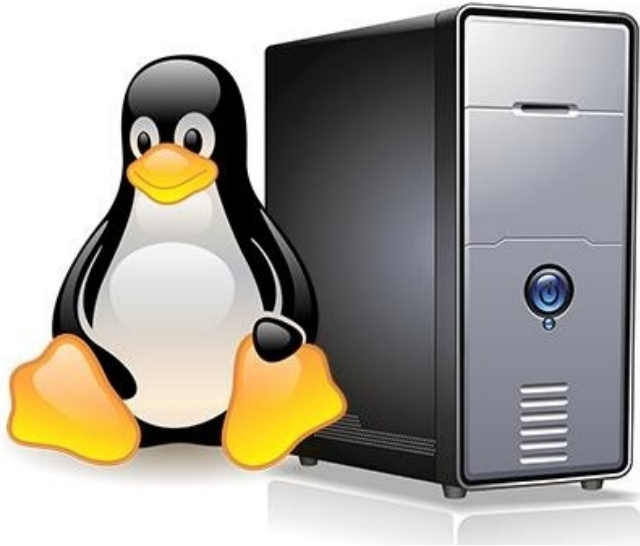# Introduction to LINUX

Penguin is a
Mascot of Linux.

- The goal of this document is to help you become familiar with the Linux operating system.

- It is designed to take you well beyond being a casual, personal user of Linux.

- You'll start with the fundamentals and progress to explore the various tools and techniques commonly used by Linux users, programmers, and system administrators to do their day-to-day work.

Focus on Three Major Linux Distribution Families
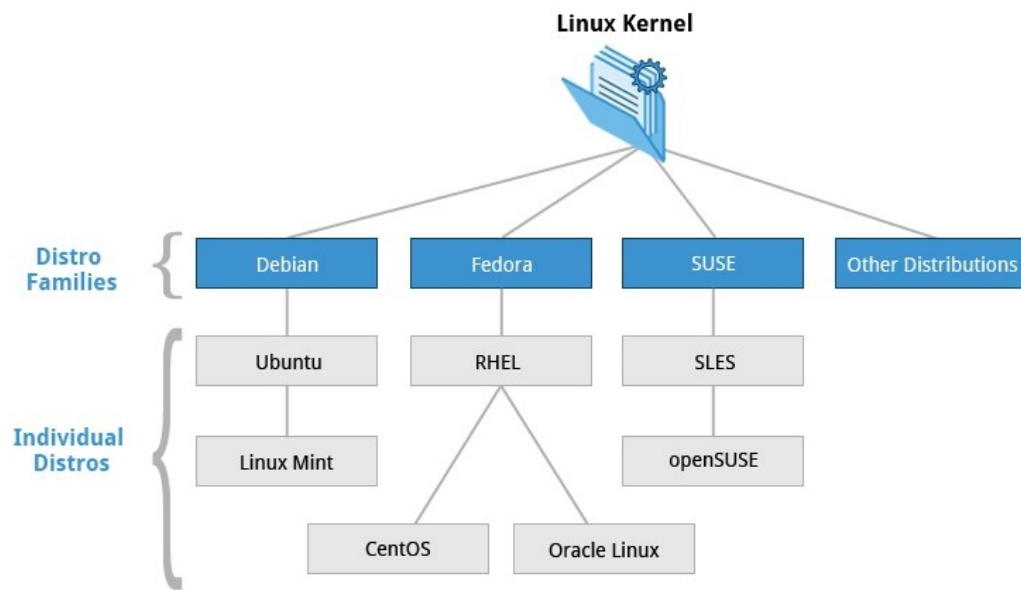
Unix is Mother of All Operating Systems, and Open-Source Means Freedom, It is a Community OS.

Focus on Three Major Linux Flavors.

There are three major distribution families within Linux: Fedora, SUSE and Debian. In this course we will work with representative members of all of these families throughout.

Linux borrows heavily from the UNIX operating system, with which its creators were well versed.

Linux accesses many features and services through files and file-like objects.

Linux is a fully multitasking, multiuser operating system, with built-in networking and service processes known as daemons.

Linux is developed by a loose confederation of developers from all over the world, collaborating over the Internet, with Linus Torvalds at the head. Technical skill and a desire to contribute are the only qualifications for participating.

The Linux community is a far reaching ecosystem of developers, vendors, and users that supports and advances the Linux operating system.

Some of the common terms used in Linux are: Kernel, Distribution, Boot loader, Service, File system, X Window system, desktop environment, and command line.

A full Linux distribution consists of the kernel plus a number of other software tools for file-related operations, user management, and software package management.

Linux File systems

Think of a refrigerator that has multiple shelves that can be used for storing various items. These shelves help you organize the grocery items by shape, size, type, etc. The same concept applies to a file system, which is the embodiment of a method of storing and organizing arbitrary collections of data in a human-usable form.

Different Types of File systems Supported by Linux:

Conventional disk file systems: ext2, ext3, ext4, XFS, Btrfs, JFS, NTFS, etc.

Flash storage file systems: ubifs, JFFS2, YAFFS, etc.

Database file systems

Special purpose file systems: procfs, sysfs, tmpfs, debugfs, etc.

This section will describe the standard file system layout shared by most Linux distributions.

Partitions and File systems

A partition is a logical part of the disk, whereas a file system is a method of storing/finding files on a hard disk (usually in a partition). By way of analogy, you can think of file systems as being like family trees that show descendants and their relationships, while the partitions are like different families (each of which has its own tree).

A comparison between file systems in Windows and Linux is given in the following table:

	Windows	Linux
Partition	Disk1	/dev/sda1
Filesystem type	NTFS/FAT32	EXT3/EXT4/XFS...
Mounting Parameters	DriveLetter	MountPoint
Base Folder where OS is stored	C drive	/

**The File system Hierarchy Standard**

Linux systems store their important files according to a standard layout called the File system Hierarchy Standard, or FHS. You can download a document that provides much greater detail here, This standard ensures that users can move between distributions without having to re-learn how the system is organized.

Linux uses the '/' character to separate paths (unlike Windows, which uses '\'), and does not have drive letters. New drives are mounted as directories in the single file system, often under /media (so, for example, a CD-ROM disc labeled FEDORA might end up being found at /media/FEDORA, and a file README.txt on that disc would be at /media/FEDORA/README.txt).
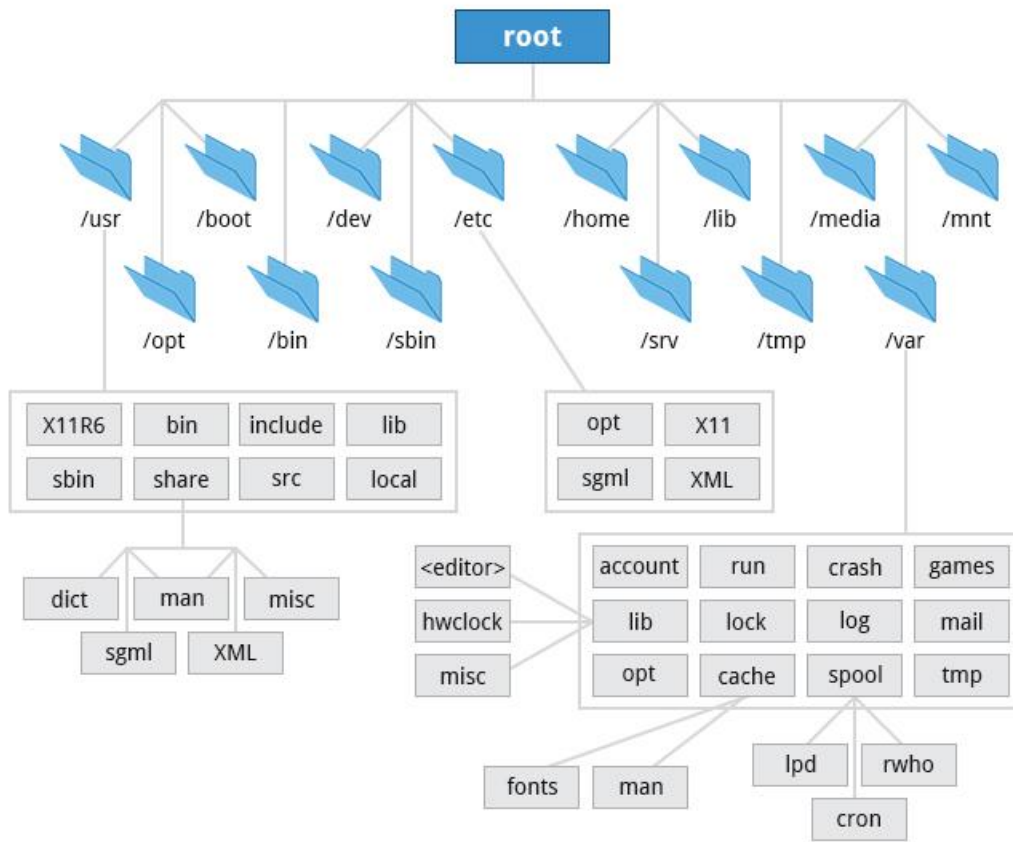
File system Hierarchy Standard

Linux Directory Tree
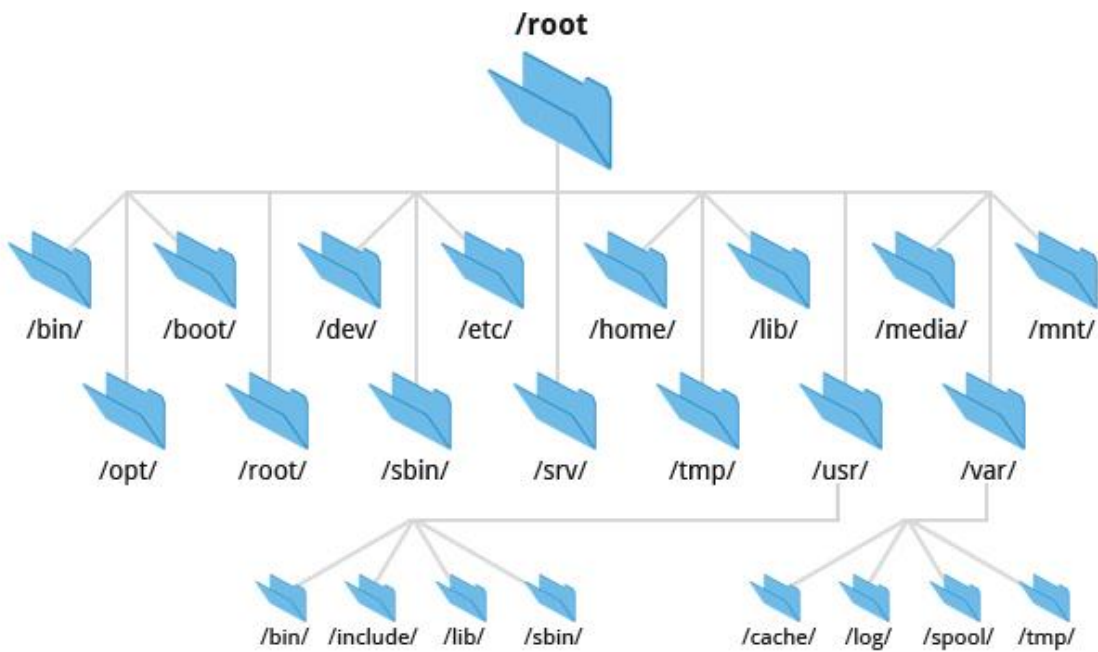
All Linux file system names are case-sensitive, so /boot, /Boot, and /BOOT represent three different directories (or folders). Many distributions distinguish between core utilities needed for proper system operation and other programs, and place the latter in directories under /usr (think "**user**"). To get a sense for how the other programs are organized, find the /usr directory in the diagram above and compare the subdirectories with those that exist directly under the system root directory (/).
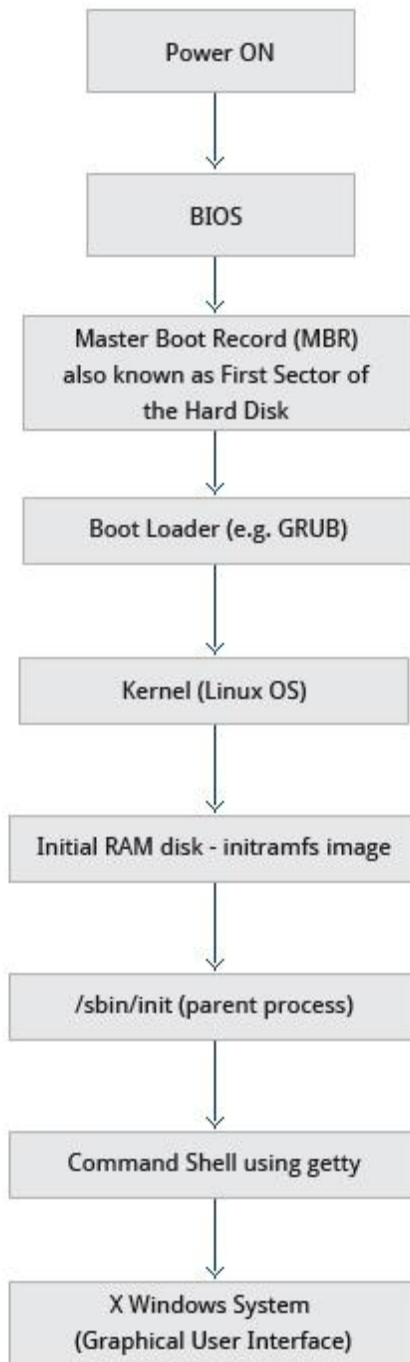
Absolute Path Begins with /data/user1/images/photo.jpg

Relational Path Begins with the Current Working Directory.

The Boot Process

Have you ever wondered what happens in the background from the time you press the **Power** button until the Linux login prompt appears?

The Linux **boot process** is the procedure for initializing the system. It consists of everything that happens from when the computer power is first switched on until the user interface is fully operational.

Once you start using Linux, you will find that having a good understanding of the steps in the boot process will help you with troubleshooting problems as well as with tailoring the computer's performance to your needs.

```
┌─────────────────┐
│    Power ON     │
└─────────────────┘
         │
         ▼
┌─────────────────┐
│      BIOS       │
└─────────────────┘
         │
         ▼
┌─────────────────────────┐
│ Master Boot Record (MBR)│
│ also known as First     │
│ Sector of the Hard Disk │
└─────────────────────────┘
         │
         ▼
┌─────────────────────────┐
│ Boot Loader (e.g. GRUB) │
└─────────────────────────┘
         │
         ▼
┌─────────────────────────┐
│   Kernel (Linux OS)     │
└─────────────────────────┘
         │
         ▼
┌──────────────────────────────────┐
│ Initial RAM disk - initramfs image│
└──────────────────────────────────┘
         │
         ▼
┌─────────────────────────┐
│ /sbin/init (parent      │
│ process)                │
└─────────────────────────┘
         │
         ▼
┌─────────────────────────┐
│ Command Shell using getty│
└─────────────────────────┘
         │
         ▼
┌─────────────────────────┐
│   X Windows System      │
│ (Graphical User Interface)│
└─────────────────────────┘
```
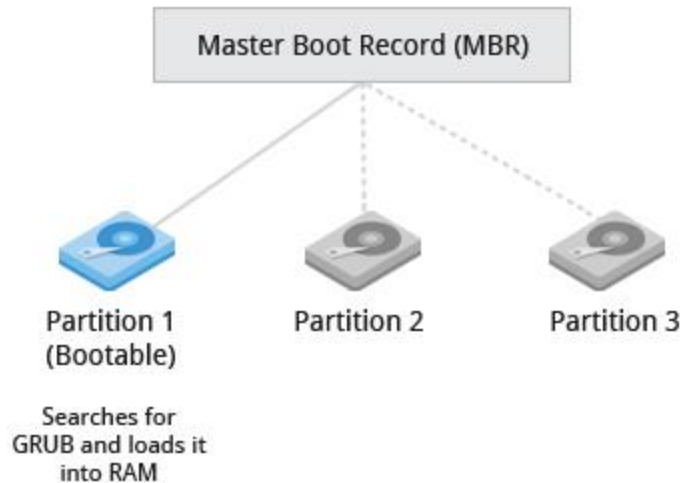
BIOS - The First Step

Starting an **x86-**based Linux system involves a number of steps. When the computer is powered on, the **Basic Input/Output System** (**BIOS)** initializes the hardware, including the screen and keyboard, and tests the main memory. This process is also called **POST** (**Power On Self Test**).

The BIOS software is stored on a ROM chip on the motherboard. After this, the remainder of the boot process is completely controlled by the operating system.

```
         ⏻
     Power On
         │
         ▼
       BIOS
(Basic Input/Output system)
         │
         ▼
        �
   [server icon]

Initializes the screen and keyboard
    and tests the main memory
```
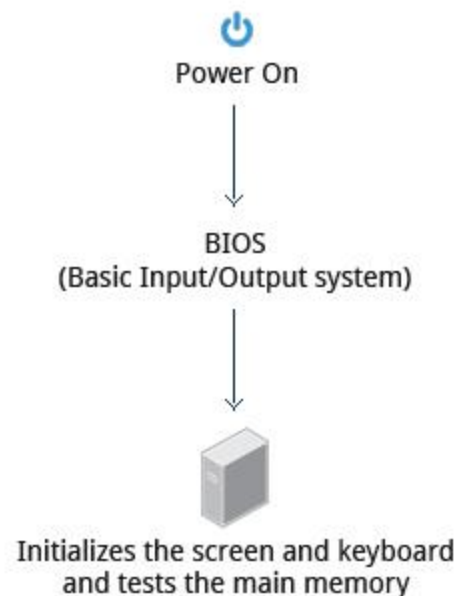
Master Boot Records (MBR) and Boot Loader

Once the **POST** is completed, the system control passes from the **BIOS** to the boot loader. The boot loader is usually stored on one of the hard disks in the system, either in the boot sector (for traditional **BIOS/MBR** systems) or the **EFI** partition (for more recent **(Unified) Extensible Firmware Interface** or **EFI/UEFI** systems). Up to this stage, the machine does not access any mass storage media. Thereafter, information on the date, time, and the most important peripherals are loaded from the **CMOS values** (after a technology used for the battery-powered memory store - which allows the system to keep track of the date and time even when it is powered off).

Master Boot Record (MBR)

Partition 1
(Bootable)

Partition 2

Partition 3

Searches for
GRUB and loads it
into RAM

A number of boot loaders exist for Linux; the most common ones are **GRUB** (for **GRand Unified Boot loader**) and **ISOLINUX** (for booting from removable media). Most Linux boot loaders can present a user interface for choosing alternative options for booting Linux, and even other operating systems that might be installed. When booting Linux, the boot loader is responsible for loading the kernel image and the initial RAM disk (which contains some critical files and device drivers needed to start the system) into memory.

Boot Loader in Action

The boot loader has two distinct stages:

First Stage:

For systems using the BIOS/MBR method, the boot loader resides at the first sector of the hard disk also known as the **Master Boot Record** (**MBR**). The size of the MBR is just 512 bytes. In this stage, the boot loader examines the partition table and finds a bootable partition. Once it finds a bootable partition, it then searches for the second stage boot loader e.g, **GRUB**, and loads it into **RAM** (**Random Access Memory**).

For systems using the **EFI/UEFI** method, **UEFI firmware** reads its **Boot Manager** data to determine which **UEFI** application is to be launched and from where (i.e., from which disk and partition the **EFI** partition can be found). The firmware then launches the **UEFI** application, for example, **GRUB**, as defined in the boot entry in the firmware's boot manager. This procedure is more complicated but more versatile than the older MBR methods.

System

Bootable
Hard Disk

Boot loader

Kernel

Initial
RAM Disk
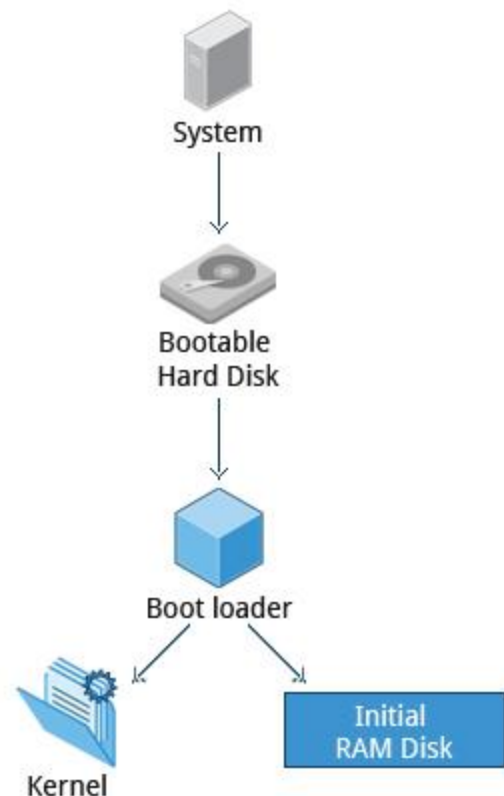
Second Stage:

The second stage boot loader resides under /boot. A **splash screen** is displayed which allows us to choose which Operating System (OS) to boot. After choosing the OS, the boot loader loads the kernel of the selected operating system into RAM and passes control to it.

The boot loader loads the selected kernel image (in the case of Linux) and passes control to it. Kernels are almost always compressed, so its first job is to uncompress itself. After this, it will check and analyze the system hardware and initialize any hardware device drivers built into the kernel.

The Linux Kernel

The boot loader loads both the kernel and an initial RAM–based file system (**initramfs**) into memory so it can be used directly by the kernel.

When the kernel is loaded in RAM, it immediately initializes and configures the computer's memory and also configures all the hardware attached to the system. This includes all processors, I/O subsystems, storage devices, etc. The kernel also loads some necessary user space applications.
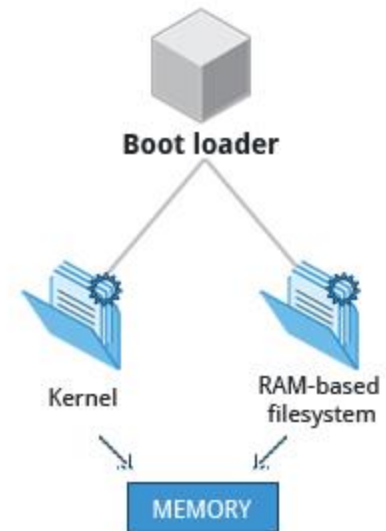
The Initial RAM Disk
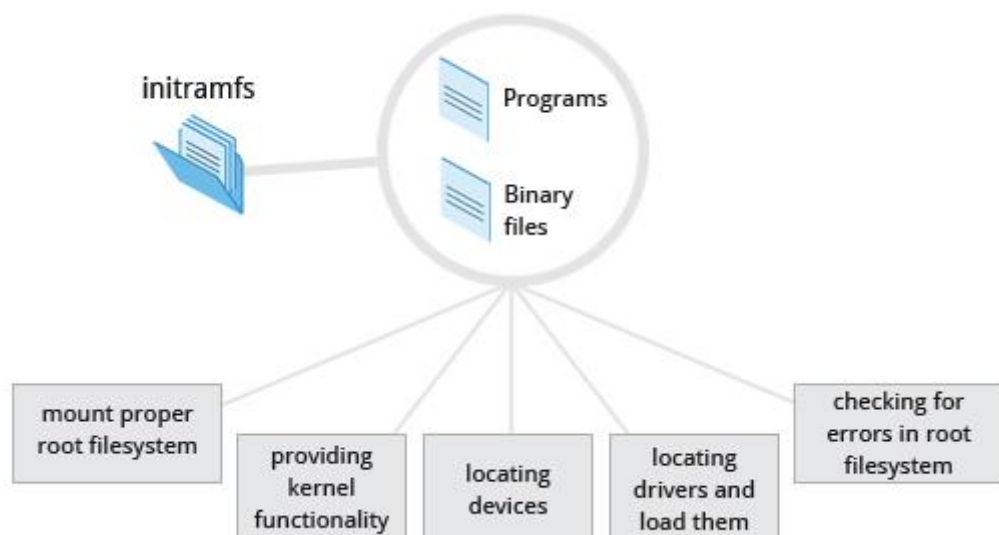
The **initramfs** file system image contains programs and binary files that perform all actions needed to mount the proper root file system, like providing kernel functionality for the needed file system and device drivers for mass storage controllers with a facility called **udev** (for **U**ser **Dev**ice) which is

responsible for figuring out which devices are present, locating the **drivers** they need to operate properly, and loading them. After the root file system has been found, it is checked for errors and mounted.

The **mount** program instructs the operating system that a file system is ready for use, and associates it with a particular point in the overall hierarchy of the file system (the **mount point**). If this is successful, the **initramfs** is cleared from RAM and the **init** program on the root file system (/sbin/init) is executed.

init handles the mounting and pivoting over to the final real root file system. If special hardware drivers are needed before the mass storage can be accessed, they must be in the **initramfs** image.
/sbin/init and Services

Once the kernel has set up all its hardware and mounted the root file system, the kernel runs the /sbin/init program. This then becomes the initial process, which then starts other processes to get the system running. Most other processes on the system trace their origin ultimately to **init**; the exceptions are kernel processes, started by the kernel directly for managing internal operating system details.

Traditionally, this process startup was done using conventions that date back to **System V UNIX**, with the system passing through a sequence of **runlevels** containing collections of scripts that start and stop services. Each runlevel supports a different mode of running the system. Within each runlevel, individual services can be set to run, or to be shut down if running. Newer distributions are moving away from the System V standard, but usually support the System V conventions for compatibility purposes.
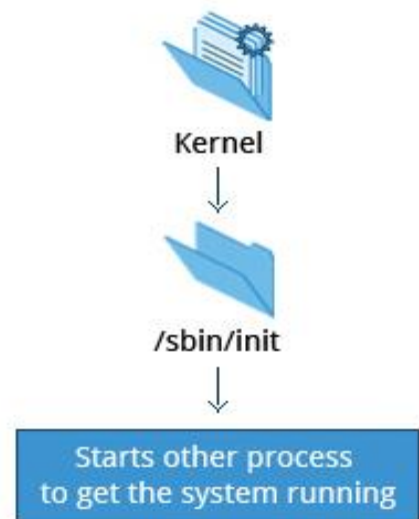
Besides starting the system, **init** is responsible for keeping the system running and for shutting it down cleanly. It acts as the "manager of last resort" for all non-kernel processes, cleaning up after them when necessary, and restarts user login services as needed when users log in and out.

Text-Mode Login

Near the end of the boot process, **init** starts a number of text-mode login prompts (done by a program called **getty**). These enable you to type your username, followed by your password, and to eventually get a command shell.

Usually, the default command shell is **bash** (the GNU **Bourne Again Shell**), but there are a number of other advanced command shells available. The shell prints a text prompt, indicating it is ready to accept commands; after the user types the command and presses **Enter**, the command is executed, and another prompt is displayed after the command is done.

As you'll learn in the chapter 'Command Line Operations', the terminals which run the command shells can be accessed using the **ALT** key plus a **function** key. Most distributions start six text terminals and one graphics terminal starting with **F1** or **F2**. If the graphical environment is also started, switching to a text
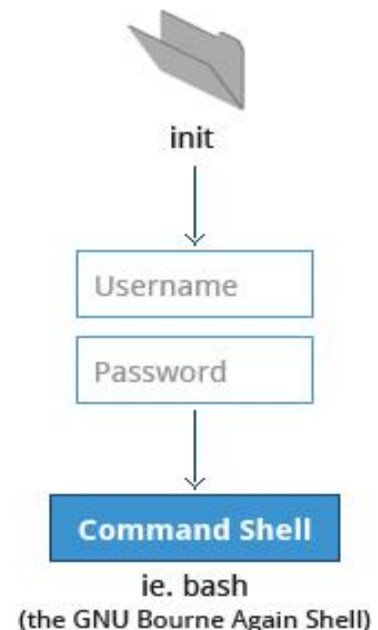
console requires pressing **CTRL-ALT +** the appropriate function key (with **F7** or **F1** being the GUI). As you'll see shortly, you may need to run the **startx** command in order to start or restart your graphical desktop after you have been in pure text mode.

X Window System

Generally, in a Linux desktop system, the **X Window System** is loaded as the final step in the boot process.

A service called the **display manager** keeps track of the displays being provided, and loads the **X server** (so-called because it provides graphical services to applications, sometimes called **X clients**). The display manager also handles graphical logins, and starts the appropriate desktop environment after a user logs in.
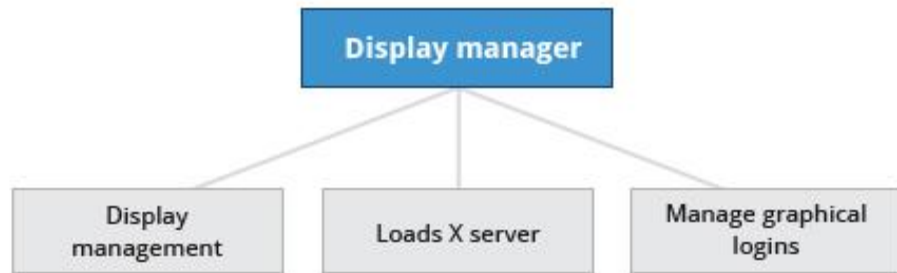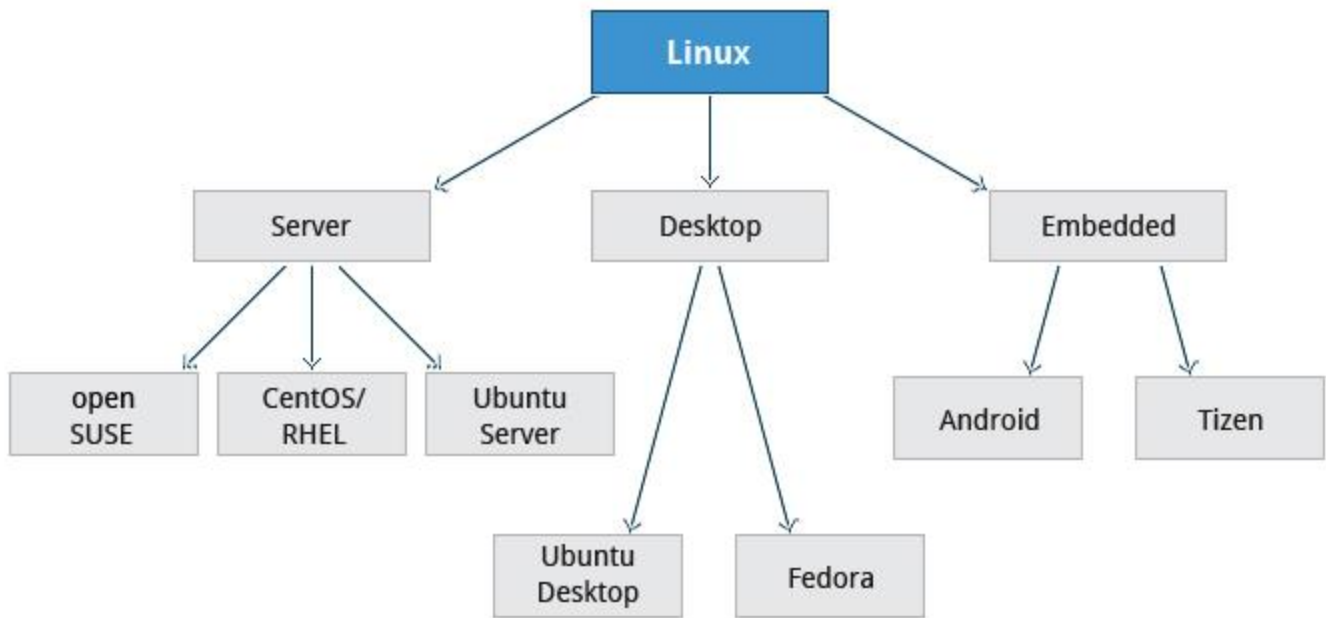
X Server

A desktop environment consists of a session manager, which starts and maintains the components of the graphical session, and the window manager, which controls the placement and movement of windows, window title-bars, and controls.

Although these can be mixed, generally a set of utilities, session manager, and window manager are used together as a unit, and together provide a seamless desktop environment.

If the display manager is not started by default in the default runlevel, you can start **X** a different way, after logging on to a text-mode console, by running **startx** from the command line.

### Questions to Ask When Choosing a Linux Distribution

Some questions worth thinking about before deciding on a distribution include:

- What is the main function of the system? (server or desktop)

- What types of packages are important to the organization? For example, web server, word processing, etc.

- How much hard disk space is available? For example, when installing Linux on an embedded device, there will be space limitations.

- How often are packages updated?

