CHARUTAR VIDYA MANDAL’S
SEMCOM
Vallabh Vidyanagar

Faculty Name: Ami D. Trivedi
Class: TYBCA (Semester-V)
Subject: US05EBCA01 (Basics of UNIX Operating System)

*UNIT – 1 (Working with UNIX-like Systems)

<u>*BRIEF HISTORY OF UNIX AND LINUX</u>

Some key dates in the development of different Unix versions:

1970	Ken Thompson suggests the name "Unix" for the fledging operating system born in 1969 at AT&T Bell Labs.
1973	The kernel (core) of Unix is re-written in the C language, making it the world’s first operating system that’s "portable"—that is, able to run on multiple kinds of hardware.
1977	First BSD (Berkeley Software Distribution) version released. Licensees must also get a license from AT&T.
1983	Version 4.2 BSD is released. By the end of 1994, more than 1,000 licenses are issued. AT&T release its commercial version "System V"
1983	AT&T releases "System V release 3." IBM, Hewlett-Packard and others base their own Unix-like systems on this version.
1991	Linus Torvalds releases version 0.02 of Linux. An open source Unix-like operating system.
1992	GNU software integrated with Linux kernel, producing a fully functional operating system
1992	Bill Jolitz releases 386/BSD, a full version of Unix with no AT&T code.
1992	Sun Microsystems release Solaris, a version of Unix based on System V release 4, incorporating many BSD features.
1994	BSD4.4-Lite is released by Berkeley. It is entirely free of legal encumbrances from the old AT&T code. Version 1.0 of Linux is also released this year; Linux incorporates features from both AT&T’s System V and BSD versions of Unix.
1995	Linux adapted to non-Intel processors.
1996	Linux supports multiple processors.
1999	Apple releases Darwin, a version of BSD Unix, and the core of the Mac OS X.
1999	Linux growth rate exceeds that of Microsoft Windows NT.

<u>STRENGTHS OF UNIX-LIKE OPERATING SYSTEMS</u>

1. Scalability

UNIX is proven to scale in very large environments. It is used on various hardware platforms, from workstations to supercomputers.

2. Mature platform

Forms of UNIX have been in place for more than 20 years. It offers a variety of software, development toolkits and utilities. Plenty of free software available particularly Internet services available on nearly every hardware platform (from PCs to mainframes).

3. Management

UNIX is managed at a very low level through a character-based interface, making it easy to access all administrative functions remotely. X Windows is network-enabled, which will allow any GUI utilities be accessed remotely.

4. Large-scale directory services

Lacks a standard directory service, but products like NIS, NIS+ and DCE directory services integrate closely with the OS and offer Unix-specific schemas by default.

5. Virtual Memory

UNIX operating system offers an efficient level of virtual memory. For the user, it means that you can use a number of programs at the same time using only a modest level of physical memory. The system can handle several programs at once without severely pulling on the system's resources.

6. Toolbox

This operating system offers a rich collection of small utilities and commands that are designed to carry out specific tasks well rather than being cluttered up with a variety of special but insignificant options. UNIX acts as a well-stocked toolbox rather than attempting to do everything at once.

7. Customization

UNIX has the ability to string different utilities and commands together in an unlimited number of configurations in order to accomplish a variety of complicated tasks. This operating system is not limited to any pre-configured menus or combinations as most ordinary personal computer systems normally are.

8. Portability

UNIX is available for use on a variety of different types of machines, making it one of the most portable operating systems in existence. UNIX can be run on both PC and Macintosh computers and many other computing machines as well.

9. Full multitasking with protected memory

Multiple users can run multiple programs each at the same time without interfering with each other or crashing the system.

10. Multiuser

UNIX can be used by multiple users. Windows is a single user system where memory, CPU and hard disk are dedicated to a single user. In UNIX, the resources are actually shared between all users.

UNIX is a multiuser system where several users are working on the same machine at the same time.

11. Access controls and security

All users must be authenticated by a valid account and password to use the system at all. All files are owned by particular accounts. The owner can decide whether others have read or write access to his files.

12. Ability to string commands and utilities together in unlimited ways to accomplish more complicated tasks -- not limited to preconfigured combinations or menus, as in personal computer systems.

13. A powerfully unified file system

Everything is a file: data, programs, and all physical devices. Entire file system appears as a single large tree of nested directories, regardless of how many different physical devices (disks) are included.

14. Kernel that does the basics for you but doesn't get in the way when you try to do the unusual.

<u>WEAKNESSES OF UNIX-LIKE OPERATING SYSTEMS</u>

1. **Not standardized**

   Incompatible versions of UNIX--applications written to one environment must be ported to another. Most portable Unix applications are not multithreaded.

2. **Cost—capital**

   Scalable, high performance RISC solutions are very expensive compared to PC hardware.

   - Reduced instruction set computing, or RISC, is a CPU design strategy based on the insight that simplified (as opposed to complex) instructions can provide higher performance if this simplicity enables much faster execution of each instruction. A computer based on this strategy is a reduced instruction set computer also called RISC.

3. **Cost—management**

   Complex OS requires experienced administrators. Most versions have simplified installation processes and each vendor offers different management utilities

4. **A large number of different Linux distributions**

   Number of available Linux distribution can be a scary thing. Each distribution has only a difference in the packages included. The best strategy is to try several distributions; unfortunately not all people have the time to do it.

