UNIX Shell Scripting

Focus Training Services

Compiled and Edited By

Mr. Mahesh Chadare

Version 2.0

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.

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 found in businesses, sciences, academia, and industry.
Many telecommunications 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 very 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.

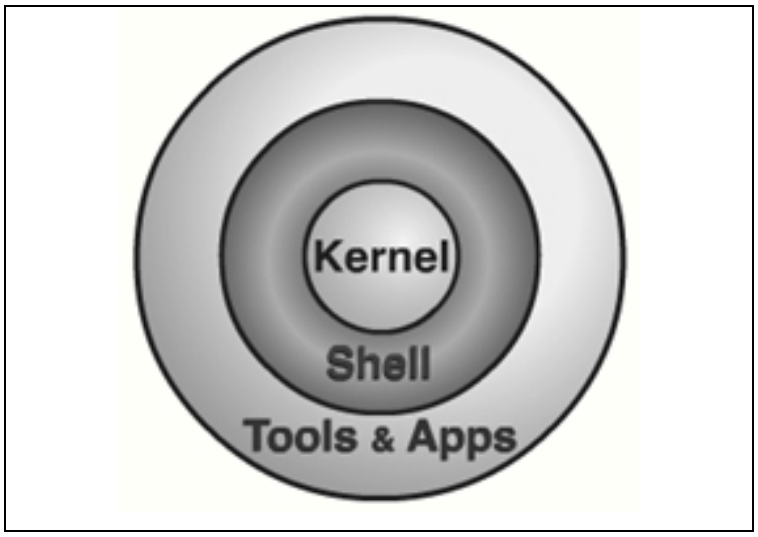
- Unix provides a consistent format for files, i.e. the byte stream, which aids in the implementation of application programs. This also provides a consistent interface for peripheral devices.

1.4 How UNIX is organized

The UNIX system is functionally organized at three levels:

The kernel, which schedules tasks and manages storage;

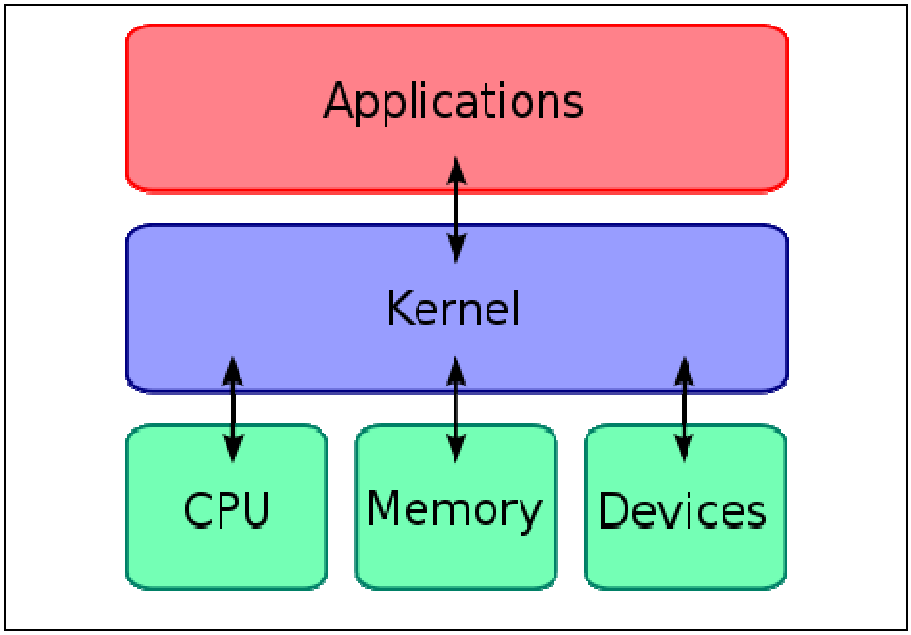
The shell, which connects and interprets users' commands, calls programs from memory, and executes them and ,the tools and applications that offer additional functionality to the operating system



1.4.1 The kernel

Kernel is heart of Unix Os.

It manages resource of Unix Os. Resources means facilities available in Unix. For e.g. Facility to store data, print data on printer, memory, file management etc .



- Kernel decides who will use this resource, for how long and when. It runs your programs (or set up to execute binary files).
- The kernel acts as an intermediary between the computer hardware and various programs/application/shell.
- The kernel controls the hardware and turns part of the system on and 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, instruction are provided using binary language, which is difficult for all of us, to read and write. So in OS 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 a command language interpreter that executes commands read from the standard input device (keyboard) or from a file.

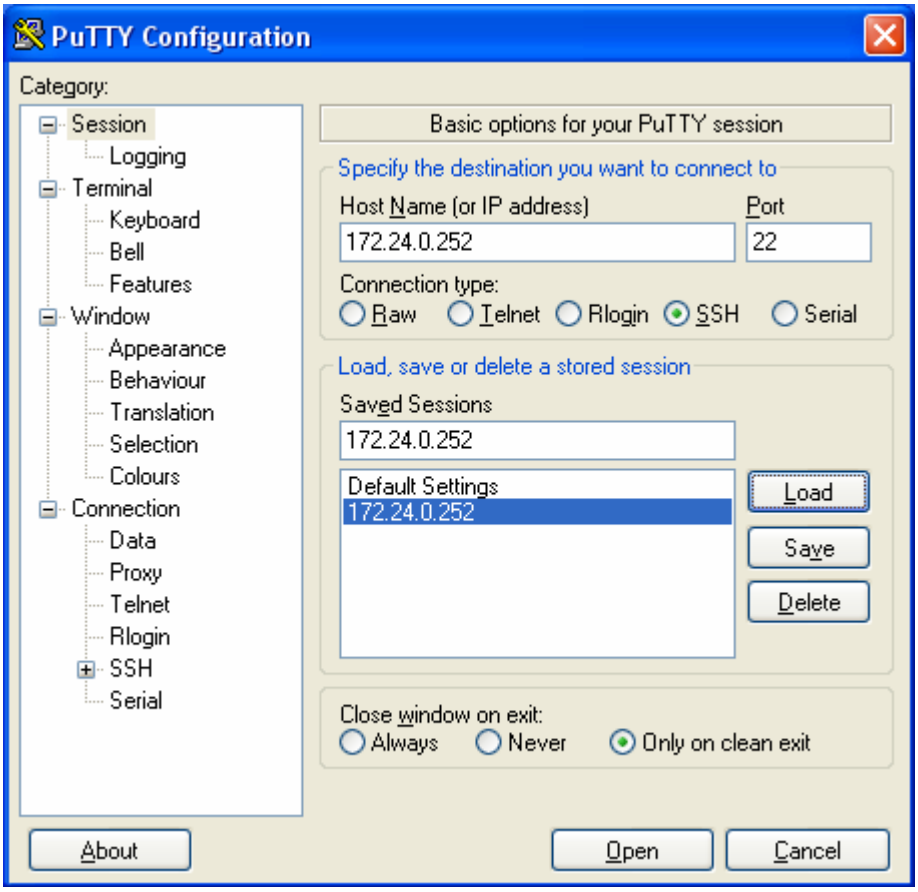
Shell is not part of system kernel, but uses the system kernel to execute programs, create files etc.

