**Presentation by**

Chaman Mandal

# Command line Presentation

## OVERVIEW

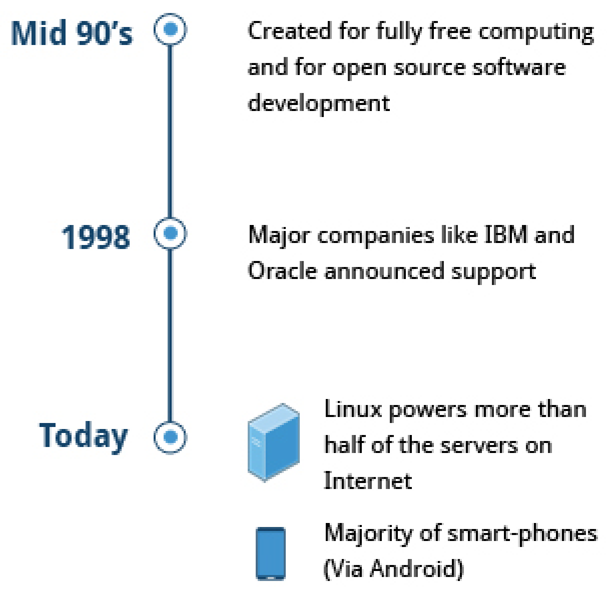
This document lists down learning objectives for the Linux/Unix command line. It will help to quickly start working in the Linux/Unix command line.

## LINUX PHILOSOPHY AND CONCEPTS

Linux is a free open source computer operating system, initially developed for **Intel x86**-based personal computers. It has been subsequently ported to many other hardware platforms.

Linus Torvalds was a student in Helsinki, Finland, in 1991, when he started a project: writing his own operating system **kernel**. He also collected together and/or developed the other essential ingredients required to construct an entire **operating system** with his kernel at the center. Soon, this became known as the **Linux** kernel.

In 1992, Linux was re-licensed using the **General Public License (GPL)** by **GNU** (a project of the Free Software Foundation or FSF, which promotes freely available software), which made it possible to build a worldwide community of developers. By combining the kernel with other system components from the GNU project, numerous other developers created complete systems called **Linux Distributions** in the mid-90's.

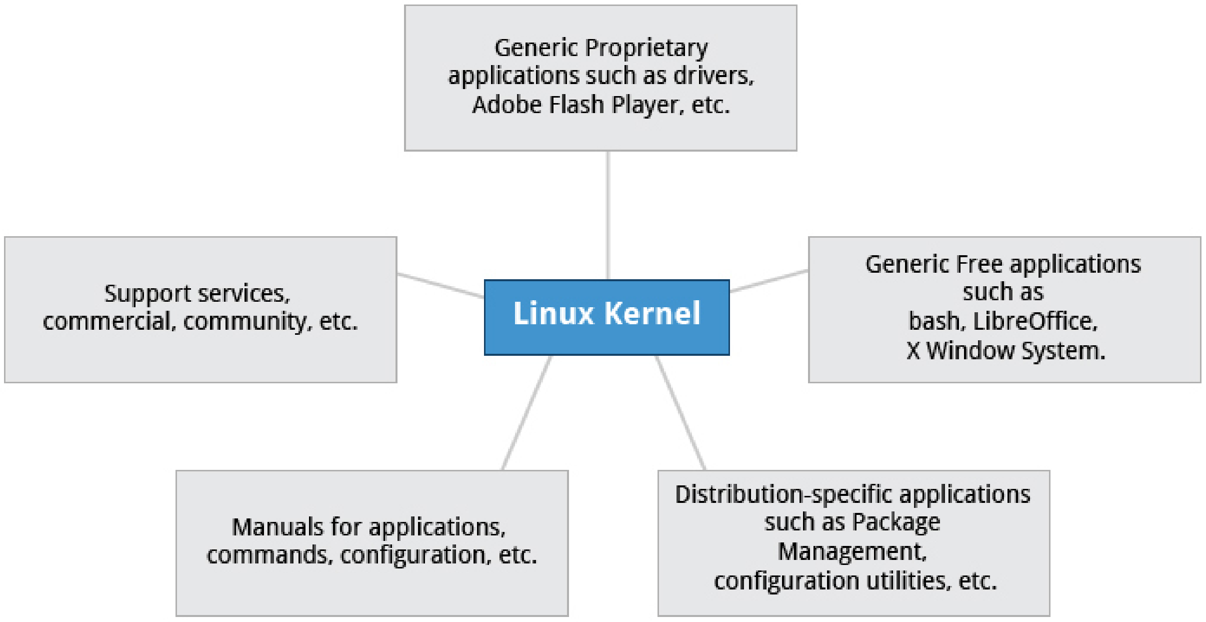

Mid 90's
Created for fully free computing and for open source software development
1998
Major companies like IBM and Oracle announced support
Today
Linux powers more than half of the servers on Internet
Majority of smart-phones (Via Android)

The Linux distributions created in the mid-90s provided the basis for fully free computing and became a driving force in the open source software movement. In 1998, major companies like **IBM** and **Oracle** announced their support for the Linux platform and began major development efforts as well.

Today, Linux powers more than half of the servers on the Internet, the majority of smartphones (via the **Android** system, which is built on top of Linux), and nearly all of the world's most powerful supercomputers.

Linux borrows heavily from the **UNIX** operating system because it was written to be a free and open source version of **UNIX**. Files are stored in a hierarchical filesystem, with the top node of the system being `root` or simply "`/`". Whenever possible, Linux makes its components available via files or objects that look like files. Processes, devices, and network sockets are all represented by file-like objects, and can often be worked with using the same utilities used for regular files.

Linux is a fully **multitasking** (i.e., multiple threads of execution are performed simultaneously), **multiuser** operating system, with built-in networking and service processes known as **daemons** in the **UNIX** world.

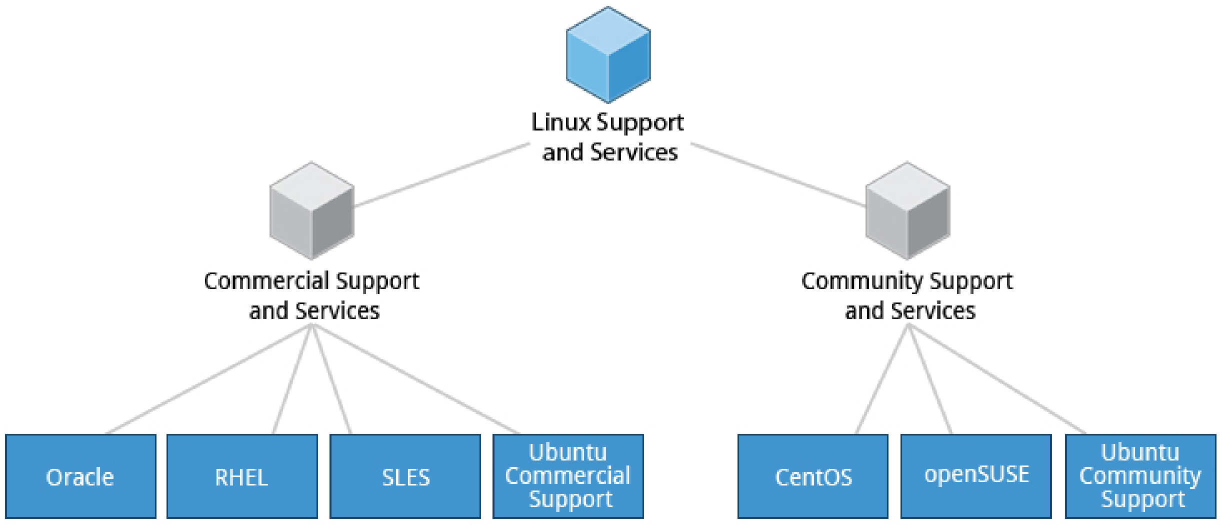

Generic Proprietary applications such as drivers, Adobe Flash Player, etc.
Support services, commercial, community, etc.
Linux Kernel
Generic Free applications such as bash, LibreOffice, X Window System.
Manuals for applications, commands, configuration, etc.
Distribution-specific applications such as Package Management, configuration utilities, etc.

