UNIX Shell Scripting

Focus Training Services

Compiled and Edited By

Rahul Mulay

Version 4.0

This page is intentionally left blank.

Preface

I am pleased to hand you this book on Unix Shell Scripting. Before you dive into the contents of the book, I would like to share with you some thoughts that steered the choice of contents and structure of this book.

The primary aim of the book is to be an aid to the students of Unix Shell scripting class. The book should reduce the burden of taking 'notes' during class so that the students can concentrate more on the proceedings of the class and spend more time on practicing the scripting. Towards the end of the book more than 30 scripts are included as examples. These should help you gaining an insight how the Shell Scripting knowledge can be applied to various real-life situations like system administrator, database administrator or even just to automate your own recurring tasks.

The book is structured such that in the first few chapters it talks about what I call building blocks for a good script. It talks about commands, variables, redirection, vi editor, conditions and loops. These are essential parts of any language. When you know these and add some programming skills to it, then only sky is the limit to a creative script. Being a multi-user system, it is essential to know about file permissions and access control. From here on, then there are chapters which talk about 'find' command, which is quite essential to find files on the multi-user system where graphical interface may not be available most of the times. Command Line Arguments is another essential topic for making the script flexible and more versatile. Cron is essential for scheduling a script whereas ssh-keygen would allow you to make an entry in some other machine without being asked password. Use of Shebang is something good to know. Functions are essentials for making repeated use of same piece of code, and similar is essential knowing what is meant by an exit status of a command (and how to make use of it). Sending a mail from command line (or from within a script) can be a great way to convey status of a process to those who are eager to listen! Using the building blocks, these tools, fillers and everything, you can build a wall, a room, a bungalow or even a housing society. The more you practice, better script you can write.

If you happen to be using soft copy of this book, do not copy-paste the code from it. The quotes, back-ticks and hyphen do not work cosrrectly in the copy-pasted script. Always type the script. That not only removes above problem, but forces you to pay attention to each word in it.

One essential point to make – Unix and Linux are very similar, in their look-and-feel as well as the commands and constructs (conditions and loops) and the way they behave. However, you should always keep in mind that the things keep evolving and the scripts executing successfully in classroom may throw an error on other machine. It is no fault of the script. It may be that the machine you used in the classroom may have different shell or different version of the shell, may have different version of the command, or may have different version of the operating system itself. For your reference, below are the details of versions of operating system and important commands on which the scripts in this book have been tested successfully.

Linux 2.6.32-71.el6.x86_64 #1 SMP Wed Sep 1 01:33:01 EDT 2010 x86_64 x86_64 x86_64 GNU/Linux

vsftpd: version 2.2.2

mail: 12.4 7/29/08

bash: GNU bash, version 4.1.2(1)-release (x86_64-redhat-linux-gnu)

csh: tcsh 6.17.00 (Astron) 2009-07-10 (x86_64-unknown-linux) options wide,nls,dl,al,kan,rh,color,filec

sed: GNU sed version 4.2.1

awk: GNU Awk 3.1.7

Acknowledgements

I thank Mr. Shekhar Tulshibagwale for being the motivation behind writing this book and for being the first person to come up with suggestions for improvements in the first draft. I thank Mr. Mahesh Chadare who created the very first version of this book. I also thank Mr. Omkar Ghotavadekar for helping me with the Oracle and MySQL related scripts. And I thank Ms. Prerana Kshirsagar for reviewing and testing the scripts written for the book.

About the Author

The author enjoys teaching Shell Scripting. He has been part of IT industry since 2001 and worked with Patni Computer Systems (now iGate) for one and half year and with Infosys Limited for over ten years. He has been working on Java projects most of the time, from being a developer to project manager. He is certified Java Programmer (Oracle Certified Java Programmer JSE6) and certified System Administrator (Redhat Certified System Administrator). Unix being a de-facto operating system for all the production environments, knowing shell scripting has been essential part of his job. He currently works as trainer on Shell Scripting, SQL, Core Java, Servlet/JSP, Struts, Hibernate and Web Services.

He is an Engineering graduate from Pune University and has done his post graduation Diploma in Advanced Computing (C-DAC).

Table of Contents

1 Introduction

1.1 Overview

As we get started, students often ask "What is this Unix thing?"

In most general terms, Unix (pronounced "yoo-niks") is an Operating System. An Operating System is a control program (or manager) for some type of hardware, often (but not always) computer hardware.

1.2 Why Unix?

Unix is the most widely used computer Operating System (OS) in the world. Unix has been ported to run on a wide range of computers, from handheld personal digital assistants (PDAs) to inexpensive home computing systems to some of the worlds' largest super-computers. Unix is a multiuser, multitasking operating system which enables many people to run many programs on a single computer at the same time. After more than three decades of use, Unix is still regarded as one of the most powerful, versatile, flexible and (perhaps most importantly) **reliable** operating systems in the world of computing.

The UNIX operating system was designed to let a number of programmers access the computer at the same time and share its resources, like RAM, the Harddisk and Processor.

The operating system controls all of the commands from all of the keyboards and all of the data being generated, and permits each user to believe he or she is the only person working on the computer.
This real-time sharing of resources makes UNIX one of the most powerful operating systems ever.
Although UNIX was developed by programmers for programmers, it provides an environment so powerful and flexible that it is extensively used in businesses, sciences, academia, and industry.
Many telecommunication switches and transmission systems also are controlled by administration and maintenance systems based on UNIX.
While initially designed for medium-sized minicomputers, the operating system was soon moved to larger, more powerful mainframe computers.
As personal computers grew in popularity, versions of UNIX found their way into these boxes, and a number of companies produce UNIX-based machines for the scientific and programming communities.

1.3 Features of UNIX

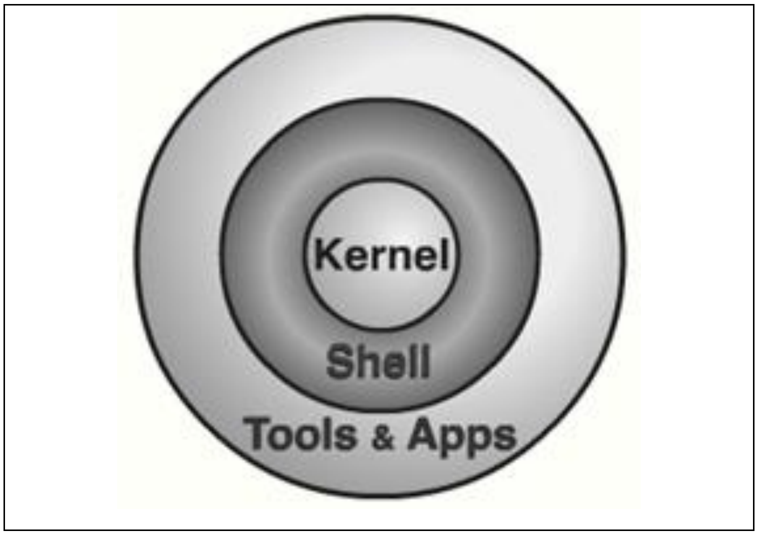
The Unix Operating System has a number of features that account for its flexibility, stability, power, robustness and success. Some of these features include:

- Unix is a multi-user, multi-tasking Operating System, which allows multiple users to access and share resources simultaneously (i.e. concurrently).
- The Unix OS is written in a modern, high-level programming language, specifically the C programming language. This makes it easy for programmers to read and modify the Unix source code, and more importantly, port this source code to other types of hardware. This accounts for its presence on a wide range of computer hardware and other devices. Prior to being written in C, the Unix OS was written in assembly language, as were most if not all, Computer Operating Systems of the time.
- Unix hides the details of the low-level machine architecture from the user, making application programs easier to port to other hardware.
- Unix provides a simple, but powerful command line User Interface (UI).
- The user interface provides primitive commands that can be combined to make larger and more complex programs from smaller programs.
- Unix implementations provide a hierarchical file system, which allows for effective and efficient implementation while providing a solid, logical file representation for the user.
- Unix provides a consistent format for files, i.e. the byte stream, which aids in the implementation of application programs. This also provides a consistent interface for peripheral devices.

1.4 How UNIX is organized

The UNIX system is functionally organized at three levels:

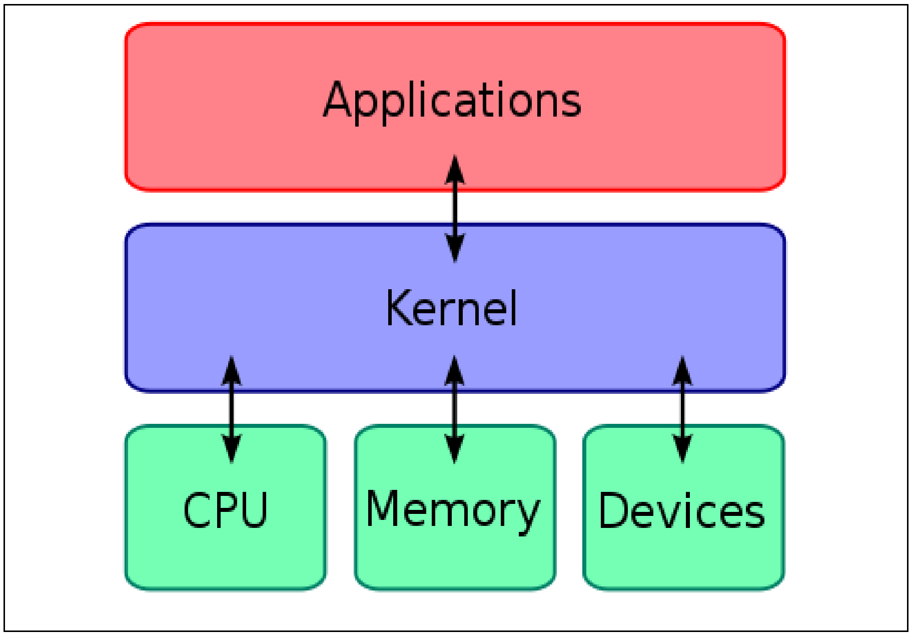
- The kernel, which schedules tasks and manages storage
- The shell which interprets users' commands, calls programs from memory, and executes them.
- The tools and applications that offer additional functionality to the operating system



1.4.1 The kernel

Kernel is heart of Unix OS.

- It manages resource of Unix OS. Resources means facilities available in Unix. E.g. Facility to store data, print data on printer, memory, file system etc.



- Kernel decides who will use this resource, for how long and when. It runs your programs (or executes binary files).
- The kernel acts as an intermediary between the computer hardware and various programs/applications/shell.
- The kernel controls the hardware and turns part of the system on or off at the programmer's command. If we ask the computer to list (**ls**) all the files in a directory, the kernel tells the computer to read all the files in that directory from the disk and display them on our screen.

1.4.2 The Shell

- Computers understand the language of 0's and 1's called binary language. In early days of computing, instructions were provided using binary language, which is difficult for all of us to read and write. So in Unix there is special program called Shell. Shell accepts your instruction or commands in English (mostly) and if it's a valid command, it is passed to kernel.

- Shell is a user program or it's an environment provided for user interaction. Shell is an interpreter that executes commands read from the standard input device (keyboard) or from a file.
- Shell is not part of system kernel, but uses the system kernel for various tasks like executeing programs and creating files etc.
- Every user gets a default shell when he logs in. A Unix can have many shells available in it and the System Administrator can decide which shell to be assigned to a user as his default shell. On Redhat Linux systems, there is bash shell by default.

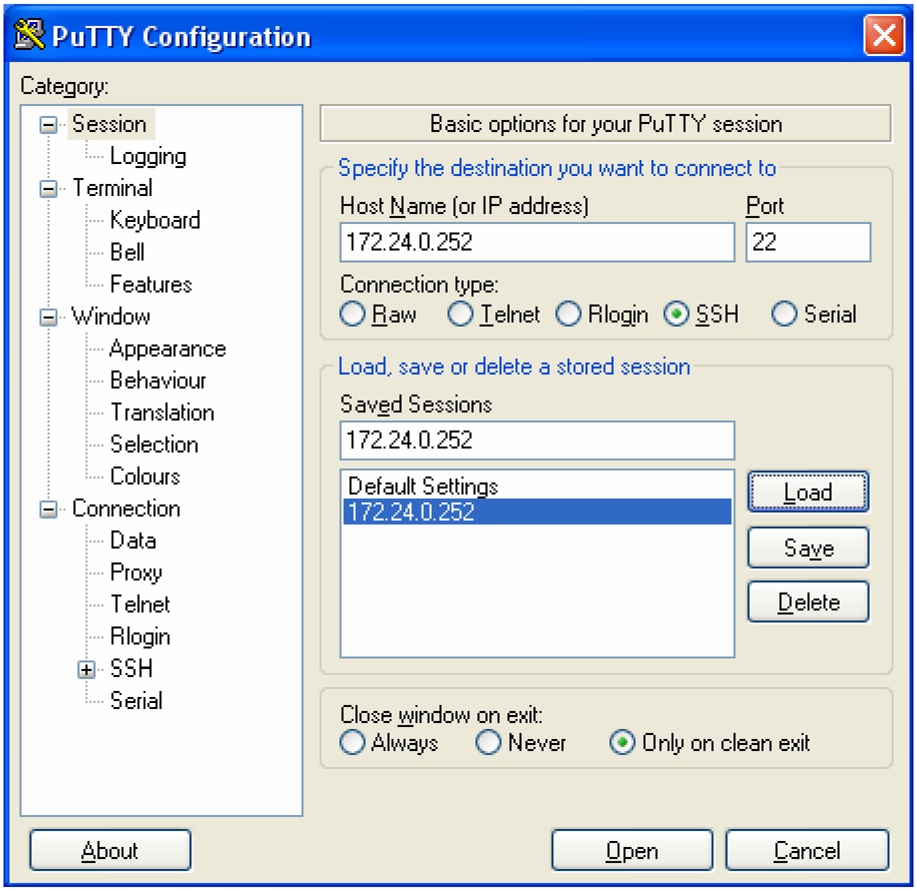
2 Unix Commands

2.1 *How to login*

Before we enter any command on the command prompt, we need to connect to and loging into some Unix machine. We need to have a login (userid and password) on that machine and we should have a software to connect to that machine.

2.1.1 putty

You can logon from Windows machine to a UNIX server using software like putty.



PuTTY Configuration

Basic options for your PuTTY session

Specify the destination you want to connect to

Host Name (or IP address): 172.24.0.252

Port: 22

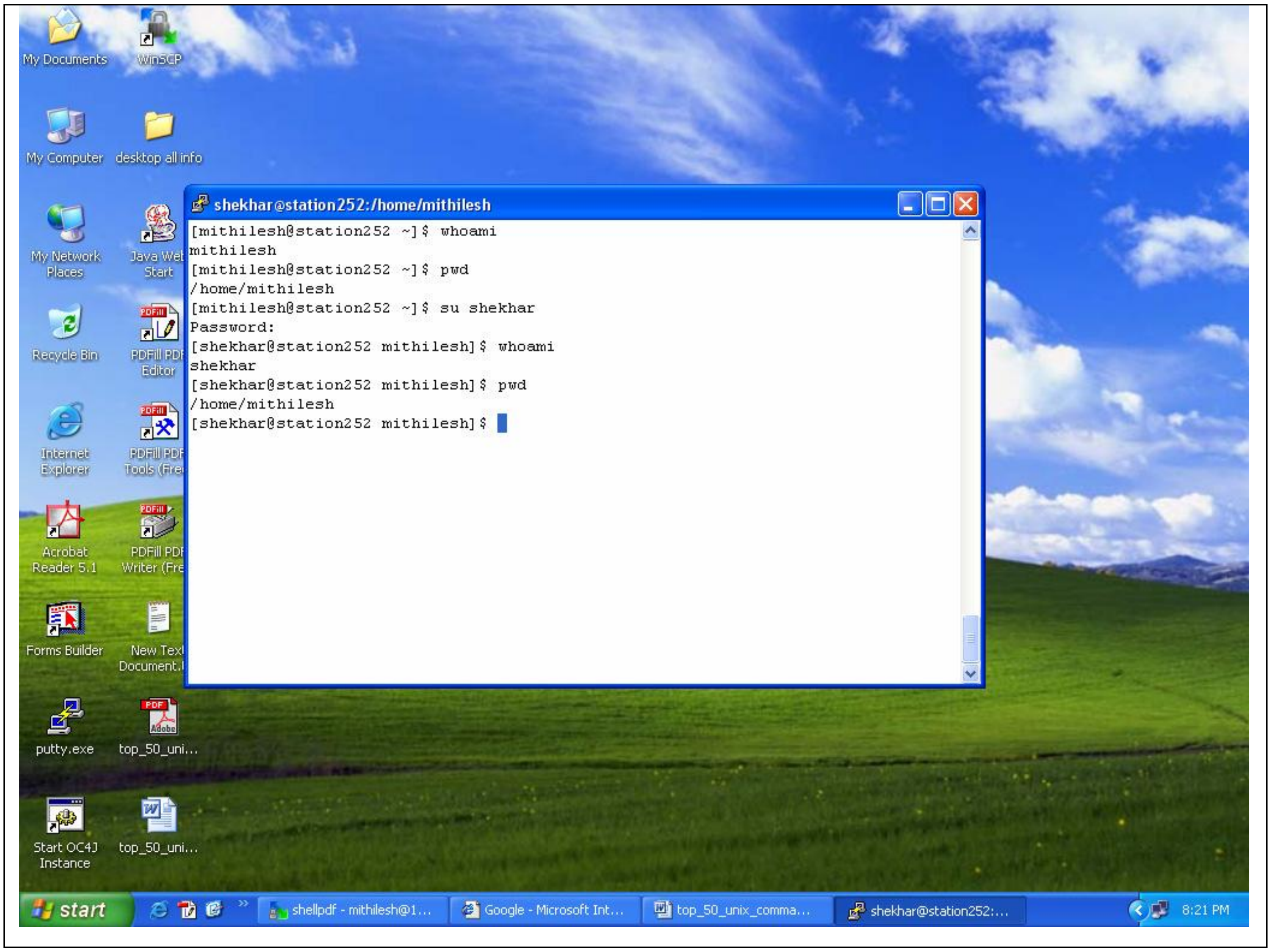
Connection type: ○ Raw ○ Telnet ○ Rlogin ● SSH ○ Serial

Load, save or delete a stored session

Saved Sessions: 172.24.0.252

Default Settings
172.24.0.252

Close window on exit: ○ Always ○ Never ● Only on clean exit



```
[mithilesh@station252 ~]$ whoami
mithilesh
[mithilesh@station252 ~]$ pwd
/home/mithilesh
[mithilesh@station252 ~]$ su shekhar
Password:
[shekhar@station252 mithilesh]$ whoami
shekhar
[shekhar@station252 mithilesh]$ pwd
/home/mithilesh
[shekhar@station252 mithilesh]$
```

2.1.2 ssh

ssh allows users of Unix workstations to secure their terminal and file transfer connections. This page shows the straight forward ways to make these secure connections. ssh provides the functional equivalent to the 'rlogin' utility, but in a secure fashion. ssh is freely available for Unix-based systems, and should be installed with an accompanying man page. ssh connects and logs into the specified hostname (with optional user name).

The user must prove his/her identity to the remote machine using one of several methods depending on the protocol version used.

General Syntax with ssh are:

```
$ ssh -l mahesh dbserver

$ ssh -l mahesh 172.24.0.252
```

Command Options

`-l` <Login name>: It specifies the user to log in as on the remot machine.

OR

```
$ ssh mahesh@172.24.0.252
```

2.2 Find Information About Your System

There are many commands on the Unix system. We are going to see some of the most frequently used commands in this section. As and when required, some new commands are introduced in relavent sections of other chapters. But before we start with this section, let us get introduced to a command we will be using to output some message on the screen or to see values of certain variables. It is `echo`.

```
$ echo “focus training institute”

$ echo “$USER”
```

USER is a system variable, whose value is accessed by saying $USER and it is being echoed in second example. We will look at system variables and user-defined variables in the chapter about variables.

2.2.1 whoami

If you are logged in with a username of “mahesh”, the `whoami` command will print ‘mahesh’ on the terminal. In another words, it prints the current userid.

```
$ whoami
 mahesh
$ ssh -l shekhar 172.24.0.252
 shekhar@172.24.0.252's password:
 Last login: Sat Dec  4 17:51:13 2010 from www.ftb.com
================================================================
            WELCOME TO FOCUS TRAINING SERVICES
=================================================================
$ whoami
shekhar
$
```

2.2.2 users

It prints ids of users who are currently logged in to the host (server).

```
$ users
apuser1 apuser1 apuser2 apuser3 apuser4 gbuser12 htuser13
htuser6 htuser7 kjuser3 kjuser4 nagnath nagnath oguser10 oracle
oracle rkuser10 rkuser18 rkuser2 rkuser32 rkuser9 root ssuser1
stuser1 stuser1
$
```

2.2.3 who

The who command displays a list of users currently logged in to the local system in detailed format.

It displays each user's

- login name
- the login device (TTY port)
- the login date and time

The command reads the binary file /var/admn/utmpx to obtain this information and information about where the users logged in from. If a user lis ogged in remotely the who command displays the remote host name or internet Protocol (IP) address in the last column of the output.

It's often a good idea to know user id's logged on to a system so we can mail them messages.

```
$ who
stuser1  pts/0        2011-12-12 09:58 (172.24.1.180)
htuser7  pts/1        2011-12-12 10:57 (172.24.0.122)
stuser1  pts/2        2011-12-12 09:56 (172.24.1.180)
apuser1  pts/3        2011-12-12 10:53 (172.24.8.40)
kjuser3  pts/4        2011-12-12 11:21 (172.24.0.130)
oracle   pts/5        2011-12-12 10:45 (172.24.8.40)
htuser6  pts/6        2011-12-12 11:09 (172.24.0.129)
htuser10 pts/7        2011-12-12 11:02 (172.24.0.241)
```

Here

- 1st column shows the userid of users who are logged on server.
- 2nd column shows device names of their respective terminal. These are the filenames associated with the terminals. (mahesh's terminal is pts/1).
- 3rd and 4th column shows date and time of logging in.
- 5th column shows machine name/ip from where the user has logged in.

The command has more options which can be used.

Command Options

`-H`: The option prints the column headers.
`-u`: The option prints with some more details like PID, IDLE time.
`-b`: The option indicates the time and date of the last reboot.

2.2.4 w

Shows who is logged and what they are doing.

Unix keeps record of what every user is doing on the system at a given time. The `w` command displays information about the users currently on the machine, and their processes.
The header in the output of this command shows the current time, how long the system has been running, how many users are currently logged on, and the system load averages for the past 1, 5, and 15 minutes in that order.

```
$ w
18:35:12 up 19:11, 7 users, load average: 0.01, 0.03, 0.00
USER TTY FROM LOGIN@ IDLE JCPU PCPU WHAT
Htuser1 pts/1 station111.examp 16:25 2.00s 2.22s 0.53s sqlplus
Stuser5 pts/3 station121.examp 18:30 9.00s 2.55s 0.77s vim
```

The following entries are displayed for each user: login name, the tty name, the remote host, login time, idle time, JCPU, PCPU, and the command line of their current process.
The JCPU time is the time used by all processes attached to the tty.
The PCPU time is the time used by the current process, named in the "what" field.

2.2.5 uname

Displays your machine’s charecteristic.

`uname` command displays certain features of the operating system running on your machine. By default it simply displays the name of operating system.

Syntax:

$ `uname` [-a] [-i] [-n] [-p] [-r] [-v]

```
$ uname
Linux
$
```

Linux system simply shows Linux.

Using suitable options you can display certain key features of operating system.

Command Options

`-r:` It displays current release

Since UNIX commands vary across versions so much so that you'll need to use -r option to find out the version of your operating system.

```
$ uname -r
2.6.18-194.el5
$
```

`-a:` It displays everything related to your machine.
It will show you following key points.

- kernel name
- node name
- kernel release
- kernel version
- machine
- processor
- hardware platform

```
$ uname -a
Linux station60.example.com 2.6.18-194.el5 #1 SMP Tue Dec 10
21:52:43EDT 2010 i686 athlon i386 GNU/Linux
$
```

`-i:` it prints the name of the hardware implementation (platform).

`-n:` it prints the nodename (the nodename is the name by which the system is known to a communications network).

2.2.6 uptime

Tells how long the system has been running.

`uptime` gives a one line display of the following information.

The current time, how long the system has been running, how many users are currently logged on, and the system load averages for the past 1, 5, and 15 minutes.

```
$ uptime
18:56:15 up 19:32, 8 users, load average: 1.60, 1.11, 0.63
$
```

2.2.7 date

The `date` command can be used to display or set the date.

Fortunately there are options to manipulate the format. The format option is preceded by a + followed by any number of field descriptors indicated by a % followed by a character to indicate which field is desired. The allowed field descriptors are:

%m	month of year (01-12)
%n	prints output to new line
%d	day of month (01-31)
%y	last two digits of year (00-99)
%D	date as mm/dd/yy
%H	hour (00-23)
%M	minute (00-59)
%S	second (00-59)
%T	time as HH:MM:SS
%j	day of year (001-366)
%w	day of week (0-6) Sunday is 0
%a	abbreviated weekday (Sun-Sat)
%h	abbreviated month (Jan-Dec)
%r	12-hour time w/ AM/PM (e.g., "03:59:42 PM")

Examples

```
$ date
Mon Jan      6 16:07:23 PST 1997

$ date '+%a %h %d %T %y'
Mon Jan 06 16:07:23 97

$ date '+%a %h %d %n %T %y'
Mon Jan 06
16:07:23 97
```

Command Options

-s datestr

-d datesr

-s datestr: Sets the time and date to the value specified in the datestr. The datestr may contain the month names, timezones, 'am', 'pm', etc. See examples for knowing how the date and time can be set. One can set date of the system if he has root permissions. Otherwise, the `date` command with this option would simply output specified date but not change the system's actual date. The `date` command can accept arguments like “yesterday” or “next Tuesday”.
-d datestr option tells date command to display the specified date instead of actual current system date.
Examples

```
$ date -s "11/20/2003 12:48:00"

$ date -d "last friday"
```

2.2.8 cal

With no arguments, prints a calendar for the current month. It prints a 12-month calendar (beginning with January) for the given year, or a one-month calendar of the given Month and Year.
Month ranges from 1 to 12. Year ranges from 1 to 9999.
Syntax:

```
$ cal [options] [[month] year]
```

-j	Display Julian dates (days numbered 1 to 365, starting from January 1).
-m	Display Monday as the first day of the week.
-y	Display entire year.

-V	Display the source of the calendar file.

