BIRLA INSTITUTE OF TECHNOLOGY, MESRA

"Shell and Kernel Programming Lab"

LAB MANUAL

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

SYLLABUS

CS310 Shell and Kernel Programming Lab

1. Use of Basic UNIX Shell Commands: ls, mkdir, rmdir, cd, cat, touch, file, wc, sort, cut, grep, dd, dfspace, du, ulimit
2. Commands related to inode, I/O redirection and piping, process control commands, mails.
3. Shell Programming: Shell script exercises based on following:
   (i) Interactive shell scripts (ii) Positional parameters (iii) Arithmetic (iv) if-then-fi, if-then- else-fi, nested if-else (v) Logical operators (vi) else + if equals elif, case structure (vii) while, until, for loops, use of break
4. Write a shell script to create a file. Follow the instructions
   (i) Input a page profile to yourself, copy it into other existing file;
   (ii) Start printing file at certain line
   (iii) Print all the difference between two file, copy the two files.
   (iv) Print lines matching certain word pattern.
5. Write shell script for-
   (i) Showing the count of users logged in,
   (ii) Printing Column list of files in your home directory
   (iii) Listing your job with below normal priority
   (IV) Continue running your job after logging out.
6. Write a shell script to change data format. Show the time taken in execution of this script.
7. Write a shell script to print files names in a directory showing date of creation & serial number of the file.
8. Write a shell script to count lines, words and characters in its input (do not use wc).
9. Write a shell script to print end of a Glossary file in reverse order using Array. (Use awk tail)
10. Write a shell script to compute gcd lcm & of two numbers. Use the basic function to find gcd & LCM of N numbers.
11. Write a shell script to find whether a given number is prime. Take a large number such as 15 digits or higher and use a proper algorithm.

Chapter 1
Introduction & Overview

1.1 Operating System

An **operating system** (**OS**) is a collection of software that manages computer hardware resources and provides common services for computer programs. The operating system is a vital component of the system software in a computer system. Application programs usually require an operating system to function.

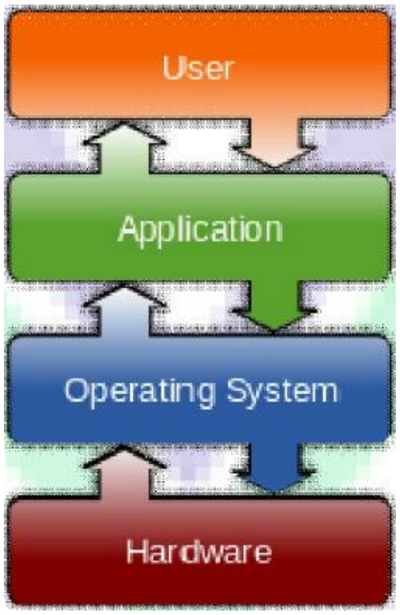


Fig 1.1 Operating System

1.2 Types Of Operating System

a. Real-time

A real-time operating system is a multitasking operating system that aims at executing real-time applications. Real-time operating systems often use specialized scheduling algorithms so that they can achieve a deterministic nature of behavior. The main objective of real-time operating systems is their quick and predictable response to events. They have an event-driven or timesharing design and often aspects of both. An event-driven system switches between tasks based on their priorities or external events while time-sharing operating systems switch tasks based on clock interrupts.

b. Multi-user

A multi-user operating system allows multiple users to access a computer system at the same time. Time-sharing systems and Internet servers can be Shell Programming Lab Manual classified as multi-user systems as they enable multiple-user access to a computer through the sharing of time. Single-user operating systems have only one user but may allow multiple programs to run at the same time.

c. **Multi-tasking vs. single-tasking**

A multi-tasking operating system allows more than one program to be running at a time, from the point of view of human time scales. A single tasking system has only one running program. Multi-tasking can be of two types: pre-emptive and co-operative. In pre-emptive multitasking, the operating system slices the CPU time and dedicates one slot to each of the programs. Unix-like operating systems such as Solaris and Linux support preemptive multitasking, as does AmigaOS. Cooperative multitasking is achieved by relying on each process to give time to the other processes in a defined manner. 16-bit versions of Microsoft Windows used cooperative multi-tasking. 32-bit versions of both Windows NT and Win9x, used preemptive multi-tasking. Mac OS prior to OS X used to support cooperative multitasking.

d. Distributed

distributed operating system manages a group of independent computers and makes them appear to be a single computer. The development of networked computers that could be linked and communicate with each other gave rise to distributed computing. Distributed computations are carried out on more than one machine. When computers in a group work in cooperation, they make a distributed system.

e. Embedded

Embedded operating systems are designed to be used in embedded computer systems. They are designed to operate on small machines like PDAs with less autonomy. They are able to operate with a limited number of resources. They are very compact and extremely efficient by design. Windows CE and Minix 3 are some examples of embedded operating systems.

f. Time-Sharing

Time-sharing operating systems schedule tasks for efficient use of the system and may also include accounting for cost allocation of processor time, mass storage, printing, and other resources.

1.3 Examples of Operating System

a. UNIX and UNIX-like operating systems

Unix was originally written in assembly language. Ken Thompson wrote B, mainly based on BCPL, based on his experience in the MULTICS project. B was replaced by C, and Unix, rewritten in C, developed into a large, complex family of inter-related operating systems which have been influential in every modern operating system.

The *UNIX-like* family is a diverse group of operating systems, with several major subcategories including System V, BSD, and Linux. The name "UNIX" is a trademark of The Open Group which licenses it for use with any operating system that has been shown to conform to their definitions. "UNIX-like" is commonly used to refer to the large set of operating systems which resemble the original UNIX.

Unix-like systems run on a wide variety of computer architectures. They are used heavily for servers in business, as well as workstations in academic and engineering environments. Free UNIX variants, such as Linux and BSD, are popular in these areas. Four operating systems are certified by the The Open Group (holder of the Unix trademark) as Unix. HP's HP-UX and IBM's AIX are both descendants of the original System V Unix and are designed to run only on their respective vendor's hardware. In contrast, Sun Microsystems's Solaris Operating System can run on multiple types of hardware, including x86 and Sparc servers, and PCs. Apple's OS X, a replacement for Apple's earlier (non-Unix) Mac OS, is a hybrid kernel-based BSD variant derived from NeXTSTEP, Mach, and FreeBSD.

Unix interoperability was sought by establishing the POSIX standard. The POSIX standard can be applied to any operating system, although it was originally created for various Unix variants.

b. Linux and GNU

Linux (or GNU/Linux) is a Unix-like operating system that was developed without any actual Unix code, unlike BSD and its variants. Linux can be used on a wide range of devices from super computers to wristwatches. The Linux kernel is released under an open source license, so anyone can read and modify its code.

It has been modified to run on a large variety of electronics. Although estimates suggest that Linux is used on 1.82% of all personal computers, it has been widely adopted for use in servers and embedded systems (such as cell phones). Linux has superseded Unix in most places, and is used on the 10 most powerful supercomputers in the world. The Linux kernel is used in some popular distributions, such as Red Hat, Debian, Ubuntu, Linux Mint and Google's Android.

The GNU project is a mass collaboration of programmers who seek to create a completely free and open operating system that was similar to Unix but with completely original code. It was started in 1983 by Richard Stallman, and is responsible for many of the parts of most Linux variants. Thousands of pieces of software for virtually every operating system are licensed under the GNU General Public License. Meanwhile, the Linux kernel began as a side project of Linus Torvalds, a university student from Finland. In 1991, Torvalds began work on it, and posted information about his project on a newsgroup for computer students and programmers. He received a wave of support and volunteers who ended up creating a full-fledged kernel. Programmers from GNU took notice, and members of both projects worked to integrate the finished GNU parts with the Linux kernel in order to create a full-fledged operating system.

c. Microsoft Windows

Microsoft Windows is a family of proprietary operating systems designed by Microsoft Corporation and primarily targeted to Intel architecture based computers, with an estimated 88.9 percent total usage share on Web connected computers. The newest version is Windows 8 for workstations and Windows Server 2012 for servers.

Windows 7 recently overtook Windows XP as most used OS. Microsoft Windows originated in 1985 as an operating environment running on top of MS-DOS, which was the standard operating system shipped on most Intel architecture personal computers at the time. In 1995, Windows 95 was released which only used MS-DOS as a bootstrap. For backwards compatibility, Win9x could run real-mode MS-DOS and 16 bit Windows 3.x drivers. Windows ME, released in 2000, was the last version in the Win9x family. Later versions have all been based on the Windows NT kernel. Current versions of Windows run on IA-32 and x86-64 microprocessors, although Windows 8 will support ARM architecture. In the past, Windows NT supported non-Intel architectures. Server editions of Windows are widely used. In recent years, Microsoft has expended significant capital in an effort to promote the use of Windows as a server operating system. However, Windows' usage on servers is not as widespread as on personal computers, as Windows competes against Linux and BSD for server market share

1.4 Filesystem architecture of Linux

Directory Content

/bin Common programs, shared by the system, the system administrator and the users.

/boot

The startup files and the kernel, vmlinuz. In some recent distributions also grub data. Grub is the GRand Unified Boot loader and is an attempt to get rid of the many different boot-loaders we know today.

/dev Contains references to all the CPU peripheral hardware, which are represented as files with special properties.

/etc Most important system configuration files are in /etc, this directory contains data similar to those in the Control Panel in Windows

/home Home directories of the common users.

/initrd (on some distributions) Information for booting. Do not remove!

/lib Library files, includes files for all kinds of programs needed by the system and the users.

/lost+found Every partition has a lost+found in its upper directory. Files that were saved

Directory Structure

Unix uses a hierarchical file system structure, much like an upside-down tree, with root (/) at the base of the file system and all other directories spreading from there.

A Unix filesystem is a collection of files and directories that has the following properties −

- It has a root directory (**/**) that contains other files and directories.
- Each file or directory is uniquely identified by its name, the directory in which it resides, and a unique identifier, typically called an **inode**.

- By convention, the root directory has an **inode** number of **2** and the **lost+found** directory has an **inode** number of **3**. Inode numbers **0** and **1** are not used. File inode numbers can be seen by specifying the **-i option** to **ls command**.
- It is self-contained. There are no dependencies between one filesystem and another.

