## Contents



# 5 Unit V

## 5.1 Introduction to Unix/Linux operating systems

All operating systems provide services for programs they run. Typical services include executing a new program, opening a file, reading a file, allocating a region of memory, getting the current time of day, and so on.

Unix is a very popular multi-user, multitasking, time-sharing operating system.

Unix has become the operating system of choice for various engineering and scientific applications. The need for Unix could be determined by the different categories of application it suffices. viz. networking, programming, multimedia, high-performance computing to name a few.

Examples of modern UNIX operating systems include **IRIX**(from SGI), **Solaris** (from Sun), **Tru64** (from Compaq) and **Linux** (from the Free Software community). Even though these different "flavors" of UNIX have unique characteristics and come from different sources, they all work alike in a number of fundamental ways.

Evolution of UNIX:

- 1965 : AT&T, GE, IBM and Project MAC join together to develop a time-sharing system named MULTICS (Multiplexed Information and Computing Service).
- 1969 : AT&T Bell Labs drops out of MULTICS project. Ken Thompson writes first version of UNICS on a PDP-7 machine. UNICS is pun on MULTICS and stands for Uniplexed Information and Computing Services. UNICS is changed to UNIX.
- 1973 : Re-written in high level language C, thus making it machine-independent.
- 1977-1981 : Unix was widely available at minimal cost and became popular for scientific applications.
- 1982 : Unix System III is released.
- 1984-85 : Unix System V is released.
- 1992-93 : 4.4 BSD is released.
- 1994 : Linux 1.0 is released.
- 2001 : Linux 2.4 is released.

### 5.1.1 Unix is available in many flavors like:

- AIX (Advanced IBM Unix)
- HP-UX ( Hewlett Packard Unix)
- MINIX (Minimal Unix)
- SCO UNIX
- SOLARIS
- XENIX
- SUN OS
- LINUX

### 5.1.2 Features Of Unix

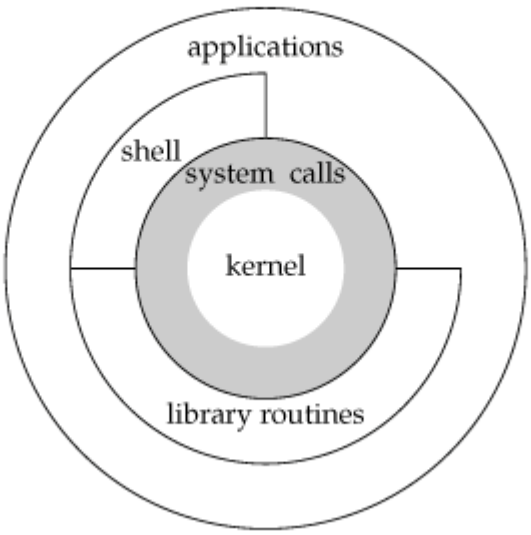
- Portability : As Unix has been re-written in C , hence it can run on machines from microcomputers to mainframe computers.
- Security : Unix is a very secure operating system. Without the proper username and passwords, it is not possible to work on Unix.
- Background Processing : Unix employs a technique of Background Processing, wherein jobs/tasks are executed in the background with the minimal interaction from the user.

- Pipes : Using the concept of pipes, a Unix user/administrator can easily link and work with multiple commands at the same time.
- Redirection Tools : These tools allow data to be re-directed between files as per the requirement of the user.
- Software Development Tools : Unix supports a wide variety of tools using which the user could create and work on different programs .viz. C Language on the Unix Operating system. Unix supports any language that has an interpreter or compiler.
- Stable and Reliable : Unix is a very reliable and stable Operating System. It is less prone to crashes.
- Easy to write programs : Its easy to write programs in Unix as it hides machine architecture from the user.
- Hierarchical File System : Unix employs a hierarchical file system which is easy to implement and maintain.
- Shells : Unix has different types of shells .viz. Bourne, C, Korn, etc.
- Communication : Unix has commands which allow communication between different users connected to the system.

### 5.1.3 UNIX Architecture

In a strict sense, an operating system can be defined as the software that controls the hardware resources of the computer and provides an environment under which programs can run. Generally, we call this software the kernel, since it is relatively small and resides at the core of the environment. Figure 1.1 shows a diagram of the UNIX System architecture.

**Figure 1.1. Architecture of the UNIX operating system**




The interface to the kernel is a layer of software called the system calls (the shaded portion in Figure 1.1). Libraries of common functions are built on top of the system call interface, but applications are free to use both. The shell is a special application that provides an interface for running other applications.

In a broad sense, an operating system is the kernel and all the other software that makes a computer useful and gives the computer its personality. This other software includes system utilities, applications, shells, libraries of common functions, and so on.
For example, Linux is the kernel used by the GNU operating system. Some people refer to this as the GNU/Linux operating system, but it is more commonly referred to as simply Linux. Although this usage may not be correct in a strict sense, it is understandable, given the dual meaning of the phrase operating system. (It also has the advantage of being more succinct.)

