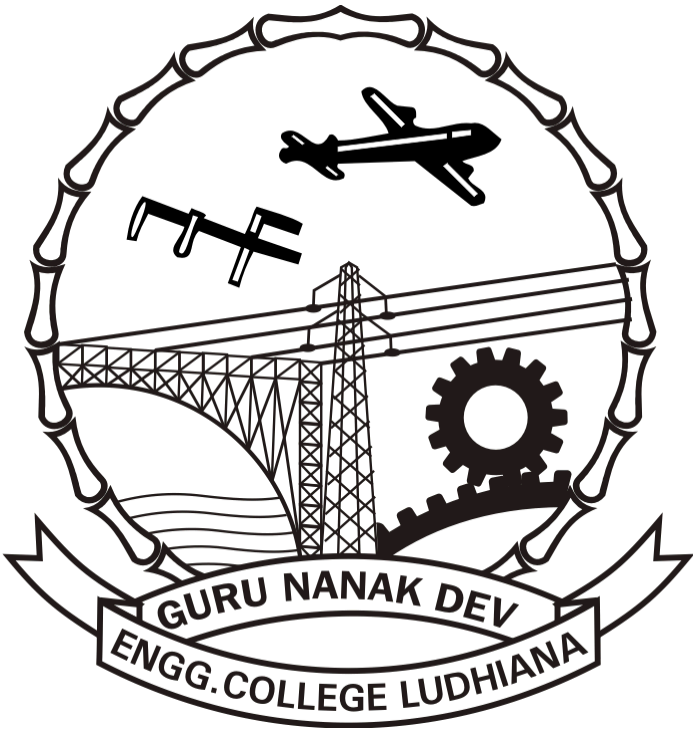
INSTRUCTION MANUAL

OPERATING SYSTEM LAB

(BTCS-406)



Prepared by

Er. Dipti kunra
Assistant Professor (CSE)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
GURU NANAK DEV ENGINEERING COLLEGE
LUDHIANA – 141006

DECLARATION

This Manual of Operating System Lab (BTCS-406) has been prepared by me as per syllabus of Operating System Lab (BTCS-406).

Signature

INDEX

Syllabus

BTCS 406 Operating System Lab

1. Installation Process of various operating systems

2. Virtualization, Installation of Virtual Machine Software and installation of Operating System on Virtual Machine

3. Commands for files & directories: cd, ls, cp, md, rm, mkdir, rmdir. Creating and viewing files using cat. File comparisons. Disk related commands: checking disk free spaces. Processes in linux, connecting processes with pipes, background processing, managing multiple processes, Manual help. Background process: changing process priority, scheduling of processes at command, batch commands, kill, ps, who, sleep. Printing commands, grep, fgrep, find, sort, cal, banner, touch, file. File related commands ws, sat, cut, grep.

4. Shell Programming: Basic of shell programming, various types of shell, Shell Programming in bash, conditional & looping statement, case statements, parameter passing and arguments, shell variables, shell keywords, creating shell programs for automate system tasks, report printing.

Practical 1

Title: Introduction of Unix and history of Unix

The University of California at Berkeley acquired Version 4 of UNIX in 1974. In 1975,
during a sabbatical leave from Bell Labs, Ken Thompson went to Berkeley and helped
install UNIX Version 6 on a PDP-11/70. ``The same year, two graduate students also
arrived at Berkeley: Bill Joy and Chuck Haley" : they were to play an important role in
the BSD developments.
Joy wrote the **vi** editor, and the **C shell**. Bill Joy is also one of the co-founders of Sun
Microsystems. In 1978, 2BSD was released. In 1979, a combination of improved 2BSD
and UNIX Version 7 was released as 3BSD.
DARPA (the Defence Advanced Research Project Agency) funded the development of
4BSD and 4.1BSD. The Fast File System was included in 4.2BSD which was released in
1983.
Finally, 4.3BSD was released in 1987.
Berkeley's development of its BSD UNIX ended with the release of its last major version,
4.4BSD, in 1992. Then, late in 1994, it released 4.4BSD*lite*.

Reasons for the popularity of Unix

- Portability
- Portable Applications Software
- Multiuser Operation
- Simple, Powerful User Interface
- Hierarchical File Structure
- Consistent File Structure
- Simple, Consistent Interface to Peripherals
- Large Number of Utilities
- Hides architecture of machine from user

Practical 2

Title: Unix Layers and Unix file system

Fig 1. Unix Layers

The File System and Security

 As with most modern operating systems, Unix maintains files in a directory structure call the Unix file system.

 The main directory is called the root directory and is indicated by a single forward
slash character: /

 A series of sub-directories can then appear below the root directory.

 Some of the main sub-directories found in more versions of Unix are seen below
in Fig 2.

5

Fig 2. Unix File System

etc - Administrative programs and configuration files
dev - Devices drivers (pointers) such as disk drives, keyboard, mouse, etc.
mnt - Mounting point for additional devices such as cdrom or remote systems
var - Temporary administrative space for logging and other system information
home - Home directories for users (in this example, holowcza, norman and smith are users)
usr - Standard programs and code libraries
/usr/sbin - Administrative programs
/usr/bin - Standard executable programs
/usr/lib - Code libraries
/usr/local/bin - Additional programs

 Files can be executable programs and scripts, text files (letters, lists, etc.), binary
files such as images and links to other files and directories.

 Directories are nested into a tree structure starting with the "root" directory /

 Each directory name is separated by the / (foreslash) character:
/home/holowcza/public_html

 Each user on the system has an assigned username

 This username is associated with a *home directory* where all of the user's files are
stored.

 To see the name of your home directory, use the pwd command right after you log
in.

 To see what files you have in your directory, use the ls command.

 To create a file, use a text editor such as pico or emacs

6

 To create a subdirectory, use the mkdir command.

 To change to another directory, use the cd command.