The directories have specific purposes and generally hold the same types of information for easily locating files. Following are the directories that exist on the major versions of Unix −

Sr.No.	Directory & Description
1	**/** This is the root directory which should contain only the directories needed at the top level of the file structure
2	**/bin** This is where the executable files are located. These files are available to all users
3	**/dev** These are device drivers
4	**/etc** Supervisor directory commands, configuration files, disk configuration files, valid user lists, groups, ethernet, hosts, where to send critical messages
5	**/lib** Contains shared library files and sometimes other kernel-related files
6	**/boot** Contains files for booting the system
7	**/home** Contains the home directory for users and other accounts

8	**/mnt** Used to mount other temporary file systems, such as **cdrom** and **floppy** for the **CD-ROM** drive and **floppy diskette drive**, respectively
9	**/proc** Contains all processes marked as a file by **process number** or other information that is dynamic to the system
10	**/tmp** Holds temporary files used between system boots
11	**/usr** Used for miscellaneous purposes, and can be used by many users. Includes administrative commands, shared files, library files, and others
12	**/var** Typically contains variable-length files such as log and print files and any other type of file that may contain a variable amount of data
13	**/sbin** Contains binary (executable) files, usually for system administration. For example, ***fdisk*** and ***ifconfig*** utlities
14	**/kernel** Contains kernel files

Navigating the File System

Now that you understand the basics of the file system, you can begin navigating to the files you need. The following commands are used to navigate the system −

Sr.No.	Command & Description
1	**cat filename** Displays a filename

2	**cd dirname** Moves you to the identified directory
3	**cp file1 file2** Copies one file/directory to the specified location
4	**file filename** Identifies the file type (binary, text, etc)
5	**find filename dir** Finds a file/directory
6	**head filename** Shows the beginning of a file
7	**less filename** Browses through a file from the end or the beginning
8	**ls dirname** Shows the contents of the directory specified
9	**mkdir dirname** Creates the specified directory
10	**more filename** Browses through a file from the beginning to the end
11	**mv file1 file2** Moves the location of, or renames a file/directory
12	**pwd** Shows the current directory the user is in

13	**rm filename** Removes a file
14	**rmdir dirname** Removes a directory
15	**tail filename** Shows the end of a file
16	**touch filename** Creates a blank file or modifies an existing file or its attributes
17	**whereis filename** Shows the location of a file
18	**which filename** Shows the location of a file if it is in your PATH

You can use Manpage Help to check complete syntax for each command mentioned here.

The df Command

The first way to manage your partition space is with the **df (disk free)** command. The command **df -k (disk free)** displays the **disk space usage in kilobytes**, as shown below −

```
$df -k
Filesystem      1K-blocks      Used   Available Use% Mounted on
/dev/vzfs        10485760   7836644     2649116  75% /
/devices                0         0           0   0% /devices
$
```

Some of the directories, such as **/devices**, shows 0 in the kbytes, used, and avail columns as well as 0% for capacity. These are special (or virtual) file systems, and although they reside on the disk under /, by themselves they do not consume disk space.

The **df -k** output is generally the same on all Unix systems. Here's what it usually includes −

Sr.No.	Column & Description
1	**Filesystem** The physical file system name
2	**kbytes** Total kilobytes of space available on the storage medium
3	**used** Total kilobytes of space used (by files)
4	**avail** Total kilobytes available for use
5	**capacity** Percentage of total space used by files
6	**Mounted on** What the file system is mounted on

You can use the **-h (human readable) option** to display the output in a format that shows the size in easier-to-understand notation.

The du Command

The **du (disk usage) command** enables you to specify directories to show disk space usage on a particular directory.

This command is helpful if you want to determine how much space a particular directory is taking. The following command displays number of blocks consumed by each directory. A single block may take either 512 Bytes or 1 Kilo Byte depending on your system.

```
$du /etc
10     /etc/cron.d
126    /etc/default
6      /etc/dfs
...
$
```

The **-h** option makes the output easier to comprehend −

```
$du -h /etc
5k    /etc/cron.d
63k   /etc/default
3k    /etc/dfs
...
$
```

Mounting the File System

A file system must be mounted in order to be usable by the system. To see what is currently mounted (available for use) on your system, use the following command −

```
$ mount
/dev/vzfs on / type reiserfs (rw,usrquota,grpquota)
proc on /proc type proc (rw,nodiratime)
devpts on /dev/pts type devpts (rw)
$
```

The **/mnt** directory, by the Unix convention, is where temporary mounts (such as CDROM drives, remote network drives, and floppy drives) are located. If you need to mount a file system, you can use the mount command with the following syntax −

```
mount -t file_system_type device_to_mount directory_to_mount_to
```

For example, if you want to mount a **CD-ROM** to the directory **/mnt/cdrom**, you can type −

```
$ mount -t iso9660 /dev/cdrom /mnt/cdrom
```

This assumes that your CD-ROM device is called **/dev/cdrom** and that you want to mount it to **/mnt/cdrom**. Refer to the mount man page for more specific information or type mount **-h** at the command line for help information.

After mounting, you can use the cd command to navigate the newly available file system through the mount point you just made.

Unmounting the File System

To unmount (remove) the file system from your system, use the **umount** command by identifying the mount point or device.

For example, **to unmount cdrom**, use the following command −

```
$ umount /dev/cdrom
```

The **mount command** enables you to access your file systems, but on most modern Unix systems, the **automount function** makes this process invisible to the user and requires no intervention.

User and Group Quotas

The user and group quotas provide the mechanisms by which the amount of space used by a single user or all users within a specific group can be limited to a value defined by the administrator.

Quotas operate around two limits that allow the user to take some action if the amount of space or number of disk blocks start to exceed the administrator defined limits −

- **Soft Limit** − If the user exceeds the limit defined, there is a grace period that allows the user to free up some space.
- **Hard Limit** − When the hard limit is reached, regardless of the grace period, no further files or blocks can be allocated.

There are a number of commands to administer quotas −

Sr.No.	Command & Description
1	**quota** Displays disk usage and limits for a user of group
2	**edquota** This is a quota editor. Users or Groups quota can be edited using this command
3	**quotacheck** Scans a filesystem for disk usage, creates, checks and repairs quota files
4	**setquota** This is a command line quota editor
5	**quotaon** This announces to the system that disk quotas should be enabled on one or more filesystems
6	**quotaoff** This announces to the system that disk quotas should be disabled for one or more filesystems
7	**repquota** This prints a summary of the disc usage and quotas for the specified file systems

File Management:

In this chapter, we will discuss in detail about file management in Unix. All data in Unix is organized into files. All files are organized into directories. These directories are organized into a tree-like structure called the filesystem.

When you work with Unix, one way or another, you spend most of your time working with files. This tutorial will help you understand how to create and remove files, copy and rename them, create links to them, etc.

In Unix, there are three basic types of files −

- **Ordinary Files** − An ordinary file is a file on the system that contains data, text, or program instructions. In this tutorial, you look at working with ordinary files.
- **Directories** − Directories store both special and ordinary files. For users familiar with Windows or Mac OS, Unix directories are equivalent to folders.
- **Special Files** − Some special files provide access to hardware such as hard drives, CD-ROM drives, modems, and Ethernet adapters. Other special files are similar to aliases or shortcuts and enable you to access a single file using different names.

Listing Files

To list the files and directories stored in the current directory, use the following command −

```
$ls
```

Here is the sample output of the above command −

```
$ls

bin          hosts   lib     res.03
ch07         hw1     pub     test_results
ch07.bak     hw2     res.01  users
docs         hw3     res.02  work
```

The command **ls** supports the **-l** option which would help you to get more information about the listed files −

```
$ls -l
total 1962188

drwxrwxr-x  2 amrood amrood      4096 Dec 25 09:59 uml
-rw-rw-r--  1 amrood amrood      5341 Dec 25 08:38 uml.jpg
drwxr-xr-x  2 amrood amrood      4096 Feb 15  2006 univ
drwxr-xr-x  2 root   root        4096 Dec  9  2007 urlspedia
-rw-r--r--  1 root   root      276480 Dec  9  2007 urlspedia.tar
drwxr-xr-x  8 root   root        4096 Nov 25  2007 usr
drwxr-xr-x  2    200    300      4096 Nov 25  2007 webthumb-1.01
-rwxr-xr-x  1 root   root        3192 Nov 25  2007 webthumb.php
-rw-rw-r--  1 amrood amrood     20480 Nov 25  2007 webthumb.tar
-rw-rw-r--  1 amrood amrood      5654 Aug  9  2007 yourfile.mid
```

```
-rw-rw-r--  1 amrood amrood    166255 Aug  9  2007 yourfile.swf
drwxr-xr-x 11 amrood amrood      4096 May 29  2007 zlib-1.2.3
$
```

Here is the information about all the listed columns −

- **First Column** − Represents the file type and the permission given on the file. Below is the description of all type of files.
- **Second Column** − Represents the number of memory blocks taken by the file or directory.
- **Third Column** − Represents the owner of the file. This is the Unix user who created this file.
- **Fourth Column** − Represents the group of the owner. Every Unix user will have an associated group.
- **Fifth Column** − Represents the file size in bytes.
- **Sixth Column** − Represents the date and the time when this file was created or modified for the last time.
- **Seventh Column** − Represents the file or the directory name.

In the **ls -l** listing example, every file line begins with a **d**, **-**, or **l**. These characters indicate the type of the file that's listed.

Sr.No.	Prefix & Description
1	**-** Regular file, such as an ASCII text file, binary executable, or hard link.
2	**b** Block special file. Block input/output device file such as a physical hard drive.
3	**c** Character special file. Raw input/output device file such as a physical hard drive.
4	**d** Directory file that contains a listing of other files and directories.
5	**l** Symbolic link file. Links on any regular file.

6	**p** Named pipe. A mechanism for interprocess communications.
7	**s** Socket used for interprocess communication.

Metacharacters

Metacharacters have a special meaning in Unix. For example, ***** and **?** are metacharacters. We use ***** to match 0 or more characters, a question mark (**?**) matches with a single character.

For Example −

```
$ls ch*.doc
```

Displays all the files, the names of which start with **ch** and end with **.doc** −

```
ch01-1.doc   ch010.doc  ch02.doc    ch03-2.doc
ch04-1.doc   ch040.doc  ch05.doc    ch06-2.doc
ch01-2.doc ch02-1.doc c
```

Here, ***** works as meta character which matches with any character. If you want to display all the files ending with just **.doc**, then you can use the following command −

```
$ls *.doc
```

Hidden Files

An invisible file is one, the first character of which is the dot or the period character (.). Unix programs (including the shell) use most of these files to store configuration information.

Some common examples of the hidden files include the files −

- **.profile** − The Bourne shell (sh) initialization script
- **.kshrc** − The Korn shell (ksh) initialization script
- **.cshrc** − The C shell (csh) initialization script
- **.rhosts** − The remote shell configuration file

To list the invisible files, specify the **-a** option to **ls** −

```
$ ls -a

.         .profile       docs     lib     test_results
..        .rhosts        hosts    pub     users
.emacs    bin            hw1      res.01  work
.exrc     ch07           hw2      res.02
.kshrc    ch07.bak       hw3      res.03
```

```
$
```

- **Single dot (.)** − This represents the current directory.
- **Double dot (..)** − This represents the parent directory.

Creating Files

You can use the **vi** editor to create ordinary files on any Unix system. You simply need to give the following command −

```
$ vi filename
```

The above command will open a file with the given filename. Now, press the key **i** to come into the edit mode. Once you are in the edit mode, you can start writing your content in the file as in the following program −

```
This is unix file....I created it for the first time.....
I'm going to save this content in this file.
```

Once you are done with the program, follow these steps −

- Press the key **esc** to come out of the edit mode.
- Press two keys **Shift + ZZ** together to come out of the file completely.

