



UNIX

Chandra Shekhar Reddy

LIST OF TOPICS

1. Introduction to Unix
2. Unix File system
3. File Management
4. Directory Management
5. File Permissions
6. User Management (Administration)
7. Unix Environment
8. IO Redirections
9. Unix Pipes and Filters
10. Process Management
11. Network Management
12. Regular Expressions

1. INTRODUCTION

UNIX is an operating system which was first developed in the 1960s, and has been under constant development ever since. By operating system, we mean the suite of programs which make the computer work. It is a stable, multi-user, multi-tasking system for servers, desktops and laptops.

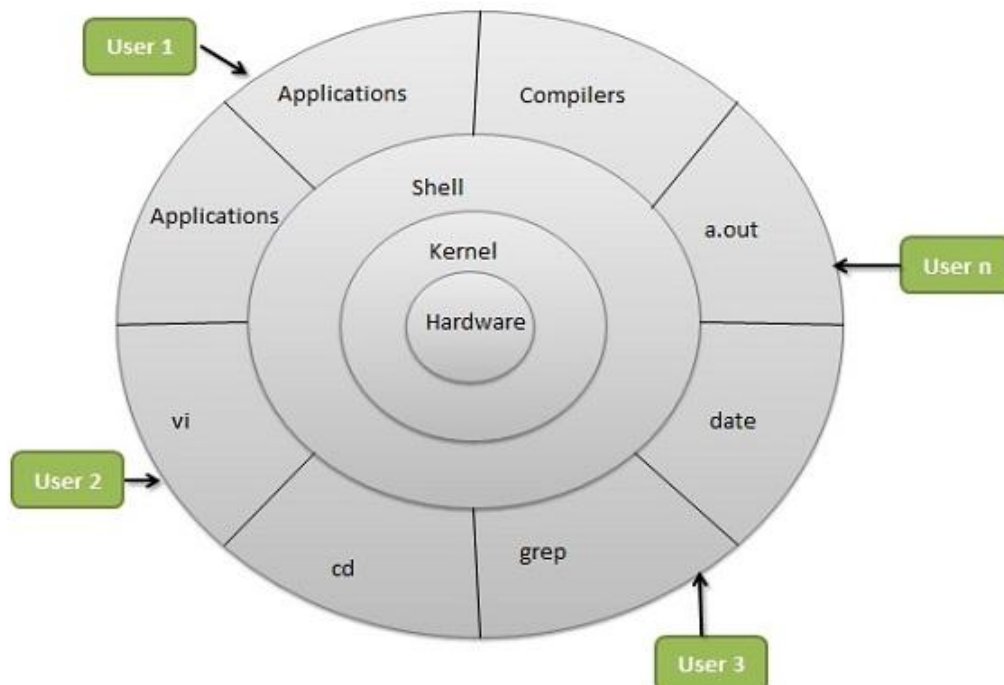
UNIX systems also have a graphical user interface (GUI) similar to Microsoft Windows which provides an easy to use environment. However, knowledge of UNIX is required for operations which aren't covered by a graphical program, or for when there is no windows interface available, for example, in a telnet session.

The UNIX operating system is a set of programs that act as a link between the computer and the user. The computer programs that allocate the system resources and coordinate all the details of the computer's internals is called the operating system or kernel. Users communicate with the kernel through a program known as the shell.

The shell is a command line interpreter; it translates commands entered by the user and converts them into a language that is understood by the kernel.

UNIX Architecture:

UNIX systems use a centralized operating system, **kernel** which manages system and process activities. All non-**kernel** software is organized into separate, **kernel**-managed processes.



UNIX Common Features:

Multitasking: Several programs running at the same time.

Multiuser: Several users on the same machine at the same time.

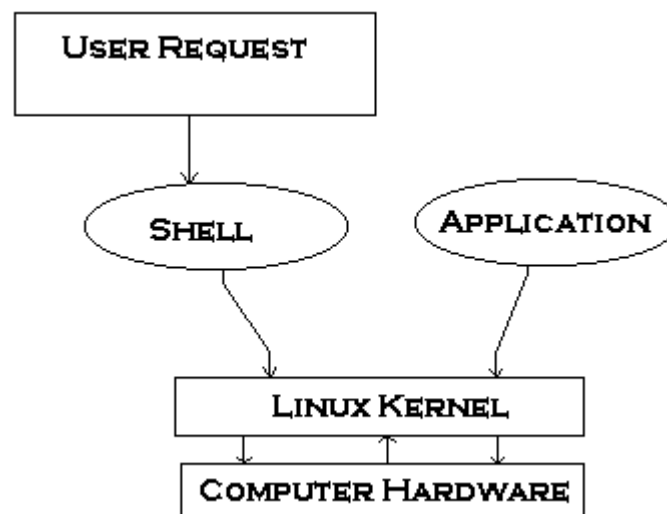
Multiplatform: Runs on many different CPUs, not just Intel.

The main concept that unites all versions of UNIX is the following four basics:

1. Kernel:

Kernel is heart of Linux OS. It manages resource of Linux OS. Resources means facilities available in Linux. For e.g. Facility to store data, print data on printer, memory, file management etc.

Kernel decides who will use this resource, for how long and when. It runs your programs (or set up to execute binary files). The kernel acts as an intermediary between the computer hardware and various programs/application/shell.



It's Memory resident portion of Linux. It performs the following tasks:-

- I/O management
- Process management
- Device management
- File management
- Memory management

2. Shell:

Computer understands the language of 0's and 1's called binary language. In early days of computing, instructions are provided using binary language, which is difficult for all of us, to read and write. So in OS there is a special program called Shell. Shell accepts your instructions or commands in English (mostly) and if it's a valid command, it passes to kernel.

Shell is a user program or its environment provided for user interaction. Shell is a command language interpreter that executes commands read from the standard input device (keyboard) or from a file. Shell is not part of system kernel, but uses the system kernel to execute programs, create files etc.

Several shell available with Linux including:

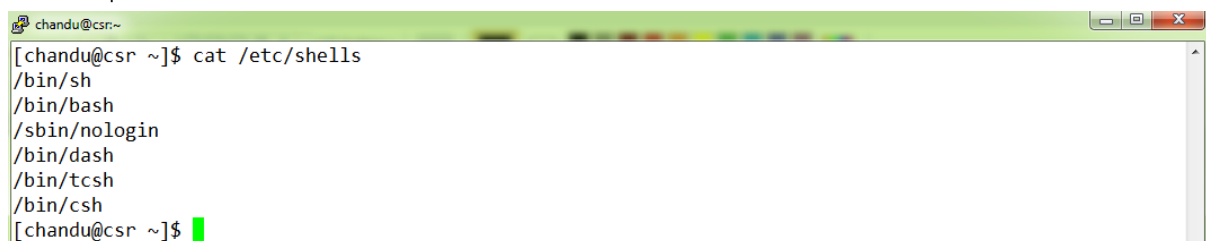
Shell Name	Developed by	Where	Remark
BASH (Bourne-Again Shell)	Brian Fox and Chet Ramey	Free Software Foundation	Most common shell in Linux. It's Freeware shell.
CSH (C Shell)	Bill Joy	University of California (For BSD)	The C shell's syntax and usage are very similar to the C programming language.
KSH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

Note: That each shell does the same job, but each understand a different command syntax and provides different built-in functions.

Any of the above shell reads command from user (via Keyboard) and tells Linux OS what users want. If we are giving commands from keyboard it is called command line interface (Usually in-front of **\$ prompt**, This prompt is depend upon your shell and Environment that you set or by your System Administrator, therefore you may get different prompt).

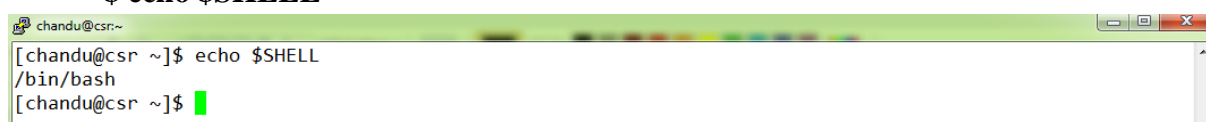
Tip: To find all available shells in your system type following command.

\$ cat /etc/shells

A terminal window titled 'chandu@csr:~' showing the command '[chandu@csr ~]\$ cat /etc/shells' and its output: '/bin/sh', '/bin/bash', '/sbin/nologin', '/bin/dash', '/bin/tcsh', '/bin/csh'. The prompt is '[chandu@csr ~]\$' with a green cursor.

Tip: To find your current shell type following command

\$ echo \$SHELL

A terminal window titled 'chandu@csr:~' showing the command '[chandu@csr ~]\$ echo \$SHELL' and its output: '/bin/bash'. The prompt is '[chandu@csr ~]\$' with a green cursor.

3. Commands and Utilities:

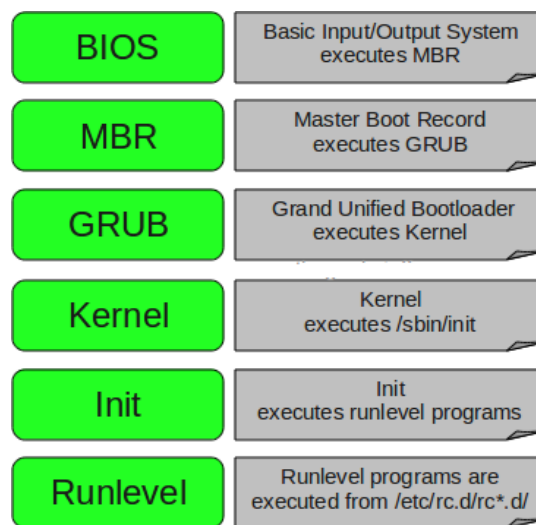
There are various command and utilities which you would use in your day to day activities. **cp**, **mv**, **cat** and **grep** etc. are few examples of commands and utilities. There are over 250 standard commands plus numerous others provided through 3rd party software. All the commands come along with various optional options.

4. Files and Directories: All data in UNIX is organized into files. All files are organized into directories. These directories are organized into a tree-like structure called the filesystem.

Linux stores data and programs in **files**. These are organized in directories. In a simple way, a directory is just a file that contains other files (or directories).

Linux Boot Process (Startup Sequence)

The following are the 6 high level stages of a typical Linux boot process.



1. BIOS

- BIOS stands for Basic Input/output System
- Performs some system integrity checks
- Searches, loads, and executes the boot loader program.
- It looks for boot loader in floppy, CD-ROM, or hard drive. You can press a key (typically F12 or F2, but it depends on your system) during the BIOS startup to change the boot sequence.
- Once the boot loader program is detected and loaded into the memory, BIOS gives the control to it.
- So, in simple terms BIOS loads and executes the MBR boot loader.

2. MBR

- MBR stands for Master Boot Record.
- It is located in the 1st sector of the bootable disk. Typically /dev/hda, or /dev/sda
- MBR is less than 512 bytes in size. This has three components 1) primary boot loader info in 1st 446 bytes 2) partition table info in next 64 bytes 3) mbr validation check in last 2 bytes.
- It contains information about GRUB (or LILO in old systems). So, in simple terms MBR loads and executes the GRUB boot loader.

3. GRUB

- GRUB stands for Grand Unified Boot loader.
- If you have multiple kernel images installed on your system, you can choose which one to be executed.
- GRUB displays a splash screen, waits for few seconds, if you don't enter anything, it loads the default kernel image as specified in the grub configuration file.
- GRUB has the knowledge of the filesystem (the older Linux loader LILO didn't understand filesystem).
- Grub configuration file is /boot/grub/grub.conf (/etc/grub.conf is a link to this). The following is sample grub.conf of CentOS.

```
#boot=/dev/sda
default=0
timeout=5
splashimage=(hd0,0)/boot/grub/splash.xpm.gz
hiddenmenu
title CentOS (2.6.18-194.el5PAE)
    root (hd0,0)
    kernel /boot/vmlinuz-2.6.18-194.el5PAE ro root=LABEL=/
    initrd /boot/initrd-2.6.18-194.el5PAE.img
```

- As you notice from the above info, it contains kernel and initrd image.
- So, in simple terms GRUB just loads and executes Kernel and initrd images.

4. Kernel

- Mounts the root file system as specified in the "root=" in grub.conf
- Kernel executes the /sbin/init program
- Since init was the 1st program to be executed by Linux Kernel, it has the process id (PID) of 1. Do a 'ps -ef | grep init' and check the PID.
- initrd stands for Initial RAM Disk.
- initrd is used by kernel as temporary root file system until kernel is booted and the real root file system is mounted. It also contains necessary drivers compiled inside, which helps it to access the hard drive partitions, and other hardware.

5. Init

- Looks at the /etc/inittab file to decide the Linux run level.
- Following are the available run levels
 - 0 – halt
 - 1 – Single user mode
 - 2 – Multiuser, without NFS
 - 3 – Full multiuser mode (Multiuser, with NFS)

- 4 – unused
- 5 – X11 (Multiuser, with NFS and GUI)
- 6 – reboot
- Init identifies the default initlevel from /etc/inittab and uses that to load all appropriate program.
- Execute ‘grep initdefault /etc/inittab’ on your system to identify the default run level
- If you want to get into trouble, you can set the default run level to 0 or 6. Since you know what 0 and 6 means, probably you might not do that.
- Typically you would set the default run level to either 3 or 5.

6. Run level programs

- When the Linux system is booting up, you might see various services getting started. For example, it might say “starting sendmail OK”. Those are the run level programs, executed from the run level directory as defined by your run level.
- Depending on your default init level setting, the system will execute the programs from one of the following directories.
 - Run level 0 – /etc/rc.d/rc0.d/
 - Run level 1 – /etc/rc.d/rc1.d/
 - Run level 2 – /etc/rc.d/rc2.d/
 - Run level 3 – /etc/rc.d/rc3.d/
 - Run level 4 – /etc/rc.d/rc4.d/
 - Run level 5 – /etc/rc.d/rc5.d/
 - Run level 6 – /etc/rc.d/rc6.d/
- Please note that there are also symbolic links available for these directory under /etc directly. So, /etc/rc0.d is linked to /etc/rc.d/rc0.d.
- Under the /etc/rc.d/rc*.d/ directories, you would see programs that start with S and K.
- Programs starts with S are used during startup. S for startup.
- Programs starts with K are used during shutdown. K for kill.
- There are numbers right next to S and K in the program names. Those are the sequence number in which the programs should be started or killed.
- For example, S12syslog is to start the syslog daemon, which has the sequence number of 12. S80sendmail is to start the sendmail daemon, which has the sequence number of 80. So, syslog program will be started before sendmail.

Basic Commands

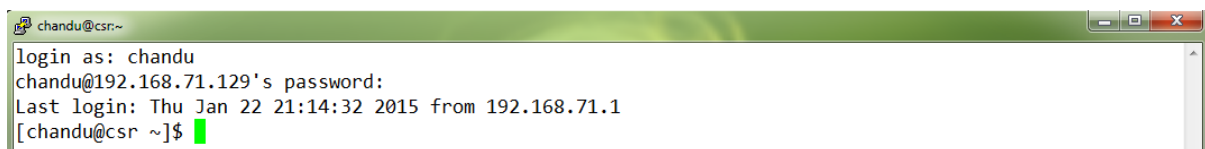
Login UNIX:

When you first connect to a UNIX system, you usually see a prompt such as the following:

A terminal window with a green title bar. The prompt is 'root@localhost:~' and the login prompt is 'login as: ' followed by a green cursor.

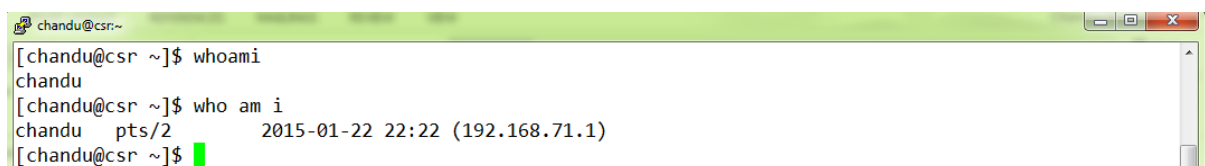
To log in:

- Have your userid (user identification) and password ready. Contact your system administrator if you don't have these yet.
- Type your userid at the login prompt, then press ENTER. Your userid is case-sensitive, so be sure you type it exactly as your system administrator instructed.
- Type your password at the password prompt, then press ENTER. Your password is also case-sensitive.
- If you provided correct userid and password then you would be allowed to enter into the system. Read the information and messages that come up on the screen something as below.

A terminal window with a green title bar. The prompt is 'chandu@csr:~'. The login sequence is shown: 'login as: chandu', 'chandu@192.168.71.129's password:', 'Last login: Thu Jan 22 21:14:32 2015 from 192.168.71.1', and the prompt changes to '[chandu@csr ~]\$'.

Who are you?

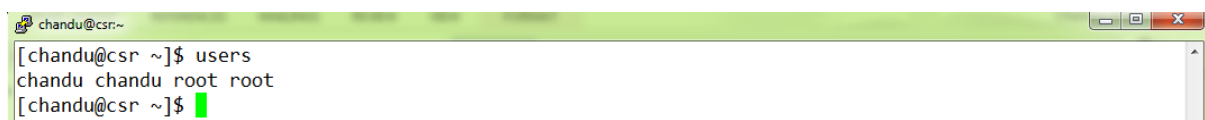
While you're logged in to the system, you might be willing to know: **Who am I?** The easiest way to find out "who you are" is to enter the **whoami** command:

A terminal window with a green title bar. The prompt is 'chandu@csr:~'. The user enters 'whoami' and the output is 'chandu'. Then the user enters 'who am i' and the output is 'chandu pts/2 2015-01-22 22:22 (192.168.71.1)'. The prompt returns to '[chandu@csr ~]\$'.

Try it on your system. This command lists the account name associated with the current login. You can try **who am i** command as well to get information about yourself.

Who is Logged In?

Sometime you might be interested to know who is logged in to the computer at the same time. There are three commands are available to get you this information, based on how much you'd like to learn about the other users: **users**, **who**, and **w**.