File Permissions

 Each file in UNIX has 3 sets of permissions (called the file *mode*):

1. Owner: Read, write and execute permissions for the owner of the file.
2. Group: Read, write and execute permissions for other members of the same group.
3. Other (or World): Read, write and execute permissions for everyone else in the world.

 The 3 permissions are:

1. Read: The owner, group or world can read this file - Denoted as r
2. Write: The owner, group or world can write to this file (modify it) - Denoted as w
3. Execute: The owner, group or world can execute this file as a program - Denoted as x

 A set of permissions looks like the following (try using the UNIX command ls -l to see the permissions):

 -rw-r--r-- my_report.txt

 -rw-rw---- group_project.txt

 -r-x------ example_program

 The file my_report.txt can be read and written by the owner and read by anyone else in the group or in the world.

 The file group_report.txt can be read and written by the owner and by the group.
Anyone outside of the group can not do anything with this file.

 The file example_program can be read and executed by the owner only. Compiled programs (e.g. from "C" source) and shell scripts are marked with the x permission.

Directory Permissions

 Work the same way as file permissions: 3 sets with 3 types of permission. The meaning is slightly different:

1. Read permissions on a directory affect all files within the directory.
2. Write permissions on a directory mean files can be created in the directory.
3. Execute permission on a directory means that the contents of the directory can be searched using the ls command.

 -rw-r--r-- my_report.txt

 -rw-rw---- group_project.txt

 -r-x------ example_program

 drwxr-x--- my_subdir

 For the subdirectory my_subdir :

1. The owner can read any file in my_subdir provided the individual file permission is set.
2. The owner can write (create) files into the directory.

3. The owner can list the files in the directory (the execute permission).
4. Others in the group can read files and can list the files in my_subdir but can't create any new files.

 Most user's home directories are created with: drwxr-xr-x permissions on it. What does this mean ?

d means it is a directory
rwx means the owner can read, write and search the directory
r-x means others in the group can read files and search the directory
r-x means others in the world can read files and search the directory

Users in Unix

 In a multi-user operating system sych as Unix, we need a way to keep each user's files and processes (executing programs) separated.
 Thus each user has an associated Unix username
 This Unix username is also given a unique identifier called the user id (or uid for short)
 Files and directories on disk, and processes (executing programs) are stamped with their owner's uid.
 The uid is also used for security purposes
 To see what your uid is, try the id program:
 $ id holowcza
 uid=10595(holowcza) gid=11301(cis)
 There is a special user on each Unix system called the root user.
 The root user acts as the system administrator and has the authority (and privileges) to create users, directories, modify any file, and so on.

Groups in Unix

 In Unix, a *group* is a convienient way to associate a collection of users.
 Each user belongs to one or more *groups*.
 A group is identified by a Group id (gid for short).
 The gid of the owner (user) is also stamped on each file and process
 To see what your gid is, try the id program:
 $ id holowcza
 uid=10595(holowcza) gid=11301(cis)

Users and Groups

 More than one "users" (think "person") can make use of the system's resources at the same time
 Resources include: Disk, memory, processing time on the CPU, etc.

Practical 3

Title: Internal and External Commands in Unix

Some commands that you type are *internal*, *built into* the shell. For example, the *cd* command is built-in. That is, the shell interprets that command and changes your current directory for you. The *ls* command, on the other hand, is an *external* program stored in the file */bin/ls*.

The shell doesn't start a separate process to run internal commands. External commands require the shell to *fork* and *exec* a new subprocess, this takes some time, especially on a busy system.

When you type the name of a command, the shell first checks to see if it is a built-in command and, if so, executes it. If the command name is an absolute pathname beginning with /, like */bin/ls*, there is no problem: the command is likewise executed. If the command is neither built-in, nor specified with an absolute pathname, the shell looks in its search path for an executable program or script with the given name.

By tradition, UNIX system programs are kept in directories called */bin* and */usr/bin*, with additional programs usually used only by system administrators in */etc* and */usr/etc*.

Many versions of UNIX also have programs stored in */usr/ucb* (named after the University of California at Berkeley, where many UNIX programs were written). There may be other directories containing programs. For example, the programs that make up the X Window System are stored in */usr/bin/X11*. Users or sites often also have their own directories where custom commands and scripts are kept, such as */usr/local/bin*.

Practical 4

Title: Introduction to vi editor.

The default editor that comes with the UNIX operating system is called vi (visual editor). The UNIX vi editor is a full screen editor and has two modes of operation:
Command mode commands which cause action to be taken on the file, and
Insert mode in which entered text is inserted into the file.
In the command mode, every character typed is a command that does something to the text file being edited; a character typed in the command mode may even cause the vi editor to enter the insert mode. In the insert mode, every character typed is added to the text in the file; pressing the <Esc> (Escape) key turns off the Insert mode.
While there are a number of vi commands, just a handful of these is usually sufficient for beginning vi users. To assist such users, this Web page contains a sampling of basic vi commands. The most basic and useful commands are marked with an asterisk (* or star) in the tables below. With practice, these commands should become automatic.
NOTE: Both UNIX and vi are case-sensitive. Be sure not to use a capital letter in place of a lowercase letter; the results will not be what you expect.

To Get Into and Out Of vi
To Start vi
To use vi on a file, type in vi filename. If the file named filename exists, then the first page (or screen) of the file will be displayed; if the file does not exist, then an empty file and screen are created into which you may enter text.

vi filename edit filenamestarting at line 1
vi -r filenamerecover filenamethat was being edited when system crashed
To Exit vi
Usually the new or modified file is saved when you leave vi. However, it is also possible to quit vi without saving the file.
Note: The cursor moves to bottom of screen whenever a colon (:) is typed. This type of command is completed by hitting the <Return> (or <Enter>) key.

