**Command Line Operations**

**Basic Operations**

**Locating Applications**

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

$ which diff

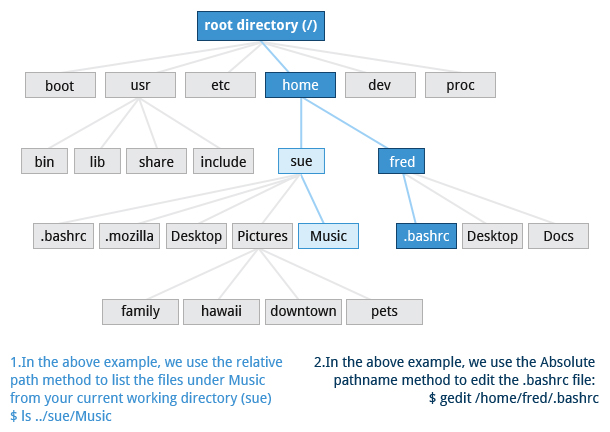
If **which** does not find the program, **whereis** is a good alternative because it looks for packages in a broader range of system directories:

$ whereis diff

**Accessing Directories**

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

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

**Absolute and Relative Paths**

There are two ways to identify paths:

1. **Absolute pathname**: An absolute pathname begins with the root directory and follows the tree, branch by branch, until it reaches the desired directory or file. Absolute paths always start with /.
2. **Relative pathname**: A relative pathname starts from the present working directory. Relative paths never start with /.

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

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

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

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

**Exploring the Filesystem**

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

The following commands can help in exploring the filesystem:

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

**Hard and Soft (Symbolic) Links**

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

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

$ ln file1 file2

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

$ ls -li file1 file2

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

file1 cannot be deleted until the link is removed. file2 will take its place.

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

$ ln -s file1 file4  
$ ls -li file1 file4

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

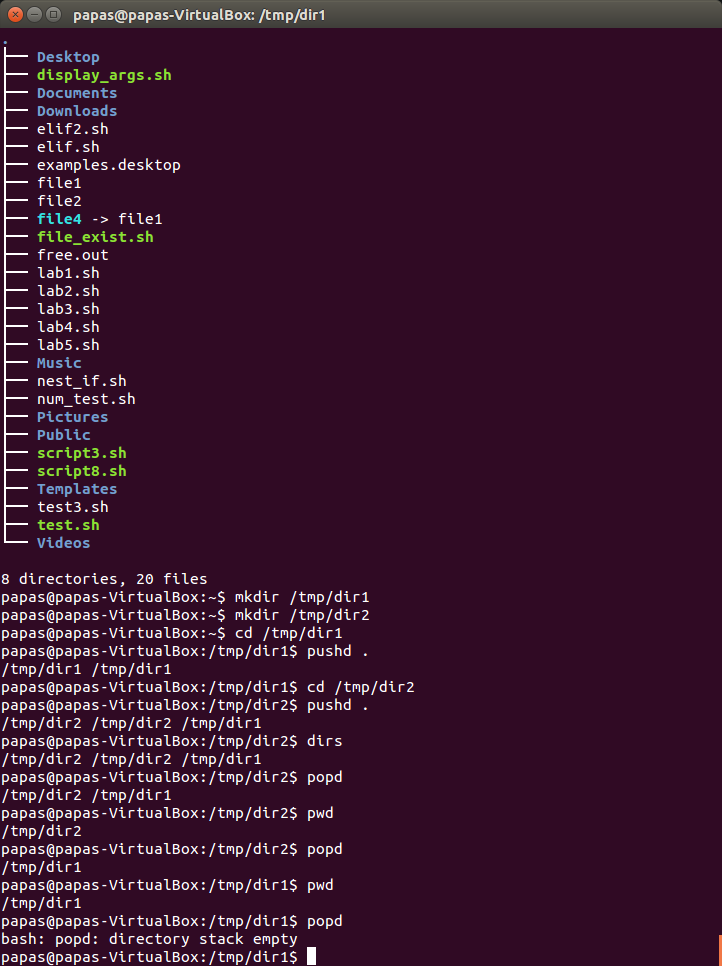
Symbolic links take no extra space on the filesystem (unless their names are very long). They are extremely convenient as they can easily be modified to point to different places. An easy way to create a shortcut from your **home** directory to long pathnames is to create a symbolic link.

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

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

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

**Navigating the Directory History**

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

**Standard File Streams**

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

In Linux, all open files are represented internally by what are called **file descriptors**. Simply put, these are represented by numbers starting at zero.

**stdin** is file descriptor 0, **stdout** is file descriptor 1, and **stderr** is file descriptor 2.

Typically, if other files are opened in addition to these three, which are opened by default, they will start at file descriptor 3 and increase from there.

**I/O Redirection**

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

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

$ do\_something < input-file

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

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

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

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

**bash**permits an easier syntax for the above:

$ do\_something >& all-output-file

**Pipes**

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

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

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

**locate**

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

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

$ locate zip | grep bin

which will list all files and directories with both "zip" and "bin" in their name . (We will cover **grep**in much more detail later.) Notice the use of **|** to pipe the two commands together.

**locate** utilizes the database created by another program, **updatedb.** Most Linux systems run this automatically once a day. However, you can update it at any time by just running **updatedb** from the command line as the root user.

**Wildcards and Matching File Names**

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

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

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

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

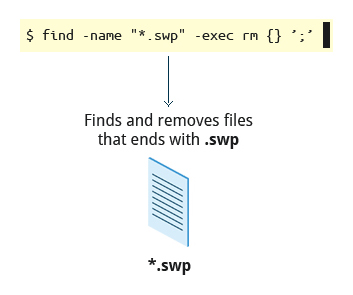
**find**

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

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

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

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

**Advanced find Options**

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

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

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

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

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

**Finding Files Based on Time and Size**

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

Finding based on time:  
$ find / -ctime 3

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

Finding based on sizes:

$ find / -size 0

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

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

**Viewing Files**

You can use the following utilities to view files:

|  |  |
| --- | --- |
| **Command** | **Usage** |
| **cat** | Used for viewing files that are not very long; it does not provide any scroll-back. |
| **tac** | Used to look at a file backwards, one line at a time. |
| **less** | Used to view larger files because it is a paging program; it pauses at each screenful of text, provides scroll-back capabilities, and lets you search and navigate within the file. Note: Use / to search for a pattern in the forward direction and ? for a pattern in the backward direction. |
| **tail** | Used to print the last 10 lines of a file by default. You can change the number of lines by doing -n 15 or just -15 if you wanted to look at the last 15 lines instead of the default. |
| **head** | The opposite of **tail**; by default it prints the first 10 lines of a file. |

**touch and mkdir**

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

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

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

**touch** provides the following options:

* The -t option allows you to set the date and time stamp of the file.
* The -c option allows you to test (or check) the directory permission (to see if you can create files in the directory).

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

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

**mkdir** is used to create a directory.

* To create a sample directory named sampdir under the current directory, type mkdir sampdir.
* To create a sample directory called sampdir under /usr, type mkdir /usr/sampdir.

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

**Removing a File**

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

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

|  |  |
| --- | --- |
| **Command** | **Usage** |
| mv | Rename a directory |
| rmdir | Remove an empty directory |
| rm -rf | Forcefully remove a directory recursively |

**Modifying the Command Line Prompt**

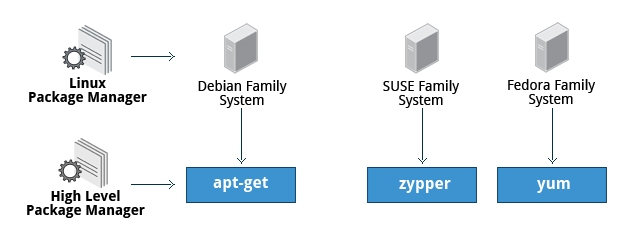
The **PS1** variable is the character string that is displayed as the prompt on the command line. Most distributions set **PS1** to a known default value, which is suitable in most cases. However, users may want custom information to show on the command line. For example, some system administrators require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

**Working With Different Package Management Systems**

* The **Advanced Packaging Tool** (apt) is the underlying package management system that manages software on Debian-based systems. While it forms the backend for graphical package managers, such as the **Ubuntu Software Center** and **synaptic**, its native user interface is at the command line, with programs that include apt-get and apt-cache.
* **Yellowdog Updater Modified** (**yum**) is an open-source command-line package-management utility for RPM-compatible Linux systems, basically what we have called the **Fedora** family. **yum** has both command line and graphical user interfaces.
* **zypper** is a package management system for **openSUSE** that is based on RPM. **zypper** also allows you to manage repositories from the command line. **zypper** is fairly straightforward to use and resembles **yum** quite closely.

