NAVODAYA INSTITUTE OF TECHNOLOGY, RAICHUR

DEPARMENT OF COMPUTER SCIENCE & ENGINEERING

MODULE 2

File Attributes & Permissions

The ls command with options:

The ls command is to obtain a list of all file names in the current directory. ls with options:

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

Listing File Attributes(ls –l):

ls command is used to obtain a list of all filenames in the current directory. The output in UNIX lingo is often referred to as the listing. Sometimes we combine this option with other options for displaying other attributes, or

ordering the list in a different sequence. ls look up the file's inode to fetch its attributes. It lists seven attributes of all files in the current directory and they:

1. File type & Permission:
   The first column shows the type and permissions associated with each file. The first character in this column is mostly a -, which indicates that file is an ordinary one.
2. Links:
   The second column indicates the number of links associated with the file. This is actually the number of filenames maintained by the system of that file.
3. Ownership:
   When you create a file, you automatically become its owner. The third column shows kumar is the owner of all of these files. The owner has full authority to tamper with a files contents and permissions—a privilege not available with others except the root user.
4. Group Ownership:
   The fourth column represents the group owner of the file. Every user is attached to a group owner. Every member of that group can access the file depending on the permission assigned.
5. File size:
   File size in bytes is displayed. It is the number of character in the file rather than the actual size occupied on disk.
6. Last modification time:
   Last modification time is the next field. If you change only the permissions or ownership of the file, the modification time remains unchanged. If at least one character is added or removed from the file then this field will be updated.
7. File name:
   The last column displays the filenames arranged in ASCII collating sequence.

For Ex:

$ ls -l total
72
-rw-r--r-- 1 kumar metal 19514 may 10 13:45 chap01 -rw-r--r--
2 kumar metal 19555 may 10 15:45 chap02 drwxr-xr-x 2

kumar metal 512 may 09 12:55 helpdir drwxr-xr-x 3 kumar metal 512 may 09 11:05 progs

Changing File Permission:

A file or a directory is created with a default set of permissions, which can be determined by umask. Let us assume that the file permission for the created file is -rw-r-- r--. Using chmod command, we can change the file permissions and allow the owner to execute his file. The command can be used in two ways:

1. In a **relative** manner by specifying the changes to the current permissions.
2. In an **absolute** manner by specifying the final permissions.

Relative Permissions:

- chmod only changes the permissions specified in the command line and leaves the other permissions unchanged.
- Its syntax is:

  chmod category operation permission filename(s) •

chmod takes an expression as its argument which contains:

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

- Category : u – user g – group o – others a - all (ugo)
- operations : + assign - remove = absolute
- permissions: r – read w – write x - execute

Example:

- Initially,

  -rw-r—r-- 1 kumar metal 1906 sep 23:38 xstart

  $chmod u+x xstart

  -rwxr—r-- 1 kumar metal 1906 sep 23:38 xstart

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

$chmod ugo+x xstart or chmod a+x xstart or chmod +x xstart

$ls –l xstart

-rwxr-xr-x 1 kumar metal 1906 sep 23:38 xstart

- chmod accepts multiple file names in command line

  $chmod u+x note note1 note3

- Let Initially,

  -rwxr-xr-x 1 kumar metal 1906 sep 23:38 xstart

  $chmod go-r xstart

  Then it becomes

  $ls –l xstart

  -rwx—x--x 1 kumar metal 1906 sep 23:38 xstart

Absolute Permissions:

- Here, we need not to know the current file permissions. We can set all nine permissions explicitly. A string of three octal digits is used as an expression. The permission can be represented by one octal digit for each category. For each category, we add octal digits. If we represent the permissions of each category by one octal digit, this is how the permission can be represented:

  Read permission – 4 (octal 100)
  Write permission – 2 (octal 010)
  Execute permission – 1 (octal 001)