:x<Return> quit vi, writing out modified file to file named in original invocation
:wq<Return> quit vi, writing out modified file to file named in original invocation
:q<Return> quit (or exit) vi
:q!<Return> quit vieven though latest changes have not been saved for this vicall

Moving the Cursor

Unlike many of the PC and MacIntosh editors, the mouse does not move the cursor within the vi editor screen (or window). You must use the the key commands listed below. On some UNIX platforms, the arrow keys may be used as well; however, since vi was designed with the Qwerty keyboard (containing no arrow keys) in mind, the arrow keys sometimes produce strange effects in vi and should be avoided.

If you go back and forth between a PC environment and a UNIX environment, you may find that this dissimilarity in methods for cursor movement is the most frustrating difference between the two.

In the table below, the symbol ^ before a letter means that the <Ctrl> key should be held down while the letter key is pressed.

jor<Return>
[ordown-arrow] move cursor down one line
k[orup-arrow] move cursor up one line
hor<Backspace>
[orleft-arrow] move cursor left one character
lor<Space>
[orright-arrow] move cursor right one character
0(zero) move cursor to start of current line (the one with the cursor)
$ move cursor to end of current line
w move cursor to beginning of next word
b move cursor back to beginning of preceding word
:0<Return>or1G move cursor to first line in file
:n<Return>ornG move cursor to line n
:$<Return>orG move cursor to last line in file

Screen Manipulation

The following commands allow the vi editor screen (or window) to move up or down several lines and to be refreshed.

^f move forward one screen
^b move backward one screen
^d move down (forward) one half screen
^u move up (back) one half screen
^l redraws the screen
^r redraws the screen, removing deleted lines

Adding, Changing, and Deleting Text

Unlike PC editors, you cannot replace or delete text by highlighting it with the mouse. Instead use the commands in the following tables.

Perhaps the most important command is the one that allows you to back up and undo your last action. Unfortunately, this command acts like a toggle, undoing and redoing your most recent action. You cannot go back more than one step.

u UNDO WHATEVER YOU JUST DID; a simple toggle

The main purpose of an editor is to create, add, or modify text for a file.

Inserting or Adding Text

The following commands allow you to insert and add text. Each of these commands puts the vi editor into insert mode; thus, the <Esc> key must be pressed to terminate the entry of text and to put the vi editor back into command mode.

i insert text before cursor, until <Esc>hit
I insert text at beginning of current line, until <Esc>hit
a append text after cursor, until <Esc>hit
A append text to end of current line, until <Esc>hit
o open and put text in a new line below current line, until <Esc>hit
O open and put text in a new line above current line, until <Esc>hit

Changing Text

The following commands allow you to modify text.

r replace single character under cursor (no <Esc>needed)
R replace characters, starting with current cursor position, until <Esc>hit

cw change the current word with new text,
starting with the character under cursor, until <Esc>hit
cNw change Nwords beginning with character under cursor, until <Esc>hit;
e.g., c5wchanges 5 words
C change (replace) the characters in the current line, until <Esc>hit
cc change (replace) the entire current line, stopping when <Esc>is hit
NccorcNc change (replace) the next N lines, starting with the current line,
stopping when <Esc>is hit

Deleting Text

The following commands allow you to delete text.

x delete single character under cursor
Nx delete N characters, starting with character under cursor
dw delete the single word beginning with character under cursor
dNw delete Nwords beginning with character under cursor;
e.g., d5wdeletes 5 words
D delete the remainder of the line, starting with current cursor position
dd delete entire current line
NddordNd delete Nlines, beginning with the current line;
e.g., 5dddeletes 5 lines

Cutting and Pasting Text

The following commands allow you to copy and paste text.

yy copy (yank, cut) the current line into the buffer

NyyoryNy copy (yank, cut) the next N lines, including the current line, into the buffer
p put (paste) the line(s) in the buffer into the text after the current line

Other Commands

Searching Text

A common occurrence in text editing is to replace one word or phase by another. To locate instances of particular sets of characters (or strings), use the following commands.

/string search forward for occurrence of stringin text
?stringsearch backward for occurrence of stringin text
n move to next occurrence of search string
N move to next occurrence of search string in opposite direction

Determining Line Numbers

Being able to determine the line number of the current line or the total number of lines in the file being edited is sometimes useful.

:.= returns line number of current line at bottom of screen
:= returns the total number of lines at bottom of screen
^g provides the current line number, along with the total number of lines, in the file at the bottom of the screen

Saving and Reading Files

These commands permit you to input and output files other than the named file with which you are currently working.

:r filename<Return> read file named filenameand insert after current line (the line with cursor)
:w<Return> write current contents to file named in original vicall
:w newfile<Return> write current contents to a new file named newfile
:12,35w smallfile<Return> write the contents of the lines numbered 12 through 35 to a new file named smallfile
:w! prevfile<Return> write current contents over a pre-existing file named prevfile

Practical 5

Title: Practice with Basic File Permissions Commands.

Chmod : Change Mode

Setting File/Directory Permissions

The UNIX chmod command is used to set permissions (short for Change Mode).
Syntax:
chmod permissions filenames
Each type of permission has a value associated with it: Read = 4, Write = 2 and Execute = 1. These are added together.
The following combinations and resulting sets of values are possible:
rwx 7
rw- 6
r-x 5
r-- 4
-wx 3
-w- 2
--x 1
Since there are three sets of permissions (Owner, Group and World/Other), three numbers are given as parameters to the chmod command.
Controlling access to your files. You can control exactly who has access to the files
stored in your account. These are the things you need to know:
What are permissions Every Unix file has a set of **permissions** associated with it that
specify who is allowed to use that file and in what way.
There are three **kinds of permissions**:
1. A **read** permission means someone can see what is in the file.
2. A **write** permission means someone can change what is in the file, or even remove
it.
3. An **execute** permission means someone can run (execute) a file, if that file contains a program.
Permissions are granted to three **kinds of users**:
 The **owner** permissions specify what the owner of the file can do.
 For a discussion of **group** permissions.
 The **world** permissions specify what anyone can do.

