1. SHELL SCRIPT

When a group of commands have to be executed regularly, they should be stored in a file, and the file itself executed as a **shell script** or **shell program**.

We use .sh extension for all shell scripts.

Every Shell script begins with special interpreter line like - #!/bin/sh or #!/bin/ksh or #!/bin/bash

This interpreter line specify which shell (Bourne/Korn/Bash) user prefer to execute shell script, and it may or may not be same as user login shell.

Example:

```
#!/bin/sh
#Sample Shell Script – sample.sh

echo "Today's Date: `date`"
echo "This Month's Calender:"
cal
```

Execution & Output:

```
$ chmod 777 sample.sh                    #Make Script Executable
$ sh sample.sh                           #Execute Script – sample.sh

Today's Date: Sun Jan 13 15:40:13 IST 2013

This Month's Calender:
January 2013
Su Mo Tu We Th Fr Sa
1 2 3 4 5
6 7 8 9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
```

2. read: MAKING SCRIPT INTERACTIVE

The read statement is the shell's internal tool for taking input from the user, i.e. making script interactive.

Example:

```
#!/bin/sh
#Sample Shell Script - simple.sh
echo "Enter Your First Name:"
read fname
echo "Enter Your Last Name:"
read lname
echo "Your First Name is: $fname"
echo " Your Last Name is: $lname"
```

Execution & Output:

```
$ chmod 777 simple.sh
$ sh simple.sh
Enter Your First Name: Henry
Enter Your Last Name: Ford
Your First Name is: Henry
Your Last Name is: Ford
```

3. USING COMMAND LINE ARGUMENTS

Shell script also accept arguments from the command line.

They can, therefore, run non-interactively and be used with redirection and pipelines.

When arguments are specified with a shell script, they are assigned to positional parameters.

The shell uses following parameters to handle command line arguments-

Shell parameter	Significance
$#	Number of arguments specified in command line
$0	Name of executed command
$1, $2, …	Positional parameters representing command line arguments
$*	Complete set of positional parameters as a single string
"$@"	Each quoted string is treated as a separate arguments, same as $*

Example:

```
#!/bin/sh
#Shell Script to demonstrate command line arguments - sample.sh
echo "The Script Name is: $0"
echo "Number of arguments specified is: $#"
echo "The arguments are: $*"
echo "First Argument is: $1"
echo "Second Argument is: $2"
echo "Third Argument is: $3"
echo "Fourth Argument is: $4"
echo "The arguments are: $@"
```

Execution & Output:

```
$ sh sample.sh welcome to hit nidasoshi [Enter]
The Script Name is: sample.sh
Number of arguments specified is: 4
The arguments are: welcome to hit nidasoshi
First Argument is: welcome
Second Argument is: to
Third Argument is: hit
Fourth Argument is: nidasoshi
The arguments are: welcome to hit nidasoshi
```

4. exit AND EXIT STATUS OF COMMAND

C program and shell scripts have a lot in common, and one of them is that they both use the same command (or function in c) to terminate a program.It has the name exit in the shell and exit( ) in C.

The command is usually run with a numeric arguments:

- exit 0 #Used when everything went fine
- exit 1 #Used when something went wrong

The shell offers a variable $? and a command test that evaluates a command's exit status.

The parameter $? stores the exit status of the last command.

It has the value 0 if the command succeeds and a non-zero value if it fails.

This parameter is set by exit's argument.

Examples:

- $ grep director emp.lst >/dev/null; echo $?
  0 #Success
- $ grep director emp.lst >/dev/null; echo $?
  1 #Failure – in finding pattern

5. THE LOGICAL OPERATORS && AND || - CONDITIONAL EXECUTION

The shell provides two operators that allow conditional execution – the && and | |.

Examples:

- $ date **&&** echo "Date Command Executed Successfully!"
  Sun Jan 13 15:40:13 IST 2013
  Date Command Executed Successfully!
- $ grep 'director' emp.lst **&&** echo "Pattern found in File!"
  1234 | Henry Ford | director | Marketing | 12/12/12 | 25000
  Pattern found in File!
- $ grep 'manager' emp.lst **| |** echo "Pattern not-found in File!"
  Pattern not-found in File!

