THE FILE SYSTEM & FILE ATTRIBUTES AND PERMISSIONS

The File system –The Basics of Files-What's in a File-Directories and File Names-Permissions-I Nodes-The Directory Hierarchy, File Attributes and Permissions-The File Command knowing the File Type-The Chmod Command Changing File Permissions-The Chown Command Changing the Owner of a File-The Chgrp Command Changing the Group of a File.

2.1The basics of flies

A file is a sequence of bytes. No structure is imposed on a file by the system, and no meaning is attached to its contents — the meaning of the bytes depends solely on the programs that interpret the file.

create a small file

```
$ cat> j
now is the time
for all good people
$
```

```
$ ls -1 j
-rw-r—r— 1 cse501 36 Sep 27 06:11 j
$
```

j is a file with 36 bytes — the 36 characters you typed while appending. To see the file,

```
$ cat j
now is the time
for all good people
$
```

cat shows what the file looks like.

The command **od (octal dump)** prints a visible representation of all the bytes of a file:

```
$ od -c j
0000000   n  o     w     i     s     t     h     e     t     i     m     e    \n
0000020   f  o     r     a     l     l     g     o     o     d     p     e     o
0000040   p  1     e    \n
0000044
$
```

The -c option means "interpret bytes as characters."

The –b option will show the bytes as octal (base 8) numbers.

The -x option tells od to print in hex.

```
$ od -cb j
0000000   n  o     w           i  s           t     h     e           t     i     m     e          \n
          156  157  167  040 151 163 040 164  150   145 040 164  151   155   145       012
0000020   f  o     r           a     l     l           g     o     o     d           p     e     o
          146 157 162 040 141 154 154 040 147 157 157 144 040 160  145 157
0000040   p  1     e          \n
          160  154   145     012
0000044
$
```

The 7-digit numbers down the left side are positions in the file, that is, ordinal number of the next character shown, in octal.

Notice that there is a character after each line, with octal value 012. This is the ASCII newline character. Other characters associated with some terminal control operation include backspace (octal value 010, printed as \b), tab (011, \t), and carriage return (015, \r).

$ stty -tabs

causes tabs to be replaced by spaces when printed on your terminal.

cat normally saves up or buffers its output to write in large chunks for efficiency, but cat -u "unbuffers" the output, so it is printed immediately as it is read:

```
$ cat
123
456
ctl-d
123
456
$ cat -u
123
123
456
456
ctl-d
$
```

cat receives each line when you press RETURN; without buffering, it prints the data as it is received.

Type some characters and then a ctl-d rather than a RETURN:

$ cat -u

123ctl-d123

Now type a second ctl-d, with no other characters:

$ cat -u

123ctl- d 123ctl-d$

The shell responds with a prompt, because cat read no characters, decided that meant end of file, and stopped, ctl-d sends whatever you have typed to the program that is reading from the terminal.

2.2 What's in a file?

The format of a file is determined by the programs that use it; there is a wide variety of file types. But file types are not determined by the file system, the kernel can't tell you the type of a file: it doesn't know it. The **file command** makes an educated guess:

```
$ file /bin   /bin/ed    /usr/src/cmd/ed . c    /usr/man/manl/ed . 1
/bin: directory
/bin/ed: pure executable
/usr/src/cmd/ed . c : c program text
/usr/man/manl/ed . 1 : roff, nroff, or eqn input text
$
```

These are four fairly typical files, all related to the editor: the directory in which it resides (/bin), the "binary" or runnable program itself (/bin/ed), the "source" or C statements that define the program (/usr/src/cmd/ed . c) and the manual page (/usr/man/manl/ed. 1).

To determine the types, file didn't pay attention to the names (although it could have), because naming conventions are just conventions, and thus not perfectly reliable. For example, files suffixed .c are almost always C source, but there is nothing to prevent you from creating a . c file with arbitrary contents.

Instead, file reads the first few hundred bytes of a file and looks for clues to the file type. Sometimes the clues are obvious. A runnable program is marked by a binary "magic number" at its beginning, od with no options dumps the file in 16-bit, or 2-byte, words and makes the magic number visible:

```
$ od /bin/ed
0000000    000410    025000    000462    011444    000000    000000    000000
000001
0000020    170011    016600    000002    005060    177776    010600    162706
000004
0000040    016616    000004    005720    010066    000002    005720    001376
020076
$
```

The octal value 410 marks a pure executable program, one for which the executing code may be shared by several processes. (Specific magic numbers are system dependent.) The bit pattern represented by 410 is not ASCII text, so this value could not be created inadvertently by a program like an editor. But you could certainly create such a file by running a program of your own, and the system understands the convention that such files are program binaries.

For text files, the clues may be deeper in the file, so file looks for words like #include to identify C source, or lines beginning with a period to identify nroff or troff input.

2.3 Directories and filenames

Each running program, that is, each process, has a current directory , and all filenames are implicitly assumed to start with the name of that directory, unless they begin directly with a slash. Your login shell, and **ls**, therefore have a current directory.

The command pwd (print working directory) identifies the current directory:

```
$ pwd
/usr/you
$
```

1. Directory related commands:

1. *Pwd (Print Working Directory)*

Name: pwd

Purpose: It displays the name of current working directory(where the user is working)

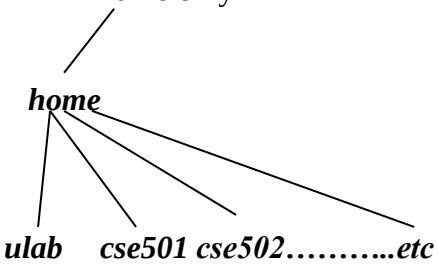
in a directory structure. *Pwd* always reports the full path of your current directory.

Examples & Description:

```
$pwd
/home/cse501
```

- The pwd command displayed a path name */home/cse501 (i.e)* the user is working with the directory *cse501* which is a subdirectory of *home.* Again *home* is a subdirectory of *root.*
- Generally all the users of UNIX will be placed under either *home* or *usr* directory

root By default the system places each user under their login name only



2. mkdir(Make directory)

Name: mkdir

Purpose: This command is used to create a new directory.

