**Module-2 Contents**

|  |  |  |
| --- | --- | --- |
| **Sl.no** | **Topics** | **Page No** |
| 1 | ls command with options | 2 |
| 2 | File Permission | 3 |
| 3 | Changing file permission | 4 |
| 4 | The Shells interpretive cycle: | 8 |
| 5 | The wild cards | 8 |
| 6 | Removing the special meanings of wild cards | 10 |
| 7 | Redirection: The Three Standard Files | 11 |
| 8 | Connecting commands: Pipe ( | ) : | 12 |
| 9 | grep – searching for a pattern | 14 |
| 10 | Basic and Extended Regular Expression | 15 |
| 11 | Shell Programming | 18 |
| 12 | Variables | 19 |
| 13 | read and readonly | 22 |
| 14 | The .Profile | 23 |
| 15 | Command line arguments | 23 |
| 16 | exit and exit status of a command | 24 |
| 17 | Logical operators | 24 |
| 18 | The test command and its shortcut | 25 |
| 19 | if | 30 |
| 20 | while | 31 |
| 21 | for | 32 |
| 22 | case | 34 |
| 23 | The set and shift commands | 36 |
| 24 | The here document | 38 |
| 25 | The trap command | 39 |

**Chapter 3**

**FILE ATTRIBUTES AND PERMISSIONS**

# ls command with options

## ls –l: listing file attributes

* 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 are:
  + File type and Permissions
  + Links
  + Ownership
  + Group ownership
  + File size
  + Last Modification date and time

### For example:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| $ ls –l |  | | | | |
| total 72 |
| -rw-r--r-- | 1 | kumar | metal | 19514 | may 10 13:45 chap01 |
| drw-r--r-- | 1 | kumar | metal | 4174 | may 10 15:01 chap02 |
| -rw-rw-rw- | 1 | kumar | metal | 84 | feb 12 12:30 dept.lst |

1. **File type and permissions:** the first column shows the type and permissions associated with each file. The first character in the column is mostly a -, which indicates that the file is an ordinary one. Directories are indicated by d at the same position.
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:** the user who creates the files automatically becomes the owner of the file. The third column shows kumar as the owner of all these files. The owner has full authority to tamper with a file’s content and permissions.
4. **Group ownership:** when opening a user account, the system administrator also assigns the user to some group. The fourth column represents the group owner of the file.
5. **File size:** the fifth column shows the size of the file in bytes, i.e. the amount of data it contains. It is only the character count of the file and not a measure of the disk space it occupies.
6. **Last modification time:** the sixth, seventh and eighth columns indicate the last modification time of the file. A file is said to be modified only if its content have changed in any way. If the file is less than a year old since its last modification date, the year won’t be displayed.
7. **Filename:** the last column displays the filenames arranged in ASCII collating sequence**.**

## ls –d: Listing Directory Attributes

* This command will not list all subdirectories in the current directory.

### For example,

$ls –ld helpdir progs

drwxr-xr-x 2 kumar metal 512 may 9 10:31 helpdir

drwxr-xr-x 2 kumar metal 512 may 9 09:57 progs

* Directories are easily identified in the listing by the first character of the first column, which here shows a d.
* The significance of the attributes of a directory differs a good deal from anordinary file.
* To see the attributes of a directory rather than the files contained in it, use ls –ld with the directory name.
* Note that simply using ls –d will not list all subdirectories in the current directory. Strange though it may seem, ls has no option to list only directories

# File Permission:

* UNIX follows a three-tiered file protection system that determines a file‘s access rights. It is displayed in the following format: Filetype owner (rwx) groupowner (rwx) others (rwx)
* For Example:

-rwxr-xr-- 1 kumar metal 20500 may 10 19:21 chap02 rwx r-x r-- owner/user group owner others

* The first group has all three permissions. The file is readable, writable and executable by the owner of the file.
* The second group has a hyphen in the middle slot, which indicates the absence of write permission by the group owner of the file.
* The third group has the write and execute bits absent. This set of permissions is applicable to others.
* You can set different permissions for the three categories of users – owner, group and others. It‘s important that you understand them because a little learning here can be a dangerous thing. Faulty file permission is a sure recipe for disaster.

# 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:
  + In a **relative** manner by specifying the changes to the current permissions
  + In an **absolute** manner by specifying the final permissions

## Relative Permission:

* 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

Examples:

* Initially, let the permission be:

-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 Permission:

* 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)

* The following table represent the octal value and its significance

|  |  |  |
| --- | --- | --- |
| **Octal** | **Permissions** | **Significance** |
| 0 | --- | no permissions |
| 1 | --x | execute only |
| 2 | -w- | write only |
| 3 | -wx | write and execute |
| 4 | r-- | read only |
| 5 | r-x | read and execute |
| 6 | rw- | read and write |
| 7 | rwx | read, write and execute |

Table 3.1 octal value in absolute permission.

* 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.
* Example:

### $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.

## Recursively Changing File Permission (-R):

* We can use chmod Recursively.

### $ chmod -R a+x shell\_scripts

* This makes all the files and subdirectories found in the shell\_scripts directory, executable by all users.
* When you know the shell meta characters well, you will appreciate that the \* doesn‘t match filenames beginning with a dot.
* The dot is generally a safer but note that both commands change the permissions of directories also.