### 5.1.4 UNIX fundamentals

UNIX has been around for a long time (over 30 years). It predates the concept of the personal computer. As such, it was designed from the ground up to be a multi-user, shared, networked operating environment. UNIX has concepts such as **Users**, **Groups**, **Permissions** and **Network-Shared Resources** (such as files, printers, other computer systems, etc.) built-in to the core of its design. This makes UNIX a uniquely powerful and flexible operating system. Along with this power and flexibility comes some unique concepts that make UNIX what it is. These concepts are relatively simple and should be understood to take full advantage of the operating system.

- **Users** - In order to make use of a UNIX system, you must first log in. This requires a **user account**, which consists of:
    - **Username:**
      This is your login name and is how you are identified to the system itself and to other users of the system.
    - **Password:**
      Along with your username, your password grants you access to the system. Don't forget or lose your password. If you write your password down, keep it in a safe place.
    - **Default group:**
      The default group that your username belongs to (see **Groups** below).
    - **Contact info:**
      So that system administrators and other users can contact you if necessary.
    - **Home directory:**
      A directory or "folder" assigned to your username. This grants you access to disk storage. This is where you will keep your files and data.

- **Default shell:** The program which manages your login and command line sessions (covered in detail later)

- **Groups** - A UNIX group is a collection of users - i.e. a list of usernames. Groups provide a mechanism to assign **permissions** (see below) to a list of users all at once. For our purposes, group associations are typically based on which research group or area of study a user is affiliated with. Each user can belong to more than one group.
- **Permissions** - Everything in UNIX is "owned" by both a user and a group. The simplest example of this would be files (but this concept is not limited only to files). By manipulating permissions, the user who owns a file can define which other users and groups can read or modify that file. In this way, users can secure sensitive files from prying eyes and keep themselves or others from accidentially deleting important data.
- **Shared Resources** - UNIX is a networked operating environment at its core. As such, nearly everything that one can access on the local system can also be accessed via the network from remote systems. This includes, among other possibilities, editing and sharing files, running software, or using printers. Even the contents of a UNIX system's display can be manipulated remotely.

The actions that an individual user is able to perform remotely is defined by the permissions assigned to that user (or any group to which the user belongs) for each of these activities. Some of these things will be discussed in greater detail later on.

## 5.2 Concept of file system

**Files and Directories/ File System**

The UNIX file system is a hierarchical arrangement of directories and files. Everything starts in the directory called root whose name is the single character /.

A directory is a file that contains directory entries. Logically, we can think of each directory entry as containing a filename along with a structure of information describing the attributes of the file.

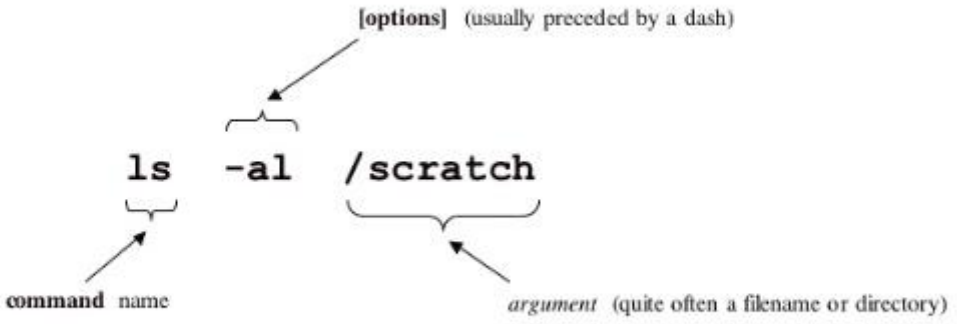
The attributes of a file are such things as type of file: *regular file* or *directory*; the size of the file; the owner of the file; permissions for the file; whether other users may access this file and when the file was last modified. The *stat* and *fstat* functions return a structure of information containing all the attributes of a file.

- **Filename**: The names in a directory are called filenames. The only two characters that cannot appear in a filename are the slash character (/) and the null character. The slash separates the filenames that form a pathname (described next) and the null character terminates a pathname. Nevertheless, it's good practice to restrict the characters in a filename to a subset of the normal printing characters. (We restrict the characters because if we use some of the shell's special characters in the filename, we have to use the shell's quoting mechanism to reference the filename, and this can get complicated.)
- Two filenames are automatically created whenever a new directory is created: **.** (called dot) and **..** (called dot-dot). Dot refers to the current directory, and dot-dot refers to the parent directory. In the root directory, dot-dot is the same as dot.
- Almost all commercial UNIX file systems support at least **255-character filenames**.
- **Pathname**: A sequence of one or more filenames separated by slashes and optionally starting with a slash, forms a pathname. A pathname that begins with a slash is called an absolute pathname; otherwise, it's called a relative pathname. Relative pathnames refer to files relative to the current directory. The name for the root of the file system (/) is a special-case absolute pathname that has no filename component.
- **Working Directory**: Every process has a working directory, sometimes called the current working directory. This is the directory from which all relative pathnames are interpreted. A process can change its working directory with the *chdir* function.
  For example, the relative pathname doc/memo/joe refers to the file or directory *joe*, in the directory memo, in the directory doc, which must be a directory within the working directory. From looking just at this pathname, we know that both doc and memo have to be directories, but we can't tell whether joe is a file or a directory. The pathname /usr/lib/lint is an absolute pathname that refers to the file or directory lint in the directory lib, in the directory usr, which is in the root directory.
- **Home Directory**: When we log in, the working directory is set to our home directory. Our home directory is obtained from our entry in the password file.