6. THE if CONDITIONAL

The if statement makes two-way decisions depending on the fulfillment of a certain condition.

In the shell, the statement uses the following forms-

if command is successful then execute commands else execute commands fi	If command is successful then execute commands fi	If command is successful then execute commands elif command is successful then execute commands else execute commands fi
Form 1	Form 2	Form 3

Example:

```
#!/bin/sh
#Shell script to illustrate if conditional
if grep 'director' emp.lst >/dev/null
then
echo “Pattern found in File!”
else
echo “Pattern not-found in File!”
fi
```

7. USING test AND [ ] TO EVALUATE EXPRESSIONS

When you use **if** to evaluate expressions, you need the **test** statement because the true or false values returned by expression's can't be directly handled by **if**.

test uses certain operators to evaluate the condition on its right and returns either a true or false exit status, which is then used by if for making decision.

test works in three ways:

- Compares two numbers
- Compares two strings or a single one for a null value.
- Checks a file's attributes

test doesn't display any output but simply sets the parameter $?.

Numeric Comparison

Numerical Comparison operators used by **test:**

Operator	Meaning
-eq	Equal to
-ne	Not equal to
-gt	Greater than
-ge	Greater than or equal to
-lt	Less than
-le	Less than or equal to

The numerical comparison operators used by test always begins with a - (hyphen), followed by a two-letter string, and enclosed on either side by whitespace.

Examples:

```
$ x=5, y=7, z=7.2
$ test $x -eq $y; echo $?
1                                   # Not Equal

$ test $x -lt $y; echo $?
0                                   #True

$ test $y -eq $z
0                                   #True- 7.2 is equal to 7
```

The last example proves that numeric comparison is restricted to integers only.

The [ ] is used as shorthand for test.

Hence, above example may be re-written as-

test $x -eq $y **or** [$x -eq $y] #Both are equivalent

String Comparison

test can be used to compare strings with yet another set of operators.

The below table shows string tests used by test-

Test	**True if**
s1 = s2	String s1 = s2
s1 != s2	String s1 is not equal to s2
-n stg	String stg is not a null string
-z stg	String stg is a null string
stg	String stg is assigned and not a null string
s1 == s2	String s1 = s2 (Korn and Bash only)

Example:

```
#!/bin/sh
#Shell script to illustrate string comparison using test – strcmp.sh
if [ $# -eq 0 ]; then
echo "Enter Your Name: \c"; read name;
```

```
if [ -z $name ]; then
echo "You have not entered your name! "; exit 1;
else
echo "Your Name From Input line is : $name "; exit 1;
fi
else
echo "Your Name From Command line is : $1 " ;
fi
```

Execution & Output:

```
$ chmod 777 strcmp.sh
$ sh strcmp.sh Henry
Your Name From Command line is : Henry

$ sh strcmp.sh
Enter Your Name: Smith [Enter]
Your Name From Input line is : Smith

$ sh strcmp.sh
Enter Your Name: [Enter]
You have not entered your name!
```

8. THE case CONDITIONAL

The case statement is similar to switch statement in C.

The statement matches an expression for more than one alternative, and permit multi-way branching.

The general syntax of the case statement is as follows:

```
case expression in
pattern1) command1 ;;
pattern2) command2 ;;
pattern3) command3 ;;
…….....
esac
```

Example:

```
#!/bin/sh
#Shell script to illustrate CASE conditional – menu.sh
echo "\t MENU\n 1. List of files\n 2. Today's Date\n 3. Users of System\n 4. Quit\n";
echo "Enter your option: \c";
read choice

case "$choice" in
1) ls -l ;;
2) date ;;
3) who ;;
4) exit ;;
*) echo "Invalid Option!"
esac
```

Execution & Output:

```
$ chmod 777 menu.sh
$ sh menu.sh
MENU
1. List of files
2. Today's Date
3. Users of System
4. Quit

Enter your option: 2
Sun Jan 13 15:40:13 IST 2013
```

9. expr: COMPUTATION AND STRING HANDLING

Shell does not have any computing features, but it rely on external **expr** command for that purpose.