Finding a files permissions : Use the ls -l command to show the permissions of a file.The
permissions appear at the beginning of each line of the output from ls -l command. Here
is an example:
-rwxrw-r-- 1 john 1659 97 Mar 10 23:46 schedule

The first character is ``-'' for ordinary files. The next three characters give the owner's
permissions; the next three show the group permissions; and the last three show the world
permissions. In this example, the owner has permissions ``rwx'' (read, write, and execute); the group permissions are ``rw-'' (read and write); and the world permissions
are ``r--'' (read only).

- **r** means read permission.
- **w** means write permission.
- **x** means execute permission.
- **-** means no permission.

Changing files permissions : To change the permissions on one or more files, use the
command
% **chmod** *mode file ...*
where *mode* tells what permissions to add, subtract, or change, and these changes are
applied to each *file*.
You can use this command for: **Setting File Permission**
To change a file's permissions to a specific value, use this command:
% **chmod *who=value*** *file...*
where *who* specifies the type of user that these permissions apply to:

- **u** to change the owner's (``user's'') permissions;
- **g** to change group permissions;
- **o** to change world (``other'') permissions; or
- **a** to change all permissions: owner, group and world.

and *value* gives the new permission values:

- **r** for read permission;
- **w** for write permission; and
- **x** for execute permission.

For example: To set up file foo so only the user can read and write it:
% **chmod u=rw foo**

- To set up file bar so anyone can read it, but no one can write or execute it:

% **chmod a=r bar**
Adding File Permissions: To add certain permissions to a file, use this command:
% **chmod *who+value*** *file...*
where *who* specifies the type of user that these permissions apply to:

- **u** to change the owner's (``user's'') permissions;
- **g** to change group permissions;
- **o** to change world (``other'') permissions; or
- **a** to change all permissions: owner, group and world.

and *value* gives the new permission values:

- **r** for read permission;

 w for write permission; and
 x for execute permission.
For example:
 To add world read permission for file foo:
% **chmod o+r foo**
 To add read and execute permissions for the owner and the group for file bar:
% **chmod ug+rx bar**
Removing Files permissions : To remove certain permissions from a file, use this
command:
% **chmod *who-value*** *file...*
where *who* specifies from which type of user these permissions are to be removed:
 u to change the owner's (``user's'') permissions;
 g to change group permissions;
 o to change world (``other'') permissions; or
 a to change all permissions: owner, group and world.
and *value* says which permissions to remove:
 r for read permission;
 w for write permission; and
 x for execute permission.

For example:
 To remove world write permission for file foo:
% **chmod o-w foo**
 To remove group and world read and write permissions from file bar:
% **chmod go-rw bar**
Limiting file access to particular people: If you want to allow a specific list of people
access to some files, Unix allows you to set up a **group** of users.
If you need to set up and use a group, please consult with a system programmer.
If what you **really** need is to share files among several users, you may find that a **source**
code control system is what you need.
Compare (cmp): The cmp utility compares two files of any type and writes the results to
the standard output. By default, cmp is silent if the files are the same; if they differ, the
byte and line number at which the first differ- ence occurred is reported. Bytes and lines
are numbered beginning with one.
Wc: Word ,line , and character count: Print byte, word, and new line counts for each
FILE, and a total line if more than one FILE is specified. With no FILE, or when FILE is
-, read

standard input.
-c, **--bytes**
print the byte counts
-m, **--chars**
print the character counts
-l, **--lines**
print the new line counts
-L, **--max-line-length**
print the length of the longest line
-w, **--words**
print the word counts
--help display this help and exit

Practical 6

Title: Study of the following commands –

Date, cal, Clear, who, who am I, banner, man script

Date : This command is used to know the current system date

Cal: This command is used to print the calender on the screen

Clear : This command is used to clear the termial

Who : This gives the information about the users for a particular system

Who am I : This gives the information about the current user logged in

Banner : This is used to print the banner on the screen

Man : Man stands for the manual. It given the information about all the commands

Script: This command is used to record the session. By default the file used in typescript

Practical 7

Title: Study of the following commands for file management –

Ls, cd, pwd, mkdir, rmdir, cp, mv, rm, find, cat

List (ls) : Finding out what files you have.
Once you log in to your account, you can see what files you have by using the **ls** command:
% **ls**
This command will give you a list of all the names of your files. If you just got your
account, you won't see any names until you actually create some files.
Here's an example. Suppose you type an **ls** command and it produces this output:
% **ls**
groceries schedule todo
This is telling you that you have three files. In alphabetical order, their names are groceries, schedule, and todo.
Change Directory (cd) and pwd : Changing your Current working Directory.
At any given time, you are located in some **current working directory** within the Unix
file tree. When you log in, your current directory is set to your **home directory**.
You can change your current working directory any time by using the cd command (this
stands for "change directory"). For example, if the current directory has a subdirectory
named details, you can change to that directory using the command:
% cd details
You can always find out the absolute pathname of the current working directory by
typing the command:
% pwd
Assuming you are using csh as your shell, you can always return to your home directory
by typing:
% cd ~
Make Directory (mkdir): You can create a directory by using the mkdir command. For
example, to create a directory named mayhem within the current working directory:

% mkdir mayhem
You may create a subdirectory within any directory where you have write permission.
For example, to create a directory called /u/trixie/bazz/fazz, assuming that directory
/u/trixie/bazz already exists:

% mkdir /u/trixie/bazz/fazz

Remove Directory (rmdir): It is used to remove a directory

Copy (cp) : Making a Copy of a File

If you want to make an exact copy of the contents of a file, figure out what you want to
name the copy, and then type the command

% **cp *oldname newname***

where *oldname* is the name of the existing file you want to copy. A new file will be
created with the name *newname* and the same contents as the original.

Move (mv) : Changing the name of a file.

The Unix **mv** command can be used to change the name of a file:

% **mv *oldname newname***

This will not affect the contents of the file. It will simply change its name from *oldname*
to *newname*

Getting Rid of unwanted files.

Once you have no more need for a disk file, you can get rid of it permanently by using
the Unix **rm** command.