You will now have a file created with **filename** in the current directory.

```
$ vi filename
$
```

Editing Files

You can edit an existing file using the **vi** editor. We will discuss in short how to open an existing file −

```
$ vi filename
```

Once the file is opened, you can come in the edit mode by pressing the key **i** and then you can proceed by editing the file. If you want to move here and there inside a file, then first you need to come out of the edit mode by pressing the key **Esc**. After this, you can use the following keys to move inside a file −

- **l** key to move to the right side.
- **h** key to move to the left side.
- **k** key to move upside in the file.
- **j** key to move downside in the file.

So using the above keys, you can position your cursor wherever you want to edit. Once you are positioned, then you can use the **i** key to come in the edit mode. Once you are done with the editing in your file, press **Esc** and finally two keys **Shift + ZZ** together to come out of the file completely.

Display Content of a File

You can use the **cat** command to see the content of a file. Following is a simple example to see the content of the above created file −

```
$ cat filename
This is unix file....I created it for the first time.....
I'm going to save this content in this file.
$
```

You can display the line numbers by using the **-b** option along with the **cat** command as follows −

```
$ cat -b filename
1   This is unix file....I created it for the first time.....
2   I'm going to save this content in this file.
$
```

Counting Words in a File

You can use the **wc** command to get a count of the total number of lines, words, and characters contained in a file. Following is a simple example to see the information about the file created above −

```
$ wc filename
2  19 103 filename
$
```

Here is the detail of all the four columns −

- **First Column** − Represents the total number of lines in the file.
- **Second Column** − Represents the total number of words in the file.
- **Third Column** − Represents the total number of bytes in the file. This is the actual size of the file.
- **Fourth Column** − Represents the file name.

You can give multiple files and get information about those files at a time. Following is simple syntax −

```
$ wc filename1 filename2 filename3
```

Copying Files

To make a copy of a file use the **cp** command. The basic syntax of the command is −

```
$ cp source_file destination_file
```

Following is the example to create a copy of the existing file **filename**.

```
$ cp filename copyfile
$
```

You will now find one more file **copyfile** in your current directory. This file will exactly be the same as the original file **filename**.

Renaming Files

To change the name of a file, use the **mv** command. Following is the basic syntax −

```
$ mv old_file new_file
```

The following program will rename the existing file **filename** to **newfile**.

```
$ mv filename newfile
$
```

The **mv** command will move the existing file completely into the new file. In this case, you will find only **newfile** in your current directory.

Deleting Files

To delete an existing file, use the **rm** command. Following is the basic syntax −

```
$ rm filename
```

Caution − A file may contain useful information. It is always recommended to be careful while using this **Delete** command. It is better to use the **-i** option along with **rm** command.

Following is the example which shows how to completely remove the existing file **filename**.

```
$ rm filename
$
```

You can remove multiple files at a time with the command given below −

```
$ rm filename1 filename2 filename3
$
```

Standard Unix Streams

Under normal circumstances, every Unix program has three streams (files) opened for it when it starts up −

- **stdin** − This is referred to as the *standard input* and the associated file descriptor is 0. This is also represented as STDIN. The Unix program will read the default input from STDIN.
- **stdout** − This is referred to as the *standard output* and the associated file descriptor is 1. This is also represented as STDOUT. The Unix program will write the default output at STDOUT
- **stderr** − This is referred to as the *standard error* and the associated file descriptor is 2. This is also represented as STDERR. The Unix program will write all the error messages at STDERR.
-

File Permission / Access Modes

In this chapter, we will discuss in detail about file permission and access modes in Unix. File ownership is an important component of Unix that provides a secure method for storing files. Every file in Unix has the following attributes −

- **Owner permissions** − The owner's permissions determine what actions the owner of the file can perform on the file.
- **Group permissions** − The group's permissions determine what actions a user, who is a member of the group that a file belongs to, can perform on the file.
- **Other (world) permissions** − The permissions for others indicate what action all other users can perform on the file.

The Permission Indicators

While using **ls -l** command, it displays various information related to file permission as follows −

```
$ls -l /home/amrood
-rwxr-xr--  1 amrood   users 1024  Nov 2 00:10  myfile
drwxr-xr--- 1 amrood   users 1024  Nov 2 00:10  mydir
```

Here, the first column represents different access modes, i.e., the permission associated with a file or a directory.

The permissions are broken into groups of threes, and each position in the group denotes a specific permission, in this order: read (r), write (w), execute (x) −

- The first three characters (2-4) represent the permissions for the file's owner. For example, **-rwxr-xr--** represents that the owner has read (r), write (w) and execute (x) permission.
- The second group of three characters (5-7) consists of the permissions for the group to which the file belongs. For example, **-rwxr-xr--** represents that the group has read (r) and execute (x) permission, but no write permission.
- The last group of three characters (8-10) represents the permissions for everyone else. For example, **-rwxr-xr--** represents that there is **read (r)** only permission.

File Access Modes

The permissions of a file are the first line of defense in the security of a Unix system. The basic building blocks of Unix permissions are the **read**, **write**, and **execute** permissions, which have been described below −

Read

Grants the capability to read, i.e., view the contents of the file.

Write

Grants the capability to modify, or remove the content of the file.

Execute

User with execute permissions can run a file as a program.

Directory Access Modes

Directory access modes are listed and organized in the same manner as any other file. There are a few differences that need to be mentioned −

Read

Access to a directory means that the user can read the contents. The user can look at the **filenames** inside the directory.

Write

Access means that the user can add or delete files from the directory.

Execute

Executing a directory doesn't really make sense, so think of this as a traverse permission.

A user must have **execute** access to the **bin** directory in order to execute the **ls** or the **cd** command.

Changing Permissions

To change the file or the directory permissions, you use the **chmod** (change mode) command. There are two ways to use chmod — the symbolic mode and the absolute mode.

Using chmod in Symbolic Mode

The easiest way for a beginner to modify file or directory permissions is to use the symbolic mode. With symbolic permissions you can add, delete, or specify the permission set you want by using the operators in the following table.

Sr.No.	Chmod operator & Description
1	**+** Adds the designated permission(s) to a file or directory.
2	**-** Removes the designated permission(s) from a file or directory.

3	**=** Sets the designated permission(s).

Here's an example using **testfile**. Running **ls -1** on the testfile shows that the file's permissions are as follows −

```
$ls -l testfile
-rwxrwxr--  1 amrood   users 1024  Nov 2 00:10  testfile
```

Then each example **chmod** command from the preceding table is run on the testfile, followed by **ls –l**, so you can see the permission changes −

```
$chmod o+wx testfile
$ls -l testfile
-rwxrwxrwx  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod u-x testfile
$ls -l testfile
-rw-rwxrwx  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod g = rx testfile
$ls -l testfile
-rw-r-xrwx  1 amrood   users 1024  Nov 2 00:10  testfile
```

Here's how you can combine these commands on a single line −

```
$chmod o+wx,u-x,g = rx testfile
$ls -l testfile
-rw-r-xrwx  1 amrood   users 1024  Nov 2 00:10  testfile
```

Using chmod with Absolute Permissions

The second way to modify permissions with the chmod command is to use a number to specify each set of permissions for the file.

Each permission is assigned a value, as the following table shows, and the total of each set of permissions provides a number for that set.

Number	Octal Permission Representation	Ref
0	No permission	---
1	Execute permission	--x
2	Write permission	-w-
3	Execute and write permission: 1 (execute) + 2 (write) = 3	-wx

4	Read permission	r--
5	Read and execute permission: 4 (read) + 1 (execute) = 5	r-x
6	Read and write permission: 4 (read) + 2 (write) = 6	rw-
7	All permissions: 4 (read) + 2 (write) + 1 (execute) = 7	rwx

Here's an example using the testfile. Running **ls -1** on the testfile shows that the file's permissions are as follows −

```
$ls -l testfile
-rwxrwxr--  1 amrood   users 1024  Nov 2 00:10  testfile
```

Then each example **chmod** command from the preceding table is run on the testfile, followed by **ls –l**, so you can see the permission changes −

```
$ chmod 755 testfile
$ls -l testfile
-rwxr-xr-x  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod 743 testfile
$ls -l testfile
-rwxr---wx  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod 043 testfile
$ls -l testfile
----r---wx  1 amrood   users 1024  Nov 2 00:10  testfile
```

Changing Owners and Groups

While creating an account on Unix, it assigns a **owner ID** and a **group ID** to each user. All the permissions mentioned above are also assigned based on the Owner and the Groups.

Two commands are available to change the owner and the group of files −

- **chown −** The **chown** command stands for **"change owner"** and is used to change the owner of a file.
- **chgrp −** The **chgrp** command stands for **"change group"** and is used to change the group of a file.

Changing Ownership

The **chown** command changes the ownership of a file. The basic syntax is as follows −

```
$ chown user filelist
```

The value of the user can be either the **name of a user** on the system or the **user id (uid)** of a user on the system.

The following example will help you understand the concept −

```
$ chown amrood testfile
$
```

Changes the owner of the given file to the user **amrood**.

NOTE − The super user, root, has the unrestricted capability to change the ownership of any file but normal users can change the ownership of only those files that they own.

Changing Group Ownership

The **chgrp** command changes the group ownership of a file. The basic syntax is as follows −

```
$ chgrp group filelist
```

The value of group can be the **name of a group** on the system or **the group ID (GID)** of a group on the system.

Following example helps you understand the concept −

```
$ chgrp special testfile
$
```

Changes the group of the given file to **special** group.

SUID and SGID File Permission

Often when a command is executed, it will have to be executed with special privileges in order to accomplish its task.

As an example, when you change your password with the **passwd** command, your new password is stored in the file **/etc/shadow**.

As a regular user, you do not have **read** or **write** access to this file for security reasons, but when you change your password, you need to have the write permission to this file. This means that the **passwd** program has to give you additional permissions so that you can write to the file **/etc/shadow**.

Additional permissions are given to programs via a mechanism known as the **Set User ID (SUID)** and **Set Group ID (SGID)** bits.

When you execute a program that has the SUID bit enabled, you inherit the permissions of that program's owner. Programs that do not have the SUID bit set are run with the permissions of the user who started the program.

This is the case with SGID as well. Normally, programs execute with your group permissions, but instead your group will be changed just for this program to the group owner of the program.

The SUID and SGID bits will appear as the letter **"s"** if the permission is available. The SUID **"s"** bit will be located in the permission bits where the owners' **execute** permission normally resides.

For example, the command −

```
$ ls -l /usr/bin/passwd
-r-sr-xr-x  1   root   bin  19031 Feb 7 13:47  /usr/bin/passwd*
$
```

Shows that the SUID bit is set and that the command is owned by the root. A capital letter **S** in the execute position instead of a lowercase **s** indicates that the execute bit is not set.

If the sticky bit is enabled on the directory, files can only be removed if you are one of the following users −

- The owner of the sticky directory
- The owner of the file being removed
- The super user, root

To set the SUID and SGID bits for any directory try the following command −

```
$ chmod ug+s dirname
$ ls -l
drwsr-sr-x 2 root root  4096 Jun 19 06:45 dirname
$
```

Pipes and Filters

In this chapter, we will discuss in detail about pipes and filters in Unix. You can connect two commands together so that the output from one program becomes the input of the next program. Two or more commands connected in this way form a pipe.