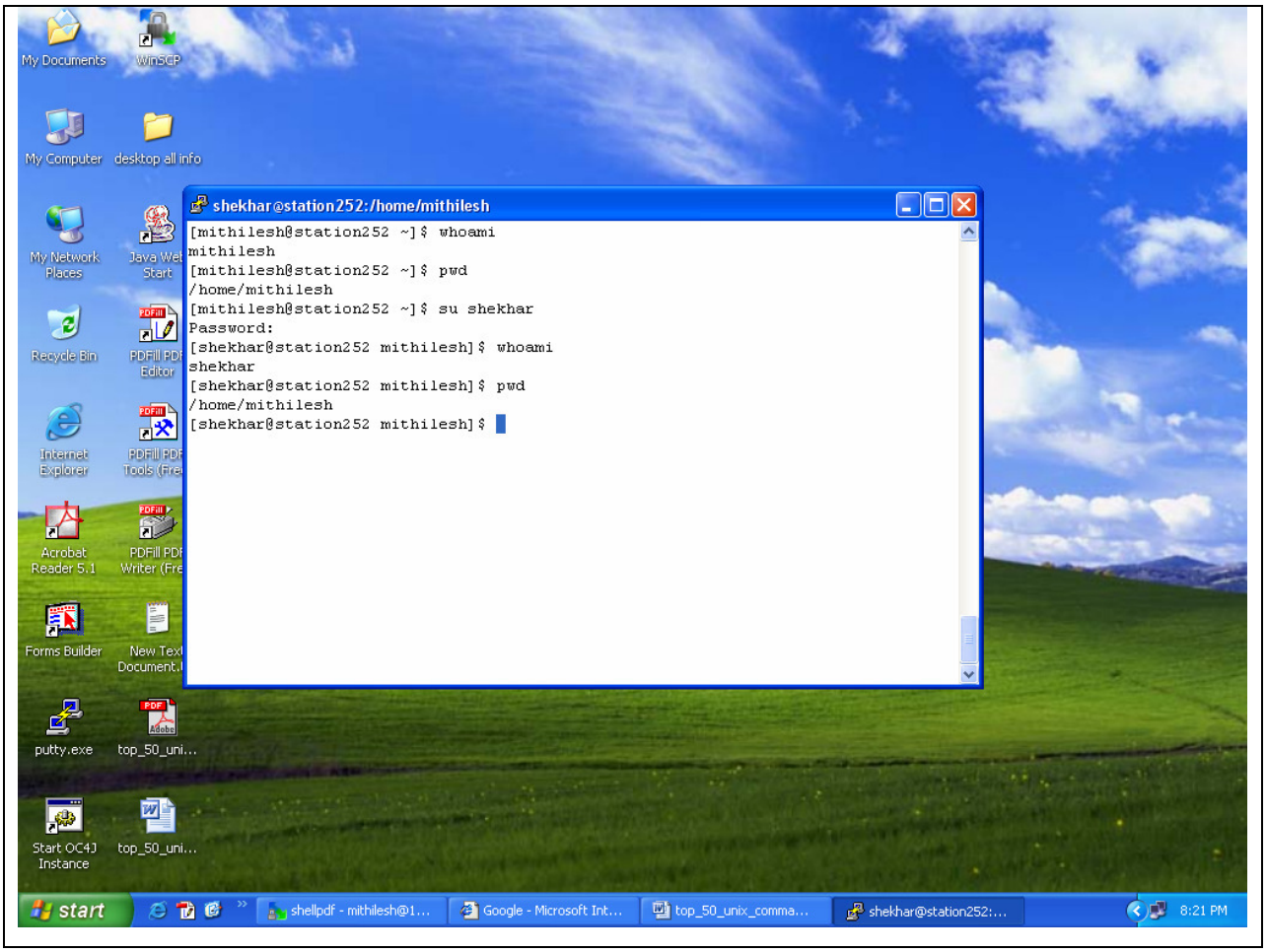
2. Basic Unix Commands

1.1 How to login

1.1.1 putty

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





1.1.2 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 usermust 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

1.2 find information about your system

1.2.1 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 effective userID.

```
[mahesh@station252 ~]$ whoami
 mahesh
[mahesh@station252 ~]$ 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
============================================================
[shekhar@station252 ~]$
```

```
[shekhar@station252 ~]$ whoami
shekhar
[shekhar@station252 ~]$
```

1.2.2 who

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

It displays each users

- 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 logged 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 their user id's so can mail them

messages. The who command displays the informative listing of users.

```
[root@sql ~]# 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 username of users who are logged on server.
- 2nd column shows device names of their respective terminal.These arethe filenames associated with the terminals.(mahesh's terminal is pts/1).
- 3rd,4th,5th column shows date and time of logging in.

  Last column shows machine name/ip from where the user has logged in.

  It has more options which can be used.

-H → The option prints the column headers.

-u → The option prints with some more details like PID, IDLE time.

-b→ Indicate the time and date of the last reboot.

1.2.3 w

Show who is logged and what they are doing.

UNIX maintains an account of all users who are logged on to system but along with that,,it also shows what that particular user doing on his machine.

It also displays information about the users currently on the machine, and their processes.

The header shows, in this order, 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.

```
[mahesh@station60 ~]$ 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.

1.2.4 uname

knowing your machine 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]

```
[mahesh@station60 ~]$ uname
Linux
[mahesh@station60 ~]$
```

Linux system simply shows Linux.

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

Command Options.

Current release(-r).

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

```
[mahesh@station60 ~]$ uname –r
2.6.18-194.el5
[mahesh@station252 ~]$
```

-a It will display everything related to your machine.

It will show you following key points.

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

```
[mahesh@station60 ~]$ 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
[mahesh@station60 ~]$
```

-i Print the name of the hardware implementation (platform).

-n Print the nodename (the nodename is the name by which the system is known to a communications network).

1.2.5 uptime

Tell 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.

```
[mahesh@station60 ~]$ uptime
18:56:15 up 19:32, 8 users, load average: 1.60, 1.11, 0.63
[mahesh@station60 ~]$
```

1.2.6 users

It prints only usernames of current users who are logged in to the current host(server).

```
 [root@sql ~]# 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
[root@sql ~]#
```

1.2.7 date

The date command can be used to display or set the date. If a user has superuser privileges, he or she can set the date by supplying a numeric string with the following command:

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
```

Set Date and Time

date [-s datestr]

-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 an example of how the date and time can be set.

Examples

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

Set the date to the date and time shown.

1.2.8 cal

Print 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. With no arguments, print a calendar for the current month.

Before we can do the calendar program we must have a file named calendar at the root of your profile. Within that file we may have something similar to:

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 us want the calendar to be displayed. Must be the numeric representation of the

month. For example: January is 1 and December is 12.

year Specifies the year that we want to be displayed.

Examples

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

1.2.9 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.

[root@station79 ~]# 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.

1.2.10 hostname

hostname command simply displays the fully qualified name of computer.

```
[mahesh@station60 ~]$ hostname
station60.example.com
[mahesh@station60 ~]$
```

1.2.11 free

Display 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.

```
[root@sql ~]# free
             total       used       free     shared    buffers     cached
Mem:       2055768    1965776      89992          0     138664    1116936
```

```
-/+ buffers/cache:          710176   1345592
Swap:     4194296               0   4194296
[root@sql ~]#
```

In above output the memory description which is displayed it is in bytes.If user wants to display it in required format that is in GB,MB or KB.

Command Options.

- $ free -k

It will show the output in Kilobytes.

- $ free -g

It will show the ouput in Gegabytes.

- $ free -m

It will show the output in Megabytes.

1.2.12 df -h

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

```
[mahesh@station60 ~]$ 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
```

2 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 systems provide 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. Similar to the concept of the process parent-child relationship, all files on a Unix system are related to one another.

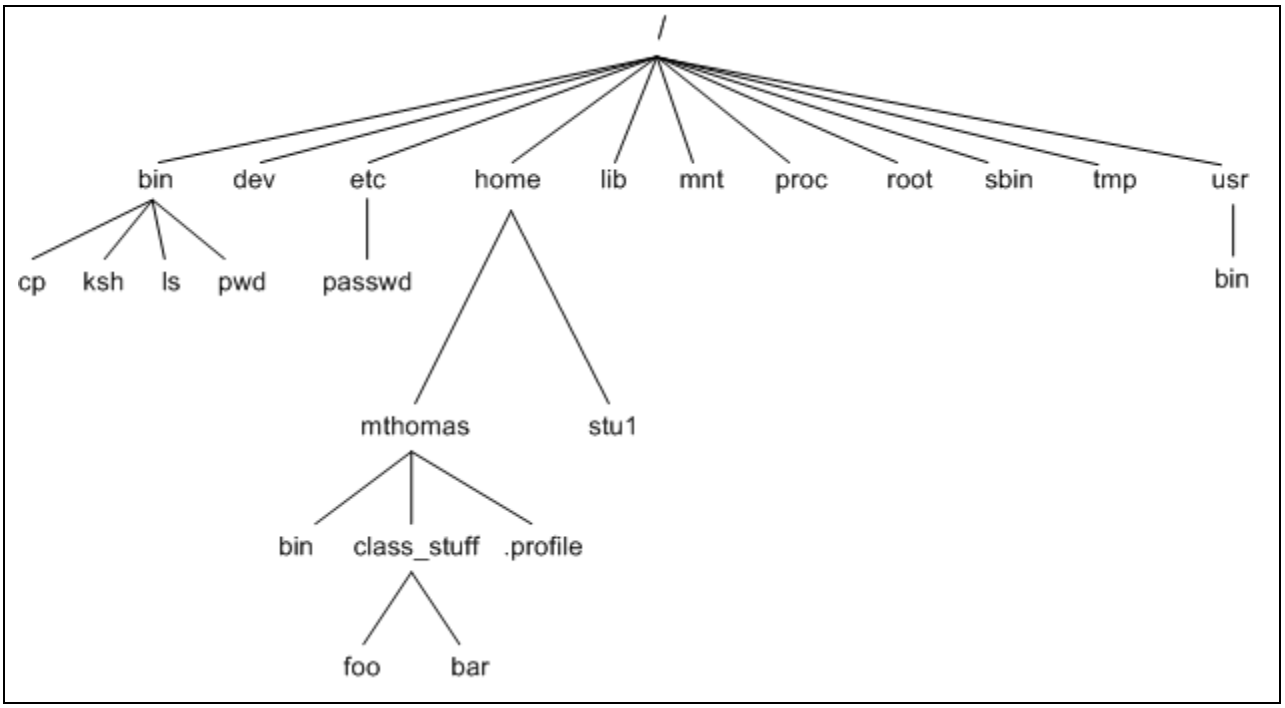
That is, files also have a parent-child existence. Thus, all files (except one) share a common parental link, the top-most file (i.e. /) being the exception.

Below is a diagram (slice) 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

- Making files available to users.
- Managing and monitoring the system's disk resources.
- Protecting against file corruption, hardware failures, user errors through backup.
- Security of these filesystems, what users need and have access to which files.
- Adding more disks, tape drives, etc when needed.

When Unix operating systems is installed, some directories depending upon the Unix being installed are created under / (or root), such as /usr /bin /etc /tmp /home /var.



- **etc** Contains all system configuration files and the files which maintain information about users and groups.
- **bin** Contains all binary executable files (command that can be used by normal user also)
- **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 will be removed when the server reboots.
- **dev** Contains all device files i.e. logical file names to physical devices.
- **home** - contains user directories and files
- **lib** - contains all library files
- **mnt** - contains device files related to mounted devices
- **proc** - contains files related to system processes
- **root** - the root users' home directory (note this is different than /)
- **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 (command which generally normal users don't have the privileges)

3 File and Directory commands

3.1 File Related Commands

3.1.1 ls

The ls command lists the files in your current working directory. When we log onto your account on UNIX, 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 options available.

Example: 1

