LINUX – INTRODUCTION - edX

OVERVIEW

Ch 1 – The Linux Foundation

Ch 2 – Linux Philosophy and Concepts

Ch 3 – Linux Basics and System Startup

Ch 4 – Graphical Interface

Ch 5 – System Configuration from the Graphical Interface

Ch 6 – Common Applications

Ch 7 – Command Line Operations

Ch 8 – Finding Linux Documentaton

Ch 9 - Processes

Ch 10 – File Operations

Ch 11 – Text Editors

Ch 12 – User Environment

Ch 13 – Manipulating Text

Ch 14 – Network Operations

Ch 15 – The Bash Shell and Basic Scripting

Ch 16 – More on Bash Shell Scripting

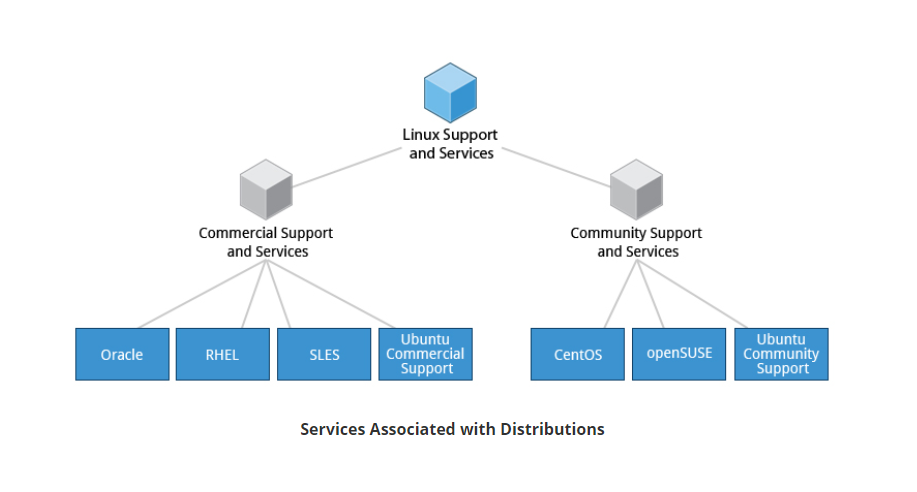
Ch 17 - Printing

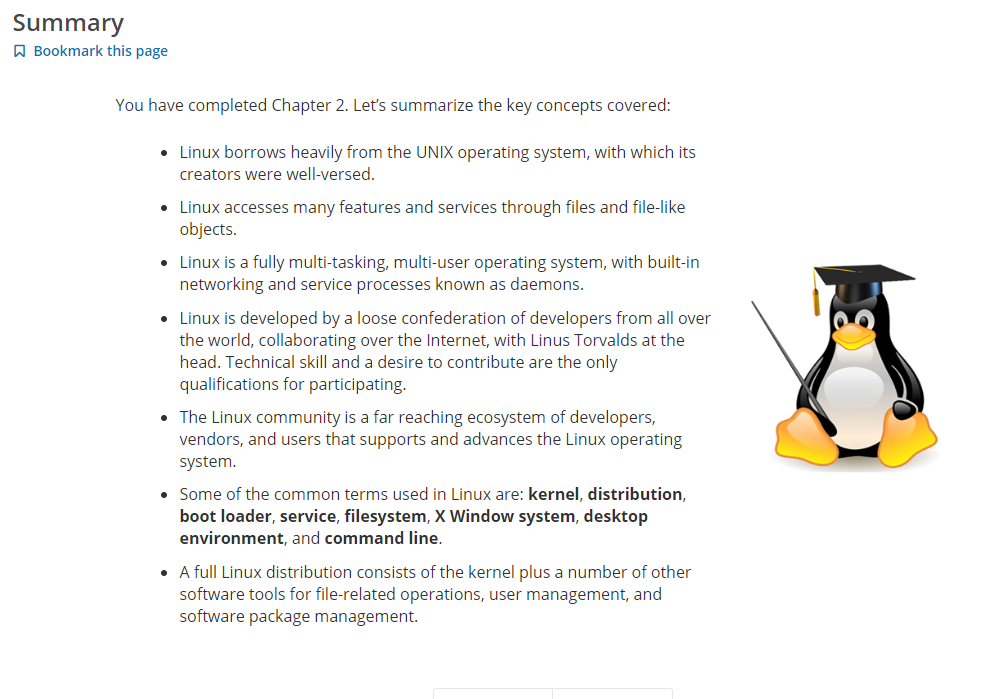
Ch 18 – Local Security Principles

**Ch 1 & 2: Linux Foundation, Philosophy and Concepts**

The Linux community is a far-reaching ecosystem consisting of developers, system administrators, users and vendors who use many different forums to connect with one another. Among the most popular are:

* Internet Relay Chat (IRC) software (such as WeeChat, HexChat, Pidgin and XChat)
* Online communities and discussion boards including Linux User Groups (both local and online)
* Many collaborative projects hosted on services such as GitHub
* Newsgroups and mailing lists, including the Linux Kernel Mailing List
* Community events, e.g. Hackathons, Install Fests, Open Source Summits and Embedded Linux Conferences.





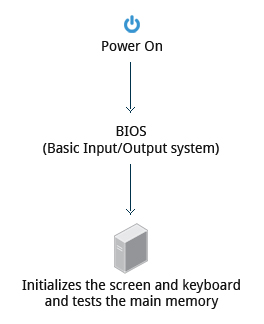
**Ch 3 – Linux Basics and System Startup**



**\*\* 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).

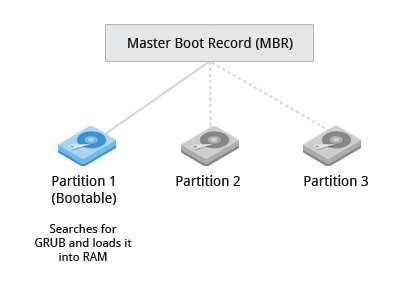
The BIOS software is stored on a ROM chip on the motherboard. After this, the remainder of the boot process is controlled by the operating system (OS).

  
**BIOS**

**\*\* MASTER BOOT RECORD (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 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 GRand Unified Boot loader), 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.



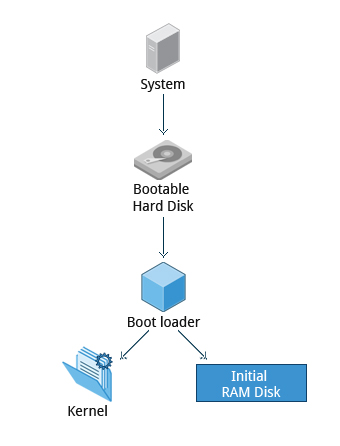
**Master Boot Record**

**\*\* Boot Loader in Action**

The boot loader has two distinct stages:

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, for example 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.

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.



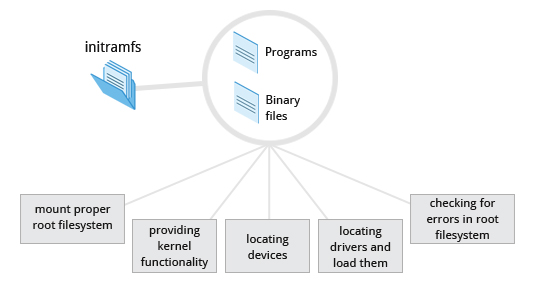
**Boot Loader in Action**

**\*\* Initial RAM Disk**

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 user device), which is responsible for figuring out which devices are present, locating the device 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 initramfsis cleared from RAM and the init program on the root filesystem (**/sbin/init**) is executed.

inithandles 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.



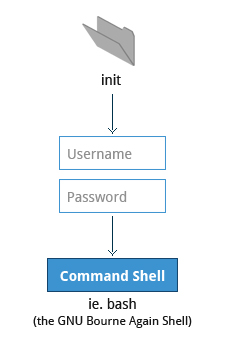
**The Initial RAM Disk**

**\*\* 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 *Chapter 7: 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**. 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).

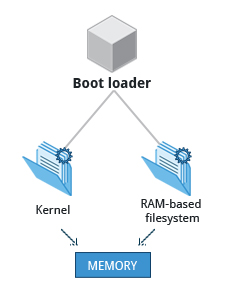
Usually, the default command shell is bash (the GNU Bourne Again Shell), but there are a number of other advanced command shells available. The shell prints a text prompt, indicating it is ready to accept commands; after the user types the command and presses **Enter**, the command is executed, and another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

  
**Text-Mode Logins**

**\*\* The Linux Kernel**

The boot loader loads both the kernel and an initial RAM–based file system (initramfs) into memory, so itcan 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 Linux Kernel**

**\*\* /sbin/init and Services**

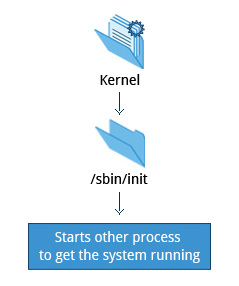
Once the kernel has set up all its hardware and mounted the root filesystem, the kernel runs **/sbin/init**. This then becomes the initial process, which then starts other processes to get the system running. Most other processes on the system trace their origin ultimately to init; exceptions include the so-called kernel processes. These are started by the kernel directly, and their job is to manage internal operating system details.

Besides starting the system, initis responsible for keeping the system running and for shutting it down cleanly. One of its responsibilities is to act when necessary as a 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 the 1980s and the System V variety of UNIX. This serial process has the system passing through a sequence of runlevelscontaining 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 newer methods, systemd and Upstart.

  
**/sbin/init and Services**

**Startup Alternatives**

SysVinitviewed 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 true, especially with mobile devices and embedded Linux systems. Some modern methods, such as the use of containers, can require almost instantaneous startup times. Thus, systems now require methods with faster and 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 was rather controversial, it has been adopted by the major distributions, and so we will not discuss the older System V method orUpstart, 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 start up 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 thing to note is that **/sbin/init**now just points to **/lib/systemd/systemd**; i.e. systemd takes over the init process.

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 (using nfsas an example) on a currently running system:  
      **$ sudo systemctl start|stop|restart nfs.service**
    - Enabling or disabling a system service from starting up at system boot:  
      **$ sudo systemctl enable|disable nfs.service**

In most cases, the **.service**can be omitted. There are many technical differences with older methods that lie beyond the scope of our discussion.

**Linux Filesystems**

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**,**squashfs**,**debugfs**, etc.

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

A partition is a physically contiguous section of a disk, or what appears to be so in some advanced setups.

A filesystem is a method of storing/finding files on a hard disk (usually in a partition).

One can think of a partition as a container in which a filesystem resides, although in some circumstances, a filesystem can span more than one partition if one uses symbolic links, which we will discuss much later.

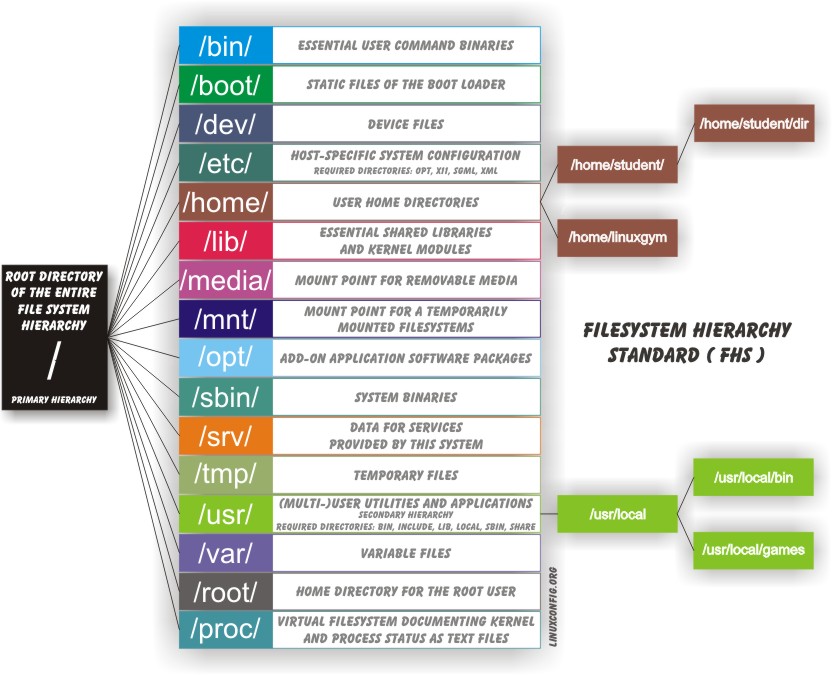
A comparison between filesystems in Windows and Linux is given in the accompanying table:

|  |  |  |
| --- | --- | --- |
|  | **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:\ | / |

**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. For more information, take a look at the following document: "[*Filesystem Hierarchy Standard*](https://refspecs.linuxfoundation.org/FHS_3.0/fhs-3.0.pdf)" created by LSB Workgroup. 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 CDs and DVDs 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 USBpen 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**.

[](https://prod-edxapp.edx-cdn.org/assets/courseware/v1/66def40e2774fd96011565107706da2d/asset-v1:LinuxFoundationX+LFS101x+3T2018+type@asset+block/dirtree.jpg)

**The Filesystem Hierarchy Standard**

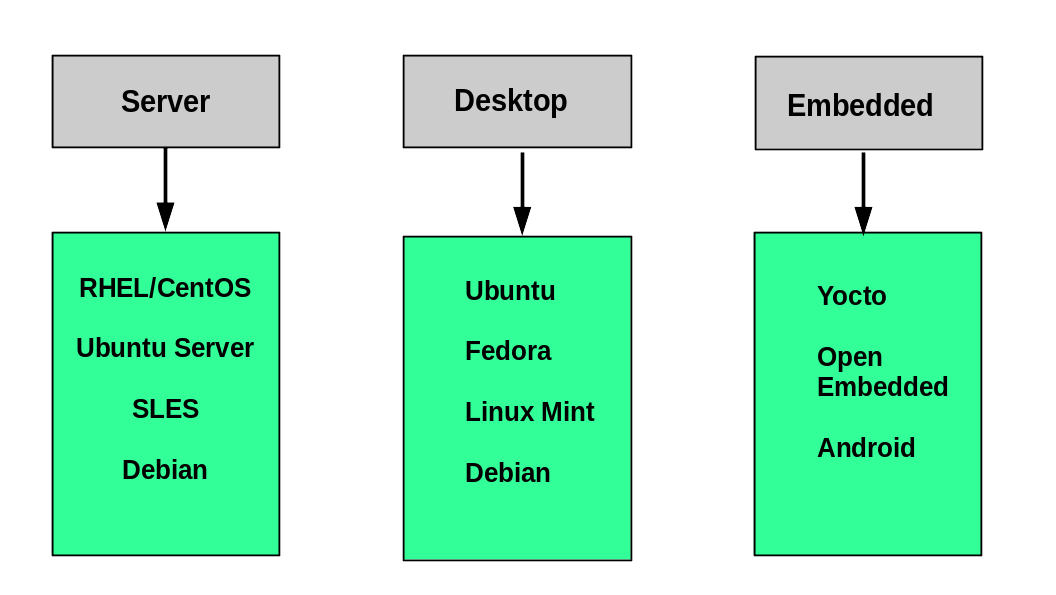
**More About the Filesystem Hierarchy Standard**

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 from the previous page and compare the subdirectories with those that exist directly under the system root directory (**/**).

**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 the size needed to fit your family in the vehicle, the type of engine and gas economy, your expected budget and available financing options, reliability record and after-sales services, etc.

Similarly, determining which distribution to deploy also requires planning. The figure shows some, but not all choices. Note that many embedded Linux systems use custom crafted contents, rather than Android or Yocto.



**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 required and how much is available? For example, when installing Linux on an embedded device, space is usually constrained.
    - 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 or a third party?
    - What hardware are you running on? For example, it might be X86, ARM, PPC, etc.
    - Do you need long-term stability? Can you accept (or need) a more volatile cutting edge system running the latest software?

**Linux Installation: Install Source**

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, in which case it is possible to perform an install without using any local media.

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

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

**Steps to Install Ubuntu (video)**

The Linux Foundation has a document: "[*Preparing Your Computer for LFS101x.2*](https://prod-edxapp.edx-cdn.org/assets/courseware/v1/3fa6f8f7a7482a6344efeb7dd0d5bdf0/asset-v1:LinuxFoundationX+LFS101x+3T2018+type@asset+block/Preparing_Your_Computer_for_LFS101x.pdf)" that describes alternate methods of installing Linux without over-writing existing data. You may want to consult it, if you need to preserve the information on your hard disk.

These alternate methods are:

1. Re-partitioning your hard disk to free up enough room to permit dual boot (side-by-side) installation of Linux, along with your present operating system.
2. Using a host machine hypervisor program (such as VMWare's products or Oracle Virtual Box) to install a client Linux Virtual Machine.
3. Booting off of and using a Live CD or USB stick and not writing to the hard disk at all.

**Steps to Install CentOS (video)**

**Steps to Install openSUSE (video)**

**You should now be able to:**

* ***Identify Linux filesystems.***
* ***Identify the differences between partitions and filesystems.***
* ***Describe the boot process.***
* ***Install Linux on a computer.***

**Ch 3 - 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 initramfsfilesystem 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.

**Ch 4 – Graphical Interface**

By the end of this chapter, you should be able to:

* Manage graphical interface sessions.
* Perform basic operations using the graphical interface.
* Change the graphical desktop to suit your needs.

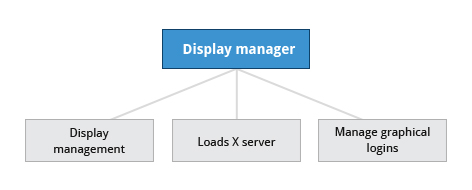
**Ch 4 – Graphical Desktop Interface**

* You can use either a Command Line Interface (CLI) or a Graphical User Interface (GUI) when using Linux. To work at the CLI, you have to remember which programs and commands are used to perform tasks, and how to quickly and accurately obtain more information about their use and options. On the other hand, using the GUI is often quick and easy. It allows you to interact with your system through graphical icons and screens. For repetitive tasks, the CLI is often more efficient, while the GUI is easier to navigate if you do not remember all the details or do something only rarely.
* We will learn how to manage sessions using the GUI for the three Linux distribution families that we cover the most in this course: Red Hat (CentOS, Fedora), SUSE (openSUSE), and Debian (Ubuntu, Mint). Since we are using the GNOME-based variant of openSUSE rather than the KDE-based one, all are actually quite similar. If you are using KDE (or other Linux desktops such as XFCE), your experience will vary somewhat from what is shown, but not in any intrinsically difficult way, as user interfaces have converged to certain well-known behaviors on modern operating systems. In subsequent sections of this course we will concentrate in great detail on the command line interface, which is pretty much the same on all distributions.

**X Window System**

Generally, in a Linux desktop system, the X Window System is loaded as one of the final steps 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 Window System is a rather old system; it dates back to the mid 1980s and, as such, has certain deficiencies on modern systems (for example, with security), as it has been stretched rather far from its original purposes. A newer system, known as [Wayland](https://wayland.freedesktop.org/), is gradually superseding it and was adopted as the default display system in Fedora 25.

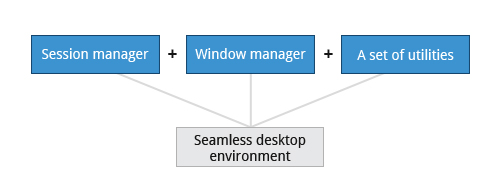


* **Display Manager**

**More About X**

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. Or, you can start the display manager (gdm, lightdm, kdm, xdm, etc.) manually from the command line. This differs from running **startx** as the display managers will project a sign in screen. We discuss them next.

* **Desktop Environment**

**GUI Startup**

When you install a desktop environment, the X display manager starts at the end of the boot process. It is responsible for starting the graphics system, logging in the user, and starting the user’s desktop environment. You can often select from a choice of desktop environments when logging in to the system.

The default display manager for GNOME is called gdm. Other popular display managers include lightdm (used on Ubuntu before version 18.04 LTS) and kdm (associated with KDE).

**GNOME Desktop Environment**

GNOME is a popular desktop environment with an easy-to-use graphical user interface. It is bundled as the default desktop environment for most Linux distributions, including Red Hat Enterprise Linux, Fedora, CentOS, SUSE Linux Enterprise, Ubuntu and Debian. GNOME has menu-based navigation and is sometimes an easy transition to accomplish for Windows users. However, as you will see, the look and feel can be quite different across distributions, even if they are all using GNOME.

Another common desktop environment very important in the history of Linux and also widely used is KDE, which has often been used in conjunction with SUSE and openSUSE. Other alternatives for a desktop environment include Unity (from on older Ubuntu, but still based on GNOME), XFCE and LXDE. As previously mentioned, most desktop environments follow a similar structure to GNOME, and we will restrict ourselves mostly to it to keep things less complex.

**Graphical Desktop Background**

Each Linux distribution comes with its own set of desktop backgrounds. You can change the default by choosing a new wallpaper or selecting a custom picture to be set as the desktop background. If you do not want to use an image as the background, you can select a color to be displayed on the desktop instead.

In addition, you can also change the desktop theme, which changes the look and feel of the Linux system. The theme also defines the appearance of application windows.

We will learn how to change the desktop background and theme.

**Lab: Recovering Deleted Files**

1. Open the file manager and navigate to your home directory. Once your user configuration is set up for it this is as simple as right clicking in the directory, and selecting **Create New->Text File** and giving it a name. (You may see something other than **Text File** With the **GNOME** file manager, there is no such option in the default setup. You must create a file called **new** in your **Templates** directory first. The easiest way to do this is to type **Alt-F2** and then in the little window that opens up

**student:/tmp> touch ~/Templates/new**

Once you have done that, you can either right click in the directory, or somewhere on the right side of the title bar to open up a dialog where there will be an option to create a new file of some type.

Why this feature is not on by default might be considered a mystery not understood by modern science. However, the logic is that new files are generally not created in a vaccuum (at least in GUIs); one expects to make them in an application. Some distributions or versions of **GNOME** may not need this stupid step.

1. This should be pretty obvious, either by dragging and dropping in the **Trash** folder on the left, or through a right click on the file.
2. To navigate down through your **.local** directory, you have to make hidden files visible. On **GNOME** systems, you can either just hit **CTRL-H**, or you can click on the little down arrow next to the gear icon and toggle showing hidden files on and off.
3. The easiest way is to click on the **Trash** icon and then pick **Restore**, which will bring it back to its original location which you can easily verify.

Note that the **GNOME** File Manager does not include a **Delete** or **Permanenly Delete** option by default. To enable this you have to go into preferences and turn it on. You can enter the preferences menu by starting the File Manager and on the top task bar on the desktop, click on its icon and get into the preference menus. Another non-obvious step :( .

**You should now be able to:**

* Manage graphical interface sessions.
* Perform basic operations using the graphical interface.
* Change the graphical desktop to suit your needs.

**Ch 4 - Summary**

* **GNOME** is a popular desktop environment and graphical user interface that runs on top of the Linux operating system.
* The default display manager for GNOME is called **gdm**.
* The gdm display manager presents the user with the login screen, which prompts for the login username and password.
* Logging out through the desktop environment kills all processes in your current **X** session and returns to the display manager login screen.
* Linux enables users to switch between logged-in sessions.
* Suspending puts the computer into sleep mode.
* For each key task, there is generally a default application installed.
* Every user created in the system will have a **home** directory.
* The *Places* menu contains entries that allow you to access different parts of the computer and the network.
* Most text editors are located in the *Accessories* submenu.
* Each Linux distribution comes with its own set of desktop backgrounds.
* GNOME comes with a set of different themes which can change the way your applications look.

**Ch 5 - Overview**

* Apply system, display, and date and time settings using the *System Settings* panel.
* Track the network settings and manage connections using *Network Manager* in Linux.
* Install and update software in Linux from a graphical interface.

***Note***: We will revisit all these tasks later, when we discuss how to accomplish them from the command line interface.

**gnome-tweak-tool**

A lot of personalized configuration settings do not appear on the settings menus we have discussed. Instead, you have to launch a tool called gnome-tweak-tool, or gnome-tweaks on very recent Linux distributions. We have not really discussed working at the command line yet, but you can always launch a program such as this by doing **Alt-F2** and typing in the command. There are distributions which have a link to the tweaks menus in the settings, but mostly for some mysterious reason, they obscure this indispensable tool's existence.

Important things you can do with this tool include selecting a theme, configuring extensions which you can get from your distribution or download from the Internet, control fonts, and set which programs start when you login.

The screenshot here is from a Red Hat 7 system with quite a few extensions installed, but not all being used.

Let's get started on managing system and display settings in OpenSUSE.

**To change the screen resolution to 1024 x 768 in OpenSUSE:**

1. On the OpenSUSE desktop, at the top left corner of screen click Activities.

2. Type settings in the Search box. The Settings icon appears.

3. Click the Settings icon.

The All Settings window appears.

4. Click the Displays icon.

The Displays window appears.

5. Select a display ID to set the resolution.

6. From the Resolution drop-down list, select 1024 x 768.

7. Click Apply.

A confirmation message is displayed.

8. Click Keep Changes.

**To change screen resolution to 1024 x 768 in Ubuntu:**

1. On the left panel, click the System Settings icon.

The System Settings window is displayed.

2. Click the Displays icon.

The Displays dialog box is displayed.

3. From the Resolution drop-down list, select 1024 x 768.

4. Click the Apply button.

A confirmation message is displayed.

5. Click Keep This Configuration.

Find out the current screen resolution for your desktop.

Change it to something else, and change it back to its original value.

***Note*:** You can also ascertain your current resolution by typing at the command line:

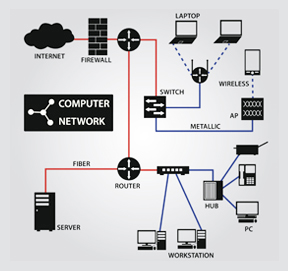
**student:/tmp> $ xdpyinfo | grep dim**  
**dimensions: 3200x1080 pixels (847x286 millimeters)**

**Wired and Wireless Connections**

Wired connections usually do not require complicated or manual configuration. The hardware interface and signal presence are automatically detected, and then *Network Manager* sets the actual network settings via Dynamic Host Configuration Protocol (DHCP).

For *static* configurations that do not use DHCP, manual setup can also be done easily through *Network Manager*. You can also change the Ethernet Media Access Control (MAC) address if your hardware supports it. The MAC address is a unique hexadecimal number of your network card.

Wireless networks are not connected to the machine by default. You can view the list of available wireless networks and see which one you are connected to by using *Network Manager*.You can then add, edit, or remove known wireless networks, and also specify which ones you want connected by default when present.



Let's demonstrate how to control wired and wireless network connections on a Red Hat 7 system.