Month: Specifies the month for which we want the calendar to be displayed. It must be the numeric representation of the month. For example: January is 1 and December is 12.
Year: Specifies the year for which we want the calendar to be displayed.
Examples

```
$ cal
$ cal -j
$ cal -m
$ cal -y
$ cal -y 1980
$ cal 12 2006
$ cal 2006
```

2.2.9 ifconfig

If a user wants to check the ip-address of his machine, he can use `ifconfig` command. `ifconfig` is used to configure the kernel-resident network interfaces. It is used at boot time to set up interfaces as necessary. After that, it is usually only needed when debugging or when system tuning is needed.

```
$ ifconfig
eth0      Link encap:Ethernet  HWaddr 52:54:00:34:1B:DD
          inet addr:172.24.0.240  Bcast:172.24.255.255  Mask:255.255.0.0
          inet6 addr: fe80::5054:ff:fe34:1bdd/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:122889 errors:0 dropped:0 overruns:0 frame:0
          TX packets:52488 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:17583628 (16.7 MiB)  TX bytes:10094346 (9.6 MiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:19454 errors:0 dropped:0 overruns:0 frame:0
          TX packets:19454 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1857754 (1.7 MiB)  TX bytes:1857754 (1.7 MiB)
```

IP address of above machine is 172.24.0.240.

2.2.10 hostname

`hostname` command simply displays the fully qualified name of computer.

```
$ hostname
station60.example.com
```

2.2.11 free

Displays amount of free and used memory in the system.

- It displays the total amount of free and used physical and swap memory in the system, as well as the buffers used by the kernel.
- The shared memory column should be ignored; it is obsolete.

```
$ free
                      total         used        free     shared    buffers    cached
Mem:         2055768              1965776      89992              0     138664    1116936
-/+ buffers/cache:              710176     1345592
Swap:      4194296                          0      4194296
$
```

In above output the memory is in bytes. If user wants to display it in required format, i.e. in KB, MB or GB then he can use `-k`, `-m`, `-g` options.

E.g.

- `$ free -k`

It will show the output in Kilobytes.

- `$ free -m`

It will show the output in Megabytes.

- `$ free -g`

It will show the ouput in Gegabytes.

2.2.12 df -h

The `df` command displays information about total space and available space on a file system.

```
$ df -h
Filesystem              Size  Used   Avail   Use% Mounted on
/dev/sda1               494M   26M  444M   6%   /boot
/dev/sda2                30G   15G   14G   52%  /
/dev/sda7               2.0G  1.3G  624M   67%  /home
/dev/sda5               6.8G  1.9G  4.6G   30%  /var
/dev/sda3               7.7G  3.8G  3.6G   51%  /usr
```

`df` stands for disk free and options `-h` means human readable. So instead bytes, the output is shown in MBs and GBs.

2.2.13 history

This command shows last few commands fired by the current user. The ‘few ‘is determined by a system variable named HISTSIZE.

2.3 Manual of a command

Before we see more commands, let us see one more command called `man`. This command stands for manual pages (or simply called man-pages) of commands. It basically gives more information about a command. E.g. the below command will give you detailed information about a command `date`.

```
$ man date
```

As an output, you will see, generally, a huge file detailing the information about that command - like its use, the options it can take, the arguments it expects, the optionality of arguments, the format of arguments (e.g. the date format), which arguments of the command can be used together, the author of the command, the related commands, licensing etc.

`info` and `whatis` are other commands which give you some information about the commands passed as arguments to them.

2.4 Filters

Filters are a set of Unix commands which do some manipulation of the text of a file or input stream (we will see what input stream is in the chapter on Redirections).

Filters are commands that alter data passed through them, typically via pipes. Some filters can be used on their own (without pipes), but the true power to manipulate streams of data to the desired output comes from the combination of pipes and filters. Summarized below are some of the useful Unix filters.

2.4.1 head

`head` command is used to display starting portion of the file. By default head command displays the top 10 lines of file.

```
$  head /etc/passwd

root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
news:x:9:13:news:/etc/news:/sbin/nologin
```

We can override the default number of lines shown by the `head` command as follows

```
$  head -5 /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
$
```

Note: Instead of 5 we can give any number. Specified number of lines starting from first line would be displayed.

2.4.2 tail

`tail` command works exactly opposite of the head. It displays the ending portion of file. By default it also displays the 10 lines from the file. We can override the behavior as follows.

```
$  tail -3 /etc/passwd

raj:x:7276:7276::/home/raj:/bin/bash
ram:x:7277:7277::/home/ram:/bin/bash
suhas:x:7278:7278::/home/suhas:/bin/bash
$
```

2.4.3 more

The `more` command in Linux is helpful when dealing with a small xterm window, or if you want to easily read a file without using an editor to do so. `more` is a filter for paging through text one screen at a time.

Let us see main uses of more here.

2.4.3.1 To View a File Using more

```
$ more fur.sh
```

This will auto clear the screen and display the start of the file.

```
#!/bin/bash
beep 659 120   #  Treble E
beep 0 120
beep 622 120   #  Treble D#
beep 0 120
--More--(5%) <---- This line shows at what line you havereached
in the file relative to the entire file size.
```

If you hit spacebar, then the more command shows you next lines in the files, again one screen at a time.

2.4.3.2 View Two Files At the 20th Line Using more -p

If you want to see 20 lines of each file at a time:

```
$ more -p20 fur.sh data.sh
```

`more` will then display the first file, followed by the second file informing you of the file change.

2.4.4 wc

To get count of lines, words, or characters from a document, use the `wc` command.

For each file, `wc` will output three numbers. The first is the line count, the second the word count, and the third is the character count.

For example, if we entered (‘a.txt’ is name of a file in current directory)

```
$ wc a.txt
```

The output would be something similar to the following:

```
38   135 847  a.txt
```

To get only a specific number, we may use one or more of the following `wc` options:

Option	Entities counted
-c	bytes
-l	lines
-m	characters
-w	words

Note: In some versions of `wc`, the `–m` option will not be available or `–c` will report characters. In some languages, one character is represented by two bytes. In that case, -m will show number half of –c. However, in most cases, the values for `–c` and `–m` are equal.

Syntax:

```
$ wc –c  abc.txt
```

For example, to find out how many bytes are in the .login file, we could enter:

```
 $ wc -c .login
```

2.4.5 sort

`sort` is a Unix command that prints the lines of its input or concatenation of all files listed in its argument list in sorted order. The `–r` option will reverse the sort order.

By default sort command sorts in ascending order

Examples:

```
$ cat phonebook
Smith,Brett   5554321
Doe,John      5551234
Doe,Jane      5553214
Avery,Cory    5554321
Fogarty,Suzie 5552314
```

```
$ sort phonebook
Avery,Cory    5554321
Doe,Jane      5553214
Doe,John      5551234
Fogarty,Suzie 5552314
Smith,Brett   5554321
```

The `-n` option makes the program to sort according to numerical value. But if the first column of file does not contain numerical data then it will not sort according to numbers. We have to provide the position of column that contains numeric values by using `-k` option

```
$ cat student.txt
harsh     10
mahesh    5
uday      55
```

```
$ sort student.txt -nk2
mahesh    5
harsh     10
uday      55
```

`-k` will work when the column separator is space. If the separator is other than space then use `-t` option ('t' for field-terminator).

```
$ cat student.txt
harsh:10
mahesh:5
uday:55
```

```
$ sort student.txt -t”:” -nk2
mahesh:5
harsh:10
uday:55
```

The `-r` option just reverses the order of the sort:

```
$ sort -r zipcode
Wendy 23456
Sam   45678
Joe   56789
Bob    34567
```

2.4.6 uniq

The `uniq` command removes the consecutive duplicate lines from the input. The consecutive word is important here. If you have duplicate lines in a file, but scattered, you should first sort the contents of that file using the `sort` command and then apply `uniq` on that sorted output. For this sort-uniq sequence to work, we should know redirection. We will revisit this command in Redirection’s chapter. To make life simpler, the `sort` command above has an option `-u`, which allows `sort` command to sort the input and then remove the duplicates before displaying the output.

```
$ cat cities.txt
Pune
Kolhapur
Kolhapur
Pune
Mumbai
```

```
Mumbai
Satara
```

```
$ uniq cities.txt
Pune
Kolhapur
Pune
Mumbai
Satara
```

One of the options is –c, which gives the count of consequently duplicates for each line.

```
$ uniq –c cities.txt
1 Pune
2 Kolhapur
1 Pune
2 Mumbai
1 Satara
```

2.4.7 cut

`cut` is a Unix command which is typically used to extract a certain range of characters from a line, usually from a file. We will use a file company.data which contains some lines as below.

```
$ cat company.data
406378:Sales:Itorre:Jan
031762:Marketing:Nasium:Jim
636496:Research:Ancholie:Mel
396082:Sales:Jucacion:Ed
```

Examples:

If you want to print just character 1 to 6 of each line (the employee serial numbers), use the -c1-6 flag, as in this command:

```
$ cut -c1-6 company.data
406378
031762
636496
396082
```

If you want to print just character 4 and 8 of each line use the c4,8 flag, as in this command:

```
$ cut -c4,8 company.data
3S
7M
4R
0S
```

`-f` specifies a field list, The line being cut is supposed to be comprised of fields. The ‘fields’ in a line are determined by delimiter specified by ‘ `-d` ’ option. Fields are incrementally numbered starting from 1.

`-d` delimiter. The character immediately following the `-d` option is the field delimiter for use in conjunction with the `-f` option.

The default delimiter is tab. Space and other characters with special meanings within the context of the shell must be quoted or escaped as necessary. And since this file has fields delimited by colons, we can pick out just the last names by specifying the `d` and `f3` options, like this:

Examples using d and f options:

If you want to access third field. Specify “:” as delimiter and 3 as field number.

```
$ cut -d”:” -f3 company.data
Itorre
Nasium
Ancholie
Jucacion
```

If you want to access multiple fields, you can separate them by comma.

```
$ cut -d”:” -f1,3 company.data
406378:Itorre
031762:Nasium
636496:Ancholie
396082:Jucacion
```

2.4.8 tr

It transforms the incoming data. It works on standard input, not on any file. The transformation can be in different ways, we will see two of them - squeezing multiple occurrences of a character into one, and switching between uppercase and lowercase characters.

```
$ tr –s “e” [Enter]
Heeeeeeeeeeeello[enter]
Hello ← this is the output that you see
Ctl+d ← Press this key-combination to say that the input is over

$ tr [a-z] [A-Z]
Hello[enter]
HELLO ← this is the output that you see
Ctl+d

$ tr [A-Z] [a-z]
HELLO [enter]
hello ← this is the output that you see
Ctl+d
```

The square brackets specify the range of characters.

2.4.9 paste

`paste` is a Unix utility tool which is used to join files horizontally (parallel merging. It is effectively the horizontal equivalent to the `cat` command which operates on the vertical plane of two (or more) files, i.e. by append contents of one file to another in order.

Example:

To paste several columns of data together, enter:

```
$ paste who where when > www
```

This creates a file named www that contains the data from the “who” file in one column, the “where” file in another, and the “when” file in a third. If the “who”, “where”, and “when” files look like:

Contents of who	Contents of Where	Contents of when
Sam Dave Sue	Detroit Edgewood Tampa	January 3 February 4 March 19

Then the www file will contain:

```
   Sam      Detroit        January 3
   Dave     Edgewood       February 4
   Sue      Tampa          March 19
```

2.4.10 grep

`grep` is one of the most frequently used text processing tools. The acronym stands for "Global Regular Expression Print".

`grep` command searches the given file for lines containing a match to the given strings or words.

By default, `grep` prints the matching lines. Use `grep` to search for lines of text that match one or many regular expressions, and outputs only the matching lines.

If you want to count of a particular word in log fileyou can use `-c` option to count the word.

Below command will print how many times word "Error" has appeared in logfile.txt

```
$ grep -c "Error" logfile.txt
```

Sometime we are interested not just in matching line but also in lines around the matching lines. This is particularly useful to see in log files what happened before any Error or Exception. `grep --context` option allows us to print lines around matching pattern. Below example of `grep` will print 6 lines around matching line of word "successful" in logfile.txt

```
$ grep --context=6 successful logfile.txt
```

This is very useful to see what is around and to print whole message if it splits around multiple lines.

You can also use command line option `-C` instead of "context" for example

```
$ grep -C6 'hello' logfile
```

If you want to do case insensitive search then use option `-i` from `grep` command.

`grep -i` will find occurrence of all of Error, error and ERROR and quite useful to display any sort of ‘Error’ from log file.

```
$ grep -i Error logfile
```

Use `grep -o` if you find whole word instead of just pattern.

```
$ grep -o ERROR logfile
```

Above command searches only for instances of 'ERROR' that are entire words.

Another useful `grep` option is " `-l`" which displays only the file names which matches the given pattern. Below command will only display file names which have ERROR.

```
$ grep -l ERROR logfile
$ grep -l 'main' *.java # search java files with main() method
```

If you want to see line number of matching lines you can use option "`grep -n`". Below command will show on which lines ‘Error’ has appeared.

```
$ grep -n ERROR logfile
```

`grep` can show matching pattern in color which is quite useful to spot the word in long line. To see matching pattern in color, use below command.

```
$ grep Exception today.log --color
```

2.5 File Related Commands

2.5.1 pwd

The `pwd` command shows the user’s present working directory. This is quite useful to determine quickly where one is on the entire file system of the Unix machine.

2.5.2 ls

The `ls` command lists the files in your current working directory. When we log onto your account on Unix machine, your current working directory is your home or personal directory. This is the directory in which we have personal disk space to put files on or to create sub-directories under. The `ls` command also has many options.

Example 1:

```
$ ls
case.sh  for.sh  hello.sh  if.sh
$
```

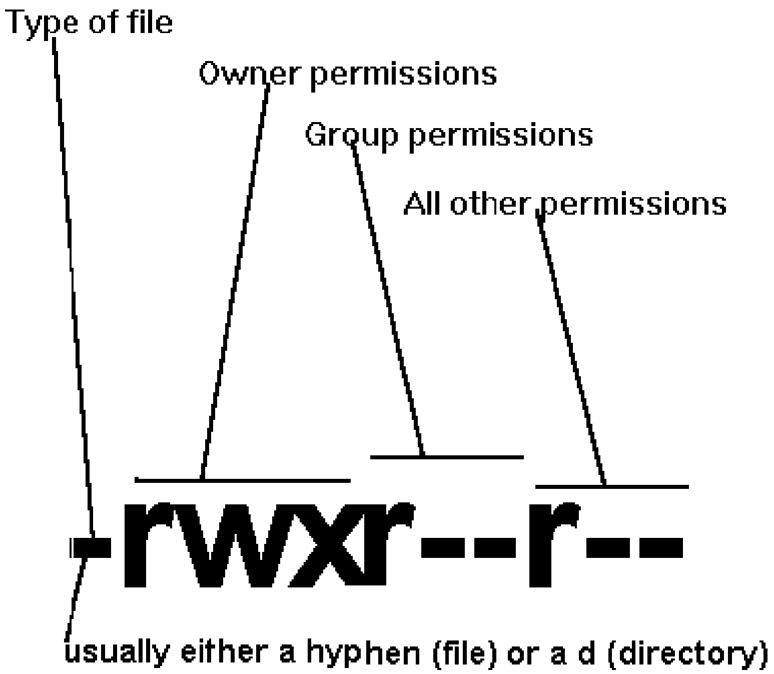
The `ls` command without any option displays the files and directories. `-l` displays the long listing of files.

Example 2:

```
$ ls  -l
total 0
-rw-rw-r-- 1 mahesh1 mahesh1 0 Dec 22 19:05 case.sh
-rw-rw-r-- 1 mahesh1 mahesh1 0 Dec 22 19:05 for.sh
-rw-rw-r-- 1 mahesh1 mahesh1 0 Dec 22 19:05 hello.sh
-rw-rw-r-- 1 mahesh1 mahesh1 0 Dec 22 19:05 if.sh
$
```

Column 1- Tells us the type of file, what privileges it has and to whom these privileges are granted. There are three types of privileges. Read and write privileges are easy to understand. The exec privilege is a little more difficult. We can make a file "executable" by giving it exec privileges. This means that commands in the file will be executed when we type the file name at the command prompt. A directory too can be executable. A directory being executable means that the user can list contents of that directory.

Privileges are granted to three levels of users:



- (hyphen) = permission not granted
r = read permission
w= write permission
x= scan or execute permission

1) The owner of the file. The owner is usually, but not always, the userid that created the file.
2) The group to which the owner belongs.
3) Everybody else who has an account on the Unix machine where the file resides.

Column 2 - Number of links

Column 3 - Owner of the file. Normally the owner of the file is the user account that originally created it.

Column 4 - Group under which the file belongs. This is by default the group to which the account belongs or first three letters of the userid. The group can be changed by the chgrp command.

Column 5 - Size of file (bytes).

Column 6 - Date of last update

Column 7 - Name of file

Example 3:

```
$ ls -ld /usr
drwxr-xr-x 16 root root 4096 Sep 20 20:06 /usr
$
```

Command Options

-d: This option shows information about directory. Rather than listing the files contained in the /usr directory, this command lists information about the /usr directory itself. This is very useful when we want to check the permissions of the directory but not the content of the directory.

-a: Shows us all files, even files that are hidden (names of hidden files/folders begin with a dot.)

Example 4:

```
$ ls -a
 .bash_logout   .bashrc  .emacs  hello.sh  .kde .bash_profile
case.sh  for.sh  if.sh     .mozilla
$
```

2.5.3 mkdir

This command is used to create a directory by name of its argument. If we specify –p option, it creates a directory and all its parents too.

```
$ mkdir myDir          # This will create a directory named myDir
$ mkdir -p all/these/are/created # This will create directory
named 'created' and all its parent directories('are', 'these'
and 'all').
```

In the second example above (with –p option), the current directory would get a directory named 'all'. Inside 'all', there would be 'these' and so on.

2.5.4 cd

This command is used to change the present working directory. If used without any arguments, the user is taken back to his home directory from wherever he is on the files ystem.

In this context, we should know ‘.’ means current directory. ‘..’ means parent directory. ‘~’ means home directory (it is shortform for /home/<userid>).

The command ‘cd –‘ would take you to previous working directory. This shortcut is very handy if you need to frequently toggle between two directories.

2.5.5 cat

The cat command is used to display contents of a file and also to create one.

Examples:

a) creating a file

```
$ cat >hello.txt
hi
welcome to unix  ←————————  Enter text and end with ctrl+d
$
```

b) displaying a file

```
$ cat hello.txt
Hi
welcome to unix
$
```

2.5.6 cp

The cp command has various flavors, depending on whether you are making copies of files, or copying files from one place to another and whether you are dealing with multiple file or whether one of the source or destination directories is current directory. Each one is discussed below.

2.5.6.1 Copying one file to another

cp command copies file or group of files. It creates exact image of the file on disk with different name. The syntax requires at least two filenames to be specified in the command line. When both are ordinary files, the first is copied to second:

```
$  cp file1 file2
```

If the destination file (file2) does not exist, it will first be created and then contents of file1 are put into it. If file2 exists, it will simply be overwritten without any warning by the system. To avoid accidental overwriting, use `–i` option. This option tells the `cp` command to ask the user confirmation before overwriting existing contents of the target file (file2 in this case).

2.5.6.2 Copying file to different directory

The `cp` is often used with the shorthand notation . (dot), to signify the current directory as the destination. For instance, to copy the file userlist.txt from /home/mahesh to your current directory, following command would do it.

```
$ cp /home/mahesh/userlist.txt .
```

2.5.6.3 Copying multiple files to a directory

`cp` can also be used to copy more than one files with a single invocation of the command. In that case the last filename must be a directory. You can use the `cp` command as follows.

```
$ cp file1 file2 file3 backup
```

2.5.6.4 To handle files with similar names

You can use wildcard * as follows

```
$ cp file* backup
```

where the all the files named `file1, file2, file3` and likewise will be copied into `backup` directory.

2.5.7 mv

`mv` command has two distinct functions.

- It renames file(or directory)
- It moves a group of files to different directory

`mv` doesn't create a copy of the file but it renames the it. No additional space is consumed on disk during renaming. To rename the file hello.sh to welcome.sh, use the following command.

```
$  mv hello.sh welcome.sh
```

`mv` simply replaces the filename in the existing directory entry with the new name.

2.5.8 rm

The `rm` command deletes one or more files. It normally operates silently and should be used with caution.

The following command deletes two files.

```
$  rm file1 file2
```

Note: `rm file*` command will be dangerous to use. In this case it will remove all the files whose name starts with “file” and end with any other characters.

Now see the following command that will delete all files in current directory.

```
$ rm *
```

Command Options

`-i` option warns the user before deleting the files.

`-r` option is for recursively deleting the directories.

```
$  rm -i for.sh
 rm: remove regular empty file `for.sh'? y
$
```

2.5.9 rmdir

This command removes directory but only when the directory is empty. That way, rm –r is more efficient than rmdir.

```
$  rmdir myDir  # results in error if myDir is not empty
```

2.5.10 touch

The touch command changes the last modified time and last access time associated with the file. This command takes file-name as its argument. If the file does not exist, the command creates a zero-byte file with that name with the current timestamp.

```
$ touch abc.txt
```

One of the command options is –d, which tells the touch command to use the specified date/time instead of current time to stamp the file. The format of date and time is explained in details in man-page of the command.

```
$ touch -d "21 Dec 2012" abc.txt
```

2.6 Other commands

2.6.1 ping

Tries to send some data to the given IP address and receive it. If data is not received in some predefined time, declares that the IP address is not reachable. This command is generally used to check if a machine is reachable or not on the network. The ping command continuously keeps sending and receiving data – it has to be killed explicitly. However if you want it to send data only specified number of times, you can use –c option to tell it so.

```
$ ping 172.24.0.240 # pings continuously until ctrl+c is pressed
$ ping -c1 172.24.0.240 # pings only once.
```

2.6.2 clear

Clears the contents of the screen and shows a blank screen having command prompt as its first line.

2.6.3 sleep

This command makes the executing shell suspend all the tasks for specified number of seconds. The shell simply remains inactive for those many seconds.

```
$ sleep 10 # sleep for 10 seconds.
```

2.6.4 time

The time command can be used to determine how much time a script/command is takeing to execute and how much of it is spent by the system (Some scripts need user input when the system is idle. The time command tells you both times separately)

```
$ time date
$ time sh myscript
```

2.6.5 alias

Sets an alias for a command. E.g. if typing clear everytime is tedious, you can set cls as its alias. This way, whenever you type cls, internally clear command is fired and you get funcationality of clear command by typing only three characterss instead of six.

```
$ alias ‘cls=clear’

$ alias ‘mydate=date +%d-%h-%Y’ # now mydate acts as a command

displaying date in 21-Dec-2012 format.
```

If you want to remove an alias, you can use `unalias`.

```
$ unalias mydate
```

3 Shell Variables

A variable is a place for storing data. They are a named entity, for example ‘abc’. The variable ‘abc’ may hold string data like “Focus” or numeric data like 15. Variables can be defined on command prompt or in a script. The value of the variable can be changed during the program execution.

The value assigned could be a number, text, filename, device, or any other type of data. A variable is nothing more than a pointer to the actual data. The shell enables you to create, assign, and delete variables.

While similar to other programming language variables, shell variables do have some different characteristics such as:

- shell variables do not need to be declared
- all shell variables are of type string, though depending on context value of them can be treated as string or number. E.g. in case of mathematical calculations a sequence of numeric characters will be treated as a number.

There are two types of variables

3.1 User defined variables

3.1.1 How to assign value to a variable

Syntax:

```
variablename=value
```

'value' is assigned to given 'variable name' and Value must be on right side = sign.

Example:

```
$no=10  # this is ok
```

```
$10=no # Error, NOT Ok, Value must be on right side of = sign.
Also, a variable name cannot be number.
```

To define variable called 'os' having value unix

```
$os=unix
```

To define variable called n having value 10

```
$ n=10
```

3.1.2 Rules for Naming variable name

Variable name must begin with alphabetic character or underscore character (_), followed by one or more Alphanumeric characters. For e.g. Valid shell variable are as follows

```
HOME
SYSTEM_VERSION
VehicleNo
```

Don't put spaces on either side of the equal sign when assigning value to variable. For e.g. In following variable declarations there will be error

```
$ no =10

$ no= 10

$ no = 10
```

Variables are case-sensitive. All four below are different variables

```
$ no=10

$ No=11

$ NO=20

$ nO=2
```

To access value of a variable, we need to use ‘`$`’ followed by the variable name. And to see value of a variable, `echo` is a good way. E.g.

`$ echo` “`$No`” will print 11 if variable ‘`No`’ was defined as above.

You can define NULL variable as follows (NULL variable is variable which has no value at the time of definition). e.g.

```
$ vech=

$ vech=""
```

Try to print it's value by issuing following command

```
$ echo $vech
```

Nothing will be shown because variable has no value i.e. NULL variable.

Do not use ?,* etc, to name your variables.

3.1.3 How to access the value of variable

Use a $ to access the value of a variable.

```
$ echo $a
10
$
```

3.1.4 Variables are not declared

If we try to access a variable that is not declared, a blank line will appear meaning the value of that variable is null or not assigned.

```
$ echo $b
$
```

In above example note that the variable `b` was not defined earlier, still the shell did not report any error. It behaved as if `b` was declared as null.

3.1.5 How to capture output of a command in a variable

Use back quote (`) above the 'tab' key on keyboard to capture the output of a command in a variable

```
$ user=`whoami`
$ echo $user
mahesh
$
```

There is another way to capture the output of a command to a variable

```
$ user=$(whoami)
$ echo $user
mahesh
$
```

3.1.6 Arithmetic on variables

As mentioned at the beginning of this section, all shell variables are of type string. This might lead one to conclude that arithmetic using shell variables is not possible because you can't do arithmetic on strings. But it is not true. You can do arithmetic on shell variables. The standard tried and tested way of doing arithmetic is using the `expr` command. In general, `expr` is used as follows:

```
expr operand1 operator operand2
```

Note the spaces on either side of the operator, these are mandatory and a source of frequent errors. Some of the possible operators include:

```
addition         +

subtraction      -
multiplication   *     # must be written with \ before the *
                        (See the example below)
division         /
modulus          %
```

Thus you can do arithmetic in the shell such as:

```
$ expr 10 + 2 [Enter]      #adding 10 + 2

  12

$ I=10

$ expr $I + 2 [Enter]     # same using a variable

  12

$ expr 10 \* 2 [Enter]                  # multiplying by 2

  20
```

The multiplication operator (*) has another meaning for shell. A * in Shell is a wildcard. It is used in conjunction with the file names. To remove this special meaning, escape it with “\”.

The `expr` command deals with integers. For decimal point calculations, use another command `bc`. On the command prompt, type `bc`. Enter. Now write expressions here and press enter. You will see the result on the screen. They are still integer. To specify how many decimals the bc should use, type 'scale=3' (for three decimal places) and press enter. All the results now will have three decimal places wherever applicable. Exit from bc by entering 'quit'. bc is a lot powerful, it can have its own loops and conditions. But that is beyond scope of this book.

```
$ bc

1+2 [Enter]
3
3*5[Enter]
15
1+2*5[Enter]
11
1/3[Enter]
0
5/3[Enter]
1
scale=3[Enter]
5/3[Enter]
1.666
1+1[Enter]
2
```

There is an alternative way to performing arithmetic calculations available in some of the newer shells (e.g. bash, ksh93). This newer method (sometimes referred to as let) uses the following syntax:

$((expression))

For example:

```
$ echo $((10 + 2)) [Enter]     # with spaces around operator

  12

$ echo $((10+2)) [Enter]                # without spaces

  12

$ X=10 [Enter]

$ echo $((X + 2)) [Enter]               # note no $ before X

  12
```

3.2 System variables

Some variables are created and maintained by Linux/Unix itself. This type of variables are defined in CAPITAL LETTERS. These variables are generally set using the export command. Some of the commonly required variables are as follows.