A terminal window with a green title bar. The prompt is 'chandu@csr:~'. The user enters 'users' and the output is 'chandu chandu root root'. The prompt returns to '[chandu@csr ~]\$'.

```
chandu@csr:~$ who
root    tty1      2015-01-22 06:25 (:0)
chandu  pts/0      2015-01-22 21:14 (192.168.71.1)
root    pts/1      2015-01-22 21:16 (:0.0)
chandu  pts/2      2015-01-22 22:22 (192.168.71.1)
chandu@csr:~$

chandu@csr:~$ w
 22:30:30 up  2:20,  4 users,  load average: 0.05, 0.02, 0.00
USER      TTY      FROM            LOGIN@   IDLE   JCPU   PCPU   WHAT
root      tty1      :0               06:25    16:06m 41.91s 41.91s /usr/bin/Xorg :0 -br -verbose -audit 4
chandu    pts/0     192.168.71.1    21:14    1:15m  0.07s  0.07s -bash
root      pts/1     :0.0            21:16    9:05   0.15s  0.15s /bin/bash
chandu    pts/2     192.168.71.1    22:22    0.00s  0.21s  0.08s w
chandu@csr:~$
```

Logging Out:

When you finish your session, you need to log out of the system to ensure that nobody else accesses your files while masquerading as you.

```
chandu@csr:~$ exit
exit
chandu@csr:~$
```

To log out just type **logout** or **exit** command at command prompt, and the system will clean up everything and break the connection

System Shutdown:

The most consistent way to shut down a UNIX system properly via the command line is to use one of the following commands:

Command	Description
halt	Brings the system down immediately
init 0	Powers off the system using predefined scripts to synchronize and clean up the system prior to shutdown
init 6	Reboots the system by shutting it down completely and then bringing it completely back up
poweroff	Shuts down the system by powering off.
reboot	Reboots the system.
shutdown	Shuts down the system.

Note: You typically need to be the superuser or root (the most privileged account on a Unix system) to shut down the system, but on some standalone or personally owned Unix boxes, an administrative user and sometimes regular users can do so.

2. UNIX File System

All data in UNIX is organized into files. All files are organized into directories. These directories are organized into a tree-like structure called the filesystem.

When you work with UNIX, one way or another you spend most of your time working with files. In UNIX there are three basic types of files:

1. Ordinary Files:

An ordinary file is a file on the system that contains data, text, or program instructions. In this tutorial, you look at working with ordinary files.

2. Directories:

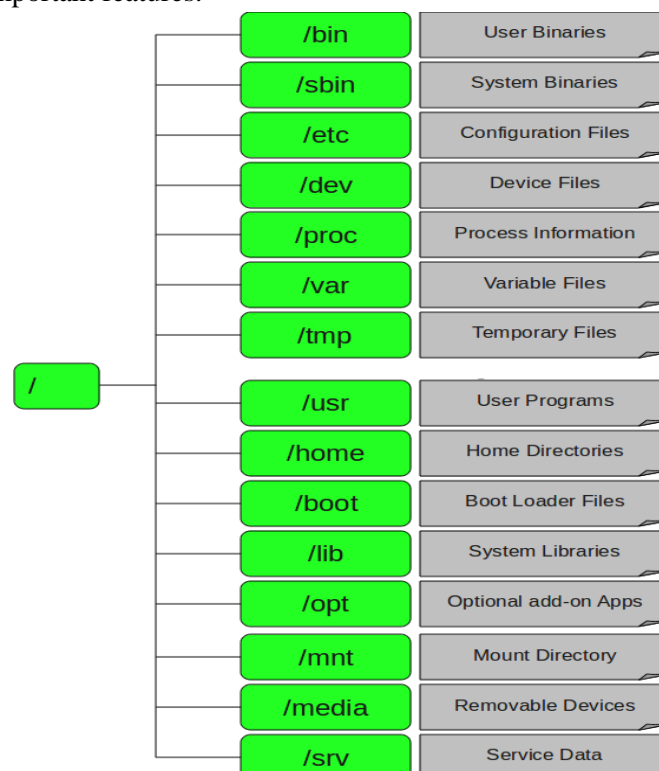
Directories store both special and ordinary files. For users familiar with Windows or Mac OS, UNIX directories are equivalent to folders.

3. Special Files:

Some special files provide access to hardware such as hard drives, CD-ROM drives, modems, and Ethernet adapters. Other special files are similar to aliases or shortcuts and enable you to access a single file using different names.

UNIX File System Structure

A file system is a logical method for organizing and storing large amounts of information in a way which makes it easy manage. The file is the smallest unit in which information is stored. The UNIX file system has several important features.



The UNIX file system is organized as a hierarchy of directories starting from a single directory called *root* which is represented by a */*(slash). Imagine it as being similar to the root system of a plant or as an inverted tree structure.

1. / – Root

- Every single file and directory starts from the root directory.
- Only root user has write privilege under this directory.
- Please note that */root* is root user's home directory, which is not same as */*.

2. /bin – User Binaries

- Contains binary executables.
- Common Linux commands you need to use in single-user modes are located under this directory.
- Commands used by all the users of the system are located here.
- For example: *ps*, *ls*, *ping*, *grep*, *cp*.

3. /sbin – System Binaries

- Just like */bin*, */sbin* also contains binary executables.
- But, the Linux commands located under this directory are used typically by system administrator, for system maintenance purpose.
- For example: *iptables*, *reboot*, *fdisk*, *ifconfig*, *swapon*

4. /etc – Configuration Files

- Contains configuration files required by all programs.
- This also contains startup and shutdown shell scripts used to start/stop individual programs.
- For example: */etc/resolv.conf*, */etc/logrotate.conf*

5. /dev – Device Files

- Contains device files.
- These include terminal devices, *usb*, or any device attached to the system.
- For example: */dev/tty1*, */dev/usbmon0*

6. /proc – Process Information

- Contains information about system process.
- This is a pseudo filesystem contains information about running process. For example: */proc/{pid}* directory contains information about the process with that particular *pid*.

- This is a virtual filesystem with text information about system resources. For example: /proc/uptime

7. /var – Variable Files

- var stands for variable files.
- Content of the files that are expected to grow can be found under this directory.
- This includes — system log files (/var/log); packages and database files (/var/lib); emails (/var/mail); print queues (/var/spool); lock files (/var/lock); temp files needed across reboots (/var/tmp);

8. /tmp – Temporary Files

- Directory that contains temporary files created by system and users.
- Files under this directory are deleted when system is rebooted.

9. /usr – User Programs

- Contains binaries, libraries, documentation, and source-code for second level programs.
- /usr/bin contains binary files for user programs. If you can't find a user binary under /bin, look under /usr/bin. For example: at, awk, cc, less, scp
- /usr/sbin contains binary files for system administrators. If you can't find a system binary under /sbin, look under /usr/sbin. For example: atd, cron, sshd, useradd, userdel
- /usr/lib contains libraries for /usr/bin and /usr/sbin
- /usr/local contains users programs that you install from source. For example, when you install apache from source, it goes under /usr/local/apache2

10. /home – Home Directories

- Home directories for all users to store their personal files.
- For example: /home/chandu, /home/shekhar

11. /boot – Boot Loader Files

- Contains boot loader related files.
- Kernel initrd, vmlinuz, grub files are located under /boot
- For example: initrd.img-2.6.32-24-generic, vmlinuz-2.6.32-24-generic

12. /lib – System Libraries

- Contains library files that supports the binaries located under /bin and /sbin
- Library filenames are either ld* or lib*.so.* For example: ld-2.11.1.so, libncurses.so.5.7

13. /opt – Optional add-on Applications

- opt stands for optional.
- Contains add-on applications from individual vendors.
- Add-on applications should be installed under either /opt/ or /opt/ sub-directory.

14. /mnt – Mount Directory

- Temporary mount directory where sysadmins can mount filesystem.

15. /media – Removable Media Devices

- Temporary mount directory for removable devices.
- For example: /media/cdrom for CD-ROM; /media/floppy for floppy drives; /media/cdrecorder for CD writer

16. /srv – Service Data

- srv stands for service.
- Contains server specific services related data.
- For example, /srv/cvs contains CVS related data.

Standard UNIX Streams:

Under normal circumstances every UNIX program has three streams (files) opened for it when it starts up:

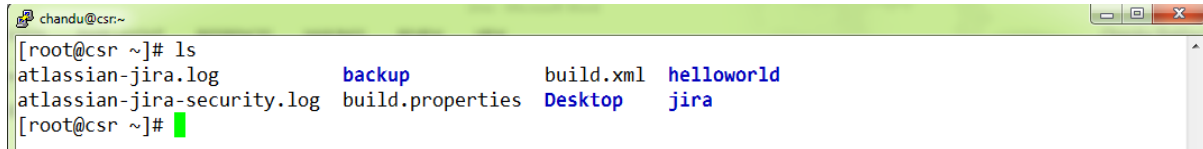
Stream	File Descriptor	Description
stdin	0	This is referred to as <i>standard input</i> and also represented as a STDIN. UNIX program would read default input from STDIN.
stdout	1	This is referred to as <i>standard output</i> and also represented as STDOUT. UNIX program would write default output at STDOUT
stderr	2	This is referred to as <i>standard error</i> and also represented as STDERR. UNIX program would write all the error message at STDERR.

4. UNIX File Management

Listing Files:

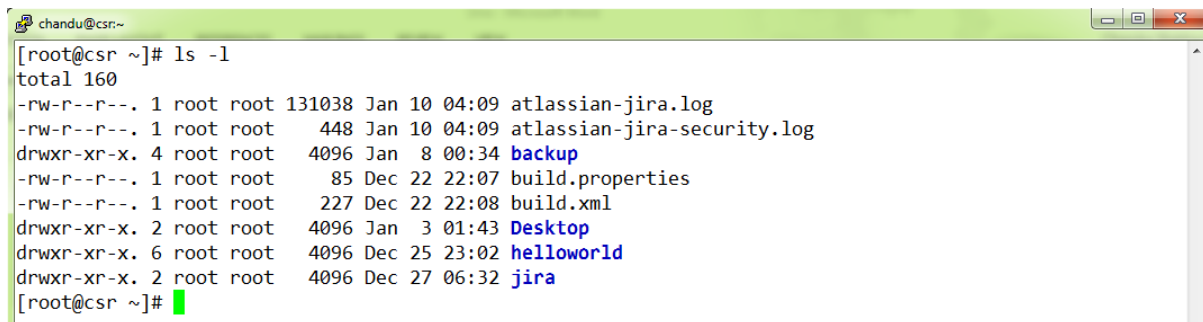
To list the files and directories stored in the current directory. Use the following command:

\$ ls



```
chandu@csr:~  
[root@csr ~]# ls  
atlassian-jira.log      backup      build.xml  helloworld  
atlassian-jira-security.log  build.properties  Desktop    jira  
[root@csr ~]#
```

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



```
chandu@csr:~  
[root@csr ~]# ls -l  
total 160  
-rw-r--r--. 1 root root 131038 Jan 10 04:09 atlassian-jira.log  
-rw-r--r--. 1 root root  448 Jan 10 04:09 atlassian-jira-security.log  
drwxr-xr-x. 4 root root  4096 Jan  8 00:34 backup  
-rw-r--r--. 1 root root   85 Dec 22 22:07 build.properties  
-rw-r--r--. 1 root root  227 Dec 22 22:08 build.xml  
drwxr-xr-x. 2 root root  4096 Jan  3 01:43 Desktop  
drwxr-xr-x. 6 root root  4096 Dec 25 23:02 helloworld  
drwxr-xr-x. 2 root root  4096 Dec 27 06:32 jira  
[root@csr ~]#
```

Here is the information about all the listed columns:

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

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

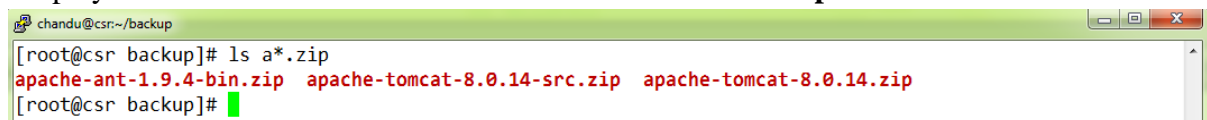
Prefix	Description
-	Regular file, such as an ASCII text file, binary executable, or hard link.
b	Block special file. Block input/output device file such as a physical hard drive
c	Character special file. Raw input/output device file such as a physical hard drive
d	Directory file that contains a listing of other files and directories.
l	Symbolic link file. Links on any regular file.
p	Named pipe. A mechanism for interprocess communications
s	Socket used for interprocess communication.

Meta Characters:

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

For Example:

Displays all the files whose name start with **a** and ends with **.zip**:



```
chandu@csr:~/backup
[root@csr backup]# ls a*.zip
apache-ant-1.9.4-bin.zip  apache-tomcat-8.0.14-src.zip  apache-tomcat-8.0.14.zip
[root@csr backup]#
```

Here `*` works as Meta character which matches with any character. If you want to display all the files ending with just **.zip** then you can use following command:



```
chandu@csr:~/backup
[root@csr backup]# ls *.zip
apache-ant-1.9.4-bin.zip  apache-tomcat-8.0.14.zip  glassfish-4.1-web.zip  nant-0.92-bin.zip
apache-tomcat-8.0.14-src.zip  cruisecontrol-bin-2.8.4.zip  log4j-1.2.17.zip  subversion-1.8.10.zip
[root@csr backup]#
```

Hidden Files:

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

Some common examples of hidden files:


```

chandu@csr:~$ ls -a
.                build.properties .esd_auth         .gtk-bookmarks   .mysql_history
..               .BuildServer      .gconf            .gvfs             .nautilus
atlassian-jira.log build.xml          .gconfd           helloworld        .pki
atlassian-jira-security.log .build.xml.swp    .gitconfig        .ICEauthority     .pulse
backup           .cache            .gnome2           .jenkins          .pulse-cookie
.bash_history    .config           .gnome2_private  jira              .ssh
.bash_logout     .cshrc            .gnote           .local            .subversion
.bash_profile    .dbus             .gnupg           .m2               .tcshrc
.bashrc          Desktop           .gstreamer-0.10  .mozilla          .viminfo
[root@csr ~]#

```

Single dot (.): This represents current directory.

Double dot (..): This represents parent directory.

ls command options:

Option	Description
ls -a	list all files including hidden file starting with '.'
ls --color	colored list [=always/never/auto]
ls -d	list directories - with '*'
ls -F	add one char of */=>@ to entries
ls -i	list file's inode index number
ls -l	list with long format - show permissions
ls -la	list long format including hidden files
ls -lh	list long format with readable file size
ls -ls	list with long format with file size
ls -r	list in reverse order
ls -R	list recursively directory tree
ls -s	list file size
ls -S	sort by file size
ls -t	sort by time & date
ls -X	sort by extension name

Creating Files:

We can create the files in many ways.

- You can create empty files using the following command:

\$ touch <filename>

```

chandu@csr:~$ touch file1.txt
[chandu@csr ~]$ ls
file1.txt
[chandu@csr ~]$

```

- By using **cat** command along with **redirections (>)**. If file already exist, it will allow to write the new text into that file, if not it will create a new file.

\$ cat > <filename>

```
chandu@csr:~$ cat > file2.txt
Hi this the text file
chandu@csr ~$ ls
file1.txt  file2.txt
chandu@csr ~$
```

- You can use **vi** or **vim** editor to create ordinary files on any UNIX system. You simply need to give following command:

\$ vi <filename>

\$ vim <filename>

Above command would open a file with the given filename. You would need to press key **i** to insert mode. Once you are in edit mode you can start writing your content in the file.

Once you are done, do the following steps:

- Press key **esc** to come out of edit mode.
- Press **:wq** together to save and exit from the editor.

Now you would have a file created with **filename** in the current directory.

```
chandu@csr:~$ vi file3.txt
chandu@csr ~$ ls
file1.txt  file2.txt  file3.txt
chandu@csr ~$
```

Display Content of a File:

You can use **cat** command to see the content of a single or multiple files at a time. Following is the simple example to see the content of above created file.

\$ cat [option] <filename> - <filename>

```
chandu@csr:~$ cat file3.txt
Hi, This file is created by using vi command.
chandu@csr ~$

chandu@csr:~$ cat file2.txt file3.txt
Hi this the text file
Hi, This file is created by using vi command.
chandu@csr ~$
```

cat command options:

Option	Description
-n	All the lines of the file output can be numbered.
-b	If file contains empty lines then you can skip numbering empty lines through -b option. This option (-b) overrides -n option. This means that even if -n is specified along with -b, empty lines will not be numbered in the output.
-E	End of every line of the file can be demarcated using -E option.
-T	If file contains tabs and it is required to know the presence of tabs then -T option can be used.

-s	If it is desired to suppress repeated empty output lines then you can use -s option.
-----------	--

```

chandu@csr:~$ cat -n file2.txt file3.txt
 1 Hi this the text file
 2
 3 Hi, This file is created by using vi command.
 4
[chandu@csr ~]$ cat -b file2.txt file3.txt
 1 Hi this the text file

 2 Hi, This file is created by using vi command.

[chandu@csr ~]$

```

Using head command:

Head prints the first N number of data of the given input. By default, it prints first 10 lines of each given file.

\$ head [option] <file>

Examples:

```

chandu@csr:~$ head /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
uucp:x:10:14:uucp:/var/spool/uucp:/sbin/nologin
[chandu@csr ~]$

chandu@csr:~$ head -2 /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
[chandu@csr ~]$

```

head command options:

Option	Description
-c, --bytes	To print N bytes from each input file.
-n, --lines	To print N lines from each input file.
-q, --silent, --quiet	Prevent printing of header information that contains file name
-v, --verbose	To print header information always.

```

chandu@csr:~$ head -n 4 /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
[chandu@csr ~]$

```

Using tail command:

Tail prints the last N number of lines from given input. By default, it prints last 10 lines of each given file.

\$ tail [option] <file>

Examples:

```
chandu@csr:~$ tail /etc/passwd
gdm:x:42:42::/var/lib/gdm:/sbin/nologin
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
tcpdump:x:72:72:::/sbin/nologin
chandu:x:500:500:CHANDRA SHEKHAR REDDY:/home/chandu:/bin/bash
rpc:x:32:32:Rpcbind Daemon:/var/cache/rpcbind:/sbin/nologin
mysql:x:27:27:MySQL Server:/var/lib/mysql:/bin/bash
babu:x:501:501:RamBabu:/home/babu:/bin/bash
takur:x:502:501:Nishant:/home/takur:/bin/bash
madhu:x:503:501:Madhu:/home/madhu:/bin/bash
ldap:x:55:55:LDAP User:/var/lib/ldap:/sbin/nologin
[chandu@csr ~]$
```

tail command options:

Option	Description
-c, --bytes	To print N bytes from each input file.
-n, --lines	To print N lines from each input file.
-q, --silent, --quiet	To prevent printing of header information.
-v, --verbose	To print header information always.
-s, --sleep-interval	To sleep for N seconds between iterations.
--retry	To keep retrying to open a file even when it is not exist or becomes inaccessible. Useful when it is used with -f.
-f, --follow	To print appended data as and when the file grows.