So, what is a Linux distribution and how does it relate to the Linux kernel?

As illustrated above, the Linux kernel is the core of a computer operating system. 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. Each of these tools provides a small part of the complete system. Each tool is often its own separate project, with its own developers working to perfect that piece of the system.

As mentioned earlier, the current **Linux** kernel (as well as earlier release versions) can be found at www.kernel.org . The various **Linux** distributions may be based on different kernel versions. For example, the very popular **RHEL 7** distribution is based on the 3.10 kernel, which is not new , but is extremely stable. Other distributions may move more quickly in adopting the latest kernel releases. It is important to note that the kernel is not an all or nothing proposition, for example, **RHEL 7** and **CentOS 7** have incorporated many of the more recent kernel improvements into their older version, as have **Ubuntu, openSUSE, SLES,** etc.

Examples of other essential tools and ingredients provided by distributions include the **C/C++** compiler, the **gdb** debugger, the core system libraries applications need to link with in order to run, the low-level interface for drawing graphics on the screen, as well as the higher-level desktop environment, and the system for installing and updating the various components, including the kernel itself.


Linux Support and Services
Commercial Support and Services
Community Support and Services
Oracle
RHEL
SLES
Ubuntu Commercial Support
CentOS
openSUSE
Ubuntu Community Support

The vast variety of Linux distributions are designed to cater to many different audiences and organizations, according to their specific needs and tastes. However, large organizations, such as companies and governmental institutions and other entities, tend to choose the major commercially-supported distributions from **Red Hat, SUSE**, and **Canonical (Ubuntu)**.

**CentOS** is a popular free alternative to **Red Hat Enterprise Linux (RHEL)** and is often used by organizations that are comfortable operating without paid technical support. **Ubuntu** and **Fedora** are popular in the educational realm. **Scientific Linux** is favored by the scientific research community for its compatibility with scientific and mathematical software packages. Both **CentOS** and **Scientific Linux** are binary-compatible with **RHEL**; i.e., binary software packages in most cases will install properly across the distributions.

Many commercial distributors, including **Red Hat**, **Ubuntu**, **SUSE**, and **Oracle**, provide long term fee-based support for their distributions, as well as hardware and software certification. All major distributors provide update services for keeping your system primed with the latest security and bug fixes, and performance enhancements, as well as provide online support resources.

Summary of the key concepts covered:

- 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 multi-tasking, multi-user 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, Filesystem, 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 BASICS AND SYSTEM STARTUP

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

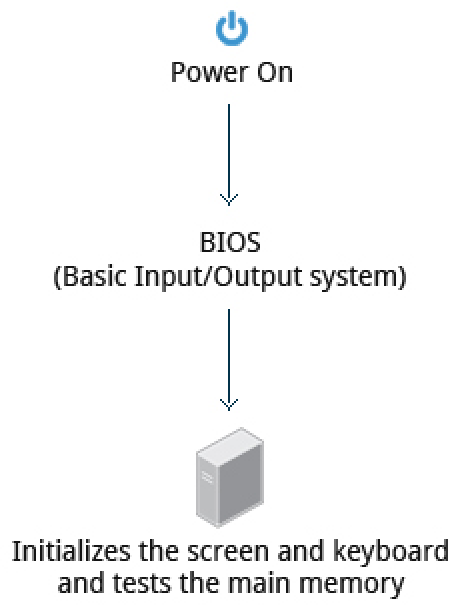
The Linux **boot process** is the procedure for initializing the system. It consists of everything that happens from when the computer power is first switched on until the user interface is fully operational.

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

On the other hand, the boot process can be rather technical. **You may want to come back and study this section later, if you want to first get a good feel for how to use a Linux system.**

## 1.The Boot Process

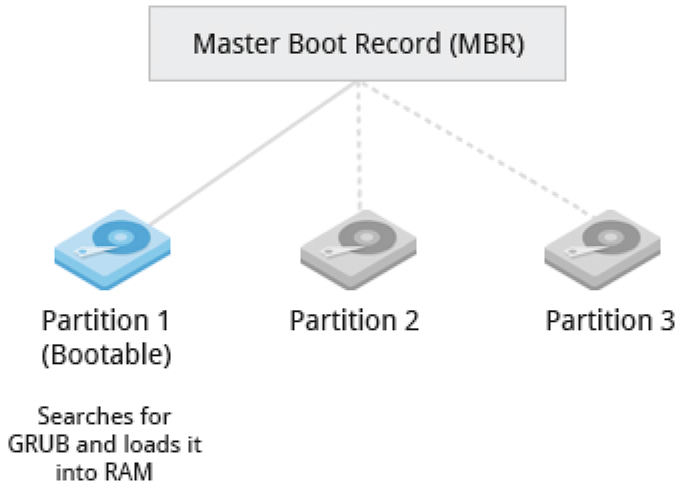
### BIOS - The First Step


Power On
BIOS
(Basic Input/Output system)
Initializes the screen and keyboard
and tests the main memory

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 controlled by the operating system (OS).

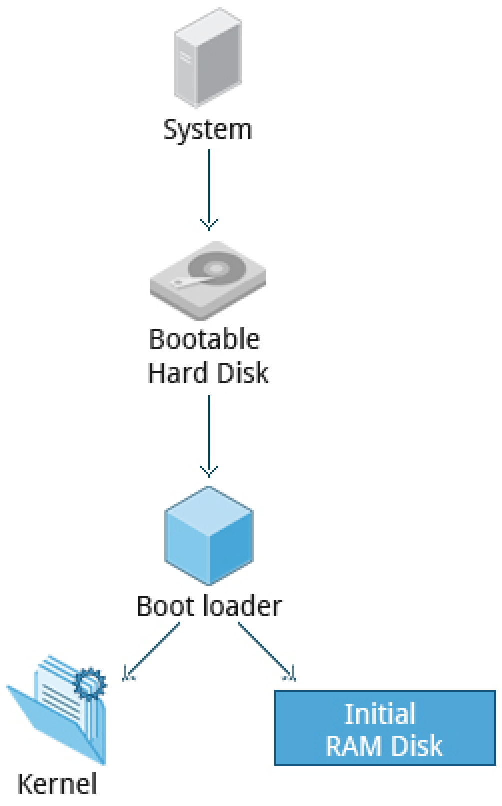
## Master Boot Record (MBR)



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).

A number of boot loaders exist for Linux; the most common ones are **GRUB** (for **GR**and **U**nified **B**oot loader) and **ISOLINUX** (for booting from removable media), and **DAS U-Boot** (for booting on embedded devices/appliances). 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 or filesystem (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**).