- How long is the support cycle for each release? For example, LTS releases have long term support.

- Do you need kernel customization from the vendor?

- What hardware are you running the Linux distribution on?
For example, **X86, ARM, PPC**, etc.

- Do you need long-term stability or short-term experimental software?

Linux Installation: Planning

- A **partition** layout needs to be decided at the time of installation because Linux systems handle partitions by mounting them at specific points in the file system. You can always modify the design later, but it always easier to try and get it right to begin with.
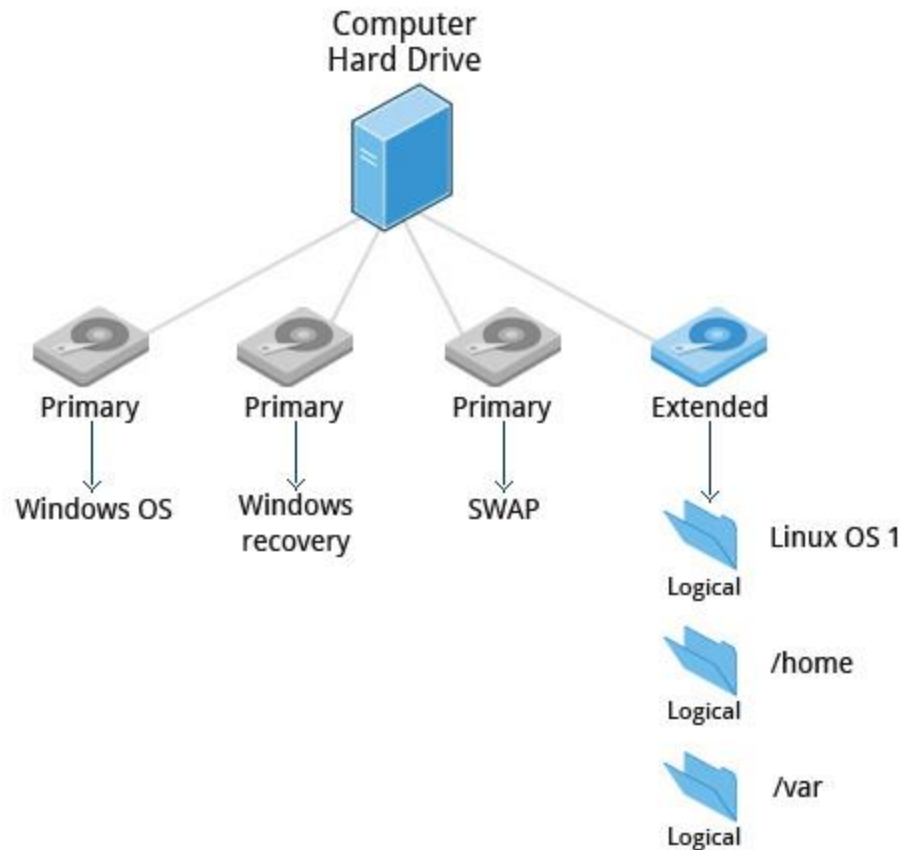

Partitions in the Linux Hard Disk
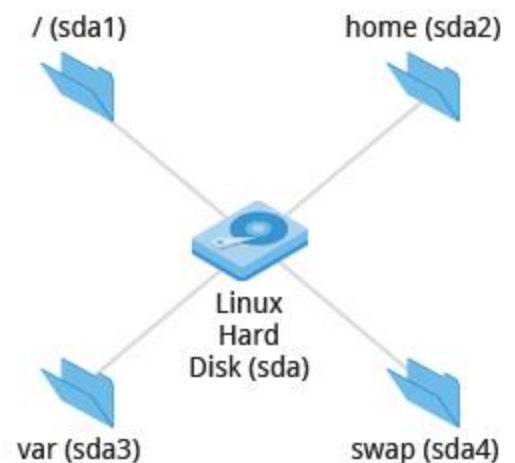
- Nearly all installers provide a reasonable file system layout by default, with either all space dedicated to normal files on one big partition and a smaller **swap** partitition, or with separate partitions for some space-sensitive areas like /home and /var. You may need to override the defaults and do something different if you have special needs, or if you want to use more than one disk.

Planning in Linux Installation

All installations include the bare minimum software for running a Linux distribution.

Most installers also provide options for adding categories of software. Common applications (such as the **Firefox** web browser and **LibreOffice** office suite), developer tools (like the **vi** and **emacs** text editors which we will explore later in this course), and other popular services, (such as the **Apache** web server tools or **MySQL** database) are usually included. In addition, a desktop environment is installed by default.

All installers secure the system being installed as part of the installation. Usually, this consists of setting the password for the superuser (**root**) and setting up an initial user. In some cases (such as **Ubuntu**), only an initial user is set up; direct root login is disabled and root access requires logging in first as a normal user and then using **sudo** as we will describe later. Some distributions will also install more advanced security frameworks, such as **SELinux** or **AppArmor.**

Computer Hard Drive

Primary → Windows OS
Primary → Windows recovery
Primary → SWAP
Extended →
Logical — Linux OS 1
Logical — /home
Logical — /var

Linux Installation: Install Source

Like other operating systems, Linux distributions are provided on optical media such as CDs or DVDs. USB media is also a popular option. Most Linux distributions support booting a small image and downloading the rest of the system over the network; these small images are usable on media or as network boot images, making it possible to install without any local media at all.

Many installers can do an installation completely automatically, using a configuration file to specify installation options. This file is called a **Kickstart** file for **Fedora**-based systems, an **AutoYAST** profile for **SUSE**-based systems,

and a **preseed file** for the **Debian**-based systems.

Each distribution provides its own documentation and tools for creating and managing these files.

Linux Installation: The Process

The actual installation process is pretty similar for all distributions.

After booting from the installation media, the installer starts and asks questions about how the system should be set up. (These questions are skipped if an automatic installation file is provided.) Then, the installation is performed.

Finally, the computer reboots into the newly-installed system. On some distributions, additional questions are asked after the system reboots.

Most installers have the option of downloading and installing updates as part of the installation process; this requires Internet access. Otherwise, the system uses its normal update mechanism to retrieve those updates after the installation is done.

You have completed this chapter. Let's summarize the key concepts covered:

- A **partition** is a logical part of the disk.

- A **file system** is a method of storing/finding files on a hard disk.

- By dividing the hard disk into partitions, data can be grouped and separated as needed. When a failure or mistake occurs, only the data in the affected partition will be damaged, while the data on the other partitions will likely survive.

- The boot process has multiple steps, starting with **BIOS**, which triggers the **boot loader** to start up the Linux kernel. From there the **initramfs** file system is invoked, which triggers the**init** program to complete the startup process.

- Determining the appropriate distribution to deploy requires that you match your specific system needs to the capabilities of the different distributions.

Introduction to the Command Line

Linux system administrators spend a significant amount of their time at a **command line** prompt. They often automate and troubleshoot tasks in this text environment. There is a saying, "*graphical user interfaces make easy tasks easier, while command line interfaces make difficult tasks possible*." Linux

relies heavily on the abundance of command line tools. The command line interface provides the following advantages:

- No GUI overhead.

- Virtually every task can be accomplished using the command line.

- You can script tasks and series of procedures.

- You can log on remotely to networked machines anywhere on the Internet.

- You can initiate graphical apps directly from the command line.

A **terminal emulator** program emulates (simulates) a stand alone terminal within a window on the desktop. By this we mean it behaves essentially as if you were logging into the machine at a pure text terminal with no running graphical interface. Most terminal emulator programs support multiple terminal sessions by opening additional tabs or windows.

By default, on **GNOME** desktop environments, the **gnome-terminal** application is used to emulate a text-mode terminal in a window. Other available terminal programs include:

- **xterm**

- **rxvt**

- **konsole**

- **terminator**

Launching Terminal Windows

To open a terminal in **CentOS**:

1. On the **CentOS** desktop, in the upper-left corner, click**Applications**.

2. From the **System Tools** menu, select **Terminal**.

To open a terminal in **openSUSE**:

1. On the **openSUSE** desktop, in the upper-left corner of the screen, click **Activities**.

2. From the left pane, click **Show Applications**.

3. Scroll-down and select the required terminal.

To open a terminal in **Ubuntu**:

1. In the left panel, click the **Ubuntu** icon.

2. Type **terminal** in the **Search** box.

If the **nautilus-open-terminal** package is installed on any of these distributions, you can always open a terminal by right clicking anywhere on the desktop background and selecting **Open in Terminal.**
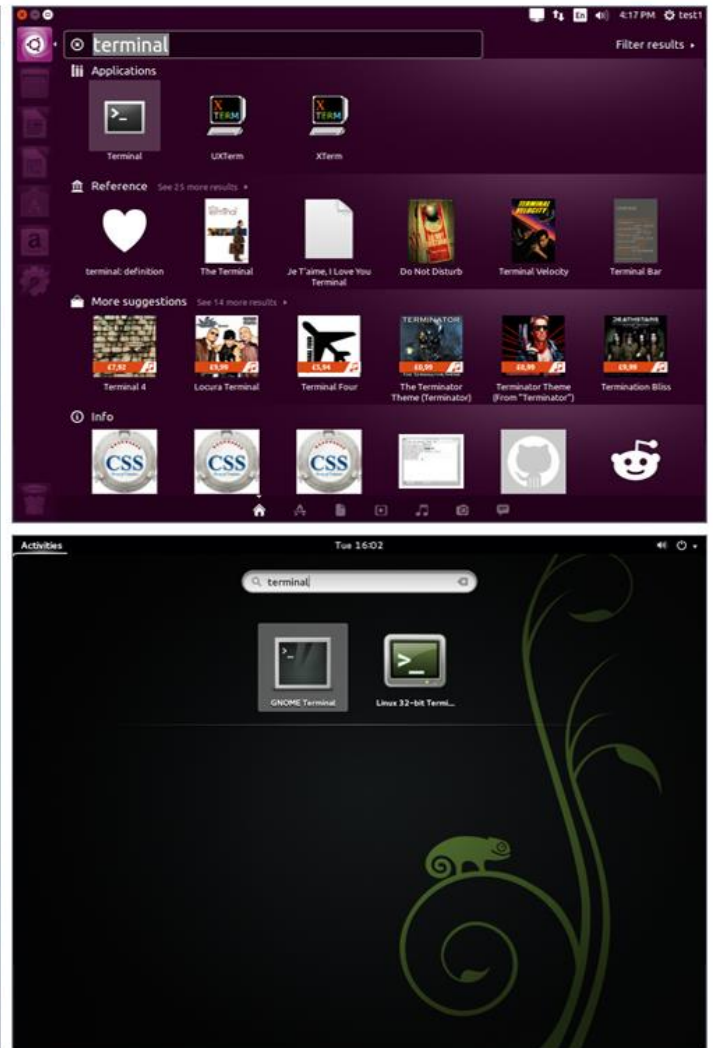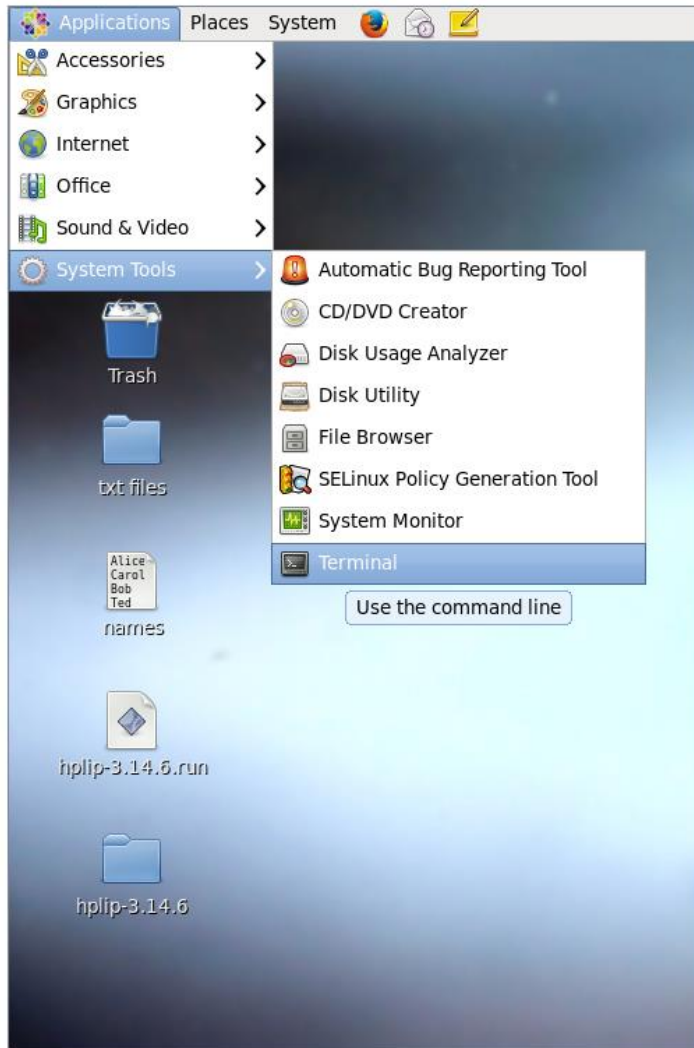
The X Window System

The customizable nature of Linux allows you to drop (temporarily or permanently) the **X Window** graphical interface, or to start it up after the system has been running. Certain Linux distributions distinguish versions of the install media between desktop (with **X**) and server (usually without **X**); Linux production servers are usually installed without **X** and even if it is installed, usually do not launch it during system start up. Removing **X** from a production server can be very helpful in maintaining a lean system which can be easier to support and keep secure.
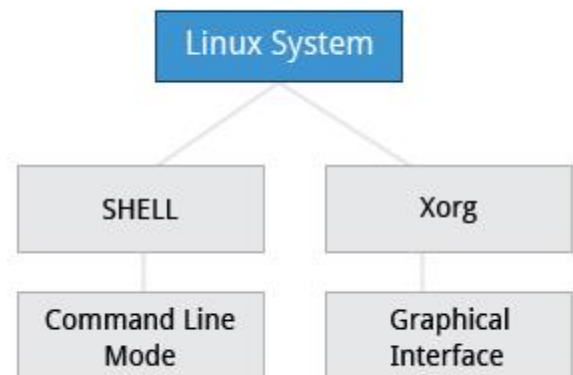
Virtual Terminals

Virtual Terminals (VT) are **console** sessions that use the entire display and keyboard outside of a graphical environment. Such terminals are considered "virtual" because although there can be multiple active terminals, only one terminal remains visible at a time. A VT is not quite the same as a command line terminal window; you can have many of those visible at once on a graphical desktop.
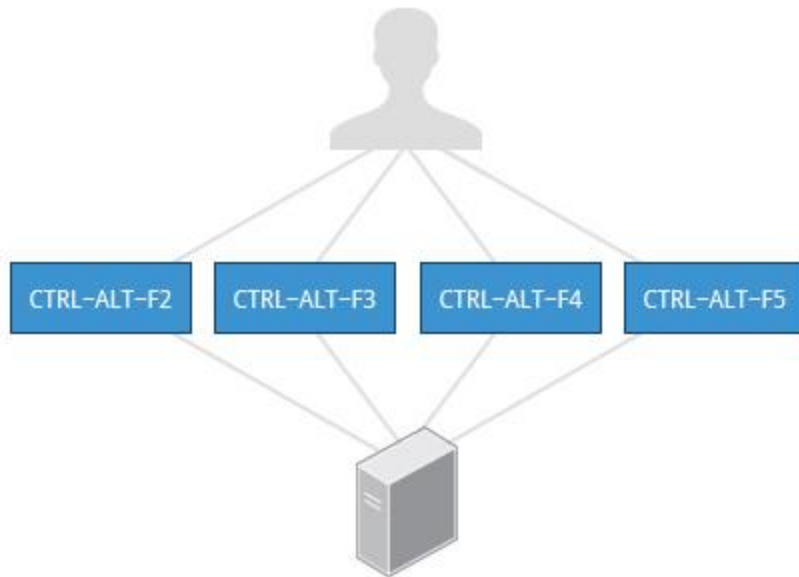
One virtual terminal (usually number one or seven) is reserved for the graphical environment, and text logins are enabled on the unused VTs. **Ubuntu** uses VT 7, but**CentOS/RHEL** and **openSUSE** use VT 1 for the graphical display.

An example of a situation where using the VTs is helpful when you run into problems with the graphical desktop. In this situation, you can switch to one of the text VTs and troubleshoot.

To switch between the VTs, press **CTRL-ALT-corresponding function key** for the VT. For example, press **CTRL-ALT-F6** for VT 6. (Actually you only have to press **ALT-F6** key combination if you are in a VT not running **X** and want to switch to another VT.)

The Command Line

Most input lines entered at the shell prompt have three basic elements:

- Command

- Options

- Arguments

The **command** is the name of the program you are executing. It may be followed by one or more **options** (or switches) that modify what the command may do. Options usually start with one or two dashes, for example,-p or--print, in order to differentiate them from **arguments**, which represent what the command operates on.

However, plenty of commands have no options, no arguments, or neither. You can also type other things at the command line besides issuing commands, such as setting environment variables.

Turning off the Graphical Desktop

Linux distributions can start and stop the graphical desktop in various ways. For **Debian**-based systems, the **Desktop Manager**runs as a service which can be simply stopped. For RPM-based systems, the **Desktop Manager** is run directly by **init** when set

to run level 5; switching to a different runlevel stops the desktop.

Use the sudo service gdm stop or sudo service lightdm stop commands, to stop the graphical user interface in **Debian**-based systems. On **RPM**-based systems typing sudo telinit 3 may have the same effect of killing the GUI.

sudo

All the demonstrations created have a user configured with**sudo** capabilities to provide the user with administrative (admin) privileges when required. **sudo** allows users to run programs using the security privileges of another user, generally root (superuser). The functionality of **sudo** is similar to that of **run as** in **Windows**.

On your own systems, you may need to set up and enable **sudo** to work correctly. To do this, you need to follow some steps that we won't explain in much detail now, but you will learn about later in this course. When running on **Ubuntu**, **sudo** is already always set up for you during installation. If you are running something in the **Fedora** or**openSUSE** families of distributions, you will likely need to set up **sudo** to work properly for you after initial installation.

Next, you will learn the steps to setup and run **sudo** on your system.

**Steps for Setting up and Running sudo**

If your system does not already have **sudo** set up and enabled, you need to do the following steps:

1. You will need to make modifications as the administrative or super user, root. While **sudo** will become the preferred method of doing this, we don't have it set up yet, so we will use **su** (which we will discuss later in detail) instead. At the command line prompt, type **su** and press **Enter.** You will then be prompted for the root password, so enter it and press **Enter**. You will notice that nothing is printed; this is so others cannot see the password on the screen. You should end up with a different looking prompt, often ending with '#'. For example: $ su Password: #

2. Now you need to create a configuration file to enable your user account to use **sudo**. Typically, this file is created in the /etc/sudoers.d/ directory with the name of the file the same as your username. For example, for this demo, let's say your username is "student". After doing step 1, you would then create the configuration file for "student" by doing this: # echo "student ALL=(ALL) ALL" > /etc/sudoers.d/student

3. Finally, some Linux distributions will complain if you don't also change permissions on the file by doing: # chmod 440 /etc/sudoers.d/student

That should be it. For the rest of this course, if you use **sudo** you should be properly set up. When using **sudo,** by default you will be prompted to give a password (your own user password) at least the first time you do it within a specified time interval. It is possible (though very insecure) to configure **sudo** to not require a password or change the time window in which the password does not have to be repeated with every **sudo** command.

Basic Operations

In this section we will discuss how to accomplish basic operations from the command line. These include how to log in and log out from the system, restart or shutdown the system, locate applications, access directories, identify the absolute and relative paths, and explore the file system.
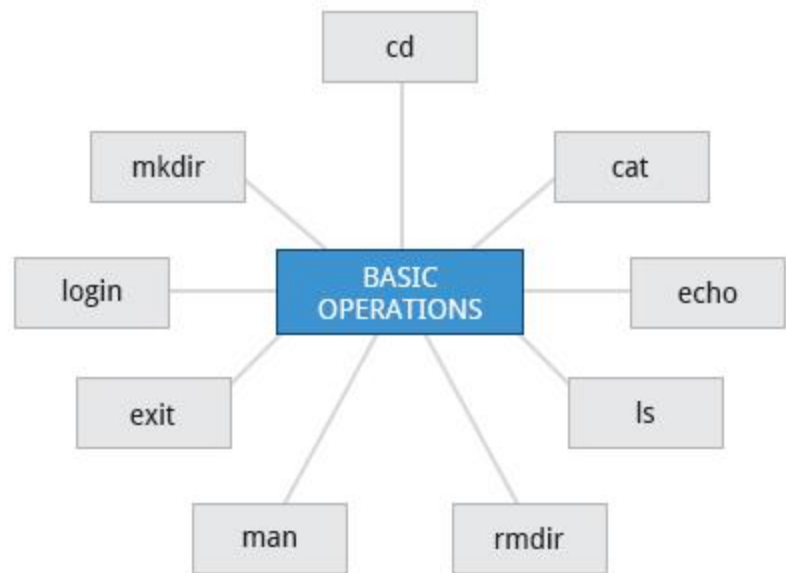
Logging In and Out

An available **text terminal** will prompt for a username (with the string login:) and password. When typing your password, nothing is displayed on the terminal (not even a * to indicate that you typed in something) to prevent others from seeing your password. After you have logged in to the system, you can perform basic operations.

Once your session is started (either by logging in to a text terminal or via a graphical terminal program) you can also connect and log in to remote systems via the **Secure Shell (SSH)** utility. For example, by typing ssh username@remote-server.com, **SSH** would connect securely to the remote machine and give you a command line terminal window, using passwords (as with regular logins) or cryptographic keys (a topic we won't discuss) to prove your identity.

Rebooting and Shutting Down

The preferred method to shut down or reboot the system is to use the **shutdown** command. This sends a warning message and then prevents further users from logging in. The**init** process will then control shutting down or rebooting the system. It is important to always shut down properly; failure to do so can result in damage to the system and/or loss of data.

The **halt** and **poweroff** commands issue shutdown -h to halt the system; **reboot** issues shutdown -r and causes the machine to reboot instead of just shutting down. Both rebooting and shutting down from the command line requires superuser (root) access.

When administering a multiuser system, you have the option of notifying all users prior to shutdown as in:

$ sudo shutdown -h 10:00 "Shutting down for scheduled maintenance."

Locating Applications

Depending on the specifics of your particular distribution's policy, programs and software packages
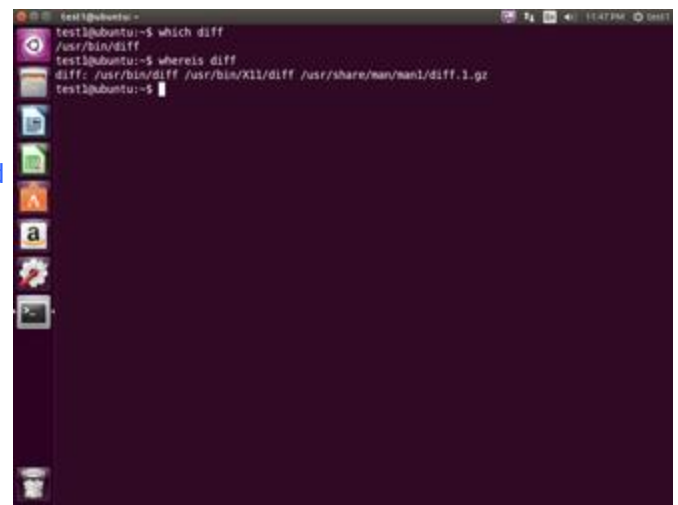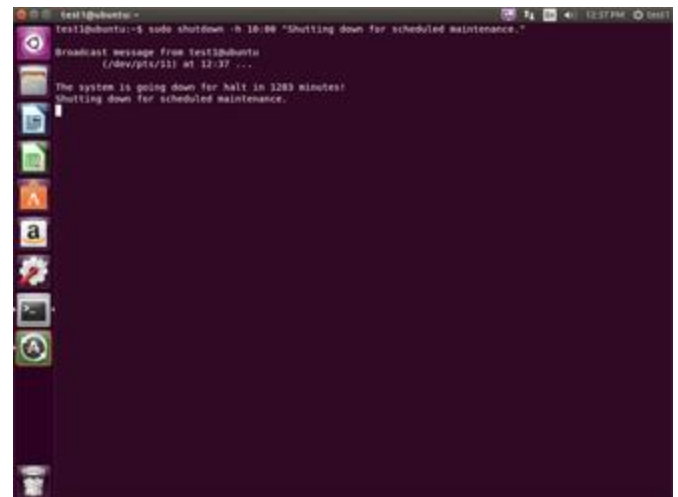
can be installed in various directories. In general, executable programs should live in the /bin, /usr/bin,/sbin,/usr/sbin directories or under /opt.

One way to locate programs is to employ the **which** utility. For example, to find out exactly where the **diff** program resides on the file system:

$ which diff

If **which** does not find the program, **whereis** is a good alternative because it looks for packages in a broader range of system directories:

$ whereis diff

Accessing Directories

When you first log into a system or open a terminal, the default directory should be your **home directory**; you can print the exact path of this by typing echo $HOME. (Note that some Linux distributions actually open new **graphical** terminals in $HOME/Desktop.) The following commands are useful for directory navigation:

Command	Result
pwd	Displays the present working directory
cd ~ orcd	Change to your home directory (short-cut name is ~ (tilde))
cd ..	Change to parent directory (..)
cd -	Change to previous directory (- (minus))

Understanding Absolute and Relative Paths

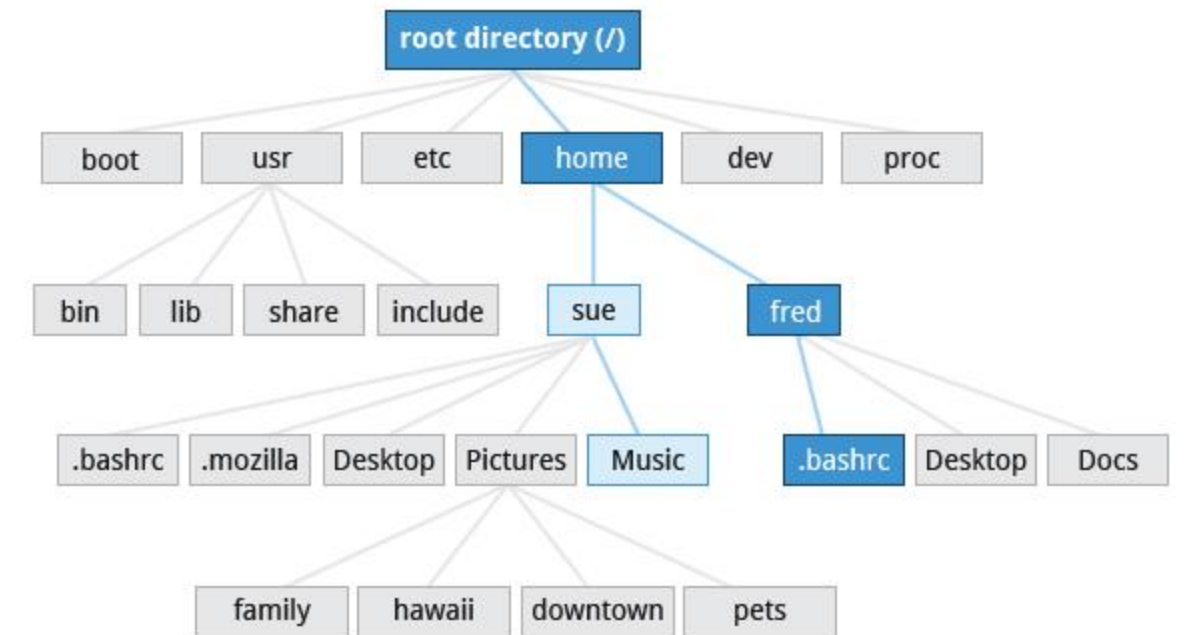There are two ways to identify paths:

1. **Absolute pathname**: An absolute pathname begins with the root directory and follows the tree, branch by branch, until it reaches the desired directory or file. Absolute paths always start with /.

1.In the above example, we use the relative path method to list the files under Music from your current working directory (sue)
$ ls ../sue/Music

2.In the above example, we use the Absolute pathname method to edit the .bashrc file:
$ gedit /home/fred/.bashrc

2. **Relative pathname**: A relative pathname starts from the present working directory. Relative paths never start with /.

Multiple slashes (/) between directories and files are allowed, but all but one slash between elements in the pathname is ignored by the system. ////usr//bin is valid, but seen as /usr/bin by the system.

Most of the time it is most convenient to use relative paths, which require less typing. Usually you take advantage of the shortcuts provided by: . (present directory), .. (parent directory) and ~ (your home directory).

For example, suppose you are currently working in your home directory and wish to move to the /usr/bin directory. The following two ways will bring you to the same directory from your home directory:

1. Absolute pathname method: $ cd /usr/bin

2. Relative pathname method: $ cd ../../usr/bin

In this case, the absolute pathname method is less typing.

Exploring the File system

Traversing up and down the file system tree can get tedious. The **tree** command is a good way to get a bird's-eye view of the file system tree. Use tree -d to view just the directories and to suppress listing file names.

The following commands can help in exploring the file system:

Command	Usage
cd /	**C**hanges your current **d**irectory to the root (/) directory (or path you supply)
ls	**Lis**t the contents of the present working directory
ls –a	**Lis**t **all** files including **hidden** files and directories (those whose name start with .)
tree	Displays a **tree** view of the file system

Hard and Soft (Symbolic) Links

ln can be used to create **hard links** and (with the -s option) **soft links**, also known as **symbolic links** or **symlinks**. These two kinds of links are very useful in UNIX-based operating systems. The advantages of symbolic links are discusssed on the following screen.

Suppose that file1 already exists. A **hard** link, called file2, is created with the command:

$ ln file1 file2

Note that two files now appear to exist. However, a closer inspection of the file listing shows that this is not quite true.

$ ls -li file1 file2

The -i option to **ls** prints out in the first column the **inode** number, which is a unique quantity for each file object. This field is the same for both of these files; what is really going on here is that it is only **one** file but it has more than one name associated with it, as is indicated by the **3** that appears in the **ls** output. Thus, there already was another object linked to file1 before the command was executed.

Symbolic Links

Symbolic (or **Soft**) links are created with the -s option as in:
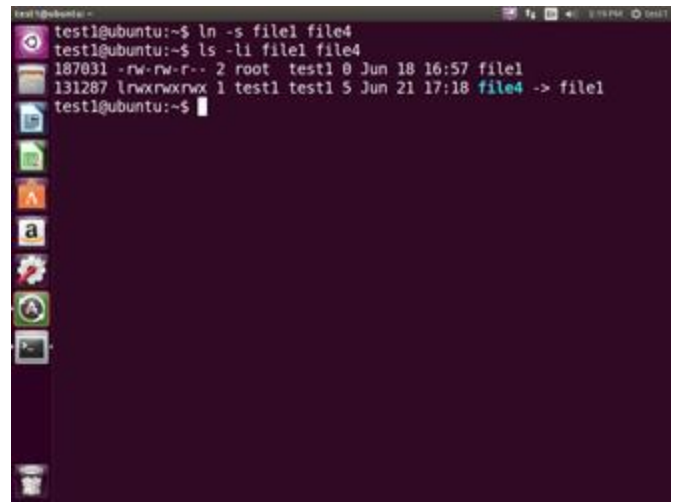
```
$ ln -s file1 file4
$ ls -li file1 file4
```

Notice file4 no longer appears to be a regular file, and it clearly points to file1 and has a different inode number.

Symbolic links take no extra space on the file system (unless their names are very long). They are extremely convenient as they can easily be modified to point to different places. An easy way to create a shortcut from your **home** directory to long pathnames is to create a symbolic link.

Unlike hard links, soft links can point to objects even on different file systems (or partitions) which may or may not be currently available or even exist. In the case where the link does not point to a currently available or existing object, you obtain a **dangling** link.

Hard links are very useful and they save space, but you have to be careful with their use, sometimes in subtle ways. For one thing if you remove either file1 or file2 in the example on the previous screen, the **inode object** (and the remaining file name) will remain, which might be undesirable as it may lead to subtle errors later if you recreate a file of that name.

If you edit one of the files, exactly what happens depends on your editor; most editors including **vi** and **gedit** will retain the link by default but it is possible that modifying one of the names may break the link and result in the creation of two objects.

Navigating the Directory History

The **cd** command remembers where you were last, and lets you get back there with cd -. For remembering more than just the last directory visited, use **pushd** to change the directory instead of **cd**; this pushes your starting directory onto a list. Using **popd** will then send you back to those directories, walking in reverse order (the most recent directory will be the first one retrieved with **popd**). The list of directories is displayed with the **dirs** command.

