Module2

UNIX looks at everything as a file and any UNIX system has thousands of files.

If we write a program , we add one more file to the system. When we compile it , we add some more.

Files grow rapidly, if they are not organized properly, we will it difficult to locate them.

Just as an office has separate file cabinets to group files of a similar nature, UNIX also organizes its own files in directories and expects us to do that as well.

The file system in UNIX is one of its simple and conceptually clean features. It lets the users access other files not belonging to them, but it also offers an adequate security mechanism so outsiders are not able to tamper with a file's content.

THE FILE:

The file is a container for storing information. We can treat it simply as a sequence of characters.

If we name a file **foo** and write three characters a,b and c into it, the **foo** will contain only the string abc and nothing else. UNIX file does not contain the eof (end-of-file) mark.

A file's size is not stored in the file, nor even its name.

All file attributes are kept in a separate area of the hard disk, not directly accessible to humans, but only to the kernel.

UNIX treats directories and devices as files as well.

A directory is simply a folder where you store filenames and other directories. All physical devices like the hard disk, memory, CD-ROM, printer and modem are treated as files. The shell is also a file , and so is the kernel. The main memory of the system is also treated as a file.

Categories of files:

1. **Ordinary file:** Also known as regular file. It contains only data as a stream of characters.
2. **Directory file:** It is commonly said that a directory contain files and other directories, but strictly speaking, it contains their names and a number associated with each name.
3. **Device file:** All devices and peripherals are represented by files. To read or write a device, you have to perform these operations on its associated file.

The reason why we make this distinction between file types is that the significance of a file's attributes often depends on its type. Read permission for an ordinary file means something quite different from that for a directory. Also, we cannot put something into a directory file, and a device file is not really a stream of characters.

Ordinary (Regular) file:

An ordinary file or a regular file is the most common file type. All programs that we write belong to this type. An ordinary file itself can be divided into two types:

- Text file
- Binary file

A **text file** contains only printable characters, and we can often view the contents and make sense out of them. All C and Java program sources, **shell** and **perl** scripts are text files.

A text file contains lines of characters where every line is terminated with the newline character, also known as a linefeed (LF). When we press [Enter] while inserting text, the LF character is appended to every line. We will not see this character normally, but there is a command (od) which can make it visible.

A **binary file,** contains both printable and unprintable characters that cover the entire ASCII range (0 to 255). Most UNIX commands are binary files, and the object code and executables that we produce by compiling C programs are also binary files.

Picture, sound and video files are binary files as well. Displaying such files with a simple cat command produces unreadable output and may even disturb the terminal's settings.

Directory File:

A directory contains no data, but keeps some details of the files and sub-directories that it contains.

The UNIX file system is organized with a number of directories and sub-directories, and we can create them as and when we need.

We often need to do that to group a set of files pertaining to a specific application. This allows two or more files in separate directories to have the same filename.

A directory file contains an entry for every file and subdirectory that it houses. If we have 20 files in a directory, there will be 20 entries in the directories. Each entry has two components:

- The filename.
- A unique identification number for the file or directory (called the inode number).

If a directory ***bar*** contains an entry for a file ***foo,*** we commonly say that the directory ***bar*** contains a file ***foo.*** Though we will be using the phrase "contains the file" rather than "contains the filename", we must interpret the statement literally.

A directory contains the filename and not the file's contents.

We cannot write a directory file, but we can perform some action that makes the kernel write a directory.

For instance, when we create or remove a file, the kernel automatically updates its corresponding directory by adding or removing the entry (***inode number and filename***) associated with the file.

Device File:

We will be printing files, installing software from CD-ROMs or backing up files to tapes. All of these activities are performed by reading or writing the file representing the device.

For instance, when we restore files from tape, we read the file associated with the tape drive.

It is advantageous to treat devices as files as some of the commands used to access an ordinary file also work with device files.

Device filenames are generally found inside a single directory structure, ***/dev.***

A device file is indeed special; it is not really a stream of characters. In fact, it does not contain anything at all. The operation of a device is entirely governed by the attributes of its associated file.

The kernel identifies a device from its attributes and then uses them to operate the device.

FILENAME:

A filename can consist of up to 255 characters.

Files may or may not have extensions, and can consist of practically any ASCII character except the / and the NULL character (ASCII value 0).

We are permitted to use control characters or other unprintable characters in a filename.

The following are valid filenames in UNIX:

.last_time

List.

^V^B^D-++bcd

-{}[]

@#$%*abcd

a.b.c.d.e

The third filename contains three control characters ([Ctrl +v] being the first). These characters should be definitely be avoided in framing filenames. Since the UNIX system has a special treatment for characters like $, ` , ? , * , & among others, it is recommended that only the following characters be used in filenames:

- Alphabetic characters and numerals
- The period (**.**),hyphen (-) and underscore (_).

UNIX imposes no rules for framing filename extensions.

A shell script does not need to have the .sh extension, even though it helps in identification. In all cases, it is the application that imposes the restriction.

Thus the C compiler expects C program filenames to end with **.c**. Oracle requires SQL scripts to have the **.sql** extension, and so forth. DOS/Windows users must also keep these two points in mind:

- A file can have as many dots embedded in its name; a.b.c.d.e is a perfectly valid filename. A filename can also begin with a dot or end with one.
- UNIX is sensitive to case; chap01, Chap01 and CHAP01 are three different filenames, and it is possible for them to coexist in the same directory.

Never use a – at the beginning of a filename. A command that uses a filename as argument often treats it as an option and reports errors.

For example, if you have a file named –z, cat –z will not display the file but interpret it as an invalid option.

The Parent – Child Relationship:

All files in UNIX are "related" to one another.

The file system in UNIX is a collection of all these related file (ordinary, directory and device files) organized in a hierarchical (an inverted tree) structure.

