands like sed that earlier seemed to you to be of limited utility are actually very useful. Shell programs are often called shell scripts.

A Linux machine can be thought of as being composed of several layers. At the lowest layer is the hardware which does all the physical tasks and without which there would be no computer and no Linux. Above that is the Linux kernel, which is the core of the operating system and does memory management, device handling and all the other mundane tasks needed to make the hardware easily usable by us. The Linux commands and utilities come next. At the top is the shell which can be considered to be the outermost layer and which enables us to run the utilities and other Linux commands. You can also construct higher application layers of your own which run above the shell. However, despite the layering that we have talked of, the shell is itself a program like any other.

Wild Cards

Wild cards characters are characters which can stand for characters other than themselves, somewhat like a joker in a pack of cards (though, unlike wild card characters, a joker has no intrinsic meaning by itself). A judicious use of wild card characters can make many commands easy to issue by saving a lot of typing and preliminary research. Suppose you have been writing a series of programs for enciphering text. You have been calling them cph01.C, cph02.C, cph03.C and so on. You suddenly realize that you have been doing this in the directory ~khanz/crypt, while actually these programs are for a particular project and you would like them in the directory ~khanz/crypt/knapsack. All you have to do to rectify the situation is to move your programs to the correct directory after creating it. So you can start off

```
[kumarr@linux kumarr] cd ~khanz/crypt
[kumarr@linux kumarr] mkdir knapsack
[kumarr@linux kumarr] mv cph01.C knapsack
[kumarr@linux kumarr] mv cph02.C knapsack
[kumarr@linux kumarr] mv cph03.C knapsack
```

Soon you get sick of typing almost the same thing again and again, even if you just use the up arrow key and keep changing the required characters in the filename. Moreover, when do you stop? If you have used a naming convention whereby the filenames have numbers from 01 onwards, you could stop as soon as the mv command reports that the file does not exist. But this method is hardly a rigorous one. What if there are gaps in the sequence, and cph08.C does not exist but you have other programs going up to chp39.C? So you would have to keep looking at the directory listing first, and also from time to time in the process, just to be sure. With many such files, the method is tedious and prone to error.

There is much less effort if you use wildcards. After making the subdirectory, just say

```
[kumarr@linux kumarr] mv cph??.C knapsack
```

The ? is a wildcard character that can stand for any single character including itself. So `cph??.C` expands to `cph` followed by any two characters and then by .C. Also remember that ? stands for exactly one character. Therefore a filename like `cph1.C` will not be matched by the command given above. To match that filename, you would need to say `cph?.C`. So the command given will leave any files from `cph1.C` to `cph9.C` in the original directory.

What do we do if our naming convention for files starts the filenames with cph but allows any characters after that, with of course .C at the end? Does it mean we have to give several mv commands with one, two, three, and many more ? characters? That would be quite tedious again. Also, after how many ? characters could we stop, knowing that there is no specific limit on the number of characters in a filename? The answer to that is another wildcard character, the *. It can expand to any number of any other characters. So all you have to do is to issue the command

```
[kumarr@linux kumarr] mv cph*.C knapsack
```

However, the * does not expand filenames starting with a leading . character. Also, in bash, a filename like `yy*.c` is not expanded unless there is at least one filename around to which it expands. So if you do not have a filename starting with `yy` and you issue a command like

```
[kumarr@linux kumarr] vi yy*.c
```

what the vi editor will create is a file called `yy*.c`. If a file like `yy1.c` already exists, then you can create `yy*.c` by escaping the * with a backslash, so that it is taken literally.

```
[kumarr@linux kumarr] vi yy\*.c
```

You can create a file called `yy?.c` in the same way, though it might be a good idea to avoid such characters in filenames to prevent confusion.

