1.1 Introduction

The definition of the word *Linux* depends on the context in which it is used. Linux means the *kernel* of the system, which is the central controller of everything that happens on the computer (more on this later). People that say their computer "runs Linux" usually refer to the kernel and suite of tools that come with it (called the *distribution*). If someone says they have "Linux experience", they are most likely talking about the programs themselves. However, they might also be talking about knowing how to add and partition a new disk or even fine-tune the kernel. Each of these components will be investigated so that you understand exactly what roles each plays.

What about *UNIX*? UNIX was originally an operating system developed at AT&T Bell Labs in the 1970's. It was modified and *forked* (that is, people modified it and those modifications served as the basis for other systems) such that now there are many different variants of UNIX. However, UNIX is now both a trademark and a specification, owned by an industry consortium called the Open Group. Only software that has been certified by the Open Group may call itself UNIX. Despite adopting all the requirements of the UNIX specification, Linux has not been certified, so Linux really isn't UNIX! It's just… UNIX-like.

1.1.1 Role of the Kernel

The three main components of an operating system are the kernel, shell and filesystem. The kernel of the operating system is like an air traffic controller at an airport. The kernel dictates which program gets which pieces of memory, it starts and kills programs, and it handles displaying text on a monitor. When an application needs to write to disk, it must ask the operating system to complete the write operation. If two applications ask for the same resource, the kernel decides who gets it, and in some cases, kills off one of the applications in order to save the rest of the system.

The kernel also handles switching of applications. A computer will have a small number of CPUs and a finite amount of memory. The kernel takes care of unloading one task and loading a new task if there are more tasks than CPUs. When the current task has run a sufficient amount of time, the CPU pauses the task so that another may run. This is called *pre-emptive multitasking*. Multitasking means that the computer is doing several tasks at once, and pre-emptive means that the kernel is deciding when to switch focus between tasks. With the tasks rapidly switching, it appears that the computer is doing many things at once.

Each application may think it has a large block of memory on the system, but it is the kernel that maintains this illusion, remapping smaller blocks of memory, sharing blocks of memory with other applications, or even swapping out blocks that haven't been touched to disk.

When the computer starts up, it loads a small piece of code called a *bootloader*. The bootloader's job is to load the kernel and get it started. If you are more familiar with operating systems such as Microsoft Windows or Apple's OS X, you probably never see the bootloader, but in the UNIX world it's usually visible so that you can tweak the way your computer boots.

The bootloader loads the Linux kernel and then transfers control. Linux then continues with running the programs necessary to make the computer useful, such as connecting to the network or starting a web server.

1.1.2 Applications

Like an air traffic controller, the kernel is not useful without something to control. If the kernel is the tower, the applications are the airplanes. Applications make requests to the kernel and receive resources, such as memory, CPU, and disk, in return. The kernel also abstracts the complicated details away from the application. The application doesn't know if a block of disk is on a solid-state drive from manufacturer A, a spinning metal hard drive from manufacturer B, or even a network file share. Applications just follow the kernel's *Application Programming Interface (API)* and in return don't have to worry about the implementation details.

When we, as users, think of applications, we tend to think of word processors, web browsers, and email clients. The kernel doesn't care if it is running something that's user facing, a network service that talks to a remote computer, or an internal task. So, from this we get an abstraction called a *process*. A process is just one task that is loaded and tracked by the kernel. An application may even need multiple processes to function, so the kernel takes care of running the processes, starting and stopping them as requested, and handing out system resources.

1.1.3 Role of Open Source

Linux started out in 1991 as a hobby project by Linus Torvalds. He made the source freely available and others joined in to shape this fledgling operating system. His was not the first system to be developed by a group, but since it was a built-from-scratch project, early adopters had the ability to influence the project's direction and to make sure mistakes from other UNIXes were not repeated.

Software projects take the form of *source code*, which is a human readable set of computer instructions. The source code may be written in any of hundreds of different programming languages, Linux just happens to be written in C, which is a language that shares history with the original UNIX.

Source code is not understood directly by the computer, so it must be compiled into machine instructions by a *compiler*. The compiler gathers all of the source files and generates something that can be run on the computer, such as the Linux kernel.

Historically, most software has been issued under a *closed-source license*, meaning that you get the right to use the machine code, but cannot see the source code. Often the license specifically says that you will not attempt to reverse engineer the machine code back to source code to figure out what it does!

Open source takes a source-centric view of software. The open source philosophy is that you have a right to obtain the software, and to modify it for your own use. Linux adopted this philosophy to great success. People took the source, made changes, and shared them back with the rest of the group.

Alongside this, was the *GNU project* (GNU's, not UNIX). While GNU (pronounced "ga-noo") was building their own operating system, they were far more effective at building the tools that go along with a UNIX operating system, such as the compilers and user interfaces. The source was all freely available, so Linux was able to target their tools and provide a complete system. As such, most of the tools that are part of the Linux system come from these GNU tools.

There are many different variants on open source, and those will be examined in a later chapter. All agree that you should have access to the source code, but they differ in how you can, or in some cases, must, redistribute changes.

1.1.4 Linux Distributions

Take Linux and the GNU tools, add some more user facing applications like an email client, and you have a full Linux system. People started bundling all this software into a *distribution* almost as soon as Linux became usable. The distribution takes care of setting up the storage, installing the kernel, and installing the rest of the software. The full featured distributions also include tools to manage the system and a *package manager* to help you add and remove software after the installation is complete.

Like UNIX, there are many different flavors of distributions. These days, there are distributions that focus on running servers, desktops, or even industry specific tools like electronics design or statistical computing. The major players in the market can be traced back to either **Red Hat** or **Debian**. The most visible difference is the software package manager, though you will find other differences on everything from file locations to political philosophies.

Red Hat started out as a simple distribution that introduced the Red Hat Package Manager (RPM). The developer eventually formed a company around it, which tried to commercialize a Linux desktop for business. Over time, Red Hat started to focus more on the server applications such as web and file serving, and released Red Hat Enterprise Linux, which was a paid service on a long *release cycle*. The release cycle dictates how often software is upgraded. A business may value stability and want long release cycles, a hobbyist or a startup may want the latest software and opt for a shorter release cycle. To satisfy the latter group, Red Hat sponsors the **Fedora Project** which makes a personal desktop comprising the latest software, but still built on the same foundations as the enterprise version.

Because everything in Red Hat Enterprise Linux is open source, a project called **CentOS** came to be, that recompiled all the RHEL packages and gave them away for free. CentOS and others like it (such as **Scientific Linux**) are largely compatible with RHEL and integrate some newer software, but do not offer the paid support that Red Hat does.

Debian is more of a community effort, and as such, also promotes the use of open source software and adherence to standards. Debian came up with its own package management system based on the .deb file format. While Red Hat leaves non Intel and AMD platform support to derivative projects, Debian supports many of these platforms directly.

Ubuntu is the most popular Debian derived distribution. It is the creation of **Canonical**, a company that was made to further the growth of Ubuntu and make money by providing support.

1.2 Hardware Platforms

Linux started out as something that would only run on a computer like Linus': a 386 with a specific hard drive controller. The range of support grew, as people built support for other hardware. Eventually, Linux started supporting other chips, including hardware that was made to run competitive operating systems!

The types of hardware grew from the humble Intel chip up to supercomputers. Later, smaller-size, Linux supported, chips were developed to fit in consumer devices, called embedded devices. The support for Linux became ubiquitous such that it is often easier to build hardware to support Linux and then use Linux as a springboard for your custom software, than it is to build the custom hardware and software from scratch.

Eventually, cellular phones and tablets started running Linux. A company, later bought by Google, came up with the Android platform which is a bundle of Linux and the software necessary to run a phone or tablet. This means that the effort to get a phone to market is significantly less. Instead of long development on a new operating system,

companies can spend their time innovating on the user facing software. Android is now one of the market leaders in the phone and tablet space.

Aside from phones and tablets, Linux can be found in many consumer devices. Wireless routers often run Linux because it has a rich set of network features. The TiVo is a consumer digital video recorder built on Linux. Even though these devices have Linux at the core, the end users don't have to know. The custom software interacts with the user and Linux provides the stable platform.

1.3 Shell

An operating system provides at least one *shell* or interface; this allows you to tell the computer what to do. A shell is sometimes called an *interpreter* because it takes the commands that a user issues and interprets them into a form that the *kernel* can then execute on the hardware of the computer. The two most common types of shells are the Graphical User Interface (GUI) and Command Line Interface (CLI).

Windows® typically use a GUI shell, primarily using the mouse to indicate what you want done. While using an operating system in this way might be considered easy, there are many advantages to using a CLI, including:

Command Repetition: In a GUI shell, there is no easy way to repeat a previous command. In a CLI there is an easy way to repeat (and also modify) a previous command.

Command Flexibility: The GUI shell provides limited flexibility in the way the command executes. In a CLI, *options* are specified with commands to provide a much more flexible and powerful interface.

Resources: A GUI shell typically uses a vast amount of resources (RAM, CPU, etc.). This is because a great deal of processing power and memory is needed to display graphics. By contrast, a CLI uses very little system resources, allowing more of these resources to be available to other programs.

Scripting: In a GUI shell, completing multiple tasks often requires multiple mouse clicks. With a CLI, a *script* can be created to execute many complex operations by typing just a single "command": the name of the script. A script is a series of commands placed into a single file. When executed, the script runs all of the commands in the file.

Remote Access: While it is possible to remotely execute commands in a GUI shell, this feature isn't typically set up by default. With a CLI shell, gaining access to a remote machine is easy and typically available by default.

Development: Normally a GUI-based program takes more time for the developers to create when compared to CLI-based programs. As a result, there are typically thousands of CLI programs on a typical Linux OS while only a couple hundred programs in a primarily GUI-based OS like Microsoft Windows®. More programs means more power and flexibility.

The Microsoft Windows® Operating System was designed to primarily use the GUI interface because of its simplicity, although there are several CLI interfaces available, too. For simple commands, there is the Run dialog box, where you can type or browse to the commands that you want to execute. If you want to type multiple commands or if you want to see the output of the command, you can use the Command Prompt, also called the DOS shell. Recently, Microsoft realized how important it is to have a powerful command line environment and, as a result, has introduced the Powershell.

Like Windows™, Linux also has both a CLI and GUI. Unlike Windows™, Linux lets you easily change the GUI shell (also called the desktop environment) that you want to use.

The two most common desktop environments for Linux are GNOME and KDE, however there are many other GUI shells available.

To access the CLI from within the GUI on a Linux operating system, the user can open a software program called a *terminal*. Linux can also be configured to only run the CLI without the GUI; this is typically done on servers that don't require a GUI, primarily to free up system resources.

1.4 Bash Shell

Not only does the Linux operating system provide multiple GUI shells, multiple CLI shells are also available. Normally, these shells are derived from one of two older UNIX shells: The Bourne Shell and the C Shell. In fact, the bash shell derives its name from the Bourne Shell: **B**ourne **A**gain **SH**ell. In this course, you will focus upon learning how to use the CLI for Linux with the bash shell, arguably the most popular CLI in Linux.
The bash shell has numerous built-in commands and features that you will learn including:

- **Aliases**: Give a command a different or shorter name to make working with the shell more efficient.
- **Re-Executing Commands**: To save retyping long command lines.
- **Wildcard Matching**: Uses special characters like ?, *, and [] to select one or more files as a group for processing.
- **Input/Output Redirection**: Uses special characters for redirecting input, <or <<, and output, >.
- **Pipes**: Used to connect one or more simple commands to perform more complex operations.
- **Background Processing**: Enables programs and commands to run in the background while the user continues to interact with the shell to complete other tasks. For example:

```
sysadmin@localhost:~/test$ sort red.txt &
```

```
[1] 108
```

The shell that your user account uses by default is set at the time your user account was created. By default, many Linux distributions use bash for a new user's shell. An administrator can use the usermod command to specify a different default shell after the account has been created.

As a user, you can use the chsh command to change your default shell.

The location where the system stores the default shell for user accounts is the /etc/passwd file.

Note: The usermod and chsh commands, as well as the /etc/passwdfile will be discussed in greater detail later in this course.

Typically, a user learns one shell and sticks with that shell, however after you have learned the basics of Linux, you may want to explore the features of other shells.

1.5 Accessing the Shell

How you access the command line shell depends on whether your system provides a GUI login or CLI login:

- GUI-based systems: If the system is configured to present a GUI, then you will need to find a software application called a *Terminal*. In the GNOME desktop environment, the Terminal application can be started by clicking the Applications menu, then the System Tools menu and Terminal icon.
- CLI-based systems: Many Linux systems, especially servers, are not configured to provide a GUI by default, so instead they present a CLI. If the system is configured to present a CLI, then the system runs a terminal application automatically for you after you login.

In the early days of computing, terminal devices were large machines that allowed users to provide input through a keyboard and displayed output by printing on paper. Over time, terminals evolved and their size shrank down into something that looked similar to a desktop computer with a video display monitor for output and a keyboard for input.

Ultimately, with the introduction of personal computers, terminals became *software emulators* of the actual hardware. Whatever you type in the terminal is interpreted by your shell and translated into a form that can then be executed by the kernel of the operating system.

If you are in a remote location, then *pseudo-terminal* connections can also be made across the network using several techniques. Insecure connections could be made using protocols such as `telnet` and programs such as `rlogin`, while secure connections can be established using programs like `putty` and protocols such as `ssh`.

1.6 Filesystems

In addition to the kernel and the shell, the other major component of any operating system is the filesystem. To the user, a filesystem is a hierarchy of directories and files with the root `/` directory at the top of the directory tree. To the operating system, a filesystem is a structure created on a disk partition consisting of tables defining the locations of directories and files. In this course, you will learn about the different Linux filesystems, filesystem benefits and how to create and manage filesystems using commands like `fsck`, `mount` and other disk and filesystem management commands.

1.7 What is a Command?

The simplest answer to the question, "What is a command?", is that a command is a software program that when executed on the command line, performs an action on the computer.

When you consider a command using this definition, you are really considering what happens when you execute a command. When you type in a command, a process is run by the operating system that can read input, manipulate data and produce output. From this perspective, a command runs a process on the operating system, which then causes the computer to perform a *job*.

However, there is another way of looking at what a command is: look at its *source*. The source is where the command "comes from" and there are several different sources of commands within the shell of your CLI:

- **Commands built-in to the shell itself**: A good example is the `cd` command as it is part of the `bash` shell. When a user types the `cd` command, the `bash` shell is already executing and knows how to interpret that command, requiring no additional programs to be started.
- **Commands that are stored in files that are searched by the shell**: If you type a `ls` command, then the shell searches through the directories that are listed in the `PATH` variable to try to find a file named `ls` that it can

execute. These commands can also be executed by typing the complete path to the command.

- **Aliases**: An alias can override a built-in command, function, or a command that is found in a file. Aliases can be useful for creating new commands built from existing functions and commands.
- **Functions**: Functions can also be built using existing commands to either create new commands, override commands built-in to the shell or commands stored in files. Aliases and functions are normally loaded from the initialization files when the shell first starts, discussed later in this section.

Consider This

While aliases will be covered in detail in a later section, this brief example may be helpful in understanding the concept of commands.

An alias is essentially a nickname for another command or series of commands. For example, the `cal 2014` command will display the calendar for the year 2014. Suppose you end up running this command often. Instead of executing the full command each time, you can create an alias called `mycal` and run the alias, as demonstrated in the following graphic:

```
sysadmin@localhost:~$ alias mycal="cal 2014"
sysadmin@localhost:~$ mycal
                              2014
      January               February              March
Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa
          1  2  3  4                  1                  1
 5  6  7  8  9 10 11   2  3  4  5  6  7  8   2  3  4  5  6  7  8
12 13 14 15 16 17 18   9 10 11 12 13 14 15   9 10 11 12 13 14 15
19 20 21 22 23 24 25  16 17 18 19 20 21 22  16 17 18 19 20 21 22
26 27 28 29 30 31     23 24 25 26 27 28     23 24 25 26 27 28 29
                                            30 31


      April                 May                   June
Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa
          1  2  3  4  5            1  2  3   1  2  3  4  5  6  7
 6  7  8  9 10 11 12   4  5  6  7  8  9 10   8  9 10 11 12 13 14
13 14 15 16 17 18 19  11 12 13 14 15 16 17  15 16 17 18 19 20 21
20 21 22 23 24 25 26  18 19 20 21 22 23 24  22 23 24 25 26 27 28
27 28 29 30           25 26 27 28 29 30 31  29 30


       July                August              September
Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa
          1  2  3  4  5            1  2     1  2  3  4  5  6
 6  7  8  9 10 11 12   3  4  5  6  7  8  9   7  8  9 10 11 12 13
```

```
13 14 15 16 17 18 19  10 11 12 13 14 15 16  14 15 16 17 18 19 20
20 21 22 23 24 25 26  17 18 19 20 21 22 23  21 22 23 24 25 26 27
27 28 29 30 31        24 25 26 27 28 29 30  28 29 30
                      31


    October           November          December
Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa
       1  2  3  4              1     1  2  3  4  5  6
 5  6  7  8  9 10 11   2  3  4  5  6  7  8   7  8  9 10 11 12 13
12 13 14 15 16 17 18   9 10 11 12 13 14 15  14 15 16 17 18 19 20
19 20 21 22 23 24 25  16 17 18 19 20 21 22  21 22 23 24 25 26 27
26 27 28 29 30 31     23 24 25 26 27 28 29  28 29 30 31
                      30
```

1.8 Commands That Are Stored In Files

Commands that are stored in files can be in several forms that you should be aware of. Most commands are written in the C programming language, which is initially stored in a human-readable text file. These text source files are then *compiled* into computer-readable binary files, which are then distributed as the command files.

Users who are interested in seeing the *source code* of compiled, GPL licensed software can find it through the sites where it originated, such as kernel.org. GPL licensed code also compels distributors of the compiled binaries, such as RedHat and Debian, to make the source code available. Often it is found in the distributors' repositories.

Consider This

Although it is not part of the Linux Essentials exam, it is possible to view available software packages at the command line. Type the following command into the terminal to view the available source packages for the GNU Compiler Collection:

```
sysadmin@localhost:~$ apt-cache search source | grep gcc
gcc-4.8-source - Source of the GNU Compiler Collection
gcc-4.4-source - Source of the GNU Compiler Collection
gcc-4.6-source - Source of the GNU Compiler Collection
gcc-4.7-source - Source of the GNU Compiler Collection
```

Command files can also contain human-readable text in the form of *script files*. A script file is a collection of commands that is typically executed at the command line.

The ability to create your own script files is a very powerful feature of the CLI. If you have a series of commands that you regularly find yourself typing in order to accomplish some task, then you can easily create a bash shell script to perform these multiple commands by typing just one command: the name of your script file. You simply need to place these commands into a file and make the file *executable* (more details on this will be provided in a later unit).

Summary of Key Terms

Command: Something that a user types in a CLI that will result in an action taking place on the system.

Compiled: The result of converting human-readable text code into system-readable binary code.

Source Code: The original human-readable text code.

Script File: A text file that contains commands and has been made executable.

1.9 Basic Command Syntax

Most commands follow a simple pattern of syntax:

```
command [options...] [arguments...]
```

In other words, you type a command, followed by one or more options (which are not always required) and one or more arguments before you press the **Enter** key. Although there are some commands in Linux that aren't entirely consistent with this syntax, most commands use this syntax.

When typing a command that is to be executed, the first step is to type the name of the command. The name of the command is often based on what it does or what the developer who created the command thinks will best describe the command's function.

For example, the `ls` command displays a *listing* of information about files. Associating the name of the command with something mnemonic for what it does may help you to remember commands more easily.

You must remember that every part of the command is normally case-sensitive, so `LS` is incorrect and will fail, but `ls` is correct and will succeed.

In the following example, the `ls` command is executed without any options or arguments, which results in the current directory contents being displayed. Many commands, like the `ls` command, can run successfully without any options or arguments, but be aware that there are commands that require you to type more than just the command alone.

```
sysadmin@localhost:~$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos  test
```

1.10 Specifying Options

If it is necessary for you to add options, they can be specified after the command name. *Short options* are specified with a hyphen - followed by a single character. Short options are how options were traditionally specified.

Often the character is chosen to be mnemonic for its purpose, like choosing the letter "a" for "all".

Multiple single options can be either given as separate options like `-a -l -r` or combined like `-alr`.

In the following example, the `-l` option is provided to the `ls` command, which results in a "long display" output:

```
sysadmin@localhost:~$ ls -l
total 0
drwxr-xr-x 1 sysadmin sysadmin   0 Sep 18 22:25 Desktop
```

```
drwxr-xr-x 1 sysadmin sysadmin   0 Sep 18 22:25 Documents
drwxr-xr-x 1 sysadmin sysadmin   0 Sep 18 22:25 Downloads
drwxr-xr-x 1 sysadmin sysadmin   0 Sep 18 22:25 Music
drwxr-xr-x 1 sysadmin sysadmin   0 Sep 18 22:25 Pictures
drwxr-xr-x 1 sysadmin sysadmin   0 Sep 18 22:25 Public
drwxr-xr-x 1 sysadmin sysadmin   0 Sep 18 22:25 Templates
drwxr-xr-x 1 sysadmin sysadmin   0 Sep 18 22:25 Videos
drwxr-xr-x 1 sysadmin sysadmin 420 Sep 18 22:25 test
```

Note: More details will be provided in a later section regarding the function of the `ls` command. For now, it is just being used to demonstrate how to execute commands on the command line.

You can type the name of a command with multiple short options. The output of all of these examples is the same, `-l` will give a long listing, while `-r` reverses the display order of the results:

- `ls -l -r`
- `ls -rl`
- `ls -lr`

```
sysadmin@localhost:~$ ls -l -r
total 0
drwxr-xr-x 1 sysadmin sysadmin 420 Sep 18 22:25 test
drwxr-xr-x 1 sysadmin sysadmin   0 Sep 18 22:25 Videos
drwxr-xr-x 1 sysadmin sysadmin   0 Sep 18 22:25 Templates
drwxr-xr-x 1 sysadmin sysadmin   0 Sep 18 22:25 Public
drwxr-xr-x 1 sysadmin sysadmin   0 Sep 18 22:25 Pictures
drwxr-xr-x 1 sysadmin sysadmin   0 Sep 18 22:25 Music
drwxr-xr-x 1 sysadmin sysadmin   0 Sep 18 22:25 Downloads
drwxr-xr-x 1 sysadmin sysadmin   0 Sep 18 22:25 Documents
drwxr-xr-x 1 sysadmin sysadmin   0 Sep 18 22:25 Desktop
```

Generally, short options can be combined with other short options in any order. The exception to this is when an option requires an argument.

For example, the `-w` option to the `ls` command specifies the width of the output desired and therefore requires an argument. If combined with other options, the `-w` option can be specified last, followed by its argument and still be valid, as in `ls -rtw 40`, which specifies an output width of 40 characters. Otherwise, the `-w` option cannot be combined with other options, and must be given separately.

```
sysadmin@localhost:~$ ls -l -r
total 0
drwxr-xr-x 1 sysadmin sysadmin 420 Sep 18 22:25 test
drwxr-xr-x 1 sysadmin sysadmin   0 Sep 18 22:25 Videos
drwxr-xr-x 1 sysadmin sysadmin   0 Sep 18 22:25 Templates
```

```
drwxr-xr-x 1 sysadmin sysadmin   0 Sep 18 22:25 Public
drwxr-xr-x 1 sysadmin sysadmin   0 Sep 18 22:25 Pictures
drwxr-xr-x 1 sysadmin sysadmin   0 Sep 18 22:25 Music
drwxr-xr-x 1 sysadmin sysadmin   0 Sep 18 22:25 Downloads
drwxr-xr-x 1 sysadmin sysadmin   0 Sep 18 22:25 Documents
drwxr-xr-x 1 sysadmin sysadmin   0 Sep 18 22:25 Desktop
```

If using multiple options that require arguments, then don't combine them. For example, the `-T` option also requires an argument. In order to accommodate both arguments, each option is given separately:

```
sysadmin@localhost:~$ ls -w 40 -T 12
Desktop    Music    Templates
Documents  Pictures  Videos
Downloads  Public    test
```

Some commands support additional options that are longer than a single character. *Long options* for commands are preceded by a double hyphen --and the meaning of the option is typically the name of the option, like `--all`. For example:

```
sysadmin@localhost:~$ ls --all
.           .bashrc   .selected_editor  Downloads  Public    test
..          .cache    Desktop         Music      Templates
.bash_logout  .profile  Documents       Pictures   Videos
```

For commands that support both long and short options, execute the command using the long and short options concurrently:

```
sysadmin@localhost:~$ ls --all --reverse -t
.profile     Videos     Pictures   Documents       .bashrc  .
.bash_logout Templates  Music      Desktop         ..
test         Public     Downloads  .selected_editor  .cache
```

Commands that support long options will often also support arguments that may be specified with or without an equal symbol (the output of both commands is the same):

- `ls --sort time`
- `ls --sort=time`

```
sysadmin@localhost:~$ ls --sort=time
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos  test
```

A special option exists, the "lone" double hyphen `--`, which can be used to indicate the end of all options for the command. This can be useful in some circumstances where it is unclear whether some text that follows the options should be interpreted as an additional option or as an argument to the command.

For example, if the `touch` command tries to create a file called `--badname`:

```
sysadmin@localhost:~$ touch --badname
touch: unrecognized option '--badname'
Try 'touch --help' for more information.
```

The command tries to interpret `--badname` as an option instead of an argument. However if the lone double hyphen `--` is placed before the filename, indicating that there are no more options, then the the filename can successfully be interpreted as an argument:

```
sysadmin@localhost:~$ touch -- --badname
sysadmin@localhost:~$ls
--badname  Documents  Music    Public    Videos
Desktop    Downloads  Pictures  Templates  test
```

Note: The file name in the previous example is considered to be "bad" because putting hyphens in the beginning of file names, while allowed, can cause problems when trying to access the file.

Consider This

A third type of option exists for a select few commands. While the options used in the AT&T version of UNIX used a single hyphen and the GNU port of those commands used two hyphens, the Berkley Software Distribution (BSD) version of UNIX used options with no hyphen at all.

This "no hyphen" syntax is fairly rare in most Linux distributions. A couple of notable commands that support the BSD UNIX style options are the `ps` and `tar` commands; both of these commands also support the single and double hyphen style of options.

In the terminal below, there are two similar commands, the first command is executed with a traditional UNIX style option (with single hyphens) and the second command is executed with a BSD style option (no hyphens).

```
sysadmin@localhost:~$ ps -u sysadmin
  PID TTY          TIME CMD
   79 ?        00:00:00 bash
  122 ?        00:00:00 ps
sysadmin@localhost:~$ ps u
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
sysadmin    79  0.0  0.0  18176  3428 ?        S    20:23   0:00 -bash
sysadmin   120  0.0  0.0  15568  2040 ?        R+   21:26   0:00 ps u
```

1.11 Specifying Arguments

After the command and any of its options, many commands will accept one or more arguments. Commands that use arguments may require one or more of them. For example, the `touch` command shown on the previous page requires at least one argument to specify the filename to act upon.

The `ls` command, on the other hand, allows for the filename argument to be specified, but it was not required. Some commands like the `cp` command (copy file) and the `mv` command (move file) require at least two arguments, the source and the destination file.

Arguments that contain unusual characters like spaces or non-alphanumeric characters will usually need to be *quoted*, either by enclosing them within double quotes or single quotes. Double quotes will prevent the shell from interpreting *some* of these special characters; single quotes prevent the shell from interpreting *any* special characters.

In most cases, single quotes are "safer" and should probably be used whenever you have an argument that contains characters that aren't alphanumeric. Quotes and special characters will be covered in greater detail in a later section, but if you want an idea of how important they are, take a look at the example in the *Consider This* section.

Consider This

To understand the importance of quotes, consider a simple scenario in which you want to go to your home directory (which can be accomplished with the `cd` command) and execute the `echo` command to display the string "hello world!!" on the screen.
The `echo` command displays text to the terminal.

You might first try the `echo` command without any quotes, unfortunately without success:

```
sysadmin@localhost:~$ cd
sysadmin@localhost:~$ echo hello world!!
echo hello worldcd
hello worldcd
```

Using no quotes failed because the shell interprets the !! characters as special shell characters; in this case they mean "replace the !! with the last command that was executed". In this case, the last command was the `cd` command, so `cd` replaced !! and then the `echo` command displayed `hello worldcd` to the screen.

You may want to try the double quotes to see if they will block the interpretation (or *expansion*) of the !! characters. The double quotes block the expansion of some special characters, but not all of them. Unfortunately, double quotes do not block the expansion of !!:

```
sysadmin@localhost:~$ cd
sysadmin@localhost:~$ echo "hello world!!"
echo "hello worldcd"
hello worldcd
```

Using double quotes preserves the literal value of all characters that they enclose except the $ (dollar sign), ` (backquote), \ (backslash) and !(exclamation point).

When you enclose text within the ' (single quote) characters, then *all* characters have their literal meaning:

```
sysadmin@localhost:~$ cd
sysadmin@localhost:~$ echo 'hello world!!'
hello world!!
```

1.12 exec Command

One exception to the basic command syntax used is the `exec` command, which takes as an argument another command to execute. What is special about the commands that are executed with `exec` is that they replace the currently running shell.

A common use of the exec command is in what are known as *wrapper scripts*. If the purpose of a script is to simply configure and launch another program, then it is known as a wrapper script.

In a wrapper script the last line of the script often uses exec program (where program is the name of another program to execute) to start some other program. A script written this way avoids having a shell continue to run while the program that it launched is running, the result is that this technique saves resources (like RAM).

Although redirection of input and output to a script are discussed in another section, it should also be mentioned that exec can be used to cause redirection for one or more statements in a script.