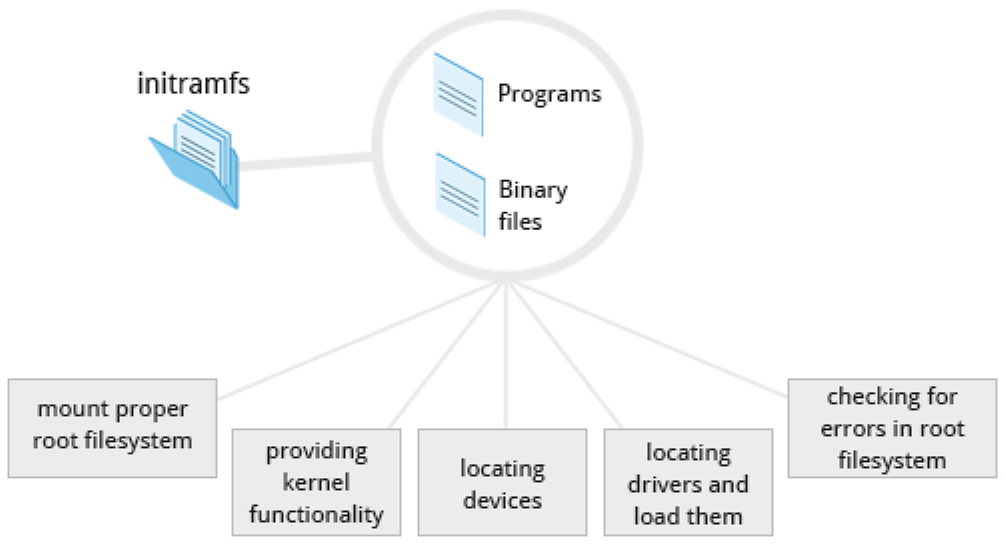
For systems using the **EFI**/**UEFI** method, **UEFI firmware** reads its **Boot Manager** data to determine which **UEFI** application is to be launched and from where (i.e., from which disk and partition the **EFI** partition can be found). The firmware then launches the **UEFI** application, for 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.

**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 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.

## Initial RAM Disk

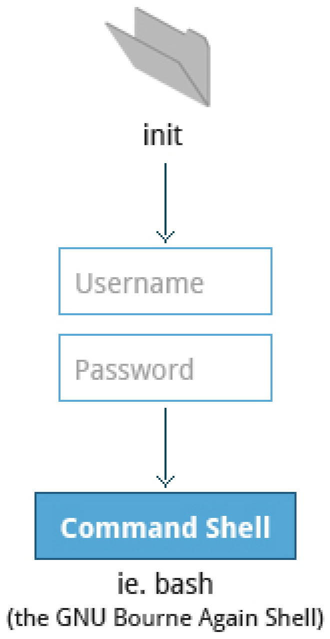

initramfs
Programs
Binary files
mount proper root filesystem
providing kernel functionality
locating devices
locating drivers and load them
checking for errors in root filesystem

The **initramfs** filesystem image contains programs and binary files that perform all actions needed to mount the proper root filesystem, like providing kernel functionality for the needed filesystem 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 filesystem has been found, it is checked for errors and mounted.

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

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

## Text-Mode Login



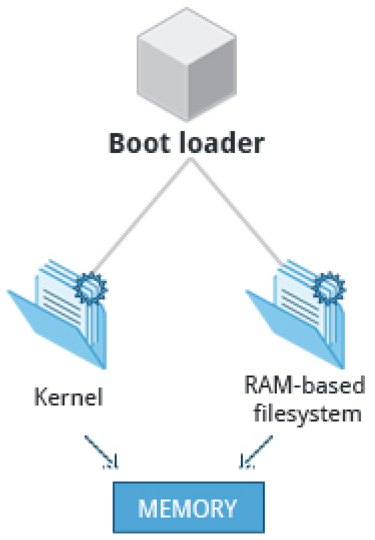
Near the end of the boot process, **init** starts a number of text-mode login prompts. These enable you to type your username, followed by your password, and to eventually get a command shell. However, if you are running a system with a graphical login interface, you will not see these at first.

As you will learn in the **Command Line Operations** section, 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`. Within a graphical environment, switching to a text console requires pressing `CTRL-ALT` + the appropriate function key (with `F7` or `F1` leading to the GUI).

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.

## 2.Kernel, init and Services

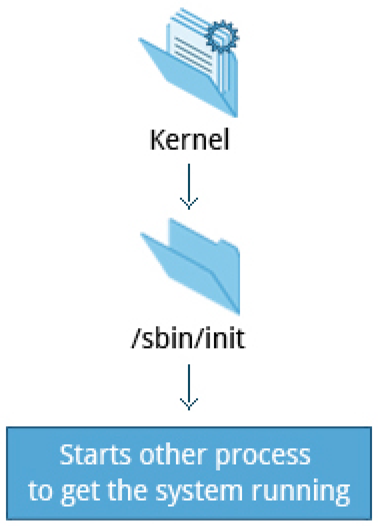
### The Linux Kernel



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.

## /sbin/init and Services



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

Besides starting the system, **init** is responsible for keeping the system running and for shutting it down cleanly. Whenever necessary, one of its responsibilities is to act as manager for all non-kernel processes; it cleans up after them upon completion, and restarts user login services as needed when users log in and out, and does the same for other background system services.

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.

However, all major recent distributions have moved away from this sequential runlevel method of system initialization, although they usually support the System V conventions for compatibility purposes. Next, we discuss the new methods, **systemd** and **Upstart**.

## Startup Alternatives: Upstart and systemd

The older startup system (**SysVinit**) viewed things as a **serial** process, divided into a series of sequential stages. Each stage required completion before the next could proceed. Thus, startup did not easily take advantage of the parallel processing that could be done on multiple processors or cores.

Furthermore, shutdown and reboot was seen as a relatively rare event; exactly how long it took was not considered important. This is no longer considered true, especially with mobile devices and embedded Linux systems. Thus, modern systems have required newer methods with enhanced capabilities. Finally, the older methods required rather complicated startup scripts which were difficult to keep universal across distribution versions, kernel versions, architectures, and types of systems. The two main alternatives developed were:

**Upstart**

- Developed by **Ubuntu** and first included in 2006
- Adopted in **Fedora 9** (in 2008) and in **RHEL 6** and its clones.

**systemd**

- Adopted by **Fedora** first (in 2011)

- Adopted by **RHEL 7** and **SUSE**
- Replaced **Upstart** in **Ubuntu 16.04**.

While the migration to **systemd** has been rather controversial, it has been pursued by the major distributions and so we will not discuss the older **System V** method or **Upstart**, which has become a dead end. Regardless of how one feels about the controversies or the technical methods of **systemd**, almost universal adoption has made learning how to work on Linux systems simpler, as there are fewer differences among distributions. We enumerate **systemd** features next.

### systemd Features

Systems with **systemd** boot are faster than those with earlier init methods. This is largely because it replaces a serialized set of steps with aggressive parallelization techniques, which permits multiple services to be initiated simultaneously.

Complicated startup shell scripts are replaced with simpler configuration files, which enumerate what has to be done before a service is started, how to execute service startup, and what conditions the service should indicate have been accomplished when startup is finished.