## 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.

**Chapter-4**

**The shells interpretive cycle**

# The Shells interpretive cycle:

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 the command is entered, the shell scans the command line for **metacharacters** 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. Now the user is free to enter another command.

# Wild cards :

* **Wild cards** are the set of characters that the shell uses to match **filenames.**
* The shell will expand all the **wild cards** before passing the command to the kernel for execution.
* The significance of the various meta-characters in the wild card set are listed in the table below:

|  |  |
| --- | --- |
| **Wild cards** | **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 the characters x  and z |
| **[!ijk]** | A single character that is not i, j or k |
| **[!x-z]** | A single character that is not within the ASCII range of the characters x and z |
| **{pat1, pat2}** | pat1, pat2, etc |

Table 4.1 wild cards

## The \* and ?

* The command **ls chap \*** to list some filenames beginning with **chap.**

### $ ls chap\*

chap chap01 chap02 chap03 chap04 chap15 chap16 chap17 chapx chapy chapz

* The command **ls chap?** Matches five character filenames beginning with chap

### $ ls chap?

**chapx chapy chapz**

*( note: \* and ? doesn’t match all files beginning with a .(dot) or the / of a pathname )*

## The character class [ ]

* The character class allows the user to frame more restrictive patterns.
* 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 [abcd] is a character class, and it matches a single character – an a, b, c, or d.

### $ ls chap0[124]

chap01 chap02 chap04

* **Range specification** is also possible inside the class with a **–** (hyphen); the two characters on either side of it form the range of characters to be matched.

### $ ls chap0[1-4]

chap01 chap02 chap04

### $ls chap[x-z]

chapx chapy chapz

* **Negating the character class (!)** you can use the ! as the first character in the class to negate the class.

**\*.[!co]** Matches all filenames with a single character extension but not the .c or .o files

**[!a-zA-Z]\*** Matches all the filenames that don’t begin with an alphabetic character.

## Matching totally dissimilar patterns { }

* The dissimilar patterns should be written within the flower brackets { and } separated by comma.

**{c, java}** Matches the pattern either c or java

**{include, bin, lib}** Matches pattern either include, bin or lib.

# Removing the special meanings of wild cards:

## Escaping:

* Providing a **\ (backslash)** before the wildcard to remove (escape) its special meaning.
* For instance, in the pattern \\*, the \ tells the shell that the asterisk has to be matched literally instead of being interpreted as a metacharacter.

**rm chap/\*** *Doesn’t remove* chap1, chap2

* The \ suppresses the wild-card nature of the \*, thus preventing the shell from performing the filename expansion on it.

**$ ls chap0\[1-3\]** Must escape the [ and ]

### chap0[1-3]

#### Escaping the space

**$ rm My\ Document.doc** without \ rm would see two files

* ***Escaping the \ itself*** sometimes we may need to interpret the \ itself literally. You need another \ before it, that’s all:

**$ echo** \\

### \

**$ echo the newline character is \\n**

the newline character is \n

## Quoting:

* Enclosing the wild-card, or even the entire pattern, within quotes, anything within these quotes are left alone by the shell and not interpreted.
* The following example shows the protection of the four special characters using single quotes:

### $ echo ‘the characters |, <, > and $ are also special’

the characters |, <, > and $ are also special

* **Single quotes** protect all special characters ( except single quotes ).
* **Double quotes** are more permissive; they don’t protect ( apart from double quotes itself) the **$ and ` (backquote)**

# Redirection: The Three Standard Files

* The Shell associates **three files** with the terminal - two for the **display** and one for the **keyboard**.
* The command performs all the terminal – related activities with these three files that are provided by the shell.
* These special files are actually streams of characters which many commands see as input and output. A stream is simply a sequence of bytes.

## Standard Input:

* The file (or stream) representing input, which is connected to the keyboard.
* When the commands are used without the filename arguments they read the file representing the

**standard input**. This file is indeed special; it can represent three input sources:

* + - 1. The **keyboard**, the default source.
      2. A file using **redirection** with the **<** symbol ( a metacharacter ).
      3. Another program using a **pipeline**.
* **Redirection ( < ) :** shell can reassign the standard input file to a disk file. This means it can redirect the standard input to originate from a file on disk. This reassignment or redirection requires the **<** symbol:

### $ wc < sample.txt 3 14 71

* wc command didn’t open sample.txt.it read the standard input file as a stream but only after the shell reassigned this stream to a disk file (sample.txt).

### $ bc < math.txt

* bc command will read input from the standard input which is assigned to math.txt using < symbol.

## Standard output:

* All commands displaying output on the terminal actually write to the standard output file as streams of characters.
* There are three possible destinations of this stream:
  + - 1. The **terminal**, the default source.
      2. A file using **redirection** with the symbols > and >> ( a metacharacter ).
      3. As input to another program using a **pipeline.**
* **Redirection ( > and >>) :** shell can reassign the standard output to a disk file. This means the default destination can be replaced with any file by using > or >> operator, followed by the filename:

### $ wc sample.txt > newfile

**$ cat newfile**

### 3 14 71 sample.txt

* The first command sends the word count of sample.txt to newfile; nothing appears on the terminal screen. If the output file doesn’t exist, the shell creates it before executing the command.
* If it exists, the shell overwrites it.
* The shell also provides >> symbol to append to a file ( prevents overwriting ).