5. **Not so easy to use (not very user friendly) and confusing for new users**

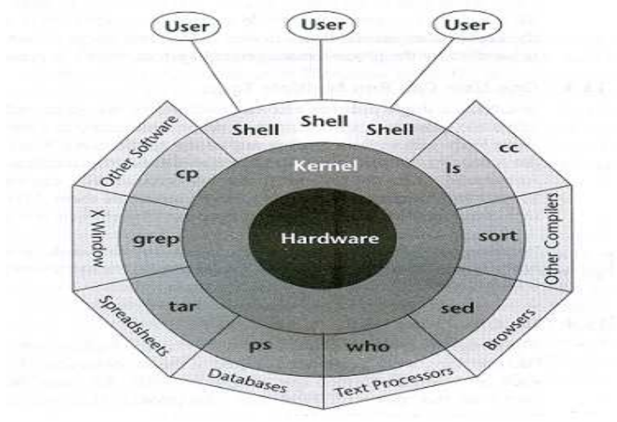
   Linux, at least the core system, it is more difficult to use than MS Windows and much more difficult than MacOS.

6. **Are open source products can be trusted?**

   Linux being free creates a question of reliability. In case of any problem or error, no one is held responsible.

<u>*BASIC CONCEPTS IN UNIX-LIKE SYSTEMS</u>

<u>KERNEL</u>



The kernel is the center of the operating system, which communicates with the hardware directly.

The kernel is the part of the UNIX system that is loaded into memory when system is booted.

Kernel

1. Manages the system resources
2. Allocates the CPU time between users and process
3. Decides process priorities
4. Does scheduling of various programs running in memory
5. Carries out all the data transfer between file system and hardware
6. Assigns storage for files
7. Runs shell programs
8. Handles interrupts &
9. Performs other tasks (about which user doesn't bother)

Other programs also access the services of the kernel through a set of functions called system calls.

The kernel is the Operating system – a program's gateway to the computer resources.

SHELLS (COMMAND INTERPRETER)

Computers don't have inherent (inbuilt) capabilities of translating commands into actions.

An interpreter is required. In Linux system, this job is done by the "outer part" of the operating system - the shell.

The shell takes command from the user, translate the special characters that it hopes to find and rebuild a simplified command line.

It finally communicates with the kernel to see that the command is executed.

Shell is actually an interface between the user and the kernel.

Shell insulates the user from kernel functions.

Types of shells

Different people implemented the interpreter function of shell in different ways. Most popular types of shell are as below.

1. **Bourne Shell**

   It is Steve Bourne's creation. Bourne Shell is the most popular shell.

2. **C Shell**

   This shell is a hit for those who are interested in Unix programming. It was created by Bill Joy.

3. **Korn Shell**

   This shell is not so widely used. It is very powerful. It is superset of Bourne Shell. It was designed by David Korn.

4. **Bash Shell (Bourne-again Shell)**

   Bash is a Unix shell written by Brian Fox. It is a free replacement for the Bourne Shell. It has been distributed widely as the default shell on Linux and Mac OS X.

   Bash is a command processor, typically run in a text window, allowing the user to type commands which cause actions.

   Bash supports filename wildcarding, piping, command substitution, variables and control structures for condition testing and iteration.

MULTIUSER

UNIX can be used by multiple users.

Windows is a single user system where memory, CPU and hard disk are dedicated to a single user.

In UNIX, the resources are actually shared between all users.

UNIX is a multiuser system where several users are working on the same machine at the same time.

In a multiuser system, the computer resources- hard disk, memory etc. are accessible to many users.

The users don't use together at the same computer, but are given different terminals from where they can operate.

A **terminal** is a keyboard and a monitor which are input and output devices for that user.

All terminals are connected to the main computer whose resources are availed by all users. Main computer is known as host machine (server or console).

So, a user at any of the terminals can use computer and the attached peripherals. E.g. printer.

This setup is very economical instead of having one computer per user. It is also convenient when same data is to be shared by all.

MULTITASKING - ONE USER CAN RUN MULTIPLE TASKS

In UNIX single user can also run multiple tasks concurrently. UNIX is capable of carrying out more than one job at a time.

For example, edit a file, print another one on the printer, send e-mail to a friend and browse the World Wide Web – all without leaving any of the application.

This is managed by dividing the CPU time between all processes which are being carried out. Depending on the priorities of the task, the operating system allots small time slots (milli or micro seconds) to each foreground and background task.

In multitasking only one job runs in the foreground while the rest run in the background process.

We can switch jobs between background and foreground. We can suspend jobs or terminate them.

The kernel is designed to handle a user's multiple needs.

Multiuser concept expects multitasking. UNIX has to be on its toes all the time to serve all the users connected to it.

Windows is also a multitasking system.

REMOTE ACCESS (REMOTE LOGIN)

Remote login means connecting to a remote machine using a username and password. After logging in, all entered commands are executed on the remote machine.

Rlogin (remote login) is a UNIX command that allows an authorized user to login to other UNIX machines (hosts) on a network. It also allows users to interact as if the user were physically at the host computer.