This **expr** command combines two functions in one:

- Performs arithmetic operations on integers
- Manipulates strings

Computation

expr can perform the four arithmetic operations as well as the modulus (remainder) function.

Examples:

$ expr 3 + 5

8

$ x=8 y=4

$ expr $x + $y

12

$ expr $x - $y

4

$ expr $x * $y #Asterisk has to be escaped to prevent from metacharacter

32

$ expr $x / $y

2

$ expr $x % $y

0

expr is often used with command substitution to assign a variable.

For example, you can set a variable z to the sum of two numbers:

$ x=6 y=2; z=expr `$x + $y`

$ echo $z

8

String Handling

For manipulating strings, **expr** uses two expressions separated by a colon.

The string to be worked upon is placed on the left of the :, and RE is placed on its right.

Depending on the composition of the expression, expr can perform three important string functions:

- Determine the length of the string
- Extract a substring
- Locate the position of a character in a string

Examples:

$ expr “abcdefghijkl” : '.*'

12 #Length of the string = number of characters

$ stg=2013

$ expr “$stg” : '..\(..\)'

13 #Extracts last two characters – as substring

$stg=abcdefgh

$ expr “$stg” : '[^d]*d'

4 #Locate position of character d in stg

10. while: LOOPING

The while statement should be quite familiar to many programmers.

It repeatedly performs a set of instructions until the control commands returns a true exit status.

The general syntax of this command is as follows:

```
while condition is true
do
commands
done
```

The commands enclosed by do and done are executed repeatedly as long as condition remains true.

Example:

```
#!/bin/sh
#Shell script to illustrate while loop – while.sh
answer=y;
while [ "$answer" = "y" ]
do
echo "Enter Branch Code and Name: \c"
read code name #Read both together
echo "$code|$name" >> newlist                    #Append a line to newlist
echo "Enter anymore (y/n)? \c"
read anymore
case $anymore in
y*|Y*) answer=y ;;                               #Also accepts yes, YES etc.
n*|N*) answer=n ;;                               #Also accepts no, NO etc.
*) answer=n ;;                                   #Any other reply means no
esac
done
```

Execution & Output:

```
$ chmod 777 while.sh
$ sh while.sh
Enter Branch Code and Name: CS COMPUTER [Enter]
Enter anymore (y/n)? y [Enter]
Enter Branch Code and Name: EC ELECTRONICS [Enter]
Enter anymore (y/n)? n [Enter]
$ cat newlist
```

CS|COMPUTER

EC|ELECTRONICS

NOTE:

The shell also offers an **until** statement which operates with a reverse logic used in **while**.

With until, the loop body is executed as long as the condition remains false.

11. for: LOOPING WITH A LIST

The shell's for loop differs in structure from the ones used in other programming language.

Unlike while and until, for doesn't test a condition, but uses a list instead.

The general syntax of for loop is as follows-

```
for variable in list
do
commands
done
```

The loop body also uses the keywords do and done, but the additional parameters here are variable and list.

Each whitespace-separated word in list is assigned to variable in turn, and commands are executed until list is exhausted.

A simple example can help you understand for loop better:

Example:

```
#!/bin/sh
#Shell script to illustrate use of for loop – forloop.sh
for file in chap1 chap2 chap3 chap4
do
cp $file ${file}.bak
echo “$file copied to $file.bak”
done
```

Execution & Output:

```
$ chmod 777 forloop.sh
$ sh forloop.sh
chap1 copied to chap1.bak
chap2 copied to chap2.bak
chap3 copied to chap3.bak
chap4 copied to chap4.bak
```

Possible sources of the list:

1. list from variables - $ for var in $x $y $z
2. list from command substitution- $ for var in `cat foo`
3. list from wild-cards- $ for file in *.htm *.html
4. list from positional parameters- $ for var in "$@"

12. set AND shift: MANIPULATING THE POSITIONAL PARAMETERS

set: Set the positional parameters

set assigns its arguments to the positional parameters $1, $2 and so on.