The UNIX file system is visually represented as shown in the following figure:

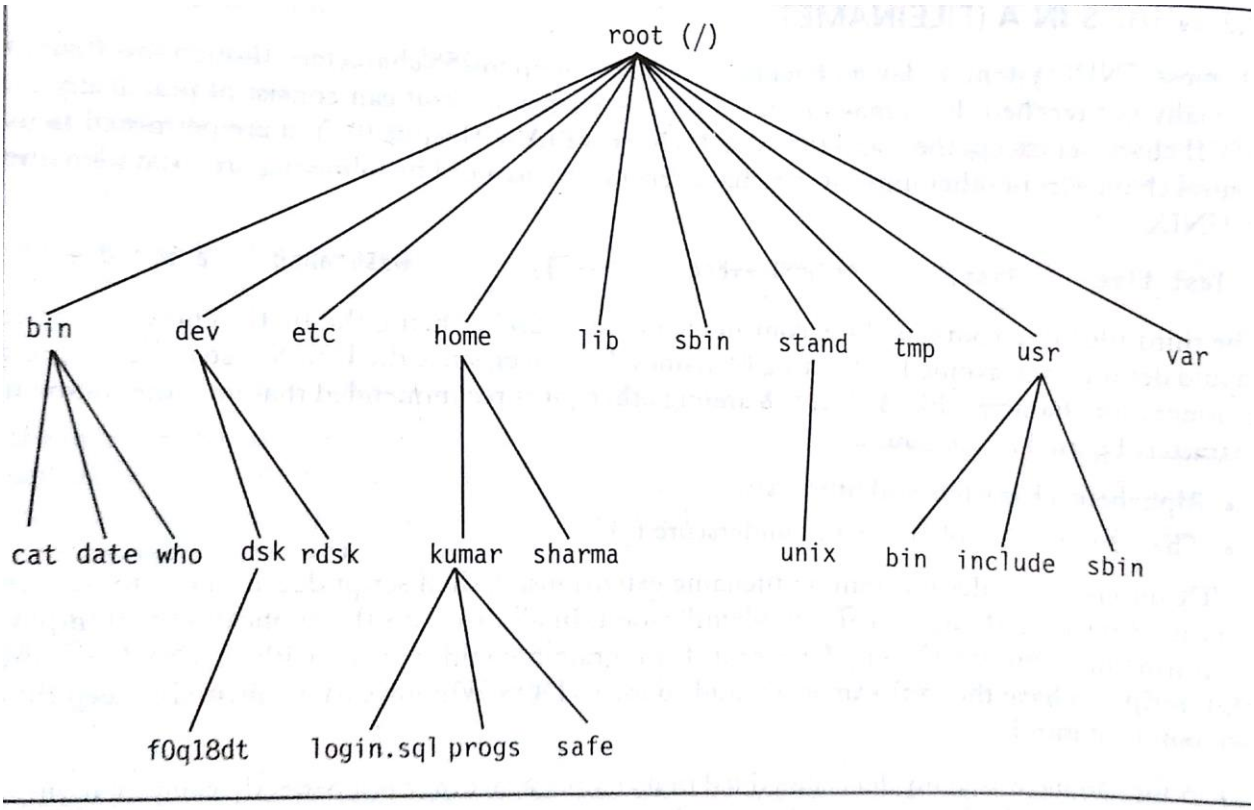

Fig. 4.1 The UNIX File System Tree

The implicit feature of UNIX file system is that there is a top, which serves as the reference point for all files.

This top is called ***root*** and is represented by a **/** (frontslash).

root is actually a directory. It is conceptually different from the user-id root used by the system administrator to log in.

The root directory (**/**) has a number of subdirectories under it. These subdirectories in turn have more subdirectories and other files under them.

For instance, ***bin*** and ***usr*** are two directories directly /, while a second ***bin*** and ***kumar*** are subdirectories under ***usr***.

Every file, apart from the root, must have a parent, and it should be possible to trace the ultimate parentage of a file to root.

Thus, the home directory is the parent of ***kumar***, while / is the parent of ***home***, and the grandparent of ***kumar.***

If we create a file ***login.sql*** under the ***kumar*** directory, then ***kumar*** will be the parent of this file.

In parent – child relationships, the parent is always a directory.

The HOME Variable: The Home Directory

When we log on to the system, UNIX automatically places us in a directory called the **home directory**. It is created by the system when a user account is opened.

If we log in using the login name ***kumar***, we will land up in a directory that could have the pathname ***/home/kumar*** (or something else).

The shell variable HOME knows our home directory:

$ echo $HOME

/home/kumar **First / represent the root directory**

What we see above is an **absolute pathname**, which is simply a sequence of directory names separated by slashes.

An absolute pathname shows a file's location with reference to the top, i.e, the root. These slashes act as delimiters to file and directory names, *except that the first slash is a synonym for root.*

The directory ***kumar*** is placed two levels below root.

It is often convenient to refer to a file ***foo*** located in our home directory as **$HOME/foo.** But, most shells also use the ~ symbol for this purpose. So, **$HOME/foo** is the same as **~/foo** in these shells.

The ~ symbol can refer to any user's home directory and not just our own.

If user sharma has the file foo in his home directory, then kumar can access it as ~sharma/foo.

The principle is:

A tilde followed a / (like ~/foo) refers to one's home directory, but when followed by a string (~sharma) refers to the home directory of that user represented by the string.

pwd: Checking our Current Directory:

UNIX encourages us to believe that, like a file, a user is placed in a specific directory of the system on logging in. We can move around form one directory to another, but at any point of time, we are located in only one directory. This directory is known as the **current directory.**

At any time, we should be able to know what our current directory is.

The **pwd** (print working directory) tells us what our current directory is.

$pwd

/home/kumar

Like HOME, **pwd** displays the absolute pathname.

cd: Changing the Current Directory:

We can move around the file system by using the **cd** (change directory) command.

When used with an argument, it changes the current directory to the directory specified as argument, for instance, **progs:**

$pwd

/home/kumar

$cd progs

$pwd

/home/kumar/progs

Though **pwd** displays the absolute pathname, cd does not need to use one.

The command **cd progs** here means this: "Change our subdirectory to progs under the current directory".

The pathname **cd /home/kumar/progs** can also be used for the same effect.

When we need to switch to the /bin directory where most of the commonly used UNIX commands are kept, we need to use the absolute pathname:

$pwd

/home/kumar/progs

$cd /bin *Absolute pathname required here because bin is*

$pwd *not in current directory*

/bin

cd can also be used without any arguments:

$pwd

/home/kumar/progs

$cd *cd used without arguments reverts to the home directory*

$pwd

/home/kumar

mkdir: Making Directories:

Directories are created with the **mkdir** (make directory) command.

The command is followed by names of the directories to be created.