To make a pipe, put a vertical bar (**|**) on the command line between two commands.

When a program takes its input from another program, it performs some operation on that input, and writes the result to the standard output. It is referred to as a ***filter***.

The grep Command

The grep command searches a file or files for lines that have a certain pattern. The syntax is −

```
$grep pattern file(s)
```

The name **"grep"** comes from the ed (a Unix line editor) command **g/re/p** which means "globally search for a regular expression and print all lines containing it".

A regular expression is either some plain text (a word, for example) and/or special characters used for pattern matching.

The simplest use of grep is to look for a pattern consisting of a single word. It can be used in a pipe so that only those lines of the input files containing a given string are sent to the standard output. If you don't give grep a filename to read, it reads its standard input; that's the way all filter programs work −

```
$ls -l | grep "Aug"
-rw-rw-rw-   1 john  doc     11008 Aug  6 14:10 ch02
-rw-rw-rw-   1 john  doc      8515 Aug  6 15:30 ch07
-rw-rw-r--   1 john  doc      2488 Aug 15 10:51 intro
-rw-rw-r--   1 carol doc      1605 Aug 23 07:35 macros
$
```

There are various options which you can use along with the **grep** command −

Sr.No.	Option & Description
1	**-v** Prints all lines that do not match pattern.
2	**-n** Prints the matched line and its line number.
3	**-l** Prints only the names of files with matching lines (letter "l")
4	**-c** Prints only the count of matching lines.
5	**-i** Matches either upper or lowercase.

Let us now use a regular expression that tells grep to find lines with **"carol"**, followed by zero or other characters abbreviated in a regular expression as ".*"), then followed by "Aug".−

Here, we are using the ***-i*** option to have case insensitive search −

```
$ls -l | grep -i "carol.*aug"
-rw-rw-r--   1 carol doc      1605 Aug 23 07:35 macros
$
```

The sort Command

The **sort** command arranges lines of text alphabetically or numerically. The following example sorts the lines in the food file −

```
$sort food
Afghani Cuisine
Bangkok Wok
Big Apple Deli
Isle of Java

Mandalay
Sushi and Sashimi
Sweet Tooth
Tio Pepe's Peppers
$
```

The **sort** command arranges lines of text alphabetically by default. There are many options that control the sorting −

Sr.No.	Description
1	**-n** Sorts numerically (example: 10 will sort after 2), ignores blanks and tabs.
2	**-r** Reverses the order of sort.
3	**-f** Sorts upper and lowercase together.
4	**+x** Ignores first **x** fields when sorting.

More than two commands may be linked up into a pipe. Taking a previous pipe example using **grep**, we can further sort the files modified in August by the order of size.

The following pipe consists of the commands **ls**, **grep**, and **sort** −

```
$ls -l | grep "Aug" | sort +4n
-rw-rw-r--  1 carol doc      1605 Aug 23 07:35 macros
-rw-rw-r--  1 john  doc      2488 Aug 15 10:51 intro
-rw-rw-rw-  1 john  doc      8515 Aug  6 15:30 ch07
-rw-rw-rw-  1 john  doc     11008 Aug  6 14:10 ch02
$
```

This pipe sorts all files in your directory modified in August by the order of size, and prints them on the terminal screen. The sort option +4n skips four fields (fields are separated by blanks) then sorts the lines in numeric order.

The pg and more Commands

A long output can normally be zipped by you on the screen, but if you run text through more or use the **pg** command as a filter; the display stops once the screen is full of text.

Let's assume that you have a long directory listing. To make it easier to read the sorted listing, pipe the output through **more** as follows −

```
$ls -l | grep "Aug" | sort +4n | more
-rw-rw-r--  1 carol doc      1605 Aug 23 07:35 macros
-rw-rw-r--  1 john  doc      2488 Aug 15 10:51 intro
-rw-rw-rw-  1 john  doc      8515 Aug  6 15:30 ch07
-rw-rw-r--  1 john  doc     14827 Aug  9 12:40 ch03
       .
       .
       .
-rw-rw-rw-  1 john  doc     16867 Aug  6 15:56 ch05
--More--(74%)
```

The screen will fill up once the screen is full of text consisting of lines sorted by the order of the file size. At the bottom of the screen is the **more** prompt, where you can type a command to move through the sorted text.

Once you're done with this screen, you can use any of the commands listed in the discussion of the more program.

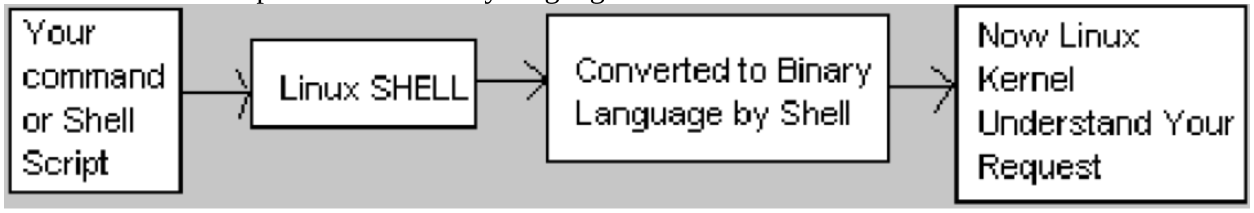
Directory Content
during failures are here.
/misc For miscellaneous purposes.
/mnt Standard mount point for external file systems, e.g. a CD-ROM or a digital camera.
/net Standard mount point for entire remote file systems
/opt Typically contains extra and third party software.
/proc
A virtual file system containing information about system resources. More information about the meaning of the files in proc is obtained by entering the command **man** ***proc*** in a terminal window. The file proc.txt discusses the virtual file system in detail.
/root The administrative user's home directory. Mind the difference between /, the root directory and /root, the home directory of the *root* user.
/sbin Programs for use by the system and the system administrator.
/tmp Temporary space for use by the system, cleaned upon reboot, so don't use this for saving any work!
/usr Programs, libraries, documentation etc. for all user-related programs.
/var

Storage for all variable files and temporary files created by users, such as log files, the mail queue, the print spooler area, space for temporary storage of files downloaded from the Internet, or to keep an image of a CD before burning it.

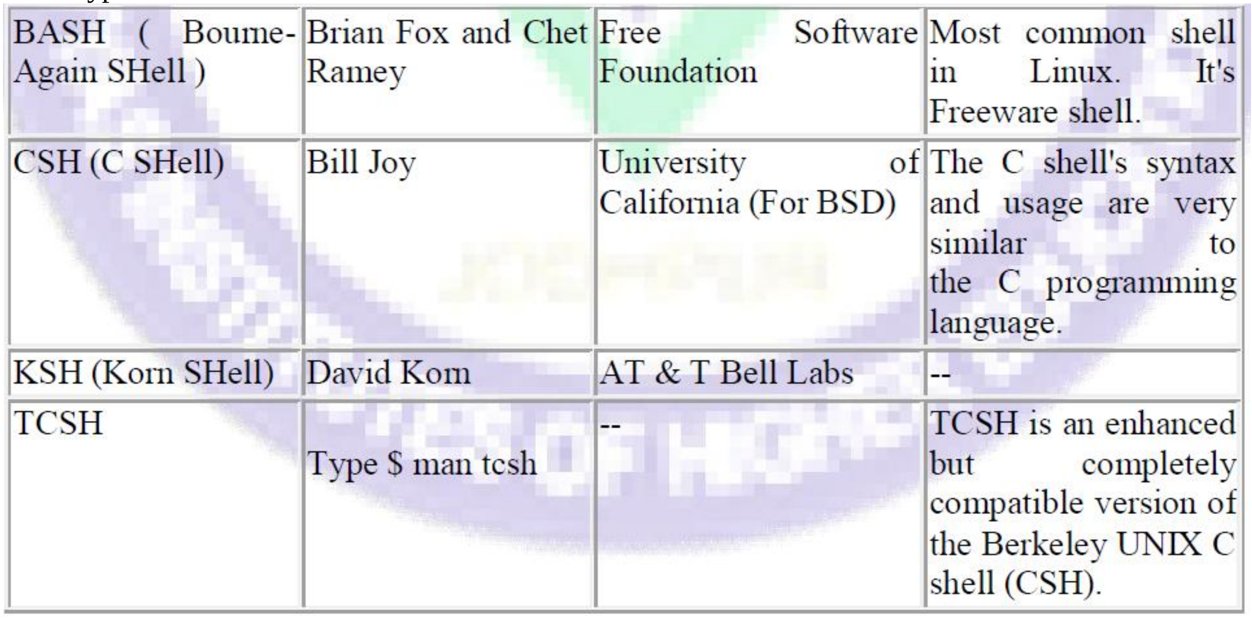
Chapter 2
Basic Linux Commands

2.1 Shell

Computer understand the language of 0's and 1's called binary language, In earlydays of computing, instruction are provided using binary language, which is difficult for all of us, to read and write. So in O/s there is special program called Shell. Shell accepts your instruction or commands in English and translates it into computers native binary language.


Your command or Shell Script
Linux SHELL
Converted to Binary Language by Shell
Now Linux Kernel Understand Your Request

2.1.1 Type **of Shell**



BASH (Bourne-Again SHell)	Brian Fox and Chet Ramey	Free Software Foundation	Most common shell in Linux. It's Freeware shell.
CSH (C SHell)	Bill Joy	University of California (For BSD)	The C shell's syntax and usage are very similar to the C programming language.
KSH (Korn SHell)	David Korn	AT & T Bell Labs	--
TCSH	Type $ man tcsh	--	TCSH is an enhanced but completely compatible version of the Berkeley UNIX C shell (CSH).

2.2 Linux Commands

2.1.1 The Manual (terminal mode)

man

This command brings up the online UNIX manual. Use it on each of the commands below.
Usage: man [command name]

eg: man pwd You will see the manual for the pwd command.

2.1.2 File Handling Commands

- **mkdir – make directories**
  create directory[ies] if they are not already exists.
  Usage: mkdir <DIREECTORY NAME>
  eg. mkdir hiren
- **ls – list directory contents**
  Usage: ls [OPTION]... [FILE]...
  eg. ls Listing Directory contents including only file names
  ls l
  Long listing of directory contents including user permissions, number
  of links, size of file or directory, day and time of last modification
  and file or directory name.
  ls –a Listing all files including hidden files
- **cd – changes directories**
  Usage: cd [DIRECTORY NAME]
  Eg. cd hiren
- **pwd – print working directory**
  Shows what directory (folder) you are in.
  In Linux, your home directory is /home/username.
  Usage: pwd
  eg. pwd show present working directory
  /home/username
- **rmdir- Remove directories**
  remove directory[ies] if they are empty.
  Usage: rmdir [DIRECTORY NAME]
  Eg. rmdir hiren remove hiren directory if this is empty.
- **rm-remove files or directories**
  remove file[s] or directory[ies]
  Usage:rm –[option] [DIRECTORY NAME OR FILE NAME]
  Eg.
  rm –i [FILENAME] remove file interactively. This will ask before removing file.
  rm –f [FILENAME] removefile forcefully.
  rm –r [FILENAME] recusively remove non empty directory.
- **mv-move**
  move or rename files or directories
  Usage: mv [SOURCE DIRECTORY] [DESTINATION DIRECTORY]
  mv [OLD FILENAME] [NEW FILENAME]
  Eg. mv linixdir hiren renaming or moving directory linixdir as hiren. After
  execution of command, the destination files are only
  available.
- **cp – copy**
  copy files and directories
  Usage: cp [OPTION] SOURCE FILE] [DESTINATION FILE]
  eg. cp sample.txt sample_copy.txt After execution of command, the both source
  and destination files are available.

