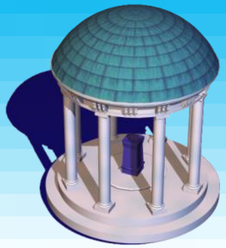


Unix Programming

Unit - I

Bourne Shell & UNIX Files

UNIX Features

- **Multi-user**

  Resources of UNIX system are shared among all user at the same time.

- **Hierarchical file system**

  Every thing in UNIX is represented hierarchically.

- **Multi-tasking**

  A user can execute many tasks at the same time. One task runs in the foreground while the rest run in the background.

- **Threads**

  Execution of jobs is using the concept of threads.

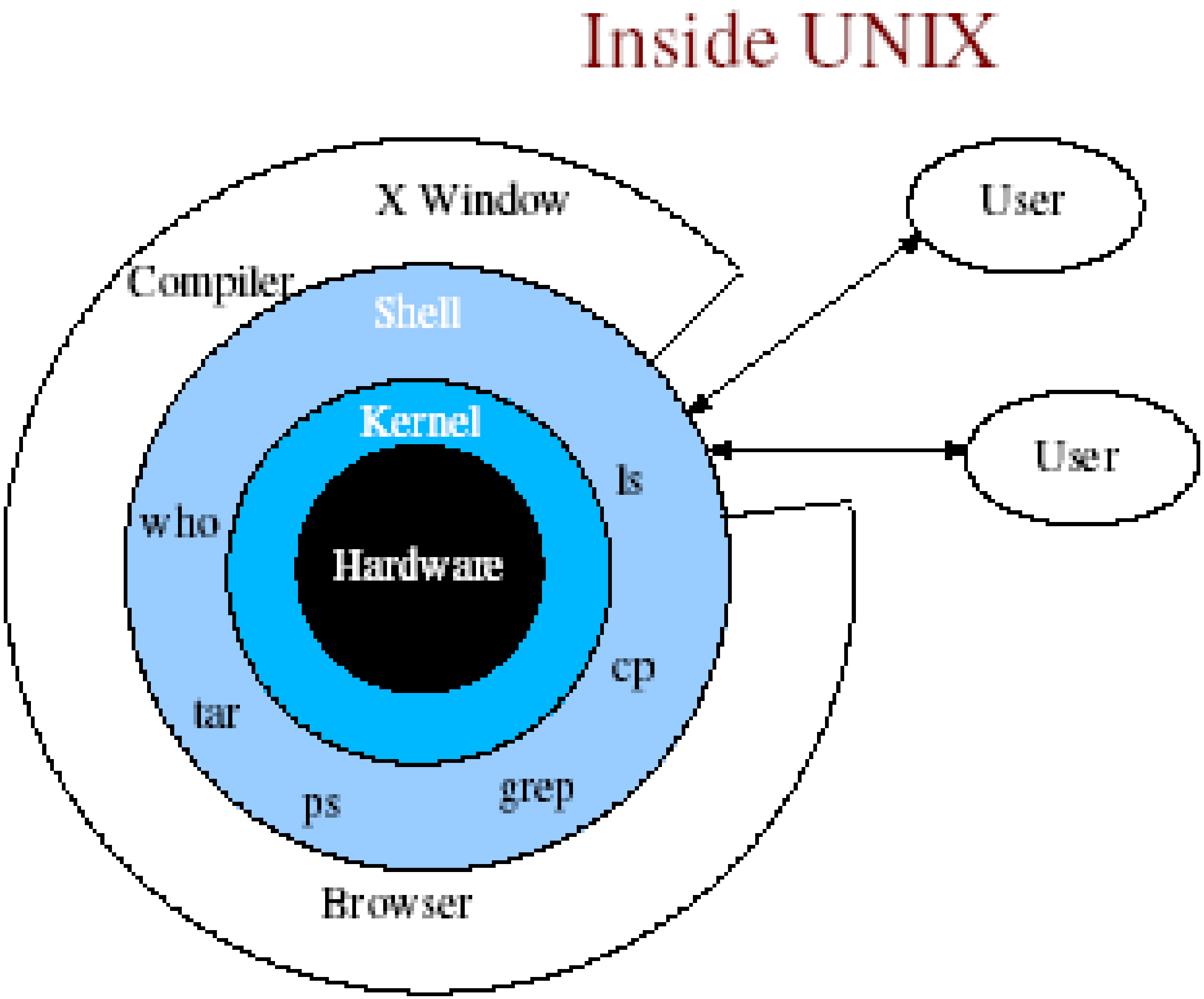
- **Built-in networking**

  All the details for networking are built in UNIX system.

