MODULE - 2

Smitha.N
CMRIT, CSE

① File Attributes and Permissions:-

The ls command with options:-

→ File has number of attributes.

→ ls -l used to display most of the attributes of a file - like

* permision
* size.
* ownership. etc.

→ o/p has seven attributes of files in current directory.

→ Eg:-

```
ls -l
total 4
-rw-r--r--  1 kumar metal 195 May 1 13:45 chap1
drwxr-xr--  1 kumar metal 84 May 2 13:45 chap
 :
 :
 :
```

→ link count
→ pointer to the file.
→ Short cut way to access.
→ more than one file name
: same file.

→ total specifies total of 4 blocks are occupied by these files on disk, each block consisting of 512 bytes.

→ File type & Permission :-

* First column shows type & permissions associated with each file.

* if its - indicates its a ordinary file else it is d which indicates its directory file.

* Series of characters r, w, x and - after first character. A file can have 3 types of permissions - read, write and execute.

→ * Links :- Second column indicates the no of links associated with a file.

A link count greater than one indicates that the file has more than one name.

→ Ownership :-

* creator becomes owner of the file.

* Owner has full authority to tamper with a files contents and Permissions.

* owner can create, modify & remove files in a directory.

→ Group owner :- when user account is opened user assigned to some group. Group members often need to work on same file.

→ File size:- Fifth column shows size of the file in bytes. Amount of data it contains. It is only character count of the file and not a measure of disk space that it occupies.

Directories size depends on list of filenames along with an identification number (inode number).

→ Last Modification Time:-

Sixth, seventh & eighth columns indicate the last modification time of the file.

→ Filename:-

Last column displays filename

(2) The -d option:- Listing Directory Attributes.

→ $ ls -ld helpdir.

drwxr-xr-x 2 kumar metal 512 May 9 10:31 h

File Ownership :-

→ If you create a file, then your owner of the file.
→ Several users may belong to a single group.
→ System administrator creates a user account. he has to assign these parameters to the user :

- → The User-id (UID) - both its name and numeric representation.
- → The group-id (GID) - both its name and numeric representation.

→ /etc/passwd maintains UID (both name + number) & GID (only number).
→ /etc/group maintains GID (both name + number)

→ $id
uid = 655537 (kumar) gid = 655535 (metal)

File Permissions :-

→
```
$ ls -l chap02 dept.lst
-rwxr-xr--  1 kumar metal 20500 May 10 19:21 Chap02
-rwxr-xr-x  1 kumar metal  890 Jan 29 23:17 dept.lst
```

→ above o/p first column represents file permissions
→ UNIX follow three tiered file protection system that determines a file's access rights

→ | - | rwx | r-x | r-- |

* Here first - represents file is an ordinary file., for directory it shall be d

* Each group represents a category contains three slots. Representing the read, write and execute permissions of the file.

→ r → read permission

→ w → write permission

→ x → execute permission

→ - → absence of the corresponding permission.

* First group (rwx) has all three permissions. i.e file is readable, writable and executable by the owner of the file.

* Second group (r-x) indicates absence of write permission by the group owner of the file.

* Third group (r--) write & execute bit is absent. This set applicable to others. (neither owner nor group)

Chmod : changing file Permissions :-

→ A file (or) directory is created with a default set of permissions.

→
```
$cat > x_start
$ls -l x_start
-rw-r--r--  1 kumar metal 1960 Sep 6 23:30 X_start
```

* Here file doesn't also have execute permission so ~~one~~ does one execute such file is with chmod command.

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

→ This command can be used in 2 ways.

* In a relative manner by specifying the changes to the current permissions.

* In an absolute manner by specifying the final permissions.

Relative Permissions :

→ chmod changes only the permission specified in the command line and leaves the other permissions unchanged.

→ Syntax:

chmod category operation Permission filename

→ chmod takes as its argument an expression which contains three components:

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

→ Suitable abbreviations can be combined to frame compact expression & then used it as a argument to chmod.

Abbreviation.

Category	Operations :	Permission
① u - user	+ - Assigns permissions	r - read
② g - Group	- Remove "	w - write
③ o - others	= → Assigns absolute permission	x - Execute
④ a - all (ugo)		

Examples :-

① $chmod u+x xstart [assigns (+) execute (x) to user (u)]

$ ls -l xstart

-rwxr--r-- 1 kumar metal 1906 May 10 20:30 xstart

others remain unchanged.