Octal	Permissions	Significance
0	---	No permissions.
1	--x	Execute only.
2	-w-	Write only
3	-wx	Write & Execute
4	r--	Read only.
5	r-x	Read & Execute.
6	rw-	Read & Write.
7	rwx	Read, write, & Execute

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

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

- Using absolute permission, we have:

  $chmod 666 xstart
  $chmod 644 xstart
  $chmod 761 xstart

- will assign all permissions to the owner, read and write permissions for the group and only execute permission to the others.
- 777 signify all permissions for all categories, but still we can prevent a file from being deleted.
- 000 signifies absence of all permissions for all categories, but still we can delete a file.
- It is the directory permissions that determine whether a file can be deleted or not.
- Only owner can change the file permissions. User cannot change other user‘s file‘s permissions.
- But the system administrator can do anything.

Directory Permissions:

- It is possible that a file cannot be accessed even though it has read permission, and can be removed even when it is write protected. The default permissions of a directory are,

rwxr-xr-x (755)

- A directory must never be writable by group and others .
- Example:

  $mkdir c_progs $ls
  –ld c_progs
  drwxr-xr-x 2 kumar metal 512 may 9 09:57 c_progs

- If a directory has write permission for group and others also, be assured that every user can remove every file in the directory. As a rule, you must not make directories universally writable unless you have definite reasons to do so.

Recursively Changing File Permissions:

It’s possible to make chmod descend a directory hierarchy and apply the expression to every file and subdirectory it finds. This is done with the –R(recursive) option:

$chmod –R a+x shell_scripts

This makes all files and subdirectories found in the tree-walk executable by all users. We can provide multiple directory and filenames.

If we want to use chmod on your home directory tree then “cd” to it and use it in one of these ways:

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


The Shell Interpretive Cycle

Introduction:

The shell sits between you and the operating system, acting as a command interpreter. It reads your terminal input and translates the commands into actions taken by the system. The shell is analogous to command.com in DOS. When you log into the system you are given a default shell. When the shell starts up it reads its startup files and may set environment variables, command search paths, and command aliases, and executes any commands specified in these files. The original shell was the Bourne shell, sh. Every Unix platform will either have the Bourne shell, or a Bourne compatible shell available.

Numerous other shells are available. Some of the more well known of these may be on your Unix system: the Korn shell, ksh, by David Korn, C shell, csh, by Bill Joy and the Bourne Again SHell, bash, from the Free Software Foundations GNU project, both based on sh, the T-C shell, tcsh, and the extended C shell, cshe, both based on csh.

Even though the shell appears not to be doing anything meaningful when there is no activity at the terminal, it swings into action the moment you key in something.

The following activities are typically performed by the shell in its interpretive cycle:

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

After a command is entered, the shell scans the command line for meta characters and expands abbreviations (like the * 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 the command execution is complete, the prompt reappears and the shell returns to its waiting role to start the next cycle. You are free to enter another command.

Pattern Matching-Wild Cards:

A pattern is framed using ordinary characters and a meta character (like *) using well- defined rules. The pattern can then be used as an argument to the command, and the shell will expand it suitably before the command is executed.

The meta characters that are used to construct the generalized pattern for matching filenames belong to a category called wild-cards. The following table lists them:

Wild Card	Matches
*	Any number of characters including none.
?	A single character.
[ijk]	A single character – either an i, j or k.
[x-z]	A single character that is within the ASCII range of characters x and z.
[!ijk]	A single character that is not an i, j or k (Not in C shell).
[!x-z]	A single character that is not within the ASCII range of the characters x and z (Not in C Shell).
{pat1,pat2...}	Pat1, pat2, etc. (Not in Bourne shell).

Examples:

To list all files that begin with chap, use, $ls chap*

To list all files whose filenames are six character long and start with chap, use , $ls chap??

Note: Both * and ? operate with some restrictions. for example, the * doesn't match all files beginning with a . (dot) or the / of a pathname. If you wish to list all hidden filenames in your directory having at least three characters after the dot, the dot must be matched explicitly.