Simple Shell Programs

Let us now look at a simple shell program. Suppose you are kumarr and are working on your cryptography project in a directory `~/prj/crypt/pkc/src`, where your source files are located. But sometimes you are looking at documentation in another location, or are also working on some other project. To look at the files in this directory you have to issue a longish command, which can be cumbersome if you have to do it often. What you can do instead is to create your own shell program, say in a file called `wd`. This you can do using any editor. The file `wd` should contain

```
ls ~/prj/crypt/pkc/src
```

Now you need to make `wd` executable by setting its permissions to 755. Otherwise you would need to invoke it by saying

```
[kumarr@linux kumarr] bash wd
```

But if you just said `wd`, it would not work because when bash is given some command to execute, it looks for the commands in different directories in a fixed sequence. Typically this sequence is /usr/local/bin, followed by /bin and /usr/bin, but this can be controlled in a way we will describe shortly. Since `wd` is not in any of those directories, Linux complains. So you can say `~/wd` to take care of this problem.

What we have now done is created a command of our own that anybody can use. Currenly it allows others to look at our directory, something we might not really care to do, but it nevertheless illustrates the point. And we have created the command by

taking an already available command and using it in a different way. This is at the core of the Linux philosophy of building on the work of others.

Many installations have locally useful commands available in /usr/local/bin. You could also be contributing to this repository of useful commands by and by, and you should first explore to see whether the task you want to do is not already accomplished by using these commands. However, before you can release shell scripts for general use, you will need to make them robust, which is something we have not looked at in the current case. If you have some commands that you feel you need but are not fit for general release, you can place them in your own bin directory like ~/bin.

Suppose the system default form of the `ls` command at your installation is not what you find useful; you would like it to be `ls -l`. One way is to create a new command like ls that contains the line `/bin/ls -l`. Note that if you put only `ls -l` in your private command, it might keep calling itself indefinitely and so will not work. Also your program will not be able to use arguments because we have not given that ability.

Variables

Variables can be defined and used in bash like in any other programming language. For example, to set the value of a variable vehicle to "bus", you can say

```
[kumarr@linux kumarr] export vehicle=bus
```

To see the value of a variable you can say

```
[kumarr@linux kumarr] echo $vehic

bus
```

If the variable has not been defined, nothing is printed. You can also print several variables together or print literals using the echo command.

```
[kumarr@linux kumarr] echo $vehicle and car

bus and car
```

To set the value of a variable to the output of a command, give the command in backquotes.

```
[kumarr@linux kumarr] export mydir='pwd'

[kumarr@linux kumarr] echo $mydir

/home/kumarr
```

To let your command take arguments, you can refer to them as $1, $2, $3 and so on up to $9. $0 stands for the command itself and $10 would be interpreted as $1 followed by 0. If you want to use all arguments then say $*. Similarly, $# stands for the number of arguments. So if you write a command ech that echoes only the first and fifth arguments, it would have

```
[kumarr@linux kumarr] cat ech

/bin/echo $1 $5
```

Now you would find the other arguments would be ignored. The shell has several inbuilt variables of its own that you can look at by using the env command, to show the environment. You would be able to recognize several of them such as HISTSIZE for the number of commands that are kept in the history buffer, LOGNAME for your login name, SHELL for the shell you are using, HOME for your home directory and so on.

f to see if it is an ordinary file. In the

The shell scripts we have seen so far have been nothing but a sequence of commands that we could have anyway issued at the prompt itself. Shell programming would not have been of much use in that case. What makes it powerful are the programming constructs available such as loops and decisions. Let us first look at a program mkupper that converts the contents of its arguments to upper case.

```
[kumarr@linux kumarr] cat mkupper
for i in $1 $2 $3
do
    tr '[a-z]' '[A-Z]' < $i > $i.up
done
```

This converts the contents of up to three files to upper case and places them in a file of the same name with a .up suffix.

Besides the for loop, you can use the while or until loops with their usual meanings. For example, the script above could be written

```
[kumarr@linux kumarr] cat mkupper
while test $# -gt 0
do
tr '[a-z]' '[A-Z]' < $1 > $1.up
shift
done
```

Here suppose there are 25 arguments to the mkupper command. The condition in the while loop tests whether there is an argument available. If so, the contents of the file are converted to upper case. After each argument has been dealt with, it is discarded and the next argument becomes the first argument. This is done by the `shift` command in the `mkupper` file. Finally when there are no arguments left, the test fails and the loop terminates. You can achieve a similar effect using the `until` loop, given below

```
if test $# -eq 0
    then echo 'No files to translate'
    exit
else
    until test $# -eq 0
    do
        tr '[a-z]' '[A-Z]' < $1 > $1.up
        shift
    done
fi
```

