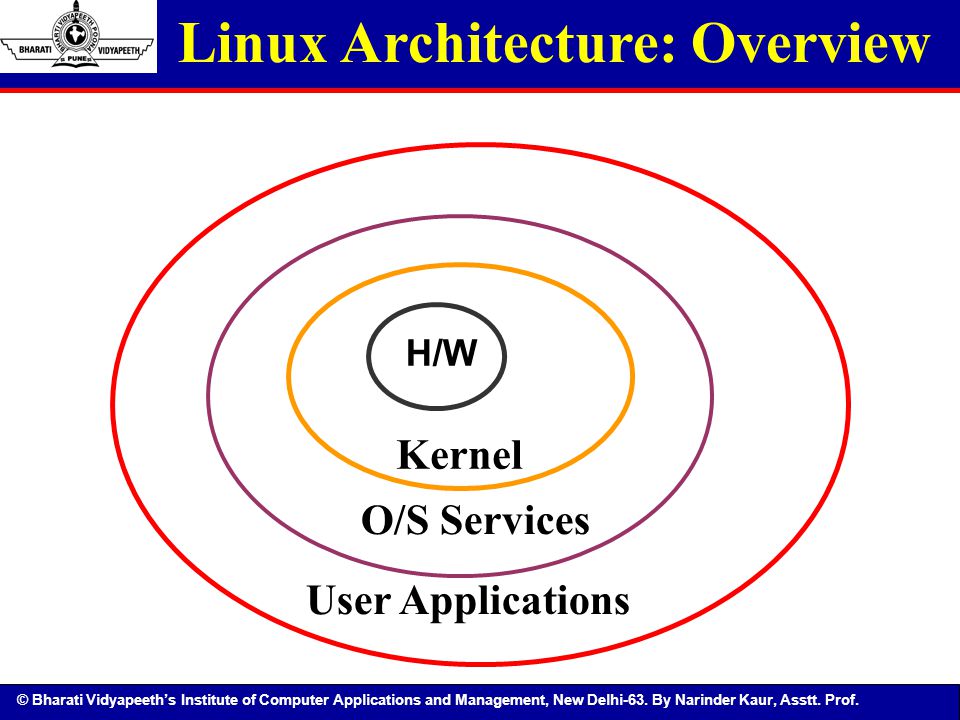
**UNIT-1**

Linux is a free and open source software operating system that can perform all the tasks that any current popular O.S can. The defining component of a Linux distribution is its kernel, which is the heart of an operating system. It controls the hardware, CPU, memory, hard disk, network card etc.

The shell acts as an interface which enables the communication between user and kernel. It interprets the inputs of the user as commands and passes them to the kernel.

#### High level architecture:



Kernel : It is the heart of the OS. It is the main program in the UNIX System. It controls Hardware, Software, CPU, memory, hard disk, network card etc.

Shell : It is the interface between user and the kernel. It interprets your commands and pass them to kernel.

Application : Provides useful functions for the OS.

#### **Linux-kernel architecture:**

63

**THE BASIC COMMANDS IN LINUX INCLUDES THE FOLLOWING:**

1.mkdir-make directory

This is used to create a directory.

Syntax: $mkdir <directory\_name>

Example: mkdir directory1

2.rmdir-remove directory

This is used to remove directory.

Syntax: $rmdir <directory\_name>

Example:rmdir directory1

3.cd-change directory

This is used to change the directory i.e getting into a particular directory.

Syntax: $cd <directory\_name>

Example: $cd directory1

4.pwd-present working directory

The pwd command writes the full pathname of the current working directory to the standard output.

Syntax: $pwd

5.date

The date command displays the current date and time. It can also be used to display or calculate the date in a format you specify.

Syntax: $date

File handling utilities

These are the Linux commands which helps to create, delete, rename, move, copy, edit and perform other related activities on Linux files.

File creating and deleting

1. cat -

In Unix file system, to create files we use this command.

Syntax: $cat <redirection symbol><filename>

Options:

-v: non-printing characters.

-n: numbering the lines.

Example: $cat > file1

Welcome

(Ctrl-d)-to save the content entered after the command.

To display the content -

$cat file1

1. rm –

In Unix file system, to remove files we use this command.

Syntax: $rm [-option] <filename>

Options:

-f : (force) forcibly removes the file though it has all r, w, x permissions.

-r : (recursive removal) removes all files and empty directories. Even if the directory has files, it removes the files first and then removes the directory.

-i : (interactive removal) remove command with interactive flag asks the user before removing the file or directory.

Example: $rm -f file1

File naming and renaming

1. move –

Syntax: $mv [-option] [file/dir] [newfile/newdir]

Options:

-i : (interactive flag) The interactive flag with mv command is useful to warn the user that the destination file already exists.

-f : (force) this is used to forcibly overwrite the already existed file.

Example: $mv -i file1 file2

Editing files

In Unix file system ,we edit a file, append or update the existing data by using vi editor.

Syntax: $vi <filename>

Example: $vi file1

Esc + i ---- insert mode(to enter the contents)

Esc : wq ---- to save the contents

File access permissions

Unix operating system supports 3 kinds of permissions.

1.read 2.write 3.execute

read : read permissions allows to read the content of a file

write : write permissions allows to edit / move/ delete a file.

execute : execute permissions allows to execute the program files and shell script files for different sets of data.

And there exists 3 kinds of users

1.user/owner(u) 2. group(g) 3. others(o)

user/owner : the creator of file or directory.

group : each user is a member of a group ; a member in a group can have access to file created by any other member of that group.

others : this type of users are everybody other than the group.

The COMMAND ‘ls’ with -l option is used to view the permissions of the files and directories.