```
[mahesh1@station60 ~]$ ls
case.sh  for.sh  hello.sh  if.sh
[mahesh1@station60 ~]$
```

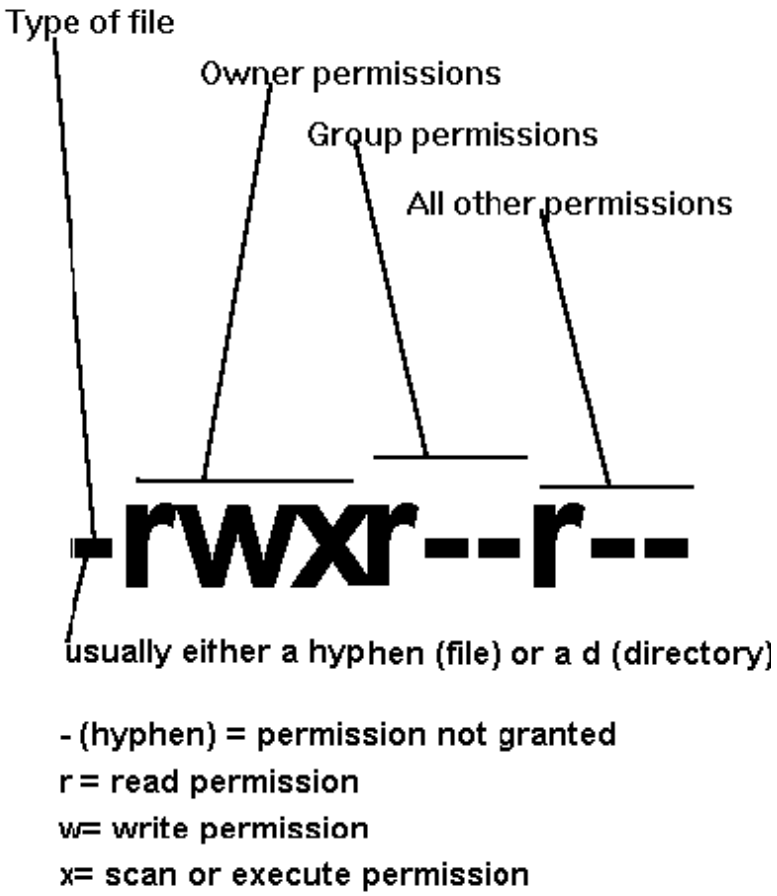
the ls command without any option displays the files and directories

-l displays the long listing of files

Example: 2

```
[mahesh1@station60 ~]$ 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
[mahesh1@station60 ~]$
```

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 in at the UNIX prompt. It also means that when a directory which, to UNIX is a file like any other file, can be "scanned" to see what files and sub-directories are in it. Privileges are granted to three levels of users:



_ **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. At GSU, the group is usually, but not always designated as the first three letters of the userid of the owner.

_ **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
[mahesh1@station60 ~]$ ls -ld /usr
drwxr-xr-x 16 root root 4096 Sep 20 20:06 /usr
[mahesh1@station60 ~]$
```

Rather than list the files contained in the /usr directory, this command lists information about the /usr directory itself (without generating a listing of the contents of /usr). 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 (these files begin with a dot.)

Example: 4

```
[mahesh1@station60 ~]$ ls -a
 .bash_logout   .bashrc  .emacs  hello.sh  .kde .bash_profile
case.sh  for.sh  if.sh     .mozilla
[mahesh1@station60 ~]$
```

3.1.2 cat

The cat command is used to read a file and also to create

Examples:

a)creating a file

```
[mahesh1@station60 ~]$ cat >hello.txt
hi
welcome to unix                    Enter text and end with ctrl-D
[mahesh1@station60 ~]$
```

b)reading a file

```
[mahesh1@station60 ~]$ cat hello.txt
Hi
welcome to unix
[mahesh1@station60 ~]$
```

3.1.3 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 screenful at a time.

I will show you the main uses of **more** here.

3.1.3.1 To View a File Using more

[mahesh1@station60 ~]$ more fur.sh

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

```
1 #!/bin/bash
2 beep 659 120  #  Treble E
3 beep 0 120
4 beep 622 120  #  Treble D#
5 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 space, then more will move down the file the height of the terminal window you have open, to display new information to you.

3.1.3.2 Pipe Output From 'cat' Into more

Sometimes you want to output an entire file, but view it slowly. For instance you may want to view a README file before doing a build or an application.

```
[mahesh1@station60 ~]$  cat data.txt | more
```

This will then display the data.txt file but use the **more** processor to view the file at your own pace.

3.1.3.3 View Two Files At The 20th Line Using more -p

If you want to see the 20th line of each file:

```
[mahesh1@station60 ~]$  more -p 20 fur.sh data.sh
```

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

3.1.3.4 View A File Starting With Line Containing Using more -p /

```
[mahesh1@station60 ~]$  more -p /Erik data.txt
```

This will display from the first line that contains Erik in it. This is quite useful if you are looking at patch files and know the filename you are looking for.

3.1.4 cp

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

```
[mahesh1@station60 ~]$ cp file1 file2
```

if the destion file (file2) does not exist ,it will first be created before coping takes place.If not it will simply be overwritten without any warinig from the system.

2.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.

```
[mahesh1@station60 ~]$cp /home/mahesh/userlist.txt .
```

3.cp can also be used to copy more than one file 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.

```
mahesh1@station60 ~]$cp file1 file2 file3 backup
```

Where the backup is a directory.

4.you can use wildcard * as follows

```
mahesh1@station60 ~]$cp file* backup
```

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

3.1.5 mv

mv command has two distincts functions.

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

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

```
[mahesh1@station60 ~]$ mv hello.sh welcome.sh
[mahesh1@station60 ~]$
```

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

To move the group of files to a directory.The following command moves files to a backup directory

3.1.6 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.

```
mahesh1@station60 ~]$rm file1 file2
Note: mahesh1@station60 ~]$ rm file*
```

command will dangerous to use in this case it will remove all the files whoes name starts with file and end with any other characters.

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

```
mahesh1@station60 ~]$rm *
```

-i option:

-i option warns the user before deleting the files.

```
[mahesh1@station60 ~]$ rm –i for.sh
 rm: remove regular empty file `for.sh'? y
[mahesh1@station60 ~]$
```

3.1.7 Using the VI editor

The VI editor is a screen–based text editor available on all Unix computers (and available for all other kinds of computers). Given that it takes some effort to learn, 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, save yourself headaches and learn the basics of VI now! In the following, **^X** denotes a control character. 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.

Starting: To edit a file named (say) "`mytext`" on a Unix computer, type the Unix command "**`vi mytext`**". Note that you must type the command with lowercase letters.

Two Modes: Pay attention, this is the crucial feature of VI! There are two **modes**, *command* and *insert*. When in ***insert mode***, everything you type appears in the document at the place where

the blinking cursor is. When in ***command mode***, keystrokes perform special functions rather than actually inserting text to the document. (This makes up for the lack of mouse, menus, etc.!) You must know which keystroke will switch you from one mode to the other:

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

Getting out: When you want to get out of the editor, switch to command mode (press **Esc**) if necessary, and then

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

Moving Around: 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

`G` to the last line

Inserting Text: From command mode, these keystrokes switch you into insert mode with new text being inserted

i just before the current cursor position

a just after the current cursor position

o into a new line below current cursor

I at the beginning of the current line

A at the end of the current line

O into a new line above current cursor

Cutting, Copying, 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
- **dd** delete (cut) current line
- **4dd** delete (cut) four lines
- **D** delete to the end of the line from the cursor
- **dw** delete to the end of the current word
- **yy** copy (without cutting) current line
- **5yy** copy (without cutting) 5 lines
- **p** paste after current cursor position/line
- **P** paste before current cursor position/line

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 next occurrence of the string "wavelet"
- **?wavelet Rtn** jump backward to the previous occurrence of the string "wavelet"
- **n** repeat the last search given by "**/**" or "**?**"

Replacing Text: This amounts to combining two steps; deleting, then inserting text.

- **r** replace 1 character (under the cursor) with another character
- **8r** replace each of the next 8 characters with a given character
- **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** replace entire line with typed input (ended with **Esc**)
- **4S** replace 4 lines with typed input (ended with **Esc**)
- **cw** replace (remainder of) word with typed input (ended with **Esc**)

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". 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)

^L clear and redraw the screen

:%s/Joe/Bob/g Rtn change every "Joe" to "Bob" throughout the document

J join this line to the next line

5J join 5 lines

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

4 What happens when we login/logout

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

4.1 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 user commands that we type at the shell prompt and the UNIX 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 diagram, 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.

4.2 .bash_profile is executed

The second thing that happens when you login is that a special file named .bash_profile automatically gets 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 everytime you login. Usually commands like “alias” are written in his .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 "----------------------------------------"
```

4.3 You are put in your HOME directory

A HOME directory is a directory is assigned to every user by the System Administrator. You can find the name of the home of 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 b 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
----------------------------------------
[root@shekhar ~]# pwd
/root  ← This is your HOME directory
[root@shekhar ~]#
```

5 Introduction to Shell Programming

5.1 What is shell

Computer understand the language of 0's and 1's called binary language.

In early days of computing, instruction are provided using binary language, which is difficult for all of us, to read and write.

So in Os there is special program called Shell. Shell accepts your instruction or commands in English (mostly) and if its a valid command, it is passed to kernel.

Shell is a user program or it's a environment provided for user interaction. Shell is an command language 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 to execute programs, create files etc.

5.2 Shell types

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