**$ wc sample.txt >> newfile** *Doesn’t disturb existing contents*

## Standard error:

* Each of the three standard files is represented by a number, called a **file descriptor**.
* A file is opened by referring to its pathname, but subsequent read and write operations identify the file by this file descriptor.
* The first three slots are generally allocated to the three standard streams in this manner:

### 0 – Standard input 1- Standard output

**2- Standard error**

* These descriptors are implicitly prefixed to the redirection symbols. For instance > and 1> mean the same thing to the shell, while < and 0< are identical.
* We need to explicitly use one of these descriptors for handing the standard error stream.
* Redirecting the standard error requires the use of the 2> symbols:

### $ cat foo 2> errorfile

**$ cat errorfile**

### cat: cannot open foo

* You can append standard error to a file :

**$ cat foo 2>> errorfile**

# Connecting commands: Pipe ( | ) :

* **Pipe** allows the standard input stream to connect with the standard output stream such that one command can take input from another.
* For example

**$ who | wc –l** *counts the number of online users*

### 5

* Here the shell connects the **who’s** standard output to **wc’s** standard input using a special operator called pipe ( | ).
* The output of **who** has been passed directly to the input of **wc,** and who is said to be **piped** to wc.
* When multiple commands are connected this way, a **pipeline** is said to be formed.
* It’s the shell that set up the connection and the commands have no knowledge of it.
* One can count the number of files in a directory by combining **ls** and **wc –w** commands using a pipe.

**$ ls | wc –w** *counts the number of files and subdirectories*

### 15

* There is no restriction in the number of commands you can use in a pipeline.

# Example Database:

* Several UNIX commands are provided for text editing and shell programming. (**emp.lst**) - each line of this file has six fields separated by five delimiters.
* The details of an employee are stored in one single line. This text file designed in fixed format and containing a personnel database. There are 15 lines, where each field is separated by the delimiter |.

**$ 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 | directir | 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. sexena | g.m. | marketing | 12/03/45 | 8000

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

# grep – searching for a pattern

* It scans the file / input for a pattern and displays lines containing the pattern, the line numbers or filenames where the pattern occurs.
* It’s a command from a special family in UNIX for handling search requirements.

**grep *options pattern filename(s)***

**grep “sales” emp.lst**

* will display lines containing sales from the file emp.lst. Patterns with and without quotes is possible. It’s generally safe to quote the pattern.
* Quote is mandatory when pattern involves more than one word. It returns the prompt in case the pattern can’t be located.

### grep president emp.lst

* When grep is used with multiple filenames, it displays the filenames along with the output.

### grep “director” emp1.lst emp2.lst

* Where it shows filename followed by the contents

### grep options

* grep is one of the most important UNIX commands, and we must know the options that POSIX requires grep to support. Linux supports all of these options.

### -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 specifies expression with this option**

### -x matches pattern with entire line

**-f file takes pattrens from file, one per line**

### -E treats pattren as an extended RE

**-F matches multiple fixed strings**

### Examples:

**grep -i ‘agarwal’ emp.lst**

### grep -v ‘director’ emp.lst > otherlist

wc -l otherlist will display 11 otherlist

### grep –n ‘marketing’ emp.lst grep –c ‘director’ emp.lst grep –c ‘director’ emp\*.lst

will print filenames prefixed to the line count

### grep –l ‘manager’ \*.lst

will display filenames *only*

### grep –e ‘Agarwal’ –e ‘aggarwal’ –e ‘agrawal’ emp.lst

will print matching multiple patterns

### grep –f pattern.lst emp.lst

all the above three patterns are stored in a separate file *pattern.lst.*

# Basic and Extended Regular Expression:

We 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

## Basic Regular Expression:

* If an expression uses meta characters, it is termed a regular expression. Some of the characters used by regular expression are also meaningful to the shell.
* The basic regular expression character subset uses an elaborate meta character set, overshadowing the shell’s wild-cards, and can perform amazing matches.

### \* Zero or more occurrences

**g\* nothing or g, gg, ggg, etc.**

### . A single character

**.\* nothing or any number of characters**

### [pqr] a single character p, q or r

**[c1-c2] a single character within the ASCII range represented by c1 and c2**

### The character class:

* grep supports basic regular expressions (BRE) by default and extended regular expressions (ERE) with the –E option.
* A regular expression allows a group of characters enclosed within a pair of [ ], in which the match is performed for a single character in the group.

### grep “[aA]g[ar][ar]wal” emp.lst

* A single pattern has matched two similar strings. The pattern [a-zA-Z0-9] matches a single alphanumeric character.
* When we use range, make sure that the character on the left of the hyphen has a lower ASCII value than the one on the right.
* Negating a class (**^**) (caret) can be used to negate the character class. When the character class begins with this character, all characters other than the ones grouped in the class are matched.

### The \*

* The asterisk refers to the immediately preceding character. \* indicates zero or more occurrences of the previous character.

### g\* nothing or g, gg, ggg, etc. grep “[aA]gg\*[ar][ar]wal” emp.lst

* Notice that we don’t require to use –e option three times to get the same output!!!!!

### The dot

* A dot matches a single character. The shell uses ? Character to indicate that.

