MODULE 2. UNIX FILE SYSTEM

2.1 INTRODUCTION

File system is one of the important pillars of UNIX system. UNIX treats everything (even a user, program, directory etc) as a file. Hence, organizing and managing the file system is very important topic to study. Here, we will discuss how to create directories, moving around the file system, listing filenames, various file attributes, security related issues like file permission, changing the file permission etc.

2.2 THE FILE

The file is a container for storing information. Unlike old DOS files, a UNIX file does not contain eof (end-of-file) character. It contains only the information stored by the user. All the attributes of a file are kept in a separate area of the hard disk, which can be accessible by only the kernel. UNIX treats directories and devices also as files. Even the physical devices like hard disk, memory, CD-ROM, printer, modem etc. are treated as files.

In UNIX, files are divided into three categories –

- Ordinary file
- Directory file
- Device file

These three types of files are discussed in detail in the following sections –

2.2.1 Ordinary (Regular) File

It is a most common type of file that contains only data as a stream of characters. An ordinary file can be one among these –

- **Text file:** contains only printable characters. Source codes of programming languages like C, Java, C++, Perl, Shell script etc. are all text files. A text file contains lines of characters where every line is terminated with a *newline* character – known as *linefeed (LF)*. Whenever you press [Enter] key while inserting text into a file, the LF character is appended. One cannot see this character, but it can be made visible using the command ***od***.
- **Binary file:** contains both printable and non-printable characters covering entire ASCII (0 – 255) set. The object codes, executable files etc. created by compiling C language are binary files. Most of the UNIX commands are binary files. Image/audio/video files are binary files. Trying to display the contents of such files using simple *cat* command would produce unreadable output.

2.2.2 Directory File

A directory contains no data, but it keeps some information about the files and subdirectories that it contains. The UNIX file system is organized with a number of directories and subdirectories. A user also can create them, as and when required. Usually, a group of related files are kept in a single directory. Sometimes, files with same name are kept in different directories.

A directory file contains an entry for every file and subdirectory it has. Each such entry has two components viz. –

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

Thus, a directory actually do not *contain* the file itself, rather, it contains only the file name and a number.

One cannot write into a directory file. But, the actions like creating a file, removing a file etc. makes kernel to update the corresponding directory by creating/removing filename and inode number associated with that file.

2.2.3 Device File

The activities like printing files, installing softwares from CD-ROM, taking backup of files into a tape/drive etc. are performed by reading or writing the file representing the device. For example, when you are printing a file in a printer, you are writing a file associated with printer.

Device filenames are generally found inside a single directory structure, /dev. A device file is not a stream of characters. In fact, it does not contain anything. The operation of a device is completely managed by the attributes of its associated file. The kernel identifies a device from its attributes and then uses them to operate the device.

2.3 WHAT IS IN A FILE NAME?

In most of the UNIX systems now a days, a filename can consists of 255 characters. They can have any printable and non-printable characters except / and the NULL character. But, as UNIX uses special characters like $, ?, *, &, ` etc for different purpose, it is better to avoid certain characters. So, ideally, a filename can contain alphabets, digits and special characters dot (.), hyphen (-) and underscore (_).

UNIX does not impose any rule for framing the extensions for filenames. Even shell scripts do not require .sh as extension. It is used only for the convention. But, underlying programming languages like C requires extension. Hence, in UNIX, a filename can contain any number of dots – say, **a.b.c.d.e** is a valid filename in UNIX. A filename can begin/end with a dot. But, UNIX is case sensitive and same is maintained in naming the files as well. Thus, test, Test, TEST all are different files.

2.4 THE PARENT – CHILD RELATIONSHIP

All files in UNIX are related to each other. The file system in UNIX is a collection of all types (ordinary, directory and device files) related files organized in a hierarchical structure. A UNIX file system has *root* represented by /, which serves as reference point for all files. The *root* is actually a directory and it is different from the user-id root used by the system admin to log-in.

The root directory (/) has a number of subdirectories under it. These subdirectories in turn have more subdirectories and files under them. Figure 2.1 shows an example of UNIX file system tree structure. Here, *bin, dev* etc. are directories under root (/). And, *mthomas, stu1* are subdirectories under *home*.