PS1	Command prompt
HOME	Your home directory
PWD	Your present working directory
LOGNAME	Your login name
USER	Same as login name
HISTSIZE	Number of history commands to be stored in history file
HOSTNAME	Your host name
SHELL	Your login shell
PPID	Process id of parent process

3.2.1 PATH variable

PATH variable holds absolute path of some directories. When we type any command on command prompt, the shell looks for the executable of the command in the directories specified in path variable. If it doesn't get the executable code in any of the directories mentioned in the PATH it displays error as command not found.

See how to set a path variable.

```
$  echo $PATH
/lib/qt3/bin:/usr/kerberos/bin:/usr/local/bin:/usr/bin:
/opt/real/RealPlayer:/home/mahesh1/bin:/bin
```

The above command will displays the path of your system. Observe the bin is present in current path.

Now type the following command. The `which` command tells us where is the executable code of a command is stored.

```
$  which ls
/bin/ls
```

`which` command tells the path of `ls` command i.e . `ls` command's executable code is stored on `bin` directory and since the `/bin` set in PATH variable we can execute the `ls` command successfully.

```
$  ls
a.txt  case.sh  for.sh  hello.sh  hello.txt  if.sh
$
```

Now remove the /bin from PATH as follows

```
$
PATH=/lib/qt3/bin:/usr/kerberos/bin:/usr/local/bin:/usr/bin:/opt
/real/RealPlayer:/home/mahesh1/bin
```

Now /bin is not present in PATH hence we cannot execute the `ls` command.

```
$  ls
-bash: ls: command not found
$  ls
```

Now set your PATH variable as it was earlier. You will be able to execute the `ls` command. The different directories in the PATH variable are separated by colone (:).

3.3 Escape character and quotes

3.3.1 Backslash

The backslash character (\) has two uses (note this differs from the frontslash (/) character). The first use is for line continuation. If a line of shell commands becomes exceedingly long, it may be useful to continue it across more than one line. Note that if you choose to do this, the split cannot be in the middle of a word, it must be done in appropriate whitespace.

The second (and perhaps more useful) use of the backslash character is to remove the special meaning of the following single character. This is sometimes referred to as escaping the special meaning of the following character.

Characters that have special meaning are referred to as metacharacters (there are many of these in Unix). We saw in the previous section that performing multiplication using `expr` required the \ to be used before the * character. This is because the * is a metacharacter, and without using the backslash to remove its special meaning, an error would result in `expr` command.

For example, if we wanted to output a statement to describe the use of the $? variable, we could try and observe:

```
$ echo The $? variable holds the exit status [Enter]
  The 0 variable holds the exit status
```

Whereas what we really want is:

```
$ echo The \$? variable holds the exit status [Enter]
  The $? variable holds the exit status
```

3.3.2 Back Quotes

The back quotation mark character (i.e. `) when used in pairs enclosing a command serve to perform command substitution. That is, when used as follows

`command`

the output of command is substituted at the location of the leftmost backquote.

For example, if we wanted to output:

My current directory is: <current directory location>

We could try the following:

```
  echo My current directory is: pwd [Enter]
```

But this would result in:

```
  My current directory is: pwd
```

To achieve what we wish, we could use back quotes as follows:

```
 $ echo My current directory is: `pwd` [Enter]
  My current directory is: /home/mthomas
```

Note the / in /home is substituted exactly at the location of the leftmost backquote. We can also perform assignment using the backquote characters, for example:

```
$ CUR_DIR=`pwd` [Enter]          # note no spaces around the =
```

Or with respect to the expr command:

```
$ I=10 [Enter]
$ I=`expr $I + 1` [Enter]
$ echo $I [Enter]
11
```

An alternative notation for command substitution (present in more modern shells) is the $(command) syntax.

```
$ echo My current directory is: $(pwd) [Enter]
 My current directory is: /home/mthomas
```

Technically, a single one of these quotes is called a grave accent, but are sometimes informally referred to as backticks, or back tick marks.

3.3.3 Single Quotes

Single quotes (not to be confused with the back quotes) serve to remove (escape) the special meaning of all characters enclosed by them. Thus, the following statement would work as follows:

```
$ echo 'My current directory is: `pwd`' [Enter]
  My current directory is: `pwd`
```

3.3.4 Double Quotes

Double quotes or front quotes (again, not to be confused with the back quotes or single quotes or two single quotes) serve to remove (escape) the special meaning of all characters enclosed by them, **except** for the $ (dollar sign) character, the \ (backslash) character, and the ` (back quote character). Thus, the following statement would work as follows:

```
$ CUR_DIR=`pwd` [Enter]
$ echo "My current directory is: $CUR_DIR" [Enter]
  My current directory is: /home/mthomas
```

Note that if single quotes were used in this example, the value stored in the $CUR_DIR variable would not be displayed. It is the practice of the author to always enclose text and variables within double quotes, and escape any special characters using the backslash.

Note that if any pair of quotes is unmatched (missing either of the quotes), a single greater than (>) character is displayed as below:

```
 $ echo "My current directory is: $CUR_DIR [Enter]
>
```

The > character in this instance is the shell environment variable named PS2 (**p**rompt **s**tring 2). Do not confuse this with an output redirection character (see next section). When this occurs, the shell is trying to parse the command and is missing one or more characters it needs to complete its parsing. If you understand what is missing, you may be able to recover at the > prompt by typing the missing characters. Otherwise, you may want to abort the command with a *ctrl+c* sequence.

4 Unix Filesystem

The Unix file system is a methodology for logically organizing and storing large quantities of data such that the system is easy to manage. A **file** can be informally defined as a collection of (typically related) data, which can be logically viewed as a stream of bytes (i.e. characters). A file is the smallest unit of storage in the Unix file system. By contrast, a **file system** consists of files, relationships to other files, as well as the attributes of each file. File attributes are information relating to the file, but do not include the data contained within a file. File attributes for a generic operating system might include (but are not limited to):

- a file type (i.e. what kind of data is in the file)
- a file name (which may or may not include an extension)
- a physical file size
- a file owner
- file protection/privacy capability
- file time stamp (time and date created/modified)

Additionally, file system provides tools which allow the manipulation of files, provide a logical organization as well as provide services which map the logical organization of files to physical devices. From the beginners' perspective, the Unix file system is essentially composed of files and **directories**. Directories are special files that may contain other files.

The Unix file system has a hierarchical (or tree-like) structure with its highest level directory called root (denoted by /, pronounced *slash*). Immediately below the root level directory are several subdirectories, most of which contain system files.

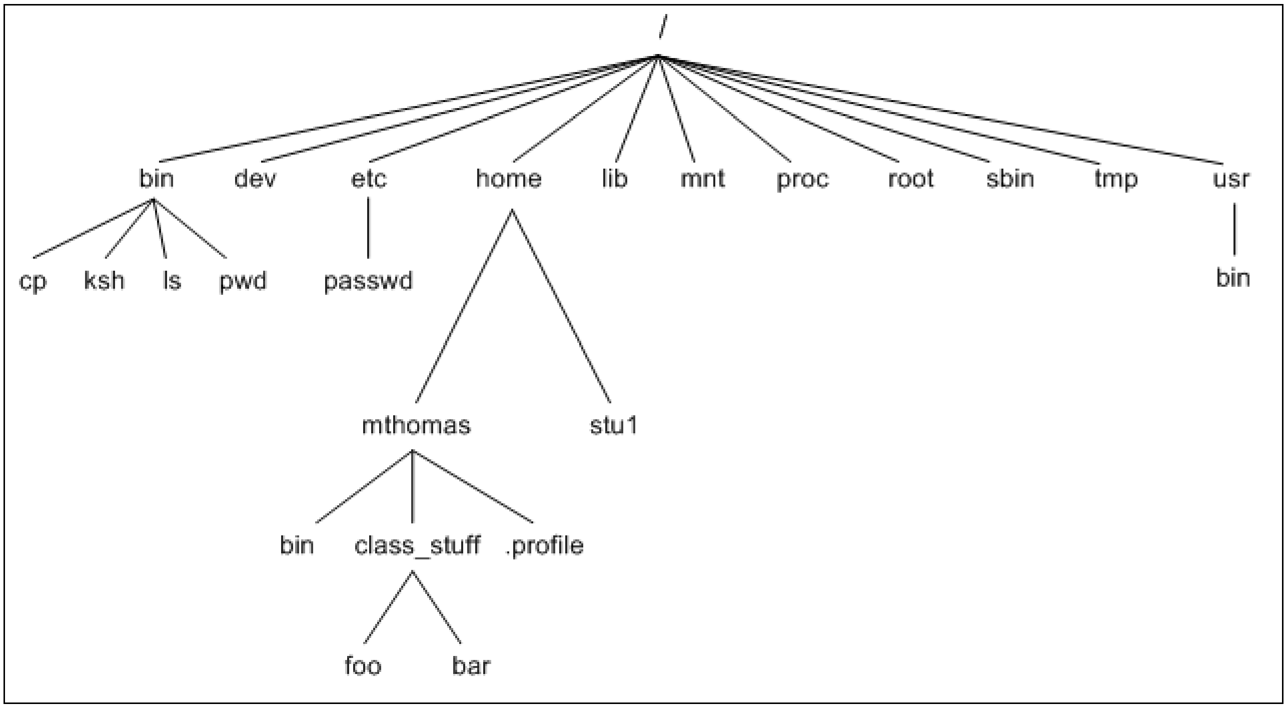
Below this can exist system files, application files, and/or user data files. All files on a Unix system are related to one another. Each directory has a parent directory; slash being the top-most parent. There is no parent to slash.

Below is a diagram of a typical Unix file system. As you can see, the top-most directory is / (slash), with the directories directly beneath being system directories. Note that as Unix implementations and vendors vary, so will this file system hierarchy. However, the organization of most file systems is similar.

Tasks carried out by file system are:

- Making files available to users.
- Managing and monitoring the system's disk resources.
- Protecting against file corruption, hardware failures, user-errors through backup.
- Adding more disks, tape drives, etc when needed.

When Unix operating system is installed, some directories are created under / (or root), such as /usr, /bin, /etc, /tmp, /home, /var. The count and names of directories may vary depending on which flavor (vendor and version) of Unix is being installed.



Purpose and contents of some of these directories:

- **etc -** Contains all system configuration files and the files which maintain information about users and groups.
- **bin -** Contains all binary executable files (commands that can be used by normal user)
- **usr -** Default directory provided by Unix OS to create users' home directories and contains manual pages - also contains executable commands.
- **tmp -** System or users create temporary files which get removed when the server reboots.
- **dev -** Contains all device files i.e. logical file names to physical devices.
- **lib** - Contains all library files
- **mnt** - Contains device files related to mounted devices
- **proc** - Contains files related to system processes
- **root** - The root user's home directory (The super user is at times called 'root' user. The top most directory too is called 'root' of file system. And this directory also is named 'root'. Based on context, the meaning of 'root' needs to be understood).
- **home -** Default directory allocated for the home directories of normal users when the administrator don't specify any other directory.
- **var -** Contains all system log files and message files.
- **sbin -** Contains all system administrator executable files (commands which generally normal users don't have the privileges to execute).

5 The vi editor

5.1 Using the vi editor

The vi editor is a screen−based text editor available on all Unix computers. It takes some effort to learn because it doesn't have user interface with buttons and menus and mouse is redundant. Then why bother with VI? Because-

- sometimes it's the only available editor
- when you log on remotely (`ssh`) to a Unix host from a Mac or PC, only the text editors (VI and emacs and pico) can be used to edit files, because they require no mouse
- mouse movements (menus, highlighting, clicking, scrolling) slow down the touch−typist

If you will be using Unix/Linux computers, especially via ssh, learn the basics of VI.

In the following text, '^' denotes a pressing of the Control key. For example, "^d" means to hold down the Control key and press the "d" key. Also "Rtn" means to press the Return (or Enter) key, while "Esc" means to press the Escape key, located in the far upper left corner of the keyboard. Also, a colon (:) after the command is only for separating command from its explanation. However a colon before the command needs to be typed while executing the command. E.g. to save the file, command is **:**`w`, where you must explicitly type colon before the 'w'.

5.1.1 Starting the vi editor

To edit a file, named say "mytext", on a Unix computer, type the command "vi mytext". Note that you must type the command with lowercase letters. If you simply typed "vi" and press enter without file name, the editor will still open with a un-named blank file.

5.1.2 Two Modes of vi editor

This is a crucial feature of `vi` editor. There are two modes, command and insert. When in insert mode, everything you type appears in the document at the blinking cursor, whereas in command mode, the keystrokes perform special functions. You must know which keystroke will switch you from one mode to the other. When you start vi editor, you are in command mode.

- To switch to insert mode: press i (or a, or o)
- To switch to command mode: press Esc

5.1.3 Moving Around in the file

When in *command mode,* you can use the arrow keys to move the cursor up, down, left, right. In addition, these keystrokes will move the cursor:

- **h:** left one character
- **l:** right one character
- **k:** up one line
- **j:** down one line
- **b:** back one word
- **f:** forward one word
- **{:** up one paragraph
- **}:** down one paragraph
- **$:** to end of the line
- **^B:** back one page
- **^F:** forward one page
- **17G:** to line #17 (that is, Press 1,7 and then shift + g)
- **G:** to the last line (that is , press shift + g)

5.1.4 Inserting Text

From command mode, these keystrokes switch you into insert mode for new text to be inserted

- **i**: insert text just before the current cursor position
- **a**: just after the current cursor position
- **o:** into a new line below current line
- **I:** at the beginning of the current line
- **A:** at the end of the current line
- **O:** into a new line above current line

5.1.5 Cutting, Copying and Pasting

From command mode, use these keystroke (or keystroke–combination) commands for the described cut/copy/paste function:

- **x:** delete (cut) character under the cursor.
- **24x**: delete (cut) 24 characters from current cursor position.
- **dd**: delete (cut) current line
- **4dd**: delete (cut) four lines, first being the current line and three below it.
- **D**: delete up to the end of the line from the current cursor position (d$ also does same)
- **dw**: delete up to the end of the current word
- **5yy**: copy (without cutting) 5 lines, first being the current line and four below it.
- **p**: paste after current cursor position/line
- **P**: paste before current cursor position/line.
- **5p:** paste five times whatever is copied using ‘yy’

5.1.6 Searching for Text

Instead of using the "Moving Around" commands, above, you can go directly forward or backward to specified text using "**/**" and "**?**". Examples:

- **/wavelet Rtn:** jump forward to the first occurrence of the string "wavelet" after current cursor position
- **?wavelet Rtn:** jump backward to the first occurrence of the string "wavelet" before current cursor position.
- In “vim” (Vi IMproved), the command **/wavelet** would highlight all the matching strings throughout the file, thereby making finding easy and making **?wavelet** a little redundant.
- **n:** The cursor jumps to the next occurrence of the word searched by by "/" or "?".
- **N:** The cursor jumps to the previous occurrence of the word searched by “/” or “?”.
- Meaning of ‘next’ and ‘previous’ interchange when used in context of forward (/) and backward (?) search.

5.1.7 Replacing Text

This amounts to combining two steps; first deleting, then inserting text.

- **r:** replace 1 character (under the cursor) with a character typed immediately after ‘r’.
- **8r**: replace each of the next 8 characters with a character typed immediately after ‘r’
- **R**: overwrite; replace text with typed input, ended with Esc
- **C**: replace from cursor to end of line, with typed input (ended with Esc)
- **S**: substitute entire line with typed input (ended with Esc)
- **4S**: substitute 4 lines with typed input (ended with Esc)
- **cw**: replace (remainder of) word with typed input (ended with Esc)

5.1.8 Miscellany

The commands on these two pages are just the start. Many more powerful commands exist in vi. More complete descriptions of all the possible commands are available on the web; search for "vi tutorial" or "vim tutorial". These are of course to be used in command mode. Useful commands include -

- **u**: undo the last change to the file (and type "u" again to re−do the change)
- **U**: undo all changes to the current line
- **^g**: show the current filename and status and line number
- **:set nu Rtn:** show all line numbers (":set nonu" gets rid of the numbers)
- **:s/Joe/Bob Rtn:** Substitute first occurrence of Joe in current line by Bob
- **:s/Joe/Bob/g Rtn:** Substitute all occurrences of Joe in current line by Bob
- **:%s/Joe/Bob/g Rtn**: Substitute every "Joe" to "Bob" throughout the document
- **J**: Joins next line at the end of current line.
- **5J**: Joins next 5 lines at the end of current line
- **xp**: exchange two characters (actually the two commands x=delete and p=paste)
- **:w Rtn** : write (save) the current text, but don’t quit VI
- **:12,17w filename Rtn:** write lines #12−17 of the current text to a (new) text file named “filename”.

5.1.9 Getting out:

When you want to get out of the editor, switch to command mode (by pressing **Esc**) and then do any of the following -

- **`:wq` :** Rtn to save the edited file and quit, or
- **`:q!` :** Rtn to quit the editor without saving changes, or
- **`ZZ` :** to save and quit (a shortcut for :wq Rtn), or
- **`:w` filename** : to save the edited file to new file "filename"

6 What happens when we login/logout

When we login to your Unix account the following three things happen.

6.1 A New Shell is started

A new shell process is started for your session. This is the login shell that has been assigned to you by the System Administrator. The name of this shell can be found out from an entry in the /etc/passwd file. This shell is acting as a middle-man between the commands that we type at the command prompt and the kernel. The process for this shell can be seen by typing the ps –f command as follows

```
[root@shekhar ~]# ps -f
UID         PID   PPID  C STIME TTY          TIME CMD
root       5794   5791  0 10:44 pts/0    00:00:00 -bash
root       5814   5794  0 10:44 pts/0    00:00:00 ps -f
```

As you can see from the above box, a process for the “bash” shell is running in the background. In this case, the PID (a unique number assigned to the process) for the shell process is 5794.

6.2 .bash_profile is executed

The second thing that happens when you login is that a special file named .bash_profile automatically executed. Every user has this file sitting in his/her home directory. Home directory is a directory assigned to each user as his/her home. If you write any Unix command in this .bash_profile file, it will get executed every time you login. Usually commands like “alias” are written in this .bash_profile file. A sample .bash_profile is show below.

```
# .bash_profile
# Get the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi


# User specific environment and startup programs
```

```
PATH=$PATH:$HOME/bin


export PATH
alias rm='rm -i'
clear
echo "----------------------------------------"
echo "Welcome $USER"
echo "----------------------------------------"
```

6.3 You are put in your HOME directory

A HOME directory is a directory that is assigned to every user by the System Administrator. You can find your home directory by reading the contents of the /etc/passwd file. The third thing that happens when a user logs into his/her account is that he/she is automatically put in his HOME directory.

You can check this by typing the “pwd” command immediately after you login. HOME directory is where the user has the privileges to create any files or directories.

```
----------------------------------------
Welcome root
----------------------------------------
$ pwd
/root  ← This is your HOME directory
$
```

If you are not the root user your home directory would be of the nature “/home/<your user name >” e.g. /home/mahesh.

7 Introduction to Shell Programming

7.1 What is shell

Computer understands the language of 0's and 1's called binary language. In early days of computing, instructions were provided using binary language, which is difficult for all of us, to read and write.

So in Unix a special program is provided named Shell. Shell accepts your instructions or commands in human-understandable language (English mostly) and if it’s a valid command, it is passed to kernel.

Shell is a user program or it iss an environment provided for user interaction. Shell is an interpreter that executes commands supplied from the standard input device (keyboard) or from a file.

Shell is not part of system kernel, but uses the system kernel for executing commands and programs..

7.2 Shell types

To know which shells are installed on your system type the following command

```
$  cat /etc/shells
/bin/sh
/bin/bash
/sbin/nologin
/bin/tcsh
/bin/csh
/bin/ksh
$
```

7.2.1 sh or Bourne Shell

The original shell still used on UNIX systems and in UNIX-related environments. This is the basic shell, a small program with few features. While this is not the standard shell, it is still available on every Linux system for compatibility with UNIX programs.

7.2.2 bash or Bourne Again shell

Bash is the standard GNU shell, intuitive and flexible. It is probably most advisable for beginners while at the same time is a powerful tool for the advanced and professional user. On Linux, bash is the standard shell for common users. This shell is a so-called *superset* of the Bourne shell, a set of add-ons and plug-ins. This means that the Bourne Again shell is compatible with the Bourne shell: commands that work in sh, also work in bash. However, the reverse is not always the case. All examples and exercises in this book use bash.

7.2.3 csh or C shell

The syntax of this shell resembles that of the C programming language.

7.2.4 tcsh or TENEX C shell

A superset of the common C shell, enhancing user-friendliness and speed. That is why some also call it the Turbo C shell.

7.2.5 ksh or the Korn shell

Sometimes appreciated by people with a UNIX background. A superset of the Bourne shell; with standard configuration. A nightmare for beginning users.

7.3 What is Shell Script?

Normally shells are interactive. It means shell accept command from you (via keyboard) and execute them. But if you frequently use a sequence of commands, then you can write this sequence of commands in text file and tell the shell to execute this file instead of you manually entering individual commands. This text file is known as shell script.

Shell script defined as:

“Shell Script is a series of commands written in a plain text file. Shell script is just like a batch file in MS-DOS but arguably is more powerful."

Shell scripting allows us to use commands in a sequence that we already use on command prompt.

We are familiar with the interactive mode of the shell. Almost everything that can be done on command prompt can be done in a script.

7.4 When to write shell scripts

Shell scripting can be applied to a wide variety of system and database tasks. Some of occasions and reasons for writing shell script are listed below.

7.4.1 Repeated Tasks

Necessity is the mother of invention. The first candidates for shell scripts will be manual tasks which are done on a regular basis.

- Backups
- Log monitoring
- Check disk space

7.4.2 Occasional Tasks

Tasks which are performed rarely enough that their procedures, or even their need may be forgotten.

- Periodic business related reports (monthly/quarterly/yearly/daily)
- Offsite backups
- Purging old data

7.4.3 Complex Manual Tasks

Some tasks must be performed manually but may be aided by scripting.

- Checking for database locks
- Killing runaway processes

7.4.4 Special Tasks

These are tasks which would not be possible without a programming language.

- Storing OS information (performance stats, disk usage, etc.) into the database
- High frequency monitoring (several times a day or more)

7.5 Our First Shell Script

Use the vi editor to create the program, for simplicity we'll call hello.sh

```
$ vi hello.sh
```

1. insert the following shell command in the `hello.sh` file:

2. Save and exit the editor program (:wq). Then run the hello.sh program.

```
#!/bin/bash
echo "Hello World!"
```

```
$ hello.sh [Enter]
```

The output you get is

```
bash: hello.sh: cannot execute - Permission denied
```

3. The problem is that you do not have ‘execute’ permission on file. So let us give execute permission on this file

```
$ chmod u+x hello.sh
```

4. Once the problem is fixed, find the path of your script. Suppose in this case the full path of hello.sh is /home/mahesh/hello.sh. Now execute the script as follows.

```
$ /home/mahesh/hello.sh [Enter]
```

5. You would see the output as below.

```
Hello World!
```

Any line starting with ‘#’ in the shell script is a comment. That line is not interpreted for execution by the shell. The newer shells also accept the ‘#’ in the middle of line. Any text beyond ‘#’ is treated as comment.

But there is a special exception to this rule. There is something called as Shebang. It starts with a ‘#’ but that is immediately followed by a ‘!’, making it ‘#!’. Shebang warrants a detailed explanation. But that is possible only after we know some syntax for shell scripting. So we will reserve that explanation for a separate chapter on Shebang.

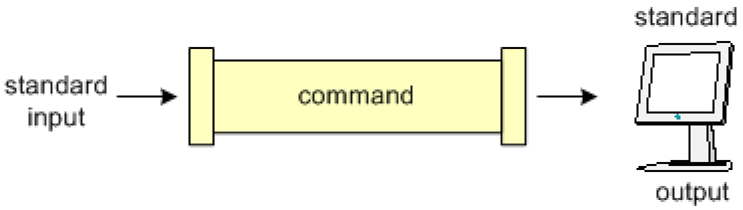
8 I/O Redirection and Pipes

This chapter describes more about the powerful Unix mechanism of redirecting input, output and errors. Topics in this chaptr include:

- Standard input, output and errors
- Redirection operators
- How to use output of one command as input for another
- How to put output of a command in a file for later referrence
- How to append output of multiple commands to a file
- Input redirection
- Handling standard error messages
- Combining redirection of input, output and error streams
- Pipes

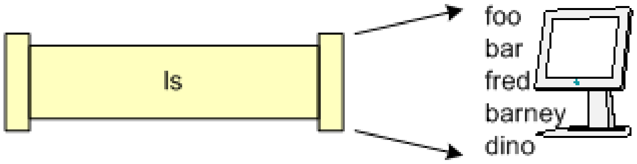
8.1 What are standard input and standard output?

Most Unix commands read input, and produce output. The input is read from keyboard or as an attribute of it and output is shown on the screen. Your keyboard is your *standard input* (stdin) device, and the screen or a particular terminal window is the *standard output* (stdout) device.



As depicted in the diagram above, input flows (by default) as a stream of bytes from standard input along a channel, is then manipulated (or generated) by the command, and command output is then directed to the standard output.

The `ls` command can be described as follows; there is really no input (other than the command itself) and the `ls` command produces output which flows to the destination of stdout (the terminal screen), as below:



8.1.1 Output redirection with >

Sometimes you will want to put

1) Output of a command in a file, or

2) Feed output of a command as input to another command. (This is important topic and is covered under its own section below called ‘Pipe Operator’)

This is known as redirecting output. Redirection is done using either the “>” (greater-than symbol), or using the “|” (pipe) operator.

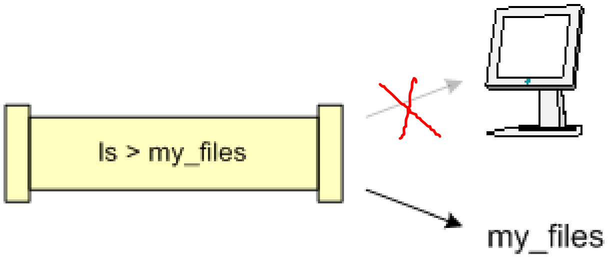
The simplest case to demonstrate this is basic output redirection. The general syntax looks as follows:

```
command > output_file_spec
```

Spaces around the redirection operator are not mandatory, but do add readability. Extending out previous example of `ls` command, we can try to redirect its output in a file named my_files:

```
$   ls > my_files [Enter]
$
```

Notice there is no output appearing after the command and you got the command prompt. This is because all output from this command was redirected to the file my_files. It is shown diagrammatically in the following diagram - no data goes to the terminal screen, but to the file instead.



Looking at the contents of my_files file, you will realize the ouput of the command has indeed gone into it.

```
$  cat my_files [Enter]
foo bar fred barney dino
$
```

In this example,

- if the file my_files does not exist, the redirection operator causes its creation, and
- if it does exist, the original contents are overwritten.