The test operation can be used to check various conditions, such as if a variable equals a number. You can also use the other relational operators with the keywords `le, gt, lt, ge` and `ne`. Other tests that can be performed are `-w or -r` for checking if the filename is writeable or readable, `-d` to check if it is a directory and -

f to see if it is an ordinary file. In the script above we print an error message and exit if there are no arguments to the file. We could also have chosen to exit silently in such a case, as we did with the `while` loop example. Or we could wait for input from the standard input. The example also shows how to make a decision using an `if` statement. The statement is terminated with the `fi` keyword. The `else` part is optional. We can also use the `exit` keyword to break out of the shell script. Several else parts can be present in an if statement, they are then introduced with `elif`. Let us take an example script called countargs that counts the number of arguments and prints the result.

```
if test $# -eq 0
    then echo "No arguments"
elif test $# -eq 1
    then echo "Only one argument"
elif test $# -eq 2
    then echo "Two arguments"
else
    echo "Many arguments"
fi
```

Within a loop you can use the `break` and `continue` statements to exit the loop or to go back to the beginning of the loop respectively. Comments are introduced by the # character. Anything after this on a line is treated as a comment. In a shell script, like in any program, it is important and recommended practice to provide comments that explain the working of the program.

You can also use the case statement in place of multiple `if...elif` statements when it is the same condition that has to be checked each time. For example, let us write a shell script that checks the first argument to decide the operation to be performed and then performs the operation on the second argument. The case is terminated with an `esac` statement.

```
if test $# -ne 2
then echo "Usage: $0 operation files"
exit
fi
case $1 in
upper) tr '[a-z]' '[A-Z]' < $2 > $2.up;;
lower) tr '[A-Z]' '[a-z]' < $2 > $2.lw;;
*) echo "Invalid operation specified";;
esac
```

You must bear in mind that the options in the case statement have to be specified in the right order. If we give the *) option first, then it would match every case and the script would do nothing.

You can also pass input from the user to a shell script. For this use the `read` command. Take the following simple script that prints out the directory listing as desired by the user. It has a friendlier interface than the Linux command ls.

```
echo "1 for long listing"
echo "2 for stream list"
echo "3 for single column list"
read x
case $x in
1) ls -l $*;;
2) ls -m $*;;
3) ls -1 $*;;
*) echo "Invalid choice"
esac
```

In the read statement you can assign values to several variables. The first value goes to the first variable, the second to the second variable and so on. If there are more values provided by the user than there are variables, the extra values go to the last variable. If there are few values, the remaining variables do not get any values. Values are delimited by spaces and a newline character causes the assignment to happen. So if you say

```
[kumarr@linux kumarr] read x y z
a b c d
[kumarr@linux kumarr] echo $x
a
[kumarr@linux kumarr] echo $y
b
[kumarr@linux kumarr] echo $z
c d
```

You can do simple arithmetic in the shell with the expr command. The operators and operands have to be delimited by spaces. When you use the * for multiplication, you have to escape it with a \, lest the * be expanded to all available filenames.

```
[kumarr@linux kumarr] expr 2 + 3
5
[kumarr@linux kumarr] expr 18 / 3
6
[kumarr@linux kumarr] expr 4 \* 5
20
[kumarr@linux kumarr] expr 7 - 5
2
```

To help debug shell scripts you can use the -v or -x options that give verbose output. The -x option precedes each command with a + sign. Thus

```
[kumarr@linux kumarr] bash goodls -v
```

will print each command as it is executed.

Check Your Progress 4

1) Try the command?
   écho*
   Write files does it neglect to furnish you with? How can you get all filenames?

   .....

2) Write a shell script that prints out the contents of some fixed file in upper case.

   .....

3) Write a shell script that prints out a list of every unique word contained in the file in alphabetical order?

   .....

3.6 GRAPHICAL USER INTERFACE

Unlike the early versions of Unix, Linux has a good graphical user interface (GUI). This makes it different from the cryptic command line interface that proved daunting to most lay users who ventured to try Unix. Most of the common operations can be performed graphically and it is not necessary to use the command line. However, the power that the command line gives is still available to power users.