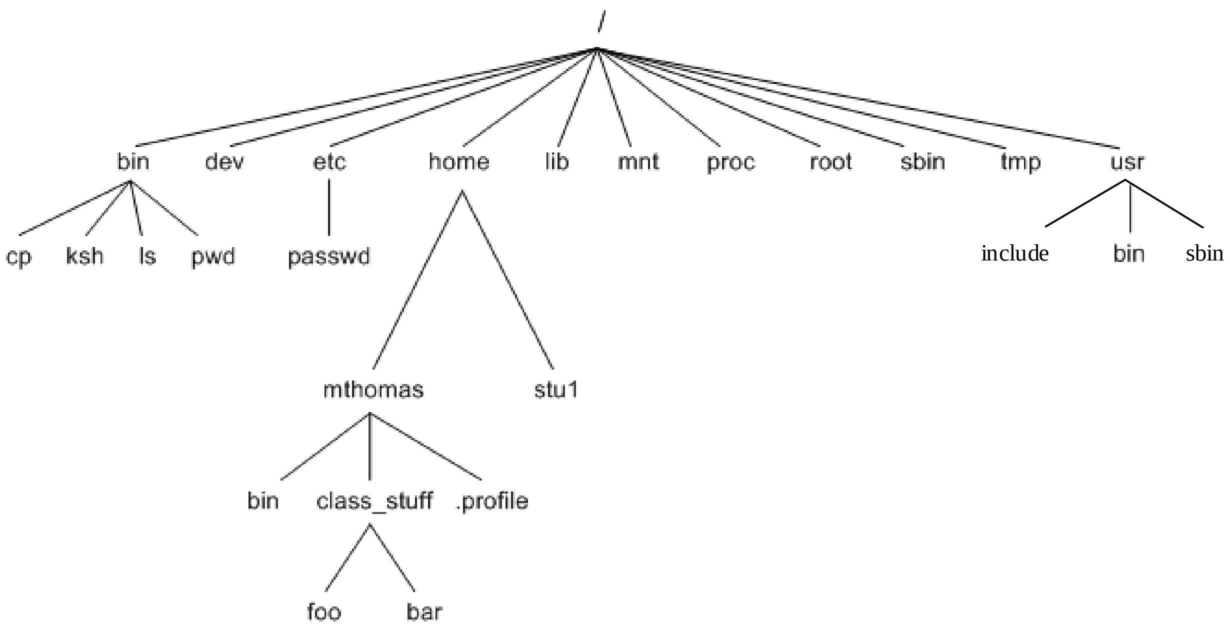

Figure 2.1 The UNIX File System Tree

Every file, apart from root, must have a parent, and there will be a parent-child relationship path from root to file. In the Figure 2.1, *cp* is child of *bin* and *bin* is child of /. That is, / is grandparent of *cp.* Note that, in a parent-child relationship, parent is always a directory.

2.5 THE *HOME* DIRECTORY

When a user logs into the system, UNIX places him into a directory called as ***home directory.*** It is created by the system when a user account is opened. If you have logged in with a user name *john,* then, your home directory would be */home/john*. This can be viewed using the shell variable $HOME as –

```
          $echo $HOME
          /home/john     # the first / represents root directory
```

The path displayed here is known as ***absolute pathname***, which is a sequence of all directory names separated by slash (/) starting from root. A file *foo* located in a home directory of the user can be referred as `$HOME/foo`. In some of the shells, it can be referred as `~/foo`. Here, the ~ symbol can be used to refer any other's file also. For example, if there is a file called *foo* in another user *richard*'s directory also, then it can be referred as `~richard/foo`.

Note that, a ~ (tilde) followed by / refers to one's own home directory, but when followed by a string (like ~richard), it refers to home directory represented by that string.

2.6 pwd: CHECKING YOUR CURRENT DIRECTORY

Once a user logs in to the UNIX system, it places him in a specific directory (usually home directory) of the file system. Though a user can move from one directory to other, for a given moment of time, he will be in one directory, known as ***current directory.*** To know current directory, the ***pwd*** (print working directory) command is used. The *pwd* command displays the absolute pathname as below –

```
$pwd
/home/john
```

One can navigate the file system using ***cd*** command.

2.7 cd: CHANGING THE CURRENT DIRECTORY

The ***cd*** command is used to move around the file system by changing the directory. This command can be used in three different ways –

- If the user *john* is in his home directory and would like to move to subdirectory called as *progs*, then the command should be given as –
```
$ cd progs     # user is moved to progs directory now
$ pwd          # verify this using pwd
/home/john/progs
```

- If the user would like to shift some other directory (but not his own subdirectory), then absolute path name can be given. For example,
```
$ pwd               #check current directory
/home/john/progs
$ cd /bin           # change directory to /bin
$ pwd               # verify new location
/bin
```

- When ***cd*** is used without argument, it will take the user back to his home directory, that is where he had logged into.

  Ex1.
```
$ pwd               #check current directory
/home/john/progs
$ cd                #no arguments given
$ pwd
/home/john          #back to home directory
```

  Ex2.
```
$cd /home/tony      #john moves to tony's home directory
$pwd                #verify
/home/tony
$cd                 #cd given without argument
$pwd                #john is back to his home directory
/home/john
```

NOTE that ***cd*** command may fail when you do not have permission to access a directory.

2.8 mkdir: MAKING DIRECTORIES

A directory can be created using the command ***mkdir***. Various ways of using this command is listed here.

- The name of the directory to be created has to be given as argument. For example,
```
$mkdir docs
```

- One can create more than one directory with a single ***mkdir*** command as below –
```
$mkdir docs progs db     # three directories created
```

- A directory tree can be created as –
```
$mkdir test test/prgms test/data
```

  Now, initially the directory *test* will be created. Then two subdirectories *prgms* and *data* will be created under *test.* Note that, while creating such directory tree, the parent directory name has to be given first.