A directory *patch* is created under the current directory like this:

mkdir patch

We can create a number of subdirectories with one **mkdir** command:

mkdir patch dbs doc

UNIX system goes further and lets us create directory trees with just one invocation of the command. For instance, the following command creates a directory tree:

mkdir pis pis/progs pis/data

This creates three subdirectories – pis and two subdirectories under pis. The order of specifying the arguments is important; we obviously cannot create a subdirectory before the creation of its parent directory.

For instance, we cannot enter

$ mkdir pis/progs pis/data pis

mkdir: Failed to make directory “pis/progs”; No such file or directory

mkdir: Failed to make directory “pis/data”; No such file or directory

We can note that even though the system failed to create two subdirectories, *progs* and *data* , it has still created the *pis* directory.

Sometimes, the system refuses to create a directory:

$ mkdir test

mkdir: Failed to make directory “test”; Permission denied

This can happen due to these reasons:

- The directory test may already exist.
- There may be an ordinary file by that name in the current directory.
- The permissions set for the current directory do not permit the creation of files and directories by the user. We will certainly get this message if we try to create a directory in /bin, /etc or any other directory that houses the UNIX system’s files.

rmdir: Removing Directories:

The **rmdir** (remove directory) command removes directories. We simply have to do this to remove the directory pis:

rmdir pis Directory must be empty

Like **mkdir, rmdir** can also delete more than one directory in one shot. For instance, the three directories and subdirectories that were just created with **mkdir** can be removed by using **rmdir** with a reversed set of arguments:

rmdir pis pis/progs pis/data

We can note that when we delete a directory and its subdirectories , a reverse logic has to be applied. The following directory sequence used by **mkdir** is invalid in **rmdir:**

$ rmdir pis pis/progs pis/data

Rmdir: directory "pis": Directory not empty

We can observe that ***rmdir*** has deleted the lowest level subdirectories ***progs*** and ***data***.

This error message leads to two important rules that we should remember when deleting directories:

- We cannot delete a directory with ***rmdir*** unless it is empty. In the above case, the pis directory could not be removed because of the existence of the subdirectories , ***progs*** and ***data***, under it.
- We cannot remove a subdirectory unless we are placed in a directory which is hierarchically above the one we have chosen to remove.

The ***mkdir*** and ***rmdir*** commands works only in directories owned by the user. Generally a user is the owner of her home directory, and she can create and remove subdirectories (as well as regular files) in this directory or in any subdirectories created by her.

Absolute Pathnames:

Many UNIX commands use file and directory names as arguments, which are presumed to exist in the current directory. For instance, the command

cat login.sql

will work only if the file *login.sql* exists in our current directory. However, if we are placed in */usr* and want to access *login.sql* in */home/kumar*, we cannot obviously use the above command, but rather the pathname of the file:

cat /home/kumar/login.sql

If the first character of a pathname is /, the file's location must be determined with respect to root (the first /). Such a pathname is called an **absolute pathname.**

When we have more than one / in a pathname, for each such /, we have to descend one level in the file system. Thus, *kumar* is one level below *home*, and two levels below *root*.

No two files in a UNIX system can have identical absolute pathnames.

If we have two files with the same name, but in different directories; their pathnames will also be different. Thus, the file */home/kumar/progs/c2f.pl* can coexist with the file */home/kumar/safe/c2f.pl*

Using the Absolute Pathname for a Command:

UNIX Command Runs By Executing Its Disk File.

When we specify the ***date*** command, the system has to locate the file *date* from a list of directories specified in the PATH variable, and then execute it.

However, if we know the location of a particular command, we can precede its name with the complete path.

Since date resides in /bin (or /usr/bin), we can also use the absolute pathname:

$ /bin/date

Thu Sep 1 09:30:49 IST 2005

For any commands that resides in the directories specified in the PATH variable, we need not use the absolute pathname.

If we execute programs residing in some other directory that isn't in PATH, the absolute pathname then needs to be specified.

For example, to execute the program **less** residing in */usr/local/bin*, we need to enter the absolute pathname:

/usr/local/bin/less

Relative Pathnames:

Consider the following commands:

cd progs

cat login.sql

Here, both *progs* and *login.sql* are presumed to be in the current directory.

Now, if *progs* also contains a directory scripts under it, we still do not need an absolute pathname to change to that directory:

cd progs/scripts

Such pathnames that does not begin with a /, are called as relative pathnames.

Using . and .. in Relative Pathnames:

UNIX offers a shortcut that uses either the current or parent directory as reference, and specifies the path relative to it. A relative pathname uses one of these cryptic symbols:

- **.** (a single dot) – This represents the current directory
- **. .** (two dots) – This represents the parent directory

Assuming that we are placed in ***/home/kumar/progs/data/text***, we can use **. .** as an argument to **cd** to move to the parent directory, ***/home/kumar/progs/data***:

$ pwd
/home/kumar/progs/data/text
$ cd . . Moves one level up
$ pwd
/home/kumar/progs/data

This method is compact and more useful when ascending the hierarchy .

The command **cd. .** translates to this: "Change your directory to the parent of the current directory".

We can combine any number of such sets of . . separated by /s. However, when a / is used with . . it acquires a different meaning; instead of moving down a level, it moves one level up.

For instance , to move to /home, we can always use **cd /home.** Alternatively, we can also use a relative pathname.

$ pwd
/home/kumar/pis
$ cd . . / . . Moves two levels up
$ pwd
/home

Any command which uses the current directory as argument can also work with a single dot. This means that the **cp** command which also uses a directory as the last argument can be used with a dot:

cp . ./sharma/ . profile . A filename can begin with a dot

This copies the file .profile to the current directory **(.)**.

ls: Listing Directory Contents:

ls command can be used to obtain the list of all filenames in the current directory.

```
$ ls
08_packets.html
TOC.sh
calendar
cptodos.sh
dept.lst
emp.lst
helpdir
progs
usdsk06x
usdsk07x
usdsk08x
```

Numerals first
Uppercase next
Then lowercase

What we see here is a complete list of filenames in the current directory arranged in **ASCII collating sequence** (numbers first, uppercase and then lowercase), with one filename in each line.