```
[root@station60 ~]# cat /etc/shells
/bin/sh
/bin/bash
/sbin/nologin
/bin/tcsh
/bin/csh
/bin/ksh
[root@station60 ~]#
```

5.2.1 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.

5.2.2 bash or Bourne Again shell:

the standard GNU shell, intuitive and flexible. Probably most advisable for beginning users while being at the same time 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**.

5.2.3 csh or C shell:

the syntax of this shell resembles that of the C programming language. Sometimes asked for by programmers.

5.2.4 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.

5.2.5 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.

5.3 What is Shell Script ?

Normally shells are interactive. It means shell accept command from you (via keyboard) and execute them. But if you use command one by one (sequence of 'n' number of commands) , the you can store this sequence of command to text file and tell the shell to execute this text file instead of entering the commands. This is know as shell script.

Shell script defined as:

"Shell Script is series of command written in plain text file. Shell script is just like batch file is MS-DOS but have more power than the MS-DOS batch file."

Shell scripting allows us to use commands we already use at the command line.

We are familiar with the interactive mode of the shell. Almost anything can be done in a script which can be done at the command line.

5.4 When to write shell scripts

Shell scripting can be applied to a wide variety of system and database tasks.

5.4.1 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

5.4.2 Occasional Tasks

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

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

5.4.3 Complex Manual Tasks

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

- Checking for database locks
- Killing runaway processes

5.4.4 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)

5.5 Our First Shell Script

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

$ vi hello.sh

1. insert the following shell command in the hw file:

```
#!/bin/bash

echo "Hello World!"
```

2. save and exit the editor program
3. run the hw program

   $ hello.sh *[Enter]*
   bash: hello.sh: cannot execute - Permission denied

   What is the problem here, and how do we fix it?

4. The problem is that you are not having execute permission on file

   so let us give execute permission on this file

   $chmod u+x hello.sh

5. once the problem above 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]*
   Hello World!

#!/bin/bash ← This first line indicates what interpreter to use when running this script

The Shebang (#!)

The "shebang" is a special comment. Since it is a comment it will not be executed when the script is run. Instead before the script is run, the shell calling the script will check for the #! pattern. If found it will invoke the script using that interpreter. f no #! is found most shells will use the current shell to run 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 shell explicitly like this assures that the script will be run with the same interpreter regardless of who executes it (or what their default shell may be.)

6 Shell Variables

The variable is place holder for storing data. The value of the variable can be changed during the program exucution.

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.

Shell variables can be used at the command line and/or used within shell programs.

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

There are two types of variables

6.1 User defined variables

6.1.1 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:

```
[mahesh@station60 ~]$no=10  # this is ok
```

```
[mahesh@station60 ~]$10=no # Error, NOT Ok, Value must be on
right side of = sign.
```

To define variable called 'os' having value unix

```
[mahesh@station60 ~]$os=unix
```

To define variable called n having value 10

```
[mahesh@station60 ~]$ n=10
```

6.1.2 Rules for Naming variable name (Both UDV and System Variable)

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

```
HOME
SYSTEM_VERSION
Vechno
```

(2) Don't put spaces on either side of the equal sign when assigning value to variable. For e.g. In following variable declaration there will be no error

```
$ no=10
```

But there will be problem for any of the following variable declaration:

```
$ no =10
$ no= 10
$ no = 10
```

(3) Variables are case-sensitive, just like filename in Linux. For e.g.

```
$ no=10
$ No=11
$ NO=20
$ nO=2
```

Above all are different variable name, so to print value 20 we have to use $ echo $NO and not any of the following

```
$ echo $no # will print 10 but not 20
$ echo $No # will print 11 but not 20
$ echo $nO # will print 2 but not 20
```

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

(5) Do not use ?,* etc, to name your variable names.

6.1.3 How to access the value of variable

Use a $ to access the value of a variable.

```
[mahesh@station60 ~]$ echo $a
10
[mahesh@station60 ~]$
```

6.1.4 Variable are not declared

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

```
[mahesh@station60 ~]$ echo $b

[mahesh@station60 ~]$
```

In above example note the b is not having any value assigned hence it returns null.

6.1.5 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

```
[mahesh@station60 ~]$ user=`whoami`
[mahesh@station60 ~]$ echo $user
mahesh
[mahesh@station60 ~]$
```

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

```
[mahesh@station60 ~]$ user=$(whoami`)
[mahesh@station60 ~]$ echo $user
mahesh
[mahesh@station60 ~]$
```

6.1.6 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. Not so fast! The standard tried and true 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
```

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 $ on X
  12
```

6.2 System variables -

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

6.2.1 PATH variable:

PATH variable simply hold the path of your system.When we type any command on prompt cheks the executable in path of the system for excution of the command.If it doesn't get the path it displays error as command not found.

See how to set a path variable.

```
[mahesh1@station60 ~]$ 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 sysem observe the bin is present in current path.

Now type the following command

```
[mahesh1@station60 ~]$ which ls
/bin/ls
[mahesh1@station60 ~]$
```

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.

```
[mahesh1@station60 ~]$ ls
a.txt  case.sh  for.sh  hello.sh  hello.txt  if.sh
[mahesh1@station60 ~]$
```

now remove the /bin from path as follows

```
[mahesh1@station60]$PATH=/lib/qt3/bin:/usr/kerberos/bin:/usr/loc
al/bin:/usr/bin:/opt/real/RealPlayer:/home/mahesh1/bin
```

now /bin is not present in path hence we can not execut the ls command.

```
[mahesh1@station60 ~]$ ls
-bash: ls: command not found
[mahesh1@station60 ~]$ ls
```

now set your path variable as it is you will be able to execut the ls command .

By assigning path of your shell script you can execut without giving full path like an executable command.

6.3 All about quotes

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

```
$ echo The $? variable returns the child exit status [Enter]
```

The 0 variable returns the child exit status

where what we really want is:

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

The $? variable returns the child exit status

6.3.2 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. Note that these are not the same character as the single quote mark. 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. This enables one do the following:

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

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

6.3.3 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`
```

6.3.4 Double Quotes

Double quotes or front quotes (again, not to be confused with the back quotes or 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 with single quotes 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 situation may arise where a command results in single greater than (>) character displayed follows:

```
$ 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 entered 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 characters missing. Otherwise, you may want to punt with a *[Ctrl-c]* sequence.

7 Additional commands for our shell script

7.1 tar

tar command background

The name "tar" stands for "tape archive". As the name implies, in the old days it was a command that Unix administrators used to deal with tape drives. Where we now use the Linux tar command to create a tar file, we used to tell it to write the tar archive to a device file

(in dev).

These days the Linux 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 another user may extract those files, with both of them using the tar command.

7.1.1 Linux tar command - Create an archive of a subdirectory

A common use of the Linux 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 letter c means "create archive".
- The letter v means "verbose", which tells tar to print all the filenames as they are added to the archive.
- The letter f tells tar that the name of the archive appears next (right after these options).

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
```

7.1.2 Linux tar command with gzip - 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 gzip'd tar archive with one 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.

7.1.3 Creating a compressed archive of the current directory

Many times when using the Linux 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.

tar command example - creating an archive in a different directory

You may also want to create a new tar archive like that previous example 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 tar archive to specify what directory the archive should be created in.

7.1.4 tar list example - 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
```

That same command can also work on a file that was tar'd and gzip'd in two separate steps (as indicated by the .tar.gz file extension):

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

I almost always list the contents of an unknown archive before I extract the contents. I think this is always good practice, especially when you're logged in as the root user.

7.1.5 tar extract example - extracting an archive

To *extract* the contents of a Linux 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
```

or 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).

7.2 The read Statement

Use to get input (data from user) from keyboard and store (data) to variable.

Syntax:

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

Following script first ask user, name and then waits to enter name from the user via keyboard. Then user enters name from keyboard (after giving name you have to press ENTER key) and entered name through keyboard is stored (assigned) to variable fname.

```
$ vi read.sh
#
#Script to read your name from key-board
#
echo "Your first name please:"
read fname
echo "Hello $fname, Lets be friend!"
```

Run it as follows:

$ chmod 755 hello.sh

$ ./sayH

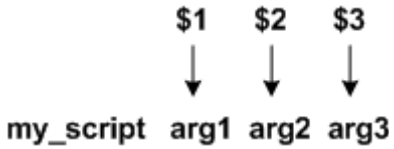
Your first name please: vivek

Hello vivek, Lets be friend!

8 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 a input to shell script called as command line arguments.

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



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
arg1
```

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:

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 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 parameter specifies 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 separate arguments.

We can write the shell script shown below to process an unknown number of command-line arguments with either the $* or $@ special parameters:

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 number 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 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).

Let's take rm command, which is used to remove file, but which file you want to remove and how you will tell this to rm command (even rm command don't ask you name of file that you would like to remove). So what we do is we write command as follows:

```
$ rm  {filename}
```

Here rm is command and filename is file which you would like to remove. This way you tail rm command which file you would like to remove. So we are doing one way communication with ourcommand by specifying filename. Also you can pass command line arguments to your script to make itmore users friendly. But how we access command line argument in our script.

