MODULE 1. INTRODUCTION OF UNIX AND SHELL

1.1 INTRODUCTION

UNIX is one of the oldest operating systems, which is a multi-user and multi-tasking system. The UNIX OS is basically written in C programming language. Under UNIX, the operating system consists of many utilities along with the master control program, called as the ***kernel***. The kernel provides services to start and stop programs, handles the file system and other common "low-level" tasks that most programs share, and schedules access to avoid conflicts when programs try to access the same resource or device simultaneously. To mediate such access, the kernel has special rights, reflected in the division between user space and kernel space. The user interacts with a UNIX system through a *command interpreter* known as the ***shell.***

1.2 HISTORY

In the mid – 1960's, the Massachusetts Institute of Technology (MIT), Bell Labs, and General Electric (GE) were trying to develop a time-sharing OS called as Multics (Multiplexed Information and Computer Services) for GE-645 mainframe. Because of the complexity of intended OS Multics, they dropped the project. But, few researchers like Dennis Ritchie, Ken Thompson, Brian Kernighan decided to redo the work in a smaller scale. They initially named it as Unics (Uniplexed Information and Computer Services), and eventually, it was renamed as UNIX.

In 1972, the UNIX was re-written in C language, migrating from assembly level language. This helped UNIX for serving more computing environments. During late 1970's and early 1980's, UNIX became available for commercial purposes apart from earlier scientific environment. It kept on taking different forms over the years, and in 1990's many UNIX – like systems came into existence, such as Linux.

1.3 ARCHITECTURE

The UNIX system is supported by several simple but useful components. The important components which embody UNIX OS architecture are discussed here.

1.3.1 Division of Labor: Kernel and Shell

UNIX OS distributes its major jobs into two agencies viz. ***kernel*** and ***shell.***

- **Kernel:** The kernel is also known as operating system. It interacts with the hardware of the machine. The kernel is the core of OS and it is a collection of routines/functions written in C. Kernel is loaded into memory when the system is booted and communicates with the hardware. The application programs access the kernel through a set of functions called as ***system calls.*** The kernel manages various OS tasks like memory management, process scheduling, deciding job priorities etc. Even if none of the user programs are running, kernel will be working in a background.
- **Shell:** The shell interacts with the user. It acts as a ***command interpreter*** to translate user's command into action. It is actually an interface between user and the kernel. Even though there will be only one kernel running, multiple shells will be

active – one for each user. When a command is given as input through the keyboard, the shell examines the command and simplifies it and then communicates with the kernel to see that the command is executed. The shell is represented by ***sh*** (Bourne Shell)***, csh*** (C Shell)***, ksh*** (Korn shell)***, bash*** (Bash shell).

The relationship between kernel and shell is shown in Figure 1.1.

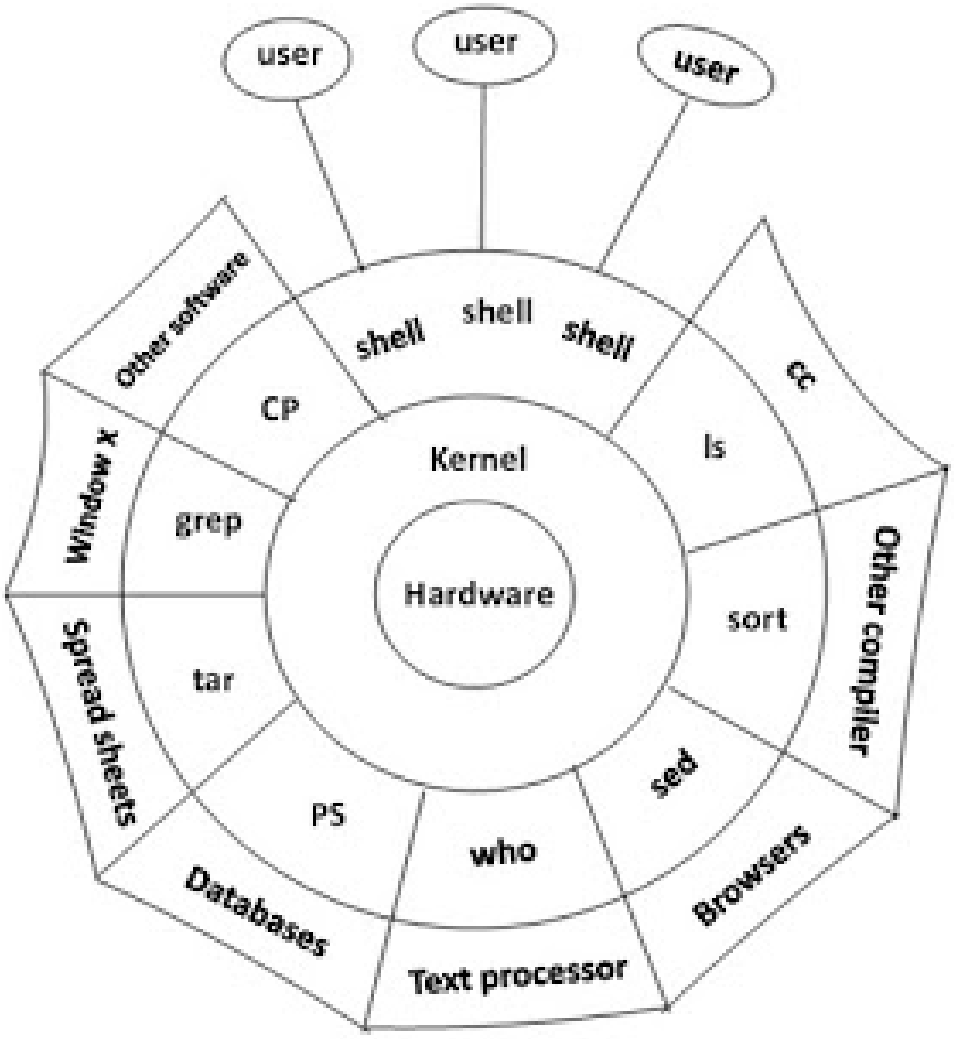

Figure 1.1 The Kernel – Shell Relationship

1.3.2 The File and Process

The file and the process are two simple entities that support UNIX.

- **File:** A file is an array of bytes and it can contain any data. Various files are related to each other by a hierarchical structure. Even a user (user name) is placed in this file system. UNIX considers directories and the devices also as the members of file system. In UNIX, the major file type is text and the behavior of UNIX is controlled mainly by text files. UNIX provides various text manipulation tools through which the files can be edited without using an editor.

- **Process:** A processing a program under execution. Processes are also belonging to separate hierarchical structure. A process can be created and destroyed. UNIX provides tools to the user to control the processes, move them between foreground and background and to kill them.

1.3.3 The System Calls

System calls are used to communicate with the kernel. There are more than thousand commands in UNIX, but they all use few set of function called as system calls for communication with kernel. All UNIX flavors (like Linux, Ubuntu etc) all use the same system calls. For example, ***write*** is a system call in UNIX. C programmer in UNIX environment can directly use this system call to write data into a file. Whereas, the C Programmer in Windows environment may need to use library function like ***fprintf()*** to write into a file. A system call ***open*** in UNIX can be used to open a file or a device. Here, the purpose is different, but the system call will be same. Such feature of UNIX allows it to have many commands for user purpose, but only few system calls internally for the actual work to be carried out in association with the kernel.

1.4 UNIX ENVIRONMENT

As UNIX is a multiuser system, to facilitate several users at a time, terminals are provided. A terminal is combination of a monitor and a keyboard. User sitting at a terminal will login to his account through his user ID and does his job. Once he finishes his work, he can logout. There will be multiple users working like this and all are accessing same CPU.

UNIX user can have his own workstation. That is, one can have dedicated CPU with RAM, hard disk etc for individual purpose. This is also known as desktop PC. Sometimes, many workstations can be connected to server via network. In such situations each workstation provides a ***terminal emulation*** facility, which will help a workstation to behave as a simple terminal. Now, the workstation will share the resources (CPU, memory etc) of the server for most of the tasks. Its own resources are used only for running terminal emulation software. When the workstation is acting like a terminal (that is, when emulation software is running), then all files created will be stored in a server and, the programs being run will use server's memory and CPU.

Logging in: A large set of networked systems is maintained by ***system administrator***. He is responsible for creating and maintaining user accounts, maintaining file systems, taking data backups, managing disk space etc. Every user of UNIX will have a separate account which has to be accessed by giving appropriate ***login-id*** or ***username*** and ***password.*** The account of system administrator is called as ***root.*** A program called ***shell*** will be always running behind and it facilitates logging process as well. It is a command interpreter between the user and the UNIX kernel.

For logging in, the prompt displayed would be –

```
login:
```

or

```
username:
```

Then the user has to provide his login id. Immediately after that, he will be asked to enter the password. And, typed password will not be hidden. For example, if the login id is *chetana,* you would see –

```
login: chetana          (press enter key)
password: *******       (invisible)
```

If any one of the login id and password is entered incorrectly, then following error message will be displayed.

```
login incorrect
login:
```

If the entered values are correct, then one may see –

```
Last login: Thu Aug 17 15:38:33 on tty2
$ _
```

The message displayed here indicates the last time the user logged in. The prompt displayed here is $ followed by a cursor blinking (the underscore _).