Syntax: *mkdir [option]* directory

Options:

Option	Description
-m mode	Set permission mode
-p	No error if existing, make parent directories as needed.
-v	Print a message for each created directory

Examples & Description:

$mkdir d1 **(**here *mkdir* creates a directory *d1* under the directory cse501)

$mkdir d2 d3 d4 **(** With a single *mkdir* command we can create more no.of directories.)

$mkdir d5 d5/d6 d5/d7 **(**We can create a directory tree with one *mkdir* command.)

The above command creates directory *d5* under *d5* it creates directories *d6, d7*

While creating the directory trees the order should be followed. i.e first parent has to be created then children

Examples & Description:

-p(creates Parent): this option creates parent directory as well as sub directories.

```
$mkdir –p x/y/z
```

creates first the parent directory ***x*** **,** inside ***x*** it creates ***y*** inside ***y*** it creates ***z***

-m mode: Each directory has permission modes(rwxrwxrwx). When a directory is created the default permissions are assigned as rwxrwxr_x by the kernel.

- The user can set his own permissions while creating the directory using –m mode option.
  Ex:

```
$mkdir –m 700 di
$ls –l
d rwx------ 0 cse501 cse501 512 Nov 19:14:17: di
```

The directory *di* has all permissions(read,write,execute) for owner. No permissions for group and other users.

3. Name: rmdir(Remove directory)

Name: rmdir

Purpose: Deletes a directory.

Syntax: rmdir [OPTION(s)] DIRECTORIE(S)

Option	Description
-p	--parents Remove DIRECTORY and its ancestors. E.g., `rmdir -p a/b/c' is similar to `rmdir a/b/c a/b a'.

Examples & Description:

$rmdir d1 (It removes directory *d1*.)
$rmdir d3,d2 (It removes directory *d2, d3* at a time.)
$rmdir d5/d6 d5/d7 d5 (It removes *d6 d7* which are subdirectories of *d5* and also removes *d5* finally).

Note1: We can't delete a directory unless it is empty. So executing the above command in the following way displays error message.

```
$rmdir d5 d5/d6 d5/d7
rmdir:d5: directory not empty
```

4.Name: cd (change directory)

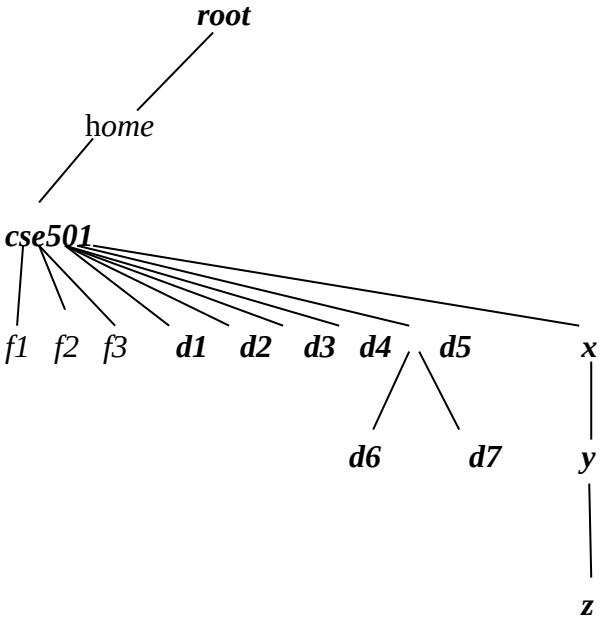
Name: cd

Purpose: cd is a command commonly used to change the directory

Syntax: cd pathname

There are no options for cd command.

Examples: consider the below directory structure for the following examples.



```
$pwd
/home/cse501
```

Here the user is working in *cse501* directory. Suppose the user wants to move to directory *d1* then cd command is used as

```
$cd d1
$pwd
/home/cse501/d1
```

From ***d1*** if the user wants to move to ***x*** then first the user has move to its parent(***cse501***) from there move to ***x***.

```
$cd ../x
$pwd
/home/cse501/x
```

cd Without arguments:

previously with whatever directory may be we are working, a simple ***cd*** command without arguments moves directly to the directory where the user originally logged onto.

```
$pwd
/home/cse501/x/y/z
$cd
$pwd
/home/cse501
```

2.du(Disk Usage):-

The command **du (disc usage)** was written to tell how much disc space is consumed by the files in a directory, including all its subdirectories.

Syntax:- du [options] [file(s)]

Examples:

ex: The 'du' command without any option (or) file name gives the output as:

```
$du
4    ./d1
2    ./d1/d2
7    ./d2
2    ./d2/d3
3    ./d2/d4
```

Here the output is for every sub-directory of current working directory the usage of disk in terms of BLOCKS is displayed. It also gives summary report for the usage of disk space for all directories under any one directory.

ex:- We can get the disk usage for a specified directory as follows

```
$du /home/sales/tml
10      /home/sales/tml/form
12      /home/sales/tml/data
40      /home/sales/tml
```

here it gives the usage of each subdirectory under a specified directory & it also gives the summary report for the disk usage of a specified directory.

Options:-

- -s :For showing only the summary report of the required directory(if we mentioned).

otherwise it gives the usage of a whole root directory.

```
$du  -s  /home/sales/tml
 40       /home/sales/tml
```

- -a : It gives the disk usage for files also.(x1,x2,x3 are files)

```
$du  -a
2      ./ d1/x1
3      ./d1/x2
5      ./d1
3      ./d2/x3
3      ./d2
16      /
```

- -b : It shows the disk usage in bytes rather than blocks.

```
$du  -b  /home/tm
11514      /home/tm/d1
12820      /home/tm/d2
75190      /home/tm
```

Here we get the disk usage in terms of bytes for sub directories d1&d2 and so on & finally for

Despite their fundamental properties inside the kernel, directories sit in the file system as ordinary files. They can be read as ordinary files. But they can't be created or written as ordinary files — to preserve its sanity and the users' files, the kernel reserves to itself all control over the contents of directories.