When you login you are presented with a text based screen, but thereafter you can have the system set up so that you reach the graphical user interface mode. This can be done by issuing the command

[kumarr@linux kumarr] `startx`

whereupon the X-window system is started up and you reach GUI mode. You have a default of 4 desktops that are available to you. Each of these can be arranged differently as you wish. To start up a terminal session, just right click the mouse on the desktop and select the "New Terminal" option. You will get a terminal window where you can use the mouse as well for editing, such as copy and paste. These options are available by right clicking the mouse or by choosing the Edit option on the menu bar in the terminal window.

The bottom tray contains some useful icons that will depend on what software packages have been installed. The leftmost icon is the red hat, the logo of the version of Linux that we are studying here. It has a small arrow to its right, clicking which brings up a list of options such as Accessories, Games, Graphics, Internet, Office and so on. Many of these options have further suboptions that you can select with your mouse.

The tray typically would contain icons for invoking the browser, email, the Open Office Writer, Presentation software and the spreadsheet, a printer manager and the icons for the desktops. To invoke any application, simply click its icon once in the bottom tray. This saves you having to know the name of the program for each application. When you move your mouse over any icon, a small explanation of the icon appears in a yellow box. It tells you the name of the program and what it does. This mouseover makes it easy for you to identify the correct icon for the program you want to run so that you do not have to rely on your memory to locate it.

If you right click on any of these icons, you get a menu where you have options to look at its properties or obtain help on the application. You can also remove the icon from the tray, but then if you want to run the application you will have to locate it in the directory structure. You can also move the icon around in the tray.

At the very left you have an icon in the shape of a red hat with an arrow pointing upwards. Clicking on this brings you to the main menu that has several options and suboptions. For instance, you can go to your home folder using one of the options. You can also copy and move files using mouse commands, or simply drag and drop files from one folder to another if you want to move them.

While in the folder window, you can right click the mouse and create a new folder. You can also rename folders or files or delete them.

3.7 EDITOR

Linux is rich in text manipulation and document preparation facilities. Here we take a brief look at `vi`, a line editor that is useful for creating and modifying text files. For example, if you want to be writing shell scripts or other programs, you will need an editor so that the program file can be created.

Text editors are different from the commands you have studied so far because you will be able to change existing files directly by using them. You need not be writing out the file to be changed to another file. Moreover, editors are interactive in that you need to be telling them what to do, command by command. This is unlike the commands you have seen so far, where you give the command once and it then does its job and terminates. Editors can be line editors or screen editors. Line editors work on a line at a time. Two line editors available in Linux are `ed` and `ex`. In contrast, screen editors present you with a screen of text and you can move around there, making changes as you want. So screen editors are more powerful and easier to use. A screen editor available in Linux is `vi`. Again because of lack of space we will only be able to look at a few basic features of `vi`. It is a programmer's editor that is not too easy to learn, though.

Unlike some other editors, `vi` does not automatically create a backup copy of the file being edited. Although editing does occur on a copy of the file, this copy is in the tmp directory that gets deleted when you save the file. The actual editing is done in a buffer in memory and the changes are written to the file only when you tell it to do so. You can therefore easily abandon a session that has gone badly wrong. But the same feature can be a problem in case of a system crash, though `vi` will try to recover as much as possible of the file that was then being edited.

It is difficult to describe an interactive command on paper and so `vi` is best understood by trying out the commands on a terminal. To start up `vi` and edit a file called linuxdoc, you say

```
[kumarr@linux kumarr] vi linuxdoc
```

At this the screen gets cleared, the file gets read into the edit buffer, the first portion of the window to the buffer appears on the screen and the cursor is at the first character of the line that you were editing when you last saved the file. This presupposes that `vi` knows how to deal with your terminal type, a task that would have been ensured by your system administrator when she set up your machine. You can resize your window at any time you wish without affecting your file. The bottom line of the screen shows the name of the file and its size in characters and lines. You can also see the current cursor position at the right of this status line. At times when you give commands to `vi`, the status bar disappears and instead you see the command that you are issuing. We will shortly look at some of these commands.