Standard File Streams

When commands are executed, by default there are three standard **file streams** (or **descriptors**) always open for use: **standard input** (standard in or **stdin**), **standard output** (standard out or **stdout**) and **standard error** (or **stderr**). Usually, **stdin** is your keyboard, **stdout** and **stderr** are printed on your terminal; often **stderr** is redirected to an error logging file. **stdin** is often supplied by directing input to come from a file or from the output of a previous command through a **pipe**. **stdout** is also often redirected into a file. Since **stderr** is where error messages are written, often nothing will go there.

In Linux, all open files are represented internally by what are called **file descriptors**. Simply put, these are represented by numbers starting at zero. **stdin** is file descriptor 0, **stdout** is file descriptor 1, and **stderr** is file descriptor 2. Typically, if other files are opened in addition to these three, which are opened by default, they will start at file descriptor 3 and increase from there.

On the next screen and in chapters ahead, you will see examples which alter where a running command gets its input, where it writes its output, or where it prints diagnostic (error) messages.

I/O Redirection

Through the command **shell** we can **redirect** the three standard filestreams so that we can get input from either a file or another command instead of from our keyboard, and we can write output and errors to files or send them as input for subsequent commands.

For example, if we have a program called **do_something** that reads from **stdin** and writes to **stdout** and **stderr**, we can change its input source by using the less-than sign (<) followed by the name of the file to be consumed for input data:

$ do_something < input-file

If you want to send the output to a file, use the greater-than sign (>) as in:
$ do_something > output-file

Because **stderr** is **not** the same as **stdout**, error messages will still be seen on the terminal windows in the above example.

If you want to redirect **stderr** to a separate file, you use **stderr's** file descriptor number (2), the greater-than sign (>), followed by the name of the file you want to hold everything the running command writes to **stderr**:
$ do_something 2> error-file

A special shorthand notation can be used to put anything written to file descriptor 2 (**stderr**) in the same place as file descriptor 1 (**stdout**): 2>&1
$ do_something > all-output-file 2>&1

bash permits an easier syntax for the above:

$ do_something >& all-output-file

Pipes

The UNIX/Linux philosophy is to have many simple and short programs (or commands) cooperate together to produce quite complex results, rather than have one complex program with many possible options and modes of operation. In order to accomplish this, extensive use of **pipes** is made; you can pipe the output of one command or program into another as its input.

In order to do this we use the vertical-bar, |, (pipe symbol) between commands as in:
 $ command1 | command2 | command3

The above represents what we often call a **pipeline** and allows Linux to combine the actions of several commands into one. This is extraordinarily efficient because **command2** and **command3** do not have to wait for the previous pipeline commands to complete before they can begin hacking at the data in their input streams; on multiple CPU or core systems the available computing power is much better utilized and things get done quicker. In addition there is no need to save output in (temporary) files between the

stages in the pipeline, which saves disk space and reduces reading and writing from disk, which is often the slowest bottleneck in getting something done.

Searching for Files

Being able to quickly find the files you are looking for will make you a much happier Linux user! You can search for files in your parent directory or any other directory on the system as needed.

In this section, you will learn how to use the **locate** and**find** utilities, and how to use **wildcards** in **bash**.

locate

The **locate** utility program performs a search through a previously constructed database of files and directories on your system, matching all entries that contain a specified character string. This can sometimes result in a very long list.

To get a shorter more relevant list we can use the **grep** program as a filter; **grep** will print only the lines that contain one or more specified strings as in:

$ locate zip | grep bin

which will list all files and directories with both "zip" and "bin" in their name . (We will cover **grep** in much more detail later.) Notice the use of **|** to pipe the two commands together.
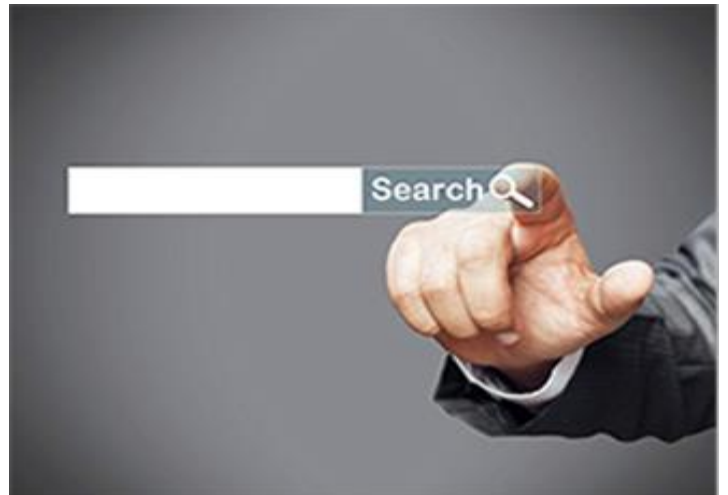
locate utilizes the database created by another program, **updatedb.** Most Linux systems run this automatically once a day. However, you can update it at any time by just running **updatedb** from the command line as the root user.

Wildcards and Matching File Names

You can search for a filename containing specific characters using **wildcards**.

Wildcard	Result

?	Matches any single character
*	Matches any string of characters
[set]	Matches any character in the set of characters, for example [adf] will match any occurrence of "a", "d", or "f"
[!set]	Matches any character not in the set of characters

To search for files using the ? wildcard, replace each unknown **character** with ?, e.g. if you know only the first 2 letters are 'ba' of a 3-letter filename with an extension of .out, type ls ba?.out .

To search for files using the * wildcard, replace the unknown **string** with *, e.g. if you remember only that the extension was .out, type ls *.out

Finding Files In a Directory

find is extremely useful and often-used utility program in the daily life of a Linux system administrator. It recurses down the file system tree from any particular directory (or set of directories) and locates files that match specified conditions. The default pathname is always the present working directory.

For example, administrators sometimes scan for large **core files** (which contain diagnostic information after a program fails) that are more than several weeks old in order to remove them. It is also common to remove files in **/tmp** (and other temporary directories, such as those containing cached files) that have not been accessed recently. Many distros use automated scripts that run periodically to accomplish such house cleaning.

Using **find**

When no arguments are given, **find** lists all files in the current directory and all of its subdirectories. Commonly used options to shorten the list include -name (only list files with a certain pattern in their name), -iname (also ignore the case of file names), and -type (which will restrict the results to files of a certain specified type, such as **d** for directory, **l** for symbolic link or **f** for a regular file, etc).

Searching for files and directories named "gcc":
$ find /usr -name gcc

Searching only for directories named "gcc":
$ find /usr -type d -name gcc

Searching only for regular files named "test1":
$ find /usr -type f -name test1

Using Advanced find Options

Another good use of **find** is being able to run commands on the files that match your search criteria. The -exec option is used for this purpose.

```
$ find -name "*.swp" -exec rm {} ';'
```

Finds and removes files
that ends with **.swp**

*.swp

To find and remove all files that end with .swp:
$ find -name "*.swp" -exec rm {} ';'

The {} (squiggly brackets) is a place holder that will be filled with all the file names that result from the **find** expression, and the preceding command will be run on each one individually.

Note that you have to end the command with either ';' (including the single-quotes) or \; Both forms are fine.

One can also use the -ok option which behaves the same as -exec except that **find** will prompt you for permission before executing the command. This makes it a good way to test your results before blindly executing any potentially dangerous commands.

Finding Files Based on Time and Size

It is sometimes the case that you wish to find files according to attributes such as when they were created, last used, etc, or based on their size. Both are easy to accomplish.

Finding based on time:
$ find / -ctime 3

Here, -ctime is when the inode meta-data (i.e., file ownership, permissions, etc) last changed; it is often, but not necessarily when the file was first created. You can also search for accessed/last read (-atime) or modified/last written (-mtime) times. The number is the number of days and can be expressed as either a number (n) that means exactly that value, +n which means greater than that number, or -n which means less than that number. There are similar options for times in minutes (as in -cmin, -amin, and -mmin).

Finding based on sizes:

$ find / -size 0

Note the size here is in 512-byte blocks, by default; you can also specify bytes (**c**), kilobytes (**k**), megabytes (**M**), gigabytes (**G**), etc. As with the time numbers above, file sizes can also be exact numbers (n), +n or -n. For details consult the **man** page for **find**.

For example, to find files greater than 10 MB in size and running a command on those files:
$ find / -size +10M -exec command {} ';'

Working with Files

Linux provides many commands that help you in viewing the contents of a file, creating a new file or an empty file, changing the **timestamp** of a file, and removing and renaming a file or directory. These commands help you in managing your data and files and in ensuring that the correct data is available at the correct location.

In this section, you will learn how to manage files.

Viewing Files

You can use the following utilities to view files:

Command	Usage
cat	Used for viewing files that are not very long; it does not provide any scroll-back.
tac	Used to look at a file backwards, one line at a time.
less	Used to view larger files because it is a paging program; it pauses at each screenful of text, provides scroll-back capabilities, and lets you search and navigate within the file. Note: Use / to search for a pattern in the forward direction and ? for a pattern in the backward direction.
tail	Used to print the last 10 lines of a file by default. You can change the number

	of lines by doing -n 15 or just -15 if you wanted to look at the last 15 lines instead of the default.
head	The opposite of **tail**; by default it prints the first 10 lines of a file.

touch and mkdir

touch is often used to set or update the access, change, and modify times of files. By default it resets a file's time stamp to match the current time.

However, you can also create an **empty** file using touch:
$ touch <filename>

This is normally done to create an empty file as a placeholder for a later purpose.

touch provides several options, but here is one of interest:

- The -t option allows you to set the date and time stamp of the file.

To set the time stamp to a specific time:
$ touch -t 03201600 myfile

This sets the file, myfile's, time stamp to 4 p.m., March 20th (03 20 1600).

mkdir is used to create a directory.

- To create a sample directory named sampdir under the current directory, type mkdir sampdir.

- To create a sample directory called sampdir under /usr, type mkdir /usr/sampdir.

Removing a directory is simply done with **rmdir.** The directory must be empty or it will fail. To remove a directory and all of its contents you have to do rm -rf as we shall discuss.

Removing a File

Command	Usage
mv	Rename a file

rm	Remove a file
rm –f	Forcefully remove a file
rm –i	Interactively remove a file

If you are not certain about removing files that match a pattern you supply, it is always good to run **rm** interactively (rm –i) to prompt before every removal.

Renaming or Removing a Directory

rmdir works only on empty directories; otherwise you get an error.

While typing rm –rf is a fast and easy way to remove a whole file system tree recursively, it is extremely dangerous and should be used with the utmost care, especially when used by root (recall that recursive means drilling down through all sub-directories, all the way down a tree). Below are the commands used to rename or remove a directory:

Command	Usage
Mv	Rename a directory
Rmdir	Remove an empty directory
rm -rf	Forcefully remove a directory recursively

Modifying the Command Line Prompt

The **PS1** variable is the character string that is displayed as the prompt on the command line. Most distributions set **PS1** to a known default value, which is suitable in most cases. However, users may want custom information to show on the command line. For example, some system administrators require the user and the host system name to show up on the command line as in:

student@quad32 $

This could prove useful if you are working in multiple roles and want to be always reminded of who you are and what machine you are on. The prompt above could be implemented by setting the PS1 variable to: \u@\h \$

For example:

```
$ echo $PS1
\$
$ PS1="\u@\h  \$ "
coop@quad64 $ echo $PS1
\u@\h \$
coop@quad64 $
```

Package Management Systems on Linux

The core parts of a Linux distribution and most of its add-on software are installed via the **Package Management System**. Each package contains the files and other instructions needed to make one software component work on the system. Packages can depend on each other. For example, a package for a Web-based application written in PHP can depend on the PHP package.

There are two broad families of package managers: those based on **Debian** and those which use **RPM** as their low-level package manager. The two systems are incompatible, but provide the same features at a broad level.

In this section, you will learn how to install, remove, or search for packages using the different package management tools.

Package Managers: Two Levels

Both package management systems provide two tool levels: a low-level tool (such as **dpkg** or **rpm**), takes care of the details of unpacking individual packages, running scripts, getting the software installed correctly, while a high-level tool (such as **apt-get**, **yum**, or **zypper**) works with groups of packages, downloads packages from the vendor, and figures out dependencies.

Most of the time users need work only with the high-level tool, which will take care of calling the low-level tool as needed. Dependency tracking is a particularly important feature of the high-level tool, as it handles the details of finding and installing each dependency for you. Be careful, however, as installing a single package could result in many dozens or even hundreds of dependent packages being installed.

Working With Different Package Management Systems

- The **Advanced Packaging Tool** (apt) is the underlying package management system that manages software on Debian-based systems. While it forms the backend for graphical package managers, such as the **Ubuntu Software Center** and **synaptic**, its
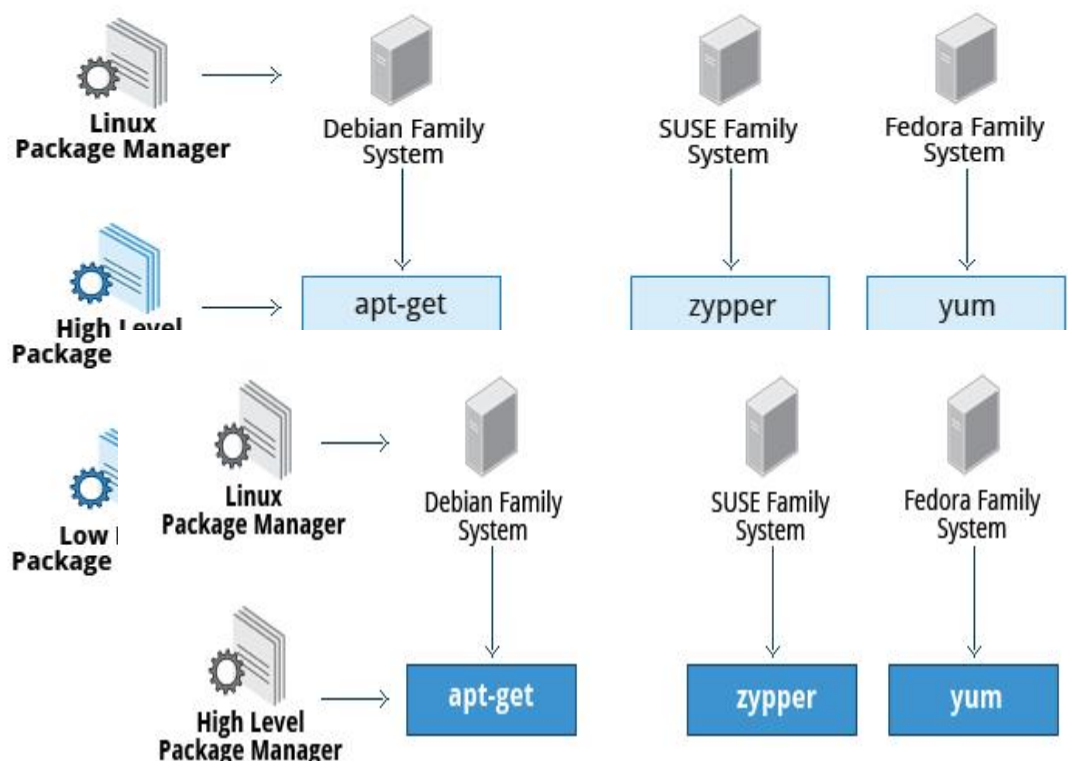
native user interface is at the command line, with programs that include apt-get and apt-cache.

- **Yellowdog Updater Modified** (**yum**) is an open-source command-line package-management utility for RPM-compatible Linux systems, basically what we have called the **Fedora** family. **yum** has both command line and graphical user interfaces.

- **zypper** is a package management system for **openSUSE** that is based on RPM. **zypper** also allows you to manage repositories from the command line. **zypper** is fairly straightforward to use and resembles **yum** quite closely.

To learn the basic packaging commands, click the link below:

Basic Packaging Commands

Operation	RPM	Deb
Install a package	rpm –i foo.rpm	dpkg --install foo.deb
Install a package with dependencies from repository	yum install foo	apt-get install foo
Remove a package	rpm –e foo.rpm	dpkg --remove foo.deb
Remove a package and dependencies using repository	yum remove foo	apt-get remove foo
Update package to a newer version	rpm –U foo.rpm	dpkg --install foo.deb
Update package using repository and resolving dependencies	yum update foo	apt-get upgrade foo
Update entire system	yum update	apt-get dist-upgrade
Show all installed packages	rpm –qa or yum list installed	dpkg --list
Get information about an installed package including files	rpm –qil foo	Dpkg --listfiles foo
Shaow available package with "foo" in name	yum list foo	apt-cache search foo
Show all available packages	yum list	apt-cache dumpavail
What packages does a file belong to?	rpm –qf file	dpkg --search file

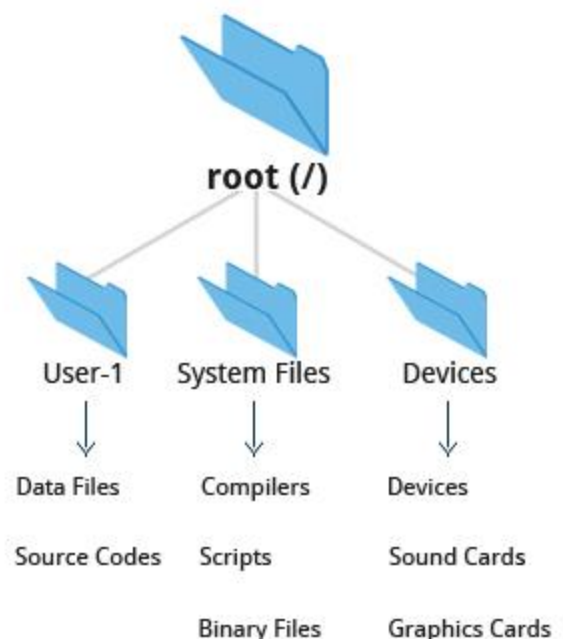**You have completed this chapter. Let's summarize the key concepts covered.**

- Virtual terminals (VT) in Linux are consoles, or command line terminals that use the connected monitor and keyboard.

- Different Linux distributions start and stop the graphical desktop in different ways.

- A terminal emulator program on the graphical desktop works by emulating a terminal within a window on the desktop.

- The Linux system allows you to either log in via text terminal or remotely via the console.

- When typing your password, nothing is printed to the terminal, not even a generic symbol to indicate that you typed.

- The preferred method to shut down or reboot the system is to use the **shutdown** command.

- There are two types of **pathnames:** absolute and relative.

- An absolute pathname begins with the root directory and follows the tree, branch by branch, until it reaches the desired directory or file.

  - A relative pathname starts from the present working directory.

  - Using **hard** and **soft** (**symbolic**) links is extremely useful in Linux.
  - **cd** remembers where you were last, and lets you get back there with cd - .

  - **locate** performs a database search to find all file names that match a given pattern.

  - **find** locates files recursively from a given directory or set of directories.

  - **find** is able to run commands on the files that it lists, when used with the -exec option.

  - **touch** is used to set the access, change, and edit times of files as well as to create empty files.

  - The **Advanced Packaging Tool** (apt) package management system is used to manage installed software on Debian-based systems.

  - You can use the **Yellowdog Updater Modified** (yum) open-source command-line package-management utility for**RPM**-compatible Linux operating systems.

  - The **zypper** package management system is based on RPM and used for openSUSE.

Introduction to File systems

In Linux (and all UNIX-like operating systems) it is often said "Everything is a file", or at least it is treated as such. This means whether you are dealing with normal data files and documents, or with devices such as sound cards and printers, you interact with them through the same kind of Input/Output (I/O) operations. This simplifies things: you open a "file" and perform normal operations like reading the file and writing on it (which is one reason why text editors, which you will learn about in an upcoming section, are so important.)

On many systems (including Linux), the **file system** is structured like a tree. The tree is usually portrayed as inverted, and starts at what is most often called the **root directory,** which marks the beginning of the hierarchical file system and is also some times referred to as the **trunk,** or simply denoted by **/**. The root directory is **not** the same as the root user. The hierarchical file system also contains other elements in the path (directory names) which are separated by forward slashes (/) as

root (/)

User-1   System Files   Devices

Data Files   Compilers   Devices

Source Codes   Scripts   Sound Cards

Binary Files   Graphics Cards

in /usr/bin/awk, where the last element is the actual file name.

In this section, you will learn about some basic concepts including the file system hierarchy as well as about **disk partitions**.

File system Hierarchy Standard

The **File system Hierarchy Standard** (**FHS**) grew out of historical standards from early versions of UNIX, such as the **Berkeley Software Distribution** (**BSD**) and others. The FHS provides Linux developers and system administrators with a standard directory structure for the file system, which provides consistency between systems and distributions.

Visit http://www.pathname.com/fhs/ for a list of the main directories and their contents in Linux systems.

Linux supports various file system types created for Linux, along with compatible file systems from other operating systems such as **Windows** and **MacOS**. Many older, legacy file systems, such as **FAT**, are supported.

Some examples of file system types that Linux supports are:

1. **ext3**, **ext4**, **btrfs**, **xfs** (native Linux file systems)

2. **vfat**, **ntfs, hfs** (file systems from other operating systems)

Here is an example of a FHS-compliant system. (Other FHS-compliant layouts are possible.)

	shareable	**unshareable**
static	/usr	/etc
	/opt	/boot
variable	/var/mail	/var/run
	/var/spool/news	/var/lock

Directory	Description
bin	Essential command binaries
boot	Static files of the boot loader
dev	Device files
etc	Host-specific system configuration
lib	Essential shared libraries and kernel modules
media	Mount point for removeable media
mnt	Mount point for mounting a filesystem temporarily
opt	Add-on application software packages
sbin	Essential system binaries
srv	Data for services provided by this system
tmp	Temporary files
usr	Secondary hierarchy
var	Variable data

Partitions in Linux

Each file system resides on a hard disk **partition**. Partitions help to organize the contents of disks according to the kind of data contained and how it is used. For example, important programs required to run the system are often kept on a separate partition (known as root or /) than the one that contains files owned by regular users of that system (/home). In addition, temporary files created and destroyed during the normal operation of Linux are often located on a separate partition; in this way, using all available space on a particular partition may not fatally affect the normal operation of the system.

(root (/)) — Important Programs
(/home) — User files
(/tmp) — Temporary files
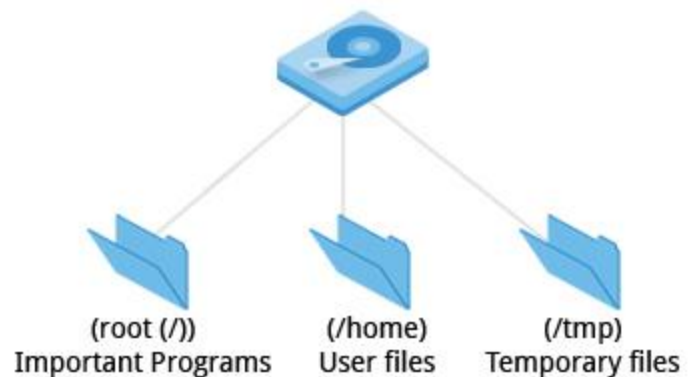
Mount Points

Before you can start using a file system, you need to **mount** it to the file system tree at a **mount point.** This is simply a directory (which may or may not be empty) where the file system is to be attached (mounted). Sometimes you may need to create the directory if it doesn't already exist.

Mount points (/)

| (/) | (/home) | (/var) |

File systems

/dev/sda1   /dev/sda5   /dev/sda6

/dev/sda
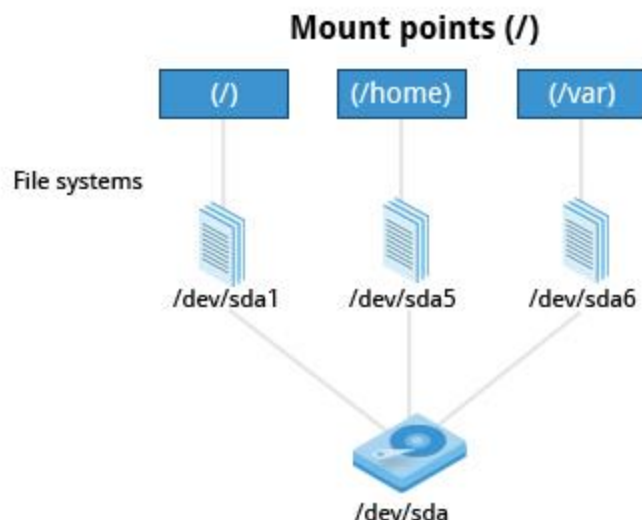
Warning: If you mount a file system on a non-empty directory, the former contents of that directory are covered-up and not accessible until the file system is unmounted. Thus mount points are usually empty directories.

Mount Points

The **mount** command is used to attach a file system (which can be local to the computer or, as we shall discuss, on a network) somewhere within the file system tree. Arguments include the **device node** and **mount point**. For example,

$ mount /dev/sda5 /home

will attach the file system contained in the disk partition associated with the /dev/sda5 device node, into the file system tree at the /home mount point. (Note that unless the system is otherwise configured only the root user has permission to run **mount**.) If you want it to be automatically available every time the system starts up, you need to edit the file/etc/fstab accordingly (the name is short for **File system Table**). Looking at this file will show you the configuration of all pre-configured file systems. man fstab will display how this file is used and how to configure it.

Typing **mount** without any arguments will show all presently mounted file systems.

The command df -Th (**disk-free**) will display information about mounted file systems including usage statistics about currently used and available space.

The Network File system

Using **NFS** (the **Network File system**) is one of the methods used for sharing data across physical systems. Many system administrators mount remote users' home directories on a **server** in order to give them access to the same files and configuration files across multiple **client** systems. This allows the users to log in to different computers yet still have access to the same files and resources.
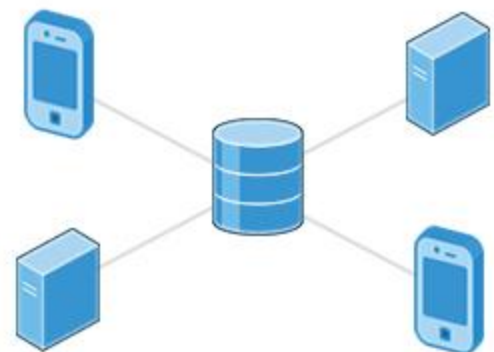
NFS on the Server

We will now look in detail at how to use NFS on the server machine.

On the server machine, NFS daemons (built-in networking and service processes in Linux) and other system servers are typically started with the following command: sudo service nfs start

The text file /etc/exports contains the directories and permissions that a host is willing to share with other systems over NFS. An entry in this file may look like the following:

/projects *.example.com(rw)

This entry allows the directory /projects to be mounted using NFS with read and write (rw) permissions and shared with other hosts in the example.com domain. As we will detail in the next chapter, every file in Linux has 3 possible permissions: **read** (r), **write** (w) and **execute** (x).

After modifying the /etc/exports file, you can use the exportfs -av command to notify Linux about the directories you are allowing to be remotely mounted using NFS (restarting NFS with sudo service nfs restart will also work, but is heavier as it halts NFS for a short while before starting it up again).

NFS on the Client

On the client machine, if it is desired to have the remote file system mounted automatically upon system boot, the /etc/fstab file is modified to accomplish this. For example, an entry in the client's /etc/fstab file might look like the following:

servername:/projects /mnt/nfs/projects nfs defaults 0 0

You can also mount the remote file system without a reboot or as a one-time mount by directly using the mount command:

$ mount servername:/projects /mnt/nfs/projects

Remember, if /etc/fstab is not modified, this remote mount will not be present the next time the system is restarted.

proc File system

Certain file systems like the one mounted at /proc are called **pseudo file systems** because they have no permanent presence anywhere on disk.

The /proc file system contains virtual files (files that exist only in memory) that permit viewing constantly varying kernel data. This file system contains files and directories that mimic kernel structures and configuration information. It doesn't contain *real* files but runtime system information (e.g. system memory, devices mounted, hardware configuration, etc). Some important files in /proc are:

/proc/cpuinfo
/proc/interrupts
/proc/meminfo
/proc/mounts
/proc/partitions
/proc/version

/proc has subdirectories as well, including:

/proc/<Process-ID-#>
/proc/sys

Overview of Home Directories

Now that you know about the basics of file systems, let's learn about the file system architecture and directory structure in Linux.

/home

FACULTY   STAFF   STUDENTS

Each user has a **home directory**, usually placed under/home. The /root (slash-root) directory on modern Linux systems is no more than the root user's home directory.

The /home directory is often mounted as a separate file system on its own partition, or even exported (shared) remotely on a network through NFS.
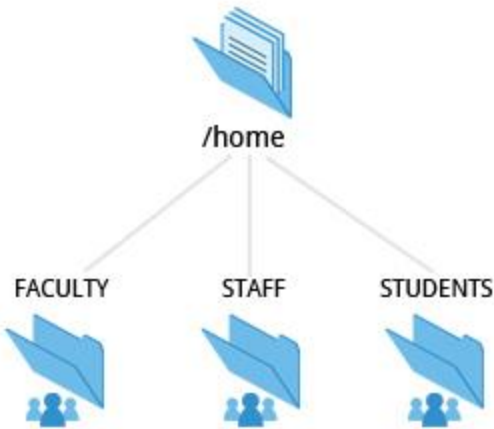
Sometimes you may group users based on their department or function. You can then create subdirectories under the/home directory for each of these groups. For example, a school may organize /home with something like the following:

/home/faculty/
/home/staff/
/home/students/

In this section, you will learn to identify and differentiate between the different directories available in Linux.

The /bin and /sbin Directories

The /bin directory contains executable binaries, essential commands used in single-user mode, and essential commands required by all system users, such as:

Command	Usage
ps	Produces a list of processes along with status information for the system.
ls	Produces a listing of the contents of a directory.
cp	Used to copy files.

To view a list of programs in the /bin directory, type: ls /bin

Commands that are not essential for the system in single-user mode are placed in the /usr/bin directory, while the /sbin directory is used for essential binaries related to system administration, such as **ifconfig** and **shutdown.**There is also a /usr/sbin directory for less essential system administration programs.

Sometimes /usr is a separate file system that may not be available/mounted in single-user mode. This was why essential commands were separated from non-essential commands. However, in some of the most modern Linux systems this distinction is considered obsolete, and /usr/bin and /bin are actually just linked together as are /usr/sbin and /sbin

The /dev Directory

The /dev directory contains **device nodes**, a type of pseudo-file used by most hardware and software devices, except for network devices. This directory is:

- Empty on the disk partition when it is not mounted

- Contains entries which are created by the **udev** system, which creates and manages device nodes on Linux, creating them dynamically when devices are found. The /dev directory contains items such as:

- /dev/sda1 (first partition on the first hard disk)

- /dev/lp1 (second printer)

- /dev/dvd1 (first DVD drive

The /var and /etc Directories

The /var directory contains files that are expected to change in size and content as the system is running (**var** stands for **variable**) such as the entries in the following directories:

- System log files: /var/log

- Packages and database files: /var/lib

- Print queues: /var/spool

- Temp files: /var/tmp

The /var directory may be put in its own file system so that growth of the files can be accommodated and the file sizes do not fatally affect the system. Network services directories such as /var/ftp (the FTP service) and /var/www (the HTTP web service) are also found under /var.

root (/)

etc   var

System log files:
/var/log

Print queues: Temp files:
/var/spool   /var/tmp

Packages and
database files:
/var/lib
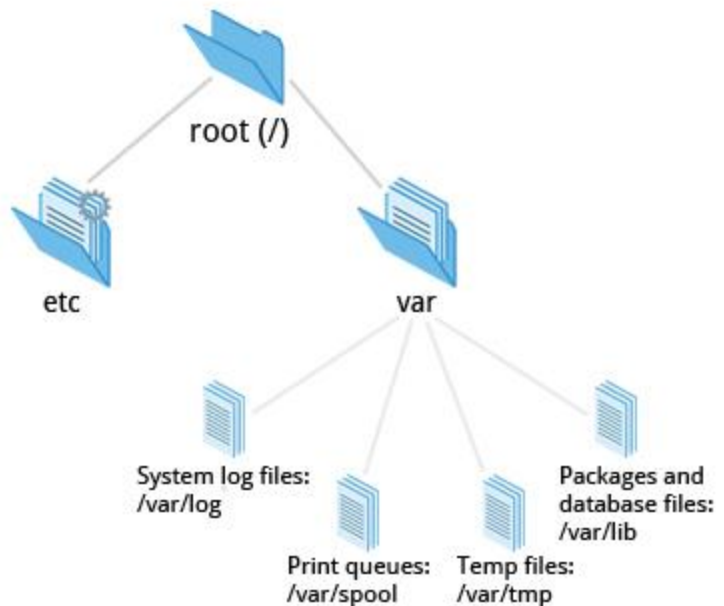
The /etc directory is the home for system configuration files. It contains no binary programs, although there are some executable scripts. For example, the file resolv.conf tells the system where to go on the network to obtain host name to IP address mappings (DNS). Files like passwd,shadow and group for managing user accounts are found in the /etc directory. System run level scripts are found in subdirectories of /etc. For example, /etc/rc2.d contains links to scripts for entering and leaving run level 2. The rc directory historically stood for *Run Commands*. Some distros extend the contents of /etc. For example, **Red Hat** adds the sysconfig subdirectory that contains more configuration files.