**Documentation**

**man**

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

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

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

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

**Manual Chapters**

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

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

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

$ man 3 printf   
$ man -a printf

**GNU Command Line Info Browser**

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

You can view help for a particular topic by typing info <topic name>. The  system then searches for the topic in all available **info** files. Some useful keys are: **q** to quit, **h** for help, and **Enter** to select a menu item.

**info Page Structure**

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

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

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

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

**help Option**

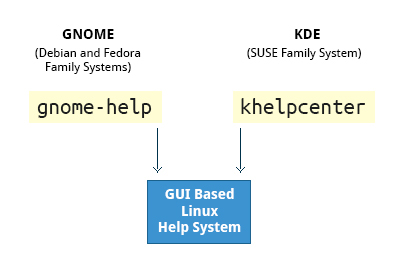
The third source of Linux documentation is use of the **help** option. **help** command displays a short synopsis of built-in shell commands.

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

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

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

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

**Desktop 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. These programs usually contain custom help for the desktop itself and some of its applications, and will often also include graphically rendered **info** and **man pages**.

You can also start the graphical help system from a graphical terminal using the following commands:

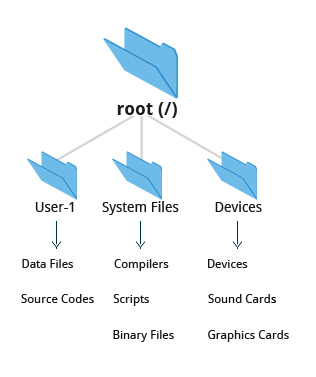
* **GNOME**: **gnome-help**
* **KDE:** **khelpcenter**

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

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

**File Operations**

**FIlesystems**

**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/awk, where the last element is the actual file name.

**More About Mount Points**

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. Arguments include the **device node** and **mount point**. For example,

$ 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. (Note that unless the system is otherwise configured only the root user has permission to run **mount**.) If you want it to be automatically available every time the system starts up, you need to edit the file /etc/fstab accordingly (the name is short for **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 usage statistics about currently used and available space.

**NFS on the Server**

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

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

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

/projects \*.example.com(rw)

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

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

**NFS on the Client**

On the client machine, if it is desired to have the remote 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 default 0 0

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

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

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

**proc Filesystem**

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

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

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

/proc has subdirectories as well, including:

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

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 disk.

**FIlesystem Architecture**

**Overview of Home Directories**

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

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

The /home directory is often mounted as a separate 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/

**The /bin and /sbin Directories**

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

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

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

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

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

**The /dev Directory**

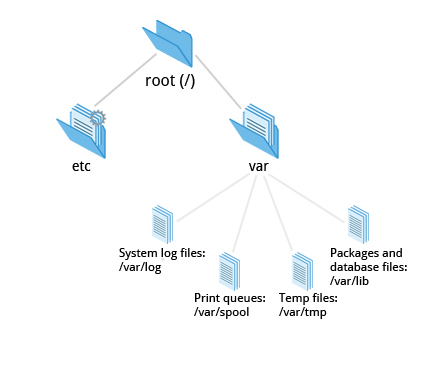
The /dev directory contains **device** **nodes**, a type of pseudo-file used by most hardware and software devices, except for network devices. This directory is:

* Empty on the disk partition when it is not mounted
* Contains entries which are created by the **udev** system, which creates and manages device nodes on Linux, creating them dynamically when devices are found. The /dev directory contains items such as:
* /dev/sda1 (first partition on the first hard disk)
* /dev/lp1 (second printer)
* /dev/dvd1 (first DVD drive)

**The /var and /etc Directories**

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

* System log files: /var/log
* Packages and database files: /var/lib
* Print queues: /var/spool
* Temp files: /var/tmp

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

**The /boot Directory**

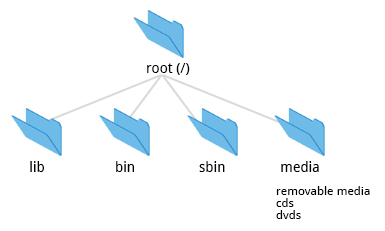
The /boot directory contains the few essential files needed to boot the system. For every alternative kernel installed on the system there are four files:

1. vmlinuz: the compressed Linux kernel, required for booting
2. initramfs: the initial ram 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.

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

**The /lib and /media Directories**

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

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

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

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

A list of additional directories under /and their use:

|  |  |
| --- | --- |
| **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. |

**Subdirectories under /usr**

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

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

**Comparing Files and File Types**

**Comparing Files**

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 utiility program has many useful options (see man diff) including:

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

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

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

**Using diff3 and patch**

You can compare three files at once using **diff3**, which uses one file as the reference basis for the other two. For example, suppose you and a co-worker both have made modifications to the same file working at the same time independently. **diff3** can show the differences based on the common file you both started with. The syntax for **diff3** is as follows:

$ diff3 MY-FILE COMMON-FILE YOUR-FILE

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

$ diff -Nur originalfile newfile > patchfile

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

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

$ patch -p1 < patchfile  
$ patch originalfile patchfile

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

**'file' utility**

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

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

**Backing Up Data**

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

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

**cp** can only copy files to and from destinations on the local machine (unless you are copying to or from a 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 [user@]host. The user@ part is optional and used if the remote user is different from the local user.

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

**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 are copied to destinationfile.

**Compressing Data**

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

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

|  |  |
| --- | --- |
| **Command** | **Usage** |
| gzip | The most frequently used Linux compression utility |
| bzip2 | Produces files significantly smaller than those produced by **gzip** |
| xz | The most space efficient compression utility used in Linux |
| zip | Is often required to examine and decompress archives from other operating systems |

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

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

**Compressing Data Using gzip**

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

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

**Compressing Data Using bzip2**

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

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

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

**Handling Files Using zip**

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

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

**Archiving and Compressing Data Using tar**

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

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

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

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

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

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

**Disk-to-Disk Copying**

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

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

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

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

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

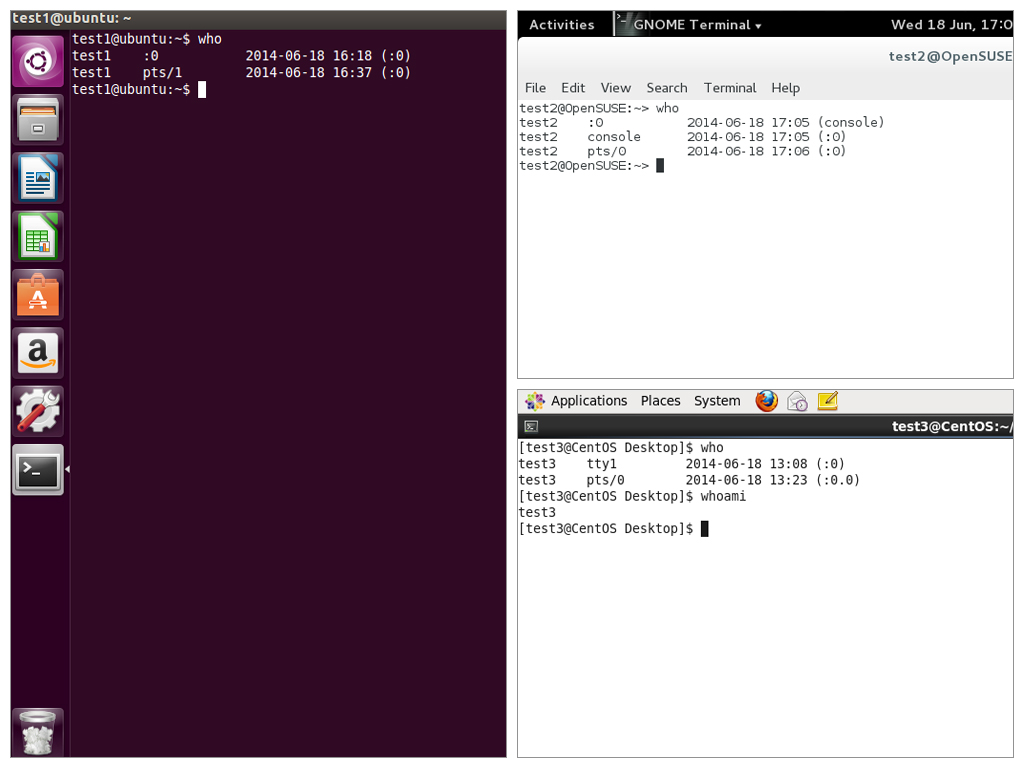
**Do not experiment with this command as written above as it can erase a hard disk!**

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

**User Enviroment**

**Accounts**

**Identifying the Current User**

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

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

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

Click the image to view an enlarged version.