Syntax: $ls -l

The differentiation between files and directories will be shown as follows.

-rwx rwx rwx ( ‘ - ‘ indicates file)

drwx rwx rwx ( ‘ d ‘ indicates directory)

In Unix file system, several file permissions and changing methods are available. The permissions of a file/directory can be changed using the COMMAND chmod.

Syntax: $chmod [-option] [mode] <file/dir>

Example: $chmod u+x file1

Process utilities

A process is any program that is running on the system i.e. any program under execution.

All processes are assigned unique process identification numbers. The commands under this:

1)ps 2)who 3)w 4)kill 5)pkill

1. ps – process status

This command displays the attributes of a process. A process consists of many attributes like a file. ps reads a process table.

Syntax: $ps [-option]

Options:

-e : shows list of running processes including user and system processes.

-f : shows entire details of user and system processes.

-a :lists the processes of all users.

-u : shows processes of specified user only.

-l : displays a long list of memory related information.

Example : $ps -e

2.who –

This command is used to know who is currently logged on to the system.

Syntax : $who [-option]

Options :

-H : displays the header of the column.

-U : provides a detailed list.

Example: $who -H

3.w –

w command is same as the who command except that this gives detailed output of user activities and many details of the system.

Syntax: $w [-options]

Options:

-H, -W

4.kill –

It kills the process id.

Syntax: $kill signal PID PID

5.pkill –

It kills the process.

Syntax: $kill signal process process

Disk utilities

1. du-

It displays the disk space used by the specified file/directory. And it displays disk usage of the current directory.

Syntax: $du [-option]

Options:

-a: (all) count of all files, not just for directory.

-b: (byte) it prints the size in bytes.

-t: (total) it prints the grand total.

-h: (human readable) it prints the size in human readable format.

2. df-

This command is used to find out the empty disk space.

Syntax: $df

Text processing utilities

The commands included under text processing utilities are:

1)head 2)tail 3)cut 4)paste 5)Tr

1. head –It is a Unix utility program and it displays first few lines of the give text file.

Syntax: $head [-option] <filename>

Options:

-n: It specifies the number of lines that are to be displayed from top of the specified file name.

Example:

$head -2 file1 $cat > file1

Abc Abc

Def Def

Ghi

Jkl

The first 2 lines of the file will be displayed.

Format:

1. tail - It is a Unix utility program and it displays last few lines of the give text file in reverse order.

Syntax: tail [-option] <filename>

Options:

+N: It displays the file contents by skipping N-1 lines from the beginning.

Example:

$tail 2 file1 $cat > file1

Ghi Abc

Jkl Def

Ghi

Jkl

1. cut – When particular columns are to be extracted from a given file ,cut command is used. It splits the file into columns and displays selected column on the output stream.

Syntax: $cut [-option] <filename>

Options:

-c: this is used to specify the characters position inorder to format the file in a fixed column.

-f: used to specify a field. The first field in the beginning of lines is field1.

Example: $cut -c 9-15 file1

This extracts the data that is stored at column 9-15.

1. paste: this is used to combine different text lines of two different text files by taking its input from specified file name.

Syntax: $paste [-option] <filename>

Options:

-d: this delimitor option is used to separate the lines.

-s: this suppress option is used for combining lines of 2 different files.

Example: $paste -d “/” file1 file2

1. tr: (translating characters) used for translating letters from character set to other.

Syntax: $tr [-option] string1 string2

This command replaces the characters specified in string1 with characters specified in string2.These strings are enclosed in double quotes. The translation is done by replacing the first character in string1 with first character in string2.

Options:

-d: this option is used to delete from text line that are matching during translation

-s: if same characters are occurring ,this is used to compress the char in the output.

-c: this option is used to display lowercase letters to uppercase.

Examples:

$tr -d “legtwy”

Yesterday I dared to struggle today I dare to win

Srda I dard o srugg oda I dar o in

$tr -s “ria” “egr”

Yesterday I dared to struggle today I dare to win

Yesteedry I dreed to steuggle todry I dreed to wgn.

$tr -c “abc” “ABC”

Stu aokabc ode

Stu Aok ABC ode

Filters

In Unix, a filter is any command that has the following features.

1)receives its input from standard input

2)manipulates input

3)sends the result to the standard output

Examples:

more, cat, comm, cut, cmp, diff , head , tail , paste , sort , tr , uniq , wc.

1. more –

It is a command to view the contents of a text file one screen at a time.

Syntax: $more [-option] <filename>

1. sort –

It sorts the contents of a file i.e. arranges the content in a particular sequence using ASCII character code values.

It is also responsible for merging and comparison.

Syntax: $sort [-option] <filename>

Options:

-c: It checks the given file that is already sorted.

-n: It sorts on numeric values.

Example:

$cat file1 $sort file1

12 12

3 13

5 22

13 33

22 3

33 5

1. GREP –

Global Regular Expression Pattern.

Its used to manage different search specifications. It does this by searching the input file by following a particular pattern inorder to print the line that matches the selection criteria.

Syntax: $grep [-option] pattern <filename>

Options:

-c: returns only the number of matches.

-i: ignores the case while searching.

-n: returns the line number as well as the text itself.

-v: returns the lines that do not match with the text.

Example: $grep “pen” file1 $cat file1

A penA pen

A car

A horse

1. EGREP –

Extended grep.

It is the most powerful utility of the grep family. Many regular expressions match the characters that are supported by this utility but not all.

Syntax: $egrep [-option] [regular expression] [filename]

1. FGREP –

It performs the searching based on string patterns but not on regular expression.

Syntax: $fgrep [-option] [string pattern] [filename]

1. wc –

word count.

The purpose of wc command is to count the number of lines,words,chars in a particular file/files.