Consider the example below:

```
$ echo "Hello World!" > my_files [Enter]
$ cat my_files [Enter]
Hello World!
```

Notice here that the previous contents of the my_files file are gone, and replaced with the string "Hello World!" This might not be the most desirable behavior, so the shell provides us with the capability to append to files.

8.1.2 The append operator is >>

We can do the following:

```
$ ls > my_files [Enter]
$ echo "Hello World!" >> my_files [Enter]
$ cat my_files [Enter]
foo bar fred barney dino
Hello World!
```

From the above Example:

When using the append redirection operator,

- If file exists this operator will append the new content to the existing content
- if the file does not exist, >> will cause its creation and, append the output (to the empty file).

8.1.3 Input Redirection

You use input redirection using the ‘<’ less-than symbol and it is usually used with a program which accepts user input from the keyboard.

The general syntax of input redirection looks as follows:

```
command < input_file_spec
```

Examples:

1. A frequent use of input redirection that I have come across is mailing the contents of a text file to another user.

```
$ mail mike@somewhere.org < mail_contents.txt
```

2. Looking in more detail at this, we will use the `wc` command. The `wc` command counts the number of bytes, words and lines in a file. Thus if we do the following using the file created above, we see:

```
$ wc my_files [Enter]
6        7       39    my_files
```

Where the output indicates 6 lines, 7 words and 39 bytes, followed by the name of the file `wc` opened.

We can also use `wc` in conjunction with input redirection as follows:

```
$ wc < my_files [Enter]
  6        7       39
```

Note here that the numeric values are as in the example above, but with input redirection, the file name is not listed. This is because the `wc` command does not know the name of the file it simply received a stream of bytes as input. In this case actually, the shell opened the file and sent the contents of my_files on input stream to `wc` command. So the `wc` command never knew whether the input came from file or keyboard. It only counted whatever was on input stream. To understand this, type only the `wc` command on command prompt, you will realize that it waits for input from keyboard. Type something and end the input stream using ctrl+d. The `wc` would show statistics of whatever you typed.

8.1.4 Combining redirections

Someone will certainly ask if input redirection and output redirection can be combined, and the answer is most definitely yes. They can be combined in following situation:

Suppose you want to find the exact number of lines, number of words and characters respectively in a text file and at the same time you want to write the counts to another file. This is achieved using a combination of input and output redirection symbols as follows:

```
$ wc < my_text_file.txt > output_file.txt
```

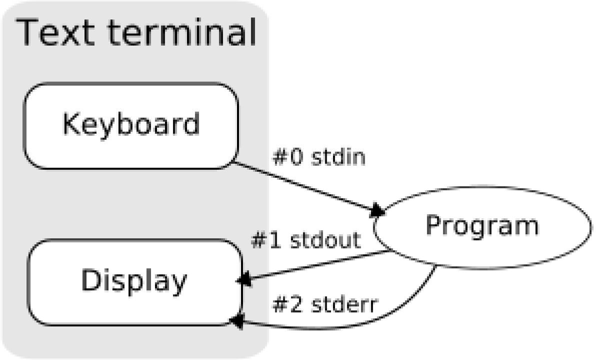
What happens above is the contents of the file my_text_file.txt are passed to the command `wc` whose output is in turn redirected to the file output_file.txt

8.1.5 Use of file descriptors

When a command outputs something on the screen, it can be a desired output from the command, or it may be an error message (e.g. the file does not exist when we invoked cat command to show contents of that file). Such error messages flow on different output stream than the standard output. The output stream for errors is called standard error. These are three types of input output streams have their own identifier called a file descriptor:

- standard input : 0
- standard output : 1
- standard error : 2

The diagram below clears the concept



In the following descriptions,

- If the file descriptor number is omitted, and the first character of the redirection operator is <, the redirection refers to the standard input (file descriptor 0).
- If the first character of the redirection operator is >, the redirection refers to the standard output (file descriptor 1).
- For redirecting an error you can't omitte the descriptor i.e. you have to write error redirection descriptor as follows.

8.1.6 Syntax of error redirection

Syntax:

```
command 2> output_file_spec
```

To show an example, we observe the following:

Here the file "foo" exist and file "bar" does not exist.

```
$ cat foo bar 2> error_file [Enter]
Hello from foo file
$ cat error_file [Enter]
cat: bar: No such file or directory
```

Note here that only the standard output appears on the screen. And when we display the contents of error_file, it contains what was previously displayed on the termimal (the error message for bar not being there).

To show another example:

```
$ ls foo bar > foo_file 2> error_file [Enter]
$ cat foo_file [Enter]
Hello from foo file
$ cat error_file [Enter]
cat: bar: No such file or directory
```

In this case both stdout and stderr were redirected to file, thus no output was sent to the terminal. The contents of each output file were what previously were displayed on the screen. Note that there are numerous ways to combine input, output and error redirection.

Another relevant topic that merits discussion here is the special file named /dev/null (sometimes referred to as the "bit bucket").

This virtual device discards all data written to it, and returns an End of File (EOF) to any process that reads from it. I informally describe this file as a "garbage can/recycle bin" like thing, except there's no bottom to it. This implies that it can never fill up, and nothing sent to it can ever be retrieved. This file is used in place of an output redirection file specification, when the redirected stream is not desired. For example, if you never care about viewing the standard output, only the standard error channel, you can do the following:

```
$ cat foo bar > /dev/null [Enter]
cat: bar: No such file or directory
```

In this case, successful command output will be discarded. The /dev/null file is typically used as a destination in cases where there is a large volume of undesired output, or cases where errors are handled internally so error messages are not warranted.

One final miscellaneous item is the technique of combining the two output streams into a single file. This is typically done with the 2>&1 command, as follows:

```
$ cat foo bar > /dev/null 2>&1 [Enter]
$
```

Here the leftmost redirection operator (>) sends stdout to /dev/null and the 2>&1 indicates that channel 2 should be redirected to the same location as channel 1, thus no output is returned to the terminal.

8.1.7 Here Document

‘<<’ redirects the standard input of the command to read from what is called a "here document". Here Documents are convenient way of placing several lines of text within the script itself, and using them as input to the command.

The << characters are followed by a single word that will later be used used to indicate the end of the Here Document. Any word can be used, however there is a common convention of using EOF (unless we need to include that word within your Here Document).

Some examples below where you can use Here Document.

Example 1: The following example shows a script that creates a file userlist.txt without waiting for user. When this auto_file.sh is executed, the userlist.txt would contain words from ‘brave’ to ‘charlie’. The lines from ‘<<EOF’ upto next ‘EOF’ are put on the input stream by the shell and given to the command as standard input.

```
#This is content of auto_file.sh
cat > userlist.txt << EOF
bravo
delta
alpha
charlie
EOF
```

Example 2: Download file from ftp server automatically using Here Document. (There is a separate chapter on FTP where these commands would be explained)

```
# This is content of auto_ftp.sh
ftp -ivn 172.24.0.254 <<EOF
quote USER anonymous
quote PASS redhat
cd pub
get test
quit
EOF
```

Example 3: Login to oracle database. Create a report using select query. Spool that report into a file and mail that file to Manager. Learning sqlplus is out of scope for this book. But the people knowing Oracle database would be able to relate.

```
# This is content of report.sh
sqlplus username/password <<EOF
spool report.txt
select e.employee_id,e.last_name,d.department_name,e.salary
FROM employees e join departments d
ON e.department_id=d.department_id;
spool off
EOF
mail -s “Salary Report” manager@focustraining.in < report.txt
```

8.1.8 Redirection Summary

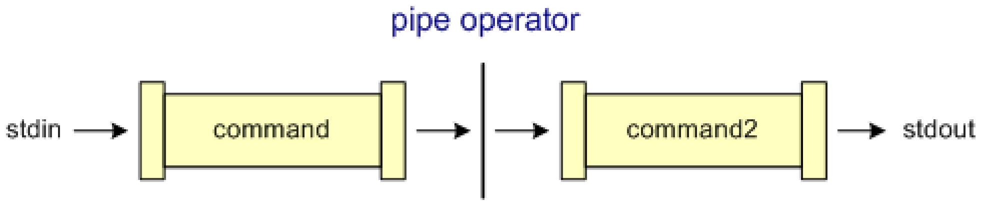
Redirection Operator	Resulting Operation
`command > file`	stdout written to file, overwriting if file exists
`command >> file`	stdout written to file, appending if file exists
`command < file`	input read from file
`command 2> file`	stderr written to file, overwriting if file exsits
`command 2>> file`	stderr written to file, appending if file exists
`command > file 2>&1`	stdout written to file, stderr written to same file descriptor

`Command <<eof`
`---`
`eof`
	The lines after '<<eof' upto 'eof' are considered as input to the command

8.1.9 Pipe Operator

A concept closely related to I/O redirection. The pipe operator is the | character (typically located above the 'Enter' key).

In computer programming, especially in Unix operating systems a pipe is a technique for passing information from one program to another as ddepicted in the following diagram..



A pipe is one-way communication only between two processes. Basically, a pipe passes the output of one process to another process which accepts it as input. The system temporarily holds the piped information until it is read by the receiving process.

Pipes are typically used when communicating between two processes running in the same Uix machine. Pipes usually have a very good throughput.

We can look at an example of pipes using the `who` and the `wc` commands. Recall that the `who` command lists each user logged into a machine, one per line as follows:

```
$ who [Enter]
      mthomas         pts/2    Oct  1   13:07
      fflintstone     pts/12   Oct  1   12:07
      wflintstone     pts/4    Oct  1   13:37
      brubble         pts/6    Oct  1   13:03
```

Also recall that the `wc` command counts characters, words and lines. Thus if we connect the standard output from the `who` command to the standard input of the `wc` (using the -l (ell) option), we can count the number of users on the system:

```
$ who | wc -l [Enter]
4
```

In the first part of this example, each of the four lines from the who command will be "piped" into the `wc` command, where the -l (ell) option will enable the wc command to count the number of lines.

While this example only uses two commands connected through a single pipe operator, many commands can be connected via multiple pipe operators

Command using Pipes	**Meaning or Use of Pipes**
$ who | sort	Output of who command is given as input to sort command so that it will print sorted list of users
$ who | sort > user_list	Same as above except output of sort is sent to (redirected) user_list file
$ who | wc -l	Output of who command is given as input to wc command so that it will output number of user who are logged on to system
$ ls -l | wc -l	Output of ls command is given as input to wc command So that it will print number of files in current directory.
$ who | grep raju	Output of who command is given as input to grep command. so that it will print if particular user name if he is logged in.

Let’s now write a small script to determine how many users are connected to the system. This can be achieved in various ways, we will list some of them, and it is up to you to decide which one you choose.

```
#This is userCount.sh
userCount=`users|wc -w`
echo "Number of users logged in is $userCount"
#Below is another way
userCount=`who | cut -d” “ -f1 | wc -l`
echo “Number of users logged in is $userCount”
#Below is yet another way
userCount=`w| tail -n +3 | cut -d” “ -f1 | wc -l`
echo “Number of users logged in is $userCount”
# In the third example, instead of ‘tail’, you can use ‘more +3’
too but that may fail if users’ list is more than screenful.
```

Report the total and used RAM in MB on the machine.

```
#This is RAMReport.sh
# In the example below, the ‘head -2’ gives us only first two
lines of output of ‘free’. Then ‘tail -1’ gives only one (last)
line of those two lines. That is the line that has our data.
$ total=$(free -m | head -2 | tail -1 | tr -s " " | cut -d" " -
f2)
$ used=$(free -m | head -2 | tail -1 | tr -s " " | cut -d" " -
f4)
$ echo "The machine `hostname` has total of ${total}MB RAM and
out of that ${used}MB is used."
```

It is prudent here to know about a command called `tee`. This command makes copies of the input and sends them to two different locations, one is screen and other is some file. You can use `tee` command at places where you want output of your script to come on screen (for your immediate scrutiny) and you want to keep a record of output in some file. `tee` is capable of creating the file if it does not exist.

```
$ echo “this goes to two places” | tee one.txt
```

By default, `tee` is in overwriting mode. That is, original contents of one.txt would have been overwritten above by ‘this goes to two places’. To put `tee` into append mode, `–a` option.

```
$ echo “this goes to two places” | tee –a one.txt
```

9 Accepting Input

Thre are different ways of accepting input into the script. Each of the ways has its syntax and purpose. We have seen one of them till now. That is using the input redirection operator(<) where the data is taken from a file, but as a stream e.g. `wc −l < myFile`.

In addition to this, there are two more ways.

9.1 The `read` command

The read command can be used to accept data from user, as well as from a file (when provided as stream). Both the forms have their own specific syntax. We see both of them in detail below.

9.1.1 Getting input from user interactively

The `read` command is used to get input from user interactively and store that to variable.

Syntax:

```
read variable 1 variable2  ...variableN
```

Following script first asks the user for his name and then waits for the user to enter name via keyboard. When the user enters name from keyboard (and presses Enter key) stores the entered name into a variable.

```
# This is hello.sh
#
#Script to read your name from key-board
#
echo "Your first name please:"
read fname
echo "Hello $fname, Lets be friends"
```

Run it as follows:

```
$ chmod 755 hello.sh
$ ./read.sh
```

On the screen first you would see the prompt. When you enter a name (vivek in this case) and press Enter, the ‘Hello Vivek…’ line is displayed.

```
Your first name please: vivek
Hello vivek, Lets be friends
```

Instead of using echo command to prompt the user to enter a value and then using read command to store input in a variable, using –p option of read command is more efficient.

```
$ read –p “Enter your name: ” name
```

In the above statement, the user is shown “Enter your name: ” and then whatever he types on keyboard is stored in variable named name.

9.2 Getting input from file (opened as stream)

In the Redirections’ chapter we saw how we can feed contents of a file to a command using the input stream operatior. E.g.

```
$ wc < inputFile
```

Below two examples show how we can use contents of file as input for loops too. We will learn more on loops in the Control Statements’ chapter, but here is a glimps.

In shell scripting it is quite easy to process data from a file. You will find that this is one of the fastest ways to process each line of a file. The first time you see this it looks a little unusual, but it works very well.

```
while read LINE
do
    echo “$LINE”
done < $FILENAME
```

By using the < $FILENAME notation after the done terminator, we feed the while loop from the bottom. When we time each technique for accessing contents of a file for processing, this method will stand out at the top of the list.

The other way of feeding file’s data to a loop is using it in for loop as below. However, here the for loop will consider each word in the file as one input for each iteration (as against one line per iteration in while loop).

```
#This script reads a file word by word
#!/bin/bash
echo "Reading file word by word : "
for var in $(cat tryfile)
do
      echo "Word = $var"
done
```

10 Exit Status

The `$?` variable represents the exit status of the previous command. Remember that it is always the exit status of IMMEDIATELY preceeding command.

Exit status is a numerical value returned by every command upon its completion. As a rule, most commands return an exit status of 0(zero) if they were successful, and other than 0 (most of the time 1) if they were unsuccessful. Unsuccessful in this context does not necessarily mean error. E.g. `grep` returns a non-zero status if it could not find the specified word in the file.

Following is the example of successful command:

```
$  ls
a.txt  case.sh  for.sh  hello.sh  hello.txt  if.sh
$  echo $?
0
```

Following is example of unsucessful command

```
$  cat abcd
cat: abcd: No such file or directory
$ echo $?
1
```

Following is an example of changing exit status depending on what the command could do. In the first grep command, we are searching an existing user in existing file. It gives zero exit status all right. In the second command however, we are trying to find a nonexisting user in the file. This results in exit code of 1. In the third instance, we are trying to find something in a file that does not exist. In this case grep exits with yet another code. In a shell script, the shell programmer can tactfully make use of such different codes to determine what happened to earlier command.

```
$ grep root /etc/passwd
root:x:0:0:root:/root:/bin/bash
$ echo $?
0
$ grep nonexistant /etc/passwd
$ echo $?
1
$ grep nonexistant fileDoesNotExist
grep: fileDoesNotExist: No such file or directory
$ echo $?
2
$ echo $? #This shows zero, earlier (echo) command was success.
0
```

11 Control Statements

While writing a shell script, there may be a situation when you need to adopt one path out of the available two paths. So you need to make use of conditional statement that allows your program to make correct decisions and perform right actions.

Unix Shell supports conditional statements which are used to perform different actions based on different conditions. Here we will see following two decision making statements:

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

11.1 Arithmetic Operators

Normal Arithmetical/ Mathematical Statements	Meaning	Mathematical Operator in Shell Script	Usage
5 == 6	is equal to	`-eq`	`if [ 5 -eq  6  ]`
5 != 6	is not equal to	`-ne`	`if [ 5 -ne  6  ]`
5 < 6	is less than	`-lt`	`if [ 5 -lt  6  ]`
5 <= 6	is less than or equal to	`-le`	`if [ 5 -le  6  ]`
5 > 6	is greater than	`-gt`	`if [ 5 -gt  6  ]`
5 >= 6	is greater than or equal to	`-ge`	`if [ 5 -ge  6  ]`

11.2 String Operators

There are following string operators supported by Bourne Shell.

For the table below, assume variable ‘a’ holds "abc" and variable ‘b’ holds "efg". Once you are gone through next section of if-else statements, try these string operators on different types of variables like –

a=”” # this is empty string

b=”   “ # this is blank string

c=”abc” # this is normal string

d= # the variable ‘d’ has nothing!

And when you are at that, surround the $variable in double quotes i.e. [“$a” = “$b”].

Operator	Description	Example
=	Checks if the values of two operands are equal	[$a = $b] is not true.

	or not, if yes then condition becomes true.	
`!=`	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	[$a != $b] is true.
`-z`	Checks if the given string operand size is zero. If it is zero length then it returns true.	[-z $a] is not true.
`-n`	Checks if the given string operand size is not null. If it is non-zero length then it returns true.	[-n $a] is true.
`str`	Check if str is not the empty string. If it is empty then it returns false.	[$a] is not false.

11.3 File test operators

Assume that a variable `file` holds name of a file a.txt and a.txt has read, write and executable permissions for the current user. It has couple of lines in it.

Operator	Discription	Example
`-r $file`	Checks if file is readable for the current user. If yes then condition becomes true.	[-r $file] is true.
`-w $file`	Check if file is writable for the current user. If yes then condition becomes true.	[-w $file] is true.
`-x $file`	Check if file is executable for the current user. If yes then condition becomes true.	[-x $file] is true.
`-s $file`	Check if file has size greater than 0. If yes then condition becomes true.	[-s $file] is true.
`-e $file`	Check if file exists. Is true even if file is a directory but exists.	[-e $file] is true.
`-f $file`	Check if file is an ordinary file as opposed to a directory or special file if yes then condition becomes true.	[-f $file] is true.

11.4 The if...else statements

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

Unix Shell supports following forms of if..else statement:

11.4.1 if...fi statement

The **if...fi** statement is the fundamental control statement that allows Shell to make decisions and execute statements conditionally.

Syntax:

```
if [ expression ]
then
   Statement(s) to be executed if expression is true
fi
```

Here Shell *expression* is evaluated. If the resulting value is *true*, given *statement(s)* are executed. If *expression* is *false* then no statement would be executed. Most of the times you will use comparison operators while making decisions.

Pay attention to the spaces between square brackets and expression. This space is mandatory otherwise you would get syntax error.

If *expression* is a shell command then it would be assumed true if it returns 0 after its execution. If it is a boolean expression then it would be true if it returns true.

Example 1:

```
#!/bin/sh
a=10
b=20
if [ $a -eq $b ]
then
   echo "a is equal to b"
fi

if [ $a -ne $b ]
then
   echo "a is not equal to b"
fi
```

This will produce following result:

```
a is not equal to b
```

Example 2: Send an email to a concerned person if there are more than 5 users logged on to the system.

```
#!/bin/bash
no_of_users_logged_in=`who | wc -l`
if [ $no_of_users_logged_in -gt 5 ]; then
     subject='High System Load'
     who > /tmp/list_of_users.txt
      mail -s ${subject} stuser20@sql.example.com
</tmp/list_of_users.txt
fi
rm -f /tmp/list_of_users.txt
```

11.4.2 if...else...fi statement

The if...else...fi statement is the next form of control statement that allows Shell to execute statements in more controlled way and making decision between two choices.

Syntax:

```
if [ expression ]
then
   Statement(s) to be executed if expression is true
else
   Statement(s) to be executed if expression is not true
fi
```

Here Shell *expression* is evaluated. If the resulting value is *true*, given *statement(s)* are executed. If *expression* is *false* then statements in the `else` part are executed (from `else`' upto `fi`).

Example 1: If we take above example then it can be written in better way using if...else statement as follows:

```
#!/bin/sh
a=10
b=20
if [ $a -eq $b ]
```

```
then
   echo "a is equal to b"
else
   echo "a is not equal to b"
fi
```

This will produce following result:

```
a is not equal to b
```

Example 2: The example below checks whether file exists

```
#!/bin/bash
#script to check whether directory exists or not.
read –p “Enter directory name: ‟ a
if [ -d “$a” ]; then
      echo " directory $a exists "
else
      echo " directory $a does not exists "
fi
```

Example 3:

```
#!/bin/bash
# Script to see whether value is positive or negative
Read –p “Enter a number:” a
if [ $a -gt 0 ]
then
     echo "$a number is positive"
else
     echo "$a number is zero or negative"
fi
```

Example 4: The example below accepts two strings from user and checks whether are equal or not

```
#!/bin/bash
echo Enter the strings(string1 string2)
read str1
read str2
if [ “$str1” = “$str2” ]
then
      echo "Both Strings are equal"
else
     echo "Given strings are not equal"
fi
```

Example 5: Write a shell script to take a bakup of project directoy. Write a script in such a way that only root user can take backup and if the same script is run by normal user the error message should be reported. (More about the `tar` command in a separate chapter on Archiving).

```
#!/bin/bash
user=`whoami`
if [ “$user” = “root” ]
then

      tar -cf /backup/project.tar /project &>/dev/null
      if [ $? -eq 0 ]
      then
            echo Backup taken successfully...
      else
            echo Error in taking backup file
      fi
else
      echo "You are not authorized to run this script"
```

```
fi
```

Example 6: Write a shell Script to pass username to shell script and check whether that user exists in your system

```
#!/bin/bash
echo "Enter user name"
read usr
cat /etc/passwd|grep -wo ^$usr &>/dev/null
if [ $? -eq 0 ]
then
      echo "$usr exists"
else
      echo "$usr does not exist"
fi
```

Example 7: Write a shell script to send database name as command line argument to shell script and check whether that database is running or not

```
# This is db_status.sh
#!/bin/bash
read -p “Enter database name: ” db
export ORACLE_SID=$db
sqlplus  / as sysdba<<EOF  &>/dev/null
spool /tmp/status.txt
select sysdate from dual;
spool off
EOF
status=`grep -c ORA /tmp/status.txt`
#If the database specified by $db was down, the ‘select’
statement would have generated error. That error would have been
put in status.txt because of spooling. Oracle’s error codes
start with ORA. We are finding count of word ‘ORA’ in status.txt
```

```
to determine whether there was an error reported.
if [ $status -gt 0 ]
then
    echo “$db is down”
else
    echo “$db is running”
fi
```

Example 8: Write a shell script to display message to user whether the databse listner is available or not.

```
# This is db_listener_status.sh
#!/bin/bash
lsnrctl status &>/dev/null
if [ $? -eq 0 ]
then
    echo “Listener is running”
else
    echo “Listener id down”
fi
```

11.4.3 if...elif...else...fi statement

```
if [ expression1 ]

then

   Statement(s) to be executed if expression1 is true

elif [ expression2 ]

then

   Statement(s) to be executed if expression2 is true

elif [ expression3 ]

then
```

```
   Statement(s) to be executed if expression3 is true

else

   Statement(s) to be executed if no expression is true
fi
```

There is nothing special about this code. It is just a series of if statements, where each `if` is part of the `else` clause of the previous statement. Here statement(s) are executed based on the true condition, if none of the conditions is true then the final `else` block is executed.

Example 1:

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

This will produce following result:

```
a is less than b
```

Note that in if..else or if..elif..else construct, only one of the blocks can get executed for a given condition. They are mutually exclusive. So, use them in places where only one condition can be true at a time. E.g. given name can be either a file, or a directory or a link. It can not be all at the same time.

On the contrary, when mutlitple conditions can be true irrespective each other, use separate blocks of `if`. E.g. a file can be readable and can be writable and can be executable or any combination of these at the same time. So while checking permissions of a file one by one, use multiple `if` blocks.

Example 2:

```
#!/bin/sh
# Script to test if..elif...else
Read –p “Enter a number :” a
if [ $a -gt 0 ]; then
   echo "$a is positive"
elif [ $a -lt 0 ]
then
      echo "$a is negative"
else
   echo "$1 is zero"
fi
```

11.5 Combining conditions to produce one result

11.5.1 -a and -o operators

There would be cases where you want to test multiple conditions simultaneously and then decide the course of your script depending on combined result of those tests. The combination that you are looking for can be of two types - both the conditions be true or any one of the condition be true.

To fulfil this need, bash provides two more operators, namely `–a` (for AND) and `–o` (for OR) for getting a combined result of two tests.

As you may already know, the `–a` operator means BOTH the conditions must be true for the entire expression to be true. The `–o` operator means AT LEAST one of the conditions must be true for the entire expression to be true.

`[ $a -gt 5 -a $a -lt 8 ]` # the entire expression is true only if `a` is greater than 5 AND less than 8. When `a` is 6 or 7 then this condition is true. For all other values, at least one of the conditions would be false and entire expression would be false.

`[ $a -lt 5 -o $a -gt 8 ]` # the entire expression is true if `a` is less than 5 OR greater than 8. When `a` is 5, 6, 7 or 8 then both the conditions are false and hence the entire expression would be false.

11.6 Negating result of a test

11.6.1 The `!` operator

Sometimes you may want to reverse the result of a test. In that case the negation operator is useful. Below are two examples describing how a negative condition can be interpreted. Below that are three examples how the operator can be used.

```
[ -z "$a" ]
```

Above expression is true when value of a is of zero length. Whereas

```
[ ! -z "$a" ]
```

above expression is false when value of a is of zero length (In other words, this conditon is true when value of a is NOT of zero length)

```
[ -f "abc" ]
```

Above expression is true when a file named "abc" exists. Whereas

```
[ ! -f "abc" ]
```

above expression is false when a file named "abc" exists (In other words, this expression is true when "abc" is NOT a file)

`[ ! $a -gt 3 ]` # expression is true when value of variable a is NOT greater than three

`[ "$b" != "$c" ]` # expression is true when values of variables b and c are NOT same

`[ -z "$d" -o ! -f "$d" ]` # this expression is true if value of d is zero length or it is not a file (If value of d was accepted from a user of your script, at this point you probably will want to warn him that he did not supply a name, or the supplied name is not a file).

Note that the opening square bracket and the corresponding closing square bracket used in `if` statement are actually shortcut for a command named `test`. So, if you want to learn more about these arithmetic, string and file operators and other operators that are availble in these catagories, check the man-page of `test` command. Also, below two statements are equivalent –

```
if [ 5 –eq 6]; then ... ……
if test 5 –eq 6 ; then… … …
```

11.7 Looping Statements

Loops are a powerful programming tool that enables you to execute a set of commands repeatedly. In this section, you would examine the following types of loops available to shell programmers:

- The while loop
- The for loop

11.7.1 The `while` loop

You would use different loops based on dfferent situation. For example `while` loop would execute given commands till the given condition remains true.