The /boot Directory

The /boot directory contains the few essential files needed to boot the system. For every alternative kernel installed on the system there are four files:

1. vmlinuz: the compressed Linux kernel, required for booting

2. initramfs: the initial ram file system, required for booting, sometimes called initrd, not initramfs

3. config: the kernel configuration file, only used for debugging and bookkeeping

4. System.map: kernel symbol table, only used for debugging

Each of these files has a kernel version appended to its name.

The **Grand Unified Bootloader** (**GRUB**) files (such as /boot/grub/grub.conf or /boot/grub2/grub2.cfg) are also found under the /boot directory.

The images show an example listing of the /boot directory, taken from a **CentOS** system that has three installed kernels. Names would vary and things would look somewhat different on a different distribution.

The /lib and /media Directories

/lib contains libraries (common code shared by applications and needed for them to run) for the essential programs in /bin and /sbin. These library filenames either start with ld or lib, for example, /lib/libncurses.so.5.7.

Most of these are what are known as **dynamically loaded libraries** (also known as **shared libraries** or **Shared Objects(SO)**). On some Linux distributions there exists a /lib64 directory containing 64-bit libraries, while /lib contains 32-bit versions.

root (/)

lib   bin   sbin   media

removable media
cds
dvds

Kernel **modules** (kernel code, often device drivers, that can be loaded and unloaded without re-starting the system) are located in /lib/modules/<kernel-version-number>.

The /media directory is typically located where removable media, such as CDs, DVDs and USB drives are mounted. Unless configuration prohibits it, Linux automatically mounts the removable media in the /media directory when they are detected.
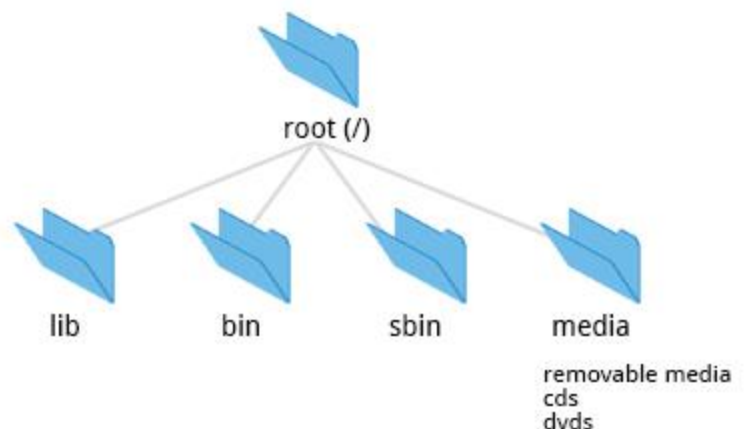
Additional Directories Under /:

The following is a list of additional directories under /and their use:

Directory name	Usage
/opt	Optional application software packages.
/sys	Virtual pseudo-file system giving information about the system and the hardware. Can be used to alter system parameters and for debugging purposes.
/srv	Site-specific data served up by the system. Seldom used.
/tmp	Temporary files; on some distributions erased across a reboot and/or may actually be a ramdisk in memory.
/usr	Multi-user applications, utilities and data.

Subdirectories under /usr

The /usr directory contains non-essential programs and scripts (in the sense that they should not be needed to initially boot the system) and has at least the following sub-directories:

Directory name	Usage
/usr/include	Header files used to compile applications.
/usr/lib	Libraries for programs in /usr/bin and /usr/sbin.
/usr/lib64	64-bit libraries for 64-bit programs in /usr/bin and /usr/sbin.
/usr/sbin	Non-essential system binaries, such as system daemons.
/usr/share	Shared data used by applications, generally architecture-independent.
/usr/src	Source code, usually for the Linux kernel.
/usr/X11R6	**X Window** configuration files; generally obsolete.
/usr/local	Data and programs specific to the local machine. Subdirectories include bin, sbin, lib, share, include, etc.
/usr/bin	This is the primary directory of executable commands on the system.

Comparing Files

Now that you know about the file system and its structure, let's learn how to manage files and directories.

diff is used to compare files and directories. This often-used utiility program has many useful options (see man diff) including:

diff Option	Usage
-c	Provides a listing of differences that include 3 lines of **context**before and after the lines differing in content
-r	Used to **recursively** compare subdirectories as well as the current directory
-i	**Ignore** the case of letters
-w	Ignore differences in spaces and tabs (**white space**)

To compare two files, at the command prompt, type diff <filename1> <filename2>

In this section, you will learn additional methods for comparing files and how to apply **patches** to files.

Using diff3 and patch

You can compare three files at once using **diff3**, which uses one file as the reference basis for the other two. For example, suppose you and a co-worker both have made modifications to the same file working

at the same time independently. **diff3** can show the differences based on the common file you both started with. The syntax for **diff3** is as follows:

$ diff3 MY-FILE COMMON-FILE YOUR-FILE

Many modifications to source code and configuration files are distributed utilizing **patches**, which are applied, not suprisingly, with the **patch** program. A patch file contains the **deltas** (changes) required to update an older version of a file to the new one. The patch files are actually produced by running **diff** with the correct options, as in:

$ diff -Nur originalfile newfile > patchfile

Distributing just the patch is more concise and efficient than distributing the entire file. For example, if only one line needs to change in a file that contains 1,000 lines, the **patch** file will be just a few lines long.

To apply a patch you can just do either of the two methods below:

$ patch -p1 < patchfile
$ patch originalfile patchfile

The first usage is more common as it is often used to apply changes to an entire directory tree, rather than just one file as in the second example. To understand the use of the -p1 option and many others, see the **man** page for **patch**.

The graphic shows a patch file produced by **diff**

Using the 'file' utility

In Linux, a file's extension often does not categorize it the way it might in other operating systems. One can not assume that a file named file.txt is a text file and not an executable program. In Linux a file name is generally more meaningful to the user of the system than the system itself; in fact most applications directly examine a file's contents to see what kind of object it is rather than relying on an extension. This is very different from the way **Windows** handles filenames, where a filename ending with .exe, for example, represents an executable binary file.

The real nature of a file can be ascertained by using the **file** utility. For the file names given as arguments, it examines the contents and certain characteristics to determine whether the files are plain text, shared libraries, executable programs, scripts, or something else.

Backing Up Data

There are many ways you can back up data or even your entire system. Basic ways to do so include use of simple copying with **cp** and use of the more robust **rsync**.

Both can be used to synchronize entire directory trees. However, **rsync** is more efficient because it checks if the file being copied already exists. If the file exists and there is no change in size or modification time, **rsync** will avoid an unnecessary copy and save time. Furthermore, because **rsync** copies only the parts of files that have actually changed, it can be very fast.

cp can only copy files to and from destinations on the local machine (unless you are copying to or from a file system mounted using NFS), but **rsync** can also be used to copy files from one machine to another. Locations are designated in the target:path form where target can be in the form of [user@]host. The user@ part is optional and used if the remote user is different from the local user.

rsync is very efficient when recursively copying one directory tree to another, because only the differences are transmitted over the network. One often synchronizes the destination directory tree with the origin, using the -r option to recursively walk down the directory tree copying all files and directories below the one listed as the source

rsync is a very powerful utility. For example, a very useful way to back up a project directory might be to use the following command:

$ rsync -r project-X archive-machine:archives/project-X

Note that **rsync** can be very destructive! Accidental misuse can do a lot of harm to data and programs by inadvertently copying changes to where they are not wanted. Take care to specify the correct options and paths. It is highly recommended that you first test your **rsync** command using the -dry-run option to ensure that it provides the results that you want.

To use **rsync** at the command prompt, type rsync sourcefile destinationfile, where either file can be on the local machine or on a networked machine.

The contents of sourcefile are copied to destinationfile.

Compressing Data

File data is often compressed to save disk space and reduce the time it takes to transmit files over networks.

Linux uses a number of methods to perform this compression including:

Command	Usage
gzip	The most frequently used Linux compression utility
bzip2	Produces files significantly smaller than those produced by **gzip**

xz	The most space efficient compression utility used in Linux
zip	Is often required to examine and decompress archives from other operating systems

These techniques vary in the efficiency of the compression (how much space is saved) and in how long they take to compress; generally the more efficient techniques take longer. Decompression time doesn't vary as much across different methods.

In addition the **tar** utility is often used to group files in an **archive** and then compress the whole archive at once.

Compressing Data Using gzip

gzip is the most oftenly used Linux compression utility. It compresses very well and is very fast. The following table provides some usage examples:

Command	Usage
gzip *	Compresses all files in the current directory; each file is compressed and renamed with a .gz extension.
gzip -r projectX	Compresses all files in the projectX directory along with all files in all of the directories under projectX.
gunzip foo	De-compresses foo found in the file foo.gz. Under the hood, gunzipcommand is actually the same as gzip –d.

Compressing Data Using bzip2

bzip2 has syntax that is similar to **gzip** but it uses a different compression algorithm and produces significantly smaller files, at the price of taking a longer time to do its work. Thus, It is more likely to be used to compress larger files.

Examples of common usage are also similar to **gzip**:

Command	Usage
bzip2 *	Compress all of the files in the current directory and replaces each file with a file renamed with a .bz2 extension.
bunzip2 *.bz2	Decompress all of the files with an extension of .bz2 in the current directory. Under the hood, bunzip2 is the same as calling bzip2 -d.

Compress Data Using xz

xz is the most space efficient compression utility used in Linux and is now used by www.kernel.org to store archives of the Linux kernel. Once again it trades a slower compression speed for an even higher compression ratio.

Some usage examples:

Command	Usage
$ xz *	Compress all of the files in the current directory and replace each file with one with a .xz extension.
xz foo	Compress the file foo into foo.xz using the default compression level (-6), and remove foo if compression succeeds.
xz -dk bar.xz	Decompress bar.xz into bar and don't remove bar.xz even if decompression is successful.
xz -dcf a.txt b.txt.xz > abcd.txt	Decompress a mix of compressed and uncompressed files to standard output, using a

	single command.
$ xz -d *.xz	Decompress the files compressed using xz.

Compressed files are stored with a .xz extension.

Handling Files Using zip

The **zip** program is not often used to compress files in Linux, but is often required to examine and decompress archives from other operating systems. It is only used in Linux when you get a zipped file from a **Windows** user. It is a legacy program.

Command	Usage
zip backup *	Compresses all files in the current directory and places them in the file backup.zip.
zip -r backup.zip ~	Archives your login directory (~) and all files and directories under it in the file backup.zip.
unzip backup.zip	Extracts all files in the file backup.zip and places them in the current directory.

Archiving and Compressing Data Using tar

Historically, **tar** stood for "tape archive" and was used to archive files to a magnetic tape. It allows you to create or extract files from an archive file, often called a **tarball**. At the same time you can optionally compress while creating the archive, and decompress while extracting its contents.

Here are some examples of the use of **tar**:

Command	Usage
$ tar xvf mydir.tar	Extract all the files in mydir.tar into the mydir directory
$ tar zcvf mydir.tar.gz mydir	Create the archive and compress with gzip
$ tar jcvf mydir.tar.bz2 mydir	Create the archive and compress with bz2
$ tar Jcvf mydir.tar.xz mydir	Create the archive and compress with xz
$ tar xvf mydir.tar.gz	Extract all the files in mydir.tar.gz into the mydir directory. Note you do **not** have to tell tar it is in gzip format.

You can separate out the archiving and compression stages, as in:

$ tar mydir.tar mydir ; gzip mydir.tar
$ gunzip mydir.tar.gz ; tar xvf mydir.tar

but this is slower and wastes space by creating an unneeded intermediary .tar file.

Disk-to-Disk Copying

The **dd** program is very useful for making copies of raw disk space. For example, to back up your **Master Boot Record** (**MBR**) (the first 512 byte sector on the disk that contains a table describing the partitions on that disk), you might type:

dd if=/dev/sda of=sda.mbr bs=512 count=1

To use **dd** to make a copy of one disk onto another, (**WARNING!**) **deleting everything that previously existed on the second disk**, type:

dd if=/dev/sda of=/dev/sdb

An exact copy of the first disk device is created on the second disk device.

Do not experiment with this command as written above as it can erase a hard disk!

Exactly what the name **dd** stands for is an often-argued item. The words **data definition** is the most popular theory and has roots in early **IBM** history. Often people joke that it means **disk destroyer** and other variants such as **delete data**!

- /var may be put in its own file system so that growth can be contained and not fatally affect the system.

- /boot contains the basic files needed to boot the system

- **patch** is a very useful tool in Linux. Many modifications to source code and configuration files are distributed with patch files as they contain the deltas or changes to go from an old version of a file to the new version of a file.

- File extensions in Linux do not necessarily mean that a file is of a certain type.

- **cp** is used to copy files on the local machine while **rsync** can also be used to copy files from one machine to another as well as synchronize contents.

- **gzip**, **bzip2**, **xz** and **zip** are used to compress files.

- **tar** allows you to create or extract files from an archive file, often called a tarball. You can optionally compress while creating the archive, and decompress while extracting its contents

- **dd** can be used to make large exact copies even of entire disk partitions efficiently.

Identifying the Current User

As you know, Linux is a multiuser operating system; i.e., more than one user can log on at the same time.

- To list the currently logged-on users, type who

- To identify the current user, type whoami

Giving **who** the -a option will give more detailed information.

Basics of Users and Groups

Linux uses **groups** for organizing users. Groups are collections of accounts with certain shared permissions. Control of group membership is administered through the /etc/group file, which shows a list of groups and their members. By default, every user belongs to a default or primary group. When a user logs in, the group membership is set for their primary group and all the members enjoy the same level of access and privilege. Permissions on various files and directories can be modified at the group level.

All Linux users are assigned a unique user ID (**uid**), which is just an integer, as well as one or more group ID's (**gid**), including a default one which is the same as the user ID.

Historically **Fedora**-family systems start **uid**'s at 500; other distributions begin at 1000.

These numbers are associated with names through the files /etc/passwd and /etc/group.

For example, the first file might contain:
george:x:1002:1002:George Metesky:/home/george:/bin/bash
and the second george:x:1002

Groups are used to establish a set of users who have common interests for the purposes of access rights, privileges, and security considerations. Access rights to files (and devices) are granted on the basis of the user and the group they belong to.

Adding and Removing Users

Distributions have straightforward graphical interfaces for creating and removing users and groups and manipulating group membership. However, it is often useful to do it from the command line or from within shell scripts. Only the root user can add and remove users and groups.

Adding a new user is done with **useradd** and removing an existing user is done with **userdel**. In the simplest form an account for the new user turkey would be done with:

$ sudo useradd turkey

which by default sets the home directory to /home/turkey, populates it with some basic files (copied from /etc/skel) and adds a line to /etc/passwd such as:

turkey:x:502:502::/home/turkey:/bin/bash

and sets the default shell to /bin/bash. Removing a user account is as easy as typing userdel turkey However, this will leave the /home/turkey directory intact. This might be useful if it is a temporary inactivation. To remove the home directory while removing the account one needs to use the **-r** option to **userdel**.

Typing **id** with no argument gives information about the current user, as in:

$ id
uid=500(george) gid=500(george) groups=106(fuse),500(george)

If given the name of another user as an argument, **id** will report information about that other user.

Adding a new group is done with **groupadd**:

$ sudo /usr/sbin/groupadd anewgroup

The group can be removed with

$ sudo /usr/sbin/groupdel anewgroup

Adding a user to an already existing group is done with **usermod**. For example, you would first look at what groups the user already belongs to:

$ groups turkey
turkey : turkey

and then add the new group:

```
$ sudo /usr/sbin/usermod -G anewgroup turkey
$ groups turkey
turkey: turkey anewgroup
```

These utilities update /etc/group as necessary. **groupmod** can be used to change group properties such as the Group ID (gid) with the -g option or its name with the -n option.

Removing a user from the group is a somewhat trickier. The **-G** option to **usermod** must give a complete list of groups. Thus if you do:

```
$ sudo /usr/sbin/usermod -G turkey turkey
$ groups turkey
turkey : turkey
```

only the **turkey** group will be left.

The root Account

The **root** account is very powerful and has full access to the system. Other operating systems often call this the **administrator** account; in Linux it is often called the **superuser** account. You must be extremely cautious before granting full root access to a user; it is rarely if ever justified. External attacks often consist of tricks used to elevate to the root account.

However, you can use the **sudo** feature to assign more limited privileges to user accounts:
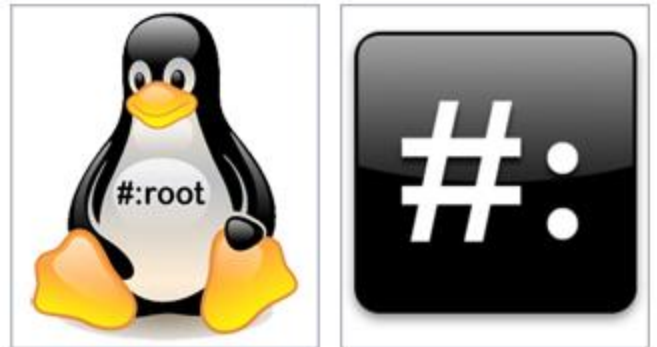
- on only a temporary basis.

- only for a specific subset of commands.

su and sudo

When assigning elevated privileges, you can use the command **su** (switch or substitute user) to launch a new shell running as another user (you must type the password of the user you are becoming). Most often this other user is root, and the new shell allows the use of elevated privileges until it is exited. It is almost always a bad (dangerous for both security and stability) practice to use **su** to become root. Resulting errors can include deletion of vital files from the system and security breaches.

Granting privileges using **sudo** is less dangerous and is preferred. By default, **sudo** must be enabled on a per-user basis. However, some distributions (such as **Ubuntu**) enable it by default for at least one main user, or give this as an installation option.

In the chapter on Security that follows shortly, we will describe and compare **su** and **sudo** in detail.

Elevating to root Account

To fully become root, one merely types **su** and then is prompted for the root password.

To execute just one command with root privilege type sudo <command>. When the command is complete you will return to being a normal unprivileged user.

sudo configuration files are stored in the /etc/sudoers file and in the /etc/sudoers.d/ directory. By default, the sudoers.d directory is empty

Startup Files

In Linux, the command shell program (generally **bash)** uses one or more startup files to configure the environment. Files in the /etc directory define global settings for all users while Initialization files in the user's home directory can include and/or override the global settings.

The startup files can do anything the user would like to do in every command shell, such as:

- Customizing the user's prompt

- Defining command-line shortcuts and aliases

- Setting the default text editor

- Setting the **path** for where to find executable programs
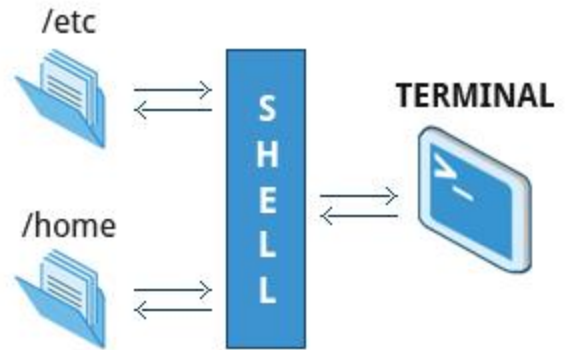
Order of the Startup Files

When you first login to Linux, /etc/profile is read and evaluated, after which the following files are searched (if they exist) in the listed order:

1. ~/.bash_profile

2. ~/.bash_login

3. ~/.profile

The Linux login shell evaluates whatever startup file that it comes across first and ignores the rest. This means that if it finds ~/.bash_profile, it ignores ~/.bash_login and ~/.profile. Different distributions may use different startup files.

However, every time you create a new shell, or terminal window, etc., you do not perform a full system login; only the ~/.bashrc file is read and evaluated. Although this file is not read and evaluated along with the login shell, most distributions and/or users include the ~/.bashrc file from within one of the three user-owned startup files. In the **Ubuntu**, **openSuse**, and **CentOS** distros, the user must make appropriate changes in the ~/.bash_profile file to include the ~/.bashrc file.

The .bash_profile will have certain extra lines, which in turn will collect the required customization parameters from .bashrc.

Environment Variables

Environment variables are simply named quantities that have specific values and are understood by the command shell, such as **bash**. Some of these are pre-set (built-in) by the system, and others are set by the user either at the command line or within startup and other scripts. An environment variable is actually no more than a character string that contains information used by one or more applications.

There are a number of ways to view the values of currently set environment variables; one can type **set**, **env**, or **export.** Depending on the state of your system, **set** may print out many more lines than the other two methods.

```
$ set
BASH=/bin/bash
BASHOPTS=checkwinsize:cmdhist:expand_aliases:extglob:extquote:force_fignore
BASH_ALIASES=()
...

$ env
SSH_AGENT_PID=1892
GPG_AGENT_INFO=/run/user/me/keyring-Ilf3vt/gpg:0:1
TERM=xterm
SHELL=/bin/bash
...

$ export
declare -x COLORTERM=gnome-terminal
declare -x COMPIZ_BIN_PATH=/usr/bin /
declare -x COMPIZ_CONFIG_PROFILE=ubuntu
```

Setting Environment Variables

By default, variables created within a script are only available to the current shell; child processes (sub-shells) will not have access to values that have been set or modified. Allowing child processes to see the values, requires use of the **export** command.

Task	Command
Show the value of a specific variable	echo $SHELL
Export a new variable value	export VARIABLE=value (or VARIABLE=value; export VARIABLE)
Add a variable permanently	1. Edit ~/.bashrc and add the line export VARIABLE=value 2. Type source ~/.bashrc or just . ~/.bashrc (dot ~/.bashrc); or just start a new

	shell by typing bash

The HOME Variable

HOME is an environment variable that represents the home (or login) directory of the user. **cd** without arguments will change the current working directory to the value of HOME. Note the tilde character (~) is often used as an abbreviation for $HOME. Thus cd $HOME and cd ~ are completely equivalent statements.

Command	Explanation
$ echo $HOME /home/me $ cd /bin	Show the value of the HOME environment variable then change directory (cd) to /bin
$ pwd /bin	Where are we? Use print (or present) working directory (pwd) to find out. As expected /bin
$ cd	Change directory without an argument . . .
$ pwd /home/me	. . . takes us back to HOME as you can now see

The PATH Variable

PATH is an ordered list of directories (the **path**) which is scanned when a command is given to find the appropriate program or script to run. Each directory in the path is separated by colons (:). A null (empty) directory name (or ./) indicates the current directory at any given time.

- :path1:path2

- path1::path2

In the example :path1:path2, there is null directory before the first colon (:). Similarly, for path1::path2 there is null directory between path1 and path2.

To prefix a private bin directory to your path:

```
$ export PATH=$HOME/bin:$PATH
$ echo $PATH
/home/me/bin:/usr/local/bin:/usr/bin:/bin/usr
```

The PS1 Variable

Prompt Statement (PS) is used to customize your**prompt** string in your terminal windows to display the information you want.

PS1 is the primary prompt variable which controls what your command line prompt looks like. The following special characters can be included in PS1 :

\u - User name
\h - Host name
\w - Current working directory
\! - History number of this command
\d - Date

They must be surrounded in single quotes when they are used as in the following example:
```
$ echo $PS1
$
$ export PS1='\u@\h:\w$ '
me@example.com:~$  # new prompt
me@example.com:~$
```

To revert the changes:
```
me@example.com:~$  export PS1='$ '
$
```

Even better practice would be to save the old prompt first and then restore, as in:
```
$ OLD_PS1=$PS1
```

change the prompt, and eventually change it back with:
```
$ PS1=$OLD_PS1
$
```

The SHELL Variable

The environment variable SHELL points to the user's default command shell (the program that is handling whatever you type in a command window, usually **bash**) and contains the full pathname to the shell:

```
$ echo $SHELL
/bin/bash
$
```

Recalling Previous Commands

bash keeps track of previously entered commands and statements in a **history** buffer; you can recall previously used commands simply by using the **Up** and **Down** cursor keys. To view the list of previously executed commands, you can just type history at the command line.

The list of commands is displayed with the most recent command appearing last in the list. This information is stored in ~/.bash_history.

Using History Environment Variables

several associated environment variables can be used to get information about the history file.

HISTFILE stores the location of the history file.

HISTFILESIZE stores the maximum number of lines in the history file.

HISTSIZE stores the maximum number of lines in the history file for the current session.

Finding and Using Previous Commands

Specific keys to perform various tasks:

Key	Usage
Up/Down arrow key	Browse through the list of commands previously executed
!! (Pronounced as **bang-bang**)	Execute the previous command
CTRL-R	Search previously used commands

If you want to recall a command in the history list, but do not want to press the arrow key repeatedly, you can press**CTRL-R** to do a reverse intelligent search.

As you start typing the search goes back in reverse order to the first command that matches the letters you've typed. By typing more successive letters you make the match more and more specific.

The following is an example of how you can use the CTRL-R command to search through the command history:

```
$ ^R                                    # This all happens on 1 line
(reverse-i-search)'s': sleep 1000    # Searched for 's'; matched "sleep"
$ sleep 1000                                  # Pressed Enter to execute the searched command
$
```

Executing Previous Commands

The table describes the syntax used to execute previously used commands.

Syntax	Task
!	Start a history substitution
!$	Refer to the last argument in a line
!n	Refer to the nth command line
!string	Refer to the most recent command starting with string

All history substitutions start with !. In the line $ ls -l /bin /etc /var !$ refers to /var, which is the last argument in the line.

Here are more examples:
$ history

1. echo $SHELL

2. echo $HOME

3. echo $PS1

4. ls -a

5. ls -l /etc/ passwd

6. sleep 1000

7. history

```
$ !1                    # Execute command #1 above
echo $SHELL
/bin/bash
$ !sl                   # Execute the command beginning with "sl"
sleep 1000
$
```

Keyboard Shortcuts

You can use keyboard shortcuts to perform different tasks quickly. The table lists some of these keyboard shortcuts and their uses.

Keyboard Shortcut	Task
CTRL-L	Clears the screen
CTRL-D	Exits the current shell
CTRL-Z	Puts the current process into suspended background
CTRL-C	Kills the current process
CTRL-H	Works the same as backspace
CTRL-A	Goes to the beginning of the line

CTRL-W	Deletes the word before the cursor
CTRL-U	Deletes from beginning of line to cursor position
CTRL-E	Goes to the end of the line
Tab	Auto-completes files, directories, and binaries

Creating Aliases

You can create customized commands or modify the behavior of already existing ones by creating **aliases**. Most often these aliases are placed in your ~/.bashrc file so they are available to any command shells you create.

Typing **alias** with no arguments will list currently defined aliases.

Please note there should not be any spaces on either side of the equal sign and the alias definition needs to be placed within either single or double quotes if it contains any spaces.

The alias projx='cd /home/staff/RandD/src' command is used to create an alias projx for the cd /home/staff/RandD/src command. You can use either double or single quotation marks in this case.

File Ownership

In Linux and other UNIX-based operating systems, every file is associated with a user who is the **owner**. Every file is also associated with a **group** (a subset of all users) which has an interest in the file and certain rights, or permissions: read, write, and execute.

The following utility programs involve user and group ownership and permission setting.

Command	Usage
chown	Used to change user ownership of a file or directory

chgrp	Used to change group ownership
chmod	Used to change the permissions on the file which can be done separately for **owner**, **group** and the rest of the world (often named as **other**.)

File Permission Modes and chmod

Files have three kinds of permissions: read (r), write (w), execute (x). These are generally represented as in rwx. These permissions affect three groups of owners: user/owner (u), group (g), and others (o).

As a result, you have the following three groups of three permissions:

```
rwx: rwx: rwx
 u:  g:  o
```

There are a number of different ways to use **chmod**. For instance, to give the owner and others execute permission and remove the group write permission:

```
$ ls -l a_file
-rw-rw-r-- 1 coop coop 1601 Mar 9 15:04 a_file
$ chmod uo+x,g-w a_file
$ ls -l a_file
-rwxr--r-x 1 coop coop 1601 Mar 9 15:04 a_file
```

where u stands for user (owner), o stands for other (world), and g stands for group.

This kind of syntax can be difficult to type and remember, so one often uses a shorthand which lets you set all the permissions in one step. This is done with a simple algorithm, and a single digit suffices to specify all three permission bits for each entity. This digit is the sum of:

- 4 if read permission is desired.

- 2 if write permission is desired.

- 1 if execute permission is desired.

Thus 7 means read/write/execute, 6 means read/write, and 5 means read/execute.

When you apply this to the **chmod** command you have to give three digits for each degree of freedom, such as in

```
$ chmod 755 a_file
$ ls -l a_file
-rwxr-xr-x 1 coop coop 1601 Mar 9 15:04 a_file
```

Let's see an example of changing file ownership using **chown:**

The first image shows the permissions for owners/groups/all users on 'file1'. The second image shows the change in permissions for the different users on "file1"

```
$ ls -l
total 4
-rw-rw-r--. 1 bob bob 0 Mar 16 19:04 file-1
-rw-rw-r--. 1 bob bob 0 Mar 16 19:04 file-2
drwxrwxr-x. 2 bob bob 4096 Mar 16 19:04 temp


$ sudo chown root file-1
[sudo] password for bob:


$ ls -l
total 4
-rw-rw-r--. 1 root bob 0 Mar 16 19:04 file-1
-rw-rw-r--. 1 bob bob 0 Mar 16 19:04 file-2
drwxrwxr-x. 2 bob bob 4096 Mar 16 19:04 temp
```

Example of chgrp

Now let's see an example of changing group ownership using **chgrp**:

The image on LHS shows the group with their permissions on 'file1'.

The image on RHS shows the change in groups and thier permissions on "file1"

```
$ sudo chgrp bin file-2
$ ls -l
total 4
-rw-rw-r--. 1 root bob 0 Mar 16 19:04 file-1
-rw-rw-r--. 1 bob bin 0 Mar 16 19:04 file-2
drwxrwxr-x. 2 bob bob 4096 Mar 16 19:04 temp
```

You have completed this chapter. Let's summarize the key concepts covered.

- Linux is a multiuser system.

- To find the currently logged on users, you can use the **who** command.

- To find the current user ID, you can use the **whoami** command.

- The root account has full access to the system. It is never sensible to grant full root access to a user.

- You can assign root privileges to regular user accounts on a temporary basis using the **sudo** command.

- The shell program (**bash**) uses multiple startup files to create the user environment. Each file affects the interactive environment in a different way. /etc/profile provides the global settings.

- Advantages of startup files include that they customize the user's prompt, set the user's terminal type, set the command-line shortcuts and aliases, and set the default text editor, etc.

- An **environment variable** is a character string that contains data used by one or more applications. The built-in shell variables can be customized to suit your requirements.

- The **history** command recalls a list of previous commands which can be edited and recycled.

- In Linux, various keyboard shortcuts can be used at the command prompt instead of long actual commands.

- You can customize commands by creating aliases. Adding an alias to ~/.bashrc will make it available for other shells.

- File permissions can be changed by typing chmod permissions filename.

- File ownership is changed by typing chown owner filename.

- File group ownership is changed by typing chgrp group filename.

Overview of Text Editors in Linux

At some point you will need to manually edit **text files**. You might be composing an email off-line, writing a script to be used for **bash** or other command interpreters, altering a system or application configuration file, or developing source code for a programming language such as **C** or **Java**.
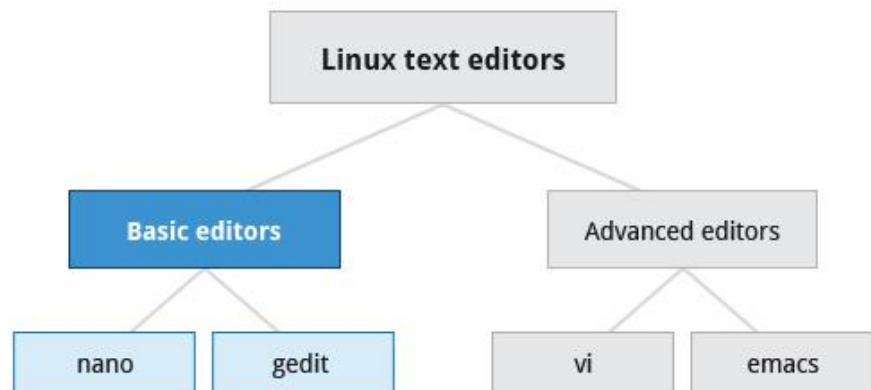
Linux Administrators quite often sidestep the text editors, by using graphical utilities for creating and modifying system configuration files. However, this can be far more laborious than directly using a text editor. Note that word processing applications such as **Notepad** or the applications that are part of office suites are not really basic text editors because they add a lot of extra (usually invisible) formatting information that will probably render system administration configuration files unusable for their intended purpose. So using text editors really is essential in Linux.