Directories often contain many files, and we may simply be interested in only knowing whether a particular file is available. In that case, just use **ls** with the filename:

$ ls calendar
Calendar

and if **perl** is not available, the system clearly says so:

$ ls perl
perl: No such file or directory

ls can also be used with multiple filenames, and has options that list most of the file attributes.

ls Options:

ls has a large number of options as listed in the table below:

Table 4.1 Options to `ls`

Option	*Description*
`-x`	Multicolumnar output
`-F`	Marks executables with *, directories with / and symbolic links with @
`-a`	Shows all filenames beginning with a dot including . and ..
`-R`	Recursive list
`-r`	Sorts filenames in reverse order (ASCII collating sequence by default)
`-l`	Long listing in ASCII collating sequence showing seven attributes of a file *(6.1)*
`-d` *dirname*	Lists only *dirname* if *dirname* is a directory *(6.2)*
`-t`	Sorts filenames by last modification time *(11.6)*
`-lt`	Sorts listing by last modification time *(11.6)*
`-u`	Sorts filenames by last access time *(11.6)*
`-lu`	Sorts by ASCII collating sequence but listing shows last access time *(11.6)*
`-lut`	As above but sorted by last access time *(11.6)*
`-i`	Displays inode number *(11.1)*

Output in Multiple Columns (-x): When we have several files, it is better to display the filenames in multiple columns. Modern versions of **ls** do that by default, but if does not that happen in our system, we use the **–x** option to produce a multicolumnar output:

$ ls –x

```
08_packets.html   TOC.sh      calendar     cptodos.sh
dept.lst          emp.lst     helpdir      progs
usdsk06x          usdsk07x    usdsk08x     ux2nd06
```

Identifying Directories and Executables (-F): The output of **ls** that we have seen so far merely showed the filenames. We did not know how many of them were directory files. To identify directories and executable files, the –F option should be used. Combining this option wiith **–x** produces a multicolumnar output as well:

```
$ ls -Fx
08_packets.html   TOC.sh*     calendar*    cptodos.sh*
dept.lst          emp.lst     helpdir/     progs/
usdsk06x          usdsk07x    usdsk08x     ux2nd06
```

The ***** indicates that the file contains executable code and the / refers to a directory. We can now identify the two subdirectories in the current directory – helpdir and progs.

Showing Hidden Files Also (-a): **ls** does not normally show all the files in a directory. There are certain hidden files (filenames beginning with a dot), often found in the home directory, that normally do not show up in the listing. The –a option (all) lists all hidden files as well:

```
$ ls -axF
./                ../               .exrc        .kshrc
.profile          .rhosts           .sh_history  .xdtsupCheck
.xinitrc          08_packets.html*  TOC.sh*      calendar*
  .....
```

The file **.profile** contains a set of instructions that are performed when a user logs in. The other file **.exrc,** contains a sequence of instructions for the **vi** editor.

The first two files (. and . .) are special directories. They represent the current and the parent directories. Whenever we create a subdirectory , these "invisible" directories are created automatically by the kernel. We cannot remove them, nor can we write into them. They help in holding the file system together.

Listing Directory Contents:

```
$ ls -x helpdir progs
helpdir:
forms.obd     graphics.obd     reports.obd

progs:
array.pl      cent2fah.pl      n2words.pl      name.pl
```

This time the contents of the directories are listed, consisting of the Oracle documentation in the *helpdir* directory and a number of **perl** program files in progs.

ls when used with directory names as arguments, does not simply show their names as it does with ordinary files.

Recursive Listing (-R)

The -R option lists all files and subdirectories in a directory tree. This traversal is done recursively until there are no subdirectories left:

```
$ ls -xR
08_packets.html  TOC.sh           calendar         cptodos.sh
dept.lst         emp.lst          helpdir          progs
usdsk06x         usdsk07x         usdsk08x         ux2nd06

./helpdir:
forms.hlp        graphics.hlp     reports.hlp

./progs:
array.pl         cent2fah.pl      n2words.pl       name.pl
```

The list shows the filenames in three sections – the ones under the home directory and those under the subdirectories **helpdir** and **progs**.

./helpdir – indicates that helpdir is a subdirectory under . (the current directory).

The UNIX File System:

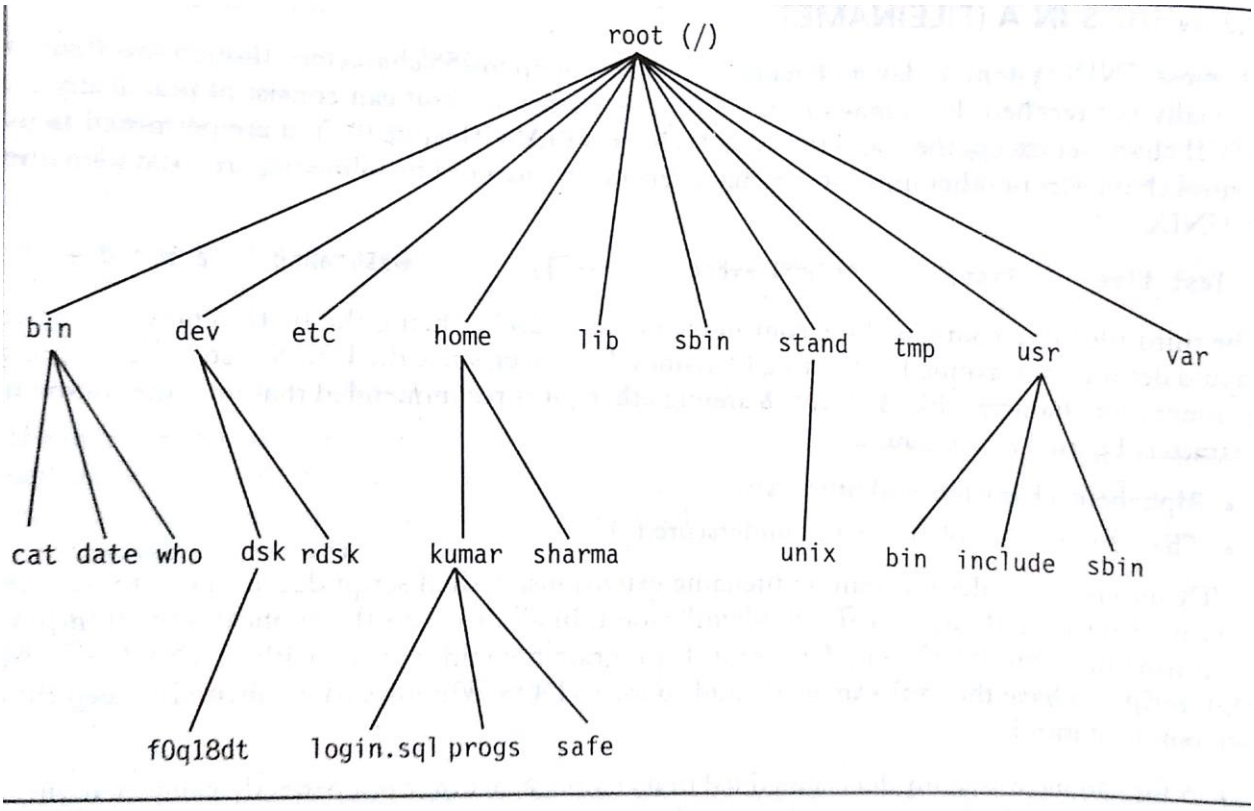

Fig. 4.1 The UNIX File System Tree

The above figure a heavily trimmed structure of a standard UNIX file system.