The time has come to look at the bytes in a directory:

```
$ od -cb .
0000000    4    ;    .   \0   \0   \0   \0   \0   \0   \0   \0   \0   \0   \0   \0   \0
           064  073  056 000  000  000  000  000  000  000  000  000  000  000 000  000
0000020    273  (    .   .   \0   \0   \0   \0   \0   \0   \0   \0   \0   \0   \0   \0
           273 050 056 056 000  000  000  000  000  000  000  000  000  000  000  000
0000040    252  d    1   d   2    \0   \0   \0   \0   \0   \0  \0
           252 144  162 144 143  000  000 000 000 000 000 000
0000060    230  =    j   \0   \0   \0   \0   \0   \0   \0   \0   \0   \0
           230  075 152  000 000 000 000 000 000 000 000 000 000
0000100
$
```

See the filenames buried in there? The directory format is a combination of binary and textual data. A directory consists of 16-byte chunks, the last 14 bytes of which hold the filename, padded with ASCII NUL's (which have value 0) and the first two of which tell the system where the administrative information for the file resides — we'll come back to that. Every directory begins with the two entries ("dot") and .' ("dot-dot").

2.4 Permissions

There is a special user on every UNIX system, called the super-user , who can read or modify any file on the system. The special login name root carries super-user privileges; it is used by system administrators when they do system maintenance. There is also a command called **su** that grants super-user status if you know the root password. Thus anyone who knows the super-user password can read your files,.

If you need more privacy, you can change the data in a file so that even the super-user cannot read (or at least understand) it, using the **crypt** command (crypt(l)). Of course, even crypt isn't perfectly secure. A super-user can change the **crypt** command itself, and there are cryptographic attacks on the crypt algorithm.

The system actually recognizes you by a number, called your user-id, or uid. In fact different login-id's may have the same uid, making them indistinguishable to the system, although that is relatively rare and perhaps undesirable for security reasons. Besides a *uid,* you are assigned a group identification, or *groupid,*which places you in a class of users.

On many systems, all ordinary users are placed in a single group called *other*, but your system may be different. The file system, and therefore the UNIX system in general, determines what you can do by the permissions granted to your uid and group-id.

Owners, Groups, and Permissions:

File ownership is an important component of UNIX that provides a secure method for storing files. Every file in UNIX has the following attributes:

***Owner permissions*:** The owner's permissions determine what actions the owner of the file can perform on the file.

Group permissions: The group's permissions determine what actions a user, who is a member of the group that a file belongs to, can perform on the file.

Other permissions *:*The permissions for others indicate what action all other users can perform on the file.

The file **/etc/passwd** is the password file ; it contains all the login information about each user. You can discover your uid and group-id, as does the system, by looking up your name in **/etc/passwd**:

$ grep user1/etc/passwd

user1 : gkmbCTrJ 0 4C0M : 604 : 1 : Sonia,RBA,S/W: /usr/you:/bin/bash

$

The fields in the password file are separated by colons and are laid out like this

login-id : encrypted-password : uid : group-id : miscellany : login-directory : shell

login name: Unique name assigned to each user, consists up to 8 characters and is case sensitive

Password: The password is present in the encrypted form, consists up to 13 characters,

UID: The unique user ID number, that represents the user in OS.

GID : The unique group ID number ,assigned to a group, for sharing accounts of users belonging to same group.

comment : A miscellaneous field also referred as GECOS/GCOS field used for specifying additional user information. It usually contains full name of user, office location, telephone number or any other number**.**

Home directory: Directory into which a user is placed by default when he is logged in**.(*ex/home/cse501)***

login program : This field contains path name of the program (of the shell) that will run for that user after login.(***bin/bash.)***

The file is ordinary text file.

Note that your password appears here in the second field, but only in an encrypted form. Anybody can read the password file (you just did), so if your password itself were there, anyone would be able to use it to masquerade as you. When you give your password to login,

it encrypts it and compares the result against the encrypted password in /etc/passwd. If they agree, it lets you log in.

For example, if your password is *ka-boom*, it might be encrypted as *gkmbCTrJ04COM*, but given the latter, there's no easy way to get back to the original.

The kernel decided that you should be allowed to read /etc/passwd by looking at the permissions associated with the file. There are three kinds of permissions for each file: read (i.e., examine its contents), write (i.e., change its contents), and execute (i.e., run it as a program).

Furthermore, different permissions can apply to different people. As file owner, you have one set of read, write and execute permissions. Your "group" has a separate set. Everyone else has a third set.

The -l option of ls prints the permissions information, among other things:

$ ls -1 /etc/passwd
-rw-r—r-- 1 root 5115 Aug 30 10:40 /etc/passwd
$ ls -lg /etc/passwd
-rw-r—r-- 1 adm 5115 Aug 30 10:40 /etc/passwd
$

These two lines may be collectively interpreted as: /etc/passwd is owned by login-id *root*, group *adm*, is 5115 bytes long, was last modified on August 30 at 10:40 AM, and has one link .

The string -rw-r--r-- is how **ls** represents the permissions on the file.

The first - indicates that it is an ordinary file. If it were a directory, there would be a **d** there.

The next three characters encode the file owner's (based on uid) read, write and execute permissions, rw- means that root (the owner) may read or write, but not execute the file. An executable file would have an x instead of a dash.

The next three characters (r--) encode group permissions, in this case that people in group adm, presumably the system administrators, can read the file but not write or execute it. The next three (also r--) define the permissions for everyone else — the rest of the users on the system.

The file **/etc/group** encodes group names and group-id's, and defines which users are in which groups, **/etc/passwd** identifies only your login group; the **newgrp** command changes your group permissions to another group.

The program to change passwords is called passwd; you will probably find it in /bin:

$ ls -1 /bin/passwd
-rwsr-xr-x 1 root 8454 Jan 4 1983 /bin/passwd
$

(Note that /etc/passwd is the text file containing the login information, while /bin/passwd, in a different directory, is a file containing an executable program that lets you change the password information.)

Directory permissions operate a little differently, but the basic idea is the
same.

$ ls –ld .
drwxrwxr-x 3 you 80 Sep 27 06:11 .
$

The **-d** option of **ls** asks it to tell you about the directory itself, rather than its contents, and the leading d in the output signifies that '.' is indeed a directory.

File Permissions

Read: If a user has *read* permissions, that person can view the contents of a file ex: **cat**

Write: A user with *write* permissions can change the contents of a file ex: open with **vi** editor to modify data.

Execute: A user with *execute* permissions can run a file as a program.

Directory Permissions:

Execute(x): The **x** bit on a directory grants access to the directory. The ***cd d1***command is executed successfully if the directory ***d1*** has execute permission. Otherwise the user can't access the directory. The read and write permissions have no effect if the access bit is not set.

