UNIX LAB MANUAL

Unix Background Information

Purpose:
Since many of the labs will require knowledge of Unix/Linux, we have included some useful background information.

Unix-Linux History
In order to understand the popularity of Linux, we need to travel back in time, about 55 years ago…

Imagine computers as big as houses, even stadiums. While the sizes of those computers posed substantial problems, there was one thing that made this even worse: every computer had a different operating system. Software was always customized to serve a specific purpose, and software for one given system didn't run on another system. Being able to work with one system didn't automatically mean that you could work with another. It was difficult, both for the users and the system administrators. Technologically the world was not quite that advanced, so they had to live with the size for another decade. In 1969, a team of developers in the Bell Labs laboratories started working on a solution for the software problem, to address these compatibility issues.

They developed a new operating system, which was

1. Simple and elegant.
2. Written in the C programming language instead of in assembly code.
3. Able to recycle code.

The Bell Labs developers named their project "UNIX." The code recycling features were very important. Until then, all commercially available computer systems were written in a code specifically developed for one system. UNIX on the other hand needed only a small piece of that special code, which is now commonly named the kernel. This kernel is the only piece of code that needs to be adapted for every specific system and forms the base of the UNIX system. The operating system and all other functions were built around this kernel and written in a higher programming language, C. This language was especially developed for creating the UNIX system. Using this new technique, it was much easier to develop an operating system that could run on many different types of hardware. The software vendors were quick to adapt, since they could sell ten times more software almost effortlessly.

Weird new situations came in existence: imagine for instance computers from different vendors communicating in the same network, or users working on different systems without the need for extra education to use another computer. UNIX did a great deal to help users become compatible with different systems. Throughout the next couple of decades the development of UNIX continued. More things became possible to do and more hardware and software vendors added support for UNIX to their products.
UNIX was initially found only in very large environments with mainframes and minicomputers (note that a PC is a "micro" computer). You had to work at a university, for the government or for large financial corporations in order to get your hands on a UNIX system. But smaller computers were being developed, and by the end of the 80s, many people had home computers. By that time, there were several versions of UNIX available for the PC architecture, but none of them were truly free and more important: they were all terribly slow, so most people ran MS DOS or Windows 3.1 on their home PCs. By the beginning of the 90s home PCs were finally powerful enough to run a full blown UNIX. While there was an academic UNIX-lookalike called Minix available at the time, its creator, Andrew S. Tanenbaum did not allow modifications that would make it more generally usable. He wanted his system to stay "clean", since he created it in order to teach computer science with it.

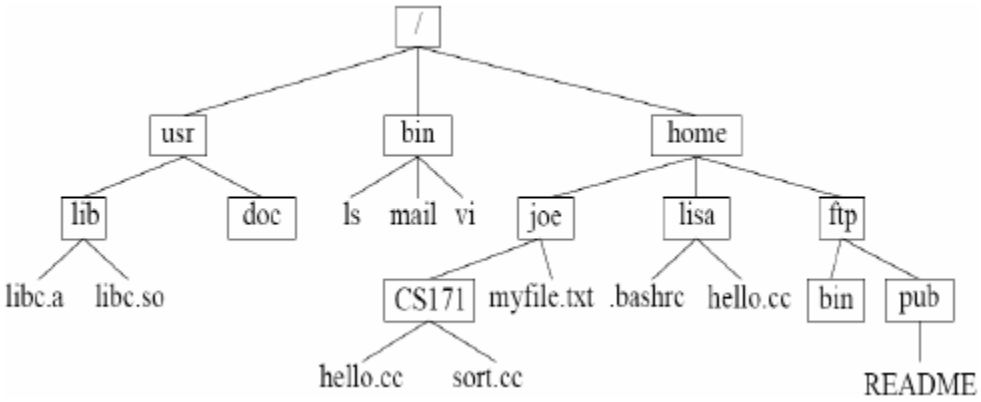
Linus Torvalds, a young man studying computer science at the University of Helsinki, used the Minix, and when he felt too constrained by its limitation, he started to code his own UNIXlookalike operating system. From the start, it was Linus' goal to have a free system that was completely compliant with the original UNIX. That is why he asked for POSIX standards, POSIX still being the standard for UNIX.