We can view the entire file system as comprising two groups of files. The first group contains the files that are made available during system installation:

- **/bin and /usr/bin** – These are the directories where all the commonly used UNIX commands (binaries , hence the name bin) are found.
- **/sbin and /usr/sbin –** These directories contain the commands that can only be executed by the system administrator.

- **/etc -** This directory conatains the configuration files of the system. We can change a very important aspect of system fucntioning by editing a text file in this directory. The user's login name and password are stored in files /etc/passwd and /etc/shadow.
- **/dev** – This directory contains all the device files. These files do not occupy space on disk.
- **/lib and /usr/lib -** Contain all library files in binary form.We will need to link our C programs with files in these directories.
- **/usr/include** – Contains the standard header files used by C programs. The statement #include<stdio.h> used in most C programs refers to the file stdio.h in this directory.
- **/usr/share/man** – This is where the man pages are stored.

The contents of the above mentioned directories would change as more software and utilities are added to the system.

Users also work with their own files; they write programs, send and receive mail and also create temporary files. These files are available in the second group shown below:

- **/tmp** – The directories where users are allowed to create temporary files. These files are wiped away regularly by the system.
- **/var** – The variable part of the file system. Contains all our print jobs and our outgoing and incoming mail.
- **/home** – On many systems the users are housed here.

cat: Displaying and creating files:

It is mainly used to display the contents of a small file on the terminal:

```
$ cat dept.lst
01|accounts|6213
02|progs|5423
03|marketing|6521
04|personnel|2365
05|production|9876
06|sales|1006
```

cat , like several other UNIX commands, also accepts more than one filename as arguments:

cat chap01 chap02

The contents of the second file are shown immediately after the first file without any header information. **cat** concatenates the two files and hence its name.

cat Options (-v and -n):

*Displaying Nonprinting Characters (**-v**):*

cat is normally used for displaying text files only. Executables, when seen with **cat,** simply display junk.

If we have nonprinting ASCII characters in our input, we can use **cat** with the –v option to display these characters.

*Numbering Lines (**-n**):*

The –n option numbers lines.

C compilers indicate the line number where errors are detected, and this numbering facility often helps a programmer in debugging programs.

Using cat to create a file:

cat can also be used to create a file.

Enter the command **cat** , followed by the > (the right chevron) character and the filename.

```
$ cat > foo
A > symbol following the command means that the
output goes to the filename following it. cat used
in this way represents a rudimentary editor.
[Ctrl-d]
$ _
```

When the command line is terminated with [*Enter*], the prompt vanishes. **cat** now waits to take input from the user.

In the above example, we entered the three lines, each followed by [*Enter*]. Finally press [*Ctrl + d*] to signify the end of input to the system. The file is written and the prompt returned.

To verify this, we can again the use the **cat** command as follows:

```
$ cat foo
A > symbol following the command means that the
output goes to the filename following it. cat used
in this way represents a rudimentary editor.
```

cat is a versatile command. It can be used to create , display, concatenate and append to files.

It does not restrict itself to handling files only; it also acts on a stream. We can supply the input to **cat** not only by specifying the filename, but also from the output of another command.

cp: Copying a File:

The cp (copy) command copies a file or a group of files.

It creates an exact image of the file on disk with a different name.

The syntax requires at least two filenames to be specified in the command line.

When both are ordinary files, the file is copied to the second:

cp chap01 unit1

If the destination file (unit1) does not exist, it will first be created before copying takes place. If not, it will simply be overwritten without any warning from the system.

If there is only one file to be copied, the destination can be either an ordinary file or directory. We then have the option of choosing our destination filename.

The following example shows two ways of copying a file to the *progs* directory:

```
cp chap01 progs/unit1        chap01 copied to unit1 under progs
cp chap01 progs              chap01 retains its name under progs
```

cp is often used with the shorthand notation, .(dot), to signify the current directory as the destination.

For instance, to copy the file .profile from /home/sharma to your current directory, we can use either of the two commands:

```
cp /home/sharma/.profile .profile      Destination is a file
cp /home/sharma/.profile .             Destination is the current director
```

The second one is preferable because it requires fewer keystrokes.

cp can also be used to copy more than one file with a single invocation of the command. In that case, the last filename must be a directory.

For instance, to copy the files chap01, chap02 and chap03 to the progs directory, we have to use **cp** like this:

cp chap01 chap02 chap03 progs

The files retain their original names in progs.

If these files are already resident in progs, they will be overwritten.

For the above command to work, the progs directory must exist because cp will not create it.

We can compress the above sequence using * to frame a pattern for matching more than one filename.

cp chap* progs *Copies all files beginning with chap*

cp Options:

Interactive Copying **(-i):**

The –i (interactive) option warns the user before overwriting the destination file.

If unit1 exists, **cp** prompts for a response:

```
$ cp -i chap01 unit1
cp: overwrite unit1 (yes/no)? y
```

Copying Directory Structures **(-R):**

The **cp – R** command behaves recursively to copy an entire directory structure, say, progs to newprogs:

cp –R progs newprogs *newprogs must not exist*

If **newprogs** does not exist, **cp** creates it along with associated subdirectories. But if **newprogs** exists, **progs** becomes a subdirectory under **newprogs.** This means that the command run twice in succession will produce different results.

rm : Deleting Files:

The **rm** command deletes one or more files.

The following command deletes three files;

rm chap01 chap02 chap03

A file once deleted cannot be recovered.

rm will not remove a directory, but it can remove files from one.