This feature is especially useful for picking up individual fields from the output of a program.

Example:

$ set `date` #Output of date command assigned to positional parameters $1, $2 & so on.

$ echo $*

Sun Jan 13 15:40:13 IST 2013

$ echo "The date today is $2 $3 $6"

The date today is Jan 13 2013

shift: Shifting Arguments Left

shift transfers the contents of a positional parameters to its immediate lower numbered one.

This is done as many times as the statement is called.

Example:

$ set `date`

$ echo $*

Sun Jan 13 15:40:13 IST 2013

$ echo $1 $2 $3

Sun Jan 13

$ shift #Shifts 1 place

Jan 13 15:40:13

$ echo $1 $2 $3

$ shift 2 #Shifts 2 places

$ echo $1 $2 $3

15:40:13 IST 2013

13. THE SAMPLE DATABASE

The following **emp.lst** is an reference database file to understand features of several UNIX commands including filters, text editors and shell programming. It's a good idea to understand the organization of emp.lst database file.

$ cat emp.lst

```
2233|a. k. shukla          |g. m.      |sales       |12/12/52|6000
9876|jai sharma            |director   |production  |12/03/50|7000
5678|sumit chakrobarty     |d. g. m.   |marketing   |19/04/43|6000
2365|barun sengupta        |director   |personnel   |11/05/47|7800
5423|n. k. gupta           |chairman   |admin       |30/08/56|5400
1006|chanchal singhvi      |director   |sales       |03/09/38|6700
6213|karuna ganguly        |g. m.      |accounts    |05/06/62|6300
1265|s. n. dasgupta        |manager    |sales       |12/09/63|5600
4290|jayant choudhury      |executive  |production  |07/09/50|6000
2476|anil aggarwal         |manager    |sales       |01/05/59|5000
6521|lalit chowdury        |director   |marketing   |26/09/45|8200
3212|shyam saksena         |d. g. m.   |accounts    |12/12/55|6000
3564|sudhir Agarwal        |executive  |personnel   |06/07/47|7500
2345|j. b. saxena          |g. m.      |marketing   |12/03/45|8000
0110|v. k. agrawal         |g. m.      |marketing   |31/12/40|9000
```

This is a text file designed in fixed format and containing a personnel database.

There are 15 lines in the file, where each line has six fields separated from one another by the delimiter |.

The details of an employee are stored in one line.

A person is identified by emp_id, name, designation, department, date of birth and salary.

14. head: DISPLAYING THE BEGINNING OF A FILE

The head command, as the name implies, displays the top of the file.

When used without an option, it displays the first ten lines of the file.

Example:

$ head -n 3 emp.lst #Displays first 3 lines from emp.lst

```
2233|a. k. shukla          |g. m.      |sales       |12/12/52|6000
9876|jai sharma            |director   |production  |12/03/50|7000
5678|sumit chakrobarty     |d. g. m.   |marketing   |19/04/43|6000
```

15. tail: DISPLAYING THE END OF A FILE

Complementing its head counterpart, the tail command displays the end of the file.

Like an head, it displays last ten lines of file by default.

Example:

$ tail -n 3 emp.lst #Displays last 3 lines from emp.lst

3564|sudhir Agarwal |executive |personnel |06/07/47|7500

2345|j. b. saxena |g. m. |marketing |12/03/45|8000

0110|v. k. agrawal |g. m. |marketing |31/12/40|9000

16. cut: SLITTING A FILE VERTICALLY

Cutting Columns (-c)

To extract specific columns, you need to follow the -c option with a list of column numbers, delimited by comma.

Examples:

$ head -n 5 emp.lst | tee shortlist #tee saves output in file shortlist & also display it on the terminal

2233|a. k. shukla |g. m. |sales |12/12/52|6000

9876|jai sharma |director |production |12/03/50|7000

5678|sumit chakrobarty |d. g. m. |marketing |19/04/43|6000

2365|barun sengupta |director |personnel |11/05/47|7800

5423|n. k. gupta |chairman |admin |30/08/56|5400

$ cut -c 6-22,24-32 shortlist

a. k. shukla g.m.

jai sharma director

sumit chakrobartyd. g. m.

barun sengupta director

n. k. gupta chairman

Cutting Fields (-f)

The -c option is useful for fixed-length line. Most UNIX files don't contain fixed-length lines.