② To enable all of them to execute this file (xstart)

$chmod ugo+x xstart

$ ls -l xstart

-rwxr-xr-x 1 kumar metal 1906 May 10 20:30 xstart

(a) $ chmod a+x xstart.

(b) $chmod +x xstart

③ chmod accepts multiple filenames in the command lines.

Eg:- $ chmod u+x note note1 note3.

④ To Remove Permission (-) operator used.

Eg:- $ ls -l xstart

-rwxr-xr-x 1 kumar metal 1906 May 10 20:30 xstart

$ chmod go-r xstart ; ls -l xstart

-rwx--x--x 1 kumar metal 1906 May 10 20:30 xstart

④. more than one permission could be set.

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

Absolute Permissions :-

→ Set all nine permission bits explicitly.

→ Expression used by chmod is String of three octal numbers. (base 8).

value 0 to 7. Set of three bits represents one octal digit.

→ Read permission — 4 (octal 100)
write " —— 2 (octal 010)
Execute " —— 1 (octal 001).

→ for instance 6 represents read + write permissions, 7 represents all permission.

Binary	octal	Permissions	Significance
000	0	---	No permissions
001	1	--x	Execute only
010	2	-w-	write only
011	3	-wx	Writable + executable
100	4	r--	Readable only
101	5	r-x	Readable + executable
110	6	rw-	Readble + writable
111	7	rwx	all

Eg:- To assign read + write permission to all three categories.

(4+2)

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

② Assign all the permission to owner, read + write permission to the group and only execute permission to the others.

```
$chmod 761 xstart.
```

The security implications:-

→ -rw-r--r-- 1 kumar metal 1906 may 10 20:30 xstart

→ $chmod 000 xstart

$ls -l xstart

---------- 1 1 ——————

Here xstart file becomes useless.

→ $chmod 777 xstart

-rwx rwx rwx 1 1 ——————

Here universal write permission is given. Where security concern is not satisfied.

Using chmod Recursively (-R).

→ chmod descend a directory hierarchy & apply the expression to every file & sub-directory it finds.

→ Using -R option it is done.

→
```
$chmod -R a+x dir_name.
```

∴ all the file & directories found in the tree-walk are all made executable to all the users.

→
```
$cd home
$ chmod -R 755 .
```

works on hidden files also

```
$chmod -R a+x *
```

leaves out hidden files.

Directory Permissions :-

* Directories also have their own permissions. and differ from those of ordinary files.

* Default permissions are.

```
$mkdir c_progs ; ls -ld c_progs
drwxr-xr-x 2 kumar metal 512 May 3 9:57 c_progs
```

Default permissions are rwxr-xr-x (@ 755).

* Directory should never be writable by group (or) others.

Smitha.N
CSE, CMRIT

The Shell:

* Shell is a unique and multi-faceted program.
* The shell is a command interpreter and a programming language rolled into one.

The Shell's Interpretive Cycle:

→ prompt appears as soon as we login to terminal & terminates as soon as we logout.

→ Shell springs into action the moment we key in something in front of the prompt.

Eg:- $ ps

PID	TTY	TIME	CMD
328	pts/2	0:00	bash

Bash shell is running at the terminal /dev/pts/2

→ When we key in command, it goes as input to shell & scans the command line for metacharacters.

Eg:- cat > foo
rm -r *
ls | more.

} these special characters (or) metacharacters like >, |, * in the command line does mean anything to command but mean something to shell.

→ Following activities are typically performed by the shell in this interpretive cycle.

* . The shell issues the prompt & waits for you to enter a command.

* . After a command is entered, the shell scans the command line for metacharacters and expands abbreviations (like * in rm *) to recreate a simplified command line.

* . It then passes on the command line to the kernel for execution.

* . The shell waits for the command to complete and normally can't do any work while the command is running.

* . After command execution is complete, the prompt reappears and the shell returns to its waiting state to start the next cycle. You are now free to enter any command

② Shell Offerings :

* UNIX offers variety of shells, grouped into 2 categories.

→ The Bourne family comprising the

* Bourne shell (/bin/sh)
* .korn shell (/bin/ksh)
* Bash shell (/bin/bash)

→ The C shell (/bin/csh) and its derivative, Tsh (/bin/tcsh)

→

→ echo $SHELL displays absolute pathname of the shell's command file.