Various shells like Bourne Shell (*bsh)*, Bourne again shell (*bash*), Korn Shell (*ksh)*, C Shell (*csh*) etc., the prompt is $ for any normal user. For the administrator, usually the prompt would be %. Some shells shows the users current directory as –

```
[home/chetana]
```

On the other hand, the Linux systems will display some informative prompts instead of % or $. The information at the prompt will be like machine name, username and current directory. For example,

```
[chetana@localhost chetana]$
```

Logging out: It is a regular practice to log out of the system when the user finishes all the jobs. The procedure for logging out of the system will be depending on the shell used. The appearance of the text '`login:`' confirms that the user has successfully logged out. Following are the various possibilities:

- `[ctrl – d]` (pressing the keys ctrl and d together, and then press enter key)
  `login:`
- **`logout`** (the command *logout* and then press enter key)
  `login:`
- **`exit`** (the command *exit* and then press enter key)
  `login:`

The C shell uses the command *logout* and the *[ctrl-d]* and *exit are* used by other shells.

1.5 FEATURES OF UNIX

As UNIX is an OS, it obviously possesses features of any other OS. But, it has its own unique features as discussed in the following sections.

1.5.1 A Multiuser System

UNIX is basically a multiprogramming system. Here, either

- Multiple users can run separate jobs or
- Singe user can run multiple jobs

In UNIX, many processes are running simultaneously. And, the resources like CPU, memory and hard disk etc are shared between all users. Hence, UNIX is a multiuser system as well.

The Unix system breaks up one time unit into several segments and each user is allotted one segment. At any point of time, the machine will be doing the job of one user. When the allotted time expires, the job is temporarily suspended and next user's job is taken up. This process continues till all processes gets one segment each and once again the first user's job is taken up. The kernel does this task several times in one second such a way that the users will never come to know about it and users cannot make out the delay in between.

1.5.2 A Multitasking System

Unix is a multitasking system, wherein a single user can run multiple jobs concurrently. A user may edit a file, print a document on a printer and open a browser etc – all at a time. In multitasking environment, a user can see one job running in the foreground and all other jobs run in the background. The jobs can be switched between background and foreground; they can be suspended or terminated.

1.5.3 The Building-Block Approach

Unix is a collection of few hundred commands, each of which is designed to perform one task. More than one command can be connected via the | (pipe) symbol to perform multiple tasks. The commands which can be connected are called as ***filters*** because, they filter or manipulate data in different ways. Many Unix tools are designed such a way that the output of one tool can be used as input to another tool. For this reason, UNIX commands do not generate lengthy or messy outputs. If a program is interactive, then user's response to it may be different. In such situations, the output of one command cannot be made as input to another command. Hence, UNIX programs are not interactive.

1.5.4 The UNIX Toolkit

Unix contains diverse set of applications like text manipulation utilities, compilers and interpreters, networked applications, system administration tools etc. The Unix kernel does many tasks with the help of these applications. Such set of tools are constantly varying with every release of UNIX. In every release, new tools are being added and old tools are either removed or modified. Most of these tools are open-source utilities and the user can download them and configure to run on one's machine.

1.5.5 Pattern Matching

Unix has very sophisticated pattern matching features. The character like * (known as a ***metacharacter***) helps in searching many files starting with a particular name. Various characters from a metacharacter set of Unix will help the user in writing ***regular expressions*** that will help in pattern matching.

1.5.6 Programming Facility

The Unix shell is a programming language as well. It provides the user to write his/her own programs using control structures, loops, variables etc. Such programs are called as ***shell scripts.*** Shell scripts can invoke Unix commands and they can control various functionalities of Unix OS.

1.5.7 Documentation

Unix provides a large set of documents to understand the working of every command and feature of it. The ***man*** command can be used on an editor to get the manual about any Unix command. Moreover, there are plenty of documents, newsgroups, forums and FAQ (Frequently Asked Questions) files available on internet, where one can get any information about Unix.

1.6 BASIC COMMANDS

The UNIX system is command-based. That is, various tasks are carried out based on the input command by the user. All UNIX commands are case-sensitive and most of them are in lower case. Some of the basic commands in UNIX are discussed here.

1.6.1 date

The ***date*** command in UNIX is used to display the current date and time of the system. The UNIX system maintains an internal clock that runs continuously. When the system is shutdown, the battery backup keeps the clock ticking. This clock actually stores the number of seconds elapsed since 1st January 1970. A 32-bit counter store these seconds and it is expected to overflow sometime in 2018.

The format of the *date* command is "Day Month date hr:min:sec IST year". For example –

```
$date
Mon Jun  30 11:35:32 IST 2017
```

This command will display the output as shown above, and will not allow the user to modify the current system date. When the system administrator uses the same command, apart from displaying the date, it allows to modify the date and time. This is possible because of the privileges given to system admin.

Suitable format specifiers can be used as an argument to *date* command to get the date/month/year etc. in required format. The format specifier is preceded by a + symbol, followed by the % operator and a single character describing the format. Following are some of the formats –

- To display only the month, one can use +%m as a specifier. For example,

```
$date +%m
10                          //indicates October.
```

- To display month name, use +%h as below –

```
$date +%h
Oct
```

- +%d for day of month (1 – 31)

- +%y for last two digits of the year
- +%H, +%M and +%S indicates hour, minute and second respectively.
- One can combine more than one option by enclosing them in double quotes, and keeping + symbol outside the quote. For example, combination of month and month name –

```
$date +"%h %m"
Oct 10
```

1.6.2 passwd

This is the command used for changing the password of a user. When this command is entered in the terminal, the system will ask for the existing password. After entering the current password correctly, the system asks for new password to enter. The new password entered must satisfy the rules framed for proper password in that system. Then system will then ask for reentering the password for confirmation purpose. When everything goes correctly, the $ prompt is returned. Next time, when this user logs in, he/she has to give the newly set password. The following is the example showing working of *passwd* command in Linux.

```
[chetana@server4 ~]$ passwd
Changing password for user chetana.
Changing password for chetana
(current) UNIX password:*******
New UNIX password:**********
Retype new UNIX password:**********
passwd: all authentication tokens updated successfully.
[chetana@server4 ~]$
```

Note that the above set of instructions and statements is just an example and the behavior of *passwd* command is heavily depending on the system.

Depending on the system how it is configured, certain checks are made on the string (password) that you enter. Some of the common messages are –

- `UX: passwd: ERROR: Passwords must differ by at least 3 positions`
- `passwd (SYSTEM): The first 6 characters of the password must contain at least two alphabetic characters and at least one numeric or special characters.`
- `passwd (SYSTEM): Password too short – must be at least 6 characters`
- `BAD PASSWORD: it does not contain enough different characters`
- `BAD PASSWORD: it is based on a dictionary word`
- `BAD PASSWORD: it too similar to the old one.`

This indicates that one cannot choose any password that he/she wishes. One has to abide by the rules set by the system.

When a user enters the password, that string is encrypted by the system, and it is stored in a file ***shadow*** in the */etc* directory. Even if a user sees this encryption, one cannot decrypt it.

When the *passwd* command is used by the system admin, its behavior will be different and it is discussed in later chapters.

1.6.3 ls

The *ls* command (meaning – list) displays all files and directories in the current directory. For example,

```
$ls
mytest.c
test1.c
Thesis
Demo
```

One can observe that here there are two files and two directories under the current directory. The *ls* command has many other options and usage which will be discussed in later chapters.

1.6.4 mkdir

This is the command used to create a new directory. Directories can be used to have a collection of files under a common name.

```
$mkdir docs
$_
```

The *mkdir* command has created directory viz. *docs*. But, the command has not shown any output, but just displays the prompt. Many a times, UNIX commands do not display any output, but does the job internally. Here, *mkdir* has created the directory and that can be confirmed using *ls* command.

Note that, UNIX internally treats a directory also as a file.

1.6.5 pwd

Present Working Directory can be displayed using the command *pwd*. For example –

```
$pwd
/home/chetana
```

The above output indicates that there is a directory with your login name, and it is within *home* directory. In UNIX, when a new user account is created, a directory is automatically created with that login name.

1.6.6 cd

To move from current working directory to a new directory, the *cd* (change directory) command is used. After using *cd* command, the *pwd* command can be used to verify the new location. For example,

```
$cd docs
```

```
$pwd
/home/chetana/docs
```

The above output is the pathname of current working directory starting from the root directory /.

1.6.7 echo

The basic job of *echo* command is to display the string passed to it as an argument. For example,

```
$ echo hello
hello
```

The echo command can be used to evaluate a variable given to it as an argument. For example,

```
$ x=5
$ echo $x
5
```

Note that, here, to initialize a variable, we will just use variable name. But, to evaluate and display its value, the $ symbol must be attached with the variable name.

The output of echo command can be used to create a new file as well. For example,

```
$echo Hello World > file1
$cat file1
Hello World
```

Here, in the first line, echo command is used to create *file1* with the content *Hello World*. Then, *cat* command is used to display the contents of *file1.*

1.6.8 wc

The command *wc* is used for counting number of lines (in fact, total number of new-line characters), words and characters in a file. For example –

```
$wc file1
1    2    12  file1
```

The above output indicates that *file1* contains one line, two words (*Hello* and *World*) and 12 characters (including space and new-line character).

1.6.9 cal

This command is used to display the calendar of current month. Various arguments can be given to *cal* command as listed below –

- $cal Dec 2017 : Displays December month calendar of the year 2017
- $cal 2017: Displays the calendar for all months of 2017.