To extract useful data from these files you need to cut fields rather than columns.

The cut used tab as default field delimiter, but can also work with different delimiter.

The two options need to be used here:

-d for the filed delimiter

-f for the field list.

The below example shows how to cut the second and third fields from file shortlist

example:

$ cut -d \| -f 2,3 shortlist

a. k. shukla |g. m.

jai sharma |director

sumit chakrobarty |d. g. m.

barun sengupta |director

n. k. gupta |chairman

17. paste: PASTING FILES

What you cut with cut can be pasted back with the paste command – but vertically rather than horizontally.

You can view two files side by side by pasting them.

Example:

$ cut -d \| -f 2,3 shortlist | tee cutlist1

a. k. shukla |g. m.

jai sharma |director

sumit chakrobarty |d. g. m.

barun sengupta |director

n. k. gupta |chairman

$ cut -d \| -f 5,6 shortlist | tee cutlist2

12/12/52|6000

12/03/50|7000

19/04/43|6000

11/05/47|7800

30/08/56|5400

$ paste cutlist1 cutlist2 #Combines two files vertically

a. k. shukla |g. m. 12/12/52|6000

jai sharma |director 12/03/50|7000

sumit chakrobarty |d. g. m. 19/04/43|6000

barun sengupta |director 11/05/47|7800

n. k. gupta |chairman 30/08/56|5400

18. sort: ORDERING A FILE

Sorting is the ordering of data in ascending or descending sequence.

The sort command orders a file.

Like cut, it identifies fields and it can sort on specified fields.

By default, sort reorders lines in ASCII collating sequence – whitespace first, then numerals, uppercase letters and finally lowercase letters.

Example:

$ sort shortlist

2233|a. k. shukla |g. m. |sales |12/12/52|6000

2365|barun sengupta |director |personnel |11/05/47|7800

5423|n. k. gupta |chairman |admin |30/08/56|5400

5678|sumit chakrobarty |d. g. m. |marketing |19/04/43|6000

9876|jai sharma |director |production |12/03/50|7000

Sort Options

Option	Description
-tchar	Uses delimiter char to identify fields
-k n	Sorts on nth filed
-k m, n	Starts sort on nth column of mth field
-u	Removes repeated lines
-n	Sorts numerically
-f	Case-insensitive sort
-c	Checks if file is sorted

19. FILE SYSTEMS AND INODES

The hard disk is split into distinct partitions (or slices), with a separate file system in each partition(or slice).

Every file system has a directory structure headed by root.

If you have three file systems on one hard disk, then they will have three separate root directories.

When the system is up, we see only a single file system with a single root directory.

Of these multiple file systems, one of them is considered to be the main one, and contains most of the essential files of the UNIX system. This is the root file system, which is more equal than others in at least one respect; its root directory is also the root directory of the combined UNIX system. At the time of booting, all secondary file systems mount(attach) themselves to the main file system, creating the illusion of a single file system to the user.

Every file is associated with a table that contains all that you could possibly need to know about a file – except its name and contents. This table is called the inode (shortened from index node) and is accessed by the inode number. The inode contains the following attributes of a file:

- file type – regular, directory, device etc.
- file permissions – the nine permissions and three more.
- Number of links – the number of aliases the file has.
- 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 pointers 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. It's the directory that stores the inode number along with the filename. When you use a command with a filename as 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.

Every file system has a separate portion set aside for storing inodes, where they are laid out in a contiguous manner. This area is accessible only to the kernel. The inode number is actually the position of the inode in this area. The kernel can locate inode number of any file using simple arithmetic. Since a UNIX machine usually comprises multiple file systems, you can conclude that the inode number for a file is unique in a single file system.

The ls command can be used with -i(inode) option to know the inode number of any file present in the UNIX file system.