Syntax: $wc [-option] <filename>

Options:

-l: gives the number of lines present in a text file

-w: gives the number of words present in a text file

-c: gives the number of characters present in a text file

-lw: gives the number of lines and words present in a text file

-wc: gives the number of words and characters present in a text file

-lc: gives the number of lines and characters present in a text file

Examples:

$wc -l file1 $cat > file1

2 ab

As

1. uniq –

This command displays unique lines in a file. It displays only single copy of each line on to the standard output.

Syntax: $uniq [-option] <filename>

Options:

-u: this option compresses the output of duplicated lines and display the only unique lines in a file.

-d: it is a complement of -u option. It displays single copy of duplicated lines.

-c: this option counts the occurrence of each line.

Comparing commands

There are 3 commands that are used to compare the contents of two files.

1)cmp 2)comm 3)diff

1. cmp –

This command is used to perform comparison of 2 files byte by byte. Its used to determine whether the file is identical/not.

Syntax: $cmp [-option] filename1 filename2

Options:

-l: this option displays the list of all differences present in a file byte by byte.

-s: this is known as suppress list option. This doesn’t display any output.

2. comm –

This command prints the lines that are common in 2 files as input.

Syntax: $comm [-option] filename1 filename2

3.diff –

This command prints the difference between 2 files by performing comparison line by line and by comparing first file with second file. When a difference is marked, the first file is altered inorder to match it with the second file.

Syntax: $diff [-option] filename1 filename2

Networking commands

The commands which we will use for communication between more than one system for sharing resources or any other issue are known as networking commands.

These are of 5 types.

1)telnet 2)finger 3)ftp 4)arp 5)rlogin

1. telnet –

It is an internet protocol ,which enables the user to connect a remote system and transfers the data. Telnet allows the user to login to a remote machine by supplying a username and password.

Syntax : $telnet hostname

2.finger –

It is used to find out the details of the user ,on both local and remote systems.

Syntax: $finger’

3.ftp –

It supports TCP/IP standards. This is used to transfer a file to 2 systems such as a client to a server ,a server to a client, between the servers.

Syntax: $ftp domain\_name

4.arp –

It is used by internet protocol specifically by IPv4.The term address resolution protocol refers to the process of finding an address of a computer in a network.

Syntax: $arp

5.rlogin –

It tells the user to log on to his identical remote account, without using either the username or password.

Syntax: $rlogin hostname

Backup utilities

Two backup utilities are used in Linux.

1)cpio 2)tar

They are used to create backup of data ,in case of system crash or loss of data.

A backup program is said to be effective if it is able to restore the lost data files easily.Backup data can be stored on secondary storage devices like magnetic tape ,hard disk, floppy disk etc.

1. cpio –

This is used to copy a file to any backup device or restore the data from the backup device. It creates a backup by storing groups of files in an archive along with the header and file contents.

It takes input from the standard input and copies in the buffer space along with its content and header and then displays on standard output.

Syntax : $cpio [-option] filename

1. tar –

This is used to save and restore the files and directories to/from different types of media like tape or floppy disk. We can mount a directory or format a disk to which we want to backup data without creating a file system on it.

Syntax: $tar -cvf tarfile/tapedevice directory

sed command

This is the most powerful filter. It stands for stream editor.editor. A stream editor is used to perform basic text transformations on an input stream (a file or input from a pipeline). While in some ways similar to an editor which permits scripted edits (such as ed), sed works by making only one pass over the input(s), and is consequently more efficient. But it is sed’s ability to filter text in a pipeline which particularly distinguishes it from other types of editors.

It reads standard input process using a file called sed script and writes the result to the standard output.

Syntax: $sed [-option] ‘address.action’ file(s)

Options:

-e: It is a default option.It indicates that the script is on the command line.

-f: It indicates that the script is in a file which immediately follows this option.

-n: It suppresses the automatic output i.e. it will not display the contents of pattern space.

Example: $sed -f myscript.sed f.txt

sed script

A sed script is a file that contains list of instructions to be applied to each line in an input file. If there is only one instruction ,it can be included as command line.

If there are more instructions, that need to be executed frequently, they should be saved in a separate file .Each instruction in a sed script consists of an address and a command separated by a compliment operator.

sed addresses

A sed address is an instruction which determines ,the lines in the input file that are to be processed or skipped by the command in the instructions.

Under this there are four:

1. single address
2. set of line addresses
3. range address
4. nested address

single address:

It specifies the exact line number in the input file. We can use dollar to specify the last line.

Example:

12 command

This displays the 12th line.

set of line address:

It is specified by a regular expression which is written between a pair of forwarded slashes.

/^A/ -It matches all the lines which start with A.

/^$/ -It matches blank lines.

Example:

$sed ‘3Q’ f1

Displays 3 lines and quits.

$sed ‘1,2 P’ f1

Prints 2 lines

range address:

It defines a set of consecutive lines to be processed by the command in the instruction. The range is specified with start address and end address.

Example:

$4,9 command

This matches all the lines between 4 and 9.

$3,/^A/

This matches the lines starting from 2 and ending with a line that starts with A.

nested address:

we can also define a nested address i.e. an address contained in another address.

Example:

11,20{

/^$/d

}

It deletes all the blank lines between 11 and 20.

awk commands

awk is a more utility. The operations of awk is similar to sed utility. It reads a standard input line by line and takes an action on a part of entire line. The actions are specified in an awk script that consists of list of instructions.

Each instruction contains a pattern and its associated action.

Syntax: awk [-option] awk\_script file(s)

awk execution

awk script is a file containing a list of instructions to be applied to each line in a input file.