### .\* signifies any number of characters or none grep “j.\*saxena” emp.lst

**Specifying Pattern Locations (^ and $)**

* Most of the regular expression characters are used for matching patterns, but there are two that can match a pattern at the beginning or end of a line.
* Anchoring a pattern is often necessary when it can occur in more than one place in a line, and we are interested in its occurance only at a particular location.

### ^ for matching at the beginning of a line

**$ for matching at the end of a line grep “^2” emp.lst**

* Selects lines where emp\_id starting with 2

### grep “7…$” emp.lst

* Selects lines where emp\_salary ranges between 7000 to 7999

### grep “^[^2]” emp.lst

* Selects lines where emp\_id doesn’t start with 2

### When meta characters lose their meaning

* It is possible that some of these special characters actually exist as part of the text. Sometimes, we need to escape these characters. For example, when looking for a pattern g\*, we have to use **\**
* To look for [, we use \[
* To look for .\*, we use \.\\*

## Extended Regular Expression (ERE) and grep

* If current version of grep doesn’t support ERE, then use egrep but without the –E option. -E option treats pattern as an ERE.

### + matches one or more occurrences of the previous character

**? matches zero or one occurrence of the previous character b+ matches b, bb, bbb, etc.**

### b? matches either a single instance of b or nothing

* These characters restrict the scope of match as compared to the \*

### grep –E “[aA]gg?arwal” emp.lst # ?include +<stdio.h>

**The ERE set**

### ch+ matches one or more occurrences of character ch

**ch? Matches zero or one occurrence of character ch**

### exp1|exp2 matches exp1 or exp2

**(x1|x2)x3 matches x1x3 or x2x3 Matching multiple patterns (|, ( and ))**

**grep –E ‘sengupta|dasgupta’ emp.lst** matches sengupta or dasgupta

* We can locate both without using –e option twice, or

**grep –E ‘(sen|das)gupta’ emp.lst**

**Chapter 5**

**SHELL PROGRAMMING**

# Shell Programming:

* When groups of command have to be executed regularly, they should be stored in a file, and the file itself executed as a shell script or a shell program by the user.
* A shell program runs in interpretive mode. It is not complied with a separate executable file as with a C program but each statement is loaded into memory when it is to be executed.
* Hence shell scripts run slower than the programs written in high-level language. .sh is used as an extension for shell scripts. However the use of extension is not mandatory.
* Shell scripts are executed in a separate child shell process which may or may not be same as the login shell.

### Example:

script.sh #! /bin/sh

# script.sh: Sample Shell Script

echo “Welcome to Shell Programming” echo “Today’s date : `date`”

echo “This months calendar:” cal `date “+%m 20%y”`

echo “My Shell :$ SHELL”

#This month’s calendar.

* The # character indicates the comments in the shell script and all the characters that follow the # symbol are ignored by the shell.
* However, this does not apply to the first line which beings with #. This because, it is an interpreter line which always begins with #! followed by the pathname of the shell to be used for running the script. In the above example the first line indicates that we are using a Bourne Shell.
* To run the script we need to first make it executable. This is achieved by using the chmod command as shown below:

### $ chmod +x script.sh

* Then invoke the script name as:

### $ script.sh

* Once this is done, we can see the following output :

Welcome to Shell Programming

Today’s date: Mon Oct 8 08:02:45 IST 2007 This month’s calendar:

October 2007

My Shell: /bin/Sh

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 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 |  |  |  |

* As stated above the child shell reads and executes each statement in interpretive mode. We can also explicitly spawn a child of your choice with the script name as argument:

### sh script.sh

*(Note: Here the script neither requires a executable permission nor an interpreter line.)*

# Variables:

* variables are used to provide information to the programs you use. You can have both global environment and local shell variables.
* Global environment variables are set by your login shell and new programs and shells inherit the environment of their parent shell.
* Local shell variables are used only by that shell and are not passed on to other processes. A child process cannot pass a variable back to its parent process.
* To declare a local shell variable we use the form *variable=value* (no spaces around =) and its evaluation requires the $ as a prefix to the variable.
* Example:

### $ count=5

**$ echo $count 5**

* A variable can be removed with **unset** and protected from reassignment by **readonly**. Both are shell internal commands.
* *Note: In C shell, we use* ***set*** *statement to set variables. Here, there either has to be whitespace on both sides of the = or none at all.*

### $ set count=5

**$ set size = 10 Uses of local shell variables**

1. Setting pathnames: If a pathname is used several times in a script, we can assign it to a variable and use it as an argument to any command.
2. Using command substitution: We can assign the result of execution of a command to a variable. The command to be executed must be enclosed in backquotes.
3. Concatenating variables and strings: Two variables can be concatenated to form a new variable.