By now you have certainly realized Linux is packed with choices; when it comes to text editors, there are many choices ranging from quite simple to very complex, including:
   - **nano**
   - **gedit**
   - **vi**
   - **emacs**

In this section, we will learn about **nano** and **gedit**; editors which are relatively simple and easy to learn. Before we start, let's take a look at some cases where an editor is not needed.

Creating Files Without Using an Editor

Sometimes you may want to create a short file and don't want to bother invoking a full text editor. In addition, doing so can be quite useful when used from within scripts, even when creating longer files. You'll no doubt find yourself using this method when you start on the later chapters that cover **bash** scripting!

If you want to create a file without using an editor there are two standard ways to create one from the command line and fill it with content.

The first is to use **echo** repeatedly:
$ echo line one > myfile
$ echo line two >> myfile
$ echo line three >> myfile

Earlier we learned that a single greater-than sign (>) will send the output of a command to a file. Two greater-than signs (>>) will **append** new output to an existing file.

The second way is to use **cat** combined with redirection:

$ cat << EOF > myfile
> line one
> line two
> line three
> EOF
$

Both the above techniques produce a file with the following lines in it:
line one
line two
line three

and are extremely useful when employed by scripts.

There are some text editors that are pretty obvious; they require no particular experience to learn and are actually quite capable if not robust. One particularly easy one to use is the text-terminal based editor **nano**. Just invoke **nano** by giving a file name as an argument. All the help you need is displayed at the bottom of the screen, and you should be able to proceed without any problem.

As a graphical editor, **gedit** is part of the **GNOME** desktop system (**kwrite** is associated with **KDE)**. The **gedit** and **kwrite** editors are very easy to use and are extremely capable. They are also very configurable. They look a lot like **Notepad** in **Windows**. Other variants such as **kedit** and **kate** are also supported by **KDE**

nano

nano is easy to use, and requires very little effort to learn. To open a file in **nano**, type nano <filename> and press **Enter**. If the file doesn't exist, it will be created.

nano provides a two line "shortcut bar" at the bottom of the screen that lists the available commands. Some of these commands are:

- CTRL-G: Display the help screen

- CTRL-O: Write to a file

- CTRL-X: Exit a file

- CTRL-R: Insert contents from another file to the current buffer

- CTRL-C: Cancels previous commands

gedit

gedit (pronounced 'g-edit') is a simple-to-use graphical editor that can only be run within a Graphical Desktop environment. It is visually quite similar to the **Notepad** text editor in **Windows**, but is actually far more capable and very configurable and has a wealth of plugins available to extend its capabilities further.

To open a new file in **gedit**, find the program in your desktop's menu system, or from the command line type gedit <filename>. If the file doesn't exist it will be created.

Using **gedit** is pretty straight-forward and doesn't require much training. Its interface is composed of quite familiar elements.

vi and emacs

Developers and administrators experienced in working on UNIX-like systems almost always use one of the two venerable editing options; **vi** and **emacs**. Both are present or easily available on all distributions and are completely compatible with the versions available on other operating systems.

Both **vi** and **emacs** have a basic purely text-based form that can run in a non-graphical environment. They also have one or more **X**-based graphical forms with extended capabilities; these may be friendlier for a less experienced user. While **vi** and **emacs** can have significantly steep learning curves for new users, they are extremely efficient when one has learned how to use them.

You need to be aware that fights among seasoned users over which editor is better can be quite intense and are often described as a holy war

Introduction to vi

Usually the actual program installed on your system is **vim** which stands for **vi Improved**, and is aliased to the name **vi**. The name is pronounced as "vee-eye".

Even if you don't want to use **vi,** it is good to gain some familiarity with it: it is a standard tool installed on virtually all Linux distributions. Indeed, there may be times where there is no other editor available on the system.

GNOME extends **vi** with a very graphical interface known as **gvim** and **KDE** offers **kvim**. Either of these may be easier to use at first.

When using **vi**, all commands are entered through the keyboard; you don't need to keep moving your hands to use a pointer device such as a mouse or touchpad, unless you want to do so when using one of the graphical versions of the editor.

vimtutor

Typing vimtutor launches a short but very comprehensive tutorial for those who want to learn their first **vi** commands. This tutorial is a good place to start learning **vi**. Even though it provides only an introduction and just seven lessons, it has enough material to make you a very proficient **vi** user because it covers a large number of commands. After learning these basic ones, you can look up new tricks to incorporate into your list of **vi** commands because there are always more optimal ways to do things in **vi** with less typing.

Modes in **vi**

vi provides three **modes** as described in the table below. It is vital to not lose track of which mode you are in. Many keystrokes and commands behave quite differently in different modes.

Mode	Feature
Command	By default, **vi** starts in Command mode. Each key is an editor command. Keyboard strokes are interpreted as commands that can modify file contents.
Insert	Type i to switch to Insert mode from Command mode. Insert mode is used to enter (insert) text into a file. Insert mode is indicated by an "**? INSERT ?**" indicator at the bottom of the screen. Press Esc to exit Insert mode and return to Command mode.
Line	Type: to switch to the Line mode from Command mode. Each key is an external command, including operations such as writing the file contents to disk or exiting. Uses line editing commands inherited from older line editors. Most of these commands are actually no longer used. Some line editing commands are very powerful. Press Esc to exit Line mode and return to Command mode.

Working with Files in vi

The table describes the most important commands used to start, exit, read, and write files in **vi**. The **ENTER** key needs to be pressed after all of these commands.

Command	Usage
vi myfile	Start the **vi** editor and edit the **myfile** file
vi -r myfile	Start **vi** and edit **myfile** in recovery mode from a system crash
:r file2	Read in **file2** and insert at current position
:w	Write to the file
:w myfile	Write out the file to **myfile**
:w! file2	Overwrite **file2**
:x or :wq	Exit **vi** and write out modified file
:q	Quit **vi**

:q!	Quit **vi** even though modifications have not been saved

Changing Cursor Positions in vi

The table describes the most important keystrokes used when changing cursor position in **vi.** Line mode commands (those following colon (:)) require the **ENTER** key to be pressed after the command is typed.

Key	Usage
arrow keys	To move up, down, left and right
j or <ret>	To move one line down
k	To move one line up
h or Backspace	To move one character left
l or Space	To move one character right
0	To move to beginning of line
$	To move to end of line
w	To move to beginning of next word
:0 or 1G	To move to beginning of file

:n or nG	To move to line n
:$ or G	To move to last line in file
CTRL-F or Page Down	To move forward one page
CTRL-B or Page Up	To move backward one page
^l	To refresh and center screen

Searching for Text in vi

The table describes the most important commands used when searching for text in **vi**. The **ENTER** key should be pressed after typing the search pattern.

Command	Usage
/pattern	Search forward for pattern
?pattern	Search backward for pattern

The table describes the most important keystrokes used when searching for text in **vi**.

Key	Usage
n	Move to next occurrence of search pattern
N	Move to previous occurrence of search pattern

Working with Text in vi

The table describes the most important keystrokes used when changing, adding, and deleting text in **vi.**

Key	Usage
A	Append text after cursor; stop upon Escape key
A	Append text at end of current line; stop upon Escape key
i	Insert text before cursor; stop upon Escape key
I	Insert text at beginning of current line; stop upon Escape key
o	Start a new line below current line, insert text there; stop upon Escape key
O	Start a new line above current line, insert text there; stop upon Escape key
r	Replace character at current position

R	Replace text starting with current position; stop upon Escape key
x	Delete character at current position
Nx	Delete N characters, starting at current position
dw	Delete the word at the current position
D	Delete the rest of the current line
dd	Delete the current line
Ndd or dNd	Delete N lines
u	Undo the previous operation
yy	Yank (copy) the current line and put it in buffer
Nyy or yNy	Yank (copy) N lines and put it in buffer
p	Paste at the current position the yanked line or lines from the buffer.

Using External Commands

Typing :sh command opens an external command shell. When you exit the shell, you will resume your **vi** editing session.

Typing :!executes a command from within **vi**. The command follows the exclamation point. This technique best suited for non-interactive commands such as:
:! wc %

Typing this will run the wc (word count) command on the file; the character % represents the file currently being edited.

The **fmt** command does simple formatting of text. If you are editing a file and want the file to look nice, you can run the file through **fmt**. One way to do this while editing is by using :%!fmt, which runs the entire file (the % part) through **fmt** and replaces the file with the results.

Introduction to emacs

The **emacs** editor is a popular competitor for **vi**. Unlike **vi**, it does not work with modes. **emacs** is highly customizable and includes a large number of features. It was initially designed for use on a console, but was soon adapted to work with a GUI as well. **emacs** has many other capabilities other than simple text editing; it can be used for email, debugging, etc.

Rather than having different modes for command and insert, like **vi**, **emacs** uses the **CTRL** and **Esc** keys for special commands

Working with emacs

The table lists some of the most important key combinations that are used when starting, exiting, reading, and writing files in **emacs**.

Key	Usage
emacs myfile	Start emacs and edit myfile
CTRL-x i	Insert prompted for file at current position
CTRL-x s	Save all files
CTRL-x CTRL-w	Write to the file giving a new name when prompted
CTRL-x CTRL-s	Saves the current file
CTRL-x CTRL-c	Exit after being prompted to save any modified files

The **emacs** tutorial is a good place to start learning basic **emacs** commands. It is available any time when in **emacs** by simply typing CTRL-h (for help) and then the letter t for tutorial.

Changing Cursor Positions in emacs

The table lists some of the keys and key combinations that are used for changing cursor positions in emacs.

Key	Usage
arrow keys	Use the arrow keys for up, down, left and right
CTRL-n	One line down
CTRL-p	One line up
CTRL-f	One character forward/right
CTRL-b	One character back/left
CTRL-a	Move to beginning of line
CTRL-e	Move to end of line
Esc-f	Move to beginning of next word
Esc-b	Move back to beginning of preceding word
Esc-<	Move to beginning of file
Esc-x	Goto-line n move to line n

Key	Usage
Esc->	Move to end of file
CTRL-v or Page Down	Move forward one page
Esc-v or Page Up	Move backward one page
CTRL-l	Refresh and center screen

Searching for Text in emacs

The table lists the key combinations that are used for searching for text in **emacs**.

Key	Usage
CTRL-s	Search forward for prompted pattern, or for next pattern
CTRL-r	Search backwards for prompted pattern, or for next pattern

Working with Text in emacs

The table lists some of the key combinations used for changing, adding, and deleting text in **emacs**:

Key	Usage
CTRL-o	Insert a blank line
CTRL-d	Delete character at current position

CTRL-k	Delete the rest of the current line
CTRL-_	Undo the previous operation
CTRL- (space or CTRL-@)	Mark the beginning of the selected region. The end will be at the cursor position
CTRL-w	Delete the current marked text and write it to the buffer
CTRL-y	Insert at current cursor location whatever was most recently deleted

You have completed this chapter. Let's summarize the key concepts covered.

- **Text editors** (rather than word processing programs) are used quite often in Linux, for tasks such as for creating or modifying system configuration files, writing scripts, developing source code, etc.

- **nano** is an easy-to-use text-based editor that utilizes on-screen prompts.

- **gedit** is a graphical editor very similar to **Notepad** in **Windows**.

- The **vi** editor is available on all Linux systems and is very widely used. Graphical extension versions of **vi** are widely available as well.

- **emacs** is available on all Linux systems as a popular alternative to **vi**. **emacs** can support both a graphical user interface and a text mode interface.

- To access the **vi** tutorial, type **vimtutor** at a command line window.

- To access the **emacs** tutorial type **Ctl-h** and then **t** from within **emacs.**

- **vi** has three modes: **Command**, **Insert**, and **Line; emacs** has only one but requires use of special keys such as Control and Escape.

- Both editors use various combinations of keystrokes to accomplish tasks; the learning curve to master these can be long but once mastered using either editor is extremely efficient.

User Accounts

The Linux kernel allows properly authenticated users to access files and applications. While each user is identified by a unique integer (the user id or **UID),** a separate database associates a **username** with each UID. Upon account creation, new user information is added to the user database and the user's home directory must be created and populated with some essential files. Command line programs such as **useradd** and **userdel** as well as GUI tools are used for creating and removing accounts.

For each user, the following seven fields are maintained in the /etc/passwd file:

Field Name	Details	Remarks
Username	User login name	Should be between 1 and 32 characters long
Password	User password (or the character **x** if the password is stored in the /etc/shadow file) in encrypted format	Is never shown in Linux when it is being typed; this stops prying eyes
User ID (UID)	Every user must have a user id (UID)	• UID 0 is reserved for root user • UID's ranging from 1-99 are reserved for other predefined accounts • UID's ranging from 100-999 are reserved for system accounts and groups (except for RHEL, which

		reserves only up to 499) • Normal users have UID's of 1000 or greater, except on RHEL where they start at 500
Group ID (GID)	The primary Group ID (GID); Group Identification Number stored in the /etc/group file	Will be covered in detail in the chapter on Processes
User Info	This field is optional and allows insertion of extra information about the user such as their name	For example: Rufus T. Firefly
Home Directory	The absolute path location of user's home directory	For example: /home/rtfirefly
Shell	The absolute location of a user's default shell	For example: /bin/bash

Types of Accounts

By default, Linux distinguishes between several account types in order to isolate processes and workloads. Linux has four types of accounts:

• root

• System

• Normal

• Network

For a safe working environment, it is advised to grant the minimum privileges possible and necessary to accounts, and remove inactive accounts. The **last** utility, which shows the last time each user logged into the system, can be used to help identify potentially inactive accounts which are candidates for system removal.

Keep in mind that practices you use on multi-user business systems are more strict than practices you can use on personal desktop systems that only affect the casual user. This is especially true with security. We hope to show you practices applicable to enterprise servers that you can use on all systems, but understand that you may choose to relax these rules on your own personal system.

Understanding the root Account

root is the most privileged account on a Linux/UNIX system. This account has the ability to carry out all facets of system administration, including adding accounts, changing user passwords, examining log files, installing software, etc. Utmost care must be taken when using this account. It has no security restrictions imposed upon it.

When you are signed in as, or acting as **root**, the shell prompt displays **'#'** (if you are using **bash** and you haven't customized the prompt as we discuss elsewhere in this course). This convention is intended to serve as a warning to you of the absolute power of this account

Operations that Require root Privileges

root privileges are required to perform operations such as:

- Creating, removing and managing user accounts.

- Managing software packages.

- Removing or modifying system files.

- Restarting system services.
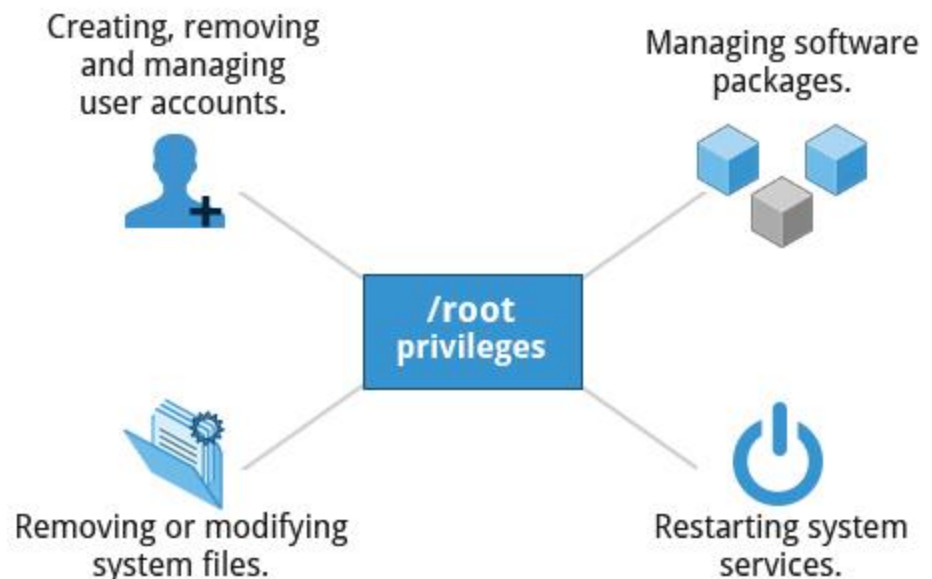
Regular account users of Linux distributions may be allowed to install software packages, update some settings, and apply various kinds of changes to the system. However, **root** privilege is required for performing administration tasks such as restarting services, manually installing packages and managing parts of the file system that are outside the normal user's directories.

Creating, removing and managing user accounts.

Managing software packages.

/root privileges

Removing or modifying system files.

Restarting system services.

Creating a New User in Linux

To create a new user account:

1. At the command prompt, as root type useradd <username> and press the **ENTER** key.

2. To set the initial password, type passwd <username> and press the **ENTER** key. The **New password**: prompt is displayed.

3. Enter the password and press the **ENTER** key.
   To confirm the password, the prompt **Retype new password**: is displayed.

4. Enter the password again and press the **ENTER** key.
   The message **passwd: all authentication tokens updated successfully**. is displayed.

Operations That Do Not Require root Privileges

A regular account user can perform some operations requiring special permissions; however, the system configuration must allow such abilities to be exercised.

SUID (Set owner User ID upon execution—similar to the Windows "run as" feature) is a special kind of file permission given to a file. SUID provides temporary permissions to a user to run a program with the permissions of the file **owner** (which may be root) instead of the permissions held by the user.

The table provides examples of operations which do not require root privileges:

Operations that do not require Root privilege	Examples of this operation
Running a network client	Sharing a file over the network
Using devices such as printers	Printing over the network
Operations on files that the user has proper permissions to access	Accessing files that you have access to or sharing data over the network
Running SUID-root applications	Executing programs such as passwd.

Comparing sudo and su

In Linux you can use either **su** or **sudo** to temporarily grant root access to a normal user; these methods are actually quite different. Listed below are the differences between the two commands.

| su	sudo
When elevating privilege, you need to enter the **root** password. Giving the root password to a normal user should **never**, **ever** be	When elevating privilege, you need to enter the user's password and not

done.	the **root** password.
Once a user elevates to the **root** account using **su,** the user can do **anything** that the **root** user can do for as long as the user wants, without being asked again for a password.	Offers more features and is considered more secure and more configurable. Exactly what the user is allowed to do can be precisely controlled and limited. By default the user will either always have to keep giving their password to do further operations with **sudo**, or can avoid doing so for a configurable time interval.
The command has limited logging features.	The command has detailed logging features.

sudo Features

sudo has the ability to keep track of unsuccessful attempts at gaining root access. Users' authorization for using **sudo** is based on configuration information stored in the /etc/sudoers file and in the /etc/sudoers.ddirectory.

A message such as the following would appear in a system log file (usually /var/log/secure) when trying to execute **sudo bash** without successfully authenticating the user:

authentication failure; logname=op uid=0 euid=0
tty=/dev/pts/6 ruser=op rhost= user=op
conversation failed
auth could not identify password for [op]
op : 1 incorrect password attempt ;
TTY=pts/6 ; PWD=/var/log ; USER=root ; COMMAND=/bin/bash
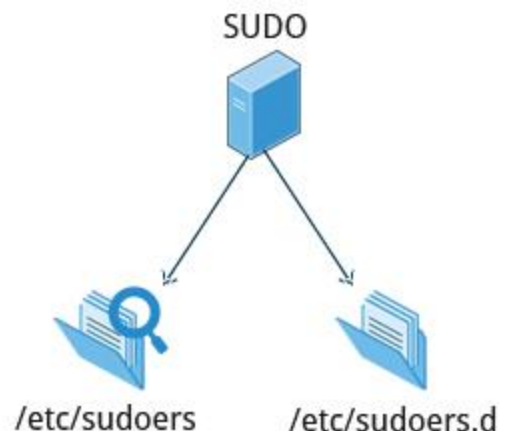
The sudoers File

Whenever **sudo** is invoked, a trigger will look at**/etc/sudoers** and the files in **/etc/sudoers.d** to determine if the user has the right to use **sudo** and what the scope of their privilege is. Unknown user requests and requests to do operations not allowed to the user even with **sudo** are reported. You can edit the **sudoers** file by using **visudo**, which ensures that only one person is editing the file at a time, has the proper permissions, and refuses to write out the file and exit if there is an error in the changes made.

The basic structure of an entry is:
who where = (as_whom) what

The file has a lot of documentation in it about how to customize. Most Linux distributions now prefer you add a file in the directory **/etc/sudoers.d** with a name the same as the user. This file contains the individual user's **sudo** configuration, and one should leave the master configuration file untouched except for changes that affect all users.

Command Logging

By default, **sudo** commands and any failures are logged in /var/log/auth.log under the **Debian** distribution family, and in /var/log/messages or /var/log/secure on other systems. This is an important safeguard to allow for tracking and accountability of **sudo** use. A typical entry of the message contains:

- Calling username

- Terminal info

- Working directory

- User account invoked

- Command with arguments

Running a command such as sudo whoami results in a log file entry such as:
Dec 8 14:20:47 server1 sudo: op : TTY=pts/6
PWD=/var/log USER=root
COMMAND=/usr/bin/whoami

Process Isolation

Linux is considered to be more secure than many other operating systems because processes are naturally **isolated** from each other. One process normally cannot access the resources of another process, even when that process is running with the same user privileges. Linux thus makes it difficult (though certainly not impossible) for viruses and security exploits to access and attack random resources on a system.

Additional security mechanisms that have been recently introduced in order to make risks even smaller are:

- **Control Groups (cgroups)**: Allows system administrators to group processes and associate finite resources to each cgroup.

- **Linux Containers (LXC)**: Makes it possible to run multiple isolated Linux systems (containers) on a single system by relying on **cgroups**.

- **Virtualization**: Hardware is emulated in such a way that not only processes can be isolated, but entire systems are run simultaneously as isolated and insulated guests (virtual machines) on one physical host.
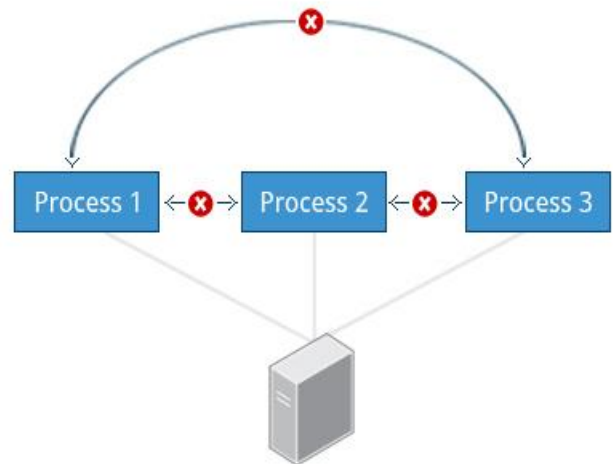
Hardware Device Access

Linux limits user access to non-networking hardware devices in a manner that is extremely similar to regular file access. Applications interact by engaging the file system layer (which is independent of the actual device or hardware the file resides on). This layer will then opens a **device special file** (often called a **device node**) under the **/dev** directory that corresponds to the device being accessed. Each device special file has standard owner, group and world permission fields. Security is naturally enforced just as it is when standard files are accessed.

Hard disks, for example, are represented as **/dev/sd***. While a root user can read and write to the disk in a **raw** fashion (for example, by doing something like:

```
$ echo hello world > /dev/sda1
```

the standard permissions as shown in the figure make it impossible for regular users to do so. Writing to a device in this fashion can easily obliterate the file system stored on it in a way that cannot be repaired without great effort, if at all. The normal reading and writing of files on the hard disk by applications is done at a higher level through the file system, and never through direct access to the device node.

Keeping Current

When security problems in either the Linux kernel or applications and libraries are discovered, Linux distributions have a good record of reacting quickly and pushing out fixes to all systems by updating their software repositories and sending notifications to update immediately. The same thing is true with bug fixes and performance improvements that are not security related.

However, it is well known that many systems do not get updated frequently enough and problems which have already been cured are allowed to remain on computers for a long time; this is particularly true with proprietary operating systems where users are either uninformed or distrustful of the vendor's patching policy as sometimes updates can cause new problems and break existing operations. Many of the most successful attack vectors come from exploiting security holes for which fixes are already known but not universally deployed.

Timely System Update

So the best practice is to take advantage of your Linux distribution's mechanism for automatic updates and never postpone them. It is extremely rare that such an update will cause new problems.

How Passwords are Stored

The system verifies authenticity and identity using user credentials. Originally, encrypted passwords were stored in the /etc/passwd file, which was readable by everyone. This made it rather easy for passwords to be cracked. On modern systems, passwords are actually stored in an encrypted format in a secondary file named **/etc/shadow**. Only those with **root access** can modify/read this file.

Password Encryption

Protecting passwords has become a crucial element of security. Most Linux distributions rely on a modern password encryption algorithm called **SHA-512** (Secure Hashing Algorithm 512 bits), developed by the U.S. National Security Agency (NSA) to encrypt passwords.

The **SHA-512** algorithm is widely used for security applications and protocols. These security applications and protocols include TLS, SSL, PHP, SSH, S/MIME and IPSec. **SHA-512** is one of the most tested hashing algorithms.

For example, if you wish to experiment with **SHA-512** encoding, the word "test" can be encoded using the program**sha512sum** to produce the **SHA-512** form

IT professionals follow several good practices for securing the data and the password of every user.

Good Password Practices

1. **Password aging** is a method to ensure that users get prompts that remind them to create a new password after a specific period. This can ensure that passwords, if cracked, will only be usable for a limited amount of time. This feature is implemented using **chage,** which configures the password expiry information for a user.

2. Another method is to force users to set strong passwords using **Pluggable Authentication Modules (PAM)**. **PAM** can be configured to automatically verify that a password created or modified using the **passwd** utility is sufficiently strong. **PAM** configuration is implemented using a library called **pam_cracklib.so**, which can also be replaced by**pam_passwdqc.so** for more options.

3. One can also install password cracking programs, such as **Jack The Ripper**, to secure the password file and detect weak password entries. It is recommended that written authorization be obtained before installing such tools on any system that you do not own.
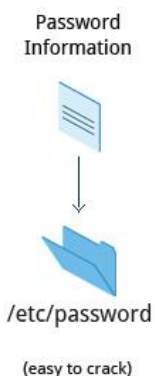
You can secure the boot process with a secure password to prevent someone from bypassing the user authentication step. For systems using the **GRUB** boot loader, for the older**GRUB** version 1, you can invoke **grub-md5-crypt** which will prompt you for a password and then encrypt as shown on the adjoining screen.

You then must edit /boot/grub/grub.conf by adding the following line below the timeout entry:

password --md5 1Wnvo.1$qz781HRVG4jUnJXmdSCZ30

You can also force passwords for only certain boot choices rather than all.

For the now more common **GRUB** version 2 things are more complicated, and you have more flexibility and can do things like use user-specific passwords, which can be their normal login password. Also you never edit the configuration file, /boot/grub/grub.cfg, directly, rather you edit system configuration files in /etc/grub.d and then run **update-grub**. One explanation of this can be found at https://help.ubuntu.com/community/Grub2/Passwords.

Older system

Password Information

/etc/password

(easy to crack)

Modern system

Password information in file accessible only by root

/etc/shadow

root user

Hardware Vulnerability

When hardware is physically accessible,
security can be compromised by:

- Key logging: Recording the real time
  activity of a computer user including the
  keys they press. The captured data can
  either be stored locally or transmitted to
  remote machines

- Network sniffing: Capturing and viewing the network packet level data on your network

- Booting with a live or rescue disk

- Remounting and modifying disk content

Your IT security policy should start with requirements on how to properly secure physical access to
servers and workstations. Physical access to a system makes it possible for attackers to easily leverage
several attack vectors, in a way that makes all operating system level recommendations irrelevant.

The guidelines of security are:

- Lock down workstations and servers

- Protect your network links such that it cannot be accessed by people you do not trust

- Protect your keyboards where passwords are entered to ensure the keyboards cannot be tampered
  with

- Ensure a password protects the BIOS in such a way that the system cannot be booted with a live or
  rescue DVD or USB key

For single user computers and those in a home environment some of the above features (like preventing
booting from removable media) can be excessive, and you can avoid implementing them. However, if
sensitive information is on your system that requires careful protection, either it shouldn't be there or it
should be better protected by following the above guidelines.

Introduction to Networking

A network is a group of computers and computing devices
connected together through communication channels, such
as cables or wireless media. The computers connected over
a network may be located in the same geographical area or
spread across the world.

A network is used to:

- Allow the connected devices to communicate with each
  other.

- Enable multiple users to share devices over the network,

such as printers and scanners.

- Share and manage information across computers easily.

Most organizations have both an internal network and an Internet connection for users to communicate with machines and people outside the organization. The Internet is the largest network in the world and is often called "the network of networks".

IP Addresses

Devices attached to a network must have at least one unique network address identifier known as the IP (**Internet Protocol)** address. The address is essential for routing **packets** of information through the network.

Exchanging information across the network requires using streams of bite-sized packets, each of which contains a piece of the information going from one machine to another. These packets contain **data buffers** together
with **headers** which contain information about where the packet is going to and coming from, and where it fits in the sequence of packets that constitute the stream. Networking protocols and software are rather complicated due to the diversity of machines and operating systems they must deal with, as well as the fact that even very old standards must be supported.

172.20.161.0   157.28.162.2
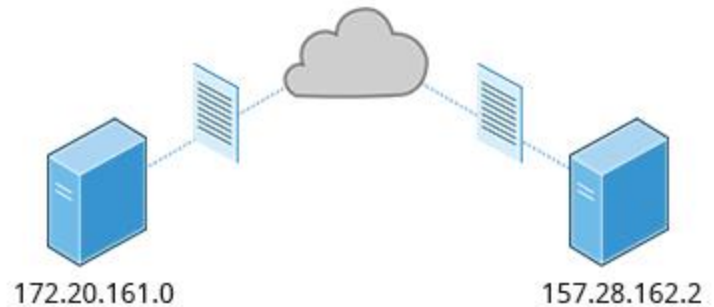
IPv4 and IPv6

There are two different types of IP addresses available: **IPv4** (version 4) and **IPv6** (version 6). **IPv4** is older and by far the more widely used, while **IPv6** is newer and is designed to get past the limitations of the older standard and furnish many more possible addresses.

IPv4
Uses 32-bits
Only 4.3 billion unique addresses.

IP Addresses

IPv6
Uses 128-bits
allows 3.4 X 10^{38} unique addresses.

IPv4 uses 32-bits for addresses; there
are **only** 4.3 billion unique addresses available. Furthermore, many addresses are allotted and reserved but not actually used. **IPv4** is becoming inadequate because the number of devices available on the global network has significantly increased over the past years.

IPv6 uses 128-bits for addresses; this allows for 3.4 X 10^{38} unique addresses. If you have a larger network of computers and want to add more, you may want to move to **IPv6**, because it provides more unique addresses. However, it is difficult to move to **IPv6** as the two protocols do not inter-operate. Due to this, migrating equipment and addresses to **IPv6** requires significant effort and hasn't been as fast as was originally intended.

Decoding IPv4 Addresses

A 32-bit IPv4 address is divided into four 8-bit sections called octets.