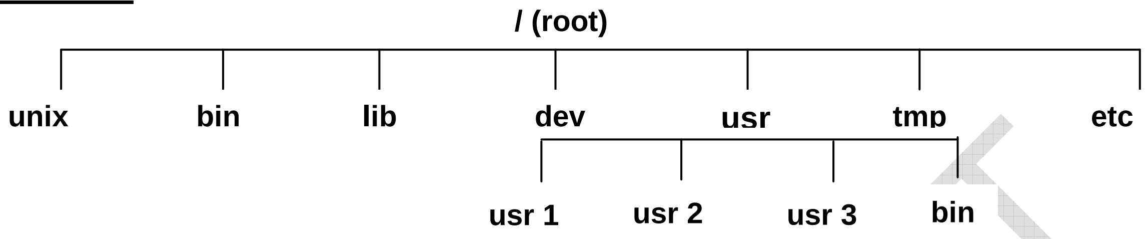
Once logged in to the host, the user can do anything that the host has given permission for, such as read, edit, or delete files.

TELNET

telnet is a TCP/IP application that allows user to log on to a remote machine after supplying a username and password.

After logging in, the user can use the remote machine like a local machine. All files are created on remote machine.

FILE SYSTEM



Before learning any Linux commands it is essential to understand the Linux file system.

Linux treats everything it knows and understand, as a file.

All applications, data in Linux stored as a file. Even a directory is treated as a file which contains several other files.

The Linux file system resembles (looks like) an upside down tree. The file system begins with a directory called root. The root directory is denoted as slash (/).

Branching from root there are several other directories called bin, lib, usr, etc, tmp and dev.

The **root** directory also contains a file called **Unix** which is Unix kernel itself.

These directories are called sub directories. Their parent directory is root. Each of these sub directories contains many files and directories known as sub-sub-directories.

Main reason behind creation of directories is to keep related files together. Other reason is to separate them from other group of related files.

Purpose of sub directories

1. The **bin** directory contains executable files for most of the UNIX commands.
2. The **lib** directory contains all the library functions provided by UNIX for programmers.
3. The **dev** directory contains device related files. These files control various input / output devices like terminals, printers, disk drives etc. In UNIX, device is implemented as a file.

   e.g. everything typed on your terminal is first written to a file associated with your terminal. And this file is present in dev directory.
4. In the **usr** directory, there are several other directories. Each directory is associated with a particular user. It is the home directory of all users.

   These directories are created by the system administrator when he creates accounts for different users. Each user is allowed to work with is directory which is often called home directory.
5. **/usr/bin** directory contains additional UNIX command files. It is another bin directory under usr directory.
6. **tmp** directory contains the temporary files created by unix or by users.

   These files are created for a temporary purpose. So, it gets automatically deleted when the system is shutdown and restarted.
7. **etc** directories contains binary executable files usually required for system administration.

PROCESS

UNIX is a multi-user multi-tasking operating system. It means that any instant, there may be several programs of several users are running in memory.

All these programs share CPU. It ensures that CPU doesn't sit idle and thus overall efficiency of the computer is improved.

There is a common misconception that a program and a process are one and same thing.

A program is elevated to the status of a process when it starts executing. So,
A **process** is an instance of running (an executing) program.

If, for example, three people are running the same program simultaneously, there are three processes there, not just one.

In fact, we might have more than one process running even with only one person executing the program, because the program can "split into two," making two processes out of one.

Keep in mind that all Unix commands, e.g. cc and mail, are programs. If 10 users are running mail right now, that will be 10 processes. At any given time, a typical Unix system will have many active processes, some of which were set up when the machine was first powered up.

UNIX identifies every process by a Process Identification Number (pid) which is assigned when the process is initiated. When we want to perform an operation on a process, we usually refer to it by its pid.

UNIX is a **timesharing** system, which means that the processes take turns running. Each turn is a called a **timeslice**. On most systems this is set at much less than one second. The reason this turns-taking approach is fairness. We don't want a 2-second job to have to wait for a 5-hour job to finish. It would happen if a job keeps the CPU until it completed.

Scheduling

There may be several processes running in memory at any given moment. But the CPU can serve to only one of these processes. Other processes wait for their turn.

A program called **scheduler** running in memory decides which process will get the CPU and when it will get.

At any given moment, a process in memory can be in one of the 6 states.

States of a process

1. **submit**

   When you execute a program, the scheduler submits your process to a queue called **process queue**. At this instant, the process is said to be in submit state.

2. **hold**

   Once submitted, the process waits for its turn in the queue for some time. At this stage, the process is said to be in hold state.

3. **ready**

   As the process advances in the queue, at some instant it would become the next one in the queue to get CPU. At this stage, it is in ready state.

4. **run**

   Finally, the process will get the CPU and starts getting executed. Now, it is in run state.

   In the middle of this execution, it may happen that the time slice allotted to this process gets over. Here, CPU starts running another process. So old process is returned to ready state and placed back in process queue.

All necessary parameters of old process are saved so that it can be retrieved when next time slice arrives.

Old process is now in ready state waiting for its next time slice to arrive.

5. **Wait**

   Some processes may require to do disk input / output. Input / Output is a slow operation. CPU can't sit idle till the I/O is over.

   So, such processes are put in wait state until their I/O is over. After that they will be placed in the ready queue.