$ ls .???*

However, if the filename contains a dot anywhere but at the beginning, it need not be matched explicitly.

Similarly, these characters don't match the / in a pathname. So, you cannot use, $cd /usr?local to change to /usr/local.

The character class:

You can frame more restrictive patterns with the character class. The character class comprises a set of characters enclosed by the rectangular brackets, [and], but it matches a single character in the class. The pattern [abd] is character class, and it matches a single character – an a,b or d.

Examples:

$ls chap0[124] - Matches chap01, chap02, chap04 and lists if found.
$ls chap[x-z] - Matches chapx, chapy, chapz and lists if found.

You can negate a character class to reverse matching criteria. For example:

To match all filenames with a single-character extension but not the .c ot .o files, use *.[!co]

To match all filenames that don't begin with an alphabetic character, use [!a-zA-Z]*

Escaping & Quoting:

Escaping is providing a \ (backslash) before the wild-card to remove (escape) its special meaning.

For instance, if we have a file whose filename is chap* (Remember a file in UNIX can be names with virtually any character except the / and null), to remove the file, it is dangerous to give command as rm chap*, as it will remove all files beginning with chap. Hence to suppress the special meaning of *, use the command rm chap*.

To list the contents of the file chap0[1-3], use,

$cat chap0\[1-3\]

A filename can contain a whitespace character also. Hence to remove a file named My Documend.doc, which has a space embedded, a similar reasoning should be followed:

rm My\ Document.doc

Quoting is enclosing the wild-card, or even the entire pattern, within quotes. Anything within these quotes (barring a few exceptions) are left alone by the shell and not interpreted. When a command argument is enclosed in quotes, the meanings of all enclosed special characters are turned off.

Examples:

$rm chap*	Removes files chap*
$rm “My Document.doc”	Removes file My Document.doc

Redirection: The three standard files

The shell associates three files with the terminal – two for display and one for the keyboard. These files are streams of characters which many commands see as input and output. When a user logs in, the shell makes available three files representing three streams. Each stream is associated with a default device: -

Standard input: The file (stream) representing input, connected to the keyboard.

Standard output: The file (stream) representing output, connected to the display.

Standard error: The file (stream) representing error messages that emanate from the command or shell, connected to the display.

The standard input can represent three input sources:

- The keyboard, the default source.
- A file using redirection with the < symbol.
- Another program using a pipeline.

The standard output can represent three possible destinations:

- The terminal, the default destination.
- A file using the redirection symbols > and >>.
- As input to another program using a pipeline.

A file is opened by referring to its pathname, but subsequent read and write operations identify the file by a unique number called a file descriptor. The kernel maintains a table of file descriptors for every process running in the system. The first three slots are generally allocated to the three standard streams as,

0 – Standard input
1 – Standard output
2 – Standard error

These descriptors are implicitly prefixed to the redirection symbols.

Examples:

Assuming file2 doesn't exist, the following command redirects the standard output to file myOutput and the standard error to file myError.

$ls –l file1 file2 1>myOutput 2>myError

Filters: Using both standard input and standard output

UNIX commands can be grouped into four categories viz.,
Directory-oriented commands like mkdir, rmdir and cd, and basic file handling commands like cp, mv and rm use neither standard input nor standard output.
2. Commands like ls, pwd, who etc. don't read standard input but they write to standard output.
3. Commands like lp that read standard input but don't write to standard output.
4. Commands like cat, wc, cmp etc. that use both standard input and standard output.

Commands in the fourth category are called filters. Note that filters can also read directly from files whose names are provided as arguments.

Example:
To perform arithmetic calculations that are specified as expressions in input file calc.txt and redirect the output to a file result.txt, use

$bc < calc.txt > result.txt6.

Connecting Commands: Pipe

With piping, the output of a command can be used as input (piped) to a subsequent command.

$ command1 | command2

Output from command1 is piped into input for command2.