One **systemd** command (**systemctl**) is used for most basic tasks. While we have not yet talked about working at the command line, here is a brief listing of its use:

- Starting, stopping, restarting a service (`fooservice` could be something like **nfsd** or the network) on a currently running system:
  ```
  $ sudo systemctl start/stop/restart fooservice
  ```

- Enabling or disabling a system service from starting up at system boot:
  ```
  $ sudo systemctl enable/disable fooservice
  ```

There are many other technical differences with older methods beyond the scope of our discussion.

# 3.Linux Filesystem basics

### Linux Filesystems

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 **filesystem**, which is the embodiment of a method of storing and organizing arbitrary collections of data in a human-usable form.

**Different Types of Filesystems Supported by Linux:**

- Conventional disk filesystems: **`ext2`**, **`ext3`**, **`ext4`**, **`XFS`**, **`Btrfs`**, **`JFS`**, **`NTFS`**, etc.
- Flash storage filesystems: **`ubifs`**, **`JFFS2`**, **`YAFFS`**, etc.
- Database filesystems
- Special purpose filesystems: **`procfs`**, **`sysfs`**, **`tmpfs`**, **`debugfs`**, etc.

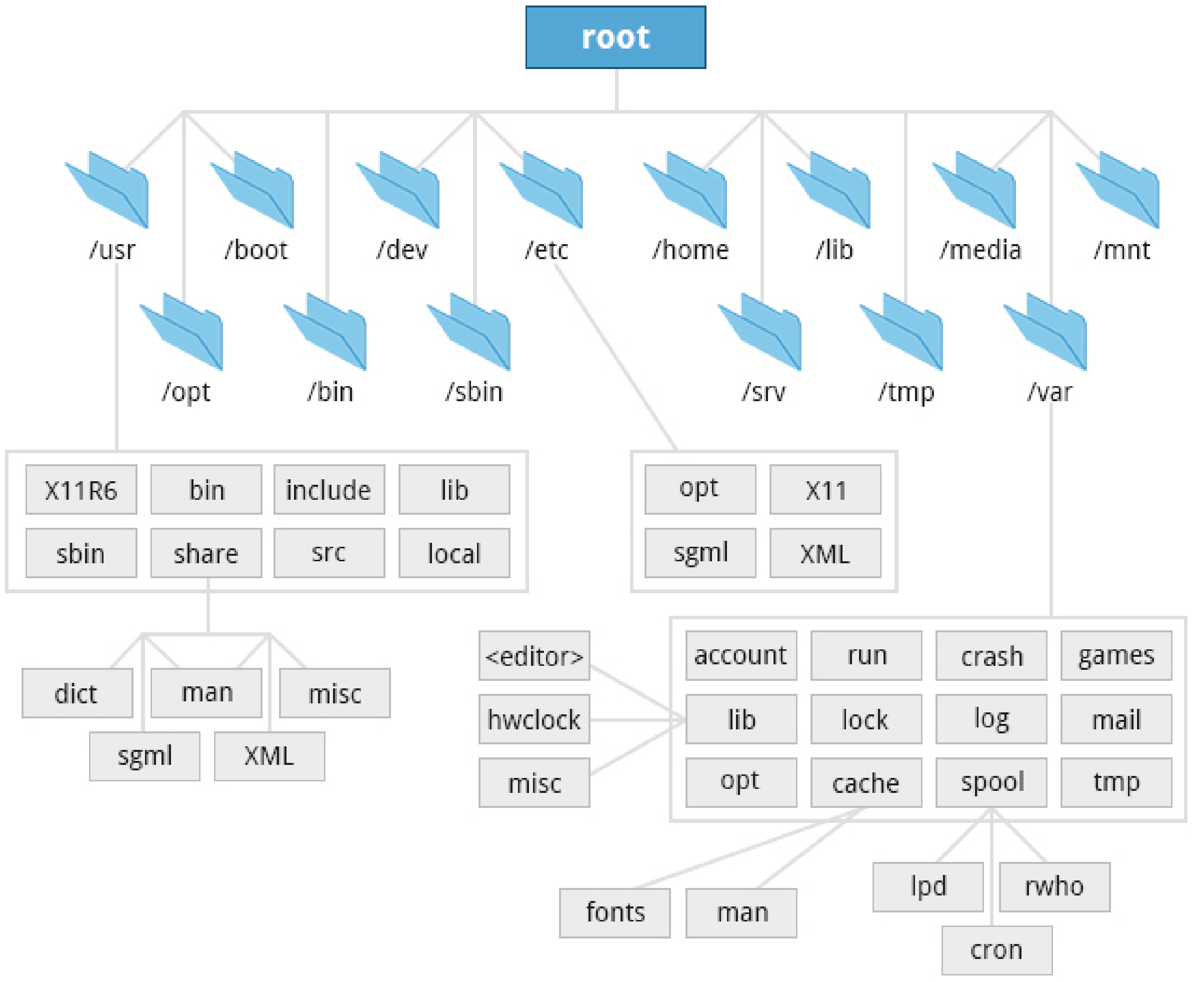
## Partitions and Filesystems

| | Windows | Linux |
|---|---|---|
| Partition | Disk1 | /dev/sda1 |
| Filesystem type | NTFS/VFAT | EXT3/EXT4/XFS/BTRFS... |
| Mounting Parameters | DriveLetter | MountPoint |
| Base Folder where OS is stored | C:\ | / |

A **partition** is a logical part of the disk, whereas a **filesystem** is a method of storing/finding files on a hard disk (usually in a partition). By way of analogy, you can think of filesystems 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 filesystems in Windows and Linux is given in the following table:

## The Filesystem Hierarchy Standard



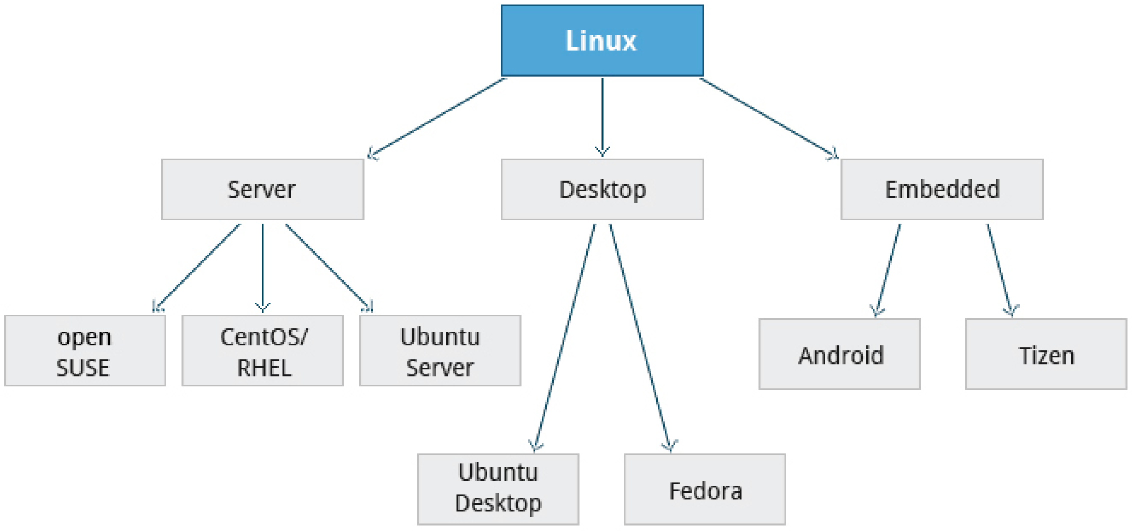
**Linux** systems store their important files according to a standard layout called the **Filesystem Hierarchy Standard (FHS)**, which has long been maintained by **The Linux Foundation**. You can read and/or download the official document that provides details from here. Having a standard is designed to ensure that users, administrators, and developers 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. Multiple drives and/or **partitions** are **mounted** as directories in the single filesystem. Removable media such as **USB** drives and **CD**s and **DVD**s will show up as mounted at `/run/media/yourusername/disklabel` for recent Linux systems, or under `/media` for older distributions. For example, if your username is `student`, a **USB** pen drive labeled **FEDORA** might end up being found at `/run/media/student/FEDORA`, and a file `README.txt` on that disc would be at `/run/media/student/FEDORA/README.txt`.