Example:

```
chandu@csr:~$ tail -n 7 /etc/passwd
chandu:x:500:500:CHANDRA SHEKHAR REDDY:/home/chandu:/bin/bash
rpc:x:32:32:Rpcbind Daemon:/var/cache/rpcbind:/sbin/nologin
mysql:x:27:27:MySQL Server:/var/lib/mysql:/bin/bash
babu:x:501:501:RamBabu:/home/babu:/bin/bash
takur:x:502:501:Nishant:/home/takur:/bin/bash
madhu:x:503:501:Madhu:/home/madhu:/bin/bash
ldap:x:55:55:LDAP User:/var/lib/ldap:/sbin/nologin
[chandu@csr ~]$
```

Counting Words in a File:

You can use the **wc** command to get a count of the total number of lines, words, and characters contained in a file.

\$ wc [option] <filename>

```
chandu@csr:~$ wc file3.txt
 3  9 48 file3.txt
[chandu@csr ~]$
```

Here is the detail of all the four columns:

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

You can give multiple files at a time to get the information about those file. Here is simple syntax:

```
chandu@csr:~$ wc file2.txt file3.txt
 1  5 22 file2.txt
 3  9 48 file3.txt
 4 14 70 total
chandu@csr ~]$
```

wc command options:

Option	Description
-c, --bytes	Print the byte counts.
-m, --chars	Print the character counts.
-l, --lines	Print the newline counts.
-w, --words	Print the word counts.
--version	Output version information, and exit.

Copying Files:

cp is the command which makes a copy of your files or directories. For instance, let's say you have a file named **demo.txt** in your working directory, and you want to make a copy of it called **newdemo.txt**. You would run the command:

Syntax: \$ cp [option] <source filename> <destination filename>

```
chandu@csr:~$ cp demo.txt newdemo.txt
chandu@csr ~$ ls
demo.txt file1.txt file2.txt file3.txt newdemo.txt passwd
chandu@csr ~$
```

cp command options:

Option	Description
-R, -r, --recursive	Copy directories recursively.
-s, --symbolic-link	Make symbolic links instead of copying.
-l, --link	Create hard links to files instead of copying.

-u, --update	Copy only when the source file is newer than the destination file or when the destination file is missing.
-f, --force	If an existing destination file cannot be opened, remove it and try again. (This option has no effect if the -n option is used)
-i, --interactive	Prompt before overwrite.
-n, --no-clobber	Do not overwrite an existing file
-v, --verbose	Verbose mode; explain what is being done.

Example:

```
chandu@csr:~$ [root@csr ~]# cp -n /etc/passwd /home/chandu/
[root@csr ~]#
```

Renaming or Moving Files:

To change the name of a file or directory we use the **mv** command. By using **mv** command we can move file(s) or directories from one place to another place (source to destination).

Syntax: \$ mv [option] <source> <destination>

Example: Renaming

```
chandu@csr:~$ [chandu@csr ~]$ ls
demo.txt file1.txt file2.txt file3.txt passwd sample.txt
[chandu@csr ~]$ mv sample.txt newsample.txt
[chandu@csr ~]$ ls
demo.txt file1.txt file2.txt file3.txt newsample.txt passwd
[chandu@csr ~]$
```

Example: Moving

```
chandu@csr:~$ [chandu@csr ~]$ ls
demo.txt file1.txt file2.txt file3.txt newsample.txt passwd
[chandu@csr ~]$ mv demo.txt /tmp
[chandu@csr ~]$ ls
file1.txt file2.txt file3.txt newsample.txt passwd
[chandu@csr ~]$ ls /tmp
demo.txt keyring-iYaRXA pulse-m7rcPgkzn77B virtual-root.kI1eKT vmware-root-860463566
keyring-BANiEU keyring-KwfqHN pulse-yIlaM6Cvghw4 virtual-root.xbo4XP
keyring-CcgWuA orbit-gdm test-svn vmware-root
keyring-gUe9J2 orbit-root virtual-root.5D0egj vmware-root-592023982
[chandu@csr ~]$
```

mv command options:

Option	Description
-f, --force	Do not prompt before overwriting existing files.
-i, --interactive	Prompt before overwriting existing files.
-n, --no-clobber	Do not overwrite any existing file
-u, --update	Perform the move only if the source file is newer than the destination file, or the destination file does not already exist.

-v, --verbose	Verbose mode; explain what is being done.
----------------------	---

Deleting Files:

To delete an existing file or directory use the **rm** command. By default, it does not remove directories

Syntax: `$ rm [option] <file or directory name>`

Note: It may be dangerous to delete a file because it may contain useful information. So be careful while using this command. It is recommended to use **-i** option along with **rm** command.

Example:

```

chandu@csr:~$ ls
file1.txt file2.txt file3.txt newsample.txt passwd
[chandu@csr ~]$ rm newsample.txt
[chandu@csr ~]$ ls
file1.txt file2.txt file3.txt passwd
[chandu@csr ~]$

```

rm command options:

Option	Description
-f, --force	Ignore nonexistent files, and never prompt before removing.
-i, --interactive	Prompt before every removal.
-I	Prompt once before removing more than three (3) files, or when removing recursively.
-r, -R, --recursive	Remove directories and their contents recursively.
-v, --verbose	Verbose mode; explain at all times what is being done.

Example: Removing a directory and their contents by using rm command options.

```

chandu@csr:~$ ls
chandu file2.txt file3.txt passwd
[chandu@csr ~]$ rm -rf chandu/
[chandu@csr ~]$ ls
file2.txt file3.txt passwd
[chandu@csr ~]$

```

The removal process unlinks a filename in a filesystem from data on the storage device, and marks that space as usable by future writes. In other words, removing files increases the amount of available space on your disk.

4. UNIX Directory Management

A directory is a file whose whole job is to store file names and related information. All files, whether ordinary, special, or directory, are contained in directories.

UNIX uses a hierarchical structure for organizing files and directories. This structure is often referred to as a directory tree. The tree has a single root node, the slash character (/), and all other directories are contained below it.

Home Directory: The directory in which you find yourself when you first login is called your home directory. You will be doing much of your work in your home directory and subdirectories that you'll be creating to organize your files.

You can go in your home directory anytime using the following command:

```
$ cd ~
```

Here ~ indicates home directory. If you want to go in any other user's home directory then use the following command:

```
$ cd ~ username
```

To go in your last directory you can use following command:

```
$ cd -
```

Path:

A path is a unique location to a file or a directory in a file system of an OS. A path to a file is a combination of / and alpha-numeric characters.

Absolute path:

An absolute path is defined as the specifying the location of a file or directory from the root directory (/). In other words we can say absolute path is a complete path from start of actual filesystem from / directory.

```
/var/ftp/pub  
/etc/samba.smb.conf  
/boot/grub/grub.conf
```

If you see all these paths started from / directory which is a root directory for every Linux/Unix machines.

Relative path:

Relative path is defined as path related to the present working directory (pwd). Suppose I am located in /var/log and I want to change directory to /var/log/kernel. I can use relative path concept to change directory to kernel.

Changing directory to /var/log/kernel by using relative path concept.

```
pwd  
/var/log  
cd cups
```


Example:

```
chandu@csr/var/log/cups
[chandu@csr log]$ pwd
/var/log
[chandu@csr log]$ cd cups/
[chandu@csr cups]$ pwd
/var/log/cups
[chandu@csr cups]$
```

Note: If you observe there is no / before cups which indicates it's a relative directory to present working directory.

Changing directory to /var/log/cups using absolute path concept.

\$ cd /var/log/cups

Note: We can use an absolute path from any location where as if you want to use relative path we should be present in a directory where we are going to specify relative to that present working directory.

Listing Directories:

To list the files in a directory you can use the following syntax:

\$ ls [option] <dirname>

Following is the example to list all the files contained in /usr/local directory:

```
chandu@csr~
[chandu@csr ~]$ ls /usr/local/
bin etc games include lib libexec sbin share src
[chandu@csr ~]$
```

Creating Directories:

Directories are created by the following command.

\$ mkdir [option] <dirname>

Here, directory is the absolute or relative pathname of the directory you want to create. For example, the command:

\$ mkdir backup

Creates the directory **backup** in the current directory. Here is another example:

\$ mkdir /tmp/mydir

This command creates the directory **mydir** in the **/tmp** directory. The **mkdir** command produces no output if it successfully creates the requested directory.

Using this command you can create multiple directories at a time.

For example:

\$ mkdir work backup test

The above command creates the three directories in current directory.

Creating Parent Directories:

Sometimes when you want to create a directory, its parent directory or directories might not exist. In this case, `mkdir` issues an error message as follows:

```
chandu@csr:~$ mkdir /tmp/work/test
mkdir: cannot create directory `/tmp/work/test': No such file or directory
[chandu@csr ~]$
```

In such cases, you can specify the **-p** option to the **mkdir** command. It creates all the necessary directories for you.

For example:

```
$ mkdir -p /tmp/work/test
```

Above command creates all the required parent directories.

mkdir command options:

Option	Description
-m, --mode	Set file mode (as in <code>chmod</code>), not <code>a=rwx - umask</code> .
-p, --parents	No error if existing, make parent directories as needed.
-v, --verbose	Print a message for each created directory.
-Z, --context	Set the SELinux security context of each created directory to CTX

Renaming or Moving Directories:

To change the name of a directory we use the **mv** command. By using **mv** command we can move file(s) or directories from one place to another place (source to destination).

Syntax: `$ mv [option] <source> <destination>`

Example: Renaming

```
chandu@csr:~$ ls
file2.txt file3.txt passwd work
[chandu@csr ~]$ mv work/ test
[chandu@csr ~]$ ls
file2.txt file3.txt passwd test
```

Example: Moving

```
chandu@csr:~$ ls
file2.txt file3.txt passwd work
[chandu@csr ~]$ mv work/ /tmp
[chandu@csr ~]$ ls
file2.txt file3.txt passwd
[chandu@csr ~]$ ls /tmp/
keyring-BAWIeU keyring-KwfqHN pulse-yIlaM6Cvghw4 virtual-root.kI1eKT vmware-root-860463566
keyring-CcgWuA orbit-gdm test virtual-root.xbo4XP work
keyring-gUe9J2 orbit-root test-svn vmware-root
keyring-iYaRXA pulse-m7rcPgkzn77B virtual-root.5D0egj vmware-root-592023982
[chandu@csr ~]$
```

mv command options:

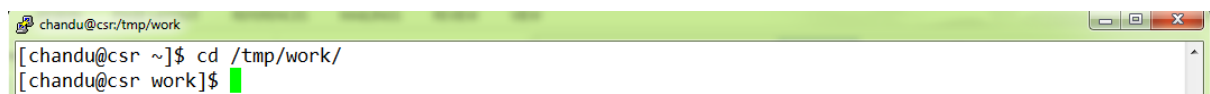
Option	Description
-f, --force	Do not prompt before overwriting existing directories.
-i, --interactive	Prompt before overwriting existing directories and its contents.
-n, --no-clobber	Do not overwrite any existing directory.
-u, --update	Perform the move only if the source directory is newer than the destination, or the destination directory does not already exist.
-v, --verbose	Verbose mode; explain what is being done.

Changing Directories:

The **cd** (change directory) command is one of the commands you will use the most at the command line in Linux. It allows you to change your working directory. You use it to move around within the hierarchy of your file system.

Syntax: `$ cd [option] <dirname>`

Example:



```
chandu@csr:/tmp/work
[chandu@csr ~]$ cd /tmp/work/
[chandu@csr work]$
```

cd command options:

Option	Description
-L	Force symbolic links to be followed. In other words, if you tell cd to move into a 'directory' which is actually a symbolic link to a directory, it moves into the directory the symbolic link points to. This is the default behavior of cd ; normally, it will always act as if -L has been specified.
-P	Use the physical directory structure without following symbolic links. In other words, only change into the specified directory if it actually exists as named; symbolic links will not be followed. This is the opposite of the -L option, and if they are both specified, this option will be ignored.
-e	If the -p option is specified, and the current working directory cannot be determined, this options tells cd to exit with an error. If -p is not specified along with this option, this option has no function.

Removing Directories:

The **rmdir** utility removes the directory entry specified by each directory argument, provided the directory is empty. Arguments are processed in the order given. In order to remove both a parent directory and a subdirectory of that parent, the subdirectory must be specified first so the parent directory is empty when **rmdir** tries to remove it.

\$ rmdir [option] <dirname>

Note: To remove a directory make sure it is empty which means there should not be any file or sub-directory inside this directory.

To remove a directory and its contents at a time use the command **rm** along with its options.

\$ rm [option] <dirname>

Example:

```
chandu@csr:~$ ls
file2.txt file3.txt passwd work
chandu@csr:~$ rm -rf work/
chandu@csr:~$ ls
file2.txt file3.txt passwd
chandu@csr:~$
```

The directories . (dot) and .. (dot dot):

The filename . (dot) represents the current working directory; and the filename .. (dot dot) represent the directory one level above the current working directory, often referred to as the parent directory.

If we enter the command to show a listing of the current working directories files and use the **-a** option to list all the files and the **-l** option provides the long listing, this is the result.

```
chandu@csr:~$ ls -al
total 68
drwx-----. 6 chandu chandu 4096 Jan 26 01:36 .
drwxr-xr-x. 6 root root 4096 Jan 8 05:32 ..
-rw-----. 1 chandu chandu 4138 Jan 25 22:53 .bash_history
-rw-r--r--. 1 chandu chandu 18 Oct 16 06:52 .bash_logout
-rw-r--r--. 1 chandu chandu 176 Oct 16 06:52 .bash_profile
-rw-r--r--. 1 chandu chandu 124 Oct 16 06:52 .bashrc
-rw-rw-r--. 1 chandu chandu 22 Jan 23 21:20 file2.txt
-rw-rw-r--. 1 chandu chandu 48 Jan 23 21:46 file3.txt
-rw-rw-r--. 1 chandu chandu 54 Dec 19 21:26 .gitconfig
drwxr-xr-x. 2 chandu chandu 4096 Nov 11 2010 .gnome2
-rw-----. 1 chandu chandu 126 Jan 23 07:06 .history
drwxr-xr-x. 4 chandu chandu 4096 Dec 1 17:22 .mozilla
-rw-r--r--. 1 root root 1745 Jan 24 00:35 passwd
drwx-----. 2 chandu chandu 4096 Dec 9 02:04 .ssh
-rw-----. 1 chandu chandu 1116 Jan 23 21:46 .viminfo
drwxrwxr-x. 3 chandu chandu 4096 Jan 26 01:36 work
chandu@csr:~$
```

rmdir command options:

Option	Description
--ignore-fail-on-non-empty	Ignore each failure that is solely because a directory is non-empty.
-p, --parents	remove directory and its ancestors; e.g., ‘rmdir -p a/b/c’ is similar to ‘rmdir a/b/c a/b a’
-v, --verbose	output a diagnostic for every directory processed

5. UNIX File Permissions

File ownership is an important component of UNIX that provides a secure method for storing files. Every file in UNIX has the following attributes:

File Users	Description
Owner	The owner's permissions determine what actions the owner of the file can perform on the file.
Group	The group's permissions determine what action a user, who is a member of the group that a file belongs to, can perform on the file.
Other	The permissions for others indicate what action all other users can perform on the file.

File Access Modes:

The permissions of a file are the first line of defense in the security of a UNIX system. The basic building blocks of UNIX permissions are the read, write, and execute permissions, which are described below:

Access Mode	Description
Read	Grants the permission to read i.e. View the contents of the file.
Write	Grants the permission to modify, or remove the content of the file.
Execute	User with execute permissions can run a file as a program.

Directory Access Modes:

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

Access Mode	Description
Read	Access to a directory means that the user can read the contents. The user can look at the filenames inside the directory.
Write	It means that the user can add or delete files to the contents of the directory.
Execute	Executing a directory does not really make a lot of sense so think of this as a traverse permission.

Changing Permissions:

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

Syntax: `$ chmod [option] [mode] <file or directory>`

Using chmod in Symbolic Mode:

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

chmod operator	Description
+	Adds the designated permission(s) to a file or directory.
-	Removes the designated permission(s) from a file or directory.
=	Sets the designated permission(s).

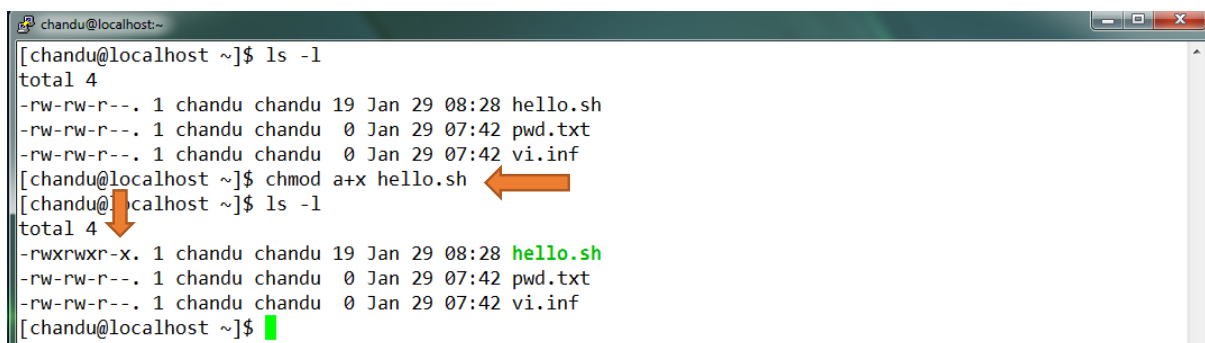
Linux (and almost all other UNIX systems) have three user classes as follows:

1. User (**u**): The owner of file
2. Group (**g**): Other user who are in group (to access files)
3. Other (**o**): Everyone else
4. All (**a**): For all the users ($u+g+o=a$)

You can setup following mode on each files. In a Linux and UNIX set of permissions is called as mode:

- ❖ Read (**r**)
- ❖ Write (**w**)
- ❖ Execute (**x**)

Example: \$ chmod a+x hello.sh

A terminal window titled 'chandu@localhost:~' showing the process of adding execute permissions to a file. The user runs 'ls -l' showing a list of files including 'hello.sh' with permissions '-rw-rw-r--'. Then, the user runs 'chmod a+x hello.sh'. A red arrow points to this command. Finally, the user runs 'ls -l' again, and a red arrow points to the updated permissions for 'hello.sh', which are now '-rwxrwxr-x'.

```
[chandu@localhost ~]$ ls -l
total 4
-rw-rw-r--. 1 chandu chandu 19 Jan 29 08:28 hello.sh
-rw-rw-r--. 1 chandu chandu  0 Jan 29 07:42 pwd.txt
-rw-rw-r--. 1 chandu chandu  0 Jan 29 07:42 vi.inf
[chandu@localhost ~]$ chmod a+x hello.sh
[chandu@localhost ~]$ ls -l
total 4
-rwxrwxr-x. 1 chandu chandu 19 Jan 29 08:28 hello.sh
-rw-rw-r--. 1 chandu chandu  0 Jan 29 07:42 pwd.txt
-rw-rw-r--. 1 chandu chandu  0 Jan 29 07:42 vi.inf
[chandu@localhost ~]$
```

In the above example **hello.sh** does not have **execute (x)** permissions first time, but using **chmod** command we issued the execute permissions to the **all users (a)**, so any user can execute that file.