Example:
IP address
→ 172 . 16 . 31 . 46
Bit format
→ 10101100.00010000.00011111.00101110

	Octet 1	Octet 2	Octet 3	Octet 4
Class A	Network ID	Host ID	Host ID	Host ID
Class B	Network ID	Network ID	Host ID	Host ID
Class C	Network ID	Network ID	Network ID	Host ID
Class D	Multicast addresses			
Class E	Reserved for future use			

Network address are divided into five classes: A, B, C, D, and E. Classes A, B, and C are classified into two parts: **Network addresses (Net ID)** and **Host address (Host ID)**. The Net ID is used to id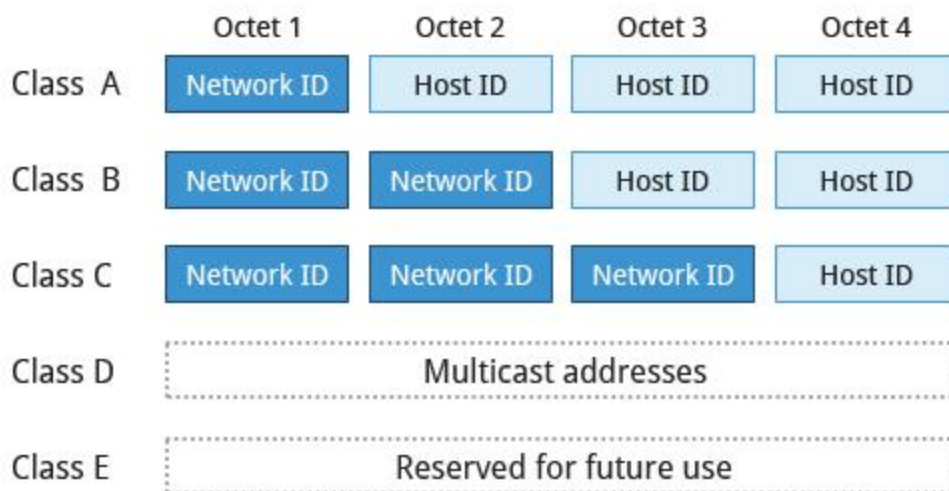entify the network, while the Host ID is used to identify a host in the network. Class D is used for special multicast applications (information is broadcast to multiple computers simultaneously) and Class E is reserved for future use. In this section you will learn about classes A, B, and C.
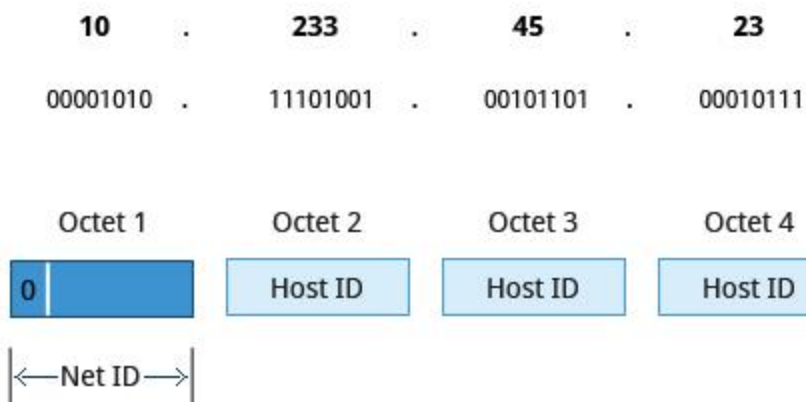
Class A Network Addresses

Class A addresses use the first octet of an IP address as their Net ID and use the other three octets as the **Host ID.** The first bit of the first octet is always set to zero. So you can use only 7-bits for unique network numbers. As a result, there are a maximum of 127 Class A networks available. Not surprisingly, this was only feasible when there were very few unique networks with large numbers of hosts. As the use of the Internet expanded, Classes B and C were added in order to accomodate the growing demand for independent networks.

An example of a Class A address is:

10	.	233	.	45	.	23
00001010	.	11101001	.	00101101	.	00010111

Octet 1	Octet 2	Octet 3	Octet 4
0	Host ID	Host ID	Host ID

|←—Net ID—→|

Each Class A network can have up to 16.7 million unique hosts on its network. The range of host address is from 1.0.0.0 to 127.255.255.255.

Note: The value of an octet, or 8-bits, can range from 0 to 255.

Class B Network Addresses

Class B addresses use the first two octets of the IP address as their Net ID and the last two octets as the Host ID. The first two bits of the first octet are always set to binary 10, so there are a maximum of 16,384 (14-bits) Class B networks. The first octet of a Class B address has values from 128 to 191. The introduction of Class B networks expanded the number of networks but it soon became clear that a further level would be needed.

An example of a Class B address is:

135	.	201	.	18	.	1
10000111 .		11001001 .		00010010 .		00000001

Octet 1	Octet 2	Octet 3	Octet 4
1 0		Host ID	Host ID

←——— Net ID ———→

Each Class B network can support a maximum of 65,536 unique hosts on its network. The range of host address is from 128.0.0.0 to 191.255.255.255.
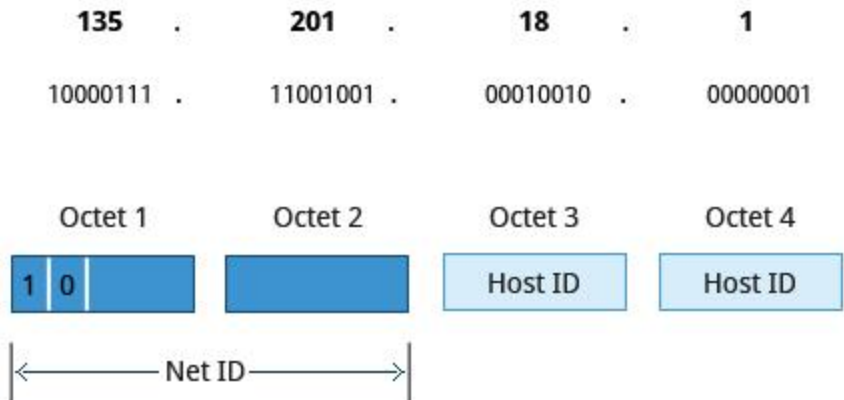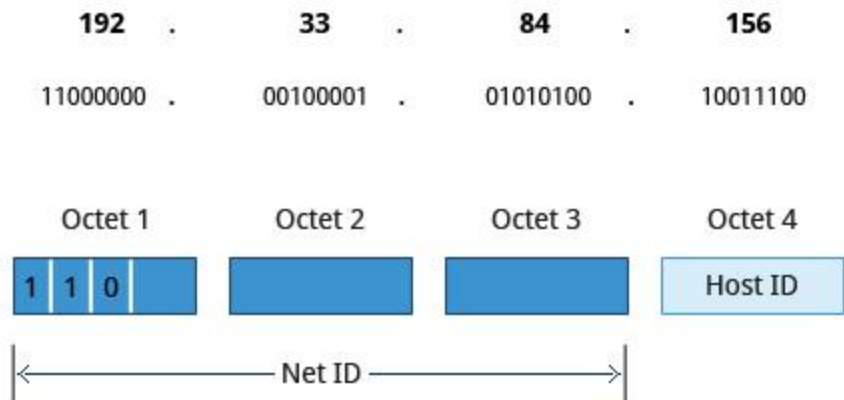
Class C Network Addresses

Class C addresses use the first three octets of the IP address as their Net ID and the last octet as their Host ID. The first three bits of the first octet are set to binary 110, so almost 2.1 million (21-bits) Class C networks are available. The first octet of a Class C address has values from 192 to 223. These are most common for smaller networks which don't have many unique hosts.

An example of a Class C address is:

192	.	33	.	84	.	156
11000000 .		00100001 .		01010100 .		10011100

Octet 1	Octet 2	Octet 3	Octet 4
1 1 0			Host ID

←——————— Net ID ———————→

Each Class C network can support up to 256 (8-bits) unique hosts. The range of host address is from 192.0.0.0 to 223.255.255.255.

IP Address Allocation
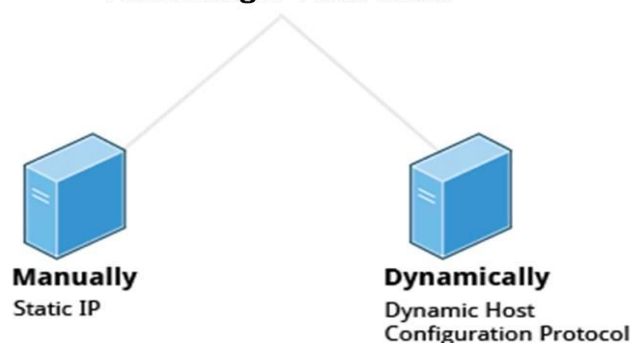
Typically, a range of IP addresses are requested from your Internet Service Provider (ISP) by your organization's network administrator. Often your choice of which class of IP address you are given depends on the size of your network and expected growth needs.

Allocating IP Addresses

Manually
Static IP

Dynamically
Dynamic Host
Configuration Protocol

You can assign IP addresses to computers over a network manually or dynamically. When you assign IP addresses manually, you add **static** (never changing) addresses to the network. When you assign IP addresses dynamically (they can change every time you reboot or even more often), the **Dynamic Host Configuration Protocol (DHCP)** is used to assign IP addresses.

Manually Allocating an IP Address

Before an IP address can be allocated manually, one must identify the size of the network by determining the host range; this determines which network class (A, B, or C) can be used. The **ipcalc** program c an be used to as certain the host range.

Note: The version of ipcalc supplied in the Fedora family of distributions does not behave as described below, it is really a different program.

Assume that you have a Class C network. The first three octets of the IP address are 192.168.0. As it uses 3 octets (i.e. 24 bits) for the network mask, the shorthand for this type of address is 192.168.0.0/24. To determine the host range of the address you can use for this new host, at the command prompt, type: ipcalc 192.168.0.0/24 and press **Enter**.

From the result, you can check the **HostMin** and **HostMax** values to manually assign a static address available from 1 to 254 (192.168.0.1 to 192.168.0.254).

Name Resolution

Name Resolution is used to convert numerical IP address values into a human- readable format known as the **hostname**. For example, 172.217.27.206 is the numeric al IP address that refers to the **google.com** hostname. Hostnames are easier to remember.

Given an IP address, you can obtain its corresponding hostname. Accessing the machine over the network becomes easier when you can type the hostname instead of the IP address.

You can view your system's hostname simply by typing **hostname** with no argument.

Note: If you give an argument, the system will try to change its hostname to match it, however, only root users can do that.

The special hostname **localhost** is associated with the IP address 127.0.0.1**,** and describes the machine you are currently on (which normally has additional net work-related IP addresses).

Note: The next two screens cover the demonstration and Try-It-Yourself activity. You can view a demonstration and practice the procedure through the Try -It-Yourself activity.

Network Interfaces

Network interfaces are a connection channel between a device and a network. Physically, network interfaces can proceed through a **network interface card** (**NIC**) or can be more abstractly implemented as software. You can have multiple network interfaces operating at once. Specific interfaces can be brought up (activated) or brought down (de-activated) at any time.

A list of currently active network interfaces is reported by the **ifconfig** utility which you may have to run as the superuser, or at least, give the full path, i.e., /sbin/ifconfig, on some distributions.

Network Configuration Files

Network configuration files are essential to ensure that interfaces function correctly.

For **Debian** family configuration, the basic network configuration file is /etc/network/interfaces. You can type /etc/init.d/networking start to start the networking configuration.

For **Fedora** family system configuration, the routing and host information is contained in /etc/sysconfig/network. The network interface configuration script is located at /etc/sysconfig/network-scripts/ifcfg-eth0.

For **SUSE** family system configuration, the routing and host information and network interface configuration scripts are contained in the /etc/sysconfig/network directory.

You can type /etc/init.d/network start to start the networking configuration for **Fedora** and **SUSE** families.

Network Configuration Commands

To view the IP address:
$ /sbin/ip addr show

To view the routing information:
$ /sbin/ip route show

ip is a very powerful program that can do many things. Older (and more specific) utilities such as **ifconfig** and **route** are often used to accomplish similar tasks. A look at the relevant **man pages** can tell you much more about these utilities.

ping

ping is used to check whether or not a machine attached to the network can receive and send data; i.e., it confirms that the remote host is online and is responding.

To check the status of the remote host, at the command prompt, type ping <hostname>.

ping is frequently used for network testing and management; however, its usage can increase network load unacceptably. Hence, you can abort the execution of **ping** by typing **CTRL-C**, or by using the **-c** option, which limits the number of packets that **ping** will send before it quits. When execution stops, a summary is displayed.

route

A network requires the connection of many nodes. Data moves from source to destination by passing through a series of routers and potentially across multiple networks. Servers maintain **routing tables** containing the addresses of each node in the network. The **IP Routing protocols** enable routers to build up a forwarding table that correlates final destinations with the next **hop** addresses.

route is used to view or change the IP routing table. You may want to change the IP routing table to add, delete or modify specific (static) routes to specific hosts or networks. The table explains some commands that can be used to manage IP routing.

Task	Command
Show current routing table	$ route –n
Add static route	$ route add -net address
Delete static route	$ route del -net address

traceroute

traceroute is used to inspect the route which the data packet takes to reach the destination host which makes it quite useful for troubleshooting network delays and errors. By using **traceroute** you can isolate connectivity issues between **hops**, which helps resolve them faster.

To print the route taken by the packet to reach the network host, at the command prompt, type traceroute <domain>.

Networking Tools

Now, let's learn about some additional networking tools. Networking tools are very useful for monitoring and debugging network problems, such as network connectivity and network traffic.

Networking Tools	Description
Ethtool	Queries network interfaces and can also set various parameters such as the speed.
Netstat	Displays all active connections and routing tables. Useful for monitoring performance and troubleshooting.

Nmap	Scans open **ports** on a network. Important for security analysis
Tcpdump	Dumps network traffic for analysis.
Iptraf	Monitors network traffic in text mode.

Graphical and Non-Graphical Browsers

Browsers are used to retrieve, transmit, and explore information resources, usually on the **World Wide Web**. Linux users commonly use both graphical and non-graphical browser applications.

The common graphical browsers used in Linux are:

- **Firefox**

- **Google Chrome**

- **Chromium**

- **Epiphany**

- **Opera**

Sometimes you either do not have a graphical environment to work in (or have reasons not to use it) but still need to access web resources. In such a case, you can use non-graphical browsers such as the following:

Non-Graphical Browsers	Description
lynx	Configurable text-based web browser; the earliest such browser and still in use.
links or elinks	Based on **lynx.** It can display tables and frames.

w3m

Newer text-based web browser with many features.

wget

Sometimes you need to download files and information but a browser is not the best choice, either because you want to download multiple files and/or directories, or you want to perform the action from a command line or a script. **wget** is a command line utility that can capably handle the following types of downloads:

- Large file downloads

- Recursive downloads, where a web page refers to other web pages and all are downloaded at once

- Password-required downloads

- Multiple file downloads

To download a webpage, you can simply type wget <url>, and then you can read the downloaded page as a local file using a graphical or non-graphical browser.
Besides downloading you may want to obtain information about a URL, such as the source code being used. **curl** can be used from the command line or a script to read such information. **curl** also allows you to save the contents of a web page to a file as does **wget**.

curl

You can read a URL using curl <URL>. For example, if you want to read http://www.google.com , type. curl www.google.co.in

To get the contents of a web page and store it to a file, type
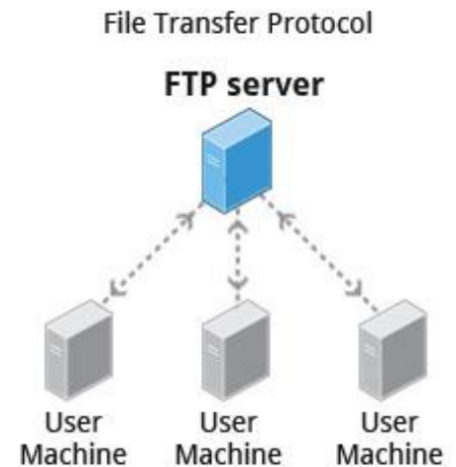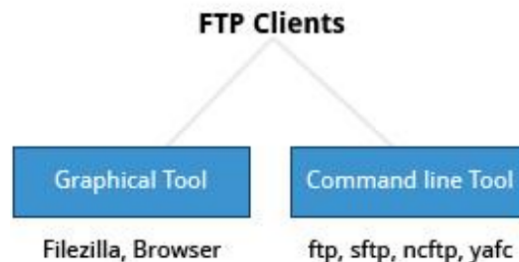curl -o saved.html www.google.com
The contents of the main index file at the website will be saved in **saved.html**

FTP (File Transfer Protocol)

When you are connected to a network, you may need to transfer files from one machine to another. **File Transfer Protocol (FTP)** is a well-known and popular method for transferring files between computers using the Internet. This method is built on a **client-server** model. FTP can be used within a browser or with standalone client programs.

FTP Clients

FTP clients enable you to transfer files with remote computers using the FTP protocol. These clients can be either graphical

File Transfer Protocol

FTP server

User Machine   User Machine   User Machine

FTP Clients

Graphical Tool   Command line Tool

Filezilla, Browser   ftp, sftp, ncftp, yafc

or command line tools. **Filezilla**, for example, allows use of the drag-and-drop approach to transfer files between hosts. All web browsers support FTP, all you have to do is give a URL like
: ftp://ftp.kernel.org where the usual http:// becomes ftp://.

Some command line FTP clients are:

- **ftp**

- **sftp**

- **ncftp**

- **yafc** (Yet Another FTP Client)

sftp is a very secure mode of connection, which uses the **Secure Shell** (**ssh)** protocol, which we will discuss shortly. **sftp**encrypts its data and thus sensitive information is transmitted more securely. However, it does not work with so-called **anonymous FTP** (guest user credentials).
Both **ncftp** and **yafc** are also powerful FTP clients which work on a wide variety of operating systems including **Windows** and **Linux**.

SSH: Executing Commands Remotely

Secure Shell (SSH) is a cryptographic network protocol used for secure data communication. It is also used for remote services and other secure services between two devices on the network and is very useful for administering systems which are not easily available to physically work on but to which you have remote access.

To run my_command on a remote system via SSH, at the command prompt, type, ssh <remotesystem> my_command and press **Enter**. **ssh** then prompts you for the remote password. You can also configure **ssh** to securely allow your remote access without typing a password each time.

Copying Files Securely with scp

```
scp <localfile> <user@remotesystem>:/home/user/
```

We can also move files securely using **Secure Copy** (**scp**) between two networked hosts. **scp** uses the SSH protocol for transferring data.

To copy a local file to a remote system, at the command prompt, type scp <localfile> <user@remotesystem>:/home/user/ and press **Enter**.

You will receive a prompt for the remote password. You can also configure **scp** so that it does not prompt for a password for each transfer.

You have completed this chapter. Let's summarize the key concepts covered:

- The **IP** (Internet Protocol) **address** is a unique logical network address that is assigned to a device on a network.

- **IPv4** uses 32-bits for addresses and **IPv6** uses 128-bits for addresses.

- Every IP address contains both a network and a host address field.

- There are five classes of network addresses available: A, B, C, D & E.

- **DNS** (Domain Name System) is used for converting Internet domain and host names to IP addresses.

- The **ifconfig** program is used to display current active network interfaces.

- The commands ip addr show and ip route show can be used to view IP address and routing information.

- You can use **ping** to check if the remote host is alive and responding.

  - You can use the **route** utility program to manage IP routing.

  - You can monitor and debug network problems using networking tools.

  - **Firefox**, **Google Chrome**, **Chromium**, and **Epiphany** are the main graphical browsers used in Linux.

  - Non-graphical or text browsers used in Linux are **Lynx**, **Links**, and **w3m**.

  - You can use **wget** to download webpages.

  - You can use **curl** to obtain information about URL's.

  - **FTP** (File Transfer Protocol) is used to transfer files over a network.

- **ftp**, **sftp**, **ncftp**, and **yafc** are command line FTP clients used in Linux.

- You can use **ssh** to run commands on remote systems.

Command Line Tools

Irrespective of the role you play with Linux (system administrator, developer, or user) you often need to browse through and parse text files, and/or extract data from them. These are **file manipulation**operations. Thus it is essential for the Linux user to become adept at performing certain operations on files.

Most of the time such file manipulation is done at the **command line** which allows users to perform tasks more efficiently than while using a GUI. Furthermore the command line is more suitable for automating often executed tasks.

Indeed, experienced system administrators write customized scripts to accomplish such repetitive tasks, standardized for each particular environment. We will discuss such scripting later in much detail.

In this section, we will concentrate on command line file and text manipulation related utilities.

cat

cat is short for concatenate and is one of the most frequently used Linux command line utilities. It is often used to read and print files as well as for simply viewing file contents. To view a file, use the following command:
$ cat <filename>

For example, cat readme.txt will display the contents of readme.txt on the terminal. Often the main purpose of**cat,** however, is to combine (concatenate) multiple files together. You can perform the actions listed in the following table using **cat**:

Command	Usage
cat file1 file2	Concatenate multiple files and display the output; i.e., the entire content of the first file is followed by that of the second file.
cat file1 file2 > newfile	Combine multiple files and save the output into a new file.
cat file >> existingfile	Append a file to the end of an existing file.

cat > file	Any subsequent lines typed will go into the file until CTRL-D is typed.
cat >> file	Any subsequent lines are appended to the file until CTRL-D is typed.

The **tac** command (**cat** spelled backwards) prints the lines of a file in reverse order. (Each line remains the same but the order of lines is inverted.) The syntax of **tac** is exactly the same as for **cat** as in:

$ tac file
$ tac file1 file2 > newfile

Using cat Interactively

cat can be used to read from standard input (such as the terminal window) if no files are specified. You can use the > operator to create and add lines into a new file, and the >> operator to append lines (or files) to an existing file.

To create a new file, at the command prompt type cat > <filename> and press the **Enter** key.

This command creates a new file and waits for the user to edit/enter the text. After you finish typing the required text, press **CTRL-D** at the beginning of the next line to save and exit the editing.

Another way to create a file at the terminal is cat > <filename> << EOF. A new file is created and you can type the required input. To exit, enter EOF at the beginning of a line.

Note that EOF is case sensitive. (One can also use another word, such as STOP.)

echo

echo simply displays (echoes) text. It is used simply as in:

$ echo string

echo can be used to display a string on **standard output** (i.e., the terminal) or to place in a new file (using the >operator) or append to an already existing file (using the >> operator).

The –e option along with the following switches is used to enable special character sequences, such as the **newline**character or horizontal **tab**.

- \n represents newline

- \t represents horizontal tab

echo is particularly useful for viewing the values of environment variables (built-in shell variables). For example, echo $USERNAME will print the name of the user who has logged into the current terminal.

The following table lists **echo** commands and their usage:

Command	Usage
echo string > newfile	The specified string is placed in a new file.
echo string >> existingfile	The specified string is appended to the end of an already existing file.
echo $variable	The contents of the specified environment variable are displayed.

Introduction to sed and awk

It is very common to create and then repeatedly edit and/or extract contents from a file. Let's learn how to use **sed** and **awk** to easily perform such operations.

Note that many Linux users and administrators will write scripts using more comprehensive language utilities such as **python** and **perl**, rather than use **sed** and **awk** (and some other utilities we'll discuss later.) Using such utilities is certainly fine in most circumstances; one should always feel free to use the tools one is experienced with. However, the utilities that are described here are much lighter; i.e., they use fewer system resources, and execute faster. There are times (such as during booting the system) where a lot of time would be wasted using the more complicated tools, and the system may not even be able to run them. So the simpler tools will always be needed.

sed

sed is a powerful text processing tool and is one of the oldest earliest and most popular UNIX utilities. It is used to modify the contents of a file, usually placing the contents into a new file. Its name is an abbreviation for **stream editor**.

sed can filter text as well as perform substitutions in data streams, working like a churn-mill.

Input Stream → Working Stream → Output Stream

Data from an input source/file (or stream) is taken and moved to a working space. The entire list of operations/modifications is applied over the data in the working space and the final contents are moved to the standard output space (or stream).

sed Command Syntax

You can invoke **sed** using commands like those listed in the following table:

Command	Usage
sed -e command <filename>	Specify editing commands at the command line, operate on file and put the output on standard out (e.g., the terminal)
sed -f scriptfile <filename>	Specify a scriptfile containing sed commands, operate on file and put output on standard out.

The -e command option allows you to specify multiple editing commands simultaneously at the command line.

sed Basic Operations

Now that you know that you can perform multiple editing and filtering operations with **sed**, let's explain some of them in more detail. The table explains some basic operations, where pattern is the current string and replace_string is the new string:

Command	Usage
sed s/pattern/replace_string/ file	Substitute first string occurrence in a line
sed s/pattern/replace_string/g file	Substitute all string occurrences in a line
sed 1,3s/pattern/replace_string/g file	Substitute all string occurrences in a range of lines
sed -i s/pattern/replace_string/g file	Save changes for string substitution in the same file

You must use the -i option with care, because the action is not reversible. It is always safer to use **sed** without the –ioption and then replace the file yourself, as shown in the following example:

```
$ sed s/pattern/replace_string/g file > file2
```

The above command will replace all occurrences of pattern with replace_string in file1 and move the contents to file2. The contents of file2 can be viewed with cat file2. If you approve you can then overwrite the original file with mv file2 file1.

Example: To convert 01/02/… to JAN/FEB/…
```
sed -e 's/01/JAN/' -e 's/02/FEB/' -e 's/03/MAR/' -e 's/04/APR/' -e 's/05/MAY/' \
-e 's/06/JUN/' -e 's/07/JUL/' -e 's/08/AUG/' -e 's/09/SEP/' -e 's/10/OCT/' \
-e 's/11/NOV/' -e 's/12/DEC
```

awk

awk is used to extract and then print specific contents of a file and is often used to construct reports. It was created at Bell Labs in the 1970s and derived its name from the last names of its authors: Alfred **A**ho, Peter **W**einberger, and Brian**K**ernighan.

awk has the following features:

- It is a powerful utility and interpreted programming language.

- It is used to manipulate data files, retrieving, and processing text.

- It works well with **fields** (containing a single piece of data, essentially a column) and **records** (a collection of fields, essentially a line in a file).

awk is invoked as shown in the following:

Command	Usage
awk 'command' var=value file	Specify a command directly at the command line
awk -f scriptfile var=value file	Specify a file that contains the script to be executed along with f

As with **sed**, short **awk** commands can be specified directly at the command line, but a more complex script can be saved in a file that you can specify using the -f option.

awk Basic Operations

The table explains the basic tasks that can be performed using **awk**. The input file is read one line at a time, and for each line, **awk** matches the given pattern in the given order and performs the requested action. The -F option allows you to specify a particular **field separator** character. For example, the /etc/passwd file uses : to separate the fields, so the -F: option is used with the /etc/passwd file.

The command/action in **awk** needs to be surrounded with apostrophes (or single-quote (')). awk can be used as follows:

Command	Usage
awk '{ print $0 }' /etc/passwd	Print entire file
awk -F: '{ print $1 }' /etc/passwd	Print first field (column) of every line, separated by a space
awk -F: '{ print $1 $6 }' /etc/passwd	Print first and sixth field of every line

File Manipulation Utilities

In managing your files you may need to perform many tasks, such as sorting data and copying data from one location to another. Linux provides several file manipulation utilities that you can use while working with text files. In this section, you will learn about the following file manipulation programs:

- sort

- uniq

- paste

- join

- split

You will also learn about **regular expressions** and **search patterns**.

sort

sort is used to rearrange the lines of a text file either in ascending or descending order, according to a sort key. You can also sort by particular fields of a file. The default sort key is the order of the ASCII characters (i.e., essentially alphabetically).

sort can be used as follows:

Syntax	Usage

sort <filename>	Sort the lines in the specified file
cat file1 file2 | sort	Append the two files, then sort the lines and display the output on the terminal
sort -r <filename>	Sort the lines in reverse order

When used with the -u option, **sort** checks for unique values after sorting the records (lines). It is equivalent to running **uniq** (which we shall discuss) on the output of **sort**.

uniq

uniq is used to remove duplicate lines in a text file and is useful for simplifying text display. **uniq** requires that the duplicate entries to be removed are consecutive. Therefore one often runs**sort** first and then pipes the output into **uniq**; if **sort** is passed the -u option it can do all this in one step. In the example shown, the file is called names and was originally Ted, Bob, Alice, Bob, Carol, Alice.

To remove duplicate entries from some files, use the following command: sort file1 file2 | uniq > file3

OR

sort -u file1 file2 > file3

To count the number of duplicate entries, use the following command: uniq -c filename

paste

Suppose you have a file that contains the full name of all employees and another file that lists their phone numbers and Employee IDs. You want to create a new file that contains all the data listed in three columns: name, employee ID, and phone number. How can you do this effectively without investing too much time?

paste can be used to create a single file containing all three columns. The different columns are identified based on delimiters (spacing used to separate two fields). For example, delimiters can be a blank space, a tab, or an **Enter**. In the image provided, a single space is used as the delimiter in all files.

paste accepts the following options:

- -d delimiters, which specify a list of delimiters to be used instead of tabs for separating consecutive values on a single line. Each delimiter is used in turn; when the list has been exhausted, paste begins again at the first delimiter.

- -s, which causes **paste** to append the data in series rather than in parallel; that is, in a horizontal rather than vertical fashion.

  - **paste** can be used to combine fields (such as name or phone number) from different files as well as combine lines from multiple files. For example, line one from file1 can be combined with line one of file2, line two from file1 can be combined with line two of file2, and so on.

  - To paste contents from two files one can do:
    $ paste file1 file2

  - The syntax to use a different delimiter is as follows:
    $ paste -d, file1 file2

 - Common delimiters are 'space', 'tab', '|', 'comma', etc.
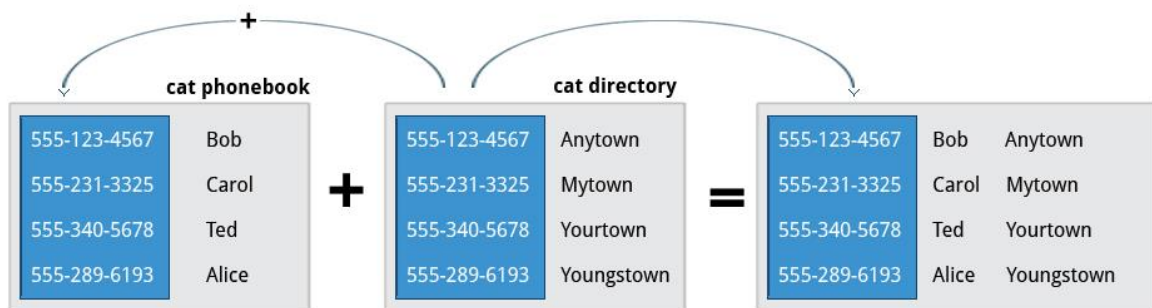
Join

- Suppose you have two files with some similar columns. You have saved employees' phone numbers in two files, one with their first name and the other with their last name. You want to combine the files without repeating the data of common columns. How do you achieve this?

- The above task can be achieved using **join**, which is essentially an enhanced version of **paste**. It first checks whether the files share common fields, such as names or phone numbers, and then joins the lines in two files based on a common field.

- To combine two files on a common field, at the command prompt type join file1 file2 and press the **Enter**key.

- For example, the common field (i.e., it contains the same values) among the phonebook and directory files is the phone number, as shown by the output of the following **cat** commands:

- $ cat phonebook
  555-123-4567 Bob
  555-231-3325 Carol

555-340-5678 Ted
555-289-6193 Alice

- $ cat directory
  555-123-4567 Anytown
  555-231-3325 Mytown
  555-340-5678 Yourtown
  555-289-6193 Youngstown

- The result of **join**ing these two file is as shown in the output of the following command:
  $ join phonebook directory
  555-123-4567 Bob Anytown
  555-231-3325 Carol Mytown
  555-340-5678 Ted Yourtown
  555-289-6193 Alice Youngstown

split

split is used to break up (or split) a file into equal-sized segments for easier viewing and manipulation, and is generally used only on relatively large files. By default **split** breaks up a file into 1,000-line segments. The original file remains unchanged, and set of new files with the same name plus an added prefix is created. By default, the **x** prefix is added. To split a file into segments, use the command split infile.
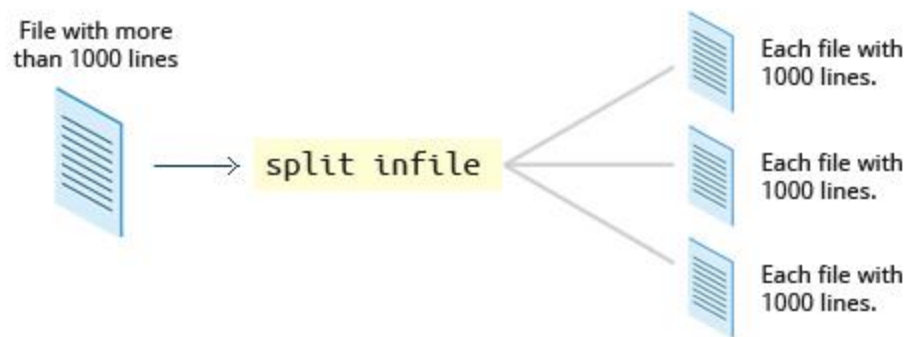
To split a file into segments using a different prefix, use the command split infile <Prefix>.