All Linux filesystem 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 (`/`).

# 4.Linux Distribution Installation

### Choosing a Linux Distribution



Suppose you intend to buy a new car. What factors do you need to consider to make a proper choice? Requirements which need to be taken into account include your expected budget, available financing options, the size needed to fit your family in the vehicle, the type of engine, after-sales services, etc.

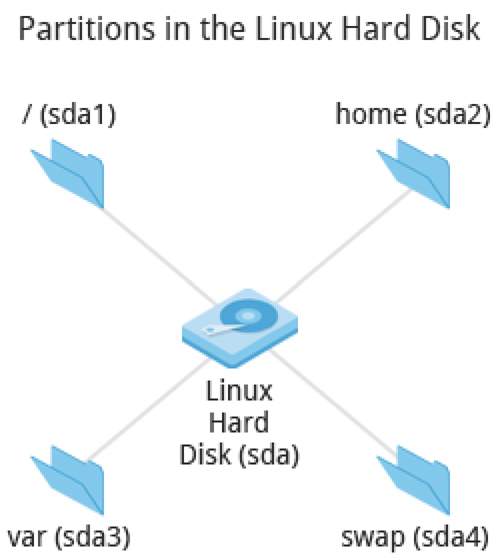
Similarly, determining which distribution to deploy also requires planning. The figure shows some, but not all choices, as there are other choices for distributions and standard embedded Linux systems are mostly neither **Android** nor **Tizen**, but slimmed down standard distributions.

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 filesystem. You can always modify the design later, but it is always easier to try and get it right to begin with.

Nearly all installers provide a reasonable filesystem layout by default, with either all space dedicated to normal files on one big partition and a smaller **swap** partition, 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.

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.**

Like other operating systems, Linux distributions are provided on removable media such as USB drives and CDs or DVDs. Most Linux distributions also 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.

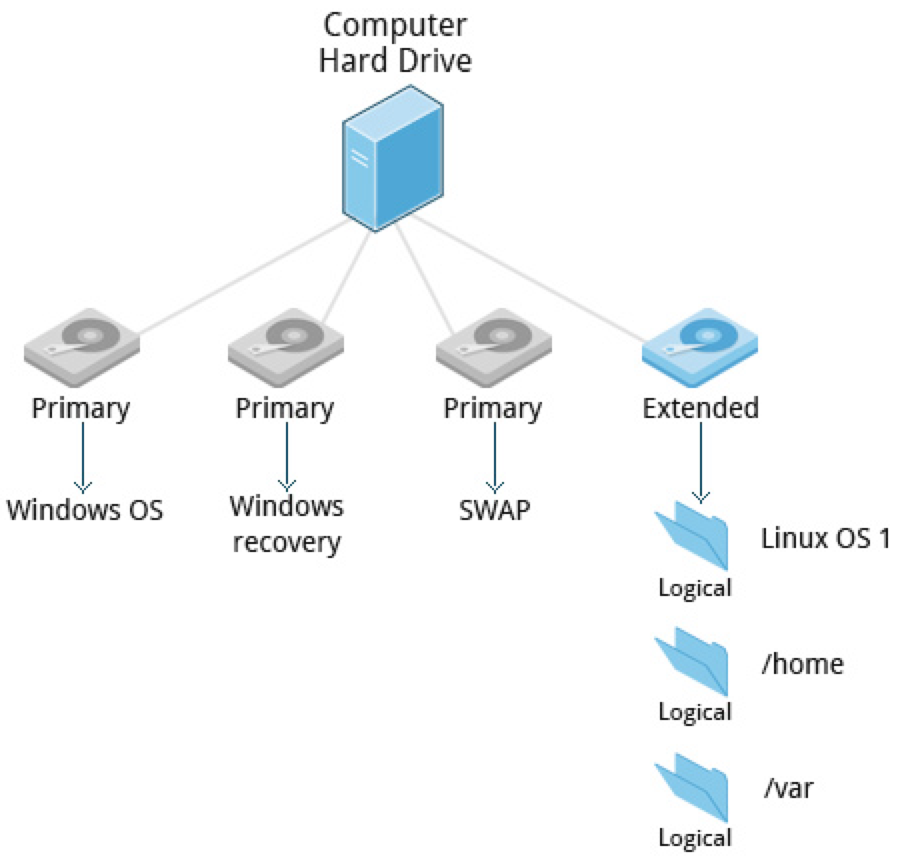
Each distribution provides its own documentation and tools for creating and managing these files.

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.



## 5.Summary

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

## COMMAND LINE OPERATIONS

# 1: Command Line Mode Options

## Some Basic Utilities

There are some basic command line utilities that are used constantly, and it would be impossible to proceed further without using some of them in simple form before we discuss them in more detail. A short list has to include:

- **cat:** used to type out a file (or combine files)
- **head:** used to show the first few lines of a file
- **tail:** used to show the last few lines of a file
- **man:** used to view documentation.

The screenshot shows elementary uses of these programs. Note the use of the **pipe** symbol (**|**) used to have one program take as input the output of another.

For the most part, we will only use these utilities in screenshots displaying various activities, before we discuss them in detail.

## 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.

## 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 will not 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 the initial installation.

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

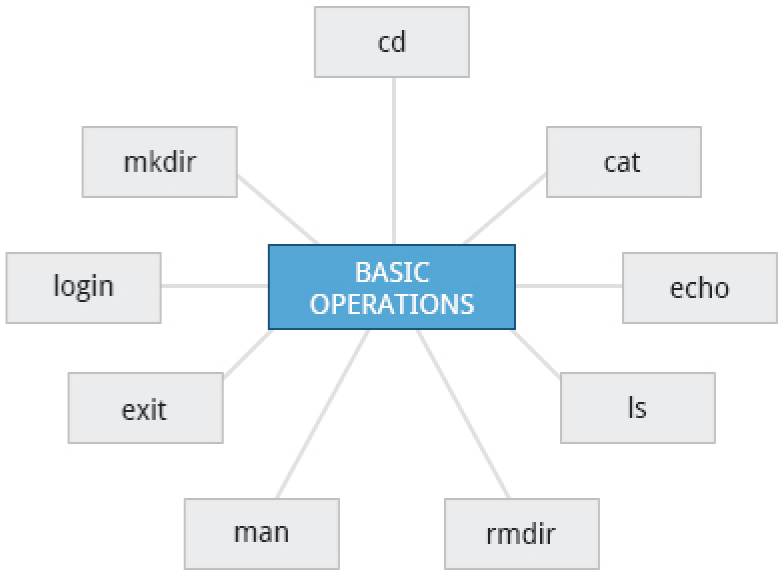
### Steps for Setting Up and Running sudo

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 superuser, root. While **sudo** will become the preferred method of doing this, we do not 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 do not 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.