2.2.3 Text Processing

- **cat – concatenate files and print on the standard output**
  Usage: cat [OPTION] [FILE]...
  eg. cat file1.txt file2.txt

cat when supplied with more than one file will concatenate the files without any header
information.

- .**cat used--to display the contents of a small file on terminal**
  usage: cat [file name]
  cat- To create file
  Usage: cat > [file name]
  NOTE: Press and hold CTRL key and press D to stop or to end file (CTRL+D)
  cat – to apend text at the end of file
  Usage: cat >> [file name]
  NOTE: Press and hold CTRL key and press D to stop or to end file (CTRL+D)
- **echo – display a line of text**
  Usage: echo [OPTION] [string] ...
  eg. echo I love India
  echo $HOME

- **wc -ommand is used to count lines, words and characters, depending on the option used.**
  Usage: wc [options] [file name]
  You can just print number of lines, number of words or number of charcters by using followi ng
  options:
  l : Number of lines
  w : Number of words
  c : Number of characters
  Eg. wc file.txt count number of lines, words and character (including whitespaces , newline etc) in a file.
- **grep print lines matching a pattern**
  Usage: grep [OPTION] PATTERN [FILE]...
  eg. grep i
  apple sample.txt
  Options:
  -i case-insensitive search
  -n show the line# along with the matched line
  -v invert match, e.g. find all lines that do NOT match
  -w match entire words, rather than substrings

2.2.4 Process Management

- **kill –**
  kill ends one or more process IDs. In order to do this you must own the process or be designated a privileged user. To find the process ID of a certain job use ps.
  Usage: kill [options] IDs
  Eg. kill 1234
- **ps-**
  The "ps" command (process statistics) lets you check the status of processes that are running on your Unix system.
  Usage:ps The ps command by itself shows minimal information
  about the processes you are running. Without any
  arguments, this command will not show information about
  other processes running on the system.

2.3 Introduction to Vi

Under Linux, there is a free version of Vi called Vim (Vi Improved). Vi (pronounced vee- eye) is an editor that is fully in text mode, which means that all actions are carried out with the help of text commands. This editor, although it may appear of little practical use at first, is very powerful and can be very helpful in case the graphical interface malfunctions.

Syntax : vi name_of_the_file
Once the file is open, you can move around by using cursors. Press I to switch to insert Mode. Press escape key to switch to command mode,

- **Basic commands**
  Description
  :q Quit the editor (without saving)
  :q! Forces the editor to quit without saving
  : wq Saves the document and quits the editor
  : filename Saves the document under the specified name
  : set nu set serial number to lines

2.4 How to write shell script

Following steps are required to write shell script:
(1) Use any editor like vi or mcedit to write shell script.
(2) After writing shell script set execute permission for your script as follows
syntax:
chmod permission your-script-name
Examples:

```
$ chmod +x your-script-name
$ chmod 755 your-script-name
```

Note: This will set read write execute(7) permission for owner, for group and other permission is read and execute only(5).
ls -l command earlier presented a long of listing file with a line like the following for each file:
-rw-r--r-- 1 root user 0 2009-04-28 08:26 newfile.txt
Here the first character in the first column (-) indicates that the file is a normal file. The next 9 characters indicate the access permissions for the file. The next set of 9 characters is divided into3 groups of 3 cha- racters. Purpose of these characters is as under:
(-) represents no permission
(r) represents 'read' permission
(w) represents 'write' permission
(x) represents 'execute' permission

Permission	Octal number	Equivalent Symbol
Read	4	r--
Write	2	-w-
Execute	1	--x

The three group represents user (owner of the file), group(to which the owner belongs) and others (any other user of the system) respectively. Three characters in each group are for 'read', 'write' and 'execute' permission respectively. In our example, the owner has 'read' and 'write' permission for the file and everyone else has only read permission. For a normal file, read, write and execute permissions are obvious. For a directory, read and write permissions mean that to read the contents of the directory and create new entries in the directory. Execute permission means that one can search in the directory but not read from or write to the directory.
You can use the chmod command to change the access permissions of a file or a directory. To specify permissions for a file with chmod, any of the following two methods can be used.

Symbol	**Meaning**
u	User
g	Group
o	Other
a	All (equals to ugo)
+	Add Permission
-	Remove a permission
r	Read Permission
w	Write permission
x	Execute permission

Syntax: chmod u+x filename
(3) Execute your script as
syntax:
bash your-script-name
sh your-script-name
./your-script-name
Examples:

```
$ bash bar
$ sh bar
$ ./bar
```

Exercise:

1. Verify that you are in your home directory.
2. Make the directory adir using the following command: mkdir adir
3. List the files in the current directory to verify that the directory adir has been made correctly.
4. Change directories to adir.
5. Verify that you have succeeded in moving to the adir directory.
6. Verify that the file testfile exists.
7. List the contents of the file testfile to the screen.
8. Make a copy of the file testfile under the name secondfile.
9. Verify that the files testfile and secondfile both exist.
10. List the contents of both testfile and secondfile to the monitor screen.
11. Delete the file testfile.
12. Verify that testfile has been deleted.
13. Clear the window.
14. Rename secondfile to thefile.
15. Issue the command to find out how large thefile is. How big is it?
16. Copy thefile to your home directory.
18. Remove thefile from the current directory.
19. Verify that thefile has been removed.
20. Copy thefile from your home directory to the current directory.
21. Verify that the file has been copied from your home directory to the current directory.

Here is a sample run of the script −

```
echo "What is your name?"
read PERSON
echo "Hello, $PERSON"
```

```
$./test.sh
```

```
What is your name?
Madan Mohan
Hello, Madan Mohan
$
```

Chapter 3
Shell Scripts-I
3.1 Variables
3.1.1 Variables
When a script starts all environment variables are turned into shell variables. New variables can be instantiated like this:
name=*value*
You must do it exactly like that, with no spaces either side of the equals sign, the name must only be made up of alphabetic characters, numeric characters and underscores; it cannot begin with a numeric character.
3.1.2 Command Line Arguments
Command line arguments are treated as special variables within the script, the reason I am calling them variables is because they can be changed with the **shift** command. The command line arguments are enumerated in the following manner *$0, $1, $2, $3, $4, $5, $6, $7, $8* and *$9*. *$0* is special in that it corresponds to the name of the script itself. *$1* is the first argument; *$2* is the second argument and so on. To reference after the ninth argument you must enclose the number in brackets like this *${nn}*.
You can use the **shift** command to shift the arguments 1 variable to the left so that *$2* becomes *$1*, *$1* becomes *$0* and so on, *$0* gets scrapped because it has nowhere to go, this can be useful to process all the arguments using a loop, using one variable to reference the first argument and **shifting** until you have exhausted the arguments list. As well as the commandline arguments there are some special builtin variables:

- ***$#*** represents the parameter count. Useful for controlling loop constructs that need to process each parameter.
- ***$@*** expands to all the parameters separated by spaces. Useful for passing all the parameters to some other function or program.
- ***$-*** expands to the flags(options) the shell was invoked with. Useful for controlling program flow based on the flags set.

- ***$$*** expands to the process id of the shell innovated to run the script. Useful for creating unique temporary filenames relative to this instantiation of the script.

3.1.3 Command Substitution
In the words of the SH manual “Command substitution allows the output of a command to be substituted in place of the command name itself”. There are two ways this can be done. The first is to enclose the command like this:
$(command)
The second is to enclose the command in back quotes like this: `command`
The command will be executed in a sub-shell environment and the standard output of the shell will replace the command substitution when the command completes.
3.1.4. Arithmetic Expansion
Arithmetic expansion is also allowed and comes in the form: $((expression))
The value of the expression will replace the substitution. Eg: echo $((1 + 3 + 4))

Will echo "8" to stdout

3.2 To evaluate Arithmetic Expressions:

There are various operators supported by each shell. We will discuss in detail about Bourne shell (default shell) in this chapter.

We will now discuss the following operators −

- Arithmetic Operators
- Relational Operators
- Boolean Operators
- String Operators
- File Test Operators

Bourne shell didn't originally have any mechanism to perform simple arithmetic operations but it uses external programs, either **awk** or **expr**.

The following example shows how to add two numbers −

```
#!/bin/sh

val=`expr 2 + 2`
echo "Total value : $val"
```

The above script will generate the following result −

```
Total value : 4
```

The following points need to be considered while adding −

- There must be spaces between operators and expressions. For example, 2+2 is not correct; it should be written as 2 + 2.
- The complete expression should be enclosed between **‘ ‘**, called the backtick.

Arithmetic Operators

The following arithmetic operators are supported by Bourne Shell.

Assume variable **a** holds 10 and variable **b** holds 20 then −

Show Examples

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator	`expr $a + $b` will give 30

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

It is very important to understand that all the conditional expressions should be inside square braces with spaces around them, for example **[$a == $b]** is correct whereas, **[$a==$b]** is incorrect.

All the arithmetical calculations are done using long integers.

Relational Operators

Bourne Shell supports the following relational operators that are specific to numeric values. These operators do not work for string values unless their value is numeric.

For example, following operators will work to check a relation between 10 and 20 as well as in between "10" and "20" but not in between "ten" and "twenty".

Assume variable **a** holds 10 and variable **b** holds 20 then −

Show Examples

Operator	**Description**	**Example**

-eq	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a -eq $b] is not true.
-ne	Checks if the value of two operands are equal or not; if values are not equal, then the condition becomes true.	[$a -ne $b] is true.
-gt	Checks if the value of left operand is greater than the value of right operand; if yes, then the condition becomes true.	[$a -gt $b] is not true.
-lt	Checks if the value of left operand is less than the value of right operand; if yes, then the condition becomes true.	[$a -lt $b] is true.
-ge	Checks if the value of left operand is greater than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -ge $b] is not true.
-le	Checks if the value of left operand is less than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -le $b] is true.

It is very important to understand that all the conditional expressions should be placed inside square braces with spaces around them. For example, **[$a <= $b]** is correct whereas, **[$a <= $b]** is incorrect.

Boolean Operators

The following Boolean operators are supported by the Bourne Shell.

Assume variable **a** holds 10 and variable **b** holds 20 then −

Show Examples

Operator	Description	Example
!	This is logical negation. This inverts a true condition into false and vice versa.	[! false] is true.

-o	This is logical **OR**. If one of the operands is true, then the condition becomes true.	[$a -lt 20 -o $b -gt 100] is true.
-a	This is logical **AND**. If both the operands are true, then the condition becomes true otherwise false.	[$a -lt 20 -a $b -gt 100] is false.

String Operators

The following string operators are supported by Bourne Shell.