**Warning! When you remove a file, it is gone forever. There is no easy way to get it
back. Please be careful that you do not remove a file that you really need.**

Remove (rm): To delete a file from your account, enter this command:

% **rm *name***

where *name* identifies the file you want to remove. You can remove more than one file at
a time. For example, this command

% **rm oldbills oldnotes badjokes**

would remove the three named files.

Find : Find files or folders based on name, date, size, ownership or other parameters

Cat : Used to create a file or view the contents of the existing file

Practical 8

Title: Study of the following commands ps, grep, tar, find

Finding your processes: To find the job numbers of all the background jobs you are
running, type:
% jobs
The job numbers will be shown in square brackets ([]), followed by the notation ``Stopped" (for suspended jobs) or ``Running" (for jobs that are still executing).
To find out all the process ID numbers of processes you are running, type:
% ps -gx
Here is an example of output from this command:
PID TT STAT TIME COMMAND
12030 p5 S 0:01 -csh (csh)
12068 p5 T 1:46 emacs foo
12788 p5 R 0:00 ps -gx
The first column is the process ID. The second column tells what terminal is controlling it
(in this case, ttyp5). Next comes the current status (S for sleeping, T for stopped, R for
running). The next column tells how much time it has spent.
Grep: Powerful command to search for a pattern on the designated files
grep -i root /etc/passwd
grep -i r[o*]t /etc/passwd
grep -i [1-9] /etc/group
find . -type f -exec grep -i ldap {} /dev/null \;
`

Find: command for searching a file
find . -name my_file /home/mine
Tar: create/extract a backup file with multiple files
1. To create all the files in the current directory to a floopy:
cd /home/mine; tar cf /dev/fd0 .

2. To compress all the files in the current directory to a file name /tmp/mine.tgz:
cd /home/mine; tar cfz /tmp/mine.tgz .
3. To extract all the files from the floopy into the currently directory:
cd /home/mine; tar xvf /dev/fd0
4. To extract all the files from a compressed tar file into the currently directory:
cd /home/mine; tar xvfz /tmp/mine.tgz
5. To extract all the files from a bzip-compressed tar file into the currently directory:
cd /home/mine; tar xvfj /tmp/mine.tar.bz2

Note: there are many options availalbe under *tar*, enter *tar --help* will show you the usage of *tar* command. In the above examples:
c: create
f: file - a file name is required. The file name can be a device (*/dev/fd0* which is floppy drive.
v: verbose - shows what files are processed **x**: extract
z: compress file
j: bzip-compress file
. stands for all the files in the current directory

Practical 9

Title: Introduction of different Shells in UNIX operating systems.

Shell : After you `log in', you are prompted for Unix commands like ls and logout. The
part of Unix that executes such commands is called a *shell*. A *shell* is the OS program
that greets you with prompt; it allows you to execute programs (editors, spelling checkers, etc) and generally makes your life easier. We will look at shell programming in
Unix, as it is more powerful (i.e., useful) than shell programming in Windows 98/NT.
Also, it is possible to get Unix shell emulators for Unix to run in Windows98/NT environments.
We have several different shells available here:
sh
The first shell, historically, was sh, also known as the Bourne shell. It is good for writing shell scripts, but not so popular for interactive use.
csh
Also known as C-Shell, csh features a syntax somewhat like the C language. It allows (among other things) adding your own commands (aliasing), history substitution (re-execution of previously typed commands), and filename completion.
tcsh
This shell allows you to edit your command line while you're typing it, using emacs-like commands. It has a number of other nifty features, but is otherwise compatible with csh.
bash
Compatible with sh for programming purposes, it has many of the good features of csh and tcsh: file name completion, job control, history substitution, emacs command-line editing, and many more.
To change your shell, use the chsh command.
csh - c shell (standard)
sh - bourne shell
ksh - korn shell (subset? of bourne)
tcsh - advanced c shell for emacs lovers
The only Unix shell programming language documented in any reasonable fashion is that
for sh. Normally, if you execute a shell program from C shell, Unix assumes the program
is a bourne shell. (This information is actually specified in your login profile, which you
usually need a system administrator to change). If necessary, 1st line of your shell
program should be:
#!/bin/sh

Shell programs are text programs that are interpreted by a shell; there is no compiling.
You must indicate to Unix that your file is executable. This is accomplished by setting the
execution bit of the file:
chmod +x <your-shell-file>
Without setting the x bit, your file cannot be executed.
Question: How does one debug shell programs? Ans: make the following the first line
(after #!/bin/sh):
set –xv

Practical 10

Title: Write a program for displaying Hello world using Shell scripts.

A First Script

For our first shell script, we'll just write a script which says "Hello World". We will then
try to get more out of a Hello World program than any other tutorial you've ever read :-)
Create a file (first.sh) as follows:
first.sh

```
#!/bin/sh
# This is a comment!
echo Hello World # This is a comment, too!
```

The first line tells Unix that the file is to be executed by /bin/sh. This is the standard
location of the Bourne shell on just about every Unix system. If you're using GNU/Linux,
/bin/sh is normally a symbolic link to bash.
The second line begins with a special symbol: #. This marks the line as a comment, and it
is ignored completely by the shell. The only exception is when the *very first* line of the
file starts with #! - as ours does. This is a special directive which Unix treats specially. It
means that even if you are using csh, ksh, or anything else as your interactive shell, that
what follows should be interpreted by the Bourne shell.
Similarly, a Perl script may start with the line #!/usr/bin/perl to tell your interactive shell
that the program which follows should be executed by perl. For Bourne shell programming, we shall stick to #!/bin/sh.
The third line runs a command: echo, with two parameters, or arguments - the first is
"Hello"; the second is "World". Note that echo will automatically put a single space
between its parameters. The # symbol still marks a comment; the # and anything following it is ignored by the shell.
Now run chmod 755 first.sh to make the text file executable, and run ./first.sh.
Your screen should then look like this:

```
$ chmod 755 first.sh
$ ./first.sh
Hello World
$
```