Syntax: $awk ‘pattern{action}’ filename

Example: $awk ‘/program/{print}’ f1.txt

Fields and records

A file is viewed as collection of fields and records by the awk utility.A field is a data unit that gives some data. Each line in a file is a record, which is a collection of several fields. However the record contains related data.

A file organized into records is called a data file.

Scripts

These are divided into 3 parts:

1. initialization
2. body
3. end of job

|  |
| --- |
| BEGIN {action} |
| pattern1 {action1}  pattern2 {action2}  .  .  patternN {actionN} |
| END {action} |

1. Initialization –

This part defines instruction for initializing variables , creating report headings , sed system variables etc. It is identified with a token BEGIN and all the instructions are enclosed with curly braces.

This part is processed only once before the awk reads the first line from the input file.

1. Body –

It consists of one or more instructions for processing the data in a file. Each instruction consists of a pattern associated with an action that will be taken when the pattern is matched.

1. End of job –

The end part is executed only once after the last line from the input file was read.

Pattern

Pattern identifies on which records in a file ,the action should be taken.

* The pattern matches the record(line) ,its associated action is taken.
* If the pattern doesn’t match the record ,its associated action is skipped.
* awk supports different types of patterns.

1. Nothing ( address action not specified )
2. Begin ( start )
3. End
4. Expression :regular

arithmetic

relational

logical

1. Range

Actions

The actions in awk are one or more statements enclosed within { }. With each pattern there exists an associated action.

* When a pattern is true ,its associated action is taken. Otherwise it is skipped.
* The statement in an action is terminated with a newline (or) semicolon (or) } .
* The valid form of pattern and action pairs are

1. pattern {statement}
2. pattern {statement1, statement2 ,…..statementn}
3. pattern {

statement1

statement2

-

-

Statementn

}

String functions

1. length() –

awk utility provides a number of built-in functions.

The length() counts the number of characters in a string and returns the count value.

Syntax: length( string )

1. index() –

This determines the first position of substring within a string. If the string isn’t found ,it returns 0.Index value starts from 1.

Syntax: index( string, substring)

1. substring() –

The substring() returns substring from the given string.

Syntax: substr( string , starting\_index ) or

substr( string , starting\_index , length )

1. split() –

This divides the string into 2 substrings using a field separator.

Syntax: split ( str ,array )

split ( str ,array ,field\_separator )

1. sub() –

The sub() substitutes one string for another string that matches your regular expression. It returns 1 if the string was substituted and returns 0 if it failed.

Syntax: sub (regular\_exp , with\_string , input\_string)

1. gsub() –

This is same as sub() except that gsub() substitutes all the occurrences of the matching string with another string.

Syntax: gsub( regular\_exp ,with\_string , input\_string )

UNIT 2: Shell Programming with BASH

Shell:-

A shell is a program which takes the command types by the user and converts it into machine understandable format.

Types of shells:-

The following are the different types of shells

1. The Bourne Shell
2. The C shell
3. The Korn shell
4. Bourne Again Shell(BASH)
5. The T C Shell (Tcsh)

1) The Bourne Shell:-

The Bourne shell, called "sh," is one of the original shells, developed for Unix computers by Stephen Bourne at AT&T's Bell Labs in 1977. Its long history of use means many software developers are familiar with it. It offers features such as input and output redirection, shell scripting with string and integer variables, and condition testing and looping.

2) The C shell:-

Developers have written large parts of the Linux operating system in the C and C++ languages. Using C syntax as a model, Bill Joy at Berkeley University developed the "C-shell," csh, in 1978. Ken Greer, working at Carnegie-Mellon University, took csh concepts a step forward with a new shell, tcsh, which Linux systems now offer.

3) The Korn shell:-

David Korn developed the Korn shell, or ksh, about the time tcsh was introduced. Ksh is compatible with sh and bash. Ksh improves on the Bourne shell by adding floating-point arithmetic, job control, command aliasing and command completion. AT&T held proprietary rights to ksh until 2000, when it became open source.

4) Bourne Again Shell (BASH):-

The popularity of sh motivated programmers to develop a shell that was compatible with it, but with several enhancements. Linux systems still offer the sh shell, but "bash" -- the "Bourne-again Shell," based on sh -- has become the new default standard. One attractive feature of bash is its ability to run sh shell scripts unchanged. Shell scripts are complex sets of commands that automate programming and maintenance chores; being able to reuse these scripts saves programmers time. Conveniences not present with the original Bourne shell include command completion and a command history.