Read(r): The read(**r)** permission on a directory enables users to view file names and directory names in that i.e use the ***ls*** command to view files and their attributes that are located in the directory.

Write(w): The write(**w**) permission on a directory is the permission to watch out for because it lets a user to add and also remove files from the directory. $***cp f1 f2 d1*** it copies **f1, f2** files to directory **d1** when **d1** has write permission otherwise copying or removing files from d1 is not possible.

A directory that grants a user only execute permission will not enable the user to view the contents of the directory or add or delete any files from the directory, but it will let the user run executable files located in the directory.

If a directory is writable, however, people can remove files in it regardless of the permissions on the files themselves.

$ cd
$ date >temp
$ chmod -w .
$ ls -ld .
dr-xr-xr-x 3 you
$ rm temp
rm: temp not removed
$ chmod 775 .
$ ls -ld .
drwxrwxr-x 3 you
$ rm temp *file is removed*
$

2.5 Inodes

A file has several components: a name, contents, and administrative information such as permissions and modification times. The administrative information is stored in the *inode* along with essential system data such as how long it is, where on the disc the contents of the file are stored, and so on.

There are three times in the inode: the time that the contents of the file were last modified (written); the time that the file was last used (read or executed); and the time that the *inode* itself was last changed, for example to set the permissions.

```
$ date
Tue Sep 27 12:07:24 EDT 1983
$ date >jiink
$ ls -l junk
-rw-rw-rw- 1 you 29 Sep 27 12:07 junk
```

```
$ ls -lu junk
-rw-rw-rw- 1 you 29 Sep 27 06:11 junk

$ ls -1c junk
-rw-rw-rw- 1 you 29 Sep 27 12:07 junk
$
```

Changing the contents of a file does not affect its usage time, as reported by **ls -lu**, and changing the permissions affects only the inode change time, as reported by **ls -1c**.

```
$ chmod 444 junk
$ ls -lu junk
-r--r--r-- 1 you 29 Sep 27 06:11 junk
$ ls -1c junk
-r--r--r-- 1 you 29 Sep 27 12:11 junk
$ chmod 666 junk
$
```

The **-t** option to **ls**, which sorts the files according to time, by default that of last modification, can be combined with -c or -u to report the order in which inodes were changed or files were read:

```
$ ls d1
d2
d3
$ ls -lut
total 2
drwxrwxrwx 4 you 64 Sep 27 12:11 d1
-rw-rw-rw- 1 you 29 Sep 27 06:11 junk
$
```

All the directory hierarchy does is provide convenient names for files. The system's internal name for a file is its i-number : the number of the *inode* holding the file's information. **ls -i** reports the i-number in decimal:

```
$ date >x
$ ls -i
15768 junk
15274 d1
15852 x
$
```

It is the *i-number* that is stored in the first two bytes of a directory, before the name, **od -d** will dump the data in decimal by byte pairs rather than octal by bytes and thus make the i-number visible.

```
$ od -d .
0000000 15156 00046 00000 00000 00000 00000 00000 00000
0000020 10427 11822 00000 00000 00000 00000 00000 00000
0000040 15274 25970 26979 25968 00115 00000 00000 00000
0000060 15768 30058 27502 00000 00000 00000 00000 00000
0000100 15852 00120 00000 00000 00000 00000 00000 00000
0000120
$
```

The first two bytes in each directory entry are the only connection between the name of a file and its contents. A filename in a directory is therefore called a link , because it links a name in the directory hierarchy to the inode, and hence to the data. The same i-number can appear in more than one directory. The **rm** command does not actually remove inodes; it removes directory entries or links. Only when the last link to a file disappears does the system remove the
inode, and hence the file itself. If the i-number in a directory entry is zero, it means that the link has been removed, but not necessarily the contents of the file — there may still be a link somewhere else. You can verify that the i-number goes to zero by removing the file:

The ***ln*** command makes a link to an existing file, with the syntax

```
$ ln old-file new-file
```

The purpose of a link is to give two names to the same file, often so it can appear in two different directories.

```
$ ln junk lnk
```

```
$ ls -li
total 3
15768 -rw-rw-rw- 2 you 29 Sep 27 12:07 junk
15768 -rw-rw-rw- 2 you 29 Sep 27 12:07 lnk
15274 drwxrwxrwx 4 you 64 Sep 27 09:34 d1
$
```

```
$ echo
one>junk
$cat junk
  one
$cat lnk
   one
```

The integer printed between the permissions and the owner is the number of links to the file.
When you change a file, access to the file by any of its names will reveal the changes, since all the links point to the same file.

```
$ ls -1
total 3
-rw-rw-rw- 2 you 2 Sep 27 12:37 junk
-rw-rw-rw- 2 you 2 Sep 27 12:37 linktojunk
drwxrwxrwx 4 you  64 Sep 27 09:34 d1
$ rm $ is -1
total 2
-rw-rw-rw- 1 you 2 Sep 27 12:37 junk
drwxrwxrwx 4 you 64 Sep 27 09:34 d1
```

After *lnk* is removed the link count goes back to one. Removing a file just breaks a link; the file remains until the last link isremoved. once the last link to a file is gone, the data is irretrievable

cp makes copies of files:

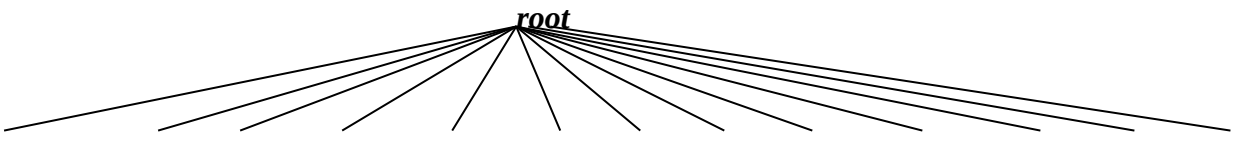
```
$ cp junk cnk
$ ls -li
total 3
15850 -rw-rw-rw- 1 you 2 Sep 27 13:13 cnk
15768 -rw-rw-rw- 1 you 2 Sep 27 12:37 junk
15274 drwxrwxrwx 4 you 64 Sep 27 09:34 d1
$
```