Using chmod with Absolute Mode:

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

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

Number	Octal Permission Representation	Ref
0	No permission	---
1	Execute permission	--X
2	Write permission	-W-
3	Execute and write permission: 1 (execute) + 2 (write) = 3	-WX
4	Read permission	r--
5	Read and execute permission: 4 (read) + 1 (execute) = 5	r-X
6	Read and write permission: 4 (read) + 2 (write) = 6	rw-
7	All permissions: 4 (read) + 2 (write) + 1 (execute) = 7	rwX

Example: \$ chmod 777 hello.sh

```

chandu@localhost:~$ chmod 777 hello.sh
[chandu@localhost ~]$ ls -l
total
-rwxrwxrwx. 1 chandu chandu 19 Jan 29 08:28 hello.sh
-rw-rw-r--. 1 chandu chandu  0 Jan 29 07:42 pwd.txt
-rw-rw-r--. 1 chandu chandu  0 Jan 29 07:42 vi.inf
[chandu@localhost ~]$

```

In the above example we have given all the permissions (read, write, execute) to the all users (owner, group, other).

```

chandu@localhost:~$ chmod 000 hello.sh
[chandu@localhost ~]$ ls -ll
total
----- 1 chandu chandu 19 Jan 29 08:28 hello.sh
-rw-rw-r--. 1 chandu chandu  0 Jan 29 07:42 pwd.txt
-rw-rw-r--. 1 chandu chandu  0 Jan 29 07:42 vi.inf
[chandu@localhost ~]$

```

In the above example we have removed all the permissions (read, write, execute) to the all users (owner, group, other).

chmod command options:

Option	Description
-c, --changes	Like verbose but report only when a change is made.
-f, --silent, --quiet	Suppress most error messages.
--reference	Use RFILE's mode instead of MODE values.
-R, --recursive	Applies permissions recursively.
-v, --verbose	Output a diagnostic for every file processed.

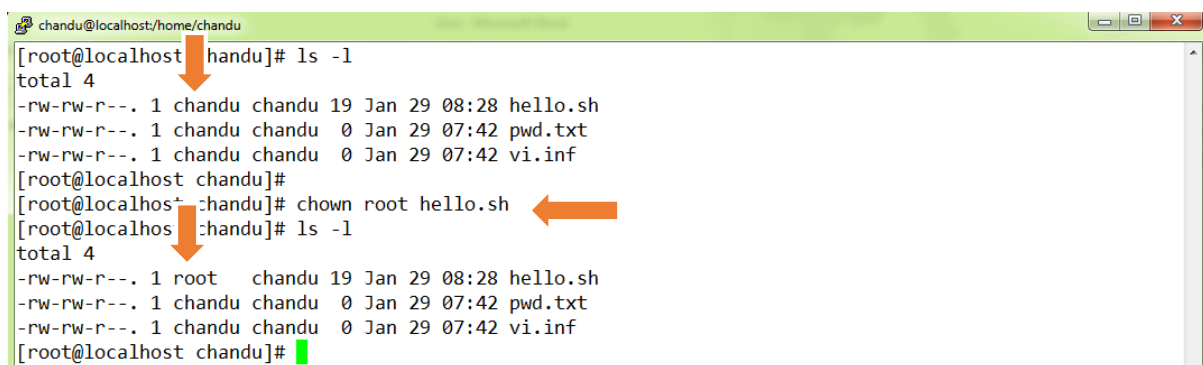
Changing Owners and Groups:

While creating an account on UNIX, it assigns an owner ID and a group ID to each user. All the permissions mentioned above are also assigned based on Owner and Groups. Two commands are available to change the owner and the group of files:

1. **Changing Ownership:** The **chown** command stands for "change owner" and is used to change the owner of a file.

Syntax: `$ chown [option] [user] <file or directory>`

Example: `$ chown root hello.sh`



```
chandu@localhost/home/chandu
[root@localhost chandu]# ls -l
total 4
-rw-rw-r--. 1 chandu chandu 19 Jan 29 08:28 hello.sh
-rw-rw-r--. 1 chandu chandu  0 Jan 29 07:42 pwd.txt
-rw-rw-r--. 1 chandu chandu  0 Jan 29 07:42 vi.inf
[root@localhost chandu]#
[root@localhost chandu]# chown root hello.sh
[root@localhost chandu]# ls -l
total 4
-rw-rw-r--. 1 root  chandu 19 Jan 29 08:28 hello.sh
-rw-rw-r--. 1 chandu chandu  0 Jan 29 07:42 pwd.txt
-rw-rw-r--. 1 chandu chandu  0 Jan 29 07:42 vi.inf
[root@localhost chandu]#
```

As shown in the example, first time **hello.sh** file owned by the **user chandu**, but using **chown** command we have changes the owner of the file to **root user**.

chown command options:

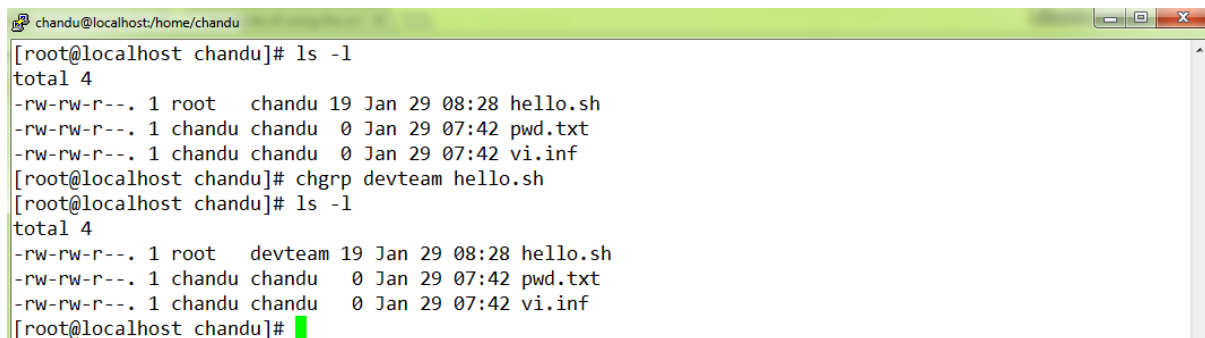
Option	Description
-c, --changes	Like verbose but report only when a change is made.
-f, --silent, --quiet	Suppress most error messages.
--reference	Use RFILE's mode instead of MODE values.
--dereference	Affect the referent of each symbolic link (this is the default), rather than the symbolic link itself.
-h, --no-dereference	Affect each symbolic link instead of any referenced file (useful only on systems that can change the ownership of a symlink).
-R, --recursive	Applies permissions recursively.
-v, --verbose	Output a diagnostic for every file processed.
-H	If a command line argument is a symbolic link to a directory, traverse it.
-L	Traverse every symbolic link to a directory encountered
-P	Do not traverse any symbolic links (default)

Note: The **-H**, **-L**, **-P** options modify how a hierarchy is traversed when the **-R** option is also specified. If more than one is specified, only the final one takes effect.

2. **Changing Group Ownership:** The **chgrp** command stands for "change group" and is used to change the group of a file. The value of group can be the name of a group on the system or the group ID (GID) of a group on the system.

Syntax: **\$ chgrp [option] [user] <file or directory>**

Example: \$ chgrp devteam hello.sh



```
chandu@localhost/home/chandu
[root@localhost chandu]# ls -l
total 4
-rw-rw-r--. 1 root  chandu 19 Jan 29 08:28 hello.sh
-rw-rw-r--. 1 chandu chandu 0 Jan 29 07:42 pwd.txt
-rw-rw-r--. 1 chandu chandu 0 Jan 29 07:42 vi.inf
[root@localhost chandu]# chgrp devteam hello.sh
[root@localhost chandu]# ls -l
total 4
-rw-rw-r--. 1 root  devteam 19 Jan 29 08:28 hello.sh
-rw-rw-r--. 1 chandu chandu 0 Jan 29 07:42 pwd.txt
-rw-rw-r--. 1 chandu chandu 0 Jan 29 07:42 vi.inf
[root@localhost chandu]#
```

Here **hello.sh** file belongs to **group chandu**, but using **chgrp** command we have changed it to **group devteam**.

chgrp command options:

All options are same as **chown** command options.

SUID and SGID File Permissions:

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

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

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

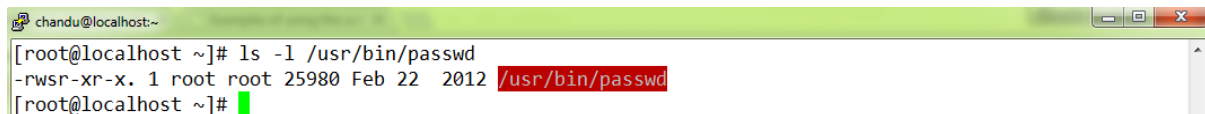
Additional permissions are given to programs via a mechanism known as the Set User ID (SUID) and Set Group ID (SGID) bits.

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

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

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

Example:

A terminal window titled 'chandu@localhost:~' showing the command 'ls -l /usr/bin/passwd' being executed. The output line is '-rwsr-xr-x. 1 root root 25980 Feb 22 2012 /usr/bin/passwd', where the file path is highlighted in red. The prompt '[root@localhost ~]#' is visible on the next line.

```
chandu@localhost:~  
[root@localhost ~]# ls -l /usr/bin/passwd  
-rwsr-xr-x. 1 root root 25980 Feb 22 2012 /usr/bin/passwd  
[root@localhost ~]#
```

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

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

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

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

\$ chmod ug+s <dirname>

Controlling File Permissions with umask:

When user create a file or directory under Linux or UNIX, she create it with a default set of permissions. In most case the system defaults may be open or relaxed for file sharing purpose. For example, if a text file has 666 permissions, it grants read and write permission to everyone. Similarly a directory with 777 permissions, grants read, write, and execute permission to everyone.

Default umask Value:

The user file-creation mode mask (umask) is use to determine the file permission for newly created files. It can be used to control the **default file permission for new files**. It is a four-digit octal number. An umask can be set or expressed using:

- Symbolic values
- Octal values

Procedure to Setup Default umask:

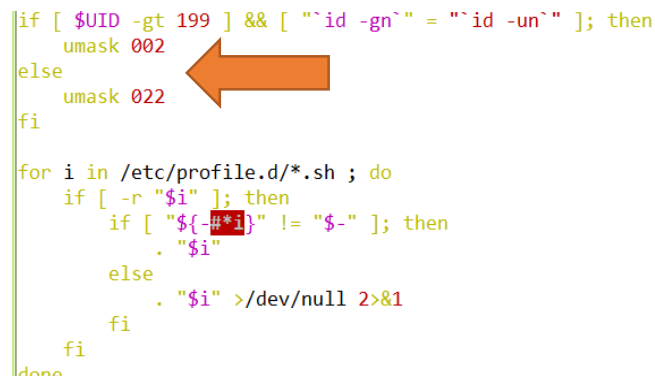
You can setup umask in `/etc/bashrc` or `/etc/profile` file for all users. By default most Linux distro set it to **0022 (022)** or **0002 (002)**.

Open `/etc/profile` or `~/.bashrc` file, enter:

Example: `$ vim /etc/profile`

```
if [ $UID -gt 199 ] && [ "`id -gn`" = "`id -un`" ]; then
    umask 002
else
    umask 022
fi

for i in /etc/profile.d/*.sh ; do
    if [ -r "$i" ]; then
        if [ "${-#*i}" != "$-" ]; then
            . "$i"
        else
            . "$i" >/dev/null 2>&1
        fi
    fi
done
```

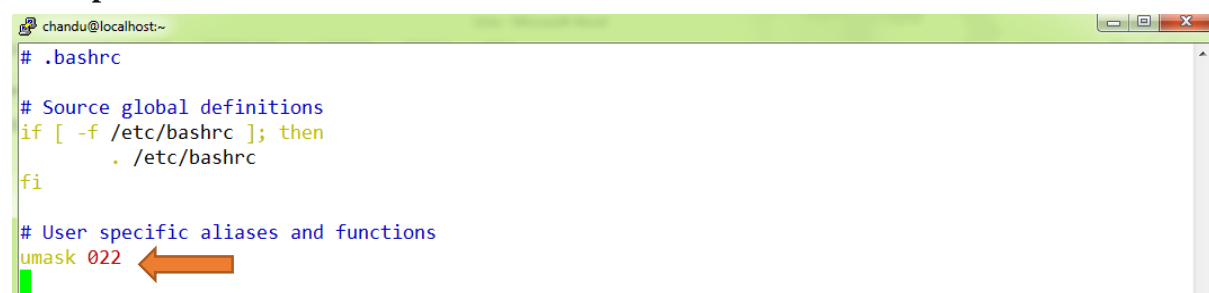


Example: `$ vim ~/.bashrc`

```
# .bashrc

# Source global definitions
if [ -f /etc/bashrc ]; then
    . /etc/bashrc
fi

# User specific aliases and functions
umask 022
```



Save and close the file (**To save and exit, Press ESC then :wq ENTER**). Changes will take effect after next login. All UNIX users can override the system umask defaults in their `/etc/profile` file, `~/.profile` (Korn / Bourne shell) `~/.cshrc` file (C shells), `~/.bash_profile` (Bash shell) or `~/.login` file (defines the user's environment at login).

For reference, the following table shows the mappings between umask values and default permissions. BE VERY CAREFUL not to confuse umask and chmod permissions, as they are entirely different (a binary inversion of each other) and are NOT INTERCHANGABLE.

Octal	Binary	Perms	Octal	Binary	Perms
0	000	rwX	4	100	-WX
1	001	rw-	5	101	-W-
2	010	r-X	6	110	--X
3	011	r--	7	111	---

To discover what umask you are currently working with, type:

\$ umask

Here are some examples of settings for umask

- umask 077 - Assigns permissions so that only you have read/write access for files, and read/write/search for directories you own. All others have no access permissions to your files or directories.
- umask 022 - Assigns permissions so that only you have read/write access for files, and read/write/search for directories you own. All others have read access only to your files, and read/search access to your directories.
- umask 002 - Assigns permissions so that only you and members of your group have read/write access to files, and read/write/search access to directories you own. All others have read access only to your files, and read/search to your directories.

If you set umask at the shell prompt, it will only apply to the current login session. It will not apply to future login sessions. To apply umask setting automatically at login, you would add the umask command to your .login file (C Shell users) or .profile (Bourne and Korn Shell users).

umask and level of security:

The umask command be used for setting different security levels as follows:

umask value	Security level	Effective permission (directory)
022	Permissive	755
026	Moderate	751
027	Moderate	750
077	Severe	700

Sample umask Values and File Creation Permissions:

If umask value set to	User permission	Group permission	Others permission
000	all	all	all
007	all	all	none
027	all	read / execute	none

Limitations of the umask:

1. The umask command can restricts permissions.
2. The umask command cannot grant extra permissions beyond what is specified by the program that creates the file or directory. If you need to make permission changes to existing file use the chmod command.

6. UNIX User Management (Administration)

There are three types of accounts on a UNIX system:

1. **Root account:**

This is also called superuser and would have complete and unfettered control of the system. A superuser can run any commands without any restriction. This user should be assumed as a system administrator.

2. **System accounts:**

System accounts are those needed for the operation of system-specific components for example mail accounts and the sshd accounts. These accounts are usually needed for some specific function on your system, and any modifications to them could adversely affect the system.

3. **User accounts:**

User accounts provide interactive access to the system for users and groups of users. General users are typically assigned to these accounts and usually have limited access to critical system files and directories.

UNIX supports a concept of *Group Account* which logically groups a number of accounts. Every account would be a part of any group account. UNIX group's plays important role in handling file permissions and process management.

Managing Users and Groups:

There are three main user administration files:

- 1) **/etc/passwd:** Keeps user account and password information. This file holds the majority of information about accounts on the UNIX system.
- 2) **/etc/shadow:** Holds the encrypted password of the corresponding account. Not all the system support this file.
- 3) **/etc/group:** This file contains the group information for each account.
- 4) **/etc/gshadow:** This file contains secure group account information.

Check all the above files using **cat** command.

Following are commands available on the majority of UNIX systems to create and manage accounts and groups:

useradd, usermod, userdel, groupadd, groupmod, and groupdel.

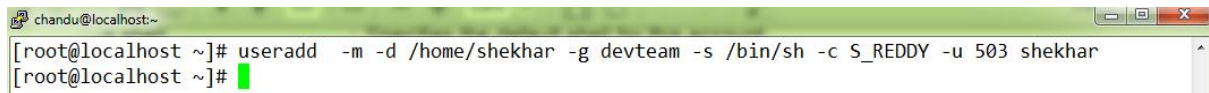
Creating a User Account:

Let us see how to create a new account on your UNIX system. Following is the syntax to create a user's account:

Syntax: useradd [option] <username>

Example:

\$ useradd shekhar

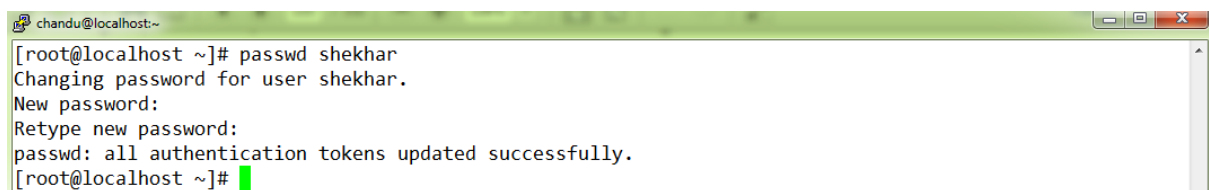
A terminal window titled 'chandu@localhost:~' showing the execution of the 'useradd' command. The prompt is '[root@localhost ~]#'. The command entered is 'useradd -m -d /home/shekhar -g devteam -s /bin/sh -c S_REDDY -u 503 shekhar'. The prompt returns to '[root@localhost ~]#' with a green cursor.

```
chandu@localhost:~  
[root@localhost ~]# useradd -m -d /home/shekhar -g devteam -s /bin/sh -c S_REDDY -u 503 shekhar  
[root@localhost ~]#
```

If you do not specify any parameter then system would use default values. The useradd command modifies the /etc/passwd, /etc/shadow, and /etc/group files and creates a home directory.