If I click on the upper right hand corner, for instance, if I want to look at the wired connection, I can just scroll down to the Wired Connection,

and then Wired Settings, and it asks me for my password; the root password, because it wants to have a super user only control network connections.

and you'll see I have 1,000 megabit connection.

If I click on the gear icon, I can control the Settings. On the Details, first screen, it says it connects automatically when the system starts and all users can use it.

On Identity, I give what the name of the interface looks like, not the name of the system. In older systems, that would have been eth0, eth1, etc.,

but as we will discuss later, newer systems have these more complex names.

Under IPv4, we control the basic internet protocols system settings. Most of the time, you'll leave it alone at Automatic (DHCP), dynamic host configuration protocol,

where the system figures out its address from the server etc.

If you understand the importance of manual settings, you can specify an address and netmask, a gateway,

you can select a particular DNS server and you can specify routes to certain addresses.

If you understand those concepts, this is where you would go to control all that.

We're not changing anything, so I'll just hit Cancel.

Now, if I want to control Wi-Fi settings, I could go back to that upper right corner, or I can just click on the Wi-Fi setting here, and once again, it's asking me for the super user password.

These are the different wireless networks that have to be... that are available at the moment. They all have locks next to them, means they require a password.

This is the one that I'm actually on, coopj, the first one.

If I click on the settings there, once again it's going to ask me for my super user password,

and I see essentially the exact same settings I had on the wired connection: the automatic start up, as other people use it, the identity and whether I want to use DHCP or set things manually or not.

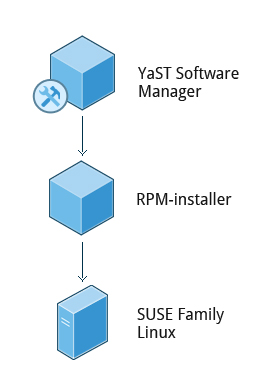
So that's all there really is in the graphical interface to control basic network connections; it's rather simple and it looks the same on all recent Linux distributions.

**openSUSE’s YaST Software Management**

The **YaST (Yet another Setup Tool)** software manager is similar to other graphical package managers. It is an RPM-based application. You can add, remove, or update packages using this application very easily. To access the YaST software manager:

1. Click *Activities*
2. In the *Search* box, type "YaST"
3. Click the *YaST* icon
4. Click *Software Management*.

openSUSE’s YaST software management application is similar to the graphical package managers in other distributions. The demonstration of the YaST software manager is shown later in this section.



**openSUSE's Software Management**

We are now going to demonstrate software package management from a graphical interface using the openSUSE Linux distribution.

The tool we want to use for doing this is YaST, y-s-t, and there's two different ways I can reach it.

If I have the GNOME shell extension installed which shows the Applications menu, which doesn't come by default, I can go down to System Tools,

and then go down to YaST, or I can always go up to the upper right hand corner, click on the Tool icon,

and then, at the bottom here, on the system, click on YaST. It will want the root password because I'm going to manipulate what's installed on the system.

And so, if I go Software, I can go to Software Management.

Now, the first thing that YaST has to do is study the system and get a database of every possible package.

So, you can look at the packages on the system in different ways.

You can look by what's called RPM Groups. So, here's all the Games, grouped under Amusements.

I have Applications, Development, and all different kinds of Development subcategories, Tools, Documentation, etc.

Or RPM Groups... No, this is RPM Groups. Okay...

Or, I can look at Package Groups, which maybe is a little more intuitive.

So, I could click on, let's say, Multimedia and see all the different programs that are involved in dealing with multimedia.

Suppose I want to install something or inquire as to where something is installed. I can click on Search and type something in here.

I'll type gnuplot and I bring up quickly all the different gnuplot programs there are.

The main one, you see, has a check mark in the box. That means it's already installed.

I will decide to install gnuplot-doc, which is a small documentation package.

And then, in order to do the install, I will click on Accept and I will go ahead and do it. It's a pretty small package. It doesn't take long.

If you needed some other packages to function, it would have installed them at the same time.

So now, I'll say Continue, and if I can, I want to remove it because I don't really need it.

I'll type it in the Search again and then I click on here until I get an X for removal. And then I say Accept and it's removing it.

So that's all I have to do to do some basic package installation and removal under GNOME software with openSUSE. Pretty simple.

**Ch 5 – Summary (System Configuration from the Graphical Interface)**

* Apply system, display, and date and time settings using the *System Settings* panel.
* Track the network settings and manage connections using *Network Manager* in Linux.
* Install and update software in Linux from a graphical interface.
* You can control basic configuration options and desktop settings through the *System Settings* panel.
* Linux always uses Coordinated Universal Time (UTC) for its own internal time-keeping. You can set the date and time settings from the *System Settings* window.
* The Network Time Protocol is the most popular and reliable protocol for setting the local time via Internet servers.
* The *Displays* panel allows you to change the resolution of your display and configure multiple screens.
* Network Manager can present available wireless networks, allow the choice of a wireless or mobile broadband network, handle passwords, and set up VPNs.
* **dpkg**and **RPM** are the most popular package management systems used on Linux distributions.
* Debian distributions use **dpkg** and **apt**-based utilities for package management.
* RPM was developed by Red Hat, and adopted by a number of other distributions, including the openSUSE, Mandriva, CentOS, Oracle Linux, and others.

**Ch 6 – Common Applications**

There are thousands of applications in Linux and new ones appear almost every day.

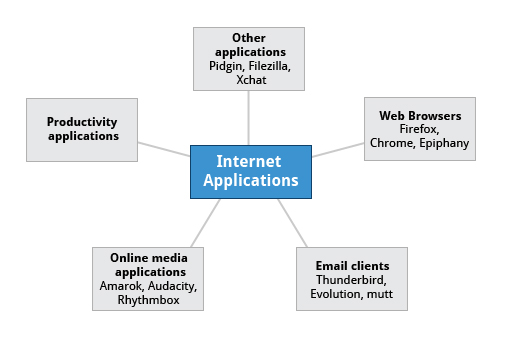
In this lesson, we will cover the common internet applications, common multimedia applications, common graphic editors, and other common tools and utilities.

By the end of this chapter, you should be familiar with common Linux applications, including:

* Internet applications such as browsers and email programs.
* Office Productivity Suites such as LibreOffice.
* Developer tools, such as compilers, debuggers, etc.
* Multimedia applications, such as those for audio and video.
* Graphics editors such as the GIMP and other graphics utilities.

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
* Streaming media applications
* Internet Relay Channels
* Conferencing Software.



**Internet Applications**

As discussed in*Chapter 4: Graphical Interface*, Linux offers a wide variety of web browsers, both graphical and text-based, including:

* Firefox
* Google Chrome
* Chromium
* Epiphany (renamed web)
* Konqueror
* linx, lynx, w3m
  + - Opera.

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, Evolution, and Claws Mail
* Text mode email clients, such as Mutt and mail
* All web browser-based clients, such as Gmail, Yahoo Mail, and Office 365.

**Email Applications**

Linux systems provide many other applications for performing Internet-related tasks. These include:

|  |  |
| --- | --- |
| **Application** | **Use** |
| **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. |
| **Hexchat** | To access Internet Relay Chat (IRC) networks. |

Most day-to-day computer systems have productivity applications (sometimes called office suites) available or installed. Each suite is a collection of closely coupled programs used to create and edit different kinds of files such as:

* Text (articles, books, reports, etc.)
* Spreadsheets
* Presentations
* Graphical objects.

Most Linux distributions offer LibreOffice, an open source office suite that started in 2010 and has evolved from OpenOffice.org. While other office suites are available as we have listed, LibreOffice is the most mature, widely used and intensely developed.

In addition, Linux users have full access to Internet-based Office Suites such as Google Docs and Microsoft Office 365.

Linux distributions come with a complete set of applications and tools that are needed by those developing or maintaining both user applications and the kernel itself.

These tools are tightly integrated and include:

* Advanced editors customized for programmers' needs, such as vi and emacs.
* Compilers (such as gcc for programs in C and C++) for every computer language that has ever existed.
* Debuggers such as gdb and various graphical front ends to it and many other debugging tools (such as valgrind).
* Performance measuring and monitoring programs, some with easy to use graphical interfaces, others more arcane and meant to be used only by serious experienced development engineers.
* Complete Integrated Development Environments (IDE's) such as Eclipse, that put all these tools together.

On other operating systems, these tools have to be obtained and installed separately, often at a high cost, while on Linux they are all available at no cost through standard package installation systems.

 Graphic editors allow you to create, edit, view, and organize images of various formats, like Joint Photographic Experts Group (JPEG or JPG), Portable Network Graphics (PNG), Graphics Interchange Format (GIF), and Tagged Image File Format (TIFF).

GIMP (GNU Image Manipulation Program) is a feature-rich image retouching and editing tool similar to Adobe Photoshop and is available on all Linux distributions. Some features of the GIMP are:

* It can handle any image file format.
* It has many special purpose plugins and filters.
* It provides extensive information about the image, such as layers, channels, and histograms.

**Ch 6 – Overview (Common Applications)**

You should now be familiar with common Linux applications, including:

* Internet applications such as browsers and email programs.
* Office Productivity Suites such as LibreOffice.
* Developer tools, such as compilers, debuggers, etc.
* Multimedia applications, such as those for audio and video.
* Graphics editors such as the GIMP and other graphics utilities.

**You have completed Chapter 6. Let’s summarize the key concepts covered:**

* + - Linux offers a wide variety of Internet applications, such as web browsers, email clients, online media applications, and others.
    - Web browsers supported by Linux can be either graphical or text-based, such as Firefox, Google Chrome, Epiphany, w3m, lynx, and others.
    - Linux supports graphical email clients, such as Thunderbird, Evolution, and Claws Mail, and text mode email clients, such as Mutt and mail.
    - Linux systems provide many other applications for performing Internet-related tasks, such as Filezilla, XChat, Pidgin, and others.
    - Most Linux distributions offer LibreOffice to create and edit different kinds of documents.
* Linux systems offer entire suites of development applications and tools, including compilers and debuggers.
* Linux systems offer a number of sound players including Amarok, Audacity, and Rhythmbox.
* Linux systems offer a number of movie players, including VLC, MPlayer, Xine, and Totem.
* Linux systems offer a number of movie editors, including Kino, Cinepaint, Blender among others.
* The GIMP (GNU Image Manipulation Program) utility is a feature-rich image retouching and editing tool available on all Linux distributions.
* Other graphics utilities that help perform various image-related tasks are eog, Inkscape, convert, and Scribus.

***By the end of this chapter, you should be able to:***

* + - Use the command line to perform operations in Linux.
    - Search for files.
    - Create and manage files.
    - Install and update software.

**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:

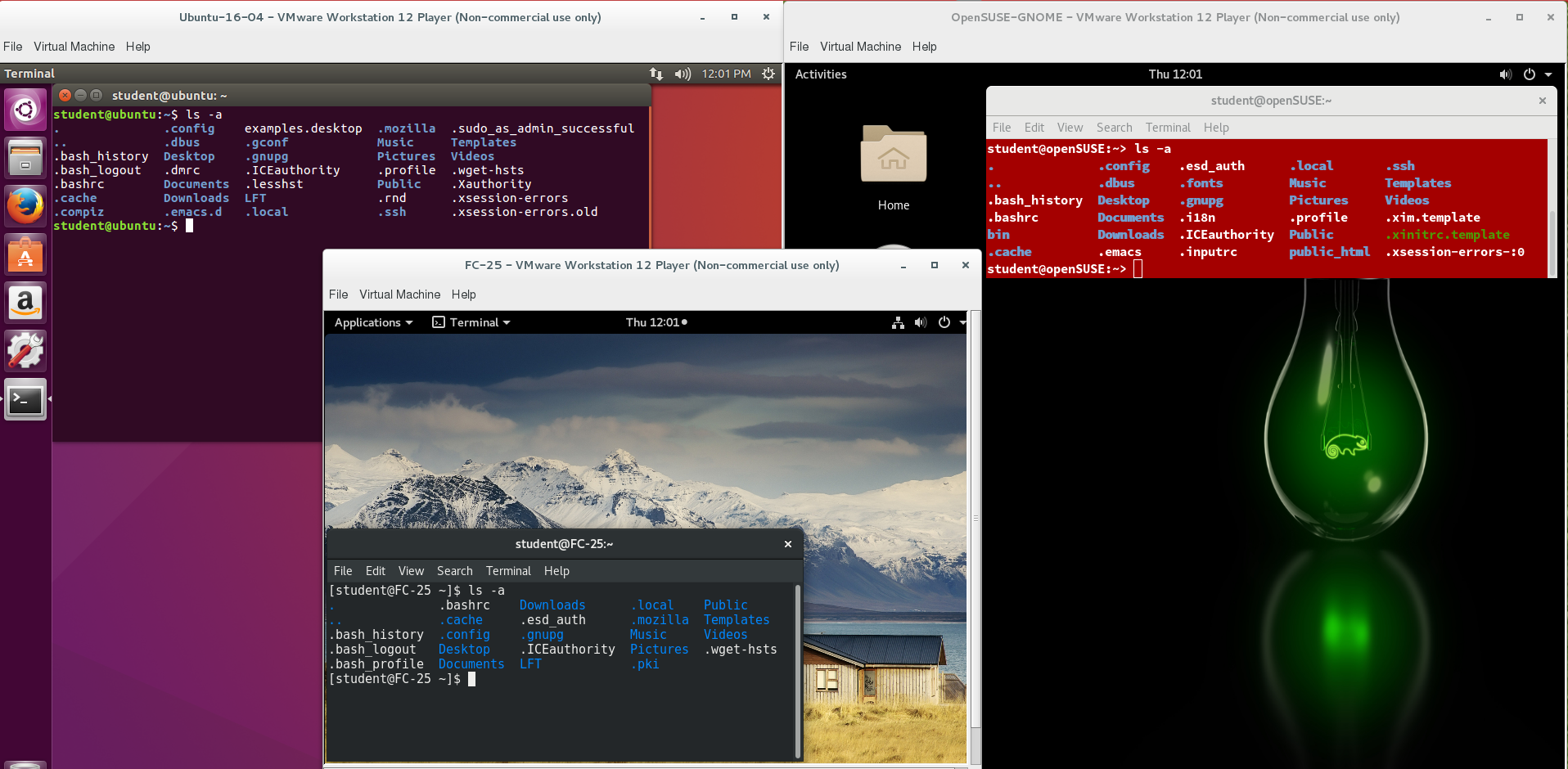
* + - No GUI overhead is incurred.
    - Virtually any and every task can be accomplished while sitting at the command line.
    - You can implement scripts for often-used (or easy-to-forget) tasks and series of procedures.
    - You can sign into remote machines anywhere on the Internet.
    - You can initiate graphical applications directly from the command line instead of hunting through menus.
    - While graphical tools may vary among Linux distributions, the command line interface does not.

**Using a Text Terminal on a Graphic Desktop**

A terminal emulator program emulates (simulates) a standalone 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.

[](https://prod-edxapp.edx-cdn.org/assets/courseware/v1/8de9705c7b1a71d12ec2d5308478064e/asset-v1:LinuxFoundationX+LFS101x+3T2018+type@asset+block/commandall.png)

**$ ls -a**

**Launching Terminal Windows**

To open a terminal on any system using a recent GNOME desktop, including CentOS 7/Red Hat 7/OpenSUSE, click on *Applications*> *System Tools*> *Terminal*.

On any but some of the most recent GNOME-based distributions, you can always open a terminal by right-clicking anywhere on the desktop background and selecting *Open in Terminal*.If this does not work, try installing the **nautilus-open-terminal** package if your Linux distribution offers it.

You can also hit**Alt-F2** and type in either gnome-terminalor konsole, whichever is appropriate.

Distributions seem to bury opening up a command line terminal and the place in menus will vary in the desktop GUI as versions evolve. It is a good idea to figure out how to "pin" the terminal icon to the panel, which might mean adding it to "favorites" on GNOME systems.

**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.

**>cat /etc/resolv.conf**

**>man head | head**

**>man tail | tail**

**>man head | grep OPTION**

**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.

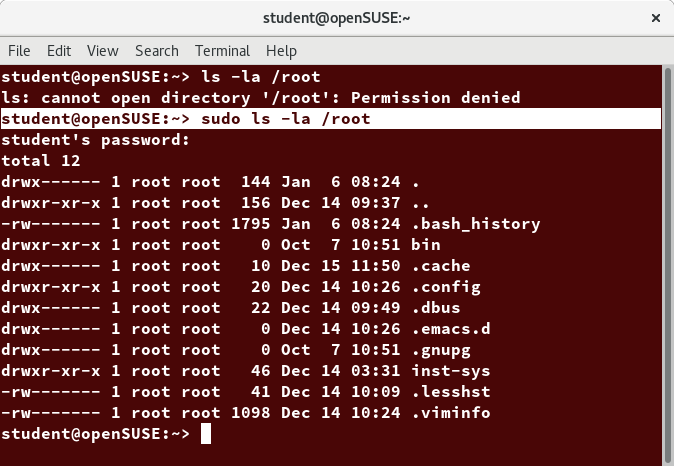
However, plenty of commands have no options, no arguments, or neither. In addition, other elements (such as setting environment variables) can also appear on the command line when launching a task.

**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 Red Hat/Fedora or SUSE/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 set up and run sudo on your system.



**sudo ls -la /root**

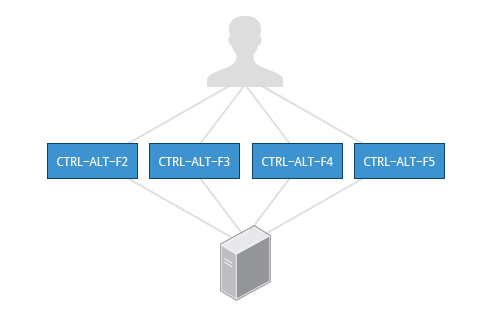
**Steps for Setting Up and Running sudo**

If your system does not already have sudoset 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 sudoyou 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.

**Virtual Terminals**



**Switching between Virtual Terminals**

**Turning Off the Graphical Desktop**

Linux distributions can start and stop the graphical desktop in various ways. The exact method differs from distribution and among distribution versions. For the newer systemd-based distributions, the display manager is run as a service, you can stop the GUI desktop with the systemctl utility and most distributions will also work with the **telinit** command, as in:

**$ sudo systemctl stop gdm**(or **sudo telinit 3**)

and restart it (after logging into the console) with:

**$ sudo systemctl start gdm**(or **sudo telinit 5**)

On Ubuntu versions before 18.04 LTS,substitute lightdm for gdm.

Methods of bringing down the GUI:

**student:/tmp> sudo systemctl stop gdm**

**student:/tmp> sudo systemctl stop lightdm**

**student:/tmp> sudo telinit 3**

**student:/tmp> sudo service gdm stop**

Methods of bringing the GUI back up:

**student:/tmp> sudo systemctl start gdm**

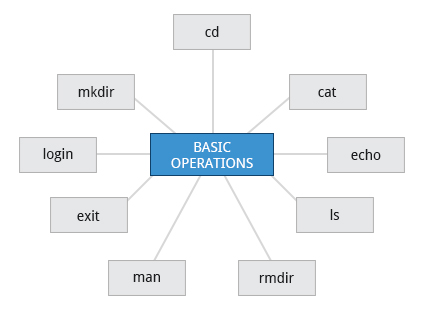
**student:/tmp> sudo systemctl start lightdm**

**student:/tmp> sudo telinit 5**

**student:/tmp> sudo service gdm start**

**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 shut down the system, locate applications, access directories, identify absolute and relative paths, and explore the filesystem.



**Basic Operations**

**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 into the system, you can perform basic operations.

Once your session is started (either by logging into a text terminal or via a graphical terminal program), you can also connect and log into 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 will not 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."**

***Note***: On Ubuntu systems, the shutdown message is not broadcast to users currently on the system. This would appear to be a bug, as it violates the man page for shutdown.

**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 and scripts should live in the **/bin**, **/usr/bin**, **/sbin**, **/usr/sbin**directories, or somewhere under **/opt**. They can also appear in **/usr/local/bin**and **/usr/local/sbin**, or in a directory in a user's account space, such as **/home/student/bin**.

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  
/usr/bin/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**  
**diff: /usr/bin/diff /usr/share/man/man1/diff.1.gz /usr/share/man/man1p/diff.1p.gz**

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 (shortcut name is **~** (tilde)) |
| **cd ..** | Change to parent directory (**..**) |
| **cd -** | Change to previous directory (**- (minus)**) |

Let’s get started with accessing directories using the command prompt.

* 1. To locate the ccache compiler using whereis:
  2. At the command prompt, type whereis ccache
  3. Press ENTER.

To display the present working directory:

* 1. Type pwd and press ENTER.

To change to the /usr/bin directory:

* 1. At the command prompt, type cd /usr/bin and press ENTER.

To change to your home directory:

* 1. At the command prompt, type cd ~ or cd.
  2. Press ENTER.

To change to the parent directory of your current directory:

* 1. At the command prompt, type cd ..
  2. Press ENTER.

To change to the previous directory:

* 1. AAt the command prompt, type cd -.
  2. Press ENTER.

To check where you are with pwd:

* 1. Type pwd and press ENTER.

**Understanding Absolute and Relative Paths**

There are two ways to identify paths:

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

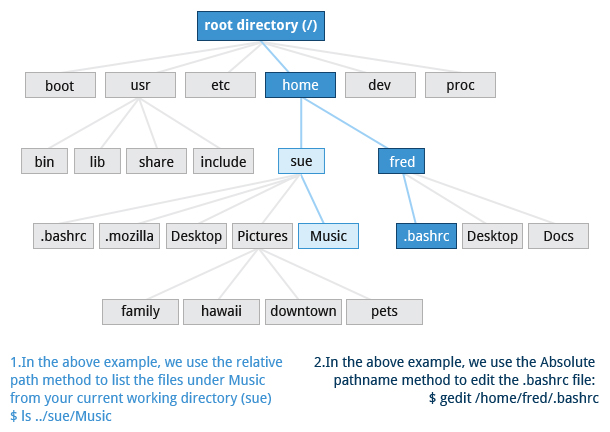
Multiple slashes (**/**) between directories and files are allowed, but all but one slash between elements in the pathname is ignored by the system. **////usr//bin**is valid, but seen as **/usr/bin**by the system.

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:

* + - Absolute pathname method  
      **$ cd /usr/bin**
    - Relative pathname method  
      **$ cd ../../usr/bin**

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



**Understanding Absolute and Relative Paths**

**Exploring the Filesystem**

tree -d

The following commands can help in exploring the filesystem:

|  |  |  |
| --- | --- | --- |
| **Command** | **Usage** | |
| **cd /** | | Changes your current directory to the root (/) directory (or path you supply) |
| **ls** | | List the contents of the present working directory |
| **ls –a** | | List all files, including hidden files and directories (those whose name start with . ) |
| **tree** | | Displays a tree view of the filesystem |

Let’s get started with how to explore the filesystem.

To access /usr/local/lib from the /usr directory using absolute path:

* 1. At command prompt, type cd /usr/local/lib and press ENTER

Go back to /usr directory, by typing cd /usr and press ENTER

To access /usr/local/lib from the /usr directory using relative path:

* 1. At command prompt, type cd local/lib and press ENTER.

To get to the root directory:

* 1. At the command prompt, type the cd / command and press ENTER.

To get a list of the files and directory in the present working directory:

* 1. At the command prompt, type the ls command and press ENTER

To get a list of hidden files and hidden directories:

* 1. At the command prompt, type the ls –a command and press ENTER

To get a tree view of the filesystem:

At the command prompt, type the tree command and press ENTER

To get a tree view of only directories in the filesystem:

At the command prompt, type the tree -d / command and press ENTER

**Hard 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**

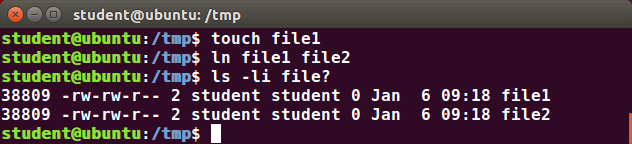
Note that two files now appear to exist. However, a closer inspection of the file listing shows that this is not quite true.

**$ 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 nameassociated 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.



**Hard Link**

**Soft (Symbolic) Links**

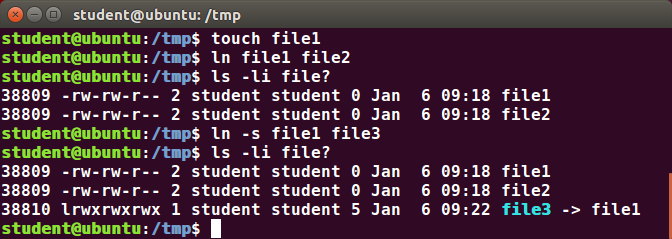
Soft (or Symbolic) links are created with the **-s**option, as in:

**$ 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.

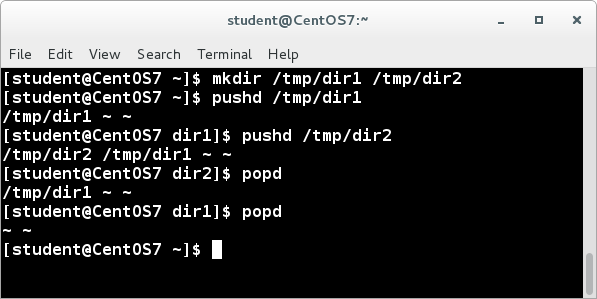
Unlike hard links, soft links can point to objects even on different filesystems, partitions, and/or disks and other media,  which may or may not be currently available or even exist. In the case where the link does not point to a currently available or existing object, you obtain a dangling link.



**Soft (Symbolic) Links**

**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.



**Navigating Through Directory History**

Let's get started with how to navigate through the directory history using the command prompt.

1. Let us say, we have two directories named dir1 and dir2 created under tmp directory and the current working directory is the test3's home directory.

2. To change from test3's home directory to dir1 under tmp directory, at the command prompt type cd /tmp/dir1 and press Enter.

3. To push the current directory to stack type pushd. and press Enter.

Now you working directory is dir1.

4. Now To change to dir2 under tmp directory, at the command prompt type cd /tmp/dir2 and press Enter.

Now you working directory is dir2.

5. To push the current directory to stack, type pushd. and press Enter.

Now, the stack contains the details of the last two directories that you have pushed.

6. To view the directories in the stack, at the command prompt type dirs and press Enter.

You can see the names of the two directories you have pushed to the stack

7. To pop the top-most directory from the stack, at the command prompt type popd and press Enter.

8. To check the current working directory, at the command prompt type pwd and press Enter.

9. To pop the top-most directory from the stack, at the command prompt type popd and press Enter.

10. To check the current working directory, at the command prompt type pwd and press Enter.

Now you're working directory is changed to dir1

11. To pop the top-most directory from the stack, at the command prompt type popd and press Enter.

Since, no entries are available in the stack you get a notification

**Find out the location of the ip network utility.**

**student:/tmp> which ip**

**/usr/sbin/ip**

**student:/tmp> whereis ip**

**ip: /usr/sbin/ip /usr/share/man/man7/ip.7.gz /usr/share/man/man8/ip.8.gz**

Note **whereis** also reports the location of the **man** page.