## 5.3 Directory and File Handling Commands

### 5.3.1 Anatomy of a UNIX Command

UNIX commands are executed by typing them at the prompt and pressing enter. The figure below illustrates the anatomy of a simple unix command.




This section describes some of the more important directory and file handling commands.

- pwd (print [current] working directory)

pwd displays the full absolute path to the your current location in the filesystem. So

$ pwd ↵
/usr/bin

implies that /usr/bin is the current working directory.

- ls (list directory)

ls lists the contents of a directory. If no target directory is given, then the contents of the current working directory are displayed. So, if the current working directory is /,

$ ls ↵
bin dev home mnt share usr var
boot etc lib proc sbin tmp vol

Actually, ls doesn't show you *all* the entries in a directory - files and directories that begin with a dot (.) are hidden (this includes the directories '.' and '..' which are always present). The reason for this is that files that begin with a . usually contain important configuration information and should not be changed under normal circumstances. If you want to see all files, ls supports the -a option:

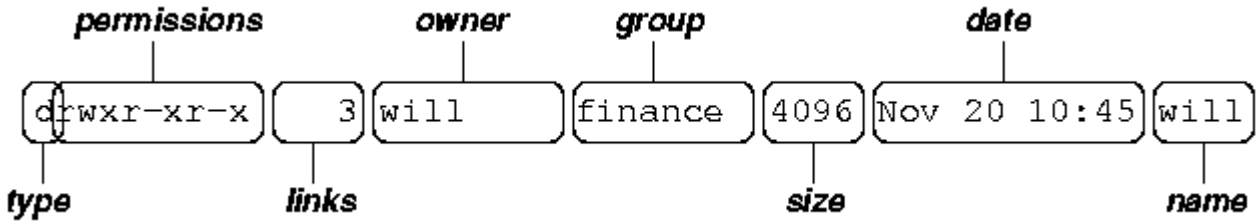
$ ls -a ↵

Even this listing is not that helpful - there are no hints to properties such as the size, type and ownership of files, just their names. To see more detailed information, use the -l option (long listing), which can be combined with the -a option as follows:

$ ls -a -l ↵
(or, equivalently,)

$ ls -al ↵

Each line of the output looks like this:




where:

- *type* is a single character which is either 'd' (directory), '-' (ordinary file), 'l' (symbolic link), 'b' (block-oriented device) or 'c' (character-oriented device).
- *permissions* is a set of characters describing access rights. There are 9 permission characters, describing 3 access types given to 3 user categories. The three access types are read ('r'), write ('w') and execute ('x'), and the three users categories are the user who owns the file, users in the group that the file belongs to and other users (the general public). An 'r', 'w' or 'x' character means the corresponding permission is present; a '-' means it is absent.
- *links* refers to the number of filesystem links pointing to the file/directory (see the discussion on hard/soft links in the next section).
- *owner* is usually the user who created the file or directory.
- *group* denotes a collection of users who are allowed to access the file according to the group access rights specified in the permissions field.
- *size* is the length of a file, or the number of bytes used by the operating system to store the list of files in a directory.
- *date* is the date when the file or directory was last modified (written to). The -u option display the time when the file was last accessed (read).
- *name* is the name of the file or directory.

ls supports more options. To find out what they are, type:

$ man ls ↵

man is the online UNIX user manual, and you can use it to get help with commands and find out about what options are supported. It has quite a terse style which is often not that helpful, so some users prefer to the use the (non-standard) info utility if it is installed:

$ info ls ↵

- cd (change [current working] directory)

$ cd *path*

changes your current working directory to *path* (which can be an absolute or a relative path). One of the most common relative paths to use is '..' (i.e. the parent directory of the current directory).

Used without any target directory

$ cd ↵

resets your current working directory to your home directory (useful if you get lost). If you change into a directory and you subsequently want to return to your original directory, use

$ cd - ↵

- mkdir (make directory)

$ mkdir *directory*

creates a subdirectory called *directory*in the current working directory. You can only create subdirectories in a directory if you have write permission on that directory.

- rmdir (remove directory)
- $ rmdir *directory*

removes the subdirectory *directory* from the current working directory. You can only remove subdirectories if they are completely empty (i.e. of all entries besides the '.' and '..' directories).

- cp (copy)

cp is used to make copies of files or entire directories. To copy files, use:

$ cp *source-file(s) destination*

where *source-file(s)* and *destination* specify the source and destination of the copy respectively. The behavior of cp depends on whether the destination is a file or a directory. If the destination is a file, only one source file is allowed and cp makes a new file called *destination* that has the same contents as the source file. If the destination is a directory, many source files can be specified, each of which will be copied into the destination directory. Section 2.6 will discuss efficient specification of source files using wildcard characters.