Once an account is created you can set its password using the **passwd** command as follows:

Syntax: \$ passwd <username>

A terminal window titled 'chandu@localhost:~' showing the execution of the 'passwd' command. The prompt is '[root@localhost ~]#'. The command entered is 'passwd shekhar'. The output shows 'Changing password for user shekhar.', 'New password:', 'Retype new password:', and 'passwd: all authentication tokens updated successfully.' The prompt returns to '[root@localhost ~]#' with a green cursor.

```
chandu@localhost:~  
[root@localhost ~]# passwd shekhar  
Changing password for user shekhar.  
New password:  
Retype new password:  
passwd: all authentication tokens updated successfully.  
[root@localhost ~]#
```

When you type *passwd accountname*, it gives you option to change the password provided you are super user otherwise you would be able to change just your password using the same command but without specifying your account name

Modifying User Account:

The **usermod** command enables you to make changes to an existing account from the command line. It uses the same arguments as the useradd command, plus **-l argument**, which allows you to change the account name.

Syntax: \$ usermod [option] <username>

For example, to change the account home directory *shekhar* to **reddy** you would need to issue following command:

\$ usermod -d /home/reddy -m shekhar

A terminal window titled 'chandu@localhost:~' showing the execution of the 'usermod' command. The prompt is '[root@localhost ~]#'. The command entered is 'usermod -d /home/reddy -m shekhar'. The prompt returns to '[root@localhost ~]#' with a green cursor.

```
chandu@localhost:~  
[root@localhost ~]# usermod -d /home/reddy -m shekhar  
[root@localhost ~]#
```

Here is the details of the parameters:

Option	Description
-b, --base-dir	The default base directory for the system if -d HOME_DIR is not specified. BASE_DIR is concatenated with the account name to define the home directory. The BASE_DIR must exist otherwise the home directory cannot be created.
-c, --comment	Any text string. It is generally a short description of the login, and is currently used as the field for the user's full name.
-d, --home	Specifies home directory for the account.
-e, --expiredate	The date on which the user account will be disabled. The date is specified in the format YYYY-MM-DD . If not specified, useradd will use the default expiry date specified by the EXPIRE variable in /etc/default/useradd , or an empty string (no expiry) by default.
-f, --inactive	<p>The number of days after a password expires until the account is permanently disabled. A value of 0 disables the account as soon as the password has expired, and a value of -1 disables the feature.</p> <p>If not specified, useradd will use the default inactivity period specified by the INACTIVE variable in /etc/default/useradd, or -1 by default.</p>
-g, --gid	Specifies a group account for this account.
-G, --groups	A list of supplementary groups which the user is also a member of. Each group is separated from the next by a comma, with no intervening whitespace. The groups are subject to the same restrictions as the group given with the -g option. The default is for the user to belong only to the initial group.
-k, --skel	<p>The skeleton directory, which contains files and directories to be copied in the user's home directory, when the home directory is created by useradd.</p> <p>This option is only valid if the -m (or --create-home) option is specified.</p> <p>If this option is not set, the skeleton directory is defined by the SKEL variable in /etc/default/useradd or, by default, /etc/skel.</p>
-m, --create-home	Creates the home directory if it does not exist.
-M	Do not create the user's home directory, even if the system wide setting from /etc/login.defs (CREATE_HOME) is set to yes.
-p, --password	The encrypted password, as returned by crypt(3) . The default is to disable the password.
-r, --system	Create a system account.

-s, --shell	Specifies the default shell for this account.
-u, --uid	You can specify a userid for this account.
-U, --user-group	Create a group with the same name as the user, and add the user to this group.
-z, --selinux-user	The SELinux user for the user's login. The default is to leave this field blank, which causes the system to select the default SELinux user.

Deleting User Account:

The **userdel** command can be used to delete an existing user. This is a very dangerous command if not used with caution.

Syntax: \$ **userdel** [option] <username>

userdel command options:

Option	Description
-f, --force	This option forces the removal of the user account, even if the user is still logged in. It also forces userdel to remove the user's home directory and mail spool, even if another user uses the same home directory or if the mail spool is not owned by the specified user. Note: This option is dangerous and may leave your system in an inconsistent state.
-r, --remove	Files in the user's home directory will be removed along with the home directory.
-z, --selinux-user	Remove SELinux user assigned to the user's login from SELinux login mapping.

For example, to remove account *shekhar*, you would need to issue following command:

```
chandu@localhost:~$ userdel -r shekhar
[root@localhost ~]#
```

If you want to keep his home directory for backup purposes, omit the **-r** option. You can remove the home directory as needed at a later time.

Creating a Group:

You would need to create groups before creating any account otherwise you would have to use existing groups at your system. You would have all the groups listed in */etc/groups* file.

All the default groups would be system account specific groups and it is not recommended to use them for ordinary accounts.

Syntax: \$ groupadd [option] <groupname>

If you do not specify any parameter then system would use default values.

Example:

\$ groupadd devteam

groupadd command options:

Option	Description
-f, --force	This option causes the command to simply exit with success status if the specified group already exists. When used with -g, and the specified GID already exists, another (unique) GID is chosen (i.e. -g is turned off).
-g, --gid	The numerical value of the group's ID. This value must be unique, unless the -o option is used. The value must be non-negative.
-o, --non-unique	This option permits to add a group with a non-unique GID.
-p, --password	The encrypted password, as returned by crypt(3). The default is to disable the password. Note: This option is not recommended because the password (or encrypted password) will be visible by users listing the processes. You should make sure the password respects the system's password policy.
-r, --system	Create a system group.

Example: with parameters (options).

```
chandu@localhost:~  
[root@localhost ~]# groupadd -g 502 -f qateam  
[root@localhost ~]#
```

Modifying the Group:

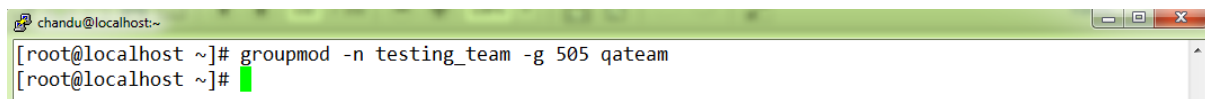
To modify the group, use the **groupmod** command.

Syntax: \$ groupmod [option] <groupname>

groupmod command options:

Option	Description
-g, --gid	The group ID of the given GROUP will be changed to GID.
-n, --new-name	The name of the group will be changed from GROUP to NEW_GROUP name.
-o, --non-unique	This option permits to add a group with a non-unique GID.

Example:



```
chandu@localhost:~  
[root@localhost ~]# groupmod -n testing_team -g 505 qateam  
[root@localhost ~]#
```

Delete a Group:

To delete an existing group, all you need are the `groupdel` command and the group name.

Syntax: `$ groupdel <groupname>`

Example:



```
chandu@localhost:~  
[root@localhost ~]# groupdel testing_team  
[root@localhost ~]#
```

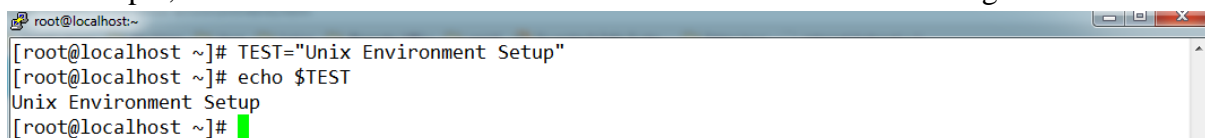
This removes only the group, not any files associated with that group. The files are still accessible by their owners.

7. UNIX Environment Setup

An important UNIX concept is the **environment**, which is defined by environment variables. Some are set by the system, others by you, yet others by the shell, or any program that loads another program.

A variable is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data.

For example, first we set a variables `TEST` and then we access its value using **echo** command:



```
root@localhost:~  
[root@localhost ~]# TEST="Unix Environment Setup"  
[root@localhost ~]# echo $TEST  
Unix Environment Setup  
[root@localhost ~]#
```

Note: The environment variables are set without using \$ sign but while accessing them we use **\$sign** as prefix. These variables retain their values until we come out shell.

When you login to the system, the shell undergoes a phase called initialization to set up various environment. This is usually a two-step process that involves the shell reading the following files:

- `/etc/profile`
- `.bash_profile`

The process is as follows:

- ❖ The shell checks to see whether the file `/etc/profile` exists.
- ❖ If it exists, the shell reads it. Otherwise, this file is skipped. No error message is displayed.

- ❖ The shell checks to see whether the file **.bash_profile** exists in your home directory. Your home directory is the directory that you start out in after you log in.
- ❖ If it exists, the shell reads it; otherwise, the shell skips it. No error message is displayed.

As soon as both of these files have been read, the shell displays a prompt:



This is the prompt where you can enter commands in order to have them execute.

Note: The shell initialization process detailed here applies to all **Bourne** type shells, but some additional files are used by **bash** and **ksh**.

The **.bash_profile** file:

The file **/etc/profile** is maintained by the system administrator of your UNIX machine and contains shell initialization information required by all users on a system.

The file **.bash_profile** is under your control. You can add as much shell customization information as you want to this file. The minimum set of information that you need to configure includes

- The type of terminal you are using
- A list of directories in which to locate commands
- A list of variables effecting look and feel of your terminal.

You can check your **.bash_profile** available in your home directory. Open it using **vi** editor and check all the variables set for your environment.

Setting the Terminal Type:

Usually the type of terminal you are using is automatically configured by either the **login** or **getty** programs. Sometimes, the auto configuration process guesses your terminal incorrectly.

If your terminal is set incorrectly, the output of commands might look strange, or you might not be able to interact with the shell properly.

To make sure that this is not the case, most users set their terminal to the lowest common denominator as follows:

```
etenv TERM vt100      ...csh, tcsh
TERM=vt100; export TERM  ...sh, ksh, zsh
export TERM=vt100      ...pdksh, bash, zsh
```

Setting the PATH:

When you type any command on command prompt, the shell has to locate the command before it can be executed.

The PATH variable specifies the locations in which the shell should look for commands. Usually it is set as follows:

A terminal window titled 'chandu@localhost:~' showing the command 'PATH=/bin:/usr/bin' being entered and executed. The prompt changes from root to ~ after execution.

```
chandu@localhost:~  
[root@localhost ~]# PATH=/bin:/usr/bin  
[root@localhost ~]#
```

Here each of the individual entries separated by the colon character, :, are directories. If you request the shell to execute a command and it cannot find it in any of the directories given in the PATH variable, a message similar to the following appears:

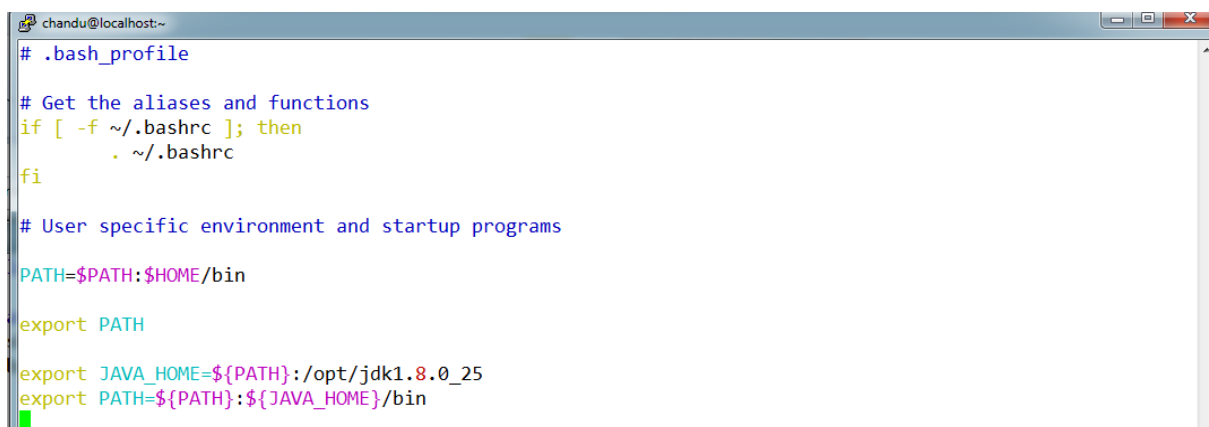
A terminal window titled 'chandu@localhost:~' showing attempts to run 'unix' and 'java' commands, both resulting in 'command not found' errors.

```
chandu@localhost:~  
[root@localhost ~]# unix  
bash: unix: command not found  
[root@localhost ~]# java  
bash: java: command not found  
[root@localhost ~]#
```

Example:

Configuring environment variables for java.

\$ vim .bash_profile

A terminal window titled 'chandu@localhost:~' showing the contents of the .bash_profile file. The file includes comments and commands to source .bashrc, set the PATH variable to include \$HOME/bin, and export JAVA_HOME and PATH to include the JDK bin directory.

```
chandu@localhost:~  
# .bash_profile  
  
# Get the aliases and functions  
if [ -f ~/.bashrc ]; then  
    . ~/.bashrc  
fi  
  
# User specific environment and startup programs  
  
PATH=$PATH:$HOME/bin  
  
export PATH  
  
export JAVA_HOME=${PATH}:/opt/jdk1.8.0_25  
export PATH=${PATH}:${JAVA_HOME}/bin
```

In .bash_profile file configure java path where your java (jdk1.8.0.25) directory existed. The path can be configured in other way also, but it is recommended.

```
chandu@localhost:~$ vim .bash_profile
[chandu@localhost ~]$ source .bash_profile
[chandu@localhost ~]$ echo $JAVA_HOME
/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/sbin:/home/chandu/bin:/home/chandu/bin:/opt/jdk1
.8.0_25
[chandu@localhost ~]$
```

A list of the commonly used variables in Linux:

System Variable	Meaning	View Variable & Value Type
BASH_VERSION	Holds the version of this instance of bash.	echo \$BASH_VERSION
HOSTNAME	The name of the computer.	echo \$HOSTNAME
CDPATH	The search path for the cd command.	echo \$CDPATH
HISTFILE	The name of the file in which command history is saved.	echo \$HISTFILE
HISTFILESIZE	The maximum number of lines contained in the history file.	echo \$HISTFILESIZE
HISTSIZE	The number of commands to remember in the command history. The default value is 500.	echo \$HISTSIZE
HOME	The home directory of the current user.	echo \$HOME
IFS	The Internal Field Separator that is used for word splitting after expansion and to split lines into words with the read built-in command. The default value is <space> <tab><newline>.	echo \$IFS
LANG	Used to determine the locale category for any category not specifically selected with a variable starting with LC_.	echo \$LANG
PATH	The search path for commands. It is a colon-separated list of directories in which the shell looks for commands.	echo \$PATH
PS1	Your prompt settings.	echo \$PS1
TMOUT	The default timeout for the read built-in command. Also in an interactive shell, the value is interpreted as the number of seconds to wait for input after issuing the	echo \$TMOUT

	command. If not input provided it will logout user.	
TERM	Your login terminal type.	echo \$TERM export TERM=vt100
SHELL	Set path to login shell.	echo \$SHELL
DISPLAY	Set X display name	echo \$DISPLAY export DISPLAY=:0.1
EDITOR	Set name of default text editor.	export EDITOR=/usr/bin/vim
HOME	Indicates the home directory of the current user; the default argument for the cd built-in command	echo \$HOME
RANDOM	Generates a random integer number between 0 and 32,767 each time it is referenced.	echo \$RANDOM
SHLVL	Increments by one each time an instance of bash is started.	
TZ	Refers to Time Zone. It can take values like GMT, AST, etc.	echo \$TZ
UID	Expand the numeric user ID of the current user, initialized at shell startup.	echo \$UID

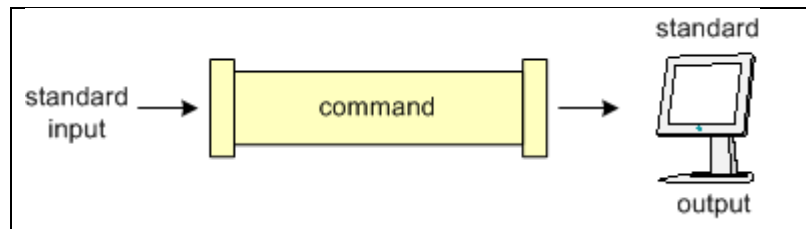
8. UNIX IO Redirections

Most UNIX system commands take input from your terminal and send the resulting output back to your terminal. A command normally reads its input from a place called standard input, which happens to be your terminal by default. Similarly, a command normally writes its output to standard output, which is also your terminal by default.

Simple redirections:

Most Linux commands read input, such as a file or another attribute for the command, and write output. By default, input is being given with the keyboard, and output is displayed on your screen. Your keyboard is your *standard input* (stdin) device, and the screen or a particular terminal window is the *standard output* (stdout) device.

However, since Linux is a flexible system, these default settings don't necessarily have to be applied. The standard output, for example, on a heavily monitored server in a large environment may be a printer.



Output Redirection:

The output from a command normally intended for standard output can be easily diverted to a file instead. This capability is known as output redirection:

If the notation `> file` is appended to any command that normally writes its output to standard output, the output of that command will be written to file instead of your terminal:

Check following **who** command which would redirect complete output of the command in users file.

```
chandu@localhost:~  
[chandu@localhost ~]$ who > users  
[chandu@localhost ~]$
```

Notice that no output appears at the terminal. This is because the output has been redirected from the default standard output device (the terminal) into the specified file. If you would check *users* file then it would have complete content:

```
chandu@localhost:~  
[chandu@localhost ~]$ who > users  
[chandu@localhost ~]$ cat users  
root    pts/0      2015-02-04 23:57 (192.168.188.1)  
chandu  pts/1      2015-02-04 23:58 (192.168.188.1)  
[chandu@localhost ~]$
```

If a command has its output redirected to a file and the file already contains some data, that data will be lost. Consider this example:

```
chandu@localhost:~  
[chandu@localhost ~]$ echo Hello > users  
[chandu@localhost ~]$ cat users  
Hello  
[chandu@localhost ~]$
```

You can use **>> operator** to append the output in an existing file as follows:

```
chandu@localhost:~  
[chandu@localhost ~]$ echo World >> users  
[chandu@localhost ~]$ cat users  
Hello  
World  
[chandu@localhost ~]$
```

Input Redirection:

Just as the output of a command can be redirected to a file, so can the input of a command be redirected from a file. As the **greater-than character** `>` is used for output redirection, the **less-than character** `<` is used to redirect the input of a command.

The commands that normally take their input from standard input can have their input redirected from a file in this manner. For example, to count the number of lines in the file *users* generated above, you can execute the command as follows:

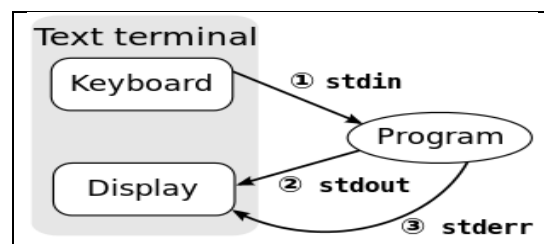
```
chandu@localhost:~  
[chandu@localhost ~]$ wc -l users  
2 users  
[chandu@localhost ~]$
```

Here it produces output 2 lines. You can count the number of lines in the file by redirecting the standard input of the **wc** command from the file *users*:

```
chandu@localhost:~  
[chandu@localhost ~]$ wc -l < users  
2  
[chandu@localhost ~]$
```

Note that there is a difference in the output produced by the two forms of the **wc** command. In the first case, the name of the file *users* is listed with the line count; in the second case, it is not.