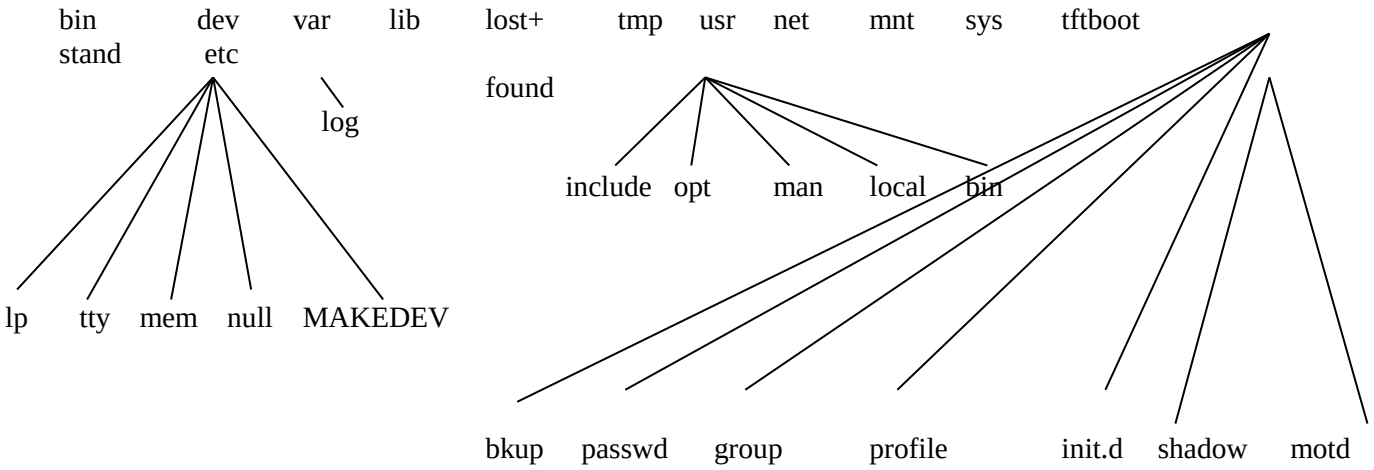
The i-numbers of junk and cnk are different, because they are different files, even though they currently have the same contents. Changing the copy of a file doesn't change the original, and removing the copy has no effect on the original.
There is one more common command for manipulating files: ***mv*** moves or renames files, simply by rearranging the links. Its syntax is the same as ***cp*** and ***ln***:

```
$ mv junk snk
$ ls -li
total 2
15274 drwxrwxrwx 4 you 64 Sep 27 09:34 d1
15768 -rw-rw-rw- 1 you 29 Sep 27 13:16 snk
```

There are 8 attributes of a file in inode table:
1.File Type : Type of a file(regular,directory, device, symbolic link, named pipe, socket etc..)
2. File Access Permissions: read(r),write(w),execute(x) permission for 3 users like owner, group, others
3. No.of links: Total no.of links of a file(created with *ln* command)
4. Owner of file
5.Group of file
6.File access time
7.file modification time
8. Size of file.

2.6 The directory hierarchy





The UNIX OS uses a hierarchically organized directories to keep track of the files, referred to as the directory hierarchy or directory tree. In this directory tree, there is a single **root** directory also referred as the parent directory and all other directories are subdirectories of the root directory. Every directory includes the files and some other directories.

/unix is the program for the UNIX kernel itself: when the system starts, /unix is read from disc into memory and started. Actually, the process occurs in two steps: first the file /boot is read; it then reads in /unix.

root Directory:-The root directory and all other sub directories are collectively called as **the file system**. The name of the root directory is the name of the file system.
The files presented in **root** directory are :
bin dev etc lib lost+found usr tmp net mnt sys var etc…

/bin Directory:- UNIX uses the bin directory to store the executable files. It also consists of most commonly used UNIX commands and tools like compilers, assemblers, editors etc.,

/dev Directory :-UNIX treats devices as if they are files. The files present in device directories are referred to as device files.
The files presented in the dev directory are as follows
lp file **tty** file **null** file **mem** file **MAKE DEV** file
/dev/lp: This file represents the printer in the terminal action or request mode on the printer must go through the /dev/lp file.
/dev/tty:-tty file represents terminal device files. It monitors the process of user's login shell.
/dev/null:- When you do not want to display the output on the screen, then the output is sent to this null file. This file also referred to as bit-bucket.
/dev/mem:- This file represents memory.
/dev/MAKE DEV:- It is used to create a device file for any device you specify.

3.The etc directory:-The programs used for system maintenance or system administration are stored in the ***etc*** directory.
bkup file passwd file group file profile file init.d file shadow file motd file
/etc/bkup:- this /etc/bkup file is used to backup and restore the system files.

/etc/group file:-The group file is used to store all the groups in the system and the list of users in each group and their passwords.
The following fields are existed in the **/etc/group** file.

group name :-unique name assigned to each group, consist up to 8 characters and is case sensitive*password* :-passwords present in an encrypted form for the group. This field is usually empty.***(ex:X123ERFD)***

GID :-the unique group ID number, that represents the group in the OS.***(ex:100)***

User-list :-An optional list of users separated by commas who are authorized to access files in that group.***(ex:cse501..)***

A sample line of the **/etc/group file** is shown below.
Student : X123ERFD : 100 : cse501, cse502,cse503,cse504

/etc/ init.d :- The */etc/init.id* file is used by the system when it changes the system state(i.e from single user mode to multiuser mode or shutdown state mode or shutdown and reboot mode or system administration mode or vice versa) also called as run levels.
/etc/**motd (**method of day):- This file usually contains information sent to the user by the system administrator(regarding shutdown of the system ,repair &maintenance of the system or any other information of the system he feels important for users to know).This file is displayed every time the users logs into the system
/etc/passwd:-The *password* file in the ***etc*** directory is the first file to grant access to users of UNIX OS.
This file is the systems user db, which lists login information for each user on the system.
A simple line of **/etc/passwd** file is.
Cse501 : AIEW23J : 201 : 100: Sonia,RBA,S/W : /user/cse501 :/bin/bash.