To copy entire directories (including their contents), use a *recursive* copy:

$ cp -rd *source-directories destination-directory*

- mv (move/rename)

mv is used to rename files/directories and/or move them from one directory into another. Exactly one source and one destination must be specified:

$ mv *source destination*

If *destination* is an existing directory, the new name for *source* (whether it be a file or a directory) will be *destination/source*. If *source* and *destination*are both files, *source* is renamed *destination*. N.B.: if *destination* is an existing file it will be destroyed and overwritten by *source* (you can use the -ioption if you would like to be asked for confirmation before a file is overwritten in this way).

- rm (remove/delete)

$ rm *target-file(s)*

removes the specified files. Unlike other operating systems, it is almost impossible to recover a deleted file unless you have a backup (there is no recycle bin!) so use this command with care. If you would like to be asked before files are deleted, use the -i option:

$ rm -i myfile ↵

rm: remove 'myfile'?

rm can also be used to delete directories (along with all of their contents, including any subdirectories they contain). To do this, use the -r option. To avoidrm from asking any questions or giving errors (e.g. if the file doesn't exist) you used the -f (force) option. Extreme care needs to be taken when using this option - consider what would happen if a system administrator was trying to delete user will's home directory and accidentally typed:

$ rm -rf / home/will ↵

(instead of rm -rf /home/will).

- **cat** (catenate/type)

$ cat *target-file(s)*

displays the contents of *target-file(s)* on the screen, one after the other. You can also use it to create files from keyboard input as follows (> is the output redirection operator, which will be discussed in the next chapter):

$ cat > hello.txt ↵

hello world! ↵

[ctrl-d]

$ ls hello.txt ↵

hello.txt

$ cat hello.txt ↵

hello world!
$

- **more and less** (catenate with pause)

$ more *target-file(s)*

displays the contents of *target-file(s)* on the screen, pausing at the end of each screenful and asking the user to press a key (useful for long files). It also incorporates a searching facility (press '/' and then type a phrase that you want to look for).

You can also use more to break up the output of commands that produce more than one screenful of output as follows (| is the pipe operator, which will be discussed in the next chapter):

$ ls -l | more ↵

less is just like more, except that has a few extra features (such as allowing users to scroll backwards and forwards through the displayed file). less not a standard utility, however and may not be present on all UNIX systems.

- Specifying multiple filenames

Multiple filenames can be specified using special pattern-matching characters. The rules are:

- '?' matches any single character in that position in the filename.
- '*' matches zero or more characters in the filename. A '*' on its own will match all files. '*.*' matches all files with containing a '.'.
- Characters enclosed in square brackets ('[' and ']') will match any filename that has one of those characters in that position.
- A list of comma separated strings enclosed in curly braces ("{" and "}") will be expanded as a Cartesian product with the surrounding characters.

For example:

1. ??? matches all three-character filenames.
2. ?ell? matches any five-character filenames with 'ell' in the middle.
3. he* matches any filename beginning with 'he'.
4. [m-z]*[a-l] matches any filename that begins with a letter from 'm' to 'z' and ends in a letter from 'a' to 'l'.
5. {/usr,}{/bin,/lib}/file expands to /usr/bin/file /usr/lib/file /bin/file and /lib/file.

Note that the UNIX shell performs these expansions (including any filename matching) on a command's arguments *before* the command is executed.

## 5.4 Concept of Shells

A shell is a command-line interpreter that reads user input and executes commands. Once we log in, some system information messages are typically displayed, and then we can type commands to the shell program. The user input to a shell is normally from the terminal (an interactive shell) or sometimes from a file (called a shell script). The common shells in use are summarized in Figure 1.2.

**Figure 1.2. Common shells used on UNIX systems**

| Name | Path | FreeBSD 5.2.1 | Linux 2.4.22 | Mac OS X 10.3 | Solaris 9 |
|---|---|---|---|---|---|
| Bourne shell | /bin/sh | • | link to bash | link to bash | • |
| Bourne-again shell | /bin/bash | optional | • | • | • |
| C shell | /bin/csh | link to tcsh | link to tcsh | link to tcsh | • |
| Korn shell | /bin/ksh | | | | • |
| TENEX C shell | /bin/tcsh | • | • | • | • |

The **Bourne shell**, developed by Steve Bourne at Bell Labs, has been in use since Version 7 and is provided with almost every UNIX system in existence. The control-flow constructs of the Bourne shell are reminiscent of Algol 68.

The **C shell**, developed by Bill Joy at Berkeley, is provided with all the BSD releases. Additionally, the C shell was provided by AT&T with System V/386 Release 3.2 and is also in System V Release 4 (SVR4). (We'll have more to say about these different versions of the UNIX System in the next chapter.) The C shell was built on the 6th Edition shell, not the Bourne shell. Its control flow looks more like the C language, and it supports additional features that weren't provided by the Bourne shell: job control, a history mechanism, and command line editing.

The **Korn shell** is considered a successor to the Bourne shell and was first provided with SVR4. The Korn shell, developed by David Korn at Bell Labs, runs on most UNIX systems, but before SVR4 was usually an extra-cost add-on, so it is not as widespread as the other two shells. It is upward compatible with the Bourne shell and includes those features that made the C shell popular: job control, command line editing, and so on.