In the first case, **wc** knows that it is reading its input from the file *users*. In the second case, it only knows that it is reading its input from standard input so it does not display file name.



File Descriptors:

There are three types of I/O, which each have their own identifier, called a file descriptor:

1. Standard Input: 0
2. Standard Output: 1
3. Standard Error: 2

In the following descriptions, if the file descriptor number is omitted, and the first character of the redirection operator is **<**, the redirection refers to the standard input (file descriptor 0). If the first character of the redirection operator is **>**, the redirection refers to the standard output (file descriptor 1).

Some practical examples will make this clearer:

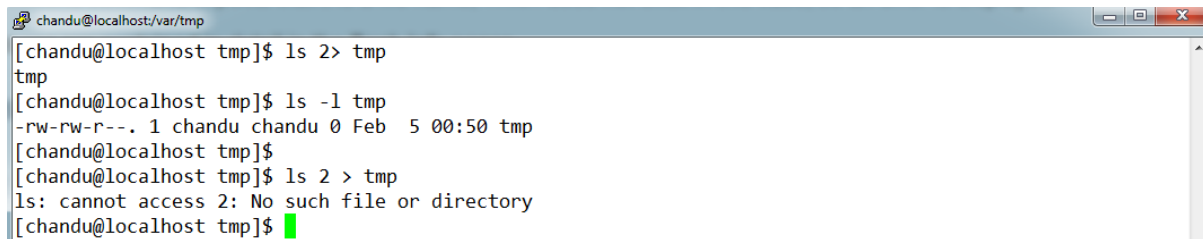
\$ ls > dirlist 2>&1

Will direct both standard output and standard error to the file *dirlist*, while the command

\$ ls 2>&1 > dirlist

Will only direct standard output to *dirlist*. This can be a useful option for programmers.

Things are getting quite complicated here, don't confuse the use of the ampersand here with the use of it in the section called “Interactive processes”, where the ampersand is used to run a process in the background. Here, it merely serves as an indication that the number that follows is not a file name, but rather a location that the data stream is pointed to. Also note that the bigger-than sign should not be separated by spaces from the number of the file descriptor. If it would be separated, we would be pointing the output to a file again. The example below demonstrates this:



```
chandu@localhost:~/var/tmp
[chandu@localhost tmp]$ ls 2> tmp
tmp
[chandu@localhost tmp]$ ls -l tmp
-rw-rw-r--. 1 chandu chandu 0 Feb  5 00:50 tmp
[chandu@localhost tmp]$
[chandu@localhost tmp]$ ls 2 > tmp
ls: cannot access 2: No such file or directory
[chandu@localhost tmp]$
```

The first command that *Nancy* executes is correct (even though no errors are generated and thus the file to which standard error is redirected is empty). The second command expects that 2 is a file name, which does not exist in this case, so an error is displayed.

Discard output:

Sometimes you will need to execute a command, but you don't want the output displayed to the screen. In such cases you can discard the output by redirecting it to the file `/dev/null`.



```
chandu@localhost:~
[chandu@localhost ~]$ cat users > /dev/null
[chandu@localhost ~]$ echo Hai > /dev/null
[chandu@localhost ~]$
```

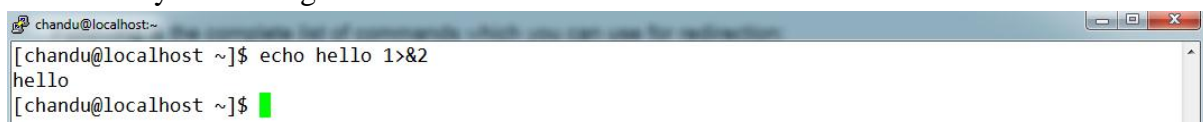
The file `/dev/null` is a special file that automatically discards all its input.

To discard both output of a command and its error output, use standard redirection to redirect `STDERR` to `STDOUT`:



```
chandu@localhost:~
[chandu@localhost ~]$ chandu
-bash: chandu: command not found
[chandu@localhost ~]$ chandu > /dev/null 2>&1
[chandu@localhost ~]$
```

Here 2 represents `STDERR` and 1 represents `STDOUT`. You can display a message on to `STDERR` by redirecting `STDIN` into `STDERR` as follows:



```
chandu@localhost:~
[chandu@localhost ~]$ echo hello 1>&2
hello
[chandu@localhost ~]$
```

Redirecting to Multiple outputs:

The standard command **tee** can redirect output from a command to several destinations.

```
chandu@localhost:~$ ls -lrt | tee chandu
total 4
-rw-rw-r--. 1 chandu chandu 12 Feb  5 00:26 users
-rw-rw-r--. 1 chandu chandu  0 Feb  5 01:16 chandu
chandu@localhost ~]$
```

This directs the file list output to both standard output and the file chandu.

Redirection Commands:

Following is the complete list of commands which you can use for redirection:

Command	Description
pgm > file	Output of pgm redirected to file.
pgm < file	Program pgm reads its input from file.
pgm >> file	Output of pgm is appended to file.
n > file	Output from stream with descriptor n appended to file.
n >> file	Output from stream with descriptor n appended to file.
n >& m	Merge output from stream n with stream m.
n <& m	Merge input from stream n with stream m.
<< tag	Standard input comes from here through next tag at start of line.
 	Takes output from one, or process, and sends it to another.

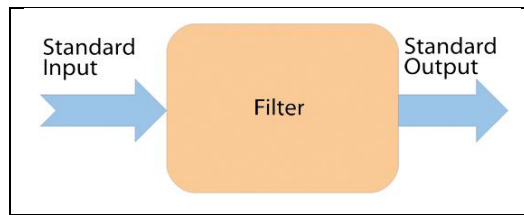
Note that file descriptor 0 is normally standard input (STDIN), 1 is standard output (STDOUT), and 2 is standard error output (STDERR).

9. UNIX Pipes & Filters

When many newbies first encounter Linux, the 'cool stuff' that often gets their attention is the incredible array of command line tools, and something called a pipe that allowed you to connect them together. Together, these provide an incredibly powerful component-based architecture designed to process streams of text-based data.

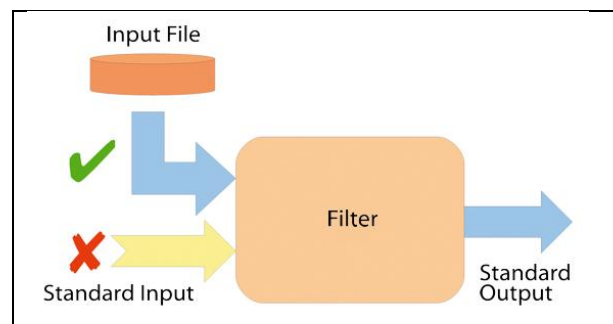
If you've never dabbled with filters and pipes before, or perhaps you've just been too scared, we want to help you out, so read on to learn how you can make powerful Linux commands just by stringing smaller bits together.

A filter is a program that reads a single input stream, transforms it in some way, and writes the result to a single output stream, as shown in Figure below. By default, the output stream (called standard output or just stdout) is connected to the terminal window that the program is running in, and the input stream (standard input, or just stdin) is connected to the keyboard, though in practice filters are rarely used to process data typed in manually at the keyboard.



A filter has a single input stream and a single output stream.

If a filter is given a file name as a command line argument, it will open that file and read from it instead of reading from stdin, as shown in Figure below. This is a much more common arrangement. There are a few sample commands in the box opposite. By themselves, many individual filters don't do anything exciting. It's what you can do with them in combination that gets interesting.



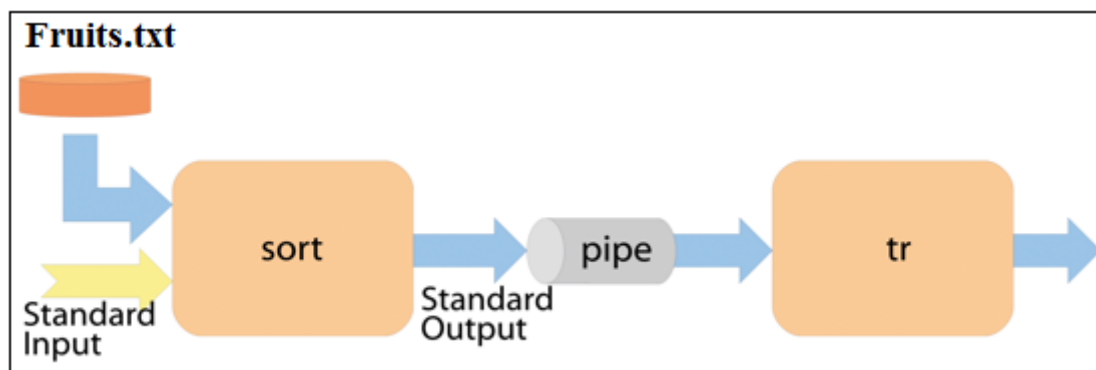
Given a filename, a filter reads it, ignoring its standard input.

Combining Pipes & Filters:

A pipe allows data to flow from an output stream in one process to an input stream in another process. Usually, it's used to connect a standard output stream to a standard input stream. Pipes are very easy to create at the shell command line using the | (vertical bar) symbol. For example, when the shell sees this command line:

```
$ sort Fruits.txt | tr '[A-Z]' '[a-z]'
```

It runs the programs sort and tr in two separate, concurrent processes, and creates a pipe that connects the standard output of sort (the 'upstream' process) to the standard input of tr (the 'downstream' process). The plumbing for this is shown in Figure below.



Plumbing for the command "sort Fruits.txt | tr '[A-Z]' '[a-z]'".

```
chandu@localhost:~$ cat Fruits.txt
Apple
Avocado
Banana
Cherry
[chandu@localhost ~]$ sort Fruits.txt | tr '[A-Z]' '[a-z]'
apple
avocado
banana
cherry
[chandu@localhost ~]$
```

This particular tr command, replaces all characters in the set [A-Z] by the corresponding character from the set [a-z]; that is, it translates upper-case to lower-case.

Capturing Standard Output:

If you need that changes permanently, just redirect that output to another file using IO Redirections.

Example: `$ sort Fruits.txt | tr '[A-Z]' '[a-z]' > changed_fruits.txt`

```
chandu@localhost:~$ sort Fruits.txt | tr '[A-Z]' '[a-z]' > changed_fruits.txt
[chandu@localhost ~]$ cat changed_fruits.txt
apple
avocado
banana
cherry
[chandu@localhost ~]$
```

Semi-filters

Linux has many command-line tools that aren't really filters, but which nonetheless write their results to stdout and so can be used at the head of a pipeline. Examples include ls, ps, df, du, but there are lots of others. So for example:

```
chandu@localhost:~$ ps -ef | wc
  114    1001    8327
[chandu@localhost ~]$
```

Will count the number of processes running on the machine.

Commonly used Filters:

Filter	What is does
cat	Copies input direct to output.
head	Shows beginning of a file (default 10 lines).
tail	Shows ending of a file (default 10 lines).
wc	Counts characters, words and lines.
sort	Sorts input lines.
grep	Shows lines that match a regular expressions.
tr	Translates or deletes specified character sets.
sed	Stream editor.

uniq	Discards all but one of successive identical lines.
awk	Highly programmable field-processing.

grep command:

grep searches the input files for lines containing a match to a given pattern list. When it finds a match in a line, it copies the line to standard output (by default), or whatever other sort of output you have requested with options.

Though grep expects to do the matching on text, it has no limits on input line length other than available memory, and it can match arbitrary characters within a line. If the final byte of an input file is not a *newline*, grep silently supplies one. Since newline is also a separator for the list of patterns, there is no way to match newline characters in a text.

Syntax: `$ grep [option] [pattern] <file>`

```

root@localhost:~/backup
[root@localhost backup]# grep ^root /etc/passwd
root:x:0:0:root:/root:/bin/bash
[root@localhost backup]#

```

First create the following demo.txt that will be used in the examples below to demonstrate grep command.

```

chandu@localhost:~
[chandu@localhost ~]$ cat demo.txt
THIS LINE IS THE 1ST UPPER CASE LINE IN THIS FILE.
this line is the 1st lower case line in this file.
This Line Has All Its First Character Of The Word With Upper Case.

Two lines above this line is empty.
And this is the last line.

[chandu@localhost ~]$

```

Search for the given string in a single file

The basic usage of grep command is to search for a specific string in the specified file as shown below.

Syntax: `$ grep "literal string" <filename>`

Example:

```

chandu@localhost:~
[chandu@localhost ~]$ grep "this" demo.txt
this line is the 1st lower case line in this file.
Two lines above this line is empty.
And this is the last line.
[chandu@localhost ~]$

```

Checking for the given string in multiple files.

This is also a basic usage of grep command. For this example, let us copy the demo.txt to demo2.txt. The grep output will also include the file name in front of the line that matched the

specific pattern as shown below. When the Linux shell sees the Meta character, it does the expansion and gives all the files as input to grep.

Syntax: \$ grep “string” <FILE_PATTERN>

Example:

```
chandu@localhost:~$ grep "lower" demo*
demo2.txt:this line is the 1st lower case line in this file.
demo.txt:this line is the 1st lower case line in this file.
[chandu@localhost ~]$
```

Case insensitive search using grep -i.

This is also a basic usage of the grep. This searches for the given string/pattern case insensitively. So it matches all the words such as “the”, “THE” and “The” case insensitively as shown below.

Syntax: \$ grep -i “String” <File>

Example:

```
chandu@localhost:~$ grep -i "the" demo.txt
THIS LINE IS THE 1ST UPPER CASE LINE IN THIS FILE.
this line is the 1st lower case line in this file.
This Line Has All Its First Character Of The Word With Upper Case.
And this is the last line.
[chandu@localhost ~]$
```

Match regular expression in files.

This is a very powerful feature, if you can use regular expression effectively. In the following example, it searches for all the pattern that starts with “lines” and ends with “empty” with anything in-between. i.e. To search “lines [anything in-between]empty” in the demo.txt.

Syntax: grep “REGEX” <filename>

Example:

```
chandu@localhost:~$ grep "lines.*empty" demo.txt
Two lines above this line is empty.
[chandu@localhost ~]$
```

A regular expression may be followed by one of several repetition operators:

Regex	Description
?	The preceding item is optional and matched at most once .
*	The preceding item will be matched zero or more times.
+	The preceding item will be matched one or more times.
{n}	The preceding item is matched exactly n times .
{n,}	The preceding item is matched n or more times .
{,m}	The preceding item is matched at most m times .

{n,m}	The preceding item is matched atleast n times , but not more than m times.
--------------	--

Look into the man page for the more options of grep command:

Syntax: `$ man [command]`

Example

`$ man grep`

sed command:

Sed is a Stream Editor used for modifying the files in UNIX (or Linux). Whenever you want to make changes to the file automatically, sed comes in handy to do this. Most people never learn its power; they just simply use sed to replace text. You can do many things apart from replacing text with sed. Here I will describe the features of sed with examples.

Syntax: `$ sed [option] [pattern] <filename>`

Consider the below text file as an input.

```
chandu@localhost:~$ cat file.txt
unix is great os. unix is opensource. unix is free os.
learn operating system.
unixlinux which one you choose.
[chandu@localhost ~]$
```

Replacing or substituting string.

Sed command is mostly used to replace the text in a file. The below simple sed command replaces the word "Unix" with "Linux" in the file.

```
chandu@localhost:~$ sed 's/unix/linux/' file.txt
linux is great os. unix is opensource. unix is free os.
learn operating system.
linuxlinux which one you choose.
[chandu@localhost ~]$
```

Here the "s" specifies the substitution operation. The "/" are delimiters. The "unix" is the search pattern and the "Linux" is the replacement string.

By default, the sed command replaces the first occurrence of the pattern in each line and it won't replace the second, third...occurrence in the line.

Replacing the nth occurrence of a pattern in a line.

Use the /1, /2 etc flags to replace the first, second occurrence of a pattern in a line. The below command replaces the second occurrence of the word "Unix" with "Linux" in a line.

```

chandu@localhost:~$ sed 's/unix/linux/2' file.txt
unix is great os. linux is opensource. unix is free os.
learn operating system.
unixlinux which one you choose.
[chandu@localhost ~]$

```

Replacing all the occurrence of the pattern in a line.

The substitute flag /g (global replacement) specifies the sed command to replace all the occurrences of the string in the line.

```

chandu@localhost:~$ sed 's/unix/linux/g' file.txt
linux is great os. linux is opensource. linux is free os.
learn operating system.
linuxlinux which one you choose.
[chandu@localhost ~]$

```

Using & as the matched string.

There might be cases where you want to search for the pattern and replace that pattern by adding some extra characters to it. In such cases & comes in handy. The & represents the matched string.

```

chandu@localhost:~$ sed 's/unix/{&}/' file.txt
{unix} is great os. unix is opensource. unix is free os.
learn operating system.
{unix}linux which one you choose.
[chandu@localhost ~]$
chandu@localhost:~$ sed 's/unix/{&&}/' file.txt
{unixunix} is great os. unix is opensource. unix is free os.
learn operating system.
{unixunix}linux which one you choose.
[chandu@localhost ~]$

```

Deleting lines.

You can delete the lines a file by specifying the line number or a range or numbers.

```

chandu@localhost:~$ sed '2 d' file.txt
unix is great os. unix is opensource. unix is free os.
unixlinux which one you choose.
[chandu@localhost ~]$

```

For more options and information see the **man page**.

uniq Command:

uniq command is helpful to remove or detect duplicate entries in a file.

Syntax: \$ uniq [options]

The following test file is used in some of the example to understand how uniq command works.


```
chandu@localhost:~  
[chandu@localhost ~]$ cat test  
aa  
aa  
bb  
bb  
bb  
xx  
[chandu@localhost ~]$
```

For example, when `uniq` command is run without any option, it removes duplicate lines and displays unique lines as shown below.

```
chandu@localhost:~  
[chandu@localhost ~]$ uniq test  
aa  
bb  
xx  
[chandu@localhost ~]$
```

uniq command options:

Option	Description
-c, --count	Prefix lines by the number of occurrences.
-d, --repeated	Only print duplicate lines.
-i, --ignore-case	Ignore differences in case when comparing.
-s, --skip-chars=N	Avoid comparing the first N characters.
-u, --unique	Only print unique lines.
-w, --check-chars=N	Compare no more than N characters in lines.

awk command:

Awk is one of the most powerful tools in UNIX used for processing the rows and columns in a file. Awk has built in string functions and associative arrays. Awk supports most of the operators, conditional blocks, and loops available in C language.

One of the good things is that you can convert Awk scripts into Perl scripts using `a2p` utility.