- Sometimes, trying to create a directory may fail and error message may get displayed as –
```
$mkdir myDir
mkdir: Failed to make directory “myDir”; Permission denied
```

  The reason may be any of these –
  - The directory *myDir* may already exist
  - There may be any ordinary file by that name in the current directory
  - The permission is set for current directory so as to not to create new files/directories within it.

2.9 rmdir: REMOVING DIRECTORIES

To remove (or delete) a directory, the ***rmdir*** command is used. Few important points about this command are discussed here –

- A directory has to be empty before removing it. That is, it should not contain any files or subdirectories.
- To remove one directory, use statement like –
```
$rmdir test         #removes the directory test
```

- More than one directory (even in tree structure) can be removed at a time. For example,
```
$mkdir test/prgms test/data test
```

  Note that, here *prgms* and *data* are two subdirectories under *test*. They should be given first and then the parent directory *test* has to be mentioned.

- To remove a particular directory, you should be hierarchically above that directory. That is you cannot remove any directory that is above your current directory and also, you cannot remove your current directory as well.

- The commands ***mkdir*** and ***rmdir*** can work only in the directories owned by the user. But they cannot be implemented on the directories of some other user.

2.10 ABSOLUTE PATHNAMES

A path of a file (or directory) which specifies the complete hierarchy of that file starting from the root (/) directory is called as ***absolute pathname*** of that file (or directory). Most of the UNIX commands that take file or directory name as arguments will assume that the specified file exists in the current working directory. That is, if you use the command

```
    $cat test.sh
```

it is assumed that the file *test.sh* is in current directory. If you would like to access the file in some other directory, you have to give the command by specifying absolute pathname of that file as –

```
    $ cat /home/john/test.sh
```

Here, one can observe that the absolute pathname of a file starts with / (indication for root) and goes one level down for every appearance of / (as a separator). That is, in the above example, *test.sh* is at three levels down from root.

We know that, more than one files with a same name may exists in different directories of UNIX system. But, their absolute pathnames will be different. That is, no two files in the UNIX file system can have same absolute pathnames.

2.10.1 Using absolute pathnames for a command

All UNIX commands like ***date, cat, cal*** etc. are basically files. When we specify any command for execution, the system has to locate the file from the list of directories mentioned in the PATH variable, and then execute it. If we know the location of a particular file, we can use it for running that command. For example, ***date*** command resides in /bin (or /usr/bin). So, we can runt *date* command as –

```
         $ /bin/date
         Sun Oct 29 11:52:43 IST 2017
```

Executing UNIX commands like this will not gain anything, it is just an illustration. But, if user's program resides in some other directory and to be executed from somewhere else, then the absolute pathname will be helpful. For example, your current directory is `/home/john`. You would like to run a file *`test.sh`* which is in `/home/richard`. Then you use the command

```
         $ sh /home/richard/test.sh
```

A shell variable PATH includes the list of various pathnames – files in that can be executed directly. If you want to execute many files within a particular directory, that pathname can be included in the PATH variable. And each time, one need not use the complete absolute path for the execution purpose.

2.11 RELATIVE PATHNAMES

A relative path is defined as the path related to the present working directory. For example, if we are currently in `/home/john` and would like to move to `/home/john/test`, then we can just give the command as –

```
          $cd test       #this is relative path usage
```

We can also give the command as

```
          $ cd /home/john/test     #this is absolute path usage
```

So, one can say that, if the pathname starts with / (root), we can say that it is absolute path. Otherwise, it is relative path.

2.11.1 Using . and .. in Relative Pathnames

We have discussed in the previous section that one can navigate from one directory to other using ***cd*** command by specifying require absolute pathname. But, when we need to navigate only within *home*, we need not use absolute pathname always. We can use relative pathname as a shortcut. Here, the relative pathname uses either the current or parent directory as a reference and specifies the path relative to it. For this purpose, a relative pathname uses following cryptic symbols:

- A single dot (`.`) – represents current directory
- Two dots (`..`) – represents parent directory.

Consider an example now. Assume that we are currently placed in `/home/john/progs/data/text` and would like to move to `/home/john/progs/data` using ***cd*** command. We can use two dots (..) for this purpose as shown below.

```
     $pwd
     /home/john/progs/data/text
     $cd ..                          #observe space between cd and ..
     $pwd                            #check now
     /home/john/progs/data
```

Thus, the usage of `..` moves the directory one level up. If we want to move more than one level up, we can use `..` more than once separated by / as shown below –

```
     $pwd
     /home/john/progs/data/text
     $cd ../../..                    #moves 3 levels up
     $pwd
     /home/john
```

Any command which uses the current directory as argument can work with single dot. For example, to run a shell script (Refer the example Shell1.sh given in Section 1.7.2 of Module1), we can use `./`(dot with slash). Also, the single dot indicating current directory is useful in copying the files. For example, assume there is a file `shell1.sh` in the current directory. And, we want to copy it into directory `myDir` which is within current directory. Now we can use the command –

```
    $cp ./shell1.sh ./myDir
```