**Basics of Users and Groups**

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

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

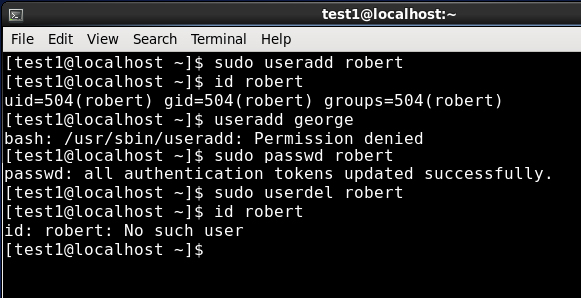
Historically **Fedora**-family systems start **uid**'s at 500; other distributions begin at 1000.

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

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

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

**Adding and Removing Users**

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

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

$ sudo useradd turkey

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

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

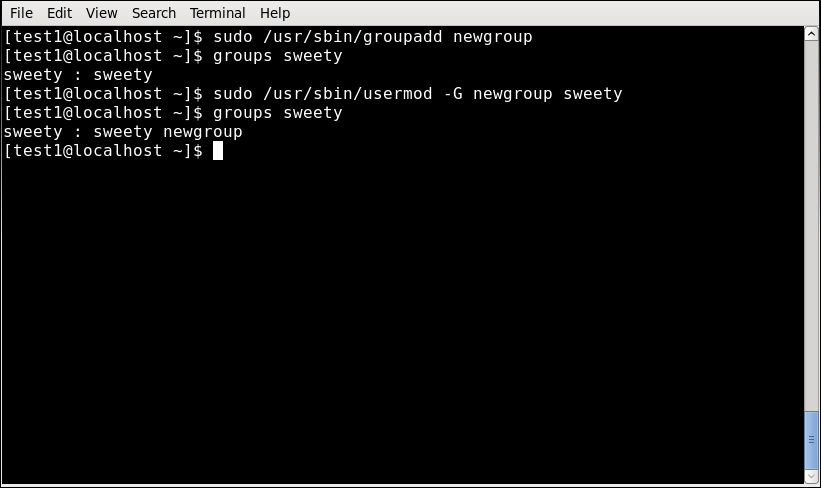
and sets the default shell to /bin/bash. Removing a user account is as easy as typing userdel turkey However, this will leave the /home/turkey directory intact. This might be useful if it is a temporary inactivation. To remove the home directory while removing the account one needs to use the **-r**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

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

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

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

**Adding and Removing Groups**

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

$ sudo /usr/sbin/groupadd newgroup

The group can be removed with

$ sudo /usr/sbin/groupdel newgroup

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

$ groups turkey  
turkey : turkey

and then add the new group:

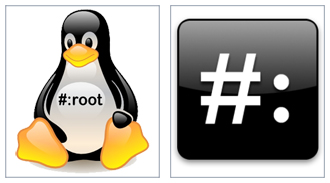
$ sudo /usr/sbin/usermod -G anewgroup turkey  
$ groups turkey  
turkey: turkey anewgroup

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

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

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

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

**The root Account**

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

However, you can use the **sudo** feature to assign more limited privileges to user accounts:

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

**su and sudo**

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

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

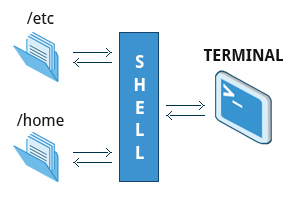
**Elevating to root Account**

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

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

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

**Startup Files**

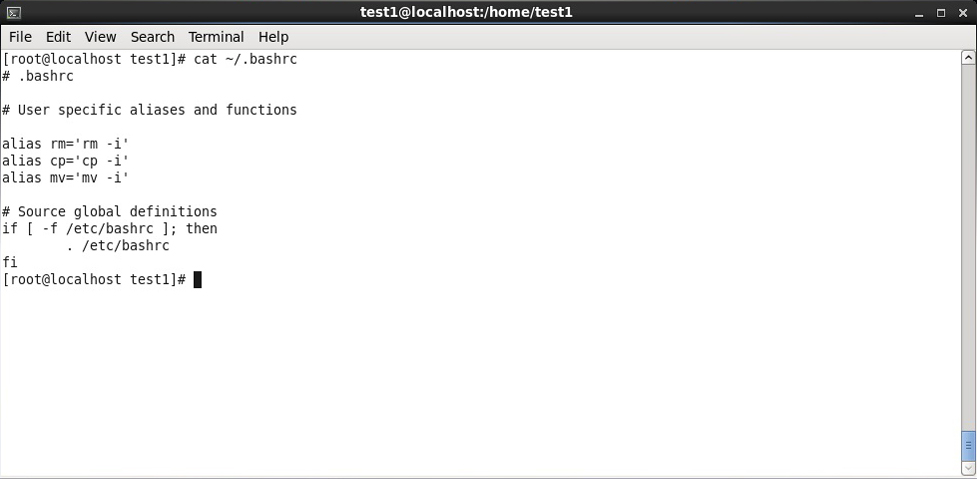
In Linux, the command shell program (generally **bash)**uses one or more startup files to configure the environment. Files in the /etc directory define global settings for all users while Initialization files in the user's home directory can include and/or override the global settings.

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

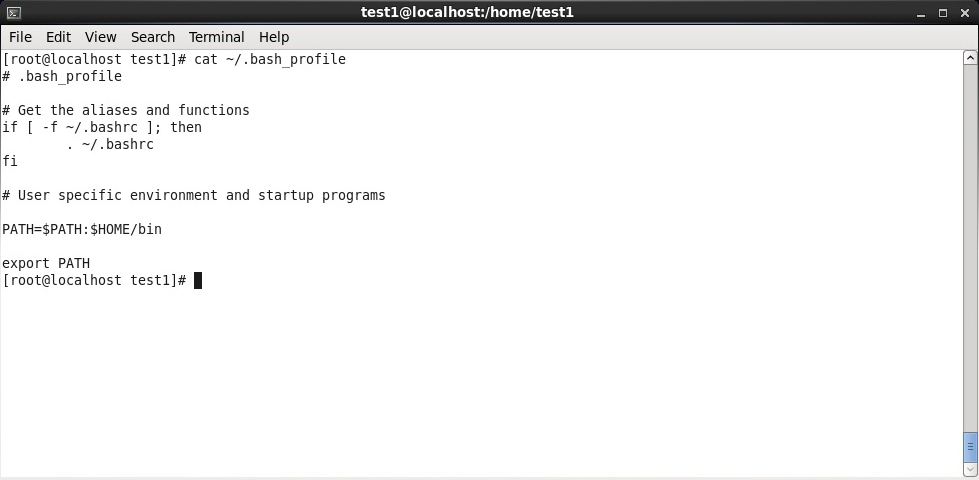
* Customizing the user's prompt
* Defining command-line shortcuts and aliases
* Setting the default text editor
* Setting the **path** for where to find executable programs

**Order of the Startup Files**

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

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

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

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

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

**Environment Variables**

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

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

**Setting Environment Variables**

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

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

**The HOME Variable**

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

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

**The PATH Variable**

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

* :path1:path2
* path1::path2

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

To prefix a private bin directory to your path:

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

**The PS1 Variable**

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

papas@papas-VirtualBox:~$ echo $PS1

\[\e]0;\u@\h: \w\a\]${debian\_chroot:+($debian\_chroot)}\u@\h:\w\$

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

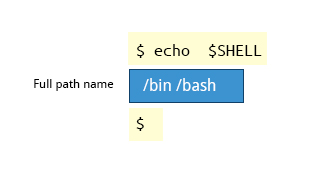
\u - User name   
\h - Host name   
\w - Current working directory   
\! - History number of this command   
\d - Date

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

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

Even better practice would be to save the old prompt first and then restore, as in:  
$ OLD\_PS1=$PS1

change the prompt, and eventually change it back with:  
$ PS1=$OLD\_PS1  
$

**The SHELL Variable**

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

$ echo $SHELL  
/bin/bash  
$

**Recalling Previous Commands**

**Using History Environment Variables**

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

HISTFILE stores the location of the history file.