- **Extensive set of utilities**

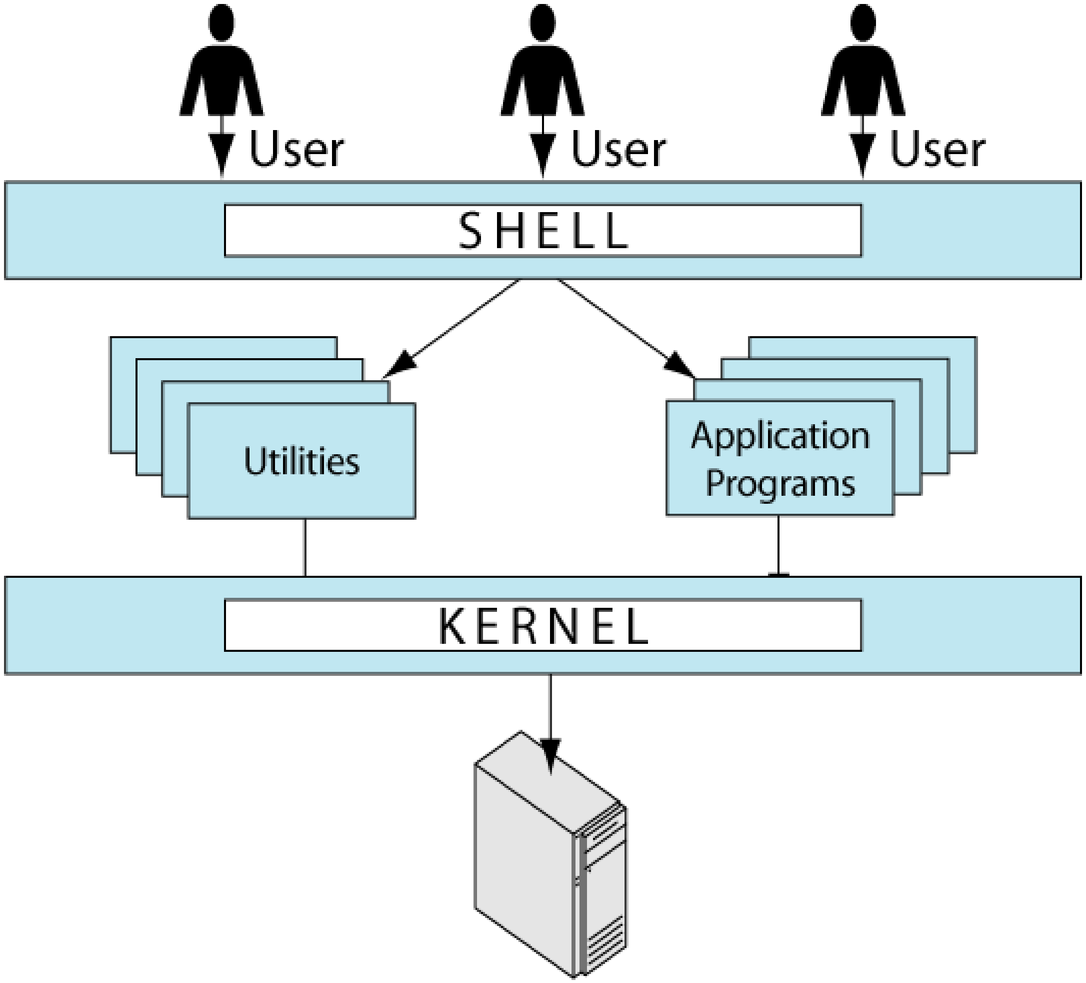
  Commands in UNIX Operating systems

Unix Structure



UNIX consists of four major components:

- Kernel
- Shell
- Standard set of utilities and
- Application programs.