6. **complete**

   A process whose execution comes to an enf goes into complete state. It is then removed from the process queue.

ENVIRONMENT AND ENVIRONMENT VARIABLES

Environment

When a shell variable is exported, it is added to the environment. The **environment** is the set of all exported variable.

$ export Note: it will display list of exported variable.

The importance of the environment is that a process can use any of the variables in its environment.

The basic environment of your login shell is determined when you log in. export commands are placed in your **.profile** file.

Usually, this environment consists of **HOME, PATH, TERM etc.** variables.

We can augment (expand) this environment with any variable that we define and export in .profile.

```
$ name=bharat
$ export name
```

Variable name is now part of the environment of current shell and all processes including shell scripts.

If you export a variable in a subshell, that variable will be part of environment only while that shell survives. When you return to parent shell, the variable will not be part of the environment.

Environment variables

UNIX system is controlled by number of special shell variables. Some variables are set during booting process and some are set after logging in.

When you start a new sub-shell, some of these variables are inherited by the sub-shell from its parent.

Environment variables or System variables have a global scope in contrast to simple shell variables which are local.

Environment variables are visible in user's local environment – the sub-shells that run a script, mail commands and editors.

Major environment variables

The major environment variables for the shell are as follows:

1.	**HOME**	:	Home directory – the directory a user is placed on logging in
2.	**PATH**	:	List of directories searched by shell to locate a command
3.	**USER or LOGNAME**	:	Login name of user
4.	**MAIL**	:	Absolute pathname of user's mailbox file
5.	MAILCHECK	:	Mail checking interval for incoming mail
6.	HISTSIZE	:	Number of commands saved in memory
7.	HISTFILESIZE	:	Number of commands saved in history file
8.	HISTFILE	:	History file
9.	**TERM**	:	Type of terminal
10.	**PWD**	:	Absolute pathname of current directory
11.	CDPATH	:	List of directories searched by cd when used with a non absolute pathname
12.	**PS1**	:	Primary prompt string
13.	**PS2**	:	Secondary prompt string
14.	**SHELL**	:	User's login shell

Note: Environment variables displayed in bold are very important.

1. HOME

Home directory – the directory a user is placed on logging in

When you log in, UNIX normally places you in a directory which is your login name. This directory is called the **home or login directory**and it is available in variable HOME.

```
$echo HOME
/home/ty100
```

A user's home directory is specified in the line corresponding to that user in /etc/passwd. This file contains a line for every user with seven fields per line. The home directory is specified in the second-to-last field.

When a user logs in, the **login** program reads the file and sets HOME and SHELL variable.

/etc/passwd can be edited only by system administrator – either manually or with useradd or usermod commands.

2. PATH

List of directories searched by shell to locate a command

PATH is one of the important system variables. This variable instructs the shell about the route to be followed to locate any executable command.

```
$echo PATH
/bin:/usr/bin:/usr/dt/bin:/home/ty100/bin:.
```

Here, we have list of 5 directories separated by a colon (:). The single **.** (dot) at end represents current directory. Placing single dot at end means the shell will look for a command in the current directory only after search in the previous directory files.

If we want to include the directory /usr/script/bin in the search list then we need to redefine this variable.

$PATH=$PATH:/usr/script/bin Note: Adding old value to new value

Here, new directory will be searched last -even after the current directory.

If there are two commands having same name in two different directories, then command will be executed whose directory name appears first in the PATH list.

3. USER or LOGNAME

Login name of user

This variable shows your username. When weroam around in the file system, we may forget our login name – especially when we have multiple accounts.

```
$echo $LOGNAME
ty100
```

We can use this variable in a script which does different things depending on user invoking the script.

4. MAIL

Absolute pathname of user's mailbox file

UNIX mail handling system doesn't inform the user that mail has arrived.

This job has to be done by shell. Shell knows the location of a user's mailbox from variable MAIL. This mailbox is generally /var/mail, /var/spool/mail or /usr/spool/mail.

5. MAILCHECK

Mail checking interval for incoming mail

MAILCHECK determines how often the shell checks the file for the arrival of new mail – typically 600 seconds on a large system.

If the shell finds the file modified since the last check, it informs the user with this familiar message:

You have mail in /var/mail/ty100

If ty100 is running a command, he/she will get this message on terminal only after the command has completed its execution.

6. HISTSIZE

Number of commands saved in memory

This variable is used to determine the size of the history list in memory.

$HISTSIZE=500 Note: It will set number of commands to be saved in memory to 500.

7. HISTFILESIZE

Number of commands saved in history file

This variable is used to determine the maximum size of the file in which the shell saves commands.

$HISTFILESIZE=1000

8. HISTFILE

History file

Shell assigns an **event number**to each command and saves all commands in a history file.

This variable contains name of a file which contains history list.

9. TERM

Type of terminal

TERM indicates the terminal type which is being used. Every terminal has certain characteristics. They are defined in a separate control file in the directory /usr/lib/terminfo.

This directory contains a number of subdirectories named from the letters of the alphabet. A terminal's control file is available in a directory having a one-letter name which is same as the first letter of the terminal name.

For example, ansi terminals use the file /usr/lib/terminfo/a/ansi.

Some utilities like vi editor are terminal dependent. And they need to know the type of terminal you are using. If TERM is not set properly, vi will not work and the display will be faulty.