* Example:

### $ base=foo ; ext=.c

**$ file=$base$ext**

**$ echo $file // prints foo.c**

* + 1. **Environment Variables:**
* The environment variables are managed by the shell. As opposed to regular shell variables, environment variables are inherited by any program you start, including another shell.
* New processes are assigned a copy of these variables, which they can read, modify and pass on in turn to their own child processes.
* The set statement display all variables available in the current shell, but env command displays only environment variables. Note than env is an external command and runs in a child process.
* There is nothing special about the environment variable names. The convention is to use uppercase letters for naming one.
* The following table shows some of the common environment variables.

|  |  |
| --- | --- |
| **Variable name** | **Stored information** |
| HISTSIZE | size of the shell history file in number of lines |
| HOME | path to your home directory |
| HOSTNAME | local host name |
| LOGNAME | login name |
| MAIL | location of your incoming mail folder |
| MANPATH | paths to search for man pages |
| PATH | search paths for commands |

|  |  |
| --- | --- |
| PS1 | primary prompt |
| PS2 | secondary prompt |
| PWD | present working directory |
| SHELL | current shell |
| TERM | terminal type |
| UID | user ID |
| USER | Login name of user |
| MAILCHECK | Mail checking interval for incoming mail |
| CDPATH | List of directories searched by cd when used with a non-  absolute pathname |

Table 5.1 :Shell Environment variables.

### The command search path (PATH):

* The PATH variable instructs the shell about the route it should follow to locate any executable command.

### Your home directory (HOME):

* When you log in, UNIX normally places you in a directory named after your login name.
* This is called the home directory or login directory. The home directory for a user is set by the system administrator while creating users (using useradd command).

### mailbox location and checking (MAIL and MAILCHECK):

* The incoming mails for a user are generally stored at /var/mail or /var/spool/mail and this location is available in the environment variable MAIL.
* MAILCHECK determines how often the shell checks the file for arrival of new mail.

### The prompt strings (PS1, PS2):

* The prompt that you normally see (the $ prompt) is the shell’s primary prompt specified by PS1. PS2 specifies the secondary prompt (>).
* You can change the prompt by assigning a new value to these environment variables.

### Shell used by the commands with shell escapes (SHELL):

* This environment variable specifies the login shell as well as the shell that interprets the command if preceded with a shell escape.

### Variables used in Bash and Korn

* The Bash and korn prompt can do much more than displaying such simple information as your user name, the name of your machine and some indication about the present working directory.
* Some examples are demonstrated next.

### $ PS1=‘[PWD] ‘

**[/home/srm] cd progs**

### [/home/srm/progs] \_

* Bash and Korn also support a *history* facility that treats a previous command as an *event* and associates it with a number. This event number is represented as !.

### $ PS1=‘[!] ‘ $ PS1=‘[! $PWD] ‘

**[42] \_ [42 /home/srm/progs] \_**

**$ PS1=“\h> “ *// Host name of the machine* saturn> \_**

# Read and Readonly : Making scripts interactive

* The read statement is the shell’s internal tool for making scripts interactive (i.e. taking input from the user). It is used with one or more variables.
* Inputs supplied with the standard input are read into these variables. For instance, the use of statement like

#### read name

* causes the script to pause at that point to take input from the keyboard. Whatever is entered by you will be stored in the variable *name*.
* Example: A shell script that uses read to take a search string and filename from the terminal.

#! /bin/sh

# emp1.sh: Interactive version, uses read to accept two inputs #

echo “Enter the pattern to be searched: \c” read pname

echo “Enter the file to be used: \c” read fname

echo “Searching for pattern $pname from the file $fname” grep $pname $fname

echo “Selected records shown above”

# No newline

# use echo –e in bash

* Running of the above script by specifying the inputs when the script pauses twice:

**$ emp1.sh**

**Enter the pattern to be searched : director Enter the file to be used: emp.lst**

### Searching for pattern director from the file emp.lst

|  |  |  |  |
| --- | --- | --- | --- |
| **9876** | **Jai Sharma** | **Director** | **Productions** |
| **356** | **Rohit** | **Director** | **Sales** |

* Selected records shown above.
* The **readonly** command is used to make variables and functions readonly, i.e. you cannot change the value of the variables.

# The .Profile :

* When logging into an interactive login shell, login will do the authentication, set the environment and start your shell. In the case of bash, the next step is reading the general profile from **/etc,** if that file exists. bash then looks for **~/.bash\_profile, ~/.bash\_login** and **~/.profile**, in that order, and reads and executes commands from the first one that exists and is readable. If none exists, **/etc/bashrc** is applied.
* When a login shell exits, bash reads and executes commands from the file, **~/.bash\_logout**, if it exists.
* The profile contains commands that are meant to be executed only once in a session. It can also be used to customize the operating environment to suit user requirements. Every time you change the profile file, you should either log out and log in again or You can execute it by using a special command (called dot).

$ **. .profile**

# Command Line Arguments

* Shell scripts also accept arguments from the command line. Therefore e they can be run non interactively and be used with redirection and pipelines.
* The arguments are assigned to special shell variables. Represented by **$1, $2**, etc; similar to C command arguments argv[0], argv[1], etc.
* The following table lists the different shell parameters.

|  |  |
| --- | --- |
| **Shell parameter** | **Significance** |
| $1, $2… | Positional parameters representing command line arguments |
| $ # | No. of arguments specified in command line |
| $ 0 | Name of the executed command |
| $ \* | Complete set of positional parameters as a single string |
| “$ @” | Each quoted string treated as separate argument |
| $ ? | Exit status of last command |
| $$ | Pid of the current shell |
| $! | PID of the last background job. |

Table 5.2 :shell parameters

# exit and Exit Status of Command:

* To terminate a program exit is used. Nonzero value indicates an error condition.