We can remove two chapters from the progs directory without having to "cd" to it:

rm progs/chap01 progs/chap02

We may sometimes need to delete all files in a directory as a part of cleanup operation.

The *, when used by itself, represents all files:

**$ rm *
$ _**

rm Options:

Interactive options **(-i)**

This option makes the command ask the user for confirmation before removing each file:

```
$ rm -i chap01 chap02 chap03
rm: remove chap01 (yes/no)? ?y
rm: remove chap02 (yes/no)? ?n
rm: remove chap03 (yes/no)? [Enter]        No response—file not deleted
```

A *y* removes any file, any other responses leaves the file undeleted.

Recursive Deletion **(-r or -R)**

With this option the rm performs a tree walk - a thorough recursive search for all subdirectories and files within these subdirectories. At each stage , it deletes everything it finds.

rm will not remove directories , but when used with is option , it will.

rm –r*

we will delete all files in the current directory and all its subdirectories.

Forcing Removal **(-f)**

rm prompts for removal if a file is write protected. The –f option to overrides this minor protection and forces removal. When we combine it with the –r option , it could be the most risky thing to do:

rm –rf*

mv: Renaming files:

The **mv** command renames (moves) files. It has two distinct functions:

- It renames a file (or directory)
- It moves a group of files to a different directory.

mv does not create a copy of the file, it just renames it.

No additional space is consumed on disk during renaming.

To rename a file chap01 to man01, we should use

mv chap01 man01

If the destination file does not exist, it will be created.

For the above example, **mv** simply replaces the existing filename in the existing directory entry with the new name. By default, **mv** does not prompt for overwriting the destination file if it exists.

A group of files can be moved to a directory. The following command moves three files to the *progs* directory:

mv chap01 chap02 chap03 progs

mv can also be used to rename a directory.

mv pis perdir

wc: Counting Lines, Words and Characters

wc is universal word-counting command that also counts lines and characters.

It takes one or more filenames as arguments and displays a four-columnar output.

Before we use **wc** on the file infile, we can use **cat** to view its contents.

```
$ cat infile
I am the wc command
I count characters, words and lines
With options I can also make a selective count
```

We can now use the **wc** command without options to make a "word count" of the data in the file.

```
$ wc infile
      3     20    103 infile
```

wc counts 3 lines, 20 words and 103 characters. The filename has also been shown in the fourth column. The meaning of these terms are as follows:

- A **line** is any group of characters not containing a newline.
- A **word** is a group of characters not containing a space, tab , or newline.
- A **character** is the smallest unit of information, and includes a space, tab and newline.

wc offers three options to make a specific count.

The –l option counts only the number of line, while the –w and –c options count words and characters, respectively:

```
$ wc -l infile
      3 infile                    Number of lines
$ wc -w infile
     20 infile                    Number of words
$ wc -c infile
    103 infile                    Number of characters
```

When used with multiple filenames, wc produces a line for each file, as well as a total count:

```
$ wc chap01 chap02 chap03
    305    4058   23179 chap01
    550    4732   28132 chap02
    377    4500   25221 chap03
   1232   13290   76532 total
```

A total as a bonus

od : Displaying data in Octal:

Displays non printing characters in a file.

The file *odfile* contains some of the characters that do not show up normally:

```
$ more odfile
White space includes a
The ^G character rings a bell
The ^L character skips a page
```

The apparently incomplete first line actually conatains a tab (entered by hitting [Tab]). The od command makes these commands visible by displaying the ASCII octal values of its input.

The –b option displays this value for each character separately .

```
$ od -b odfile
0000000 127 150 151 164 145 040 163 160 141 143 145 040 151 156 143 154
0000020 165 144 145 163 040 141 040 011 012 124 150 145 040 007 040 143
.....Output trimmed....
```

Each line displays 26 bytes of data octal, preceded by the offset (position) in the file of the first byte in the line.

When the –b and –c (character) options are combined, we get a better readable output.

```
$ od -bc odfile
0000000 127 150 151 164 145 040 163 160 141 143 145 040 151 156 143 154
          W   h   i   t   e       s   p   a   c   e       i   n   c   l
0000020 165 144 145 163 040 141 040 011 012 124 150 145 040 007 040 143
          u   d   e   s       a       \t  \n  T   h   e       007     c
0000040 150 141 162 141 143 164 145 162 040 162 151 156 147 163 040 141
          h   a   r   a   c   t   e   r       r   i   n   g   s       a
0000060 040 142 145 154 154 012 124 150 145 040 014 040 143 150 141 162
              b   e   l   l  \n   T   h   e      \f       c   h   a   r
  .....
```

Each line is now replaced with two.

The octal representations are shown in the first line.

The printable characters and escape characters are shown in the second line.

The first character in the first line is the letter W having the octal value 127.

The tab character, [Ctrl -i], is shown as \t and the octal value 011.

The bell character, [Ctrl-g], is shown as 007. Some of the systems show it a \a.

The formfeed character, [Ctrl-l], is shown as \f and 014

The LF (linefeed or newline) character, [Ctrl-j], is shown as \n and 012.

File Attributes:

The *ls – l* displays most of the file attributes – like its permissions, size and ownership details.

ls looks up the file's inode to fetch its attributes.

```
$ ls -l
total 72
-rw-r--r--   1 kumar    metal      19514 May 10 13:45 chap01
-rw-r--r--   1 kumar    metal       4174 May 10 15:01 chap02
-rw-rw-rw-   1 kumar    metal         84 Feb 12 12:30 dept.lst
-rw-r--r--   1 kumar    metal       9156 Mar 12  1999 genie.sh
drwxr-xr-x   2 kumar    metal        512 May  9 10:31 helpdir
drwxr-xr-x   2 kumar    metal        512 May  9 09:57 progs
```

The list is precede by the words **total 72,** which indicates that a total of 72 blocks are occupied by these files on disk, each block consisting of 512 bytes (1024 in Linux).

The following are the different fields in the above output:

1. **File Type and Permission:**

   The first column shows the type and permissions associated with each file. The first character in this column is mostly a -, which indicates that the file is an ordinary file. But for directory files *helpdir* and *progs* , we can see a **d** at the same position.

   We then see a series of characters that can take the values r, w, x and -. In the UNIX system, a file can have three types of permissions – read , write and execute.

2. **Links:**

   The second column indicates the number of links associated with the file. This is actually the number of filenames maintained by the system of that file. UNIX lets a file have as many names as we want it to have, even though there is a single file on disk.