Syntax: `$ awk 'BEGIN {start_action} {action} END {stop_action}' <filename>`

Here the actions in the begin block are performed before processing the file and the actions in the end block are performed after processing the file. The rest of the actions are performed while processing the file.

Examples:

Create a file **test** with the following data. This file can be easily created using the output of `ls -l`.

```
$ ls -l | cat > test
```

```
chandu@localhost:~  
[chandu@localhost ~]$ ls -l  
total 16  
-rw-rw-r--. 1 chandu chandu 235 Feb 10 06:26 demo2.txt  
-rw-rw-r--. 1 chandu chandu 235 Feb 10 06:20 demo.txt  
-rw-rw-r--. 1 chandu chandu 111 Feb 10 07:01 file.txt  
-rw-rw-r--. 1 chandu chandu 222 Feb 10 09:19 test  
[chandu@localhost ~]$
```

From the data, you can observe that this file has rows and columns. The rows are separated by a new line character and the columns are separated by a space characters. We will use this file as the input for the examples discussed here.

Extracting the Columns:

\$ awk '{print \$1}' <filename>

Here \$1 has a meaning. \$1, \$2, \$3... Represents the first, second, third columns... in a row respectively. This awk command will print the first column in each row as shown below.

```
chandu@localhost:~  
[chandu@localhost ~]$ awk '{print $1}' test  
total  
-rw-rw-r--.  
-rw-rw-r--.  
-rw-rw-r--.  
-rw-rw-r--.  
[chandu@localhost ~]$
```

Here is best real-time example:

I want to get the used disk space percentage, from this it has to raise the warning if it exceeds 80%. So in this case we need to extract the single value from it.

Carefully watch the below example. It is the combination of commands used.

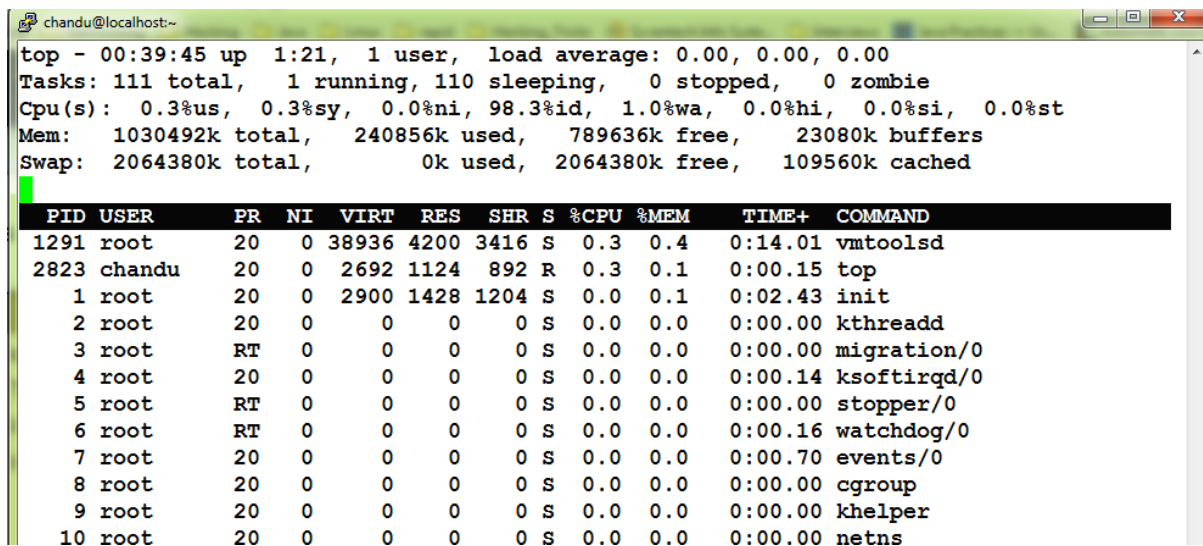
```
chandu@localhost:~  
[chandu@localhost ~]$ df -h  
Filesystem      Size  Used Avail Use% Mounted on  
/dev/sda2       18G   4.3G   13G   26% /  
tmpfs           504M   72K   504M   1% /dev/shm  
/dev/sda1       283M   50M   218M  19% /boot  
[chandu@localhost ~]$ df -h | sed -n '2p'  
/dev/sda2       18G   4.3G   13G   26% /  
[chandu@localhost ~]$ df -h | sed -n '2p' | awk '{print $5}'  
26%  
[chandu@localhost ~]$
```

10. Unix Process Management

A program is a series of instructions that tell the computer what to do. When we run a program, those instructions are copied into memory and space is allocated for variables and other stuff required to manage its execution. This running instance of a program is called a process and its processes which we manage.

Linux, like most modern OS's is a multitasking operating system. This means that many processes can be running at the same time. As well as the processes we are running, there may be other users on the system also running stuff and the OS itself will usually also be running various processes which it uses to manage everything in general. If we would like to get a snapshot of what is currently happening on the system we may use a program called **top**.

\$ top



```
top - 00:39:45 up 1:21, 1 user, load average: 0.00, 0.00, 0.00
Tasks: 111 total, 1 running, 110 sleeping, 0 stopped, 0 zombie
Cpu(s): 0.3%us, 0.3%sy, 0.0%ni, 98.3%id, 1.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 1030492k total, 240856k used, 789636k free, 23080k buffers
Swap: 2064380k total, 0k used, 2064380k free, 109560k cached

  PID USER      PR  NI  VIRT  RES  SHR  S  %CPU  %MEM    TIME+  COMMAND
 1291 root        20   0 38936 4200 3416 S   0.3   0.4   0:14.01 vmtoolsd
 2823 chandu     20   0 2692 1124  892 R   0.3   0.1   0:00.15 top
    1 root        20   0 2900 1428 1204 S   0.0   0.1   0:02.43 init
    2 root        20   0      0      0      0 S   0.0   0.0   0:00.00 kthreadd
    3 root        RT   0      0      0      0 S   0.0   0.0   0:00.00 migration/0
    4 root        20   0      0      0      0 S   0.0   0.0   0:00.14 ksoftirqd/0
    5 root        RT   0      0      0      0 S   0.0   0.0   0:00.00 stopper/0
    6 root        RT   0      0      0      0 S   0.0   0.0   0:00.16 watchdog/0
    7 root        20   0      0      0      0 S   0.0   0.0   0:00.70 events/0
    8 root        20   0      0      0      0 S   0.0   0.0   0:00.00 cgroup
    9 root        20   0      0      0      0 S   0.0   0.0   0:00.00 khelper
   10 root        20   0      0      0      0 S   0.0   0.0   0:00.00 netns
```

When you execute a program on your UNIX system, the system creates a special environment for that program. This environment contains everything needed for the system to run the program as if no other program were running on the system.

Whenever you issue a command in UNIX, it creates, or starts, a new process. When you tried out the **ls** command to list directory contents, you started a process. A process, in simple terms, is an instance of a running program.

The operating system tracks processes through a **four digit ID** number known as the **PID** or process ID. Each process in the system has a unique PID.

PID's eventually repeat because all the possible numbers are used up and the next PID rolls over and starts over. At any one time, no two processes with the same PID exist in the system because it is the PID that UNIX uses to track each process.

Starting a Process:

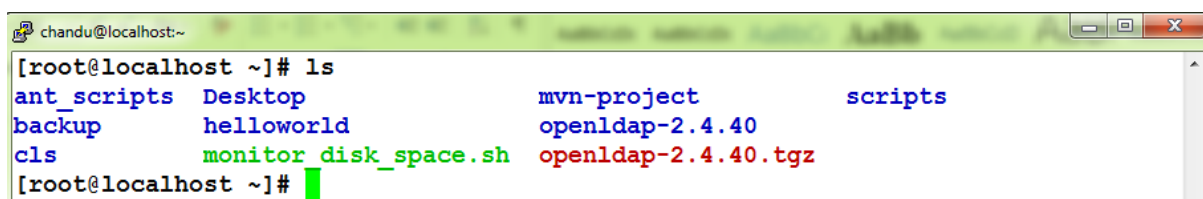
When you start a process (run a command), there are two ways you can run it:

- Foreground processes
- Background processes

Foreground Processes:

By default, every process that you start runs in the foreground. It gets its input from the keyboard and sends its output to the screen.

You can see this happen with the **ls** command. If I want to list all the files in my current directory, I can use the following command.

A terminal window titled 'chandu@localhost:~' showing the execution of the 'ls' command. The output lists files and directories in a color-coded format: 'ant_scripts' (blue), 'Desktop' (blue), 'mvn-project' (blue), and 'scripts' (blue) on the first line; 'backup' (blue), 'helloworld' (blue), 'openldap-2.4.40' (blue), and an empty space on the second line; 'cls' (blue), 'monitor_disk_space.sh' (green), and 'openldap-2.4.40.tgz' (red) on the third line. The prompt '[root@localhost ~]#' is visible at the bottom.

```
[root@localhost ~]# ls
ant_scripts Desktop mvn-project scripts
backup helloworld openldap-2.4.40
cls monitor_disk_space.sh openldap-2.4.40.tgz
[root@localhost ~]#
```

The process runs in the foreground, the output is directed to my screen, and if the **ls** command wants any input (which it does not), it waits for it from the keyboard.

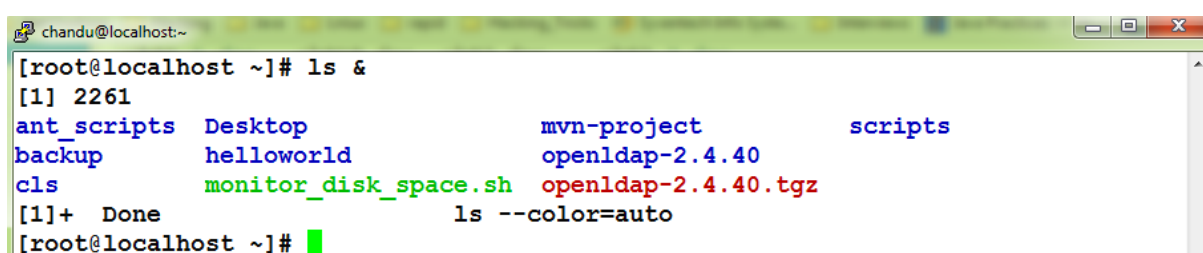
While a program is running in foreground and taking much time, we cannot run any other commands (start any other processes) because prompt would not be available until program finishes its processing and comes out.

Background Processes:

A background process runs without being connected to your keyboard. If the background process requires any keyboard input, it waits.

The advantage of running a process in the background is that you can run other commands; you do not have to wait until it completes to start another.

The simplest way to start a background process is to add an ampersand (&) at the end of the command.

A terminal window titled 'chandu@localhost:~' showing the execution of the 'ls &' command. The output is identical to the previous screenshot. Below the file list, the prompt changes to '[1] 2261' on the next line, followed by '[1]+ Done' on the line after, and then the prompt returns to '[root@localhost ~]#'. The command 'ls --color=auto' is also visible in the output.

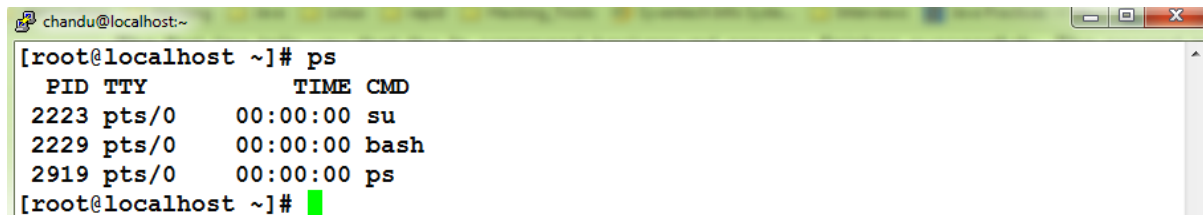
```
[root@localhost ~]# ls &
[1] 2261
ant_scripts Desktop mvn-project scripts
backup helloworld openldap-2.4.40
cls monitor_disk_space.sh openldap-2.4.40.tgz
[1]+ Done ls --color=auto
[root@localhost ~]#
```

Here if the **ls** command wants any input (which it does not), it goes into a stop state until I move it into the foreground and give it the data from the keyboard.

That first line contains information about the background process - the job number and process ID. You need to know the job number to manipulate it between background and foreground.

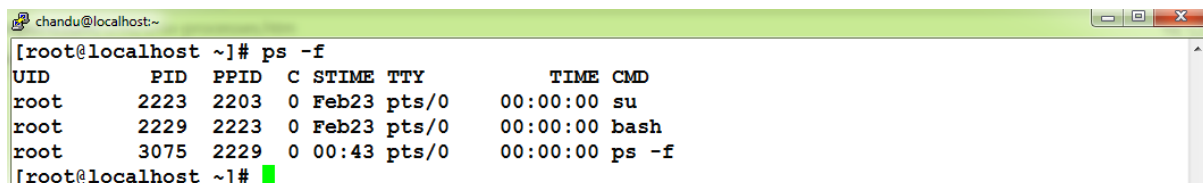
Listing Running Processes:

To list out the processes that are running by your own, you can use the **ps** (process status) command.



```
chandu@localhost:~  
[root@localhost ~]# ps  
  PID TTY          TIME CMD  
 2223 pts/0    00:00:00 su  
 2229 pts/0    00:00:00 bash  
 2919 pts/0    00:00:00 ps  
[root@localhost ~]#
```

One of the most commonly used flags for **ps** is the **-f** (f for full) option, which provides more information as shown in the following example:



```
chandu@localhost:~  
[root@localhost ~]# ps -f  
UID          PID  PPID  C  STIME TTY          TIME CMD  
root         2223   2203  0  Feb23 pts/0    00:00:00 su  
root         2229   2223  0  Feb23 pts/0    00:00:00 bash  
root         3075   2229  0  00:43 pts/0    00:00:00 ps -f  
[root@localhost ~]#
```

Here is the description of all the fields displayed by **ps -f** command:

Column	Description
UID	User ID that this process belongs to (the person running it).
PID	Process ID.
PPID	Parent process ID (the ID of the process that started it).
C	CPU utilization of process.
STIME	Process start time.
TTY	Terminal type associated with the process
TIME	CPU time taken by the process.
CMD	The command that started this process.

ps command options:

Option	Description
-a	Shows information about all users.
-e	Displays extended information.
-f	Display the full information
-u	Shows additional information like -f option.
-x	Shows information about processes without terminals.

EXAMPLES

To see every process on the system using standard syntax:

```
ps -e
```

```
ps -ef
```

```
ps -eF
```

```
ps -ely
```

To see every process on the system using BSD syntax:

```
ps ax
```

```
ps axu
```

To print a process tree:

```
ps -ejH
```

```
ps axjf
```

To get info about threads:

```
ps -eLf
```

```
ps axms
```

To get security info:

```
ps -eo euser,ruser,suser,fuser,f,comm,label
```

```
ps axZ
```

```
ps -eM
```

To see every process running as root (real & effective ID) in user

```
ps -U root -u root u
```

To see every process with a user-defined format:

```
ps -eo pid,tid,class,rtprio,ni,pri,psr,pcpu,stat,wchan:14,comm
```

```
ps axo stat,euid,ruid,tt,tpgid,sess,pgrp,ppid,pid,pcpu,comm
```

```
ps -eopid,tt,user,fname,tmout,f,wchan
```

Print only the process IDs of syslogd:

```
ps -C syslogd -o pid=
```

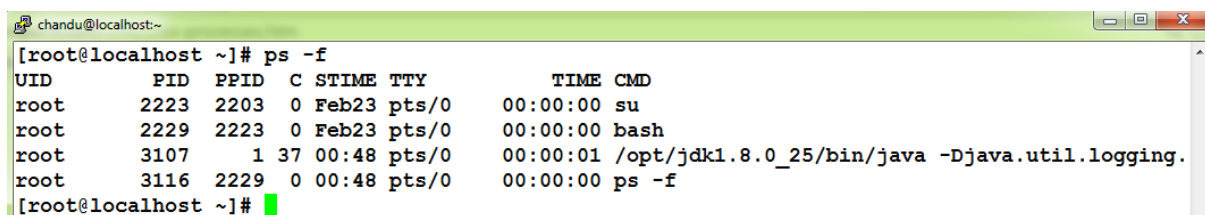
Print only the name of PID 42:

```
ps -q 42 -o comm=
```

Stopping or Killing the Processes:

Ending a process can be done in several different ways. Often, from a console-based command, sending a CTRL + C keystroke (the default interrupt character) will exit the command. This works when process is running in foreground mode.

If a process is running in background mode then first you would need to get its Job ID using **ps** command and after that you can use **kill** command to kill the process as follows:

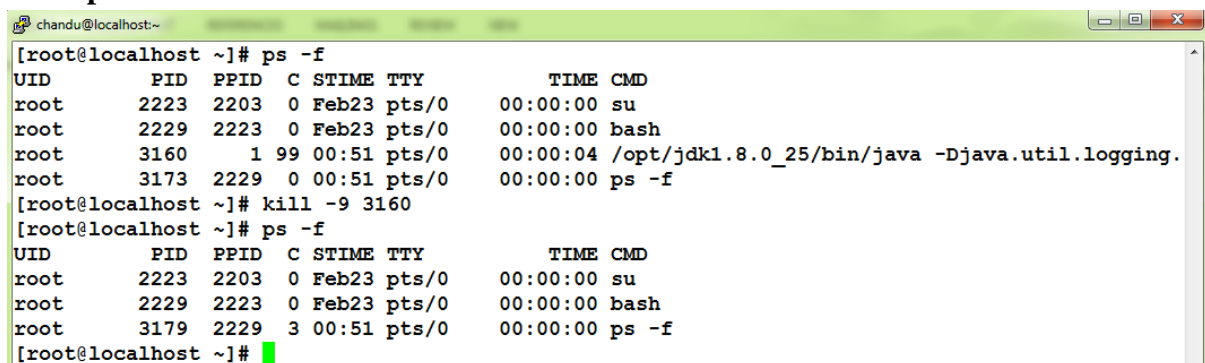


```
chandu@localhost:~  
[root@localhost ~]# ps -f  
UID      PID  PPID  C  STIME TTY          TIME CMD  
root     2223   2203  0  Feb23 pts/0      00:00:00 su  
root     2229   2223  0  Feb23 pts/0      00:00:00 bash  
root     3107     1  37  00:48 pts/0      00:00:01 /opt/jdk1.8.0_25/bin/java -Djava.util.logging.  
root     3116   2229  0  00:48 pts/0      00:00:00 ps -f  
[root@localhost ~]#
```

Here **kill** command would terminate first_one process. If a process ignores a regular kill command, you can use **kill -9** followed by the process ID as follows.