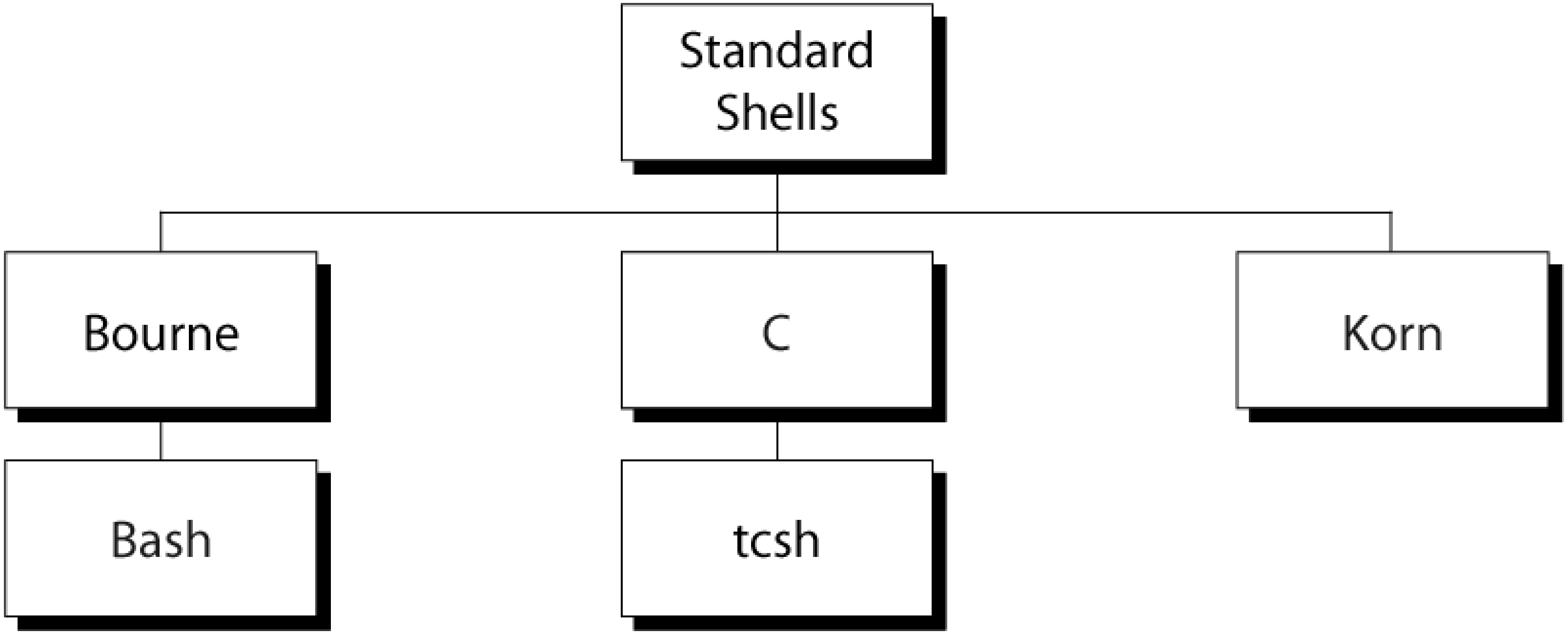


Shell

- Computers cannot translate commands into actions directly.
- An interpreter is needed for the work.
- In UNIX system this is handled by shell (command line interpreter (CLI)).
- Shell is the outer part of UNIX that is visible to the user.
- It is the interface between the user and the kernel.
- It effectively insulates the user from the knowledge of kernel functions.
- It takes a command from the user, converts and rebuilds a simplified command line.
- Finally, communicates with the kernel to see that the command is executed.

- There are two major parts of a shell.
  - **Interpreter** that reads the commands and works with the kernel to execute them.
  - **Programming capability** that allows programmers to write a shell script.
- A shell script is a file that contains shell commands that perform a useful function.