In those days plug-and-play wasn't invented yet, but so many people were interested in having a UNIX system of their own, that this was only a small obstacle. New drivers became available for all kinds of new

hardware, at a continuously rising speed. Almost as soon as a new piece of hardware became available, someone bought it and submitted it to the Linux test, as the system was gradually being called, releasing more free code for an ever wider range of hardware. These coders didn't stop at their PCs; every piece of hardware they could find was useful for Linux.

Two years after Linus' post, there were 12000 Linux users. The project, popular with hobbyists, grew steadily, all the while staying within the bounds of the POSIX standard. All the features of UNIX were added over the next couple of years, resulting in the mature operating system Linux has become today. Linux is a full UNIX clone, fit for use on workstations as well as on middlerange and high-end servers. Today, a lot of the important players in the hardware and software market each have their team of Linux developers; at your local dealers you can even buy preinstalled Linux systems with official support–though there is still some hardware and software not supported.

Linux Filesystem



A simple example of a hierarchical file system is shown in the above figure. Each boxed name represents a directory, while the unboxed names are files. Linux file names are case sensitive and may contain almost any character. File names may or may not be followed by an extension like .txt or .cc or html. In fact, the period in a file name is not given any special significance by a shell, and extensions are rarely required for a file to be opened by a particular application. However, it is usually a good idea to include an extension for a file so it is easier for you to figure out what kind of file it is. By convention, executable programs in Linux usually have no extension.

Any directory that is not the root is usually called a subdirectory. For example, in the figure, usr is a subdirectory of / and doc is a subdirectory of usr. The directory usr is also called the parent directory of doc and / is the parent directory of usr. The root directory is the only directory without a parent; by convention, the root directory is its own parent.

In a Linux filesystem, the bin subdirectory contains programs that correspond to core Linux commands. The usr subdirectory contains many other parts of the basic Linux system. The home subdirectory contains the home directories of all the users with accounts on the system. If your username were joe, you could store your files in the joe subdirectory of home.

The pathname of a file contains a sequence of directories to follow to reach the file. For example, the pathname of the joe subdirectory is /home/joe. The pathname of the file myfile.txt in the joe subdirectory is /home/joe/myfile.txt. The pathnames above are called absolute pathnames because they contain all the information needed to find a file. On the other hand, a relative pathname gives the information necessary to find a file from a particular point in the tree. For example, from the directory /home, the relative pathname of myfile.txt is just joe/myfile.txt. Notice that you can tell the difference between an absolute and a relative pathname by looking for the leading forward slash.

Linux Disks and Partitions

Linux treats its devices as files. The special directory where these "files" are maintained is "/dev".

DISKS

- Floppy (a:) /dev/fd0
- Floppy (b:) /dev/fd1
- 1st Hard disk (master, IDE-0) /dev/hda
- Hard disk (slave, IDE-0) /dev/hdb
- Hard disk (master, IDE-1) /dev/hdc, etc.
- 1st SCSI hard disk /dev/sda
- 2nd SCSI hard disk /dev/sdb, etc.

PARTITIONS

- 1st Hard disk (master, IDE-0) /dev/had
  - 1st Primary partition /dev/hda1
  - 2nd Primary partition /dev/hda2, etc.
  - 1st Logical drive (on ext'd part) /dev/hda5
  - 2nd Logical drive /dev/hda6, etc.
- 2nd Hard disk (slave, IDE-0) /dev/hdb
  - 1st Primary partition /dev/hdb1, etc
- CDROM or 3rd disk (master, IDE-1) /dev/hdc
- CDROM (SCSI) /dev/scd0
- 1st SCSI disk /dev/sda
  - 1st Primary partition /dev/sda1, etc.

The pattern described above is fairly easy to follow. If you are using a standard IDE disk, it will be referred to as "hdx" where the "x" is replaced with an "a" if the disk is connected to the primary IDE controller as master and a "b" if the disk is connected to the primary IDE controller as a slave device. In the same way, the IDE disks connected to the secondary IDE controller as master and slave will be referred to as "hdc" and "hdd" respectively Note: Before a filesystems on devices can be used, they must be mounted. In order to mount them, you must know what they arr called. So for example, if you use a parallel ZIP drive or USB disk (thumb drive, memory stick, etc.), it will be accessed as /dev/sda (assuming no other SCSI devices) or /dev/sdb.