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

```
while condition-command

  do

      command1
      command2

      ...

  done
```

The example below loops over two statements as long as the variable i is less than or equal to ten. Store the following code in a file named while1.sh and execute it.

Example 1:

```
#!/bin/bash
#Illustrates implementing a counter with a while loop
#Notice how we increment the counter with expr in backquotes
          i=1
          while [ $i -le 10 ]
          do
                     echo "i is $i"
                     i=`expr $i + 1`
          done
```

Example 2: Lock the user accounts whoes uid is between the range of 500 to 510

```
#!/bin/bash
while read line
do
      uname=`echo $line|cut -d":" -f1`
      id=`echo $line|cut -d":" -f3`
      if [ $id -ge 500 -a $id -le 510 ]
          usermod -L $uname &>/dev/null
          echo "User $uname Locked...."
      fi
done</etc/passwd
```

Example 3: Change the shell of users to csh whoes id is between the range of 500 to 520 and login shell is bash.

```
#!/bin/bash
while read line
do
      uname=`echo $line|cut -d":" -f1`
      id=`echo $line|cut -d":" -f3`
      shell=`echo $line|cut -d":" -f7`
      if [ $id -ge 500 -a $id -le 520 -a $shell = “/bin/bash” ]
          usermod -s /bin/csh &>/dev/null
          echo "Shell of $uname changed to /bin/csh"
      fi
done</etc/passwd
```

11.7.2 The `for` loop

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

Syntax:

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

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

Here is a simple example that uses for loop to span through the given list of numbers:

Example 1:

```
#!/bin/sh
for var in 0 1 2 3 4 5 6 7 8 9
do
   echo $var
done
```

This will produce following result:

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

Example 2:

```
#!/bin/bash
a=$(seq 1 1 5) # seq command generates sequence of numbers
for i in $a
do
    echo "Value of i = $i"
done
```

Following is the example to display all the files starting with .bash and available in your home. I'm executing this script as root.

Example 3:

```
#!/bin/sh
for FILE in $HOME/.bash*
do
   echo $FILE
done
```

This will produce following result when executed by root. Root user’s home directory is /root.

```
/root/.bash_history
/root/.bash_logout
/root/.bash_profile
/root/.bashrc
```

Example 4:

```
#!/bin/bash
read -p “Enter first three parts of IP address e.g. 172.24.0” ip
for i in {1..10} # as good as space-separated list of 1 to 10
do
    ping -c1  $ip.$i   > /dev/null 2>&1
    if [ $? -eq 0 ]; then
        echo "Node ${ip}.$i is up"
    else
         echo "Node ${ip}.$i is down"
    fi
done
```

The above script checks if that a machine is up or down, using the ping command and using its Exit status.

The loop runs on a sequence of range of IP addresses to check which machine on the network is up. The first three sections of the IP address are taken as input from the user. Inside the for loop, the IP address is built by appending host number to the network part of the IP address (${ip}.$i).

Once the full ip address is built as above, the machine is pinged once (-c1 makes sure it pings only once). The output generated by the command is discarded into /dev/null/. The script then checks for the exit status of the ping command. If $? (exit status) is equal to zero then the machine is up else it is down.

Example 5: Below script takes back up of each of the folders in the /home directory on a machine. The command basename captures the last string in the argument, thereby giving names of the users' home directories. This name bname) is then used to build the name of the tar file.

```
#!/bin/bash
for i in $(ls /home)
do
    bname=$(basename ${i})
    echo "backing up $i...."
    tar -cf /backup/$bname.tar /home/$i > /dev/null 2>&1
done
echo "backup done"
```

11.7.3 The infinite loop

All the loops have a limited life and they come out once the condition is false or true depending on the type of loop (for or while).

A loop may continue forever due to required condition is not met. Such a loop executes an infinite number of times. For this reason, such loops are called infinite loops.

Example:

Here is a simple example that uses the value of “a” as condition.

```
#!/bin/sh
a=20
while [ $a -gt 10 ]
do
   echo $a
done
```

This loop would continue forever because `a` is alway greater than 10 and it would never become less than 10.

11.8 `break` and `continue` statements

So far you have looked at creating and working with loops to accomplish different tasks. Sometimes you need to stop a loop or skip iterations of the loop.

Following two statements are used to control loops:

1. The `break` statement
2. The `continue` statement

11.8.1 The `break` statement

The `break` statement is used to terminate the execution of the loop after completing the execution of all of the lines of code up to the break statement. Execution then continues from the code following the end of the loop.

Syntax:

The following `break` statement would be used to come out of a loop:

```
break
```

Example 1:

Here is a simple example which shows that loop would terminate as soon as ‘a’ becomes 5:

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

This will produce following result:

```
0
1
2
3
4
5
```

Example 2:

```
#!/bin/bash
read “Enter a filename: ‟ file
while [ 1 = 1 ]
do
        if [ -f $file ]; then
                break
           else
                sleep 1
                 echo "file doesnt exist yet !!"
        fi
done
         echo " file is present in your directory "
```

The above script is in infinite loop, waiting for some file to appear/arrive. The file name is supplied by the user after he is prompted. If the file is present, the loop breaks. If the file doesn't exist, the script sleeps for a second, prints a statement and again checks for the existence of the file. The loop breaks as soon as file appears.

11.8.2 The `continue` statement

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

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

Syntax:

```
continue
```

Example:

The following loop makes use of `continue` statement which aborts the current iteration and starts processing next iteration. It skips those statements in the loop which are after `continue`:

```
#!/bin/sh
NUMS="1 2 3 4 5 6 7"
for NUM in $NUMS
do
   Q=`expr $NUM % 2`
   if [ $Q -eq 0 ]
   then
      echo "Number is an even number!!"
      continue
   fi
   echo "Found odd number"
done
```

This will produce following result:

```
Found odd number
Number is an even number!!
```

```
Found odd number
Number is an even number!!
Found odd number
Number is an even number!!
Found odd number
```

12 Command Line Arguments & Special Variables

We can accept user input by using the `read` command that waits to take value from user. There is another way to send an input to shell script called as Command Line Argument.

In the example below, the name of the shell program is my_script, and the first positional parameter (named 1) gets the value as arg1 within the script, the second variable gets the value as arg2, etc.

```
             1    2    3
             ↓    ↓    ↓
$ my_script arg1 arg2 arg3
```

Thus, if we wish to see the value stored in the first postional parameter, we could do the following from within the my_script program (note that this only works from within the my_script program):

```
$ echo $1
```

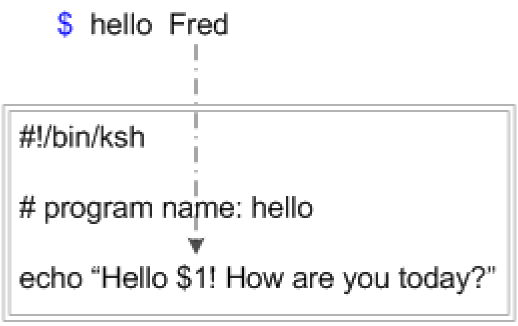
Positional parameters provide the programmer with a powerful way to "pass data into" a shell program while allowing the data to vary. If we had a shell program named hello that contained the following statement:

```
$ echo Hello Fred! How are you today?
```

This would not be very interesting to run, unless perhaps your name was Fred. However if the program was modified like this:

```
$ echo "Hello $1! How are you today?"
```

This would allow us to pass single data values "into" the program via positional parameters as illustrated in the following diagram:



```
$ hello Fred
```

```
#!/bin/ksh

# program name: hello

echo "Hello $1! How are you today?"
```

We could then run the program as follows, using varying values to pass into the positional variable $1.

```
 $ hello Fred [Enter]
   Hello Fred! How are you today?
 $ hello Barney [Enter]
   Hello Barney! How are you today?
```

It should be obvious that this would be a much more useful program. Keep in mind that many behaviors of standard variables are also behaviors of positional variables. For example, if you did not assign a value to the first positional variable, you would not get an error, rather a behavior as follows:

```
 $ hello [Enter]
  Hello! How are you today?
```

Similarly, if there are more command line arguments than positional variables, the extra arguments are simply ignored, for example:

```
$ hello Fred Barney Dino [Enter]
  Hello Fred! How are you today?
```

Special Parameters $* and $@:

There are special parameters that allow accessing all of the command-line arguments at once. $* and $@ both will act the same unless they are enclosed in double quotes, "".

Both the parameters specify all command-line arguments but the "$*" special parameter takes the entire list as one argument with spaces between and the "$@" special parameter takes the entire list and separates it into individual arguments.

The following table shows a number of special variables that you can use in your shell scripts:

Variable	Description
$0	The filename of the current script.
$n	These variables correspond to the arguments with which a script was invoked. Here n is a positive decimal number corresponding to the position of an argument (the first argument is $1, the second argument is $2, and so on).
$#	The count of arguments supplied to a script.
$*	All the arguments are double quoted. If a script receives two arguments, $* is equivalent to $1 $2.
$@	All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2.
$?	The exit status of the last command executed.
$$	The process number of the current shell. For shell scripts, this is the process ID under which they are executing.
$!	The process number of the last background command.

Why Command Line arguments are required

1. Telling the command/utility which option to use.

2. Informing the utility/command which file or group of files to process (reading/writing of files).

```
$ myshell foo bar
```

Shell Script name here is myshell. First command line argument passed to myshell is foo. Second command line argument passed to myshell is bar.

Here $# (built in shell variable) will be 2 (Since foo and bar are two arguments). Please note that at a time such 9 arguments can be used from $1..$9, You can also refer all of them by using $* (which expand to`$1,$2...$9`). Note that $1..$9 i.e command line arguments to shell script are also know as "positional parameters".

Following script is used to print command ling argument and will show you how to access them:

```
$ vi demo
```

```
#!/bin/bash
# Script that demos, command line args
echo "Total number of command line argument are $#"
echo "$0 is script name"
echo "$1 is first argument"
echo "$2 is second argument"
echo "All of them are :- $* or $@"
```

What if you have more than 9 arguments to be passed to your program? Something like

```
$ my_script jan feb mar apr may jun jul aug sep oct nov dec
```

In this case, inside the my_script program you can access only upto $9, that is ‘sep’. If you try to access $10, you get jan0. This is because, the shell simply doesn’t understand $10. $10 is interpreted as $1 followed by a zero. Similarly $11 is interpreted as $1 followed by 1, resulting in jan1.

But there has to be a way to access arguments beyond 9th position. This is achieved by a command shift. This command shifts the positions of the arguments to left. i.e. in the above example, if we use the shift command, the argument oct will be accessible at $9. shift takes a numeric argument. Instead of only shift, if we had said shift 3, then dec would come in $9, nov would be in $8 and oct would be in $7. Consequently, apr would be in $1. What happened to jan, feb and mar then? They are lost… And if you observe keenly, $# itself changes.

```
#!/bin/bash
# this is my_script
echo "Total number of command line arguments is $#"
echo "$8 is 8th arg before shift" #shows aug
echo "$9 is 9th arg before shift" #shows sep
shift 2
echo "Total number of command line arguments after shift $#"
echo "$8 is 8th arg after shift" #shows oct
echo "$9 is 9th arg after shift" #shows nov
```

If you, for some reason have to set command line arguments from within the schell script, you can do so by using a command named `set`.

Write a script as below and execute it. You will see that prior to `set` command the positional arguments do not have values, but they gain values after the `set` command.

```
#!/bin/bash
# demo for setting the CLAs
echo "Total number of command line arguments is $#" # shows zero
echo "$1 is 1st arg before setting" #shows blank
echo "$2 is 2nd arg before setting" #shows blank
set summer spring monsoon autumn fall winter
echo "Number of CLAs after setting $#" # shows six
echo "$3 is 3rd arg after set" #shows monsoon
echo "$2 is 2nd arg after set" #shows spring
```

13 Shebang

You must have observed a line starting with '#!' in many shell scripts. It is called shebang. What follows the '#!' sequence is path of an executable. In our scripts it would be #!/bin/bash. That is, the shebang is pointing to the executable for bash shell.

When a script is given execute permissions and executed ate command prompt, the default shell executing the script looks for the '#!' pattern. If the shebang is found, it invokes the executable mentioned in the shebang and let that executable (generally an interpreter shell) execute the script. If no '#!' is found then default shell itself will execute the script.

Since the shells are installed in different locations on different systems you may have to alter the '#!' line. For example, the bash shell may be in /bin/bash, /usr/bin/bash or /usr/local/bin/bash.

Setting the interpreter explicitly like this ensures that the script will be executed by the stated interpreter regardless of who starts it (or what the default shell may be). This is important because shells have variations in syntaxes. Script written with one shell in mind may not work in another shell.

Shebang takes effect only if your script is executed as an executable rather than as below. In below case, the calling shell ('sh') will consider the shebang as a comment.

```
$ sh hello.sh [Enter]
```

Also remember that, the shebang is treated as shebang only when it is first line of the script. Otherwise, it is a plain comment starting with '#'.

The comments in the blow script explain where the shebang takes effect and where don't. And why shebang is necessary for scripts which are supposed to be executed on various machines.

```
#!/bin/csh

# Below script has syntax which is very specific to C-Shell.
# That means, if this script is executed using commands like
below, it will result in errors. This file is cShellSyntax.sh
# $>sh cShellSyntax.sh
# $>bash cShellSyntax.sh

# The reason for errors is that, the script has syntax of C-
shell and we are using Shell (sh) and Bourne Again Shell (bash)
to execute it.
# Since the 'sh' and 'bash' do not understand this syntax, they
cannot interpret the script correctly and report error.
```

```
# To make sure that the script is executed correctly, we have
two options -
# 1.
# Execute the script using CShell (csh). Like -
# $ csh cShellSyntax.sh
# OR
#2.
# Give execute permission to the script (chmod u+x
cShellSyntax.sh) and then execute the script like -
# $>./cShellSyntax.sh

# When the second option of script execution is used, the
default shell starts reading the cShellSyntax.sh from the top
line.
# From the first line, the Shebang, the defult shell infers that
it should invoke the shell mentioned in the shebang and let THAT
shell execute the remaining code.
# In this case, the shebang is specifying the c-shell. That way,
the the default shell invokes c-shell and allows it to execute
the code. That way, it is always the c-shell that would be
executing the code. And hence the c-shell specific syntax is
always executed successfully irrespective the default shell.
# That is the reason, most of the scripts would always start
with a shebang, indicating which shell the code in the script
adheres to.

echo " The home directory is $home. " #A variable 'home' (all
lower case) is understood by c-shell, but not by bash or sh.
bash or sh understand HOME.

echo " The number of command line arguments is $#argv" # In bash
and sh, the number of CLAs is shown by $#
date>/dev/null
echo "The exit status of last command is $status" # The bash and
sh use $?

set i=6 # In bash or sh we simply say i=6. But in csh, we need
to use 'set' for setting values of variable.
if ($i == 5) then   # The 'then' is needed on same line. If
'then' is not be used, the first line in if-block should start
on this line.
    echo "equal this is "
else
    echo "not equal"
endif # There is no 'fi' in cshell. 'fi' is for bash and sh.
```

14 Functions

Functions enable you to break down the overall functionality of a script into smaller, logical sections, which can then be called upon to perform their individual tasks when needed.

Using functions to perform repetitive tasks is an excellent way to reuse code. Code reuse is an important part of modern object-oriented programming principles.

Shell functions are similar to subroutines, procedures, methods or functions in other programming languages.

14.1 Creating Functions

To declare a function, simply use the following syntax:

```
Function fname () {
   list of commands
}
```

The name of above function is `f_name`, and that's what you will use to call it from elsewhere in your scripts. The function name must be followed by parentheses, which are followed by a list of commands enclosed within curly braces (this is body of the function). If you use keyword “function” before function, then you can omit parentheses.

Example:

Following is the simple example of using function:

```
#!/bin/sh
# Define your function here
Hello () {
   echo "Hello World"
}
# Invoke your function
Hello
```

When you would execute above script it would produce following result:

```
$ sh ./test.sh
Hello World
```

14.2 Pass Parameters to a Function

You can pass parameters to a function when calling them. These parameters would be represented by $1, $2 and so on in the function's body.

Following is an example where we pass two parameters Zara and Ali and then we capture and print these parameters in the function.

```
#!/bin/sh
# Define your function here
Hello () {
   echo "Hello World $1 $2"
}
# Invoke your function
Hello Zara Ali
```

This would produce following result:

```
$ sh ./test.sh
Hello World Zara Ali
$
```

14.3 Returning Values from Functions

If you execute an exit command from inside a function, its effect is not only to terminate execution of the function but also of the shell program that called the function.

If you instead want to just terminate execution of the function, then there is a way to come out of a function. It is the `return` statement.

Based on the situation, you can return any value from your function using the **return** command whose syntax is as follows:

```
return code
```

Here `code` can be anything you choose, but obviously you should choose something that is meaningful or useful in the context of your script as a whole. `code` must be numeric, between 0 and 255. It should more be used for indicating whether the function could do its job completely or not (like exit status of a command).

Example:

Following function returns a value 10:

```
#!/bin/sh
# Define your function here
Hello () {
   echo "Hello World $1 $2"
   return 10
}
# Invoke your function
Hello Zara Ali
# Capture value returnd by last command
echo "Return value is $?"
```

This would produce following result.

```
$ sh ./test.sh
Hello World Zara Ali
Return value is 10
$
```

14.4 Function Call from Prompt:

You can put definitions for commonly used functions inside your .bash_profile file so that they'll be available whenever you log in and you can use them at command prompt.

Alternatively, you can group the definitions in a file, say test.sh, and then source the file in the current shell by typing:

```
$ . test.sh
```

This has the effect of causing any functions defined inside test.sh to be defined in the current shell. Supposing that the test.sh had a function named number_one which had two echo statements, you can call the function as follows after sourcing test.sh in current shell:

```
$ number_one [Enter]
This is the first line speaking...
This is now the second line speaking...
$
```

To remove the definition of a function from the shell, you use the `unset` command with the `–f` option. This is the same command you use to remove the definition of a variable to the shell.

```
$ unset -f function_name
```

15 File Transfer Protocol (FTP)

Many a times a person working on Linux systems would require to move files from one machine to another. This can be achieved using FTP, the file transfer protocol. Other way of doing it is to use `scp` command, but we will see it in chapter on password-less entry to other machines.

In the FTP scenario, there is a FTP server running on some machine. And to interact with that server there has to be a FTP client. In our examples below, we are assuming ourselves to be client, having FTP client installed on our machines. And there is a server somewhere on the network, accessible to us and the administrator of that machine has granted permissions to us to login, get and put files from/at that server. The server admin at remote server has specified in which directory we will land in when we login. The IP address of the server is something like xxx.xxx.8.40, the loginid is named mirza and the password is 1234.

15.1 Logging into FTP server

With all this information at hand, we are ready to transfer files between server and our machine.

```
$ ftp xxx.xxx.8.40
```

This would take you to ftp prompt. The ftp prompt is an indication that now the program that would listen to your commands is the FTP program. Assuming that xxx.xxx.8.40 machine was up and connection was established, we will need to provide userid and password for logging in.

```
ftp> user mirza 1234
```

Above command would log us in into ftp server with mirza as user and 1234 as his password. You will be back to the ftp prompt. But now you are connected to server with successful login. At this prompt now you can fire commands.

```
ftp> pwd
```

Above command would tell you name of the present working directory on the remote machine (server).

```
ftp> !pwd
```

The ‘!’ in the beginning tells the ftp prompt to invoke local machine’s (client’s) shell and fire the command on it. Consquently, ‘!pwd’ shows you present working directory on your machine.

The commands that we can fire at the ftp prompt are those which are understood by the ftp client. The 'pwd', 'cd' etc commands that we fire on ftp prompt look and behave very similar to Unix commands with same name, but these are being executed by ftp client, not by the shell.

15.2 Getting and putting files

```
ftp> get a.txt
```

The `get` command gets files from server to local machine one at a time. It is assumed that a file a.txt is in present working directory of the server machine and it will be transferred to present working directory (pwd) of the client machine. The FTP would show you how much data was transferred and in how much time etc.

```
ftp> put y.sh
```

The `put` command would transfer the file from local machine's pwd to remote machine's pwd.

```
ftp> cd /var/ftp/pub
```

The `cd` command (which gets fired on server) would change directory to /var/ftp/pub.

```
ftp> mget c.txt d.sh c.java
```

The `mget` command is used to get multiple files from the server in one go. The ftp will ask for each of the files whether you want to get the file.

```
ftp> mput a.txt b.txt *.sh
```

The `mput` command is used to put multiple files on the server. The ftp will ask you for each file whether you want to transfer it.

15.3 Types of transfer

The data in the file can be either characters (text files, shell scripts) or binary (tar files, executables etc). FTP client uses two different modes for transferring different types of files. You can use '`ascii`' command to switch to ascii mode (for character based files) or '`binary`' for switching to binary mode (for binary files). By default, when ftp client starts, it is in binary mode.

15.4 Automating FTP using shell script

With this knowledge at our disposal, let's try and write some shell scripts which would automate some ftp tasks. Before that we should know that the ftp client command by default tries to connect to the server using the current user’s id. To suppress that behavior, we should use −n option so that we ourselves can supply a user id. Also, to suppress the behavior of asking question for every file during mget/mput commands, use −i.

Let’s write first shell script to ftp some files to server automatically.

```
ftp -ni xxx.xxx.8.40 <<EOF
user mirza 1234
cd /var/ftp/pub
mput cities*
mget mirza.*
bye
EOF
```

Now here, the things are very much hardcoded. The IP address, the userid, the password are all hardcoded. This makes script of limited use for a specified machine and specific user.

Let’s modify it to accept the IP, user and passwd from command line so that they can be specified at the time of execution. This lets us use the same script for any machine and any user. But when it comes to user input, there have to be a lot of validations. Let’s add those to script. And yes, if the validation fails, we will have to show some error message too. And then again, for establishing an FTP connection to the FTP server, firstly the machine on which FTP server is running has to be up.

```
function showHelp() {
    echo "Usage: `basename $0` ip userid passwd"
    echo "$1"
    exit $2
}

if [ $# -ne 3 ]; then
    showHelp “You supplied only $# arguments” 25
fi

ping -qc1 $1 &> /dev/null
```

```
if [ $? -ne 0 ]; then
    showHelp "The IP Address $1 is not accessible. Will not
attempt FTP" 26
fi

ftp -ni $1  <<EOF
user $2 $3
cd /var/ftp/pub
mput cities*
mget mirza.*
bye
EOF
```

But the names of the files being transferred are still hardcoded. Now if we be more creative, we can actually get input from a file - A file which lists ip, user, passwd, destination directory and a list of files to be put. Below two codes would do it. The first is the input file and the other is the script which reads this file and does the actual file transfer. cities, cities1, cities2, cities3 and cities4 are the names of the files to be put. –f specifies that whatever comes after it is a space-separated-list of files to be put on server.

The second input line generates error. Read the script and determine the reason.

```
xxx.xxx.8.40 mirza 1234 /var/ftp/pub -f cities cities1
xxx.xxx.8.40 1234 /var/ftp/pub cities2 cities3
xxx.xxx.8.40 rahul s0mething /var/ftp/pub -f cities4
```

If above contents are in a file named inputFile, and name of the below script is auto_ftp.sh, it is to be invoked as

$ sh auto_ftp.sh inputFile

```
function showHelp() {
    echo "Usage: `basename $0` inputFile"
    echo "$1"
    exit $2
}

function showRequestFormat () {
    echo "The request must be a space-separated line with following columns in that order."
    echo "ip userid password destinationDirectory -f filesToBeUploadedSeparatedBySpaces"
    echo "Sequence of these fileds must be maintained. The '-f' must be specified before list of files."
    echo -e "The cause of error is -\n$1"
}

if [ $# -ne 1 ]; then
    showHelp "The input file is not specified." 25
fi

if [ ! -f $1 ]; then
    showHelp "The specified input file does not exist." 26
fi
while read line
do
    unset ip user password destDir fileIndicator files
    ip=`echo $line | cut -d" " -f1`
    user=`echo $line | cut -d" " -f2`
    password=`echo $line | cut -d" " -f3`
    destDir=`echo $line | cut -d" " -f4`
    fileIndicator=`echo $line | cut -d" " -f5`
    files=`echo $line | cut -d" " -f6-`

    if [ "$fileIndicator" != "-f" -o -z "$files"  ]; then
```

```
        showRequestFormat "This line has error-\n $line"
        continue
    fi

    ping -qc1 $ip &> /dev/null
    if [ $? -ne 0 ]; then
        showRequestFormat "The server is inaccessible. Won’t
attempt FTP for this request: $line."
        continue
    fi
    ftp -ni $ip  <<EOF > /dev/null
    user $user $password
    cd "$destDir"
    mput $files
    close
    bye
EOF
done < "$1"
```

Ponder a little at this point. The core funcationality of the script still remain in last few lines of it. Most of the code is to handle user input – its acceptance, its validation, error handling and error reporting. Generally this would be the case with any script. More lines would be for validations and error handling than the core functionality.

One more thing to note is, we should put ample amount of comments in our scripts. Above script has no comment at all. But a real-life script should explain the functionality of the script, at least at the complex lines.

16 Sending and receiving mails

If the mailing utility is installed on the machine it is possible to send mails to other users on the same machine. If there are gateways provided and the firewall alows, then mails can be sent outside of the network too, but let’s restrict ourselves to sending mails to the users on the same machine.

16.1 Sending mail

We will start with sending mail.

```
$ mail -s “this is subject line” -a a.txt -a b.txt
rahul,amol,milind [Enter]
This is body of the mail.
Second line of the body.
Third line
Regards,
Signature
Ctrl+d ← end the input by pressing this key-combination
```

The `mail` command gets the body of the mail from the standard input. So the user has to type it from the keyboard. Let us explore other ways of giving input later, but first understand the command above.

The `-s` option is for specifying the subject. `-a` is for specifying attachment. If there need to be multiple attachments, there need to be `-a` option for each of these. Then there is list of receipients. Here thre are three, viz rahul, amol and milind. Single receipient stands as it is; multiple receipients are separated by comma.

Most of the times, we would like to mail the status of some activity after that activity is performed by an automated script (e.g. checking available disk space on various servers, or the names of the users who have logged in multiple times to the server). But the mail command needs input from input stream demanding probably the programmer to sit at the maching while mail is being sent. But that defeats the motto of automation. To get around the problem, we can use the input redirection operator (<) to get input from a file.

```
$ mail -s “hello” rahul < message.txt  # here, message.txt is a
file whose contentents would go as body of the mail.
```

But then, that file will have to be created every time the automated script runs. And if the is command is used in a script, then for successful execution of it, both the files need to be always kept together - the script-file as well as the message.txt. Instead of that, if we use here-document (<<), we can have message as well as script in one file. This also has the advantage that you can use script’s variables in the body of the mail.