# 2: Basic Operations



## 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 filesystem:

```
$ 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
```

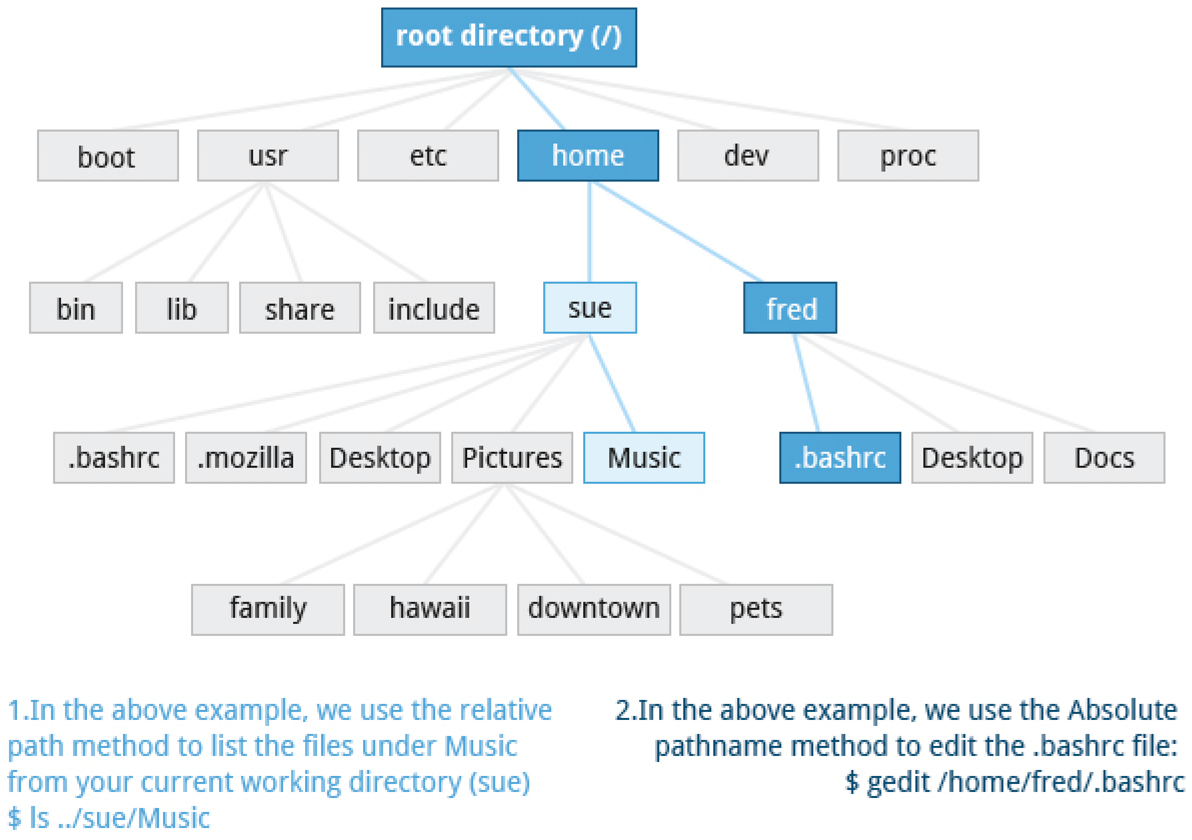
as well as locating **source** and **man** files packaged with the program.

## 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 $HOME`. (Many Linux distributions actually open new **graphical** terminals in `$HOME/Desktop`.) The following commands are useful for directory navigation:

| Command | Result |
|---|---|
| `pwd` | Displays the present working directory |
| `cd ~` or `cd` | 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



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

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 `/`.
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 FileSystem

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

The following commands can help in exploring the filesystem:

| Command | Usage |
|---|---|
| `cd /` | **C**hanges your current **d**irectory to the root (/) directory (or path you supply) |
| `ls` | **Li**st the contents of the present working directory |
| `ls -a` | **Li**st **all** files including **hidden** files and directories (those whose name start with . ) |
| `tree` | Displays a **tree** view of the filesystem |

## Hard File Links

The **ln** utility is 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.

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 **2** that appears in the **ls** output. Thus, there was already another object linked to `file1` before the command was executed.

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, 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.

## Soft (Symbolic) Links

**Soft** (or **Symbolic** links are created with the `-s` option as in:

```
$ ln -s file1 file3

$ ls -li file1 file3
```

Notice `file3` 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 filesystem (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 filesystems (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.

## 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.

# 3: Working with Files

Linux provides many commands that help you with 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.

## 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, starting with the last line. |
| **less** | Used to view larger files because it is a paging program; it pauses at each screen full 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. (An older program named **more** is still used, but has fewer capabilities.) |
| **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 `myfile` file'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 filesystem 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 $
```

# 4: Searching for Files

## 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**).

| Name | Symbolic Name | Value | Example |
|---|---|---|---|
| standard input | **stdin** | 0 | keyboard |
| standard output | **stdout** | 1 | terminal |
| standard error | **stderr** | 2 | log file |

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.

## I/O Redirection

Through the command **shell** we can **redirect** the three standard file streams 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.

We will now 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 the 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**.

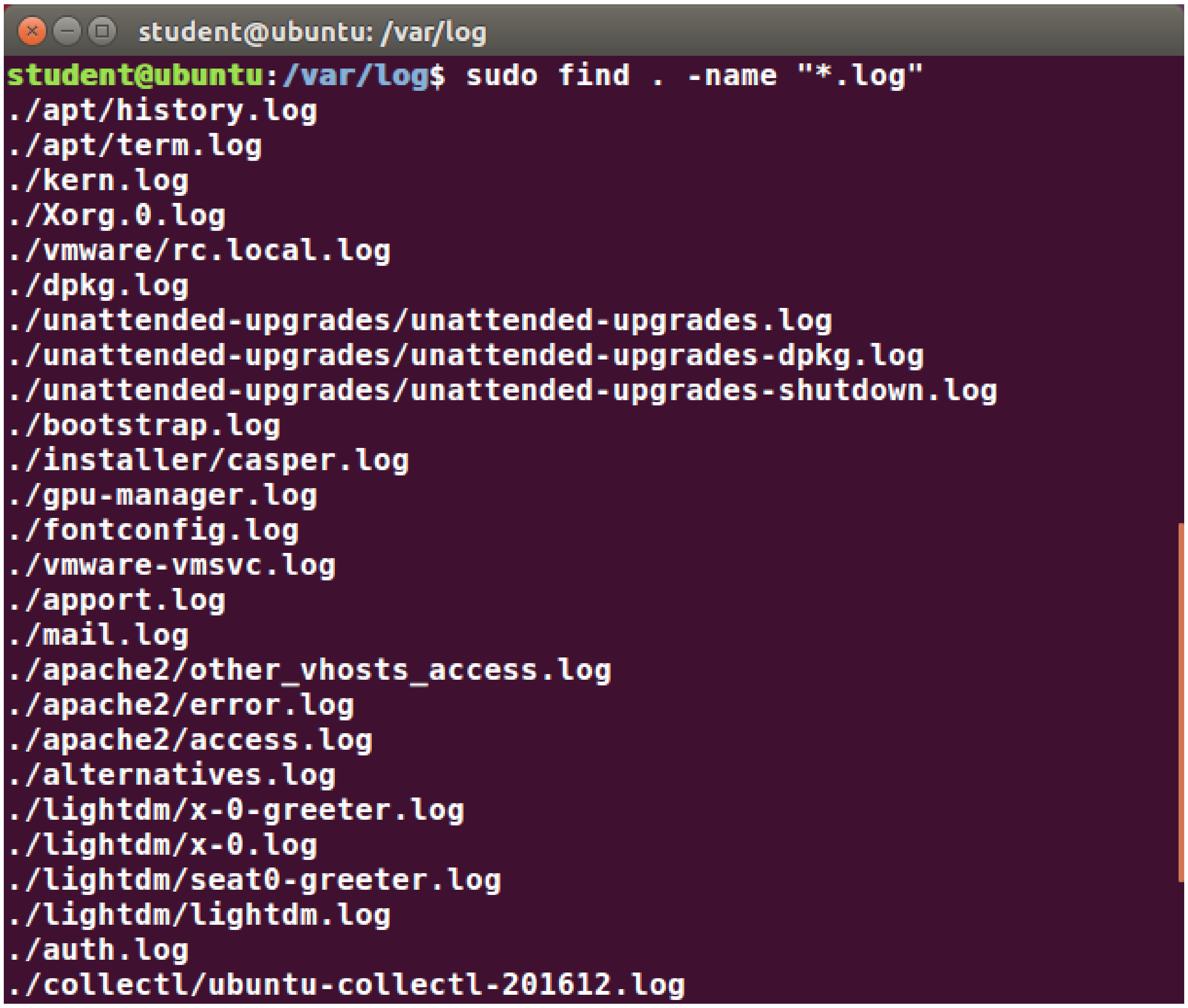
| Wildcard | Result |
|---|---|
| `?` | Matches any single character |
| `*` | Matches any string of characters |
| `[set]` | Matches any character in the set of characters, for example [adf] will match any occurrence of "a", "d", or "f" |

| | |
|---|---|
| `[!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



student@ubuntu: /var/log