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

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

**Finding and Using Previous Commands**

Specific keys to perform various tasks:

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

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

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

The following is an example of how you can use the CTRL-R command to search through the command history:  
  
$ ^R                                                              # This all happens on 1 line  
(reverse-i-search)'s': sleep 1000    # Searched for 's'; matched "sleep"  
$ sleep 1000                                              # Pressed Enter to execute the searched command  
$

**Executing Previous Commands**

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

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

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

Here are more examples:  
$ history

1. echo $SHELL
2. echo $HOME
3. echo $PS1
4. ls -a
5. ls -l /etc/ passwd
6. sleep 1000
7. history

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

**Keyboard Shortcuts**

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

|  |  |
| --- | --- |
| **Keyboard Shortcut** | **Task** |
| **CTRL-L** | Clears the screen |
| **CTRL-D** | Exits the current shell |
| **CTRL-Z** | Puts the current process into suspended background |
| **CTRL-C** | Kills the current process |
| **CTRL-H** | Works the same as backspace |
| **CTRL-A** | Goes to the beginning of the line |
| **CTRL-W** | Deletes the word before the cursor |
| **CTRL-U** | Deletes the entire line |
| **Tab** | Auto-completes files, directories, and binaries |

**Command Aliases**

**Creating Aliases**

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

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

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

**File Ownership**

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

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

|  |  |
| --- | --- |
| **Command** | **Usage** |
| chown | Used to change user ownership of a file or directory |
| chgrp | Used to change group ownership |
| chmod | Used to change the permissions on the file which can be done separately for **owner**, **group** and the rest of the world (often named as **other**.) |

**File Permission Modes and chmod**

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

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

rwx: rwx: rwx  
 u:   g:   o

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

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

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

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

* 4 if read permission is desired.
* 2 if write permission is desired.
* 1 if execute permission is desired.

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

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

$ chmod 755 a\_file  
$ ls -l a\_file  
-rwxr-xr-x 1 coop coop 1601 Mar 9 15:04 a\_file

**Example of chown**

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

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

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

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

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

**Example of chgrp**

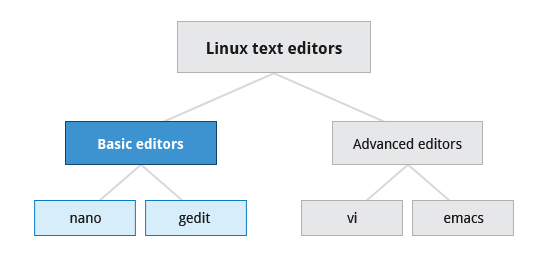
Now let’s see an example of changing group ownership using **chgrp**:

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

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

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

**Text Editors**

**Overview of Text Editors in Linux**

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

Linux Administrators quite often sidestep the text editors, by using graphical utilities for creating and modifying system configuration files. However,  this can be far more laborious than directly using a text editor. Note that word processing applications such as **Notepad** or the applications that are part of office suites are not really basic text editors because they add a lot of extra (usually invisible) formatting information that will probably render system administration configuration files unusable for their intended purpose. So using text editors really is essential in Linux.

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

**Creating Files Without Using an Editor**

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

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

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

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

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

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

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

and are extremely useful when employed by scripts.

**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. One particularly easy one to use is the text-terminal based editor **nano**. Just invoke **nano**by giving a file name as an argument. All the help you need is displayed at the bottom of the screen, and you should be able to proceed without any problem.

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

**nano**

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

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

* CTRL-G: Display the help screen
* CTRL-O: Write to a file
* CTRL-X: Exit a file
* CTRL-R: Insert contents from another file to the current buffer
* CTRL-C: Cancels previous commands

**gedit**

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

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

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

**Local Security Principles**

**User Accounts**

The Linux kernel allows properly authenticated users to access files and applications. While each user is identified by a unique integer (the user id or **UID),** a separate database associates a **username** with each UID. Upon account creation, new user information is added to the user database and the user's home directory must be created and populated with some essential files. Command line programs such as **useradd** and **userdel**as well 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 (except for RHEL, which reserves only up to 499) * Normal users have UID's of 1000 or greater, except on RHEL where they start at 500 |
| Group ID (GID) | The primary Group ID (GID); Group Identification Number stored in the /etc/group file | Will be covered in detail in the chapter on Processes |
| User Info | This field is optional and allows insertion of extra information about the user such as their name | For example: Rufus T. Firefly |
| Home Directory | The absolute path location of user's home directory | For example: /home/rtfirefly |
| Shell | The absolute location of a user's default shell | For example: /bin/bash |

**Types of Accounts**

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

* root
* System
* Normal
* Network

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

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

**Understanding the root Account**

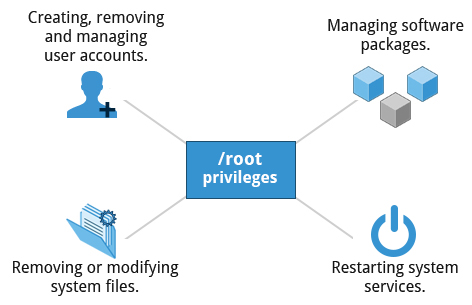
**root** is the most privileged account on a Linux/UNIX system. This account has the ability to carry out all facets of system administration, including adding accounts, changing user passwords, examining log files, installing software, etc. Utmost care must be taken when using this account. It has no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

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

**Usage of the Root Account**

**Operations that Require root Privileges**

**root**privileges are required to perform operations such as:

* Creating, removing and managing user accounts.
* Managing software packages.
* Removing or modifying system files.
* Restarting system services.

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

**Creating a New User in Linux**

1. At the command prompt, as root type useradd <username> and press the **ENTER** key.
2. To set the initial password, type passwd <username>  and press the **ENTER** key. The **New password**: prompt is displayed.
3. Enter the password and press the **ENTER** key.  
   To confirm the password, the prompt **Retype new password**: is displayed.
4. Enter the password again and press the **ENTER** key.  
   The message **passwd: all authentication tokens updated successfully**. is displayed.

**Operations That Do Not Require root Privileges**

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

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

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

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

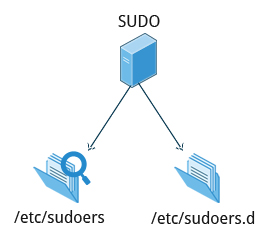
**Using sudo, Importance of Process Isolation, Limiting Hardware Access and Keeping Systems Current.**

**omparing 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. |

**sudo Features**

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

**The sudoers File**

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

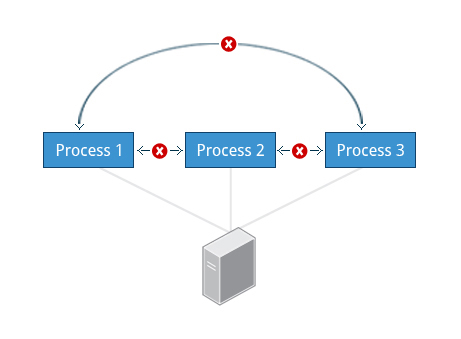
The basic structure of an entry is:  
who where = (as\_whom) what

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

**Command Logging**

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

* Calling username
* Terminal info
* Working directory
* User account invoked
* Command with arguments

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

**Process Isolation**

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

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

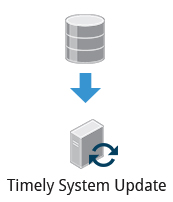
* **Control Groups (cgroups)**: Allows system administrators to group processes and associate finite resources to each cgroup.
* **Linux Containers (LXC)**: Makes it possible to run multiple isolated Linux systems (containers) on a single system by relying on **cgroups**.
* **Virtualization**: Hardware is emulated in such a way that not only processes can be isolated, but entire systems are run simultaneously as isolated and insulated guests (virtual machines) on one physical host.

**Hardware Device Access**

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

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

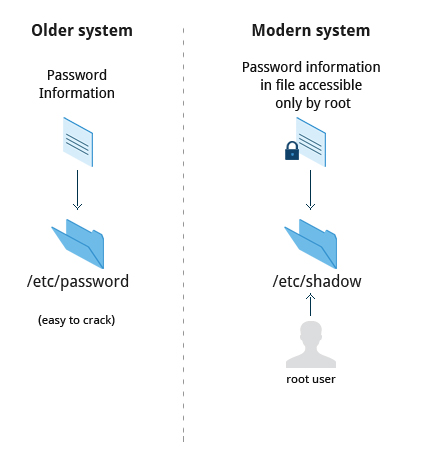
 $ echo hello world > /dev/sda1

the standard permissions as shown in the figure make it impossible for regular users to do so.  Writing to a device in this fashion can easily obliterate the 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.