1.13 uname Command

The uname command displays system information. This command will output Linux by default when it is executed without any options.

```
sysadmin@localhost:~$ uname
Linux
```

Options for the uname command are as follows:

Short Option	Long Option	Prints
-a	--all	All information
-s	--kernel-name	Kernel name
-n	--node-name	Network node name
-r	--kernel-release	Kernel release
-v	--kernel-version	Kernel version
-m	--machine	Machine hardware name
-p	--processor	Processor type or unknown
-i	--hardware-platform	Hardware platform or unknown
-o	--operating-system	Operating system
	--help	Help information

Short Option	Long Option	Prints
	--version	Version information

The uname command is useful for several reasons, including when you need to determine the name of the computer as well as the current version of the kernel that is being used.

1.14 pwd Command

One of the simplest commands available is the pwd command, which is mnemonic for "**p**rint **w**orking **d**irectory". When executed without any options, the pwd command will display the name of the directory where the user is currently located in the file system.

```
sysadmin@localhost:~$ pwd
/home/sysadmin
```

1.15 Command Completion

A useful tool of the bash shell is the ability to automatically complete commands and their arguments. Like many command line shells, bash offers command line completion, where you type a few characters of a command (or its filename argument) and then press the **Tab** key. The bash shell will complete the command (or its filename argument) automatically for you. For example, if you type ech and press **Tab**, then the shell will automatically complete the command echo for you.

There will be times when you type a character or two and press the **Tab** key, only to discover that bash does not automatically complete the command. This will happen when you haven't typed enough characters to match only one command. However, pressing the **Tab** key a second time in this situation will display the possible completions (possible commands) available.

A good example of this would be if you typed ca and pressed **Tab**, then nothing would be displayed. If you pressed **Tab** a second time, then the possible ways to complete a command starting with ca would be shown:

```
sysadmin@localhost:~$ ca
cal             capsh           cat             cautious-launcher
calendar        captoinfo       catchsegv
caller          case            catman
sysadmin@localhost:~$ ca
```

Another possibility may occur when you have typed too little to match a single command name uniquely. If there are more possible matches to what you've typed than can easily be displayed, then the system will ask you if you want to display all possibilities.

For example, if you just type c and press the **Tab** key twice, the system may provide you with a prompt like:

```
Display all 102 possibilities? (y or n)
```

You should probably respond with `n` in a situation like this and then continue to type more characters to achieve a more refined match.

A common mistake when typing commands is to misspell the command. Not only will you type commands faster, but you will type more accurately if you use command completion. Using the **Tab** key to automatically complete the command helps to ensure that the command is typed correctly.

Note that completion also works for arguments to commands when the arguments are file or directory names.

Chapter Objectives

Chapter 1: Using the Shell

This chapter will cover the following exam objectives:

103.1: Work on the command line

Weight: 4

Description: Candidates should be able to interact with shells and commands using the command line. The objective assumes the Bash shell.

Key Knowledge Areas:

- Use single shell commands and one line command sequences to perform basic tasks on the command line
  Section 1.7 | Section 1.9 | Section 1.10 | Section 1.11

KEY TERMS

Chapter 1: Using the Shell

bash

Bourne Again SHell - an sh-compatible command language interpreter that executes commands read from the standard input or from a file.
Section 1.3

echo

Echo the STRING(s) to standard output. Useful with scripts.
Section 1.11

ls

Command that will list information about files. The current directory is listed by default.
Section 1.9

pwd

Print the name of the current working directory.
Section 1.14

uname

Print certain system information such as kernel name, network node hostname, kernel release, kernel version, machine hardware name, processor type, hardware platform, and operating system, depending on options provided.
Section 1.13

2.1 Introduction

As previously mentioned, UNIX was the Operating System from which the Linux foundation was built. The developers of UNIX created help documents called man pages (man stands for manual).

Referring to the man page for a command will provide you with the basic idea behind the purpose of the command, as well as details regarding the options of the command and a description of its features.

2.2 Viewing man Pages

To view a man page for a command, execute man *command* in a terminal window. For example, the following graphic shows the partial man page for the cal command:

```
CAL(1)          BSD General Commands Manual          CAL(1)

NAME
     cal, ncal -- displays a calendar and the date of Easter

SYNOPSIS
     cal [-3hjy] [-A number] [-B number] [[month] year]
     cal [-3hj] [-A number] [-B number] -m month [year]
     ncal [-3bhjJpwySM] [-A number] [-B number] [-s country_code] [[month]
       year]
     ncal [-3bhJeoSM] [-A number] [-B number] [year]
     ncal [-CN] [-H yyyy-mm-dd] [-d yyyy-mm]

DESCRIPTION
     The cal utility displays a simple calendar in traditional format and ncal
     offers an alternative layout, more options and the date of Easter.  The
     new format is a little cramped but it makes a year fit on a 25x80 termi-
     nal.  If arguments are not specified, the current month is displayed.

     The options are as follows:

     -h      Turns off highlighting of today.
```

Consider This

To send a man page to the default printer, execute the command as follows:

```
man -t command | lp
```

2.3 Controlling the man Page Display

The man command uses a "pager" to display documents. Typically, this pager is the less command, but on some distributions it may be the more command. Both are very similar in how they perform and will be discussed in more detail in a later chapter.

If you want to view the various movement commands that are available, you can type the letter h while viewing a man page. This will display a help page. (Note: If you are working on a Linux distribution that uses the more command as a pager, your output will be different from the partial example shown here.):

```
          SUMMARY OF LESS COMMANDS


      Commands marked with * may be preceded by a number, N.
      Notes in parentheses indicate the behavior if N is given.
      A key preceded by a caret indicates the Ctrl key; thus ^K is ctrl-K.


  h  H              Display this help.
  q  :q  Q  :Q  ZZ     Exit.
 ---------------------------------------------------------------------------

                           MOVING

  e  ^E  j  ^N  CR  *  Forward  one line   (or N lines).
  y  ^Y  k  ^K  ^P  *  Backward one line   (or N lines).
  f  ^F  ^V  SPACE  *  Forward  one window (or N lines).
  b  ^B  ESC-v      *  Backward one window (or N lines).
  z                 *  Forward  one window (and set window to N).
  w                 *  Backward one window (and set window to N).
  ESC-SPACE         *  Forward  one window, but don't stop at end-of-file.
  d  ^D             *  Forward  one half-window (and set half-window to N).
  u  ^U             *  Backward one half-window (and set half-window to N).
  ESC-)  RightArrow *  Left  one half screen width (or N positions).
  ESC-(  LeftArrow  *  Right one half screen width (or N positions).
  F                    Forward forever; like "tail -f".
  r  ^R  ^L            Repaint screen.
  R                    Repaint screen, discarding buffered input.
        ---------------------------------------------------
        Default "window" is the screen height.
        Default "half-window" is half of the screen height.
 ---------------------------------------------------------------------------
```

If your distribution uses the less command, you might be a bit overwhelmed with the large number of "commands" that are available. The following table provides a summary of the more useful commands:

Command	Function
Return (or **Enter**)	Go down one line
Space	Go down one page
`/term`	Search for *term*
`n`	Find next search item
`1G`	Go to beginning of the page
`G`	Go to end of the page
`h`	Display help
`q`	Quit `man` page

2.4 man Page Sections

Each `man` page is broken into sections. Each section is designed to provide specific information about a command. While there are common sections that you will see in most man pages, some developers also create sections that you will only see in a specific man page.

The following table describes some of the more common sections that you will find in `man` pages:

Section name	Purpose
NAME	Provides the name of the command and a very brief description.
SYNOPSIS	A brief summary of the command or function's interface. A summary of how the command line syntax of the program looks.
DESCRIPTION	Provides a more detailed description of the command.
OPTIONS	Lists the options for the command as well as a description of how they are used. Often this information will be found in the `DESCRIPTION` section

Section name	Purpose
	and not in a separate OPTIONSsection.
FILES	Lists the files that are associated with the command as well as a description of how they are used. These files may be used to configure the command's more advanced features. Often this information will be found in the DESCRIPTIONsection and not in a separate FILES section.
AUTHOR	The name of the person who created the man page and (sometimes) how to contact the person.
REPORTING BUGS	Provides details on how to report problems with the command.
COPYRIGHT	Provides basic copyright information.
SEE ALSO	Provides you with an idea of where you can find additional information. This also will often include other commands that are related to this command.

2.5 man Pages Synopsis

The SYNOPSIS section of a man page can be difficult to understand, but it is a valuable resource since it provides a concise example of how to use the command. For example, consider an example SYNOPSIS for the calcommand:

```
SYNOPSIS
   cal [-3hjy] [-A number] [-B number] [[month] year]
```

The square brackets [ ] are used to indicate that this feature is not required to run the command. For example, [-3hjy] means you can use the options -3, -h, -j, -y, but none of these options are required for the cal command to function properly.

The second set of square brackets in the cal SYNOPSIS, [-A number]allows you to specify a number of months to be added to the end of the display.

The third set of square brackets in the cal SYNOPSIS, [-B number] allows you to specify a number of months to be added to the beginning of the display.

The fourth set of square brackets in the cal SYNOPSIS, [[month] year]demonstrates another feature; it means that you can specify a year by itself, but if you specify a month you must also specify a year.

Another component of the SYNOPSIS that might cause some confusion can be seen in the SYNOPSIS of the date command:

```
SYNOPSIS
```

```
date [OPTION]... [+FORMAT]
date [-u|--utc|--universal] [MMDDhhmm[[CC]YY][.ss]]
```

In this SYNOPSIS there are two syntaxes for the `date` command. The first one is used to display the date on the system while the second is used to set the date.

The ellipses following [OPTION]..., indicate that [OPTION] may be repeated.

Additionally the [-u|--utc|--universal] notation means that you can either use the `-u` option or the `--utc` option or the `--universal` option. Typically this means that all three options really do the same thing, but sometimes this format (use of the | character) is used to indicate that the options can't be used in combination, like a logical *or*.

2.6 Searching Within a man Page

In order to search a man page for a term, type `/` and type the *term* followed by the **Enter** key. The program will search from the current location down towards the bottom of the page to try to locate and highlight the term.

If the term is not found, or you have reached the end of the matches, then the program will report "Pattern not found (press Return)". If a match is found and you want to move to the next match of the term, press `n`. To return to a previous match of the term, press `N`.

```
   -s country_code
          Assume the switch from Julian to Gregorian Calendar at the date
          associated with the country_code.  If not specified, ncal tries
          to guess the switch date from the local environment or falls back
          to September 2, 1752.  This was when Great Britain and her
          colonies switched to the Gregorian Calendar.
/date
```

If you haven't already, notice here that commands to the pager are viewed on the final line of the screen.

2.7 man Page Sections

Until now, we have been displaying man pages for commands. However, sometimes configuration files also have man pages. Configuration files (sometimes called system files) contain information that is used to store information about the operating system or services.

Additionally, there are several different types of commands (user commands, system commands, and administration commands) as well as other features that require documentation, such as libraries and kernel components.

As a result, there are thousands of man pages on a typical Linux distribution. To organize all of these man pages, the pages are categorized by sections, much like each individual man page is broken into sections.

By default there are nine default sections of man pages:

- Executable programs or shell commands
- System calls (functions provided by the kernel)
- Library calls (functions within program libraries)
- Special files (usually found in /dev)

- File formats and conventions, e.g. /etc/passwd
- Games
- Miscellaneous (including macro packages and conventions), e.g. man(7), groff(7)
- System administration commands (usually only for root)
- Kernel routines [Non standard]

When you use the man command, it searches each of these sections in order until it finds the first "match". For example, if you execute the command man cal, the first section (Executable programs or shell commands) is searched for a man page called **cal**. If not found, then the second section is searched. If no man page is found after searching all sections, you will receive an error message:

```
sysadmin@localhost:~$ man zed
No manual entry for zed
```

Consider This

Typically, there are many more sections than the standard nine. Custom software designers make man pages, using non-standard section "names", like 3p.

2.8 Determining Which Section

To determine which section a specific man page belongs to, look at the numeric value on the first line of the output of the man page. For example, if you execute the command man cal, you will see that the cal command belongs to the first section of man pages:

```
CAL(1)          BSD General Commands Manual          CAL(1)
```

2.9 Specifying a Section

In some cases you will need to specify the section in order to display the correct man page. This is necessary because sometimes there will be man pages with the same name in different sections.

For example, there is a command called passwd that allows you to change your password. There is also a file called passwd that stores account information. Both the command and the file have a man page.

The passwd command is a "user" command, so the command man passwdwill display the man page for the passwd command by default:

```
PASSWD(1)          User Commands          PASSWD(1)


NAME
    passwd - change user password
```

To specify a different section, provide the number of the section as the first argument of the man command. For example, the command man 5 passwdwill look for the passwd man page in section 5 only:

```
PASSWD(5)       File Formats and Conversions       PASSWD(5)


NAME
    passwd - the password file
```

2.10 Searching by Name

Sometimes it isn't clear which section a man page is stored in. In cases like this, you can search for a man page by name.

The -f option to the man command will display man pages that match, or partially match, a specific name and provide a brief description of each manpage:

```
sysadmin@localhost:~$ man -f passwd
passwd (5)        - the password file
passwd (1)        - change user password
passwd (1ssl)     - compute password hashes
```

Note that on most Linux distributions, the whatis command does the same thing as man -f. On those distributions, both will produce the same output.

```
sysadmin@localhost:~$ whatis passwd
passwd (5)        - the password file
passwd (1)        - change user password
passwd (1ssl)     - compute password hashes
```

2.11 Searching by Keyword

Unfortunately, you won't always remember the exact name of the man page that you want to view. In these cases you can search for man pages that match a keyword by using the -k option to the man command.

For example, what if you knew you wanted a man page that displays how to change your password, but you didn't remember the exact name? You could run the command man -k password:

```
sysadmin@localhost:~$ man -k password
chage (1)         - change user password expiry information
chgpasswd (8)     - update group passwords in batch mode
chpasswd (8)      - update passwords in batch mode
cpgr (8)          - copy with locking the given file to the password or gr...
cppw (8)          - copy with locking the given file to the password or gr...
```

When you use this option, you may end up with a large amount of output. The preceding command, for example, provided over 60 results.

Recall that there are thousands of man pages, so when you search for a keyword, be as specific as possible. Using a generic word, such as "the" could result in hundreds or even thousands of results.

Note that on most Linux distributions, the apropos command does the same thing as man -k. On those distributions, both will produce the same output.

Consider This

Want to learn more about man pages? Suggestion: execute the command man man.

2.12 info Command

man pages are great sources of information, but they do tend to have a few disadvantages. One example of a disadvantage is that each man page is a separate document, not related to any

other man page. While some man pages have a SEE ALSO section that may refer to other man pages, they really tend to be unrelated sources of documentation.

The info command also provides documentation on operating system commands and features. The goal of this command is slightly different from man pages: to provide a documentation resource that provides a logical organizational structure, making reading documentation easier.

Within info documents, information is broken down into categories that work much like a table of contents that you would find in a book. Hyperlinks are provided to pages that contain information on individual topics for a specific command or feature. In fact, all of the documentation is merged into a single "book" in which you can go to the top level of documentation and view the table of contents representing all of the documentation available.

Another advantage of info over man pages is that the writing style of info documents is typically more conducive to learning a topic. Consider man pages to be more of a reference resource and info documents to be more of a learning guide.

2.13 Displaying info Documentation

To display the info documentation for a command, execute info *command* (replace *command* with the name of the command that you are seeking information about). For example, the following demonstrates the output of the command info ls:

```
File: coreutils.info,  Node: ls invocation,  Next: dir invocation,  Up: Directory listing

10.1 `ls': List directory contents
==================================

The `ls' program lists information about files (of any type, including
directories).  Options and file arguments can be intermixed
arbitrarily, as usual.

  For non-option command-line arguments that are directories, by
default `ls' lists the contents of directories, not recursively, and
omitting files with names beginning with `.'.  For other non-option
arguments, by default `ls' lists just the file name.  If no non-option
argument is specified, `ls' operates on the current directory, acting
as if it had been invoked with a single argument of `.'.

  By default, the output is sorted alphabetically, according to the
locale settings in effect.(1) If standard output is a terminal, the
output is in columns (sorted vertically) and control characters are
output as question marks; otherwise, the output is listed one per line
and control characters are output as-is.
--zz-Info: (coreutils.info.gz)ls invocation, 58 lines --Top----------------------
Welcome to Info version 5.2. Type h for help, m for menu item.
```

Notice that the first line provides some information that tells you where you are in the info documentation. This documentation is broken up into "nodes" and in the example above you are currently in the ls invocation node. If you went to the next node (like going to the next chapter in a book), you would be in the dir invocation node. If you went up one level you would be in the Directory listing node.

2.14 Moving Around While Viewing an info Document

Similar to the man command, you can view a listing of movement commands by typing the letter h while reading the info documentation:

```
Basic Info command keys


l        Close this help window.
q        Quit Info altogether.
H        Invoke the Info tutorial.

Up       Move up one line.
Down     Move down one line.
DEL      Scroll backward one screenful.
SPC      Scroll forward one screenful.
Home     Go to the beginning of this node.
End      Go to the end of this node.

TAB      Skip to the next hypertext link.
RET      Follow the hypertext link under the cursor.
l        Go back to the last node seen in this window.

[        Go to the previous node in the document.
]        Go to the next node in the document.
p        Go to the previous node on this level.
n        Go to the next node on this level.
u        Go up one level.
-----Info: *Info Help*, 466 lines --Top-----------------------------------------
```

Note that if you want to close the help screen, you type the letter l. This brings you back to your document and allows you to continue reading. To quit entirely, you type the letter q.

The following table provides a summary of useful commands:

Command	Function
Down Arrow	Go down one line

Command	Function
Space	Go down one page
s	Search for term
[	Go to previous node
]	Go to next node
u	Go up one level
Tab	Skip to next hyperlink
Home	Go to beginning
End	Go to end
h	Display help
L	Quit help page
q	Quit info command

If you scroll though the document, you will eventually see the menu for the `ls`command:

```
* Menu:


* Which files are listed::
* What information is listed::
* Sorting the output::
* Details about version sort::
* General output formatting::
* Formatting file timestamps::
* Formatting the file names::
```

The items under the menu are hyperlinks that can take you to nodes that describe more about the `ls` command. For example, if you placed your cursor on the line `* Sorting the output::` and pressed the **Enter** key, you would be taken to a node that describes sorting the output of the `ls` command:

```
File: coreutils.info,  Node: Sorting the output,  Next: Details about version sort,  Prev: What information is listed,  Up: ls invo
cation


10.1.3 Sorting the output
-------------------------


These options change the order in which `ls' sorts the information it
outputs.  By default, sorting is done by character code (e.g., ASCII
order).


`-c'
`--time=ctime'
`--time=status'
    If the long listing format (e.g., `-l', `-o') is being used, print
    the status change time (the `ctime' in the inode) instead of the
    modification time.  When explicitly sorting by time (`--sort=time'
    or `-t') or when not using a long listing format, sort according
    to the status change time.


`-f'
    Primarily, like `-U'--do not sort; list the files in whatever
    order they are stored in the directory.  But also enable `-a' (list
--zz-Info: (coreutils.info.gz)Sorting the output, 68 lines --Top----------
```

Note that by going into the node about sorting, you essentially went into a sub-node of the one in which you originally started. To go back to your previous node, you can use `u`. While `u` will take you to the start of the node one level up, you could also use `l` to return you exactly to the previous location that you were before entering the sorting node.

2.15 Exploring info Documentation

Instead of using `info` documentation to look up information about a specific command or feature, consider exploring the capabilities of Linux by reading through the `info` documentation. If you execute the `info` command without any arguments, you are taken to the top level of the documentation. From there, you can explore many features:

```
File: dir,      Node: Top      This is the top of the INFO tree


 This (the Directory node) gives a menu of major topics.
 Typing "q" exits, "?" lists all Info commands, "d" returns here,
 "h" gives a primer for first-timers,
 "mEmacs<Return>" visits the Emacs manual, etc.
```

```
 In Emacs, you can click mouse button 2 on a menu item or cross reference
 to select it.

* Menu:

Basics
* Common options: (coreutils)Common options.
* Coreutils: (coreutils).      Core GNU (file, text, shell) utilities.
* Date input formats: (coreutils)Date input formats.
* File permissions: (coreutils)File permissions.
                        Access modes.
* Finding files: (find).       Operating on files matching certain criteria.

C++ libraries
* autosprintf: (autosprintf).   Support for printf format strings in C++.
-----Info: (dir)Top, 211 lines --Top-----------------------------------------
Welcome to Info version 5.2. Type h for help, m for menu item.
```

2.16 Additional Sources of Help

In many cases, you will find that either man pages or info documentation will provide you with the answers you need. However, in some cases, you may need to look in other locations.

Using the --help Option

Many commands will provide you basic information, very similar to the SYNOPSIS found in man pages, when you apply the `--help` option to the command. This is useful to learn the basic usage of a command:

```
sysadmin@localhost:~$ head --help
Usage: head [OPTION]... [FILE]...
Print the first 10 lines of each FILE to standard output.
With more than one FILE, precede each with a header giving the file name.
With no FILE, or when FILE is -, read standard input.

Mandatory arguments to long options are mandatory for short options too.
 -c, --bytes=[-]K        print the first K bytes of each file;
                     with the leading '-', print all but the last
                     K bytes of each file
 -n, --lines=[-]K        print the first K lines instead of the first 10;
                     with the leading '-', print all but the last
                     K lines of each file
 -q, --quiet, --silent    never print headers giving file names
```

```
-v, --verbose          always print headers giving file names
    --help     display this help and exit
    --version  output version information and exit
```

Additional System Documentation

On most systems, there is a directory where additional documentation is found. This will often be a place where vendors who create additional (third party) software can store documentation files.

Typically, this will be a place where system administrators will go to learn how to set up more complex software services. However, sometimes regular users will also find this documentation to be useful.

These documentation files are often called "readme" files, since the files typically have names such as `README` or `readme.txt`. The location of these files can vary depending on the distribution that you are using. Typical locations include `/usr/share/doc` and `/usr/doc`.

13.1 Introduction

In order to access or modify a file or directory, the correct *permissions* must be set. There are three different permissions that can be placed on a file or directory: read, write, and execute.

The manner in which these permissions apply differs for files and directories, as shown in the chart below:

Permission	Effects on File	Effects on Directory
read (`r`)	Allows for file contents to be read or copied.	Without execute permission on the directory, allows for a non-detailed listing of files. With execute permission, `ls -l` can provide a detailed listing.
write (`w`)	Allows for contents to be modified or overwritten. Allows for files to be added or removed from a directory.	For this permission to work, the directory must also have execute permission.
execute (`x`)	Allows for a file to be run as a process, although script files require read permission, as well.	Allows a user to change to the directory if parent directories have write permission as well.

When listing a file with the `ls -l` command, the output includes permission information:

```
sysadmin@localhost:~$ ls -l /bin/ls
-rwxr-xr-x. 1 root root 118644 Apr  4  2013 /bin/ls
```

Below is a refresher on the fields relevant to permissions.

File Type Field

```
-rwxr-xr-x. 1 root root 118644 Apr  4  2013 /bin/ls
```

The first character of this output indicates the type of a file. Recall if the first character is a `-`, as in the previous example, this is a regular file. If the character was a `d` it would be a directory.

Consider This

If the first character of the output of the `ls -l` command is an `l`, then the file is a symbolic link. A symbolic link file "points to" another file in the file system.

To access a file that a symbolic link points to, you need to have the appropriate permissions on both the symbolic link file and the file that it points to.

Permissions Field

```
-rwxr-xr-x. 1 root root 118644 Apr  4  2013 /bin/ls
```

After the file type character, the permissions are displayed. The permissions are broken into three sets of three characters:

- `rwx`r-xr-x: The first set for the user who owns the file.
- rwx`r-xr-x`:The second set for the group that owns the file.
- rwxr-x`r-x`: The last set for everyone else, which means "anyone who is not the user that owns the file or a member of the group that owns the file".

User Owner Field

```
-rwxr-xr-x. 1 root root 118644 Apr  4  2013 /bin/ls
```

After the link count, the file's user owner is displayed.

Group Owner Field

```
-rwxr-xr-x. 1 root root 118644 Apr  4  2013 /bin/ls
```

After the user owner field, the file group owner is displayed.

13.2 Changing Ownership

Changing the User Owner

Initially, the owner of a file is the user who creates it. The user owner can only be changed by a user with root privileges using the `chown` command. By executing `chown <newuser> pathname`, the root user can change the owner of the file `<pathname>` to the `<newuser>` account.

Changing the Group Owner

When a user creates a file or directory, their primary group will normally be the group owner. The `id` command reveals the current user's identity, current primary group and all group memberships.

To create a file or directory that will be owned by a group different from your current primary group, one option is to change your current primary group to another group you belong to by using the `newgrp` command. For example, to change the current primary group from `student`, to a secondary group called `circles`, execute: `newgrp circles`. After executing that command, any new files or directories created would be group owned by the `circles` group.

The `newgrp` command opens a new shell and assigns the primary group for that shell to the specified group. To go back to the default primary group, use the `exit` command to close the new shell that was started by the `newgrp`command.

Note: You must be a member of the group that you want to change to, additionally, administrators can password protect groups.

Another option is for the user owner of the file to change the group owner by using the `chgrp` command. For example, if you forgot to change your primary group to `circles` before you created `myfile`, then you can execute the `chgrp circles myfile` command.

The `chgrp` command does not change your current primary group, so when creating many files, it is more efficient to just use the `newgrp` command instead of executing the `chgrp` command for each file.

Note: Only the owner of the file and the root user can change the group ownership of a file.

13.3 Understanding Permissions

User and group owner information are important considerations when determining what permissions will be effective for a particular user. To determine which set applies to a particular user, first examine the user's identity. By using the `whoami`command or the `id` command, you can display your user identity.

To understand which permissions will apply to you, consider the following:

- `rwx`r-xr-x: If your current account is the user owner of the file, then the first set of the three permissions will apply and the other permissions have no effect.
- rwx`r-x`r-x: If your current account *is not* the user owner of the file and you *are* a member of the group that owns the file, then the group permissions will apply and the other permissions have no effect.
- rwxr-x`r-x`: If you are not the user who owns the file or a member of the group that owns the file, the third set of permissions applies to you. This last set of permissions is known as the permissions for "others"; it is sometimes referred to as the "world permissions".

For example, use the `ls -l /etc` command in the virtual terminal and examine the permissions displayed.

Answer the following questions:

- Who owns the files in the /etc directory?
- Can members of the root group modify (write to) these files?
- Can users who are not root, nor members of the root group modify these files?

Consider This

Understanding which permissions apply is an important skill set in Linux. For example, consider the following output of the `ls -l` command:

```
-r--rw-rwx. 1 bob staff 999 Apr 10 2013 /home/bob/test
```

In this scenario, the user `bob` ends up having less access to this file than members of the `staff` group or everyone else. The user `bob` only has the permissions of `r--`. It doesn't matter if `bob` is a member of the `staff`group; once user ownership has been established, only the user owner's permissions apply.

13.4 Changing Basic File Permissions

The `chmod` command is used to change the permissions of a file or directory. Only the root user or the user who owns the file is able to change the permissions of a file.

Consider This

Why is the command called `chmod` instead of `chperm`? Permissions used to be referred to as "modes of access", so the command `chmod` really means "change the modes of access".

There are two techniques of changing permissions with the `chmod` command: symbolic and octal (also called the numeric method). The symbolic method is good for changing one set of permissions at a time. The octal or numeric method requires knowledge of the octal value of

each of the permissions and requires all three sets of permissions (user, group, other) to be specified every time. When changing more than one permission set, the octal method is probably the better method.

The Symbolic Method

To use the symbolic method of `chmod` you will use the following symbols to represent which set of permissions you are changing:

Symbol	Meaning
`u`	user: the user who owns the file
`g`	group: the group who owns the file
`o`	others: people other than the user owner or member of the group owner
`a`	all: to refer to the user, group and others

Use the above symbols for who you are specifying along with an action symbol:

Symbol	Meaning
`+`	Add the permission, if necessary
`=`	Specify the exact permission
`-`	Remove the permission, if necessary

After an action symbol, specify one or more permissions (`r`=read, `w`=write and `x`=execute), then a space and the pathnames for the files to assign those permissions. To specify permissions for more than one set, use commas (and no spaces) between each set.

Examples

Add execute permission for the user owner:

```
sysadmin@localhost:~$  chmod u+x myscript
```

Remove write permission from the group owner:

```
sysadmin@localhost:~$  chmod g-w file
```

Assign others to have only the read permission, removes write permission from the group owner, and adds execute permission for the user owner:

```
sysadmin@localhost:~$  chmod o=r,g-w,u+x myscript
```

Assign everyone no permission:

```
sysadmin@localhost:~$  chmod a=- file
```

In the following example, a file is created with the `touch` command and then its permissions are modified using the symbolic method:

```
sysadmin@localhost:~$  touch sample
sysadmin@localhost:~$  ls -l sample
-rw-rw-r-- 1 sysadmin sysadmin 0 Oct 17 18:05 sample
sysadmin@localhost:~$  chmod g-w,o-r sample
sysadmin@localhost:~$  ls -l sample
-rw-r----- 1 sysadmin sysadmin 0 Oct 17 18:05 sample
```

The Octal Method

Using the octal method requires that the permissions for all three sets be specified. To do so, add together the octal value for the read, write and execute permission for each set:

Permission	Octal Value
read (`r`)	`4`
write (`w`)	`2`
execute (`x`)	`1`

For example, set the permissions of a file to `rwxr-xr-x`. Using the values found in the previous table, this would calculate as `7` for the user owner's permission, `5` for the group owner's permissions and `5` for others:

These numbers are easy to derive by just adding together the octal value for the permissions shown. So, for read, write and execute, add `4 + 2 + 1` to get `7`. Or, for read, not write and execute, add `4 + 0 + 1` to get `5`.