2.12 THE UNIX FILE SYSTEM

The structure of UNIX file system is discussed here. The Figure 2.1 given earlier depicts a sample of UNIX file system. The root (/) directory has many subdirectories under it, but only few are shown in the diagram. From system administrator point of view, the entire file system consists of two groups of files. The first group contains the files that are made available during system installation as given below –

- **/bin and /usr/bin:** Commonly used UNIX commands (binaries, hence the name bin) are found here. PATH variable always shows these directories in its list.
- **/sbin and /usr/sbin :** The commands which can be executed only by system administrator (but not by a normal user) are kept in this directory. The PATH variable of system administrator contains these directories.
- **/etc :** This directory contains the configuration files of the system. Modification of any text file in this directory may affect the functionality of the system. Login name and password of users will be stored in the files `/etc/passwd` and `/etc/shadow` respectively.
- **/dev :** This directory contains all device files. These files do not occupy space on the disk. There can be more subdirectories like `pts, dsk` and `rdsk` in this directory.
- **/lib and /usr/lib :** This directory contains all library files in binary form. If you want to run C program, you may need to link those programs with files in these directories.
- **/usr/include :** Contains standard header files like `stdio.h, stdlib.h` etc. used by C programs.
- **/usr/share/man :** All the man (manual) pages are stored here. There are separate subdirectories like `man1, man2` etc that contain the pages for each section.

The second group contains the files/programs/mails created by the users as shown below –

- **/tmp :** The directories where users are allowed to create temporary files. These files are removed regularly by the system.
- **/var:** It is a variable part of the file system. Contains all printable jobs and outgoing/incoming mails.
- **/home:** Home directories of all the users are stored here.

2.13 ls: LISTING DIRECTORY CONTENTS

We have discussed the command ***ls*** in previous chapter (Section 1.6.3 of Module 1), that it used to list the files and directories in the current working directory. This command is discussed here with more details and some of the options.

When ***ls*** command is used to display the files, one can observe that (in most of UNIX systems, but not always in Linux) the files will be displayed in alphabetical (rather ASCII) order. That is, files starting with numeric first, then uppercase letters and then lowercase letters. This is known as ***ASCII collating sequence.*** The files and directories are listed as a single column (older versions of ***ls*** do so) and, if you are using Linux systems, then they might be showed up in different colors.

To check whether a particular file exists in a directory or not, one can give the file name along with ***ls*** command. For example,

```
$ ls test      #checking whether the file test is there
test           #file name is displayed if it exists
```

Here, the file name is displayed if that file exists in that directory. Otherwise, an error message will be displayed as shown below –

```
$ ls test
test: No such file or directory
```

2.13.1 ls Options

The command ***ls*** has multiple options for various purposes. Some of them are discussed here.

- **Output in Multiple Columns (–x) :** When there are many files, it is better to display them in multiple columns. Modern versions of ***ls*** do that by default (without any options), but if it doesn't happen in your system, you can use **–x** option as –

```
$ ls -x
Thesis    Shell1.sh      Shell2.sh      ShellPgms
Emp.txt   cmd.c          helpdir
```

- **Identifying Directories and Executables (–F) :** The ***ls*** command displays files as well as directories. To know which of them are directories and executable files, one can use –F option. When –F option is combined with –x, it produces multicolor output.

```
$ ls -Fx
Thesis/   Shell1.sh*     Shell2.sh*     ShellPgms/
Emp.txt   cmd.c          helpdir/
```

Here, the symbols * and / are type indicators. The * indicates that the file contains executable code, and / refers to a directory.

- **Showing Hidden Files Also (–a):** If we want to see hidden files also, we use –a (all) option for ***ls.*** There are certain hidden files starting with a dot, which normally don't get displayed with just ***ls*** command.

```
$ ls -a
.    ..        .exrc     Thesis
.emacs         .gnome2   Shell1.sh
```

Note that, the first two files displayed are `.` and `..` indicating current and parent directories.

- **Listing specific directory contents:** If you want to display the contents of only specific subdirectories, you can give the name along with ***ls*** as shown below –

```
$ ls ACMPaper ShellPgms
ACMPaper:
acm.aux        acm.bib   acm.pdf   runtex
acm.tex        sig.pdf

ShellPgms:
caseEx.sh      first.sh  menu.sh   test
hello.c
```

In the above example, `ACMPaper` and `ShellPgms` are directory names. When they are given along with ***ls*** command, the files within them are displayed separately.

- **Recursive Listing (–R):** This option lists all files and subdirectories in a directory tree. That is, contents of subdirectories also will be displayed recursively till there is no subdirectory is left out.

```
$ ls –R
.:
Thesis    Shell1.sh      Shell2.sh      ShellPgms
Emp.txt   cmd.c

./Thesis:
Chap1.aux      Chap1.bib      Chap1.tex      Chap1.pdf
Annex.aux      Annex.pdf

./ShellPgms:
caseEx.sh      first.sh       menu.sh   test
hello.c

./ShellPgms/helpdir:
Test.c         here.sh        try.sh
```

One can observe that, the –R option starts display with the current directory (`.`). Then it displays the contents of all subdirectories under current directory. Later it goes one level down and so on.