**Working with Files**

You can use the following command line 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: "less is more". |
| **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. |

Let's get started on viewing files using the different available commands.

To view the entire messages file on screen:

1. At the command prompt, type cat messages.

2. Press ENTER.

To view the entire messages file with scroll-back:

1. At the command prompt, type less messages.

2. Press ENTER.

To view last ten lines of the messages file:

1. At the command prompt, type tail messages.

2. Press ENTER.

To view first ten lines of the messages file:

1. At the command prompt, type head messages.

2. Press ENTER.

To view last three lines of the messages file:

1. At the command prompt, type tail -3 messages.

2. Press ENTER.

To view first three lines of the messages file:

1. At the command prompt, type head -3 messages.

2. Press ENTER.

**touch**

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

However, you can also create an empty file using **touch**:

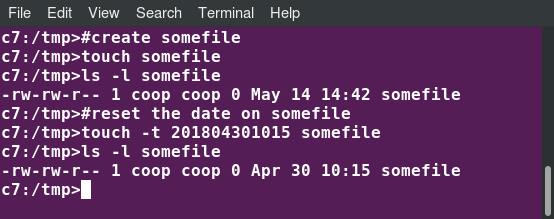
**$ touch <filename>**

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

**touch**provides several useful options. For example, the **-t** option allows you to set the date and timestamp of the file to a specific value, as in:

**$ touch -t 12091600 myfile**

This sets the **myfile** file's timestamp to 4 p.m., December 9th (12 09 1600).



**touch**

**mkdir and rmdir**

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

However, you can also create an empty file using **touch**:

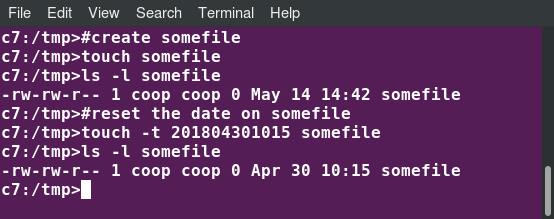
**$ touch <filename>**

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

**touch**provides several useful options. For example, the **-t** option allows you to set the date and timestamp of the file to a specific value, as in:

**$ touch -t 12091600 myfile**

This sets the **myfile** file's timestamp to 4 p.m., December 9th (12 09 1600).



**touch**

**Moving, Renaming or Removing a File**

Note that **mv** does double duty, in that it can:

* + - Simply rename a file
    - Move a file to another location, while possibly changing its name at the same time.

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.

|  |  |
| --- | --- |
| **Command** | **Usage** |
| **mv** | Rename a file |
| **rm** | Remove a file |
| **rm –f** | Forcefully remove a file |
| **rm –i** | Interactively remove a file |

**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).

|  |  |
| --- | --- |
| **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 $**

By convention, most systems are set up so that the root user has a pound sign (**#**) as their prompt.

Let’s get started on working with files and directories using command prompt

To create two empty files test1 & test2 with timestamp : 14 march 2018 2:00 PM:

1. At the command prompt, type touch -t 1803141400 test1 test2.

2. Press ENTER.

To check if files are created:

1. At the command prompt, type ls -l.

2. Press ENTER.

To rename the file test1 to new\_test1:

1. At the command prompt, type mv test1 new\_test1.

2. Press ENTER.

To remove test2 interactively:

1. At the command prompt, type rm -i test2.

2. Press ENTER.

The Confirmation message is displayed.

3. Type y and press ENTER.

To remove new\_test1 forcefully:

1. At the command prompt, type rm -f new\_test1.

2. Press ENTER.

To create a directory dir1:

1. At the command prompt, type mkdir dir1.

2. Press ENTER.

To remove a directory dir1: mdir dir1.

2. Press ENTER.

**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, and stdout and stderr are printed on your terminal. stderr is often redirected to an error logging file, while stdin is 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, usually 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 page and in the 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 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 use them to provide 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**

***Note***: By the same logic, **do\_something 1> output-file** is the same as **do\_something > output-file**.

A special shorthand notation can send anything written to file descriptor **2** (stderr) to the same place as file descriptor **1** (stdout): **2>&1**.

**$ do\_something > all-output-file 2>&1**

bashpermits an easier syntax for the above:

**$ do\_something >& all-output-file**

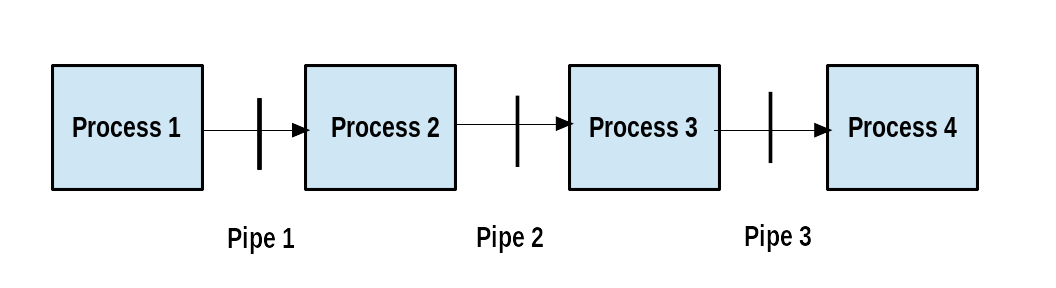
**Pipes**

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.

Furthermore, 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.



**Pipeline**

**locate**

The **locate** utility program performs a search taking advantage of 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 (and possibly 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 a database created by a related utility, **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.

Let's get started on how to find files using the command prompt.

1. To find a file named passwd, at the command prompt, type the locate passwd command and press Enter.

You can see the list of directories and files with passwd as the part of its name

2. To copy the /etc/passwd file to test1.txt under your home directory, at the command prompt, type cp /etc/passwd test1.txt and press Enter.

3. Type Y and press ENTER

4. To update locate database, at the command prompt, type sudo updatedb and press Enter.

5. To locate the file test1.txt using locate, at the command prompt, type locate test 1.txt and press Enter.

You can see the details of the test1.txt

**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 **?**. For example, if you know only the first two letters are 'ba' of a three-letter filename with an extension of **.out**, type **ls ba?.out**.

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

Let's get started on how to use the different wildcards to search for files.

To view all files in the current directory using the ls command:

1. At the command prompt, type ls –a command and press ENTER.

2. All the files under current directory are displayed

To search for files using the Bash wildcard:

1. At the command prompt, type the ls ba?.? command.

2. Press ENTER.

All the filenames starting with "ba", plus a single character,

plus a single character extension are shown.

To search for files using the \* Bash wildcard:

1. At the command prompt, type the ls \*.out command.

2. Press ENTER.

All the filenames having the .out extension are displayed.

To search for files using the set Bash wildcard notation:

1. At the command prompt, type the ls ba[a-s].c command.

2. Press ENTER.

All the filenames starting with "ba" and have an alphabetic character between a and s are displayed.

To search for files using the !set Bash wildcard notation:

1. At the command prompt, type the ls ba[!a-s].c command.

2. Press ENTER.

All the filenames starting with "ba" and not having an alphabetic character between a and s are displayed.

**The find Program**

**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 potentially 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 inessential or outdated files in **/tmp** (and other volatile directories, such as those containing cached files) that have not been accessed recently. Many Linux distributions use shell scripts that run periodically (through **cron** usually) to perform such house cleaning.

 $ sudo find . –name “\*.log”

**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 **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.

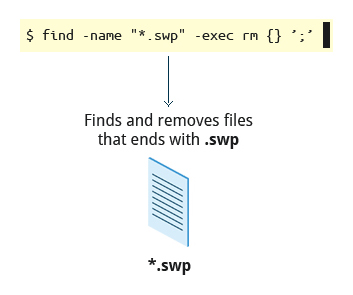
To find and remove all files that end with **.swp**:

**$ find -name "\*.swp" -exec rm {} ’;’**

The **{}**(squiggly brackets) is a placeholder 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 and Removing Files that Ends with .swp**

**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. It is easy to perform such searches.

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 manpage 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 {} ’;’**

Let’s get started on how to search for files in a directory.

To search for a file based on a filename:

1. At the command prompt, type the find / usr -name gcc command.

2. Press ENTER.

All the files with GCC as the file name will be displayed.

To search for a directory based on a filename:

1. At the command prompt, type the find / usr -type d -name gcc command.

2. Press ENTER.

To search files in current directory which were modified today:

1. At the command prompt, type find -type f -mtime 0.

2. Press ENTER.

To search files with size 0 bytes:

1. At the command prompt, type find -type f -size 0

2. Press ENTER.

Lab 7.4: Find the **init.d** directory, starting from **/**, and then create a symbolic link from within your home directory to this directory.

Note that this SysVinit directory is no longer used much in systemd-based systems, but is kept for backwards compatibility reasons.

**student:/tmp> find / -type d -name init.d**

**student:/tmp> cd ~**

**student:/home/student> ln -s /etc/init.d .**

Note you will get a lot of noise about trying to look at files and directories normal users are not allowed to examine. If you preface the **find** command with **sudo** these will not occur.)

**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 well and cooperate with the other components that comprise the entire 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.

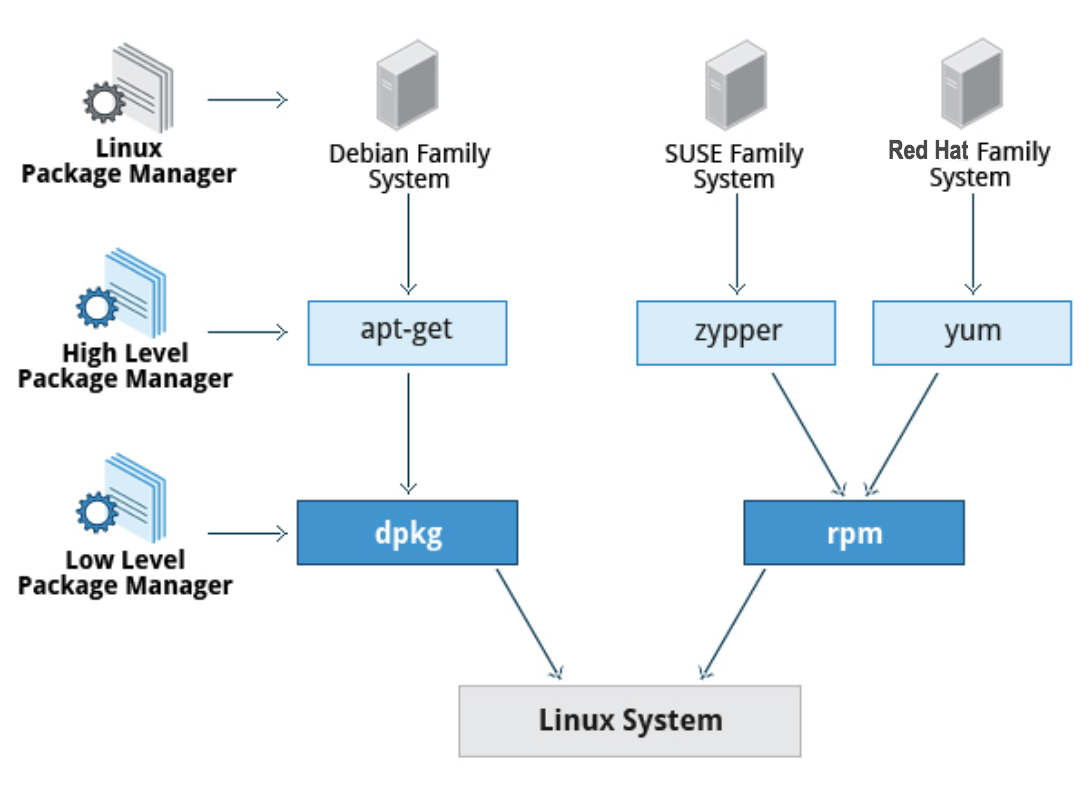
There are two broad families of package managers: those based on Debian and those which use RPM as their low-level package manager. The two systems are incompatible, but broadly speaking, provide the same features and satisfy the same needs. There are some other systems used by more specialized Linux distributions.

In this section, you will learn how to install, remove, or search for packages from the command line using these two package management systems.

**Package Managers: Two (2) Levels**

Both package management systems operate on two distinct 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*,* dnf or zypper) works with groups of packages, downloads packages from the vendor, and figures out dependencies.

Most of the time users need to work only with the high-level tool, which will take care of calling the low-level tool as needed. Dependency resolution 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.



**Package Managers: Two Levels**

**Low-Level Debian Package Management with dpkg**

Let's get some practice with the basic low-level command for the Debian packaging system, dpkg or d-package.

So, to get a list of all the packages on the system, I can simply type "dpkg –list | less".

and I'll just pipe that into "less" because it's a long list.

Okay. Actually, if I don't use "less", for some reason, it's more compact.

Okay. So, there are a lot of different packages on the system.

Now, if I want to see some information just about a particular package, I could just grep let's say bzip2, to see what's going on with bzip2.

dpkg –list | grep bzip2

And that's the information about bzip2. That tells us the version number, the architecture, which here is amd64, and its description as a high quality block sorting file compressor utility.

So, it's a more advanced program and with better compression than gzip or the old-fashioned zip program.

If I want to see what's actually contained in that package, I can do "dpkg --listfiles" and then the name of it [bzip2], and then I'll go slower by putting it into "less",

and you'll see, there's the executables in the /bin directory, and then, what's under /usr is basically documentation, under /usr/share/doc, usr/share/man, etc.

If I try to remove the package, let's see what happens.

So, do "sudo dpkg". I, of course, have to have root privilege to remove the package.

So, I'm removing bzip2, "dpkg --remove bzip2", and it's telling me I cannot do this because I need the developing package dpkg-dev. That would be headers and stuff for programs which use this in a library version.

And file-roller, which is an archive extractor that's common on Linux systems. So, I would have to remove all three of them.

We'll see later, when we use tools like apt-get, it's easier to deal with these dependencies.

**Low-Level RPM Package Management with rpm**

rpm –qa | grep bzip2

rpm –qil bzip2 | less

ls –lF $(rpm – ql bzip2) | less

**High-Level Package Management with yum on CentOS 7**

$ sudo yum list “\*bzip2\*”

$ sudo yum info lbzip2

Now, let’s install lbzip2 utilities;

$ sudo yum install lbzip2-utils

Error, must install lbzip2 package before you can install utilities. Then prompts “Install package?”

**High-Level Package Management with zypper on openSUSE**

Let’s first search for gnuplot packages;

**$ zypper search gnuplot**

**$ sudo zipper install gnuplot-doc**

Warning, yes.

Info on install;

**$ rpm –qi gnuplot-doc**

$ sudo zipper remove gnuplot

Error, must delete gnuplot-doc as well, are you sure?

**High-Level Package Management with apt on Ubuntu**

$ sudo apt-cache search wget2

$ sudo apt-get install wget2-dev

Now, let’s remove it

$ sudo apt-get remove wget2

**Lab 7.5: Installing and Removing Software Packages**

Using the upper-level package management system appropriate for your Linux distribution, do the following:

* + 1. Install the dump package on your system.
    2. Remove the dump package from your system.

***Note***: If dump is already installed (you will be told so when you try to install), then do things in opposite order, i.e. remove and then install.

**student:/tmp> apt-get install dump**

**student:/tmp> apt-get remove dump**

or

**student:/tmp> yum install dump**

**student:/tmp> yum remove dump**

or

**student:/tmp> zypper install dump**

**student:/tmp> zypper remove dump**

**Learning Objectives (Review)**

You should now be able to:

* Use the command line to perform operations in Linux.
* Search for files.
* Create and manage files.
* Install and update software.

**Ch 7 – Summary (Command Line Operations)**

You have completed Chapter 7. Let’s summarize the key concepts we 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.
    - Usinghard 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.

**Ch 8 – Learning Objectives –-help & man**

By the end of this chapter, you should be able to:

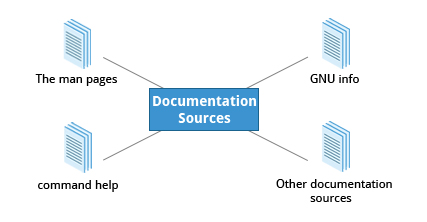
* Use different sources of documentation.
* Use the man pages.
* Access the GNU info System.
* Use the **help** command and **--help** option.
* Use other documentation sources.

**Ch 8 - Linux Documentation Sources**

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. [Gentoo Handbook](https://www.gentoo.org/support/documentation/) or [Ubuntu Documentation](https://help.ubuntu.com/community/CommunityHelpWiki).



**Linux Documentation Sources**

**Ch 8 – 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, and programming APIs for system calls, library routines, and the kernel. They are present on all Linux distributions and are always at your fingertips.

The manpages infrastructure was first introduced in the early UNIX versions, at the beginning of 1970s. The name man is just an abbreviation for manual.

Typing man with a topic name as an argument retrieves the information stored in the topic's man pages.

man pages are often converted to other formats, such as PDF documents and web pages. To learn more, take a look at [Linux man pages online](http://man7.org/linux/man-pages/). Many web pages have a graphical interface for help items, which may include man pages.

The man program searches, formats, and displays the information contained in the man page system. Because many topics have copious amount of relevant 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.

A given topic may have multiple pages associated with it and there is a default order determining which one is displayed when no options or section number is specified. To list all pages on the topic, use **-f**option. To list allpages that discuss a specified topic (even if the specified subject is not present in the name), use the **–k** option.

**man –f**generates the same result as typing **whatis**.

**man –k**generates the same result as typing **apropos.**

The default order is specified in **/etc/man\_db.conf**and is roughly (but not exactly) in ascending numerical order by section.



**man**

The man pages are divided into chapters numbered 1 through 9. In some cases, 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, as in:

**$ man -a socket**

**$ man socket**

**$ man –f socket**

**$ whatif socket**

**$ man 7 socket**

**$ man –a socket**

**$ man –k socket ;all with socket.**

**Lab 8 – Working with man**

Now, try to do the following:

* + 1. Finding man pages  
       From the command line, bring up the man page for man itself. Scroll down to the **EXAMPLES** section.
    2. Finding man pages by topic  
       What man pages are available that document file compression?
    3. Finding man pages by section  
       From the command line, bring up the man page for the **printf** library function. In which manual page section are library functions found?

**student:/tmp> man man**

Use the **Page Down** key or search in less with the '**/**' key for the **EXAMPLES** section.

**student:/tmp> man -k compress**

or

**student:/tmp> apropos compress**

will bring up a long list of programs and references, including **gzip**, **bzip2** and **xz**, and a number of file utilities that work with compressed files, such as **zless**, **zgrep**, **bzcat**, and **xzdiff**.

**student:/tmp> man 3 printf**

(**man printf** will bring up the command-line utility (section 1) of the same name.)

**Ch 8 – The GNU Info System**

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

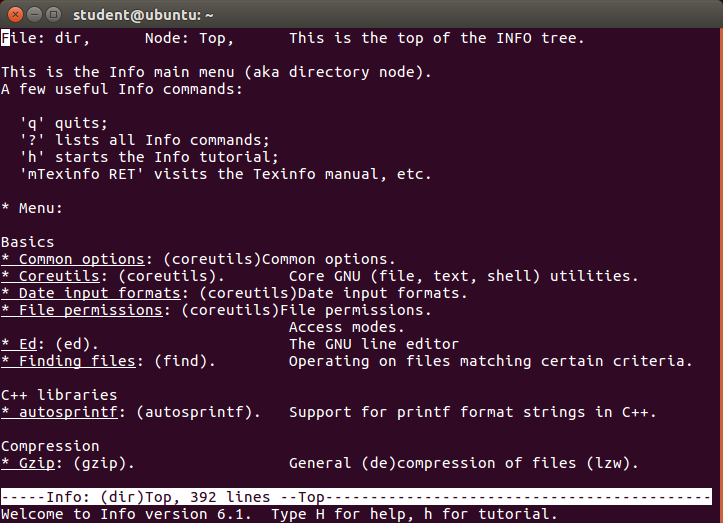
This is the GNU project's standard documentation format, which it prefers as an alternative to man. The Info System is basically free-form, and supports linked subsections.

Functionally, info 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.

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**

The topic which you view in an info page is called a node. The table lists the basic keystrokes for moving between nodes.

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

Items function like browser links and 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.

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

Examples;

$ info make

Select Next, Previous or Up (top of section).

**Lab 8.2: Working with info**

From the command line, bring up the info page for cpio. Bring up the tutorial.

$ info cpio

**Ch 8 – 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 infopages.

**Ch 8 – The help Command**

When run within a bash command shell, some popular commands (such as **echo**and **cd**) actually run especially built into bash versions of the commands rather than the usual binaries found on the file system, say under **/bin** or **/usr/bin**. It is more efficient to do so as execution is faster because fewer resources are used. We will discuss command shells such as bash in detail later. One should note that there can be some (usually small) differences in the two versions of the command.

To view a synopsis of these built-in commands, you can simply type **help**as shown in the screenshot.

For these built-in commands, **help**performs the same basic function as the **-h** and **--help** arguments perform for standalone programs.

**Lab 8.3 – Working with Command Line help**

**Ch 8 -**

List the available options for the **mkdir** command, in more than one way.

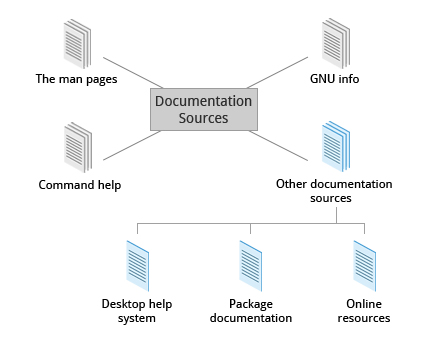
$ mkdir –help

$ man mkdir

**Ch 8 – Other Documentation Sources**

In addition to the man pages, the GNU info System, and the **help** command, there are other sources of Linux documentation, some examples of which include:

* + - Desktop help system
    - Package documentation
    - Online resources.



**Other Documentation Sources**

**Ch 8 – Graphical Help Systems**

All Linux desktop systems have a graphical help application. This application is usually displayed as a question-mark icon or an image of a ship’s life-preserver, and can also always be found within the menu system. These programs usually contain custom help for the desktop itself and some of its applications, and will sometimes also include graphically-rendered info and man pages.

If you don't want to spend time hunting for the right icon or menu item to launch the help application, you can also start the graphical help system from a terminal window or command prompt by using one of the following commands:

* + - GNOME: **gnome-help**or **yelp**
    - KDE: **khelpcenter**

**Ch 8 – 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, grouped in subdirectories named after each package, perhaps including the version number in the name.

**Ch 8 – 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 book has been well-reviewed by other users of this course. It is a free, downloadable command line compendium under a Creative Commons license: "[*The Linux Command Line*](http://linuxcommand.org/tlcl.php)" by William Shotts.

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 Documentation](https://help.ubuntu.com/)
* [CentoS Documentation](https://wiki.centos.org/Documentation)
* [openSUSE Documentation](https://doc.opensuse.org/)
* [Gentoo Documentation](https://www.gentoo.org/support/documentation/)
* [Fedora Documentation](https://docs.fedoraproject.org/).

**Ch 8 – Summary (Finding Linux Documentation)**

You should now be able to:

* + - Use different sources of documentation.
    - Use the man pages.
    - Access the GNU info System.
    - Use the **help** command and **--help** option.
    - Use other documentation sources.

You have completed Chapter 8. Let’s summarize the key concepts covered:

* + - 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.

**Ch 9 – Processes**

In Linux, all the application programs we are running are processes.

In this lesson, we are going to explore the concept of processes.

We are going to find out which processes are running; how to automatically start processes and how to list the resources that a process consumes.

**Ch 9 – Learning Objectives**

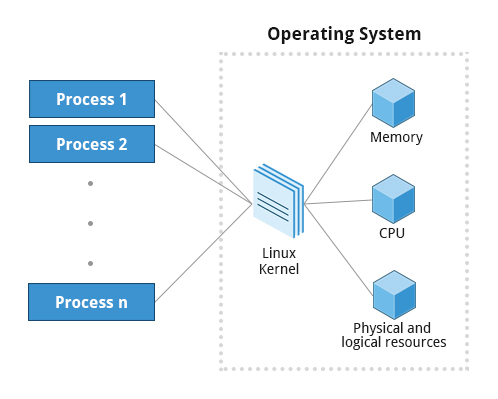
By the end of this chapter, you should be able to:

* Describe what a process is and distinguish between types of processes.
* Enumerate process attributes.
* Manage processes using **ps** and **top**.
* Understand the use of load averages and other process metrics.
* Manipulate processes by putting them in background and restoring them to foreground.
* Use **at**, **cron**, and **sleep** to schedule processes in the future or pause them.

**Ch 9 – 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 command 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 optimized system utilization.



**Processes**

**Ch 9 – 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 during system startup and then wait for a user or system request indicating that their service is required. | **httpd, xinetd, 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. | **firefox, gnome-terminal-server** |
| 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. | **kthreadd, migration, ksoftirqd** |

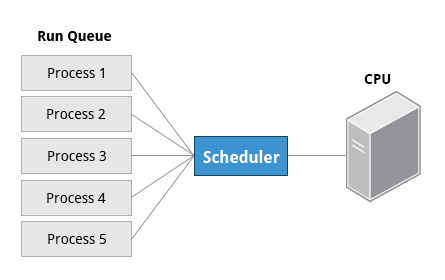
**Ch 9 – Process Scheduling and States**

A critical kernel function called the scheduler constantly shifts processes on and off the CPU, sharing time according to relative priority, how much time is needed and how much has already been granted to a task.

When a process is in a so-called running state, it means it is either currently executing instructions on a CPU, or is waiting to be granted a share of time (a time slice) so it can execute. 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 on 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 Scheduling and States**

**Ch 9 – 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:

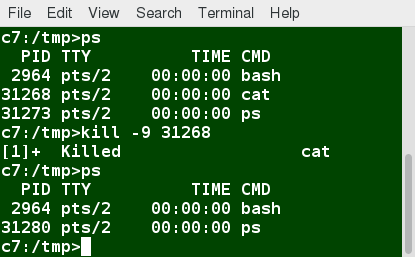
|  |  |
| --- | --- |
| **ID Type** | **Description** |
| Process ID (PID) | Unique Process ID number |
| Parent Process ID (PPID) | Process (Parent) that started this process. If the parent dies, the PPID will refer to an adoptive parent; on recent kernels, this is kthreadd which has PPID=2. |
| Thread 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. |

**Ch 9 – Terminating a Process**

At some point, one of your applications may stop working properly. How do you eliminate it?