---

③ Pattern Matching - The wild cards

→ special set of characters that the shell uses to match filenames.

→ metacharacters are used to construct the generalized pattern for matching filenames belong to a category called wild-cards.

Eg:- ls chap chap01 chap02 chap03

ls chap* → metacharacter

ordinary.

Shell scans command line, expands for execution.

The Shells wild-cards.

Wild-Card.	Matches
*	Any number of characters including none.
?	A single character
[i j k]	A single character- either an i, j @ k.
[x-z]	A single character that is within the Ascii range of the character x and z
[!i j k]	A single character that is not an i, j @ k &.
[! x-z]	A single character that is not within the Ascii range of the characters x and z.
{patt, pat 2..}	pat 1, pat 2 .. etc.

The * and ? :-

* :-

→ any no of characters including none.

→ Eg:- ls chap*

chap chap01 chap02 chap03.

* When shell encounters *, it identifies it as wild-card and expands it.

Like ls chap chap01 chap02 chap03.

* Shell hands over this to kernel which uses its process creation facilities to run the command.

Eg2:- echo *

List all files in the current directory.

Eg-3:- rm *

delete all the files in current dir

? :-

→ matches single character

Eg1:- ls chap? // matches 5-character

chapx chapy chap2

Eg2:- ls chap?? // matches 6 characters.

Chap01 Chap02 Chap03

Matching the Dot :-

→ To all the hidden filenames in current directory having atleast 3 character after the dot

```
$ ls .???*
.bash_profile   .exrc   .profile.
```

→ if dot present anywhere inbetween, need not be matched explicitly.

Eg:- ls emp*lstt

emp1stt emp1.lstt emp221lstt

The character class :-

→ We can frame more restrictive patterns with the character class.

→ character class comprises a set of characters enclosed by the rectangular brackets [and]. but it matches a single character in the class.

→ Eg:- ls chap0[124]

chap01 chap02 chap04

→ Range specification is also possible inside the class with a - (hyphen)

Eg:- $ ls chap0[1-4]

chap01 chap02 chap03 & chap04

$ ls chap[x-z]

chapx

chapy

chapz

Negating the character class (!)

→ framing pattern that reverses the matching criteria.

Eg:- ls *.[!co]

Matches all filenames with a single character extension but not the .c @ .o files.

Eg:- [!a-zA-Z]* matches all the filenames that doesn't begin with an alphabetic character.

Escaping And Quoting :-

→ there may be chances to have special characters as part of filenames themselves.

Eg:- ls chap*

Chap (chap*) chap01

if we try to remove filename chap* using rm chap* → it would result in deleting all the files ∴ * metacharacter is used.

→ The shell provides two solution to prevent its own interference:

*. Escaping :- Providing a \ (backslash) before the wild-card to remove (escape) its special meaning.

*. Quoting :- Enclosing the wild-card, or even the entire pattern, within Quotes

Eg:- 'chap *'

Escaping :-

→ placing a \ (backslash) immediately before a metacharacter turns off its special meaning.

Eg:① rm chap* → deletes only file name chap*.

\ suppresses the wild-card nature of the *.

This feature is known as escaping.

②$ echo > chapo[1-3] creates a file chapo[1-3]

$ ls chapo\[1-3\]

chapo[1-3] must escape the [and]

→ Escaping the space :-

* $ rm My\ Document.Doc

without the \ rm would see two files.

→ Escaping the \ itself :-

* Sometimes we may need to interpret \ itself then we need another \ before it.

Eg: $ echo \\

\

$ echo The newline char is \n
The newline char is \n.

→ Escaping the newline character :-

* Some command lines that use several arguments can be long enough to overflow to the next line.
To ensure better readability, we need to split the wrapped line into two lines, but we need to input a \ before pressing [Enter].

$ find /usr/local/bin /usr/bin \ [Enter]
> -size -2048

Here \ escapes the meaning of the newline character generated by [Enter].

Quoting :-

→ another way to turn off the meaning of a metacharacter. by enclosing command argument in quotes.

→ $ echo '|'
|

$ rm 'chap*'
removes file chap*

$ rm "My document.doc"
removes file My document.doc.

→ Single quotes is flexible because they protect all special characters.

Eg:- $ echo 'The characters |, <, > and $ are special'.
The characters |, <, > and $ are special.