Consider below script which reports names of users with multiple logins. The mail is constructed and sent from within the script, thereby making human intervention unnecessary.

```
list=`who | cut -d" " -f1 | sort | uniq -c`
date=`date +%d%b%y`
time=`date +%H:%M`
filename=${date}_${time}users.lst
read -p "Specify the number of allowed sessions:" threshold
read -p "Specify mail id of the person to be informed:" mailto

echo "$list" | while read line
do
    #echo "current line is $line"
    user=`echo "$line" | cut -d " " -f2`
    count=`echo "$line" | cut -d " " -f1`
    if [ $count -gt $threshold ]; then
        echo "The user $user has logged in from $count
terminals" >> $filename
    fi
done

mail -s "Multiple Login Status" -a $filename $mailto <<EOM
Hello,
Please find attached a list of users who had more than
$threshold sessions opened on `hostname` on $date at $time.

Regards,
$USER
EOM

rm -f $filename # once the file is sent as an attachment, lets
remove it from the filesystem. Good citizens always cleanup
after themselves.


#Excercise: Above script has bugs. If the user specifies
threshold of 3 and no user has more than three sessions open,
then the $filename will never be created and the 'mail' command
will cry when trying to attach a non-existant file.
# So before attachment, we must check existance of file. Not
only that, if there is no attachment, we will have to alter the
body of mail. Think...!!!
# If the specified ‘threshold’ is not number, then too there is
```

```
problem. Find a solution to check if a number is numeric. Hint:
You can use grep combined with regular expression (^[0-9]+$).
```

16.2 Receiving Mail

To recive a mail, we use `mail` command.

```
$ mail [Enter]
```

You will see your inbox there. What you see is actually called ‘hearders’, which has mail number, whether it new (N) or unread (U), the sender date and time, and subject.

It is followed by the command prompt (&) of the `mail` command. Here you can issue commands to the mail program. To know the list of commands, enter ‘list’ here.

```
& list [Enter]
```

If you already know commands you can proceed by entering those. Some of most frequently used commands are –

```
& 1
```

When you want to see a mail, enter its number at the ‘&’prompt and press enter. Above command will show you mail listed at number 1.

```
& h
```

After reading the mail, if you want to go back to headers enter ‘h’.

```
& d2
```

When you want to delete a mail, press ‘`d`’ followed by mail number. If you want to delete range of mail, use it as ‘`d3-7`’.

```
& quit
```

`Quit` and `Exit` both are valid commands. `Exit` is for discarding the changes to mailbox and `Quit` is for making the changes permanent before quitting.

17 Creating Archives

17.1 tar

The name "tar" stands for "tape archive". As the name implies, it is a command that Unix administrators used to deal with tape drives. That was in old days; now a days no one sees tapes for backups, but the name sticks.

These days the `tar` command is more often used to create compressed archives that can easily be moved around, from disk to disk, or computer to computer. One user may archive a large collection of files and send them to someone else. The other user may extract those files. Both of the suers using the `tar` command.

17.1.1 Create an archive of a subdirectory

A common use of the `tar` command is to create an archive of a subdirectory. For instance, assuming there is a subdirectory named MyProject in the current directory, you can use `tar` to create an *uncompressed* archive of that directory with this command:

```
$ tar -cvf MyProject.20090816.tar MyProject
```

Where MyProject.20090816.tar is the name of the archive (file) you are creating, and MyProject is the name of your subdirectory. It's common to name an *uncompressed* archive with the .tar file extension.

In that command, I used three options to create the tar archive:

- The option `c` means "create archive".
- The option `v` means "verbose", which tells `tar` to print all the filenames as they are added to the archive.
- The option `f` tells `tar` that the name of the archive follows (immediately after this option).

The `v` flag is completely optional, but I usually use it so I can see the progress of the command.

The general syntax of the tar command when creating an archive looks like this:

```
$ tar [flags] archive-file-name  files-to-archive
```

17.1.2 Creating a compressed archive

You can compress a `tar` archive with the `gzip` command after you create it, like this:

```
$ gzip MyProject.20090816.tar
```

This creates the file MyProject.20090816.tar.gz.

But these days it's more common to create a compressed archive in one step using `tar` command, like this:

```
$ tar -czvf MyProject.20090816.tgz MyProject
```

As you can see, I added the '`z`' flag there (which means "compress this archive with gzip"), and I changed the extension of the archive to .tgz, which is the common file extension for files that have been tar'd and gzip'd in one step.

17.1.3 Creating a compressed archive of the current directory

Many times when using the `tar` command you will want to create an archive of all files in the current directory, including all subdirectories. You can easily create this archive like this:

```
$ tar -czvf mydirectory.tgz .
```

In this tar example, the '.' at the end of the command is how you refer to the current directory.

17.1.4 Creating an archive in a different directory

You may also want to create a new tar archive in a different directory, like this:

```
$ tar -czvf /tmp/mydirectory.tar.gz .
```

As you can see, you just add a path before the name of your archive to specify what directory the archive should be created in.

17.1.5 Listing the contents of a tar archive

To list the contents of an *uncompressed* tar archive, just replace the `c` flag with the `t` flag, like this:

```
$ tar -tvf my-archive.tar
```

This lists all the files in the archive, but does not extract them.

To list all the files in a *compressed* archive, add the z flag like before:

```
tar -tzvf my-archive.tar.gz
```

The zipped tar file at times is named .tar.gz too.

17.1.6 Extracting an archive

To *extract* the contents of a tar archive, now just replace the t flag with the x ("extract") flag. For uncompressed archives the extract command looks like this:

```
$ tar -xvf my-archive.tar
```

For compressed archives the tar extract command looks like this:

```
tar -xzvf my-archive.tar.gz
```

Additional information

Keep the following in mind when using the tar command:

- The order of the options sometimes matters. Some versions of tar require that the f option be immediately followed by a space and the name of the tar file being created or extracted.
- Some versions require a single dash before the option string (e.g., -cvf).

18 Finding file on the filesystem

18.1 find

find searches the directory tree starting from the given directory and in all its subdirectories for the files specified by criteria.

Syntax:

```
  find   which_dir   [criteria]
```

Some examples of find command:

To understand the find command and various criteria, let us first create some files using touch command. My current directory is /root at the moment, since I am root user. Your current directory may differ and so you will need to alter the examples below by changing /root to your home directory.

```
 touch  file_{1,2,3}
 touch  FILE_{1,2,3}
 touch  file_{pune,mumbai,nagpur}
```

Now lets use find command on these files.

```
$ find /root  -name  "file_?"
```

The above command will search all the files in /root directory(and its subdirectories) whose name starts with 'file_' and has any single character after it. The question-mark is a wild card which represents 'any one character'. Contrast it with asterik (*) which represent 'any number of multiple characters'.

```
$ find /root  -iname  "file_*"
```

The above command will search the files in /root whose name is starting with 'file_' and will have any number of character at the end and also this will do a case insensitive search. The double quotes aoround the file_* are necessary.

```
$ find /etc  -name  "*.conf"
```

The above command will search all the files which have .conf at the end of filename in /etc directory.

```
$ find /var -user root 2>/dev/null
```

The above command will find files owned by user “root” in /var directory and its sub directory. The error output is ignored.

```
$ find /etc -name “*.conf” -exec cp -av {} {}.bak \; 2>
/dev/null
```

The above command means – ‘in the /etc directory and its sub-directories, find files whose name ends in “.conf” and make a copy of them with “.bak” appended to original filename.’ The `-exec` tells the `find` command to execute another command (`cp` in above case). The pair of opening and closing curly-braces represents the result of the `find` command. Essentially, above command tries to find files whose name end in .conf and then copy each of such files with a name having .bak appended to original name. The ‘\;’ is mandatory and is the closing sequence that tells `find` that the command specified by `-exec` has ended.

```
$ find /etc -name “*.conf.bak” -ok rm {} \;  # -ok is like -
exec, but asks user for confirmation
```

In the /etc directory and its sub-directories, find files whose name ends in .conf.bak and remove them.

```
$ find /tmp  -type f  2> /dev/null
```

The above command finds under /tmp directory anything of type file.

We saw above that the criteria for finding can be name, user of type of the file being found. The find command supports many more criteria which are neatly documented in the man page of it. We will see here couple of more criteria which may be useful. Those are finding by access time and finding by permissions.

```
$ find . -atime 3 # Find files whose access time is 3*24 hours
ago.
$ find . -atime +3 # Find files whose access time is more 72
hours ago and prior to that.
$ find . -atime -3 # Find files whose access time is between now
and upto 3 days ago.
```

Note here that even if the criterion is specified in number of days, internally the hours are calculated. And since an exact match is difficult to find it applies some approximation.

Experiement with the command to get a feel of its behavior. Hints: `touch` command has a `–d` option that lets you dectate what should be the timestamp to be given to the file. `stat` is a command that shows access, modification and creating time of a file.

```
$ find . -perm 644 # Find in the current directory the files
having exact permissions of 644
$ find . -perm /644 # Find in the current directory the files
having either rw to user OR r to group OR r to others. Any one
permission match will do.
$ find . -perm -664 # Find in the current directory the files
having minimum 664 permissions. 666 will be found, 764 will be
found but 464 won't be found.
```

Create some files, change their permissions using `chmod` and experiment with above commands.

19 Password-less access to other machine

Unix being a multi-user operating system, it can have multiple users on one single machine. And of course, it has ability to be connected to multiple machines. This raises lot of security concern regarding who can access whose data and on which machine. At the file level, there are user-group-others distinction coupled with read-write-execute permissions. Then there are sticky-bit, suid, sgid and even Access Control Lists. At the user level, if you wish to pose as another user and login with his id, you will have to supply his password. And when accessing another machine, there are checks at even the machine level, whether this machine should connect to another machine. The point is, there is very tight security for individual's data and it is hardly possible to access someone else's data on some other Unix machine without proper authentication and authority.

Even when all the above security measures are warranted, there are occasions when one wants a password-less entry to home directory of another user on another machine. Consider a situation when a system administrator needs to report status of each machine every morning. How much free disk space the machine has or how much RAM is installed on it. Logging into each machine to get this information is one way of doing it, but it would mean manual intervention to enter password for each machine. The process would not only be slow, but person-depandant. Won't it be good if he can sit at one machine, run one script and the script does everything for him, without ever asking password? In the other example, a person has three different logins in three machines and he has to move files from one machine to another frequently. Entering password for each file's transfer would be too boring. Or may be you are a system administrator and your responsibility to take backup of files on 5 machines and copy them to a server somewhere. You will have to login into each of the five machines to start backup and from there copy the file onto the server by providing password of server!

In such cases, we need something like password-less entry. It is not exactly password-less – it can not be. It will be a security breach in that case. However, Unix provides a mechanism, by which one can build trust with another user on another machine so that passwords are not asked for – at least after the first connection.

To understand the mechanism lets think of an anology. You want to make a new friend. First you will hesitate to visit the new place (his housing society). But once you are convinced it is a secure place, next time you won't hesitate. Next, after reching his place, his security guard will ask your new friend whether he (the guard) should allow you in the premises. The security guard will always ask your friend whether to allow you whenever you visit there. But your friend can make an arrangement with the security guard that if you produce some kind of identification when you come and if it matches with a copy that the guard has, the guard should allow you to enter the premises without asking your friend.

This is what exactly we are going to do for establishing trust between two users of two machines. Tell your machine that the other machine (firend's society) is a good place to connect to. And then create identification and give a copy to your friend's security guard. Each time you visit friend's place with identification, the gurad matches that with his own copy and then let you in without asking for password (firend's nod).

19.1 Establishing trust between two machines

When you try to connect to any machine, say using ssh, your machine first asks you whether you trust that other machine and whether you will want to keep trusting it. If you say yes, an entry for the other machine is made in your home directory's .ssh/known_hosts file. Once the entry is there, it means now your machine won't hesitate to connect to the other machine. Next time you do ssh for the same machine, you will not be asked this question. This is like you telling your mind that you trust your friend's premises.

If you remove the entry of the other machine from your known_hosts file, next time the other machine is treated as an unknown machine and you are asked whether you trust that machine.

19.2 Establishing trust between you on your machine and the other user on other machine

The next step is to generate your identification and give it to the security guard of other user (your friend) living in another building.

19.2.1 Generating your identification

In your home directory, go to .ssh folder (it is a hidden folder, so you won't see it there unless you do `ls -a` in your home directory). On the command prompt, issue a command like below.

```
$ ssh-keygen
```

This command would generate a pair of keys, one is private and other is public. The private key is named id_rsa by default and is generated in current directory (~/.ssh). Let us accept defaults. The other key would be named id_rsa.pub. If a passphrase is asked for, let it be blank (press Enter without any input on that prompt). The public key is what we are going to give to your friend's guard.

19.2.2 Copying the identification to other machine

On the command prompt in your machine, issue a command like below from ~/.ssh folder.

```
$ ssh-copy-id friend@hisIPAddress
```

This command would try to copy your public key to firend's machine. But this would need your friend's password. It is like the first occasion when your friend will introduce you to his security gurad and let the guard keep a copy of your identification. So the other user needs to provide his password now. (This is anyway logical. You are trying to copy something from your machine to someone else's machine without having any trust between two. Though we are trying to build the trust now, it is not there yet!)

Once the friend enters his password, your public key gets copied in his ~/.ssh/authenticated_users file. Remember that it is in his machine. It is as good as his security guard is keeping copy of your identify with himself.

Now as long as the entry of your public key is in his authenticated_users file, you will not be asked for his password when connecting to his machine.

19.2.3 Some points to note

The public and private keys are plain text files. You can see your public key on your machine and see your public key in friend's authenticated_users file. They are exactly same. If you remove the entry from his file, you will again be required to enter his password before you enter his machine next time. Note the mention of userid and IP address in the key. That means, the public key is specific to a user on a machine. And when you copy it to some other machine using `ssh-copy-id` command, then you need to specify the user and IP on another machine. This means that, you establish trust with only one user only on that machine. In other words, this trust is specific to these two users on these two machines. Kyes are specific to users and machines. You can neither go to other person's home in your friend's building, nor you can enter some other building without password.

The other point is that this trust is one way. That means, if someone has accepted your key, then you can enter his home directory without his password. But he can not access your home directory without your password. For him to get password-less entry to your machine, he will have to share his public key with you and you need to accept it.

Some commands where password-less entry can be used:

ssh : This command allows you to connect to another machine. This is what we generally do to login from our local machine to remote machine. The remote machine invariably asks for password of the user using which you are logging into it. But using ssh-keygen as above, you can enter into remote machine without password. Generate key on your local machine, and copy it to another machine using ssh-copy-id command. The remote machine will ask for password for the first time, but once the key of local machine is accepted by remote machine, next time it will accept login without password. This is great feature for system administrators, who need to fetch information from multiple machines multiple times. Once the key is generated, they don't have to bother entering passwords at every connection, infact, they don't even have to remember passwords of multiple machines. And the keys are independent of password. That means, even if the password of other machine is changed later, the key remains same and password-less entry still works.

The ssh command, when used in the following way, will establish the connection with stated machine with stated user and then execute the list of commands given in double quotes on the remote machine and show result on the local machine.

```
$ ssh rahul@172.24.xx.xx "hostname;df -h; free"
```

This command would connect to 172.24.xx.xx machine using rahul's id (assuming rahul has accepted public key of this user already) without password and execute `hostname`, `df -h` and `free` commands on rahul's machine and display the results on local machine.

scp: This command is used to copy files between two machines securely. While copying file from your machine to remote machine, the remote machine invariably asks for password of the user using which you are trying to connect to it. But if the key is already generated on local machine and accepted by the remote machine, the `scp` command would not ask password while copying. Once key is generated and share you can copy files to and from the remote machine. The command's arguments are as below.

```
$ scp a.txt rahul@172.24.xx.xx:/home/rahul/fromMyFriend
$ scp rahul@172.24.xx.xx:/home/rahul/toMyFriend/b.txt .
```

The first command copies a.txt from local macine's current directory to fromMyFriend directory in home directory of a user named rahul on remote machine (124.24.xx.xx).

In the second command a file named b.txt in the toMyFriend directory in home directory of user rahul is copied in current directory of local machine.

20 Scheduling a task using crontab

cron is used to schedule tasks to be executed periodically. You can setup commands or scripts, which will repeatedly run at a set time.

cron is one of the most useful tools in Linux or Unix-like operating systems. The cron service (daemon) runs in the background and constantly checks the /etc/crontab file, /etc/cron.*/ directories. It also checks the /var/spool/cron/ directory.

crontab is the command used to install, deinstall or list the tables (cron configuration file) used to drive the cron daemon in Vixie Cron. Each user has his own crontab file, and though these are files in /var/spool/cron/crontabs, they are not intended to be edited directly. You should use crontab command for editing or setting up your own cron jobs.

20.1 Install / Create / Edit Cronjobs

To edit your crontab file, type the following command at the command prompt:

```
$ crontab -e
```

20.2 Syntax of crontab (Field Description)

Your cron job looks as follows for user jobs:

1 2 3 4 5 /path/to/command arg1 arg2
OR

1 2 3 4 5 /root/backup.sh
Where,

1 is minute’s place which can take values from 0 to 59
2 is hour’s place which can take values from 0 to 23
3 is place for day of the month (date) which takes values from 1 to 31
4 is place for month which takes values from 0 to 12 [12 == December] or jan to dec.
5 is place for day of the week which takes values from 0 to 7 [7 and 0 == sunday] or mon to sun.
/path/to/command - Script or command name to schedule

If you wished to have a script named /root/backup.sh run every day at 3am, your crontab entry would look like as follows. First, let the crontab command create a temporary file for you to enter the schedule by running the following command:

```
$   crontab -e
```

Add the following entry (append to existing entries, if any, on a new line):

```
0 3 * * * /root/backup.sh
```

Save and close the file. After closing the file you will get a message 'crontab installed successfully'. This means the way you specified schedule was recognized and accepted. If you made any mistake (e.g. entering a number more than 59 in minute's field or more than 23 in hour's field) then the crontab lets you know which column has an unacceptable entry.

More Examples:

To run /path/to/command five minutes after midnight, every day –

```
5 0 * * * /path/to/command
```

Run /path/to/script.sh at 2:15pm on the first of every month –

```
15 14 1 * * /path/to/script.sh
```

Run /scripts/phpscript.php at 10 pm on weekdays –

```
0 22 * * 1-5 /scripts/phpscript.php
```

In the above example 'weekdays' are assumed to be from Monday to Friday and the range is specified as 1-5. You could have written mon-fri too. For a range, we use start-end format. For specific values it can be comma separated list. E.g. for a job which is to be executed on Monday, Wednesday, thursday and saturday, the column for day would contain mon,wed,thu,sat

Run /root/scripts/perl/perlscript.pl at 23 minutes after midnight, 2am, 4am ..., everyday-

```
23 0-23/2 * * * /root/scripts/perl/perlscript.pl
```

Run /path/to/unixcommand at 5 past 4 every Sunday-

```
5 4 * * sun /path/to/unixcommand
```

20.2.1 How Do I Use Operators?

An operator allows you to specifying multiple values in a field. There are three operators:

- The asterisk (*): This operator specifies all possible values for a field. For example, an asterisk in the hour time field would be equivalent to every hour or an asterisk in the month field would be equivalent to every month.
- The comma (,): This operator specifies a list of values, for example: "1,5,10,15,20,25".
- The dash (-): This operator specifies a range of values, for example: "5-15" days , which is equivalent to typing "5,6,7,8,9,....,13,14,15" using the comma operator.

20.2.2 How Do I Disable Email Output?

By default the output of a command or a script (if any produced), will be emailed to your local email account. To stop receiving email output from crontab you need to append >/dev/null 2>&1.

For example:

```
0 3 * * * /root/backup.sh >/dev/null 2>&1
```

20.2.3 How do I list existing cron jobs?

Type the following command:

```
$ crontab -l # list own crontabs
$ crontab -u username -l  # list some other user’s jobs
(possible only to root user)
```

To remove or erase all crontab jobs use the following command

```
$ crontab -r
$ crontab -r -u username
```

21 Signal Trapping

Signals are software interrupts sent to a program to indicate that an important event has occurred. The events can vary from user requests to illegal memory access errors. Some signals, such as the interrupt signal, indicate that a user has asked the program to do something that is not in the usual flow of control.

The following are some of the more common signals you might encounter and want to use in your programs:

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 + \).
`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).

21.1 List of Signals

There is an easy way to list down all the signals supported by your system. Just issue `kill -l` command and it would display all the supported signals:

```
 $ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF     28) SIGWINCH
```

```
29) SIGIO        30) SIGPWR       31) SIGSYS       34) SIGRTMIN
35) SIGRTMIN+1   36) SIGRTMIN+2   37) SIGRTMIN+3   38) SIGRTMIN+4
39) SIGRTMIN+5   40) SIGRTMIN+6   41) SIGRTMIN+7   42) SIGRTMIN+8
43) SIGRTMIN+9   44) SIGRTMIN+10  45) SIGRTMIN+11  46) SIGRTMIN+12
47) SIGRTMIN+13  48) SIGRTMIN+14  49) SIGRTMIN+15  50) SIGRTMAX-14
51) SIGRTMAX-13  52) SIGRTMAX-12  53) SIGRTMAX-11  54) SIGRTMAX-10
55) SIGRTMAX-9   56) SIGRTMAX-8   57) SIGRTMAX-7   58) SIGRTMAX-6
59) SIGRTMAX-5   60) SIGRTMAX-4   61) SIGRTMAX-3   62) SIGRTMAX-2
63) SIGRTMAX-1   64) SIGRTMAX
```

The actual list of signals varies between Solaris, HP-UX, and Linux.

21.2 Default Actions

Every signal has a default action associated with it. The default action for a signal is the action a script or program performs when it receives a signal.

Some of the possible default actions are:

- Terminate the process.
- Ignore the signal.
- Dump core. This creates a file called core containing the memory image of the process when it received the signal.
- Stop the process.
- Continue a stopped process.

21.3 Sending Signals

There are several methods of delivering signals to a program or script. One of the most common is for a user to type Ctrl+c or the INTERRUPT key while a script is executing. When you press the *Ctrl+c* key a SIGINT is sent to the script and as per defined default action script terminates.

The other common method for delivering signals is to use the `kill` command whose syntax is as follows:

```
$ kill -signal pid
```

Here `signal` is either the number or name of the signal to deliver and `pid` is the process ID that the signal should be sent to. For Example:

```
$ kill -1 1001
```

Sends the HUP or hang-up signal to the program that is running with process ID 1001. To send a kill signal to the same process use the following command:

```
$ kill -9 1001
```

This would kill the process running with process ID 1001.

21.4 Trapping Signals

When you press the *Ctrl+c* or at your terminal during execution of a shell program, normally that program is immediately terminated, and your command prompt returned. This may not always be desirable. For instance, you may end up leaving a bunch of temporary files that won't get cleaned up.

Trapping these signals is quite easy, and the trap command has the following syntax:

```
$ trap ‘commands’ signals
```

Here `commands` can be any valid Unix commands, or even a user-defined function, and `signals` can be a list of any number of signals you want to trap.

There are following common uses for `trap` in shell scripts:

1. Clean up temporary files
2. Ignore signals

Following script demonstrates the tarping the SIGINT signal when it is sent to shell script

```
#!/bin/bash
trap ‘echo “trapped the signal SIGINT”; exit’ SIGINT
echo "starting infinite looop"
while [ 1 -eq 1 ]
do
    echo "Sleeping for 1 sec..."
    sleep 1
done
```

Execute above script. It will go in an infinite loop. Then press ctrl+c. This would send SIGINT signal to the script. Because of the trap statement for SIGINT the signal would be trapped and you will see ‘trapped the signal SIGENT’ message due to the echo that the `trap` statement mentions. Then the `exit` command would take place and the script would exit. If there weren’t `exit` statement there, the script would have continued even after ctrl+c. Once the signal is trapped, it loses its effect.

Keep in mind that –

1. If signal is received by a script before the interpreter interpreted the `trap` statement, the `trap` is ineffective. This is quite logical. If the shell doesn’t yet know something is to be trapped, how can it trap it! So, generally the `trap` statements are written immediately after shebang and before any other lines in the script.
2. A script can have multiple `trap` statements. They can be for different signals or same signal. But if there are multiple `trap` statements for same signal, then only the latest takes effect.
3. If a signal is trapped, its effect is lost. So if you have trapped SIGINT and did not mention `exit` in the trap statement, you will not be able to stop the script using ctrl+c.
4. Signal 9 (SIGKILL) can not be trapped. You can write a `trap` statement for this signal, but it is ineffective.

22 Sed

`sed` is a stream editor. Basically it is editor for stream of data, coming from either a file or a pipe or the keyboard. It is capable of processing lines from the file. The lines to be processed can be 'addressed' using line-numbers or specifying some string to be found in the line. What processing needs to be done on that line is specified by the commands defined by `sed`.

One thing to be remembered is, when `sed` operates on a file, it operates on the stream of data coming from the file. It does not change the contents of the original file unless asked it to do so explicitly.

Another thing to remember is, `sed` operates on whole line. So it is not much useful if you wish to cut the line in various sections and process each section separately. For such a need you will need to use '`awk`'. In some ways, `sed` can be seen as less capable than `awk`, but to understand `awk` better, `sed` must be understood first.

The best way to understand `sed` would be to write couple of files to be processed and then use `sed` on them.

We will write emp.lst file having list of employees working in some company. This is a pipe-delimited files. i.e. each of the fields is delimited by a pipe character (|). emp.lst is filxed-width as well as pipe-delimited.

As told earlier, sed does not understand various fields in a line. So being pipe-delimited has no importance in this chapter, but we are going to use same file in chapter on `awk` too.

The file looks as below.

```
2367|b.d. khurana     |g.m.     |sales     |12/10/52|6000
2645|archana saxena   |g.m.     |sales     |17/03/45|8000
3562|kokil chakrobarty|d.g.m    |production|12/05/50|7000
5682|n.k. mukherjee   |chairman |admin     |27/08/56|5400
1209|g.n. shelar      |director |sales     |12/03/50|7000
3465|sumit aggarwal   |manager  |sales     |01/06/59|5000
2078|kapil jagdale    |director |sales     |13/09/38|6700
9805|asha Choudhury   |executive|production|23/04/50|6000
2365|sumi tendon      |director |personnel |11/05/47|7800
4698|d.k. dasgupta    |manager  |production|12/09/63|5600
4563|justice ganguly  |g.m.     |accounts  |05/12/62|6300
6521|d.n. chowdury    |director |marketing |26/10/45|8200
0832|v.k. singh       |g.m.     |marketing |31/11/40|9000
5972|nikhil saksena   |d.g.m.   |accounts  |12/12/55|6000
9396|kapil Agarwal    |executive|personnel |21/08/47|7500
```

The sed commands are of the nature -

```
$ sed 'addressAndAction' inputFile
```

The single quotes are mandatory, inputFile is not mandatory, but in that case input would be sought from stdin (keyboard). addressAndAction specifies on which lines what action needs to be taken. Address and Action both can be omitted (resulting in empty double quotes). Or only Action can be specified, in which case, action is taken on each line in the file. But specifying only the address would result in error.

The addressAndAction is an expression that sed evaluates to understand what needs to be done. The addressAndAction part is also sometimes called script. Lines can be addressed using line numbers or any string to be found in that line. Actions can be quit, print, delete, write, substitute.

In addition to script (that is, addresses and action), sed accepts options like -i and -n which we will see at appropriate places below.

The contents of this chapter are structured such that there are a couple of lines of explanation and then the actual command.