**Keeping Current**

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

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

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

**Working with Passwords**

**How Passwords are Stored**

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

**Password Encryption**

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

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

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

papas@papas-VirtualBox:~$ echo -n free.out | sha512sum

e37c6191ee82d2ca46f192f492a132b930ccd5d44f0b14a42e3baa0a65ae57aa26201fd1b75397c875f1a246d393536d21bb8eab46ec9744e1e09de3b28f9e36 -

**Good Password Practices**

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

1. **Password aging** is a method to ensure that users get prompts that remind them to create a new password after a specific period. This can ensure that passwords, if cracked, will only be usable for a limited amount of time. This feature is implemented using **chage,** which configures the password expiry information for a user.
2. Another method is to force users to set strong passwords using **Pluggable Authentication Modules (PAM)**. **PAM** can be configured to automatically verify that a password created or modified using the **passwd**utility is sufficiently strong. **PAM** configuration is implemented using a library called **pam\_cracklib.so**, which can also be replaced by **pam\_passwdqc.so** for more options.
3. One can also install password cracking programs, such as **Jack The Ripper**, to secure the password file and detect weak password entries. It is recommended that written authorization be obtained before installing such tools on any system that you do not own.

**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 **GRUB** boot loader, for the older **GRUB** version 1, you can invoke **grub-md5-crypt** which will prompt you for a password and then encrypt as shown on the adjoining screen.

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

password --md5 $1$Wnvo.1$qz781HRVG4jUnJXmdSCZ30

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

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

**Hardware Vulnerability**

When hardware is physically accessible, security can be compromised:

* 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

**Network Operations**

**Introduction to Networking**

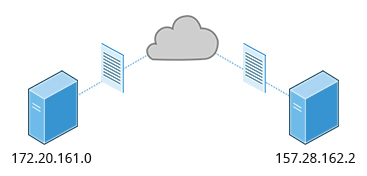
**Introduction to Networking**

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

A network is used to:

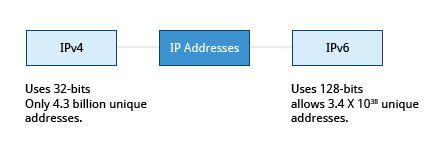
* Allow the connected devices to communicate with each other.
* Enable multiple users to share devices over the network, such as printers and scanners.
* Share and manage information across computers easily.

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

**IP Addresses**

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

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

**IPv4 and IPv6**

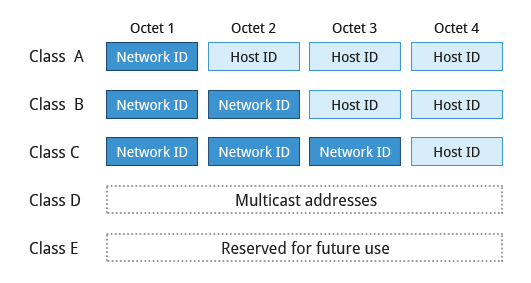
There are two different types of IP addresses available: **IPv4** and **IPv6**. **IPv4** is older and by far the more widely used, while **IPv6** is newer and is designed to get past the limitations of the older standard and furnish many more possible addresses.

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

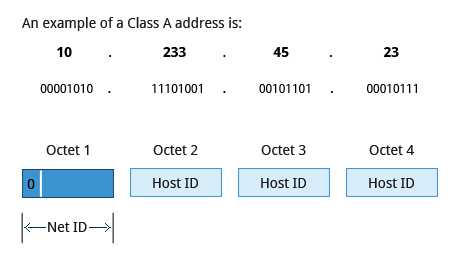
**IPv6** uses 128-bits for addresses; this allows for 3.4 X 1038 unique addresses. If you have a larger network of computers and want to add more, you may want to move to **IPv6**, because it provides more unique addresses. However, it is difficult to move to **IPv6** as the two protocols do not inter-operate. Due to this, migrating equipment and addresses to **IPv6** requires significant effort and hasn't been as fast as was originally intended.

**Decoding IPv4 Addresses**

A 32-bit IPv4 address is divided into four 8-bit sections called [octets](http://en.wikipedia.org/wiki/Octet_%28computing%29).

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

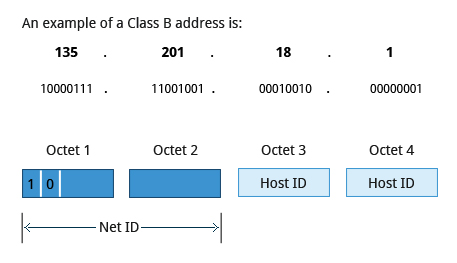
Network address are divided into five classes: A, B, C, D, and E. Classes A, B, and C are classified into two parts: **Network addresses (Net ID)** and **Host address (Host ID)**. The Net ID is used to identify the network, while the Host ID is used to identify a host in the network. Class D is used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

**Class A Network Addresses**

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

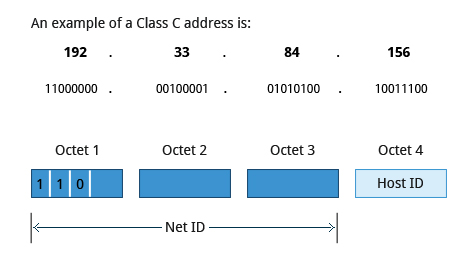
Each Class A network can have up to 16.7 million unique hosts on its network. The range of host address is from 1.0.0.0 to 127.255.255.255.

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

**Class B Network Addresses**

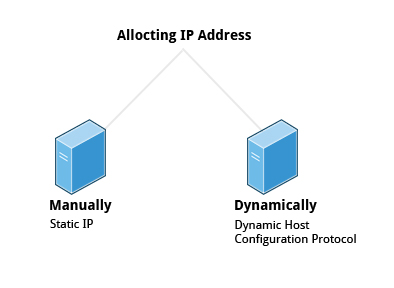
Class B addresses use the first two octets of the IP address as their Net ID and the last two octets as the Host ID. The first two bits of the first octet are always set to binary 10, so there are a maximum of 16,384 (14-bits) Class B networks. The first octet of a Class B address has values from 128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

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

**Class C Network Addresses**

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

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

**IP Address Allocation**

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

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

**Manually Allocating an IP Address**

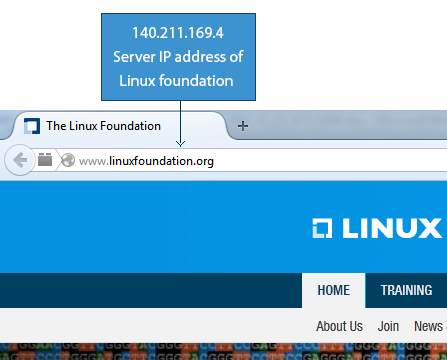
Before an IP address can be allocated manually, one must identify the size of the network by determining the host range; this determines which network class (A, B, or C) can be used. The **ipcalc**program can be used to ascertain the host range.

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

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

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

**Name Resolution**

**Name Resolution** is used to convert numerical IP address values into a human-readable format known as the **hostname**. For example, 140.211.169.4 is the numerical IP address that refers to the **linuxfoundation.org** hostname. Hostnames are easier to remember.

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

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

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

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

papas@papas-VirtualBox:~$ cat /etc/**hosts**

127.0.0.1 localhost

127.0.1.1 papas-VirtualBox

# The following lines are desirable for IPv6 capable hosts

::1 ip6-localhost ip6-loopback

fe00::0 ip6-localnet

ff00::0 ip6-mcastprefix

ff02::1 ip6-allnodes

ff02::2 ip6-allrouters

papas@papas-VirtualBox:~$ cat /etc/**resolv.conf**

# Dynamic resolv.conf(5) file for glibc resolver(3) generated by resolvconf(8)

# DO NOT EDIT THIS FILE BY HAND -- YOUR CHANGES WILL BE OVERWRITTEN

nameserver 127.0.1.1

search lan

To lookup host names using DNS:

papas@papas-VirtualBox:~$ **host** Linuxfoundation.org

Linuxfoundation.org has address 140.211.169.4

Linuxfoundation.org mail is handled by 10 ASPMX3.GOOGLEMAIL.COM.

Linuxfoundation.org mail is handled by 1 ASPMX.L.GOOGLE.COM.