You will probably have expected that! You could even just run:

```
$ echo Hello World
Hello World
$
```

Now let's make a few changes. First, note that echo puts ONE space between its parameters. Put a few spaces between "Hello" and "World". What do you expect the output to be? What about putting a TAB character between them? As always with shell programming, try it and see. The output is exactly the same! We are calling the echo program with two arguments; it doesn't care any more than cp does about the gaps in between them. Now modify the code again:

```
#!/bin/sh
# This is a comment!
echo "Hello World" # This is a comment, too!
```

This time it works. You probably expected that, too, if you have experience of other programming languages. But the key to understanding what is going on with more complex command and shell script, is to understand and be able to explain: WHY?

echo has now been called with just ONE argument - the string "Hello World". It prints this out exactly. The point to understand here is that the shell parses the arguments BEFORE passing them on to the program being called. In this case, it strips the quotes but passes the string as one argument. As a final example, type in the following script.

first2.sh

```
#!/bin/sh
# This is a comment!
echo "Hello World" # This is a comment, too!
echo "Hello World"
echo "Hello * World"
echo Hello * World
echo Hello World
echo "Hello" World
echo Hello " " World
echo "Hello \"*\" World"
echo `hello` world
echo 'hello' world
```

Just about every programming language in existance has the concept of *variables* - a symbolic name for a chunk of memory to which we can assign values, read and

manipulate its contents. The bourne shell is no exception, and this section introduces
idea. Let's look back at our first Hello World example. This could be done using variables

(though it's such a simple example that it doesn't really warrant it!)
Enter the following code into var.sh:
var.sh
```
#!/bin/sh
MY_MESSAGE="Hello World"
echo $MY_MESSAGE
```
This assigns the string "Hello World" to the variable MY_MESSAGE then echoes out the
value of the variable.
Note that we need the quotes around the string Hello World. Whereas we could get away
with echo Hello World because echo will take any number of parameters, a variable can
only hold one value, so a string with spaces must be quoted to that the shell knows to
treat it all as one. Otherwise, the shell will try to execute the command World after
assigning MY_MESSAGE=Hello

Practical 11

Title: Shell program to add two numbers

```
#!/bin/Bash
echo "Enter the two numbers to be added:"
read n1
read n2
answer=$(($n1+$n2))
echo $answer
```

Practical 12

Title: Shell program to find largest of three numbers

```
#!/bin/bash
a=$1
b=$2
c=$3

if [ $# -lt 3 ]
then
        echo "$0 n1 n2 n3"
        exit 1
fi

if [ $a -gt $b -a $a -gt $c ]
then
        echo "$a is largest integer"
elif [ $b -gt $a -a $b -gt $c ]
then
        echo "$b is largest integer"
elif [ $c -gt $a -a $c -gt $b ];
then
        echo "$c is largest integer"
else
        echo "Sorry cannot guess number"
fi
```

Practical 13

Title: Shell program to exchange the values of two variables

```
echo Enter valuefor a:
read a
echo Enter valuefor b:
read b
clear
echo Values of variables before swaping
echo A=$a
echo B=$b
echo Values of variables after swaping
a=`expr $a + $b`
b=`expr $a - $b`
a=`expr $a - $b`
echo A=$a
echo B=$b
```

Practical 14

Title: Use of Different Wildcards & Escape Characters

Wildcards are really nothing new if you have used Unix at all before.
It is not necessarily obvious how they are useful in shell scripts though. This section is
really just to get the old grey cells thinking how things look when you're in a shell script -
predicting what the effect of using different syntaxes are. Think first how you would
copy all the files from /tmp/a into /tmp/b. All the .txt files? All the .html files?
Hopefully you will have come up with:

```
$ cp /tmp/a/* /tmp/b/
$ cp /tmp/a/*.txt /tmp/b/
$ cp /tmp/a/*.html /tmp/b/
```

Now how would you list the files in /tmp/a/ without using ls /tmp/a/?
How about echo /tmp/a/*? What are the two key differences between this and the ls
output? How can this be useful? Or a hinderance?
How could you rename all .txt files to .bak? Note that

```
$ mv *.txt *.bak
```

will not have the desired effect; think about how this gets expanded by the shell before it
is passed to mv. Try this using echo instead of mv if this helps.

Escape Characters

Certain characters are significant to the shell; we have seen, for example, that the use of
double quotes (") characters affect how spaces and TAB characters are treated, for
example:

```
$ echo Hello World
Hello World
$ echo "Hello World"
Hello World
```

So how do we display: Hello "World" ?

```
$ echo "Hello \"World\""
```

The first and last " characters wrap the whole lot into one parameter passed to echo so
that the spacing between the two words is kept as is. But the code:

```
$ echo "Hello "World""
```

would be interpreted as three parameters:

- "Hello "
- World
- ""

So the output would be

```
Hello World
```

Note that we lose the quotes entirely. This is because the first and second quotes mark off
the Hello and following spaces; the second argument is an unquoted "World" and the
third argument is the empty string; "".