Standard UNIX Shells



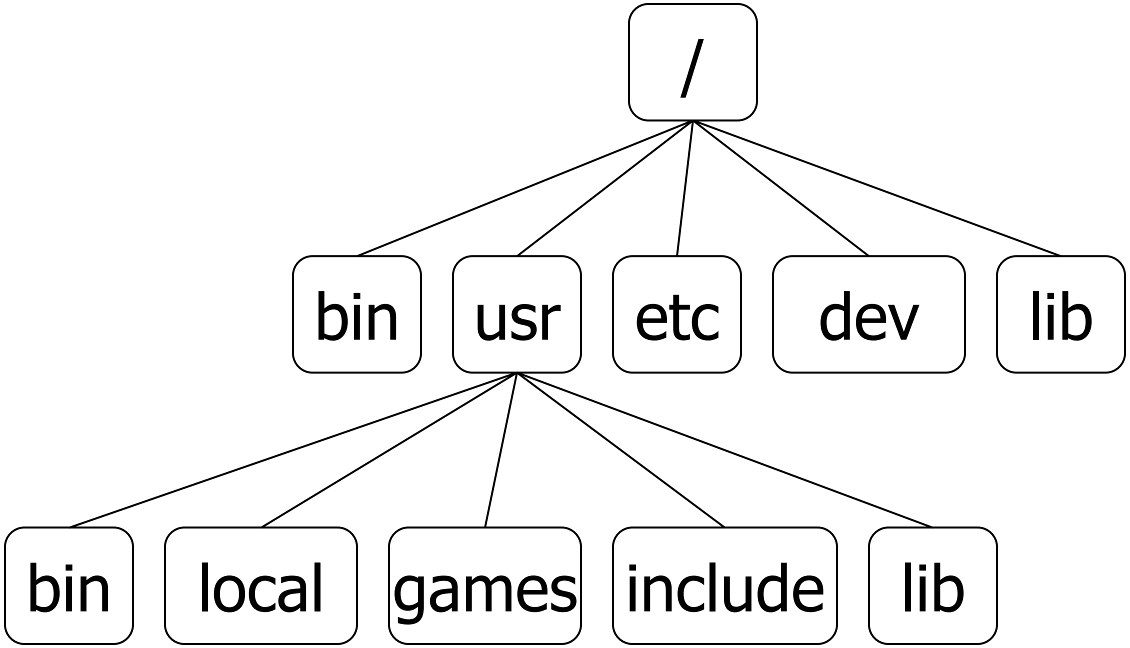
This is represented by /bin/sh(Bourne shell), /bin/csh (C shell), /bin/ksh (korn shell)

UNIX Organization

- Every thing inside Unix is considered as a file.
- File is a container for storing information.
- Four types of files
  - Ordinary file
  - Directory file
  - Linked file
  - Device files

Unix file system structure

The file system is a hierarchical structure resembling a tree, anchored at the root (“/”):