For example,

```
$cal
```

```
      May 2017
 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
```

1.6.10 calendar

This command is a useful in setting reminders. It is available only in UNIX, but not in Linux. A user has to create a file named *calendar* with all the tasks he would like to remember. Then, when this command is used, it invokes all the tasks listed for today (current system date) and tomorrow. While creating a task list in a file, one has to be careful about the date format that can be recognized by the system. Consider an example –

```
$cat calendar
Meeting with principal on Oct 23
October 24 – call for staff meeting
Give notes to students on Oct 24th
Car service – Oct 23, 2017
Oct 25 – Krupa birthday
```

Here, a file has been created with a name *calendar* and with some five lines of contents. Now, assume that today is October 23, 2017 and we are running the command as –

```
$calendar
Meeting with principal on Oct 23
October 24 – call for staff meeting
Car service – Oct 23, 2017
```

It has displayed the output containing the tasks to be done on 23rd and 24th October 2017. Observe that, it didn't display one task mentioned as 'Oct 24th". Because, it is not in a recognizable format for the system.

Note that, by default, the *calendar* command assumes the current year, when the year is not mentioned in the file. Also, when the current date is a Friday, then it displays the activities of Monday (the next working day), including the weekend days - Saturday and Sunday.

1.6.11 bc

UNIX provides two types of calculators – a graphical (GUI) calculator (similar to the one available in windows OS) and a character based ***bc*** command. A visual calculator can be available using ***xcalc*** command and it is available only on X Window system, but not on command-line based terminals.

The calculator available through ***bc*** command is a very powerful, but sadly a most neglected tool in UNIX. When ***bc*** command is invoked without any argument, it does

nothing but waits for the input from the keyboard. Once the job is done, *ctrl+d* has to be pressed to release the command and to get a prompt.

The usage of ***bc*** command is illustrated here with examples.

- **Basic operations:**

```
$bc
3+5
8
5*6
30
6-10
-4
[ctrl+d]
```

- **To perform more than one operation in a single line:**

```
$bc
2^4; 3+6        //using semicolon as a separator
16
9
[ctrl+d]
```

- **Setting scale for required precision during division operation:**
  By default, ***bc*** performs truncated division (or integer division). For example,

```
$bc
9/5
1
```

  Here, the output 1, instead of 1.8. To avoid such truncation, one can set the precision after the decimal point. For example,

```
$bc
scale=2
9/5
1.80
22/7
3.14
```

- **Converting numbers from one base to the other:**
  One can change the base of a number by setting ***ibase*** (input base) or ***obase*** (output base). For example –

```
$bc
ibase=2         //setting input base as 2
1100
12              //decimal equivalent of 1100
11001110
206             //decimal equivalent of 11001110
```

The reverse is possible through ***obase*** as shown below –

```
$bc
obase=2          //setting output base as 2
14
1110             //binary equivalent of 14
308
100110100        //binary equivalent of 308
obase=16         //setting output base as 16 (hexa)
14
E                //hexadecimal equivalent of 14
```

- **Storing variables:**
  One can store values in variables and then use them. But, ***bc*** supports only single lowercase letters (a-z) and hence, one can use only 26 variables at a time. For example –

```
$bc
a=5; b=3; c=2
p=a+b*c
p                //use variable name to display the result
11
```

Note that, ***bc*** is a pseudo-programming language that supports arrays, functions, conditional structures (if) and looping structures (for and while). It also supports library of some scientific functionalities. It can handle very large numbers. If the result of some calculation is 900 digits, the ***bc*** command shows every digit of it!!

1.6.12 who: To know users of the system

Normally a UNIX system is used by multiple users at a time. One user may needs to know the list of other users who are using the system currently. The *who* command is used for this purpose. This command displays name of the users (login ID used to log in), name of the terminal and date and time of login. For example –

```
$who
root      :0             Sept 04 10:12
chetana   tty01          Sept 04 11:11
raghu     tty02          Sept 04 12:35
ram       tty03          Sept 04 14:08
```

Here the first column shows the user-ids of four users who are currently logged in. In the second column, *tty01* etc. are name of the terminals and the last column shows date and time of their respective login. This indicates that currently (while giving *who* command), four users have logged in. The term *tty* indicates *teletype.* The machine identifies a person with his/her username. So, user will be the owner of file he has created. When a file created by one user, say *chetana* is sent to another user, the machine will inform the recipient that a mail has arrived from *chetana.*

Some of the systems display as below when *who* command is used –

```
$who
chetana   pts/1          Sept 04 11:11
```

Here, `pts/1` is the name of the terminal. The term *pts* stands for *pseudo-terminal slave.* Most of the UNIX/Linux systems have software implementation as an interface to interact with real terminal. It is called as *pseudo-terminal* or *pseudo-teletype* represented by *pty.* The *pts* is a slave part of *pty.*

The header option *–H* can be used along with *–u* option to get more information on the ***who*** command.

```
$who –Hu
NAME     LINE     TIME               IDLE    PID        COMMENT
root     :0      2007-01-12 04:49    ?      5595
chetana  pts/1   2007-01-13 05:39    .     24081  (172.16.4.205)
```

Here, first three columns are same as before. The fourth column *IDLE* indicates from how long the user is idle. The dot (.) indicates the respective user was active in the last one minute. The question mark (?) indicates that the user is idle from quite a long time, which is unknown. The fifth column PID (process identifier) will be discussed in later chapters. The comment line indicates some special comment, if any. In the above example, for the user *chetana*, it is showing the IP address of the machine.

It is possible to have multiple user names for a single user. So, sometimes, a user would like to know who he/she is – that is, with which user name he/she logged in. For this purpose, *who am i* command is used. For example,

```
$who am i
chetana   tty01     Sept 04 11:11
```

The additional terms *am i* used with *who* are known as ***arguments.*** The arguments change the behavior of UNIX commands and there are many UNIX commands with arguments that will be discussed in later chapters.

1.6.13 tty: Knowing terminal name

The command *tty* (teletype) is used to know name of the terminal. For example,

```
$tty
/dev/tty01
```

The above statement indicates that *`tty01`* is the name of the terminal and it is within the directory *dev.* The *dev* is under *`root`* directory.

NOTE that, UNIX treats all devices as files, and *`tty01`* is one of the files under file system. In UNIX, just like users, even terminals, disks and printers also have the name and all these are treated as files. Even the commands are also files in UNIX.

Some the systems may display the statement as below when *tty* command is given –

```
$tty
/dev/pts/1
```

Here, `pts/1` is the name of the terminal.

1.6.14 Knowing Shell and Terminal Type

It is known that, there are different types of shells. If a user would like to know the shell on which he/she is working, one can use ***$SHELL*** argument with ***echo*** command. The basic feature of ***echo*** is to simply display the string argument passed to it. The term ***$SHELL*** is a variable used by UNIX to know name of the shell.

```
echo $SHELL
/bin/bash
```

Here, *bash* is name of the shell stored in *bin* folder. In UNIX, all commands are files and shell is also a file. Shell variables have values associated with them just like programming variables. The *echo* command is used here to evaluate the value of the variable by using $ symbol attached with ***SHELL.***

Another variable ***TERM*** is used for knowing type of the terminal being used. Terminal type or emulation specifies how your computer and the host computer (server) exchange the information. Terminal type of one's computer has to be set properly so that both terminal and the server can communicate in a same way. Otherwise, your terminal will not have enough information to perform actions like clearing the screen, moving the cursor, placing a character etc.

```
$echo $TERM
vt220
```

The type of terminal may be vt100, vt220, xterm etc. Each one has different meaning and feature, which is beyond the scope of this study.

1.6.15 stty

This command is used to set terminal characteristics. The terminal is a device with which user communicates. Each terminal is configured differently depending on the user's choice. For example, a user can decide

- what should be the abort key (like Ctrl+c or Delete key etc),
- whether a character has to be deleted or not when backspace key is used
- what should be the end-of-file character when *cat* command is used (like Ctrl+d or Ctrl+a etc)

The ***stty*** command helps the user in setting all such characteristics and also to revoke existing characteristics.

NOTE that the setting users does may not be effective in some terminals (especially on Linux machines).

The command ***stty*** shows the output as given below –

```
$stty
speed 9600 baud; line = 0;
-brkint -imaxbel
```

Initially it displays ***baud rate*** of the terminal as 9600. The number of characters that a terminal can transmit per second is known as baud rate. The ***line*** indicates the line discipline of Unix terminal. It does the input processing in the kernel. The ***brkint*** indicates that whether or not (with – or without –) an interrupt signal has to be generated when there is a break in the script. The ***imaxbel*** indicates to beep and do not flush a full input buffer on a character.

The ***–a*** (all) option with this command will display the current settings as shown below –

```
$stty –a
speed 9600 baud; rows 24; columns 80; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol =
<undef>;
eol2 = <undef>; swtch = <undef>; start = ^Q; stop = ^S; susp = ^Z;
rprnt = ^R;
werase = ^W; lnext = ^V; flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 -hupcl -cstopb cread -clocal -crtscts -cdtrdsr
-ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl
ixon –ixoff -iuclc -ixany -imaxbel -iutf8
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0
bs0 vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop
–echoprt echoctl echoke
```

In the first line, we can see number of rows (24) and number of columns (80) that a terminal can display at a time. After this, there are many settings, which are shortcut keys for certain tasks. In this terminal, the interrupt key is *Ctrl+C,* the erase character is *Ctrl+?,* the kill character is *Ctrl+u,* the end-of-file is *Ctrl + d* etc. There are some keywords in the above display, where some are with – (hyphen) symbol and some without that. If an option is without – (hyphen) symbol means, it is turned on. One can use ***stty*** to on or off these options. Some of the examples are discussed here.