Act on first three lines and then quit. The default action of sed is to print the input on the screen. So below command prints first three lines on screen and then quits. q stands for quit.

```
$ sed '3q' emp.lst
```

Print first to fourth line of emp.lst. The default action being to print all the lines in input file, this command prints entire file's contents in addition to first through fourth lines. p stands for print.

```
$ sed '1,4p' emp.lst
```

To supress the default behavior of printing all lines, add -n option. Below command prints only the addressed lines, i.e. first through fourth.

```
$ sed -n '1,4p' emp.lst
```

Avoid printing first two lines. Give address of first two lines, followed by negated action (NOT print).

```
$ sed -n '1,2!p' emp.lst
```

It is not necessary that you print only the consequent lines; sections of files too can be printed. (Note: There is an ENTER pressed after p in the first line of this command).

```
$ sed -n '9,12p
```

```
14p' emp.lst
```

The same as above can be achieved by using `-e` options. `e` stands for expression.

```
$ sed -n -e '9,12p' -e '14p' emp.lst
```

Print the last line of the file. The '$' stands for last line.

```
$ sed -n '$p' emp.lst
```

Insert `something` before third line. `3` gives address and `i` is action (to insert). For better readability of this command, use other version as given further below.

```
$ sed '3isomething' emp.lst
```

```
$ sed '3i\

something

' emp.lst
```

Insert `something` after line numbers 4 through 7 line. “`4,7`” gives address and `a` is action (to append). For better readability of this command, use other version as given further below.

```
$ sed '4,7asomething' emp.lst
```

```
$ sed '4,7a\

something

' emp.lst
```

Append a new record at the end of emp.lst and store the output in empnew.lst. The `$a` is the addressAndAction. `$` means last line, `a` means append. 'Append at the end of file' is the actual meaning of `$a`.

```
$ sed '$a\

9546|v.n tiwary       |manager  |marketing |22/02/53|5800

' emp.lst > empnew.lst
```

Below demonstrates the need of '\' at the end of the record. It is evident when we more than one lines to be inserted/appended. Without the '\', `sed` cannot understand it is a new record that follows.

```
$ sed '$a\
9546|v.n tiwary       |manager  |marketing |22/02/53|5800\
9547|v.n1 tiwary      |manager  |marketing |22/02/53|5800
' emp.lst > empnew.lst
```

Lines need not only be addressed by line numbers or '$', but can be addressed using string-patterns too. Addressing-by-pattern would mean 'those lines which contain the stated pattern'. Below command means print all those lines which contain string 'director' in them. While specifying the pattern, remember that sed is case sensitive.

```
$ sed -n '/director/p' emp.lst
```

Below command means print all the lines starting at the line containing 'dasgupta' upto the line containing 'saksena'.

```
$ sed -n '/dasgupta/,/saksena/p' emp.lst
```

Number-addressing and pattern-addressing can be combined. Below command prints from first line upto the line containing 'saksena'

```
$ sed -n '1,/saksena/p' emp.lst
```

Delete all the lines which contain the word 'director' and store the resulting list into file named 'olist'. d stands for delete. After this command, verify in emp.lst that its contents are intact. sed did not delete lines in source file.

```
$ sed '/director/d' emp.lst > olist
```

Write the lines containing 'director' in another file named 'dlist'. w stands for write.

```
$ sed -n '/director/w dlist' emp.lst
```

sed can be used for substitution too. Below command substitutes word 'director' by 'member ' in line number 1 to 5. s stands for substitute. The source file, emp.lst remains unaffected. 'member ' has spaces towards end to maintain fixed-width nature of the data.

```
$ sed '1,5s/director/member  /' emp.lst
```

If no address is specified, the action is taken on entire file.

```
$ sed 's/director/member  /' emp.lst
```

The above substitution takes place in each matching line but only for the first occurance of the word. That is because the s command operates only on first occurance. As another example, the below command substitutes 'or' by 'oo' in the file. Since it replaces only the first occurance, the Chakrobarty who is in Production department, sees change only in his name, not his department's. For the 'ro' in Production to change, we would need to do the substituion globally (in the entire line). Use g flag for that.

```
$ sed 's/ro/oo/' emp.lst
```

```
$ sed 's/ro/oo/g' emp.lst
```

sed accepts regular expressing for specifying patterns. Below pattern would match both spellings of Choudhury in emp.lst. Discussion of regular expressions is out of scope for this book.

```
$ sed -n '/[cC]ho[wu]dh*ury/p' emp.lst
```

Below command prints all the lines which have '5' and '0' as 7th and 6th characters respectively from the end of the line.

```
$ sed -n '/50.....$/p' emp.lst
```

When there would be large number of characters to be counted, above form may be tiresome and difficult to maintain. It can be compressed by another pattern which specifies the count of characters. Below line looks for the lines which have fifty characters (\{50\}) from the beginning of line (^) and then '5' and 0' as 51st and 52nd characters respecitvely.

```
$ sed -n '/^.\{50\}50/p' emp.lst
```

Replacing a pattern in the selected lines. e.g change designation of 'director' to 'memeber' where department is 'marketing'. The below command means - look for the lines containing word 'marketing' and in those lines substitute 'director' by 'member ' and print those lines.

```
$ sed -n '/marketing/s/director/member  /p' emp.lst
```

sed can remember the pattern which was used for addressing the lines. This can be useful when search-pattern is part of substitution. sed remembers the search pattern (director) and you can use it during substitution by way of '//'.

```
$ sed '/director/s//member  /' emp.lst
```

When a pattern in the source string also occurs in the replacement, you can use the special character '&' to represent it. Below command would find lines containing ‘director’ and then substitute 'director' by 'executive director' in those lines.

```
$ sed '/director/s//executive &/' emp.lst
```

Sometimes you require a part of the search pattern to be present in the target string. This can also be achieved with the special `sed` featue of attaching a tag to a pattern. Suppose you want to substitute the word 'Production Manager' by 'Manager-Producation'. Here both the words in search pattern are needed in target string.

```
$ echo "Production Manager" | sed 's/\(Production\)
\(Manager\)/\2-\1/'
```

Though not related to the feature being discussed here, notice that there is no file to process - the `sed` is taking input from a pipe.

Delete the blank lines from a file. (remove all the lines which have either nothing, only spaces and/or tabs in it). You will have to first alter the emp.lst file to introduce some blank lines and lines with spaces and tabs into it.

```
$ sed '/^[ \t]*$/d' emp.lst
```

If you want to apply edits to original file rather than editing only the stream, use `–i` option. The below command deletes all the lines containing ‘director’ word from the original file. If you open the file after executing this command, you will realize the original contents of the file are altered.

```
$ sed  -i ‘/director/d’ emp.lst
```

For the `awk`’s chapter, we would require original contents of emp.lst. So restore this file before proceeding for `awk`’s chapter.

23 awk

As the `sed` is useful for editing the streams, `awk` too is used for similar reason. But `awk` is more powerful than `sed`. awk not only searches and processes the data in file, but also formats the output in a desired way. It also offers functionality to add headers and footers to the group of processeds lines. This is wonderful way to create reports on data in text files. Let us see how it works. We will start with basic search-and-print funcationality and then move to manipulating individual fields, searching within fields, comparing the field values, formatting the output and then adding headers and footers to the output.

`awk` can be seen as a programming language in itself where you can declare variables, have conditional statements and even have loops. This chapter will not go deep into those language-like constructs, but has enough examples to give you an idea of `awk` 's capabilities.

In `awk` you can define your own variables as well as it has its inbuilt variables. One of in-built variables is NR which holds the line number in the source. We would use this variable to address the lines.

General format of `awk` command is

```
$ awk 'addressAndAction' inputFile
```

Single quotes are mandatory. If only address is specified, default action is printing. If only action is provided, it is applied to all the lines in inputFile. If both are omitted, nothing is outputted. `awk` is case sensitive by default.

In addition to emp.lst from sed's chapter we will use empn.lst, which has same data, but its columns are not fixed-width. The extra spaces in each of the columns are removed.

```
2367|b.d. khurana|g.m.|sales|12/10/52|6000
2645|archana saxena|g.m.|sales|17/03/45|8000
3562|kokil chakrobarty|d.g.m|production|12/05/50|7000
5682|n.k. mukherjee|chairman|admin|27/08/56|5400
1209|g.n. shelar|director|sales|12/03/50|7000
3465|sumit aggarwal|manager|sales|01/06/59|5000
2078|kapil jagdale|director|sales|13/09/38|6700
9805|asha Choudhury|executive|production|23/04/50|6000
2365|sumi tendon|director|personnel|11/05/47|7800
4698|d.k. dasgupta|manager|production|12/09/63|5600
4563|justice ganguly|g.m.|accounts|05/12/62|6300
6521|d.n. chowdury|director|marketing|26/10/45|8200
0832|v.k. singh|g.m.|marketing|31/11/40|9000
5972|nikhil saksena|d.g.m.|accounts|12/12/55|6000
9396|kapil Agarwal|executive|personnel|21/08/47|7500
```

Just as sed, printing is the default action. So, in the first example, lets print 2nd through 5th line.

```
$ awk 'NR==2, NR==5' emp.lst
```

To explicitly mention action of printing use below command. Notice that the actions in awk are mentioned in curly braces.

```
$ awk 'NR==2, NR==5{print}' emp.lst
```

The lines in input file can be addressed using search pattern too. Below command finds the lines containing word 'director'. Search patterns are put within pair of '/'s as in sed.

```
$ awk '/director/{print}' emp.lst
```

You can sepcify two patterns e.g. Find lines containing 'director' or 'manager'.

```
$ awk '/manager|director/{print}' emp.lst
```

The patterns can be regular expression. Below line searches both the occurances of differently spelled Saxena.

```
$ awk '/sa[kx]s*ena/{ print} ' emp.lst
```

In all the above examples, we tried to find a line in a file and printed the whole line. It was similar to sed. From here onwards, we we see the differentiating capabilities of awk. Let us start with splitting line in different fields based on a field-separator. Our file has pipe (|) as field separator. We can inform awk about the field-separator character and then access individual fields by $1, $2, $3 etc for first, second, third fields respectively.

```
$ awk -F "|" '/sa[kx]s*ena/{ print $1 $2} ' emp.lst
```

The above command prints fields 1 and 2 from the file. But they are concatenated to each other in the output. That is the default behivor of print command. If you want to have a separator between two fields in output, you need to put a comma between two field names. Default output field separator (OFS) is space. You can change value of this inbuilt variable (OFS), but we will see that later.

```
$ awk -F "|" '/sa[kx]s*ena/{ print $1,$2} ' emp.lst
```

Now that we have separated the fields in the file, we can even specify address of lines based on values of the field. Below command searches for those lines whose third field is either 'director' or 'chairman'. Notice that we are using empn.lst here. This file does not have any extra spaces. This command wont work if we change the input file to emp.lst since emp.lst has spaces after 'chairman' and 'director' words.

```
$ awk -F"|" '$3 == "chairman" ||  $3 == "director" {print NR,
$2,$3,$6}' empn.lst
```

If you want to have list of only those employees who are neither director nor chairman you can use != operator.

```
$ awk -F"|" '$3 != "chairman" &&  $3 != "director" {print NR,
$2,$3,$6}' empn.lst
```

Notice the value of NR in the output of above two commands. It is the line number in the input file, not the serial number in output.

If your search pattern is a regular expression, then for field comparision, you need to use ~ as operator and regular expression in pair of '/'.

```
$ awk -F"|" '$2 ~ /sa[kx]s*ena/ {print NR, $2,$3,$6}' empn.lst
```

For negating the regular express match, use !~. Below command shows everyone except for both Saxena.

```
$ awk -F"|" '$2 !~ /sa[kx]s*ena/ {print NR, $2,$3,$6}' empn.lst
```

If you are comparing two values for the same column (e.g. you need either saxena or choudhury in $2) you can use a shortcut like below.

```
$ awk -F"|" '$2 ~ /[cC]ho[wu]dh?ury|sa[xk]s?ena/ {printf "%4d %s
%-20s\n", NR, $1, $2}' empn.lst
```

Till now we have been printing the lines in the input file as it is, or printing selective fields with default delimiter. But if we want to have a nice-looking table-like structure, we need to format the output. We use `printf` statements for printing the formatted output. It accepts a string which tells what the format should be, and then the fields which need to be formatted.

The formatting strings have to follow some syntax. Read the next command in conjunction with this explanation of formatting strings. The '`d`' stands for digits, '`s`' stands for string. Numbers before '`d`' and '`s`' specify the total number of spaces that will be allocated for that column. The '`-`' sign means left justified. No sign mean right justification. '`\n`' adds new line after each printed line. (without it, the output lines will be printed next to each other). In the following command, `NR` is numeric and other fields are strings. The command finds any line which contains any variation of Agarwal. Then splits the line into various fields considering '`|`' as field separator. Then prints the line number and fields numbered two, three and six. The printing is formatted as –

i. The line number appears in first column of output and gets three spaces and would be right justified. It will be treated as number(digits).

ii. The second field, the name, gets 20 spaces and would be left justified in second column of output. It will be treated as string.

iii. The third field is treated as string, gets 12 spaces and is left justified.

iv. The last column in output, i.e. sixth field in input record is treated as number and we are not specifying how many spaces it will be allocated.

v. This all is followed by a new line character so that the output is only one employee record in one row of table.

```
$ awk -F"|" '/[aA]gg?[ar]+wal/ { printf "%3d %-20s %-12s %d\n",
NR, $2, $3, $6 }' empn.lst
```

`awk` allows comparasion operators too on the fields. And based on the operator being used it converts the field into numeric type for comparision. In the below command we are trying to find employees earning more than 7500. Notice that the comparision with $6 is numeric and in the same command we are formatting $6 as string in printf statement.

```
$ awk -F"|" '$6 > 7500 {printf "%4d %s %-20s %s\n", NR, $1, $2,
$6}' empn.lst
```

The address part need not mention only one field at a time. It can contain conditions on multiple columns. e.g. Below command finds any employees who are earning more than 8000 OR born in year 45.

Observe the matching pattern for fifth field. We are looking for a fifth field (birth dates of employees) ending in 45.

```
$ awk -F"|" '$6 > 8000 || $5 ~/45$/ {printf "%4d %-20s %8s
%d\n", $1, $2, $5, $6}' empn.lst
```

This goes on lines of SQL as - select emp_id, emp_name, dob, salary from employees where salary > 8000 or to_char(dob, 'yy') = '45'.

You can setup kind of a mini-database using `awk` on data in a text file!

The action in `awk` need not only limited to printing or formatted printing. Action can be anything like comparision, loops, numeric functions, string functions, time functions, user defined functions, input/outpt statements etc. `awk` is like a full programming language in itself.

We will see variable declaration and mathematical expression in next two commands. Variables need not be declared before using them. A numeric variable has value of zero when it is used first time.

Here we are defining a variale called '`kount`' and using that as line number of output (instead of NR - the line number in the source file).

```
$ awk -F"|" '/director/ { kount=kount+1; printf "%3d %-20s\n",
kount, $2}' empn.lst
```

Below command is computing and displaying 20% increased salary of directors along with their names.

```
$ awk -F"|" '/director/ { k+=1; printf "%3d %-20s %d\n", k, $2,
$6*1.2}' empn.lst
```

In addition to all this, `awk` also provides features of `BEGIN` and `END` blocks. One can put any a `awk`-specific code in those blocks. The `BEGIN` block is executed only once before processing of the file starts. The `END` block is executed only once after processing of the file ends. These blocks can be effectively used for creating headers and footers for reports.

However, when we use those, the script part of `awk` command (the part between single quotes) becomes too large to be written on command prompt. Hence we use `'-f'` option of `awk` command. This option allows us to write the script part in a text file and then let `awk` read that file to get its script.

Let our script be called emp.awk and it contains as below.

```
BEGIN {
    # Since we are specifying the input file separator (FS)
here, we can avoid specifying '-F' option for awk command when
we call this script later.
    FS="|"
    printf "\n\t\tEmployee abstract\n\n"
}

$6 > 7500 { # find people with salary more than 7500
    kount++; total += $6
    printf "%3d %-20s %-12s %d\n", kount, $2, $3, $6
}
END {
    # The use of OFS is an impractical example here. But it is
effective to convey how variable's value can be changed and
used.
    # ' and ' appears in place of comma between 'kount' and
'total' variables in output.
    OFS=" and ";
    print "\n\t Count of employees earning more than 7500 and
their collective salary are " kount, total " respectively";
    printf "\n\t The average basic pay is %6d\n\n\n",
total/kount;

}
```

Now call this script at Unix command prompt.

```
$ awk -f emp.awk emp.lst
```

The `BEGIN` block executes first where we are assigning a value to the input field separator (FS) and printing the header for the report. Then each line from input file is processed by the `awk` which counts the people having salary more than 7500, sums their salaries and prints their information. Then the `END` block executes which sets the output field separator (OFS) and prints two statements on the output. OFS's use here is a far-stretched imagination, but it can be put to a far better use than this.

By the way, `$0` stands for entire line. And `print` by default prints the entire line. So below line produces same output with or without `$0`. It prints entire line where ‘director’ word appears.

```
$ awk ‘/director/{print $0}’ emp.lst
```

24 Useful Assignments

24.1 Shell Scripting Assignments for Linux Admins

1) Write a shell script for download a file from ftp server.

```
#!/bin/bash


read -p "ENTER IP ADDRESS: " ip # Accepts IP ADDRESS from the User
HOST=$ip                        # IP Address is assigned to variable HOST

read -p "ENTER USERNAME: " u    # Accepts the USER name
USER=$u                         # Username is assigned to variable USER
read -p "Enter Password: " p    # Accepts the Password
PASSWD=$p                       # Password is assigned to variable PASSWD
# -------------------------------------------------------
#  Now we will use ftp command to login to the FTP server
#  to the provided IP by the User.
#  by passing the User name and Password.
# -------------------------------------------------------
ftp -ivn $HOST <<END_SCRIPT
quote USER $USER
quote PASS $PASSWD
cd pub
mget a.txt aaaa.txt
bye
END_SCRIPT
```

2) Check the status (ping) of the server by shell script. If IP is not specified give an error message.

```
#!/bin/bash

echo -n "Enter IP Address: "
read ip                     # Accepts the IP ADDRESS From the user

# ---------------------------------------------------------
    # Now let us check the value of variable 'ip'
    # If the IP is not provided will show the Error Message
# ---------------------------------------------------------

if [ ! $ip ]; then # will check for Value of $ip
      echo "IP was not Provided..!!!!"
    else
      echo "The entered ip: $ip"  # Displays the IP ADDRESS that the User had Entered
      # we need to ping this server to
      # check which server is working and is Down
      ping -c1 $ip> /dev/null

      # ---------------------------------------------------
      # Now let us check the exit status of the ping command
      # Exit status of 0 means the server is up
      # ---------------------------------------------------
      if [ $? -eq 0 ]; then
            echo "Ping Successfully done"
          else
            echo "Ping was Unsuccessfull"
      fi
fi
```

3) Write a shell that will chechk the status (Ping) of entire network.

- Send the list of down servers as a email attachment to respective authority
- The list of servers to be pinged is kept in a file named ip.txt

```
#!/bin/bash
>/tmp/$$_machineup.txt   # this will clean up the machinedown.txt file's contents if any
(from previous run).
>/tmp/$$_machinedown.txt  # this will clean up the machinedown.txt file's contents if any
(from previous run).

if [ ! -e ip.txt ]; then
    echo "File ip.txt does not exist."
    echo "Canot proceed further to ping the network"
    exit 9  # exit the shell script with exit status of 9 (other than 0)
fi

# ---------------------------------------------------------------
# The following for loop will read ip addresses from a file
# named ip.txt
# ---------------------------------------------------------------
for i in `cat ip.txt`
do
    # We need to ping these servers to
    # check which server is working and which is down
    ping -c1 $i> /dev/null # variable i contains the ip address # -c1 means ping for
'c'ount '1'
    # ---------------------------------------------------
    # Now let us check the exit status of the ping command
    # Exit status of 0 means the server is up
    # ---------------------------------------------------
    if [ $? -eq 0 ]; then
        echo $i is up >>/tmp/$$_machineup.txt
    else
        echo $i is down >>/tmp/$$_machinedown.txt
    fi
done
mail -s "Working Servers" rahul </tmp/$$_machineup.txt

    #---------------------------------------------------
    # The mail command will send contents of $$_machineup.txt to rahul in a mail with
subject line "Working Servers"

rm -f /tmp/$$_machineup.txt
rm -f /tmp/$$_machinedown.txt

# As an excercise, read the input file's name ( the ip.txt ) from command line.
# If user did not specify any input file, then use name ip.txt.
# In another varient of excercise, if user doesn’t specify input file's name, show usage.
# In yet another varient of excercise, show the usage in a funcation and call the
funcation if input file is not specified.
```

4) Write a shell script for generating File system space utilization report.

- Indent the content of report nicely.
- The report should be sent every morning at 8:00 AM.

```
#!/bin/bash
#----------------------------------------------------------------------------
# df -h command gives you the filesystem and mount details.
# Here it redirects the details of the command to the part.txt file

# Note that depending on the file system's names, the output of 'df -h' may vary
in format and composition of output.
# This script may need some modification in cut, awk and sed commands' options for
desired output.
#----------------------------------------------------------------------------

df -h | cut -c23-1000 | tr -s " " ":" | awk -F ":" '{ print $4 "    " $5}' | sed
'/^$/d' | sed 's/%//' | grep -v Use | sed '/ $/d' >part.txt

#----------------------------------------------------------------------------
#    The while loop Reads the part.txt file and checks for the % usage
#    of the mount points. if it is greater then 75 it gives a message.
#----------------------------------------------------------------------------
while read line
do
part=`echo $line | awk '{ print $2 }'`
size=`echo $line | awk '{ print $1 }'`

if [ $size -gt 75 ]; then
    echo "$part --- >> has less space"
else
    echo "$part --- >> has more space"
fi
done <part.txt
#----------------------------------------------------------------------------
#   For sending the report at 8am every day set a crontab for the script
#   and use mail command for sending the part.txt file which gives the Report
#----------------------------------------------------------------------------

# Excercise: The threshold to declare 'more space' or 'less space' (75 here) to be
put in a variable(named 'threashold').
# and use that variable in 'if' condition as well as in the echo message.
# On the screen the user should see output something like below
# "/dev/shm has consumed more than 60% space"
# where the number 60 has come from $threshold.
# In order to understand this script, start on command prompt by typing 'df -h'
and observe output. Then pipe that to 'cut -c23-1000' command and observe output.
# Likewise do it till the last 'sed' command in the pipeline above.
```

5) Write a shell script to Lock all users having UID Greater than 500 and and less than 530 .

```
#!/bin/bash

# ------------------------------------------------------------------------
# The following for loop will read the USERNAME and UID of the Users from
# a file /etc/passwd
# ------------------------------------------------------------------------

while read line
do
   u=`echo $line | cut -d":" -f1`   # Username will be stored in 'u'.
   id=`echo $line | cut -d":" -f3`  # UID will be stored in 'id'.

    # --------------------------------------------------------------------
    # Now let us check the value of UID
    # We Require the user name with UID of Range 500 to 530
    # --------------------------------------------------------------------

if [ $id -ge 500 -a $id -le 530 ]; then
   # The below line, which actually locks the user, is intentionally
commented to prevent
   # accidental locking of users. Once you are sure of what you are doing,
uncomment it!
   # usermod -L $u &>/dev/null   #this will lock the User
    echo "$u with user id $id was locked successfully."
fi

done < /etc/passwd

# Excercise: accept the lower and upper limit of userids from command
prompt.
# First CLA would be lower limit ( 500 in above script)
# Second CLA would be upper limit (530 in above script)
```

6) Write a shell script Lock all users whose names are in a file called users.txt

```
#!/bin/bash

if [ ! -e users.txt ]; then
    echo "File users.txt does not exist."
    exit 9  # exit the shell script with exit status of 9 (other than 0)
fi


# ---------------------------------------------------------------------------
# The following for loop will read the USERNAME of the Users from
# a file users.txt
# ---------------------------------------------------------------------------

while read line
do

 # -----------------------------------------------------------------------
 # Now the users.txt will be read line-by-line,
 # and this script will lock the users present in the file.
 # If the user can not be locked for any reason, e.g. user does not exist,
 # it would give an error message.
 # -----------------------------------------------------------------------

   # The below line, which actually locks the user, is intentionally
commented to prevent
   # accidental locking of users. Once you are sure of what you are doing,
uncomment it!
    #usermod -L $line &>/dev/null   #this will lock the User

    # The 'id' command exits with zero status if the user exists and with
non-zero status otherwise.
    # We are using this behavior of 'id' command to simulate the 'usermod -L'
command's exit status.
    id $line &> /dev/null  # Comment this line when script is not for testing
purpose

    if [ $? -eq 0 ]; then
        echo "$line successfully locked "
    else
        echo "$line user not found "
    fi

done < users.txt
# Excercise: Take the names of users from the user at command prompt instead
of from users.txt.
# Hint: In a while loop, keep prompting for usernames until the user enters
'endoflist'. Store these names in a variable.
# Use that variable in another loop to read one user at a time and lock.
(hint2: 'for' loop may be easier in this case to use).
```

7) Write a shell script recycle log files in /var directory

- remove oldest lines.
- Put this in a crontab.
- make sure that you leave atleast 1000 lines in the file.

```
Note: To rotate the log files in /var/log directory you can select each file or
you can create a file which will contain the the name of files which are being
rotated.
Let us do this by creating file which will contain the list of files to be
rotated.
The file will be as follows
Name and Pathe of File :/var/log/rotate_list.txt
/var/log/messages
/var/log/secure
/var/log/cron
/var/log/dmesg
/var/log/boot.log
/var/log/vsftpd.log
----------------------------------------------------------------------------------
#!/bin/bash
while read old_name
do
   temp_name=${old_name}.bak # Make a temporary name from current file name
   tail -5 $old_name > $temp_name # 'tail' the last few (5 here) lines from old
file and redirect the outptu to temporary file
   rm -f $old_name # remove the old file
   mv  $temp_name  $old_name # rename the temporary file as old file
done < rotate_list.txt

# Exercise:
# 1. Handle a case of blank lines in rotate_list.txt.
# 2. Take the number of lines to be preserved (5 in the above script) from command
prompt.
# 3. Mention the number of lines to be preserved against each file name in
rotate_list.txt and use that number as count of lines to be preserved.
# For the purpose of third excercise, the entry in rotate_list.txt would look like
below.
# ftp_23-Sep-2013.log 1000
# mail_23-Sep-2013.log  300
# If you are left with some energy, modify the script so that lines starting with
'#' in rotate_list.txt are ignored.
```

8) Write a shell script to Identify list of user who have executed more than 10 jobs yesterday.

```
#!/bin/bash
users=`awk -F":" '$3>499{print $1}' /etc/passwd` # $3 is numeric user id and
we are checking for uids more than 499.
#The grep command below will display the number of time and the #username
that occurs in var log cron file
#w - exact match of user
#o - cut the mathing word only
#E - extended regular expression can be user to search multiple words

cron_log="/var/log/cron" # The name of the file in which the data is to be
searched for.
yesterday_date=$(date -d yesterday '+%b %d') # yesterday's date in 'mon dd'
format
grep "$yesterday_date" $cron_log | grep -oE "$users" |sort | uniq -c >
cront_job_count_for_user.txt
awk '$1>=10 {print $2 " executed cron job " $1 " times on $yesterday_date" }'
cront_job_count_for_user.txt

# Exercise:
# 1. Show all entries of cron job execution, irrespective of how many times
the job was executed.
# 2. Instead of displaying the output sorted by user ids alphabetically,
display it sorted on the count of job-execution.
```