Assume variable **a** holds "abc" and variable **b** holds "efg" then −

Show Examples

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

File Test Operators

We have a few operators that can be used to test various properties associated with a Unix file.

Assume a variable **file** holds an existing file name "test" the size of which is 100 bytes and has **read**, **write** and **execute** permission on −

Show Examples

Operator	Description	Example
-b file	Checks if file is a block special file; if yes, then the condition becomes true.	[-b $file] is false.
-c file	Checks if file is a character special file; if yes, then the condition becomes true.	[-c $file] is false.
-d file	Checks if file is a directory; if yes, then the condition becomes true.	[-d $file] is not true.
-f file	Checks if file is an ordinary file as opposed to a directory or special file; if yes, then the condition becomes true.	[-f $file] is true.
-g file	Checks if file has its set group ID (SGID) bit set; if yes, then the condition becomes true.	[-g $file] is false.
-k file	Checks if file has its sticky bit set; if yes, then the condition becomes true.	[-k $file] is false.
-p file	Checks if file is a named pipe; if yes, then the condition becomes true.	[-p $file] is false.
-t file	Checks if file descriptor is open and associated with a terminal; if yes, then the condition becomes true.	[-t $file] is false.
-u file	Checks if file has its Set User ID (SUID) bit set; if yes, then the condition becomes true.	[-u $file] is false.
-r file	Checks if file is readable; if yes, then the condition becomes true.	[-r $file] is true.
-w file	Checks if file is writable; if yes, then the condition becomes true.	[-w $file] is true.

-x file	Checks if file is executable; if yes, then the condition becomes true.	[-x $file] is true.
-s file	Checks if file has size greater than 0; if yes, then condition becomes true.	[-s $file] is true.
-e file	Checks if file exists; is true even if file is a directory but exists.	[-e $file] is true.

exprexpr
writes the result of the expression on the standard output. This command is primarily intended for arithmetic and string manipulation.
Syntax:
Operators:

expr1 | *expr2*
results in the value *expr1* if *expr1* is true; otherwise it results in the value of *expr2*.
expr1 & *expr2*
results in the value of *expr1* if both expressions are true; otherwise it results in 0
expr1 <= *expr2*
expr1 < *expr2*
expr1 = *expr2*
expr1!= *expr2*
expr1 >= *expr2*
expr1 > *expr2*
If both *expr1* and *expr2* are numeric, **expr** compares them as numbers; otherwise it compares them as strings. If the comparison is true, the expression results in 1; otherwise it results in 0.
expr1 + *expr2*
expr1 - *expr2*
performs addition or subtraction on the two expressions. If either expression is not a number, **expr** exits with an error.
expr1 * *expr2*
expr1 / *expr2*
expr1 % *expr2*
performs multiplication, division, or modulus on the two expressions. If either expression is not a number, **expr** exits with an error.

3.3 Reading from Standard input

read
reading values from standard input.
Syntax: read variable_name1 variable_name2 …..
Eg: read num

Exercise:

3.1 Write a shell script to evaluate arithmetic operations.

Script1:

```
echo “enter two integer number”
read a
read b
c=`expr $a + $b`
echo “sum=$c”
c=`expr $a - $b`
echo “sub=$c”
c=`expr $a / $b`
echo “div=$c”
c=`expr $a \* $b`
echo “multiplication=$c”
c=`expr $a % $b`
echo “remainder=$c”
```

3.2 Write a shell script to calculate simple interest.

Script2:

```
echo “enter the principal value, rate of interest and time period”
read p
read r
read t
si=$(expr $p \* $r \* $t / 100)
echo “simple interset=$si”
```

3.4 Control Constructs

The flow of control within SH scripts is done via four main constructs; if...then...elif..else, do...while, for and case.

3.4.1 If..Then..Elif..Else

This construct takes the following generic form, The parts enclosed within ([) and (])
are optional:

if *list*
then *list*
[elif *list*
then *list*] ...
[else *list*]
fi

When a Unix command exits it exits with what is known as an *exit status*, this indicates to anyone who wants to know the degree of success the command had in performing whatever task it was supposed to do, usually when a command executes without error it terminates with an exit status of zero. An exit status of some other value would indicate that some error had occurred, the details of which would be specific to the command. The commands' manual pages detail the exit status messages that they produce.

A list is defined in the SH as "a sequence of zero or more commands separated by newlines, semicolons, or ampersands, and optionally terminated by one of these three characters.", hence in the generic definition of the *if* above the list will determine which of the execution paths the script takes. For example, there is a command called **test** on Unix which evaluates an expression and if it evaluates to true will return zero and will return one otherwise, this is how we can test conditions in the *list* part(s) of the *if* construct because **test** is a command.

We do not actually have to type the **test** command directly into the *list* to use it, it can be implied by encasing the test case within ([) and (]) characters.

Numerical Comparision

Symbol	Description	Usage
-eq	Equals	$A -eq $B
-ne	Not equal	$A -ne $B
-lt	Less than	$A –le $B
-le	Less than or equal to	$A -le $B
-gt	Greater than	$A -ge $B
-ge	Greater than or equal to	$A –ge $B

String Comparision

Symbol	Description	Usage
=	Equals	$A = $B
!=	Not Equal	$A != $B
-n	Not a null string	-n $str
-z	Null String	-z $str

Testing For File

Symbol	Description	Usage
-r	Read Permission	-r $filename
-w	Write Permission	-w $filename
-x	Execution Permission	-x $filename
-f	File Exists	-f $filename
-d	Directory Exits	-d $filename
-c	Special Character File	-c $filename
-b	Block Special File	-b $filename
-s	Size of File is not zero	-s $filename

3.4.2 Do...While

The *Do...While* takes the following generic form:

```
while list
        do list
done
```

In the words of the SH manual "The two lists are executed repeatedly while the exit status of the first list is zero." there is a variation on this that uses until in place of while which executes *until* the exit status of the first list is zero

3.4.3 For

The syntax of the for command is:

```
for variable in word ...
do list
done
```

The SH manual states “The words are expanded, and then the list is executed repeatedly with the variable set to each word in turn.”. A word is essentially some other variable that contains a list of values of some sort, the *for* construct assigns each of the values in the word to variable and then variable can be used within the body of the construct, upon completion of the body variable will be assigned the next value in word until there are no more values in word.

3.4.4 Case

The case construct has the following syntax:

```
        case word in
        pattern) list ;;
        ...
        esac
```

An example of this should make things clearer:

```
!#/bin/sh
case $1 in
1) echo 'First Choice';;
2) echo 'Second Choice';;
*) echo 'Other Choice';;
esac
```

"1", "2" and "*" are patterns, word is compared to each pattern and if a match is found the body of the corresponding pattern is executed, we have used "*" to represent everything, since this is checked last we will still catch "1" and "2" because they are checked first. In our example word is "$1", the first parameter, hence if the script is ran with the argument "1" it will output "First Choice", "2" "Second Choice" and anything else "Other Choice". In this example we compared against numbers (essentially still a string comparison however) but the pattern can be more complex,

3.4.5 Functions

The syntax of an SH function is defined as follows:

```
name ( ) command
```

It is usually laid out like this:

```
name() {
commands
}
```

A function will return with a default exit status of zero, one can return different exit status' by using the notation return *exit status*. Variables can be defined locally within a function using local *name*=*value*. The example below shows the use of a user defined increment function:

Example: Increment Function Example

```
#!/bin/sh
inc()
{ # The increment is defined first so we can use it
echo $(($1 + $2))
# We echo the result of the first parameter plus the second parameter
}
# We check to see that all the command line
arguments are present
if [ "$1" "" ] || [ "$2" = "" ] || [ "$3" = "" ] then
        echo USAGE:
        echo " counter startvalue incrementvalue endvalue"
        else
        count=$1 # Rename are variables with clearer names
        value=$2
        end=$3
        while [ $count -lt $end ] # Loop while count is less than end
        do
                echo $count
                count=$(inc $count $value)
                # Call increment with count and value as parameters
        done # so that count is incremented by value
fi
```

Exercise:

3.3 Write a shell Script to determine largest among three integer number.

Script3:

```
echo “enter three integer number”
read a
read b
read c
if [ $a –ge $b ]
then
if [ $a –ge $c ]
then
echo “$a is largest number”
else
echo “$c is largest number”
fi
elif [ $b –ge $c ]
then
echo “$b is largest number”
else
echo “$c is largest number”
fi
```

3.4 Write a shell script to determine a given year is leap year or not.

Script4:

```
echo “enter any year”
read y
if [ $(expr $y % 100) –eq 0 ]
then
if [$(expr $y % 400) –eq 0 ]
then
echo “$y is leap year”
else
echo “$y is leap year”
fi
elif [ $(expr $y % 4 ) –eq 0 ]
then
echo “$y is leap year”
else
 echo “$y is leap year”
fi
```

3.5 Write a shell script to print multiplication table of given number using while statement.

Script5:

```
echo “enter any num”
read n
i=1;
while [ $i –le $n ]
do
m=$(expr $n \* $i);
echo “$n * $i = $m”
i=$(expr $i + 1);
done
```

3.6 Write a shell script to compare two string.

Script6:

```
echo “enter two string”
read a
read b
if [ -z $a ]
```

```
then
echo " First String is empty: Null String"
fi
if [ -z $b ]
then
echo " First String is empty: Null String"
fi
if [ $a = $b ]
then
echo "Strings are equal: strings Matched"
else
echo "Strings are not equal: Strings not match"
fi
```

3.7 Write a shell script to read and check the directory exists or not, if not make directory.

Script7:

```
echo "enter name of directory"
read dir
if [ -d $dir ]
then
echo "Directory $dir Exits!"
else
mkdir $dir
 fi
```

3.8 Write a shell script to read and check the directory exists or not, if not make file.

Script7:

```
echo "enter name of file"
read filename
if [ -f $filename ]
then
echo "File $filename Exits!"
else
touch $filename
fi
```

3.9 Write a shell script to implement menu driven program to perform all arithmetic operation using case statement.

Script9:

```
echo "enter two integer values"
read a
read b
echo –e "Menu \n 1 for Addition \n 2 for Substraction \n 3 for Multiplication \n 4 for
Division \n 5 for Remainder"
echo "enter choice"
read ch
case $ch in
1) echo "Sum=$(expr $a + $b)";;
2) echo "Substraction=$(expr $a - $b)";;
3) echo "Multiplication=$(expr $a \* $b)";;
4) echo "Division=$(expr $a / $b)";;
5) echo "Remainder=$(expr $a % $b)";;
6) echo "invalid Choice:Try Again!"
esac
```

3.10 Write a shell script to do:
(i). display list of directory contents
(ii). Name of current directory
(iii). Who is logged on
(iv). Long listing of directory contents
according to choice of user.