login name: Unique name assigned to each user, consists up to 8 characters and is case sensitive.**(ex: *Cse501)***
Password: The password is present in the encrypted form, consists up to 13 characters, which includes .,/, 0to9, A to Z. ***(*ex: AIEW23J)**
UID: The unique user ID number, that represents the user in OS.**(*ex:201)***
GID : The unique group ID number ,assigned to a group, for sharing accounts of users belonging to same group.**(*ex:100)***
comment : A miscellaneous field also referred as GECOS/GCOS field used for specifying additional user information. It usually contains full name of user, office location, telephone number or any other number**.**
(*Sonia,RBA,S/W)*
Home directory: Directory into which a user is placed by default when he is logged in**.(*ex/home/cse501)***
login program : This field contains path name of the program (of the shell) that will run for that user after login.(***bin/bash.)***
/etc/Profile:-This file contains shell configuration files of Bourne and korn shells.
This file is used by the system administrator for setting up any environment for a user who is different from the system default, and to make sure that all user have a common environment.
/etc/shadow:- This file contains the secured password information. The read access of this file is restricted only to the super user and all other users doesn't have capability to access this file.

- Each line contains entries of each user in fields delimited by colons.
- Fields present in the file are listed below.
  - *Login name*

- *Encrypted password.*
- *Last change of the password in days.*
- *Minimum number of days required b/n password changes.*
- *Maximum no. of days the password is valid.*
- *No. of days to issue a warning message about the expiry of password.etc..*

.

4.The Lib directory :-

The UNIX OS uses the *lib* directory to store functional and procedural librariesThe functional and procedural libraries in the '*lib'* directory includes library functions for c,c++ languages(such as stdio.h, iostream.h,math.h etc..),system calls(such as open(),chdir(),mount(),etc..) and I/O routines users can also add their own custom routines to the '*lib'* directory.

5.usr directory: Most UNIX systems place the users directory in their home directory, which is a subdirectory for the *usr* directory. It is used to store all user accounts and mail commands. The user directory was intended to be the central storage place for all user related commands.

The files present in the user directory are : *opt directory ,bin directory, local directory, man directory, include directory*

/usr/opt:- As the name implies it is an optional directory, the optional S/W packages are usually stored here.

/usr/bin :-This directory contains the various executable binary files and additional UNIX commands not included in the */bin* directory.

/usr/local:- The */user/local* directory is used to store local programs man pages(help) and libraries necessary for the UNIX OS.

/usr/man:-The source text for the various commands are stored in a special directory called as */usr/man* ..

/usr/ include *:-*This directory contains special files referred as header files (with an extension .h)

/usr/adm system administration: accounting info., etc.
/usr/dict dictionary (words) and support for spell(l)
/usr/games game programs
/usr/lib libraries for C, FORTRAN, etc.
/usr/mdec hardware diagnostics, bootstrap programs, etc.
/usr/src source code for utilities and libraries
/usr/src/cmd source for commands in /bin and /usr/bin
/usr/src/lib source code for subroutine libraries
/usr/spool working directories for communications programs
/**usr/spool/lpd** line printer temporary directory
/usr/spool /mail mail in-boxes
/usr/you your login directory

6. /lost +found:-. there are chances of problems or failures due to non-synchronization of the computer with the file system. These files can be recovered by the UNIX kernel from problems and are placed in a special directory named lost+found directory.

7*. / tmp:-* The *'tmp'* directory is a special directory that is available for storing temporary files.

The files stored in this directory are deleted as soon as the system is shutdown.

Some other directories present in the UNIX system of less interest are **: *net*** *directory,* ***tft boot*** *directory* **,** ***var*** *directory* **,** ***sys*** *directory,* ***mnt*** *directory,* ***stand*** *directory*

8./net directory :-.The word *'net'* stands for network is a directory in UNIX that contains files for accessing other computers on your N/W.

9./tft boot directory :- The *tft* (trivial file transfer protocol boot) directory is relatively new feature of UNIX, which contains versions of the kernel suitable for X windows system.

***10./var directory* :-**The *var* directory is used to hold files with constantly varying contents as the system runs. The *var* directory contains a directory named log, which is used to store messages generated by the system.

11./sys directory :-This directory contains the files indicating the system configuration.

***12.mnt directory* :-**.The *mnt* directory is said to be a common place to mount external media like hard disk, removable cartridges, driver, floppy disks, CD-ROMS and so on.

***13.stand directory* :-**It contains programs and configuration files used when booting the system.

2.7 THE file COMMAND—KNOWING THE FILE TYPE

Sometimes, apart from classifying Unix files as regular files, directories, device files and other files they are also classified as text files, executable files, and directories. This classification is based on the contents of the file. The file command is used to identify the type of the files on the basis of their contents. When this command is used, it reads either the header or first few hundreds of bytes of the file (given as an argument) and an educated guess is made on the type of the file. More often this guess is correct. One might argue that a filetype could be guessed or even concluded looking at the extension names. But Unix has nothing to do with extension names. This is because Unix puts no restriction on extensions in filenames. Certain category of files such as executables are recognized by the information stored on their headers—the information stored in the first-byte. This first byte information is known as the magic number. This magic number is consistent for similar file types between files and systems. The correlation between magic numbers and file types is contained in the file /etc/magic. For example the octal 410 is the magic number of executable files. These magic numbers can be verified by taking the octal dump of the relevant file.

For text files, the clues may not be available directly with the magic numbers. Rather, such clues will be available deeper in the file. For example, the clue for identifying text files could be, the use of a new line character at the end of every line. The presence of words such as #include indicate a C source file, lines beginning with a period may indicate nroff or troff input and so on. The study of following examples will give a better understanding of the usefulness of this command.

```
$file mgv
mgv: ASCII text
$file /bin
/bin: directory
$file mac.c
mac.c: ASCII C program text
$touch liju
$file liju
liju: empty
$cd /bin
```

$file csh
csh: symbolic link tcsh
$
In all the examples shown above, filenames have been given in the form of relative pathnames. Filenames can be given in the form of absolute pathnames also. Here it may be recalled that the listing command ls with the flag option F also gives an idea about the filetypes but in a limited way.

2.8 THE chmod COMMAND—CHANGING FILE PERMISSIONS

The chmod command is used to change the permissions of a file after its creation. Only the owner or the super user can change file permissions. The general syntax of this command is

$chmod assigenment_expression filename

The assignment expression holds the following information.