Examples

Change permissions to `rwxrw-r--`:

```
sysadmin@localhost:~$  chmod 764 myscript
```

Change permissions to `rw-r--r--`:

```
sysadmin@localhost:~$  chmod 644 myfile
```

Change permissions to `rwxr--r--`:

```
sysadmin@localhost:~$  chmod 744 myscript
```

Change permissions to `---------`:

```
sysadmin@localhost:~$  chmod 000 myfile
```

Considering the last example, what can be done with a file if there are no permissions for anyone on the file? The owner of the file can always use the `chmod` command at some point in the future to grant permissions on the file. Also, with write and execute permission `-wx` on the directory that contains this file, a user can also remove it with the `rm` command.

In the following example, the permissions of the sample file that was created previously are modified using the octal method:

```
sysadmin@localhost:~$  ls -l sample
-rw-r----- 1 sysadmin sysadmin 0 Oct 17 18:05 sample
sysadmin@localhost:~$  chmod 754 sample
sysadmin@localhost:~$  ls -l sample
-rwxr-xr-- 1 sysadmin sysadmin 0 Oct 17 18:05 sample
```

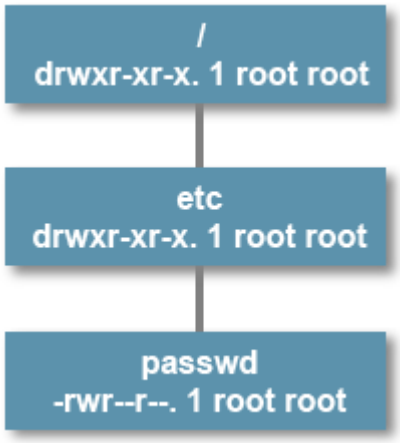
13.5 Permission Scenarios

Before discussing changing or setting permissions, a solid understanding of how permissions really work is required. Many beginning Linux users get hung up on the short descriptions of "read", "write" and "execute", but these words don't really define what a user can *do* with a file or directory.

To understand how permissions behave, several scenarios will be presented. While you review these scenarios, keep in mind the following table from a previous slide:

Permission	Effect of File	Effect on Directory
read (`r`)	Allows for file contents to be read or copied.	Without execute permission on the directory, allows for a non-detailed listing of files. With execute permission, `ls -l` can provide a detailed listing.
write (`w`)	Allows for contents to be modified or overwritten. Allows for files to be added or removed from a directory.	For this permission to work, the directory must also have execute permission.
execute (`x`)	Allows for a file to be run as a process, although script files require read permission, as well.	Allows a user to change to the directory if parent directories have execute permission as well

For each scenario below, a diagram is provided to describe the directory hierarchy. In this diagram, key permission information is provided for each file and directory. For example, consider the following:

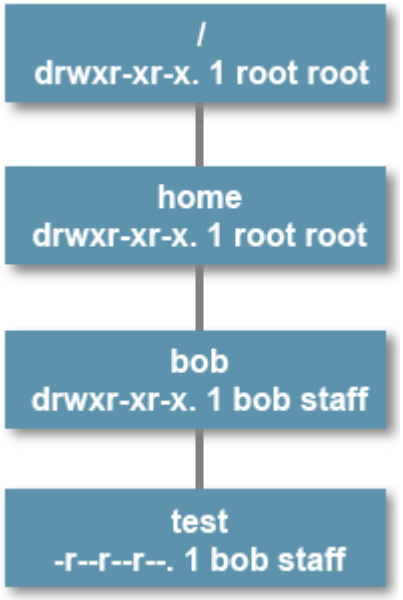


The first box describes the `/` directory. The permissions are `rwxr-xr-x`, the user owner is `root` and the group owner is `root`.

The second box describes the `etc` directory, which is a subdirectory under the `/` directory. The third box describes the `passwd` file, which is a file under the `etc` directory.

Scenario #1

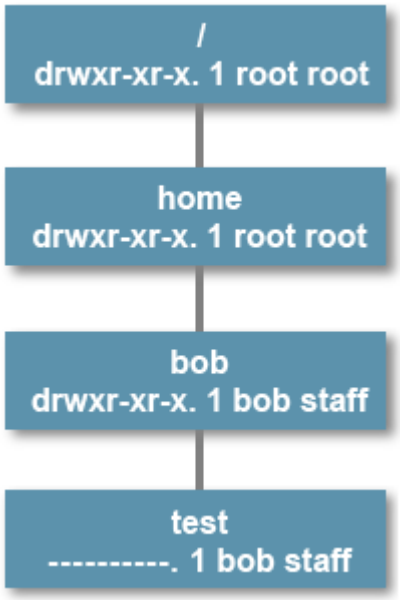
Question: Given the following diagram, what can the user `bob` do with the /home/bob/test file?



Answer: The user `bob` can view and copy the file because this user has the permissions of `r--` on the file.

Scenario #2

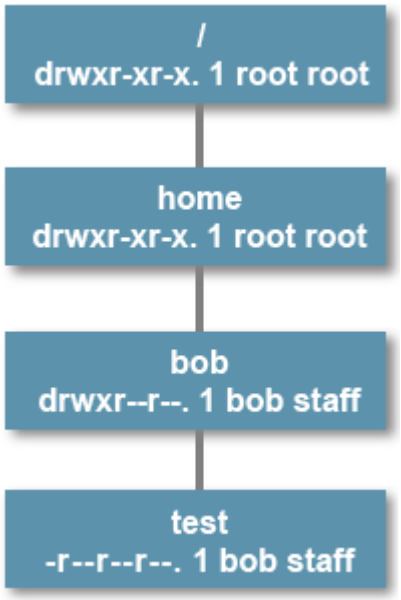
Question: Given the following diagram, can the user `bob` delete the `/home/bob/test` file?



Answer: Yes. While it may seem like the user `bob` shouldn't be able to delete this file because he has no permissions on the file itself, to delete a file a user needs to have write permission in the directory that the file is stored in. File permissions do not apply when deleting a file.

Scenario #3

Question: Given the following diagram, can the user `sue` view the contents of the `/home/bob/test` file?

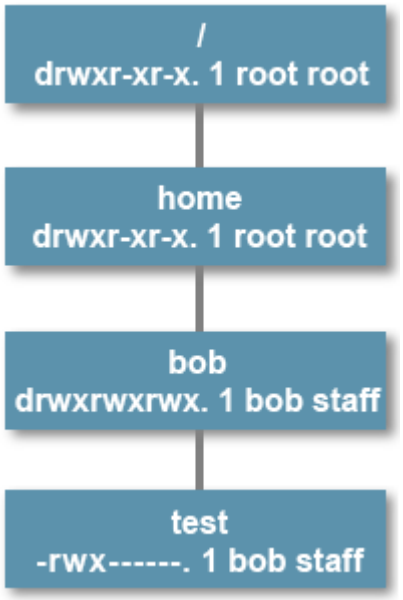


Answer: No. Initially it may seem that the user `sue` should be able to view the contents of the `/home/bob/test` file because all users have read permission on the file.

However, when checking permissions, it is important to check the permissions on the directories "above" the file in question. In this case, only the user `bob` has the ability to "get into" the `/home/bob` directory because only the user `bob` has execute permission on that directory. If you can't get into a directory, it doesn't matter what the permissions are on the files in that directory are set to.

Scenario #4

Question: Given the following diagram, who can delete the `/home/bob/test`file?



Answer: All users on the system can. Recall that to delete a file, a user needs write and execute permissions on the directory that the file is stored in. All users have write and execute permission on the `/home/bob` directory. This is a huge mistake because now all users can delete all files in bob's home directory.

13.6 Changing Advanced File Permissions

The following permissions are considered advanced because typically they are only set by the administrator (the root user) and they perform very specialized functions. They can be set using the `chmod` command, using either the symbolic or octal method.

Permission	Symbol	Octal Value	Purpose
setuid on a file	An `s` replaces the `x` for the user owner permissions Set with `u+s`	`4000`	Causes an executable file to execute under user owner identity, instead of the user running the command.
setgid on a file	An `s` replaces the `x` for the group owner permissions Set with `g+s`	`2000`	Causes an executable file to execute under group owner identity, instead of the user running the command.
setgid on a directory	An `s` replaces the `x` for the group owner permissions Set with `g+s`	`2000`	Causes new files and directories that are created inside to be owned by the group that owns the directory.
sticky on a directory	A `t` replaces the `x` for the others permissions Set with `o+t`	`1000`	Causes files inside directory to be able to be removed only by the user owner, or the root user.

Note: A leading `0`, such as `0755`, will remove all special permissions from a file or directory.

13.6.1 setuid Permission

The setuid permission is used on executable files to allow users who are not the root user to execute those files as if they were the root user.

For example, the `passwd` command has the setuid permission. The `passwd`command modifies the `/etc/shadow` file in order to update the value for the user's password. That file is not normally modifiable by an ordinary user; in fact, ordinary users normally have no permissions on the file.

Since the `/usr/bin/passwd` command is owned by the root user and has setuid permission, it executes with a "duel personality", which allows it to access files either as the person who is running the command or as the root user. When the `passwd` command attempts to update the `/etc/shadow` file, it uses the credentials of the root user to modify the file (the term *credentials* is akin to "authority").

The following demonstrates what this setuid executable file looks like when listed with `ls -l`:

```
sysadmin@localhost:~$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 47032 Feb 17  2014 /usr/bin/passwd
```

Notice from the listing, the fourth character is an `s`, where there would normally be an `x` if this file was just executable. When this character is a lowercase `s`, it indicates that both the setuid and execute permissions are set. An uppercase `S`in the fourth character position means that the file does *not* have the execute permission, only the setuid permission. Without the execute permission for the user owner, the setuid permission is ineffective.

13.6.2 setgid Permission

On a File

The setgid permission on a file works very similarly to the setuid permission, except that instead of executing as the user who owns the file, setgid permission will execute as the group that owns the file. The following displays the setgid executable file `/usr/bin/wall` when listed with the `ls -l` command:

```
sysadmin@localhost:~$ ls -l /usr/bin/wall
-rwxr-sr-x 1 root tty 19024 Jun  3 20:54 /usr/bin/wall
```

Notice from the listing, the seventh character is an `s` where there would normally be an `x` for the group execute permission. The lowercase `s` indicates that this file has both the setgid and execute permission set. An `S` instead of the `s`means the file lacks execute permission for the group. Without execute permission for the group, the setgid permission will be ineffective.

Consider This

The way that the Linux kernel restricts both setuid and setgid executables is stricter than some UNIX implementations. While Linux doesn't prevent users from setting these permissions on script files, the Linux kernel will not execute scripts with setuid or setgid permissions.

In order to effectively set these permissions, the Linux kernel will only honor them on executable files that are in a binary format. This increases the security of the system.

On a Directory

In most cases, once an installation of a Linux distribution is complete, the files that require setuid and setgid permission should have those permissions automatically set. While some of the setuid and setgid permissions on files may be removed for security reasons, very seldom are new setuid or setgid files created. On the other hand, it is fairly common for administrators to add setgid to *directories*.

Using setgid on a directory can be extremely helpful for users trying to share a directory of files with other users who are in different groups. To understand why, consider the following scenario.

Users from different groups in your organization have asked the administrator to set up a directory in which they can share files. For this example, there will be three users: `joe` who is a member of the `staff` group, `maya` who is a member of the `payroll` group and `steve` who is a member of the `acct` group. To accomplish this, the administrator takes the following steps:

- **Step #1:** Create a new group for the three users:

```
root@localhost:~# groupadd -r common
```

- **Step #2:** Add the users to the group:

```
root@localhost:~# usermod -aG common joe
root@localhost:~# usermod -aG common maya
```

- **Step #3:** Create a directory and make common the group owner:

```
root@localhost:~# mkdir /shared
root@localhost:~# chgrp common /shared
```

- **Step #4:** Make the directory writable for the group:

```
root@localhost:~# chmod 770 /shared
```

This solution is almost perfect: the users `joe`, `maya` and `steve` can now access the `/shared` directory and add new files. However, there is a potential problem: for example, when the user `joe` makes a new file in the `/shared`directory, it will likely end up looking like the following when displayed with the `ls -l` command:

```
-rw-rw----. 2 joe staff 8987 Jan 10 09:08 data.txt
```

The problem with this is that neither `maya` or `steve` are members of the `staff`group. As a result, their permissions are `---`, which means they can't access the contents of this file.

The `joe` user could have used the `newgrp` command to switch to the `common`group before creating the file. Or, after creating the file, the `joe` user could have used the `chgrp` command to change the group ownership to the `common` group. However, users won't always remember to run these commands; some may not even know that these commands exist.

Instead of having to train users how to use the `newgrp` and `chgrp` commands, the administrator can set up a directory with setgid permission. If a directory is setgid, then all new files created or copied into the directory will *automatically* be owned by the group that owns the directory. This means users don't have to use the `newgrp` or `chgrp` commands because the group ownership will be managed automatically.

So, a better solution to this scenario would be to use the following command for Step #4:

```
root@localhost:~# chmod 2775 /shared
```

Listing the details of the directory after running the previous command would result in the following output:

```
drwxrwsr-x. 2 root common 4096 Jan 10 09:08 /shared
```

After completing these steps, the `joe`, `maya` and `steve` users would now be able to easily create files in the `/shared` directory that would automatically be owned by the `common` group.

13.6.3 sticky bit Permission

The final advanced permission to discuss is the sticky permission, which is sometimes called the "sticky bit". Setting this permission can be important to prevent users from deleting other user's files.

Recall that to be able to delete a file, only the write and execute permission on the directory are necessary. The write and execute permission is also necessary to add files to a directory. So, if an administrator wants to make a directory where any user can add a file, the required permissions mean that user can also delete any other user's file in that directory.

The sticky permission on a directory modifies the write permission on a directory so that only specific users can delete a file within the directory:

- The user who owns the file
- The user who owns the directory that has the sticky bit set (typically this is the root user)
- The root user

On a typical Linux installation there will normally be two directories that have the sticky permission set by default: the `/tmp` and `/var/tmp` directories. Besides user home directories, these two directories are the only locations that regular users have write permission by default.

As their name suggests, these directories are intended to be used for temporary files. Any user can copy files to the `/tmp` or `/var/tmp` directory to share with others.

Since the /tmp or /var/tmp directories have the sticky permission, you don't have to worry about users deleting the shared files. However, there is a reason that these directories are considered temporary: these directories are automatically purged of their contents on a regular schedule. If you put a file in them, it will be removed automatically at some point in the future.

When a directory has the sticky permission, a `t` will replace the `x` in the set of permissions for others. For example, the following shows the output of the `ls -ld /tmp` command:

```
sysadmin@localhost:~$ ls -ld /tmp
drwxrwxrwt 1 root root 0 Oct 17 19:17 /tmp
```

The `/tmp` directory has an octal permission mode of `1777`, or full permissions for everyone plus the sticky permission on the directory. When the “others” permission set includes the execute permission, then the `t` will be in lowercase. If there is an uppercase `T`, it means that the execute permission is not present for others. This does not mean that the directory does not have an effective sticky permission, however it does mean that the users who are affected by the "others" permission can't change to this directory or create files in the directory.

There are cases in which users may not want to have execute for others, but still have the sticky bit permission on a directory. For example, when creating shared directories with the setgid permission. Consider the following command:

```
root@localhost:~# chmod 3770 /shared
```

Notice that the special permissions of setgid and stickybit can be added together, the `2000` permission plus the `1000` permission gives you `3000` for the special permissions. Add this to the basic permissions for the user, group and others; The owner and group have full access and others get none. Listing the directory after making the changes with the `ls -ld /shared` command results in the following output:

```
drwxrws--T. 2 root common 4096 Jan 10 09:08 /shared
```

The uppercase `T` in the execute permissions position for others indicates there is no execute permission for others. However, since multiple users of the group still have access, the sticky permission is effective for the `common` group.

Consider This

The `stat` command can displays permissions both in symbolic and octal notation, as well as user owner and group owner information. For example, look at the first line that starts with `Access`:

```
sysadmin@localhost:~$ stat /tmp
  File: '/tmp'
  Size: 0               Blocks: 0          IO Block: 4096   directory
Device: 3ah/58d Inode: 1493        Links: 1
Access: (1777/drwxrwxrwt)  Uid: (    0/    root)   Gid: (    0/    root)
Access: 2014-09-18 23:37:24.364779732 +0000
Modify: 2014-10-17 19:17:01.430467987 +0000
Change: 2014-10-17 19:17:01.430467987 +0000
 Birth: -
```

13.7 Default File Permissions

Unlike other operating systems where the permissions on a new directory or file may be inherited from the parent directory, Linux sets the default permissions on these new objects based upon the value of the creator's UMASK setting. The `umask` command is used to both set the UMASK value and display it.

The `umask` command is automatically executed when a shell is started. To have a persistent setting for UMASK, a custom `umask` command can be added to the `~/.bashrc` file.

To customize the UMASK value, it is important to understand what the UMASK value does. The value of the UMASK value only affects the permissions placed on new files and directories at the time they are created. It only affects the basic permissions for the user owner, the group owner and others. The UMASK value does not affect the special advanced permissions of setuid, setgid or sticky bit.

The UMASK is an octal value based upon the same values that you saw earlier in this section:

Permission	Octal Value
read	`4`
write	`2`
execute	`1`
none	`0`

When using the `chmod` command with octal values, add together the values for the user, the group and others. However, the UMASK octal value used to specify permissions to be *removed*. In other words, the octal value set for the UMASK is subtracted from the maximum possible permission to determine the permissions that are set when a file or directory is created.

Understanding UMASK for Files

By default, the maximum permissions that will be placed on a brand new file are `rw-rw-rw-`, which can be represented octal as `666`. The execute bit is turned off for security reasons. The UMASK value can be used to specify which of these default permissions to remove for new files. Three octal values will be provided: the value of the permissions to remove for the user owner, the group owner and others.

For example, to set a UMASK value for new files that would result in full permissions for the owner (result: `rw-`), remove or *mask* write permissions for the group owner (result: `r--`) and remove all permissions from others (result: `---`), calculate the UMASK value as follows:

- The first digit for the user owner would be a `0`, which would not mask any of the default permissions.
- The second digit would be a `2`, which would mask only the write permission.
- The third digit would be a `6`, which would mask the read and write permissions.

As a result, the UMASK value `026` would result in new files having the permissions of `rw-r-----`.

Another example: To set a UMASK value for new files that would remove write permissions for the owner (result: `r--`) and remove read and write permissions for the group and others (result: `---`), calculate the UMASK value as follows:

- The first digit for the user owner would be a `2`, which would mask only the write permission.
- The second digit would be a `6`, which would mask the read and write permissions.
- The third digit would be a `6`, which would mask the read and write permissions.

As a result, the UMASK value `266` would result in new files having the permissions of `r--------`.

Understanding UMASK for Directories

The way the UMASK is applied to regular files is different from directory files. For security reasons, regular files are not allowed to be executable at the time they are created. It would be dangerous to allow for a file to be able to run as a process without a user explicitly assigning the execute permission. So, regardless of whether you include the execute permission in the UMASK value, it will not apply to regular files.

For directories, the execute permission is critical to properly access the directory. Without the execute permission, you cannot navigate to a directory and the write permission is not functional. Essentially, directories are not very useful at all without execute permission, so new directories are allowed to be executable by default. The default permissions for new directories is `rwxrwxrwx` or `777`.

For example, if you wanted a UMASK value for new directories that would result in full permissions for the user owner (result: `rwx`), remove or mask write permissions for the group owner (result: `r-x`) and remove all permissions from others (result: `---`), then you could calculate the UMASK value as follows:

- The first digit for the user owner would be a `0`, which would not mask any of the default permissions.
- The second digit would be a `2`, which would mask only the write permission.
- The third digit would be a `7`, which would mask the read and write permissions.

As a result, the UMASK value `027` would result in new directories having the permissions of `r-xr-x---`.

Very Important: While the UMASK value effects the permissions for new files and directories differently (because of different maximum permissions), there is not a separate UMASK value for files and directories; the single UMASK value applies to both, as you can see from the following table of commonly used UMASK values:

umask	File Permissions	Directory Permissions	Description
`002`	`664` or `rw-rw-r--`	`775` or `rwxrwxr-x`	Default for ordinary users
`022`	`644` or `rw-r--r--`	`755` or `rwxr-xr-x`	Default for root user
`007`	`660` or `rw-rw----`	`770` or `rwxrwx---`	No access for others
`077`	`600` or `rw-------`	`700` or `rwx------`	Private to user

The following commands will display the current UMASK value, sets it to a different value and displays the new value:

```
sysadmin@localhost:~$ umask
0002
sysadmin@localhost:~$ umask 027
sysadmin@localhost:~$ umask
```

```
0027
```

Note: The initial `0` in the UMASK value is for special permissions (setuid, setgid and sticky bit). Since those are never set by default, the initial `0` is not necessary when setting the UMASK value.

Chapter 13: File Permissions and Ownership

This chapter will cover the following exam objectives:

104.5: Manage file permissions and ownership

Weight: 3

Description: Candidates should be able to control file access through the proper use of permissions and ownerships.

Key Knowledge Areas:

- Manage access permissions on regular and special files as well as directories
  Section 13.4 | Section 13.6
- Use access modes such as suid, sgid and the sticky bit to maintain security
  Section 13.6.1 | Section 13.6.2 | Section 13.6.3
- Know how to change the file creation mask
  Section 13.7
- Use the group field to grant file access to group members
  Section 13.4 | Section 13.6.2

Chapter 13: File Permissions and Ownership

chgrp

Command that is used to change the primary group of a file. Essentially it changes what group is the owner of the FILE.
Section 13.1

chmod

Command that is used to change the mode bits of a FILE. The chmod utility can be used to change the files permissions. For example setting the read, write, and execute bits.
Section 13.4

chown

Command that is used to change the ownership of a FILE. The chown utility can also be used to change the primary group of a FILE as well.
Section 13.1

umask

Command that sets the calling process's file mode creation mask. The umask utility will set the default permissions for FILEs when they are created.
Section 13.6

5.1 Introduction

Since everything is considered a file in Linux, file management is an important topic. Not only are your documents considered files, but hardware devices such as hard drives and memory as well. Even directories are considered files, since they are special files that are used to contain other files.

The essential commands needed for file management are often named by short abbreviations of their functions. You can use the ls command to list files, the cpcommand to copy files, the mv command to move or rename files and the rmcommand to delete files.

For working with directories, use the mkdir command to make directories, the rmdir command to remove directories (if they are empty) and the rm command to recursively delete directories containing files.

5.2 ls Command

While technically not a file management command, the ability to list files using lsis critical to file management. By default, the ls command will list the files in the current directory:

```
sysadmin@localhost:~$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos  test
```

For arguments, the ls command will accept an arbitrary number of different pathnames to attempt to list:

```
sysadmin@localhost:~$ ls . test /usr/local
.:
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos  test

/usr/local:
bin  etc  games  include  lib  man  sbin  share  src

test:
adjectives.txt          alpha.txt    linux.txt     people.csv
alpha-first.txt         animals.txt  longfile.txt  profile.txt
alpha-first.txt.original  food.txt     newhome.txt   red.txt
alpha-second.txt        hidden.txt   numbers.txt
alpha-third.txt         letters.txt  os.csv
```

Note the use of the . character. When used where a directory is expected, it represents the current directory. However, when the . character is used at the beginning of a file or directory name, it makes the file hidden by default. These *hidden files* are not normally displayed when using the ls command, though many can be commonly found in user home directories. Hidden files and directories are typically used for individual specific data or settings. Using the -aoption will display all files, including hidden files:

```
sysadmin@localhost:~$ ls -a
.           .bashrc   .selected_editor  Downloads  Public     test
..          .cache    Desktop          Music      Templates
.bash_logout  .profile  Documents        Pictures   Videos
```

The -A option to the ls command will display almost all files of a directory. The current directory file . and the parent directory file .. are omitted.

```
sysadmin@localhost:~$ ls -A
.bash_logout  .profile       Documents  Pictures  Videos
.bashrc       .selected_editor  Downloads  Public    test
.cache        Desktop        Music      Templates
```

To learn the details about a file, such as the type of file, the permissions, ownerships or the timestamp, perform a long listing. To get a detailed view, use the -l option to the ls command. In the example below, the /var/logdirectory is being listed because of its variety of output:

```
sysadmin@localhost:~$ ls -l /var/log/
total 1812
-rw-r--r-- 1          root        root        14100 Jul 23 18:45 alternatives.log
drwxr-xr-x 1          root        root        38 Jul 23 18:43 apt
-rw-r----- 1          syslog      adm         23621 Aug 23 15:17 auth.log
-rw-r--r-- 1          root        root        47816 Jul 17 03:36 bootstrap.log
-rw-rw---- 1          root        utmp        0 Jul 17 03:34 btmp
-rw-r----- 1          syslog      adm         8013 Aug 23 15:17 cron.log
-rw-r----- 1          root        adm         85083 Jul 23 18:45 dmesg
-rw-r--r-- 1          root        root        245540 Jul 23 18:45 dpkg.log
-rw-r--r-- 1          root        root        32064 Jul 23 18:45 faillog
drwxr-xr-x 1          root        root        32 Jul 17 03:36 fsck
-rw-r----- 1          syslog      adm         416 Aug 22 15:43 kern.log
-rw-rw-r-- 1          root        utmp        292584 Aug 20 18:44 lastlog
-rw-r----- 1          syslog      adm         0 Aug 22 06:25 syslog
-rw-r----- 1          syslog      adm         1087150 Aug 23 15:17 syslog.1
drwxr-xr-x 1          root        root        0 Apr 11 21:58 upstart
-rw-rw-r-- 1          root        utmp        384 Aug 20 18:44 wtmp
```

Viewing the above output as fields that are separated by spaces, they indicate:

- **File Type**

  - - rw-r--r-- 1 root root 14100 Jul 23 18:45 alternatives.log
  - d rwxr-xr-x 1 root root 38 Jul 23 18:43 apt

  The first field actually contains eleven characters, where the first character indicates the type of file and the next ten specify permissions. The file types are:

  - d for directory
  - - for regular file
  - l for a symbolic link
  - s for a socket
  - p for a pipe
  - b for a block file
  - c for a character file

Note that alternatives.log is a regular file -, while the apt is a directory d.

Permissions

```
d rwxr-xr-x 1 root root              0 Apr 11 21:58 upstart
```

The permissions are read r, write w, and execute x. Breaking this down a bit:

- rwx : Owner Permissions
- r-x : Group Permissions
- r-x : Everyone Else's Permissions

- **Hard Link Count**

```
-rw-r----- 1 syslog adm 23621 Aug 23 15:17 auth.log
```

There is only one directory name that links to this file. Hard links will be discussed in detail later in the course.

- **User Owner**

```
-rw-r----- 1 syslog adm 416 Aug 22 15:43 kern.log
```

User syslog owns this file. Every time a file is created, the ownership is automatically assigned to the user who created it.

- **Group Owner**

```
-rw-rw-r-- 1 root utmp 292584 Aug 20 18:44 lastlog
```

Although root created this file, any member of the utmp group has read and write access to it (as indicated by the group permissions).

- **File Size**

```
-rw-r----- 1 syslog adm 1087150 Aug 23 15:17 syslog.1
```

The size of the file in bytes. In the case of a directory, it might actually be a multiple of the block size used for the file system.

- **Timestamp**

```
drwxr-xr-x 1 root root 32 Jul 17 03:36 fsck
```

This indicates the time that the file's contents were last modified. This time will be listed as just the date and year if the file was last modified more than six months from the current date. Otherwise, the month, day, and time is displayed. Using the ls command with the --full-time option will display timestamps in full detail.

- **Filename**

```
-rw-r--r-- 1 root root 47816 Jul 17 03:36 bootstrap.log
```

The final field contains the name of the file or directory. In the case of symbolic links, the link name will be shown along with an arrow and the pathname of the file that is linked is shown.

```
lrwxrwxrwx. 1 root root 22 Nov 6 2012 /etc/grub.conf -> ../boot/grub/grub.conf
```

Sorting

The output of the ls command is sorted alphabetically by file name. It can sort by other methods as well:

- `-S` : Sort by files by size
- `-t` : Sort by timestamp
- `-r` : Reverses any type of sort

With both time and size sorts, add the `-l` option or the `--full-time` option, to be able to view those details:

```
sysadmin@localhost:~$ ls -t --full-time
total 0
drwxr-xr-x 1 sysadmin sysadmin   0 2014-09-18 22:25:18.000000000 +0000 Desktop
drwxr-xr-x 1 sysadmin sysadmin   0 2014-09-18 22:25:18.000000000 +0000 Documents
drwxr-xr-x 1 sysadmin sysadmin   0 2014-09-18 22:25:18.000000000 +0000 Downloads
drwxr-xr-x 1 sysadmin sysadmin   0 2014-09-18 22:25:18.000000000 +0000 Music
drwxr-xr-x 1 sysadmin sysadmin   0 2014-09-18 22:25:18.000000000 +0000 Pictures
drwxr-xr-x 1 sysadmin sysadmin   0 2014-09-18 22:25:18.000000000 +0000 Public
drwxr-xr-x 1 sysadmin sysadmin   0 2014-09-18 22:25:18.000000000 +0000 Templates
drwxr-xr-x 1 sysadmin sysadmin   0 2014-09-18 22:25:18.000000000 +0000 Videos
drwxr-xr-x 1 sysadmin sysadmin 420 2014-09-18 22:25:18.000000000 +0000 test
```

Recursion

When managing files it is important to understand what the term *recursive option* means. When the recursive option is used with file management commands, it means to apply that command to not only the specified directory, but also to all subdirectories and all of the files within all subdirectories.