### Example 1:

**$ cat foo**

### Cat: can’t open foo

* Returns nonzero exit status. The shell variable $? Stores this status.

### Example 2:

**grep director emp.lst > /dev/null:echo $? 0**

* Exit status is used to devise program logic that braches into different paths depending on success or failure of a command.

# The logical Operators && and ||

* The shell provides two operators that aloe conditional execution, the && and ||.

### Usage:

**cmd1 && cmd2**

### cmd1 || cmd2

* && delimits two commands. cmd 2 executed only when cmd1 succeeds.

### Example1:

**$ grep ‘director’ emp.lst && echo “Pattern found” Output:**

|  |  |  |  |
| --- | --- | --- | --- |
| 9876 | Jai Sharma | Director | Productions |
| 2356 | Rohit | Director | Sales |

### Example 2:

**Example 3:**

# test and [ ] :

### Pattern found

**$ grep ‘clerk’ emp.lst || echo “Pattern not found” Output:**

### Pattern not found

**grep “$1” $2 || exit 2**

### echo “Pattern Found Job Over”

* Test statement is used to handle the true or false value returned by expressions, and it is not possible with if statement.
* 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 decisions. Test works in three ways:

### Compare two numbers

* **Compares two strings or a single one for a null value**

### Checks files attributes

* Test doesn’t display any output but simply returns a value that sets the parameters $?
  + 1. **Numeric Comparison**

|  |  |
| --- | --- |
| **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 |

Table 5.3 : Operators

* Operators always begin with a – (Hyphen) followed by a two word character word and enclosed on either side by whitespace.
* Numeric comparison in the shell is confined to integer values only, decimal values are simply truncated.

### Example:

$x=5; y=7; z=7.2

$ test $x –eq $y; echo $?

#### 1 Not equal

$ test $x –lt $y; echo $?

#### True

$ test $z –gt $y; echo $?

#### 7.2 is not greater than 7

$ test $z –eq $y ; echo $y

#### 0 7.2 is equal to 7

* The script emp.sh uses test in an if-elif-else-fi construct (Form 3) to evaluate the shell parameter $#

#!/bin/sh

#emp.sh: using test, $0 and $# in an if-elif-else-fi construct #

If test $# -eq 0; then

Echo “Usage : $0 pattern file” > /dev/tty Elfi test $# -eq 2 ;then

Grep “$1” $2 || echo “$1 not found in $2”>/dev/tty Else

echo “You didn’t enter two arguments” >/dev/tty fi

* It displays the usage when no arguments are input, runs grep if two arguments are entered and displays an error message otherwise.
* Run the script four times and redirect the output every time

### $ emp31.sh >f oo

* **Usage :** emp.sh pattern file

$emp31.sh ftp>foo

You didn’t enter two arguments

$emp31.sh henry /etc/passwd>foo Henry not found in /etc/passwd

$emp31.sh ftp /etc/passwd>foo ftp:\*:325:15:FTP User:/user1/home/ftp:/bin/true

* **Shorthand for test [ and ]** can be used instead of test.
* The following two forms are equivalent

### Test $x –eq $y

and

**[ $x –eq $y ]**

## String Comparison:

* Test command is also used for testing strings.
* Test can be used to compare strings with the following set of comparison operators as listed below.

|  |  |
| --- | --- |
| **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 null |
| s1= =s2 | String s1=s2 |

Table 5.4 : String test used by test

### Example:

#!/bin/sh

#emp1.sh checks user input for null values finally turns emp.sh developed previously #

if [ $# -eq 0 ] ; then

echo “Enter the string to be searched :\c” read pname

if [ -z “$pname” ] ; then

echo “You have not entered th e string”; exit 1 fi

echo “Enter the filename to be used :\c” read flname

if [ ! –n “$flname” ] ; then

echo “ You have not entered the flname” ; exit 2 fi

emp.sh “$pname” “$flname” else

emp.sh $\* fi

**Output1:**

### $emp1.sh

**Enter the string to be searched *:[Enter]***

### You have not entered the string

**Output2:**

### $emp1.sh

**Enter the string to be searched :root**

### Enter the filename to be searched :/etc/passwd Root:x:0:1:Super-user:/:/usr/bin/bash

* When we run the script with arguments emp1.sh bypasses all the above activities and calls emp.sh to perform all validation checks

### $ emp1.sh jai

**You didn’t enter two arguments**

### $ emp1.sh jai emp,lst

**9878|jai sharma|director|sales|12/03/56|70000**

### $emp1.sh “jai sharma” emp.lst You didn’t enter two arguments

* Because $\* treats jai and sharma are separate arguments. And $# makes a wrong argument count. Solution is replace $\* with “$@” (with quote” and then run the script.

## File Tests:

* Test can be used to test various file attributes like its type (file, directory or symbolic links) or its permission (read, write. Execute, SUID, etc).

|  |  |  |  |
| --- | --- | --- | --- |
| **Example:** |  |  | |
|  | $ ls –l emp.lst |
|  | -rw-rw-rw- | 1 kumar group | 870 jun 8 15:52 emp.lst |
|  | $ [-f emp.lst] ; | echo $? |  Ordinary file |
|  | 0 |  |  |
|  | $ [-x emp.lst] ; | echo $? |  Not an executable. |
|  | 1 |  |  |

$ [! -w emp.lst] || echo “False that file not writeable” False that file is not writable.