Syntax: \$ kill -9 PID

Example:



```
chandu@localhost:~  
[root@localhost ~]# ps -f  
UID      PID  PPID  C  STIME TTY          TIME CMD  
root     2223   2203  0  Feb23 pts/0      00:00:00 su  
root     2229   2223  0  Feb23 pts/0      00:00:00 bash  
root     3160     1  99  00:51 pts/0      00:00:04 /opt/jdk1.8.0_25/bin/java -Djava.util.logging.  
root     3173   2229  0  00:51 pts/0      00:00:00 ps -f  
[root@localhost ~]# kill -9 3160  
[root@localhost ~]# ps -f  
UID      PID  PPID  C  STIME TTY          TIME CMD  
root     2223   2203  0  Feb23 pts/0      00:00:00 su  
root     2229   2223  0  Feb23 pts/0      00:00:00 bash  
root     3179   2229  3  00:51 pts/0      00:00:00 ps -f  
[root@localhost ~]#
```

kill command:

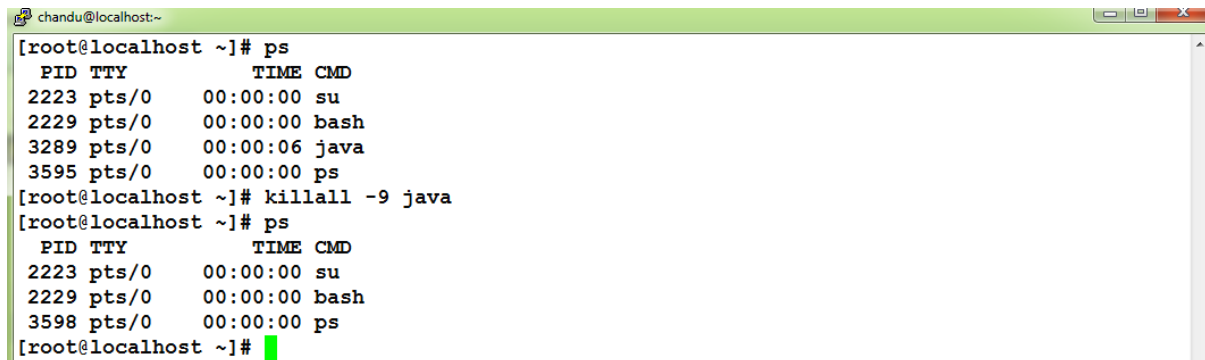
Kill command is use to send signal to a process or to kill a process. We typically use kill - SIGNAL PID, where you know the PID of the process.

Syntax: \$ kill -9 PID

killall command:

Instead of specifying a process by its PID, you can specify the name of the process. If more than one process runs with that name, all of them will be killed.

Syntax: \$ killall -9 <process name>



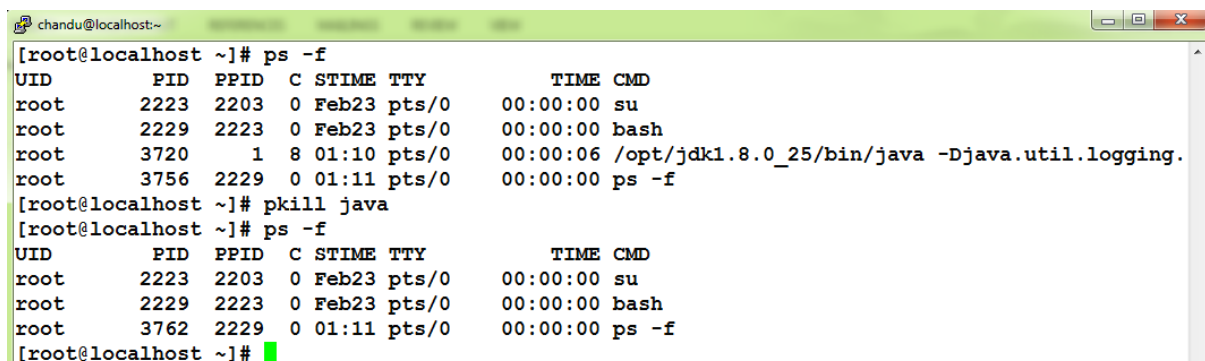
```
chandu@localhost:~  
[root@localhost ~]# ps  
  PID TTY          TIME CMD  
 2223 pts/0    00:00:00 su  
 2229 pts/0    00:00:00 bash  
 3289 pts/0    00:00:06 java  
 3595 pts/0    00:00:00 ps  
[root@localhost ~]# killall -9 java  
[root@localhost ~]# ps  
  PID TTY          TIME CMD  
 2223 pts/0    00:00:00 su  
 2229 pts/0    00:00:00 bash  
 3598 pts/0    00:00:00 ps  
[root@localhost ~]#
```

pkill command:

You can send signal to any process by specifying the full name or partial name. So there is no need for you to find out the PID of the process to send the signal.

Syntax: \$ pkill <process name>

Example:



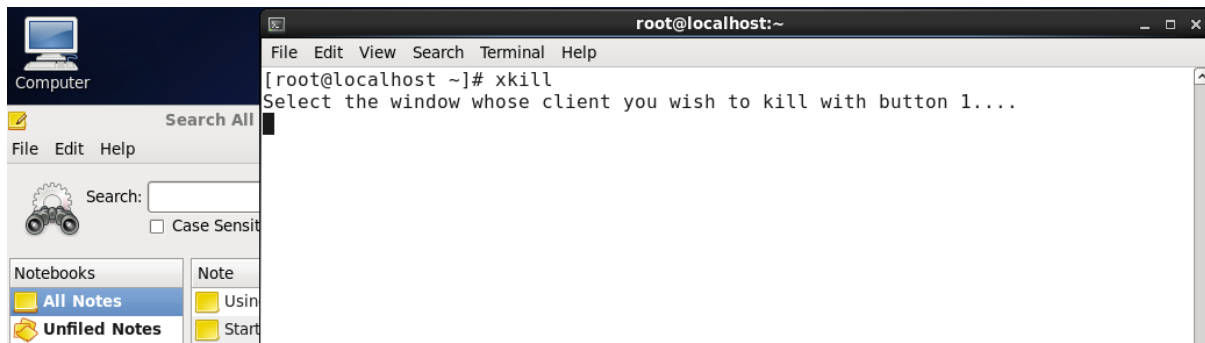
```
chandu@localhost:~  
[root@localhost ~]# ps -f  
UID          PID  PPID  C  STIME TTY          TIME CMD  
root          2223   2203  0  Feb23 pts/0    00:00:00 su  
root          2229   2223  0  Feb23 pts/0    00:00:00 bash  
root          3720     1   8  01:10 pts/0    00:00:06 /opt/jdk1.8.0_25/bin/java -Djava.util.logging.  
root          3756   2229  0  01:11 pts/0    00:00:00 ps -f  
[root@localhost ~]# pkill java  
[root@localhost ~]# ps -f  
UID          PID  PPID  C  STIME TTY          TIME CMD  
root          2223   2203  0  Feb23 pts/0    00:00:00 su  
root          2229   2223  0  Feb23 pts/0    00:00:00 bash  
root          3762   2229  0  01:11 pts/0    00:00:00 ps -f  
[root@localhost ~]#
```

xkill command:

xkill is the simplest way to kill a malfunctioning program. When you want to kill a process, initiate xkill which will offer a cross-hair cursor. Click on the window with left button which will kill that process.

Syntax: \$ xkill

Example:



Note: This command works only when you are in GUI mode.

Parent and Child Processes:

Each UNIX process has two ID numbers assigned to it: Process ID (pid) and Parent process ID (ppid). Each user process in the system has a parent process.

Most of the commands that you run have the shell as their parent. Check `ps -f` example where this command listed both process ID and parent process ID.

Zombie and Orphan Processes:

Normally, when a child process is killed, the parent process is told via a SIGCHLD signal. Then the parent can do some other task or restart a new child as needed. However, sometimes the parent process is killed before its child is killed. In this case, the "parent of all processes," **init** process, becomes the new PPID (parent process ID). Sometime these processes are called orphan process.

When a process is killed, a `ps` listing may still show the process with a Z state. This is a zombie, or defunct, process. The process is dead and not being used. These processes are different from orphan processes. They are the processes that has completed execution but still has an entry in the process table.

Daemon Processes:

Daemons are system-related background processes that often run with the permissions of root and services requests from other processes.

A daemon process has no controlling terminal. It cannot open `/dev/tty`. If you do a "`ps -ef`" and look at the `tty` field, all daemons will have a `?` for the `tty`.

More clearly, a daemon is just a process that runs in the background, usually waiting for something to happen that it is capable of working with, like a printer daemon is waiting for print commands.

If you have a program which needs to do long processing then it's worth to make it a daemon and run it in background.

Job ID versus Process ID:

Background and suspended processes are usually manipulated via job number (job ID). This number is different from the process ID and is used because it is shorter.

In addition, a job can consist of multiple processes running in series or at the same time, in parallel, so using the job ID is easier than tracking the individual processes.

11. Unix Network Management

When you work in a distributed environment then you need to communicate with remote users and you also need to access remote UNIX machines.

There are several UNIX utilities which are especially useful for users computing in a networked, distributed environment. These commands are listed below.

hostname:

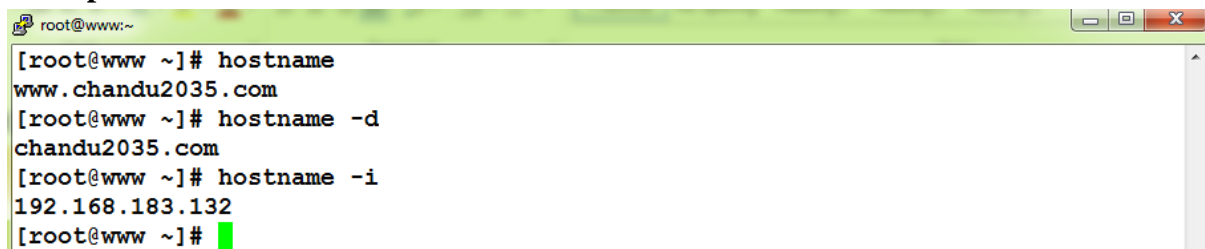
This command is used to display the host/domain name and IP address.

Syntax: `$ hostname`

Options:

Option	Description
-a, --alias	Display the alias name of the host (if used).
-d, --domain	Display the name of the DNS domain. Don't use the command domainname to get the DNS domain name because it will show the NIS domain name and not the DNS domain name. Use dnsdomainname instead.
-I, --ip-address	Display the IP address of the host.
-s, --short	Display the short host name. This is the host name cut at the first dot.

Example:

A terminal window titled 'root@www:~' showing the execution of the 'hostname' command with various options. The output shows the full domain name, the domain part only, the IP address, and the short host name.

```
root@www:~  
[root@www ~]# hostname  
www.chandu2035.com  
[root@www ~]# hostname -d  
chandu2035.com  
[root@www ~]# hostname -i  
192.168.183.132  
[root@www ~]#
```

ping:

The ping command sends an echo request to a host available on the network. Using this command you can check if your remote host is responding well or not.

The ping command is useful for the following:

- Tracking and isolating hardware and software problems.
- Determining the status of the network and various foreign hosts.
- Testing, measuring, and managing networks.

Syntax: `$ ping <host_address / host_name >`

Example: `$ ping 192.168.0.1` (or) `$ ping www.google.com`

```
root@localhost:~# ping 192.168.0.1
PING 192.168.0.1 (192.168.0.1) 56(84) bytes of data.
64 bytes from 192.168.0.1: icmp_seq=1 ttl=128 time=228 ms
64 bytes from 192.168.0.1: icmp_seq=2 ttl=128 time=3.76 ms
64 bytes from 192.168.0.1: icmp_seq=3 ttl=128 time=4.15 ms
^C
--- 192.168.0.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2817ms
rtt min/avg/max/mdev = 3.764/78.790/228.448/105.824 ms
root@localhost ~]#
```

Note: Above command would start printing a response after every second. To come out of the command you can terminate it by pressing **CTRL + C** keys.

ifconfig:

Ifconfig is used to configure the kernel-resident network interfaces. It is used at boot time to set up interfaces as necessary. After that, it is usually only needed when debugging or when system tuning is needed.

Syntax: `$ ifconfig <interface> [options]`

Options:

Option	Description
-a	Display information for all network interfaces, even if they are down.
-s	Display a short list in a format identical to the command netstat -i .
up	This flag causes the interface to be activated.
Down	This flag can be used to turn off the interface.

Example:

```
root@localhost:~# ifconfig
eth1      Link encap:Ethernet  HWaddr 00:0C:29:35:FD:25
          inet addr:192.168.183.132  Bcast:192.168.183.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe35:fd25/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:2077 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1197 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:329786 (322.0 KiB)  TX bytes:143373 (140.0 KiB)
          Interrupt:19 Base address:0x2000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:399 errors:0 dropped:0 overruns:0 frame:0
          TX packets:399 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:41208 (40.2 KiB)  TX bytes:41208 (40.2 KiB)

root@localhost ~]#
```

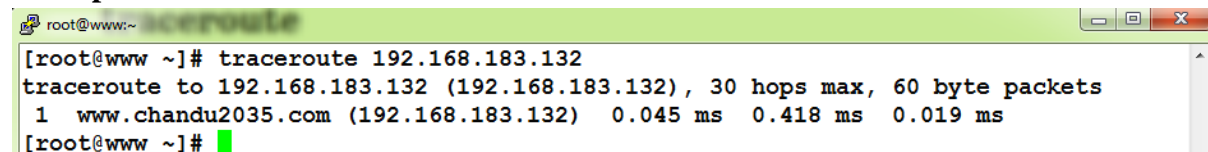
traceroute:

The traceroute command will attempt to provide a list of all the routers your connections cross when reaching out to a remote system. The output also provides some information on how long each segment of the path takes, thus giving you some notion of the quality of a connection.

→ Print the route packets trace to network host.

Syntax: `$ traceroute [option] <IP/Host>`

Example:



```
root@www:~# traceroute 192.168.183.132
traceroute to 192.168.183.132 (192.168.183.132), 30 hops max, 60 byte packets
 1  www.chandu2035.com (192.168.183.132)  0.045 ms  0.418 ms  0.019 ms
root@www ~]#
```

netstat:

The netstat command can tell you about ongoing connections on the local system and ports (i.e., services) that are listening, indicating that services are waiting for requests to come through. By itself, netstat gives you a *lot* information with help its options.

It prints network connections, routing tables, interface statistics, masquerade connections, and multicast memberships

Syntax: `$ netstat [option]`

netstat -anp | grep port → will display process id of application which is using that port

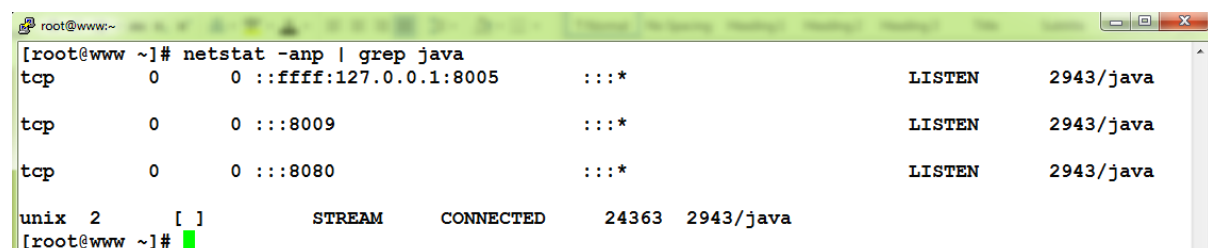
netstat -a or netstat -all → will display all connections including TCP and UDP

netstat -tcp or netstat -t → will display only TCP connection

netstat --udp or netstat -u → will display only UDP connection

netstat -g → will display all multicast network subscribed by this host.

For example, in my Linux system apache tomcat server is running, so I want to know this service is running on which port.



```
root@www:~# netstat -anp | grep java
tcp        0      0 :::ffff:127.0.0.1:8005 :::*           LISTEN        2943/java
tcp        0      0 :::8009          :::*           LISTEN        2943/java
tcp        0      0 :::8080          :::*           LISTEN        2943/java
unix  2      [ ]          STREAM  CONNECTED  24363  2943/java
root@www ~]#
```

nslookup:

Nslookup is a program to query Internet domain name servers. Nslookup has two modes: interactive and non-interactive. Interactive mode allows the user to query name servers for

information about various hosts and domains or to print a list of hosts in a domain. Non-interactive mode is used to print just the name and requested information for a host or domain.

Arguments:

Interactive mode is entered in the following cases:

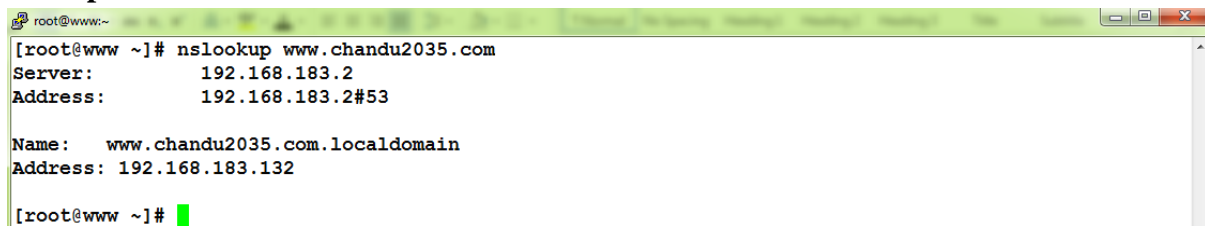
1. When no arguments are given (the default name server will be used)
2. When the first argument is a hyphen (-) and the second argument is the host name or Internet address of a name server.

Non-interactive mode is used when the name or Internet address of the host to be looked up is given as the first argument. The optional second argument specifies the host name or address of a name server.

If you know the IP address it will display hostname. To find all the IP addresses for a given domain name, the command nslookup is used. You must have a connection to the internet for this utility to be useful.

Syntax: \$ nslookup [option] <hostname/domainname>

Example:

A screenshot of a terminal window with a green title bar. The terminal shows the command 'nslookup www.chandu2035.com' being executed. The output displays the server IP (192.168.183.2), the address (192.168.183.2#53), the name (www.chandu2035.com.localdomain), and the address (192.168.183.132). The prompt '[root@www ~]#' is visible at the bottom.

```
[root@www ~]# nslookup www.chandu2035.com
Server:      192.168.183.2
Address:     192.168.183.2#53

Name:   www.chandu2035.com.localdomain
Address: 192.168.183.132

[root@www ~]#
```

finger:

View user information, displays a user's login name, real name, terminal name and write status. This is pretty old UNIX command and rarely used now days.

12. Unix Regular Expressions

A regular expression is a string that can be used to describe several sequences of characters. Regular expressions are used by several different Linux commands, like **sed**, **awk**, **grep** and, to a more limited extent, **vi**.