Example:

$ ls -il test.c

13109062 -rwxrwxrwx 1 chandrakant chandrakant 639 Dec 10 13:18 test.c

The file test.c has the inode number 13109062. This inode number is unique in this file system.

20. HARD LINKS

Why is the filename not stored in the node? So that a file can have multiple filenames.

When that happens, we say the file has more than one link.

We can then access file by any of its links.

All names provided to a single file have the same inode number.

Example:

$ ls -li backup.sh restore.sh

478274 -rwxrwxrwx 2 chandrakant chandrakant 163 Dec 10 13:18 backup.sh

478274 -rwxrwxrwx 2 chandrakant chandrakant 163 Dec 10 13:18 restore.sh

Here, both “files” indeed have the same inode number, so there's actually only one file with a single copy on disk.

We can't really refer to them as two “files”, but only as two “filenames”.

This file simply has two aliases, changes made in one alias (link) are automatically available in others.

There are two entries for this file in the directory, both having the same inode number.

ln: CREATING HARD LINKS

A file is linked with the ln(link) command, which takes two filenames as arguments.

The ln command can create both a hard link and a soft link.

The following command links emp.lst with employee:

Examples:

$ ln emp.lst employee #Ensure employee must not exist

$ ls -li emp.lst employee

13110850 -rw-rw-r-- 2 chandrakant chandrakant 811 Jan 21 15:41 employee

13110850 -rw-rw-r-- 2 chandrakant chandrakant 811 Jan 21 15:41 emp.lst

Where to use Hard Links?

1. Whenever you reorganized your directory structure and moved file from one place to another, create and use hard link to refer the moved file with same old pathname in all the programs in that you are previously using the same file, without modifying the program.
2. Hard links provides some protection against accidental deletion, especially when they exist in different directories.
3. Because of links, we don't need to maintain two programs as two separate disk files if there is very little difference between them.

21. SYMBOLIC LINKS AND ln

The hard link have two limitations:

- You can't have two linked filenames in two file systems. In other words, you can't link a filename in the /usr file system to another in the /home file system.
- You can't link a directory even within the same file system.
- This serious limitation was overcome by symbolic links.
- Until now, we have divided files into three categories – ordinary, directory and device.
- The symbolic link is the fourth file type considered in this text.
- Unlike the hard link, a symbolic link doesn't have the file's contents, but simply provides the pathname of the file that actually has the contents.
- A symbolic link is also known as soft link.
- Windows shortcuts on desktops are more like symbolic links.
- Symbolic links can also be used with relative pathnames.
- Unlike hard links, they can also span multiple file systems and also link directories.

How to create soft link?

The ln command with -s option creates symbolic links.

Example:

$ ln -s emp.lst employee #Ensure employee must not exist

$ ls -li emp.lst employee

13110850 -rw-rw-r-- 1 chandrakant chandrakant 80 Jan 21 15:41 emp.lst

13112164 lrwxrwxrwx 1 chandrakant chandrakant 7 Jan 26 19:34 employee -> emp.lst

Here, you can identify symbolic link by the character l seen in the second (permissions) field of ls -l output.

The pointer notation ->emp.lst suggest that employee contains the pathname for the filename emp.lst.

It's emp.lst, not employee, that actually contains the data.

When you use cat employee, you don't actually open the symbolic link, employee, but the file the link points to.

22. THE DIRECTORY

- A directory has its own permissions, owners and links.
- However, the significance of the file attributes change a great deal when applied to a directory.
- For example, size of directory is not related to the size of files that exist in the directory, but rather to the number of files housed by it.
- The higher the number of files, the larger is the directory size.

Examples:

$ mkdir sample #Create new directory, sample

$ ls -l -d sample #Check directory attributes by ls -l -d option

drwxrwxr-x 2 chandrakant chandrakant 4096 Jan 26 19:51 sample

Here,

first character in permissions field is d which shows file is of directory type.

Read Permission

- Read permission for a directory means that the list of filenames stored in that directory is accessible.
- Since ls reads the directory to display filenames, if a directory's read permission is removed, ls won't work.
- Consider removing the read permission first from the directory sample.