To terminate a process, you can type **kill -SIGKILL <pid>** or **kill -9 <pid>**.

Note, however, you can only kill your own processes; those belonging to another user are off limits, unless you are root.



**Terminating a Process**

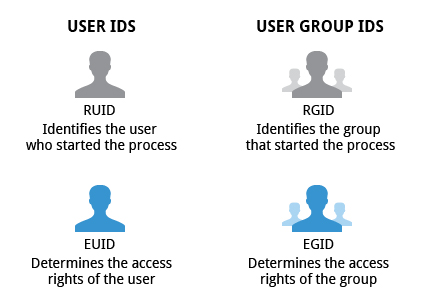
**Ch 9 – User and Group IDs**

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 (RGID). The access rights of the group are determined by the Effective Group ID (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).



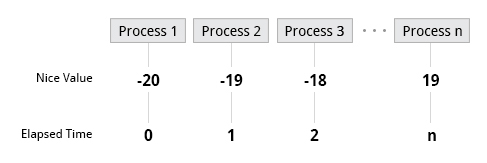
**User and Group IDs**

**Ch 9 – 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.

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.



**Nice Values**

**Ch 9 – Load Averages**

Load average is the average of the load number for a givenperiod of time. It takes into account processes that are:

* Actively running on a CPU
* Considered runnable, but waiting for a CPU to become available
* Sleeping: i.e. waiting for some kind of resource (typically, I/O) to become available.

***Note***: Linux differs from other UNIX-like operating systems in that it includes the sleeping processes. Furthermore, it only includes so-called uninterruptiblesleepers, those which cannot be awakened easily.

The load average can be viewed by running **w**, **top** or **uptime**. We will explain the numbers on the next page.

**Ch 9 – Interpreting Load Averages**

The load average is displayed using three different sets of numbers, as shown in the following example:

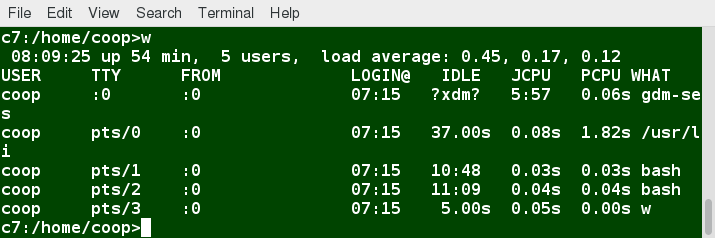
The last piece of information is the average load of the system. Assuming our system is a single-CPU system, the three load average numbers are interpreted as follows:

* **0.45**: For the last minute the system has been 45% utilized on average.
* **0.17**: For the last 5 minutes utilization has been 17%.
* **0.12**: For the last 15 minutes utilization has been 12%.

If we saw a value of 1.00 in the second position, that would imply that the single-CPU system was 100% utilized, on average, over the past 5 minutes; this is good if we want to fully use a system. A value over 1.00 for a single-CPU system implies that the system was over-utilized: there were more processes needing CPU than CPU was available.

If we had more than one CPU, say a quad-CPU system, we would divide the load average numbers by the number of CPUs. In this case, for example, seeing a 1 minute load average of 4.00 implies that the system as a whole was 100% (4.00/4) utilized during the last minute.

Short-term increases are usually not a problem. A high peak you see is likely a burst of activity, not a new level. For example, at start up, many processes start and then activity settles down. If a high peak is seen in the 5 and 15 minute load averages, it may be cause for concern.



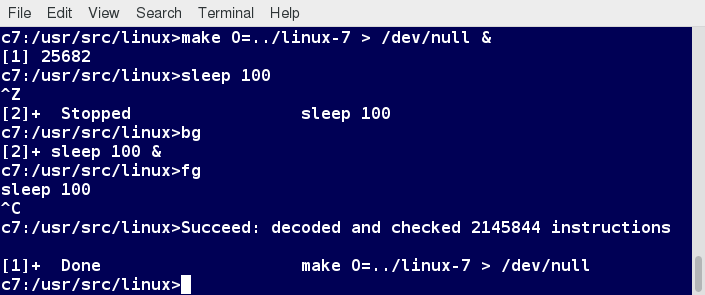
**Interpreting Load Averages**

**Ch 9 – Background and Foreground Processes**

Linux supports background and foreground job processing. A job in this context is just a command launched from a terminal window. Foreground jobs run directly from the shell, and when one foreground job is running, other jobs need to wait for shell access (at least in that terminal window if using the GUI) until it is completed. This is fine when jobs complete quickly. But this can have an adverse effect if the current job is going to take a long time (even several hours) to complete.

In such cases, you can run the job in the background and free the shell for other tasks. The background job will be executed at lower priority, which, in turn, will allow smooth execution of the interactive tasks, and you can type other commands in the terminal window while the background job is running. By default, all jobs are executed in the foreground. You can put a job in the background by suffixing **&**to the command, for example: **updatedb &**.

You can either use **CTRL-Z** to suspend a foreground job or **CTRL-C** to terminate a foreground job and can always use the **bg** and **fg**commands to run a process in the background and foreground, respectively.



**Background and Foreground Processes**

**Ch 9 – Managing Jobs**

The **jobs** utility displays all jobs running in background. The display shows the job ID, state, and command name, as shown here.

**jobs -l**provides the same information as **jobs**, including the PID of the background jobs.

The background jobs are connected to the terminal window, so, if you log off, the **jobs** utility will not show the ones started from that window.

**Lab 9.1 – Getting uptime and Load Averages**

Ascertain how long your system has been up.

Display its load averages.

$ uptime

$ top | head

$ w

**Lab 9.2 – Background and Foreground Jobs**

We are going to launch a graphical program from a terminal window, so that one can no longer type in the window. **gedit** is an easy choice but you can substitute any other program that does this.

1. Open **gedit** on a new file as in

**$ gedit somefile**

While you can still type in the terminal window, the shell will not pay attention to what you input.

1. While your pointer is over the terminal window, hit **CTRL-Z**.

**^Z**

**[3]+ Stopped gedit somefile**

You can no longer type in the **gedit** window.

1. With **jobs -l**, see what processes have been launched from this terminal window:

**$ jobs -l**

**[1] 17705 Running evince \*pdf &**

**[2]- 18248 Running emacs /tmp/hello.tex &**

**[3]+ 19827 Stopped gedit somefile**

1. Now put the most recent job (**gedit somefile**) in background:

**$ bg**

**[3]+ gedit somefile &**

Now you should be able to type in the **gedit** window.

1. Put the process in foreground again:

**$ fg**

**gedit somefile**

Note you once again input to the terminal window. has no effect

1. To clean up, suspend the process again and then use **kill** to terminate it:

^Z

**Ch 9 – The ps Command (System V Style)**

**ps**provides information about currently running processes keyed by PID. If you want a repetitive update of this status, you can use **top** or other commonly installed variants, such as **htop**or **atop**, from the command line, or invoke your distribution's graphical system monitor application.

**ps**has many options for specifying exactly which tasks to examine, what information to display about them, and precisely what output format should be used.

Without options, **ps**will display all processes running under the current shell. You can use the **-u** option to display information of processes for a specified username. The command **ps -ef** displays all the processes in the system in full detail. The command **ps -eLf** goes one step further and displays one line of information for every thread (remember, a process can contain multiple threads).

$ ps axo stat,priority,pid,pcpu,comm

**Ch 9 – The Process Tree**

**pstree**displays the processes running on the system in the form of a tree diagramshowing the relationship between a process and its parent process and any other processes that it created. Repeated entries of a process are not displayed, and threads are displayed in curly braces.



**The Process Tree**

**Ch 9 – top**

While a static view of what the system is doing is useful, monitoring the system performance live over time is also valuable. One option would be to run **ps**at regular intervals, say, every two minutes. A better alternative is to use **top** to get constant real-time updates (every two seconds by default), until you exit by typing **q**. **top** clearly highlights which processes are consuming the most CPU cycles and memory (using appropriate commands from within **top**).

The **first line** of the **top** output displays a quick summary of what is happening in the system, including:

* How long the system has been up
* How many users are logged on
* What is the load average.

The load average determines how busy the system is. A load average of 1.00 per CPU indicates a fully subscribed, but not overloaded, system. If the load average goes above this value, it indicates that processes are competing for CPU time. If the load average is very high, it might indicate that the system is having a problem, such as a runaway process (a process in a non-responding state).

The **second line** of the **top** output displays the total number of processes, the number of running, sleeping, stopped, and zombie processes. Comparing the number of running processes with the load average helps determine if the system has reached its capacity or perhaps a particular user is running too many processes. The stopped processes should be examined to see if everything is running correctly.

The **third line** of the **top**output indicates how the CPU time is being divided between the users (**us**) and the kernel (**sy**) by displaying the percentage of CPU time used for each.

The percentage of user jobs running at a lower priority (**niceness - ni**) is then listed. Idle mode (**id**) should be low if the load average is high, and vice versa. The percentage of jobs waiting (**wa**) for I/O is listed. Interrupts include the percentage of hardware (**hi**) vs. software interrupts (**si**). Steal time (**st**) is generally used with virtual machines, which has some of its idle CPU time taken for other uses.

The **fourth and fifth lines** of the **top**output indicate memory usage, which is divided in two categories:

* Physical memory (RAM) – displayed on line 4.
* Swap space – displayed on line 5.

Both categories display total memory, used memory, and free space.

You need to monitor memory usage very carefully to ensure good system performance. Once the physical memory is exhausted, the system starts using swap space (temporary storage space on the hard drive) as an extended memory pool, and since accessing disk is much slower than accessing memory, this will negatively affect system performance.

If the system starts using swap often, you can add more swap space. However, adding more physical memory should also be considered.

**Ch 9 – Process List of the top Output**

Each line in the process list of the **top**output displays information about a process. By default, processes are ordered by highest CPU usage. The following information about each process is displayed:

* Process Identification Number (PID)
* Process owner (USER)
* Priority (PR) and nice values (NI)
* Virtual (VIRT), physical (RES), and shared memory (SHR)
* Status (S)
* Percentage of CPU (%CPU) and memory (%MEM) used
* Execution time (TIME+)
* Command (COMMAND).

**Ch 9 – Interactive Keys with top**

Besides reporting information, **top** can be utilized interactively for monitoring and controlling processes. While **top** is running in a terminal window, you can enter single-letter commands to change its behavior. For example, you can view the top-ranked processes based on CPU or memory usage. If needed, you can alter the priorities of running processes or you can stop/kill a process.

The table lists what happens when pressing various keys when running **top**:

|  |  |
| --- | --- |
| **Command** | **Output** |
| **t** | Display or hide summary information (rows 2 and 3) |
| **m** | Display or hide memory information (rows 4 and 5) |
| **A** | Sort the process list by top resource consumers |
| **r** | Renice (change the priority of) a specific processes |
| **k** | Kill a specific process |
| **f** | Enter the **top** configuration screen |
| **o** | Interactively select a new sort order in the process list |

**Ch 9 – top Command**

Let's get started with using the top command.

To use the top command:

1. Open the command prompt.

2. Type top and press Enter.

Real-time updates of the running processes are displayed.

3. To hide the memory information, press m.

The memory information is hidden.

4. To display the memory information, press m again.

The memory information is displayed.

5. To renice a process, press r.

You are prompted to enter the PID of the process that you want to renice.

6. Enter the process’s PID and press Enter.

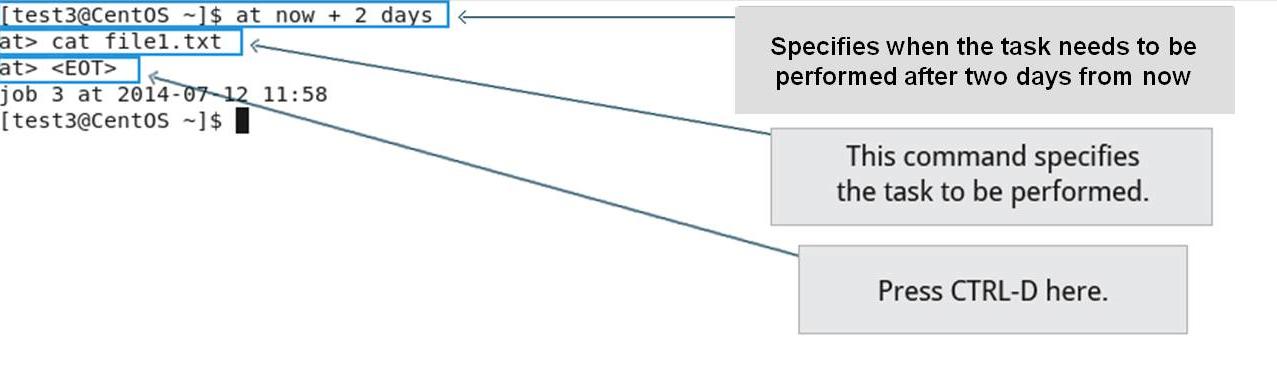
You are prompted to enter renice value.

7. Type the required value; for example, type 5 and press Enter.

The renice value of the process is updated.

**Ch 9 – Scheduling Future Processes Using at**

Suppose you need to perform a task on a specific day sometime in the future. However, you know you will be away from the machine on that day. How will you perform the task? You can use the **at** utility program to execute any non-interactive command at a specified time, as illustrated in the diagram:



**Scheduling Future Processes Using at**

**Ch 9 – cron**

**cron** is a time-based scheduling utility program. It can launch routine background jobs at specific times and/or days on an on-going basis. **cron**is driven by a configuration file called **/etc/crontab** (cron table), which contains the various shell commands that need to be run at the properly scheduled times. There are both system-wide **crontab** files and individual user-based ones. Each line of a **crontab** file represents a job, and is composed of a so-called **CRON** expression, followed by a shell command to execute.

The **crontab -e** command will open the crontab editor to edit existing jobs or to create new jobs. Each line of the **crontab** file will contain 6 fields:

|  |  |  |
| --- | --- | --- |
| **Field** | **Description** | **Values** |
| **MIN** | Minutes | 0 to 59 |
| **HOUR** | Hour field | 0 to 23 |
|  |  |  |
| **DOM** | Day of Month | 1-31 |
| **MON** | Month field | 1-12 |
| **DOW** | Day Of Week | 0-6 (0 = Sunday) |
| **CMD** | Command | Any command to be executed |

**Examples:**

* + - The entry **\* \* \* \* \* /usr/local/bin/execute/this/script.sh** will schedule a job to execute '**script.sh**' every minute of every hour of every day of the month, and every month and every day in the week.
    - The entry **30 08 10 06 \* /home/sysadmin/full-backup** will schedule a full-backup at 8.30 a.m., 10-June, irrespective of the day of the week.

**Ch 9 – sleep**

Sometimes, a command or job must be delayed or suspended. Suppose, for example, an application has read and processed the contents of a data file and then needs to save a report on a backup system. If the backup system is currently busy or not available, the application can be made to sleep (wait) until it can complete its work. Such a delay might be to mount the backup device and prepare it for writing.

**sleep**suspends execution for at least the specified period of time, which can be given as the number of seconds (the default), minutes, hours, or days. After that time has passed (or an interrupting signal has been received), execution will resume.

The syntax is:  
  
**sleep NUMBER[SUFFIX]...**  
  
where **SUFFIX** may be:

* **s** for seconds (the default)
* **m** for minutes
* **h** for hours
* **d** for days.

**sleep** and **at** are quite different; **sleep** delays execution for a specific period, while **at** starts execution at a later time.

**Lab 9.3 – Using at for Future Batch Processing**

Schedule a very simple task to run at a future time from now. This can be as simple as running **ls** or date and saving the output. You can use a time as short as one minute in the future.

Note that the command will run in the directory from which you schedule it with **at**.

Do this:

1. From a short bash script.
2. Interactively.

**Lab: Solution**

* 1. Create the file **testat.sh** containing:

**#!/bin/bash**

**date > /tmp/datestamp**

and then make it executable and queue it up with **at**:

**$ chmod +x testat.sh**

**$ at now + 1 minute -f testat.sh**

You can see if the job is queued up to run with **atq**:

**$ atq**

**17 Wed Apr 22 08:55:00 2015 a student**

Make sure the job actually ran:

**$ cat /tmp/datestamp**

**Wed Apr 22 08:55:00 CDT 2015**

What happens if you take the **/tmp/datestamp** out of the command? (Hint: type **mail** if not prompted to do so!)

* 1. Interactively it is basically the same procedure. Just queue up the job with:

**$ at now + 1 minute**

**at> date > /tmp/datestamp**

**CTRL-D**

**$ atq**

**Lab 9.4 – Scheduling a Periodic Task with cron**

Set up a cron job to do a simple task every day at 10:00am.

**Lab: Solution:**

Set up a **cron** job to do some simple task every day at 10 AM. Create a file named **mycrontab** with the following content:

**0 10 \* \* \* /tmp/myjob.sh**

and then create **/tmp/myjob.sh** containing:

**#!/bin/bash**

**echo Hello I am running $0 at $(date)**

and make it executable:

**$ chmod +x /tmp/myjob.sh**

Put it in the **crontab** system with:

**$ crontab mycrontab**

and verify it was loaded with:

**$ crontab -l**

**0 10 \* \* \* /tmp/myjob.sh**

**$ sudo ls -l /var/spool/cron/student**

**-rw------- 1 student student 25 Apr 22 09:59 /var/spool/cron/student**

**$ sudo cat /var/spool/cron/student**

**0 10 \* \* \* /tmp/myjob.sh**

Note you if don't really want this running every day, printing out messages like:

**Hello I am running /tmp/myjob.sh at Wed Apr 22 10:03:48 CDT 2015**

and mailing them to you, you can remove it with:

**$ crontab -r**

If the machine is not up at 10 AM on a given day, **anacron** will run the job at a suitable time.

**Ch 9 – Overview (Processes)**

You should now be able to:

* Describe what a process is and distinguish between types of processes.
* Enumerate process attributes.
* Manage processes using **ps** and **top**.
* Understand the use of load averages and other process metrics.
* Manipulate processes by putting them in background and restoring them to foreground.
* Use **at**, **cron**, and **sleep** to schedule processes in the future or pause them.

**Ch 9 – Summary**

You have completed Chapter 9. Let’s summarize the key concepts covered:

* Processes are used to perform various tasks on the system.
* Processes can be single-threaded or multi-threaded.
* Processes can be of different types, such as interactive and non-interactive.
* Every process has a unique identifier (PID) to enable the operating system to keep track of it.
* The nice value, or niceness, can be used to set priority.
* **ps**provides information about the currently running processes.
* You can use**top** to get constant real-time updates about overall system performance, as well as information about the processes running on the system.
* Load average indicates the amount of utilization the system is under at particular times.
* Linux supports background and foreground processing for a job.
* **at**executes any non-interactive command at a specified time.
* **cron**is used to schedule tasks that need to be performed at regular intervals.

**Ch 10 – File Operations**

By the end of this chapter, you should be able to:

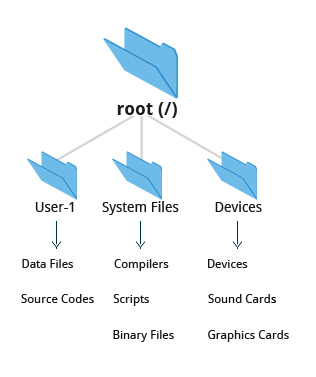
* Explore the filesystem and its hierarchy.
* Explain the filesystem architecture.
* Compare files and identify different file types.
* Back up and compress data.

**Ch 10 – Introduction to Filesystems**

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 filesystem 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 filesystem and is also sometimes referred to as the trunk, or simply denoted by **/**. The root directory is not the same as the root user. The hierarchical filesystem also contains other elements in the path (directory names), which are separated by forward slashes (**/**), as in **/usr/bin/emacs**, where the last element is the actual file name.

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



**Filesystem**

**Ch 10 – Filesystem Varieties**

Linux supports a number of native filesystem types, expressly created by Linux developers, such as:

* ext3
* ext4
* squashfs
* btrfs.

It also offers implementations of filesystems used on other alien operating systems, such as those from:

* Windows (ntfs, vfat)
* SGI (xfs)
* IBM (jfs)
* MacOS (hfs, hfs+).

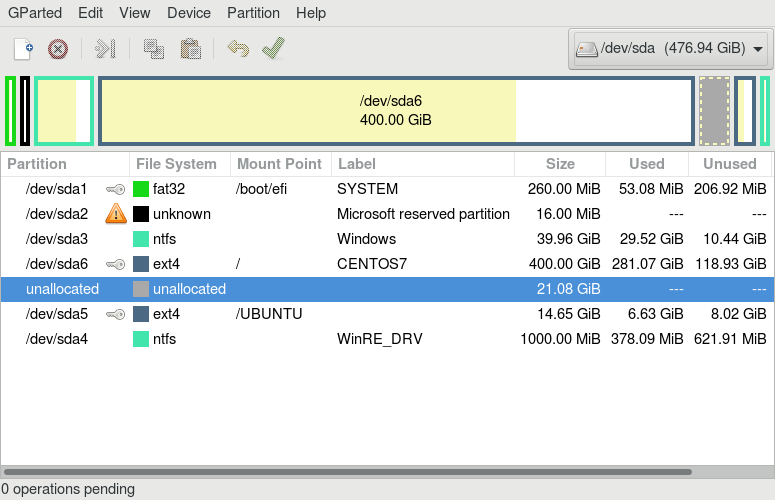
Many older, legacy filesystems, such as FAT, are also supported.

It is often the case that more than one filesystem type is used on a machine, based on considerations such as the size of files, how often they are modified, what kind of hardware they sit on and what kind of access speed is needed, etc. The most advanced filesystem types in common use are the journaling varieties: ext4, xfs, btrfs, and jfs. These have many state-of-the-art features and high performance, and are very hard to corrupt accidentally.

**Ch 10 – Linux Partitions**

As we discussed earlier, each filesystem on a Linux system occupies a hard disk partition. Partitions help to organize the contents of disks according to the kind and use of the data contained. 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 may be located on dedicated partitions. One advantage of this kind of isolation by type and variability is that when all available space on a particular partition is exhausted, the system may still operate normally.

The pictures shows the use of a utility called gparted, which displays the partition layout on a system which has three operating systems on it: Windows 10, RHEL 7, and Ubuntu 18.04.

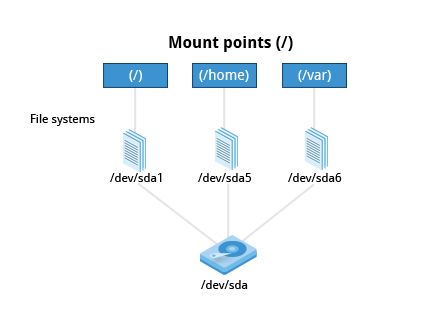


**Linux Partitions: gparted**

**Ch 10 – Mount Points**

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

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



**Mount Points**

**Ch 10 – Mounting and Unmounting**

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

**$ sudo mount /dev/sda5 /home**

will attach the filesystem contained in the disk partition associated with the **/dev/sda5** device node, into the filesystem tree at the **/home**mount point. There are other ways to specify the partition other than the device node, such as using the disk label or UUID.

To unmount the partition, the command would be:

**$ sudo umount /home**

Note the command is **umount**, not unmount! Only a root user (logged in as root, or using sudo) has the privilege to run these commands, unless the system has been otherwise configured.

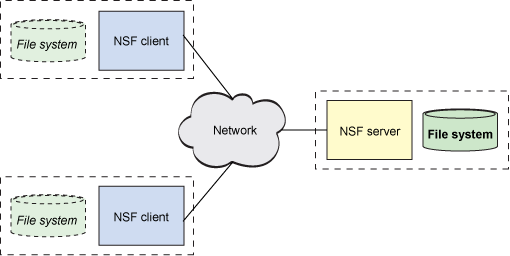
If you want it to be automatically available every time the system starts up, you need to edit **/etc/fstab** accordingly (the name is short for filesystem table). Looking at this file will show you the configuration of all pre-configured filesystems. **man fstab** will display how this file is used and how to configure it.

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

The command **df -Th** (disk-free) will display information about mounted filesystems, including the filesystem type, and usage statistics about currently used and available space.

**Ch 10 – NFS and Network Filesystems**

It is often necessary to share data across physical systems which may be either in the same location or anywhere that can be reached by the Internet. A network (also sometimes called distributed) filesystem may have all its data on one machine or have it spread out on more than one network node. A variety of different filesystems can be used locally on the individual machines; a network filesystem can be thought of as a grouping of lower level filesystems of varying types.



**The Client-Server Architecture of NFS**(retrieved from [www.ibm.com](https://www.ibm.com/developerworks/library/l-network-filesystems/))

 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.

The most common such filesystem is named simply NFS (the Network Filesystem). It has a very long history and was first developed by Sun Microsystems. Another common implementation is CIFS (also termed SAMBA), which has Microsoft roots. We will restrict our attention in what follows to NFS.

**Ch 10 – NFS on the Server**

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

On the server machine, NFS uses daemons (built-in networking and service processes in Linux) and other system servers are started at the command line by typing:

**$ sudo systemctl start nfs**

The text file **/etc/exports** contains the directories and permissions that a host is willing to share with other systems over NFS. A very simple 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 three 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. You can also restart NFS with **sudo systemctl restart** **nfs**, but this is heavier, as it halts NFS for a short while before starting it up again. To make sure the NFSservice starts whenever the system is booted, issue **sudo systemctl enable nfs**.

**Ch 10 – NFS on the Client**

On the client machine, if it is desired to have the remote filesystem 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 filesystem without a reboot or as a one-time mount by directly using the **mount** command:

**$ sudo 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. Furthermore, you may want to use the **nofail** option in **fstab** in case the NFS server is not live at boot.

**Lab 10.1 – Exploring Mounted Filesystems**

Issue the command:

**student:/tmp> cat /etc/fstab**

Now type:

**student:/tmp> mount**

Compare the results. What are the differences?

Find another way to see a list of the mounted filesystems, by examining the **/proc**pseudo-filesystem.

Typically, **mount** will show more filesystems mounted than are shown in **/etc/fstab** , which only lists those which are explicitly requested.

The system, however, will mount additional special filesystems required for normal operation, which are not enumerated in **/etc/fstab** .

Another way to show mounted filesystems is to type:

**student:/tmp> cat /proc/mounts**

which is essentially how the utility gets its information.

**Ch 10 –**

**Ch 10 –**

**Ch 10 –**

**Ch 10 – Overview of User Home Directories**

In this section, you will learn to identify and differentiate between the most important directories found in Linux. We start with ordinary users' home directory space.

Each user has a home directory, usually placed under **/home**. The **/root** ("slash-root") directory on modern Linux systems is no more than the home directory of the root user (or superuser, or system administrator account).