Linuxfoundation.org mail is handled by 5 ALT1.ASPMX.L.GOOGLE.COM.

Linuxfoundation.org mail is handled by 5 ALT2.ASPMX.L.GOOGLE.COM.

Linuxfoundation.org mail is handled by 10 ASPMX2.GOOGLEMAIL.COM.

To lookup server names interactively:

papas@papas-VirtualBox:~$ **nslookup** Linuxfoundation.org

Server: 127.0.1.1

Address: 127.0.1.1#53

Non-authoritative answer:

Name: Linuxfoundation.org

Address: 140.211.169.4

To lookup domain name information from nameserver:

papas@papas-VirtualBox:~$ **dig** Linuxfoundation.org

; <<>> DiG 9.9.5-3-Ubuntu <<>> Linuxfoundation.org

;; global options: +cmd

;; Got answer:

;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 2119

;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:

; EDNS: version: 0, flags:; udp: 4096

;; QUESTION SECTION:

;Linuxfoundation.org. IN A

;; ANSWER SECTION:

Linuxfoundation.org. 10702 IN A 140.211.169.4

;; Query time: 10 msec

;; SERVER: 127.0.1.1#53(127.0.1.1)

;; WHEN: Wed Aug 20 16:40:44 EEST 2014

;; MSG SIZE rcvd: 64

**Network Interfaces**

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

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

papas@papas-VirtualBox:~$ ifconfig

eth0 Link encap:Ethernet HWaddr 08:00:27:b5:47:12

inet addr:10.0.2.15 Bcast:10.0.2.255 Mask:255.255.255.0

inet6 addr: fe80::a00:27ff:feb5:4712/64 Scope:Link

UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1

RX packets:2510 errors:0 dropped:0 overruns:0 frame:0

TX packets:1324 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:1000

RX bytes:1851686 (1.8 MB) TX bytes:109056 (109.0 KB)

lo Link encap:Local Loopback

inet addr:127.0.0.1 Mask:255.0.0.0

inet6 addr: ::1/128 Scope:Host

UP LOOPBACK RUNNING MTU:65536 Metric:1

RX packets:74 errors:0 dropped:0 overruns:0 frame:0

TX packets:74 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:0

RX bytes:6351 (6.3 KB) TX bytes:6351 (6.3 KB)

**Network Configuration Files**

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

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

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

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

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

**Network Configuration Commands**

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

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

**ip** is a very powerful program that can do many things. Older (and more specific) utilties 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 utilties.

**ping**

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

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

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

**route**

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

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

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

**traceroute**

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

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

**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. |

**Browsers**

**Graphical and Non-Graphical Browsers**

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

|  |  |
| --- | --- |
| **Non-Graphical Browsers** | **Description** |
| [lynx](http://lynx.browser.org/) | Configurable text-based web browser; the earliest such browser and still in use. |
| [links or elinks](http://elinks.or.cz/) | Based on**lynx.** It can display tables and frames. |
| [w3m](http://w3m.sourceforge.net/) | Newer text-based web browser with many features. |

**wget**

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

* Large file downloads
* Recursive downloads, where a web page refers to other web pages and all are downloaded at once
* Password-required downloads
* Multiple file downloads

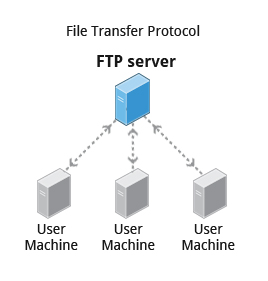
To download a webpage, you can simply type wget <url>, and then you can read the downloaded page as a local file using a graphical or non-graphical browser.

**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> , type curl <http://www.linuxfoundation.org>.

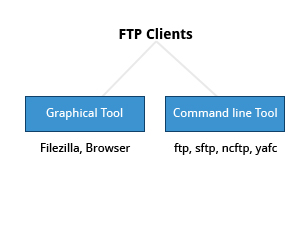
To get the contents of a web page and store it to a file, type curl -o saved.html <http://www.mysite.com>. The contents of the main index file at the website will be saved in saved.html.

**Transfering Files**

**FTP (File Transfer Protocol)**

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

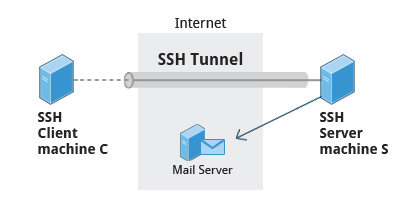
**FTP Clients**

FTP clients enable you to transfer files with remote computers using the FTP protocol. These clients can be either graphical or command line tools. **Filezilla**, for example, allows use of the drag-and-drop approach to transfer files between hosts. All web browsers support FTP, all you have to do is give a URL like : ftp://ftp.kernel.org where the usual http:// becomes ftp://.

Some command line FTP clients are:

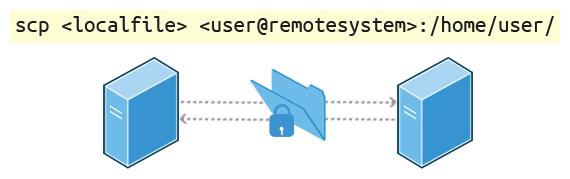
* **ftp**
* **sftp**
* **ncftp**
* **yafc** (Yet Another FTP Client)

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

**SSH: Executing Commands Remotely**

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

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

**Copying Files Securely with scp**

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.

**Manipulating Text**

**cat and echo**

**Command Line Tools**

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

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

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

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

**cat**

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

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

|  |  |
| --- | --- |
| **Command** | **Usage** |
| cat file1 file2 | Concatenate multiple files and display the output; i.e., the entire content of the first file is followed by that of the second file. |
| cat file1 file2 > newfile | Combine multiple files and save the output into a new file. |
| cat file >> existingfile | Append a file to the end of an existing file. |
| cat > file | Any subsequent lines typed will go into the file until CTRL-D is typed. |
| cat >> file | Any subsequent lines are appended to the file until CTRL-D is typed. |

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

$ tac file  
$ tac file1 file2 > newfile

**Using cat Interactively**

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

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

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

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

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

**echo**

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

$ echo string

**echo** can be used to display a string on **standard output** (i.e. 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

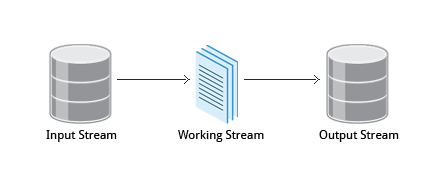
**echo**is particularly useful for viewing the values of environment variables (built-in shell variables). For example, echo $USERNAME will print the name of the user who has logged into the current terminal.

|  |  |
| --- | --- |
| **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. |

**sed and awk**

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

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

**sed**

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

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

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

**sed Command Syntax**

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

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

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

**sed Basic Operations**

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

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

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

$ sed s/pattern/replace\_string/g file > file2

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

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

papas@papas-VirtualBox:~$ cat > test1

this is a sample file for sed command

it is used to edit contents in a file

it replaces words in a line of file

papas@papas-VirtualBox:~$ sed /s/is/are/ test1

s/are/

this is a sample file for sed command

s/are/

it is used to edit contents in a file

s/are/

it replaces words in a line of file

papas@papas-VirtualBox:~$ cat test1

this is a sample file for sed command

it is used to edit contents in a file

it replaces words in a line of file

papas@papas-VirtualBox:~$ sed 1,2s/is/are/g test1

thare are a sample file for sed command

it are used to edit contents in a file

it replaces words in a line of file

papas@papas-VirtualBox:~$ cat test1

this is a sample file for sed command

it is used to edit contents in a file

it replaces words in a line of file

papas@papas-VirtualBox:~$ sed s/is/are/g test1

thare are a sample file for sed command

it are used to edit contents in a file

it replaces words in a line of file

papas@papas-VirtualBox:~$ cat test1

this is a sample file for sed command

it is used to edit contents in a file

it replaces words in a line of file

It doesn’t save the changes since we do not save the output of sed command

**awk**

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

**awk** has the following features:

* It is a powerful utility and interpreted programming language.
* It is used to manipulate data files, retrieving, and processing text.
* It works well with **fields** (containing a single piece of data, essentially a column) and **records** (a collection of fields, essentially a line in a file).