Lets take ls command

```
$ ls –a /*
```

This command has 2 command line argument -a and /* is another. For shell script,

```
$ myshell foo bar
```

Shell Script name i.e. myshell

First command line argument passed to myshell i.e. foo

Second command line argument passed to myshell i.e. bar

In shell if we wish to refer this command line argument we refer above as follows

```
      myshell it is $0
      foo it is $1
      bar it is $2
```

Here $# (built in shell variable) will be 2 (Since foo and bar only two Arguments), Please note 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 is 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 $@"
```

$ vi isnump_n

9 Exit Status:

The **$?** variable represents the exit status of the previous 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 if they were successful, and other than o (most of the time 1)if they were unsuccessful.

Some commands return additional exit statuses for particular reasons. For example, some commands differentiate between kinds of errors and will return various exit values depending on the specific type of failure.

Following is the example of successful command:

```
[mahesh1@station60 ~]$ ls
a.txt  case.sh  for.sh  hello.sh  hello.txt  if.sh
[mahesh1@station60 ~]$ echo $?
0
[mahesh1@station60 ~]$
```

Following is example of unsucessful command

```
[mahesh1@station60 ~]$ cat abcd
cat: abcd: No such file or directory
[mahesh1@station60 ~]$
echo $?
1
[mahesh1@station60 ~]$
```

10 Advanced UNIX commands(Filters)

A filter is a Unix command that does some manipulation of the text of a file

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 more useful Unix filters.

10.1 head

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

```
[root@station60 ~]# 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:
 [root@station60 ~]#
```

We can override the default behaviour of the head command as follows

```
[root@station60 ~]# 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

[root@station60 ~]#
```

Note :Instead of 5 we can give any number regardless of it is less or greater than 10.

10.2 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 behaviour as follows.

```
[root@station60 ~]# 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

[root@station60 ~]#
```

suppose I want to retrive a line on particular position from the file then combination of head and tail commands can be used as follows

```
[root@station60 ~]# cat /etc/passwd|head -50|tail -1
mahesh:x:523:501::/home/mahesh:/bin/bash
[root@station60 ~]#
```

in above the 50th line from /etc/passwd file will be displayed.

10.3 wc

In Unix, to get the line, word, or character count of a document, use the wc command. At the Unix shell prompt.

wc filename Replace file name with the file or files for which we want information. 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

[root@station60 ~]# wc login

the output would be something similar to the following:

38 135 847 login

To narrow the focus of your query, 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. However, in most cases, the values for -c and -m are equal.

Syntax:

To count the characters in a file. Here it counts the no of characters in the file abc.txt

```
$ Wc –c / abc.txt
```

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

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

We may also pipe standard output into wc to determine the size of a stream. For example, to find out how many files are in a directory, enter:

```
/bin/ls -l | wc -l
```

10.4 Sort

sort is a standard Unix command line program that prints the lines of its input or concatenation of all files listed in it's argument list in sorted order. The r flag will reverse the sort order.

1) 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
```

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

1) The n option makes the program to sort according to numerical value:

```
$ du /bin/* | sort n
4      /bin/domainname
4     /bin/echo
4     /bin/hostname
4     /bin/pwd
...
24    /bin/ls
30    /bin/ps
44    /bin/ed
54    /bin/rmail
80    /bin/pax
102  /bin/sh
304  /bin/csh
```

2) If the first column of file does not contains numerical data then it will not sort according to numbers we have to provide the position of column by using k option

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

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

3) k will work when there the column seprator is space. If the delimiter is other than space then use t option

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

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

4) The r option just reverses the order of the sort:

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

10.5 cut

cut is a Unix command which is typically used to extract a certain range of characters from a line, usually from a file.

Syntax

```
cut [c][flist] [ddelim] [file]
```

Flags which may be used include

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

-c Characters: a list following c specifies a range of characters which will be returned,

Examples:

1) If you want to print just columns 1 to 6 of each line (the employee serial numbers), use the c16 flag, as in this command:

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

1) If you want to print just columns 4 and 8 of each line(the first letter of the department and the fourth digit of the serial number), use the c4,8 flag, as in this command:

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

-f Specifies a field list, separated by a delimiter list A comma separated or blank separated list of integer denoted fields, incrementally ordered.

The indicator may be supplied as shorthand to allow inclusion of ranges of fields

-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 in use must be enquoted or escaped as necessary.And since this file obviously has fields delimited by colons, we can pick out just the last names by specifying the d and f3 flags, like this:

Examples using d and f options

1) If you want to access single field.

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

2) If you want to access multiple fields.

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

10.6 paste

Paste is a Unix utility tool which is used to join files horizontally (parallel merging), e.g. to join two similar length files which are comma delimited. It is effectively the horizontal equivalent to the utility cat command which operates on the vertical plane of two (or more) files, i.e. by adding 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 names file in one column, the places file in another,and the dates file in a third. If the names, places, and dates file look like:

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

then the www file will contain:

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

10.7 grep

"grep" one of the most frequently used TEXT PROCESSING TOOLS 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 e the matching lines.

1) If you want to count of a particular word in log file
   you can use grep 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
```

2) Sometime we are not just interested on matching line but also on lines around matching lines particularly useful to see what happens before any Error or Exception. grep –context option allows us toprint lines around matching pattern. Below example of grep command in UNIX will print 6 lines around matching line of word "successful" in logfile.txt

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

Show additional six lines after matching 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 –C 'hello' logfile
```

Prints two lines of context around each matching line.

3) If you want to do case insensitive search than use i
option from grep command in UNIX. Grep I will find occurrence of both Error, error and ERROR and quite useful to display any sort of Error u from log file.

```
grep –i Error logfile
```

4) Use grep -o command in UNIX if you find whole word instead of just pattern.

```
grep –o ERROR logfile
```

Above grep command in UNIX searches only for instances of 'ERROR' that are entire words; it does not match

5) Another useful grep command line option is "grep -l"
which display 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
```

will list the names of all Java files in the current directory whose contents mention`main'.

6) If you want to see line number of matching lines you can use option "grep -n" below command will show on which lines w Error has appeared.

```
grep –n ERROR logfile
```

7) grep command in UNIX can show matching pattern in color which is quite useful to highlight the matching section , to see matching pattern in color use below command.

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

11 I/O Redirection and Pipes

Abstract

This chapter describes more about the powerful UNIX mechanism of redirecting input, output and errors. Topics 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
- Output filters

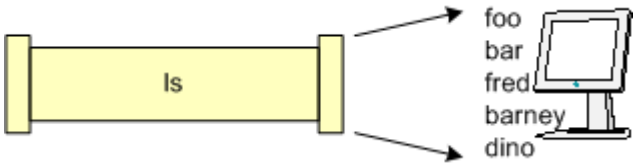
11.1 What are standard input and standard output?

Most Uinux commands read input, such as a file or another attribute for the command, and write output. By default, input is being given with the keyboard, and output is displayed on your screen. Your keyboard is your ***standard input*** (stdin) device, and the screen or a particular terminal window is the ***standard output*** (stdout) device.

Commands typically get their input from a source referred to as **standard input** (stdin) and typically display their output to a destination referred to as **standard output** (stdout) as pictured below:

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 then 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:



11.2 The redirection operators

11.2.1 Output redirection with > and |

Sometimes you will want to put

1) output of a command in a file, or

2) You may want to issue another command on the output of one command.

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

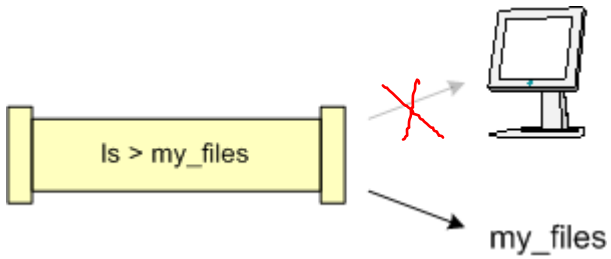
The simplest case to demonstrate this is basic **output redirection .** The general syntax looks as follows:

```
command > output_file_spec
```

Spaces around the redirection operator are not mandatory, but do add readability to the command. Thus in our ls example from above, we can observe the following use of output redirection:

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

Notice there is no output appearing after the command, only the return of the prompt. Why is this, you ask? This is because all output from this command was *redirected* to the file my_files. Observe in the following diagram, no data goes to the terminal screen, but to the file instead.



Examining the file as follows results in the contents of the my_files being displayed:

```
$ 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 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 output to files.

11.2.2 The append operator is the >>.

Thus 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 exist it will append to the existing file
- if the file does not exist, >> will cause its creation and ,
- append the output (to the empty file).

11.3 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 legendary 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
```

If the user *mike* exists on the system, you don't need to type the full address. If you want to reach somebody on the Internet, enter the fully qualified address as an argument to **mail**.

2. Looking in more detail at this, we will use the **wc** (**w**ord **c**ount) command. The wc command counts the number of bytes, word 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, only that it received a stream of bytes to count.

11.4 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 it 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

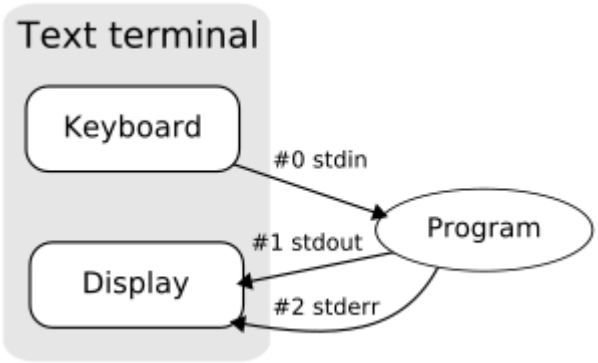
11.5 Advanced redirection features

11.5.1 Use of file descriptors

There are three types of I/O, which each 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 a error you can't omitte the descriptor i.e. you have to write error redirection descriptor as follows.

11.6 Syntax of error redirection:

```
command 2> output_file_spec
```

Thus to show an example, we observe the following:

```
$ 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 once the standard

error stream is redirected into the file named error_file,

and when we display the contents of error_file, it contains what was previously displayed on the termimal. 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 was what was previously displayed on the screen.