- **Whether backspacing should erase character (echoe):** You might have observed that in some of the terminals, the backspace key will not erase the characters. This is the behavior of ***echoe*** option available in ***stty*** command. If you want the backspace to remove the character, use –

  ```
  $stty echoe
  ```

  If you want the backspace to not to remove a character, use –

  ```
  $stty –echoe
  ```

  NOTE that, this option is not effective in Linux.
- **Entering a Password through a Shell Script (echo):** The option ***echo*** has to be set to let shell programs to accept a password – like string (invisible characters). By default, the option is turned on, but one can turn it off using –

```
$stty –echo
```

After using this command in a terminal, every character/text what you write will be invisible in terminal. To make characters visible, set as –

```
$stty echo
```

Note that, this sentence is invisible, but subsequent commands will be visible.

1.6.16 uname

The command ***uname*** is a short-form for UNIX name, which displays the details like name and version of the machine and OS currently running. It can display various details based on the option given to it as an argument. Consider following situations:

- `$uname`
  `Linux`
  The command without any options displays the name of underlying OS.
- `$uname –a`
  `Linux server4 2.6.18-128.el5xen #1 SMP Wed Dec 17 12:01:40 EST 2008 x86_64 x86_x`
  This has displayed details like kernel name, node name, kernel release, kernel version etc.
- `$uname –n`
  `server4`
  When your system is connected to network, it prints the name of the machine in network. This name is required while copying the files from remote machine using ***ftp*** command.

1.6.17 printf

The ***printf*** command can be used as an alternative to ***echo*** command. Several usages of *printf* are explained here with suitable examples.

- `$printf “Hello World”`
  `Hello World$`
  Note that, *printf* has printed the string and the prompt got displayed immediately after the string without inserting a newline character (Whereas, *echo* adds a newline character). To have a newline character, one can use the escape sequence \n as below –

  ```
  $printf “Hello World\n”
  Hello World
  $
  ```

- *`printf`* can be used without double quotes. But, *printf* terminates when a space is encountered. For example –

  ```
  $printf Hello World
  Hello$
  ```

  Note that the term *World* didn’t get printed here.

- Just like *printf()* function of C language, in UNIX also, *printf* command can use format specifiers like %s, %d, %f, %o, %x etc. And, the values of variables can be displayed along with *printf* command. For example,

```
$printf "Current shell is %s \n" $SHELL
Current shell is /bin/bash
```

Observe that, there is no comma separator between the string to be printed and the variable.

- The format specifiers can be used to convert the numbers as below –

```
$printf "Value of 138 is %x in Hexa and %o in Octal\n" 138 138
Value of 138 is 8a in Hexa and 212 in Octal
```

1.6.18 script

The ***script*** command is used to record the session in a file. When you have are doing some important work, and would like to keep a log of all your activities, you should use ***script*** command immediately after logging in. For example,

```
$script
Script started, file is typescript
$
```

Now onwards, whatever you type, that will be stored in the file *typescript*. Once the recording is over, you can terminate the session by using ***exit*** command.

```
$exit
Script done, file is typescript
$
```

To view the file *typescript*, one can use ***cat*** command.

Note that, the usage of *script* command overwrites any existing file with name *typescript.* If you want to append the new content to existing file, then use ***–a*** as below –

```
$script –a
```

Now, the previous *typescript* will be appended with the activities of this session.

If you want to create your own file instead of *typescript* file, then give the required filename as –

```
$script mylogfile
```

Now, the activities of this session will be stored in the file *mylogfile.*

NOTE that, some activities like the commands used in the full-screen mode like ***vi*** editor will not be recorded properly when we record session using *script* command.

1.6.19 spell and ispell

The ***spell*** command is used to check the spelling in a text file. When the name of a file is given an argument to this command, it lists out all the mistakes (words without proper

meaning as per the understanding of UNIX). To understand this command, let us first create a file as below –

```
$cat >test.txt
hello hw are yu?
Im doin fine
```

Now, apply ***spell*** command on the file *test.txt* as shown –

```
$ spell test.txt
doin
hw
Im
yu
```

One can observe that the words with spelling mistakes have been displayed in the order. Now, if you want to correct these words, ***ispell*** command can be used. ***ispell*** is actually an interactive editor, which displays various suggestions for a mistaken word. Then, user has to choose one of the possible suggestions listed and the mistaken word will be replaced by the corrected word. For example,

$ispell test.txt

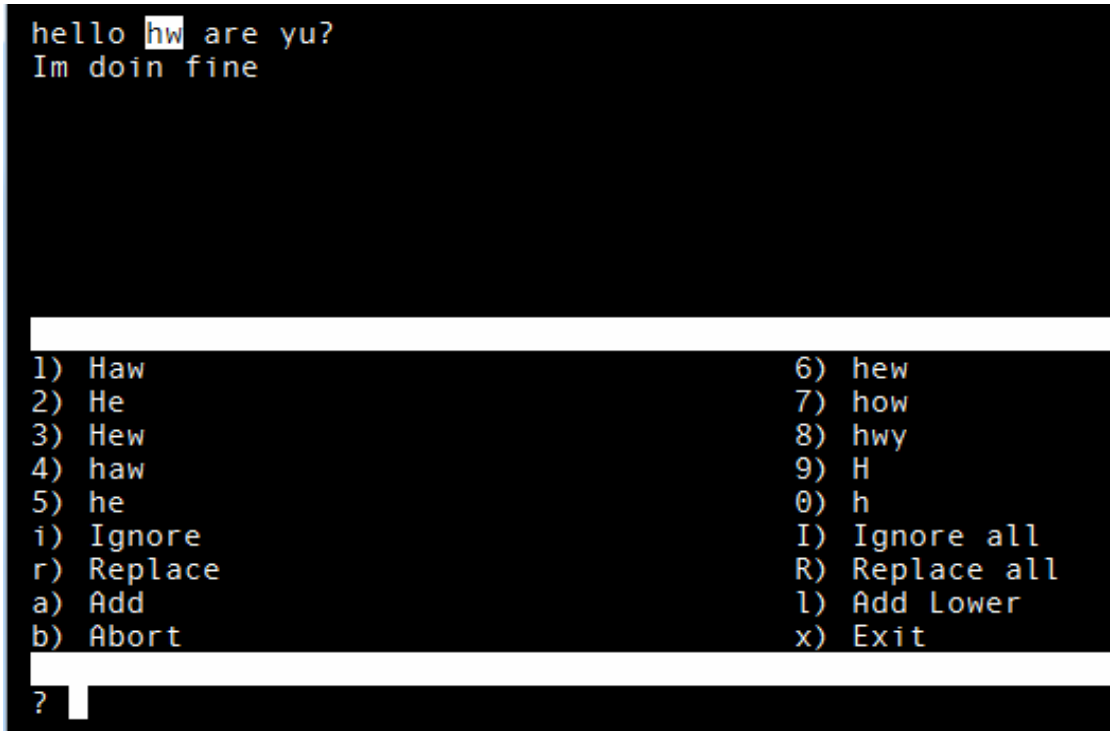

Figure 1.2 Showing suggestions for spelling mistakes

Observe in Figure 1.2 that, the word *hw* is highlighted and the options are shown. Here, we wanted the word *how*, which is suggestion number 7 in the editor. So, one has to type 7 in front of the question mark displayed at the end. Immediately, the word *hw* will be replaced

by *how* and the next mistake will be highlighted with suitable suggestions as shown in Figure 1.3.

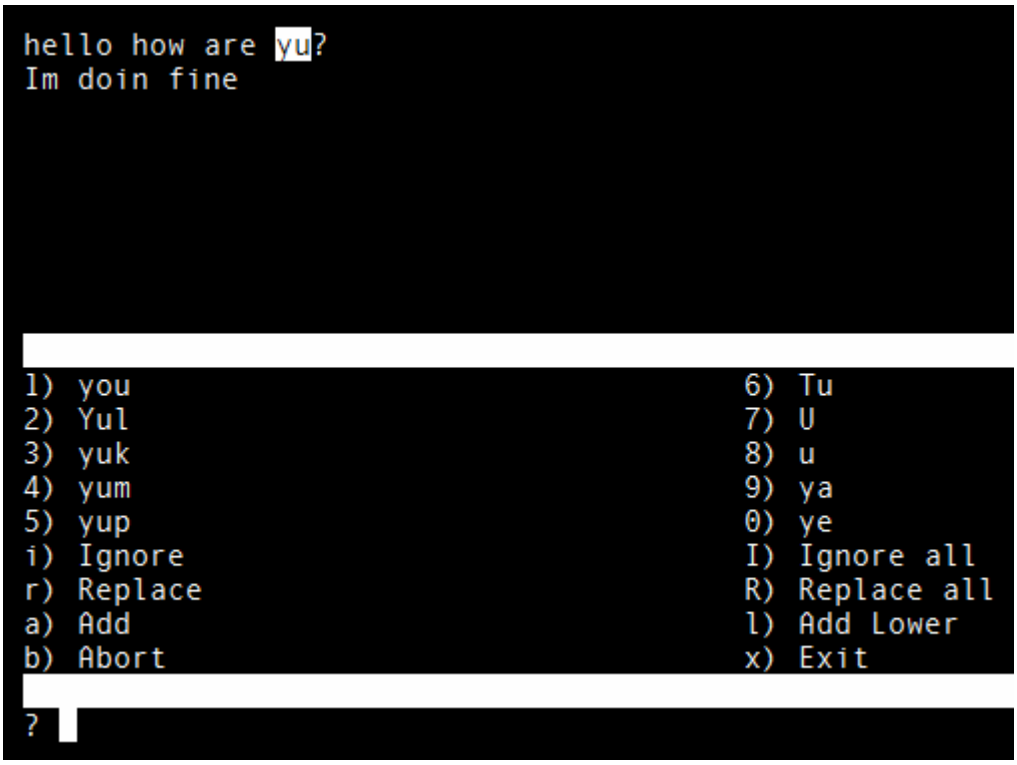

Figure 1.3 Correction made as per user's selection

The procedure continues till all mistakes are rectified. You can also observe that, there are few other options like *Ignore, Ignore All, Replace, Replace All* etc. User can take decision accordingly for every mistaken word.