1. The information about the category of users {user –u, group –g, others –o, all –a}.
2. The information about granting or denial of the permission {the operators +, – and =}.
3. The information about the type of permission {read –r, write –w, execute –x}.

Although we generally consider only three types of users such as the owner, the group and others, a fourth category called all {a} that refers to all the three conventional categories is also considered. Further the + (plus) operator is used for granting the permission, the – (minus) operator is used for removing the permission and the = (equal to) operator is used for assigning absolute permission. Obviously the different permissions that are either granted or denied are the read permission (r), the write permission (w) and the execute permission (x).

Some examples that illustrate the use of the chmod command with reference to a file named sample with initial permissions of –rw–r– –r– – are given here.

$chmod u+x sample
$ls –l sample
–rwxr—r— 1 mgv csd 5180 Jan 07 12:06 sample

In this example u+x is the argument expression. The user has been granted the execution permission. As already mentioned above u stands here for user, x for execution and + for granting.

$chmod ugo+x sample;ls –l sample
–rwxr–xr–x 1 mgv csd 5180 Jan 07 12:06 sample
$

Execution permission has been granted (because of +) to all categories of users, that is, owner (because of u), group (because of g) and others (because of o). The expression ugo+x can also be written as a+x where a means all. Whenever the category is all, the category of user need not be mentioned explicitly. It is used by implication when the argument expression is just as +x. Thus the previous command can be rewritten in either of the following two ways.

$chmod a+x sample; ls –l sample

or

$chmod +x sample; ls –l sample

The chmod command can work on more than one file at a time as shown in the following example.

```
$chmod u+x sample1 sample2 sample3
$ls –l sample1 sample2 sample3
–rwxr––r––      1       mgv     csd     5180    Jan 07 12:06  sample1
–rwxr––r––      1       mgv     csd     6191    Jan 07 01:16  sample2
–rwxr––r––      1       mgv     csd     7101    Jan 07 02:26  sample3
$
```

More than one permission can be set using multiple argument expressions like u-x, go+x.

$chmod u-x, go+x sample

$ls –l sample

–rw-r-xr-x 1 mgv csd 5180 Jan 07 12:06 sample

$

Relative and Absolute Permissions Assignment

In all the examples discussed hitherto changes made were relative to the present settings. In other words, an expression like u+x sets the execute permissions to the user. It will not disturb other settings of either this or any other category. This type of permission assignment is called *relative permission* assignment.

The use of the = operator in the chmod expression assigns or grants only specified permissions and removes all other permissions. This type of granting permissions is called *absolute permission* assignment. Below is given an example where absolute assignment is made.

$chmod a=r sample; ls –l sample

–r––r––r–– 1 mgv csd 5180 Jan 07 12:06 sample

$

From the output of the above example, one may observe that all have been given read permissions after removing the permissions associated with the file earlier.

Permissions with Octal Numbers

File permissions can also be assigned using octal numbers. In this representation

1. 4_8 (as it is equivalent to 100_2) assigns read permission, 2_8 (as it is equivalent to 010_2) assigns write permission and 1_8 (as it is equivalent to 001_2) assigns the execute permission and so on.
2. Permission assignments made using octal numbers are always absolute assignments. In other words, octal numbers cannot be used for relative permissions assignment.

For example, a 6_8 $(110)_2$ assigns both read and write permissions and denies the execute permission 5_8 $(101)_2$ assigns read and execute permissions and denies write permission.

Because there are three categories of users, one has to use *three octal digits* in the expression field, as shown in the following example.

$chmod 644 sample; ls –l sample

–rw–r––r–– 1 mgv csd 5180 Jan 07 12:06 sample

$

The $chmod 761 sample is the octal notation equivalent of the following command.

$chmod u=rwx, g=rw, o=x sample

$

Permissions can be granted to all the files and sub-directories in a directory by using the recursive option (–R) with the chmod command. For this the argument must be the directory

name. For example, the execute permission to all category of users with respect to all files and directories under the current directory can be granted using the command given below.

```
$chmod –R a+x
$
```

2.9 THE chown COMMAND—CHANGING THE OWNER OF A FILE

As already mentioned, every file has a owner. When a file is created, the creator becomes the owner of the file. Only the owner can change the major attributes of a file (of course, the system administrator also can do it).

Sometimes it is necessary to change the ownership of a file. There are two ways in which the ownership can be changed—by copying the file in to the target user's directory, and by using the chown command.

For example, the file sample from the directory of hmk is *copied* to the home directory of someone else, say mgv. Then mgv becomes the new owner of the file sample. If, now, the oldfile and newfile are listed using the ls –l command, one sees that every detail will be same except the owner.

The copying method of changing the ownership has the following disadvantages:

- it creates an additional file and thus uses extra space.
- the new owner should have the knowledge about the permissions of the file.

Changing the owner of a file using the chown command is more simpler and direct method of changing the ownership. This command takes two arguments, login name of the new user and the name of the file. An example is given below.

```
$ls –l sample
–rwxr– –r–x 1   rajcsd    425  May 10 20:30   sample
$chown kumar sample ; ls –l sample
–rwxr– –r–x        1        kumar     csd     425       May 10 20:30 sample
$
```

It should be noted that the ownership once surrendered can not be reinstated. Also moving a file does not change the ownership. Further this command can use the –R option—the recursive option. When this option is used the ownership of all the files in the current directory are changed.

2.10 THE chgrp COMMAND—CHANGING THE group OF A FILE

In Unix, all files not only belong to an owner but also to a group. One may need to change the group of a file under certain circumstances such as when new groups are set up on a system or when files are copied to a new system. This is done by using the chgrp command. Only the owner of a file can change the group (of course, the system administrator also can do the same). Changing the group using the chgrp command is also straightforward.

This command also takes two arguments; the name of the new group and the name of the file. For example, $chgrp planning sample $

```
$chgrp                    planning                    sample
$
```

As shown above, the name of the new group must appear as the first argument and the name of the file has to appear as the second argument. The recursive option –R can also be used with this command. When used with the –R option, the group of all the files under the current directory is changed.

File Commands cp(Copying Files)

Name: cp

Purpose: The ***cp*** command copies files or directories from one place to another.