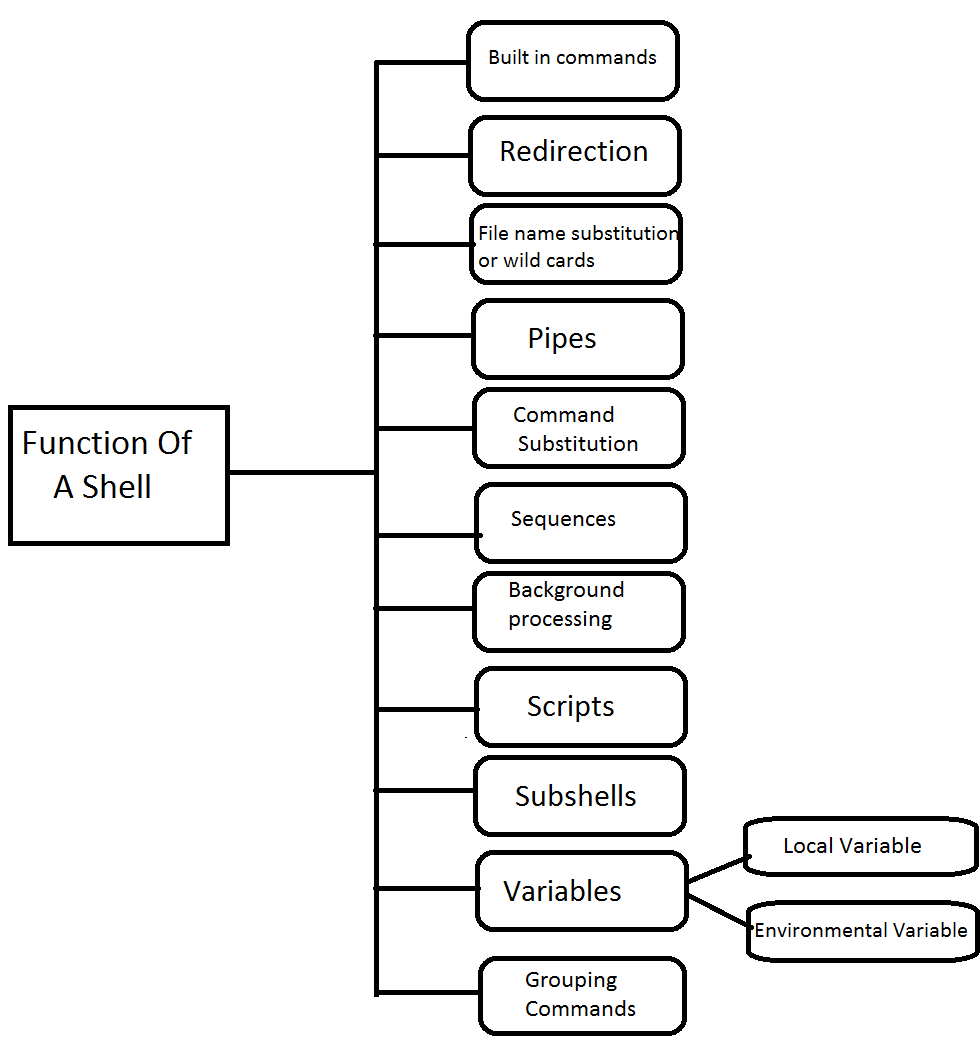
5) The T C Shell

Tcsh fixed problems in csh and added command completion, in which the shell makes educated "guesses" as you type, based on your system's directory structure and files. Tcsh does not run bash scripts, as the two have substantial differences.

Shell script:-

The usage of more number of commands in one file is called Shell script.

\*Shell responsibilities



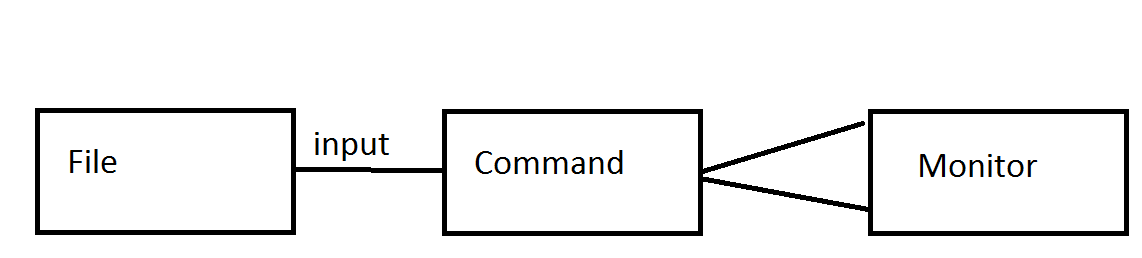
Built in commands:-  
A shell contains several present commands upon giving a command $ls.  
ls - A shell searches the /bin directory for the executable program ls. If it is found, execute the command and it displays all the files present in the directory.  
  
Wild cards or File name substitution:-  
A wild card is also known as the file name substitution. A shell offers a wild card facility that helps in selecting files from a file system, that satisfies the specific pattern.  
A pattern is an argument to a command rather than supplying a long list of file names which a pattern represents.  
Eg:- $ls -l f-name\*  
  
The pattern fname\* matches the filenames starting with a string fname specified in a command line.  
When the shell encounters a meta character '\*', then it replaces the list of files from the current directory that matches the pattern.  
  
Command substitution:-  
The shell executes the command surrounded by a backward quotes.  
Syntax:- `command`  
Eg:- $echo enter the date is `date`  
  
Sequences:-  
A shell executes a series of commands in a sequence from left to right and the sequence of commands are separated by a semicolon.  
Eg:- $cat > file1 ; ls ; pwd  
  
Background processing:-  
When a user gives a command or a series of commands followed by a "&" symbol and a "&metacharacter", a subshell is created to execute the commands in a background which will run simultaneously as parent shell and act as the background process.  
Eg:- $fact.c & date&  
  
Subshells:-  
A shell consists of a parent shell and a child shell. A current shell creates a new shell to perform a specific task.  
  
Shell has two data areas  
Environmental area  
Local variable area  
  
Variables:-  
There are two types of shell variables:-  
Local variables  
Environmental variables  
  
The data in these variables is stored as string.  
A child shell inherits a copy of the parent shell which is called as the environmental variables but not local variable.  
  
The useful information transmitted by using environmental variable.  
  
Grouping commands:-  
Shell allows group of commands separated by using semicolon by placing between the parenthesis.  
  
The group of commands is executed by using sub shell or child shell.  
Eg:- $(who;pwd;date;ls)

\*Pipes:-  
Piping is a process of combining 2 or more commands using a pipe operator("|").  
Eg:- $who | ls  
The output from command on the left will be the input to the other command.