1.6.20 `clear` and `tput`

The command ***clear*** is used to clear the screen and to keep the cursor at top left corner of the screen. The usage is –

```
$clear
```

The command ***tput*** also clears the screen, but only by giving *clear* argument to it. That is,

```
$tput clear
```

The about command will clear the screen and keep the cursor at top left corner of the screen.

The ***tput*** command can be used to keep the cursor at a required position by using ***cup*** argument. For example,

```
$tput cup 10 20
```

This will keep the cursor at 10^{th} row and 20^{th} column of the screen. This is usually used in shell script to display the text message in a required position. For example, in a shell script,

if the above command is used before an *echo* command, then the text to be displayed through *echo* will appear at 10th row and 20th column position.

The command ***tput*** uses the argument ***smso*** to render the text in reverse combination of background and foreground colors. The term *smso* indicates "Start Mode Stand-Out". Assume that background color of the terminal is white and foreground color of the text is black. When the argument ***smso*** is set using ***tput***, the further entries in the command prompt will appear as – background with black color and foreground text with white (That is reverse of the previous settings). To stop this setting, the argument ***rmso*** (Reset Mode Stand-Out) is used. For example,

```
$tput smso
$echo "hello world"        //foreground background colors will be reverse from now
hello world
$tput rmso                 //becomes normal
```

1.6.21 cat

It is a short-form of the term *concatenate.* This command is basically used for viewing the contents of a file. But, it has many other usages like creating a file, joining more than one file etc. Here, few of the usages of *cat* command are discussed.

- **To create a new file:** Following is an example to create a new file –

```
$ cat >test
hello how are you?
I'm doing good.
what about you?
[Ctrl+d]
$
```

  When *cat* command is used with > operator and the file name, the cursor waits for the user to enter the contents of the file. Once the entry is over, user will press *ctrl+d* to exit. This action would create a file with a specified name and the contents. Note that, if the file with that name already exists, it will be overwritten by new contents.

- **To display contents of a file:** The ***cat*** command is used with filename to display the contents of the file as shown below –

```
$cat t1
This is first file
$
```

- **Display Multiple files at once:** The ***cat*** command can be used to display contents of multiple files by passing different filenames as arguments through space-separated list. For example,

```
$ cat t1 t2
This is first file
This is second file          //content of the file t2
$
```

- **Giving line Numbers:** The contents of any file can be displayed along with line number by using ***–n*** option as below –

```
$cat –n test
     1    hello how are you?
     2    I'm doing good.
     3    what about you?
$
```

- **Use redirection operator to copy the contents of one file to other:** A duplicate copy of one file can be created using redirection operator as below –

```
$cat t1 > t3
$ cat t3                    // check the contents of t3
This is first file
$
```

Now, a new file *t3* got created and the contents of *t1* are copied into *t3.* Note that, if the file with name *t3* already existed, then it will be overwritten.

- **Append contents of one file at the end of other file:** Using the >> operator, the contents of one file can be appended at the end of other file as shown below –

```
$ cat t1 >>t2
$ cat t2                    //display the contents of t2
This is first file          //original contents of t1
This is second file         //appended portion from t2
$
```

- **Creating single file with the contents of multiple files:** One can create a new file which contains the contents of more than one file as below –

```
$ cat t1 t2  > t4
$ cat t4
This is first file
This is second file
$
```

1.6.22 touch

This command is used to create one or more empty files. These files can be then filled with required content through vi editor. Usage is as follows –

```
$touch t1 t2 t3
$
```

Now, 3 files with names *t1, t2* and *t3* got created and that can be verified by using *ls* command.

1.7 INTRODUCTION TO SHELL SCRIPTING

Various basic commands of UNIX have been discussed till now. And, it is understood that the shell is helping to perform all these command-line tasks. But, shell is not just limited to command interpretation; instead, it also has many internal commands and has a programming capability. Shell supports many programming constructs like variables, conditional structures, looping structures etc, which are borrowed from C language. But, most of these constructs are much simpler and compact to use compared to that in C.

Shell programs run in interpretive mode – one statement at a time. That is, the output of every line is displayed immediately after executing that line. Shell will not wait till the end of the program. Also, there will not be any intermediate files (like .obj or .exe etc) for a shell script when you run it. Since shell scripts are interpreted codes, they run slower compared to higher–level languages. Still, shell scripts are powerful because external UNIX commands mingle with internal constructs of shell very easily.

1.7.1 Shell Variables

Variables are used for storing the values. In shell programming, variable name must start with a letter, but can contain digits and the underscore. A variable is assigned a value using = operator, and evaluated by prefixing $ symbol with its name. For example,

```
$ x=5
$ echo "value of x is " $x
value of x is  5
$
```

The command ***unset*** can be used to remove the variable from shell. For example,

```
$ unset x                    //value of x removed
$ echo “value of x is “ $x
value of x is                // x value not got printed
$
```

More than one variable can be concatenated by placing them together, and without using any operator. For example,

```
$ x=5;y=6
$ z=$x$y
$ echo $z
56
```

Shell may use alternative method for evaluating a variable with the help of curly braces as below –

```
$echo ${x}
5
```

If user wants to concatenate a value of a variable with a string, then the above format is useful. For example –

```
$echo ${x} “is the value of x”
5 is the value of x
```

1.7.2 Shell Scripts

When a group of commands have to be executed to solve a single problem, they are usually kept inside a file. This is called as ***shell script*** or ***shell program.*** Usually, the extension used for these files is ***.sh***, though it is not mandatory to do so. Shell scripts are executed in a separate child shell process, and this sub-shell need not be of the same type as that of your login shell. That is, even if your login shell is Bourne, you can use Korn shell as a sub-shell to run the script. Though, child and parent shells belong to the same type by default, one can provide a special ***interpreter line*** in the first line of the script to specify the type of shell of your script.

Consider the first shell program, with the name **Shell1.sh** as given below. One can use vi editor to create this file.

Shell1.sh

```
#!/bin/sh
#My first shell program
echo "Today's date: `date`"
echo "This month's calendar:"
cal `date "+%m 20%y"`
echo "my shell is: $SHELL"
```

To run this script, make it as executable file using the command –

```
$chmod +x Shell1.sh
```

Then, use the following command to run the script –

```
$./Shell1.sh
Today's date: Tue Jan 23 03:40:59 IST 2007
This month's calendar:
    January 2007
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

my shell is: /bin/bash
```

Few important points about Shell Scripts:

- The lines starting with # symbols are comment lines in a shell script. That is, the shell ignores those lines. When we give the command ./Shell1.sh, then we are

actually requesting a kernel to run an executable file. So, now the kernel checks the file. In the file, if first two characters are # and ! (together known as shebang), then the kernel uses the next string (the string which follows #!) to run the script. Hence, in the above script the string after `#!` is - `/bin/sh` – being used to run the script. The path `/bin/sh` makes the underlying shell (bash, ksh, csh etc) to run the script and to give the output. Note that, the line `#!/bin/sh` is known as interpreter line.

- To run the script, we are using the command `./Shell1.sh` here. The meaning of `.(dot)` is current directory. If the PATH variable consists of the current directory, we can remove `./` and can just use

  ```
  $Shell1.sh
  ```

  to run the script. To include current directory in a PATH variable, we should use the command

  ```
  $ export PATH=$PATH:.
  ```

  As PATH variable in UNIX is a colon separated list, we are using colon (:) here to concatenate existing contents of PATH and the dot (.).

- To run the script, it must be in an executable mode. So, we will change the mode of the file and make it executable. If you directly use the shell to run a shell script, then you don't need a script in executable mode. For example, in the above example, before using chmod command, one can use –

  ```
  $sh Shell1.sh
  ```

  to run this script. Here, the usage of `sh` make the underlying parent shell to spawn a child shell and that child is being used to run the script. Now, the interpreter line is being ignored by shell.

  But, the above type of usage has a disadvantage as well. As we know, there is a slight difference between the syntax of various shell scripts. That is, some of the commands in bash, ksh, csh etc may not be compatible with each other. Now consider a situation – you are writing a script in ksh format and you have given interpreter line accordingly. Assume that the underlying shell in your system is bash. Now, if you directly use `sh` to run a script, you may get unexpected results or sometimes errors as well. Because, you are trying to run a ksh script using bash shell, which is not possible always.