3. **Ownership:**

   When we create a file, we automatically become its owner. The third column shows *kumar* as the owner of all these files. The owner has the full authority to tamper with a file's contents and permissions - a privilege not available to others except the root user. Similarly we can create , modify or remove files in a directory if we are the owner of the directory.

4. **Group Ownership:**

   When opening a user account, the system administrator also assigns the user to some group. The fourth column represents the *group owner* of the file. The concept of a group of users also owning a file has acquired importance today as group members often need to work on the same file. It is generally desirable that the group have a set of privileges distinct from others as well as the owner.

5. **File Size:**

   The fifth column shows the size of the file in bytes, i.e., the amount of data it contains. It is only a character count of the file and not a measure of disk space that it occupies. The space occupied by a file on disk is usually larger than this figure since files are written to disk in blocks of 1024 bytes or more.

   The two directories show smaller file sizes (512 bytes each). This is to be expected as a directory maintains a list of filenames along with an identification number (the inode number) for each file.

6. **Last Modification Time:**

   The sixth, seventh and eighth columns indicate the last modification time of the file, which is stored to the nearest second. A file is said to be modified only if its contents have changed in any way. If we change only the permissions or ownership of the file, the modification time remain unchanged.

   If the file is less than a year old since its last modification date, the year will not be displayed.

7. **Filename:**

   The last column displays the filenames arranged in ASCII collating sequence. The UNIX filenames can be very long (up to 255 characters).

The –d Option: Listing Directory Attributes:

ls when used with directory names, lists files in the directory rather than the directory itself.

To force **ls** to list the attributes of a directory, rather than its contents, we need to use the **–d** (directory) option:

```
$ ls -ld helpdir progs
drwxr-xr-x   2 kumar    metal         512 May  9 10:31 helpdir
drwxr-xr-x   2 kumar    metal         512 May  9 09:57 progs
```

Directories are easily identified in the listing by the first character of the first column, which in the above output shows a d.

For ordinary files, this slot always shows a – (hypen), and for device files, either a b or c.

File Ownership:

When we create a file, our username shows up in the third column of the file's listing. It specifies the **owner** of the file.

Our group name is seen in the fourth column; our group is the group owner of the file.

If we copy someone else's file, we are the owner of the copy.

If we cannot create files in other user's home directories, it is because those directories are not owned by us (and the owner has not allowed us write access).

Several users may belong to a single group.

People working on a project are generally assigned a common group, and all files created by group members (who have separate user-ids) will have the same group owner.

The privileges of the group are set by the owner of the file and not by the group members.

When the system administrator creates a user account, he has to assign these parameters to the user:

- The user-id (UID) – both its name and numeric representation.
- The group-id (GID) – both its name and numeric representation.

The file */etc/passwd* maintains the UID (both the number and name) and GID (but only the number). */etc/group* contains the GID (both number and name).

File Permissions:

UNIX has a simple and well defined system of assigning permissions to files.

Consider the **ls-l** command to view the permissions of few files:

```
$ ls -l chap02 dept.lst dateval.sh
-rwxr-xr--   1 kumar    metal      20500 May 10 19:21 chap02
-rwxr-xr-x   1 kumar    metal        890 Jan 29 23:17 dateval.sh
-rw-rw-rw-   1 kumar    metal         84 Feb 12 12:30 dept.lst
```

The first column in the above output represents the file permissions. These permissions are also different for the three files.

UNIX follows a three-tiered file protection system that determines a file's access rights.

To understand how this system works, let us break up the permissions string of the file ***chap02*** into three groups.

The initial – (in the first column) represents an ordinary file and is left out of the permission string:

r w x r – x r - -

Each group here represents a category and contains three slots, representing the read, write and execute permissions of the file , in that order.

r indicates read permission, which means **cat** can display the file.

w indicates write permission; we can edit such a file with an editor.

x indicates execute permission, the file can be executed as a program.

The – shows the absence of the corresponding permission.

The first group (rwx) has all three permissions. The file is readable, writable and executable by the *owner* of the file. The third column shows *kumar* as the owner and the first permissions group applies to kumar. We have to login with the username kumar for these privileges to apply to us.

The second group (r-x) has a hyphen in the middle slot, which indicates the absence of write permission by the *group owner* of the file. This group owner is metal, and all users belonging to the metal group have read and write permissions only.

The third group (r- -) has only read permission. This set of permissions is applicable to *others*, i.e., those who are neither the owner nor belong to the metal group. This category (others) is often referred to as the world. The file is not world-writable.

We can set different permissions for the three categories of users – owner, group and others.

chmod: Change File Permissions:

A file or directory is created within a default set of permissions.

Generally, the default setting write – protects a file from all except the user (owner), though all user may have access.

```
$ cat /usr/bin/startx > xstart                    Actually copies the file startx
$ ls -l xstart
-rw-r--r--   1 kumar    metal       1906 Sep  5 23:38 xstart
```

In the above output, it seems, by default, a file does not also have execute permission.

The permissions of the file can be changed using the **chmod** command.

The **chmod (**change mode**)** command is used to set the permissions of one or more files for all the three categories of users (user, group and others).

It can be run only by the user (the owner) and the superuser.

The command can be used in two ways:

- In a relative manner by specifying the changes to the current permissions.
- In an absolute manner by specifying the final permissions.

Relative Permissions:

When changing permissions in a relative manner, **chmod** only changes the permissions specified in the command line and *leaves all other permissions unchanged.*

In this mode it uses the following syntax:

chmod *category operation permission filename (s)*

chmod takes as its arguments an expression comprising some letters and symbols that completely describe the user category and type of permission being assigned or removed.

The expression contains three components:

- User category (user, group , others)
- The *operation* to be performed (assign or remove a permission)
- The type of *permission* (read, write, execute)

By using suitable abbreviations for each of these components , we can form compact expression and then use it as an argument to **chmod.** The abbreviations used for these three components are shown in the following table:

Table 6.1 Abbreviations Used by **chmod**

Category	*Operation*	*Permission*
u—User	+—Assigns permission	r—Read permission
g—Group	-—Removes permission	w—Write permission
o—Others	=—Assigns absolute permission	x—Execute permission
a—All (ugo)		

Example:

To assign execute permission to the user of the file *xstart*, we need to frame a suitable expression by using appropriate characters from each of the three columns of the above table.