```
student@ubuntu:/var/log$ sudo find . -name "*.log"
./apt/history.log
./apt/term.log
./kern.log
./Xorg.0.log
./vmware/rc.local.log
./dpkg.log
./unattended-upgrades/unattended-upgrades.log
./unattended-upgrades/unattended-upgrades-dpkg.log
./unattended-upgrades/unattended-upgrades-shutdown.log
./bootstrap.log
./installer/casper.log
./gpu-manager.log
./fontconfig.log
./vmware-vmsvc.log
./apport.log
./mail.log
./apache2/other_vhosts_access.log
./apache2/error.log
./apache2/access.log
./alternatives.log
./lightdm/x-0-greeter.log
./lightdm/x-0.log
./lightdm/seat0-greeter.log
./lightdm/lightdm.log
./auth.log
./collectl/ubuntu-collectl-201612.log
```

**find** is an extremely useful and often-used utility program in the daily life of a Linux system administrator. It recurses down the filesystem 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.

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 "gcc":

```
$ find /usr -type f -name gcc
```

## 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.

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.

Please 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.

To find files based on time:

```
$ find / -ctime 3
```

Here, **`-ctime`** is when the inode metadata (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`**).

To find files 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 {} ’;’
```

# 5: Installing Software

## 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. There are some other systems used by more specialized Linux distributions.

## 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.

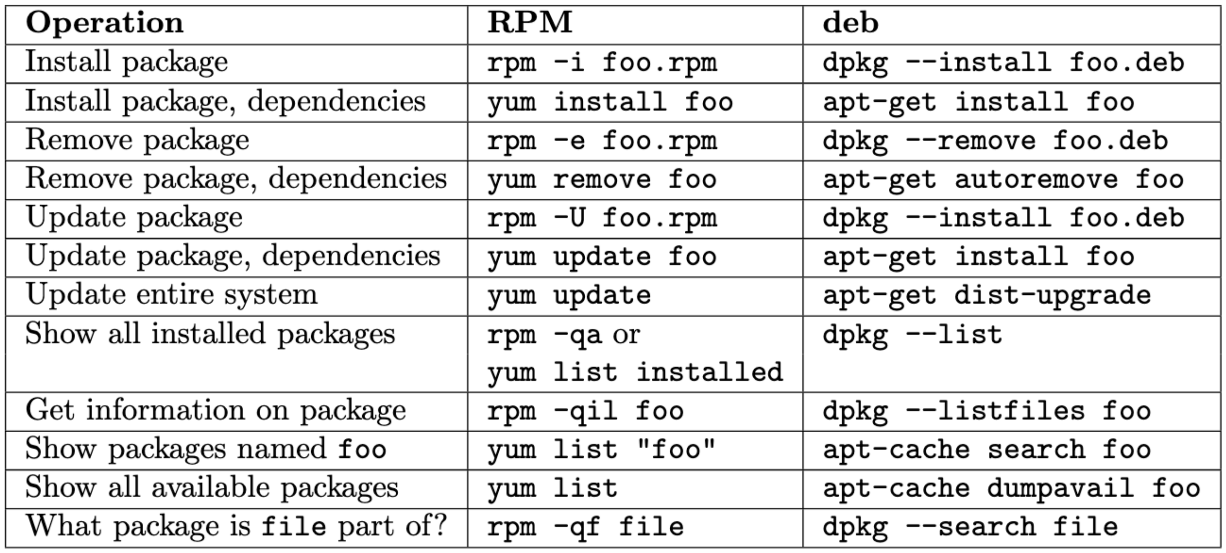
## Working With Different Package Management Systems

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. Recent **Fedora** versions have replaced **yum** with a new utility called **dnf**, which has less historical baggage, has nice new capabilities and is mostly backwards compatible with **yum** for day-to-day commands.

**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:



| Operation | RPM | deb |
|---|---|---|
| Install package | `rpm -i foo.rpm` | `dpkg --install foo.deb` |
| Install package, dependencies | `yum install foo` | `apt-get install foo` |
| Remove package | `rpm -e foo.rpm` | `dpkg --remove foo.deb` |
| Remove package, dependencies | `yum remove foo` | `apt-get autoremove foo` |
| Update package | `rpm -U foo.rpm` | `dpkg --install foo.deb` |
| Update package, dependencies | `yum update foo` | `apt-get install foo` |
| Update entire system | `yum update` | `apt-get dist-upgrade` |
| Show all installed packages | `rpm -qa` or `yum list installed` | `dpkg --list` |
| Get information on package | `rpm -qil foo` | `dpkg --listfiles foo` |
| Show packages named `foo` | `yum list "foo"` | `apt-cache search foo` |
| Show all available packages | `yum list` | `apt-cache dumpavail foo` |
| What package is `file` part of? | `rpm -qf file` | `dpkg --search file` |

## 5:Summary

- 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.