Note 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 an empty destination in such cases where there is a large volume of extraneous 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.

11.7 Here Document

<< redirects the standard input of the command to read from what is called a "here document". Here documents are convenient ways 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 is used to indicate the end of file word for the Here Document. Anyword can be used, however there is a common convention of using EOF (unless we need to include that word within your here document).

Example:1 The following example creates a file userlist.txt without waiting for user,after running a file auto_file.sh.

```
$Vim auto_file.sh
$ cat > userlist.txt << EOF
bravo
delta
alpha
chrlie
EOF
```

Example:2 Download file from ftp server automatically using here document

```
$vim 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 crete a report using select query spool that report into a file and mail that file to Manager.

```
$vim 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
```

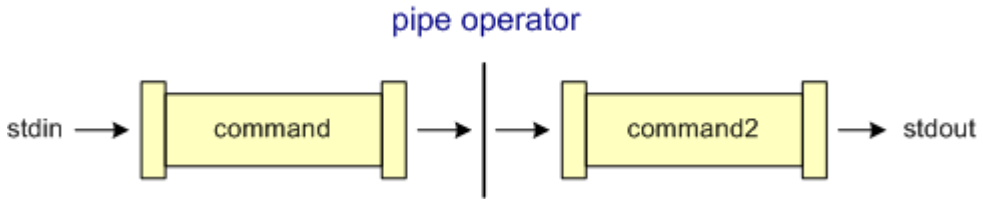
2.1.1 Redirection Summary

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 are considered as input to the command

11.8 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 process to another in the following way.



Unlike other forms of interprocess communication (IPC), a pipe is one-way communication only. Basically, a pipe passes a parameter such as 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.

The UNIX domain sockets (UNIX Pipes) are typically used when communicating between two processes running in the same UNIX machine. UNIX 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 will list 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 send to (redirected) user_list file
$ who | wc –l	Output of who command is given as input to wc command So that it will number of user who logon 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 logon or nothing is printed (To see particular user is logon or not)

12 Control Statements

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

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

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

12.1 Operators

12.1.1 For Mathematics, or numerical comparision use following

Operators in Shell Script

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

12.2 String Operators:

There are following string operators supported by Bourne Shell.

Assume variable a holds "abc" and variable b holds "efg" then:

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

12.3 File test operators

Operator	Discription	Example
-r file	Checks if file is readable if yes then condition becomes true.	[-r $file] is true.
-w file	Check if file is writable if yes then condition becomes true.	[-w $file] is true.
-x file	Check if file is execute 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	[-f $file] is true.

	yes then condition becomes true.	

12.4 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:

12.4.1 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 not executed. Most of the times you will use comparison operators while making decisions.

Give you attention on the spaces between braces 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 return 0 after its execution. If it is a boolean expression then it would be true if it returns true.

2.2 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
```

12.4.2 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 no statement would be not executed.

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
echo “Enter the filename”
 read file1
 if [ ! –s file1 ]
 then
        echo "file1 is empty or does not exist."
        ls –l > file1
 else
         echo "File file1 already exists."
 Fi
```

```
#!/bin/bash
#script to check whether directory exists or not.
#dir=$(pwd)
a=$1
if [ –d $a ]; then
     echo " directory $a exists "
else
     echo " directory $a does not exists "

fi
```

Example:3

```
$ vi positive.sh
```

```
#!/bin/bash
# Script to see whether argument is positive or negative
if [ $# -eq 0 ]
then
     echo "$0 : You must give/supply one integers"
     exit 1
fi
if [ $1 -gt 0 ]
then
     echo "$1 number is positive"
else
     echo "$1 number is 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 I f the same script is run by normal user the error Message should be reported.

```
#!/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 $0"
fi
```

Example:6 Write a shell Script to pass username to shell script and check whether that user exists or not 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 exists"
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

```
$vim db_status.sh
#!/bin/bash
export ORACLE_SID=$1
sqlplus  / as sysdba<<EOF  &>/dev/null
spool status.txt
select sysdate from dual;
spool off
EOF
status=`grep –c 'ORA' status.txt`
if [ $status –gt 0 ]
then
    echo $1 is running…
else
    echo $1 id down…
fi
```

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

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

12.4.3 if...elif...else...fi statement

```
if [ expression 1 ]
then
   Statement(s) to be executed if expression 1 is true
```

```
elif [ expression 2 ]

then

   Statement(s) to be executed if expression 2 is true

elif [ expression 3 ]

then

   Statement(s) to be executed if expression 3 is true

else

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

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 condition is true then 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"
fi
```

This will produce following result:

a is less than b

Example 2:

$ vi elf.sh

```
#!/bin/sh
# Script to test if..elif...else
if [ $1 -gt 0 ]; then
   echo "$1 is positive"
elif [ $1 -lt 0 ]
then
     echo "$1 is negative"
elif [ $1 -eq 0 ]
then
   echo "$1 is zero"
else
   echo "Opps! $1 is not number, give number"
fi
```

12.5 Looping Statements

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

- The while loop
- The for loop
- The until loop

12.5.1 The while loop

You would use different loops based on dfferent situation. For example while loop would execute given commands until given condition remains true where as until loop would execute until a given condition becomes 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.

Similar to the basic if statement, except the block of commands is repeatedly executed as long as the condition is met.

```
while condition-command
  do
      command1
      command2
      ...
  done
```

As with if statements, a semicolon (;) can be used to remove include the do keyword on the same line as the while condition-command statement.

The example below loops over two statements as long as the variable i is less than or equal to ten. Store the following 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 accounte 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 520 ]
          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
```

12.5.2 The for loop

The for loop operate 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 the variable var is set to the next word in the list of words, word1 to wordN.

Example:

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)
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 from my root:

Example:

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

This will produce following result:

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

Example:3

```
#!/bin/bash
echo "You want to ping $1 network"
for i in $(seq 1 1 10)
do
    ping -c1  $1.$i   > /dev/null 2>&1
    if [ $? -eq 0 ]; then
        echo "Node $1.$i is up"
    else
         echo "Node $1.$i is down"
    fi
done
```

Example:4

Example:

```
#!/bin/bash

#for i in /home/ganeshn /home/prashatw /home/abhijit

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"
```

12.5.3 The until loop

he while loop is perfect for a situation where you need to execute a set of commands while some condition is true. Sometimes you need to execute a set of commands until a condition is true.

Syntax:

```
until command
do
   Statement(s) to be executed until command is true
Done
```

Here Shell *command* is evaluated. If the resulting value is *false*, given *statement(s)* are executed. If *command* is *true* then no statement would be not executed and program would jump to the next line after done statement.

Example:

Here is a simple example that uses the until loop to display the numbers zero to nine:

```
#!/bin/sh
a=0
until [ ! $a -lt 10 ]
do
   echo $a
   a=`expr $a + 1`
done
```

This will produce following result:

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

12.6 Break and Continue statements

So far you have looked at creating loops and working with loops to accomplish different tasks.

Sometimes you need to stop a loop or skip iterations of the loop.

In this tutorial you will learn following two statements used to control shell loops:

1. The **break** statement
2. The **continue** statement

The infinite Loop:

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

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

Example:

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

```
#!/bin/sh
a=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. So this true example of infinite loop.

12.6.1 The break statement:

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

Syntax:

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

break

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

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

break n

Example:

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

The break statement will cause the shell to stop executing the current loop and continue on after its end.

```
#!/bin/sh
files=`ls`
count=0
for i in $files
do
   count=`expr $count + 1`
   if [ $count -gt 100 ]
   then
        echo "There are more than 100 files in the current
                                                    directory"
        break
   fi
done
```

```
#!/bin/bash
while [ 1 = 1 ]
do
        if [ -f $1 ]; then
                break
           else
                sleep 1
               echo "file doesnt exists !!"
        fi
done
         echo " file is present in your directory "
```

12.6.2 continue statement:

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

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

Syntax:

```
Continue
```

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

```
continue n
```

Here n specifies the nth enclosing loop to continue from.

Example:

The following loop makes use of continue statement which returns from the continue statement and start processing next statement:

```
#!/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
```

13 Reading from file

you may have written programms for processing file in c language or any other programming language.But for that you may have performed lot of steps only for reading from a file.For example in c language you have to declare a file pointer writing loops,closing pointers etc.
But in shell scripting it is quite easier 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 loop terminator
we feed the while loop from the bottom, which greatly increases the input throughput
to the loop. When we time each technique, this method will stand out at the top of the list.

```
#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
```

14 Wildcards in UNIX

14.1 How to use UNIX Wildcards

Many computer operating systems provide ways to select certain files without typing complete filenames. For example, we may wish to remove all files whose names end with "old". Unix allows us to use wildcards (more formally known as **metacharacters**) to stand for one or more characters in a filename.