  Hence, it is always better to have an interpreter line, then changing the script to executable mode and then to run a script.

Explanation about the program Shell1.sh:

- The first line in the program (after an interpreter line and a comment line) is –

  ```
  echo "Today's date: `date`"
  ```

  Here, a string within double quotes is being passed to `echo` command. But, observe the term `` `date` `` within that string. The symbol ` (in the keyboard, it is at the left of key 1, and it is along with another symbol tilde ~) is known as

backquote or backticks. The command given within a pair of backquote in UNIX is known as ***command expansion*** (or ***command substitution***). The command expansion will execute the underlying command (that is, the command given within backquotes) and the output is substituted at that place itself.

Hence, in the above line the command `date` is executed and its output is made as a part of the string. Thus, we will get the output as expected.

Some of the shells use the $ symbol and a pair of brackets as an alternative to backquotes for command expansion purpose. For example, one can use the following line to get the same result as shown in the script –

```
echo "Today's date: $(date)"
```

- The third line in the script is -

```
cal `date "+%m 20%y"`
```

In the section 1.6.1, we have seen `date` command and its options like %m and %y. These are given as arguments to `date` command to get month number and last two digits of the year. Then the output is substituted and passed as option to `cal` command to get the calendar for respective month of the year.

- In the last line of the script, a PATH variable $SHELL is used to display the type of shell being used.

1.7.3 read

The ***read*** statement is the internal tool of the shell for taking input from the user. This will help the scripts to become interactive. Consider the following example –

Shell2.sh

```
#!/bin/sh
#Illustration of read statement
echo "Enter your name:"
read fname
echo "Hello $fname"
```

When you run this script, the output would be –

```
Enter your name:
Chetana
Hello Chetana
```

Observe that, when the statement `read fname` is encountered, the script pauses to receive the user input. After getting the input, it continues with the next line in the script.

The `read` statement can be used to read multiple variables in a single line as below –

```
read fname lname
```

Here `fname` and `lname` are different variables. The input has to be given with a space between two values.

Note that, if the numbers of inputs are less than the number of variables, then the leftover variables will not be assigned any value. Whereas, if number of inputs are more than the number of variables, then the extra inputs are concatenated and assigned to the last variable. Consider following examples –

Ex1:

```
read fname lname          #read two variables
chetana                   #give only one input
echo "$fname  $lname"     #try to display both variables
chetana                   #displays first variable, second is empty
```

Ex2:

```
read fname                #read one variable
chetana hegde             #give two inputs
echo "$fname"             #display one variable
chetana hegde             #displays both values
```

1.7.4 Command Line Arguments

Arguments can be passed to a shell script through the command line, while running the script. Such command line arguments are assigned to special variables known as ***positional parameters.*** The name of the program itself is treated as first argument and stored in positional parameter $0. Further arguments given by the user are sequentially stored in parameters $1, $2 and so on. Note that, these are not called as shell variables (because, name of shell variables starts with a character). Some of the special parameters used by the shell are listed in Table 1.1.

Table 1.1 Special parameters used by Shell

Shell Parameter	Significance
$0	Name of the executed command (that is script file name – in most of the cases)
$1, $2 etc.	Positional Parameters representing command line arguments
$#	Total number of arguments specified in command line (excluding file name)
$*	Complete set of positional parameters as a single string
"$@"	Each quoted string treated as a separate argument
$?	Exit status of last command
$$	PID of the current shell
$!	PID of the last background job

Consider a shell script Shell3.sh as below –

Shell3.sh

```
#!/bin/sh
#Illustration of Command line arguments
echo "Your file name is $0"
echo “Argument one: $1”
echo “Argument two: $2”
echo “you have totally $# arguments as: $*”
```

Output:

```
$sh Shell3.sh Hello World
Your file name is Shell3.sh
Argument one: Hello
Argument two: World
you have totally 2 arguments as: Hello World
```

Here, along with script name Shell3.sh, two arguments Hello and World have been passed and they will be stored in $1 and $2 respectively. The parameter $# is used to display number of arguments and $* is used to display all arguments as a single string.

1.7.5 Exit Status of a Command

Whenever a shell command gets executed, its execution status is stored in a special parameter $?. If the command is executed successfully, then 0 will be stored (indicating *true*). Otherwise, some non-zero number (indicating *false)* is stored in $?. It is called as ***exist status*** of a command. This parameter always contains the exit status of the last command which has been executed. Consider few examples:

Ex1: Successful Execution

```
$echo “Date: `date`” ; echo “Exit Status: $?”
Date: Wed Oct 18 21:25:29 IST 2017
Exit Status: 0
```

Here, the `date` command is executed within `echo`. As, it could execute successfully, the exit status parameter will be obviously 0.

Ex2: File not found error

```
$cat test; echo “Exit Status: $?”
cat: test: No such file or directory
Exit Status: 1
```

Here, we are trying to open a file test, which actually does not exist. So, the appropriate error message is displayed. Hence, the exit status parameter will now contain 1.

Ex3: Command not found error

```
$try ; echo “Exit status: $?”
```

```
bash: try: command not found
Exit status: 127
```

Here, we are giving some random text *try* as if it is a command. Hence, no such command exists, there will be a *command not found* error. Now the exit status is 127. Note that the number 127 indicates – either the command not found; or the command is found, but the library required to run that command is not found in the specified path.

NOTE: Knowing the exit status of a command is very essential in real time programming. Because, one may need to take separate action based on successful/unsuccessful execution of a particular command.

1.7.6 Relational Operators in Shell Scripts

In shell scripts, when we need to compare two numeric values we use some special operators as listed in Table1.2.

Table 1.2 Relational Operators

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

Example Syntax:

```
operand1 -eq operand2
```

Note that these operators will work only on integers. The numbers with decimal points will be simply truncated.

1.7.7 Numeric Comparison using *test* and [ ]

In shell scripts, to evaluate an expression (especially in *if* conditional construct) we use ***test*** command. Generally, *test* command will use numeric comparison operators to evaluate the condition. It returns either *true* or *false* exit status. Hence the result of *test* command can be known using the shell parameter $?, which stores the exit status of previous command. The uses of *test* command are:

- Comparing two numbers
- Comparing two strings
- Comparing single string with a null value
- Checking file's attributes

Here, we will discuss comparison of two numbers. Consider the following example –

```
$ x=5; y=7
$ test $x –eq $y
$ echo $?
1              #Not equal

$ test $x –lt $y; echo $?
0              #5 less than 7 is true
```

Observe that when x and y are compared for checking equality, the result is 1 – a non-zero number indicating false; whereas, x is less than y and hence the result is true (indicated by zero).

A pair of square brackets [ ] also does the similar job that of ***test*** for evaluating numbers. For example,

```
$ [ 10 –lt 20 ]
$echo $?
0         #10 less than 20 is true
```

NOTE that, there must be a space between a square bracket and the operand. Otherwise, shell will throw an error.

1.7.8 The *if* Condition

One of the important requirements in programming is conditional structures. In shell programming, the conditional construct ***if*** can be used in the following ways –

```
if command is successful
then
        execute commands
fi
```

Format 1

```
if command is successful
then
        execute commands
else
        execute commands
fi
```

Format 2

```
if command is successful
then
        execute commands
elif command is successful
then
        execute commands
elif command is successful
then
        ……..
else
        ……..
fi
```

Format 3

Consider the following example to illustrate *if* statement.

```
#!/bin/sh
#Illustration of if statement

x=5
y=10

if test $x -lt $y
then
  echo "$x is less than $y"
else
  echo "$y is less than $x"
fi

if [ $x -ne $y ]; then
  echo "$x and $y are not equal"
fi
```

Output:

```
5 is less than 10
5 and 10 are not equal
```

1.7.9 The Logical Operators && and ||

The shell script allows the user to use two logical operators && and || for combining more than one conditions/commands. Syntax:

- **cmd1 && cmd2**: The cmd2 is executed only if cmd1 is true. If both cmd1 and cmd2 executed successfully, then the exit status would contain zero, otherwise some non-zero number. For example,

  Ex1:

  ```
  $ a=10
  $ b=10
  $ test $a -eq $b && echo "equal"
  equal
  $ echo $?
  0
  ```

  Here the values of a and b are equal. Hence, the second command `echo "equal"` got executed. The exit status is found to be zero indicating successful execution of both the commands.

  Ex2.

  ```
  $ a=10
  $ b=20
  $ test $a -eq $b && echo "not equal"
  $ echo $?
  1
  ```

  Observe that value of a and b are different. Hence, the first command

  ```
  test $a -eq $b
  ```

itself is false. So, the second command will not be executed. And, the exit status is found to be 1 indicating false.

- **cmd1 || cmd2**: If any one of cmd1 and cmd2 executes successfully, then the exit status will contain zero.
  Ex1:

```
$ a=10
$ b=20
$ test $a -eq $b && echo "equal"
equal
$ echo $?
0
```

  Here the values of a and b are not same. Still, we are checking the equality. In fact, the command `test $a -eq $b` results in false. Still, because of OR (||) operator, the second command `echo "equal"` gets executed.