Most characters (*, ', etc) are not interpreted (ie, they are taken literally) by means of
placing them in double quotes ("). They are taken as is and passed on the the command
being called. An example using the asterisk (*) goes:
$ echo *
case.shtml escape.shtml first.shtml functions.shtml hints.shtml index.shtml ip-primer.txt
raid1+0.txt
$ echo *txt
ip-primer.txt raid1+0.txt
$ echo "*"
*$
echo "*txt"
*txt
In the first example, * is expanded to mean all files in the current directory.
In the second example, *txt means all files ending in txt.
In the third, we put the * in double quotes, and it is interpreted literally.
In the fourth example, the same applies, but we have appended txt to the string.
However, ", $, `, and \ are still interpreted by the shell, even when they're in double
quotes.
The backslash (\) character is used to mark these special characters so that they are not
interpreted by the shell, but passed on to the command being run (for example, echo).
So to output the string: (Assuming that the value of $MY_VAR is 5):
A quote is ", backslash is \, backtick is `, a few spaces are and dollar is $. $MY_VAR is
5.
we would have to write:
$ echo "A quote is \", backslash is \\, backtick is \`, a few spaces are and dollar is \$.
\$MY_VAR is ${MY_VAR}."
A quote is ", backslash is \, backtick is `, a few spaces are and dollar is $. $MY_VAR is
5. We have seen why the " is special for preserving spacing. Dollar is special because it
marks a variable, so $MY_VAR is replaced by the shell with the contents of the variable

MY_VAR. Backslash is special because it is itself used to mark other characters off; we
need the following options for a complete shell:
$ echo "This is \\ a backslash"
This is \ a backslash

Practical 15

Title: Use of if statement, for loop & While loop in Shell Scripting.

If Statement

The if statement has the following format. (Note again that if, then, and fi begin different
lines):

if <this command is successful>
then ...
fi
or
if
then ...
else ...
fi

The "command" is really a process; depending on its exit value will the interpretation
"true" or "false". exit(0) is "true"; exit(-1) or anything nonzero is "false". (Exactly
opposite of what you'd think if you were a C programmer!)

Here are two files, test1 and testr

if testr
then echo "successful"
else echo "failure"
fiw
= "stupid" > myerror

run test1. Now go and change testr to w = = = "stupid". (You'll get: Usage: w [-hlsuw] [
user]. type `man w' to find out what w is...). Change testr to w="stupid" and rerun test1.

Now set w==="stupid" (it works, but why???) Put a echo $w inside testr and see what
happens.

Exit

To exit from a shell program, use:

- exit
- exit n - n is returned as the exit value of the program.

Most languages have the concept of loops: If we want to repeat a task twenty times, we
don't want to have to type in the code twenty times, with maybe a slight change each
time.

As a result, we have FOR and WHILE loops in the Bourne shell. This is somewhat fewer

features than other languages, but nobody claimed that shell programming has the power
of C.

For Loops

for loops iterate through a set of values until the list is exhausted:

for.sh

```
#!/bin/sh
for i in 1 2 3 4 5
do echo "Looping ... number $i"
done
```

Try this code and see what it does. Note that the values can be anything at all:

for2.sh

```
#!/bin/sh
for i in hello 1 * 2 goodbye
do echo "Looping ... i is set to $i"
done
```

is well worth trying. Make sure that you understand what is happening here. Try it
without the * and grasp the idea, then re-read the Wildcards section and try it again with
the * in place. Try it also in different directories, and with the * surrounded by double
quotes, and try it preceded by a backslash (*)

While Loops

while loops can be much more fun! (depending on your idea of fun, and how often you
get out of the house...)

while.sh

```
#!/bin/sh
INPUT_STRING=hello
while [ "$INPUT_STRING" != "bye" ]
do echo "Please type something in (bye to quit)"
read INPUT_STRING
echo "You typed: $INPUT_STRING"
done
```

What happens here, is that the echo and read statements will run indefinitely until you
type "bye" when prompted.

The colon (:) always evaluates to true; whilst using this can be necessary sometimes, it is
often preferrable to use a real exit condition. Compare quitting the above loop with the
one below; see which is the more elegant. Also think of some situations in which each
one would be more useful than the other:

while2.sh
```
#!/bin/sh
while :
do echo "Please type something in (^C to quit)"
read INPUT_STRING
echo "You typed: $INPUT_STRING"
done
```
Another useful trick is the while read f loop. This example uses the case statement, which
we'll cover later. It reads from the file myfile, and for each line, tells you what language it
thinks is being used. Thanks to *Asmus* for a number of improvements to this section.
while3a.sh
```
#!/bin/sh
while read f
do case $f in
hello) echo English ;;
howdy) echo American ;;
gday) echo Australian ;;
bonjour) echo French ;;
"guten tag") echo German ;;
*) echo Unknown Language: $f ;;
esac
done < myfile
```
On many Unix systems, this can be also be done as
while3b.sh
```
#!/bin/sh
while f=`line`
do .. process f ..
done < myfile
```
But since the while read f works with any *nix, and doesn't depend on the external
program line, the former is preferable. See External Programs to see why this method
uses the backtick (`).

Thanks again to *sway* for pointing out that I referred to $i in the default ("Unknown
Language") case above - you will get no warnings or errors in this case, even though $i
has not been declared or defined. For example:
```
$ i=THIS_IS_A_BUG
$ export i
$ ./while3.sh something
Unknown Language: THIS_IS_A_BUG
```

$
So make sure that you avoid typos. This is also another good reason for using ${x} and
not just $x - if x="A" and you want to say "A1", you need echo ${x}1, as echo $x1 will
try to use the variable x1, which may not exist, or may be set to B2.
A handy Bash (but not Bourne Shell) project is:
mkdir rc{0,1,2,3,4,5,6,S}.d
instead of the more cumbersome:
for runlevel in 0 1 2 3 4 5 6 S
do
mkdir rc${runlevel}.d
done
And this can be done recursively, too:
$ cd /
$ ls -ld {,usr,usr/local}/{bin,sbin,lib}
drwxr-xr-x 2 root root 4096 Oct 26 01:00 /bin
drwxr-xr-x 6 root root 4096 Jan 16 17:09 /lib
drwxr-xr-x 2 root root 4096 Oct 27 00:02 /sbin
drwxr-xr-x 2 root root 40960 Jan 16 19:35 usr/bin
drwxr-xr-x 83 root root 49152 Jan 16 17:23 usr/lib
drwxr-xr-x 2 root root 4096 Jan 16 22:22 usr/local/bin
drwxr-xr-x 3 root root 4096 Jan 16 19:17 usr/local/lib
drwxr-xr-x 2 root root 4096 Dec 28 00:44 usr/local/sbin
drwxr-xr-x 2 root root 8192 Dec 27 02:10 usr/sbin

PRACTICAL 16

TITLE: Program for passing parameters and arguments using shell script.