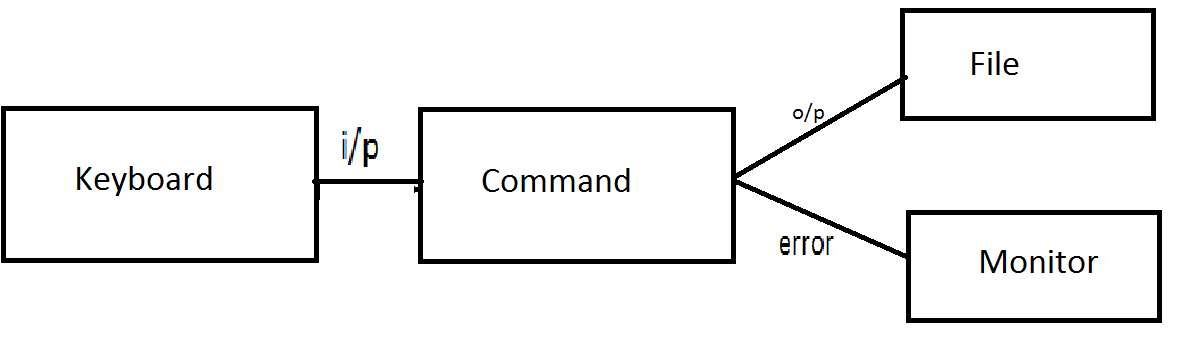
Redirection:-

Redirection is a process in which we use a file in place of one of the standard streams.

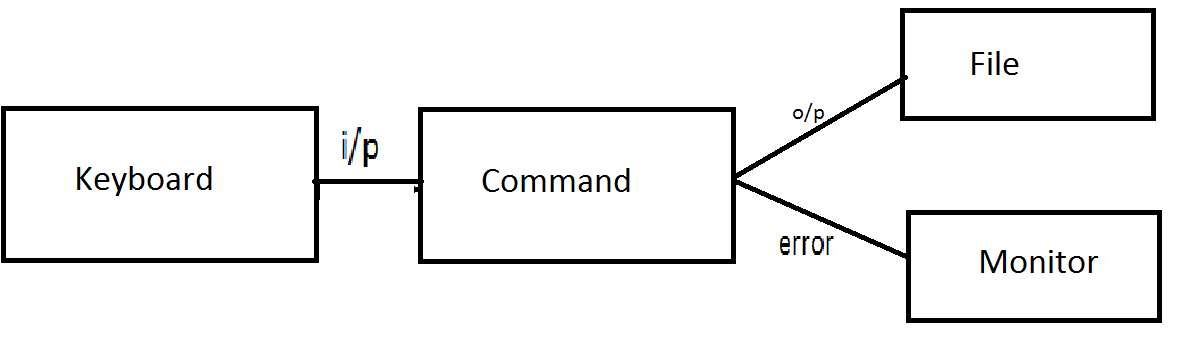
a) Standard input Streams



b) Standard output streams



c) Standard error streams



Here Document

A *here document* is a special-purpose code block. It uses a form of [I/O redirection](http://tldp.org/LDP/abs/html/io-redirection.html#IOREDIRREF) to feed a command list to an interactive program or a command, such as [ftp](http://tldp.org/LDP/abs/html/communications.html#FTPREF), [cat](http://tldp.org/LDP/abs/html/basic.html#CATREF), or the *ex* text editor

**Syntax : here<<string**

metacharacters

The command options, option arguments and command arguments are separated by the space character. However, we can also use special characters called metacharacters in a Linux command that the shell interprets rather than passing to the command.

Quotes

There are mainly four types of quotes characters used in Linux , These are

**1.** **"** (double quote):

      The double quote ( "quote" ) protects everything enclosed between two double quote marks except $, ', " and \.Use the double quotes when you want only **variables and command substitution**.

**2**. **'** (single quote):

     The single quote ( 'quote' ) protects everything enclosed between two single quote marks. It is used to      **turn off the special meaning** of all characters.

**3.** **`** (back quote):

Backtick is not a quotation sign, it has a very special meaning. Everything you type between backticks is evaluated (executed) by the shell before the main command (like chown in your examples), and the *output* of that execution is used by that command, just as if you'd type that output at that place in the command line.

**4. \** (back slash):

The backslash ( \ ) alters the special meaning of the ' and " i.e. it will escape or cancel the special meaning of the next character.

**Control Structures**

The *control flow* commands alter the order of execution of commands within a shell script. They include the **if...then, for...in, while, until,** and **case** statements. In addition, the **break** and **continue** statements work in conjunction with the control flow structures to alter the order of execution of commands within a script.

1. If.. then.. fi

      The if statement tests the result of a command and then conditionally executes a group of statements. It is a nested conditional flow control.

Programs

1. Write a shell script using two arguments.

  Ans.

            $ vi file1

            Echo “My First number is $1”

            Echo “My Second number is $2”

            Echo “total number of arguments are $#”

2. Write a shell script using arithmetic operator

Ans.

   $vi file 2

   #!/bin/bash

   a=5

   b=10

   Echo ‘expr $a+$b’

3. Write a shell script that accepts a file name starting and ending line numbers as arguments and display         .all lines between given line number.

Ans.  cat<line.txt>

        $awk ‘NR<2||NR>4 { print $0 }’ line.txt

4.Write a shell script that displays a list of all files in the current directory to which users can read, write and execute permissions.

Ans.

  Echo ”List of files which have r,w,x permissions in current directory”

  for file in \*

  do

  If [-r $file -a -w $file -a -x $file]

then

echo $file

done

5. Write a shell script to list all the files in a directory.

Ans.

 $vi file

 #!/bin/bash [location of yout OS]

 for x in \*

 do

 ls $x

 done

6. Write a shell script to print 1-10 numbers

Ans.