Linux File Permissions

Every file or folder in Linux has access permissions. There are three types of permissions (what allowed to do with a file):

- read access
- write access
- execute access

Permissions are defined for three types of users:

- the owner of the file
- the group that the owner belongs to
- other users

Thus, Linux file permissions are nine bits of information (3 types x 3 type of users), each of them may have just one of two values: allowed or denied. Simply put, for each file it can be specified who can read or write from/to the file. For programs or scripts it also can be set if they are allowed to be executed. It is used in Linux long directory listings. It consists of 10 characters. The first character shows the file type. Next 9 characters are permissions, consisting of three groups: owner, group, others. Each group consists of three symbols: **rwx** (in this order), if some permission is denied, then a dash "-" is used instead. Example:
-rwxr--r--
0123456789
Symbol in the position 0 ("-") is the type of the file. It is either:

d = directory
- = regular file
l = symbolic link
s = Unix domain socket
p = named pipe
c = character device file
b = block device file

- Symbols in positions 1 to 3 ("rwx") are permissions for the owner of the file.
- Symbols in positions 4 to 6 ("r--") are permissions for the group.
- Symbols in positions 7 to 9 ("r--") are permissions for others.

r Read access is allowed
w Write access is allowed
x Execute access is allowed
- Replaces "r", "w" or "x" if according access type is denied

Examples:
-rwxr-xr-x
File,
Owner has read, write, and execute permissions,
Group: only read and execute permissions,
Others: only read and execute permissions.

dr-x------
Directory,
owner has read and execute access,
group and others have no access

If a numeric representation is used (like in chmod command, for example), then it is in the octal format (with the base of 8), and digits involved are 0 to 7. Octal format is used for the simplicity of understanding: every octal digit combines read, write and execute permissions together. Respective access rights for owner, group and others (in this order) are the last three digits of the numeric file permissions representation. Example: "0644". Here the second digit ("6" in the example) stands for rights of the owner, the third digit ("4" in the example) stands for rights of the group, the fourth digit ("4" in the example) stands for rights of others.

This table shows what numeric values mean:

Octal digit	Text equivalent	Binary value	Meaning
0	---	000	All types of access are denied
1	--x	001	Execute access is allowed only
2	-w-	010	Write access is allowed only
3	-wx	011	Write and execute access are allowed
4	r--	100	Read access is allowed only
5	r-x	101	Read and execute access are allowed
6	rw-	110	Read and write access are allowed
7	rwx	111	Everything is allowed

Useful Commands

Directories

Linux "folders" are called *directories*. The top-level, root directory is called /. Your home directory is **/home/***username*. From anywhere you can get back there by typing simply **cd** The short-hand name for the directory you happen to be in at any time is called "**.**" and the directory in which the current directory resides is called "**..**". Typing "**cd ..**" will move you to the next higher level directory. Several useful commands for directories are listed below.

Command	**Function**	**Examples**
cd	Change directory	**cd**, **cd ..**, **cd /home/catyp**
pwd	Print working directory	**pwd**
mkdir	Make a new subdirectory	**mkdir** *newdirectory*
rmdir	Remove a directory	**rmdir** *emptydirectory*
ls	List files in a directory	**ls**, **ls –l**

Files:

Files reside in directories. Use the **ls** command (or **ls -l** for more information) to see all the files in a directory. Useful commands for manipulating files include:

Command	**Function**	**Examples**
mv	Rename (move) a file	**mv** *oldname newname*
cp	Copy a file	**cp** *oldname newname* **cp** *oldname dirname/*
rm	Delete (remove) a file	**rm** *filename* **rm** *file1 file2 file3* **rm** *-r dirname*
cat	Output the contents of a file to the screen	**cat** *filename*
file	Identify the type of file	**file** *filename*
head	Display the first few lines of a text file.	**head** *filename*
tail	Display the last few lines of a text file.	**tail** *filename*
chmod	Change access permissions on files	**chmod** mode *filename*
ln	Creates symbolic link	**ln -s** targetfile *linkname*

Other Commands:

Command	**Function**	**Examples**
passwd	Change your password.	**Passwd**
ps	List running processes of the current terminal. If you want to see all processes of the current user, use "-a", if you're still not satisfied with the output, try "-e".	**ps -aux**
kill	Stop a process by passing its process-id (shown by ps as PID).	**kill –TERM** *process-id* **kill -9** *process-id*
tar	Create / expand / query archives.	**tar cfv** *arch.tar somedir/* **tar xfv** *arch.tar*
mount	Make a device visible in the filesystem. For users, this is typically used to access CD-Roms or floppies.	**mount** */mnt/floppy*
umount	Un-mount a device from a given point in the file system.	**umount** *floppy*
df	Reports on used disk space on the partition containing file.	**df** *file*
find	Find files in the file system hierarchy	**find** *path expression*
grep	Print lines in file containing the	**grep** *PATTERN file*

search pattern.
ifconfig Configures a network interface **ifconfig**
exit or logout Leave this session **Exit**

vi editor:
The most popular editor under unix environment is vi editor, a visual editor. It supports most programming languages and their syntax. This will be in three modes:

1. Command mode: The default mode where every key pressed is interpreted as command.
2. Input mode: This mode allows us type in the text.
3. ex mode (or last line mode): This mode is used to handle files.

To create or modify a text file, type *vi filename <enter>* at the shell prompt.
Example: …]$ vi hello.c
The vi editor displays the contents of the file, if it exists. The editor will be in the command mode. Press *i* to insert text (Input mode). To return to command mode press [Esc] key. Press : here to get to ex mode.

Commonly used commands

S.No.	Command	Action
1	h	Moves the cursor to the previous character
2	l	Moves the cursor to the next character
3	k	Moves the cursor to the previous line
4	j	Moves the cursor to the next line
5	x	Deletes the character at the current cursor position
6	:wq<enter>	Saves and quits the editor
7	:w<enter>	Saves the file
8	:q<enter>	Quits the editor
9	:q!<enter>	Forcibly quits even if some changes were made
10	:e<filename><enter>	Opens the specified file
11	:r<filename><enter>	Reads and inserts the contents of the file after the current line
12	:w<filename><enter>	Writes to specified file
13	:w! <filename><enter>	Forcibly Writes to specified file
14	:!<command><enter>	Executes a shell command

Insert and replace commands

S.No.	Command	Action
1	a	Append text after the current character
2	A	Append text at the end of the line
3	i	Insert text before the current character
4	I	Insert text at the beginning of the line
5	o	Insert blank line below the current line and insert
6	O	Insert blank line above the current line and insert

Navigation Commands

S.No.	Command	Action
1	<ctrl>d	Scroll down half-screen
2	<ctrl>u	Scroll up half-screen
3	<ctrl>F	Move a page farward
4	<ctrl>B	Move a page backward
5	0 (zero)	Move to the beginning of the line
6	$	Move to the end of the line
7	w	Move to the next word
8	b	Move to the previous word
9	e	Move the the end of the word

Editing commands

S.No.	Command	Action
1	dw	Delete aword

2	dd	Delete a line
3	cw	Change a word
4	cc	Change a line
5	J	Join lines
6	u	Undo the last change
7	.	Repeat the last change
8	Y or yy	Yank the current line
9	*n*Y or *n*yy	Yank *n* lines including the current line
10	p	Place the yanked text after the current line
11	P	Place the yanked text before the current line

Searching commands

S.No.	Command	Action
1	/pattern<enter>	Find the next line containing the pattern
2	?pattern<enter>	Find the previous line containing the pattern

Week – 1

Exercise 1 (Getting Familiar with Linux)

Enter these commands at the UNIX prompt, and try to interpret the output:

- echo hello world
- passwd
- date
- hostname
- arch
- uname -a
- dmesg | more (you may need to press q to quit)
- uptime
- who am i
- who
- id
- last
- finger
- w
- top (you may need to press q to quit)
- echo $SHELL
- echo {con,pre}{sent,fer}{s,ed}
- man "automatic door"
- man ls (you may need to press q to quit)
- man who (you may need to press q to quit)
- lost
- clear
- cal 2000
- cal 9 1752 (do you notice anything unusual?)
- bc -l (type quit or press Ctrl-d to quit)
- echo 5+4 | bc -l
- yes please (you may need to press Ctrl-c to quit)
- time sleep
- history

Exercise 2 (Getting Familiar with Linux)

Try the following command sequence:

- cd
- pwd
- ls –al
- cd .
- pwd
- cd ..
- pwd
- ls –al
- cd ..
- pwd (what happens now)
- cd /etc
- ls –al | more
- cat passwd
- cd ~
- pwd

1. Change back into your home directory.
2. Make subdirectories called **work** and **play.**
3. Delete subdirectory called **work.**
4. Copy file /etc/passwd into your home directory.
5. Move it into the subdirectory **play**.