Since the fields needs to executable only by the user, the expression required is u+x:

```
$ chmod u+x xstart
$ ls -l xstart
-rwxr--r--   1 kumar    metal        1906 May 10 20:30 xstart
```

The command assigns (+) execute (x) permission to the user (u), but other permissions remain unchanged.

We can now execute the file if you are the owner of the file but other categories (i.e. group and others) still cannot.

To enable all of them to execute this file, we have to use multiple characters to represent the user category (ugo):

```
$ chmod ugo+x xstart ; ls -l xstart
-rwxr-xr-x   1 kumar    metal        1906 May 10 20:30 xstart
```

The string ugo combines all the three categories – user, group and others.

UNIX also offers a shorthand symbol a (all) to act as a synonym for the string.

There is also a shorter form that combines these three categories. When it is not specified, the permission applies to all categories. So the previous sequence can be replaced by either of the following:

chmod a+x xstart	a implies ugo
chmod +x xstart	By default, a is implied

chmod accepts multiple filenames in the command line.

When you need to assign the same set of permissions to a group of files, this is what we should do:

chmod u+x note note1 note3

Permissions are removed with the – operator. To remove the read permissions from both group and others, use the expression go –r:

```
$ ls -l xstart
-rwxr-xr-x   1 kumar    metal        1906 May 10 20:30 xstart
$ chmod go-r xstart ; ls -l xstart
-rwx--x--x   1 kumar    metal        1906 May 10 20:30 xstart
```

chmod also accepts multiple expressions delimited by commas.

For instance, to restore the original permissions to the file ***xstart,*** we have to remove the execute permission from all (a-x) and assign read permission to group and others (go+r):

```
$ chmod a-x,go+r xstart ; ls -l xstart
-rw-r--r--   1 kumar    metal        1906 May 10 20:30 xstart
```

More than one permission can also be set.

u+rw is a valid chmod expression. So setting write and execute permissions for others is no problem:

```
$ chmod o+wx xstart ; ls -l xstart
-rw-r--rwx   1 kumar    metal        1906 May 10 20:30 xstart
```

Absolute Permissions:

Using absolute permissions we can set all nine permission nits explicitly.

The expression used by **chmod** here is a string of three octal numbers (base 8).

Octal numbers use the base 8, and octal digits have the value 0 to 7.This means that a set of three bits can represent one octal digit.

If we represent the permissions of each category by one octal digit, this is how permissions can be represented.

- Read Permission - 4 (Octal 100)
- Write Permission - 2 (Octal 010)
- Execute Permission - 1 (Octal 001)

For each category we add up the numbers.

For instance, 6 represents read and write permissions and 7 represents all permissions as can easily be understood from the following table:

Binary	*Octal*	*Permissions*	*Significance*
000	0	---	No permissions
001	1	--x	Executable only
010	2	-w-	Writable only
011	3	-wx	Writable and executable
100	4	r--	Readable only
101	5	r-x	Readable and executable
110	6	rw-	Readable and writable
111	7	rwx	Readable, writable and executable

We have three categories and three permissions for each category, so three octal digits can describe a file's permissions completely.

The most significant digit represents user and the least one represents others. **chmod** can use this three-digit string as the expression.

We can use this method to assign read and write permission to all three categories.

```
$ chmod 666 xstart ; ls -l xstart
-rw-rw-rw-   1 kumar    metal        1906 May 10 20:30 xstart
```

The 6 indicates read and write permissions (4 + 2). To restore the original permissions to the file, we need to remove the write permission (2) from group and others.

```
$ chmod 644 xstart ; ls -l xstart
-rw-r--r--   1 kumar    metal        1906 May 10 20:30 xstart
```

To assign all permissions to the owner, read and write permissions to the group, and only execute permissions to the others, we can use this:

chmod 761 xstart

The expression 777 signifies all permissions for all categories, while 000 indicates absence of all permissions for all categories.

Using chmod Recursively (-R):

The –R option makes **chmod** descend a directory hierarchy and apply the expression to every file and subdirectory it finds.

chmod –R a+x shell_scripts

This makes all files and subdirectories found in the tree-walk (that commences from the shell_scripts directory) executable by all users.

We can provide multiple directory and filenames.

If we want to use **chmod** on our home directory tree, then "cd" to it and use it in one of these ways:

chmod –R 755 . *Works on hidden files also*

chmod –R a+x * *Leaves out hidden files*

Directory Permissions:

Directories also have their own permissions and the significance of these permissions differ from those of ordinary files.

The read and write access to an ordinary file are also influenced by the permissions of the directory housing them.

It is possible that a file cannot be accessed even though it has read permission, and can be removed even it is write protected.

Consider the following example:

$ mkdir c_progsl; ls –ld c_progs

drwxr-xr-x 2 kumar metal 512 May 9 09:57 c_progs

The default permissions of a directory in the above example are rwxr-xr-x (or 755).

A directory must never be writable by *group* and *others*.

Changing File Ownership:

There are two commands meant to change the ownership of a file or directory:

1. chown
2. chgrp

chown: Changing File owner:

The command is used in this way:

chown ***options owner [:group] file(s)***

chown (change owner) transfers ownership of a file to a user.
The chown command requires the user-id (UID) of the recipient, followed by one or more filenames.
Changing ownership requires superuser permission.

$ **su**
Password: ********** *This is the root Password*
_

After the password is successfully entered, **su** returns a # prompt.

To now change the ownership of the file note to *sharma*, we can use **chown** in the following way:

```
# ls -l note
-rwxr----x   1 kumar    metal        347 May 10 20:30 note
# chown sharma note ; ls -l note
-rwxr----x   1 sharma   metal        347 May 10 20:30 note
```

#exit
$ _

To change both owner and group, **chown** can be used as follows:

chown sharma:dba dept.lst

chgrp: Changing Group owner:

By default, the group owner of a file is the group to which the owner belongs.
The **chgrp** (change group) command changes a file's group owner.

In the following example, *kumar* changes the group ownership of **dept.lst** to **dba** (no superuser permission is required):

```
$ ls -l dept.lst
-rw-r--r--   1 kumar    metal        139 Jun  8 16:43 dept.lst
$ chgrp dba dept.lst ; ls -l dept.lst
-rw-r--r--   1 kumar    dba          139 Jun  8 16:43 dept.lst
```