## FINDING LINUX DOCUMENTATION

Whether you are an inexperienced user or a veteran, you will not always know (or remember) the proper use of various Linux programs and utilities: what is the command to type, what options does it take, etc. You will need to consult help documentation regularly. Because Linux-based systems draw from a large variety of sources, there are numerous reservoirs of documentation and ways of getting help. Distributors consolidate this material and present it in a comprehensive and easy-to-use manner.

Important Linux documentation sources include:

- The **man pages** (short for manual pages)
- GNU **Info**
- The **help** command and **`--help`** option
- Other Documentation Sources, e.g. https://www.gentoo.org/doc/en/ or https://help.ubuntu.com/community/CommunityHelpWiki

## The man pages

The **man pages** are the most often-used source of Linux documentation. They provide in-depth documentation about many programs and utilities, as well as other topics, including configuration files, system calls, library routines, and the kernel. They are present on all Linux distributions and are always at your fingertips. (**man** is actually an abbreviation for **manual**.)

Typing **man** with a topic name as an argument retrieves the information stored in the topic's **man pages**. Some Linux distributions require every installed program to have a corresponding **man** page, which explains the depth of coverage. The **man** pages infrastructure was first introduced in the early UNIX versions of the early 1970s.

**man** pages are often converted to:

- Web pages (See http://man7.org/linux/man-pages/ )

- Published books
- Graphical help
- Other formats.

## man

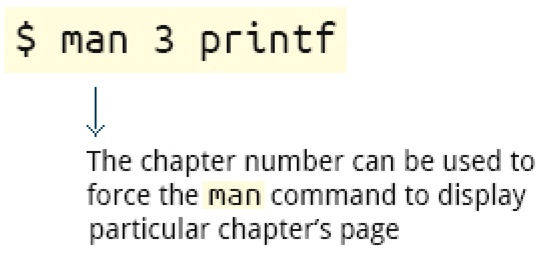
The **man** program searches, formats, and displays the information contained in the **man** pages. Because many topics have a lot of information, output is piped through a **pager** program such as **less** to be viewed one page at a time; at the same time, the information is formatted for a good visual display.

When no options are given, by default one sees only the dedicated page specifically about the topic. You can broaden this to view all pages containing a string in their name by using the `-f` option. You can also view all pages that discuss a specified subject (even if the specified subject is not present in the name) by using the `-k` option.

`man -f` generates the same result as typing **whatis**.

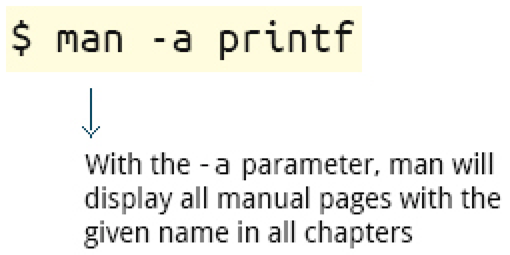
`man -k` generates the same result as typing **apropos.**

## Manual Chapters



```
$ man 3 printf
```

The chapter number can be used to force the man command to display particular chapter's page



```
$ man -a printf
```

With the - a parameter, man will display all manual pages with the given name in all chapters

The **man pages** are divided into nine numbered chapters (1 through 9). Sometimes, a letter is appended to the chapter number to identify a specific topic. For example, many pages describing part of the **X Window** API are in chapter 3X.

The chapter number can be used to force **man** to display the page from a particular chapter; it is common to have multiple pages across multiple chapters with the same name, especially for names of library functions or system calls.

With the `-a` parameter, **man** will display all pages with the given name in all chapters, one after the other.

```
$ man 3 printf
```

```
$ man -a printf
```

## GNU Info System

The next source of Linux documentation is the **GNU Info System**.

This is the **GNU** project's standard documentation format (**info**) which it prefers as an alternative to **man**. The **info** system is more free-form and supports linked sub-sections.

Functionally, the **GNU Info System** resembles **man** in many ways. However, topics are connected using links (even though its design predates the World Wide Web). Information can be viewed through either a command line interface, a graphical help utility, printed or viewed online.

## Command Line Info Browser

Typing **info** with no arguments in a terminal window displays an index of available topics. You can browse through the topic list using the regular movement keys: **arrows**, **Page Up**, and **Page Down**.

You can view help for a particular topic by typing `info <topic name>`. The  system then searches for the topic in all available **info** files.

Some useful keys are: `q` to quit, `h` for help, and `Enter` to select a menu item.

## info Page Structure

The topic which you view in the **info** page is called a **node**.

Nodes are similar to sections and subsections in written documentation. You can move between nodes or view each node sequentially. Each node may contain **menus** and linked subtopics, or **items**.

Items can be compared to Internet hyperlinks. They are identified by an asterisk (`*`) at the beginning of the item name. Named items (outside a menu) are identified with double-colons (`::`) at the end of the item name. Items can refer to other nodes within the file or to other files. The table lists the basic keystrokes for moving between nodes.

| Key | Function |
|---|---|
| `n` | Go to the next node |
| `p` | Go to the previous node |
| `u` | Move one node up in the index |

## Introduction to the --help Option

Another important source of Linux documentation is use of the `--help` option.

Most commands have an available short description which can be viewed using the `--help` or the `-h` option along with the command or application. For example, to learn more about the **man** command, you can run the following command:

```
$ man --help
```

The `--help` option is useful as a quick reference and it displays information faster than the **man** or **info** pages.

Some popular commands (such as **echo**), when run in a **bash** command shell, silently run their own built-in versions of system programs or utilities, because it is more efficient to do so. (We will discuss command shells in great detail later.) To view a synopsis of these built-in commands, you can simply type **help**.

For these built-in commands, **help** performs the same basic function as the `-h` and `--help` arguments (which we will discuss shortly) perform for stand-alone programs.

The next screen covers a demonstration on how to use **man**, **info**, and the **help** option.

## Package Documentation

Linux documentation is also available as part of the package management system. Usually, this documentation is directly pulled from the upstream source code, but it can also contain information about how the distribution packaged and set up the software.

Such information is placed under the `/usr/share/doc` directory in a subdirectory named after the package, perhaps including the version number in the name.

## Online Resources

There are many places to access online Linux documentation, and a little bit of searching will get you buried in it. The following site has been well reviewed by other users of this course and offers a free, downloadable command line compendium under a Creative Commons license:

**LinuxCommand.org:** http://linuxcommand.org/tlcl.php

You can also find very helpful documentation for each distribution. Each distribution has its own user-generated forums and wiki sections. Here are just a few links to such sources:

**Ubuntu:** https://help.ubuntu.com/

**CentOS:** https://www.centos.org/docs/

**OpenSUSE:** http://en.opensuse.org/Portal:Documentation

**GENTOO:** http://www.gentoo.org/doc/en

Moreover, you can use online search sites to locate helpful resources from all over the Internet, including blog posts, forum and mailing list posts, news articles, and so on.

## Summary

- The main sources of Linux documentation are the **man pages**, **GNU Info**, the **help** options and command, and a rich variety of online documentation sources.
- The **man** utility searches, formats, and displays **man pages**.
- The **man pages** provide in-depth documentation about programs and other topics about the system, including configuration files, system calls, library routines, and the kernel.
- The **GNU Info System** was created by the **GNU** project as its standard documentation. It is robust and is accessible via command line, web, and graphical tools using **info**.

- Short descriptions for commands are usually displayed with the `-h` or `--help` argument.
- You can type **help** at the command line to display a synopsis of built-in commands.
- There are many other help resources both on your system and on the Internet.