**awk** is invoked as shown in the following:

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

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

**awk Basic Operations**

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

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

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

**File manipulation Utilities**

**sort**

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

**sort** can be used as follows:

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

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

papas@papas-VirtualBox:~$ sort names

archemedes

carol

earth

mars

mercury

nepture

venus

papas@papas-VirtualBox:~$ cat names

mercury

nepture

archemedes

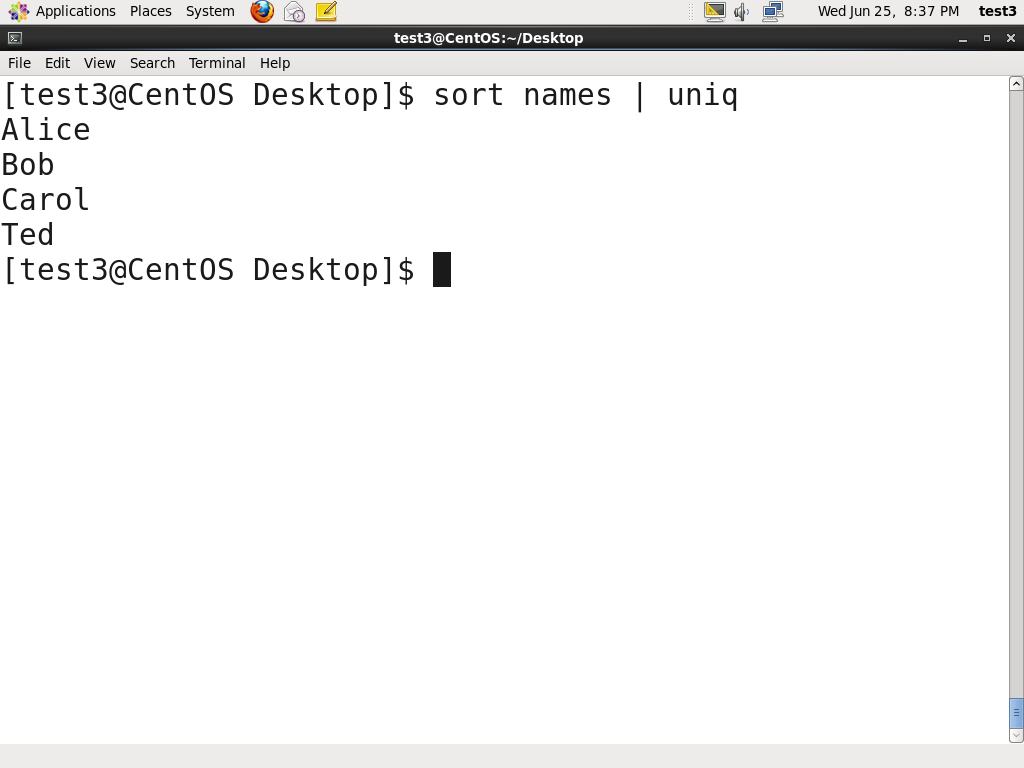
venus

carol

earth

mars

**uniq**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch12_screen_25a.jpg)**uniq**is used to remove duplicate lines in a text file and is useful for simplifying text display. **uniq**requires that the duplicate entries to be removed are consecutive. Therefore one often runs **sort**first and then pipes the output into **uniq**;if **sort** is passed the -u option it can do it all this in one step. In the example shown, the file is called names and was originally Ted, Bob, Alice, Bob, Carol, Alice.

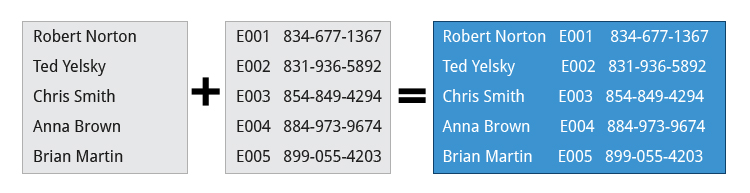
To remove duplicate entries from some files, use the following command: sort file1 file2 | uniq > file3

**OR**

sort -u file1 file2 > file3

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

**paste**



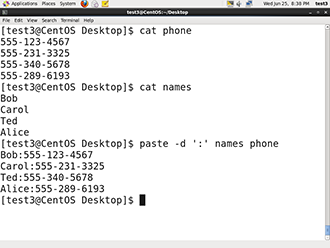
Suppose you have a file that contains the full name of all employees and another file that lists their phone numbers and Employee IDs. You want to create a new file that contains all the data listed in three columns: name, employee ID, and phone number. How can you do this effectively without investing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

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

**paste**accepts the following options:

* -d delimiters, which specify a list of delimiters to be used instead of tabs for separating consecutive values on a single line. Each delimiter is used in turn; when the list has been exhausted, paste begins again at the first delimiter.
* -s, which causes **paste** to append the data in series rather than in parallel; that is, in a horizontal rather than vertical fashion.

**Using paste**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch12_screen_28a.jpg)**paste** can be used to combine fields (such as name or phone number) from different files as well as combine lines from multiple files. For example, line one from file1 can be combined with line one of file2, line two from file1 can be combined with line two of file2, and so on.

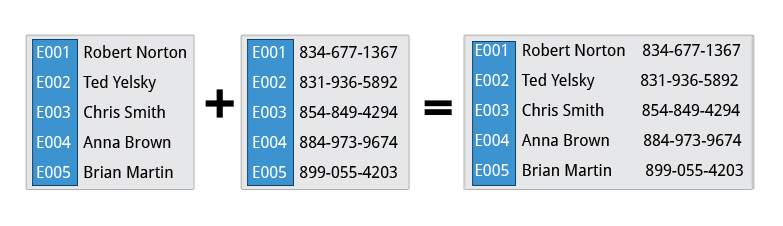
To paste contents from two files one can do:  
$ paste file1 file2

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

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

Click the image to view an enlarged version.

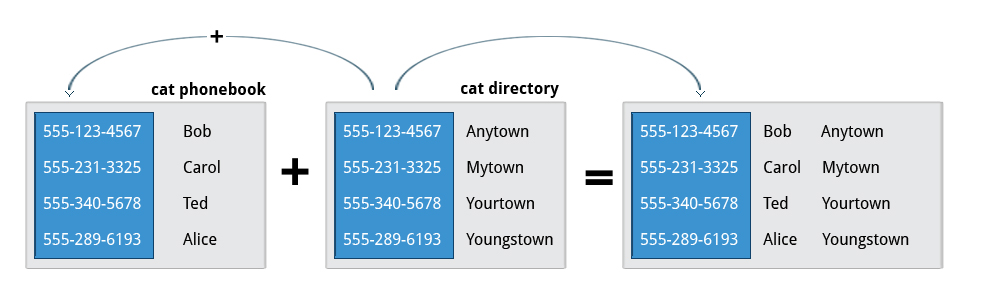
**join**



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.

**Using join**

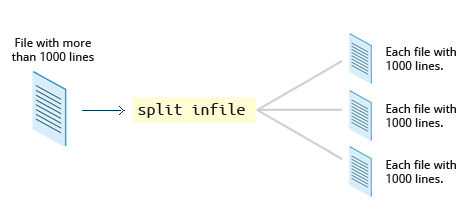


To combine two files on a common field, at the command prompt type join file1 file2 and press the **Enter** key.

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

$ cat phonebook  
555-123-4567 Bob  
555-231-3325 Carol  
555-340-5678 Ted  
555-289-6193 Alice

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

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

**split**

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

To split a file into segments using a different prefix, use the command split infile <Prefix>.

**Using split**

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

$ wc -l american-english  
99171 american-english

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

$ split american-english dictionary

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

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

**Regular Expressions and Search Patterns**

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

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

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

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

**Using Regular Expressions and Search Patterns**

For example, Consider the following sentence:

**the quick brown fox jumped over the lazy dog**

Some of the patterns that can be applied to this sentence are as follows:

|  |  |
| --- | --- |
| **Command** | **Usage** |
| a.. | matches azy |
| b.|j. | matches both br and ju |
| ..$ | matches og |
| l.\* | matches lazy dog |
| l.\*y | matches lazy |
| the.\* | matches the whole sentence |

**grep**

**grep**is an 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. |

The fundamental building blocks are the regular expressions that match a single character. Most characters, including all letters and digits, are regular expressions that match themselves. Any metacharacter with special meaning may be quoted by preceding it with a backslash. A *bracket expression* is a list of characters enclosed by **[** and **]**. It matches any single character in that list; if the first character of the list is the caret **^** then it matches any character *not* in the list. For example, the regular expression **[0123456789]** matches any single digit.