The text that you enter in `vi` is organised in lines. The editor is not a word processor and works only in non-document mode. So you need to explicitly tell `vi` when a line has ended. Otherwise it will continue to add text to the line until it reaches its limit for the length of a line. How do such long lines look on the screen? They are wrapped

onto the next physical line on the screen if the width of the screen has been reached. If you increase the width of the window you can see the line getting redrawn. Similarly if you reduce the width of the window you will be able to see the line being rearranged on the screen. While by mere visual inspection you cannot make out whether multiple lines are actually the same line or are separate, you can easily find out, say by resizing the screen or going to the end of the line by the $ command.

You can start up `vi` with more than one filename, and you can even give wildcards.

```
[kumarr@linux kumarr] vi linuxdoc xx xx.c
```

or

```
[kumarr@linux kumarr] vi *.c
```

In such a case the files are presented to you one by one. After you are done with the first file, you are presented with the next one unless you choose to exit the whole operation, in which case the rest of the files are not presented to you. In this respect vi differes from other word processors where you can only bring up one file at a time. In fact you can start up `vi` without any filename at all. If you do that you will be presented with a blank screen and can then start entering commands. You can give the file a name when you save it.

Another option with which you can start up `vi` is the `-x` option that allows you to encrypt the file. You have to supply a key (you need to retype it to confirm) and when you save the file it is encrypted with that key. To now read the file you must supply the same key, without which it will not be intelligible. While not a foolproof method, it will certainly safeguard you against the casual busybody. One more option to vi is to open it in readonly mode with `-R`. You can navigate and examine the file as you wish but cannot make accidental changes as you have to use the force options to commands that alter the file. This mode can also be called up by using the command `view` instead of `vi`.

The -r mode of vi tries its best to recover from system crashes. If you want to start editing from a line other than the first, you can use the `+n` option where `n` is the line number you want to be at. To go to the last line of the file, just omit the number and simply say +.

From the number of different ways in which you can start it up, you would have had some idea of the kinds of commands available and the bewildering array of options that must be supported by `vi`. Now that you know how to enter `vi`, let us also learn how to come out of it. You can save the file with :w and force a save, say when you are in readonly mode, by `:w!`. Similarly `:q` will quit the file without saving changes and `:q!` Will quit without trying to save any changes. To save and quit you can also say `:x` or simply `ZZ`.

Navigating Around the File

When you open up a file in `vi` you will find the cursor at the line you asked for during the invocation. Unlike a word processor, `vi` starts up in command mode. It does not start inserting the text you enter. So any key you press is taken to be a command. `Vi` has many commands, and just about any key is likely to be taken as one. Once a command is given, subsequent keys pressed will pertain to the command until you exit the command. If you press a key that is not a valid command in command mode, nothing will happen and vi will emit a beep.

For example, to move around the file, you can use the arrow keys. If your terminal type is not properly set and you have some problem with them, you can use `j, k, l` and `h` for down, up, right and left respectively. The commands l or h will work only within the line. At the last or first character respectively of the line, they have no effect. Similarly pressing k at the first line or `j` at the last line of the file has no effect. To scroll forward half a screen, use `^D` and to scroll half a screen back, use

Similarly, ^F and ^B will take you forward and backward one whole screenful respectively. If you are at the bottom of the screen and you press j or the down arrow key, the display scrolls up by one line and your cursor continues to remain at the last line. To scroll forward one line without changing the line at which the cursor is, use ^Y. Likewise, ^E will scroll backward one line without moving the cursor. The commands 0 and $ take you to the beginning and end of the current line respectively.

Operations on the window can blank out the status line, which you can always get back by ^G. To move forward one word, say w and use b to move backward one word. But unlike the character at a time commands, this works througout the file, which means that pressing b at the first word of a line will take you to the last word of the previous line. The command will not have any effect if you are at the first word of the file. Similarly w will have no effect at the end of the file. Note that b will bring you to the beginning of the current word if you are not already there and to the beginning of the previous word otherwise. Similarly e brings you to the end of the next word if you are at the end of the current one, and to the end of the current word itself otherwise. These lower case commands define a word in a way that is similar to the definition of an identifier in programming languages like C. This is very convenient while programming, as vi is really a programmer's editor. If you want to consider a word to be a sequence of characters delimited by spaces, you can use the upper case equivalents W, B and E.