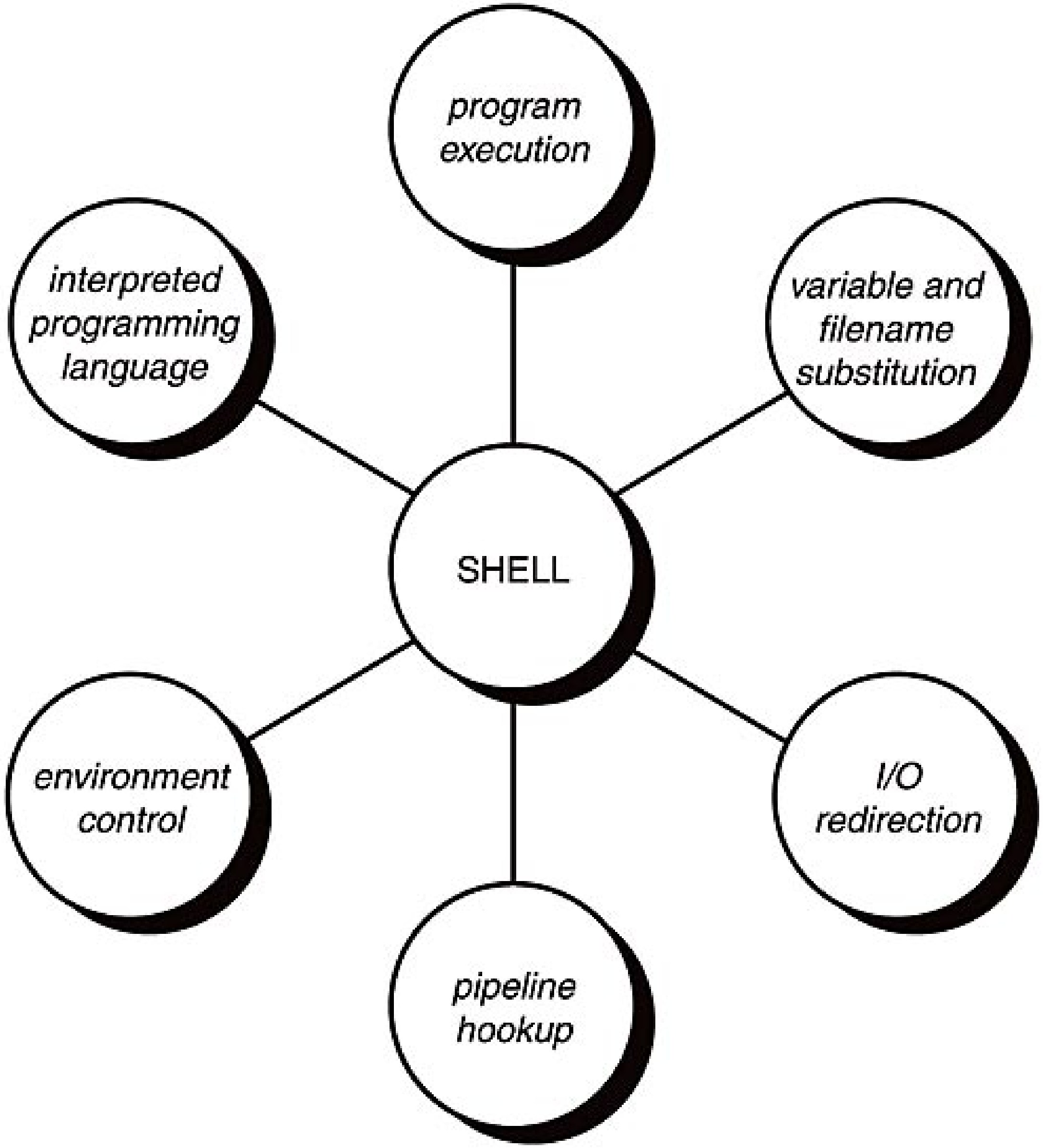


Directory	Typical Contents
/	The "root" directory
/bin	Essential low-level system utilities
/lib	Program libraries (collections of system calls that can be included in programs by a compiler) for low-level system utilities
/sbin	Super user system utilities (for performing system administration tasks)
/usr/bin	Higher-level system utilities and application programs

Directory	Typical Contents
/usr/lib	Program libraries for higher-level user programs
/tmp	Temporary file storage space (can be used by any user)
/home or /homes	User home directories containing personal file space for each user. Each directory is named after the login of the user.
/etc	UNIX system configuration and information files

Directory	Typical Contents
/dev	Hardware devices
/proc	A pseudo-file system which is used as an interface to the kernel. Includes a sub-directory for each active program (or process).

Shell Responsibilities



Program Execution:

- The shell is responsible for the execution of all programs that are requested from the terminal.
- The shell analyzes the given command line and determines the name of the program to be executed and what arguments to be passed to the program.

Variable and Filename Substitution:

- Like any other programming language, the shell allows for assigning values to variables.
- Whenever these variables are used in the command line, preceded by a dollar sign, the shell substitutes the value assigned to the variable at that point.

I/O Redirection:

- Shell scans the command line for the occurrence of the special redirection characters <, >, or >>.
- Instead of input coming from the keyboard and output and error going to the terminal, they can be **redirected** to come from or go to any file or some other device.

Standard o/p: It has 3 sources.

- The **terminal** which is the default source
- Redirected to a file using redirections **>, >>**
  - ‘>’ creates a new file with data or overrides an existing file with new data.
  - ‘>>’ creates a new file with data or appends an existing file with new data.

- To another program using a pipeline.

Standard i/p:

- The **keyboard** is the default source
- A file using redirection with **<**
- A file using redirection with **<<** (Here document)
- A here document is used when data read from standard input must be given to a command Ex: cat <<a (enter) fghfgdfghfdd a

Standard Error:

- The standard error stream can also be redirected to a file.
- Each of the standard files has a number called a file descriptor, which is used for identification.

Pipeline Hookup:

- As the shell scans the command line looking for redirection characters, it also looks for the pipe character ‘|’.
- For each such character that it finds, it connects the standard output from the command preceding the ‘|’ to the standard input of the one following the ‘|’.
- It then initiates execution of both programs.