On multi-user systems, the **/home** directory infrastructure is often mounted as a separate filesystem 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/**

**Ch 10 – The /bin and /sbin Directories**

The **/bin** directory contains executable binaries, essential commands used to boot the system or in single-user mode, and essential commands required by all system users, such as cat, cp, ls, mv, ps, and rm.

Likewise, the **/sbin**directory is intended for essential binaries related to system administration, such as fsckand shutdown.To view a list of these programs, type:

**$ ls /bin /sbin**

Commands that are not essential (theoretically) for the system to boot or operate in single-user mode are placed in the **/usr/bin** and**/usr/sbin**directories. Historically, this was done so **/usr** could be mounted as a separate filesystem that could be mounted at a later stage of system startup or even over a network. However, nowadays most find this distinction is obsolete. In fact, many distributions have been discovered to be unable to boot with this separation, as this modality had not been used or tested for a long time.

Thus, on some of the newest Linux distributions **/usr/bin**and **/bin**are actually just symbolically linked together, as are **/usr/sbin**and **/sbin**.

**Ch 10 – The /proc Filesystem**

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

The **/proc** filesystem contains virtual files (files that exist only in memory) that permit viewing constantly changing kernel data. This filesystem contains files and directories that mimic kernel structures and configuration information. It does not 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**

The first example shows there is a directory for every process running on the system, which contains vital information about it. The second example shows a virtual directory that contains a lot of information about the entire system, in particular its hardware and configuration. The **/proc** filesystem is very useful because the information it reports is gathered only as needed and never needs storage on the disk.

**Ch 10 – 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:
  1. **/dev/sda1** (first partition on the first hard disk)
  2. **/dev/lp1** (second printer)
  3. **/dev/random**(a source of random numbers).

**Ch 10 – The /var Directory**

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**
* Temporary files: **/var/tmp**.

The **/var** directory may be put on its own filesystem 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**.



**The /var Directory**

**Ch 10 – The /etc Directory**

The **/etc** directory is the home for system configuration files. It contains no binary programs, although there are some executable scripts. For example, **/etc/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. While some distributions have historically had their own extensive infrastructure under **/etc** (for example, Red Hat and SUSE have used **/etc/sysconfig**), with the advent of systemd there is much more uniformity among distributions today.

Note that **/etc**is for system-wide configuration files and only the superuser can modify files there. User-specific configuration files are always found under their home directory.

**Ch 10 – 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 filesystem, 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.

**Ch 10 – The /lib and /lib64 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.5.9**.

Most of these are what is 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.

**Ch 10 – Removable media: the /media, /run and /mnt Directories**

One often uses removable media, such as USB drives, CDs and DVDs. To make the material accessible through the regular filesystem, it has to be mounted at a convenient location. Most Linux systems are configured so any removable media are automatically mounted when the system notices something has been plugged in.

While historically this was done under the **/media**directory, modern Linux distributions place these mount points under the **/run** directory. For example, a USB pen drive with a label "myusbdrive" for a user name "student" would be mounted at **/run/media/student/myusbdrive**.

The **/mnt** directory has been used since the early days of UNIX for temporarily mounting filesystems. These can be those on removable media, but more often might be network filesystems with NFS, which are not normally mounted. Or these can be temporary partitions, or so-called loopback filesystems, which are files which pretend to be partitions.

**Ch 10 – Additional Directories Under /:**

There are some additional directories to be found under the root directory:

|  |  |
| --- | --- |
| **Directory Name** | **Usage** |
| **/opt** | Optional application software packages |
| **/sys** | Virtual pseudo-filesystem 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 |

**Ch 10 – The /usr Directory**

The **/usr** directory tree contains theoretically 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/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 |

**Ch 10 – Comparing Files with diff**

Now that you know about the filesystem and its structure, let’s learn how to manage files and directories.

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

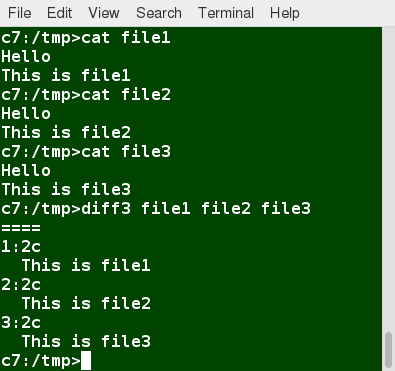
To compare two files, at the command prompt, type **diff [options] <filename1> <filename2>**. **diff** is meant to be used for text files; for binary files, one can use **cmp**.

**Ch 10 – 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**

The graphic shows the use of **diff3**.

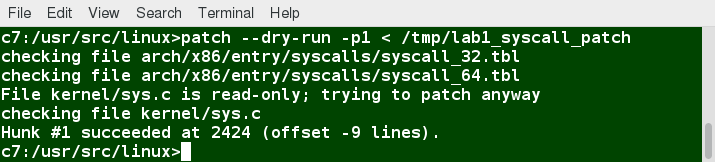


**Using diff3**

 Many modifications to source code and configuration files are distributed utilizing patches, which are applied, not surprisingly, 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 1000 lines, the patch file will be just a few lines long.



**Using patch**

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**.

**Ch 10 – Using the file Utility**

In Linux, a file's extension often does not categorize it the way it might in other operating systems. One cannot assume that a file named **file.txt** is a text file and not an executable program. In Linux, a filename 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.

**Lab 10.2 – Using diff and patch**

Linux and other open source communities often use the **patch** utility to disseminate modifications and updates. Here, we will give a practical introduction to using **diff** and **patch**.

It would be a good idea to read the man pages for both **patch** and **diff** to learn more about advanced options and techniques, that will help one to work more effectively with **patch**. In particular, the form of patches has a lot to do with whether they can be accepted in their submitted form.

1. Change to the **/tmp**directory.
2. Copy a text file to **/tmp**. For example, copy **/etc/group**to**/tmp**.
3. **dd** cannot only copy directly from raw disk devices, but from regular files as well. Remember, in Linux, everything is pretty much treated as a file. **dd** can also perform various conversions. For example, the **conv=ucase** option will convert all of the characters to upper-case characters. We will use **dd** to copy the text file to a new file in **/tmp**while converting characters to upper-case, as in: **student:/tmp> dd if=/tmp/group of=/tmp/GROUP conv=ucase**.
4. According to the man page for **patch**, the preferred options for preparing a patch with **diff** are **-Naur** when comparing two directory trees recursively. We will ignore the **-a** option, which means treat all files as text, since **patch** and **diff** should only be used on text files anyway. Since we are just comparing two files, we do not need to use the **N**or **r** options to **diff**, but we could use them anyway as it will not make a difference. Compare **group**and **GROUP** using **diff**, and prepare a proper patch file.
5. Use **patch** to patch the original file, **/tmp/group** , so its contents now match those of the modified file, **/tmp/GROUP** . You might try with the **--dry-run** option first!
6. Finally, to prove that your original file is now patched to be the same one with all upper-case characters, use **diff** on those two files. The files should be the same and you will not get any output from **diff**.

**Ch 10 – Solution 10.2**

For this exercise, you could use any text file, but we will use **/etc/group** as described.

1. **student:/tmp> cd /tmp**
2. **student:/tmp> cp /etc/group /tmp**
3. **student:/tmp> dd if=/tmp/group of=/tmp/GROUP conv=ucase**

**1+1 records in**

**1+1 records out**

**963 bytes (963 B) copied, 0.000456456 s, 2.1 MB/s**

1. **student:/tmp> diff -Nur group GROUP > patchfile**

**student:/tmp> cat patchfile**

**--- group 2015-04-17 11:03:26.710813740 -0500**

**+++ GROUP 2015-04-17 11:15:14.602813740 -0500**

**@@ -1,68 +1,68 @@**

**-root:x:0:**

**-daemon:x:1:**

**-bin:x:2:**

**-sys:x:3:**

**....**

**-libvirtd:x:127:student**

**-vboxsf:x:999:**

**+ROOT:X:0:**

**+DAEMON:X:1:**

**+BIN:X:2:**

**+SYS:X:3:**

**.....**

1. **student:/tmp> patch --dry-run group patchfile**

**checking file group**

**student:/tmp> patch group patchfile**

**patching file group**

1. Note you could have also done either of these two commands:

**student:/tmp> patch group < patchfile**

**student:/tmp> patch < patchfile**

1. **student:/tmp> diff group GROUP**

**student:/tmp>**

**Ch 10 – Backing Up Data**

There are many ways you can back up data or even your entire system. Basic ways to do so include the 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 filesystem 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 **someone@host**. The **someone@** 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

**Ch 10 – Using rsync**

**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** will be copied to **destinationfile**.

A good combination of options is shown in:

**$ rsync --progress -avrxH  --delete sourcedir destdir**

**Ch 10 – Compressing Data (gzipm, bzip2, xz)**

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 does not 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.

**Ch 10 – 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 |

**Ch 10 – 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 |

**Ch 10 – Disk-to-Disk Copying (dd)**

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**

**WARNING!**

Typing:

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

to make a copy of one disk onto another, will delete everything that previously existed on the second disk.

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 IBMhistory. Often, people joke that it means disk destroyer and other variants such as delete data!

**Lab 10.3 – Archiving (Backing Up) the Home Directory**

Archiving (or backing up) your files from time to time is essential good hygiene. You might type a command and thereby unintentionally clobber files you need and did not mean to alter.

Furthermore, while your hardware may be deemed fairly reliable, all devices do fail in some fashion eventually (even if it is just an unexpected power failure). Often, this happens at the worst possible time. Periodically backing up files is a good habit to get into.

It is, of course, important to do backups to external systems through a network, or onto external storage, such as an external drive or USB stick. Here, we will be making a back up archive on the same system, which is very useful, but won’t help if the drive fails catastrophically, or your computer is stolen or the building gets zapped by an asteroid or a fire.

First, using **tar**, back up all files and subdirectories under your home directory. Place the resulting tarball file in the **/tmp** directory, giving it the name **backup.tar** .

Second, accomplish the same task with **gzip** compression using the **-z** option to **tar**, creating **/tmp/backup.tar.gz**.

Compare the size of the two files (with **ls -l**).

**Solution – Lab 10.3**

To construct a tarball archive of your home directory you can do:

**student:/tmp> tar -cvf /tmp/backup.tar ~**

or equivalently

**student:/tmp> tar -cvf /tmp/backup.tar /home/student**

Note you can have omitted the **-** in the options with no change. In the following we will not bother using the **-v** option for verbose. To create archives with all three compression utilities:

**student:/tmp> tar zcf /tmp/backup.tar.gz ~**

**student:/tmp> tar jcf /tmp/backup.tar.bz2 ~**

**student:/tmp> tar Jcf /tmp/backup.tar.xz ~**

Comparing the sizes (first using the **-h** option to **ls** to make it **human**-readable):

**student@ubuntu:~student:/tmp> ls -lh /tmp/backup\***

**-rw-rw-r-- 1 student student 8.3M Apr 17 10:14 /tmp/backup2.tar.gz**

**-rw-rw-r-- 1 student student 12M Apr 17 10:13 /tmp/backup.tar**

**-rw-rw-r-- 1 student student 8.4M Apr 17 10:15 /tmp/backup.tar.bz2**

**-rw-rw-r-- 1 student student 8.3M Apr 17 10:14 /tmp/backup.tar.gz**

**-rw-rw-r-- 1 student student 8.2M Apr 17 10:15 /tmp/backup.tar.xz**

and then without it:

**student@ubuntu:~student:/tmp> ls -l /tmp/backup\***

**-rw-rw-r-- 1 student student 8686942 Apr 17 10:14 /tmp/backup2.tar.gz**

**-rw-rw-r-- 1 student student 12226560 Apr 17 10:13 /tmp/backup.tar**

**-rw-rw-r-- 1 student student 8720491 Apr 17 10:15 /tmp/backup.tar.bz2**

**-rw-rw-r-- 1 student student 8686929 Apr 17 10:14 /tmp/backup.tar.gz**

**-rw-rw-r-- 1 student student 8551064 Apr 17 10:15 /tmp/backup.tar.xz**

Note in this case there is not much difference in the different archiving methods, but this particular directory was a bad choice because it already contained a lot of compressed files. A somewhat better example involving more text files:

**student:/tmp> tar cf /tmp/doc.tar /usr/share/doc**

**student:/tmp> tar zcf /tmp/doc.tar.gz /usr/share/doc**

**student:/tmp> tar jcf /tmp/doc.tar.bz2 /usr/share/doc**

**student:/tmp> tar Jcf /tmp/doc.tar.xz /usr/share/doc**

**student:/tmp> ls -lh /tmp/doc.tar\***

**-rw-rw-r-- 1 student student 85M Apr 17 10:34 /tmp/doc.tar**

**-rw-rw-r-- 1 student student 31M Apr 17 10:35 /tmp/doc.tar.bz2**

**-rw-rw-r-- 1 student student 34M Apr 17 10:34 /tmp/doc.tar.gz**

**-rw-rw-r-- 1 student student 28M Apr 17 10:36 /tmp/doc.tar.xz**

which shows **xz** did best, followed by **bz2** and then **gz**. You may have noticed, however, the inverse relationship between the size reduction of the compression and how long it took!

**Ch 10 – Overview**

You should now be able to:

* Explore the filesystem and its hierarchy.
* Explain the filesystem architecture.
* Compare files and identify different file types.
* Back up and compress data.

**Ch 10 – Summary**

You have completed Chapter 10. Let’s summarize the key concepts covered:

* The filesystem tree starts at what is often called the root directory (or trunk, or **/**).
* The  Filesystem Hierarchy Standard (FHS) provides Linux developers and system administrators a standard directory structure for the filesystem.
* Partitions help to segregate files according to usage, ownership, and type.
* Filesystems can be mounted anywhere on the main filesystem tree at a mount point. Automatic filesystem mounting can be set up by editing **/etc/fstab**.
* NFS (Network File System) is a useful method for sharing files and data through the network systems.
* Filesystems like **/proc** are called pseudo filesystems because they exist only in memory.
* **/root** (slash-root) is the home directory for the root user.
* **/var** may be put in its own filesystem 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.

**Ch 11 – Text Editors**

By the end of this chapter, you should be familiar with:

* How to create and edit files using the available Linux text editors.
* nano, a simple text-based editor.
* gedit, a simple graphical editor.
* vi and emacs,two advanced editors with both text-based and graphical interfaces.

**Ch 11 – Overview of Text Editors in Linux**

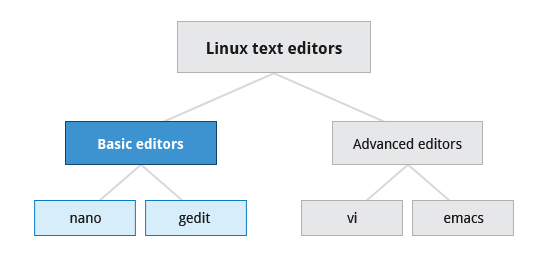
At some point, you will need to manually edit text files. You might be composing an email off-line, writing a script to be used for bash or other command interpreters, altering a system or application configuration file, or developing source code for a programming language such as C or Java.

Linux Administrators quite often sidestep the text editors, by using graphical utilities for creating and modifying system configuration files. However, this can be more laborious than directly using a text editor, and be more limited in capability. Note that word processing applications including those 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.



**Text Editors in Linux**

**Ch 11 – Creating Files Without Using an Editor**

ometimes, 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 will 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**

Note that while a single greater-than sign (**>**) will send the output of a command to a file, two of them (**>>**) will append the 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 techniques produce a file with the following lines in it:  
  
**line one  
line two  
line three**

and are extremely useful when employed by scripts.

**Ch 11 – nano and gedit**

There are some text editors that are pretty obvious; they require no particular experience to learn and are actually quite capable, if not robust. A particularly easy to use one is the text terminal-based editor nano. Just invoke nanoby 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 kate are also supported by KDE.

**Ch 11 – 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 does not 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.

**Ch 11 – 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 does not exist, it will be created.

Using gedit is pretty straightforward and does not require much training. Its interface is composed of quite familiar elements.

**Lab 11.1 – Using nano**

Using nano, we are going to create a file named **myname.txt**, and have it include your name on the first line, and the current date on the last line. To do this:

1. Start nano by typing **nano myfile.txt**.
2. Add your name in the first line of the file.
3. Add the date in the last line of the file.
4. Close the file.

**Lab 11.2 – Using gedit**

Using gedit, we are going to either create or reuse a file named **myname.txt**, and have it include your street address on the second line and the name of your city on the last line:

1. Start gedit by typing **gedit myfile.txt**.
2. Add your street address in the second line of the file.
3. Add the name of your city on the last line of the file.
4. Close the file.

**Ch 11 – vi and emacs (advanced editors)**

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 graphical interface forms with extended capabilities; these may be friendlier for a less experienced user. While viand emacs can have significantly steep learning curves for new users, they are extremely efficient when one has learned how to use them.

**Ch 11 – 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 do not 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 do not 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.

**Ch 11 – vimtutor**

Typing **vimtutor** launches a short but very comprehensive tutorial for those who want to learn their first vicommands. 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 vicommands because there are always more optimal ways to do things in vi with less typing.

**Ch 11 –**

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 inCommand 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. |

**Ch 11 – 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 |

**Ch 11 – vi**

Let's get started on using modes and cursor movements in vi.

1. Open vi by typing vi followed by the filename.

Vi opens in Command mode.

2. Type i to enter Insert mode.

Vi's Insert mode is displayed.

3. Type the following sentences: The quick brown fox jumped over the lazy dog. Nobody expects the Spanish Inquisition!

4. To exit Insert mode and switch to Command mode, press ESC

5. To exit vi and save the file, type :wq and press Enter.

6. Open the recently saved file and place the cursor on the letter k in the word quick in the first sentence.

To move the cursor four characters to the left, type h four times.

The cursor is moved to letter q of the word quick.

7. To move the cursor to the next line, type j.

8. To move the cursor to the beginning of the next word, type w.

The cursor is moved to the beginning of the word expects.

9. To move the cursor to the end, type $.

The cursor is moved to the end of the second sentence.

10. Type i to enter Insert mode.

Vi insert mode is displayed.

11. To insert text at the end of the second sentence, type A and type History.

The word History is displayed to the end of the sentence.

12. To exit Insert mode and switch to Command mode, press ESC

13. To exit vi and save the file type :wq and press Enter.

**Ch 11 – Using External Commands**

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

Typing **:!**executes a command from within vi. The command follows the exclamation point. This technique is 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.

1. Open vi editor by typing VI followed by the filename.

vi opens in Command mode.

2. Type i to enter Insert mode.

Vi's Insert mode is displayed.

3. Type the following sentences.

Nobody expects the Spanish Inquisition! Nobody expects the 2020 revolution!

4. To exit Insert mode and switch to Command mode, press Esc.

5. To write and quit the file, type :wq.

The file is updated with the changes and closed.

6. Open vi by typing VI followed by the filename.

7. To count the words in the current file, type :!wc %.

The word count is displayed.

8. Press ENTER to continue editing.

9. To quit if no edits were made in the file, type :q.

The file is closed.

10. Open vi by typing VI followed by filename.

11. To quit without saving the file, type :q!.

The file is closed without saving the changes.

**Ch 11 – 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. For example, it can be used for email, debugging, etc.

Rather than having different modes for command and insert, like vi, emacs uses the **CTRL** and Meta**(Alt**or **Esc)** keys for special commands.

We will now demonstrate some of the operations you can do with Emacs in your daily work.

So first, let's get a file to edit. So, working in the /tmp directory, let's get a copy of /etc/passwd and bring it over here [cp /etc/passwd].

And then, in order to work on it, I just have to say "emacs passwd &" the name of the file.

Now, the default font here is a little small. So, I'll play a trick and hit ctrl + the middle button of the mouse to control the size of the font,

and then I'll make the window a little bigger.

Okay. So, we've done that. Now, suppose I want to look for a certain string, let's say ftp, I would hit Ctrl-s and type into the very bottom line here ftp.

So you can see down here.

Then you'll see it brought me to ftp. I'll go back to the beginning of the line by hitting Ctrl-a ,

and then, let's say I want to change all occurrences of the string ftp to something else, I hit Escape % mark, and then I'll say ftp again in the bottom line.

And then, let's just do it backwards in capitals PTF. You see, I hit Space, it does the first one, Space, it does the second one, Space, it does the third one.

If I had hit an exclamation point, it would have done all of them in the entire file.

Suppose I want to remove a line. I can just hit Ctrl-K and it's gone. Ctrl-K is gone again.

Suppose I want to remove a range of lines. I hit Ctrl-space and then I go down a few lines with either the arrow key or Ctrl- N, I hit Ctrl-W and they're gone.

If I want to move them further on in the file, I go down a few lines and I hit Ctrl-Y for yank and they're back in there.

One nice thing I can do with Emacs is open up multiple windows at the same time.

So, let me hit Ctrl-X-2 and now I've got two windows and I can switch to the bottom window by hitting Ctrl-X-O, for other,

and then I hit Ctrl-X-F and I could put a different file in that buffer.

So, let's say I put it in /etc/group. Ok, once again, I'll make the font a little bigger by hitting Ctrl and the middle button on the mouse. ok

And you'll notice the bottom line it's actually right protected, so I can't really delete anything because it belongs to root.

If I try to delete this line, it won't let me do it. It says buffer is read-only.

But, in general, like if I had two files with the same permissions, I could cut and paste and go from one to the other.

To get back to just one [window], I can do Ctrl-X-1, and I have only this. But if I really wanted the other window, I can hit Ctrl-X-B, and I'm back in the passwd file.

If I want to rewrite it, I can hit Ctrl-X-W... Ctrl-X Ctrl-W and that would let me write it as a different name. So I'll say passwdrevised, and it's there.

If I want to terminate the program, I say Ctrl-X-S. Make sure things have been saved, and then, Ctrl-X-C, and I'm done.

So you can see we use the Ctrl key quite a bit in emacs. The position most keyboards put it in these days is a little unnatural, all the way at the bottom left or right.

So veteran Emacs users tend to remap the keyboard so that the caps lock key also works as Ctrl.

So, that's some of the basic operations you would do in day-to-day use of Emacs.

**Ch 11 – Learning Objectives (Review)**

You should now be familiar with:

* How to create and edit files using the available Linux text editors.
* nano, a simple text-based editor.
* gedit, a simple graphical editor.
* vi and emacs,two advanced editors with both text-based and graphical interfaces.

You have completed Chapter 11. Let’s summarize the key concepts covered:

* Text editors (rather than word processing programs) are used quite often in Linux, for tasks such as 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 viare 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 emacstutorial type **Ctl-h** and then **t**from within emacs**.**
* **vi** has three modes: *Command*, *Insert*, and *Line*. emacshas 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.

**Ch 12 – Learning Objectives**

By the end of this chapter, you should be able to:

* Use and configure user accounts and user groups.
* Use and set environment variables.
* Use the previous shell command history.
* Use keyboard shortcuts.
* Use and define aliases.
* Use and set file permissions and ownership.

**Ch 12 – Identifying the Current User**

As you know, Linux is a multi-user operating system, meaning more than one user can log on at the same time.

* To identify the current user, type **whoami**
* To list the currently logged-on users, type **who**.

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

**Ch 12 – User Startup Files**

In Linux, the command shell program (generally bash) uses one or more startup files to configure the user 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 prompt
* Defining command line shortcuts and aliases
* Setting the default text editor
* Setting the path for where to find executable programs.

**Ch 12 – Order of the Startup Files**

The standard prescription is that 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:

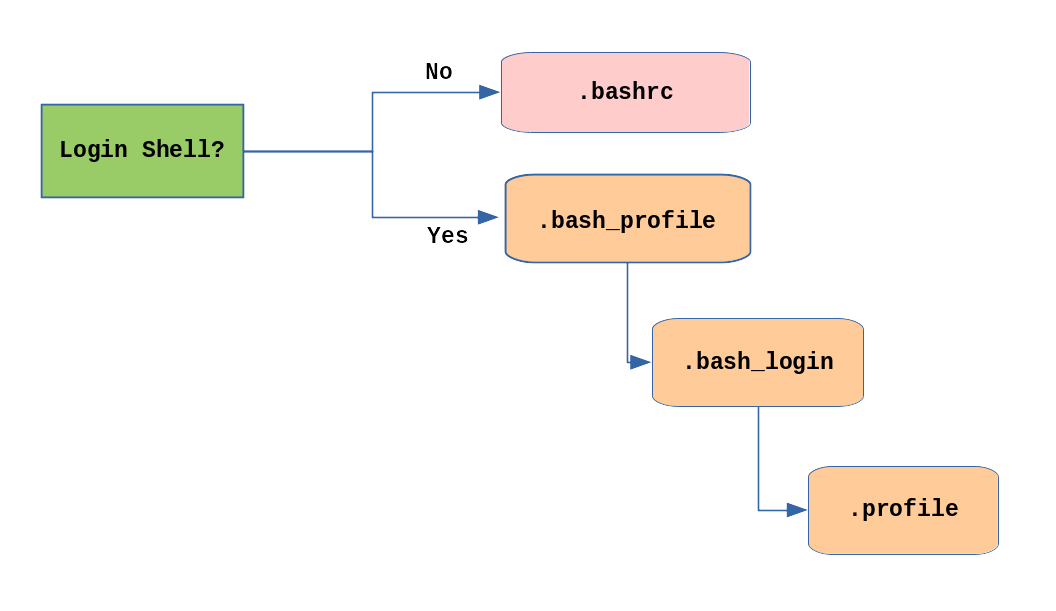
1. **~/.bash\_profile**
2. **~/.bash\_login**
3. **~/.profile**

where  **~/.**denotes the user's home directory. 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 a file named **~/.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.

Most commonly, users only fiddle with **~/.bashrc**, as it is invoked every time a new command line shell initiates, or another program is launched from a terminal window, while the other files are read and executed only when the user first logs onto the system.

Recent distributions sometimes do not even have **.bash\_profile**or **.bash\_login**(Ubuntu) and some just have it do little more than include **.bashrc**.



**Order of the Startup Files**

**Ch 12 – 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. **unalias**removes an alias.

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.

**Ch 12 – Basics of Users and Groups**

All Linux users are assigned a unique user ID (uid), which is just an integer; normal users start with a uid of 1000 or greater.

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.

Users also have one or more group IDs (gid), including a default one which is the same as the user ID. These numbers are associated with names through the files **/etc/passwd** and **/etc/group**. 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.

For example, **/etc/passwd**might contain **george:x:1002:1002:George Metesky:/home/george:/bin/bash** and **/etc/group**might contain **george:x:1002**.