9) Write a shell script to find files larger than a specific size. Report it to a concerned person through email.

```
#!/bin/bash
read -p "Enter the size you need to find: " s

# ---------------------------------------------------------
# we need to search the files greater than specific size
# we will use FIND command for searching the files.
# ---------------------------------------------------------

find  /  -size  +$s >size.txt  2> /dev/null

mail -s "file size details" -a size.txt rahul <<EOF
EOF


# Excercise: make the directory to be looked into ( / in this example )
configurable.
# What would you choose between
 (a) command line argument and
 (b) reading from file
to get the directory to be looked into?
# Why would you choose that option in this case?
```

10) Generate and email Security check report daily

- List of users with 0 UID
- List of users in visudo file
- List of files on your system with 777 permissions
- List of users who did su to the root account that day only

```
#!/bin/bash


# --------------------------------------------------------------------------------
#  List of Users with UID Zero is displayed using awk command
#  from the file /etc/passwd. the output is sent in a file op.txt
# --------------------------------------------------------------------------------
>op.txt # Empty the existing contents of op.txt

echo "Users with zero UID : " >> op.txt

awk -F ":" '$3~"^0$"{ print $3 "   "  $1 }' /etc/passwd >> op.txt


# --------------------------------------------------------------------------------
#  For list of Users in VISUDO file.
#  The usernames from /etc/passwd are kept in secure.txt file
#  We use while loop to check the username in the visudo file
# --------------------------------------------------------------------------------

>secure.txt # Empty the existing contents of secure.txt

echo -e "\n\nUsers in VISUDO : " >> op.txt

cat /etc/passwd | cut -d  ":" -f1 > secure.txt # Get users from /etc/passwd file

while read line
do
u=`grep -wo "^$line" /etc/sudoers | uniq ` # Find all those lines in /etc/sudoers
which start with users from secure.txt file

if [ $line = "$u" ]; then
    echo "$line is there in visudo file" >> op.txt
fi
done < secure.txt


# --------------------------------------------------------------------------------
#  For Files with 777 Permissions.
#  We use find command for searching the files with 777 Permissions
#  on the system.
# --------------------------------------------------------------------------------
```

```
echo -e "\n\nFile with 777 Permisions : " >> op.txt
find / -perm 777 >> op.txt 2> /dev/null

# --------------------------------------------------------------------------------
#  For user who did SU on a particular day
#  we need to search the system log for checking who did "su" on
#  a particular day
# --------------------------------------------------------------------------------
su_logs="/var/log/secure"
dt_format="mmm dd" # Change value of this variable according to the format in your
$su_logs file
read -p "Enter the date on which you want to see who used SU ($dt_format):" d

echo -e "\n\nThe users who used su " >> op.txt
if [ ! -z "$d" ]; then
    cat $su_logs | grep -w "su" | grep -w "opened" | grep -iw "$d" | awk '{ print
$11 }' | sort | uniq -c >> op.txt
else
    echo "The date was not entered; did not attempt to find." >> op.txt
fi


# --------------------------------------------------------------------------------
# For mailing the report we use mail command. Below we are sending mail with
op.txt as attachment and without any message to rahul
# mail -s "security check" -a op.txt rahul <<EOF
# EOF
# --------------------------------------------------------------------------------
```

11) Write a shell script to Reset passwords of all users listed in a file user.list

```
#!/bin/bash
# --------------------------------------------------------------------------------

if [ ! -e  user.list ]; then
    echo "File users.list does not exist."
    exit 9  # exit the shell script with exit status of 9 (other than 0)
fi


# --------------------------------------------------------------------------------
# The While Loop Will read the username of the users from the
# file users.list to change the password of the user.
# -------------------------------------------------------------------------------
while read line
do
echo "$line" | passwd --stdin $line &>/dev/null # See the explanation at bottom
of this script
if [ $? -eq 0 ]; then
        echo "${line}'s password successfully Changed "
    else
        echo "$line User not found "
fi

done < user.list

# Explanation: The passwd command is used for changing password.
# The command can be used by any user without any argument.
# However, the root user can use this command with an argument of userid.
# We are using the second varient here - passwd with a user id (the variable
'line' contains userid).
# The '--stdin' option tells the 'passwd' command that the input would come from
stdin.
# The 'echo' command is piped into passwd command, which means the string that is
echoed (the value of 'line') is on stdin of passwd command.
# This all together effectively means, the value of variable 'line' is given on
stdin to passwd command and that is set as password for user.
```

12) Script to check the system load and notify the administrator

```
#!/bin/bash
#
# Script to notify admin user if Linux load crossed certain limit
# It will send an email notification to admin.
# Set up limit below
NOTIFY="6.0"

# admin user email id
TO="rahul"

# Subject for email
SUBJECT="Alert $(hostname) load average"

# ----------------------------------------------------------------------------

# Os Specifc tweaks do not change anything below ;)
OS="$(uname)"
TRUE="1"
if [ "$OS" == "FreeBSD" ]; then
    TEMPFILE="$(mktemp /tmp/$(basename $0).tmp.XXX)"
    FTEXT='load averages:'
elif [ "$OS" == "Linux" ]; then
    TEMPFILE="$(mktemp)"
    FTEXT='load average:'
fi


# get first 5 min load
F5M="$(uptime | awk -F "$FTEXT" '{ print $2 }' | cut -d, -f1 | sed 's/ //g')"
# 10 min
F10M="$(uptime | awk -F "$FTEXT" '{ print $2 }' | cut -d, -f2 | sed 's/ //g')"
# 15 min
F15M="$(uptime | awk -F "$FTEXT" '{ print $2 }' | cut -d, -f3 | sed 's/ //g')"


# Look if it crossed limit
# compare it with last 15 min load average
RESULT=$(echo "$F15M > $NOTIFY" | bc)

# if so send an email
if [ "$RESULT" == "$TRUE" ]; then
    # mail message
    {
    echo -n "" > $TEMPFILE #Remove the old contents of $TEMPFILE
    echo "Load average Crossed allowed limit $NOTIFY."
    echo "Load average is $F15M"
    echo "Hostname: $(hostname)"
    echo "Local Date & Time : $(date)"
    }>> $TEMPFILE
    mail -s "$SUBJECT" "$TO" < $TEMPFILE
fi

# remove file
#rm -f $TEMPFILE
```

13) Shell script to find the number of files present in the current directory without using WC command.

```
#!/bin/bash
# Shell script to find the number of files present in the current
# directory without using WC command.
# NOTE/TIP:
# If allowed to use wc command then it should be as follows:
# ls | wc -l

FILE="filecount.txt"

echo -n "Enter directory name : "
read dname

ls $dname > $FILE

echo "No of files in directory : $(grep [^*$] $FILE -c)"
#rm -f $FILE
```

14)Shell script to find out the files having suid.

```
#!/bin/bash
# Shell script to find all programs and scripts with setuid bit set on.
#

if [ $# -ne 1 ] ; then
    echo "Usage: $($BASENAME $0) directory"
    exit 1
fi

DIRNAME="$1"
FIND $DIRNAME -xdev -type f -perm +u=s -print
```

15) Shell script to rename given file names to from uppercase to lowercase OR lowercase to uppercase

```
#!/bin/bash
#
# Shell script to rename given file names to from uppercase to
# lowercase OR lowercase to uppercase
#
ME="$(basename $0)"
FILES="$1"
OPTION="$2"

# function to display message and exit with given exit code
function die(){
echo -e "$1"
exit $2
}

# exit if no command line argument given
[ "$FILES" == "" ] && die "Syntax: $ME {file-name} option\nExamples:\n $ME xyz 2upper\n
$ME \"*.jpg\" 2lower" 1 || :
# exit if no OPTION given
[ "$OPTION" == "" ] && die "Option must be specified as either 2upper or 2lower" || :
# scan for all input file
for i in $FILES
do
    # see if upper to lower OR lower to upper by command name
    [ "$2" == "2upper" ] && N="$(echo "$i" | tr [a-z] [A-Z])" || N="$(echo "$i" | tr [A-
Z] [a-z])"
    # if source and dest file not the same then rename it
    [ "$i" != "$N" ] && mv "$i" "$N" || :
done


# Exercise: Write the functionality for the case where user specified option of 2lower.
# Explore what is [[:lower:]] and [[:upper:]] and replace the [a-z] and [A-Z] by those.

Hint: look for man page on 'grep' and read ' Character Classes and Bracket Expressions'.
```

16) Script to update user password in batch mode.

```
#!/bin/bash
# Script to update user password in batch mode
# You must be a root user to use this script
# /root is good place to store clear text password
FILE="/root/batch.passwd"

# get all non-root user account
# By default on most linux non-root uid starts
# from 1000
USERS=$(awk -F: '$3 > 1000 { print $1}' /etc/passwd)

# create file with random password
echo "Generating file, please wait..."

# overwrite file, This is bash specific, A better solution is cat > $FILE

for u in $USERS
do
p=change_1t_n0w
echo "$u:$p" >> $FILE # save USERNAME:PASSWORD pair
done
echo ""
echo "Random password and username list stored in $FILE file"
echo "Review $FILE file and once satisfied execute command: "
echo "chpasswd < $FILE"

# Uncomment following line if you want immediately update all users password,
# be careful with this option, it is recommended that you review $FILE first
# chpasswd < $FILE
```

17) A shell script to add mysql database, username and password

```
#!/bin/bash
# A shell script to add mysql database, username and password.
# It can also grant remote access on fly while creating the database.
_db="$1"
_user="$2"
_pass="$3"
_dbremotehost="$4"
_dbrights="$5"

## Path to mysql bins ##
mysql="/usr/bin/mysql"
## Mysql root settings ##
_madminuser='root'
_madminpwd='MySQL-PassWord'
_mhost='localhost'

# make sure we get at least 3 args, else die
[[ $# -le 2 ]] && { echo "Usage: $0 'DB_Name' 'DB_USER' 'DB_PASSORD'
['remote1|remote2|remoteN'] ['DB_RIGHTS']"; exit 1; }

# fallback to ALL rights
[[ -z "${_dbrights}" ]] && _dbrights="ALL"

# build mysql queries
_uamq="${mysql} -u "${_madminuser}" -h "${_mhost}" -p'${_madminpwd}' -e 'CREATE
DATABASE ${_db};'"
_upermq1="${mysql} -u "${_madminuser}" -h "${_mhost}" -p'${_madminpwd}' -e \"GRANT
${_dbrights} ON ${_db}.* TO ${_user}@localhost IDENTIFIED BY '${_pass}';\""


# run mysql queries
$_uamq
$_upermq1


# read remote host ip in a bash loop
# build queires to grant permission to all remote webserver or hosts via ip using
the same username
IFS='|'
for i in ${_dbremotehost}
do
_upermq2="${mysql} -u "${_madminuser}" -h "${_mhost}" -p'${_madminpwd}' -e \"GRANT
${_dbrights} ON ${_db}.* TO ${_user}@${i} IDENTIFIED BY '${_pass}';\""
$_upermq2
done
```

How Do I Use This Script?

Add a database called bar with username tom and password jerry, enter:

./script.sh bar tom jerry

Add a database called bar with username tom, password jerry and allow remote access from 192.168.1.5 and 192.168.1.11, enter:

./script.sh bar tom jerry '192.168.1.5|192.168.1.11'

Add a database called bar with username tom, password jerry, allow remote access from 192.168.1.5 & 192.168.1.11, and only grant SELECT,INSERT,UPDATE,DELETE, enter:

./script.sh bar tom jerry '192.168.1.5|192.168.1.11' 'SELECT,INSERT,UPDATE,DELETE'

18) A simple shell script to backup all MySQL Server Database

```
#!/bin/bash
# A simple shell script to backup all MySQL Server Database
# Dump all MySQL database every hour from raid10 db disk to /nas/mysql
# Each dump will be line as follows:
# Directory: /nas/mysql/mm-dd-yyyy
# File: mysql-DBNAME.04-25-2008-14:23:40.gz
# Full path: /nas/mysql/mm-dd-yyyy/mysql-DBNAME.04-25-2008-14:23:40.gz
# - Make sure NAS really mounted on $NAS
# --------------------------------------------------------------------------
NOW=$(date +"%m-%d-%Y") # mm-dd-yyyy format
FILE="" # used in a loop
NASBASE="/nas" # NAS Mount Point
BAK="${NAS}/mysql/${NOW}" # Path to backup dir on $NAS

### Server Setup ###
#* MySQL login user name *#
MUSER="root"

#* MySQL login PASSWORD name *#
MPASS="YOUR-PASSWORD"

#* MySQL login HOST name *#
MHOST="127.0.0.1"

#* MySQL binaries *#
MYSQL="$(which mysql)"
MYSQLDUMP="$(which mysqldump)"
GZIP="$(which gzip)"

# Make sure nas is really mounted
mount | awk '{ print $3}' |grep -w $NASBASE >/dev/null
if [ $? -ne 0 ]
then
echo "Error: NAS not mounted at $NASBASE, please mount NAS server to local
directory and try again."
exit 99
fi

### NAS MUST BE MOUNTED in Advance ###
# assuming that /nas is mounted via /etc/fstab
if [ ! -d $BAK ]; then
mkdir -p $BAK
else
:
fi

# get all database listing
DBS="$($MYSQL -u $MUSER -h $MHOST -p$MPASS -Bse 'show databases')"
```

```
# start to dump database one by one
for db in $DBS
do
FILE=$BAK/mysql-$db.$NOW-$(date +"%T").gz
# gzip compression for each backup file
$MYSQLDUMP -u $MUSER -h $MHOST -p$MPASS $db | $GZIP -9 > $FILE
done
```

19)Shell Script to Monitor Apache Process.

```
#!/bin/bash
# Apache Process Monitor
# Restart Apache Web Server When It Goes Down
# RHEL / CentOS / Fedora Linux restart command
RESTART="/sbin/service httpd restart"

# uncomment if you are using Debian / Ubuntu Linux
#RESTART="/etc/init.d/apache2 restart"

#path to pgrep command
PGREP="/usr/bin/pgrep"

# Httpd daemon name,
# Under RHEL/CentOS/Fedora it is httpd
# Under Debian 4.x it is apache2
HTTPD="httpd"

# find httpd pid
$PGREP ${HTTPD} &> /dev/null

if [ $? -ne 0 ] # if apache not running
then
     # restart apache
     $RESTART
else
     echo "HTTPD Process is already running with following pids "
     $PGREP ${HTTPD}
fi

# You may want to schedule it to run every few minutes to make sure that even if
Apache server goes down sometime, it is brought up automatically (using this
script) within minutes.
```

20) Shell Script To List All Top Hitting IP Address to your webserver

```
#!/bin/bash
# Shell Script To List All Top Hitting IP Address to your webserver.
# This may be useful to catch spammers and scrappers.

# where to store final report?
DEST=/var/www/reports/ips

# domain name
DOM=$1

# log file location
LOGFILE=/var/log/httpd/*$DOM-access_log-20121021             #latest access_log
file name

# die if no domain name given
[ $# -eq 0 ] && exit 1

# make dir
[ ! -d $DEST ] && mkdir -p $DEST

# ok, go though log file and create report
if [ -f $LOGFILE ]
then
echo "Processing log for $DOM..."
awk '{ print $1}' $LOGFILE | sort | uniq -c | sort -nr > $DEST/$DOM.txt
echo "Report written to $DEST/$DOM.txt"
fi
```

21) Simple iptables IP/subnet block script

```
---------------------------------------------------------------------------------
Note:-
Create /root/blocked.ips file as follows with list of ips and subnets to block
entering your dedicated server:

192.168.1.0/24
202.54.1.2
# spam
202.5.1.2
---------------------------------------------------------------------------------
Call following script from your existing shell script:

#!/bin/bash
# Simple iptables IP/subnet block script
IPT=/sbin/iptables
SPAMLIST="spamlist"
SPAMDROPMSG="SPAM LIST DROP"
BADIPS=$(egrep -v -E "^#|^$" /root/blocked.ips)

# create a new iptables list
$IPT -N $SPAMLIST

for ipblock in $BADIPS
do
$IPT -A $SPAMLIST -s $ipblock -j LOG --log-prefix "$SPAMDROPMSG"
$IPT -A $SPAMLIST -s $ipblock -j DROP
done

$IPT -I INPUT -j $SPAMLIST
$IPT -I OUTPUT -j $SPAMLIST
$IPT -I FORWARD -j $SPAMLIST
```

22) Script to take selective backup

```
#!/bin/bash

# This script is used to archive certain directories,
# move them to another machine and then mail the status of activity to
someone.


# This funcation varifies correctness of the Command Line Arguments
# If there is any error or warning, mention it in developer's log file and
the final report.
function verify_CLAs {

    if [ ! -f "$1" ]; then    # If first CLA is not a valid file, report
the problem and exit.
        echo "ERROR: The specified file\"$1\" is not a valid file." >>
$DEVLOG
        echo -e "Further processing is not possible. Exitting...\n\n\n"
>> $DEVLOG
        exit 251     # another non-zero exit status
    fi

# If first CLA is a valid file,but can't read from or write to it.

    if [ ! -r "$1" -o ! -w "$1" ]; then
        echo "ERROR: The specified file\"$1\" is not readable or writable
file." >>  $DEVLOG
        echo -e "Further processing is not possible till the permissions
are changed. Exitting...\n\n\n" >> $DEVLOG
        exit 251     # another non-zero exit status
    fi

# If the second CLA is not a valid directory,report the problem but do not
exit.

    if [ ! -d "$2" ]; then
        echo "WARNING: The backup directory does not exist. It will be
created during processing." >> $DEVLOG
    fi

# the backup directory exists, but is not writable.

    if [ -d "$2" -a ! -w "$2" ]; then
        echo "ERROR: Backup directory is not writable." >> $DEVLOG
        echo "Storing the tar files will not be possible. Exitting..." >>
$DEVLOG
        exit 252
    fi

}
```

```
# This function verifies that the directory to be backed up indeed exists.
function check_input_exists {
    if [ -d "$1" ]; then
        return 0     # zero indicates existance of directory
    else
        return 1     # non-zero indicates absence of directory
    fi
}


# This function keeps only those lines in the input file which start with
'/'.
# It also keeps the original contents in another file (to be restored to
original name later
function sanitize_input_file {
    cp "$1" backup_list.txt.tmp # Copy the original input file to a file
named backup_lsit.txt.tmp
    grep "^/" "$1" >  temp  # Select only those lines which start with '/'
i.e. first column is an absolute path.
    mv -f temp "$1"         # Rename the 'temp' file to the original file.
}


# This is the function which really takes the backup, i.e. make a tar and
move it to specified folder.
# If the specified folder is not present, it creates it.
# The process's messages are logged into developer's log and the report as
appropriate.
function backup {
    date_string=`date +%d_%m_%Y`
    fn=$1
    file_name=${fn//\//_}           #replace all occurances of '/' by '_'
    file_name=${file_name/_/}       #replace first '_' by nothing

    full_file_name=${file_name}_${date_string}.tar #build the tar file
name as specified in the requirement
    tar -cf $full_file_name $1 &> /dev/null #take the backup; ignore its
messages
                                            #like
#'removing leading /' or '... is an archive. not dumped'
    if [ $? -eq 0 ]; then           #if the tar command was successful,
then
        if [ ! -d "$2" ]; then      #if the backup directory does not
exist, then
            mkdir "$2"              #make the backup directory
        fi
        mv $full_file_name "$2"     #move the tar file to the backup
directory
        if [ $? -eq 0 ]; then       #if the movement was successful, then
            echo "Backed up file $full_file_name moved to backup
directory" >> $DEVLOG
            echo "`ls -l $2/$full_file_name | tr -s " " | cut -d" " -
f5,9`" >> $REPORT
```

```
        else
            echo "The $full_file_name could not be moved to backup
directory" >>  $DEVLOG
        fi
    else                             #take this route if the tar command
was not successful
        echo "The directory $1 could not be backedup" >> $DEVLOG
    fi

}

######################################
# Here starts the processing of this script.
# First it clears the screen, then checks for existance
#of two command line arguments.
# If the number of CLAs is not two,
#displays error message and exit
# If the number of CLAs is two, it verify them for
#correctness and then starts while loop to process each line from the
file.
######################################

DEVLOG=developer.log   # A log file for the developer of
                  #this script for debugging purpose.

REPORT=report.msg      # A file which will be mailed to concerned
                  #prsons ( may be 'awk' can be used to better format
this one)
> $DEVLOG #empty the previous contents
> $REPORT #empty the previous contents
remote_location=rahulm@172.24.0.240:/home/rahulm

clear

if [ $# -ne 2 ]; then      # If CLA is not two, show the usage and then
exit
    echo "ERROR: Incorrect number of Command Line Arguments. See usage
below."
    echo "Usage: $0 backup_list backup_directory."
    echo "Where -"
    echo "backup_list is a file having list of directories and directive
to back it up or otherwise."
    echo "backup_directory is the location to keep archives."
    echo -e "Further processing aborted. Exitting...\n\n\n"
    exit 250                # Some non-zero exit status
fi


verify_CLAs "$1" "$2"

sanitize_input_file "$1"
```

```
# Read the backup_list.txt file, one line at a time. Then determine from
its
#second column whether it is to be backed up. If so, check if the
directory
#indeed exists. If directory doesnt exist, make a note in a file. If
exists,
#process it(tar and move to /backup directory)

while read line
do
    echo "Line being processed is $line" >> $DEVLOG
    backup_flag=`echo $line | cut -d" " -f2`
    input_directory=`echo $line | cut -d" " -f1`

    if [ "$backup_flag" = "$input_directory" ]; then    # if the 'line'
contained only one word,
                                                        #f1 and f2 would
both return same value. Avoid those lines.
        echo "backup_flag and input_directory are same. $backup_flag and
$input_directory" >> $DEVLOG
        echo "------------------------------------------------------------
--" >> $DEVLOG
        continue
    fi

    if [ "Y" == "$backup_flag" -o "y" == "$backup_flag" ]; then
        check_input_exists $input_directory
        if [ $? -eq 0 ]; then
            backup $input_directory "$2"
        else
            echo "The directory $input_directory is not a valid directory.
Skipping..." >> $DEVLOG
        fi
    else
        echo "The directory $input_directory need not be backedup.
Skipping... " >> $DEVLOG
    fi
    echo "--------------------------------------------------------------"
>> $DEVLOG

done < "$1"

mv backup_list.txt.tmp backup_list.txt              # Rename the temporary
file to the original name.

tar_to_copy_to_remote=final_`date +%d_%m_%Y`.tar   #Name of the file to be
moved to
                                                   #remote machine


tar -cf $tar_to_copy_to_remote $2 >> $DEVLOG
scp $tar_to_copy_to_remote $remote_location &>> $DEVLOG
if [ $? -eq 0 ]; then           # If the scp command was successful
```

```
    echo "Successfully moved the tar of tars to remote machine.
$tar_to_copy_to_remote" >> $DEVLOG
    echo "final_file_name $tar_to_copy_to_remote" >> $REPORT
    rm $tar_to_copy_to_remote          # Remove the now-redundant file
fi


# The following 'awk' command prints header in BEGIN block. Then if the
first column in report.msg is not "final_file_name",
#it prints a table of file names and sizes.
# If the first first column is 'final_file_name', it prints corresponding
line. Then in the END block there are footer lines.
# CAUTION: There are two risky assumptions here. The 'final_file_name' is
hard-coded string in second
#'echo' statement after the success of 'scp' command. Also,
#according to current logic, it is the last line in the $REPORT file. This
'awk' script works based on these two.
#If a developer changes 'awk' logic, or any logic above, he is responsible
to keep both in synch.
awk -F" " '
BEGIN { printf "The following files are created as back up. Also listed
are their sizes in bytes.\n\n"
    printf "File name \t\t\t\t\t Size\n"
    printf "----------\t\t\t\t\t-----\n"
}
{ if ( $1 != "final_file_name" ) { printf "%s %25s\n", $2, $1
} else { printf "\n\nName of the file copied to remote location is %s", $2
}
}
END { printf "\n\nFor any queries, contact the system administrator or the
developer of the script which generated this report\n"
    printf "This is a system generated report, no signature required!\n"
} ' report.msg > report_formatted.msg
# send mail, having body as the formatted report,
#to the list mentioned in recepients.list

mail -s "backup status" `cat recepients.list` < report_formatted.msg

rm $REPORT report_formatted.msg     # Remove the now-redundant files.
```

24.2 Shell Scripting Assignment for Oracle

1. Backup of oracle database (RMAN)

```
#!/bin/sh
LOG_PATH=/u01/app/oracle/admin/rahul/logfile
export ORACLE_HOME=/u01/app/oracle/product/11.2.0/db_1
export ORACLE_SID=rahul
echo
echo Performing HOT Backup using RMAN
echo
$ORACLE_HOME/bin/rman target / log=$LOG_PATH/rman_hot_backup.log <<EOF
run {
     report schema;
     backup database plus archivelog delete input;
     delete noprompt obsolete;
    }
exit;
EOF

var1=`grep -c "ORA" $LOG_PATH/rman_hot_backup.log`
if [ ! $var1 -eq 0 ]; then
        mail -s "Backup Failed" rahul@focustraining.in
else
        mail -s "Backup Succesfully Completed" rahul@focustraining.in
fi
```

2. Shell script to take cold backup.

```
#!/bin/sh
DT=`date +%Y.%m.%d`
export ORACLE_BASE=/u01/app/oracle
export ORACLE_HOME=/u01/app/oracle/product/10.2.0/db_1
export ORACLE_SID=rahul
export BACKUP_DIR=$ORACLE_BASE/admin/$ORACLE_SID/exp/$DT
export TMP_DIR=/tmp
echo "Creating backup directory $BACKUP_DIR"
mkdir $BACKUP_DIR
echo
echo Backup for $DT
echo Selecting the files that must be backed up....
echo

$ORACLE_HOME/bin/sqlplus -S '/ as sysdba' <<EOFSQL >/dev/null 2>&1
set termout off
set pages 0
set lines 120
set feedback off
set trimspool on
spool $TMP_DIR/files.2.backup
select name    from v\$datafile;
select name    from v\$controlfile;
select member from v\$logfile;
select '$PFILE' from dual;
select '$LISTENER_FILE' from dual;
select '$TNSNAMES_FILE' from dual;
spool off
shutdown immediate
exit
EOFSQL

echo
echo "Database successfully shut down"
echo "now backing up files to $BACKUP_DIR"
echo


for i in `cat $TMP_DIR/files.2.backup`
do
   cp $i $BACKUP_DIR/
   echo "back-ed up : $i "
done

echo
echo "Backup successful"
echo "Restarting database "
echo
$ORACLE_HOME/bin/sqlplus -S '/ as sysdba' <<EOFSQL
startup
exit
EOFSQL
```