Some commands use `-r` for recursive while others use `-R`. The `-R` option is used for recursive with the `ls` command because the `-r` option is used for reversing how the files are sorted:

```
sysadmin@localhost:~$ ls -lR /var/log
/var/log:
total 804
-rw-r--r-- 1 root   root  14100 Sep 18 22:25 alternatives.log
drwxr-xr-x 1 root   root     38 Sep 18 22:23 apt
-rw-r----- 1 syslog adm    1343 Sep 29 21:17 auth.log
-rw-r--r-- 1 root   root  47816 Sep 17 03:35 bootstrap.log
-rw-rw---- 1 root   utmp      0 Sep 17 03:34 btmp
-rw-r----- 1 syslog adm     546 Sep 29 21:17 cron.log
-rw-r----- 1 root   adm   85083 Sep 18 22:25 dmesg
-rw-r--r-- 1 root   root 252496 Sep 18 22:25 dpkg.log
-rw-r--r-- 1 root   root  32064 Sep 18 22:25 faillog
drwxr-xr-x 1 root   root     32 Sep 17 03:35 fsck
-rw-r----- 1 syslog adm     108 Sep 29 18:48 kern.log
-rw-rw-r-- 1 root   utmp 292584 Sep 29 18:48 lastlog
-rw-r----- 1 syslog adm   58355 Sep 29 21:17 syslog
drwxr-xr-x 1 root   root      0 Apr 11 21:58 upstart
-rw-rw-r-- 1 root   utmp    384 Sep 29 18:48 wtmp
```

```
/var/log/apt:
total 20
-rw-r--r-- 1 root root 13672 Sep 18 22:25 history.log
-rw-r----- 1 root adm    402 Sep 18 22:25 term.log

/var/log/fsck:
total 8
-rw-r----- 1 root adm 31 Sep 17 03:35 checkfs
-rw-r----- 1 root adm 31 Sep 17 03:35 checkroot

/var/log/upstart:
total 0
```

Use the recursive option carefully because its impact can be huge! With the `ls`command, using the `-R` option can result in a large amount of output in your terminal. For the other file management commands, the recursive option can impact a large amount of files and disk space.

The `-d` option to the `ls` command also important. When listing a directory, the `ls` command normally will show the contents of that directory, but when the `-d`option is added, then it will display the directory itself:

```
sysadmin@localhost:~$ ls -ld /var/log
drwxrwxr-x 1 root syslog 216 Sep 29 18:48 /var/log
```

5.3 file Command

The `file` command "looks at" the contents of a file to report what kind of file it is; it does this by matching the content to known types stored in a "magic" file. Many commands that you will use in Linux expect that the file name provided as an argument is a text file (rather than some sort of binary file). As a result, it is a good idea to use the `file` command before attempting to access a file to make sure that it does contain text. For example:

```
sysadmin@localhost:~$ file test/newhome.txt
test/newhome.txt: ASCII text
sysadmin@localhost:~$ file /bin/ls
/bin/ls: ELF 64-bit LSB  executable, x86-64, version 1 (SYSV), dynamically linked (uses shared libs), for G
NU/Linux 2.6.24, BuildID[sha1]=64d095bc6589dd4bfbf1c6d62ae985385965461b, stripped
```

Consider This

If you use a command that is expecting a text file argument, you may get strange results if you provide a non-text file. Your terminal may become disrupted by attempting to output binary content; as a result, the terminal may not be able to function properly any more.

The result will be "weird looking" characters being displayed in the terminal. To correct this situation, typing the command `reset` will often work. If this does not solve the problem, closing the terminal and opening a new one will solve the problem.

5.4 touch Command

The `touch` command performs two functions:

- create an empty file
- update the modification timestamp on an existing file

If the argument provided is an existing file, then the `touch` command will update the file's timestamp:

```
sysadmin@localhost:~$ ls -l test/red.txt
-rw-r--r-- 1 sysadmin sysadmin 51 Sep 18 22:25 test/red.txt
sysadmin@localhost:~$ touch test/red.txt
sysadmin@localhost:~$ ls -l test/red.txt
-rw-r--r-- 1 sysadmin sysadmin 51 Sep 29 22:35 test/red.txt
```

If the argument is a file that does not exist, a new file is created with the current time for the timestamp. The contents of this new file will be empty:

```
sysadmin@localhost:~$ touch newfile
sysadmin@localhost:~$ ls
Desktop  Downloads  Pictures  Templates  newfile
Documents  Music    Public   Videos    test
sysadmin@localhost:~$ ls -l newfile
-rw-rw-r-- 1 sysadmin sysadmin 0 Sep 29 22:39 newfile
```

Each file has three timestamps:

- The last time the file's contents were modified. This is the timestamp provided by the `ls -l` command by default. The `touch` command modified this timestamp by default.
- The last time the file was accessed. To modify this timestamp, use the `touch -a` command.
- The last time the file attributes, like permissions or ownership, were changed. These file attributes are also called the file's *metadata*. To modify this timestamp, use the `touch -c` command.

The `touch` command will normally update the specified time to the current time, but `-t` option with a timestamp value to can be used instead.

```
sysadmin@localhost:~$ touch -t 201412251200 newfile
sysadmin@localhost:~$ ls -l newfile
-rw-rw-r-- 1 sysadmin sysadmin 0 Dec 25  2014 newfile
```

In order to view all three timestamps that are kept for a file, use the `stat`command with the path to the file as an argument:

```
sysadmin@localhost:~$ stat test/alpha.txt
  File: 'test/alpha.txt'
  Size: 390           Blocks: 8          IO Block: 4096   regular file
Device: 58h/88d Inode: 13987       Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1001/sysadmin)   Gid: ( 1001/sysadmin)
Access: 2014-09-23 13:52:03.070012387 +0000
Modify: 2014-09-18 22:25:18.000000000 +0000
Change: 2014-09-23 13:52:03.070012387 +0000
```

Birth: -

5.5 cp Command

Creating copies of files can be useful for numerous reasons:

- If a copy of a file is created before changes are made, then it is possible to revert back to the original.
- It can be used to transfer a file to removable media devices.
- A copy of an existing document can be made, so to take advantage of the existing layout and content to get started more quickly than from scratch.

Consider This

While permissions will be covered in greater detail later in the course, they can have an impact on file management commands, such as the cpcommand.

For example, in order to be able to copy a file, you will need to have execute permission to access the directory where the file is located and the read permission for the file you are trying to copy.

You will also need to have write and execute permission on the directory where you want to put the copied file. Typically, there are two places where you should always have write and execute permission on the directory: your home directory (for "permanent" files) and the /tmp directory (for "temporary" files).

The copy command cp takes at least two arguments: the first argument is the path to the file to be copied and the second argument is the path to where the copy will be placed. The files to be copied are sometimes referred to as the source and the place to where the copies are placed in is called the destination.

Recall that the ~ character that represents your home directory; it is commonly used as a source or destination.

```
sysadmin@localhost:~$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos  test
```

For example, to copy the /etc/services file to your home directory, you could use the following command:

```
sysadmin@localhost:~$ cp /etc/services ~
```

The result of executing the previous command would create a copy of the contents of the /etc/services file in your home directory; the new file in your home directory would also be named services.

```
sysadmin@localhost:~$ ls
Desktop    Downloads  Pictures  Templates  services
Documents  Music      Public    Videos     test
```

To copy a file and rename the file in one step you can execute a command like:

```
sysadmin@localhost:~$ cp /etc/services ~/ports
```

The previous command would copy the contents of the /etc/services file into a new file named ports in the home directory.

To copy multiple files into a directory, additional file names to copy can be included as long as the last argument is a destination directory. Although they may only represent one argument, a wildcard pattern can be expanded to match multiple pathnames. These will be covered in detail in the next

chapter. The following command copies all files in the test directory that start with an n to the home directory, as a result newhome.txt and numbers.txt are copied.

```
sysadmin@localhost:~$ cp test/n* ~
sysadmin@localhost:~$ ls
Desktop   Downloads  Pictures  Templates  newhome.txt  ports    test
Documents  Music     Public    Videos     numbers.txt  services
```

An issue that frequently comes up when using wildcard patterns with the cpcommand is that sometimes they match the names of directories as well as regular files. If an attempt is made to copy a directory, then the cp command will print an error message. Using the -v option makes the cp command print verbose message, so you can always tell what copies succeed as well as those that fail. In the following wildcard example we've marked in red the lines which indicate that the directories were matched, but not copied.

```
sysadmin@localhost:~$ cp -v /etc/b* ~
'/etc/bash.bashrc' -> '/home/sysadmin/bash.bashrc'
cp: omitting directory '/etc/bash_completion.d'
cp: omitting directory '/etc/bind'
'/etc/bindresvport.blacklist' -> '/home/sysadmin/bindresvport.blacklist'
'/etc/blkid.conf' -> '/home/sysadmin/blkid.conf'
```

In order to copy a directory, its contents must be copied as well, including all files within the directories and all of its subdirectories. This can be done by using a recursive option of either -r or -R.

```
sysadmin@localhost:~$ cp -R -v /etc/perl ~
'/etc/perl' -> '/home/sysadmin/perl'
'/etc/perl/CPAN' -> '/home/sysadmin/perl/CPAN'
'/etc/perl/Net' -> '/home/sysadmin/perl/Net'
'/etc/perl/Net/libnet.cfg' -> '/home/sysadmin/perl/Net/libnet.cfg'
```

Consider This

The archive option -a copies the contents of the file and also attempts to maintain the original timestamps and file ownership. For regular users, only original timestamps can be maintained, since regular users can't create files owned by other users. If the root user uses the -a option, then the cpcommand will create a new file owned by the original file owner and also use the original file's timestamps.

The -a option also implies that recursion will be done, so it can also be used to copy a directory.

5.6 mv Command

The mv command is used to move a file from one pathname in the filesystem to another. When you move a file, it's like creating a copy of the original file with a new pathname and then deleting the original file.

Consider This

While permissions will be covered in greater detail later in the course, they can have an impact on file management commands, such as the mvcommand. Moving a file requires write and execute permissions on both the origin and destination directories.

If you move a file from one directory to another and you don't specify a new name for the file, then it will retain its original name. For example, the following will move the ~/test/red.txt file into the home directory ("~" refers to the user's home directory) and the resulting file name will be red.txt:

```
sysadmin@localhost:~/test$ mv red.txt ~
sysadmin@localhost:~/test$ ls ~
Desktop  Downloads  Pictures  Templates  red.txt
Documents  Music    Public   Videos    test
```

Like the cp command, the mv command will allow you to specify multiple files to move, as long as the final argument provided to the command is a directory. For example:

```
sysadmin@localhost:~/test$ mv animals.txt food.txt alpha.txt /tmp
sysadmin@localhost:~/test$ ls /tmp
alpha.txt  animals.txt  food.txt
```

There is no need for a recursive option with the mv command. When a directory is moved, everything it contains is automatically moved as well.

Moving a file within the same directory is an effective way to rename it. For example, in the following command the people.csv file is moved from the current directory to the current directory and given a new name of founders.csv. In other words people.csv is renamed founders.csv:

```
sysadmin@localhost:~/test$ mv people.csv founders.csv
```

5.7 rm Command

The rm command is used to delete files and directories. It is important to keep in mind that deleted files and directories do not go into a "trash can" as with desktop oriented operating systems. When a file is deleted with the rm command, it is permanently gone.

Without any options, the rm command is typically used to remove regular files:

```
sysadmin@localhost:~test$ rm alpha.txt
sysadmin@localhost:~test$ ls alpha.txt
ls: cannot access alpha.txt: No such file or directory
```

Extra care should be applied when using wildcards to specify which files to remove, as the extent to which the pattern might match files may be beyond what was anticipated. To avoid accidently deleting files when using globbing characters, use the -i option. This option makes the rm command confirm "interactively" every file that you delete:

```
sysadmin@localhost:~test$ rm -i a*
rm: remove regular file 'adjectives.txt'? y
rm: remove regular file 'alpha-first.txt'? y
rm: remove regular file 'alpha-first.txt.original'? y
rm: remove regular file 'alpha-second.txt'? y
rm: remove regular file 'alpha-third.txt'? y
rm: remove regular file 'animals.txt'? y
```

Some distributions make the -i option a default option by making an alias for the rm command.

The rm command will ignore directories that it's asked to remove; to delete a directory, use either the -r or -R options. Just be careful as this will delete all files and all subdirectories:

```
sysadmin@localhost:~$ rm test
```

```
rm: cannot remove 'test': Is a directory
sysadmin@localhost:~$ rm -r test
```

Consider This

While permissions will be covered in greater detail later in the course, they can have an impact on file management commands, such as the `rm`command.

To delete a file within a directory, a user must have write and execute permission on a directory. Regular users typically only have this type of permission in their home directory and its subdirectories.

The /tmp and /var/tmp directories do have the special permission called *sticky bit* set on them, so that files in these directories can only be deleted by the user that owns them (with the exception of the root user who can delete any file in any directory). So this means if you copy a file to the /tmpdirectory, then other users of the system will not be able to delete your file.

5.8 mkdir Command

The `mkdir` command allows you to create a directory. Creating directories is an essential file management skill, since you will want to maintain some functional organization with your files and not have them all placed in a single directory.

Typically, you only have a handful of directories in your home directory by default. Exactly what directories you have will vary due to the distribution of Linux, what software has been installed on your system and actions that may have been taken by the administrator.

For example, upon successful graphical login on a default installation of CentOS, the following directories have already been created automatically in the user's home directory: Desktop, Documents, Downloads, Music, Pictures, Public, Templates and Videos.

The bin directory is a common directory for users to create in their home directory. It is a useful directory to place *scripts* that the user has created. In the following example, the user creates a bin directory within their home directory:

```
sysadmin@localhost:~$mkdir bin
sysadmin@localhost:~$ ls
Desktop  Downloads  Pictures  Templates  bin
Documents  Music  Public  Videos  test
```

The `mkdir` command can accept a list of space separated pathnames for new directories to create:

```
sysadmin@localhost:~$mkdir one two three
sysadmin@localhost:~$ ls
Desktop  Downloads  Pictures  Templates  bin  test  two
Documents  Music  Public  Videos  one  three
```

Consider This

While permissions will be covered in greater detail later in the course, they can have an impact on file management commands, such as the `mkdir`command.

Creating a directory requires write and execute permissions for the parent of the proposed directory. For example, to issue the following command, it is necessary to have write and execute permission in the /etc directory: `mkdir /etc/test`

Directories are often described in terms of their relationship to each other. If one directory contains another directory, then it is referred to as the parent directory. The subdirectory is referred to as a child directory. In the following example /home is the parent directory and /sysadmin is the child directory:

```
sysadmin@localhost:~$ pwd
/home/sysadmin
```

When creating a child directory, its parent directory must first exist. If an administrator wanted to create directory /blue inside of /home/sysadmin/redwithout its existence, there would be an error:

```
sysadmin@localhost:~$ mkdir /home/sysadmin/red/blue
mkdir: cannot create directory '/home/sysadmin/red/blue': No such file or directory
```

By adding the `-p` option, the mkdir command automatically creates the parent directories for any child directories about to be created. This is especially useful for making "deep" pathnames:

```
sysadmin@localhost:~$ mkdir -p /home/sysadmin/red/blue/yellow/green
sysadmin@localhost:~$ ls -R red
red:
blue

red/blue:
yellow

red/blue/yellow:
green

red/blue/yellow/green:
```

5.9 rmdir Command

The `rmdir` command is used to remove *empty* directories.

```
sysadmin@localhost:~$ rmdir bin
```

Using the `-p` option with the `rmdir` command will remove directory paths, but only if all of the directories contain other empty directories.

```
sysadmin@localhost:~$ rmdir red
rmdir: failed to remove 'red': Directory not empty
sysadmin@localhost:~$ rmdir -p red/blue/yellow/green
```

Otherwise, if a directory contains anything except other directories, you'll need to use the command `rm` with a recursive option:

```
sysadmin@localhost:~$ rmdir test
rmdir: failed to remove 'test': Directory not empty
sysadmin@localhost:~$ rm -r test
```

Consider This

While permissions will be covered in greater detail later in the course, they can have an impact on file management commands, such as the `rmdir` command.

To delete a directory with the rmdir command, you must have write and execute permission on the parent directory. For example, to issue the following command, you would need to have write and execute permission in the /etc directory: rmdir /etc/test

Chapter 5: File Manipulation

This chapter will cover the following exam objectives:

103.3: Perform basic file management

Weight: 4

Description: Candidates should be able to use the basic Linux commands to manage files and directories.

Key Knowledge Areas:

- Copy, move and remove files and directories individually
  Section 5.1 | Section 5.5 | Section 5.6 | Section 5.7 | Section 5.8 | Section 5.9
- Copy multiple files and directories recursively
  Section 5.5
- Remove files and directories recursively
  Section 5.7 | Section 5.9
- Use simple and advanced wildcard specifications in commands
  Section 5.5 | Section 5.7

Chapter 5: File Manipulation

cp

Command used to copy files and directories. cp will copy a SOURCE to a DEST, or multiple SOURCES to a DIRECTORY.
Section 5.5

file

Command used to determine the type of file. file tests each argument in an attempt to classify it. There are three sets of tests, preformed in this order: filesystem test, magic tests, and language tests.
Section 5.3

ls

Command that will list information about files. The current directory is listed by default.
| Section 5.2

mkdir

Command used to create directories, if they do not already exist.
Section 5.8

mv

Command that can move files from one location to another, as well as renaming files.
Section 5.6

rm

Command used to remove files or directories. By default the rm command will not remove directories.
Section 5.7

rmdir

Command that is used to remove empty directories in the filesystem.
Section 5.9

touch

Command used to change the file timestamps. touch will allow a user to update the access and modification times of each FILE to the current time.
Section 5.4

8.1 Introduction

A large number of the files in a typical filesystem are *text files*. Text files contain simply text, no formatting features that you might see in a word processing file.

Because there are so many of these files on a typical Linux system, a great number of commands exist to help users manipulate text files. There are commands to both view and modify these files in various ways.

In addition, there are features available for the shell to control the output of commands, so instead of having the output placed in the terminal window, the output can be *redirected* into another file or another command. These redirection features provide users with a much more flexible and powerful environment to work within.

Searching and Extracting Data from Files

- Weight: 4
- Description: Search and extract from files in the home directory.
- Key Knowledge Areas:
    - Command line pipes
    - I/O re-direction
    - Partial POSIX Regular Expressions (.,[],*,?)
- The following is a partial list of the used files, terms, and utilities:
    - find
    - grep
    - less
    - head, tail
    - sort
    - cut
    - wc
- Thing that are nice to know:
    - Partial POSIX Basic Regular Expressions ([^], ^, $)
    - Partial POSIX Extended Regular Expressions (+,(),|)
    - xargs

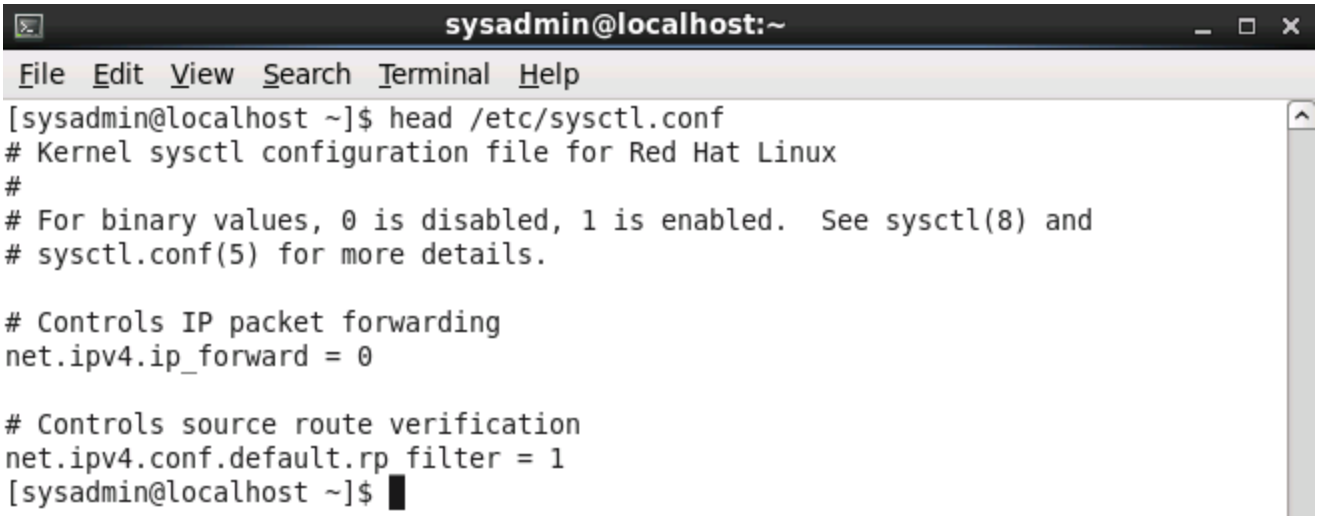
8.3 Command Line Pipes

Previous chapters discussed how to use individual commands to perform actions on the operating system, including how to create/move/delete files and move around the system. Typically, when a command has output or generates an error, the output is displayed to the screen; however, this does not have to be the case.

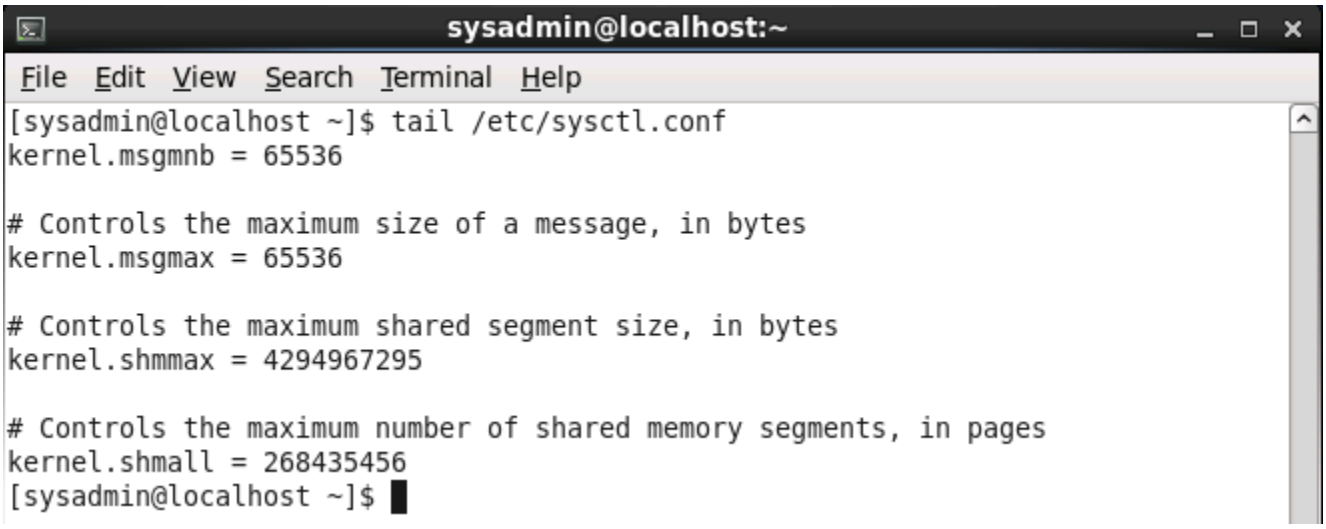
The *pipe* (|) character can be used to send the output of one command to another. Instead of being printed to the screen, the output of one command becomes input for the next command. This can be a powerful tool, especially when looking for specific data; *piping* is often used to refine the results of an initial command.

The `head` and `tail` commands will be used in many examples below to illustrate the use of pipes. These commands can be used to display only the first few or last few lines of a file (or, when used with a pipe, the output of a previous command).

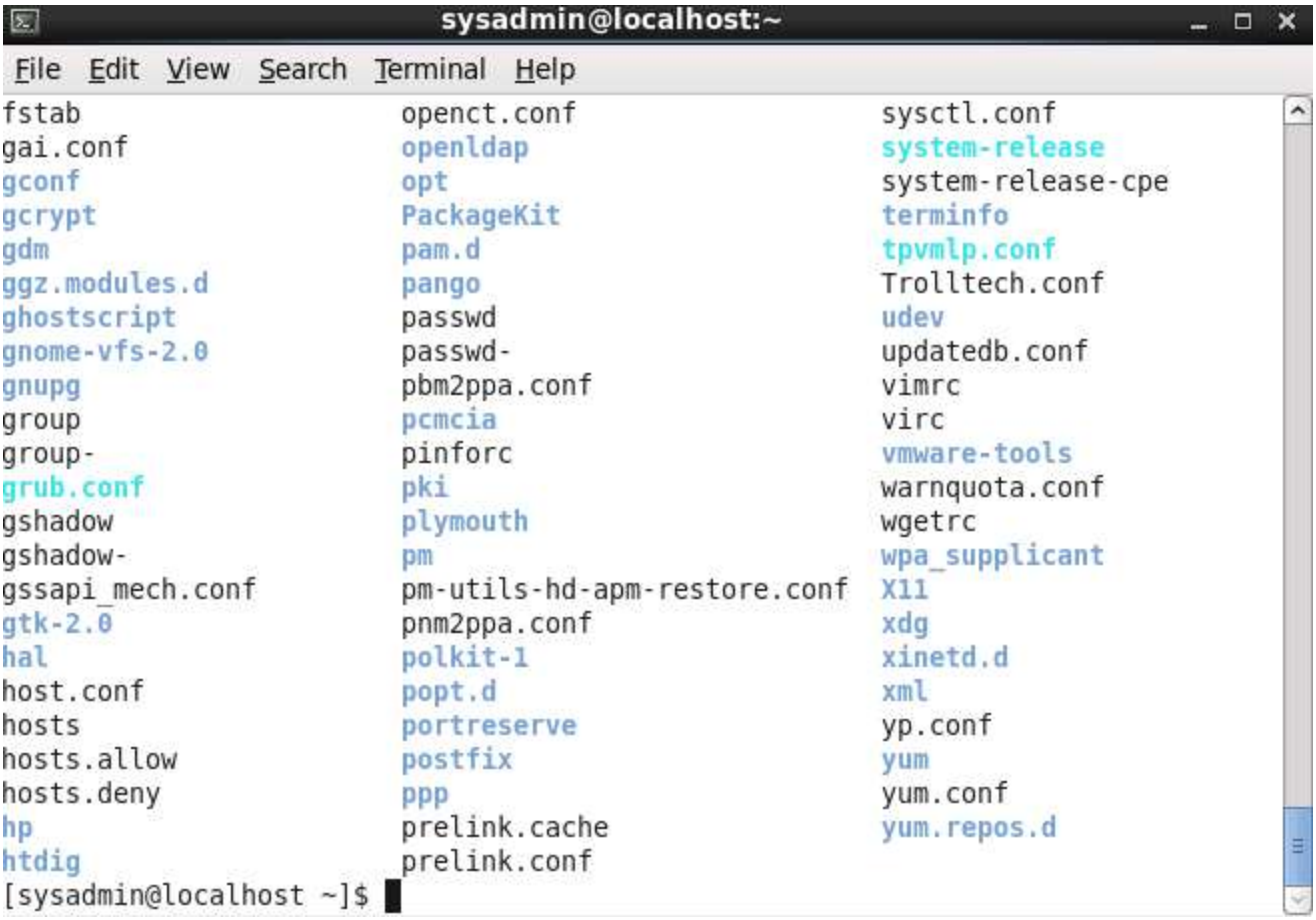
By default the `head` and `tail` commands will display ten lines. For example, the following command will display the first ten lines of the `/etc/sysctl.conf` file:



In the next example, the last ten lines of the file will be displayed:

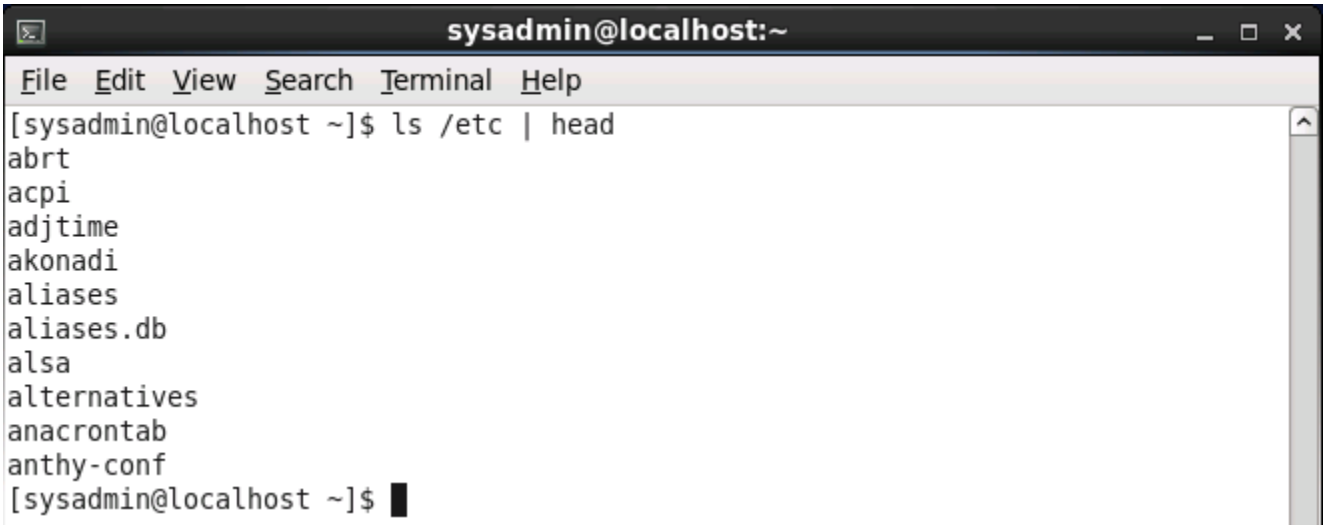


The pipe character will allow users to utilize these commands not only on files, but on the output of other commands. This can be useful when listing a large directory, for example the `/etc` directory:



If you look at the output of the previous command, you will note that first filename is `fstab`. But there are other files listed "above" that can only be viewed if the user uses the scroll bar. What if you just wanted to list the first few files of the `/etc` directory?