1. Creating a file :
   nano pop.sh
2. add contents to file:
   echo $1, $2
   for i in 'seg $1, $2'
   do
     echo “red $i”;
   done
   exit 0
3. save file and exit:
   ctrl+x
   Y
4. Give executable permissions:
   chmod 777 pop.sh
5. Executable file :
   sh pop.sh

PRACTICAL NO. 17

TITLE: Installation of windows operating system on computer

Steps for installing windows.

1. Turn your computer on, press Del or F2 to enter system BIOS.
2. Go to Boot-menu and choose Boot from CD/DVD.
3. Press F10 to save configuration and exist BIOS, then restart computer.
4. Insert windows DVD into your DVD drive, this will start up computer, windows will be loading files.
5. After your computer reboots, a black screen will appear showing the message " Press any key to boot from CD".
6. Wait for a few seconds until the blue screen pops up until the Title windows set up.
7. Select a drive from windows to be installed, and choose its file system (FAT 32 a NTFS).
8. After a number of restarts, the previous screen with 'Press any key to Boot from CD' will appear. Ignore it this time.
9. Set the region and language and Network settings and let the setup install windows. You can keep track of the progress on the Green bar at the left side of the screen.
10. Install basic security software such as a firewall, antivirus program, and an anti-spyware program, the built-in windows firewall, Gus oftes AVG free and safer networking's spy boot S & D should work fine.
11. Update windows and security software. This should help protect against some viruses and improve stability.
12. Make sure that all your hardware is working. Unlike OEM installs, there can be some issues here. You might want to go to the hardware.
13. You are done with installing windows.

PRACTICAL – 18

TITLE: Installation process of LINUX operating system.

Steps for installing LINUX involves two methods. First method is by booting from a Cd or within windows itself . Ubuntu is one of the most popular form of linux operating system . It is available for free and will run on almost any computer. So the steps are:

1. Downloading ISO file: we can get ISO file from ubuntu website and ISO file is CD image that will need to be burned before you can use it.If you have windows 8 PC or a Pc with 64 bit architecture, download 64 bit version. Most older machines should download 32 version.
2. Burn the ISO file: use your favorite CD/DVD burning software to burn .
3. Boot to disc: Once you have finished burning the disc, restart your computer and choose to boot from disc.You may have to change your boot preferences by hitting setup key while your computer is restarting. This is typically f1,f2,del or assist key.
4. Install ubuntu: your computer will need atleast 45 Gb of free space. You will want more than this if you want to install program and create files. Check the "download updates automatically" box. It will allow you to play mp3 files as well as flash videos, once you check"install this third party software" box.
5. If we have windows installed on system , we will be given couple of options on how we would like to install ubuntu . We ca install it along side our previous window or we can replace windows.
6. If install it along side old version of windows , we will be given option to choose our OS each time. We reboot our computer. Windows file and programs remain untouched.
7. We can replace windows with ubuntu but all windows file, documents and programs will be deleted.
8. Set partition size: if we are installing ubuntu alongside windows, we can use slider to adjust how much space we would like to designate for ubuntu. Remember ubuntu will take upto 4.5 Gb when installed so be sure to leave some space for programs files . Once you are satisfied click install.
9. Choose your location. If you are connected , this should be done automatically, verify time zone and click continue button.
10. Set keyboard layout by choosing from list of options or click detect layout button to have ubuntu automatically pick correct option.
11. Enter your login information.
12. Once you choose login info the installation will begin. During the steps, various tips for using ubuntu will be displayed once it is finished, restart and ubuntu willl load.

PRACTICAL – 19

TITLE : Installation of virtual machine software

Introduction :

Virtual Box is an open source software,is freely available and performs as virtual machine. It can be installed in most popular operating system such as windows XP and Vista, Machintosh and LINUX Hosts . This is a big advantage since you can have two opertaing system in same screen at same time without restarting your machine.

Steps for installation of virtual box:

1. Download the virtual box application.
2. A window will indicate the download progress. Download will take approx. 3 minutes.
3. Once the download is complete , go to the location where you have saved both file and double click on it.A welcome window will open , hit next and accept the conditions, click next.
4. When you get to the custom setup window you can choose in which program you would like to save the program . By default it goes to your program file in your C drive . You have the option of choosing another folder, a partition where program will be stored
5. Once you select the place where you are going to save virtual box,hit next.
6. Next window will ask you to install virtual box go ahead and hit install.
7. Virtual box will be installed in your computer.Hit finish and open the virtual box in your computer.

Go to the start >All Programs>Virtual box.

PRACTICAL 20

TITLE : Installation of Operating System on Virtual Machine.

Procedure :

1. Download Microsoft Virtual PC from Microsoft website.
2. Install the program. You must be running windows XP or higher. However, the problem may run on older system.
3. Once you start the program, it must ask you to make a virtual machine. If not,click on New button.
4. Click on Create a Virtual machine button and click next.
5. Type a name for thew machine button and click next.
6. Select the OS you are going to install this setup,tye recommended space for your install machine. If the OS you are going to install is not there, then click “Other”.
7. Depending on the OS you are using, you may want to adjust the amount of RAM, it will use. Don't choose more RAM than your real computer has OS it will running too.\
8. Click “ A New Virtual Hard Disk” and click next allow you get to choose where to put the virtual hard disk. Usually the default is line. Also set the size of hard disk in MegaByte.
9. Finish the wizard:
   You should see something new in the virtual PC. It should have your virtual OS.
10. Click on it and click “Start”. You should see a bundle of lines or text similar to the line you first entered.

References

1. Unix Concepts and Applications by Sumitbha Das, Tata McGraw Hill Publisher

2. Operating System By Mukesh Kumar, Kalyani Publishers

3. Basis of OS, Unix and Shell Programming by ISRD Group, Tata McGraw Hill Publisher