1.7.10 case

The ***case*** statement in shell is similar to switch – statement in C language. When there is a possibility that the statement matches an expression for more than one alternative, we use *case* statement. It is useful for multi-way branching. The general syntax is –

```
case expression in
     pattern1) commands1 ;;
     pattern2) commands2 ;;
     pattern3) commands3 ;;
     …….
esac
```

The *case* matches *expression* with *pattern1*. If they match, then *commands1* will be executed. If the match fails, then *pattern2* is compared and so on. Here *commands1* etc may be one or more commands. Each command list is terminated by a pair of semicolons (similar to *break* in C language). And, entire *case* construct is ended with *esac* (the reverse of *case*).

Consider the following shell script :

menu.sh

```
#!/bin/sh
#Menu driven script to illustrate case

echo –e "        MENU  \n
1. List of files \n   2. Display date \n 3. Users of
system 4. Quit \n Enter your option: \c"

read choice

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

Output:

```
        MENU

 1. List of files
 2. Display date
 3. Users of system
 4. Quit
Enter your option: 2
Tue Jan 30 09:59:46 IST 2007
```

In the above example, the multi-line *echo* command uses *–e* option to enable escape sequences like \n. The script displays various options as shown in the output. Based on the user's choice, the respective option is matched using *case* statement to execute respective block of code. The last pattern in the *case* is *), which is used only when all previous options fails (similar to ***default*** keyword in ***switch-case*** of C language). Note that a pair of semicolons is not necessary for this last option.

The ***case*** can be used in following ways as well:

- **Matching multiple patterns:** More than one pattern can be combined to perform a specific action using ***case***. For example, consider the following code snippet:

```
echo "Do you want to continue? (y/n):"
read ans
case "$ans" in
     y|Y)  ……..  ;;  #do something
     n|N) exit;;
esac
```

Here, when user would like to continue, he/she may press y or Y (lower case or upper case letter). So, the | symbol is used to combine both the patterns. In the same way, n or N is used.

- **Using wild-cards with *case*:** String matching features using wild-cards is possible with ***case*** in shell script. The metacharacters like * and ? etc can be used to match strings. For example,

```
echo "Do you want to continue? (yes/no):"
read ans
case "$ans" in
     [yY][eE]*)  ……  ;; #do something
     [nN][oO]) exit;;
esac
```

Here, user can give the input for `ans` in any combination of uppercase and lowercase letters. That is, `ans` can be one among: *Yes, yes, yEs, YEs* etc. Similarly, a 'no' can be given as *No, nO, NO, no*.

1.7.11 expr

The shell does not have computing features. For the evaluation of expressions, it has to use the ***expr*** command. It can perform arithmetic operations on integers and manipulate strings up to some extent. As it is not a shell statement, instead a stand – alone UNIX command, its usage in shell script will slowdown the operation. It can perform the basic arithmetic operations – addition, subtraction, multiplication, division and modulus (remainder after division). Consider the following example –

```
$ x=3 y=5
$ expr $x + $y      #note the space before and after +
8

$ expr $x - $y
-2

$ expr $x \* $y     #the *  used with \ as escape sequence
15

$ expr $y / $x
1                   # only integer part will be displayed

$expr 18 % 4
2                   # remainder after dividing 18 by 4
```

NOTE:

- While using ***expr*****,** the operands +, - etc. have to be enclosed with white space on both sides.

- For doing multiplication, the operator * has to be escaped to prevent the shell from interpreting it as a metacharacter.
- To assign the evaluated value of an expression to another variable, the back quotes must be used as command substitution. For example,

```
$ x= 4 y=6
$ z = `expr $x + $y`
$ echo $z
10
```

1.7.12 sleep And wait

Sometimes in shell scripts, we need to introduce some delay to let the user to see some message before doing anything else, or to check the regular time intervals for an event to occur etc. In such situations, a UNIX command ***sleep*** can be used along with the argument indicating number of seconds to sleep or pause. After these many seconds gets elapsed, the shell resumes its execution. For example,

```
$ sleep 100;  echo "100 seconds have elapsed"
100 seconds have elapsed
```

The output of *echo* command will be displayed after 100 seconds. That is, we need to wait for 100 seconds to see the output. The ***sleep*** command doesn't incur much overhead while it is sleeping.

A shell build-in command ***wait*** is used to check whether all background processes have been completed. This is useful when we have to run a job in the background and now want to make sure whether it is completed before starting another job. If one needs to wait till a particular background process to be processed, then this command is used with an argument indicating the required process id (PID). The syntax is –

```
$ wait       #waits for completion of all background processes
$ wait 128   #waits for completion of process having PID 128
```

1.7.13 while

Shell scripts support three types of looping structures – ***while***, ***until*** and ***for***. Looping structures are used in any program when a set of instructions have to be executed repeatedly for few times based on the condition.

The ***while*** statement performs a set of instructions till the control command returns a true exit status. It is similar to ***while*** loop of C language. The syntax is –

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

Note the keywords ***do*** and ***done*** here. The set of instructions enclosed within ***do*** and ***done*** are executed as long as the *condition* remains true. The *condition* may be any UNIX command or an expression involving ***test, expr*** etc.

Consider the shell script **menu.sh** used earlier (section 1.7.10) and the modified version is given here –

```
#!/bin/sh
#Illustration of while loop

ans='y'
while [ "$ans" = "y" ]
do
     echo -e "         MENU   \n
       1. List of files \n 2. Display date \n 3. Users of system
       4. Quit \n Enter your option:\c"

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

      echo -e "Do you want to continue (y/n)?: \c"
      read ans
done
```

Output:

```
      MENU
1. List of files
2. Display date
3. Users of system
4. Quit
Enter your option: 2
Tue Jan 30 09:59:46 IST 2007
Do you want to continue (y/n)? : y

      MENU
1. List of files
2. Display date
3. Users of system
4. Quit
Enter your option: 3
chetana   pts/1          Sept 04 11:11
Do you want to continue (y/n)? : n
```

In the above example, the menu options are repeatedly shown to the user until he/she chooses to quit. So, in a single run of the script, one can choose multiple menu options one after the other.

1.7.14 until

The ***until*** statement is a complement of ***while*** loop. Here, the block of statement is executed until the condition remains false. For example, consider the following shell script –

```
#!/bin/sh
#Illustration of until loop

a=0
until [ $a –gt 5 ]
do
      echo $a
      a=`expr $a + 1`
done
```

Output:

0
1
2
3
4
5

In the above script, the variable *a* is initialized to zero. Then the condition is checked whether *a* is greater than 5. Since the condition is false, the statement inside the until-loop starts getting executed. Here, the value of *a* is being printed and then incremented by 1. When value of *a* is 6, then the condition '6 greater than 5' becomes true. And now, the loop gets terminated.

1.7.15 for

It is very important to note that (especially those who know higher programming languages) the ***for*** loop in shell script is NOT same as that in other languages. One can neither increment/decrement the values, nor specify the condition to be met. Instead, it just iterates over the elements in a list. A set of commands are executed until the list gets exhausted.

The syntax is –

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

Consider an example –

```
$ for x in 1 5 10 4           # list has 4 strings
> do
>    echo "Value of x is $x"  # executed 4 times
> done
Value of x is 1
Value of x is 5
Value of x is 10
Value of x is 4
```

Here, the list contains four strings 1, 5, 10 and 4 (note that, they are treated as strings, not as integers). Each item in the *list* is assigned to *variable* (here x) and *echo* command is executed.

Ex: Write a shell script to find sum of numbers provided through command line.

sum.sh

```
#!/bin/sh
#sum of numbers given through command line

sum=0

for x in $*
do
     sum=`expr $sum + $x`
done

echo "Sum = $sum"
```

Output:

While running this script, give command line arguments similar to –

$ sh sum.sh 10 20 30
Sum=60

1.7.16 "$@"

We have observed earlier that $* is used to list out all the arguments given via command line. In the script ***sum.sh*** given in previous section, it is observed that a string separated by space is treated as a different argument by $*. But, sometimes we need text within double quotes separated by space to be treated as single string. In such situations, $@ is used. The $@ will treat the text given in double quotes as one single string. To do that, $@ itself has to be enclosed in double quotes as "$@".

Consider the script given below –

cmdArg.sh

```
#!/bin/sh
#illustration of $@

echo "The command line arguments are:"
for x in "$@"
do
        echo $x
done
```

Output:
While running this script, give command line arguments similar to –

```
$sh cmdArg.sh "MCA01 Chetana" "MCA02 Raghu" "MCA03 Rajatha"
The command line arguments are:
MCA01 Chetana
MCA02 Raghu
MCA03 Rajatha
```

Observe that, the arguments are given within double quotes. And each argument contains string involving space in between. If $* would have used in the above script, the shell would treat `MCA01` as separate string, `Chetana` as another string and so on.

1.7.17 set And shift: Manipulating the Positional Parameters

To extract a field from single line output statements, we have filters like *grep*, *head* etc. (will be discussed in later chapters). But, there is an internal command in UNIX viz. ***set.*** This command assigns its arguments to positional parameters like $1, $2 etc. Also, it assigns proper values to $* and $# as well. For example,

```
$ set 13 45 32
$ echo “Argument one:$1, Argument 2:$2, Argument 3:$3”
Argument one:13, Argument 2:45, Argument 3:32
$ echo “There are $# number of arguments”
There are 3 number of arguments
```

Now, this feature of ***set*** command can be used in commands like ***date.*** For example,