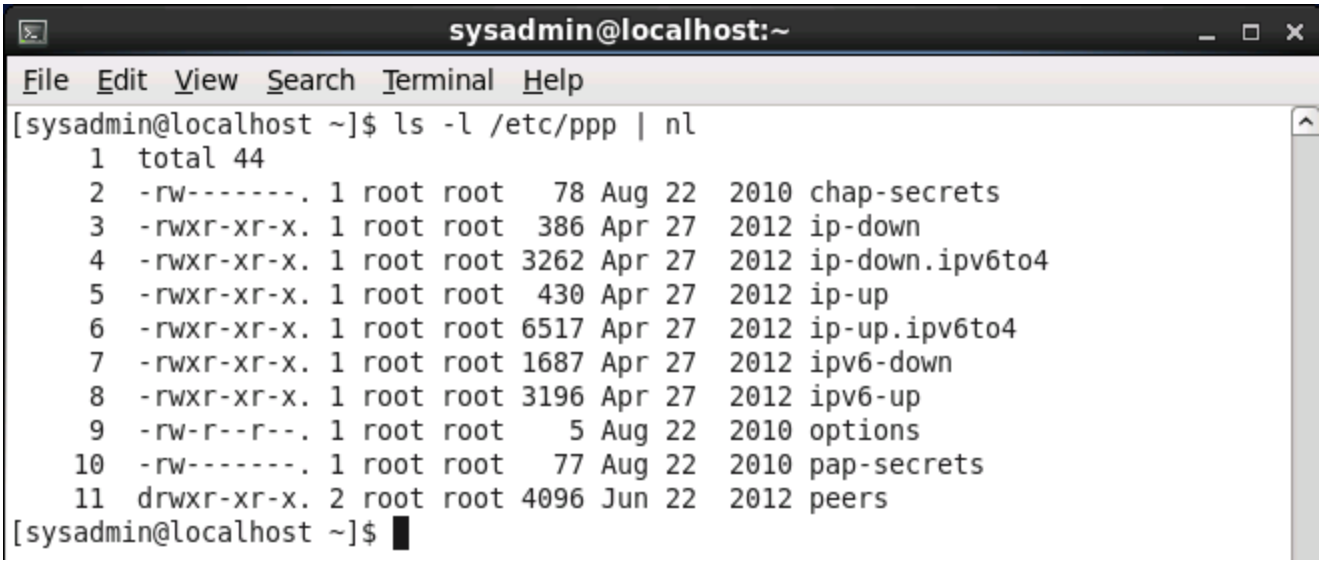
Instead of displaying the full output of the above command, piping it to the `head` command will display only the first ten lines:



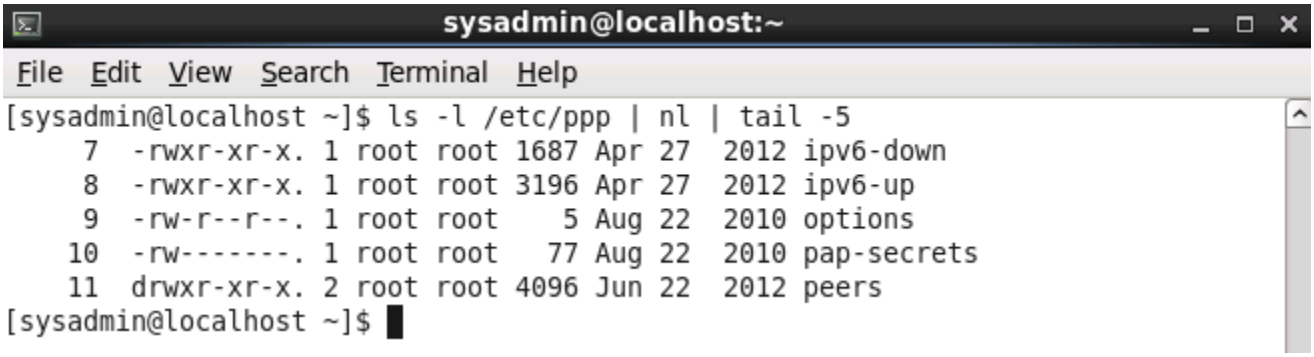
The full output of the `ls` command is passed to the `head` command by the shell instead of being printed to the screen. The `head` command takes this output (from ls) as "input data" and the output of `head` is then printed to the screen.

Multiple pipes can be used consecutively to link multiple commands together. If three commands are piped together, the first command's output is passed to the second command. The output of the second command is then passed to the third command. The output of the third command would then be printed to the screen.

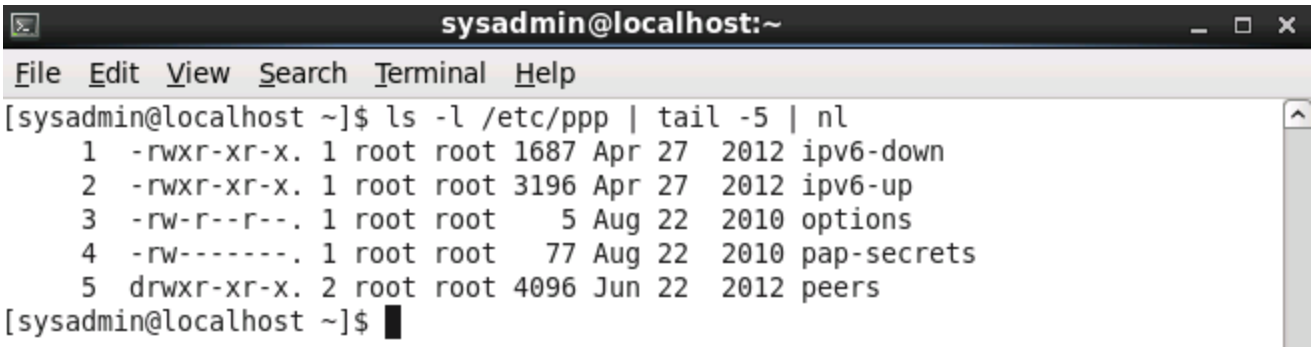
It is important to carefully choose the order in which commands are piped, as the third command will only see input from the output of the second. The examples below illustrate this using the `nl` command. In the first example, the `nl` command is used to number the lines of the output of a previous command:



In the next example, note that the `ls` command is executed first and its output is sent to the `nl` command, numbering all of the lines from the output of the `ls` command. Then the `tail` command is executed, displaying the last five lines from the output of the `nl` command:



Compare the output above with the next example:



Notice how the line numbers are different. Why is this?

In the second example, the output of the `ls` command is first sent to the `tail` command which "grabs" only the last five lines of the output. Then the `tail` command sends those five lines to the `nl` command, which numbers them 1-5.

Pipes can be powerful, but it is important to consider how commands are piped to ensure that the desired output is displayed.

8.4 I/O Redirection

Input/Output (I/O) redirection allows for command line information to be passed to different *streams*. Before discussing redirection, it is important to understand standard streams.

8.4.1 STDIN

Standard input, or STDIN, is information entered normally by the user via the keyboard. When a command prompts the shell for data, the shell provides the user with the ability to type commands that, in turn, are sent to the command as STDIN.

8.4.2 STDOUT

Standard output, or STDOUT, is the normal output of commands. When a command functions correctly (without errors) the output it produces is called STDOUT. By default, STDOUT is displayed in the terminal window (screen) where the command is executing.

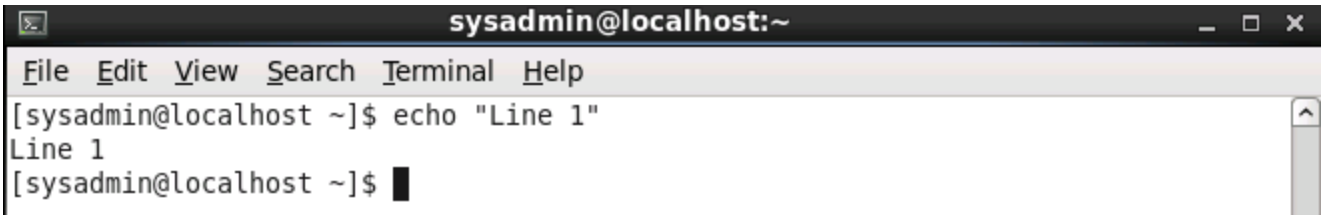
8.4.3 STDERR

Standard error, or STDERR, are error messages generated by commands. By default, STDERR is displayed in the terminal window (screen) where the command is executing.

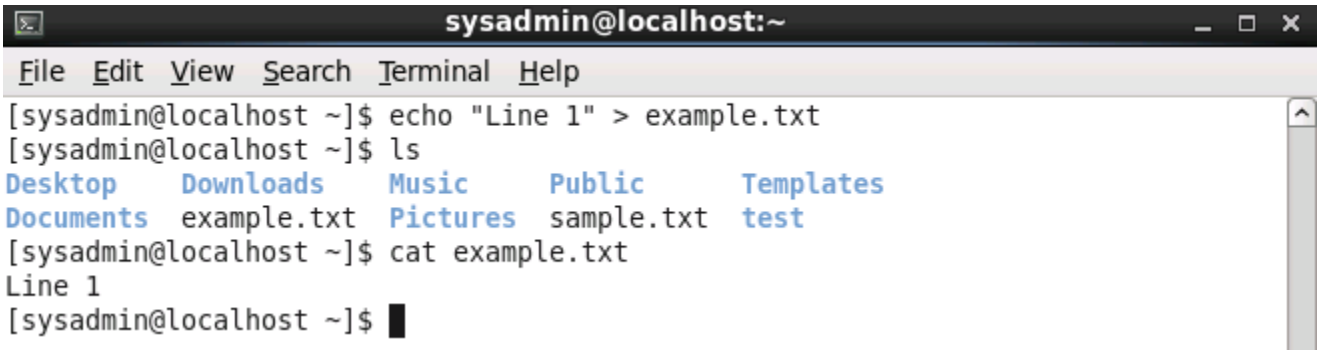
I/O redirection allows the user to redirect STDIN so data comes from a file and STDOUT/STDERR so output goes to a file. Redirection is achieved by using the arrow characters: (<) and (>).

8.4.4 Redirecting STDOUT

STDOUT can be directed to files. To begin, observe the output of the following command which will display to the screen:

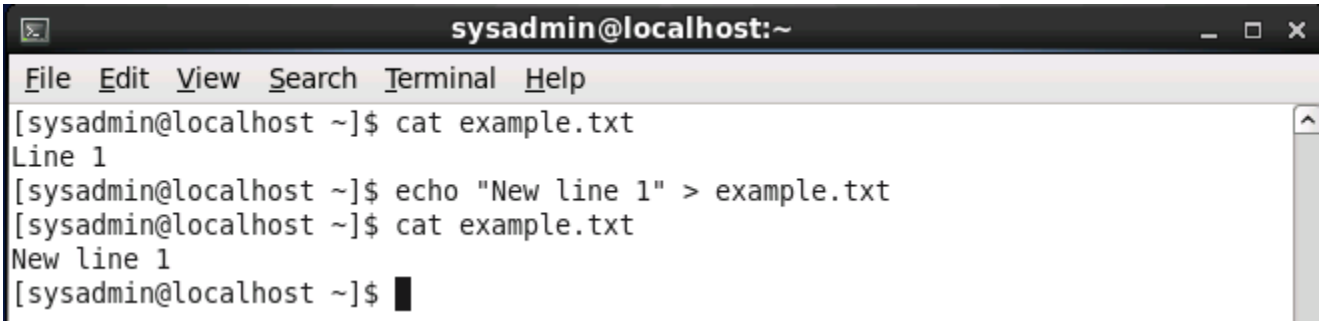


Using the > character the output can be redirected to a file:



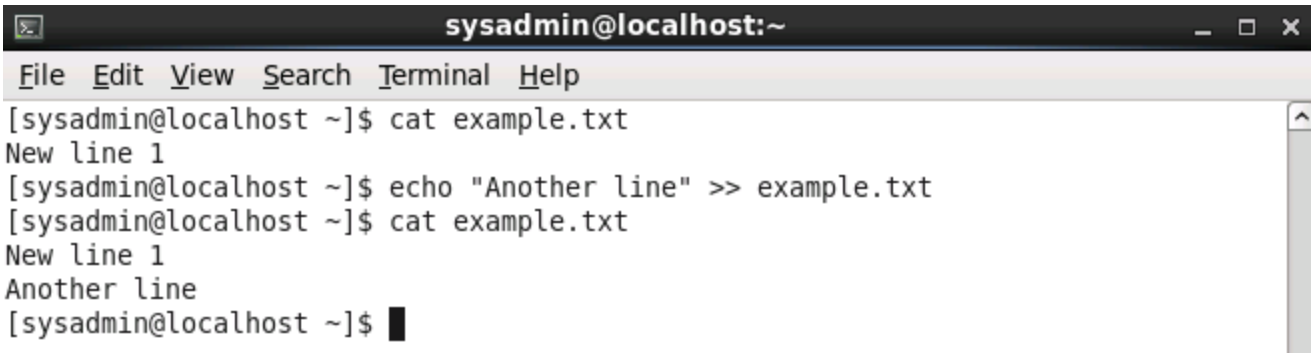
This command displays no output, because STDOUT was sent to the file `example.txt` instead of the screen. You can see the new file with the output of the `ls` command. The newly-created file contains the output of the `echo` command when the file is viewed with the `cat` command.

It is important to realize that the single arrow will overwrite any contents of an existing file:



The original contents of the file are gone, replaced with the output of the new `echo` command.

It is also possible to preserve the contents of an existing file by appending to it. Use "double arrow" (>>) to append to a file instead of overwriting it:



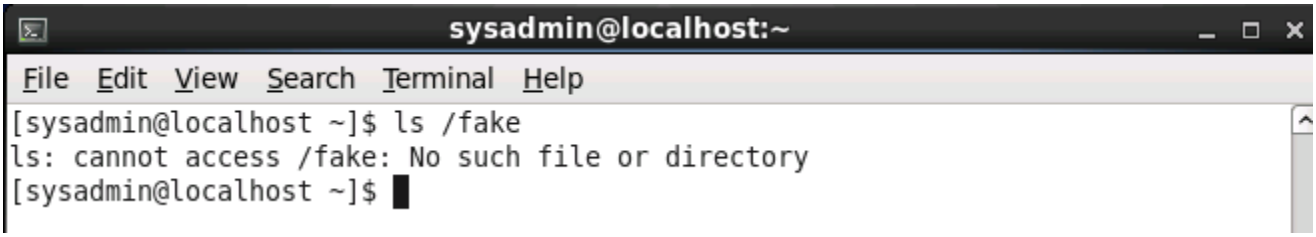
Instead of being overwritten, the output of the most recent `echo` command is added to the bottom of the file.

8.4.5 Redirecting STDERR

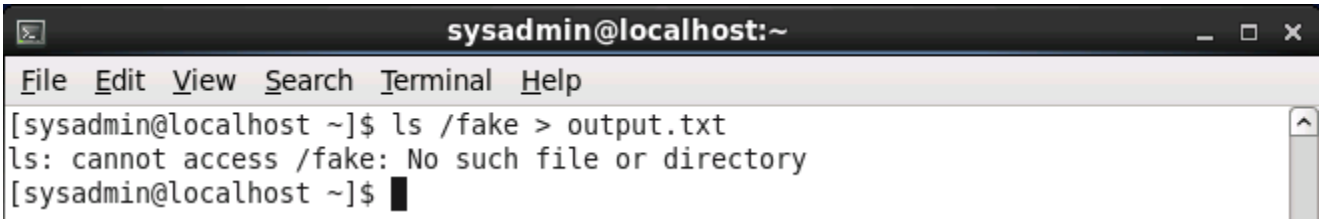
STDERR can be redirected in a similar fashion to STDOUT. STDOUT is also known as *stream* (or*channel*) #1. STDERR is assigned stream #2.

When using arrows to redirect, stream #1 is assumed unless another stream is specified. Thus, stream #2 must be specified when redirecting STDERR.

To demonstrate redirecting STDERR, first observe the following command which will produce an error because the specified directory does not exist:

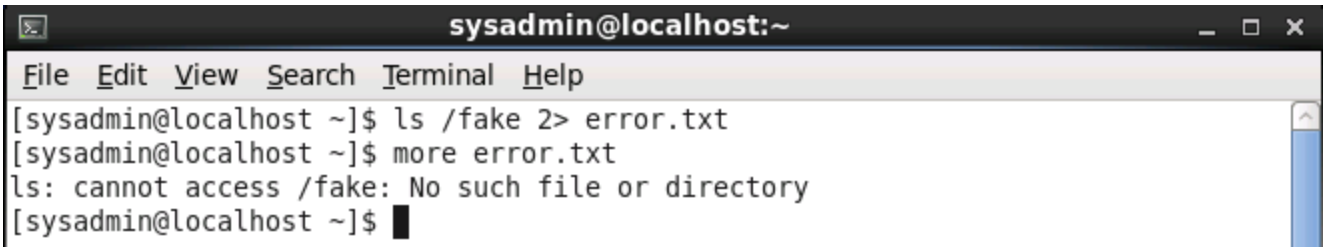


Note that there is nothing in the example above that implies that the output is STDERR. The output is clearly an error message, but how could you tell that it is being sent to STDERR? One easy way to determine this is to redirect STDOUT:



In the example above, STDOUT was redirected to the `output.txt` file. So, the output that is displayed can't be STDOUT because it would have been placed in the `output.txt` file. Because all command output goes either to STDOUT or STDERR, the output displayed above must be STDERR.

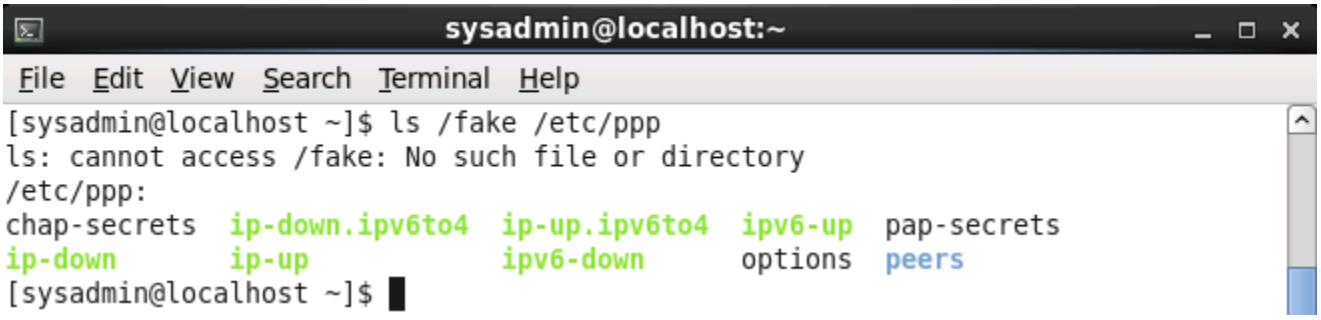
The STDERR output of a command can be sent to a file:



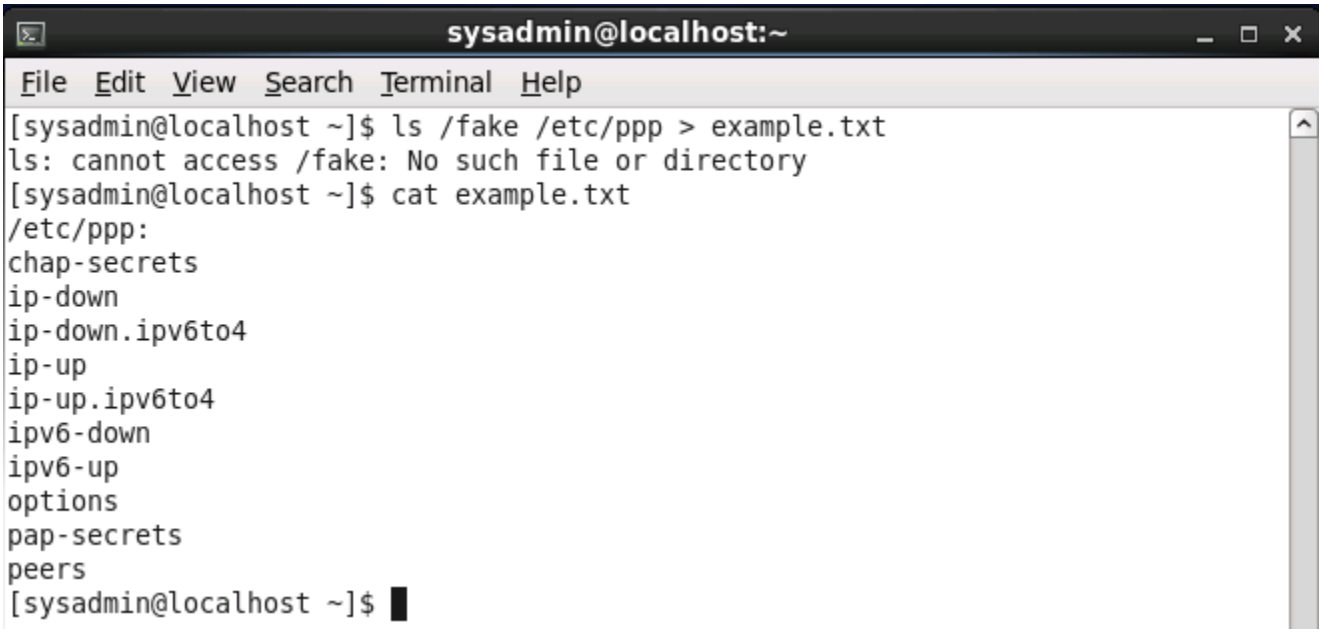
In the command above, the `2>` indicates that all error messages should be sent to the file `error.txt`.

8.4.6 Redirecting Multiple Streams

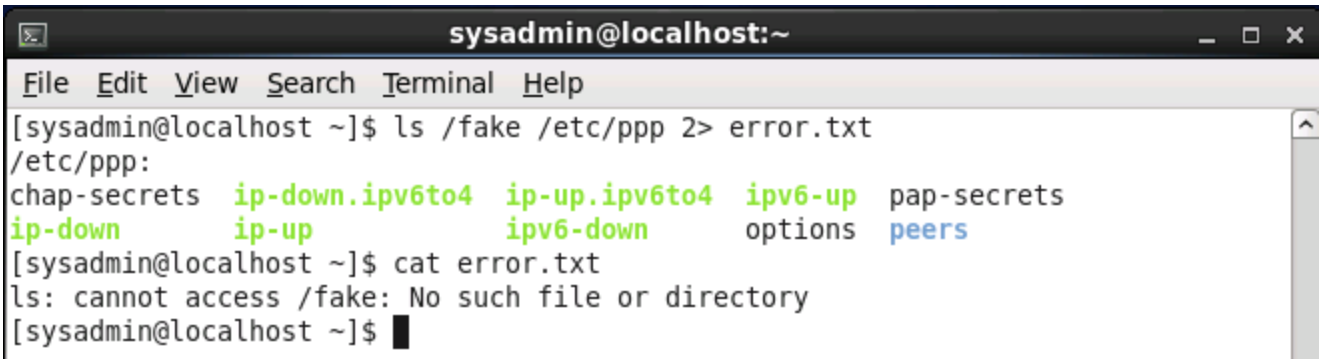
It is possible to direct both the STDOUT and STDERR of a command at the same time. The following command will produce both STDOUT and STDERR because one of the specified directories exists and the other does not:



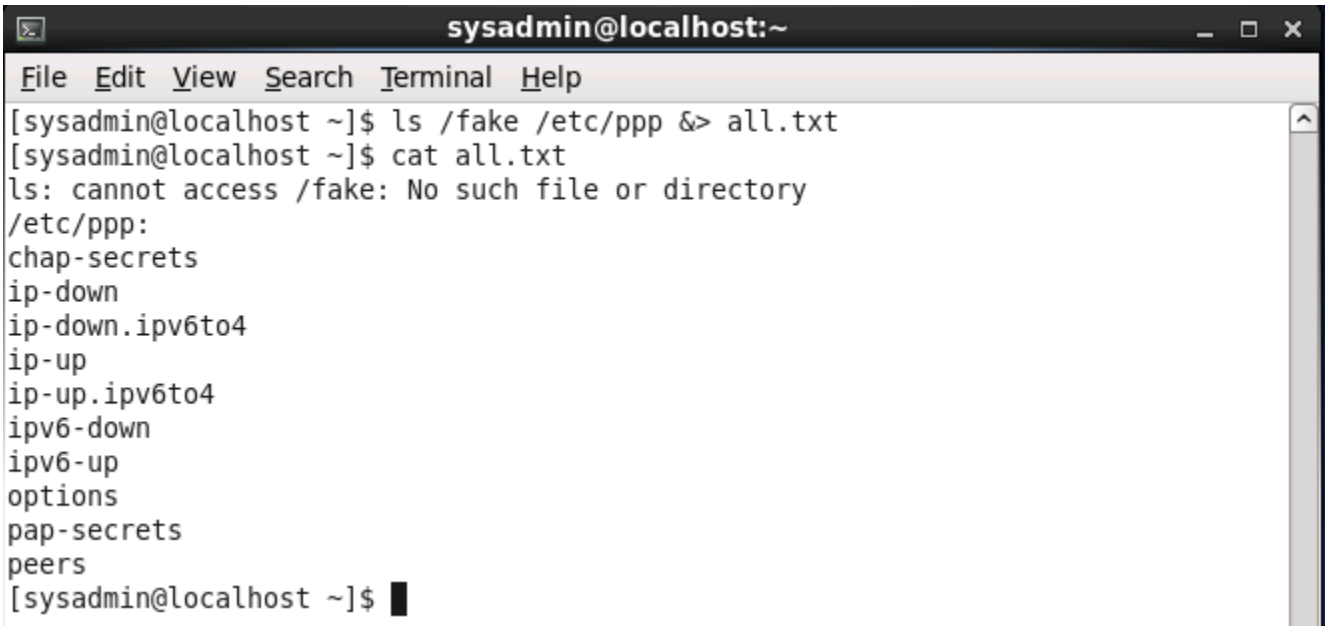
If only the STDOUT is sent to a file, STDERR will still be printed to the screen:



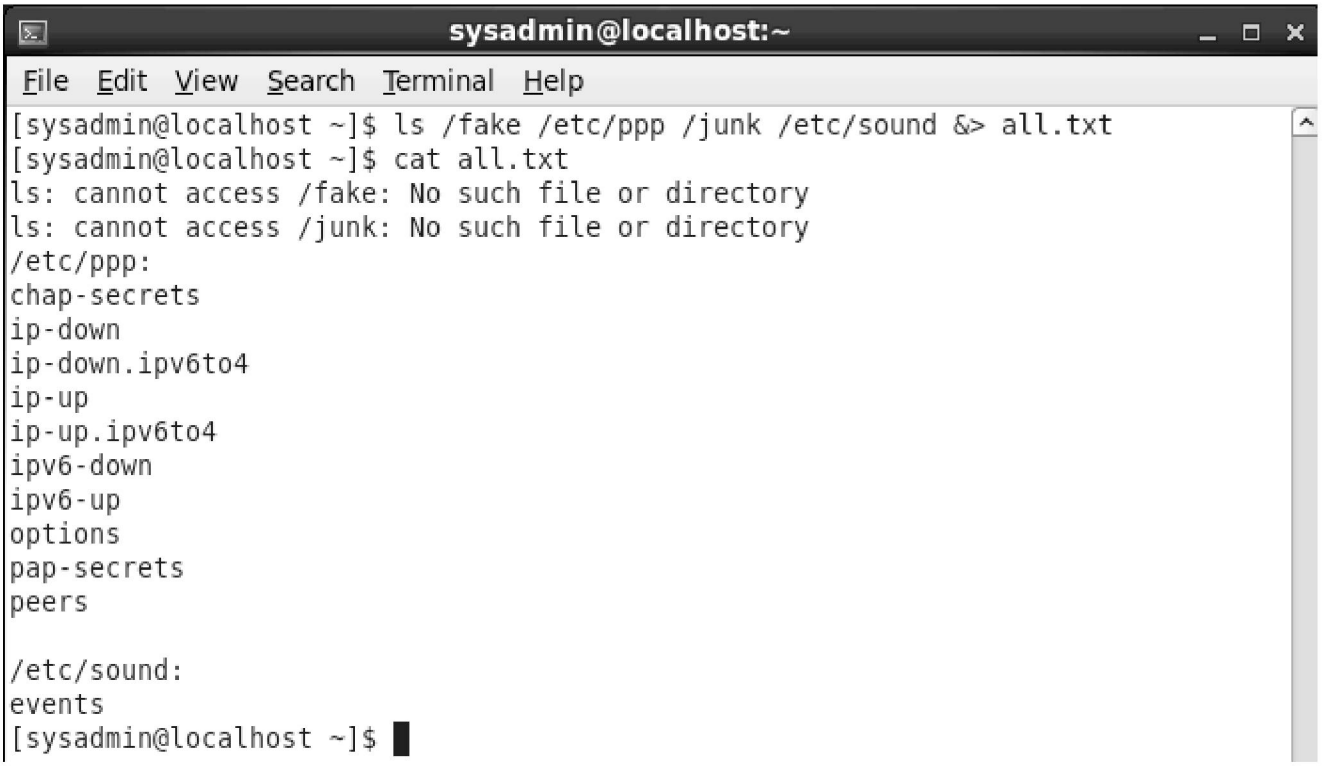
If only the STDERR is sent to a file, STDOUT will still be printed to the screen:



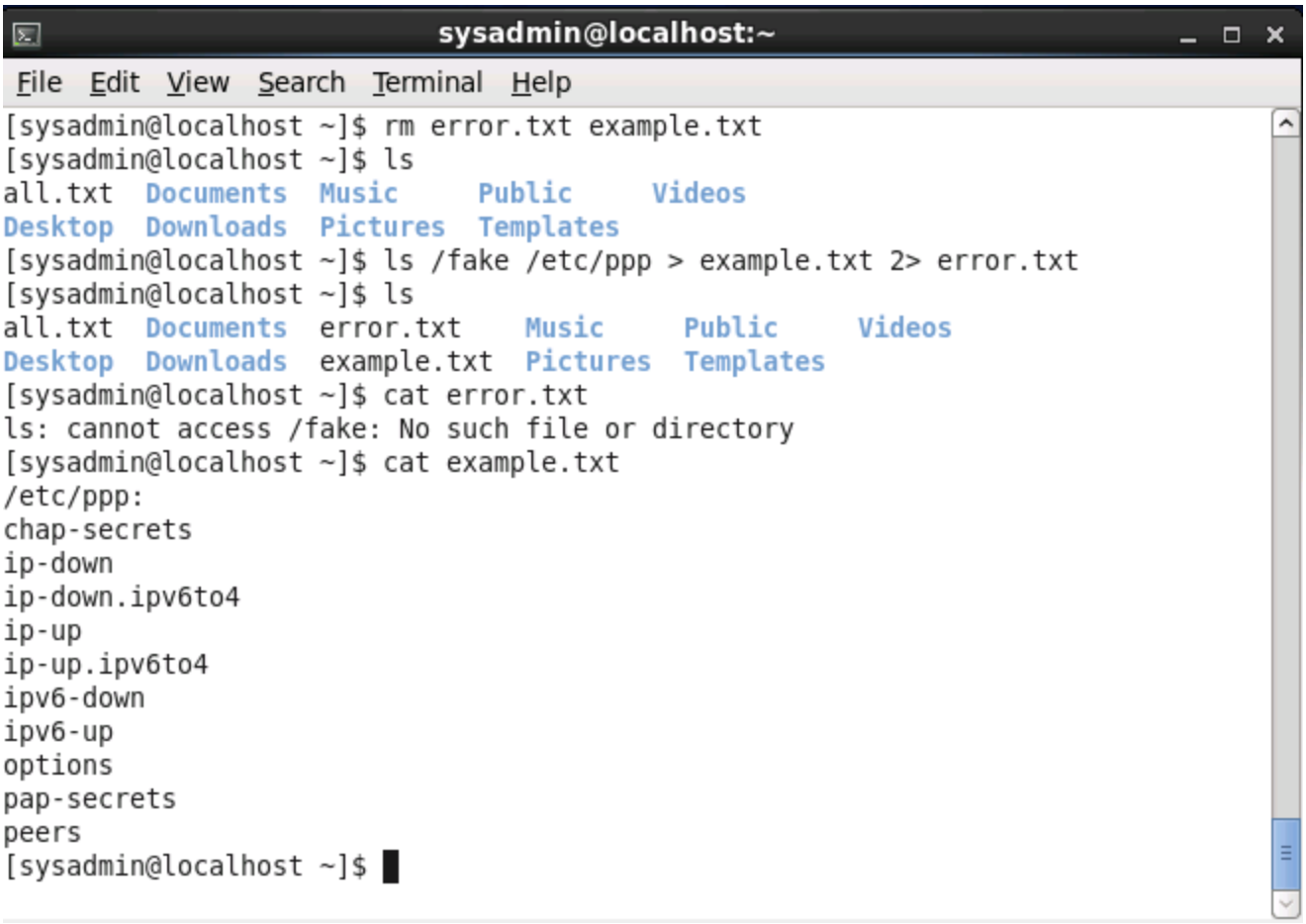
Both STDOUT and STDERR can be sent to a file by using &>, a character set that means "both 1> and 2>":



Note that when you use &>, the output appears in the file with all of the STDERR messages at the top and all of the STDOUT messages below all STDERR messages:



If you don't want STDERR and STDOUT to both go to the same file, they can be redirected to different files by using both > and 2>. For example:



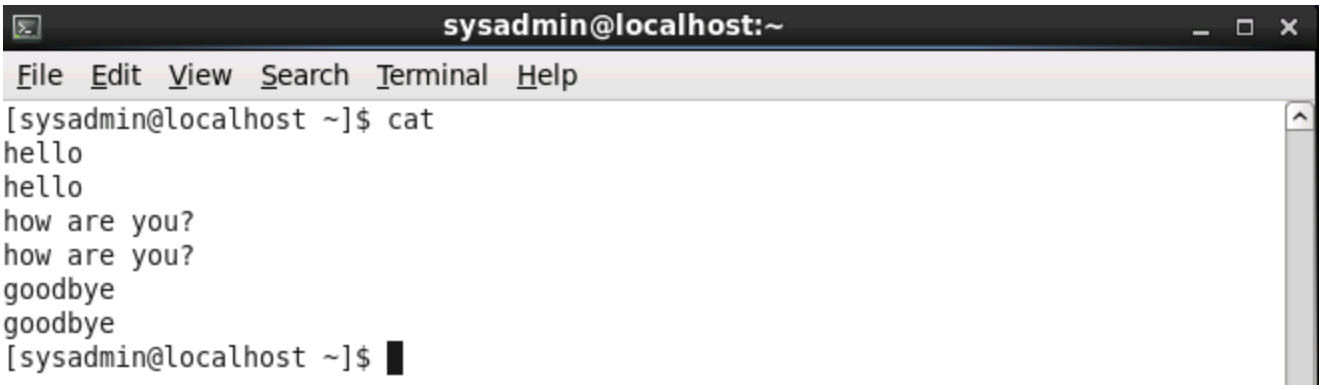
The order the streams are specified in does not matter.

8.4.7 Redirecting STDIN

The concept of redirecting STDIN is a difficult one because it is more difficult to understand *why*you would want to redirect STDIN. With STDOUT and STDERR, the answer to *why* is fairly easy: because sometimes you want to store the output into a file for future use.

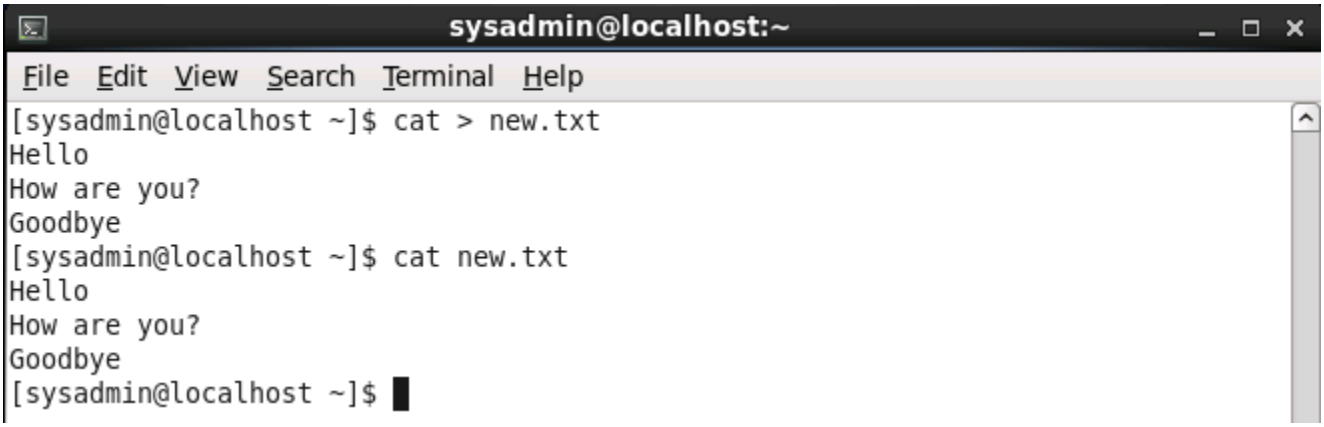
Most Linux users end up redirecting STDOUT routinely, STDERR on occasion and STDIN...well, very rarely. There are very few commands that require you to redirect STDIN because with most commands if you want to read data from a file into a command, you can just specify the filename as an argument to the command. The command will then look into the file.

For some commands, if you don't specify a filename as an argument, they will revert to using STDIN to get data. For example, consider the following `cat` command:



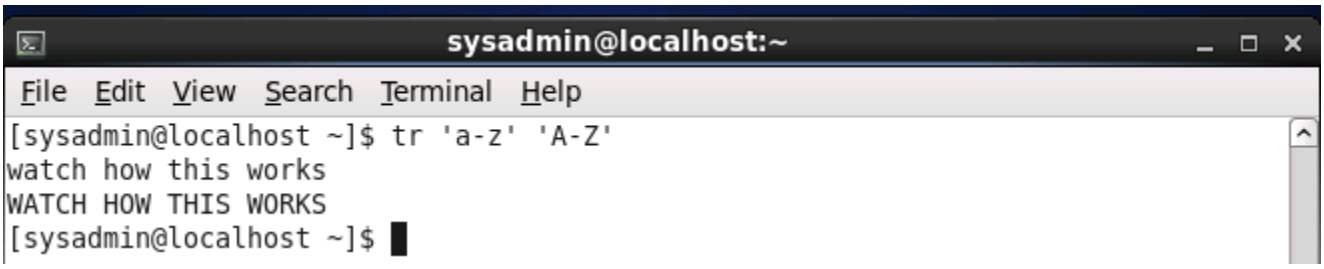
In the example above, the `cat` command wasn't provided a filename as an argument. So, it asked for the data to display on the screen from SDTIN. The user typed "hello" and then the `cat` command displayed "hello" on the screen. Perhaps this is useful for lonely people, but not really a good use of the `cat` command.

However, perhaps if the output of the `cat` command were redirected to a file, then this method could be used either to add to an existing file or to place text into a new file:



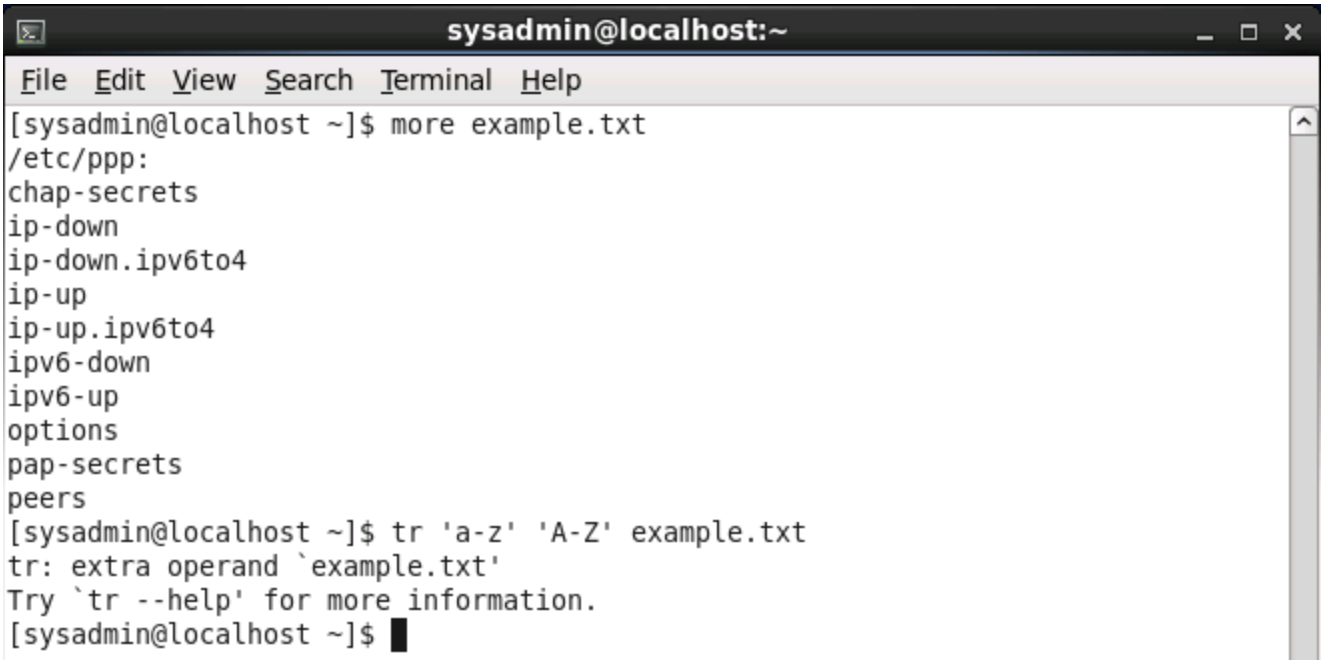
While the previous example demonstrates another advantage of redirecting STDOUT, it doesn't address why or how STDIN can be directed. To understand this, first consider a new command called `tr`. This command will take a set of characters and translate them into another set of characters.

For example, suppose you wanted to capitalize a line of text. You could use the `tr` command as follows:

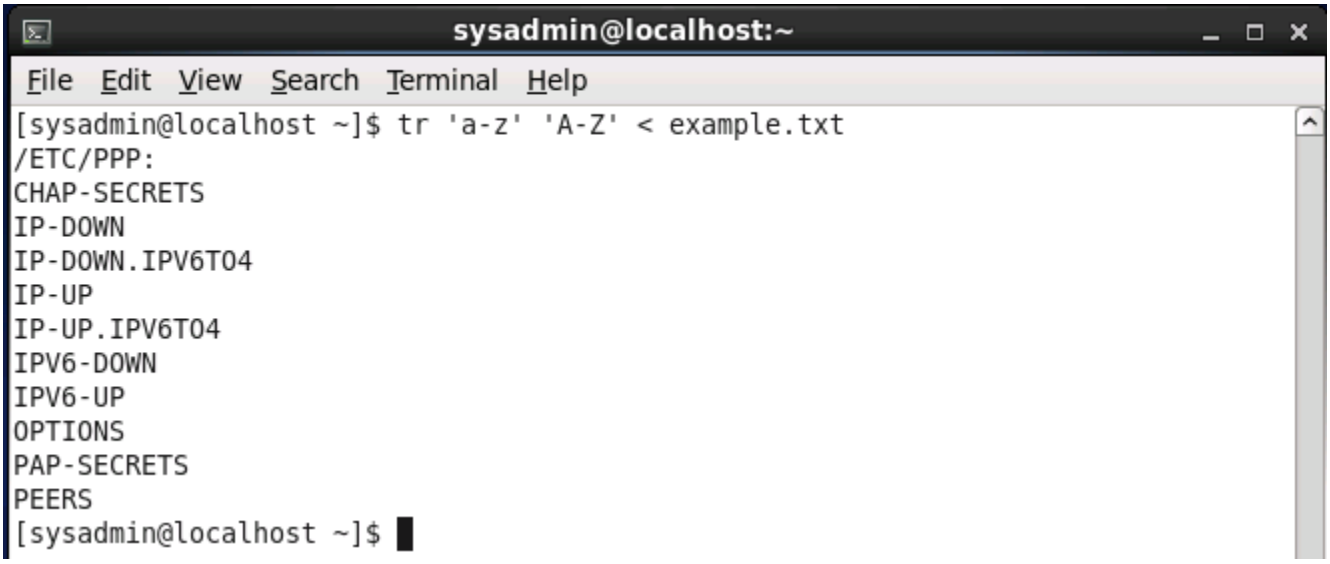


The `tr` command took the STDIN from the keyboard ("watch how this works") and converted all lower case letters before sending STDOUT to the screen ("WATCH HOW THIS WORKS").

It would seem that a better use of the `tr` command would be to perform translation on a file, not keyboard input. However, the `tr` command does not support filename arguments:

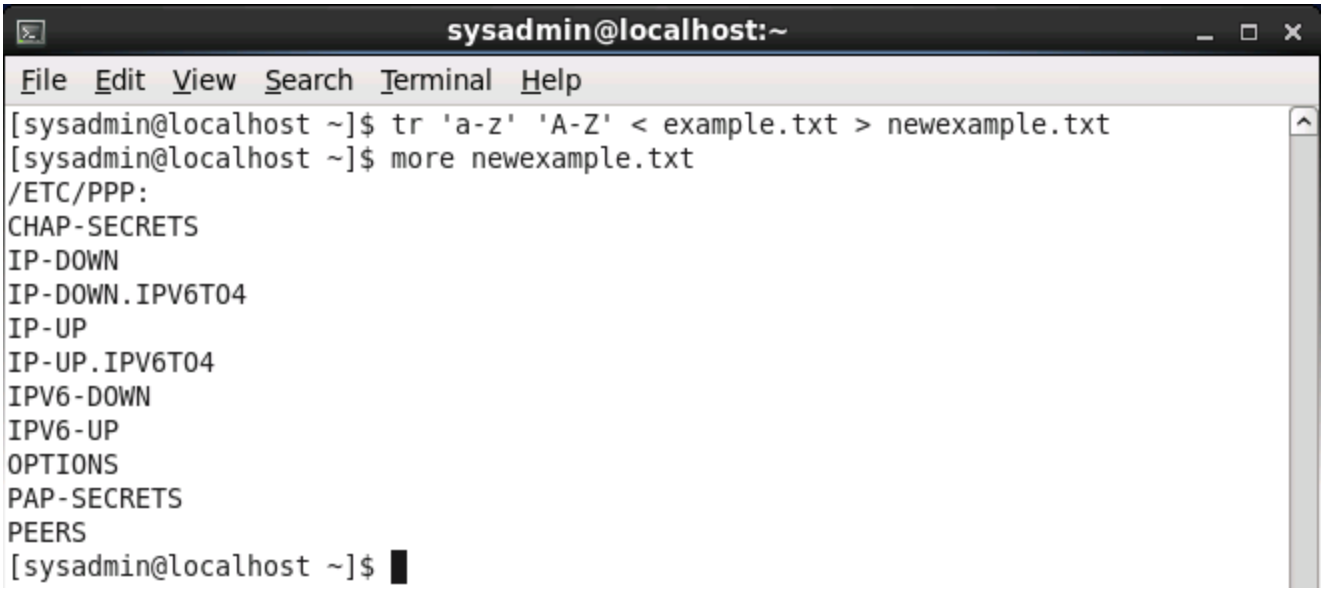


You can, however, tell the shell to get STDIN from a file instead of from the keyboard by using the < character:



This is fairly rare because most commands do accept filenames as arguments. But, for those that do not, this method could be used to have the shell read from the file instead of relying on the command to have this ability.

One last note: In most cases you probably want to take the resulting output and place it back into another file:



8.5 Searching for Files Using the Find Command

One of the challenges that users face when working with the filesystem, is trying to recall the location where files are stored. There are thousands of files and hundreds of directories on a typical Linux filesystem, so recalling where these files are located can pose challenges.

Keep in mind that most of the files that you will work with are ones that you create. As a result, you often will be looking in your own home directory to find files. However, sometimes you may need to search in other places on the filesystem to find files created by other users.

The `find` command is a very powerful tool that you can use to search for files on the filesystem. This command can search for files by name, including using wildcard characters for when you are not certain of the exact filename. Additionally, you can search for files based on file metadata, such as file type, file size and file ownership.

The syntax of the `find` command is:

```
find [starting directory] [search option] [ search criteria] [result option]
```

A description of all of these components:

Component	Description
[starting directory]	This is where the user specifies where to start searching. The `find` command will search this directory and all of its subdirectories. If no starting directory is

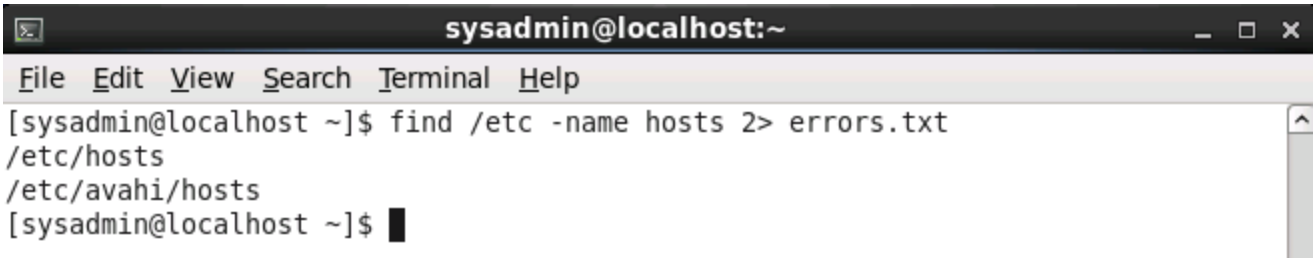
Component	Description
	provided, then the current directory is used for the starting point.
[search option]	This is where the user specifies an option to determine what sort of metadata to search for; there are options for file name, file size and many other file attributes.
[search criteria]	This is an argument that compliments the search option. For example, if the user uses the option to search for a file name, the search criteria would be the filename.
[result option]	This option is used to specify what action should be taken once the file is found. If no option is provided, the file name will be printed to STDOUT.

8.5.1 Search by File Name

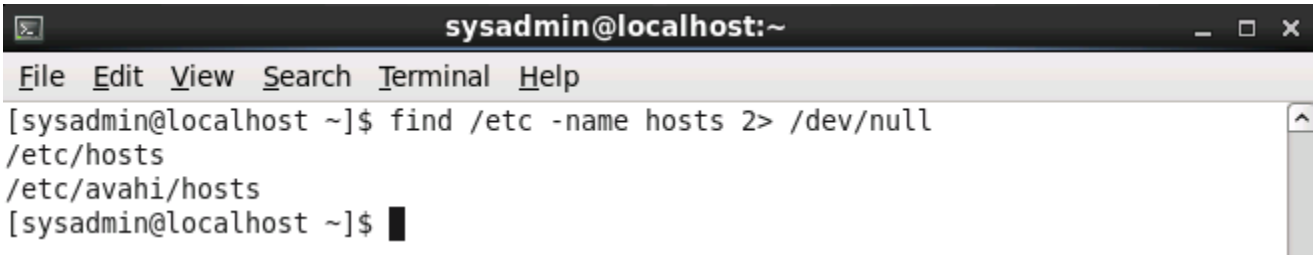
To search for a file by name, use the `-name` option to the `find` command:

Note that two files were found: `/etc/hosts` and `/etc/avahi/hosts`. The rest of the output was STDERR messages because the user who ran the command didn't have the permission to access certain subdirectories.

Recall that you can redirect STDERR to a file so you don't need to see these error messages on the screen:



While the output is easier to read, there really is no purpose to storing the error messages in the `error.txt` file. The developers of Linux realized that it would be good to have a "junk file" to send unnecessary data; any file that you send to the `/dev/null` file is discarded:

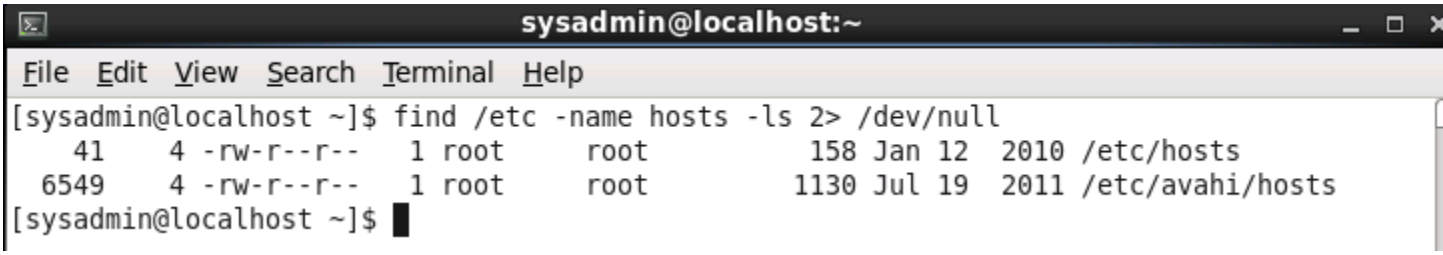


8.5.2 Displaying File Detail

It can be useful to obtain file details when using the `find` command because just the file name itself might not be enough information for you to find the correct file.

For example, there might be seven files named hosts; if you knew that the host file that you needed had been modified recently, then the modification timestamp of the file would be useful to see.

To see these file details, use the `-ls` option to the `find` command:

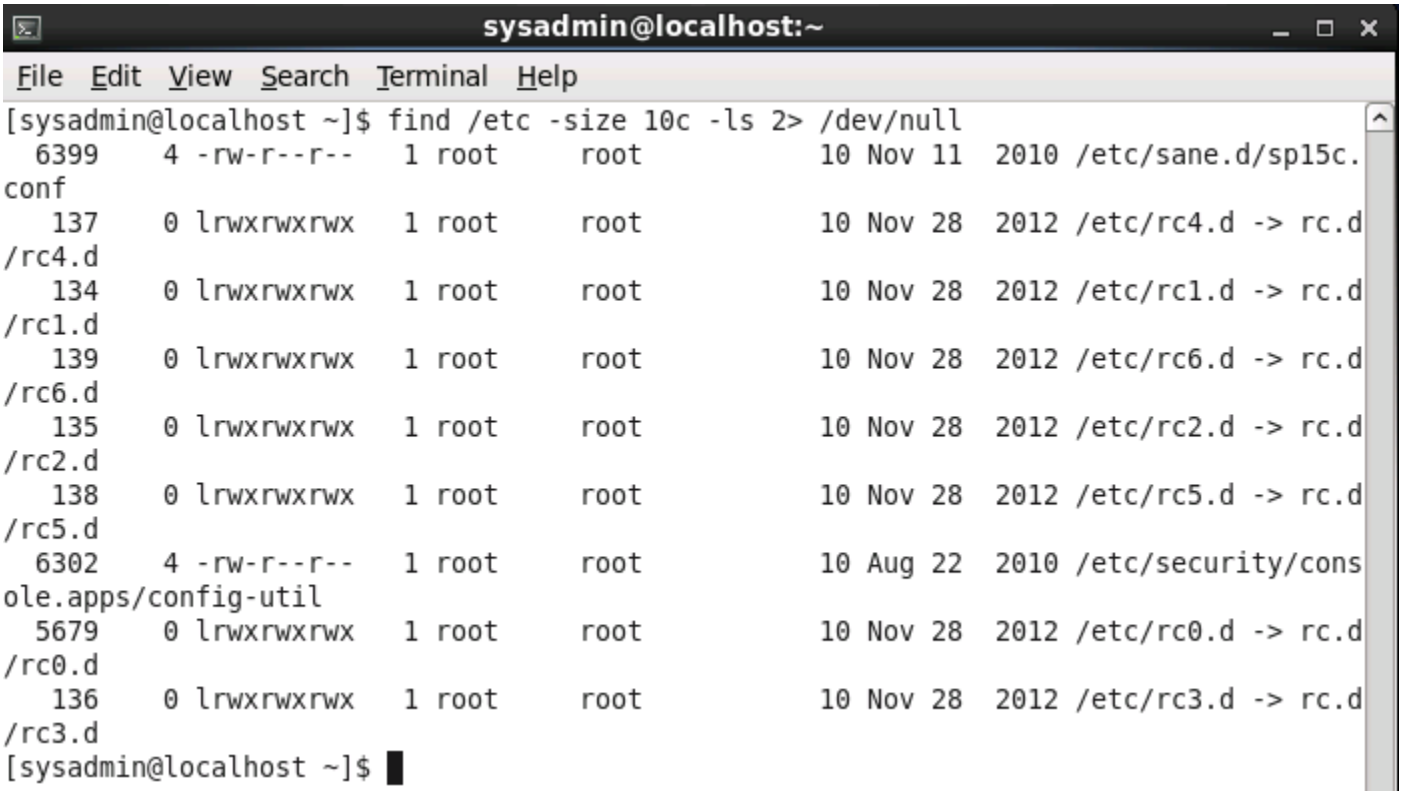


The first two columns of the output above are the *inode number* of the file and the number of *blocks* that the file is using for storage. Both of these are beyond the scope of the topic at hand. The rest of the columns are typical output of the `ls -l` command: file type, permissions, hard link count, user owner, group owner, file size, modification timestamp and file name.

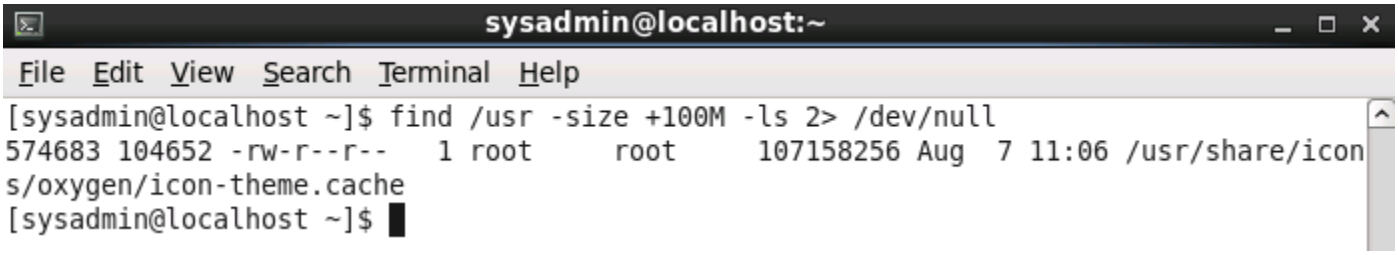
8.5.3 Searching for Files by Size

One of the many useful searching options is the option that allows you to search for files by size. The `-size` option allows you to search for files that are either larger than or smaller then a specified size as well as search for an exact file size.

When you specify a file size, you can give the size in bytes (c), kilobytes (k), megabytes (M) or gigabytes (G). For example, the following will search for files in the /etc directory structure that are exactly 10 bytes large:



If you want to search for files that are larger than a specified size, you place a + character before the size. For example, the following will look for all files in the `/usr` directory structure that are over 100 megabytes in size:



To search for files that are smaller than a specified size, place a - character before the file size.