  $vi file 1

  #!/bin/bash

  for i in {1..10}

  do

  echo $i

  done

7. Write a shell script for string comparison.

Ans.

 word = abc

 If [$word==”abc”]

 then

echo “condition is true”

else

echo “condition is false”

fi

count =10

if[$count -eq 10]

then

echo “condition is true”

else

echo “condition is false”

fi

8.Write a  Shell script  to find factorial of a given integer

Ans.

    # !/bin/bash  
echo "enter a number"  
read num  
fact=1  
while [ $num -ge 1 ]  
do  
fact=`expr  $fact\\* $num`  
num=’expr $num – 1’  
done  
echo "factorial of $n is $fact"

9.Write about Environment variable.

Ans:

Environment variable

An environment variable is a dynamic-named value that can affect the way running processes will behave on a computer.

They are part of the environment in which a process runs.

For example, a running process can query the value of the TEMP environment variable to discover a suitable location to store temporary files,

or the HOME or USERPROFILE variable to find the directory structure owned by the user running the process.

9.Write about Local variable.

Ans:

Local Variable

In computer science, a local variable is a variable that is given local scope.

Local variable references in the function or block in which it is declared override the same variable name in the larger scope.

In programming languages with only two levels of visibility, local variables are contrasted with global variables.

On the other hand, many ALGOL-derived languages allow any number of nested levels of visibility, with private variables, functions, constants and types hidden within them, either by nested blocks or nested functions.

Local variables are fundamental to procedural programming, and more generally modular programming: variables of local scope are used to avoid issues with side-effects that can occur with global variables.

10.Write about Arthematic operators.

Ans:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + (Addition) | Adds values on either side of the operator | `expr $a + $b` will give 30 |
| - (Subtraction) | Subtracts right hand operand from left hand operand | `expr $a - $b` will give -10 |
| \* (Multiplication) | Multiplies values on either side of the operator | `expr $a \\* $b` will give 200 |
| / (Division) | Divides left hand operand by right hand operand | `expr $b / $a` will give 2 |
| % (Modulus) | Divides left hand operand by right hand operand and returns remainder | `expr $b % $a` will give 0 |
| = (Assignment) | Assigns right operand in left operand | a = $b would assign value of b into a |
| == (Equality) | Compares two numbers, if both are same then returns true. | [ $a == $b ] would return false. |
| != (Not Equality) | Compares two numbers, if both are different then returns true. | [ $a != $b ] would return true. |

11.Write about Assignment operators.

Ans:

Assignment Operators

There are following assignment operators supported by LINUX PROGRAMMING −

[Show Examples](https://www.tutorialspoint.com/cplusplus/cpp_assignment_operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Simple assignment operator, Assigns values from right side operands to left side operand. | C = A + B will assign value of A + B into C |
| += | Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand. | C += A is equivalent to C = C + A |
| -= | Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand. | C -= A is equivalent to C = C - A |
| \*= | Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand. | C \*= A is equivalent to C = C \* A |
| /= | Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand. | C /= A is equivalent to C = C / A |
| %= | Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand. | C %= A is equivalent to C = C % A |
| <<= | Left shift AND assignment operator. | C <<= 2 is same as C = C << 2 |
| >>= | Right shift AND assignment operator. | C >>= 2 is same as C = C >> 2 |
| &= | Bitwise AND assignment operator. | C &= 2 is same as C = C & 2 |
| ^= | Bitwise exclusive OR and assignment operator. | C ^= 2 is same as C = C ^ 2 |
| |= | Bitwise inclusive OR and assignment operator. | C |= 2 is same as C = C | 2 |

12.Write about Comparison operators.

Ans:

## **Comparison Operators**

There are following comparison operators supported by LINUX PROGRAMMING language

Assume variable A holds 10 and variable B holds 20 then −

[Show Examples](https://www.tutorialspoint.com/php/php_comparison_operators_examples.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| == | Checks if the value of two operands are equal or not, if yes then condition becomes true. | (A == B) is not true. |
| != | Checks if the value of two operands are equal or not, if values are not equal then condition becomes true. | (A != B) is true. |
| > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | (A > B) is not true. |
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | (A < B) is true. |
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (A >= B) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | (A <= B) is true. |

## **Syntax**

if [ expression 1 ]  
then  
  Statement(s) to be executed if expression 1 is true  
elif [ expression 2 ]  
then  
  Statement(s) to be executed if expression 2 is true  
elif [ expression 3 ]  
then  
  Statement(s) to be executed if expression 3 is true  
else  
  Statement(s) to be executed if no expression is true  
fi

This code is just a series of *if* statements, where each *if* is part of the *else*clause of the previous statement. Here statement(s) are executed based on the true condition, if none of the condition is true then *else* block is executed.

## **Example**

#!/bin/sh  
  
a=10  
b=20  
  
if [ $a == $b ]  
then  
  echo "a is equal to b"  
elif [ $a -gt $b ]  
then  
  echo "a is greater than b"  
elif [ $a -lt $b ]  
then  
  echo "a is less than b"  
else  
  echo "None of the condition met"  
fi

Upon execution, you will receive the following result −

a is less than b

In this chapter, we will understand shell decision-making in Unix. While writing a shell script, there may be a situation when you need to adopt one path out of the given two paths. So you need to make use of conditional statements that allow your program to make correct decisions and perform the right actions.

Unix Shell supports conditional statements which are used to perform different actions based on different conditions. We will now understand two decision-making statements here −

* The **if...else** statement
* The **case...esac** statement

## **The if...else statements**

If else statements are useful decision-making statements which can be used to select an option from a given set of options.

Unix Shell supports following forms of **if…else** statement −

* [if...fi statement](https://www.tutorialspoint.com/unix/if-fi-statement.htm)
* [if...else...fi statement](https://www.tutorialspoint.com/unix/if-else-statement.htm)
* [if...elif...else...fi statement](https://www.tutorialspoint.com/unix/if-elif-statement.htm)

Most of the if statements check relations using relational operators discussed in the previous chapter.