```
$set `date`
$echo $*
Sun Oct 29 08:57:52 IST 2017
$echo $1
Sun
$echo $3
29
$ echo “Today’s date is $2 $3, $6”
Today’s date is Oct 29, 2017
```

Shifting Arguments towards left: The *shift* command

We have seen that in many of the scripts, the first argument ($0) is a filename. If we don't want this to happen, we can shift the argument towards left using ***shift*** command. This will transfer the contents of a positional parameter to its immediate lower numbered positional parameter. When ***shift*** is called once, $2 will become $1, $3 will become $2 and son on. For example,

```
$set `date`
$echo $1 $2 $3
Sun Oct 29
$shift
$ echo $1 $2 $3
Oct 29 08:57:52
$shift 2            #shifting two positions at a time
$ echo $1 $2 $3
08:57:52 IST 2017
```

1.8 REDIRECTION

Usually UNIX commands generate some output or error message on to a terminal, and require input from the terminal. In the context of redirection, the word terminal indicates the screen, display or keyboard. The activities related to terminal are done through 3 files viz.

- Standard Input – The file (or stream) representing input, which is connected to keyboard
- Standard Output – The file (or stream) representing output, which is connected to display
- Standard Error – the file (or stream) representing error messages that originate from the command or shell. This is also connected to the display.

These special files are actually streams of characters which many commands see as input and output. A stream is a sequence of bytes. When a user logs in, the shell makes these three files available representing three streams. The files are closed when the command completes execution. Even though the shell associates each of these files with a default physical device, this association is not permanent. The shell can easily unhook a stream from its default device and connect it to a disk file the moment it sees some special characters in the command line. The user has to instruct the shell to do what by using symbols like > and < in the command line.

These three files are discussed in detail in the following sections.

1.8.1 Standard Input

The standard input file represents three input sources viz.

- The keyboard (default source)
- A file using *redirection* with the < symbol
- Another program using a *pipeline*

The commands like ***cat*** and ***wc,*** when used without any special symbols like < and | in the command line, takes the input from the default source. For example –

```
$ wc
```

```
Standard input can be redirected
It can come from a file
Or a pipeline
[ctrl – d]
     3    14   71
```

Here, when we ***wc*** command and press enter, it waits for keyboard input as no filename is provided. We have given some 3 lines of text and then pressed *ctrl-d*. Then it displays the output (3 lines 14 words and 71 characters). In fact, the ***wc*** command is reading from standard input file here.

The shell can reassign or ***redirect*** the standard input file to a disk file using < symbol as shown –

```
$ wc < sample.txt   #existing file containing above 3 lines
     3    14   71
```

Here, the ***wc*** command reads the *sample.txt* and gives the input. The working of this command is explained here –

1. On seeing the < symbol, the shell opens the disk file *sample.txt* for reading
2. Shell unplugs the standard input file from its default source and assigns it to *sample.txt*
3. **wc** reads from standard input which has earlier been reassigned by the shell to *sample.txt*

Note that, ***wc*** has no idea where the stream came from, and also it is not even aware that the shell had to open the file *sample.txt* on its behalf.

NOTE:

1. When < is used, the shell is opening a file and the command is not aware of that. But, when filename is used directly without < symbol, then the command itself will open a file and does the job.
2. In the above discussion, the example of ***wc*** command is taken. But, same explanation holds good for other commands like ***cat*** as well.
3. It is possible to take input from both standard input as well as file. While taking input from multiple sources, the – symbol has to be used to indicate the sequence of taking input. For example,

   ```
   $cat – test
   ```

   The above command takes input from standard input and then from the file *test.*

   ```
   $cat test – test1
   ```

   This line takes input from file *test*, and then from standard input and finally from the file *test1.*

1.8.2 Standard Output

All the commands displaying the output on the terminal actually write to the ***standard output*** file as a stream of characters, but not directly to the terminal. There are three possible destinations for the output stream –

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

The default destination (terminal or the monitor display) of standard output can be redirected to any file by using >. For example,

```
$wc  sample.txt > newfile
$cat newfile
     3    14   71  sample.txt
```

Here, the ***wc*** command operates on *sample.txt* to count number of lines, words and characters. But, instead of displaying the result on the terminal, output is redirected to another file *newfile*. Then use *cat* command to view the contents of *newfile*, which is nothing but the result of *wc* command.

In the similar lines, one can use >> symbol to append the output of one command at the end of existing contents of any file.

The working of the command `wc  sample.txt > newfile` is as given here –

1. The shell opens the disk file *newfile* for writing once it sees > symbol.
2. It unplugs the standard output file from its default destination and assigns it to *newfile.*
3. Now, the command ***wc*** (not the shell) opens the file *sample.txt* for reading.
4. Then, ***wc*** writes to standard output which has earlier been reassigned by the shell to *newfile*.

And, all these procedure happens without ***wc*** knowing that it is writing to *newfile*.

The redirection is useful feature when output of many commands or files has to be concatenated.

1.8.3 Standard Error

Before understanding the concept of standard error, we should understand the concept of ***file descriptor.*** A file descriptor is a number attached to each of the three standard files. A file is opened by refereeing to its pathname, but subsequent read and write operations identify the file by this file descriptor. The kernel maintains a table of file descriptors for every process running in the system. The first 3 slots are generally allocated to three standard streams as –

0 – standard input
1 – standard output
2 – standard error

These descriptors are implicitly prefixed with the redirection symbols. That is –

> and 1> both are same
< and 0< both are same

If our program opens a file now, that file may be allocated with the file descriptor as 3.

One need to use these file descriptors while handling standard error stream. Consider the following example, where we are trying to open a non-existing file –

```
$cat foo
cat: cannot open foo
```

Here, the error message has been displayed on the terminal (monitor display). We may want to redirect this error message into a file. But, ordinary redirection symbol > cannot do this job. That is,

```
$cat foo > errFile
cat: cannot open foo
```

As we can observe, the error message is displayed on the terminal, but not being redirected to *errFile*. Though standard output and standard error both use the terminal as a default destination, the shell treats them differently. So, to redirect the standard error, we need to use the file descriptor as –

```
$cat foo 2> errFile
$cat errFile
cat: cannot open foo
```

One can append the error file using 2>>. Also, both output and error streams can be redirected separately in a single line as –

```
$ test.sh > out.txt  2>err.txt
```

Here, the output of shell script *test.sh* into the file *out.txt* and any possible error messages of the script into *err.txt*.

1.9 trap: INTERRUPTING A PROGRAM

When we press *ctr+c* or *break* in the terminal during the execution of any shell program, the shell script will be terminated. But, by doing so, some of the temporary files remain in stack and they will not be cleaned up. Hence, UNIX gives an option to terminate the program using ***trap*** command with appropriate signals. Syntax is –

```
trap command_list signal_list
```

Here, *command_list* can be any valid UNIX command or user-defined function, *signal_list* is a list of any number of signals you want to trap.

Some of the common signals you want to encounter in your program may be as below –

Signal Name	Signal Number	Description
SIGHUP	1	Hang up detected on controlling terminal or death of controlling process
SIGINT	2	Issued if the user sends an interrupt signal (Ctrl + C)
SIGQUIT	3	Issued if the user sends a quit signal (Ctrl + D)
SIGFPE	8	Issued if an illegal mathematical operation is attempted
SIGKILL	9	If a process gets this signal it must quit immediately and will not perform any clean-up operations
SIGALRM	14	Alarm clock signal (used for timers)
SIGTERM	15	Software termination signal (sent by kill by default)

There are two common uses for trap in shell scripts −

- Clean up temporary files
- Ignore signals

If the command listed for trap is null, the specified signal will be ignored when received. For example, the command −

```
$ trap '' 2
```

This specifies that the interrupt signal is to be ignored. You might want to ignore certain signals when performing an operation that you don't want to be interrupted. You can specify multiple signals to be ignored as follows −

```
$ trap ' '1 2 3 15
```

1.10 THE *here document*

Sometimes, the shell uses the << symbols to read data from the same file containing the script. This is referred to as here document, indicating that the data is here only, not in a separate file. Any command using standard input can also take input from a here document.

Consider an interactive script, that is, a shell script which reads some input from the keyboard.

hereDoc.sh

```
#!/bin/sh
#Illustration of here document

echo "Enter your name:"
read fname
echo "Enter your age:"
read age

echo “Your name is $fname, Your age is $age”
```

When we run this script in a normal way, it would look something like this –

```
$ sh hereDoc.sh
Enter your name:
Ramu
Enter your age:
21
Your name is Ramu, Your age is 21
```

Now, let us see, how to use ***here document*** for this script. Run the above script as shown below –

```
$ sh hereDoc.sh <<END
>Ramu          #shell waits for your input from this line
>21
>END                # till this line
Enter your name:
Enter your age:
Your name is Ramu, Your age is 21
```

Observe the above lines. While running the script, we have used a term <<END. Here, the symbol << indicates that the file *hereDoc.sh* will be reading an input from the ***here document*** but not from the keyboard. The word *END* used is just an example for delimiter, and one can use any word (not UNIX command). After the first line, user can keep giving the inputs. Once the input is done, the delimiter has to be provided. Immediately after seeing the delimiter word for the second time, the *hereDoc.sh* file starts executing and the *read* commands inside the script will not wait for the user input from the keyboard, instead, it will be taken from the ***here document*** created already.