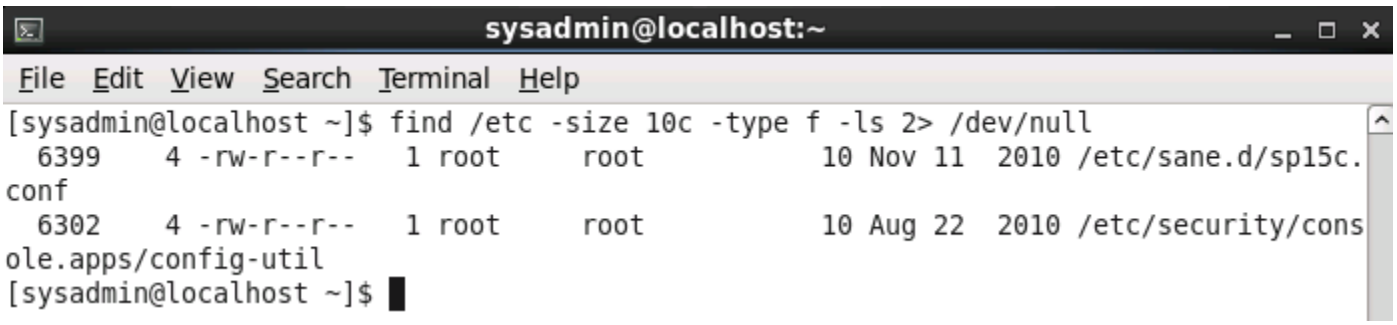
8.5.4 Additional Useful Search Options

There are many search options. The following table illustrates a few of these options:

Option	Meaning
-maxdepth	Allows the user to specify how deep in the directory structure to search. For example, -maxdepth 1 would mean only search the specified directory and its immediate subdirectories.
-group	Returns files owned by a specified group. For example, -group payroll would return files owned by the payroll group.
-iname	Returns files that match specified filename, but unlike -name, -iname is case insensitive. For example, -iname hosts would match files named hosts, Hosts, HOSTS, etc.
-mmin	Returns files that were modified based on modification time in minutes. For example, -mmin 10 would match files that were modified 10 minutes ago.
-type	Returns files that match file type. For example, -type f would return files that are regular files.
-user	Returns files owned by a specified user. For example, -user bob would return files owned by the bob user.

8.5.5 Using Multiple Options

If you use multiple options, they act as an "and", meaning for a match to occur, all of the criteria must match, not just one. For example, the following command will display all files in the `/etc` directory structure that are 10 bytes in size *and* are plain files:



8.6 Viewing Files Using the less Command

While viewing small files with the `cat` command poses no problems, it is not an ideal choice for large files. The `cat` command doesn't provide any way to easily pause and restart the display, so the entire file contents are dumped to the screen.

For larger files, you will want to use a *pager* command to view the contents. Pager commands will display one page of data at a time, allowing you to move forward and backwards in the file by using movement keys.

There are two commonly used pager commands:

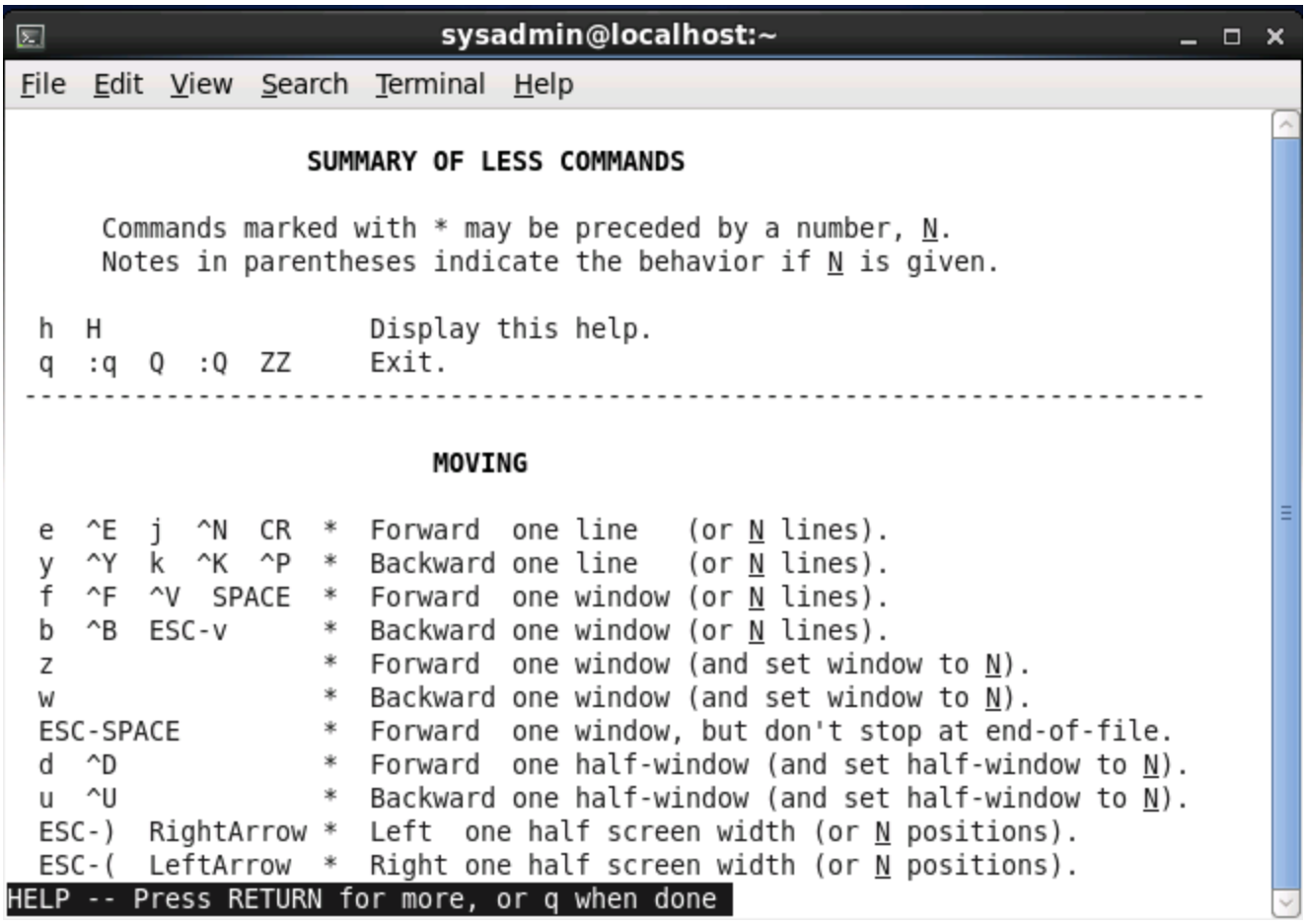
- The `less` command: This command provides a very advanced paging capability. It is normally the default pager used by commands like the man command.
- The `more` command: This command has been around since the early days of UNIX. While it has fewer features than the `less` command, it does have one important advantage: The `less` command isn't always included with all Linux distributions (and on some distributions, it isn't installed by default). The `more` command is always available.

When you use the `more` or `less` commands, they will allow you to "move around" a document by using *keystroke commands*. Because the developers of the `less` command based the command from the functionality of the `more` command, all of the keystroke commands available in the`more` command also work in the `less` command.

For the purpose of this manual, the focus will be on the more advanced command (`less`). The`more` command is still useful to remember for times when the `less` command isn't available. Remember that most of the keystroke commands provided work for both commands.

8.6.1 Help Screen in less

When you view a file with the `less` command, you can use the **h** key to display a help screen. The help screen allows you to see which other commands are available. In the following example, the `less /usr/share/dict/words` command is executed. Once the document is displayed, the**h** key was pressed, displaying the help screen:



8.6.2 less Movement Commands

There are many movement commands for the `less` command, each with multiple possible keys or key combinations. While this may seem intimidating, remember you don't need to memorize all of these movement commands; you can always use the **h** key whenever you need to get help.

The first group of movement commands that you may want to focus upon are the ones that are most commonly used. To make this even easier to learn, the keys that are identical in `more` and `less` will be summarized. In this way, you will be learning how to move in `more` and `less` at the same time:

Movement	Key
Window forward	Spacebar
Window backward	b

Movement	Key
Line forward	Enter
Exit	q
Help	h

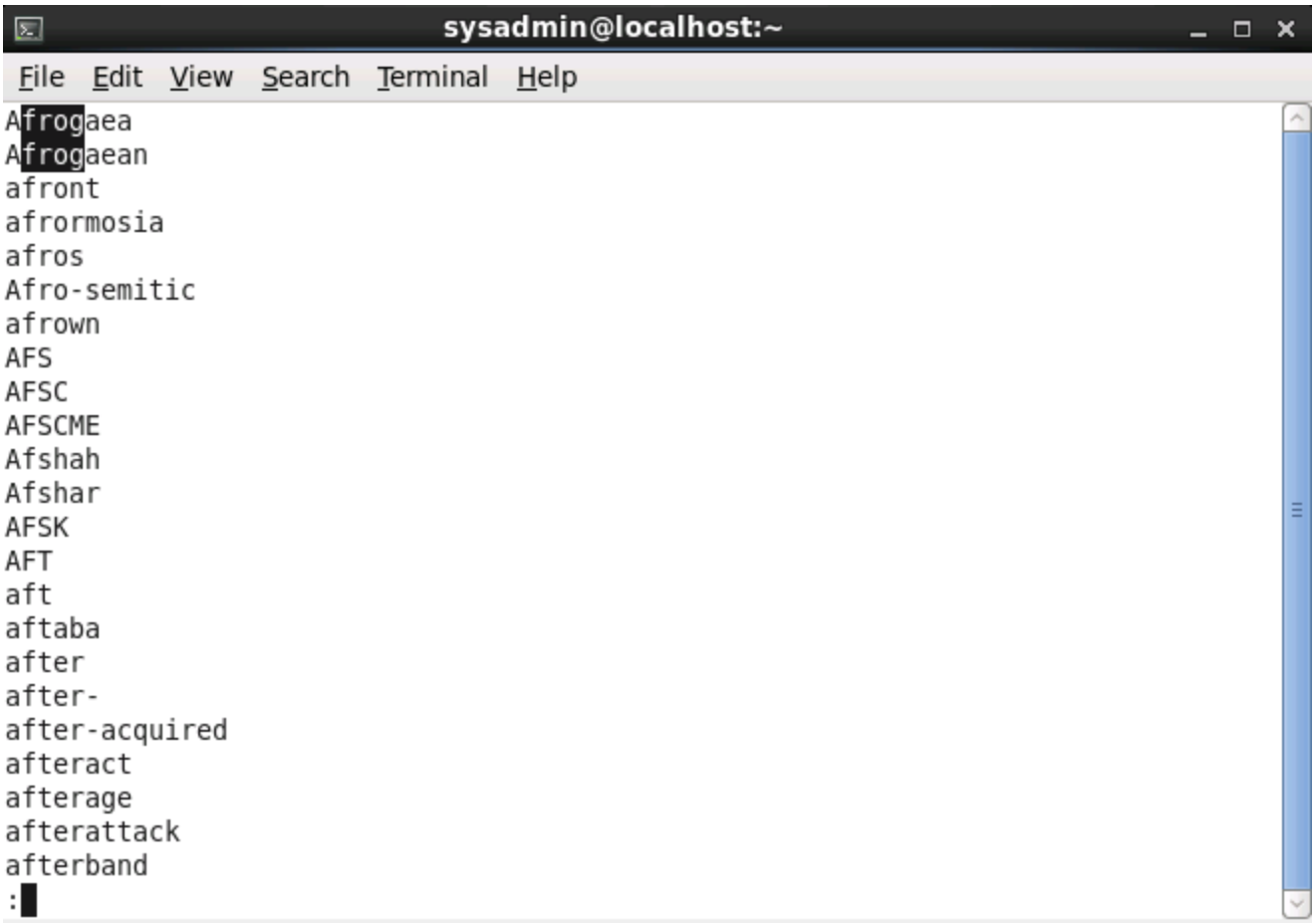
When simply using `less` as a pager, the easiest way to advance forward a page is to press the spacebar.

8.6.3 less Searching Commands

There are two ways to search in the `less` command: you can either search forward or backwards from your current position using patterns called regular expressions. More details regarding regular expressions are provided later in this chapter.

To start a search to look forward from your current position, use the **/** key. Then, type the text or pattern to match and press the **Enter** key.

If a match can be found, then your cursor will move in the document to the match. For example, in the following graphic the expression "frog" was searched for in the `/usr/share/dict/words` file:



Notice that "frog" didn't have to be a word by itself. Also notice that while the `less` command took you to the first match from the current position, all matches were highlighted.

If no matches forward from your current position can be found, then the last line of the screen will report "Pattern not found":

To start a search to look backwards from your current position, press the **?** key, then type the text or pattern to match and press the **Enter** key. Your cursor will move backward to the first match it can find or report that the pattern cannot be found.

If more than one match can be found by a search, then using the **n** key will allow you to move to the next match and using the **N** key will allow you to go to a previous match.

8.7 Revisiting the head and tail Commands

Recall that the `head` and `tail` commands are used to filter files to show a limited number of lines. If you want to view a select number of lines from the top of the file, you use the `head` command and if you want to view a select number of lines at the bottom of a file, then you use the `tail` command.

By default, both commands display ten lines from the file. The following table provides some examples:

Command Example	Explanation of Displayed Text
head /etc/passwd	First ten lines of /etc/passwd
head -3 /etc/group	First three lines of /etc/group
head -n 3 /etc/group	First three lines of /etc/group
help | head	First ten lines of output piped from the help command
tail /etc/group	Last ten lines of /etc/group
tail -5 /etc/passwd	Last five lines of /etc/passwd
tail -n 5 /etc/passwd	Last five lines of /etc/passwd
help | tail	Last ten lines of output piped from the help command

As seen from the above examples, both commands will output text from either a regular file or from the output of any command sent through a pipe. They both use the `-n` option to indicate how many lines to output.

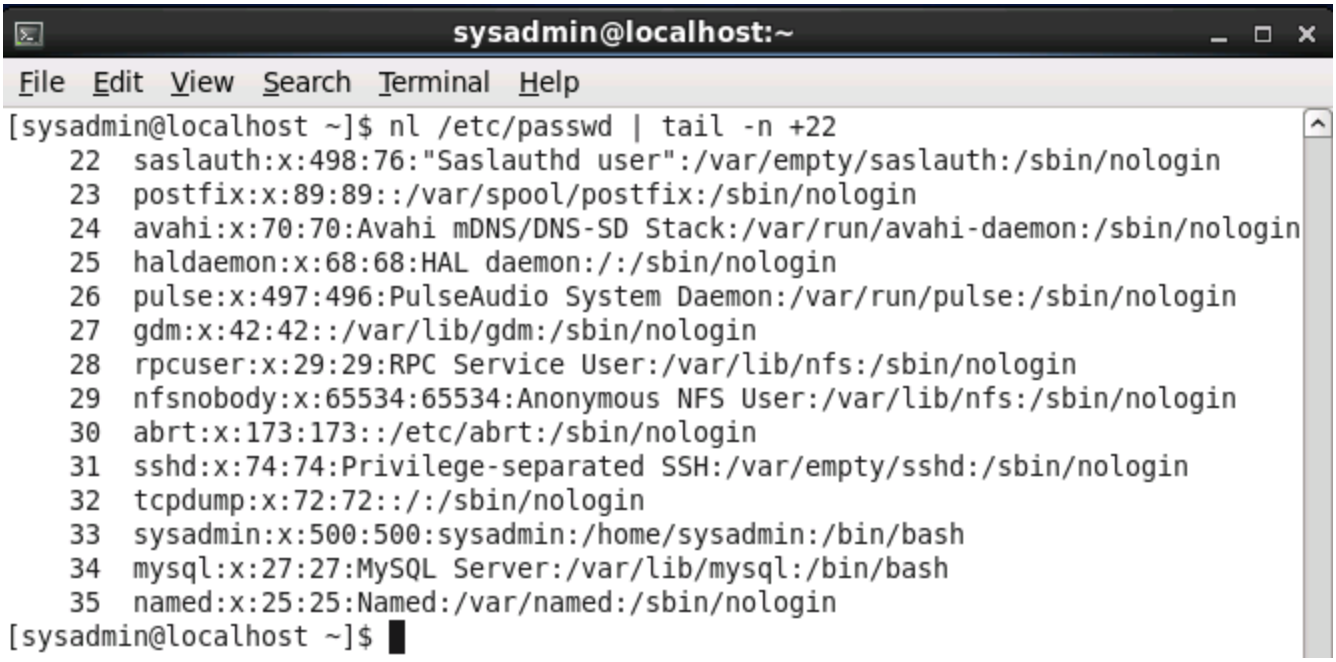
8.7.1 Negative Value with the -n Option

Traditionally in UNIX, the number of lines to output would be specified as an option with either command, so `-3` meant show three lines. For the `tail` command, either `-3` or `-n -3` still means show three lines. However, the GNU version of the `head` command recognizes `-n -3` as show **all but the first three lines**, and yet the `head` command still recognizes the option `-3` as show the first three lines.

8.7.2 Positive Value With the tail Command

The GNU version of the `tail` command allows for a variation of how to specify the number of lines to be printed. If you use the `-n` option with a number prefixed by the plus sign, then the `tail` command recognizes this to mean to display the contents starting at the specified line and continuing all the way to the end.

For example, the following will display line #22 to the end of the output of the `nl` command:



8.7.3 Following Changes to a File

You can view live file changes by using the `-f` option to the `tail` command. This is useful when you want to see changes to a file as they are happening.

A good example of this would be when viewing log files as a system administrator. Log files can be used to troubleshoot problems and administrators will often view them "interactively" with the `tail` command as they are performing the commands they are trying to troubleshoot in a separate window.

For example, if you were to log in as the root user, you could troubleshoot issues with the email server by viewing live changes to its log file with the following command: `tail -f /var/log/mail.log`

8.8 Sorting Files or Input

The `sort` command can be used to rearrange the lines of files or input in either dictionary or numeric order based upon the contents of one or more fields. Fields are determined by a field separator contained on each line, which defaults to whitespace (spaces and tabs).

The following example creates a small file, using the `head` command to grab the first 5 lines of the `/etc/passwd` file and send the output to a file called `mypasswd`.

```
sysadmin@localhost:~$ head -5 /etc/passwd > mypasswd
sysadmin@localhost:~$
```

```
sysadmin@localhost:~$ cat mypasswd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
sysadmin@localhost:~$
```

Now we will sort the `mypasswd` file:

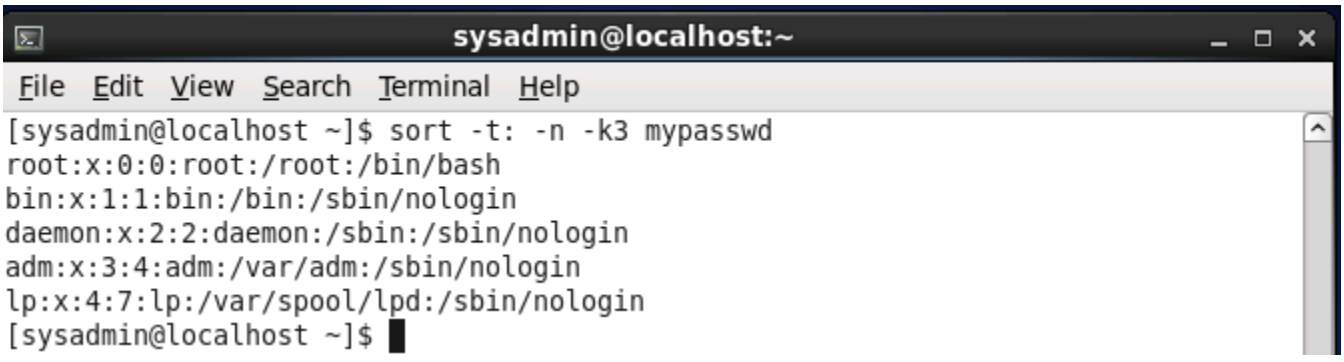
```
sysadmin@localhost:~$ sort mypasswd
bin:x:2:2:bin:/bin:/bin/sh
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
root:x:0:0:root:/root:/bin/bash
sync:x:4:65534:sync:/bin:/bin/sync
sys:x:3:3:sys:/dev:/bin/sh
sysadmin@localhost:~$
```

8.8.1 Fields and Sort Options

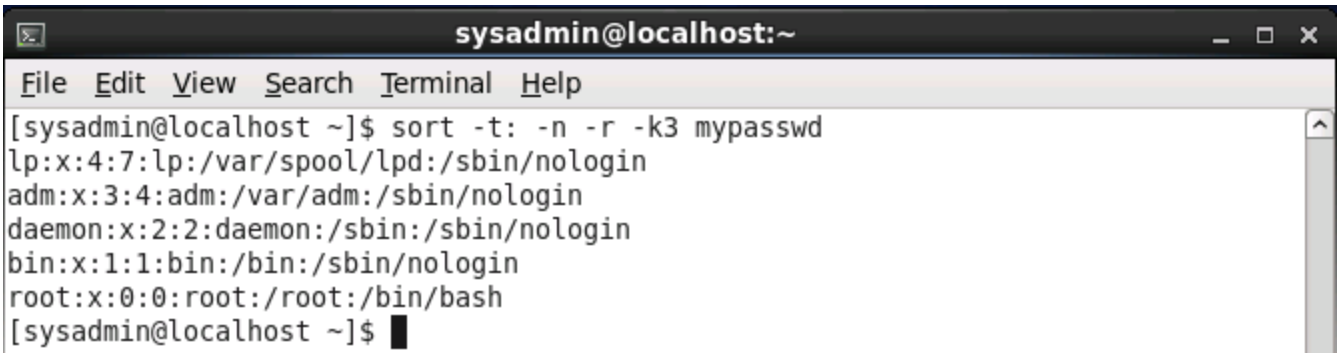
In the event that the file or input might be separated by another delimiter like a comma or colon, the `-t` option will allow for another field separator to be specified. To specify fields to sort by, use the `-k` option with an argument to indicate the field number (starting with 1 for the first field).

The other commonly used options for the `sort` command are the `-n` to perform a numeric sort and `-r` to perform a reverse sort.

In the next example, the `-t` option is used to separate fields by a colon character and performs a numeric sort using the third field of each line:



Note that the `-r` option could have been used to reverse the sort, making the higher numbers in the third field appear at the top of the output:



Lastly, you may want to perform more complex sorts, such as sort by a primary field and then by a secondary field. For example, consider the following data:

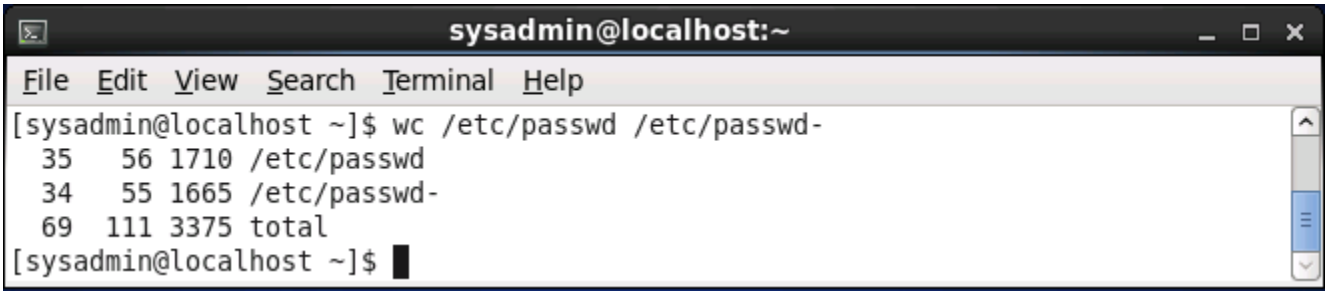
```
bob:smith:23
nick:jones:56
sue:smith:67
```

You might want to sort first by the last name (field #2) and then first name (field #1) and then by age (field #3). This can be done with the following command:

```
sort -t: -k2 -k1 -k3n filename
```

8.9 Viewing File Statistics With the wc Command

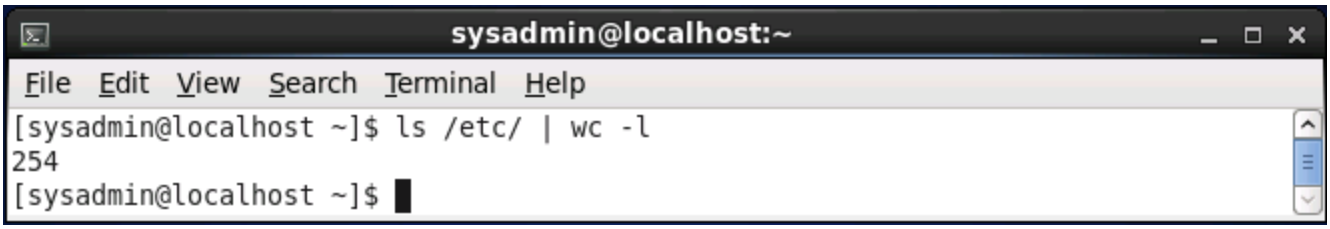
The `wc` command allows for up to three statistics to be printed for each file provided, as well as the total of these statistics if more than one filename is provided. By default, the wc command provides the number of lines, words and bytes (1 byte = 1 character in a text file):



The above example shows the output from executing: `wc /etc/passwd /etc/passwd-`. The output has four columns: number of lines in the file, number of words in the file, number of bytes in the file and the file name or "total".

If you are interested in viewing just specific statistics, then you can use `-l` to show just the number of lines, `-w` to show just the number of words and `-c` to show just the number of bytes.

The `wc` command can be useful for counting the number of lines output by some other command through a pipe. For example, if you wanted to know the total number of files in the `/etc` directory, you could execute `ls /etc | wc -l`:



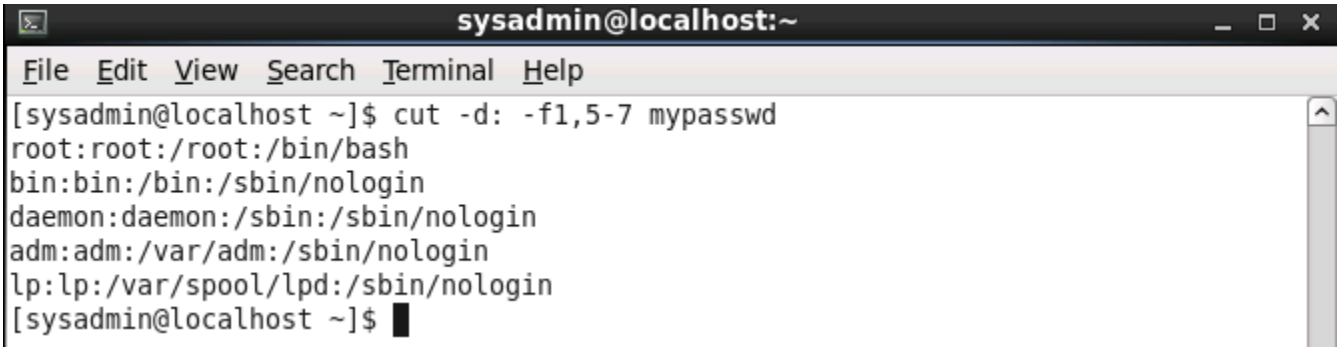
8.10 Using the cut Command to Filter File Contents

The `cut` command can extract columns of text from a file or standard input. A primary use of the `cut` command is for working with delimited database files. These files are very common on Linux systems.

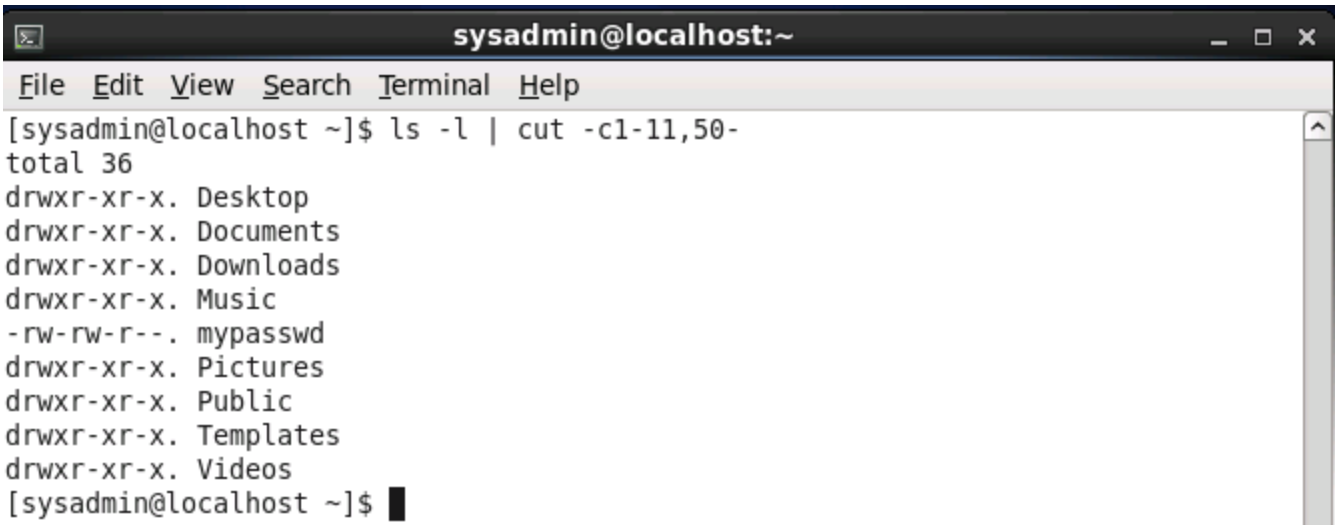
By default, it considers its input to be separated by the **Tab** character, but the `-d` option can specify alternative delimiters such as the colon or comma.

Using the `-f` option, you can specify which fields to display, either as a hyphenated range or a comma separated list.