This is equivalent to, but more efficient than:

```
$ command1 > temp
$ command2 < temp
$ rm temp
```

Examples:

$ ls -l | wc –l	Displays number of file in current directory
$ who | wc –l	Displays number of currently logged in users

grep: Searching for a pattern

You often need to search a file for a pattern, either to see the lines containing (or not containing) it or to have it replaced with something else. This chapter discusses two important filters that are specially suited for these tasks- grep and sed. This chapter also takes up one of the fascinating features of UNIX – regular expressions (RE).

To discuss all the examples in this chapter we use following emp.lst as the reference 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
```

grep scans its input for a pattern displays lines containing the pattern, the line numbers or filenames where the pattern occurs. The command uses the following syntax:

$grep options pattern filename(s)

grep searches for pattern in one or more filename(s), or the standard input if no filename is specified.

The first argument (except the options) is the pattern and the remaining arguments are filenames.

Examples:
$ grep “sales” emp.lst

2233|a. k. shukla |g. m. |sales |12/12/52|6000
1006|chanchal singhvi |director |sales |03/09/38|6700
1265|s. n. dasgupta |manager |sales |12/09/63|5600
2476|anil aggarwal |manager |sales |01/05/59|5000

here, grep displays 4 lines containing pattern as “sales” from the file emp.lst.

$ grep president emp.lst #No quoting necessary here
$_ #No pattern (president) found

here, grep silently returns the prompt because no pattern as “president” found in file emp.lst.

$ grep “director” emp1.lst emp2.lst

emp1.lst: 9876|jai sharma |director |production |12/03/50|7000
emp1.lst: 2365|barun sengupta |director |personnel |11/05/47|7800
emp1.lst: 1006|chanchal singhvi |director |sales |03/09/38|6700
emp2.lst: 6521|lalit chowdury |director |marketing |26/09/45|8200

Here, first column shows file name. when grep is used with multiple filenames, it displays the filenames along with the output.

grep options:
The below table shows all the options used by grep.

Option	Significance
-i	Ignores case for matching.

-v	Doesn't display lines matching expression
-n	Displays line numbers along with lines
-c	Displays count of number of occurrences
-l	Displays list of filenames only.
-e exp	Matches multiple patterns
-f filename	Takes patterns from file, one per line
-E	Treats patterns as an ERE
-F	Matches multiple fixed strings

Ignoring case (-i):
When you look for a name but are not sure of the case, use the -i (ignore) option.

```
$ grep -i 'agarwal' emp.lst

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

This locates the name Agarwal using the pattern agarwal.

Deleting Lines (-v):
The -v option selects all the lines except those containing the pattern.
It can play an inverse role by selecting lines that does not containing the pattern.

```
$ grep -v 'director' empl.lst

    2233|a. k. shukla |g. m. |sales |12/12/52|6000
    5678|sumit chakrobarty |d. g. m. |marketing |19/04/43|6000
    5423|n. k. gupta |chairman |admin |30/08/56|5400
    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
    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
```

Displaying Line Numbers (-n):
The -n(number) option displays the line numbers containing the pattern, along with the lines.

```
$ grep -n 'marketing' emp.lst

    3: 5678|sumit chakrobarty |d. g. m. |marketing |19/04/43|6000
    11: 6521|lalit chowdury |director |marketing |26/09/45|8200
    14: 2345|j. b. saxena |g. m. |marketing |12/03/45|8000
```

15: 0110|v. k. agrawal |g. m. |marketing |31/12/40|9000

here, first column displays the line number in emp.lst where pattern is found.

Counting lines containing Pattern (-c):

How many directors are there in the file emp.lst?