To demonstrate the use of **split,** we'll apply it to an american-english dictionary file of over 99,000 lines:

$ wc -l american-english
99171 american-english

where we have used the **wc** program (soon to be discussed) to report on the number of lines in the file. Then typing:

$ split american-english dictionary

will split the american-english file into equal-sized segments named 'dictionary'.

$ ls -l dictionary*
-rw-rw-r 1 me me 8552 Mar 23 20:19 dictionaryab
-rw-rw-r 1 me me 8653 Mar 23 20:19 dictionaryaa

Regular Expressions and Search Patterns

Regular expressions are text strings used for matching a specific **pattern**, or to search for a specific location, such as the start or end of a line or a word. Regular expressions can contain both normal characters or so-called**metacharacters**, such as * and $.

Many text editors and utilities such as **vi**, **sed**, **awk**, **find** and **grep** work extensively with regular expressions. Some of the popular computer languages that use regular expressions include **Perl**, **Python** and **Ruby**. It can get rather complicated and there are whole books written about regular expressions; we'll only skim the surface here.

These regular expressions are different from the wildcards (or "metacharacters") used in filename matching in command shells such as **bash** (which were covered in the earlier Chapter on Command Line Operations)**.** The table lists search patterns and their usage.

Search Patterns	Usage
.(dot)	Match any single character
a|z	Match a or z
$	Match end of string
*	Match preceding item 0 or more times

grep

grep is extensively used as a primary text searching tool. It scans files for specified patterns and can be used with regular expressions as well as simple strings as shown in the table.

Command	Usage
grep [pattern] <filename>	Search for a pattern in a file and print all matching lines
grep -v [pattern] <filename>	Print all lines that do **not** match the pattern

grep [0-9] <filename>	Print the lines that contain the numbers 0 through 9
grep -C 3 [pattern] <filename>	Print context of lines (specified number of lines above and below the pattern) for matching the pattern. Here the number of lines is specified as 3.

tr

In this section, you will learn about some additional text utilities that you can use for performing various actions on your Linux files, such as changing the case of letters or determining the count of words, lines, and characters in a file.

The **tr** utility is used to **translate** specified characters into other characters or to delete them. The general syntax is as follows:

$ tr [options] set1 [set2]

The items in the square brackets are optional. **tr** requires at least one argument and accepts a maximum of two. The first, designated set1 in the example, lists the characters in the text to be replaced or removed. The second, set2, lists the characters that are to be substituted for the characters listed in the first argument. Sometimes these sets need to be surrounded by apostrophes (or single-quotes (')) in order to have the shell ignore that they mean something special to the shell. It is usually safe (and may be required) to use the single-quotes around each of the sets as you will see in the examples below.

For example, suppose you have a file named city containing several lines of text in mixed case. To translate all lower case characters to upper case, at the command prompt type cat city | tr a-z A-Z and press the **Enter** key.

Command	Usage
$ tr abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ	Convert lower case to upper case
$ tr '{}' '()' < inputfile > outputfile	Translate braces into parenthesis

$ echo "This is for testing" | tr [:space:] '\t'	Translate white-space to tabs
$ echo "This is for testing" | tr -s [:space:]	Squeeze repetition of characters using -s
$ echo "the geek stuff" | tr -d 't'	Delete specified characters using -d option
$ echo "my username is 432234" | tr -cd [:digit:]	Complement the sets using -c option
$ tr -cd [:print:] < file.txt	Remove all non-printable character from a file
$ tr -s '\n' ' ' < file.txt	Join all the lines in a file into a single line

tee

tee takes the output from any command, and while sending it to standard output, it also saves it to a file. In other words, it "tees" the output stream from the command: one stream is displayed on the standard output and the other is saved to a file.

For example, to list the contents of a directory on the screen and save the output to a file, at the command prompt type ls -l | tee newfile and press the **Enter** key.

Typing cat newfile will then display the output of ls –l.

wc

wc (word count) counts the number of lines, words, and characters in a file or list of files. Options are given in the table below.

By default all three of these options are active.

For example, to print the number of lines contained in a file, at the command prompt type wc -l filename and press the **Enter** key.

Option	Description
–l	display the number of lines.
-c	display the number of bytes.
-w	display the number of words.

cut

cut is used for manipulating column-based files and is designed to extract specific columns. The default column separator is the**tab** character. A different delimiter can be given as a command option.

For example, to display the third column delimited by a blank space, at the command prompt type ls -l | cut -d" " -f3and press the **Enter** key.
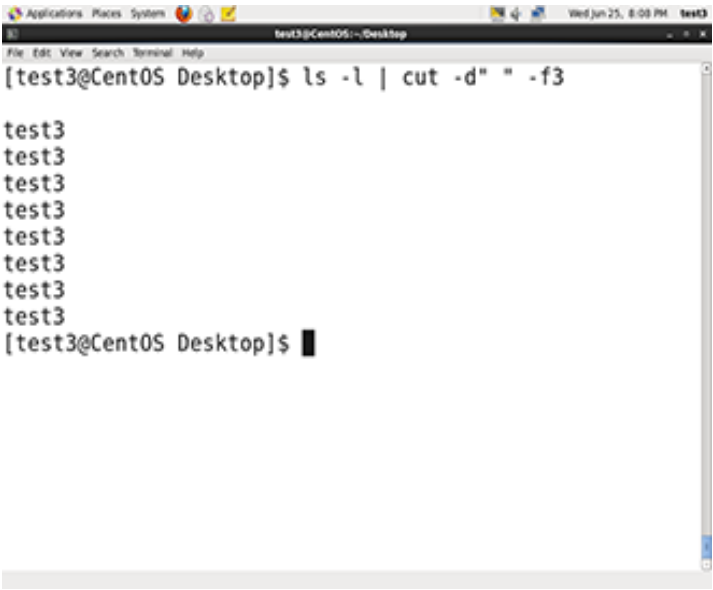
You have completed this chapter. Let's summarize the key concepts covered:

- The **command line** often allows the users to perform tasks more efficiently than the GUI.

- **cat**, short for **concatenate**, is used to read, print and combine files.

- **echo** displays a line of text either on standard output or to place in a file.

- **sed** is a popular **stream editor** often used to filter and perform substitutions on files and text data streams.

- **awk** is a interpreted programming language typically used as a data extraction and reporting tool.

- **sort** is used to sort text files and output streams in either ascending or descending order.

- **uniq** eliminates duplicate entries in a text file.

- **paste** combines fields from different files and can also extract and combine lines from multiple sources.

- **join** combines lines from two files based on a common field. It works only if files share a common field.

- **split** breaks up a large file into equal-sized segments.

- **Regular expressions** are text strings used for **pattern matching**. The pattern can be used to search for a specific location, such as the start or end of a line or a word.

- **grep** searches text files and data streams for patterns and can be used with regular expressions.

- **tr** translates characters, copies standard input to standard output, and handles special characters.

- **tee** accepts saves a copy of standard output to a file while still displaying at the terminal.

- **wc** (**word count**) displays the number of lines, words and characters in a file or group of files.

- **cut** extracts columns from a file.

- **less** views files a page at a time and allows scrolling in both directions.

- **head** displays the first few lines of a file or data stream on standard output. By default it displays 10 lines.

- **tail** displays the last few lines of a file or data stream on standard output. By default it displays 10 lines.

- **strings** extracts printable character strings from binary files.

- The **z** command family is used to read and work with compressed files.

Introduction to Printing

To manage printers and print directly from a computer or across a networked environment, you need to know how to configure and install a printer. Printing itself requires software that converts information from the application you are using to a language your printer can understand. The Linux standard for printing software is the **Common UNIX Printing System (CUPS)**.

CUPS Overview

CUPS is the software that is used behind the scenes to print from applications like a web browser descriptions produced by your line there, and so forth) and then or **LibreOffice**. It converts page application (put a paragraph here, draw a sends the information to the printer. It acts

as a **print server** for local as well as network printers.

Printers manufactured by different companies may use their own particular print languages and formats. **CUPS** uses a modular printing system which accommodates a wide variety of printers and also processes various data formats. This makes the printing process simpler; you can concentrate more on printing and less on how to print.

Generally, the only time you should need to configure your printer is when you use it for the first time. In fact, **CUPS** often figures things out on its own by detecting and configuring any printers it locates.

How Does CUPS Work?

CUPS carries out the printing process with the help of its various components:

- Configuration Files
- Scheduler
- Job Files
- Log Files
- Filter
- Printer Drivers
- Backend

You will learn about each of these components in detail in the next few screens.
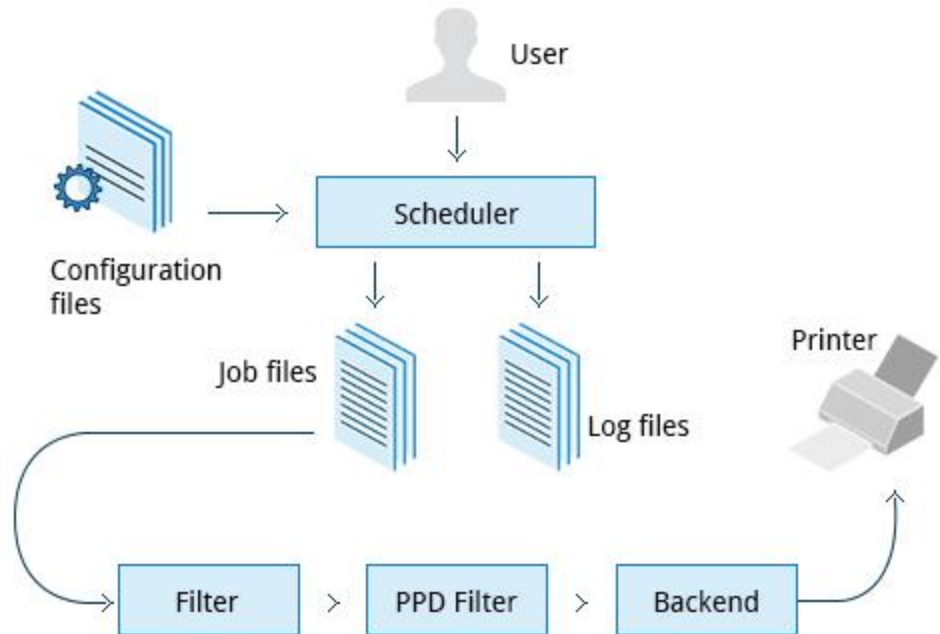
Scheduler

CUPS is designed around a **print scheduler** that manages print jobs, handles administrative commands, allows users to query the printer status, and manages the flow of data through all **CUPS** components.

As you'll see shortly, CUPS has a browser-based interface which allows you to view and manipulate the order and status of pending print jobs.
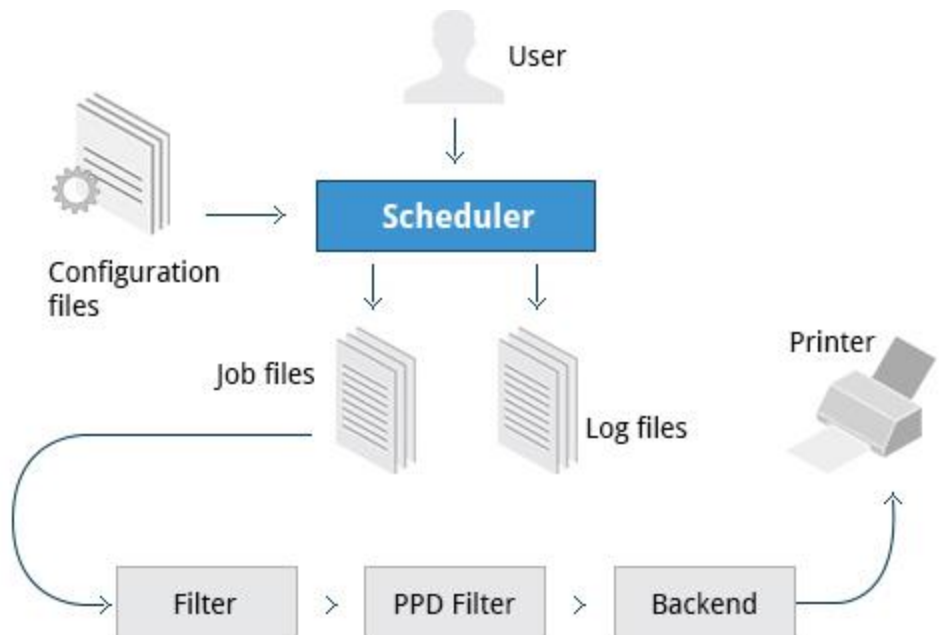
Configuration Files

The print scheduler reads server settings from several configuration files, the two most important of which are cupsd.conf and printers.conf. These and all other **CUPS** related configuration files are stored under the /etc/cups/ directory.

cupsd.conf is where most system-wide settings are located; it does not contain any printer-specific details. Most of the settings available in this file relate to network security, i.e. which systems can access **CUPS** network capabilities, how printers are advertised on the local network, what management features are offered, and so on.

printers.conf is where you will find the printer-specific settings. For every printer connected to the system, a corresponding section describes the printer's status and capabilities. This file is generated only after adding a printer to the system and should not be modified by hand.

You can view the full list of configuration files by typing: ls -l /etc/cups/
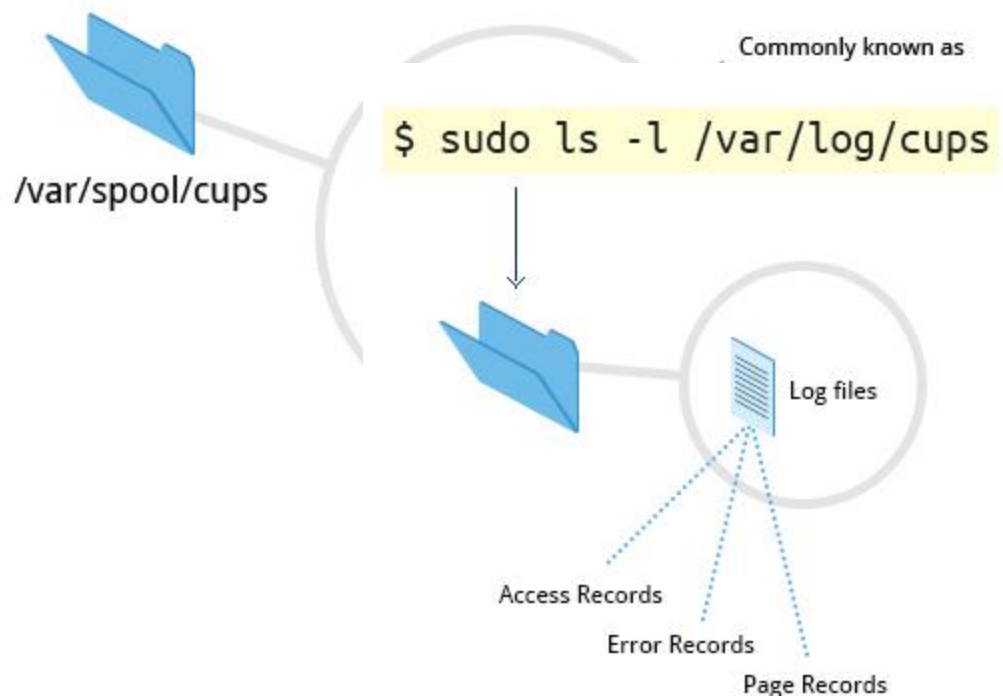
Job Files

CUPS stores print requests as files under the /var/spool/cups directory (these can actually be accessed before a document is sent to a printer). Data files are prefixed with the letter **d** while control files are prefixed with the letter **c**. After a printer successfully handles a job, data files are automatically removed. These data files belong to what is commonly known as the **print queue**.

Log Files

Log files are placed in /var/log/cups and are used by the scheduler to record activities that have taken place. These files include access, error, and page records.

To view what log files exist, type:
sudo ls -l /var/log/cups

(Note on some distributions permissions are set such that you don't

Commonly known as

$ sudo ls -l /var/log/cups

/var/spool/cups

Log files

Access Records

Error Records

Page Records

need the **sudo**.) You can view the log files with the usual tools.

Filters, Printer Drivers, and Backends

CUPS uses **filters** to convert job file formats to printable formats.
Printer **drivers** contain descriptions for currently connected and configured printers, and are usually stored under /etc/cups/ppd/. The print data is then sent to the printer through a filter and via a **backend** that helps to locate devices connected to the system.

So In short, when you execute a print command, the scheduler validates the command and processes the print job creating job files according to the settings specified in configuration files. Simultaneously, the scheduler records activities in the log files. Job files are processed with the help of the filter, printer driver, and backend, and then sent to the printer.
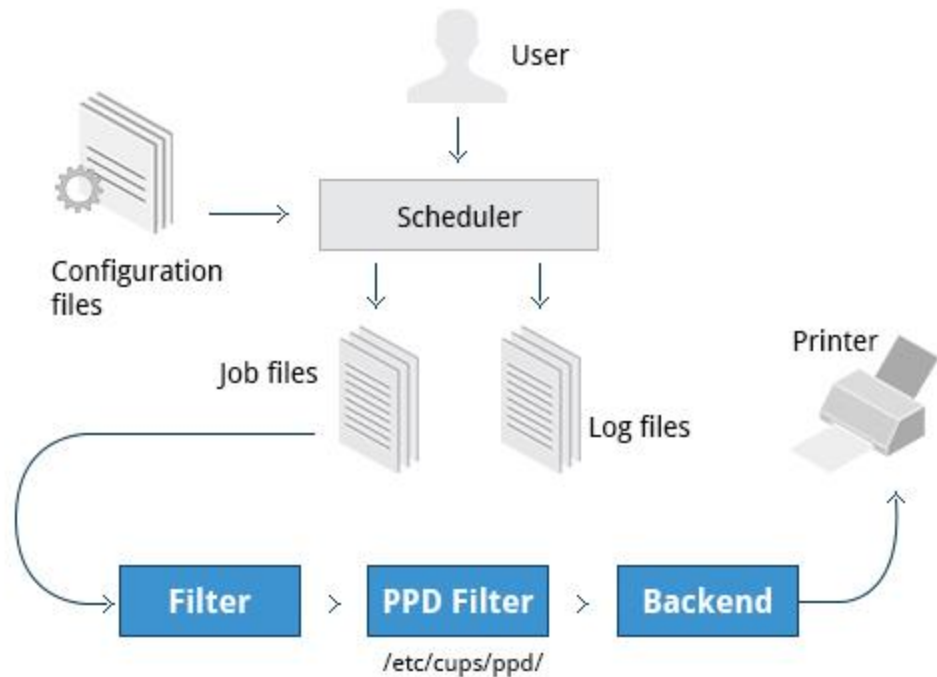
Installing CUPS

Due to printing being a relatively important and fundamental feature of any Linux distribution, most Linux systems come with **CUPS** preinstalled. In some cases, especially for Linux server setups, **CUPS** may have been left uninstalled. This may be fixed by installing the corresponding package manually. To install **CUPS**, please ensure that your system is connected to the Internet.

Demonstration of Installing CUPS

You can use the commands shown below to manually install **CUPS**:

  - CentOS: $ sudo yum install cups
  - OpenSUSE: $ sudo zypper install cups
  - Ubuntu: $ sudo apt-get install cups

The video below demonstrates this procedure for **Ubuntu**, the other two are similar, once the correct install command is provided.

Note: CUPS features are also supported by other packages such as cups-common and libcups2, which contains the core **CUPS** libraries. The above install command will make sure any needed packages are also installed.

After installing **CUPS**, you'll need to start and manage the **CUPS** daemon so that **CUPS** is ready for configuring a printer. Managing the **CUPS** daemon is simple; all management features are wrapped around the **cups** init script, which can be easily started, stopped, and restarted.

Configuring a Printer from the GUI

Each Linux distribution has a GUI application that lets you add, remove, and configure local or remote printers. Using this application, you can easily set up the system to use both local and network printers. The following screens show how to find and use the appropriate application in each of the distribution families covered in this course.

When configuring a printer, make sure the device is currently turned on and connected to the system; if so it should show up in the printer selection menu. If the printer is not visible, you may want to troubleshoot using tools that will determine if the printer is connected. For common USB printers, for example, the **lsusb** utility will show a line for the printer. Some printer manufacturers also require some extra software to be installed in order to make the printer visible to **CUPS**, however, due to the standardization these days, this is rarely required.

Adding Printers from the CUPS Web Interface

A fact that few people know is that **CUPS** also comes with its own web server, which makes a configuration interface available via a set of CGI scripts.

This web interface allows you to:

CUPS 1.7.2

CUPS is the standards-based, open source printing system developed by Apple Inc. for OS® X and other UNIX®-like operating systems.

CUPS for Users

Overview of CUPS

Command-Line Printing and Options

What's New in CUPS 1.7

User Forum

CUPS for Administrators

Adding Printers and Classes

Managing Operation Policies

Printer Accounting Basics

Server Security

Using Kerberos Authentication

Using Network Printers

cupsd.conf Reference

Find Printer Drivers

CUPS for Developers

Introduction to CUPS Programming

CUPS API

Filter and Backend Programming

HTTP and IPP APIs

PPD API

Raster API

PPD Compiler Driver Information File Reference

Developer Forum

- Add and remove local/remote printers

- Configure printers:

  – Local/remote printers

  – Share a printer as a CUPS server

- Control print jobs:

  – Monitor jobs

  – Show completed or pending jobs

– Cancel or move jobs

The **CUPS** web interface is available on your browser at: http://localhost:631

Some pages require a username and password to perform certain actions, for example to add a printer. For most Linux distributions, you must use the root password to add, modify, or delete printers or classes.

Printing from the Graphical Interface

Many graphical applications allow users to access printing features using the **CTRL-P** shortcut. To print a file, you first need to specify the printer (or a file name and location if you are printing to a file instead) you want to use; and then select the page setup, quality, and color options. After selecting the required options, you can submit the document for printing. The document is then submitted to **CUPS**. You can use your browser to access the **CUPS** web interface at http://localhost:631/ to monitor the status of the printing job. Now that you have configured the printer, you can print using either the Graphical or Command Line interfaces.

Printing from the Command-Line Interface

CUPS provides two command-line interfaces, descended from the **System V** and **BSD** flavors of UNIX. This means that you can use either **lp** (System V) or **lpr** (BSD) to print. You can use these commands to print text, PostScript, PDF, and image files.

These commands are useful in cases where printing operations must be automated (from shell scripts, for instance, which contain multiple commands in one file). You will learn more about the shell scripts in the upcoming chapters on**bash** scripts.

lp is just a command line front-end to the **lpr** utility that passes input to **lpr**. Thus, we will discuss only **lp** in detail. In the example shown here, the task is to print the file called test1.txt.

Using lp

lp and **lpr** accept command line options that help you perform all operations that the GUI can accomplish.**lp** is typically used with a file name as an argument.

Some **lp** commands and other printing utilities you can use are listed in the table.

Command	Usage
lp <filename>	To print the file to default printer
lp -d printer <filename>	To print to a specific printer (useful if multiple printers are available)

program | lp echo string | lp	To print the output of a program
lp -n number <filename>	To print multiple copies
lpoptions -d printer	To set the default printer
lpq -a	To show the queue status
lpadmin	To configure printer queues

The **lpoptions** utility can be used to set printer options and defaults. Each printer has a set of **tags** associated with it, such as the default number of copies and authentication requirements. You can execute the command lpoptions help to obtain a list of supported options. **lpoptions** can also be used to set system-wide values, such as the default printer.

Managing Print Jobs

You send a file to the shared printer. But when you go there to collect the printout, you discover another user has just started a 200 page job that is not time sensitive. Your file cannot be printed until this print job is complete. What do you do now?

In Linux, command line print job management commands allow you to monitor the job state as well as managing the listing of all printers and checking their status, and cancelling or moving print jobs to another printer.

Some of these commands are listed in the table.

| Command	Usage
lpstat -p -d	To get a list of available printers, along with their status
lpstat -a	To check the status of all connected printers,

	including job numbers
cancel job-id OR lprm job-id	To cancel a print job
lpmove job-id newprinter	To move a print job to new printer

Working with PostScript

PostScript is a standard **page description language.** It effectively manages scaling of fonts and vector graphics to provide quality printouts. It is purely a text format that contains the data fed to a PostScript interpreter. The format itself is a language that was developed by **Adobe** in the early 1980s to enable the transfer of data to printers.

Features of PostScript are:

- It can be used on any printer that is PostScript-compatible; i.e., any modern printer

- Any program that understands the PostScript specification can print to it

- Information about page appearance, etc. is embedded in the page
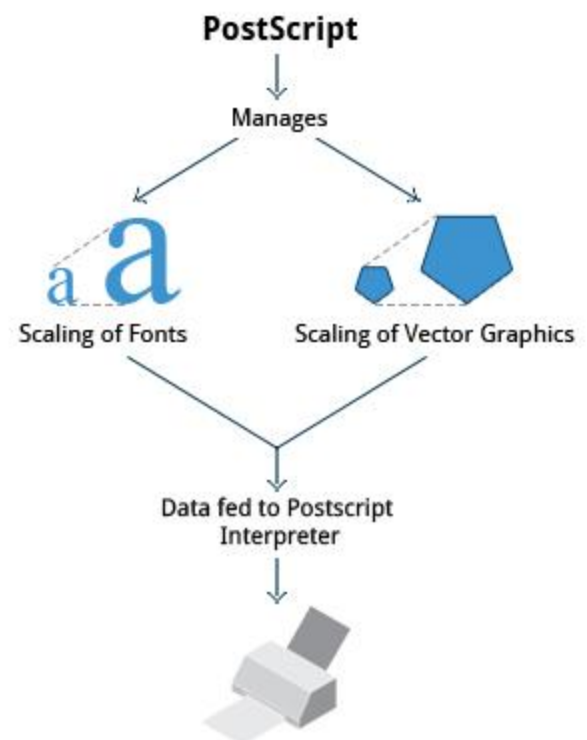
Working with enscript

enscript is a tool that is used to convert a text file to PostScript and other formats. It also supports **Rich Text Format (RTF)** and **HyperText Markup Language (HTML)**. For example, you can convert a text file to two column (-2) formatted **PostScript** using the command: enscript -2 -r -p psfile.ps textfile.txt. This command will also rotate (-r) the output to print so the width of the paper is greater than the height (aka landscape mode) thereby reducing the number of pages required for printing.

The commands that can be used with **enscript** are listed in the table below (for a file called 'textfile.txt').

Command	Usage
enscript -p psfile.ps textfile.txt	Convert a text file to PostScript (saved to psfile.ps)
enscript -n -p psfile.ps textfile.txt	Convert a text file to n columns where n=1-9 (saved in psfile.ps)
enscript textfile.txt	Print a text file directly to the default printer

Modifying PDFs with pdftk

At times, you may want to merge, split, or rotate PDF files; not all of these operations can be achieved while using a PDF viewer. A great way to do this is to use the "PDF Toolkit", **pdftk**, to perform a very large variety of sophisticated tasks. Some of these operations include:

- Merging/Splitting/Rotating PDF documents

- Repairing corrupted PDF pages

- Pulling single pages from a file

- Encrypting and decrypting PDF files

- Adding, updating, and exporting a PDF's **metadata**

- Exporting bookmarks to a text file

- Filling out PDF forms

In short, there's very little **pdftk** cannot do when it comes to working with PDF files; it is indeed the Swiss Army knife of PDF tools.

Installing pdftk on Different Family Systems

To install **pdftk** on **Ubuntu**, use the following command:

```
$ sudo apt-get install pdftk
```

On **CentOS:** `$ sudo yum install pdftk`

On **openSUSE**: `$ sudo zypper install pdftk`

You may find that **CentOS** (and **RHEL**) don't have **pdftk** in their packaging system, but you can obtain the PDF Toolkit directly from the PDF Lab's website by downloading from:
http://www.pdflabs.com/docs/install-pdftk-on-redhat-or-centos/

Using pdftk

You can accomplish a wide variety of tasks using **pdftk** including:

Command	Usage
pdftk 1.pdf 2.pdf cat output 12.pdf	Merge the two documents 1.pdf and 2.pdf. The output will be saved to 12.pdf.
pdftk A=1.pdf cat A1-2 output new.pdf	Write only pages 1 and 2 of 1.pdf. The output will be saved to new.pdf.
pdftk A=1.pdf cat A1-endright output new.pdf	Rotate all pages of 1.pdf 90 degrees clockwise and save result in new.pdf.

Encrypting PDF Files

If you're working with PDF files that contain confidential information and you want to ensure that only certain people can view the PDF file, you can apply a password to it using the user_pw option. One can do this by issuing a command such as:

```
$ pdftk public.pdf output private.pdf user_pw PROMPT
```
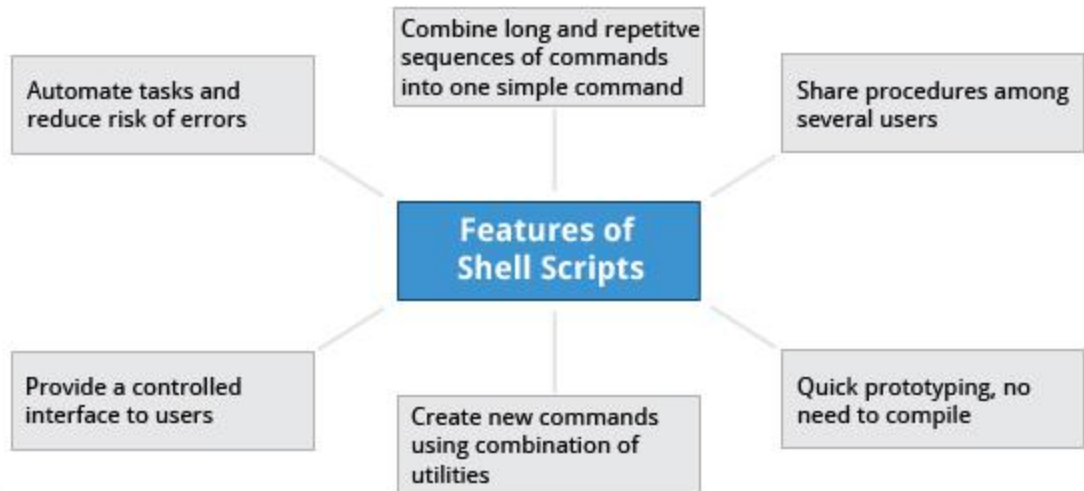
When you run this command, you will receive a prompt to set the required password, which can have a maximum of 32 characters. A new file, private.pdf, will be created with the identical content as public.pdf, but anyone will need to type the password to be able to view it.

You have completed this chapter. Let's summarize the key concepts covered:

- **CUPS** provides two command-line interfaces: the **System V** and **BSD** interfaces.

- The CUPS interface is available at http://localhost:631

- **lp** and **lpr** are used to submit a document to **CUPS** directly from the command line.

- **lpoptions** can be used to set printer options and defaults.

- PostScript effectively manages scaling of fonts and vector graphics to provide quality prints.

- **enscript** is used to convert a text file to PostScript and other formats.

  - **Portable Document Format (PDF)** is the standard format used to exchange documents while ensuring a certain level of consistency in the way the documents are viewed.

  - **pdftk** joins and splits PDFs; pulls single pages from a file; encrypts and decrypts PDF files; adds, updates, and exports a PDF's metadata; exports bookmarks to a text file; adds or removes attachments to a PDF; fixes a damaged PDF; and fills out PDF forms.

  - **pdfinfo** can extract information about PDF documents.

  - **flpsed** can add data to a PostScript document.

  - **pdfmod** is a simple application with a graphical interface that you can use to modify PDF documents.

Introduction to Scripts

Suppose you want to look up a filename, check if the associated file exists, and then respond accordingly, displaying a message confirming or not confirming the file's existence. If you only need to do it once, you can just type a sequence of commands at a terminal. However, if you need to do this multiple times, automation is the way to go. In order to automate sets of commands you'll need to learn how to write **shell scripts**, the most common of which are used with **bash**. The graphic illustrates several of the benefits of deploying scripts.

Features of Shell Scripts

- Automate tasks and reduce risk of errors
- Combine long and repetitve sequences of commands into one simple command
- Share procedures among several users
- Provide a controlled interface to users
- Create new commands using combination of utilities
- Quick prototyping, no need to compile

Introduction to Shell Scripts

Remember from our earlier discussion, a **shell** is a command line **interpreter** which provides the user interface for terminal windows. It can also be used to run scripts, even in non-interactive sessions without a terminal window, as if the commands were being directly typed in. For example typing: find . -name "*.c" -ls at the command line accomplishes the same thing as executing a script file containing the lines:

```
#!/bin/bash
find . -name "*.c" -ls
```

The #!/bin/bash in the first line should be recognized by anyone who has developed any kind of script in UNIX environments. The first line of the script, that starts with #!, contains the full path of the command interpreter (in this case /bin/bash) that is to be used on the file. As we will see on the next screen, you have a few choices depending upon which scripting language you use.