**Ch 12 – 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**bjmoose** would be done with:

**$ sudo useradd bjmoose**

Note that for openSUSE, **useradd**is not in the normal user's PATH, so the command should be:

**$ sudo /usr/sbin/useradd bjmoose**

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

**bjmoose:x:1002:1002::/home/bjmoose:/bin/bash**

and sets the default shell to **/bin/bash**. Removing a user account is as easy as typing **userdel bjmoose**. However, this will leave the**/home/bjmoose** 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=1002(bjmoose) gid=1002(bjmoose) groups=106(fuse),1002(bjmoose)**

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

**Ch 12 – Adding and Removing Groups**

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 rjsquirrel**  
**bjmoose : rjsquirrel**

and then add the new group:

**$ sudo /usr/sbin/usermod -a -G anewgroup rjsquirrel**

**$ groups rjsquirrel**  
**rjsquirrel: rjsquirrel anewgroup**

These utilities update **/etc/group** as necessary. Make sure to use the **-a**option, for append, so as to avoid removing already existing groups. **groupmod** can be used to change group properties, such as the Group ID (gid) with the **-g** option or its name with then **-n** option.

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

**$ sudo /usr/sbin/usermod -G rjsquirrel rjsquirrel**

**$ groups rjsquirrel  
rjsquirrel : rjsquirrel**

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

**Ch 12 – 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:

* Only on a temporary basis
* Only for a specific subset of commands.

**Ch 12 – 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 *Chapter 18: Local Security Principles*, we will describe and compare **su**and **sudo**in detail.

**Ch 12 – Elevating to root Account**

**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.

**Lab 12 – Deploying aliases**

Typing long commands and filenames over and over gain gets rather tedious, and leads to a lot of trivial errors, such as typos.

Deploying aliases allows us to define shortcuts to alleviate the pain of all of this typing.

Suppose you are a member of a project team that works in a common, shared directory for your project. This directory is located in **/home/staff/RandD/projects/projectX/src** .

When you are working on Project X, you often need to create and modify your files in this directory. It does not take too long before typing in: **cd/home/staff/RandD/projects/projectX/src**gets tedious.

Define and use an alias named "projx" to do the above **cd** command for you.

The **alias** line would look like this:

**student:/tmp> alias projx='cd /home/staff/RandD/projects/projectX/src'**

Note you can use double quotes instead of single quotes, or use no quotes at all since their is no white space in your defined alias. All you have to do now to change to the directory is

**student:/tmp> projx**

To make the alias persistent, just place it in your **$HOME/.bashrc** file.

**Ch 12 – Environment Variables**

Environment variables are quantities that have specific values which may be utilized by the command shell, such as bash,  or other utilities and applications. Some environment variables are given preset values by the system (which can usually be overridden), while others are set directly by the user, either at the command line or within startup and other scripts.

An environment variable is actually just 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.

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** |

You can also set environment variables to be fed as a one shot to a command as in:

**$ SDIRS=s\_0\* KROOT=/lib/modules/$(uname -r)/build make modules\_install**

which feeds the values of the**SDIRS**and**KROOT**environment variables to the command **make modules\_install**.

**Ch 12 – 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. |

**Ch 12 – 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 a null directory before the first colon (**:**). Similarly, for **path1::path2** there is a null directory between **path1** and **path2**.

To prefix a private **bin** directory to your path:

**$ export PATH=$HOME/bin:$PATH**  
**$ echo $PATH**  
**/home/student/bin:/usr/local/bin:/usr/bin:/bin/usr**

**Ch 12 – 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  
$**

**Ch 12 – The PS1 Variable and the Command Line Prompt**

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$ '**  
**student@example.com:~$ # new prompt**  
**student@example.com:~$**

To revert the changes:  
  
**student@example.com:~$ export PS1='$ '**  
**$**

An 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=$OLD\_PS1**  
**$**

**Lab 12 – Adding /tmp to Your Path**

Create a small file **/tmp/ls**, which contains just the line:

**echo HELLO, this is the phony ls program.**

Then, make it executable by doing:

**$ chmod +x /tmp/ls**

* 1. Append **/tmp** to your path, so it is searched only after your usual path is considered. Type **ls** and see which program is run: **/bin/ls** or **/tmp/ls**?
  2. Pre-pend **/tmp** to your path, so it is searched before your usual path is considered. Once again, type**ls** and see which program is run: **/bin/ls** or **/tmp/ls**?

What are the security considerations in altering the path this way?

First create the phony ls program, using an editor, or just simply doing:

**student:/tmp>echo "echo HELLO, this is the phony ls program." > /tmp/ls**

**student:/tmp>chmod +x /tmp/ls**

For the next two steps it is a good idea to work in another terminal window, or just start a new shell, so the changes do not persist on later issued commands. You can start a new shell by just typing **bash**.

1. **student:/tmp>bash**

**student:/tmp>PATH=$PATH:/tmp**

**student:/tmp>ls /usr**

**bin etc games include lib lib64 libexec local sbin share src tmp**

**student:/tmp>exit**

1. **student:/tmp>bash**

**student:/tmp>PATH=/tmp:$PATH**

**student:/tmp>ls /usr**

**HELLO, this is the phony ls program.**

**student:/tmp>exit**

Note the second form is a very dangerous thing to do, and is a trivial way to insert a **Trojan Horse** program; if someone can put a malicious program in **/tmp**, they can trick you into running it accidentally.

**Lab 12 – Changing the Command Line Prompt**

It is nice to have your current working directory as part of your prompt so that a quick glance will give you some information without typing **pwd**every time.

If you often work on multiple computers, especially if you network from one into another with ssh, it is very convenient to have the computer name be part of your prompt.

* + 1. Put your current working directory in your command line prompt.
    2. Put your computer (machine) name in your prompt.
    3. Put both your current directory and computer name in your prompt.

How can you make this persistent, so that whenever you start a bash command shell, this is your prompt?

1. **$ echo $PWD**

**/tmp**

**$ PS1='\w>'**

**/tmp>**

1. **PS1='\h>'**

**student>**

1. **PS1='\h:\w>'**

**student:/tmp>**

**Ch 12 – 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**. If you have multiple terminals open, the commands typed in each session are not saved until the session terminates.

**Ch 12 – Using History Environment Variables**

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

* + - **HISTFILE**The location of the history file.
    - **HISTFILESIZE**The maximum number of lines in the history file (default 500).
    - **HISTSIZE**   
      The maximum number of commands in the history file.
    - **HISTCONTROL**How commands are stored.
    - **HISTIGNORE**Which command lines can be unsaved.

For a complete description of the use of these environment variables, see man bash.

**Ch 12 – Finding and Using Previous Commands**

Specific keys to perform various tasks:

|  |  |
| --- | --- |
| **Key** | **Usage** |
| **Up**/**Down**arrow keys | 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 have 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)  
**$**

**Ch 12 – 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**  
**$**

**Ch 12 – Keyboard Shortcuts**

You can use keyboard shortcuts to perform different tasks quickly. The table lists some of these keyboard shortcuts and their uses. Note the case of the "hotkey" does not matter, e.g. doing **CTRL-a** is the same as doing **CTRL-A** .

|  |  |
| --- | --- |
| **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 |

**Ch 12 – Command History**

The **history** command is the way to display the commands you have typed in:

**student:/tmp> history**

**1 cd /**

**2 ls**

**3 cd**

**4 pwd**

**5 echo $SHELL**

**6 ls /var/**

**7 ls /usr/bin**

**8 ls /usr/local/bin**

**9 man fstab**

**10 ls**

**. . .**

In order to re-run a previous command, you have a few choices. Letâ€™s say that you wanted to re-run the **man** command you ran way back when you first logged in. You could type:

**student:/tmp> !9**

to re-run the command listed as **#9** . If this was the only **man** command that you typed in, you could also type:

**student:/tmp> !man**

now that you remember the command name that you typed. Finally, if you had typed a few man commands, you could use **CTRL-R** to search backward in your history to find the specific man command that you want to re-run, and then just hit **Return** to execute it.

**Ch 12 – 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) |

**Ch 12 – 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 somefile  
-rw-rw-r-- 1 student student 1601 Mar 9 15:04 somefile  
$ chmod uo+x,g-w somefile  
$ ls -l somefile  
-rwxr--r-x 1 student student 1601 Mar 9 15:04 somefile**

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:

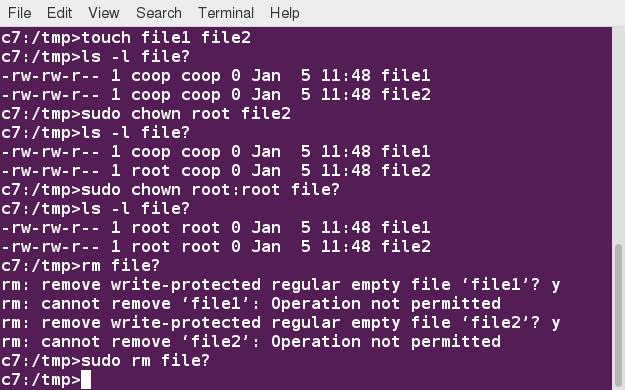
**$ chmod 755 somefile**  
**$ ls -l somefile**  
**-rwxr-xr-x 1 student student 1601 Mar 9 15:04 somefile**

**Ch 12 – Example of chown**

Let's see an example of changing file ownership using **chown**,as shown in the screenshot to the right. First, we create two empty files using **touch.**

Notice it requires sudo to change the owner of **file2** to root. The second **chown** command changes both owner and group at the same time!

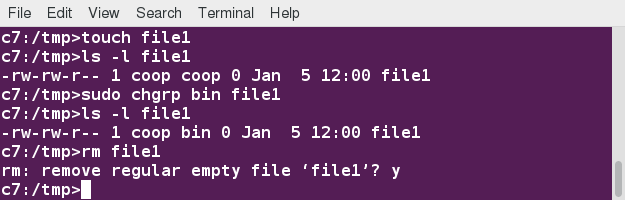
Finally, only the superuser can remove the files.



**chown**

**Ch 12 – Example of chgrp**

Now, let’s see an example of changing the group ownership using **chgrp**:



**chgrp**

**Ch 12 – Learning Objectives (Review)**

You should now be able to:

* Use and configure user accounts and user groups.
* Use and set environment variables.
* Use the previous shell command history.
* Use keyboard shortcuts.
* Use and define aliases.
* Use and set file permissions and ownership.

**Ch 12 – Summary**

You have completed Chapter 12. Let's summarize the key concepts covered:

* Linux is a multi-user 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**.

**Ch 13 – Manipulating Text**

In this lesson we are going to work ***on editing, sorting, manipulating, and searching*** strings of text.

By the end of this chapter, you should be able to:

* Display and append to file contents using **cat** and **echo**.
* Edit and print file contents using **sed** and **awk**.
* Search for patterns using **grep**.
* Use multiple other utilities for file and text manipulation.

**Ch 13 – Command Line Tools for Manipulating Text Files**

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.

**$ tail /etc/services**

**$ cat /etc/services | sort | uniq | awk ‘{print $1}’ | tail**

**Ch 13 – 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. However, the main purpose of **cat** is often to combine (concatenate) multiple files together. You can perform the actions listed in the table using **cat**.

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**

|  |  |
| --- | --- |
| **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 |

1. To create the test1 file, type cat > test1 and press the Enter key.

2. Type the sentence This is Sample File1 and press the Enter key and press CTRL-D to save the file.

3. To create the test2 file, type cat > test2 and press the Enter key.

4. Type the sentence This is Sample File2 and press the Enter key and press CTRL-D to save the file.

5. To create the test3 file, type cat > test3 and press the Enter key.

6. Type the sentence This is Sample File3 and press the Enter key and press CTRL-D to save the file.

7. To combine the files test1 and test2 into the file newtest, type cat test1 test2 > newtest and press the Enter key.

8. To view the newtest file, type cat newtest and press the Enter key. The file newtest is displayed at the command prompt.

9. To append a file to the end of newtest file, type cat test3 >> newtest and press the Enter key.

10. To view the newtest file, type cat newtest and press the Enter key. The file newtest is displayed at the command prompt.

11. To append the newtest file, type cat >> newtest and press the Enter key.

12. Type the sentence The Sample File3 is appended with another line press the Enter key and press CTRL-D.

You will return to the command prompt

13. At the command prompt, type cat newtest and press the Enter key.

The newtest file is displayed.

**Ch 13 – 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 |

**Ch 13 – Working with Large Files – less | head | tail**

**$ less somefile**

**$ cat somefile | less**

If you want to print the first 5 lines from **grub.cfg**, use the following command:

**$ head –n 5 grub.cfg**

**$ tail –n 15 somefile.log**

To continually monitor new output in a growing log file:

**$ tail -f somefile.log**

**Ch 13 – Viewing Compressed Files**

When working with compressed files, many standard commands cannot be used directly. For many commonly-used file and text manipulation programs, there is also a version especially designed to work directly with compressed files. These associated utilities have the letter "z" prefixed to their name. For example, we have utility programs such as **zcat**, **zless**, **zdiff** and **zgrep**.

Here is a table listing some **z** family commands:

|  |  |
| --- | --- |
| **Command** | **Description** |
| **$ zcat compressed-file.txt.gz** | To view a compressed file |
| **$ zless somefile.gz** or **$ zmore somefile.gz** | To page through a compressed file |
| **$ zgrep -i less somefile.gz** | To search inside a compressed file |
| **$ zdiff file1.txt.gz file2.txt.gz** | To compare two compressed files |

Note that if you run **zless** on an uncompressed file, it will still work and ignore the decompression stage. There are also equivalent utility programs for other compression methods besides **gzip**, for example, we have **bzcat** and **bzless** associated with **bzip2**,and **xzcat**and **xzless**associated with **xz**.

**Ch 13 – 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 comprehensive scripting languages such as Python and perl, rather than use **sed** and **awk** (and some other utilities we will 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 situations (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** 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.

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).

You can invoke **sed**using commands like those listed in the accompanying 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. It is unnecessary if you only have one operation invoked.

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 every line |
| **sed s/pattern/replace\_string/g file** | Substitute all string occurrences in every 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 **–i** option and then replace the file yourself, as shown in the following example:

**$ sed s/pattern/replace\_string/g file1 > 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** 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 Aho, Peter Weinberger, and Brian Kernighan.

**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:

$ awk –F: ‘{print “name: “$1” shell:” $7}’ \

/etc/passwd | head -10

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.

|  |  |
| --- | --- |
| **Command** | **Usage** |
| **awk ‘command’  file** | Specify a command directly at the command line |
| **awk -f scriptfile file** | Specify a file that contains the script to be executed |

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 $7 }' /etc/passwd** | Print first and seventh field of every line |

**Lab 13 –**

Search for all instances of the user command interpreter (shell) equal to **/sbin/nologin** in **/etc/passwd** and replace them with **/bin/bash**.

To get output on standard out (terminal screen):

**student:/tmp> sed s/'\/sbin\/nologin'/'\/bin\/bash'/g /etc/passwd**

or to direct to a file:

**student:/tmp> sed s/'\/sbin\/nologin'/'\/bin\/bash'/g /etc/passwd > passwd\_new**

Note this is kind of painful and obscure because we are trying to use the forward slash ( **/** ) as both a string and a delimiter between fields. One can do instead:

**student:/tmp> sed s:'/sbin/nologin':'/bin/bash':g /etc/passwd**

where we have used the colon ( **:** ) as the delimiter instead. (You are free to choose your delimiting character!) In fact when doing this we do not even need the single quotes:

**student:/tmp> sed s:/sbin/nologin:/bin/bash:g /etc/passwd**

works just fine.

**Ch 13 – File Manipulation Utilities - sort, uniq, paste, join & split**

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**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, according to the characters at the beginning of each line |
| **cat file1 file2 | sort** | Combine the two files, then sort the lines and display the output on the terminal |
| **sort -r <filename>** | Sort the lines in reverse order |
| **sort -k 3 <filename>** | Sort the lines by the 3rd field on each line instead of the beginning |

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**removes duplicate consecutive lines in a text file and is useful for simplifying the text display.

Because **uniq**requires that the duplicate entries must be consecutive, one often runs sortfirst and then pipes the output into **uniq**;if sort is used with the **-u**option, it can do all this in one step.

To remove duplicate entries from multiple files at once, use the following command:   
  
**sort file1 file2 | uniq > file3**

OR

**sort -u file1 file2 > file3**

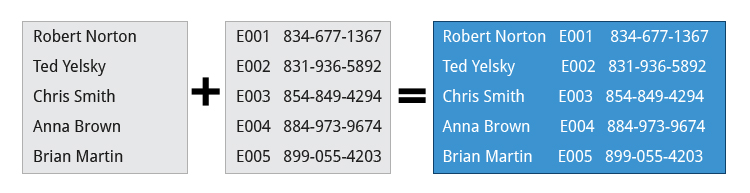
To count the number of duplicate entries, use the following command:   
  
**uniq -c filename**

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**

**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.

**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 1000-line segments. The original file remains unchanged, and a 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>**.

We will apply **split** to an American-English dictionary file of over 99,000 lines:

**$ wc -l american-english  
99171 american-english**

where we have used **wc**(word count, 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 100 equal-sized segments named **'dictionaryxx**. The last one will of course be somewhat smaller.

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

**Ch 13 – 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 |

**Ch 13 – strings**

**strings**is used to extract all printable character strings found in the file or files given as arguments. It is useful in locating human-readable content embedded in binary files; for text files one can just use **grep**.

For example, to search for the string **my\_string**in a spreadsheet:  
  
**$ strings book1.xls | grep my\_string**

**Ch 13 – tr**

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

|  |  |
| --- | --- |
| **Option** | **Description** |
| **–l** | Displays the number of lines |
| **-c** | Displays the number of bytes |
| **-w** | Displays the number of words |

By default, all three of these options are active.

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

**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" " -f3** and press the **Enter** key.

**Ch 13 – Summary**

You should now be able to:

* Display and append to file contents using **cat** and **echo**.
* Edit and print file contents using **sed** and **awk**.
* Search for patterns using **grep**.
* Use multiple other utilities for file and text manipulation.

You have completed Chapter 13. 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 an 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. It 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** 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.

**Ch 14 – Network Operations**

By the end of this chapter, you should be able to:

* Explain basic networking concepts, including types of networks and addressing issues.
* Configure network interfaces and use basic networking utilities, such as **ifconfig**, **ip**, **ping**, **route** and **traceroute**.
* Use graphical and non-graphical browsers, such as Lynx, w3m, Firefox, Chrome and Epiphany.
* Transfer files to and from clients and servers using both graphical and text mode applications, such as Filezilla, ftp, sftp, curl and wget.

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.

**Ch 14 – 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 small 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.

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 limitations inherent in the older standard and furnish many more possible 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 considered inadequate for meeting future needs because the number of devices available on the global network has increased enormously in recent years.

IPv6 uses 128-bits for addresses; this allows for 3.4 X 1038 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 can be complex to migrate to IPv6; the two protocols do not always inter-operate well. Thus, moving equipment and addresses to IPv6 requires significant effort and has not been quite as fast as was originally intended. We will discuss IPv4 more than IPv6 as you are more likely to deal with it.

A 32-bit IPv4 address is divided into four 8-bit sections called [octets](https://en.wikipedia.org/wiki/Octet_(computing)).

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

***Note***: Octet is just another word for byte.

Class A, B & C.

**Ch 14 – Using Domain Name System (DNS) and Name Resolution**

$ cat /etc/resolve.conf (hosts)

$ host linuxfoundation.org

$ nslookup linuxfoundation.org

**Ch 14 – Network Configuration Files**

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.

Information about a particular network interface or all network interfaces can be reported by the **ip** and **ifconfig** utilities, which you may have to run as the superuser, or at least, give the full path, i.e. **/sbin/ifconfig**, on some distributions. **ip** is newer than **ifconfig**and has far more capabilities, but its output is uglier to the human eye. Some new Linux distributions do not install the older **net-tools** package to which **ifconfig**belongs, and  so you would have to install it if you want to use it.

**Ip**

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**

One can use the **route** utilityor the newer **ip route** command to view or 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** or **ip route** |
| Add static route | **$ route add -net address** or **ip route add** |
| Delete static route | **$ route del -net address** or **ip route del** |

**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 <address>.**

$ traceroute www.linuxfoundation.org

**Ch 14 – More 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 |
| **mtr** | Combines functionality of ping and traceroute and gives a continuously updated display |
| **dig** | Tests DNS workings. A good replacement for host and nslookup |

For All active connections & routing tables;

$ netstat –r

To scan open ports;

$ sudo nmap

**Ch 14 – Browsers – Graphical & Non-Graphical**

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
* Konqueror
* 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](http://lynx.browser.org/) | Configurable text-based web browser; the earliest such browser and still in use |
| [ELinks](http://lynx.browser.org/) | Based on Lynx. It can display tables and frames |
| [w3m](http://w3m.sourceforge.net/) | Another 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 web page, 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.

**curl**

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.**

You can read a URL using **curl <URL>**. For example, if you want to read [http://www.linuxfoundation.org](https://www.linuxfoundation.org/), type **curl**[**http://www.linuxfoundation.org**](http://www.linuxfoundation.org/).

To get the contents of a web page and store it to a file, type **curl -o saved.html**[**http://www.mysite.com**](http://www.mysite.com/). The contents of the main index file at the website will be saved in **saved.html**.

**ftp**

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 stand-alone client programs.

FTP is one of the oldest methods of network data transfer, dating back to the early 1970s. As such, it is considered inadequate for modern needs, as well as being intrinsically insecure. However, it is still in use and when security is not a concern (such as with so-called anonymous FTP) it can make sense. However, many websites, such as [kernel.org](https://www.kernel.org/), have abandoned its use.

**FTP Clients**

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

FTP has fallen into disfavor on modern systems, as it is intrinsically insecure, since passwords are user credentials that can be transmitted without encryption and are thus prone to interception. Thus, it was removed in favor of using **rsync** and web browser https access for example. As an alternative, 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).

**Ch 14 – Connecting to an FTP Server**

$ ftp –p aristotle.learningmate.com

* + - * username:
      * password:

$ quit

$ ftp –p [ftp.gnu.org](ftp://ftp.gnu.org)

$ anonymous

$ password

$ get welcome.msg

**Ch 14 – SSH – Executing Commands Remotely**

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 login to a remote system using your same user name you can just type **ssh some\_system**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.

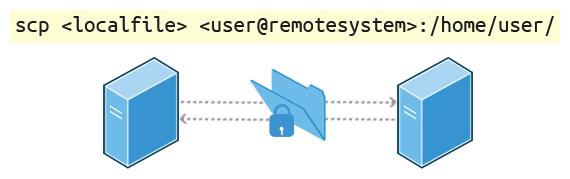
If you want to run as another user, you can do either **ssh -l someone some\_system**or **ssh someone@some\_system**. To run a command on a remote system via SSH, at the command prompt, you can type **ssh some\_system my\_command**.

**Ch 14 – Copying Files Securily with scp**

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.



**Copying Files Securely with scp**

**Ch 14 – Using SSH between Two Virtual Machines**

Ubuntu and CentOS

$ ip –brief addr show ! on both machines.

$ ssh [student@172.16.249.129](mailto:student@172.16.249.129)

$ scp –r /home/student 172.16.249.133:/tmp !Copies to tmp directory (local).

$ exit ! Each machine.

**Lab 14 – Network Troubleshooting**

Troubleshooting network problems is something that you will often encounter if you haven't already. We are going to practice some of the previously discussed tools, that can help you isolate, troubleshoot and fix problems in your network.

The solution file contains a step-by-step procedure for exercising many of the tools we have studied. Please repeat the steps, substituting your actual network interface names, alternative network addresses and web sites, etc.

$ /shin/ifconfig

$ ip addr show

$ sudo systemctl restart NetworkManager

$ sudo systemctl restart network

$ sudo service NetworkManager restart

$ sudo service network restart

$ sudo dhclient eth0

$ sudo ping –c 3 google.com

$ sudo ping 189x128vkjs.com

$ host 189x128vkjs.com

$ host google.com

$ dig google.com

$ sudo traceroute 8.8.8.8

$ ip route show

$ sudo mtr –report –cycles 3 8.8.8.8 ! mtr – enhanced trouceroute utility.

$

**Ch 14 – Summary**

You should now be able to:

* Explain basic networking concepts, including types of networks and addressing issues.
* Configure network interfaces and use basic networking utilities, such as **ifconfig**, **ip**, **ping**, **route** and **traceroute**.
* Use graphical and non-graphical browsers, such as Lynx, w3m, Firefox, Chrome and Epiphany.
* Transfer files to and from clients and servers using both graphical and text mode applications, such as Filezilla, ftp, sftp, curl and wget.

You have completed Chapter 14. 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 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 URLs.
* 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.

**Ch 15 – The Bash Shell and Basic Scripting**

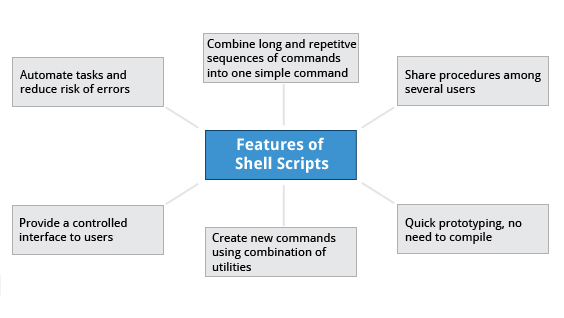
By the end of this chapter, you should be able to:

* Explain the features and capabilities of bash shell scripting.
* Know the basic syntax of scripting statements.
* Be familiar with various methods and constructs used.
* Test for properties and existence of files and other objects.
* Use conditional statements, such as if-then-else blocks.
* Perform arithmetic operations using scripting language.

**Ch 15 – Shell Scripting**

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 will 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.

***Note*:** Many of the topics discussed in this and the next chapter have already been introduced earlier, while discussing things that can be done at the command line. We have elected to repeat some of that discussion in order to make the sections on scripting stand on their own, so the repetition is intentional, not just sloppy editing.



**Features of Shell Scripts**

**Ch 15 –**

**Ch 15 –**

**Ch 15 –**

**Ch 15 –**

**Ch 15 –**

**Ch 15 – Command Shell Choices**

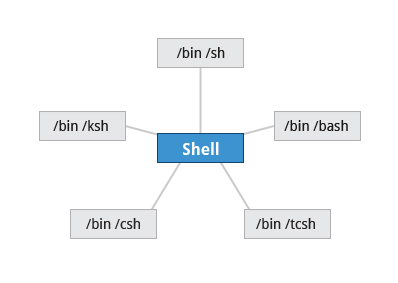
The command interpreter is tasked with executing statements that follow it in the script. Commonly used interpreters include: **/usr/bin/perl**,**/bin/bash**, **/bin/csh**, **/usr/bin/python** and **/bin/sh**.

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 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  
/bin/zsh**

Most Linux users use the default bash shell, but those with long UNIX backgrounds with other shells may want to override the default.



**Command Shell Choices**

**Ch 15 – 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 have noted, you have quite a few choices for the scripting language you can use, such as **/usr/bin/perl**, **/bin/csh, /usr/bin/python,**etc.