**Example:** filetest.sh

#! /bin/usr #

if [! –e $1] : then

Echo “File doesnot exist” elif [! –r S1]; then

Echo “File not readable” elif[! –w $1]; then

Echo “File not writable”

else

Echo “File is both readable and writable”\

fi

### Output:

**$ filetest.sh emp3.lst**

### File does not exist

**$ filetest.sh emp.lst**

### File is both readable and writable

* The following table depicts file-related Tests with test:

|  |  |
| --- | --- |
| **Test** | **True if** |
| -f file | File exists and is a regular file |
| -r file | File exists and readable |
| -w file | File exists and is writable |
| -x file | File exists and is executable |
| -d file | File exists and is a directory |
| -s file | File exists and has a size greater than zero |
| -e file | File exists (Korn & Bash Only) |
| -u file | File exists and has SUID bit set |
| -k file | File exists and has sticky bit set |
| -L file | File exists and is a symbolic link (Korn & Bash Only) |
| f1 –nt f2 | File f1 is newer than f2 (Korn & Bash Only) |
| f1 –ot f2 | File f1 is older than f2 (Korn & Bash Only) |
| f1 –ef f2 | File f1 is linked to f2 (Korn & Bash Only) |

# The if Conditional

Table 5.5 : file-related Tests with test

* The if statement makes two way decisions based on the result of a condition. The following forms of if are available in the shell:

**Form 1**

if *command is successful*

then

*execute commands*

fi

**Form 2**

if *command is successful*

then

*execute commands*

else

*execute commands*

fi

**Form 3**

if *command is successful*

then

*execute commands*

elif *command is successful*

then...

else... fi

* If the command succeeds, the statements within if are executed or else statements in else block are executed (if else present). Example:

#! /bin/sh

if grep “^$1” /etc/passwd 2>/dev/null then

echo “Pattern Found”

else

echo “Pattern Not Found”

fi

### Output1:

$ emp3.sh ftp

ftp: \*.325:15:FTP User:/Users1/home/ftp:/bin/true Pattern Found

### Output2:

$ emp3.sh mail Pattern Not Found

# While: Looping

* To carry out a set of instruction repeatedly shell offers three features namely while, until and for.

### Syntax:

while condition is true do

Commands

done

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

### Example:

#! /bin/usr ans=y

while [“$ans”=”y”] do

echo “Enter the code and description : \c” > /dev/tty read code description

echo “$code $description” >>newlist echo “Enter any more [Y/N]”

read any case $any in

Y\* | y\* ) answer =y;; N\* | n\*) answer = n;;

\*) answer=y;;

esac

done

**Input:**

Enter the code and description : 03 analgestics Enter any more [Y/N] :y

Enter the code and description : 04 antibiotics Enter any more [Y/N] : [Enter]

Enter the code and description : 05 OTC drugs Enter any more [Y/N] : n

### Output:

$ cat newlist 03 | analgestics

04 | antibiotics 05 | OTC drugs

# for: Looping with a List

* for is also a repetitive structure.

### Synatx:

for variable in list do

Commands

done

* list here comprises a series of character strings. Each string is assigned to variable specified.

### Example:

for file in ch1 ch2 do

cp $file ${file}.bak

echo $file copied to $file.bak

done

Output:

ch1 copied to ch1.bak ch2 copied to ch2.bak

* + 1. **Sources of list:**

**List from variables**: Series of variables are evaluated by the shell before executing the loop

### Example:

$ for var in $PATH $HOME; do echo “$var” ; done Output:

/bin:/usr/bin;/home/local/bin;

/home/user1

**List from command substitution**: Command substitution is used for creating a list. This is used when list is large.

### Example:

$ for var in `cat clist`

**List from wildcards**: Here the shell interprets the wildcards as filenames.

### Example:

$ for file in \*.htm \*.html ; do

sed ‘s/strong/STRONG/g

s/img src/IMG SRC/g’ $file > $$ mv $$ $file

done

**List from positional parameters**: **Example**: emp.sh

#! /bin/sh

for pattern in “$@”; do

grep “$pattern” emp.lst || echo “Pattern $pattern not found” done

Output:

$emp.sh 9876 “Rohit”

|  |  |  |  |
| --- | --- | --- | --- |
| 9876 | Jai Sharma | Director | Productions |
| 2356 | Rohit | Director | Sales |

# The case Conditional

* The case statement is the second conditional offered by the shell. It doesn’t have a parallel either in C (Switch is similar) or perl.
* The statement matches an expression for more than one alternative, and uses a compact construct to permit multiway branching.
* case also handles string tests, but in a more efficient manner than if.

### Syntax:

case expression in

Pattern1) commands1 ;; Pattern2) commands2 ;; Pattern3) commands3 ;;

…

Esac

* Case first matches expression with pattern1. if the match succeeds, then it executes commands1, which may be one or more commands.
* If the match fails, then pattern2 is matched and so forth.
* Each command list is terminated with a pair of semicolon and the entire construct is closed with esac (reverse of case).