The -c(count) option counts the number of lines containing the pattern.

```
$ grep -c 'director' emp.lst
4
```

Matching Multiple Patterns (-e):

With the -e option, you can match the three agarwals by using the grep like this:

```
$ grep -e “Agarwal” -e “aggarwal” -e “agrawal” emp.lst

2476|anil aggarwal |manager |sales |01/05/59|5000
3564|sudhir Agarwal |executive |personnel |06/07/47|7500
0110|v. k. agrawal |g. m. |marketing |31/12/40|9000
```

Taking patterns from a file (-f):

You can place all the patterns in a separate file, one pattern per line. Grep uses -f option to take patterns from a file:

```
$ cat
patterns.lst
director
manager
chairman

$ grep -f patterns.lst emp.lst
9876|jai sharma |director |production |12/03/50|7000
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
1265|s. n. dasgupta |manager |sales |12/09/63|5600
2476|anil aggarwal |manager |sales |01/05/59|5000
6521|lalit chowdury |director |marketing |26/09/45|8200
```

Basic Regular Expression:

Like the shell's wild-cards which matches similar filenames with a single expression, grep uses an expression of a different type to match a group of similar patterns. Unlike shell's wild-cards, grep uses following set of metacharacters to design an expression that matches different patterns.

If an expression uses any of these meta-characters, it is termed as Regular Expression (RE).

The below table shows the BASIC REGULAR EXPRESSION(BRE) character set-

Symbols/Expression	Matches
*	Zero or more occurrences of the previous character.
g*	Nothing or g, gg, ggg, gggg, etc.
.(Dot)	A single character.
.*	Nothing or any number of characters.
[pqr]	A single character p, q or r
[c1-c2]	A single character withing ASCII range shown by c1 and c2
[0-9]	A digit between 0 and 9
[^pqr]	A single character which is not a p, q or r
[^a-zA-z]	A non-alphabetic character
^pat	Pattern pat at beginning of line
pat$	Pattern pat at end of line
^bash$	A bash as the only word in line
^$	Lines containing nothing

The character class:
A RE lets you specify a group of characters enclosed within a pair of rectangular brackets, [ ], in which case the match is performed for a single character in the group.

```
$ grep '[aA]g[ar][ar]wal' emp.lst

3564|sudhir Agarwal |executive |personnel |06/07/47|7500
0110|v. k. agrawal |g. m. |marketing |31/12/40|9000
```

The *
The * (asterisk) refers to the immediately preceding character. Here, it indicates that the previous character can occur many times, or not at all.

```
$ grep '[aA]gg*[ar][ar]wal' emp.lst

2476|anil aggarwal |manager |sales |01/05/59|5000
```

3564|sudhir Agarwal |executive |personnel |06/07/47|7500
0110|v. k. agrawal |g. m. |marketing |31/12/40|9000

The Dot:
A . matches a single character. The pattern 2... matches a fourcharacter patten beginning with a 2. The pattern .* matches any number of characters, or none.

$ grep 'j.*saxena' emp.lst

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

Specifying pattern locations (^ and $):
^ (carat) – For matching at the beginning of a line.
$ (dollar) – For matching at the end of a line

$ grep '^2' emp.lst

2233|a. k. shukla |g. m. |sales |12/12/52|6000
2365|barun sengupta |director |personnel |11/05/47|7800
2476|anil aggarwal |manager |sales |01/05/59|5000
2345|j. b. saxena |g. m. |marketing |12/03/45|8000
$ grep '7...$' emp.lst
9876|jai sharma |director |production |12/03/50|7000
2365|barun sengupta |director |personnel |11/05/47|7800
3564|sudhir Agarwal |executive |personnel |06/07/47|7500

To display all lines that don't begins with a 2
$ grep '^[^2]' emp.lst

9876|jai sharma |director |production |12/03/50|7000
5678|sumit chakrobarty |d. g. m. |marketing |19/04/43|6000
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
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
0110|v. k. agrawal |g. m. |marketing |31/12/40|9000

To display only directories using ls -l and grep

```
$ ls -l | grep '^d'
drwxr-xr-x 2 chandrakant chandrakant 4096 Jan 16 12:18 Desktop
drwxr-xr-x 2 chandrakant chandrakant 4096 Jan 13 07:54 Documents
drwxr-xr-x 7 chandrakant chandrakant 4096 Jan 16 09:41 Downloads
```

Extended Regular Expression (ERE) & egrep:

ERE make it possible to match dissimilar patterns with a single expression. grep uses ERE characters with -E option. egrep is another alternative to use all the ERE characters without -E option. This ERE uses some additional characters set shown in below table.