The two basic wildcard characters are ? and *. The wildcard ? matches any one character. The wildcard * matches any grouping of zero or more characters. Some examples may help to clarify this. (Remember that Unix is casesensitive).

Assume that your directory contains the following files:

```
Chap bite bin
bit Chap6 it
test.new abc
Lit site test.old
Big snit bin.old
The ? wildcard
```

The command ls will list all the files. The command

```
$ ls ?bit
 bit
```

lists only the files Lit and bit. The file snit was not listed because it has two characters before "it". The file it was not listed because it has no characters before "it".

The ? wildcard may be used more than once in a command. For

```
Example,
$ ls ?i?
Lit big bin bit
```

Finds any files with "i" in the middle, one character before and one character after.

The * wildcard

The * wildcard is more general. It matches zero or any number of characters, except that it will not match a period
that is the first character of a name.

```
$ ls *t
lit bit it snit
```

Using this wildcard finds all the files with "it" as the last two characters of the name (although it would not have found
a file called .bit).
We could use this wildcard to remove all files in the directory whose names begin with "test". The command to do this
is

```
$rm test*
```

Be careful when using the * wildcard, especially with the rm command. If we had mistyped this command by adding a
space between test and *, Unix would look first for a file called test, remove it if found, and then proceed to remove all
the files in the directory!

```
Matching a range of characters with [ ]
```

The ? wildcard matches any one character. To restrict the matching to a particular character or range of characters, use square brackets [ ] to include a list. For example, to list files ending in "ite", and beginning with only "a", "b", "c",
or "d" we would use the command:

```
$ ls [abcd]ite
```

This would list the file bite, but not the file site. Note that the sequence [ ] matches only one character. If we had a file
called delite, the above command would not have matched it.
We can also specify a range of characters using [ ]. For instance, [1-3] will match the digits 1, 2 and 3, while[A-Z]
matches all capital letters.

```
ls [A-Z]it
```

Will find any file ending in "it" and beginning with a capital letter (in this case, the file Lit). Wildcards can also be combined with [ ] sequences. To list any file beginning with a capital letter, we would use:

```
$ ls [A-Z]*
Chap1 Chap6 Lit
```

Matching a string of characters with { }

The method described in the previous section matches a single character or range of characters. It is also possible to match a particular string by enclosing the string in { } (braces). For example, to list only the files ending in the string "old", we would use

```
$ ls *{old}
bin.old test.old
```

To list all files ending in either "old" or "new", use

$ ls *{old,new}

15 Functions

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

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

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

15.1 Creating Functions:

To declare a function, simply use the following syntax:

```
function_name () {
   list of commands
}
```

The name of your function is function_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 braces.

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:

```
amrood]$./test.sh
Hello World
[amrood]$
```

15.2 Pass Parameters to a Function:

You can define a function which would accept parameters while calling those function. These parameters would be represented by $1, $2 and so on.

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:

```
[amrood]$./test.sh
Hello World Zara Ali
[amrood]$
```

15.3 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 way to come out of a defined function.

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 here, but obviously you should choose something that is meaningful or useful in the context of your script as a whole.

Example:

Following function returns a value 1:

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

This would produce following result:

[amrood]$./test.sh
Hello World Zara Ali
Return value is 10
[amrood]$

15.4 Function Call from Prompt:

You can put definitions for commonly used functions inside your *.profile* 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 execute the file in the current shell by typing:

```
[amrood]$. test.sh
```

This has the effect of causing any functions defined inside test.sh to be read in and defined to the current shell as follows:

```
[amrood]$ number_one
This is the first function speaking...
This is now the second function speaking...
[amrood]$
```

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.

```
[amrood]$unset .f function_name
```

16 Arrays

A shell variable is capable enough to hold a single value. This type of variables are called scalar variables.

Shell supports a different type of variable called an array variable that can hold multiple values at the same time. Arrays provide a method of grouping a set of variables. Instead of creating a new name for each variable that is required, you can use a single array variable that stores all the other variables.

All the naming rules discussed for Shell Variables would be applicable while naming arrays.

16.1 Defining Array Values:

The difference between an array variable and a scalar variable can be explained as follows.

Say that you are trying to represent the names of various students as a set of variables. Each of the individual variables is a scalar variable as follows:

```
NAME01="Zara"
NAME02="Qadir"
NAME03="Mahnaz"
NAME04="Ayan"
NAME05="Daisy"
```

We can use a single array to store all the above mentioned names. Following is the simplest method of creating an array variable is to assign a value to one of its indices. This is expressed as follows:

```
array_name[index]=value
```

Here *array_name* is the name of the array, *index* is the index of the item in the array that you want to set, and value is the value you want to set for that item.

As an example, the following commands:

```
NAME[0]="Zara"
NAME[1]="Qadir"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
```

If you are using **ksh** shell the here is the syntax of array initialization:

```
set -A array_name value1 value2 ... valuen
```

If you are using **bash** shell the here is the syntax of array initialization:

```
array_name=(value1 ... valuen)
```

16.2 Accessing Array Values:

After you have set any array variable, you access it as follows:

```
${array_name[index]}
```

Here *array_name* is the name of the array, and *index* is the index of the value to be accessed. Following is the simplest example:

```
#!/bin/sh
NAME[0]="Zara"
NAME[1]="Qadir"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
echo "First Index: ${NAME[0]}"
echo "Second Index: ${NAME[1]}"
```

This would produce following result:

```
[amrood]$./test.sh
First Index: Zara
Second Index: Qadir
```

You can access all the items in an array in one of the following ways:

```
${array_name[*]}
${array_name[@]}
```

ere array_name is the name of the array you are interested in. Following is the simplest example:

```
#!/bin/sh
NAME[0]="Zara"
NAME[1]="Qadir"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
echo "First Method: ${NAME[*]}"
echo "Second Method: ${NAME[@]}"
```

This would produce following result:

```
[amrood]$./test.sh
First Method: Zara Qadir Mahnaz Ayan Daisy
Second Method: Zara Qadir Mahnaz Ayan Daisy
```

17 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 + D).
SIGFPE	8	Issued if an illegal mathematical operation is attempted
SIGKILL	9	If a process gets this signal it must quit immediately and will not perform any clean-up operations
SIGALRM	14	Alarm Clock signal (used for timers)
SIGTERM	15	Software termination signal (sent by kill by default).

17.1 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:

```
[amrood]$ 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.

17.2 Default Actions:

Every signal has a default action associated with it. The default action for a signal is the action sithat 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.

17.3 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 CONTROL-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:

```
[amrood]$ 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:

[amrood]$ 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 folloing command:

[amrood]$ kill -9 1001

This would kill the process running with process ID 1001.

17.4 Trapping Signals:

When you press the *Ctrl+C* or Break key 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 *command* can be any valid Unix command, or even a user-defined function, and signal 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

Example

Following script dehmonstrate the tarping the TERM signal when it is send to shell script

```
vi trap.sh
#!/bin/bash
trap "echo 'trapped the signal TERM'" TERM
echo "starting infinite looop"
i=1
while [ 1 -eq 1 ]
do
    echo "$i. sleeping for 1 sec..."
    sleep 1
    i=$(( $i + 1 ))
done
```

18 awk

```
awk options 'selection criteria { action }' file(s)
```

AWK is a parser and a powerful report building tool.

Let's go ahead and start playing around with awk to see how it works. At the command line, enter the following command:

```
$ awk '{ print }' /etc/passwd
```

You should see the contents of your /etc/passwd file appear before your eyes. Now, for an explanation of what awk did. When we called awk, we specified /etc/passwd as our input file. When we executed awk,

- it evaluated the print command for each line in /etc/passwd, in order.
- All output is sent to stdout, and we get a result identical to catting /etc/passwd.

Now, for an explanation of the { print } code block. In awk, curly braces are used to group blocks of code together, similar to C.

Inside our block of code, we have a single print command. In awk, when a print command appears by itself, the full contents of the current line are printed.

Here is another awk example that does exactly the same thing:

```
$ awk '/director/ {print}' emp.lst
```

This will find out all lines in emp.lst file that contain
the word “director” and print them to the stdout. Same as
grep director emp.lst

$ awk ‘/director/ ’ emp.lst Print is the default action

$ awk '{ print $0 }' /etc/passwd

In awk, the $0 variable represents the entire current line, so print and print $0 do exactly the same thing. If you'd like, you can create an awk program that will output data totally unrelated to the input data. Here's an example:

$ awk '{ print "" }' /etc/passwd

Whenever you pass the "" string to the print command, it prints a blank line. If you test this script, you'll find that awk outputs one blank line for every line in your /etc/passwd file. Again, this is because awk executes your script for every line in the input file. Here's another example:

$ awk '{ print "hiya" }' /etc/passwd

Running this script will fill your screen with hiya's. :)

Multiple fields

Awk is really good at handling text that has been broken into multiple logical fields, and allows you to effortlessly reference each individual field from inside your awk script. The following script will print out a list of all user accounts on your system:

```
$ awk -F":" '{ print $1 }' /etc/passwd
```

Above, when we called awk, we use the -F option to specify ":" as the field separator. When awk processes the print $1 command, it will print out the first field that appears on each line in the input file. Here's another example:

```
$ awk -F":" '{ print $1 $3 }' /etc/passwd
```

Here's an excerpt of the output from this script:

```
halt7
operator11
root0
shutdown6
sync5
bin1
....etc.
```

As you can see, awk prints out the first and third fields of the /etc/passwd file, which happen to be the username and uid fields respectively. Now, while the script did work, it's not perfect -- there aren't any spaces between the two output fields! If you're used to programming in bash or python, you may have expected the print $1 $3 command to insert a space between the two fields. However, when two strings appear next to each other in an awk program, awk concatenates them without adding an intermediate space. The following command will insert a space between both fields:

```
$ awk -F":" '{ print $1 " " $3 }' /etc/passwd
```

When you call print this way, it'll concatenate $1, " ", and $3, creating readable output. Of course, we can also insert some text labels if needed:

```
$ awk -F":" '{ print "username: " $1 "\t\tuid:" $3" }' /etc/passwd
```

This will cause the output to be:

username: halt uid:7

username: operator uid:11

username: root uid:0

username: shutdown uid:6

username: sync uid:5

username: bin uid:1

....etc.