These commands all take counts. This means 20W will take you 20 words forward and 90b will take you 90 words backward, with the appropriate definition of word. The % command takes you to the matching opening or closing (, { or [character. To move forward a sentence, use (and) to move a sentence back.

Similarly [and] move forward and backward one paragraph. These too can be preceded by a number that specifies the count.

Adding, Deleting and Changing Text

By now you have a good idea of the kind of commands vi takes. With your feet wet, it will not be hard to understand the commands for actually changing text. First let us see how to insert text. You can do this by using the i command. This puts you into insert mode. In this mode, whatever you type becomes part of the file. The text is added starting from before the current position of the cursor. To come out of insert mode back into command mode, you can say ESC or ^[. You can also use I to insert text at the beginning of the current line. Similarly, the a command adds text after the current position of the cursor, and A adds text at the end of the current line. To open up a new line under the current line, type o and use O if you want to open up a line before the current line. To add a control character to the file, type ^V followed by the character. The s command deletes the current character and puts into insert mode, while S deletes all text on the current line and puts you into insert mode.

Deleting text is similarly straightforward. The x command deletes the character at the current cursor position but does not move the cursor. At the end of the line, the command brings the previous character under the cursor. So pressing x repeatedly anywhere on a line will finally leave you with a blank line. The X command deletes characters to the left of the current cursor position and on reaching the beginning of the line no more characters are deleted.

To delete more than one character use the d command. This has to be followed by another letter to indicate what is to be deleted. So dw deletes a word, db deletes a word in the backword direction and dW and dB delete a word in the forward and backward directions according to the respective definitions. Word deletions happen across line boundaries. When deleting words on the next or previous line, the two lines are joined together. To delete the rest of a sentence from the current position, use d(, and d) will do the same up to the beginning of the sentence. The same can be done for paragraphs as well with the d{ or d} commands.

All the above deletion commands can be preceded by a number to indicate how many times they should be performed. To delete a line completely, say `dd`, or use `D` to delete the rest of the line from the current cursor position. Similarly d^ will delete a line from the beginning till the current cursor position. These commands will not remove the newline character. To delete a block of lines, give the range preceded by a colon, as in `:4,10d` to delete from the 4th to the 10th line (both inclusive). Here you can use a . to indicate the current line and $ to indicate the end of the file. So :.,$d will delete the rest of the file starting from the current line. You can also use the + sign to indicate the number of lines to delete, as in `:15,+8d` to delete 8 lines starting from the 15th line.

Internally `vi` maintains the line number of each line in the edit buffer. You can display line numbers by saying `:set number` or `:se nu`. All commands that begin with a colon have to be followed by the ENTER key before they will take effect.

Changing text is done in much the same way. The r command will replace the current character with whatever you type next, while R places you in replace mode. Then each character you type will replace the next character until you press ESC. To replace a fixed number of characters by the same character, precede r with the count, such as 23r. The ~ command changes the case of each character that is a letter. It too can be preceded by a count.

For changing blocks of text, use the c command followed by what you want to change, such as cw for a word, `cW` to use the other definition of a word, c$ to change text upto the end of the line and `c^` to change text upto the beginning of the line. To change the complete line on which the cursor is located, use cc. You can precede cw, cW, cb, cB or cc by a count to indicate how many words or lines are to be changed. All c commands place you in insert mode that you need to exit in the usual way with ESC or ^[.

The . command can be used to repeat the last action. To join the next line to the current one by deleting the newline character between them use the J command.

Searching for, Copying and Moving Text

Let us now see how to search for text. The f command searches for the next character you give on the same line. This can be preceded by a count, so saying 3fx will look for the 3rd x from the current cursor position. Similary F looks for the character in the backward direction. A ; continues the search for the same character in the same direction, while a , continues to search for the same character in the reverse direction. Both these commands work only on the current line, but the ; or , commands can be used on a different line after repositioning the cursor.