Expression	**Significance**
ch+	Matches one or more occurrences of character ch
ch?	Matches zero or one occurrence of character ch
Exp1 | Exp2	Matches exp1 or exp2
GIF | JPEG	Matches GIF or JPEG
(x1|x2)x3	Matches x1x3 or x2x3
(hard|soft)ware	Matches hardware or software

The + and ?

+ - Matches one or more occurrences of the previous character ?

- Matches zero or one occurrence of the previous character.

```
$ grep -E “[aA]gg?arwal” emp.lst

2476|anil aggarwal |manager |sales |01/05/59|5000
3564|sudhir Agarwal |executive |personnel |06/07/47|7500
```

Matching Multiple Patterns(|, (and))

```
$ grep -E 'sengupta|dasgupta' emp.lst

2365|barun sengupta |director |personnel |11/05/47|7800
1265|s. n. dasgupta |manager |sales |12/09/63|5600

$ grep -E '(sen|das)gupta' emp.lst

2365|barun sengupta |director |personnel |11/05/47|7800
1265|s. n. dasgupta |manager |sales |12/09/63|5600
```

SHELL PROGRAMMING

Introduction:

A shell is a program that acts as the interface between user & the linux system, allowing user to enter commands for the operating system to execute. A linux shell is both a command interpreter and a programming language.

As a command interpreter, the shell provides the user interface to the rich set of GNU utilities. The programming language features allow these utilities to be combined. Files containing commands can be created and become commands themselves. These new commands have the same status as system commands in directories such as /bin, allowing users or groups to establish custom environments to automate their commands tasks.

Shells may be used interactively or non-interactively, in interactive mode, they accept input typed from the keyboard. When executing noninteractively, shells execute commands read from a file. A shell allows execution of GNU commands both synchronously & asynchronously. The shell waits for synchronous commands to complete before accepting more

input; asynchronously commands continue to execute in parallel with the shell while it reads & executes additional commands.

The most commonly used shells are SH(Bourne Shell) CSH(C Shell) and KSH(Korn Shell). KSH is based on SH & so is BASH(Bourne again shell). TCSH(Extended C Shell) is based on CSH.

Bash Shell:

Bash is the shell or command language interpreter, for the gnu OS. The name is an acronym for the "Bourne-Again Shell".

Bash is largely compatible with sh and incorporates useful features from the Korn shell ksh and the C shell csh.

Bash is based on the Bourne shell, sh, originally written by Stephen Bourne.

Bash is extremely portable and can be built on most UNIX systems because many of the environment dependent variables are determined at build time. Bash has been ported as a shell for several non-UNIX platforms like QNX, Minix, OS/2, windows 95 & Windows NT.

Bourne Shell:

The Bourne shell is one of number of Unix shell. Like the others, it is both a command language and a programming language. As a command language it provides a user interface to Unix/Linux. It executes commands entered by the user or from a file.

Files containing commands allow users build their own commands thus tailoring the system to their own needs. Such files are called: shell scripts, shell programs, or command files. These commands have access to the command line parameters and have the same status as others Unix Commands.

As a programming language each shell provides I/O, variables, conditionals loops and switches. The syntax is aimed at ease of use at a terminal so that strings for example do not have to be quoted. Each shell(Bourne C Bash etc) has its own syntax.

System administration can build shell programs to add new users, to shutdown the system and so on. Some examples are given in these notes.

C Shell:

The C shell was written by Bill Joy at the University of California at Berkele. Csh is new a command language interpreter good features of other shells and a history mechanism similar to the redo of INTERLISP. While incorporating many features of other shells which make writing shell programs easier, most of the features unique to chs are designed more for the interactive UNIX user.