The **Bourne-again shell** is the GNU shell provided with all Linux systems. It was designed to be POSIX-conformant, while still remaining compatible with the Bourne shell. It supports features from both the C shell and the Korn shell.

The **TENEX C shell** is an enhanced version of the C shell. It borrows several features, such as command completion, from the TENEX operating system

(developed in 1972 at Bolt Beranek and Newman). The TENEX C shell adds many features to the C shell and is often used as a replacement for the C shell.
Linux uses the Bourne-again shell for its default shell. In fact, /bin/sh is a link to /bin/bash. The default user shell in FreeBSD and Mac OS X is the TENEX C shell, but they use the Bourne shell for their administrative shell scripts because the C shell's programming language is notoriously difficult to use. Solaris, having its heritage in both BSD and System V, provides all the shells shown in Figure 1.2. Free ports of most of the shells are available on the Internet.
Throughout the text, we will use parenthetical notes such as this to describe historical notes and to compare different implementations of the UNIX System. Often the reason for a particular implementation technique becomes clear when the historical reasons are described.

## 5.5 vi Editor

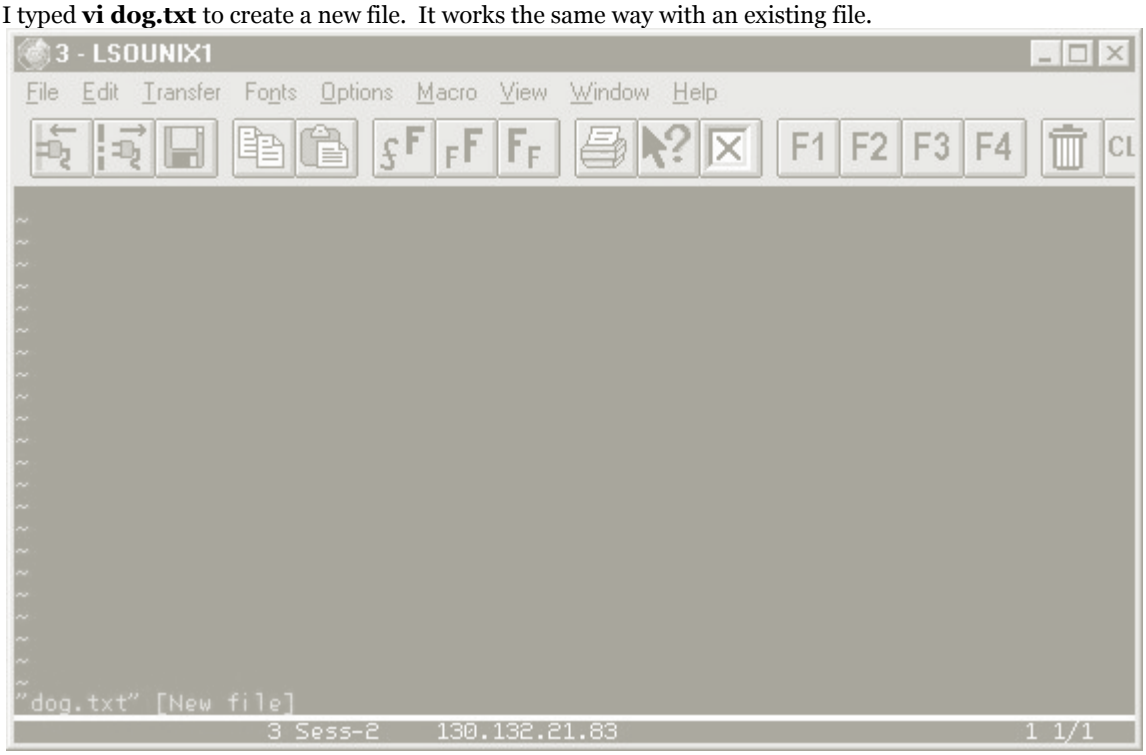
- Introduction

The default editor that comes with the UNIX operating system is called vi (**vi**sual editor). [Alternate editors for UNIX environments include pico and emacs, a product of GNU.]

- Working

**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 **start**, type **vi *filename*** at the command line, and you'll see a screen like this:
I typed **vi dog.txt** to create a new file. It works the same way with an existing file.




### ➢ Editing Modes

The most important thing to know about Vi (and the most confusing) is that it has two modes, **Command Mode**and **Insert Mode**.

In **Command Mode**, you can **invoke editing commands**, **move the cursor**, **save** or **exit the file**, **invoke the shell**, or **enter Insert Mode**.

In **Insert Mode**, you can **insert new text**.

By default, vi **starts in Command Mode**.

### ➢ Insert Mode

Pretty straightforward--in **Insert Mode**, you can insert text. Use the **Backspace** key to correct errors as you type.