Many UNIX tools do not work or produce garbage on the screen because of incorrect setting of TERM.

10. <u>PWD</u>

Absolute pathname of current directory

Refer PWD from unit-1.

11. <u>CDPATH</u>

List of directories searched by cd when used with a non absolute pathname

We may be regularly visiting other directories located elsewhere in the file system.

For example, you have two directories prog1 and prog2 under your home directory. You are currently in prog1. If you want to move to prog2, you can use **cd../prog2**.

You can cut a few keystrokes by first setting CDPATH/

CDPATH:.:..:/home/ty100

This string contains 3 subdirectories. Shell searches it in sequence to look for a directory.

When we use cd prog2 form prog1, it will search the current directory (**.**) for prog2. Failing to which it will search the parent directory (**..**). Here prog1 and prog2 are at same hierarchical level, it will locate prog2.

```
$pwd
/home/ty100/prog1
$cd prog2
$pwd
/home/ty100/prog2
```

12. <u>PS1</u>

Primary prompt string

The shell has two prompts stored in PS1 and PS2. The primary prompt string PS1 is the one you normally see.

If you continue a command to the next line, shell responds with a >. The > is secondary prompt string stored in PS2.

Normally PS1 in the Bourne shell are set to character $. You can change the primary prompt string to for eg. C>.

```
$PS1="C>"
C>_
```

$ is most commonly used primary prompt string, the system administrator uses the # as the prompt.

13. PS2

Secondary prompt string

If you continue a command to the next line, shell responds with a >. The > is secondary prompt string stored in PS2.

$cpfile1[Enter]
>file2

Normally PS2 in the Bourne shell are set to character >.You can change the secondary prompt string to for eg. +.

$PS2="+"
$cpfile1[Enter]
+file2

14. SHELL: User's login shell

SHELL tells you the shell you are using.

Refer types of shells from Unit-1.

COMMAND LINE

For eg. $ ls –l file1 file2

Here, ls is command. -l, file1 and file2 are **arguments**.

-l is a special argument which begins with – symbol. Arguments that start with – symbol are known as **options**.

The command with its arguments and options is entered in one line that is referred as **Command Line**.

This line can be considered complete only after the user press Enter key. The complete line is then given to shell as its input for interpretation and execution.

Combining Options

Options that begins with – sign, can be combined with only one – sign.
\
For e.g. **ls –l –a** can be written as **ls –la**. We can write options in any sequence. E.g **ls -al**

Command line can overflow

A command is normally entered in a line by typing from the keyboard. A terminal width is restricted to 80 characters.

But we can enter a command whose width is more than 80 characters. In this situation, the command overflows to next line.

Sometimes it is necessary and sometimes desirable to spread the command into multiple lines. Shell issues a **secondary prompt**. Secondary prompt is generally > sign. It indicates that the command line is not complete.

$ echo "This is
>a message"
This is
a message.

Command line argument

- **Positional parameters**

On many occasions we need to convey information to a program.

A convenient way of doing this is by specifying arguments at command line. How does the shell script know that what has been passed to it?

For this, shell uses special variables called **Positional parameters**. These are the variables defined by the shell. They are nine in number, **$1 to $9**.

If we execute shell script script1 with following command

```
$ sh script1 Hello Hi
```

then each word will be automatically stored in positional parameters.

Script name – **script1** is read by **s$0.** First argument Hello will be read by **$1** and second argument Hi will be read by **$2**.

$# will contain total number of arguments.

$* will contain complete set of positional parameters as a single string.

- **Passing command line arguments**

The knowledge of positional parameters opens up a wider scope to programming. It also makes things easier.
e.g. We want to write a shell script which accepts two file names from the command line. And copies the first file to second file and then display it.

```
# copyfile
# Usage: copyfile <source filename> <target filename>

cp  $1 $2
cat $2
```

```
$ sh copyfile file1 file2
Hello
How are you?
I am fine.
```

The statement **cp $1 $2** is translated by the shell as **cp file1 file2. $1** receives first argument and **$2** receives the second.

ONLINE MANUAL (man: ON LINE HELP)

UNIX documentation comes in number of forms of varying complexities.

Most important form is viewed with the **man-**command, often called **man documentation.**

It is most complete and authoritative guide to UNIX system. The **man** facility is present in every system. To view the manual page of the C shell, you can use **man** with csh as argument.

```
        man csh
```

The entire **man** page pertaining to the **csh** command is dumped on the screen.

man presents first page and the pauses. We need to press spacebar or enter key to see the next page.

To quit from man, press q.

We can see man pages of multiple commands with single man command.

```
$ man cp mv rm
```

This command first shows the page for cp command. to look for next command press q. a q at the page for rm will quit man. To abort in middle, use ctrl-c or Delete character.

USING THE vi EDITOR

While working with UNIX, we may need to edit some files. vi is a full screen editor used by many Unix users. The vi editor has powerful features to help programmers.

vi offers some mnemonic (symbolic) internal commands for editing work. It makes near-complete use of the keyboard. Every key has a function.

- **Starting the vi Editor**

The vi editor lets a user create new files or edit existing files. The command to start the vi editor is vi, followed by the filename.