2.13.2 ls –l option: Listing File Attributes

The –l option of ***ls*** command is used for listing the various attributes like permissions, size, ownership etc. of a file. The output of ***ls –l*** is referred to as the ***listing***. The –l option can be combined with other options for displaying other attributes, or ordering the list in a different sequence. The command ***ls*** use inode of a file to fetch its attributes. Consider the following example of ***ls –l*** which displays seven attributes of all files in the current directory.

```
$ ls -l
total 144
-rw-rw-r-- 1 john john  280 Jan 30 09:56 caseEx.sh
-rw-rw-r-- 1 john john  104 Feb  3 06:40 cmdArg.sh
```

```
-rw-rw-r-- 1 john john  199 Jan 29 10:58 ifEx.sh
-rw-rw-r-- 1 john john  217 Jan 19 09:25 logfile
drwxrwxr-x 2 john john 4096 Feb  6 05:48 myDir
-rwxrwxr-x 1 john john   29 Jan 22 10:04 myFirstShell
-rw-rw-r-- 1 john john   43 Jan 22 10:44 second.sh
```

The output of ***ls –l*** starts with `total 144`, indicates that a total of 144 blocks are occupied by these files on disk. The 7 types of attributes/fields displayed by the command are discussed below –

- **File Type and Permissions:** The first column shows the type and permissions associated with each file. If the first character in this column is – (hypen), then it is an ordinary file. On the other hand, if the first character is d, then it is a directory. Then, there is a series of r, w, x and – (hyphen) indicating file permissions read, write and execute. The hyphen indicates absence of particular permission.

- **Links:** The second column indicates the number of links associated with the file. It is a number of filenames maintained by the system of that file. Usually for ordinary files, it will be just 1. But for directories, it will be number of files contained within that directory (including current directory).

- **Ownership:** The creator of the file would be its owner. In the third column, it shows `john` as the owner. The owner has full authority to modify the contents and permissions of a file. Similarly, the owner of a directory can create modify or remove files in that directory.

- **Group Ownership:** While opening a user account, a system administrator assigns the user to some group. The fourth column represents the group owner of that file.

- **File Size:** Size of the file in bytes is shown as fifth column. The size is only a character count of the file, but not the amount of space it occupies in the disk.

- **Last Modification Time:** The 6th, 7th and 8th columns indicate the last modification time of the file. A file is said to be modified only if its contents have changed. If you change only the permission or ownership of the file, its last modification time field will not get affected.

- **Filename:** The last field is the name of the file, usually in ASCII collating sequence.

2.13.3 ls –d option: Listing Directory Attributes

If we want to list the attributes of only the directory, but not its contents, we can use –d option as below –

```
$ ls –ld myDir
drwxrwxr-x 2 john john 4096 Feb  6 05:48 myDir
```

2.14 FILE OWNERSHIP

The person who creates a file will be the owner of that file. The login name of that person will be showed as the owner when ***ls –l*** option is used. The group to which the person belongs to, will be the group owner of the file. If you copy someone's file, you will be the owner only for that copy. One cannot create files in other's home directory, because one may not have permission to do so.

Several users may belong to a single group. The files created by group members will have the same group owner. However, the privileges of the group are set by the owner of the file, but not by group members.

When the system administrator creates a user account, he has to assign the following 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 name and number) and GID (only the number). The file `/etc/group` contains the GID (both name and number). To know your UID and GID, the ***id*** command can be used as –

```
$id
uid=821(chetana) gid=822(STAFF)
```

2.15 FILE PERMISSIONS

UNIX has a simple and well-defined system of assigning permissions to files. UNIX follows three-tiered file protection system to determine a file's access rights. Consider the ***ls –l*** command to view the permissions of few files:

```
$ ls –l chap02  dept.lst  shell1.sh
-rwxr-xr--    1 john richard   20500    Sep 29 11:53  chap02
-rwxr-xr-x    1 john richard     850    Oct 02 10:12  dept.lst
-rw-rw-rw-    1 john richard      48    Nov 07 08:03  shell1.sh
```

Observe the first column representing permission for the file chap02. Here, the first character says whether the file is ordinary file or directory. So, leaving it apart, consider next 9 characters as a group of 3 characters each –

```
rwx  r-x  r--
```

Each group represents a ***category*** viz. ***owner, group owner*** and ***others (or world)*** respectively. Every group contains any of the characters ***r, w, x*** and **-**. The meaning of these is as below –

- **r:** indicates read permission – means, ***cat*** command can display the file
- **w:** indicates write permission – file can be edited with an editor
- **x:** indicates execute permission – the file can be executed as a program
- **-:** indicates absence of the corresponding permission

Usually, the owner of a file will have all the three permissions. In the above example, the group owner of the file *chap02* has only read and execute permissions. The public or others or world has only read permission.

Note that the owner of the file is also termed as ***user***.