Before you can **do** anything in **Insert Mode**, you have to **get there,** and there are **several ways** to do that:

- **a** append new text after cursor
- **A** append new text at end of line
- **c** begin change operation
- **C** change to end of line
- **i** insert new text before cursor
- **I** insert new text at beginning of line
- **o** open a new line below current line
- **O** open a new line above current line
- **R** begin overwriting text

- **s** substitute a character
- **S** substitute entire line

To **get back to Command Mode**, hit the **Esc** key.

**Command Mode**

Command mode is a little more complex, other than **entering Insert Mode**, you can **Move, Delete, Search, Change** and **Save**.

**Commands can be applied to multiple objects**. For example, **dd** deletes the current line. If I type **5dd**, it deletes 5 lines.

Changing Text

- **cw** change word
- **cc** change line
- **r** replace character
- **R** replace text beginning at cursor

Deleting Text

- **dd** delete current line
- **D** delete the remainder of the line
- **dw** delete word
- **dG** delete to the end of the file
- **x** delete current cursor position
- **X** delete back one character

Copying and Moving Text

- **yy** copy (yank) current line
- **ye** copy to end of word
- **p** paste yanked text (deleted text can also be pasted)

Cursor Movement in Command Mode

- by Character
    - **h** move one character left
    - **l** move one character right (yes, letter **l** moves you **right**!)
    - **j** move one character down
    - **k** move one character up
    - **Backspace** move back one character
- by Line
    - **0** (zero) move to the beginning of a line
    - **$** move to the end of a line
    - **Return** first character of next line
- by Word
    - **w** forward by word
    - **b** backward by word
    - **e** end of word
- by Screen
    - **CTRL-f** forward one screen

- **CTRL-b** backward one screen

Searching

- **/*text*** search forward for *text*
- **?*text*** search backward for *text*
- **n** repeat previous search
- **N** repeat previous search in opposite direction

Undoing Changes and Recovery (see also section on Saving and Exiting)

- **u** undo last change
- **U** restore current line

Saving and Exiting

- **ZZ** quit vi and write the file if changes were made
- **:w** write file
- **:w *file*** save a copy to *file*
- **:q** quit file
- **:q!** quit file and discard edits
- **:e!** return to version of current file at time of last write (this command and the one previous are useful if you make a serious mistake)

**Note** that when you use a command that **begins with a colon**, or you do a search with **/** or **?,** the cursor jumps to a command line as shown in the following screen:
After editing this document, I hit **Esc** to go back into **Command Mode** and then I typed **:w**
Now, when I hit **Return,** my changes will be written to **dog.txt**.

**Tip**: **If you forget** which mode you are in, hit **Esc** to make sure you are in **Command Mode**--You can always re-enter **Insert Mode**, and you won't mess up your typing.

# Quick reference for different commands used in vi editor

**Inserting and typing text:**

```
i           insert text (and enter input
mode)
$a          append text (to end of line)
ESC         re-enter command mode
J           join lines
```

**Cursor movement:**

```
h           left
j           down
k           up
l           right
^           beginning of line
$           end of line
1 G         top of document
G           end of document
<n> G       go to line <n>
^F          page forward
^B          page backward
w           word forwards
b           word backwards
```

**Deleting and moving text:**

```
Backspace   delete character before cursor
            (only works in insert mode)
x           delete character under cursor
dw          delete word
dd          delete line (restore with p or P)
<n> dd      delete n lines
d$          delete to end of line
dG          delete to end of file
yy          yank/copy line (restore with p or
P)
<n> yy      yank/copy <n> lines
```

**Search and replace:**

```
%s/<search string>/<replace string>/g ↵
```

**Miscellaneous:**

```
u           undo
:w          save file
:wq         save file and quit
ZZ          save file and quit
:q!         quit without saving
```

## 5.6 Basic file attributes in Unix, chmod command

- **Basic File Attributes**

The UNIX file system allows the user to access other files not belonging to them and
without infringing on security. A file has a number of attributes (properties) that are
stored in the ***inode***.
There are seven attributes of all files in the current directory and they are:

1. File type and Permissions
2. Links
3. Ownership
4. Group ownership
5. File size
6. Last Modification date and time
7. File name

The file type and its permissions are associated with each file. Links indicate the number of file names maintained by the system. This does not mean that there are so many copies of the file. File is created by the owner. Every user is attached to a group owner. File size in bytes is displayed. Last modification time is the next field. If you change only the permissions or ownership of the file, the modification time remains unchanged. In the last field, it displays the file name.
***ls*** command is used to obtain a list of all filenames in the current directory.
For example,
$ ls –l
Will display →