## **The case...esac Statement**

You can use multiple **if...elif** statements to perform a multiway branch. However, this is not always the best solution, especially when all of the branches depend on the value of a single variable.

Unix Shell supports **case...esac** statement which handles exactly this situation, and it does so more efficiently than repeated **if...elif** statements.

There is only one form of **case...esac** statement which has been described in detail here −

* [case...esac statement](https://www.tutorialspoint.com/unix/case-esac-statement.htm)

The **case...esac** statement in the Unix shell is very similar to the **switch...case** statement we have in other programming languages like **C** or **C++** and **PERL**, etc.

In this chapter, we will discuss shell loops in Unix. A loop is a powerful programming tool that enables you to execute a set of commands repeatedly. In this chapter, we will examine the following types of loops available to shell programmers −

* [The while loop](https://www.tutorialspoint.com/unix/while-loop.htm)
* [The for loop](https://www.tutorialspoint.com/unix/for-loop.htm)
* [The until loop](https://www.tutorialspoint.com/unix/until-loop.htm)
* [The select loop](https://www.tutorialspoint.com/unix/select-loop.htm)

You will use different loops based on the situation. For example, the **while**loop executes the given commands until the given condition remains true; the **until** loop executes until a given condition becomes true.

Once you have good programming practice you will gain the expertise and thereby, start using appropriate loop based on the situation. Here, **while** and **for** loops are available in most of the other programming languages like **C**, **C++** and **PERL**, etc.

## **Nesting Loops**

All the loops support nesting concept which means you can put one loop inside another similar one or different loops. This nesting can go up to unlimited number of times based on your requirement.

Here is an example of nesting **while** loop. The other loops can be nested based on the programming requirement in a similar way −

## **Nesting while Loops**

It is possible to use a while loop as part of the body of another while loop.

### **Syntax**

while command1 ; # this is loop1, the outer loop  
do  
  Statement(s) to be executed if command1 is true  
  
  while command2 ; # this is loop2, the inner loop  
  do  
     Statement(s) to be executed if command2 is true  
  done  
  
  Statement(s) to be executed if command1 is true  
done

### **Example**

Here is a simple example of loop nesting. Let's add another countdown loop inside the loop that you used to count to nine −

#!/bin/sh  
  
a=0  
while [ "$a" -lt 10 ]    # this is loop1  
do  
  b="$a"  
  while [ "$b" -ge 0 ]  # this is loop2  
  do  
     echo -n "$b "  
     b=`expr $b - 1`  
  done  
  echo  
  a=`expr $a + 1`  
done

This will produce the following result. It is important to note how **echo -n**works here. Here **-n** option lets echo avoid printing a new line character.

0  
1 0  
2 1 0  
3 2 1 0  
4 3 2 1 0  
5 4 3 2 1 0  
6 5 4 3 2 1 0  
7 6 5 4 3 2 1 0  
8 7 6 5 4 3 2 1 0  
9 8 7 6 5 4 3 2 1 0

In this chapter, we will discuss shell loop control in Unix. So far you have looked at creating loops and working with loops to accomplish different tasks. Sometimes you need to stop a loop or skip iterations of the loop.

In this chapter, we will learn following two statements that are used to control shell loops−

* The **break** statement
* The **continue** statement

## **The infinite Loop**

All the loops have a limited life and they come out once the condition is false or true depending on the loop.

A loop may continue forever if the required condition is not met. A loop that executes forever without terminating executes for an infinite number of times. For this reason, such loops are called infinite loops.

### **Example**

Here is a simple example that uses the **while** loop to display the numbers zero to nine −

#!/bin/sh  
  
a=10  
  
until [ $a -lt 10 ]  
do  
  echo $a  
  a=expr $a + 1`  
done

This loop continues forever because **a** is always **greater than** or **equal to 10**and it is never less than 10.

## **The break Statement**

The **break** statement is used to terminate the execution of the entire loop, after completing the execution of all of the lines of code up to the break statement. It then steps down to the code following the end of the loop.

### **Syntax**

The following **break** statement is used to come out of a loop −

break

The break command can also be used to exit from a nested loop using this format −

break n

Here **n** specifies the **nth** enclosing loop to the exit from.

### **Example**

Here is a simple example which shows that loop terminates as soon as **a**becomes 5 −

#!/bin/sh  
  
a=0  
  
while [ $a -lt 10 ]  
do  
  echo $a  
  if [ $a -eq 5 ]  
  then  
     break  
  fi  
  a=`expr $a + 1`  
done

Upon execution, you will receive the following result −

0  
1  
2  
3  
4  
5

Here is a simple example of nested for loop. This script breaks out of both loops if **var1 equals 2** and **var2 equals 0** −

#!/bin/sh  
  
for var1 in 1 2 3  
do  
  for var2 in 0 5  
  do  
     if [ $var1 -eq 2 -a $var2 -eq 0 ]  
     then  
        break 2  
     else  
        echo "$var1 $var2"  
     fi  
  done  
done

Upon execution, you will receive the following result. In the inner loop, you have a break command with the argument 2. This indicates that if a condition is met you should break out of outer loop and ultimately from the inner loop as well.

1 0  
1 5

## **The continue statement**

The **continue** statement is similar to the **break** command, except that it causes the current iteration of the loop to exit, rather than the entire loop.

This statement is useful when an error has occurred but you want to try to execute the next iteration of the loop.

### **Syntax**

continue

Like with the break statement, an integer argument can be given to the continue command to skip commands from nested loops.

continue n

Here **n** specifies the **nth** enclosing loop to continue from.