2.16 chmod: CHANGING FILE PERMISSIONS

A file or directory is created with a default set of permissions. Generally, in the default setting, write permission is not given to group and others. That is, only the user (owner) the file can write a file. But, read permission will be given to all. The ***chmod*** (change mode) command is used for assigning/removing different permissions to/from *category* (user, group, others). This command can be run only by the user (owner) and the super-user (admin). The ***chmod*** 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

Consider the permissions of an existing file *test* as below –

```
$ls –l test
-rw-r--r--   1 john richard   853  Sep 5 23:38 test
```

It is observed here that, by default the execute permission is not there even for the user (owner). Keeping this status of the file *test* as a base, let us discuss different ways of using ***chmod*** command.

2.16.1 Relative Permissions

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

`chmod` *`category operation permission filenames`*

The argument for ***chmod*** is an expression consisting of some letters and symbols describing user category and type of permission being assigned/removed. The expression contains three components:

- User category (user: **u**, group: **g**, others: **o**, All: **a**)
- The operation to be performed (assign: **+**, remove: **–**, assign absolute permission: **=**)
- The type of permission (read: **r**, write: **w**, execute: **x**)

Now, consider the example of the file *test* taken before, and assign execute permission to it as below –

```
$ chmod u+x test         #assign(+) x(execute) to u(user)
$ ls –l test
-rwxr--r--   1 john richard   853  Sep 5 23:38 test
```

Now, the user *john* got permission to execute the file *test.* If you want to assign execute permission on file *test* to group and others also, then use the command as –

```
$ chmod ugo+x test   #assign(+) x to u(user, group, others)
$ ls –l test
-rwxr-xr-x   1 john richard   853  Sep 5 23:38 test
```

The string ugo can be replaced by *a* indicating *all* as shown below –

```
$ chmod a+x test     #assign(+) x to a(all)
$ ls –l test
-rwxr-xr-x   1 john richard   853  Sep 5 23:38 test
```

When you are willing to assign a particular permission to *all*, then even the character *a* can be omitted as below –

```
$ chmod +x test      #assign(+) x to all
```

When same set of permissions has to be assigned to more than one file, then we can give multiple files separated with space as –

```
$ chmod u+x test  test1  test2
```

To remove a permission, the – (hyphen or minus) operator is used. For example, to remove read permission from group and other, we can do as below –

```
$ ls –l test        #check current status
-rwxr-xr-x   1 john richard   853  Sep 5 23:38 test

$ chmod go-r test   #remove r permission from group & others

$ ls –l test        #verify
-rwx—x--x   1  john richard   853  Sep 5 23:38 test
```

Multiple expressions separated by comma can be given to ***chmod*** command. For example, to remove the execute permission from all and then to assign read permission to group and others, a single statement can be used as –

```
$ chmod a-x, go+r test
$ ls –l test
-rw-r--r--   1 john richard   853  Sep 5 23:38 test
```

More than one permission can also be set as below –

```
$chmod o+wx test
$ ls –l test
-rw-r--rwx   1 john richard   853  Sep 5 23:38 test
```

Here, write and execute permissions are set to *others*.

2.16.2 Absolute Permissions

Irrespective of existing permissions for a file, we may need to assign a new set of permissions. That is, we wish to set all nine permission bits explicitly. This is known as *absolute permissions*. For this purpose, ***chmod*** uses a string of three octal numbers. Various permissions are given a specific digit as below –

- Read permission – 4
- Write permission – 2
- Execute permission – 1

Every possible combination of three different permissions is shown in binary representation in Table 2.1.

Table 2.1 Digits used for absolute pathnames

Binary	Octal	Permission	Significance
000	0	- - -	No permission
001	1	- - x	Execute only
010	2	- w -	Write only
011	3	- wx	Write and execute
100	4	r- -	Read only
101	5	r – x	Read and execute
110	6	rw-	Read and write
111	7	rwx	Read, write and execute

Now, let us see some examples of using the absolute permissions with the help of octal digits.

Ex 1. Assigning read and write(4+2=6) permissions to all –

```
$ chmod 666 test
$ ls –l test
-rw-rw-rw-   1 john richard   853  Sep 5 23:38 test
```

Ex 2. To remove the write permission from group and others:

```
$ chmod 644 test
$ ls –l test
-rw-r--r--   1 john richard   853  Sep 5 23:38 test
```

Note that, there is nothing like removing some permission. It is just reassignment of new set of permissions to all.