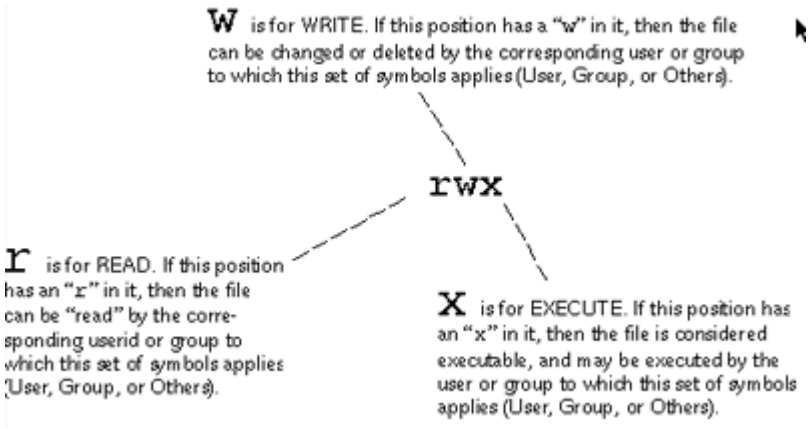
```
total 72
-rw-r--r-- 1 kumar metal 19514 may 10 13:45 chap01
-rw-r--r-- 1 kumar metal 4174 may 10 15:01 chap02
```

-rw-rw-rw- 1 kumar metal 84 feb 12 12:30 dept.lst -rw-r--r-- 1 kumar metal 9156 mar 12 1999 genie.sh
drwxr-xr-x 2 kumar metal 512 may 9 10:31 helpdir
drwxr-xr-x 2 kumar metal 512 may 9 09:57 progs

The total line tells how many blocks (usually 1024 bytes per block) are contained in this directory.

- File Permissions

UNIX follows a three-tiered file protection system that determines a file's access
rights. It is displayed in the following format:
Filetype owner (rwx) groupowner (rwx) others (rwx)
For Example:
-rwxr-xr-- 1 kumar metal 20500 may 10 19:21 chap02
r w x r - x r - - owner/user group owner others
The first group has all three permissions. The file is readable, writable and executable by the owner of the file. The second group has a hyphen in the middle slot,
which indicates the absence of write permission by the group owner of the file. The third group has the write and execute bits absent. This set of permissions is applicable to others.




## The `chmod` Command

We use the `chmod` command to **ch**ange the access **mod**e of a file. This command comes in many flavors, but we'll be talking primarily about one of them.

`chmod` *`who=permissions filename`*

This gives "**who**" the specified **permissions** for a given **filename**.

*who*

The "who" is a list of letters that specifies whom you're going to be giving permissions to. These may be specified in any order.

| Letter | Meaning |
|---|---|
| `u` | The **u**ser who owns the file (this means "you.") |
| `g` | The **g**roup the file belongs to. |
| `o` | The **o**ther users |
| `a` | **a**ll of the above (an abbreviation for `ugo`) |

*permissions*

Of course, the permissions are the same letters that you see in the directory listing:

| | |
|---|---|
| `r` | Permission to **r**ead the file. |
| `w` | Permission to **w**rite (or delete) the file. |
| `x` | Permission to e**x**ecute the file, or, in the case of a directory, search it. |

- `chmod` Examples

Let's change some of the permissions as we discussed a couple of pages ago. Here's the way our files are now:

ls –l

will give:

```
-rwxr-xr-x  joe  acctg  archive.sh
-rw-rw-r--  joe  acctg  orgchart.gif
-rw-rw-r--  joe  acctg  personnel.txt
-rw-r--r--  joe  acctg  publicity.html
drwxrwxr-x  joe  acctg  sales
-rw-r-----  joe  acctg  topsecret.inf
-rwxr-xr-x  joe  acctg  wordmatic
```

First, let's prevent outsiders from executing `archive.sh`

| | |
|---|---|
| Before: | `-rwxr-xr-x  archive.sh` |
| Command: | `chmod o=r archive.sh` |
| After: | `-rwxr-xr--  archive.sh` |

Take away all permissions for the group for `topsecret.inf` We do this by leaving the permissions part of the command empty.

| | |
|---|---|
| Before: | `-rw-r-----  topsecret.inf` |
| Command: | `chmod g= topsecret.inf` |
| After: | `-rw-------  topsecret.inf` |

Open up `publicity.html` for reading and writing by anyone.

| | |
|---|---|
| Before: | `-rw-r--r--  publicity.html` |
| Command: | `chmod og=rw publicity.html` |
| After: | `-rw-rw-rw-  publicity.html` |

Other example:

| | |
|---|---|
| Before: | `-rw-r--r--  publicity.html` |
| Command: | `chmod u=rwx, g=rx, o=x publicity.html` |
| After: | `-rwxr-x--x  publicity.html` |