Command Shell Choices

The command **interpreter** is tasked with executing statements that follow it in the script. Commonly used interpreters include: /usr/bin/perl, /bin/bash, /bin/csh, /usr/bin/python and /bin/sh.

Typing a long sequence of commands at a terminal window can be complicated, time consuming, and error prone. By deploying shell scripts, using the command-line becomes an efficient and quick way to launch complex sequences of steps. The fact that shell scripts are saved in a file also makes it easy to use them to create new script variations and share standard procedures with

Shell

- /bin /sh
- /bin /ksh
- /bin /bash
- /bin /csh
- /bin /tcsh
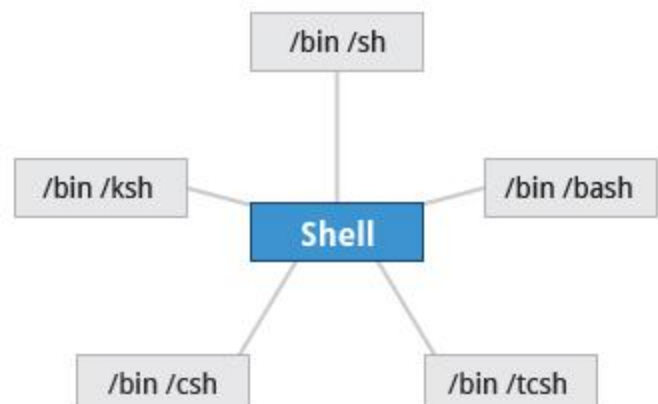
several users.

Linux provides a wide choice of shells; exactly what is available on the system is listed in /etc/shells. Typical choices are:

/bin/sh
/bin/bash
/bin/tcsh
/bin/csh
/bin/ksh

Most Linux users use the default **bash** shell, but those with long UNIX backgrounds with other shells may want to override the default.

bash Scripts

Let's write a simple **bash** script that displays a two-line message on the screen. Either type

```
$ cat > exscript.sh
  #!/bin/bash
  echo "HELLO"
  echo "WORLD"
```

and press **ENTER** and **CTRL-D** to save the file, or just create exscript.sh in your favorite text editor. Then, type chmod +x exscript.sh to make the file executable. (The chmod +x command makes the file executable for all users.) You can then run it by simply typing ./exscript.sh or by doing:

```
$ bash exscript.sh
  HELLO
  WORLD
```

Note if you use the second form, you don't have to make the file executable.

Interactive Example Using bash Scripts

Now, let's see how to create a more interactive example using a **bash** script. The user will be prompted to enter a value, which is then displayed on the screen. The value is stored in a temporary variable, sname. We can reference the value of a shell variable by using a $ in front of the variable name, such as $sname. To create this script, you need to create a file named ioscript.sh in your favorite editor with the following content:

```
  #!/bin/bash
  # Interactive reading of variables
  echo "ENTER YOUR NAME"
  read sname
  # Display of variable values
  echo $sname
```

Once again, make it executable by doing chmod +x ioscript.sh.
In the above example, when the script ./ioscript.sh is executed, the user will receive a prompt ENTER YOUR NAME. The user then needs to enter a value and press the **Enter** key. The value will then be printed out.

Additional note: The hash-tag/pound-sign/number-sign (#) is used to start comments in the script and can be placed anywhere in the line (the rest of the line is considered a comment).

Return Values

All shell scripts generate a **return value** upon finishing execution; the value can be set with the exit statement. Return values permit a process to monitor the exit state of another process often in a parent-child relationship. This helps to determine how this process terminated and take any appropriate steps necessary, contingent on success or failure.
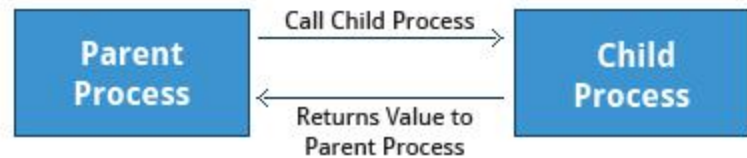
Viewing Return Values

As a script executes, one can check for a specific value or condition and return success or failure as the result. By convention, success is returned as 0, and failure is returned as a non-zero value. An easy way to demonstrate success and failure completion is to execute **ls** on a file that exists and one that doesn't, as shown in the following example, where the return value is stored in the environment variable represented by $?:

```
$ ls /etc/passwd
/etc/passwd
```

```
$ echo $?
0
```

In this example, the system is able to locate the file /etc/passwd and returns a value of 0 to indicate success; the return value is always stored in the $? environment variable. Applications often translate these return values into meaningful messages easily understood by the user.

Basic Syntax and Special Characters

Scripts require you to follow a standard language **syntax**. Rules delineate how to define variables and how to construct and format allowed statements, etc. The table lists some special character usages within **bash** scripts:

Character	Description
#	Used to add a comment, **except** when used as \#, or as #! when starting a script
\	Used at the end of a line to indicate continuation on to the next line
;	Used to interpret what follows as a new command

$	Indicates what follows is a variable

Note that when # is inserted at the beginning of a line of commentary, the whole line is ignored.
This line will not get executed.

**Splitting Long Commands
Over Multiple Lines**

Users sometimes need to combine several commands and statements and even conditionally execute them based on the behaviour of operators used in between them. This method is called **chaining of commands**.

Line 1 : Command 1 starts here   scp abc@server1.linux.com:\
Line 2 : Command 1 continues   /var/ftp/pub/userdata/custdata/read \
Line 3 : Command 1 continues   abc@server3.linux.co.in\
Line 4 : Command 1 ends   :/opt/oradba/master/abc/

The **concatenation operator** (****) is used to concatenate large commands over several lines in the shell.

For example, you want to copy the file **/var/ftp/pub/userdata/custdata/read** from **server1.linux.com** to the **/opt/oradba/master/abc** directory on **server3.linux.co.in**. To perform this action, you can write the command using the \ operator as:

scp abc@server1.linux.com:\
/var/ftp/pub/userdata/custdata/read \
abc@server3.linux.co.in:\
/opt/oradba/master/abc/

The command is divided into multiple lines to make it look readable and easier to understand. The \ operator at the end of each line combines the commands from multiple lines and executes it as one single command.

**Putting Multiple Commands on a
Single Line**

Sometimes you may want to group multiple commands on a single line. The ; (semicolon) character is used to separate these commands and execute them sequentially as if they had been typed on separate lines.

Line 1 : Command 1 ; Command 2 ; Command 3

cd /   ;   ls   ; cd /home/student

The three commands in the following example will all execute even if the ones preceding them fail:
$ make ; make install ; make clean

However, you may want to abort subsequent commands if one fails. You can do this using the && (and) operator as in:

$ make && make install && make clean

If the first command fails the second one will never be executed. A final refinement is to use the || (or) operator as in:

```
$ cat file1 || cat file2 || cat file3
```

In this case, you proceed until something succeeds and then you stop executing any further steps.

Functions

A **function** is a code block that implements a set of operations. Functions are useful for executing procedures multiple times perhaps with varying input variables. Functions are also often called **subroutines.** Using functions in scripts requires two steps:

1. Declaring a function
2. Calling a function

The function declaration requires a name which is used to invoke it. The proper syntax is:

```
function_name () {
   command...
}
```

For example, the following function is named display:

```
display () {
   echo "This is a sample function"
}
```

The function can be as long as desired and have many statements. Once defined, the function can be called later as many times as necessary. In the full example shown in the figure, we are also showing an often-used refinement: how to pass an argument to the function. The first argument can be referred to as $1, the second as $2, etc

Built-in Shell Commands

Shell scripts are used to execute sequences of commands and other types of statements. Commands can be divided into the following categories:

- Compiled applications

- Built-in **bash** commands

- Other scripts

Compiled applications are binary executable files that you can find on the filesystem. The shell script always has

Types of commands

Compiled applications	Built-in bash commands	Other scripts
rm	cd	
ls	pwd	
df	echo	
vi	read	
gzip	logout	
	printf	
	let	
	ulimit	
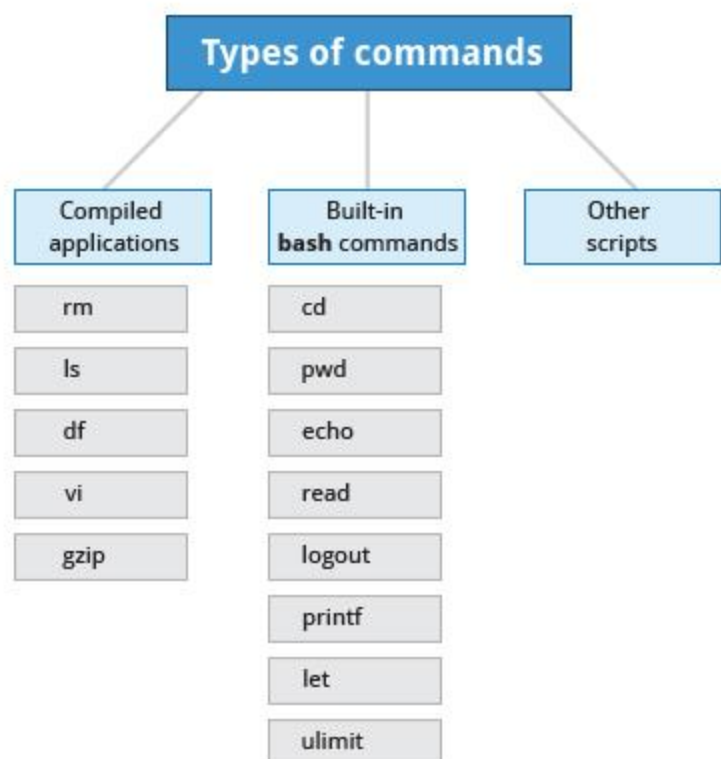
access to compiled applications such as **rm**, **ls**, **df**,**vi**, and **gzip**.

bash has many **built-in** commands which can only be used to display the output within a terminal shell or shell script. Sometimes these commands have the same name as executable programs on the system, such as**echo** which can lead to subtle problems. **bash** built-in commands include and cd, pwd, echo, read, logout,printf, let, and ulimit.

A complete list of **bash** built-in commands can be found in the **bash man** page, or by simply typing help.

Command Substitution

At times, you may need to substitute the result of a command as a portion of another command. It can be done in two ways:

- By enclosing the inner command with backticks (`)

- By enclosing the inner command in $( )

No matter the method, the innermost command will be executed in a newly launched shell environment, and the standard output of the shell will be inserted where the command substitution was done.

Virtually any command can be executed this way. Both of these methods enable command substitution; however, the $( ) method allows command nesting. New scripts should always use this more modern method. For example:

$ cd /lib/modules/$(uname -r)/

In the above example, the output of the command "uname –r" becomes the argument for the cd command.

Environment Variables

Almost all scripts use **variables** containing a value, which can be used anywhere in the script. These variables can either be user or system defined. Many applications use such **environment variables** (covered in the "User Environment" chapter) for supplying inputs, validation, and controlling behaviour.

Some examples of standard environment variables areHOME, PATH, and HOST. When referenced, environment variables must be prefixed with the $ symbol as in $HOME. You can view and set the value of environment variables. For example, the following command displays the value stored in the PATH variable:

$ echo $PATH

However, no prefix is required when setting or modifying the variable value. For example, the following command sets the value of the MYCOLOR variable to blue:

$ MYCOLOR=blue

You can get a list of environment variables with the **env**, **set**, or **printenv** commands.

Exporting Variables

By default, the variables created within a script are available only to the subsequent steps of that script. Any child processes (sub-shells) do not have automatic access to the values of these variables. To make them available to child processes, they must be promoted to environment variables using the **export** statement as in:

export VAR=value

or

VAR=value ; export VAR

While child processes are allowed to modify the value of exported variables, the parent will not see any changes; exported variables are not shared, but only copied.

Script Parameters

Users often need to pass parameter values to a script, such as a filename, date, etc. Scripts will take different paths or arrive at different values according to the parameters (command arguments) that are passed to them. These values can be text or numbers as in:

$ ./script.sh /tmp
$ ./script.sh 100 200

Within a script, the parameter or an argument is represented with a $ and a number. The table lists some of these parameters.

Parameter	Meaning
$0	Script name
$1	First parameter
$2, $3, etc.	Second, third parameter, etc.
$*	All parameters
$#	Number of arguments

Using Script Parameters

Using your favorite text editor, create a new script file named **script3.sh** with the following contents:

```
#!/bin/bash
echo The name of this program is: $0
echo The first argument passed from the command line is: $1
echo The second argument passed from the command line is: $2
echo The third argument passed from the command line is: $3
echo All of the arguments passed from the command line are : $*
echo
echo All done with $0
```

Make the script executable with **chmod +x**. Run the script giving it three arguments as in: script3.sh one two three, and the script is processed as follows:

$0 prints the script name: script3.sh
$1 prints the first parameter: one
$2 prints the second parameter: two
$3 prints the third parameter: three
$* prints all parameters: one two three
 The final statement becomes: All done with script3.sh

Output Redirection

Most operating systems accept input from the keyboard and display the output on the terminal. However, in shell scripting you can send the output to a file. The process of diverting the output to a file is called output **redirection**.

The > character is used to write output to a file. For example, the following command sends the output of **free** to the file /tmp/free.out:

$ free > /tmp/free.out

To check the contents of the /tmp/free.out file, at the command prompt type cat /tmp/free.out.

Two > characters (>>) will append output to a file if it exists, and act just like > if the file does not already exist.

Input Redirection

Just as the output can be redirected **to** a file, the input of a command can be read **from** a file. The process of reading input from a file is called input redirection and uses the < character. If you create a file called script8.sh with the following contents:

```
#!/bin/bash
echo "Line count"
wc -l < /temp/free.out
```

and then execute it with chmod +x script8.sh ; ./script8.sh, it will count the number of lines from the **/temp/free.out** file and display the results.

The if Statement

Conditional decision making using an if statement, is a basic construct that any useful programming or scripting language must have.

When an if statement is used, the ensuing actions depend on the evaluation of specified conditions such as:

- Numerical or string comparisons

- Return value of a command (0 for success)

- File existence or permissions

In compact form, the syntax of an if statement is:

if TEST-COMMANDS; then CONSEQUENT-COMMANDS; fi

A more general definition is:

```
if condition
then
    statements
else
    statements
fi
```

Using the if Statement

The following if statement checks for the /etc/passwd file, and if the file is found it displays the message /etc/passwd exists.:

```
if [ -f /etc/passwd ]
then
    echo "/etc/passwd exists."
fi
```

Notice the use of the square brackets ([]) to delineate the test condition. There are many other kinds of tests you can perform, such as checking whether two numbers are equal to, greater than, or less than each other and make a decision accordingly; we will discuss these other tests.

In modern scripts you may see doubled brackets as in [[-f /etc/passwd]]. This is not an error. It is never wrong to do so and it avoids some subtle problems such as referring to an empty environment variable without surrounding it in double quotes; we won't talk about this here.

Testing for Files

You can use the if statement to test for file attributes such as:

- File or directory existence

- Read or write permission

- Executable permission
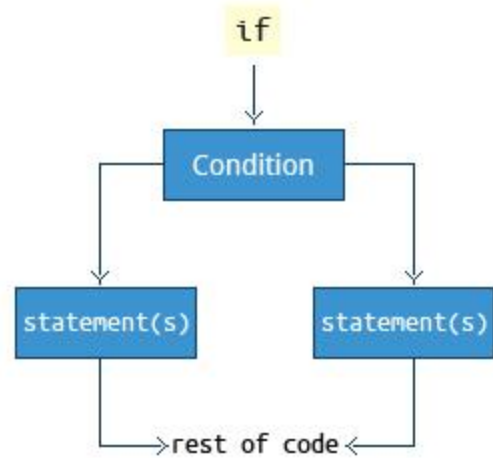
For example, in the following example:
```
if [ -f /etc/passwd ] ; then
    ACTION
fi
```

the if statement checks if the file /etc/passwd is a regular file.
Note the very common practice of putting **"; then"** on the same line as the **if** statement.

bash provides a set of **file conditionals**, that can used with the if statement, including:

Condition	Meaning
-e file	Check if the file exists.
-d file	Check if the file is a directory.
-f file	Check if the file is a regular file (i.e., not a symbolic link, device node, directory, etc.)
-s file	Check if the file is of non-zero size.
-g file	Check if the file has sgid set.
-u file	Check if the file has suid set.
-r file	Check if the file is readable.
-w file	Check if the file is writable.
-x file	Check if the file is executable.

You can view the full list of file conditions using the command man 1 test.

Example of Testing of Strings

You can use the `if` statement to compare strings using the operator `==` (two equal signs). The syntax is as follows:

```
if [ string1 == string2 ] ; then
    ACTION
fi
```

Let's now consider an example of testing strings.

In the example illustrated here, the `if` statement is used to compare the input provided by the user and accordingly display the result.

Numerical Tests

You can use specially defined operators with the `if` statement to compare numbers. The various operators that are available are listed in the table.

Operator	Meaning
-eq	Equal to
-ne	Not equal to
-gt	Greater than
-lt	Less than
-ge	Greater than or equal to
-le	Less than or equal to

The syntax for comparing numbers is as follows:
```
exp1 -op exp2
```

Arithmetic Expressions

Arithmetic expressions can be evaluated in the following three ways (spaces are important!):

- Using the **expr** utility: **expr** is a standard but somewhat deprecated program. The syntax is as follows:

```
expr 8 + 8
echo $(expr 8 + 8)
```

- Using the `$((...))` syntax: This is the built-in shell format. The syntax is as follows:

```
echo $((x+1))
```

- Using the built-in shell command `let`. The syntax is as follows:

```
let x=( 1 + 2 ); echo $x
```

In modern shell scripts the use of **expr** is better replaced with `var=$((...))`

You have completed this chapter. Let's summarize the key concepts covered:

- Scripts are a sequence of statements and commands stored in a file that can be executed by a shell. The most commonly used shell in Linux is **bash.**

- Command substitution allows you to substitute the result of a command as a portion of another command.

- Functions or routines are a group of commands that are used for execution.

- Environmental variables are quantities either pre-assigned by the shell or defined and modified by the user.

  - To make environment variables visible to child processes, they need to be **exported**.

  - Scripts can behave differently based on the parameters (values) passed to them.

  - The process of writing the output to a file is called output redirection.

  - The process of reading input from a file is called input redirection.

  - The `if` statement is used to select an action based on a condition.

  - Arithmetic expressions consist of numbers and arithmetic operators, such as +, -, and *.

What Is a Process?

A process is simply an instance of one or more related **tasks (threads)** executing on your computer. It is not the same as a**program** or a **command**; a single program may actually start several processes simultaneously. Some processes are independent of each other and others are related. A failure of one process may or may not affect the others running on the system.

Processes use many system resources, such as memory, CPU (central processing unit) cycles, and peripheral devices such as printers and displays. The operating system (especially the kernel) is responsible for allocating a proper share of these resources to each process and ensuring overall optimum utilization.
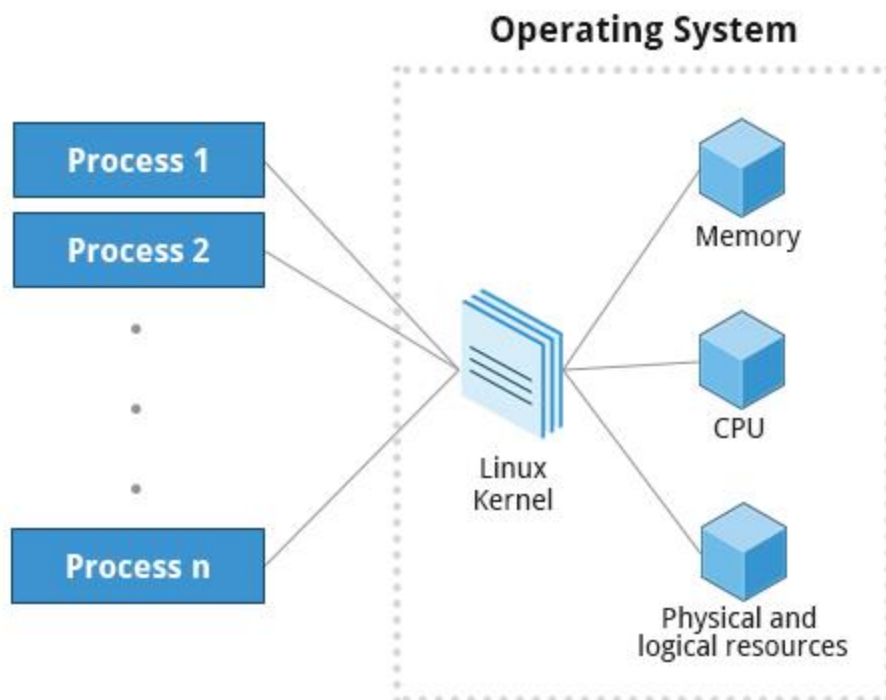
Process Types

A terminal window (one kind of command shell), is a process that runs as long as needed. It allows users to execute programs and access resources in an interactive environment. You can also run programs in the **background**, which means they become **detached** from the shell.

Processes can be of different types according to the task being performed. Here are some different process types along with their descriptions and examples.

Process Type	Description	Example
Interactive Processes	Need to be started by a user, either at a command line or through a graphical interface such as an icon or a menu selection.	**bash, firefox, top**
Batch Processes	Automatic processes which are scheduled from and then disconnected from the terminal. These tasks are queued and work on a **FIFO** (First In, First Out) basis.	**updatedb**
Daemons	Server processes that run continuously. Many are launched	**httpd, xinetd,**

	during system startup and then wait for a user or system request indicating that their service is required.	**sshd**
Threads	Lightweight processes. These are **tasks** that run under the umbrella of a main process, sharing memory and other resources, but are scheduled and run by the system on an individual basis. An individual thread can end without terminating the whole process and a process can create new threads at any time. Many non-trivial programs are multi-threaded.	**gnome-terminal, firefox**
Kernel Threads	Kernel tasks that users neither start nor terminate and have little control over. These may perform actions like moving a thread from one CPU to another, or making sure input/output operations to disk are completed.	**kswapd0, migration, ksoftirqd**

Process Scheduling and States

When a process is in a so-called **running** state, it means it is either currently executing instructions on a CPU, or is waiting for a share (or **time slice)** so it can run. A critical kernel routine called the **scheduler** constantly shifts processes in and out of the CPU, sharing time according to relative priority, how much time is needed and how much has already been granted to a task. All processes in this state reside on what is called a **run queue** and on a computer with multiple CPUs, or cores, there is a run queue on each.

However, sometimes processes go into what is called a **sleep** state, generally when they are waiting for something to happen before they can resume, perhaps for the user to type something. In this condition a process is sitting in a **wait**queue.
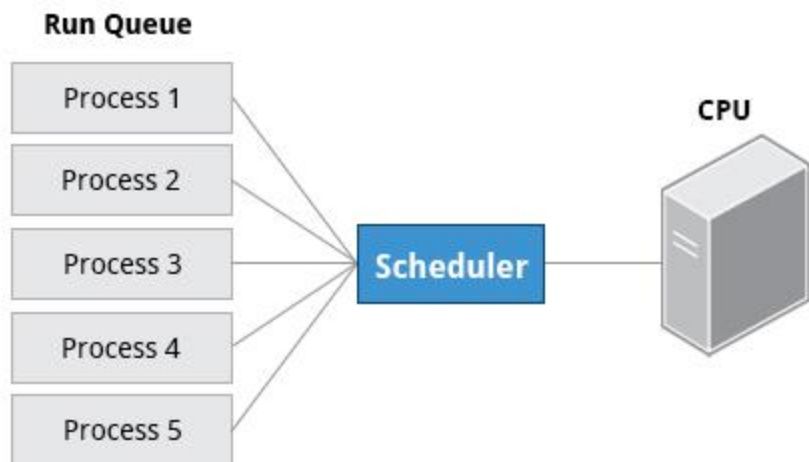
There are some other less frequent process states, especially when a process is terminating. Sometimes a child process completes but its parent process has not asked about its state. Amusingly such a process is said to be in a **zombie** state; it is not really alive but still shows up in the system's list of processes.

Process and Thread IDs

At any given time there are always multiple processes being executed. The operating system keeps track of them by assigning each a unique **process ID** (**PID**) number. The PID is used to track process state, cpu usage, memory use, precisely where resources are located in memory, and other characteristics.

New PIDs are usually assigned in ascending order as processes are born. Thus PID 1 denotes the **init** process (initialization process), and succeeding processes are gradually assigned higher numbers.

The table explains the PID types and their descriptions:

ID Type	Description
Process ID (PID)	Unique Process ID number
Parent Process ID (PPID)	Process (Parent) that started this process
Thread ID (TID)	Thread ID number. This is the same as the PID for single-threaded processes. For a multi-threaded process, each thread shares the same PID but has a unique TID.

User and Group IDs

Many users can access a system simultaneously, and each user can run multiple processes. The operating system identifies the user who starts the process by the Real User ID (**RUID**) assigned to the user.

The user who determines the access rights for the users is identified by the Effective UID (**EUID**). The EUID may or may not be the same as the RUID.

Users can be categorized into various groups. Each group is identified by the Real Group ID, or **RGID**. The access rights of the group are determined by the Effective Group ID, or **EGID**. Each user can be a member of one or more groups.

Most of the time we ignore these details and just talk about the User ID (**UID**).

USER IDS

RUID
Identifies the user
who started the process

EUID
Determines the access
rights of the user

USER GROUP IDS

RGID
Identifies the group
that started the process

EGID
Determines the access
rights of the group

More About Priorities

At any given time, many processes are running (i.e., in the run queue) on the system. However, a CPU can actually accommodate only one task at a time, just like a car can have only one driver at a time. Some processes are more important than others so Linux allows you to set and manipulate process **priority**. Higher priority processes are granted more time on the CPU.

	Process 1	Process 2	Process 3	· · ·	Process n
Nice Value	-20	-19	-18		19
Elapsed Time	0	1	2		n

The priority for a process can be set by specifying a **nice value**, or **niceness**, for the process. The lower the nice value, the higher the priority. Low values are assigned to important processes, while high values are assigned to processes that can wait longer. A process with a high nice value simply allows other processes to be executed first. In Linux, a nice value of -20 represents the highest priority and 19 represents the lowest. (This does sound kind of backwards, but this convention, the nicer the process, the lower the priority, goes back to the earliest days of UNIX.)

You can also assign a so-called **real-time priority** to time-sensitive tasks, such as controlling machines through a computer or collecting incoming data. This is just a very high priority and is not to be confused with what is called **hard real time** which is conceptually different, and has more to do with making sure a job gets completed within a very well-defined time window.
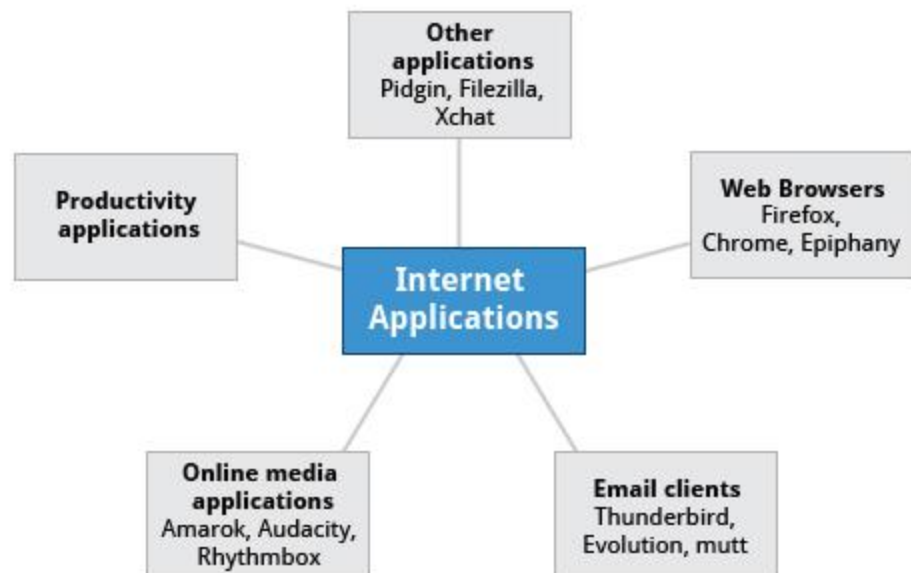
Internet Applications

The Internet is a global network that allows users around the world to perform multiple tasks such as searching for data, communicating through emails and online shopping. Obviously, you need to use network-aware applications to take advantage of the Internet. These include:

- Web browsers

- Email clients

- Online media applications

- Other applications

Other applications
Pidgin, Filezilla, Xchat

Productivity applications

Web Browsers
Firefox, Chrome, Epiphany

Internet Applications

Online media applications
Amarok, Audacity, Rhythmbox

Email clients
Thunderbird, Evolution, mutt

Web Browsers

As discussed in the earlier chapter on Network Operations, Linux offers a wide variety of web browsers, both graphical and text based, including:

- **Firefox**

- **Google Chrome**

- **Chromium**

- **Epiphany**

- **Konqueror**

- **w3m**

- **lynx**

Email Applications

Email applications allow for sending, receiving, and reading messages over the Internet. Linux systems offer a wide number of **email clients**, both graphical and text-based. In addition many users simply use their browsers to access their email accounts.

Most email clients use the **Internet Message Access Protocol (IMAP)** or the older **Post Office Protocol (POP)** to access emails stored on a remote mail server. Most email applications also display **HTML** (**HyperText Markup Language**) formatted emails that display objects, such as pictures and hyperlinks. The features of advanced email applications include the ability of importing address books/contact lists, configuration information, and emails from other email applications.
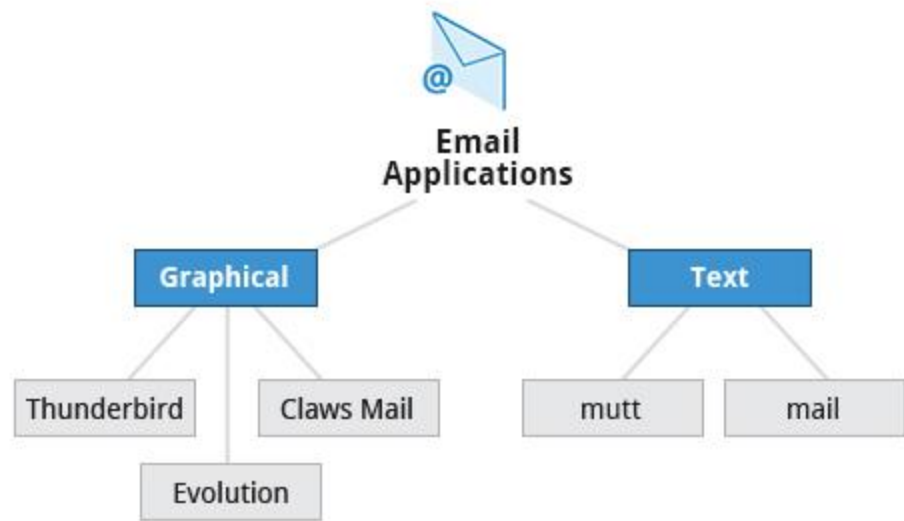
Linux supports the following types of email applications:

- Graphical email clients, such as **Thunderbird** (produced by **Mozilla)**, **Evolution**, and **Claws Mail**

- Text mode email clients such as **mutt** and **mail**

  - **Other Internet Applications**
  - Linux systems provide many other applications for performing Internet-related tasks. These include:

Application	Use

FileZilla
  Pidgin
  XChat
  Ekiga	

FileZilla	Intuitive graphical **FTP** client that supports **FTP**, **Secure File Transfer Protocol (SFTP)**, and **FTP Secured (FTPS).** Used to transfer files to/from**(FTP)** servers.
Pidgin	To access **GTalk**, **AIM**, **ICQ**, **MSN**, **IRC** and other messaging networks
Ekiga	To connect to **Voice over Internet Protocol (VoIP)** networks
XChat	To access **Internet Relay Chat (IRC)** networks

Not to be confused with Unix, Unix-like, or Linux.

POSIX (/ˈpɒzɪks/ *POZ-iks*), an acronym for **Portable Operating System Interface**,[1] is a family of standards specified by the IEEE for maintaining compatibility between operating systems. POSIX defines the application programming interface (API), along with command lineshells and utility interfaces, for software compatibility with variants of Unix and other operating systems.