Ex 3. To assign all permissions to owner, read and write permissions to group and only execute permission to others –

```
$ chmod 761 test
$ ls –l test
-rwxrw---x   1 john richard   853  Sep 5 23:38 test
```

2.16.3 Using chmod Recursively (–R)

The *chmod* command can be used to apply required permissions on all files (and files within subdirectory) in a given directory. It is done using –R option as below –

```
    $chmod –R a+x ShellPgms
```

This makes all files and subdirectories found in the tree-walk (starting from `ShellPgms` directory, includes all files in subdirectories) executable by all users. One can provide multiple directory and filenames for this purpose. If *chmod* has to be applied on home directory tree, one can use any one of the following –

```
    $chmod –R 755 .          #works on hidden files also
    $chmod –R a+x *          #leaves out hidden files
```

2.17 CHANGING FILE OWNERSHIP

It has been discussed that the creator of a file will be the owner. And the group to which user belongs to, will be the group owner. But, except the owner, other group members cannot alter the permissions of any file. UNIX provides two commands for changing the ownership of a file or directory viz. ***chown*** and ***chgrp***. The usage of these commands differs from system to system. On BSD – based systems, only the system administrator can change the ownership of a file using ***chown***. On the other systems, only the owner can use *chown* and *chgrp.*

(NOTE: Berkeley Software Distribution (**BSD**) was a Unix operating system derivative developed and distributed by the Computer Systems Research Group (CSRG) of the University of California, Berkeley, from 1977 to 1995.)

2.17.1 chown: Changing File Owner

The syntax of ***chown*** command in BSD-based systems is –

```
    chown options owner [:group] files
```

To use ***chown*** command in BSD-based systems, we need the super-user permission. For that, the ***su*** command is used as below –

```
    $su
    Password: ********           (this is root password)
    #_                           (This is another shell)
```

The command *su* lets us to acquire superuser status (Note that # is the prompt for admin). Now, try to change the ownership of a file *note* which is currently owned by *john* as below–

```
    # ls –l note
    -rwxr----x    1    john   metal   347  May 10 20:30   note
    # chown ricky note
    # ls –l note
    -rwxr----x    1    ricky  metal   347  May 10 20:30   note
```

Here, the ownership of the file *note* has been changed from *john* to *ricky.* The file permissions previously assigned to *john* will now be of *ricky*. Now onwards, *john* is not the owner of this file and he cannot read/write this file.

2.17.2 chgrp: Changing Group Owner

By default, the group owner of a file is the group to which the owner belongs to. But, the ***chgrp*** command changes a file's group owner. A user can be a member of more than one group. So, in BSD – based systems, group can be changed only among the set of groups for which the user is a member.

Assume that *john* is a member of two groups *metal* and *dba*. And he has created a file *dept.txt* in *metal* group. He can change the group owner as below –

```
    $ls –l dept.txt
    -rw-r--r--     1    john  metal  129  Jun 8 16:42  dept.txt
    $chgrp dba dept.txt
    -rw-r--r--     1    john  dba    129  Jun 8 16:42  dept.txt
```

When the user is not a member of particular group, he cannot change the group owner of any file to that group. Only superuser can do so.

2.18 SECURITY AND FILE PERMISSIONS

There are certain security issues while using ***chmod*** command. Consider removing all the permissions for a file as below –

```
    $chmod 000 test
    $ls –l test
    ---------- 1 john richard  830 May 10 20.30 test
```

Now the file is virtually useless, as no one can do anything with it. But, user can still delete this file. At the same time, one must be aware that giving all permissions to everyone is dangerous. That is, having the following statement makes the file readable/writable/executable for everyone.

```
    $chmod 777 test
```

So, anyone can modify the contents of the file and it is a threat on security.

2.19 MORE FILE ATTRIBUTES

Every file is associated with a table that contains various attributes of a it, except its name and contents. This table is called as ***inode*** (index node) and accessed by the ***inode number***. The ***inode*** contains following attributes of a file –

- File type (ordinary, directory, device etc)
- File permissions
- Number of links
- The UID of the owner
- The GID of the group owner
- File size in bytes
- Date and time of last modification
- Date and time of last access
- Date and time of last change of the inode
- An array of pointer that keep track of all disk blocks used by the file

Observe that neither the name of the file nor the inode number is stored in the inode. The directory stores these details along with the filename. When you use any command with filename as an argument, the kernel first locates the inode number of the file from the directory and then reads the inode to fetch data relevant to the file.

The ***ls*** command uses inode to fetch the attributes of a file. The ***–i*** option with ***ls*** command can be used to check inode number of a file.

```
$ls –il test
9059 –rw-r--r--     1 john    Richard   839  Jan 31 11:23 test
```

The first entry in the above output is the inode number of the file *test*. No other file in the same file system can have this inode number. Once the file is removed, the inode number can be assigned to some other file.

2.20 HARDLINKS

We have seen in the previous section that, the inode table does not contain the name of a file. Reason is – a file can have multiple names. In that case, we say that the file has more than one ***link***. Multiple names provided to one single file are all having same inode number. The link count is displayed when ***ls –l*** is used. It is usually one.

A link can be created to a file using ***ln*** command. The following command is used to create hard link for an existing file *emp.lst* with a non-existing file *employee*:

```
$ln emp.lst employee
$ ls –li emp.lst employee
29314 –rwxr-xr-x 2 john metal 915 May 5 03:34 emp.lst
29314 lrwxr-xr-x 2 john metal 915 May 5 03.34 employee
```

Observe that both the files `emp.lst` and `employee` have same inode number, and the link count is 2. One can link one more file say, `emp.dat` as below –

```
$ln emp.lst emp.dat
$ ls –li emp*
29314 –rwxr-xr-x 3 john metal 915 May 5 03:34 emp.lst
29314 -rwxr-xr-x 3 john metal 915 May 5 03.34 employee
29314 -rwxr-xr-x 3 john metal 915 May 5 03.34 emp.dat
```

Now the link count is 3.