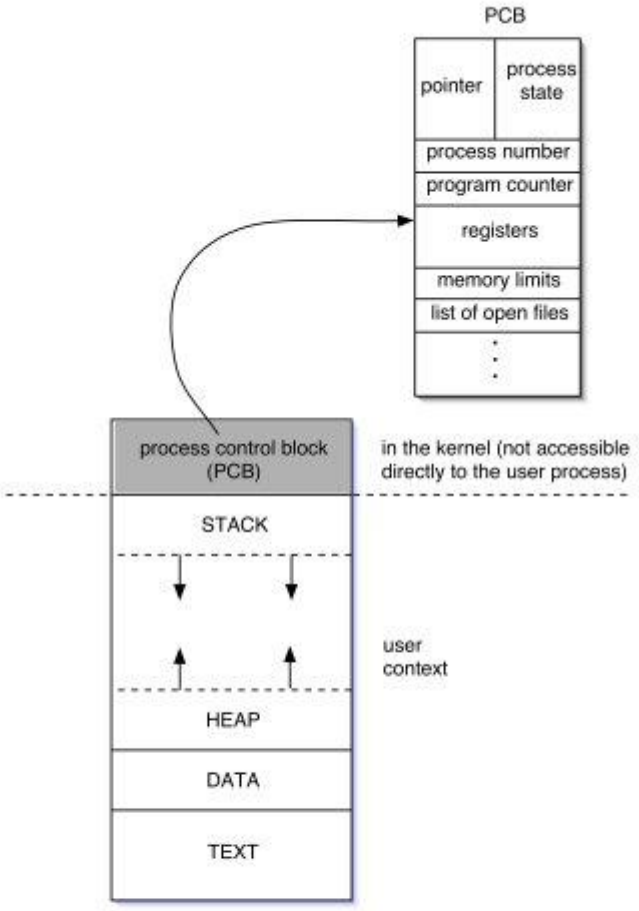
## 5.7 concept of process, working with ps command.

- UNIX Processes

While a **program** can be defined as an executable file, *a process is an instance of a program that is being executed by the operating system*. Some operating systems use the term "task" instead of process. Operating systems that are capable of executing more than one task (process) at a time are called multi-tasking systems.

The ***unix*** kernel (which is what the operating system is called) provides many "access points" through which an active process can obtain services from the kernel. These are called system calls. The standard unix C library provides a C interface to each system call; as a result, the actual system calls appear as normal C functions to the programmer.

A process consists of several components: the executable program code itself (referred to as the "code" or sometimes the "text" portion of the process), the data on which the program will execute, the resources required for the execution (such as memory workspace and access to various files), and information about the state of the process. The data portion contains items such as program variables and their values. Among the resources required for execution is memory space, which can be divided into two types: heap and stack.




The *heap* is a portion of memory allocated dynamically (as needed, at runtime) for the use of the process. Whenever the malloc or calloc functions are used in C for instance, they reserve space in heap memory. The *stack* portion of memory, on the other hand, is used by the process to aid the invocation of functions. Every time a function is called, the process reserves a portion of stack memory to store the values of parameters

passed to the functions as well as for results returned by the functions and the local variables used within the functions. The stack is also where space for all declared data types and structures is reserved at compile time.
All processes in ***unix*** exist in a hierarchy of parent-child relationships. Any process that creates or spawns another process becomes the parent of the created process. The created process itself is called the child of the creating process. A process can have multiple child processes, but a child process can have only one parent process.
Every process has a unique process ID (or PID). The PID is an integer that is assigned by the kernel when the process is created. The process with PID 0 is a special kernel process called the "swapper" (or sometimes called the "scheduler") which implements the concurrent execution of multiple processes on a single CPU as mentioned above. PID 1 is also a special process called "init" which initializes the system and makes it ready for use by users. init is considered the parent of all other processes since it creates them.

- UNIX command: ps

The **ps** command displays active processes.

The syntax for the **ps** command is:

```
ps [options]
```

### Some useful options

| | |
|---|---|
| -a | Displays all processes on a terminal, with the exception of group leaders. |
| -e | Displays all processes. |
| -f | Displays a full listing. |
| -l | Displays a long listing |
| -p*list* | Displays data for the *list* of process IDs. |
| -s*list* | Displays data for the *list* of session leader IDs. |
| -t*list* | Displays data for the *list* of terminals. |
| -u*list* | Displays data for the *list* of usernames. |

### Examples

```
ps -ef
ps -aux
```

## Examples

ps

Typing ps alone would list the current running processes. Below is an example of the output that would be generated by the ps command.

```
PID   TTY   TIME   CMD
6874  pts/9  0:00    ksh
6877  pts/9  0:01    csh
418   pts/9  0:00    csh
```

| field | description |
|---|---|
| **UID** | the user who owns the process |
| **PID** | the *process id*, a unique identifier assigned to each process |
| **PPID** | the *parent process id*, the process that spawned the current process |
| **C** | this field is obsolete |
| **STIME** | the start time for the current process |
| **TTY** | the controlling terminal for the current process |
| **TIME** | the amount of CPU time accumulated by the current process |
| **CMD** | the command used to invoke the process |

Use ***ps -ef*** to get a full listing of all processes on the system. Since there are generally many processes to be listed, you'll want to pipe the output into a pager, such as ***more***.

```
% ps -ef | more
     UID   PID  PPID  C    STIME TTY  TIME CMD
    root     0     0  0   Sep 18 ?    0:17 sched
    root     1     0  0   Sep 18 ?    0:54 /etc/init -
    root     2     0  0   Sep 18 ?    0:00 pageout
    root     3     0  0   Sep 18 ?    6:15 fsflush
    root   418     1  0   Sep 18 ?    0:00 /usr/lib/saf/sac -t 300
  daemon   156     1  0   Sep 18 ?    0:00 /usr/lib/nfs/statd
     .      .     .  .      .   .     .      .
     .      .     .  .      .   .     .      .
     .      .     .  .      .   .     .      .
```