To search for any text, precede it by a / character. It place the cursor at the beginning of the first occurrence of the text pattern. To go to the next occurrence, say n and say N to find the same pattern in the reverse direction. If the pattern, say hello, is not found, the message

```
Pattern not found: hello
```

is displayed on the status line at the bottom of the window. You can start the search in the backward direction from the current curson position by preceding the text with ?. Both the / and ? will wrap around the file end or beginning. So after reaching the end, the search will continue from the beginning of the file. You can use the d command to delete text from the current cursor position to an occurrence of a search string, say `hello`, by saying d/`hello`.

You can bookmark a position in the file with m followed by any letter. So mx will bookmark the current cursor position. Now you can reach that position from anywhere in the file by saying 'x. To reach the beginning of the line containing the bookmark x, you can say 'x. Bookmarks are valid only for that edit session.

You can undo the effect of most commands by the u command, but repeating it merely brings back the previous situation. So the command works like a toggle. However, you can use U to undo all changes on a line if you have not moved away from it. Deleted text goes into an unnamed buffer, from where it can be retrieved by using the p command and placed after the current cursor position. The P command places it before the current cursor position. You can use these commands repeatedly to place the contents of the buffer at different points in the file. Thus to take care of a transposition error involving two characters, say `xp` at the first character.

The y command yanks a block of text into the unnamed buffer without deleting it. So you can use yw or yy (also Y) to yank a word or a line. These can be preceded by a count. You can then place the text wherever you want by first going to that point and then using p or P.

Finally, you can use any of 26 named buffers by preceding the command with " followed by the buffer name, which can be any lower case letter of the alphabet. Thus "m4yy will yank 4 lines and place them in the named buffer m. To put back the text at some point in the file, just go to that point and say "mp. You can append text to a named buffer by using the same buffer name but in upper case.

This completes our very quick overview of the vi editor. Linux also has available an office suite that includes a full fledged word processor suitable for editing text documents, while vi is useful for editing programs.

Linux has a full screen windows based editor, `gedit`, that creates text files. But it does not offer the wealth of commands and features that vi does, in spite of the fact that it is windows based. For a programmer, it might yet be better to use `vi`.

3.8 SUMMARY

In this chapter we have covered a lot of ground and the treatment of the topics has had to be brief. We have looked at some text manipulation commands that are frequently useful. We studied the meaning of permission modes for files that you see in the directory long listing, and also saw how to change them for files that we own. Next we saw how we can construct our own commands easily by joining together filters with pipes. This also brought us to the concept of the standard input, standard output and standard error. We then looked at how to write simple shell scripts in the bash shell. The graphical facilities available in Linux were then touched upon followed by a brief discussion of the programmer's editor vi.

3.9 SOLUTIONS/ ANSWERS

Check Your Progress 1

1) Linux does not type out the file. On trying to type out a directory called "linux" that does not exist, we get an error message

   cat: linux: Is a directory

2) On trying to cat a file that does not exist, we get an error message

   cat: highest: No such file or directory

3) If there are two files file1 and file2 that are the same after byte offsets offset1 and offset 2, you can verify this using cmp using

   $ cmp file1 file2 offset1 offset2

4) You can use wc on a binary file but the results are not meaningful for such files.

Check Your Progress 2

1) You will not be able to do so and will get an error message in both cases.

2) No. This is because of the definition of search permission. But you might be able to find out the files in it if you give it as the argument to the ls command.

3) Yes, you only need write permission on the directory. It does not matter whether you can write to the files or not or even whether you own them or not.

Check Your Progress 3

1) Try yourself

2) You can say
ls -l | wc -l

3) It is useful when you not only want to pipe the output of a command to another command, but also want to see or save that output.

Check Your Progress 4

1) It gives all the filenames except those that are hidden, that is, those that start with a period (.). To get these you can say
echo .*

2) Try yourself

3) Try yourself

3.10 FURTHER READINGS

There are a host of resources available for further reading on the subject of Red Hat Linux version 9.0.

1) http://www.redhat.com/docs/manuals/linux

2) http://www.linux.org gives among other information, a list of good books on Red Hat Linux.

3) Consider joining a good Linux mailing list.