For example to edit a file called temp, you would type vi temp and then press enter key. You can start vi without a filename, but when you want to save your work, you will have to tell filename to vi.

When you start vi for the first time, you will see a screen filled with tildes (A tilde looks like this: ~) on the left side of the screen. At the bottom of your screen, the filename should be shown, if you specified an existing file, and the size of the file will be shown as well, like this:
"filename" 21 lines, 385 characters

If the file you specified does not exist, then it will tell you that it is a new file, like this:
"newfile" [New file]

- **Modes of operation and switching between them**

VI editor has three modes: **command, insert / input and ex / last line.**

1. **Command mode**

Command mode is default mode of the editor. In this mode every key pressed is interpreted as a command to act on text. These commands are usually one or two characters long.

The command mode allows the entry of commands to manipulate text. If we hit **escape** key being in command mode, then there will be a beep sound to tell you that you are already in that mode.

We can not use command mode to enter or replace text. To enter a text, we have to leave the command mode and enter the input mode.

2. **Insert or Input mode**

Insert or Input mode is used to enter new text, or edit existing text. This mode is invoked by pressing one of the keys (i, a, I, A, o, O, R, s, S). The most commonly used commands to get into insert mode are a and i.

To move to command mode, press escape key in this mode. This mode is also known as input-text mode.

3. **ex or Last line mode**

In ex or Last line mode, commands are entered in last line of the screen. The bottom line of vi screen is called command line. These commands act on text.

It is used for saving, perform substitution (replacing one string with another) etc.

To invoke this mode, press **:** in the command mode. Then you can enter ex mode command followed by enter key. After the command is run, you are back to the default command mode.

- **Text navigation**

Navigation

h	move the cursor to the left one character position.
j	move the cursor down one line.
k	move the cursor up one line.
l	move the cursor to the right one character position.

Word navigation

w	Moves cursor to the right, to the first character of the next word.
b	Moves cursor back to the first character of the previous word.
e	Moves cursor to the end of the current word.

Moving to line extremes

0	Moves cursor to beginning of current line.
|	Moves cursor to beginning of current line. It takes a repeat factor and using it you can position cursor on a certain column. E.g 30| will take cursor on column 30.
$	Moves cursor to end of the current line.
G	To move to specific line. E.g. 40G will take cursor to line number 40.1G goes to line number 1. Only G goes to end of file.

Scrolling

^f	Scroll forwards one page. A count scrolls that many pages. E.g. 10^f will scroll 10 pages.
^b	Scroll backwards one page. A count scrolls that many pages.
^d	Scroll forwards half a window. A count scrolls that many lines.
^u	Scroll backwards half a window. A count scrolls that many lines.

- **Editing text**

Insert & Append

a	Enter insert mode, the characters will be inserted after the current cursor position. If you specify a count, all the text that had been inserted will be repeated that many times.
i	Enter insert mode, the characters will be inserted before the current cursor position. If you specify a count, all the text that had been inserted will be repeated that many times.
A	Append at the end of the current line.
I	Insert from the beginning of a line.

Replace text

r	Replace one character under the cursor. Specify a count to replace a number of characters.
R	Replace characters on the screen with a set of characters entered, ending with the Escape key.
s	Substitute one character under the cursor, and go into insert mode. Specify a count to substitute a number of characters. A dollar sign ($) will be put at the last character to be substituted.
S	Change an entire line

Open a line

o	Enter insert mode in a new line below the current cursor position.
O	Enter insert mode in a new line above the current cursor position.

- **Saving and quitting**

:w	To save and continue editing work. Save and resume.
:x	To save and quit vi. Same as :wq. Needs one less keystroke.
:wq	To save and quit vi. Same as :x. Needs one extra keystroke.
ZZ	To save and quits vi.
:q	To quit vi. If your file has been modified, the editor will warn you of this, and not allow you to quit. To ignore this message, the command to quit out of vi without saving is **:q!**. This allows you to quit vi without saving any of the changes.

- **Using buffers (cut-copy-paste)**

d	d is not a command but it is used as dd for deletion of line.
y	y is not a command but it is used as yy for yanking (copying) line(s).
d^	deletes from current cursor position to the beginning of the line.
d$	deletes from current cursor position to the end of the line.
dd	Deletes the current line. It takes repeat factor. E.g. 3dd deletes three lines from current cursor position downwards.
yy	It is used to yank (copy) one or more lines. **yy** yanks (copy) current line. 10yy yanks current line and 9 lines below.

Moving (Pasting)

p	p pastes the specified or general buffer after the cursor position. Specifying count before the paste command pastes text the specified number of times.
P	P pastes the specified or general buffer before the cursor position. Specifying count before the paste command pastes text the specified number of times.

Joining

J	Used to join current line and the line following current line. **4J** joins following 3 lines with current line.

- **Pattern searching and replacement**

Search for a pattern (/ and ?)

The vi editor has two kinds of **searches: string and character**. For a string search, the / and ? commands are used.

The **/** command searches **forwards** (downwards) in the file, while the **?** command searches **backwards** (upwards) in the file.

To locate the first occurrence of string UNIX, just enter
/UNIX [Enter]

When you press /, it is echoed in the last line of the screen. Enter the pattern to be searched after /, and then press [Enter].