Within a bracket expression, a *range expression* consists of two characters separated by a hyphen. It matches any single character that sorts between the two characters, inclusive, using the locale's collating sequence and character set. For example, in the default C locale, **[a-d]** is equivalent to **[abcd]**. Many locales sort characters in dictionary order, and in these locales **[a-d]** is typically not equivalent to **[abcd]**; it might be equivalent to **[aBbCcDd]**, for example. To obtain the traditional interpretation of bracket expressions, you can use the C locale by setting the **LC\_ALL** environment variable to the value **C**.

Finally, certain named classes of characters are predefined within bracket expressions, as follows. Their names are self explanatory, and they are **[:alnum:]**, **[:alpha:]**, **[:cntrl:]**, **[:digit:]**, **[:graph:]**, **[:lower:]**, **[:print:]**, **[:punct:]**, **[:space:]**, **[:upper:]**, and **[:xdigit:].** For example, **[[:alnum:]]** means **[0-9A-Za-z]**, except the latter form depends upon the C locale and the ASCII character encoding, whereas the former is independent of locale and character set. (Note that the brackets in these class names are part of the symbolic names, and must be included in addition to the brackets delimiting the bracket list.) Most metacharacters lose their special meaning inside lists. To include a literal **]** place it first in the list. Similarly, to include a literal **^** place it anywhere but first. Finally, to include a literal **-** place it last.

The period **.** matches any single character. The symbol **\w** is a synonym for **[[:alnum:]]** and **\W** is a synonym for **[^[:alnum]]**.

The caret **^** and the dollar sign **$** are metacharacters that respectively match the empty string at the beginning and end of a line. The symbols **\<** and **\>** respectively match the empty string at the beginning and end of a word. The symbol **\b** matches the empty string at the edge of a word, and **\B** matches the empty string provided it's *not* at the edge of a word.

Two regular expressions may be joined by the infix operator `|'; the resulting regular expression matches any string matching either subexpression.

use the backslashed versions: \? ,\+ ,\{ ,\| ,\( , and \)

? The preceding item is optional and will be matched at most once.

\* The preceding item will be matched zero or more times.

+ The preceding item will be matched one or more times.

(with \?, \\*, \+ etc)

**1. How can I list just the names of matching files?**

grep -l 'main' \*.c

lists the names of all C files in the current directory whose contents mention `main'.

**2. How do I search directories recursively?**

grep -r 'hello' /home/gigi

searches for `hello' in all files under the directory `/home/gigi'. For more control of which files are searched, use find', `grep' and `xargs'. For example, the following command searches only C files:

find /home/gigi -name '\*.c' -print | xargs grep 'hello' /dev/null

**3. What if a pattern has a leading `-'?**

grep -e '--cut here--' \*

searches for all lines matching `--cut here--'. Without `-e', `grep' would attempt to parse `--cut here--' as a list of options.

**4. Suppose I want to search for a whole word, not a part of a word?**

grep -w 'hello' \*

searches only for instances of `hello' that are entire words; it does not match `Othello'. For more control, use `\<' and `\>' to match the start and end of words. For example:

grep 'hello\>' \*

searches only for words ending in `hello', so it matches the word `Othello'.

**5. How do I output context around the matching lines?**

grep -C 2 'hello' \*

prints two lines of context around each matching line.

**6. How do I force grep to print the name of the file?**

Append `/dev/null':

grep 'eli' /etc/passwd /dev/null

**7. Why do people use strange regular expressions on `ps' output?**

ps -ef | grep '[c]ron'

If the pattern had been written without the square brackets, it

would have matched not only the `ps' output line for `cron', but

also the `ps' output line for `grep'.

**8. Why does `grep' report "Binary file matches"?**

If `grep' listed all matching "lines" from a binary file, it would

probably generate output that is not useful, and it might even

muck up your display. So GNU `grep' suppresses output from files

that appear to be binary files. To force GNU `grep' to output

lines even from files that appear to be binary, use the `-a' or

`--text' option.

**9. Why doesn't `grep -lv' print nonmatching file names?**

`grep -lv' lists the names of all files containing one or more

lines that do not match. To list the names of all files that

contain no matching lines, use the `-L' or `--files-without-match'

option.

**10. I can do OR with `|', but what about AND?**

grep 'paul' /etc/motd | grep 'franc,ois'

finds all lines that contain both `paul' and `franc,ois'.

**11. How can I search in both standard input and in files?**

Use the special file name `-':

cat /etc/passwd | grep 'alain' - /etc/motd

**tr**

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

The **tr**utilityis 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 uppercase characters to lowercase, at the command prompt type cat city | tr a-z A-Z and press the **Enter** key.

|  |  |
| --- | --- |
| **Command** | **Usage** |
| $ tr abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ | Convert lower case to upper case |
| $ tr '{}' '()' < inputfile > outputfile | Translate braces into parenthesis |
| $ echo "This is for testing" | tr [:space:] '\t' | Translate white-space to tabs |
| $ echo "This   is   for    testing" | tr -s [:space:] | Squeeze repetition of characters using -s |
| $ echo "the geek stuff" | tr -d 't' | Delete specified characters using -d option |
| $ echo "my username is 432234" | tr -cd [:digit:] | Complement the sets using -c option |
| $ tr -cd [:print:] < file.txt | Remove all non-printable character from a file |
| $ tr -s '\n' ' ' < file.txt | Join all the lines in a file into a single line |

**tee**

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

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

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

**wc**

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

By default all three of these options are active.

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

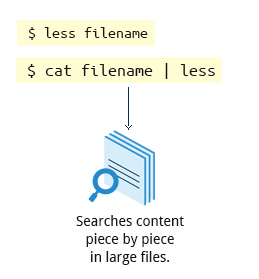
|  |  |
| --- | --- |
| **Option** | **Description** |
| –l | display the number of lines. |
| -c | display the number of characters. |
| -w | display the number of words. |

**cut**

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

For example, to display the third column delimited by a blank space, at the command prompt type ls -l | cut -d" " -f3 and press the **Enter** key.

**Note: The next two screens cover the Try-It-Yourself activities through which you can practice the procedures.**

**Working with Large Files**

System administrators need to work with configuration files, text files, documentation files, and log files. Some of these file may be large or become quite large as they accumulate data with time. These files will require both viewing and administrative updating. In this section, you will learn how to manage such large files.

For example, a banking system might maintain one simple large log file to record details of all of one day's ATM transactions. Due to a security attack or a malfunction, the administrator might be forced to check for some data by navigating within the file. In such cases, directly opening the file in an editor will cause issues, due to high memory utilization, as an editor will usually try to read the whole file into memory first. However, one can use **less** to view the contents of such a large file, scrolling up and down page by page without the system having to place the entire file in memory before starting. This is much faster than using a text editor.

Viewing the file can be done by typing either of the two following commands:

$ less <filename>  
$ cat <filename> | less

By default, manual (i.e., the **man** command) pages are sent through the **less** command.

**head**

**head**reads the first few lines of each named file (10 by default) and displays it on standard output. You can give a different number of lines in an option.

For example, If you want to print the first 5 lines from atmtrans.txt, use the following command:

$ head –n 5 atmtrans.txt

(You can also just say head -5 atmtrans.txt.)

**tail**

**tail**prints the last few lines of each named file (10 by default) and displays it on standard output. By default, it displays the last 10 lines. You can give a different number of lines as an option. **tail** is especially useful when you are troubleshooting any issue using log files as you probably want to see the most recent lines of output.

For example, to display the last 15 lines of atmtrans.txt, use the following command:

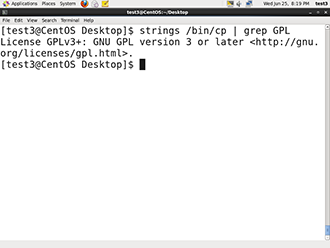
$ tail -n 15 atmtrans.txt

(You can also just say tail -15 atmtrans.txt.) To continually monitor new output in a growing log file:

$ tail -f atmtrans.txt

This command will continuously display any new lines of output in **atmtrans.txt** as soon as they appear. Thus it enables you to monitor any current activity that is being reported and recorded.

**strings**

[](https://courses.edx.org/c4x/LinuxFoundationX/LFS101x/asset/LFS01_ch12_screen_53a.jpg)**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

**The z Command Family**

When working with compressed files many standard commands can not 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 <filename>.gz or $ zmore <filename>.gz | To page through a compressed file |
| $ zgrep -i less test-file.txt.gz | To search inside a compressed file |
| $ zdiff filename1.txt.gz filename2.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**; i.e, we have **bzcat** and **bzless** associated with **bzip2,**and **xzcat** and **xzless**associated with **xz**.