Environment Control:

- The shell provides certain commands that helps for customizing the environment.

Interpreted Programming Language:

- The shell has its own built-in programming language. This language is interpreted, the shell analyzes each statement in the language one line at a time and then executes it.
- Programs developed in interpreted programming languages are typically easier to debug.
- They usually take much longer to execute.

Shell meta-characters

Types of meta-characters:

- File substitution
- I/O redirection
- Process execution
- Quoting meta-characters
- Positional parameters
- Special characters

Filename substitution:

These metacharacters are used to match the filenames in a directory.

Metacharacter	significance
*	matches any no. of characters
?	matches a single character
[ijk]	matches a single character either i, j, k
[!ijk]	matches a single character that is not an i, j, k

I/O redirection:

These special characters specify from where to take i/p & where to send o/p.

- **>** used to send the o/p to a specific file
- **<** used to take i/p from specific location but not from keyboard.
- **>>** used to save the o/p in a particular file at the end of that file without overwriting it.
- **<<** used to take i/p from standard i/p file. (here document)

Process execution:

- **;** used when more than a command is executed.

  Ex: date; cat f1 > f2

- **()** used to group the commands.

  Ex : (date; cat f1) >f2

- **&** used to execute the commands in background mode.

  Ex: ls &

- **&&** execute second command only if the first command is executed

  Ex: grep Unix f1 && echo Unix found

- **| |** used to execute the second command if first command fails.

  Ex: grep unix f1 | | echo no unix

Quoting:

- \ (backslash)- negates the special property of the single character following it.

  Ex: echo \? * \? o/p : ?*?

- ‘ ‘(pair of single quotes)-negates the special properties of all enclosed characters.

  Ex: echo ‘send $100 to whom?’

- ” “(pair of double quotes)-negates the special properties of all enclosed characters except $,`,\ .

```
Ex:    echo “pwd is $PWD”
        echo “pwd is `pwd` “
```

Positional parameters:

- **$0** gives the name of the command which is being executed.
- **$*** gives the list of arguments.
- **$#** gives no. of arguments.
- **$@** String containing the command line arguments

Special parameters:

- **$$** gives PID of the current shell.
- **$?** gives the exit status of the last executed command.
- **$!** gives the PID of last background process.
- **$-** gives the current setting of shell.

Shell Programming

- Grouping set of commands and storing in a file called a shell script or a shell program.

- Reasons for using shell scripts are
  - To execute a set of commands regularly
  - Typing every time every command is laborious & time consuming
  - To have control on the sequence of commands to be executed based on previous results

- **Shell Commands**

- read
- printf
- comment
- exit status command
- exit
- set
- expr
- eval
- shift
- export

Shell commands

1. **read**

   - The read statement is a tool for taking input from the user i.e. making scripts interactive.
   - It is used with one or more variables.
   - Data given through the standard input is read into these variables.

   Ex : `read name`

2. **printf:**

   - printf is used to print formatted o/p.
   - Syntax : printf "format" arg1 arg2 ...

   Ex: `printf "This is a number: %d\n" 10`

   o/p : `This is a number: 10`

3. **Exit status of a command:**

    - Every command returns a value after execution. This value is called the exit status or return value of a command.
    - This value is said to be true if the command executes successfully and false if it fails.
    - **$?** stores the exit status of a command.

4. **exit**:

    - The exit statement is used to prematurely terminate a program.

5. **set:**

- ♦ Set is used to produce the list of currently defined variables.

Ex: set

- ♦ Set is used to assign values to the positional parameters.

Ex : set a=10

6. **The do-nothing(: or #) or comment Command**

- ♦ It is a null command.
- ♦ This is used at the start of a line to introduce a comment.

7. expr:

The **expr** command evaluates its arguments as an expression:

```
Ex  :   expr 8 + 6

        o/p  :  14

Ex  : x=`expr 12 / 4 `

          echo $x

          o/p  : 3
```

9. eval:

- **eval** is useful when **command** contains something which needs to be evaluated by the shell.
- eval scans the command line twice before executing it.
- Syntax : **eval command-line**

Ex: **a=10; x=a**	**a with the value '10' and x with the value ‘a'.**
echo $x	**result will be the string 'a'**
eval echo ‘$’$x	**Output will be 10**

10. ${n}

- If more than 9 arguments are given to a program then those arguments cannot be accessed with $10, $11, and so on.
- ${n} must be used for accessing them directly, where ‘n’ is the argument number.
- To directly access argument 10, use ${10}

11. Shift command:

- The shift command allows to effectively left shift the positional parameters.
- Syntax : shift
- whatever was previously stored inside $2 will be assigned to $1, whatever was previously stored in $3 will be assigned to $2, and so on. The old value of $1 will be irretrievably lost.