Script10:

```
echo –e “Menu \n 1 for listing directory content \n 2 for print name of current
directory \n 3 for Show who is logged on \n 4 Show directory content using long
listing format “
echo “enter your choice “
read ch
case $ch in
1) ls;;
 2) pwd;;
3) who;;
4) ls –l;;
*) echo “Invalid Choice: Try Again!!”
esac
```

3.11 Write a shell script to print following pattern.

```
*
* *
* * *
* * * *
```

Script11 :

```
echo “ enter number of rows”
read n
i=1
while [ $i –le $n ]
do
j=1
while [ $j –le $i ]
do
echo –n “*”
j=$(expr $j + 1)
done
echo
i=$(expr $i + 1)
done
```

3.12 Write a shell script to read the file word by word with serial number.

Script12:

```
i=1
for line in $( cat file1)
do
echo –e “$i \t ”
echo –e “$line \n”
i=$(expr $i + 1 )
done
```

3.13 Write a shell script to read the file word by word with serial number.
Script12:

```
i=1
while read line
do
echo –e “$i \t ”
echo –e “$line \n”
i=$(expr $i + 1 )
done < file
```

3.14 Write a shell script to do
(i). counting number of user logged in
(ii). Printing column list of your current directory
(iii). Running your job after logging out.
Script12:

```
echo “ Number of users logged in”
i=0
who > file # result of who command is appended to file
while read line
do
i=$(expr $i + 1)
done < file
echo “ $i”
i=1
for line in $( cat file)
do
echo –e “$i \t ”
echo –e “$line \n”
i=$(expr $i + 1 )
done
nohup sort file
```

3.15. compare two integers using the numeric comparison operator to determine if they are equal in value. Remember, 0 signals true, while 1 indicates false:

```
#!/bin/bash
string_a="UNIX"
string_b="GNU"
echo "Are $string_a and $string_b strings equal?"
[ $string_a = $string_b ]
echo $?

num_a=100
num_b=100
echo "Is $num_a equal to $num_b ?"
[ $num_a -eq $num_b ]
echo $?
```

Save the above script as eg. `comparison.sh` file, make it executable and execute:

```
$ chmod +x compare.sh


$ ./compare.sh
```

```
Are UNIX and GNU strings equal?

1

Is 100 equal to 100 ?

0
```

3.5 running job after logout
nohup
used to continue runing job / task after running out and appending output to nohup.out output file by default.
Usage: nohup [command_name]

3.6 Important Linux commands
3.6.1 date

Chapter 4: Shell Script to Create a File

After reviewing the above backup.sh script, you will notice the following changes to the code:

- we have defined a new function called `total_files`. The function utilized the `find` and `wc` commands to determine the number of files located within a directory supplied to it during the function call
- we have defined a new function called `total_directories`. Same as the above `total_files` function it utilized the `find` and `wc` commands however it reports a number of directories within a directory supplied to it during the function call

Q1. This bash script is used to backup a user's home directory to /tmp/.

```
#!/bin/bash
user=$(whoami)
input=/home/$user
output=/tmp/${user}_home_$(date +%Y-%m-%d_%H%M%S).tar.gz

# The function total_files reports a total number of files for a given
directory.
function total_files {
        find $1 -type f | wc -l
}

# The function total_directories reports a total number of directories
# for a given directory.
function total_directories {
        find $1 -type d | wc -l
}

tar -czf $output $input 2> /dev/null

echo -n "Files to be included:"
total_files $input
echo -n "Directories to be included:"
total_directories $input

echo "Backup of $input completed!"

echo "Details about the output backup file:"
ls -l $output
```

Q2. to perform a sanity check by comparing the difference between the total number of the files before and after the backup command. Here is the new updated `backup.sh` script:

```
#!/bin/bash

user=$(whoami)
input=/home/$user
output=/tmp/${user}_home_$(date +%Y-%m-%d_%H%M%S).tar.gz

function total_files {
        find $1 -type f | wc -l
}

function total_directories {
        find $1 -type d | wc -l
}

function total_archived_directories {
        tar -tzf $1 | grep  /$ | wc -l
}

function total_archived_files {
        tar -tzf $1 | grep -v /$ | wc -l
}

tar -czf $output $input 2> /dev/null

src_files=$( total_files $input )
src_directories=$( total_directories $input )

arch_files=$( total_archived_files $output )
arch_directories=$( total_archived_directories $output )

echo "Files to be included: $src_files"
echo "Directories to be included: $src_directories"
echo "Files archived: $arch_files"
echo "Directories archived: $arch_directories"

if [ $src_files -eq $arch_files ]; then
        echo "Backup of $input completed!"
        echo "Details about the output backup file:"
        ls -l $output
else
        echo "Backup of $input failed!"
fi
```

There are few additions to the above script. Highlighted are the most important changes.

Lines 15 - 21 are used to define two new functions returning a total number of files and directories included within the resulting compressed backup file. After the backup **Line 23** is executed, on **Lines 25 - 29** we declare new variables to hold the total number of source and destination files and directories.

The variables concerning backed up files are later used on **Lines 36 - 42** as part of our new conditional if/then/else statement returning a message about the successful backup on **Lines 37 - 39**only if the total number of both, source and destination backup files is equal as stated on **Line 36**.

Here is the script execution after applying the above changes:

```
$ ./backup.sh

Files to be included: 24

Directories to be included: 4

Files archived: 24

Directories archived: 4

Backup of /home/linuxconfig completed!

Details about the output backup file:

-rw-r--r-- 1 linuxconfig linuxconfig 235569 Sep 12 12:43
/tmp/linuxconfig_home_2017-09-12_124319.tar.gz
```

Shell Script Examples

This section presents several shell script examples.

Hello World

Example 9. Hello World

```
#!/bin/sh
echo "Hello world"
```

Using Arguments

Example 10. Shell Script Arguments

```
#!/bin/bash

# example of using arguments to a script
echo "My first name is $1"
echo "My surname is $2"
echo "Total number of arguments is $#"
```

Save this file as `name.sh`, set execute permission on that file by typing **chmod a+x name.sh** and then execute the file like this: **./name.sh**.

```
$ chmod a+x name.sh
$ ./name.sh Hans-Wolfgang Loidl
My first name is Hans-Wolfgang
My surname is Loidl
Total number of arguments is 2
```

Version 1: Line count example

The first example simply counts the number of lines in an input file. It does so by iterating over all lines of a file using a **while** loop, performing a **read** operation in the loop header. While there is a line to process, the loop body will be executed in this case simply increasing a counter by **((counter++))**. Additionally the current line is written into a file, whose name is specified by the variable `file`, by echoing the value of the variable `line` and redirecting the standard output of the variable to **$file**. the current line to file. The latter is not needed for the line count, of course, but demonstrates how to check for success of an operation: the special variable **$?** will contain the return code from the previous command (the redirected **echo**). By Unix convention, success is indicated by a return code of 0, all other values are error code with application specific meaning.

Another important issue to consider is that the integer variable, over which iteration is performed should always *count down* so that the analysis can find a bound. This might require some restructuring of the code, as in the following example, where an explicit counter `z` is introduced for this purpose. After the loop, the line count and the contents of the last line are printed, using **echo**. Of course, there is a Linux command that already implements line-count functionality: **wc** (for word-count) prints, when called with option **-l**, the number of lines in the file. We use this to check wether our line count is correct, demonstrating numeric operations on the way.

```
#!/bin/bash
# Simple line count example, using bash
#
# Bash tutorial: http://linuxconfig.org/Bash_scripting_Tutorial#8-2-read-
file-into-bash-array
# My scripting link:
http://www.macs.hw.ac.uk/~hwloidl/docs/index.html#scripting
#
# Usage: ./line_count.sh file
# -----------------------------------------------------------------------
--

# Link filedescriptor 10 with stdin
exec 10<&0
# stdin replaced with a file supplied as a first argument
exec < $1
# remember the name of the input file
in=$1

# init
file="current_line.txt"
let count=0

# this while loop iterates over all lines of the file
while read LINE
do
    # increase line counter
    ((count++))
    # write current line to a tmp file with name $file (not needed for
counting)
    echo $LINE > $file
    # this checks the return code of echo (not needed for writing; just for
demo)
    if [ $? -ne 0 ]
     then echo "Error in writing to file ${file}; check its permissions!"
    fi
done

echo "Number of lines: $count"
echo "The last line of the file is: `cat ${file}`"

# Note: You can achieve the same by just using the tool wc like this
echo "Expected number of lines: `wc -l $in`"

# restore stdin from filedescriptor 10
# and close filedescriptor 10
exec 0<&10 10<&-
```

As documented at the start of the script, it is called like this (you must have a file `text_file.txt` in your current directory):

```
$  ./line_count.sh text_file.txt
```

Sample text file

You can get a sizable sample text file by typing:

```
$ cp /home/msc/public/LinuxIntro/WaD.txt text_file.txt
```

Several versions of line counting across a set of files

This section develops several shell scripts, each counting the total number of lines across a set of files. These examples elaborate specific shell features. For counting the number of lines in one file we use **wc -l**. As a simple exercise you can replace this command with a call to the line counting script above.

Version 1: Explicit For loop

We use a for-loop to iterate over all files provided as arguments to the script. We can access all arguments through the variable `$*`. The sed command matches the line count, and replaces the entire line with just the line count, using the back reference to the first substring (\1). In the for-loop, the shell variable n is a counter for the number of files, and s is the total line count so far.

```
#!/bin/bash
# Counting the number of lines in a list of files
# for loop over arguments

if [ $# -lt 1 ]
then
  echo "Usage: $0 file ..."
  exit 1
fi

echo "$0 counts the lines of code"
l=0
n=0
s=0
for f in $*
do
        l=`wc -l $f | sed 's/^\([0-9]*\).*$/\1/'`
        echo "$f: $l"
        n=$[ $n + 1 ]
        s=$[ $s + $l ]
done

echo "$n files in total, with $s lines in total"
```

Version 2: Using a Shell Function

In this version we define a function **count_lines** that counts the number of lines in the file provided as argument. Inside the function the value of the argument is retrieved by accessing the variable `$1`.

```
#!/bin/bash
# Counting the number of lines in a list of files
# function version

count_lines () {
  local f=$1
  # this is the return value, i.e. non local
  l=`wc -l $f | sed 's/^\([0-9]*\).*$/\1/'`
}

if [ $# -lt 1 ]
then
  echo "Usage: $0 file ..."
  exit 1
fi

echo "$0 counts the lines of code"
l=0
n=0
s=0
while [ "$*" != ""  ]
do
        count_lines $1
        echo "$1: $l"
        n=$[ $n + 1 ]
        s=$[ $s + $l ]
        shift
done

echo "$n files in total, with $s lines in total"
```

Version 3: Using a return code in a function

This version tries to use the return value of the function to return the line count. However, this fails on files with more than 255 lines. The return value is intended to just provide a return code, e.g. 0 for success, 1 for failure, but not for returning proper values.

```
#!/bin/bash
# Counting the number of lines in a list of files
# function version using return code
# WRONG version: the return code is limited to 0-255
#  so this script will run, but print wrong values for
#  files with more than 255 lines

count_lines () {
  local f=$1
  local m
  m=`wc -l $f | sed 's/^\([0-9]*\).*$/\1/'`
  return $m
}

if [ $# -lt 1 ]
then
  echo "Usage: $0 file ..."
  exit 1
fi

echo "$0 counts the lines of code"
l=0
n=0
s=0
while [ "$*" != ""  ]
do
        count_lines $1
        l=$?
        echo "$1: $l"
        n=$[ $n + 1 ]
        s=$[ $s + $l ]
        shift
done

echo "$n files in total, with $s lines in total"
```

Version 4: Generating the file list in a shell function

```
#!/bin/bash
# Counting the number of lines in a list of files
# function version

# function storing list of all files in variable files
get_files () {
  files="`ls *.[ch]`"
}

# function counting the number of lines in a file
count_lines () {
  local f=$1  # 1st argument is filename
  l=`wc -l $f | sed 's/^\([0-9]*\).*$/\1/'` # number of lines
}

# the script should be called without arguments
if [ $# -ge 1 ]
then
  echo "Usage: $0 "
  exit 1
fi

# split by newline
IFS=$'\012'

echo "$0 counts the lines of code"
# don't forget to initialise!
l=0
```

```
n=0
s=0
# call a function to get a list of files
get_files
# iterate over this list
for f in $files
do
        # call a function to count the lines
        count_lines $f
        echo "$f: $l"
        # increase counter
        n=$[ $n + 1 ]
        # increase sum of all lines
        s=$[ $s + $l ]
done

echo "$n files in total, with $s lines in total"
```

Version 5: Using an array to store all line counts

The example below uses shell arrays to store all filenames (`file`) and its number of lines (`line`). The elements in an array are referred to using the usual [ ] notation, e.g. `file[1]` refers to the first element in the array `file`. Note, that bash only supports 1-dimensional arrays with integers as indizes.