NOTE:

1. Links provide protection against accidental deletion, especially when they exist in different directories. Assume that a file `emp.lst` is in `/home/data` directory and you have created a link for it in `/home/imp_files` directory. The deletion of the file in one directory will not remove the file in the other directory and hence, you can always get the file back.

2. Having multiple names for a single file is **NOT** same as having multiple copies of single file. When a copy of one file is created, those two copies will have different inode numbers, and their link count will be one. But, when a file is linked, then all such files will have unique inode numbers and their link count will be more than one.

2.21 SYMBOLIC LINK

The hard links discussed in the previous section has certain limitations –

- One cannot have two linked filenames in two file systems. That is, one cannot link a filename in the `/usr` file system to another in `/home` file system.
- One cannot link a directory even within the same file system.

The ***symbolic links*** or ***soft links*** will overcome these limitations. The symbolic link can be thought of as a fourth type of a file (apart from 3 types discussed till now – ordinary, directory and device). Unlike the hard link, a symbolic link doesn't have the file's contents. But, it simply provides the pathname of the file that actually has the contents. Shortcut keys in windows are the best examples for symbolic links.

The ***ln*** command with ***–s*** option is used to create symbolic link as below –

```
$ ln –s note note.sym
$ ls –li note note.sym
9948 –rw-r--r-- 1 john metal 915 May 5 03:34 note
9952 lrwxrwxrwx 1 john metal   4 May 5 03:34 note.sym->note
```

Compared to hard links, one can find following differences in the listing of symbolic link file–

- Original file and symbolic linked file have different inode numbers.
- File type of `note.sym` shows **l** (el) indicating it is not an ordinary file, but a symbolic link file.
- Size of the file `note.sym` is just 4 – which is the length of the pathname it contains (`note`).
- The pointer notation at the end `note.sym->note` indicates that `note.sym` contains the pathname for the filename `note.`

When we use ***cat*** command on `note.sym`, we are not opening the symbolic link file, but the original `note` file. Removing `note.sym` will not affect much, because we can always create a link again. But, if we remove `note`, we would lose the file containing data. In this case, `note.sym` would point to a non-existent file and become a ***dangling*** symbolic link.

2.22 umask: DEFAULT FILE AND DIRECTORY PERMISSIONS

When you create files and directories, the permissions assigned to them depend on the default setting of the system. The UNIX system has the following default permissions for all files and directories –

- `rw-rw-rw-` (octal 666) for ordinary files
- `rwxrwxrwx` (octal 777) for directories

This default setting is transformed by subtracting the ***user mask*** from it to remove one or more permissions. To understand this, we should know the current value of mask by using ***umask*** command without arguments as –

```
$umask
0022
```

This is an octal number which has to be subtracted from the system default to obtain the actual default. Hence, the actual default of files will be –

- 666 – 022 = 644 for ordinary files
- 777 – 022 = 755 for directories

Hence, when we create a new file on this system, the default permission of it would be –

```
rw-r--r--
```

The ***umask*** is a shell built-in command, though it exists as an external command. A user can use this command to set a new default. For example,

```
$umask 000
```

The above command sets the ***umask*** to 000 and hence, any new file created will have the permissions as 666-000 = 666 permission. That is, it would be rw-rw-rw-, which is dangerous because anyone can write the file.

Similarly, if you set

```
$umask 666
```

Then, all files created will have permission as 666-666 =000, and it will be a useless file. So, the mask has to be set carefully.

2.23 find: LOCATING FILES

The ***find*** command is one of the powerful tools of the UNIX system. It recursively examines a directory tree to search for files matching some criteria, and then takes some action on the selected files. The syntax of this command is –

```
find path_list selection_criteria action
```

The working of ***find*** command is as below –

- Initially, it recursively examines all files in the directories specified in *`path_list`*
- Then it matches each file for one or more *`selection_criteria`*
- Finally, it takes some *`action`* on those selected files.

Consider an example to understand working of ***find*** command –

```
$ find /home/john -name "*.sh" -print
/home/john/Shell2.sh
/home/john/caseEx.sh
/home/john/cmdArg.sh
```

```
/home/john/test.sh
/home/john/test2.sh
/home/john/ifEx.sh
/home/john/test1.sh
/home/john/Shell1.sh
/home/john/myDir/Shell1.sh
/home/john/second.sh
```

In the above example, *`path_list`* to be searched for is `/home/john`. The *`selection_criteria`* is –

```
-name "*.sh"
```

The criteria for selection has certain options (read UNIX manual page to know more options), each one starting with – (hyphen). In the above example, it means that name of the files having pattern `*.sh` have to be searched. That is, all files with `.sh` as extension have to be searched in the given path. Then, the *`action`* to be taken is just `–print`. Hence, all the files with extension `.sh` are printed in the given path `/home/john`.