Example:

$ ls -ld sample #Before removing read permission of sample

drwxrwxr-x 2 chandrakant chandrakant 4096 Jan 26 19:51 sample

$ chmod -r sample #Removes read permission of sample

$ ls -ld sample #After removing read permission of sample

sample: Permission denied

Write Permission

- Be aware that you can't write to a directory file; only kernel can do that.
- Write permission for a directory implies that you are permitted to create or remove files in it.
- To try that out, restore the read permission and remove the write permission from the directory sample.

Example:

$ chmod 555 sample #Restore read permission $ Removes write permission

$ ls -ld sample

dr-xr-xr-x 2 chandrakant chandrakant 4096 Jan 26 19:51 sample

$ cp emp.lst sample

cp: cannot create regular file 'sample/emp.lst' : Permission denied

The directory sample doesn't have write permission; you can't create copy or delete a file in it.

Execute Permission

- Executing a directory just doesn't make any sense, so what does its execute privilege mean?
- It only means that a user can “pass through” the directory in searching for subdirectories.
- When you use a pathname with any command – **cat /home/chandrakant/sample/emp.lst** you need to have execute permission for each of the directories involved in the complete pathname.
- The directory home contains entry for chandrakant, and the directory chandrakant contains entry for sample, and so forth.
- If a single directory in this pathname doesn't have execute permission, then it can't be searched for the name of the next directory.
- That's why the execute privilege of a directory is often referred to as the **search** permission.

Example:

$ chmod 666 sample #Restore write permission & Removes execute permission

$ ls -ld sample

drw-rw-rw- 2 chandrakant chandrakant 4096 Jan 26 19:51 sample

$ cd sample

cd: sample: Permission denied

23. umask: DEFAULT FILE AND DIRECTORY PERMISSIONS

When you create files and directories, the permissions assigned to them depend on the system's default settings.

The UNIX has the following default permissions for all files and directories:

- rw-rw-rw- (octal 666) for regular files.
- rwxrwxrwx (octal 777) for directories.

However, you don't see these permissions when you create a file or directory.

Actually, this default is transformed by subtracting the user mask from it to remove one or more permissions.

To understand what this means, let's evaluate the current value of the mask by using umask without arguments:

Example:

$ umask

022

This is an octal number which has to be subtracted from the system default to obtain the actual default.

- This becomes 644 (666 – 022) for ordinary files.
- This becomes 755 (777 – 022) for directories.

24. MODIFICATION AND ACCESS TIMES

A UNIX file has three time stamps associated with it-

- Time of last modification shown by ls -l
- Time of last access shown by ls -lu
- Time of last inode modification shown by ls -lc

Whenever you write to a file, the time of last modification is updated in the file's inode.

A directory can be modified by changing its entries – by creating, removing and renaming files in the directory.

Even though ls -l and ls -u shows the last modification and access time, respectively, the sort order remains standard, i.e. ASCII.

However, when you add the -t option to -l or -lu, the files are actually displayed in order of the respective time stamps:

ls -lt Displays file listing in order of their modification time

ls -lut Displays file listing in order of their access time.

touch: Changing the Time Stamps

You may sometimes need to set the modification and access times to predefined values.

The touch command changes these times, and is used in the following manner:

touch options expression filename(s)

when touch is used without option or expression, both times are set to the current times.

The -m and -a options change the modification and access times, respectively.

The expression consists of an eight-digit number using the format MMDDhhmm (month, day, hour and minute).

Examples:

$ touch 03161430 emp.lst #Change both modification and access time

$ ls -l emp.lst

-rw-rw-r-- 1 chandrakant chandrakant 80 Mar 16 14:30 emp.lst

$ touch -m 03161430 emp.lst #Change modification time

$ ls -l emp.lst

-rw-rw-r-- 1 chandrakant chandrakant 80 Mar 16 14:30 emp.lst

$ touch -a 03161430 emp.lst #Change access time

$ ls -lu emp.lst

-rw-rw-r-- 1 chandrakant chandrakant 80 Mar 16 14:30 emp.lst