### Example:

#! /bin/sh #

echo “

Menu\n

1. List of files\n2. Processes of user\n3. Today’s Date

1. Users of system\n5.Quit\nEnter your option: \c” read choice

case “$choice” in

* 1. ls –l;;
  2. ps –f ;;
  3. date ;;
  4. who ;;
  5. exit ;;

\*) echo “Invalid option”

esac

**Output**

$ menu.sh Menu

1. List of files
2. Processes of user
3. Today’s Date
4. Users of system
5. Quit

Enter your option: 3

Mon Oct 8 08:02:45 IST 2007

### Matching Multiple Patterns:

* case can also specify the same action for more than one pattern . For instance to test a user response for both y and Y (or n and N).

### Example:

Echo “Do you wish to continue? [y/n]: \c” Read ans

Case “$ans” in

Y | y );;

N | n) exit ;;

esac

### Wild-Cards: case uses them:

* case has a superb string matching feature that uses wild-cards.
* It uses the filename matching metacharacters \*, ? and character class (to match only strings and not files in the current directory).

### Example:

Case “$ans” in

[Yy] [eE]\* );; *Matches YES, yes, Yes, yEs, etc*

[Nn] [oO]) exit ;; *Matches no, NO, No, nO*

\*) echo “Invalid Response”

esac

# set and shift:

## set:

* The set statement assigns positional parameters **$1**, **$2** and so on, to its arguments. This is used for picking up individual fields from the output of a program.

### Example 1:

**$ set 9876 2345 6213**

* This assigns the value 9876 to the positional parameters $1, 2345 to $2 and 6213 to $3. It also sets the other parameters $# and $\*.

### Example 2:

**Example 3:**

### $ set `date`

**$ echo $\***

### Mon Oct 8 08:02:45 IST 2007

**$ echo “The date today is $2 $3, $6” The date today is Oct 8, 2007**

## Shift: Shifting Arguments Left

* Shift transfers the contents of positional parameters to its immediate lower numbered one.
* This is done as many times as the statement is called. When called once, $2 becomes $1, $3 becomes S2 and so on.

### Example 1:

**$ echo “$@”** *$@ and $\* are interchangeable*

### Mon Oct 8 08:02:45 IST 2007

**$ echo $1 $2 $3 Mon Oct 8**

### $ shift

**$echo $1 $2 $3 Mon Oct 8 08:02:45**

**$ shift 2** *Shifts 2 places*

### $echo $1 $2 $3 08:02:45 IST 2007

**Example 2**: emp.sh

#! /bin/sh Case $# in

0|1) echo “Usage: $0 file pattern(S)” ;exit ;;

\*) fname=$1 shift

for pattern in “$@” ; do

grep “$pattern” $fname || echo “Pattern $pattern not found”

done;;

esac

Output:

$emp.sh emp.lst

Insufficient number of arguments

$emp.sh emp.lst Rakesh 1006 9877

|  |  |  |  |
| --- | --- | --- | --- |
| 9876 | Jai Sharma | Director | Productions |
| 2356 | Rohit | Director | Sales |

Pattern 9877 not found.

### Set -- : Helps Command Substitution

* Inorder for the set to interpret - and null output produced by UNIX commands the – option is used . If not used – in the output is treated as an option and set will interpret it wrongly.
* In case of null, all variables are displayed instead of null.

### Example:

**Example2:**

### $ set `ls –l chp1`

Output:

### -rwxr-xr-x: bad options

**$set `grep usr1 /etc/passwd`**

Correction to be made to get correct output are:

### $ set -- `ls –l chp1`

**$ set -- `grep usr1 /etc/passwd`**

# 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.
  + 1. **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 |

* 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**

**QUESTION BANK**

1. Describe the significance of the seven fields of the ‘ls –l’ command in detail.
2. The current permission of a regular file “unix” are **rw--w---x** write the chmod expression to change it to the following:
   1. –wxrwxr-x
   2. ---r-xrw-
   3. rwx---x---
   4. r---wx---

Using both absolute and relative method of assigning permissions.

1. What are wild cards ? Explain the various shell wild cards with suitable examples.
2. Apply the shell’s wild card and write the output. i) [a-z][1-4]\*.txt

ii) \*.[!c][!p][!p]

iii) \*[0-3][A-Z]

iv) chap\*[!0-9]

v) chap[0-1][0-9] vi) [A-Z][a-z][0-9]\*

1. Explain grep command with all its options.
2. Explain briefly BRE and ERE metacharacters with suitable examples.
3. Explain the three standard files.
4. Explain the process of connecting commands using pipes.
5. Explain the shells interpretive cycle.
6. What is a shell programming? Write a shell program to get the following details from the student: NAME, AGE, USN and GENDER. Output all the details to the terminal and also display message whether the student is eligible to vote or not.
7. Write a shell script to create a simple calculator that performs addition, subtraction, multiplication and division based on the user’s input.
8. Discuss the difference between user defined variable and environmental variable.
9. Explain the .Profile of the user.
10. What are the command line arguments?
11. Explain the exit and exit status of the command with suitable examples.
12. Write and explain the syntax of for and while loops in shell programming.
13. Explain the three different forms of if conditional statement.
14. Write a menu driven shell program to perform the following:
    1. List of users
    2. Files in the directory
    3. Today’s date
    4. Total number of files in the directory.
15. Explain the set and shift command for handling positional parameters with example.
16. Explain the working of test command.
17. Explain the here (<<) document.
18. Explain the trap command