AWK Variables

You can use variables in awk and assign values to them. They do not need a $ sign in front of them like shell variables. Variables do not have data type. Variables are not declared. String variables are always double quoted. Numbera are initialized to 0 and strings are initialized to null (empty string)

X = “5”

Y =10

Z=”A”

Print X + Y Prints 15

Print XY Prints 510

Print Y + Z Prints 10. Z is converted to 0 since it does not have numerals

```
$ awk –F"|" '/director/ { kount = kount + 1
> printf "%3d %-20s", kount, $2}
```

Prints formatted output with line numbers

AWK Expressions

AWK expressions either have a value of true or false. A positive number or a not empty string is true.

X="AAAA"

Y=-1

If (X) will return true

If (Y) will return false

We will learn more about if statement later

How to check the value of an expression in awk This is only way. Another way is to use if statement

```
$ awk –F"|" ' $3 == "director" { print $0}' emp.lst
```

Print entire line from the emp.lst only if the third field is "director"

```
$ awk –F"|" ' $3 == "director" || $3 == "chairman" { print $0}' emp.lst
```

Print entire line from the emp.lst only if the third field is "director" or "chairman"

```
$ awk –F"|" ' $6 > 8000 && $3 == "supervisor" { print $0}' emp.lst
```

Print entire line from the emp.lst only if the employee is a supervisor and has a salary more than 8000"

|| OR operator

&& AND operator

== is equql to

!= is not equal to

> is less than

< is greater than

>= is greater than or equal to

<= is less than or equal to

External scripts

Passing your scripts to awk as a command line argument can be very handy for small one-liners, but when it comes to complex, multi-line programs, you'll definitely want to compose your script in an external file. Awk can then be told to source this script file by passing it the -f option:

```
$ awk -f myscript.awk myfile.in
```

Putting your scripts in their own text files also allows you to take advantage of additional awk features. For example, this multi-line script does the same thing as one of our earlier one-liners, printing out the first field of each line in /etc/passwd:

```
BEGIN {
    FS=":"
}
{ print $1 }
```

The difference between these two methods has to do with how we set the field separator. In this script, the field separator is specified within the code itself (by setting the FS variable), while our previous example set FS by passing the -F":" option to awk on the command line. It's generally best to set the field separator inside the script itself, simply because it means you have one less command line argument to remember to type. We'll cover the FS variable in more detail later in this article.

The BEGIN and END blocks

Normally, awk executes each block of your script's code once for each input line. However, there are many programming situations where you may need to execute initialization code before awk begins processing the text from the input file. For such situations, awk allows you to define a BEGIN block. We used a BEGIN block in the previous example. Because the BEGIN block is evaluated before awk starts processing the input file, it's an excellent place to initialize the FS (field separator) variable, print a heading, or initialize other global variables that you'll reference later in the program.

Awk also provides another special block, called the END block. Awk executes this block after all lines in the input file have been processed. Typically, the END block is used to perform final calculations or print summaries that should appear at the end of the output stream.

Regular expressions and blocks

Awk allows the use of regular expressions to selectively execute an individual block of code, depending on whether or not the regular expression matches the current line. Here's an example script that outputs only those lines that contain the character sequence foo:

```
/foo/ { print }
```

Of course, you can use more complicated regular expressions. Here's a script that will print only lines that contain a floating point number:

```
/[0-9]+\.[0-9]*/ { print }
```

Expressions and blocks

There are many other ways to selectively execute a block of code. We can place any kind of boolean expression before a code block to control when a particular block is executed. Awk will execute a code block only if the preceding boolean expression evaluates to true. The following

example script will output the third field of all lines that have a first field equal to fred. If the first field of the current line is not equal to fred, awk will continue processing the file and will not execute the print statement for the current line:

```
$1 == "fred" { print $3 }
```

Awk offers a full selection of comparison operators, including the usual "==", "<", ">", "<=", ">=", and "!=". In addition, awk provides the "~" and "!~" operators, which mean "matches" and "does not match". They're used by specifying a variable on the left side of the operator, and a regular expression on the right side. Here's an example that will print only the third field on the line if the fifth field on the same line contains the character sequence root:

```
$1 ~ /root/ { print $3 }
```

Conditional statements

Awk also offers very nice C-like if statements. If you'd like, you could rewrite the previous script using an if statement:

```
{
 if ( $5 ~ /root/ ) {
       print $3
 }
}
```

19 UNIX Interview Questions

1.What OS are you running

uname

2.What is your kernel version

uname -r

3.For how long your machine is running

uptime

4.How many users are logged on to your system

w

5.What is user "shekhar" doing now

w|grep shekhar

6.When did user rohit logged in

w|grep rohit

7.From which machine user prakash has logged in

who|grep prakash

8.what is the login shell of user roshan

cat /etc/passwd|grep roshan|cut -d":" -f7

9.What is the home directory of user prakash

cat /etc/passwd|grep roshan|cut -d":" -f6

10.How many users do not have bash shell assigned to them

cat /etc/passwd|grep -v /bin/bash|wc -l

11.Create an alias named i to display ip address of your machine. Make is parmanent.

vim .bash_profile

alias i='ifconfig'

:wq

12.What group/s do you belong to

groups

13.What group/s are assigned to shekhar

groups shekhar

14.What are the last 10 commands that you have executed

tail .bash_history

15.How many times you have executed cd command receltly

cat .bash_history|grep cd

16.Can you read a file named /etc/fstab.

yes (but can't write to this file)

17.Can user shekhar read the same file

yes

18.Do you have the permission to modify /etc/sysctl.conf file. Who can modify it.

no.only root user.

19.How much of RAM do you have on your machine

free -m

20.What is the size of your swap

free -m|grep -i swap

21.How many processes are currently running (not sleeping)

top

22.Which process is taking the maximum CPU

top

23.Which process is taking the maximum memory

top

24.When I type ifconfig command I get the error message "command not found". What is the problem and how can I solve it permaneltly.

--if normal user is not having permission of running ifconfig command.can solved by making entry

in /etc/sudoers file by root user

--or if permission is there but path is not fount then set the path for ifonfig command

ex-- PATH=$PATH:/sbin/ifconfig

25.Is oracle running on your server

ps -ef|grep oracle

26.How many oracle databases are running on the server

27.How can you change the default permissions of a file to r--r--r when the file gets created

by using umask command

ex:umask 0222

28.What is the current date on your system

date +%D

29.What is the current time on your system

date +%T

30.How to find out the time required to run a specific command

time command

30. Find out all files in /tmp directory whose owner is shekhar

find /tmp -user shekhar

31.Find out all files older than 30 days whose extension is log and delete them - in one command

Vi Questions

a. Go to the end of the file

G

b. GO to the begining of the file

gg or :1 enter

c. Find a word "tom" and replace it by "dick"

:1 $s/tom/dick/g

d. How to undo any mistake

ESC u

e. How to quit the file without saving and discard changes that you have made

q!

f. How to copy 30 lines and paste at the very begining

30 yy

ESC gg

p

20 Useful Assignments

20.1 Shell Scripting Assignments for Linux Admins

1) Write a shell script for download a file from ftp server.

- schedule it to run at a specific time.
- send success or failure email.
- Use command line arguments for sending ip of the ftp server and loginID & password

2) Check the status (ping) of the server by shell script.

- Server ip should be sent as a command line arg.
- If not sent give an error message.
- Send email of failure.

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 server is kept in a file named /opt/server_list.txt

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.

5) Write a shell script that will Clone a VirtualMachine

- Make it an interactive script i.e the location to store the clone
- size reqired to VM etc should be provided by user.
- Note:Disk space shpould be checked before cloning

6) Write a shell script Lock all users between UID 500 and 530

7) Write a shell script Lock all users whose names are in a file called users.txt

8) 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.

9) Write a shell script to Identify list of user who have executed more than 10 jobs yesterday.
send this report in an email.

10) Write a shell script to find files larger that a specific size.

- report it to a concerned person through email.

11) Generate and email Security check report daily

- List uf 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

12) Write a shell script to Reset passwords of all users listed in a file user.list

20.2 Shell Scripting Assignment for Oracle

---

1. Load data from a file

2. Extract data from a file and then ftp that file to another server.
Schedule it daily.
Send email on success or failure

3. Generate a report of Top 5 customers and their revenue for the last year

and email this report to all participants whose email addresses are present in a file named email_list.txt

4. Backup of oracle database (RMAN)

5. Export of oracle database. Name of the DB should be sent as a command line argument. send email.

6.Check instance availability.

7.Check the listener avalability.