Syntax1 : cp [options] source dest

Here The *source* is the name of the file you want to copy, *dest* is the name of the new file.

Syntax2: cp [options] source(s) destdir

If you specify a *dest* that is a directory, **cp** will put a copy of the *source(s)* in the directory. The filename will be the same as the filename of the source file

Option	Descritpion
-i	Copies interactively
-p	Preserves modification time and access permissions.
-r	Recursive copying through subdirectories

Description and Examples:

1. The *cp* command copies a file or group of files into a directory or directory structure.
2. It creates an exact image of the source file on the disk with different name.

Examples: 1.

```
$cp  x.txt y.txt
```

This command copies the content of *x.txt* file into *y.txt*. Before copying if *y.txt* is not existed then cp command creates the new file *y.txt* and copies. If *y.txt* is already existed before copying and it has some data that data will be overwritten with the new data of *x.txt.*

2. With cp command we can copy more number of source files into a directory. If more than two inputs are given, ***cp*** treats the last argument as the destination and the other files as sources. This works only if the sources are files and the destination is a directory, as in the following

```
$cp  x.txt a b f1 f2 d1
```

This command places all 5 files *x.txt,a,b,f1,f2* files into a directory ***d1.***all these files are copied with the same names into the directory .

Options:

-i (Interactive Mode):

-i option of the cp command warns the user before overwriting the destination file.

```
$ cp -i a.txt c.txt
overwrite c.txt? (y/n)
```

If you choose y (yes), the file will be overwritten. If you choose n (no), the file c.txt isn't changed.

-p(preserves modification time):. With –p option the modification time of destination file is assigned to the modification time of source file.

```
$cp x.txt y.txt
$ls –l x.txt y.txt
_rw--r-- r--   1   cse501  cse501   40   May 25 15:46   x.txt
_rw--r-- r--   1   cse501  cse501   40   Nov 15  14:02  y.txt
```

By using –p option we can preserve the modification time.

```
$cp –p x.txt y.txt
$ls –l x.txt y.txt
_rw--r-- r--   1   cse501  cse501   40   May 25 15:46   x.txt
_rw--r-- r--   1   cse501  cse501   40   May 25 15:46   y.txt
```

-r(recursive copying): To copy all files in directrories as well as files in subdirectories and files in sub-sub directories use –r opiton.

$cp -r d1/* d2/d3

3. rm(Deleting Files)

Name: rm

Purpose: To delete files

Syntax: rm option(s) file(s)

Options:

-f	Forcily remove files that donot have write permission
-i	Interactively removes the files by confirming with the user before removing each file
-r	Recursively deletes the entire contents of the directory as well as subdirectories.

Description & Examples:

1. ***rm f1*** this command deleted file f1.

2.With single ***rm*** command we can delete more than one file.

$rm f2 f3 f4 deletes f2 ,f3,f4 files at a time**.**

1. **$rm d1/d2/f1** deletes a file **f1** which is in directory **d2**
2. **$ rm *** deletes all files in the current working directory.
3. **$rm f*** deletes all files ,for which the file name started with **f.**

-i(interactive): this option deletes files interactively. It confirms from user before deleting each file.

```
$rm –i a.txt b.txt c.doc z y
a.txt:? y
b.txt:? n
c.doc :? n
z:?y
a:?y
```

In the above while removing each file the command display **file:?** If user press **'y'** then the file is removed. Otherwise the file is not removed.

-f(forcibly):

-f option removes a file forcibly ,even though a file doesn't have write permission. **–f** removes the file without user confirmation.

-r(recursive): with –r option ***rm*** deletes all files, subdirectories, files under subdirectories, sub-sub directories, files under sub-sub directories.

Generally ***rm*** can't remove directories ,but with **–r** option it will delete subdirectories & files under it.

$rm –r * deletes current directory tree completely. All files in current directory, as well as all subdirectories and their files will be removed.

4. mv(Renaming Files)

Name: mv

Purpose: To rename or move the files

Syntax: mv option(s) file1 file2

Options:

-i	Interactively moves the files
-r	Recursively moves the files, subdirectories and files in sub

	directories.
-p	Preserves modification time.

Description & Examples:

1. **mv** command doesn't create a copy of the source file, but merely renames it to destination file. So **mv** renames files and directories
2. After moving the source file name is deleted, only the destination file name is remained.

Example:

1. **$mv f1 f2** it renames the file **f1** with **f2**.(deletes the f1 name from directory and places name f2).
2. **$mv f1 f1** it gives error message.
3. **$mv d1/f1 d2/f2** the command moves the file **f1** from directory **d1** to directory **d2** and names the file as **f2** after moving **f1** will be removed from **d1.**
4. **$mv d1/f1 d2/** it moves the file **f1** from directory **d1** to directory **d2**.and maintains the same name (**f1**). After moving **f1** will be removed from **d1**.
5. **$mv d1/f1** when the destination path is not specified the file is moved to current working directory.

-i(interactive): mv with –i option moves files interactively.

$mv –i f1 f2 If **f2** is already existed in current working directory the **–i** option displays the message as

mv:Overwrites f2? If user press '**y**' then **f2** data is erased and **f1** data is placed. If user press '**n'** then the **mv** command failed.

-p(preserves modification time):

When mv command is executed successfully, the modification time of a destination file is assigned to the new time. (i.e the time when mv is executed). But to preserve the modification time of original file we can use –p option.

```
$ls –l x.txt
rw--r-- r--    1    cse501  cse501    40    May 25 15:46   x.txt
$mv x.txt y.txt
$ls –l  y.txt
_rw--r-- r--   1    cse501  cse501    40    Nov 15  14:02  y.txt
```

```
$ls –l x.txt
rw--r-- r--    1    cse501  cse501    40    May 25 15:46   x.txt
$mv -p x.txt y.txt
$ls –l  y.txt
_rw--r-- r--   1    cse501  cse501    40    May 25 15:46   y.txt
```

-r(recursive):

To move all the files under a directory, and to move all the subdirectories, files under subdirectories, sub-sub directories and files under sub-sub directories use **–r** option.

$mv d3/* d2/ it moves all files of directory **d3** to a directory **d2** but subdirectories of **d3** are not moved.

$mv -r d3/* d2/ it moves all files ,subdirectories, sub-subdirectories, and files of d2 to d3.