**Ch 15 – ASimple bash Script**

Let's write a simple bash script that displays a one line message on the screen. Either type:

**$ cat > hello.sh**  
**#!/bin/bash**  
**echo "Hello Linux Foundation Student"**

and press **ENTER** and **CTRL-D** to save the file, or just create **hello.sh** in your favorite text editor. Then, type **chmod +x hello.sh** to make the file executable by all users.

You can then run the script by  typing **./hello.sh** or by doing:

**$ bash hello.sh  
  Hello Linux Foundation Student**

***Note***: If you use the second form, you do not have to make the file executable.

**Ch 15 – 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, **name**. We can reference the value of a shell variable by using a **$** in front of the variable name, such as **$name**. To create this script, you need to create a file named **getname.sh** in your favorite editor with the following content:

**#!/bin/bash**  
**# Interactive reading of a variable**  
**echo "ENTER YOUR NAME"**  
**read name**  
**# Display variable input**  
**echo The name given was :$name**

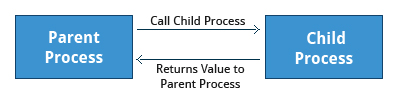
Once again, make it executable by doing **chmod +x getname.sh**.

In the above example, when the user types**./getname.sh**and the script is executed, the user is prompted with the string  **ENTER YOUR NAME**. The user then needs to enter a value and press the **Enter** key. The value will then be printed out.

***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). However, note the special magic combination of **#!**, used on the first line, is a unique exception to this rule.

**Ch 15 – Return Values**

All shell scripts generate a return value upon finishing execution, which can be explicitly set with the **exit** statement. Return values permit a process to monitor the exit state of another process, often in a parent-child relationship. Knowing how the process terminates enables taking any appropriate steps which are necessary or contingent on success or failure.



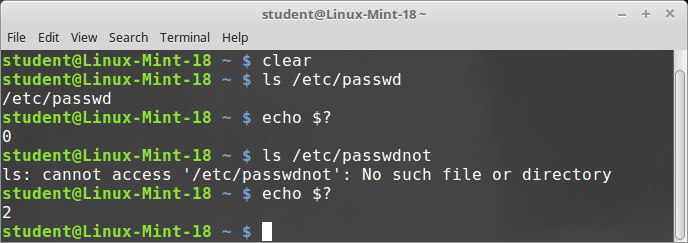
**Return Values**

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 as well as one that does not, the return value is stored in the environment variable represented by **$?**:

**$ ls /etc/logrotate.conf  
/etc/logrotate.conf**

**$ echo $?  
0**

In this example, the system is able to locate the file **/etc/logrotate.conf** and lsreturns a value of **0** to indicate success. When run on an non-existing file, it returns **2**. Applications often translate these return values into meaningful messages easily understood by the user.



**Viewing Return Values**

**Lab 15 – Exit Status Codes**

Create a file named **testls.sh**, with the content below.

**#!/bin/bash**

**#**

**# check for non-existent file, exit status will be 2**

**#**

**ls SoMeFiLe.ext**

**echo "status: $?"**

**# create file, and do again, exit status will be 0**

**touch SoMeFiLe.ext**

**ls SoMeFiLe.ext**

**echo "status: $?"**

**# remove the file to clean up**

**rm SoMeFiLe.ext**

Make it executable and run it:

**student:/tmp> chmod +x testls.sh**

**student:/tmp> ./testls.sh**

**ls: cannot access SoMeFiLe.ext: No such file or directory**

**status: 2**

**SoMeFiLe.ext**

**status: 0**

**Ch 15 – 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 to be executed next |
| **$** | Indicates what follows is an environment variable |
| **>** | Redirect output |
| **>>** | Append output |
| **<** | Redirect input |
| **|** | Used to pipe the result into the next command |

There are other special characters and character combinations and constructs that scripts understand, such as **(..)**, **{..}**, **[..]**, **&&**, **||**, **'**, **"**,**$((...))**, some of which we will discuss later.

**Ch 15 – Splitting Long Commands Over Multiple Lines**

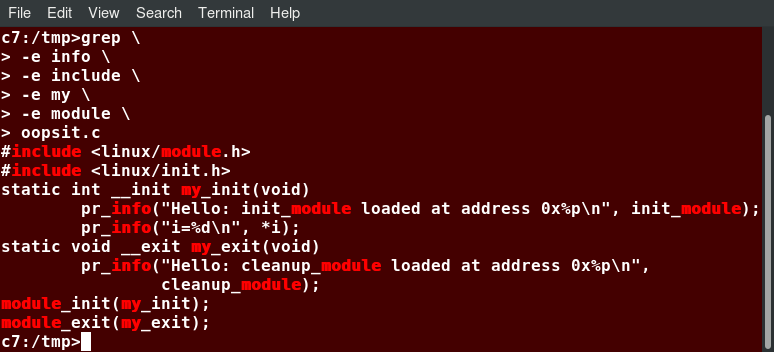
Sometimes, commands are too long to either easily type on one line, or to grasp and understand (even though there is no real practical limit to the length of a command line).

In this case, the concatenation operator (**\**), the backslash character, is used to continue long commands over several lines.

Here is an example of a command installing a long list of packages on a system using Debian package management:

**$~/> cd $HOME  
$~/> sudo apt-get install autoconf automake bison build-essential   
    chrpath curl diffstat emacs flex gcc-multilib g++-multilib \   
    libsdl1.2-dev libtool lzop make mc patch \   
    screen socat sudo tar texinfo tofrodos u-boot-tools unzip \   
    vim wget xterm zip**

The command is divided into multiple lines to make it look readable and easier to understand. The **\** operator at the end of each line causes the shell to combine (concatenate) multiple lines and executes them as one single command.



**Splitting Long Commands Over Multiple Lines**

**Ch 15 – Putting Multiple Commands on a Single Line**

Users sometimes need to combine several commands and statements and even conditionally execute them based on the behavior of operators used in between them. This method is called chaining of commands.

There are several different ways to do this, depending on what you want to do. The **;** (semicolon) character is used to separate these commands and execute them sequentially, as if they had been typed on separate lines. Each ensuing command is executed whether or not the preceding ones succeed.

Thus, the three commands in the following example will all execute, even if the ones preceding them fail:  
  
**$ make ; make install ; make clean**

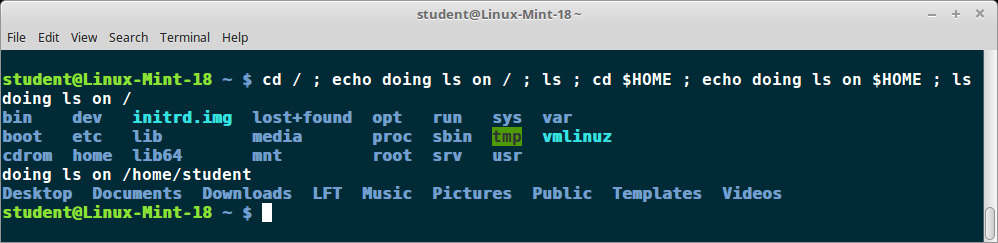
However, you may want to abort subsequent commands when an earlier one fails. You can do this using the **&&** (and) operator as in:

**$ 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.



**Putting Multiple Commands on a Single Line**

**Ch 15 – Output Redirection**

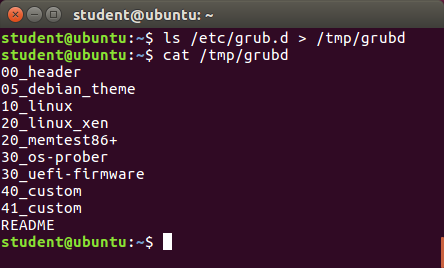
Most operating systems accept input from the keyboard and display the output on the terminal. However, in shell scripting you can send the output to a file. The process of diverting the output to a file is called output redirection. We have already used this facility in our earlier sections on how to use the command line.

The **>** character is used to write output to a file. For example, the following command sends the output of freeto **/tmp/free.out**:

**$ free > /tmp/free.out**

To check the contents of **/tmp/free.out**, 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.



**Output Redirection**

**Ch 15 – 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.

The following three commands (using the **wc** program to count the number of lines, words and characters in a file) are entirely equivalent and involve input redirection, and a command operating on the contents of a file:

**$ wc < /etc/passwd  
49  105 2678 /etc/passwd**

**$ wc /etc/passwd  
49  105 2678 /etcpasswd**

**$ cat /etc/passwd | wc  
49  105 2678**

**Ch 15 – Built-In Shell Commands**

Shell scripts execute sequences of commands and other types of statements. These commands can be:

* + - Compiled applications
    - Built-in bash commands
    - Shell scripts or scripts from other interpreted languages, such as perl and Python.

Compiled applications are binary executable files, generally residing on the filesystem in well-known directories such as **/usr/bin**. Shell scripts always have access to applications such as **rm**, **ls**, **df**, **vi**, and **gzip**, which are programs compiled from lower level programming languages such as C.

In addition, 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**. Thus, slightly different behavior can be expected from the built-in version of a command such as **echo** as compared to **/bin/echo**.

A complete list of bash built-in commands can be found in the bash man page, or by simply typing **help**, as we review on the next page.



**Built-In Shell Commands**

**Ch 15 – 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 or special character. 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 |

If you type in the script shown in the figure, make the script executable with **chmod +x param.sh**. Then, run the script giving it several arguments, as shown. The script is processed as follows:  
  
**$0** prints the script name: **param.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 four five**  
The final statement becomes: **All done with param.sh**

**Ch 15 – 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 in **$( )**
* By enclosing the inner command with backticks **(`)**

The second, backticks form, is deprecated in new scripts and commands. No matter which method is used, the specified command will be executed in a newly launched shell environment, and the standard output of the shell will be inserted where the command substitution is done.

Virtually any command can be executed this way. While both of these methods enable command substitution, the **$( )**method allows command nesting. New scripts should always use of this more modern method. For example:

**$ ls /lib/modules/$(uname -r)/**

In the above example, the output of the command **uname –r** (which will be something like **4.18.2**) is inserted into the argument for the **ls**command.

**Ch 15 – Environment Variables**

Most 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 (already covered in some detail in *Chapter 12: User Environment*) for supplying inputs, validation, and controlling behavior.

As we discussed earlier,  some examples of standard environment variables are **HOME**, **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.

**Ch 15 – Exporting Environment Variables**

While we discussed the exportof environment variables in the section on the "*User Environment*", it is worth reviewing this topic in the context of writing bash scripts.

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, they are only copied and inherited.

Typing export with no arguments will give a list of all currently exported environment variables.

**Ch 15 – 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.

**Lab 15 – Working with Files and Directories in a Script**

Create a file named **testfile.sh**, with the content below.

**#!/bin/bash**

**# Prompts the user for a directory name and then creates it with mkdir.**

**echo "Give a directory name to create:"**

**read NEW\_DIR**

**# Save original directory so we can return to it (could also just use pushd, popd)**

**ORIG\_DIR=$(pwd)**

**# check to make sure it doesn't already exist!**

**[[ -d $NEW\_DIR ]] && echo $NEW\_DIR already exists, aborting && exit**

**mkdir $NEW\_DIR**

**# Changes to the new directory and prints out where it is using pwd.**

**cd $NEW\_DIR**

**pwd**

**# Using touch, creates several empty files and runs ls on them to verify they are empty.**

**for n in 1 2 3 4**

**do**

**touch file$n**

**done**

**ls file?**

**# (Could have just done touch file1 file2 file3 file4, just want to show do loop!)**

**# Puts some content in them using echo and redirection.**

**for names in file?**

**do**

**echo This file is named $names > $names**

**done**

**# Displays their content using cat**

**cat file?**

**# Says goodbye to the user and cleans up after itself**

**cd $ORIG\_DIR**

**rm -rf $NEW\_DIR**

**echo "Goodbye My Friend!"**

Make it executable and run it:

**$ chmod +x testfile.sh**

**./testfile.sh**

**Give a directory name to create:**

**/tmp/SOME\_DIR**

**/tmp/SOME\_DIR**

**file1 file2 file3 file4**

**This file is named file1**

**This file is named file2**

**This file is named file3**

**This file is named file4**

**Goodbye My Friend**

**Lab 15.2 – Passing Arguments**

Create a file named **testarg.sh**, with the content below.

**#!/bin/bash**

**#**

**# check for an argument, print a usage message if not supplied.**

**#**

**if [ $# -eq 0 ] ; then**

**echo "Usage: $0 argument"**

**exit 1**

**fi**

**echo $1**

**exit 0**

Make it executable and run it:

**student:/tmp> chmod +x testarg.sh**

**student:/tmp> ./testarg.sh Hello**

**Hello**

**student:/tmp>./testarg.sh**

**Usage: ./testarg.sh argument**

**student:/tmp>**

**Lab 15 – Environment Variables**

Create a file named **testenv.sh**, with the content below.

**#!/bin/bash**

**echo "Enter 1 or 2, to set the environmental variable EVAR to Yes or No"**

**read ans**

**# Set up a return code**

**RC=0**

**if [ $ans -eq 1 ]**

**then**

**export EVAR="Yes"**

**else**

**if [ $ans -eq 2 ]**

**then**

**export EVAR="No"**

**else**

**# can only reach here with a bad answer**

**export EVAR="Unknown"**

**RC=1**

**fi**

**fi**

**echo "The value of EVAR is: $EVAR"**

**exit $RC**

Make it executable and run it:

**student:/tmp> chmod +x testenv.sh**

**student:/tmp> ./testenv.sh**

**Enter 1 or 2, to set the environmental variable EVAR to Yes or No**

**1**

**The value of EVAR is: Yes**

**student:/tmp> ./testenv.sh**

**Enter 1 or 2, to set the environmental variable EVAR to Yes or No**

**2**

**The value of EVAR is: No**

**student:/tmp> ./testenv.sh**

**Enter 1 or 2, to set the environmental variable EVAR to Yes or No**

**3**

The value of EVAR is: Unknown

**Lab 15 – Functions**

Create a file named **testfun.sh**, with the content below.

**#!/bin/bash**

**# Functions (must be defined before use)**

**func1() {**

**echo " This message is from function 1"**

**}**

**func2() {**

**echo " This message is from function 2"**

**}**

**func3() {**

**echo " This message is from function 3"**

**}**

**# Beginning of the main script**

**# prompt the user to get their choice**

**echo "Enter a number from 1 to 3"**

**read n**

**# Call the chosen function**

**func$n**

Make it executable and run it:

**student:/tmp> chmod +x testfun.sh**

**student:/tmp> ./testfun.sh**

**Enter a number from 1 to 3**

**2**

**This message is from function 2**

**$ ./testfun.sh**

**Enter a number from 1 to 3**

**7**

**./testfun.sh: line 21: func7: command not found**

**Ch 15 – 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**

**Ch 15 – Using the if Statement**

In the following example, an **if** statement checks to see if a certain file exists, and if the file is found, it displays a message indicating success or failure:

**if [ -f "$1" ]**  
**then**  
**echo file "$1 exists"**  
**else**  
**echo file "$1" does not exist**  
**fi**

We really should also check first that there is an argument passed to the script (**$1**) and abort if not.

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 will not talk about this here.

**Ch 15 – The elif Statement**

You can use the **elif** statement to perform more complicated tests, and take action appropriate actions. The basic syntax is:

**if [ sometest ] ; then**  
**echo Passed test1**  
**elif [ somothertest ] ; then**  
**echo Passed test2**  
**fi**

In the example shown we use strings tests which we will explain shortly, and show how to pull in an environment variable with the **read**statement.

$ cat ./show\_elif.sh

$ ./show\_elif.sh

$

**Ch 15 – Testing for Files**

bash provides a set of file conditionals, that can be used with the **if** statement, including those in the table.

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 [ -x /etc/passwd ] ; then**  
**ACTION**  
**fi**

the **if** statement checks if the file **/etc/passwd** is executable, which it is not. Note the very common practice of putting:

**; then**

on the same line as the **if** statement.

You can view the full list of file conditions typing:

**man 1 test**.

|  |  |
| --- | --- |
| **Condition** | **Meaning** |
| **-e file** | Checks if the file exists. |
| **-d file** | Checks if the file is a directory. |
| **-f file** | Checks if the file is a regular file (i.e. not a symbolic link, device node, directory, etc.) |
| **-s file** | Checks if the file is of non-zero size. |
| **-g file** | Checks if the file has **sgid** set. |
| **-u file** | Checks if the file has **suid** set. |
| **-r file** | Checks if the file is readable. |
| **-w file** | Checks if the file is writable. |
| **-x file** | Checks if the file is executable. |

**Ch 15 – Boolean Expressions**

Boolean expressions evaluate to either TRUE or FALSE, and results are obtained using the various Boolean operators listed in the table.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Operation** | **Meaning** |
| **&&** | **AND** | The action will be performed only if both the conditions evaluate to true. |
| **||** | **OR** | The action will be performed if any one of the conditions evaluate to true. |
| **!** | **NOT** | The action will be performed only if the condition evaluates to false. |

Note that if you have multiple conditions strung together with the **&&** operator, processing stops as soon as a condition evaluates to false. For example, if you have **A && B && C** and A is true but B is false, C will never be executed.

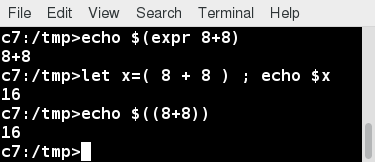
Likewise, if you are using the **||** operator, processing stops as soon as anything is true. For example, if you have **A || B || C** and A is false and B is true, you will also never execute C.

**Ch 15 – 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=$((...))**.



**Arithmetic Expressions**

**Lab 15 – Arithmetic Expressions in a Bash Shell (.sh)**

Create a file named **testmath.sh**, with the content below.

**#!/bin/bash**

**# Functions. must be before the main part of the script**

**# in each case method 1 uses $((..))**

**# method 2 uses let**

**# method 3 uses expr**

**add() {**

**answer1=$(($1 + $2))**

**let answer2=($1 + $2)**

**answer3=`expr $1 + $2`**

**}**

**sub() {**

**answer1=$(($1 - $2))**

**let answer2=($1 - $2)**

**answer3=`expr $1 - $2`**

**}**

**mult() {**

**answer1=$(($1 \* $2))**

**let answer2=($1 \* $2)**

**answer3=`expr $1 \\* $2`**

**}**

**div() {**

**answer1=$(($1 / $2))**

**let answer2=($1 / $2)**

**answer3=`expr $1 / $2`**

**}**

**# End of functions**

**#**

**# Main part of the script**

**# need 3 arguments, and parse to make sure they are valid types**

**op=$1 ; arg1=$2 ; arg2=$3**

**[[ $# -lt 3 ]] && \**

**echo "Usage: Provide an operation (a,s,m,d) and two numbers" && exit 1**

**[[ $op != a ]] && [[ $op != s ]] && [[ $op != d ]] && [[ $op != m ]] && \**

**echo operator must be a, s, m, or d, not $op as supplied**

**# ok, do the work!**

**if [[ $op == a ]] ; then add $arg1 $arg2**

**elif [[ $op == s ]] ; then sub $arg1 $arg2**

**elif [[ $op == m ]] ; then mult $arg1 $arg2**

**elif [[ $op == d ]] ; then div $arg1 $arg2**

**else**

**echo $op is not a, s, m, or d, aborting ; exit 2**

**fi**

**# Show the answers**

**echo $arg1 $op $arg2 :**

**echo 'Method 1, $((..)),' Answer is $answer1**

**echo 'Method 2, let, ' Answer is $answer2**

**echo 'Method 3, expr, ' Answer is $answer3**

Make it executable and run it:

**student:/tmp> chmod +x testmath.sh**

**student:/tmp> ./testmath.sh**

student:/tmp> for n in a s m d x ; do ./testmath.sh $n 21 7 ; done

**21 a 7 :**

**Method 1, $((..)), Answer is 28**

**Method 2, let,**

**Answer is 28**

**Method 3, expr,**

**Answer is 28**

**21 s 7 :**

**Method 1, $((..)), Answer is 14**

**Method 2, let,**

**Answer is 14**

**Method 3, expr,**

**Answer is 14**

**21 m 7 :**

**Method 1, $((..)), Answer is 147**

**Method 2, let,**

**Answer is 147**

**Method 3, expr,**

**Answer is 147**

**21 d 7 :**

**Method 1, $((..)), Answer is 3**

**Method 2, let,**

**Answer is 3**

**Method 3, expr,**

**Answer is 3**

**operator must be a, s, m, or d, not x as supplied**

**x is not a, s, m, or d, aborting**

**Ch 15 – Review**

You should now be able to:

* Explain the features and capabilities of bash shell scripting.
* Know the basic syntax of scripting statements.
* Be familiar with various methods and constructs used.
* Test for properties and existence of files and other objects.
* Use conditional statements, such as if-then-else blocks.
* Perform arithmetic operations using scripting language.

**Ch 15 – Summary**

You have completed Chapter 15. 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 preassigned 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 **\***.

**Ch 16 – More on Bash Shell Scripting**

manipulating strings, evaluating Boolean expressions, and everybody’s favorite, debugging the scripts you just wrote.

**Ch 16 – Objectives**

By the end of this chapter, you should be able to:

* Manipulate strings to perform actions such as comparison and sorting.
* Use Boolean expressions when working with multiple data types, including strings or numbers, as well as files.
* Use **case** statements to handle command line options.
* Use looping constructs to execute one or more lines of code repetitively.
* Debug scripts using set **-x** and set **+x**.
* Create temporary files and directories.
* Create and use random numbers.

**Ch 16 – String Manipulation**

Let’s go deeper and find out how to work with strings in scripts.

A string variable contains a sequence of text characters. It can include letters, numbers, symbols and punctuation marks. Some examples include: **abcde**, **123**, **abcde 123**, **abcde-123**, **&acbde=%123**.

String operators include those that do comparison, sorting, and finding the length. The following table demonstrates the use of some basic string operators:

|  |  |
| --- | --- |
| **Operator** | **Meaning** |
| **[[ string1 > string2 ]]** | Compares the sorting order of **string1** and **string2**. |
| **[[ string1 == string2 ]]** | Compares the characters in **string1** with the characters in **string2**. |
| **myLen1=${#string1}** | Saves the length of **string1** in the variable **myLen1**. |

$ #!/bn/bash

echo Give two strings to compare;

echo “”

read str1 str2

echo “”

if [ “$str1” = “$str2” ]; then

echo “The first string: $str1, is the same as the second string: $str2”

else

echo “The first string: $str1, is not the same as the second string: $str2”

fi

$ ./str\_demo1.sh

$ #!/bn/bash

echo Give a filename to check on

read file

echo “”

if [ -f “$file” ] ; then

echo ‘$file exists”

else

echo “$file does NOT exist”

fi

At times, you may not need to compare or use an entire string. To extract the first **n**characters of a string we can specify: **${string:0:n}**. Here, **0** is the offset in the string (i.e. which character to begin from) where the extraction needs to start and **n** is the number of characters to be extracted.

To extract all characters in a string after a dot (**.**), use the following expression: **${string#\*.}**.

$ NAME=Eddie.Haskel

$ first=${NAME:0:5} ; echo first name = $first

$ last=${NAME#\*.} ; echo last name = $last

**Ch 16 – The case Statement**

The **case** statement is used in scenarios where the actual value of a variable can lead to different execution paths. **case** statements are often used to handle command-line options.

Below are some of the advantages of using the **case** statement:

* It is easier to read and write.
* It is a good alternative to nested, multi-level **if-then-else-fi** code blocks.
* It enables you to compare a variable against several values at once.
* It reduces the complexity of a program.
* Here is the basic structure of the **case** statement:
* **case expression in  
     pattern1) execute commands;;  
     pattern2) execute commands;;  
     pattern3) execute commands;;  
     pattern4) execute commands;;  
     \* )       execute some default commands or nothing ;;  
  esac**

**Lab 16 –**

Create a file named **testcase.sh**, with the content below.

**#!/bin/bash**

**# Accept a number between 1 and 12 as**

**# an argument to this script, then return the**

**# the name of the month that corresponds to that number.**

**# Check to see if the user passed a parameter.**

**if [ $# -eq 0 ]**

**then**

**echo "Error. Give as an argument a number between 1 and 12."**

**exit 1**

**fi**

**# set month equal to argument passed for use in the script**

**month=$1**

**################################################**

**# The example of a case statement:**

**case $month in**

**1) echo "January" ;;**

**2) echo "February" ;;**

**3) echo "March" ;;**

**4) echo "April" ;;**

**5) echo "May" ;;**

**6) echo "June" ;;**

**7) echo "July" ;;**

**8) echo "August" ;;**

**9) echo "September" ;;**

**10) echo "October" ;;**

**11) echo "November" ;;**

**12) echo "December" ;;**

**\*)**

**echo "Error. No month matches: $month"**

**echo "Please pass a number between 1 and 12."**

**exit 2**

**;;**

**esac**

**exit 0**

Make it executable and run it:

**student:/tmp> chmod +x testcase.sh**

**student:/tmp> ./testcase.sh 5**

**May**

**student:/tmp> ./testcase.sh 12**

**December**

**student:/tmp> ./testcase.sh 99**

**Error. No month matches: 99**

**Please pass a number between 1 and 12**

**student:/tmp>**

**Ch 16 – Looping Contructs**

By using looping constructs, you can execute one or more lines of code repetitively, usually on a selection of values of data such as individual files. Usually, you do this until a conditional test returns either true or false, as is required.

Three type of loops are often used in most programming languages:

* **for**
* **while**
* **until**.

All these loops are easily used for repeating a set of statements until the exit condition is true.

**Ch 16 – The for Loop**

The **for**loop operates on each element of a list of items. The syntax for the **for** loop is:

**for *variable-name* in *list*  
do  
    execute one iteration for each item in the *list* until the *list* is finished**  
**done**

In this case, **variable-name** and **list** are substituted by you as appropriate (see examples). As with other looping constructs, the statements that are repeated should be enclosed by **do** and **done**.

The screenshot here shows an example of the **for** loop to print the sum of numbers 1 to 10.

**Ch 16 – The while Loop**

The **while** loop repeats a set of statements as long as the control command returns true. The syntax is:  
  
**while condition is true  
do  
    Commands for execution  
    ----  
done**

The set of commands that need to be repeated should be enclosed between **do** and **done**. You can use any command or operator as the condition. Often, it is enclosed within square brackets (**[]**).