The C shell has three separate files which are used for customizing its environment. These three files are .chrc, .login, & .logout. because these files begin with a period (.) they do not usually appear when one types the ls command. In order to see all files beginning with periods. The -a option is used with the ls command.

The .cshrc files contains commands, variable definitions and aliases used any time the C shell is run. When one logs in, the C shell starts by reading the .cshrc file, & sets up any variables and aliases.

The C shell reads the .login file after it has read the .cshrc file. This file read once only for login shells. This file should be used to set up terminal settings, for example, backspace suspend and interrupt characters.

The .logout file contains commands that are run when the users logs out of the system.

Unix Shell:

Shell	System Name	Prompt
Bourne Shell	/bin/sh	$
Csh	/bin/sh	%
Tcsh	/bin/tcsh	>
Korn Shell	(shells/ksh93)	$
Z shell	(shells/zsh)	%

Ordinary Variables:

Within a shell, a shell parameter is associated with a value that is accessible to the user. There are several kinds of shell parameters, in the korn shell, all data are stored as strings.

User Defined Variable:

User defined variable is created and maintained by the user. This type of variable defined may use any valid variable name, but it is a good practice to avoid all uppercase names as many are used by the shell.

The .profile:

The .profile file is present in user home ($HOME) directory. It is customize as user individual working environment. Because the .profile file is hidden, use the ls -a command to list it.

The .profile file contains your individual profile that overrides the variables set iin /etc/profile file. The .profile file is often used to set exported environment variables & terminal modes. You can customize your environment by modifying the .profile file. Use the .profile file to control the following defaults.

1. Shells to open.
2. Prompt appearance.
3. Keyboard sound.

This hidden file is read every time you open a shell in unix and you can put in commands to define the experience you want when you interact with a shell.

The following are the profile files of the commonly used shells:

Shell	Profile File
Ksh	.profile
Bourne	.profile
Bash	.bash_profile
Tcsh	.login
Csh	.login

You can put aliases into your .profile file, which customize commands.

read & readonly commands:

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
```

read only statement:

Shell provides a way to mark variables as read-only by using the read-only command. After a variable is marked read-only, its value cannot be changed.

For example, the following script generates an error while trying to change the value of NAME –

```
#!/bin/sh

NAME="Ramu" readonly NAME
NAME="Bantu"
```

/bin/sh: NAME: This variable is read only.

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, $3,…….	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
```

exit & 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:
Ex1:

```
$ grep director emp.lst >/dev/null; echo $?

 0                      #Success
```

Ex2:

```
$ grep director emp.lst >/dev/null; echo $?

 1                      #Failure – in finding pattern
```

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 OCT 13 15:40:13 IST 2020
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!
```

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 Comparision:

Numerical Comparison operators used by test:

Operator	**Meaning**
-eq	Equal to
-ne	Not equal to
-gt	Greater than
-ge	Greater the 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.
```

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:

Form 1	Form 2	Form 3
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

Example:

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

The case Conditional:

The second form of branching is the case statement. A case differs from an if block in that if branches based upon the value of one variable.

An if does not have such restrictions because it uses the test function and a test can be any string of logical expressions as has been shown.

The general syntax of the case statement is as follows:

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

the case block takes as its argument a variable. To denote the variable, it must be surrounded by parenthesis. The case compares the variables value against against each pattern.

The pattern may be any legal regular expression. If variable’s value matches the pattern, then the shell executes the command block immediately following the pattern. The command block terminates with a pair of double semi colons.