Week – 2

SHELL PROGRAMMING

Exercise 3

Write a shell script to generate a multiplication table.

a) Interactive version: The program should accept an integer *n* given by the user and should print the multiplication table of that *n.*
b) Command line arguments version: The program should take the value of *n* from the arguments followed by the command.
c) Redirection version: The value of *n* must be taken from a file using input redirection.

Use the commands **read, echo, expr, while,** *or* **for**.

Exercise 4

Write a shell script that copies multiple files to directory.

a) Interactive version
b) Command line arguments version

Use the commands **echo, read, cp, mkdir**.

Exercise 5

Write a shell script which counts the number of lines and number of words present in a given file.

a) Interactive version
b) Command Line arguments version

Use the commands **echo, read, wc**.

Exercise 6

Write a shell script which displays the list of all files in a given directory.

a) Interactive version
b) Command Line arguments version

Use the commands **echo, read, ls**.

Exercise 7

Write a shell script (small calculator) that adds, subtracts, multiplies and divides the two given numbers. There are two division options: one returns the quotient and the other remainder. The script requires three arguments: the operation to be used and the two integers. The operation are specified by options:

Add	-a
Subtract	-s
Multiply	-m
Quotient	-c
Remainder	-r

Use the **if** *and* **case** *structures*.

Week – 3

Exercise 8

Write a shell script to determine whether a given number is a prime number or not.

a) Interactively.
b) By command line arguments.

Exercise 9

Write a shell script to print all the primes below a given number is a prime number.

a) Interactively.
b) By command line arguments.

Exercise 10

Write a shell script to print the first *n* Fibonacci numbers.

a) Interactively.
b) Using Command line arguments.

Exercise 11

Write a shell script to find the gcd of two given numbers.

a) Interactively.
b) By command line arguments.

Exercise 12

Write a shell script to reverse the rows and columns of a matrix.

a) Interactively.
b) By command line arguments.

Exercise 13

Write a shell script to find the scalar product of two vectors.

a) Interactively.

b) By command line arguments.

Week – 4

Exercise 14

Write a C program that counts the number of blanks in a text file.
a) Using standard I/O
b) Using system calls

Exercise 15

Write a C program to count the number of words, lines and characters of a given text file.
a) Interactively
b) Command line arguments
c) Using input redirections

Exercise 16

Implement in C the following unix commands using system calls.
a) cat
b) ls
c) mv

Exercise 17

Write a C program that takes one or more file/directory names as command line input and reports the following information on the file:
a) File type
b) Number of links
c) Time of last access
d) Read, write, and execute permission

Exercise 18

Write a program in C that illustrates how to execute two commands concurrently with a command pipe.

Exercise 19

Write a c program that illustrates the creation of a child process using fork system call.

Week – 5

Exercise 20

Write a C program that displays the real time of a day every 60 seconds.

Exercise 21

Write a C program that illustrates for file locking using semaphores.

Exercise 22

Write a C program that implements a producer-consumer system with two processes (using semaphores).

Exercise 23

Write a C program that illustrates inter process communication using shared memory system calls.

Exercise 24

Write a C Program that illustrates the following:

a) Creating message queue.
b) Writing to a message queue.
c) Reading from a message queue.

Week – 6

Exercise 25
Write a C program to copy a file into another using system calls.

Exercise 26

Write a C program that reads a file and writes in the reverse order.

Exercise 27
Write a C program to print all error messages.

Exercise 28
Write a C program to list only directories.

Exercise 29
Write a C program to check all 12 permission bits of a file.

Exercise 30
Write a C program to determine a file's access rights using UID and GID.

Week – 7

Exercise 31
Write a C program to set a file's time stamps to those of another file.

Exercise 32
Write a C program using fork, exec, and wait to run a UNIX command.

Exercise 33
Write a C program to accept user input as a command to be executed and then uses the strtok library function for parsing command line.

Exercise 34
Write a C program to run two programs in a pipeline (Child runs cat, parent runs tr).

Exercise 35

Write a program that uses fork and exec to run a user-defined program and kills it if does not complete in 5 seconds.