See the section on arrays in the Advanced Bash-Scripting Guide:.

```

#!/bin/bash
# Counting the number of lines in a list of files
# function version

# function storing list of all files in variable files
get_files () {
  files="`ls *.[ch]`"
}

# function counting the number of lines in a file
count_lines () {
  f=$1  # 1st argument is filename
  l=`wc -l $f | sed 's/^\([0-9]*\).*$/\1/'` # number of lines
}

# the script should be called without arguments
if [ $# -ge 1 ]
then
  echo "Usage: $0 "
  exit 1
fi

# split by newline
IFS=$'\012'

echo "$0 counts the lines of code"
# don't forget to initialise!
l=0
n=0
```

```
s=0
# call a function to get a list of files
get_files
# iterate over this list
for f in $files
do
        # call a function to count the lines
        count_lines $f
        echo "$f: $l"loc
        # store filename in an array
        file[$n]=$f
        # store number of lines in an array
        lines[$n]=$l
        # increase counter
        n=$[ $n + 1 ]
        # increase sum of all lines
        s=$[ $s + $l ]
done

echo "$n files in total, with $s lines in total"
i=5
echo "The $i-th file was ${file[$i]} with ${lines[$i]} lines"
```

Version 6: Count only files we own

```
#!/bin/bash
# Counting the number of lines in a list of files
# for loop over arguments
# count only those files I am owner of

if [ $# -lt 1 ]
then
  echo "Usage: $0 file ..."
  exit 1
fi

echo "$0 counts the lines of code"
l=0
n=0
s=0
for f in $*
do
  if [ -O $f ] # checks whether file owner is running the script
  then
      l=`wc -l $f | sed 's/^\([0-9]*\).*$/\1/'`
      echo "$f: $l"
      n=$[ $n + 1 ]
      s=$[ $s + $l ]
  else
      continue
  fi
done

echo "$n files in total, with $s lines in total"
```

Version 7: Line count over several files

The final example supports options that can be passed from the command-line, e.g. by **./loc7.sh -d 1 loc7.sh**. The `getopts` shell function is used to iterate over all options (given in the following string) and assigning the current option to variable `name`. Typically it is used in a while loop, to set shell variables that will be used later. We use a pipe of **cat** and **awk** to print the header of this file, up to the first empty line, if the help option is chosen. The main part of the script is a for loop over all non-option command-line arguments. In each iteration, `$f` contains the name of the file to process. If the date options are used to narrow the scope of files to process, we use the **date** and an if-statement, to compare whether the modification time of the file is within the specified interval. Only in this case do we count the number of lines as before. After the loop, we print the total number of lines and the number of files that have been processed.

Example 11. Version 7: Line count over several files

```
#!/bin/bash
############################################################################
#
# Usage: loc7.sh [options] file ...
#
# Count the number of lines in a given list of files.
# Uses a for loop over all arguments.
#
# Options:
#  -h     ... help message
#  -d n ... consider only files modified within the last n days
#  -w n ... consider only files modified within the last n weeks
#
# Limitations:
#  . only one option should be given; a second one overrides
#
############################################################################

help=0
verb=0
weeks=0
# defaults
days=0
m=1
str="days"
getopts "hvd:w:" name
while [ "$name" != "?" ] ; do
  case $name in
   h) help=1;;
   v) verb=1;;
   d) days=$OPTARG
      m=$OPTARG
      str="days";;
   w) weeks=$OPTARG
      m=$OPTARG
      str="weeks";;
  esac
  getopts "hvd:w:" name
done

if [ $help -eq 1 ]
 then no_of_lines=`cat $0 | awk 'BEGIN { n = 0; } \
                                /^$/ { print n; \
                                       exit; } \
                                     { n++; }'`
      echo "`head -$no_of_lines $0`"
      exit
fi

shift $[ $OPTIND - 1 ]

if [ $# -lt 1 ]
then
  echo "Usage: $0 file ..."
  exit 1
fi

if [ $verb -eq 1 ]
  then echo "$0 counts the lines of code"
fi

l=0
n=0
s=0
for f in $*
do
  x=`stat -c "%y" $f`
  # modification date
  d=`date --date="$x" +%y%m%d`
  # date of $m days/weeks ago
  e=`date --date="$m $str ago" +%y%m%d`
  # now
  z=`date +%y%m%d`
  #echo "Stat: $x; Now: $z; File: $d; $m $str ago: $e"
  # checks whether file is more recent then req
  if [ $d -ge $e -a $d -le $z ] # ToDo: fix year wrap-arounds
  then
      # be verbose if we found a recent file
      if [ $verb -eq 1 ]
        then echo "$f: modified (mmdd) $d"
      fi
      # do the line count
      l=`wc -l $f | sed 's/^\([0-9]*\).*$/\1/'`
      echo "$f: $l"
      # increase the counters
      n=$[ $n + 1 ]
      s=$[ $s + $l ]
  else
      # not strictly necessary, because it's the end of the loop
      continue
  fi
done

echo "$n files in total, with $s lines in total"
```

Chapter 11. Write a shell script to compute gcd lcm & of two numbers. Use the basic function to find GCD & LCM of N numbers.

Find GCD:

```
# Argument check
ARGS=2
E_BADARGS=85

if [ $# -ne "$ARGS" ]
then
  echo "Usage: `basename $0` first-number second-number"
  exit $E_BADARGS
fi
# -------------------------------------------------------


gcd ()
{

  dividend=$1             #  Arbitrary assignment.
  divisor=$2              #! It doesn't matter which of the two is larger.
                          #  Why not?

  remainder=1             #  If an uninitialized variable is used inside
                          #+ test brackets, an error message results.

  until [ "$remainder" -eq 0 ]
  do    #  ^^^^^^^^^^  Must be previously initialized!
    let "remainder = $dividend % $divisor"
    dividend=$divisor     # Now repeat with 2 smallest numbers.
    divisor=$remainder
  done                    # Euclid's algorithm

}                         # Last $dividend is the gcd.


gcd $1 $2

echo; echo "GCD of $1 and $2 = $dividend"; echo


# Exercises :
# ---------
# 1) Check command-line arguments to make sure they are integers,
#+   and exit the script with an appropriate error message if not.
# 2) Rewrite the gcd () function to use local variables.

exit 0
```

Chapter 12: Write a shell script to find whether a given number is prime. Take a large number such as 15 digits or higher and use a proper algorithm.

Example A-15. Generating prime numbers using the modulo operator

```
#!/bin/bash
# primes.sh: Generate prime numbers, without using arrays.
# Script contributed by Stephane Chazelas.

#  This does *not* use the classic "Sieve of Eratosthenes" algorithm,
#+ but instead the more intuitive method of testing each candidate number
#+ for factors (divisors), using the "%" modulo operator.


LIMIT=1000                    # Primes, 2 ... 1000.

Primes()
{
 (( n = $1 + 1 ))             # Bump to next integer.
 shift                        # Next parameter in list.
#  echo "_n=$n i=$i_"

 if (( n == LIMIT ))
 then echo $*
 return
 fi

 for i; do                    # "i" set to "@", previous values of $n.
#   echo "-n=$n i=$i-"
   (( i * i > n )) && break   # Optimization.
   (( n % i )) && continue    # Sift out non-primes using modulo operator.
   Primes $n $@               # Recursion inside loop.
   return
   done

   Primes $n $@ $n            #  Recursion outside loop.
                              #  Successively accumulate
                             #+ positional parameters.
                              #  "$@" is the accumulating list of primes.
}

Primes 1

exit $?

# Pipe output of the script to 'fmt' for prettier printing.

#  Uncomment lines 16 and 24 to help figure out what is going on.

#  Compare the speed of this algorithm for generating primes
#+ with the Sieve of Eratosthenes (ex68.sh).
```

Alternate Method

Example 16-46. Generating prime numbers

```
#!/bin/bash
# primes2.sh

#  Generating prime numbers the quick-and-easy way,
#+ without resorting to fancy algorithms.

CEILING=10000   # 1 to 10000
PRIME=0
E_NOTPRIME=

is_prime ()
{
  local factors
  factors=( $(factor $1) )  # Load output of `factor` into array.

if [ -z "${factors[2]}" ]
#  Third element of "factors" array:
#+ ${factors[2]} is 2nd factor of argument.
#  If it is blank, then there is no 2nd factor,
#+ and the argument is therefore prime.
then
  return $PRIME             # 0
else
  return $E_NOTPRIME        # null
fi
}

echo
for n in $(seq $CEILING)
do
  if is_prime $n
  then
    printf %5d $n
  fi   #    ^  Five positions per number suffices.
done   #       For a higher $CEILING, adjust upward, as necessary.

echo

exit
```

Write a shell script program to implement Tower of Hanoi Problem.

```
E_NOPARAM=66  # No parameter passed to script.
E_BADPARAM=67 # Illegal number of disks passed to script.
Moves=        # Global variable holding number of moves.
              # Modification to original script.

dohanoi() {   # Recursive function.
```

```
    case $1 in
    0)
        ;;
    *)
        dohanoi "$(($1-1))" $2 $4 $3
        echo move $2 "-->" $3
        ((Moves++))          # Modification to original script.
        dohanoi "$(($1-1))" $4 $3 $2
        ;;
    esac
}

case $# in
    1) case $(($1>0)) in     # Must have at least one disk.
       1)  # Nested case statement.
           dohanoi $1 1 3 2
           echo "Total moves = $Moves"   # 2^n - 1, where n = # of disks.
           exit 0;
           ;;
       *)
           echo "$0: illegal value for number of disks";
           exit $E_BADPARAM;
           ;;
       esac
    ;;
    *)
       echo "usage: $0 N"
       echo "       Where \"N\" is the number of disks."
       exit $E_NOPARAM;
       ;;
esac
```