As soon as it reaches them, the shell continues past the end of the case block as denoted by the 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
Thu Oct 22 15:40:13 IST 2020
```

While Looping:

A while loop is different from a for loop in that is uses a test condition to determine whether or not to execute its command block. As long as the condition returns true when tested, the loops executes the commands.

If it returns false, the script skips the loop and proceeds beyond its terminate point. Consequently, the command set could conceiveably never run. This is further illustrated by the loop's syntax:

Syntax:

```
while condition
do
```

```
        command1
        .
        .
        .
        commandn
done
```

because the loop starts with the test, if the condition fails, then the program continues at whatever action immediately follows the loops done statement. If the condition passes then the script executes the enclosed command block.

After performing the last command in the block, the loop starts tests the condition again & determine whether to proceed beyond the loop or fail through it once more.

Example:
Here is the example that uses while loop to open through the given list of numbers:

```
i=1
while [ $i -le 5 ]
do
        echo $i
        i=`expr $i + 1`
done
```

this will produce following result:

```
1
2
3
4
5
```

for loop:

The **for** loop operates on lists of items. It repeats a set of commands for every item in a list.

Syntax:

```
for var in word1 word2 ... wordN
do
        Statement(s) to be executed for every word.
Done
```

Here *var* is the name of a variable and word1 to wordN are sequences of characters separated by spaces (words). Each time the for loop executes, the value of the variable var is set to the next word in the list of words, word1 to wordN.

Example:

```
#!/bin/sh

for var in 1 2 3 4 5
do    echo
$var done
```

Output:
1
2
3
4
5

set & shift commands: Handling Positional Parameters:

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 $*
Thu Oct 22 15:40:13 IST 2020

$ echo “The date today is $2 $3 $6”
The date today is Oct 22 2020
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

The Here Document(<<):
The shell uses the << symbol to read data from the same file containing the script. This is referred to as a here document, signifying that the data is here rather than in aspirate file. Any command using standard input can slo take input from a here document.

Example:
mailx kumar << MARK

Your program for printing the invoices has been executed on `date`.Check the print queue The updated file is $flname MARK

The string (MARK) is delimiter. The shell treats every line following the command and delimited by MARK as input to the command. Kumar at the other end will see three lines of message text with the date inserted by command. The word MARK itself doesn't show up.

Using Here Document with Interactive Programs:

A shell script can be made to work non-interactively by supplying inputs through here document.

Example:

$ search.sh << END
> director
>emp.lst
>END

Output:
Enter the pattern to be searched: Enter the file to be used: Searching for director from file emp.lst

9876	Jai Sharma	Director	Productions
2356	Rohit	Director	Sales

Selected records shown above.

The script search.sh will run non-interactively and display the lines containing “director” in the file emp.lst.

trap: interrupting a Program

Normally, the shell scripts terminate whenever the interrupt key is pressed. It is not a good programming practice because a lot of temporary files will be stored on disk. The trap statement lets you do the things you want to do when a script receives a signal. The trap statement is normally placed at the beginning of the shell script and uses two lists:

trap ‘command_list’ signal_list

When a script is sent any of the signals in signal_list, trap executes the commands in command_list. The signal list can contain the integer values or names (without SIG prefix) of one or more signals – the ones used with the kill command.

Example: To remove all temporary files named after the PID number of the shell:

trap ‘rm $$* ; echo “Program Interrupted” ; exit’ HUP INT TERM trap is a signal handler. It first removes all files expanded from $$*, echoes a message and finally terminates the script when signals SIGHUP (1), SIGINT
(2) or SIGTERM(15) are sent to the shell process running the script.

A script can also be made to ignore the signals by using a null command list.

Example:
trap ‘’ 1 2 15

Programs
#!/bin/sh IFS="|"
While echo "enter dept code:\c"; do
Read dcode
Set -- `grep "^$dcode"<<limit

01|ISE|22
02|CSE|45
03|ECE|25
04|TCE|58
limit`

Case $# in echo "dept name :$2 \n emp id:$3\n" *) echo "invalid code"; continue
esac

done

Output:
$valcode.sh

Enter dept code:88
Invalid code

Enter dept code:02
Dept name : CSE
Emp-id :45

Enter dept code: <ctrl c>