Control Structures

- Syntax of simple if statement

```
if [condition]
then
  execute commands
fi
```

- Syntax of if – else statement

```
if [condition]
then
  execute commands
else
execute commands
fi
```

- Syntax if else if ladder statement

```
if [condition]
then
  execute commands
elif [condition]
then..
else  execute commands
fi
```

- Syntax of case conditional

```
case expr in

pattern1) command1;;

pattern2) command2;;

*) command;;
esac
```

- while loop

```
while [condition]
  do
      execute commands
  done
```

- until loop: while’s complement

```
until [condition]
  do
      execute commands
  done
```

- for loop

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

list here comprises a series of character strings separated by whitespace

File Conditions

-d file	Tue if the file is a directory
-f file	True if the file is a regular file
-e file	True if the file exists
-u file	True if set-user-id is set on file
-g file	True if set-group-id is set on file
-r file	True if the file is readable
-w file	True if the file is writeable
-x file	True if the file is executable
-s file	True if the file has non-zero size

Shell Scripts

- **Shell Script to display all the given command line arguments**

```
echo "The" $# "arguments entered were:" $@
```

- **Shell script for copy multiple files on to a directory and display the contents of directory**

```
cp  *.sh  files

cd files

ls
```

- Shell script to display the contents of a directory by taking the name of directory from the user

```
echo “enter the name of the directory”
read name      ls $name
```

- Shell script to display the count of words, line in a file by taking file name from user

```
echo “name of file”
read fname
wc  -wl $fname
```

- **Shell script to display the arguments given at prompt separately using for loop**

```
for word in $*
do
echo $word
done
```

- **Shell script to display multiplication table**

```
echo “enter the number”
read n
i=1
while [ $i –le 10 ]
do
c=`expr $n \* $i`
echo $n “*” $i “=” $c
    i=`expr $i + 1`
    done
```

- **Shell script to display a menu for add, sub, mul, division**

```
echo “enter 2 numbers”
read a
read b
echo”1. add 2. sub 3. mul.4 division”
read c
case $c in
1)d=`expr $a + $b`
echo $d;;
2)d=`expr $a - $b`
echo $d;;
3)d=`expr $a \* $b`
echo $d;;
4)echo “a.quotient b.remainder”
read ch
case $ch in
        a)d=`expr $a / $b`
             echo $d;;
        b)d=`expr $a % $b`
              echo d;;
        *) echo “wrong choice”;;
esac
;;
*) echo “proper option”;;
esac
```

Shell script to display the array elements

```
echo " enter the number of elements"
read n
i=0
echo "enter the elements"
while [ $i -lt $n ]
do
echo "enter the " $i "element"
read a$i
i=`expr $i + 1`
done
echo "element are"
i=0
while [ $i -lt $n ]
do
eval echo \$a$i
i=`expr $i + 1`
done
```

- **Shell script to read 3 arguments which are filename, starting line, ending line and display the line in between them by using while loop**

```
 exec < $1
 nol=0
 while read line
 do
 nol=`expr $nol + 1`
if [ $nol -ge $2 -a $nol -lt $3 ] then
   echo $line
fi
done
```

- **Shell script to merge the given two files and create a new file using exec**

```
 echo "enter file1"
 read f1
 echo "second file"
 read f2
 exec < $f1
while read line
do
echo $line >> f3
done
exec < $f2
while read line
do
echo $line >> f3
done
```

- **Write a shell script for checking the existence of a file in a directory**

```
echo "enter directory name"
read dirname
if [ -d $dirname ]
then
echo "$dirname is directory"
echo "enter file name"
read filename
ls $dirname | grep $filename
if [ $? -eq 0 ]
then
echo "$filename is in $dirname"
else
echo "$filename is not in
 $dirname"
fi
else
echo "$dirname is not a
directory"
fi
```