The cursor is then positioned on the first character of the first occurrence of this pattern. Note that search will start from the present location of the cursor.

By default, the search **wraps around** the end of the text and resume from the beginning of the file if the pattern can't be located up to the end of the file.

The / and ? differ only in the direction where the search takes place. To search in the **reverse direction**, we need to use ? instead of /.

?UNIX [Enter]

Searches most previous occurrence of UNIX i.e. it searches in backward direction.

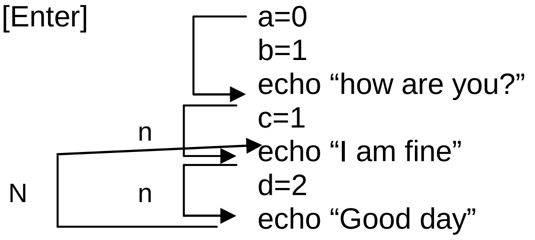
Repeating the last pattern search (n and N)

For repeating searches vi uses **n and N** commands.

For repeating a search in the same direction in which previous search is made with / or ?, then use **n**.

For repeating a search in the reverse direction than the previous search, then use **N**.

/echo [Enter]



```
a=0
b=1
echo "how are you?"
c=1
echo "I am fine"
d=2
echo "Good day"
```

Substitution - search and replace (:s)

It allows to replace a pattern in the file with some another pattern.

SYNTAX: :address/source_pattern/target_pattern/flags

The source_pattern is replaced with target_pattern in all lines specified by address. The address can be one or a pair of numbers separated by comma.

For e.g. **1,$** addresses all lines in a file.

Most commonly used flag is g which carries out the substitution for all occurrences of pattern in a line For e.g

:1,$s/director/member/g will replace all director with member globally throughout the file.

If the pattern can not be found, vi gives message "substitute pattern match failed". If you leave out g, then the substitution will be carried out for the first occurrence in each addressed line.

The taget pattern is optional. If you leave out, then you will delete all instances of the source pattern in all lines matched by the address. For e.g.

:1,50s/unsigned//g deletes "unsigned" everywhere in lines 1 to 50.

:3,10s/director/member/g	substitute lines 3 through 1
:.s/director/member/g	only the current lin
:$s/director/member/g	Only the last line

:1,$s/director/member/gc

Each line is selected in turn followed by a sequence of carets in the next line just below the pattern that requires substitution. Cursor is positioned at the end of this caret sequence waiting for your response.

```
1034|somu sengupta   |director   |sales        |12/13/65|5000
                      ^^^^^^^^n
1234|lalit chowdhary |director   |marketing    |09/26/72|6000
                      ^^^^^^^^y
```

Only y performs substitution.

- **Undoing and Repeating commands**

Undoing

u	undo the last change to the file. Typing u again will re-do the change.
U	Remove all changes have been made to a current line.

Repeating

Vi have the facility to repeat the last editing instruction. This repetition applies to commands of Input and Command mode.

To repeat the command at other places, we have to use a simple **. (dot)** command.

BASIC COMMANDS RELATED TO HANDLING FILES AND THE FILE SYSTEM

1. pwd (present working directory)

pwd is used to display the working directory.

SYNTAX: pwd

We can move around from one directory to another. But at any point of time, we are located in only one directory. This directory is known as **current directory**.

The pwd command displays the absolute pathname of the current (working) directory. i.e. this pathname shows your location with reference to the top which is root.

The pwd command requires no options or arguments.

```
$ pwd
/home/ty100
```

/ is root and also it is parent of home directory. home is parent directory of ty100.

2. cd (Changing the working directory)

cd command is used to change the working directory. If we specify pathname of the new working directory as an argument then cd command will change the current directory to the specified directory.

For example, to change the working directory to the /temp directory, type:

```
$ cd /temp
$ pwd
$ /home/ty100/temp
```

If you attempt to change the working directory to a directory that doesn't exist, Linux displays an error message:

```
$ cd nowhere
bash: nowhere: No such file or directory
```

Use of .. with cd

To move to parent directory, we can use .. (two dots) with cd.

```
$ pwd
/home/ty100/scripts
$ cd ..                 Note: Moves one level up
$ pwd
/home/ty100
```

```
$ pwd
/home/ty100/scripts
$ cd .. / ..            Note: Moves two level up
$ pwd
/home
```

This method is compact and more useful for moving in hierarchy. We can combine any number of sets of **.. (two dots)** separated by /.

If we want to move to a directory with same parent then we can make use of **.. (two dots)** to frame relative pathname.

```
$ pwd
/home/ty100
$ cd ../ty200       Note: One level up and then down
$ pwd
/home/ty200
```

3. mkdir (Making Directories)

Directories can be created with the **mkdir** (make directory) command. The command is followed by the names of the directories to be created.

SYNTAX: **mkdir** [option] directory...

e.g. **$ mkdir student** Note: Create student directory in current directory.

mkdir can take multiple arguments. We can create number of subdirectories with one mkdir commands.

$ mkdir ty100 ty200 ty300 Note: 3 subdirectories created.

We can create a directory tree with single mkdir command.

$ mkdir ty ty/ty100 ty/ty200 Note: create directory tree

This command will create total 3 directories - 1 directory ty and two subdirectories ty100 and ty200 under ty.