**Ch 16 – The until Loop**

The **until** loop repeats a set of statements as long as the control command is false. Thus, it is essentially the opposite of the **while** loop. The syntax is:

**until condition is false  
do  
    Commands for execution  
    ----  
done**

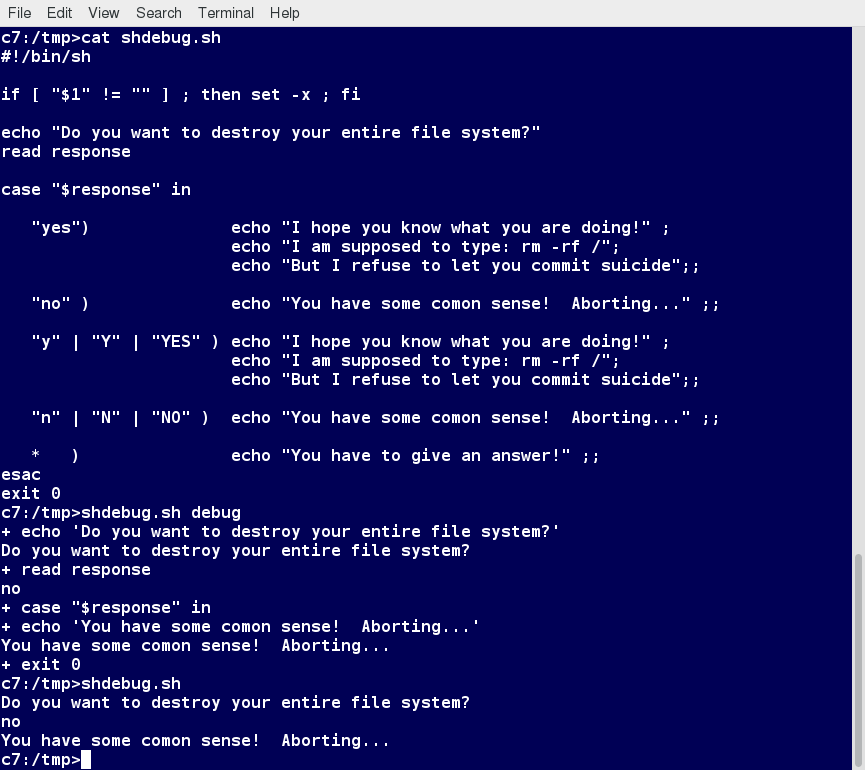
**Ch 16 – Debugging bash Scripts**

Before fixing an error (or bug), it is vital to know its source.

You can run a bash script in debug mode either by doing **bash –x ./script\_file**,  or bracketing parts of the script with **set -x**and **set +x**. The debug mode helps identify the error because:

* + - It traces and prefixes each command with the **+** character.
    - It displays each command before executing it.
    - It can debug only selected parts of a script (if desired) with:  
        
      **set -x    # turns on debugging**  
      **...**  
      **set +x    # turns off debugging**

The screenshot shown here demonstrates a script which runs in debug mode if run with any argument on the command line.



**Script Debug Mode**

In UNIX/Linux, all programs that run are given three open file streams when they are started as listed in the table:

|  |  |  |
| --- | --- | --- |
| **File stream** | **Description** | **File Descriptor** |
| **stdin** | Standard Input, by default the keyboard/terminal for programs run from the command line | 0 |
| **stdout** | Standard output, by default the screen for programs run from the command line | 1 |
| **stderr** | Standard error, where output error messages are shown or saved | 2 |

Using redirection, we can save the stdout and stderr output streams to one file or two separate files for later analysis after a program or command is executed.

**Ch 16 – Creating Temporary Files and Directories**

Consider a situation where you want to retrieve 100 records from a file with 10,000 records. You will need a place to store the extracted information, perhaps in a temporary file, while you do further processing on it.

Temporary files (and directories) are meant to store data for a short time. Usually, one arranges it so that these files disappear when the program using them terminates. While you can also use touchto create a temporary file, in some circumstances this may make it easy for hackers to gain access to your data. This is particularly true if the name and the file location of the temporary file are predictable.

The best practice is to create random and unpredictable filenames for temporary storage. One way to do this is with the mktemputility, as in the following examples.

The **XXXXXXXX** is replaced by the mktemputility with random characters to ensure the name of the temporary file cannot be easily predicted and is only known within your program.

|  |  |
| --- | --- |
| **Command** | **Usage** |
| **TEMP=$(mktemp /tmp/tempfile.XXXXXXXX)** | To create a temporary file |
| **TEMPDIR=$(mktemp -d /tmp/tempdir.XXXXXXXX)** | To create a temporary directory |

Sloppiness in creation of temporary files can lead to real damage, either by accident or if there is a malicious actor. For example, if someone were to create a symbolic link from a known temporary file used by root to the **/etc/passwd** file, like this:

**$ ln -s /etc/passwd /tmp/tempfile**  
There could be a big problem if a script run by root has a line in like this:

**echo $VAR > /tmp/tempfile**

The password file will be overwritten by the temporary file contents.

To prevent such a situation, make sure you randomize your temporary file names by replacing the above line with the following lines:

**TEMP=$(mktemp /tmp/tempfile.XXXXXXXX)**  
**echo $VAR > $TEMP**

**Ch 16 – Discarding Output with /dev/null**

Certain commands (like **find**) will produce voluminous amounts of output, which can overwhelm the console. To avoid this, we can redirect the large output to a special file (a device node) called **/dev/null**. This pseudofile is also called the bit bucket or black hole.

All data written to it is discarded and write operations never return a failure condition. Using the proper redirection operators, it can make the output disappear from commands that would normally generate output to stdout and/or stderr:

**$ ls -lR /tmp > /dev/null**

In the above command, the entire standard output stream is ignored, but any errors will still appear on the console. However, if one does:

**$ ls -lR /tmp >& /dev/null**

both stdout and stderrwill be dumped into **/dev/null**.

**Ch 16 – Random Numbers and Data**

**$RANDOM**

It is often useful to generate random numbers and other random data when performing tasks such as:

* Performing security-related tasks
* Reinitializing storage devices
* Erasing and/or obscuring existing data
* Generating meaningless data to be used for tests.

Such random numbers can be generated by using the **$RANDOM** environment variable, which is derived from the Linux kernel’s built-in random number generator, or by the OpenSSL library function, which uses the FIPS140(Federal Information Processing Standard) algorithm to generate random numbers for encryption

Create a file named **testrandom.sh**, with the content below.

**#!/bin/bash**

**##**

**# check to see if the user supplied in the parameter.**

**[[ $# -eq 0 ]] && echo "Usage: $0 word" && exit 1**

**echo "$1-$RANDOM"**

**exit 0**

Make it executable and run it:

**student:/tmp> chmod +x testrandom.sh**

**student:/tmp> ./testrandom.sh strA**

**strA-29294**

**student:/tmp>./testrandom.sh strB**

**strB-23911**

**student:/tmp>./testrandom.sh strC**

**strC-27782**

**student:/tmp>**

/dev/urandom & /dev/random

**Ch 16 – Review**

You should now be able to:

* Manipulate strings to perform actions such as comparison and sorting.
* Use Boolean expressions when working with multiple data types, including strings or numbers, as well as files.
* Use **case** statements to handle command line options.
* Use looping constructs to execute one or more lines of code repetitively.
* Debug scripts using set **-x** and set **+x**.
* Create temporary files and directories.
* Create and use random numbers.

You have completed Chapter 16. Let’s summarize the key concepts covered:

* You can manipulate strings to perform actions such as comparison, sorting, and finding length.
* You can use Boolean expressions  when working with multiple data types, including strings or numbers, as well as files.
* The output of a Boolean expression is either true or false.
* Operators used in Boolean expressions include the **&&** (AND), **||**(OR), and **!**(NOT) operators.
* We looked at the advantages of using the **case** statement in scenarios where the value of a variable can lead to different execution paths.
* Script debugging methods help troubleshoot and resolve errors.
* The standard and error outputs from a script or shell commands can easily be redirected into the same file or separate files to aid in debugging and saving results
* Linux allows you to create temporary files and directories, which store data for a short duration, both saving space and increasing security.
* Linux provides several different ways of generating random numbers, which are widely used.

**Ch 17 – Printing**

By the end of this chapter, you should know how to:

* Configure a printer on a Linux machine.
* Print documents.
* Manipulate postscript and PDF files using command line utilities.

**Ch 17 – CUPS Overview**

CUPS is the underlying software almost all Linux systems use to print from applications like a web browser or LibreOffice. It converts page descriptions produced by your application (put a paragraph here, draw a line there, and so forth) and then sends the information to the printer. It acts as a print server for both local and 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.

CUPS carries out the printing process with the help of its various components:

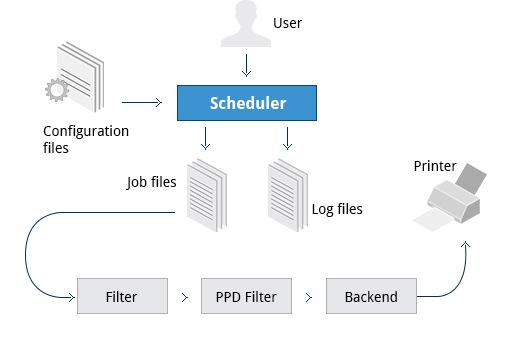
* Configuration Files
* Scheduler
* Job Files
* Log Files
* Filter
* Printer Drivers
* Backend.

You will learn about each of these components on the next few pages.

**Ch 17 – 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.

We will look at the browser-based interface that can be used with CUPS,  which allows you to view and manipulate the order and status of pending print jobs.



**Scheduler**

**Ch 17 – 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.

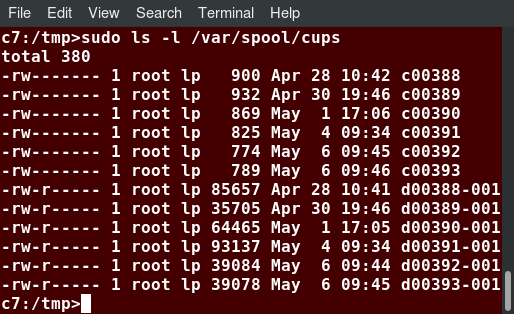
**cupsd.conf** is where most system-wide settings are located; it does not contain any printer-specific details. Most of the settings available in this file relate to network security, i.e. which systems can access CUPS network capabilities, how printers are advertised on the local network, what management 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/**.

**Ch 17 – 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".



**/var/spool/cups Directory**

 After a printer successfully handles a job, data files are automatically removed. These data files belong to what is commonly known astheprint queue.

**Ch 17 – 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**

**Ch 17 – 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 the 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.

**Ch 17 – Managing CUPS Daemon**

Assuming CUPS has been installed 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 can be done with the systemctlutility:

**$ systemctl status cups**

**$ sudo systemctl [enable|disable] cups**

**$ sudo systemctl [statis|start|stop|restart|disable|enable] cups**

**Ch 17 – 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:

* + - 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](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.

**Ch 17 – Printing from the Graphical Interface**

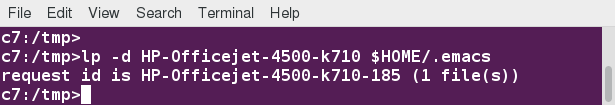
CTRL-P

**Ch 17 – 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).

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**.



**Printing from the Command-Line Interface**

**lp**and **lpr** acceptcommand 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.

To print to Joe-the-Printer

$ lp –d Joe-the-Printer <filename>

$ lp –n 2 <filename> ! 2 copies

$ lpoptions –d Joe-the-Printer ! Sets Joe-the-Printer as default.

$ lpq –a ! Printer Queu status.

**Ch 17 – 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 canceling 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 |

**Ch 17 – Working with PostScript and PDF**

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.

Postscript has been for the most part superseded by the PDF format (Portable Document Format) which produces far smaller files in a compressed format for which support has been integrated into many applications. However, one still has to deal with postscript documents, often as an intermediate format on the way to producing final documents.

**Ch 17 – 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 columns (**-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 |

**Ch 17 – Converting between PostScript and PDF**

Most users today are far more accustomed to working with files in PDF format, viewing them easily either on the Internet through their browser or locally on their machine. The PostScript format is still important for various technical reasons that the general user will rarely have to deal with.

From time to time, you may need to convert files from one format to the other, and there are very simple utilities for accomplishing that task. **ps2pdf** and pdf2ps are part of the **ghostscript**package installed on or available on all Linux distributions. As an alternative, there are **pstopdf** and **pdftops** which are usually part of the **poppler** package, which may need to be added through your package manager. Unless you are doing a lot of conversions or need some of the fancier options (which you can read about in the man pages for these utilities), it really does not matter which ones you use.

Another possibility is to use the very powerful convert program, which is part of the **ImageMagick** package.

Some usage examples:

|  |  |
| --- | --- |
| **Command** | **Usage** |
| **pdf2ps file.pdf** | Converts **file.pdf** to**file.ps** |
| **ps2pdf file.ps** | Converts **file.ps** to**file.pdf** |
| **pstopdf input.ps output.pdf** | Converts **input.ps**to **output.pdf** |
| **pdftops input.pdf output.ps** | Converts **input.pdf** to **output.ps** |
| **convert input.ps output.pdf** | Converts **input.ps** to **output.pdf** |
| **convert input.pdf output.ps** | Converts **input.pdf** to **output.ps** |

**Ch 17 – Viewing PDF Content**

Linux has many standard programs that can read PDF files, as well as many applications that can easily create them, including all available office suites, such as LibreOffice.

The most common Linux PDF readers are:

* + 1. Evince is available on virtually all distributions and the most widely used program.
    2. Okular is based on the older kpdf and available on any distribution that provides the KDE environment.
    3. GhostView is one of the first open source PDF readers and is universally available.
    4. Xpdf is one of the oldest open source PDF readers and still has a good user base.

All of these open source PDF readers support and can read files following the PostScript standard unlike the proprietary Adobe Acrobat Reader, which was once widely used on Linux systems, but, with the growth of these excellent programs, very few Linux users use it today.

**Ch 17 – Manipulating PDFs**

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. 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 order to accomplish these tasks there are several programs available:

* **qpdf**
* **pdftk**
* **ghostscript**

**qpdf**  is widely available on **Linux** distributions and is very full-featured.  **pdftk** was once very popular but depends on an obsolete unmaintained package (**libgcj**) and a number of distributions have dropped it; thus we recommend avoiding it.  **Ghostscript** (often invoked using **gs**) is widely available and well-maintained.  However, its usage is a little complex.

**Ch 17 – Encrypting PDF Files with pdftk**

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.

**Ch 17 – Using Ghostscript**

Ghostscript is widely available as an interpreter for the Postscript and PDF languages. The executable program associated with it is abbreviated to gs.

This utility can do most of the operations pdftk can, as well as many others; see man gs for details. Use is somewhat complicated by the rather long nature of the options. For example:

* + - Combine three PDF files into one:  
      **$ gs -dBATCH -dNOPAUSE -q -sDEVICE=pdfwrite  -sOutputFile=all.pdf file1.pdf file2.pdf file3.pdf**
    - Split pages 10 to 20 out of a PDF file:  
      **$ gs -sDEVICE=pdfwrite -dNOPAUSE -dBATCH -dDOPDFMARKS=false -dFirstPage=10 -dLastPage=20\  
      -sOutputFile=split.pdf file.pdf**

**Ch 17 – Using pdftk**

To **merge** two documents, type pdftk 1.pdf 2.pdf cat output 12.pdf and press Enter.

$ pdftk 1.pdf 2.pdf cat output 12.pdf ! merges

To **split pages** of one document into a new document, type pdftk A=1.pdf cat A1-2 output new.pdf and press Enter.

$ pdftk A=1.pdf cat a1-2 output new.pdf

To **rotate** a document 90 degrees clockwise, type pdftk A=1.pdf cat A1-endright output new.pdf and press Enter.

$ pdftk A=1.pdf cat A1-endright output new.pdf

To **set password** for a PDF file, type pdftk public.pdf output private.pdf user\_pw PROMPT and press Enter.

$ pdftk public.pdf output private.pdf user\_pw PROMPT

**Ch 17 – Using Additional Tools**

You can use other tools to work with PDF files, such as:

* **pdfinfo**   
  It can extract information about PDF files, especially when the files are very large or when a graphical interface is not available.
* **flpsed**   
  It can add data to a PostScript document. This tool is specifically useful for filling in forms or adding short comments into the document.
* **pdfmod**   
  It is a simple application that provides a graphical interface for modifying PDF documents. Using this tool, you can reorder, rotate, and remove pages; export images from a document; edit the title, subject, and author; add keywords; and combine documents using drag-and-drop action.

For example, to collect the details of a document, you can use the following command:  
  
**$ pdfinfo /usr/share/doc/readme.pdf**

**Lab 17 – Creating PostScript and PDF from Text Files**

1. Check if the enscript package has been installed, try;

**which enscript**

**/usr/bin/enscript**

If you do not get a positive result, install with whichever command is appropriate for your **Linux** distribution:

**apt-get install enscript**

**yum install enscript**

**zypper install enscript**

1. Using enscript, convert /var/dmesq to PostScipt;

**enscript -p /tmp/dmesg.ps /var/log/dmesg**

**evince /tmp/dmesg.ps**

1. Convert the PostScript document to PDF, using ps2pdf

**ps2pdf/tmp/dmesg.ps**

**ls -lh /var/log/dmesg /tmp/dmesg.ps /tmp/dmesg.pdf**

**-rw-rw-r-- 1 coop coop 28K Apr 22 13:00 /tmp/dmesg.pdf**

**-rw-rw-r-- 1 coop coop 80K Apr 22 12:59 /tmp/dmesg.ps**

**-rw-r--r-- 1 root root 53K Apr 22 11:48 /var/log/dmesg**

**evince /tmp/dmesg.ps /tmp/dmesg.pdf**

Note the difference in sizes. PostScript files tend to be large, while PDF is a compressed format.

1. You may want to scan the **man** pages for **enscript** and **ps2pdf** to figure out how to use standard input or standard output instead of files.

**student:/tmp> enscript -p - /var/log/dmesg | ps2pdf - dmesg\_direct.pdf**

**[ 15 pages \* 1 copy ] left in -**

**85 lines were wrapped**

**student:/tmp> ls -l dmesg\*pdf**

**-rw-rw-r-- 1 coop coop 28177 Apr 22 13:20 dmesg\_direct.pdf**

**-rw-rw-r-- 1 coop coop 28177 Apr 22 13:00 dmesg.pdf**

1. Using pdfinfo, determine what is the PDF version used to encode the file, the number of pages, the page size, and other metadata about the file.

**student:/tmp> pdfinfo dmesg.pdf**

**Ch 17 – Overview/Summary**

You should now know how to:

* Configure a printer on a Linux machine.
* Print documents.
* Manipulate postscript and PDF files using command line utilities.

You have completed Chapter 17. Let’s summarize the key concepts covered:

* CUPS provides two command-line interfaces: the System V and BSD.
* The CUPS interface is available at [http://localhost:631](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.

**Ch 18 – Local Security Principles**

By the end of this chapter, you should be able to:

* Have a good grasp of best practices and tools for making Linux systems as secure as possible.
* Understand the powers and dangers of using the root (superuser) account.
* Use the **sudo** command to perform privileged operations while restricting enhanced powers as much as feasible.
* Explain the importance of process isolation and hardware access.
* Work with passwords, including how to set and change them.
* Describe how to secure the boot process and hardware resources.

**Ch 18 – 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 userdelas well asGUI 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 * Normal users have UID's of 1000 or greater |
| **Group ID (GID)** | The primary Group ID (GID); Group Identification Number stored in the **/etc/group** file | Is 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** |

**Ch 18 – 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.

**Ch 18 – 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.

**Ch 18 - Operations Requiring root Privileges**

rootprivileges 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 might be allowed to install software packages, update some settings, use some peripheral devices, 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 filesystem that are outside the normal user’s directories.

**Ch 18 – Operations NOT Requiring 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.

Ex;

Running a network client Sharing a file over the network

Using devices (printer) Printing over the network

Operations on files that the Accessing files that you have access to or share data access over the network

Running SUID-root apps Executing programs such as **passwd**

**Ch 18 – 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 done. | When elevating privilege, you need to enter the user’s password and not 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. |

**Ch 18 – sudo Features**

sudohas 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.d** directory.

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:

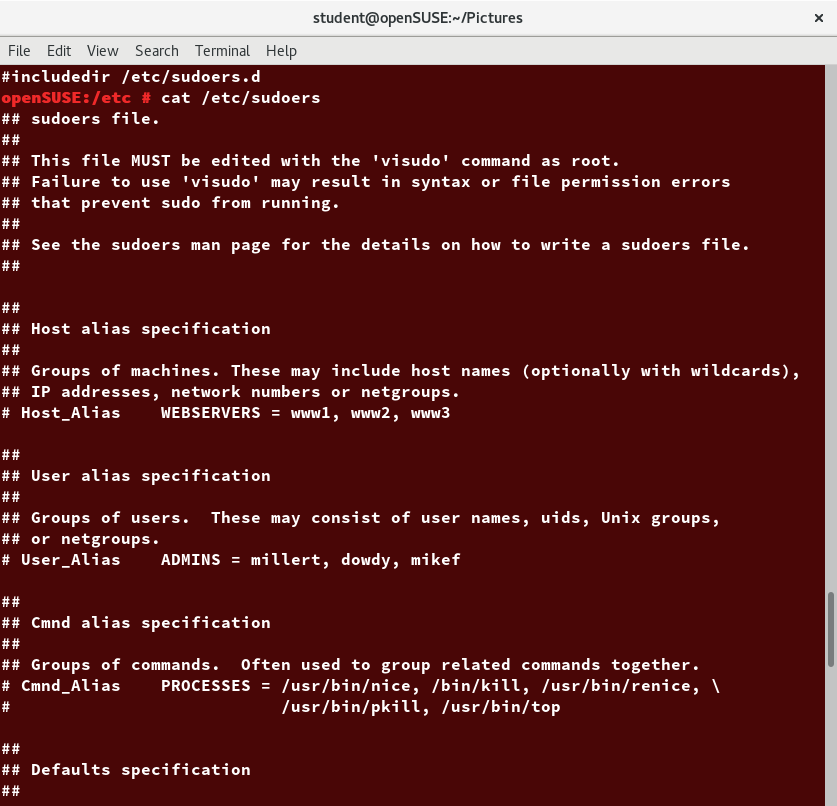
**authentication failure; logname=op uid=0 euid=0 tty=/dev/pts/6 ruser=op rhost= user=op**  
**conversation failed**  
**auth could not identify password for [op]**  
**op : 1 incorrect password attempt ;**  
**TTY=pts/6 ; PWD=/var/log ; USER=root ; COMMAND=/bin/bash**

**Ch 18 – The sudoers File**

Whenever sudois 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 sudoare 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.



**The sudoers File**

**Ch 18 – 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** and/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**

**Ch 18 – 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.

More recent additional security mechanisms that limit risks even further include:

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

**Ch 18 – 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 filesystem layer (which is independent of the actual device or hardware the file resides on). This layer will then open 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 filesystem 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 filesystem, and never through direct access to the device node.

**$ cd /dev**

**$ ls –l/dev/sd\***

**Lab 18 –**

1. Create a new user, using **useradd**, and give the user an initial password with **passwd**.

**sudo useradd newuser**

**sudo passwd newuser**

1. Configure this user to be able to use **sudo**.

With root privileges, (use sudo visudo) add this line to /etc/sudoers

**newuser ALL=(ALL) ALL**

Alternatively, create a file names /etc/soders.d/newuers with just that one line as content

1. Login as or switch to this new user and make sure you can execute a command that requires root privilege.

**sudo su newuser**

Or

**ssh newusuer@localhost**

Which will require giving newuser’s password, and is probably a better solution. Instead of localhost you can give your hostname. IP address or 127.0.0.1. Then as newuser just type

**sudo ls /root**

**Ch 18 – 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.

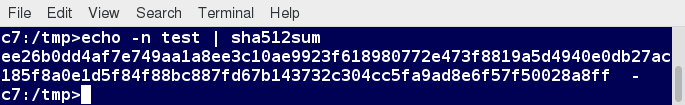
On modern systems, passwords are actually stored in an encrypted format in a secondary file named **/etc/shadow**. Only those with root access can modify/read this file.

**Ch 18 – Password Algorithm**

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 (see graphic):



**Password Encryption: sha512sum**

chage –list student

**Ch 18 – Password Aging**

With the newly created user from the previous exercise, look at the password aging for the user.

Modify the expiration date for the user, setting it to be something that has passed, and check to see what has changed.

When you are finished and wish to delete the newly created account, use **userdel**, as in:

**$ sudo userdel newuser**

**Ans:**

**chage --list newuser**

**sudo chage -E 2014-31-12 newuser**

**chage --list newuser**

**sudo userdel newuser**

**Note:** The example solution works in the US. **chage -E** uses the default date format for the local keyboard setting.

**Ch 18 – Requiring Boot Loader Passwords**

You can secure the boot process with a secure password to prevent someone from bypassing the user authentication step. For systems using the older GRUB boot loader, version 1, you can invoke **grub-md5-crypt** which will prompt you for a password and then encrypt.

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

**password --md5 $1$74r8m1$NmkE69eAjXre.oF1k0cyk/**

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

However, for the GRUBversion 2 (which has taken over almost completely now) things are more complicated. While you have more flexibility, you can take advantage of more advanced features, such as 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**, or the equivalent utility on your Linux distribution.

**Ch 18 – 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.

**Ch 18 – Review (Local Security Principles)**

You should now be able to:

* Have a good grasp of best practices and tools for making Linux systems as secure as possible.
* Understand the powers and dangers of using the root (superuser) account.
* Use the **sudo** command to perform privileged operations while restricting enhanced powers as much as feasible.
* Explain the importance of process isolation and hardware access.
* Work with passwords, including how to set and change them.
* Describe how to secure the boot process and hardware resources.

You have completed Chapter 18. Let’s summarize the key concepts covered:

* The root account has authority over the entire system.
* root privileges may be required for tasks, such as restarting services, manually installing packages and managing parts of the filesystem that are outside your home directory.
* In order to perform any privileged operations such as system-wide changes, you need to use either **su**or**sudo.**
* Calls to **sudo**trigger a lookup in the**/etc/sudoers** file, or in the **/etc/sudoers.d** directory, which first validates that the calling user is allowed to use**sudo**and that it is being used within permitted scope.
* One of the most powerful features of **sudo**is its ability to log unsuccessful attempts at gaining root access. By default, **sudo** commands and failures are logged in **/var/log/auth.log**under the Debian family and **/var/log/messages** in other distribution families.
* One process cannot access another process’ resources, even when that process is running with the same user privileges.
* Using the user credentials, the system verifies the authenticity and identity.
* The SHA-512 algorithm is typically used to encode passwords. They can be encrypted, but not decrypted.
* Pluggable Authentication Modules (PAM) can be configured to automatically verify that passwords created or modified using the **passwd**utility are strong enough (what is considered strong enough can also be configured).
* Your IT security policy should start with requirements on how to properly secure physical access to servers and workstations.
* Keeping your systems updated is an important step in avoiding security attacks.