Order of specifying argument is important. We can't create a subdirectory before creation of its parent directory.

-p option with mkdir allows us to create multiple generations of directories with one command. It creates all the parent directories specified in given path.

$ mkdir –p stud/ty/ty100/scripts

The –p option will first create stud, then within it ty, next its child directory ty100 and lastly scripts within ty100.

4. rmdir (Removing Directories)

Directories can be removed (deleted) with the **rmdir** (remove directory) command. The command is followed by the names of the directories to be removed.

SYNTAX: **rmdir** DIRECTORY...

e.g. **$ rmdir student** Note: Remove student directory from current directory.

Like **mkdir**, **rmdir** can also delete more than one directory in one shot.

$ rmdir ty100 ty200 ty300 Note: 3 subdirectories deleted.

We can delete a directory tree with single rmdir command. Subdirectories are specified before parent directory while using rmdir command.

$ rmdir ty/ty100 ty/ty200 ty Note: delete directory tree

This command will delete total 3 directories - 1 directory ty and two subdirectories ty100 and ty200 under ty.

The following sequence is invalid in rmdir.

```
$ rmdir ty ty/ty100 ty/ty200
rmdir: ty: Directory not empty
```

There are 2 important rules to be kept in mind when deleting directories:

1) We can't use rmdir to delete a directory unless it is empty. In above example, ty could not be deleted because of two subdirectories exist for ty - ty100 and ty200.
2) We can't remove a subdirectory unless you are there in a directory which is hierarchically above than the directory to be removed. (This restriction doesn't apply in Linux).

```
$ cd ty100
$ pwd
/home/ty/ty100
$ rmdir .                Note: Trying to remove current directory
rmdir: .: Can't remove current directory or ..
```

To remove this directory, you must position yourself in the directory above ty100.

UNIX **rm** command can be used with a special option –**r** to remove a complete directory structure.

5. cp (Copying files)

The cp command can be used to copy a file or group of files. It creates an exact image of the file on disk with a different name.

The syntax requires at least two file names in the command line.

To copy file1 to file2 under the current directory:

```
$ cp file1 file2
```

If the destination file (target file) i.e. file2 does not exist, it will be created first. Then copying process will takes place.

If file2 already exists, it will be simply overwritten with the contents of the source file i.e. file1. No warning will be given from the system. So be careful while choosing destination file name.

We can copy multiple files into the directory using single cp command. Here, the last file name must be a directory.

```
$ cp file1 file2 file3 scripts    Note: it will copy file1, file2 and file3 in scripts directory.
$ cp file* scripts          Note: it will copy all files starting with word file into scripts directory.
```

Interactive copying (-i)

The –i (interactive) option warns the user before overwriting the destination file. If file2 exists, cp command prompts for a response:

```
$ cp –i file1 file2
cp: overwrite file2? y
```

A **y** at this prompt will overwrite the file. Any other response will not allow copy.

Copying directory structure (-r)

It is possible to copy entire directory structure with –r (recursive) option. The following command will copy all files and subdirectories of scripts to newscripts.

```
$ cp –r scripts newscripts
```

6. mv (Renaming files)

mv renames (moves) files and directories. It doesn't create a copy of the file.

It means that additional space is not consumed in the disk by using mv.

It has two functions:

- Renames a file (or directory)
- Moves a group of files to a different directory

To rename the file file1 to script1:
$ mv file1 script1

If the destination file (script1) doesn't exist then it will be created. By default, mv doesn't prompt for overwriting the destination file if it exists.

A group of files can be moved. But it can be moved only to a directory.

$ mv file1 file2 file3 scripts Note: scripts must be a directory

mv can be used to **rename a directory**.

Interactive renaming (-i)

The –i (interactive) option warns the user before overwriting the destination file. If file2 exists, mv command prompts for a response:

$ mv –i file1 file2
mv: overwrite file2? **y**

A **y** at this prompt will overwrite the file. Any other response will not allow renaming.

7. rm (Deleting files)

The rm command deletes files and makes space available on disk. It normally operates silently. So it should be used carefully.

It can delete more than one file with a single command.

$ rm file1 file2 file3 Note: it will delete 3 files- file1, file2 and file3

$ rm file* Note: it will delete all files starting with word file.

We can remove two files from same subdirectory without using cd.

$ rm scripts/file1 scripts/file2 Note: or use **rm scripts/file[12]**

To delete all files from a directory:

$ rm * Note: all files deleted.

When you delete files in this way, system will not prompt with any kind of message. The $ prompt will return silently.

Interactive deletion (-i)

The –i (interactive) option warns the user before deleting the file. rm with -i command prompts for a response:

$ rm –i file1 file2 file2
file1: ? **y**
file2: ? **n**
file3: ? **y**

A **y** at this prompt will remove (delete) the file. Any other response will not allow deletion.

Recursive (and Dangerous) Deletion (-r)

rm –r performs a tree walk – a recursive search for all subdirectories and files within these subdirectories. Then it deletes all of them.

Normally rm won't remove directories but with –r option it will remove directories.

$ rm –r * Note: Current directory tree will be completely removed.

Disclaimer

The study material is compiled by Ami D. Trivedi. The basic objective of this material is to supplement teaching and discussion in the classroom in the subject. Students are required to go for extra reading in the subject through library work.