In the following example, the first, fifth, sixth and seventh fields from `mypasswd` database file are displayed:



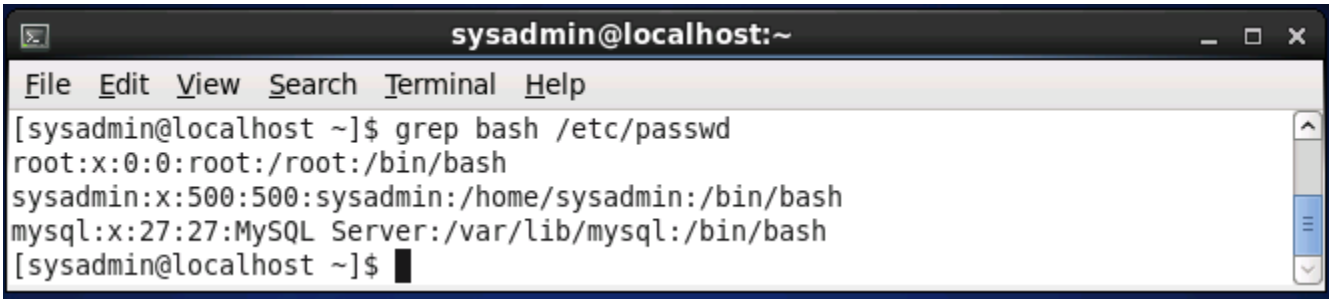
Using the `cut` command, you can also extract columns of text based upon character position with the `-c` option. This can be useful for extracting fields from fixed-width database files. For example, the following will display just the file type (character #1), permissions (characters #2-10) and filename (characters #50+) of the output of the `ls -l` command:



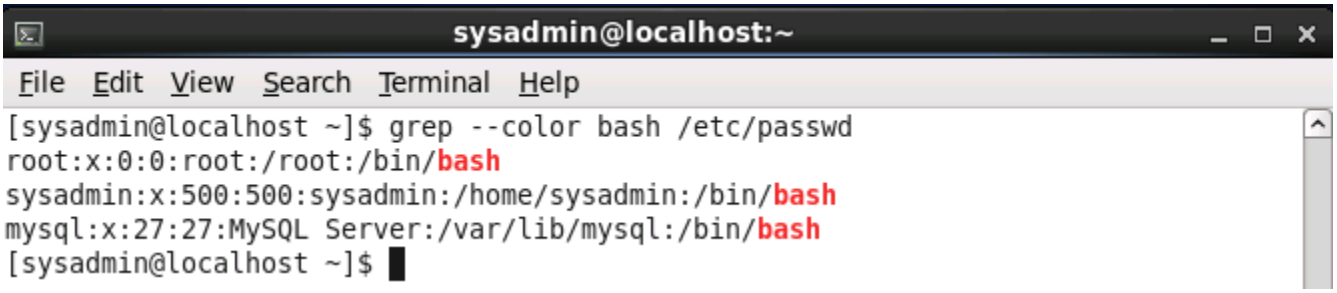
8.11 Using the grep Command to Filter File Contents

The `grep` command can be used to filter lines in a file or the output of another command based on matching a pattern. That pattern can be as simple as the exact text that you want to match or it can be much more advanced through the use of regular expressions (discussed later in this chapter).

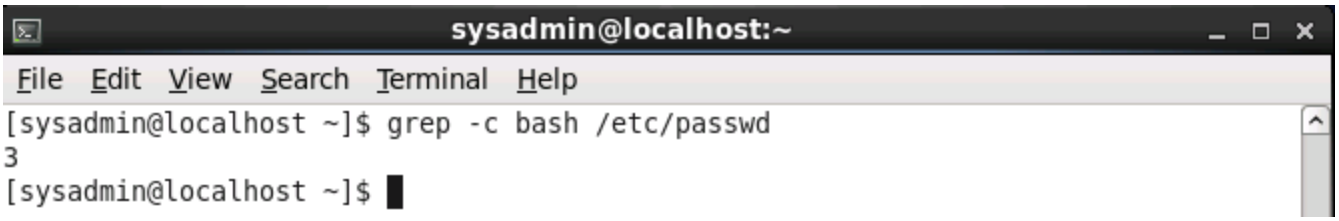
For example, you may want to find all the users who can login to the system with the BASH shell, so you could use the `grep` command to filter the lines from the `/etc/passwd` file for the lines containing the characters "bash":



To make it easier to see what exactly is matched, use the `--color` option. This option will highlight the matched items in red:

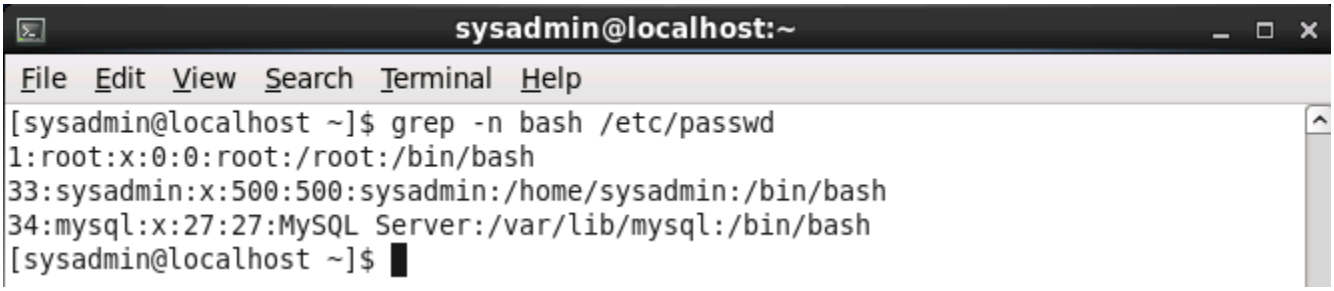


In some cases you don't care about the specific lines that match the pattern, but rather how many lines match the pattern. With the `-c` option, you can get a count of how many lines that match:



When you are viewing the output from the `grep` command, it can be hard to determine the original line numbers. This information can be useful when you go back into the file (perhaps to edit the file) as you can use this information to quickly find one of the matched lines.

The `-n` option to the `grep` command will display original line numbers:



Some additional useful `grep` options:

Examples	Output
grep -v nologin /etc/passwd	All lines not containing "nologin" in the /etc/passwd file
grep -l linux /etc/*	List of files in the /etc directory containing "linux"
grep -i linux /etc/*	Listing of lines from files in the /etc directory containing any case (capital or lower) of the character pattern "linux"
grep -w linux /etc/*	Listing of lines from files in the /etc directory containing the word pattern "linux"

8.12 Basic Regular Expressions

A *Regular Expression* is a collection of "normal" and "special" characters that are used to match simple or complex patterns. Normal characters are alphanumeric characters which match themselves. For example, an "a" would match an "a".

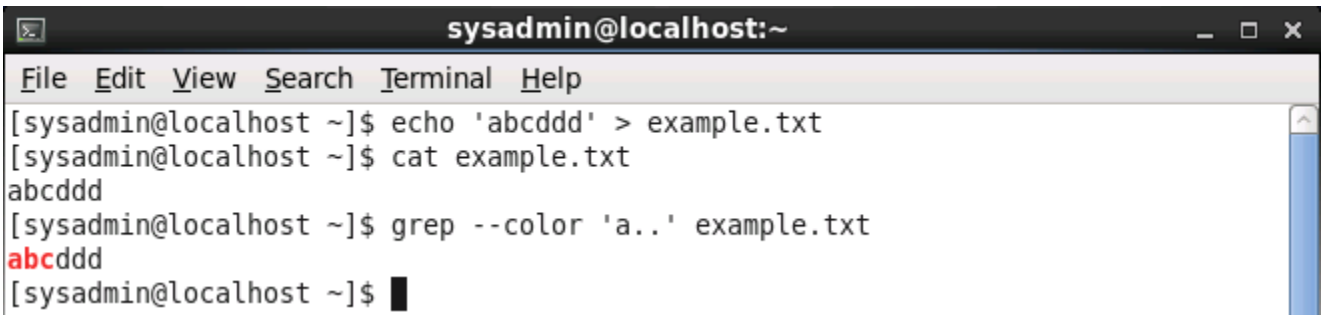
Some characters have special meanings when used within patterns by commands like the `grep` command. There are both *Basic Regular Expressions* (available to a wide variety of Linux commands) and *Extended Regular Expressions* (available to more advanced Linux commands). Basic Regular Expressions include the following:

Regular Expression	Matches
.	Any single character
[ ]	A list or range of characters to match one character, unless the first character is the caret "^", and then it means any character not in the list
*	Previous character repeated zero or more times
^	Following text must appear at beginning of line
$	Preceding text must appear at the end of the line

The `grep` command is just one of many commands that support regular expressions. Some other commands include the `more` and `less` commands. While some of the regular expressions are unnecessarily quoted with single quotes, it is a good practice to use single quotes around your regular expressions to prevent the shell from trying to interpret special meaning from them.

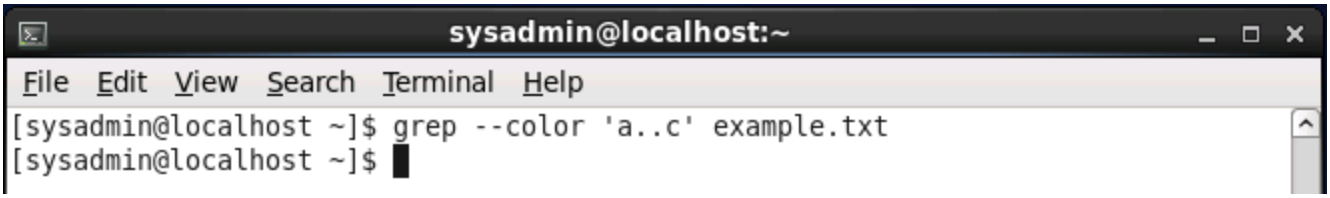
8.12.1 Basic Regular Expressions - the . Character

In the example below, a simple file is first created using redirection. Then the `grep` command is used to demonstrate a simple pattern match:



In the previous example, you can see that the pattern "a.." matched "abc". The first . character matched the "b" and the second matched the "c".

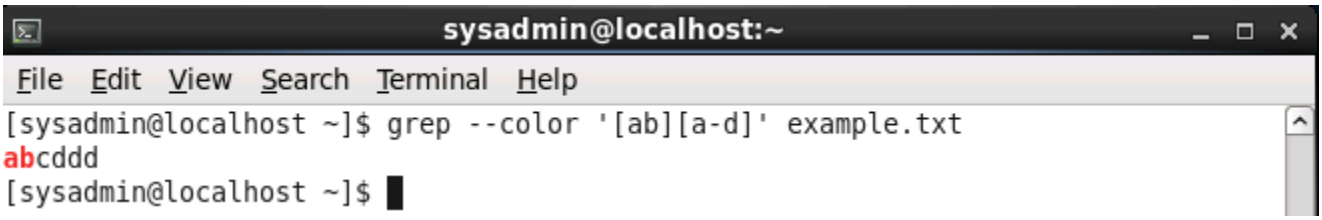
In the next example, the pattern "a..c" won't match anything, so the `grep` command will not product any output. For the match to be successful, there would need to be two characters between the "a" and the "c":



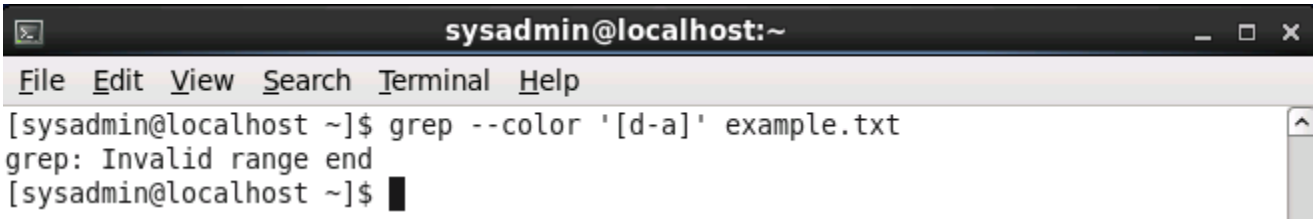
8.12.2 Basic Regular Expressions - the [ ] Characters

If you use the . character, then any possible character could match. In some cases you want to specify exactly which characters you want to match. For example, maybe you just want to match a lower-case alpha character or a number character. For this, you can use the [ ] Regular Expression characters and specify the valid characters inside the [ ] characters.

For example, the following command matches two characters, the first is either an "a" or a "b" while the second is either an "a", "b", "c" or "d":



Note that you can either list out each possible character ([abcd]) or provide a range ([a-d]) as long as the range is in the correct order. For example, [d-a] wouldn't work because it isn't a valid range:



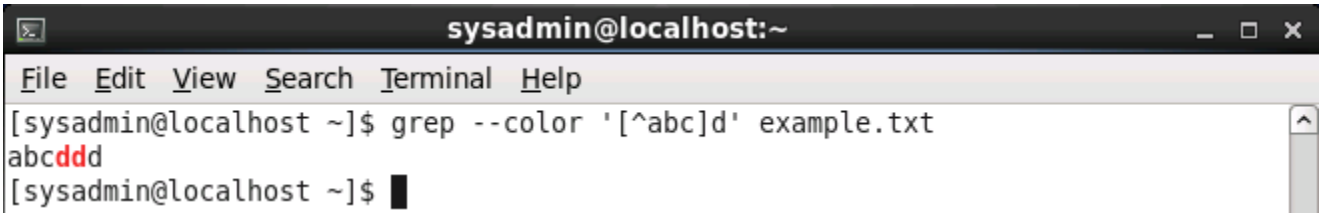
The range is specified by a standard called the ASCII table. This table is a collection of all printable characters in a specific order. You can see the ASCII table with the `man ascii` command. A small example:

```
041   33    21    !                    141   97    61    a
042   34    22    "                    142   98    62    b
043   35    23    #                    143   99    63    c
044   36    24    $                    144   100   64    d
045   37    25    %                    145   101   65    e
046   38    26    &                    146   102   66    f
```

Since "a" has a smaller numeric value (141) then "d" (144), the range a-d includes all characters from "a" to "d".

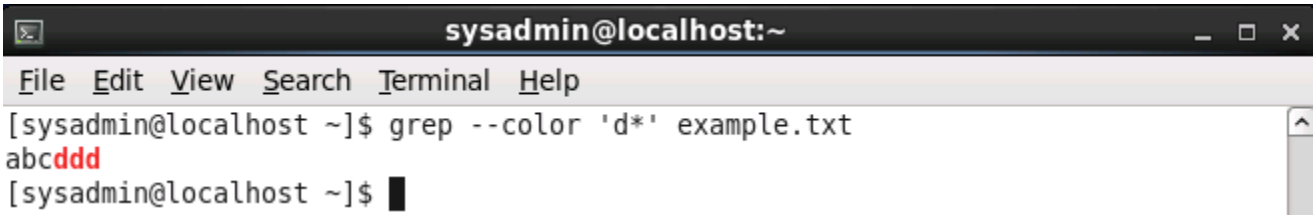
What if you want to match a character that can be anything but an "x", "y" or "z"? You wouldn't want to have to provide a [ ] set with all of the characters except "x", "y" or "z".

To indicate that you want to match a character that is not one of the listed characters, start your [ ] set with a ^ symbol. For example, the following will demonstrate matching a pattern that includes a character that isn't an "a", "b" or "c" followed by a "d":



8.12.3 Basic Regular Expressions - the * Character

The * character can be used to match "zero or more of the previous character". For example, the following will match zero or more "d" characters:



8.12.4 Basic Regular Expressions - the ^ and $ Characters

When you perform a pattern match, the match could occur anywhere on the line. You may want to specify that the match occurs at the beginning of the line or the end of the line. To match at the beginning of the line, begin the pattern with a ^ symbol.

In the following example, another line is added to the example.txt file to demonstrate the use of the ^ symbol:

```
sysadmin@localhost:~$ echo "xyzabc" >> example.txt
sysadmin@localhost:~$ cat example.txt
abcddd
xyzabc
sysadmin@localhost:~$ grep --color "a" example.txt
abcddd
xyzabc
sysadmin@localhost:~$ grep --color "^a" example.txt
abcddd
sysadmin@localhost:~$
```

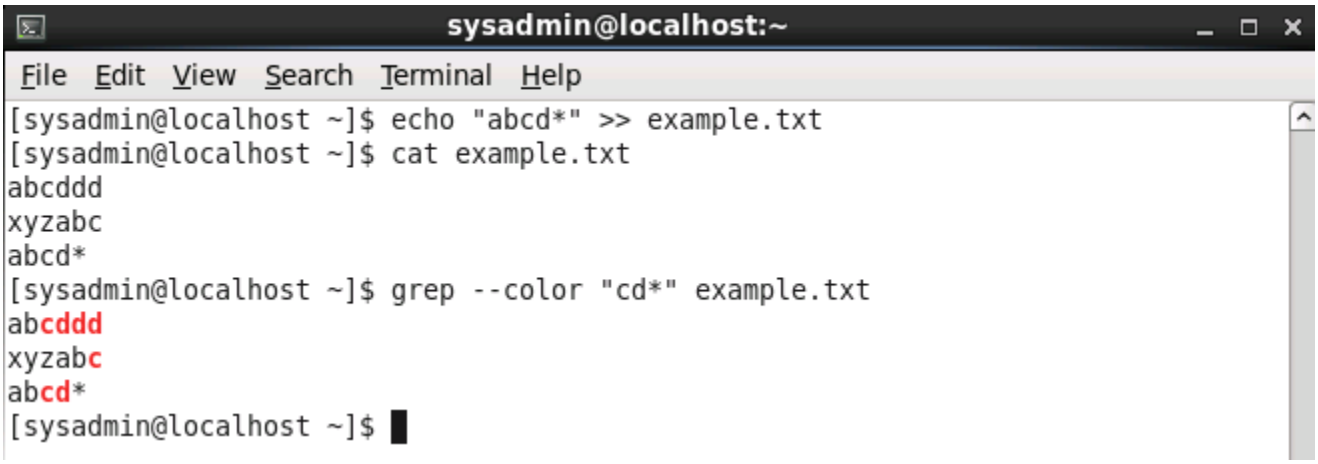
Note that in the first grep output, both lines match because they both contain the letter "a". In the second grep output, only the line that began with the letter "a" matched.

In order to specify the match occurs at the end of line, end the pattern with the $ character. For example, in order to only find lines which end with the letter "c":

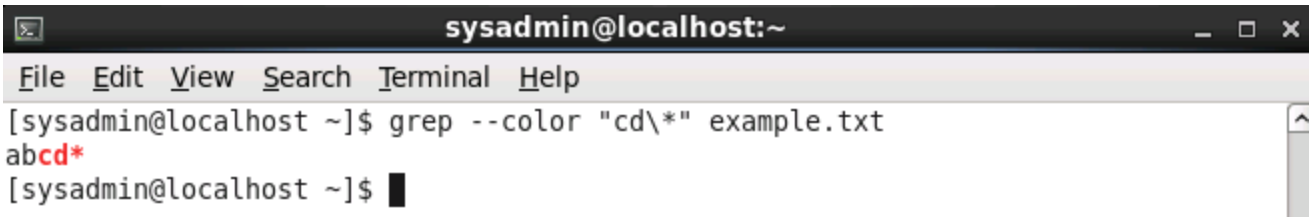
```
sysadmin@localhost:~$ grep "c$" example.txt
xyzabc
sysadmin@localhost:~$
```

8.12.5 Basic Regular Expressions - the \ Character

In some cases you may want to match a character that happens to be a special Regular Expression character. For example, consider the following:



In the output of the `grep` command above, you will see that every line matches because you are looking for a 'c' character followed by zero or more 'd' characters". If you want to look for an actual * character, place a \ character before the * character:



8.13 Extended Regular Expressions

The use of Extended Regular Expressions often requires a special option be provided to the command to recognize them. Historically, there is a command called `egrep`, which is similar to grep, but is able to understand their usage. Now, the `egrep` command is deprecated in favor of using grep with the `-E` option.

The following regular expressions are considered "extended":

RE	Meaning
?	Matches previous character zero or one time, so it is an optional character
+	Matches previous character repeated one or more times
|	Alternation or like a logical or operator

Some extended regular expressions examples:

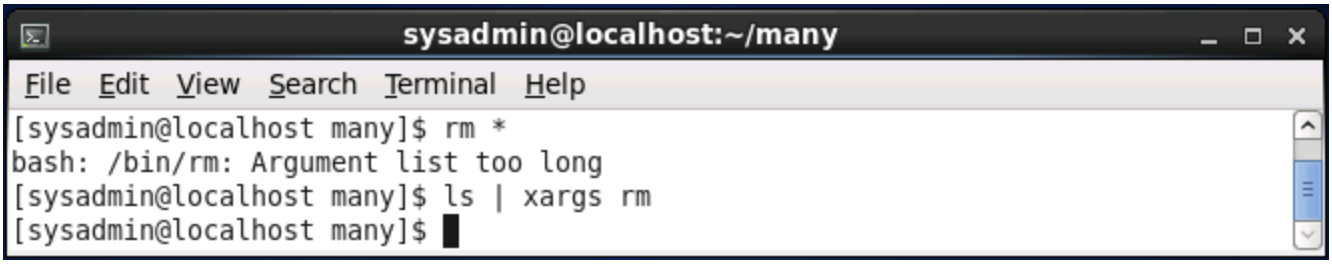
Command	Meaning	Matches
grep -E 'colou?r' 2.txt	Match 'colo' following by zero or one 'u' character	color colour
grep -E 'd+' 2.txt	Match one or more 'd' characters	d dd ddd dddd
grep -E 'gray|grey' 2.txt	Match either 'gray' or 'grey'	gray grey

8.14 xargs Command

If you receive an error about an argument list being too long when trying to execute a command, then it's probably time to think about using `xargs` with that command. The `xargs` command also has a useful option `-0` that helps to eliminate problems with files that have spaces or tabs in their names.

The `xargs` command is useful for allowing commands to be executed more efficiently. Its goal is to build the command line for a command to execute as few as times as possible with as many arguments as possible, rather than to execute the command many times with one argument each time.

The following example shows a scenario where the `xargs` command allowed for many files to be removed, where using a normal wildcard (glob) character failed:



Quoting in Linux

There are three types of quotes used by the Bash shell: single quotes ('), double quotes (") and back quotes (`). These quotes have special features in the Bash shell as described below.

To understand single and double quotes, consider that there are times that you don't want the shell to treat some characters as *special*. For example, the * character is used as a *wildcard*. What if you wanted the * character to just mean a literal asterisk?

- Single ' quotes prevent the shell from "interpreting" or expanding all special characters. Often single quotes are used to protect a string (a sequence of characters) from being changed by the shell, so that the string can be interpreted by a command as a parameter to affect the way the command is executed.
- Double " quotes stop the expansion of glob characters like the asterisk (*), question mark (?), and square brackets ([]). Double quotes *do* allow for both variable expansion and command substitution (see back quotes) to take place.
- Back ` quotes cause *command substitution* which allows for a command to be executed within the line of another command.

When using quotes, they must be entered in pairs or else the shell will not consider the command complete.

While single quotes are useful for blocking the shell from interpreting one or more characters, the shell also provides a way to block the interpretation of just a single character called "escaping" the character. To *escape* the special meaning of a shell metacharacter, the backslash \ character is used as a prefix to that one character.

Execute the following command to use back quotes ` (found under the ~ character on some keyboards) to execute the date command within the line of the echo command:

```
echo Today is `date`
```

Your output should be similar to the following:

```
sysadmin@localhost:~$ echo Today is `date`
Today is Mon Dec 3 21:29:45 UTC 2018
```

You can also place $(before the command and) after the command to accomplish command substitution:

```
echo Today is $(date)
```

Your output should be similar to the following:

```
sysadmin@localhost:~$ echo Today is $(date)
Today is Mon Dec 3 21:33:41 UTC 2018
```

Why two different methods that accomplish the same thing? Backquotes look very similar to single quotes, making it harder to "see" what a command is supposed to do. Originally, shells used backquotes; the `$(command)` format was added in a later version of the Bash shell to make the statement more visually clear.

If you don't want the backquotes to be used to execute a command, place single quotes around them. Execute the following:

```
echo This is the command '`date`'
```

Your output should be similar to the following:

```
sysadmin@localhost:~$ echo This is the command ''`date`''
This is the command `date`
sysadmin@localhost:~$
```

Note that you could also place a backslash character in front of each backquote character. Execute the following:

```
echo This is the command \`date\`
```

Your output should be similar to the following:

```
sysadmin@localhost:~$ echo This is the command \`date\`
This is the command `date`
sysadmin@localhost:~$
```

Double quote characters don't have any effect on backquote characters. The shell will still use them as command substitution. Execute the following to see a demonstration:

```
echo This is the command "`date`"
```

Your output should be similar to the following:

```
sysadmin@localhost:~$ echo This is the command "`date`"
This is the command Mon Dec  3 21:37:33 UTC 2018
```

5.6.6 Step 6

Double quote characters will have an effect on wildcard characters, disabling their special meaning. Execute the following:

```
echo D*
echo "D*"
```

Your output should be similar to the following:

```
sysadmin@localhost:~$ echo D*
Desktop Documents Downloads
sysadmin@localhost:~$ echo "D*"
D*
sysadmin@localhost:~$
```

Important

Quoting may seem trivial and weird at the moment, but as you gain more experience working in the command shell, you will discover that having a good understanding of how different quotes work is critical to using the shell.

5.7 Control Statements

Typically, you type a single command and you execute it when you press **Enter**. The Bash shell offers three different statements that can be used to separate multiple commands typed together.

The simplest separator is the semicolon (`;`). Using the semicolon between multiple commands allows for them to be executed one right after another, sequentially from left to right.

The `&&` characters create a logical "and" statement. Commands separated by && are conditionally executed. If the command on the left of the `&&` is successful, then the command to the right of the `&&` will also be executed. If the command to the left of the `&&` fails, then the command to the right of the `&&` is not executed.

The `||` characters create a logical "or" statement, which also causes conditional execution. When commands are separated by `||`, then only if the command to the left fails, does the command to the right of the `||` execute. If the command to the left of the `||` succeeds, then the command to the right of the `||` will not execute.

To see how these control statements work, you will be using two special executables: `true` and `false`. The `true` executable always succeeds when it executes, whereas, the `false` executable always fails. While this may not provide you with realistic examples of how `&&` and `||` work, it does provide a means to demonstrate how they work without having to introduce new commands.

5.7.1 Step 1

Execute the following three commands together separated by semicolons:

```
echo Hello; echo Linux; echo Student
```

As you can see the output shows all three commands executed sequentially:

```
sysadmin@localhost:~$ echo Hello; echo Linux; echo Student
Hello
Linux
Student
sysadmin@localhost:~$
```

5.7.2 Step 2

Now, put three commands together separated by semicolons, where the first command executes with a failure result:

```
false; echo Not; echo Conditional
```

Your output should be similar to the following:

```
sysadmin@localhost:~$ false; echo Not; echo Conditional
Not
Conditional
sysadmin@localhost:~$
```

Note that in the previous example, all three commands still executed even though the first one failed. While you can't see from the output of the `false` command, it did execute. However, when commands are separated by the `;` character, they are completely independent of each other.

5.7.3 Step 3

Next, use logical "and" to separate the commands:

```
echo Start && echo Going && echo Gone
```

Your output should be similar to the following:

```
sysadmin@localhost:~$ echo Start && echo Going && echo Gone
Start
Going
Gone
sysadmin@localhost:~$
```

Because each `echo` statement executes correctly, a return value of success is provided, allowing the next statement to also be executed.

5.7.4 Step 4

Use logical "and" with a command that fails as shown below:

```
echo Success && false && echo Bye
```

Your output should be similar to the following:

```
sysadmin@localhost:~$ echo Success && false && echo Bye
Success
sysadmin@localhost:~$
```

The first `echo` command succeeds and we see its output. The `false` command executes with a failure result, so the last `echo` statement is not executed.

5.7.5 Step 5

The "or" characters separating the following commands demonstrates how the failure before the "or" statement causes the command after it to execute; however, a successful first statement causes the command to not execute:

```
false || echo Fail Or
true || echo Nothing to see here
```

Your output should be similar to the following:

```
sysadmin@localhost:~$ false || echo Fail Or
Fail Or
sysadmin@localhost:~$ true || echo Nothing to see here
sysadmin@localhost:~$
```

9.1 Introduction

The premier text editor for Linux and UNIX is a program called `vi`. While there are numerous editors available for Linux that range from the tiny editor `nano` to massive `emacs` editor, there are several advantages to the `vi` editor:

- The `vi` editor is available on every Linux distribution in the world. This is not true of any other editor.
- The `vi` editor can be executed both in a CLI interface and a GUI interface. While graphical editors, like `gedit` from the Gnome desktop environment or `kedit` from K desktop environment, are easier to use, they require a GUI, which servers won't always have running.
- While new features have been added to the `vi` editor, the core functions have been around for decades. This means if someone learned the `vi`editor in the 1970s, they could use a modern version without any problem. While that seems trivial, it may not seem to so trivial twenty years from now.

Consider This

The correct way to pronounce "the vi editor" is "the vee-eye editor". The letters vi stand for "**vi**sual", but it was never pronounced "vi" by the developers, but rather the letter "v" followed by the letter "i".

The original `vi` editor was written by Bill Joy, the co-founder of Sun Microsystems. Since `vi` is part of the Single UNIX Specification (SUS), it is required that conforming UNIX-based systems have it. Since the Linux Standards Base (LSB) mirrors the requirements of SUS, Linux systems that conform to LSB must also include the `vi` editor.

In reality, most Linux systems don't include the original `vi`, but an improved version of it known as `vim`, for **vi im**proved. This fact may be hidden by most Linux distributions. On some the `vi` file will link to `vim`, while on others an alias exists that will execute `vim` when the `vi` command is run:

```
[sysadmin@localhost ~]$ which vi
alias vi='vim'
        /usr/bin/vim
```

For the most part, `vim` works just like `vi`, but has additional features. For the topics that will be covered in this course, either `vi` or `vim` will work.

To get started using `vi`, simply type the command followed by the pathname to the file to edit or create:

```
sysadmin@localhost:~$ vi newfile
```

9.2 Command Mode Movement

There are three modes used in `vi`: command mode, insert mode, and ex mode. Initially, the program starts in command mode. Command mode is used to type commands, such as those used to move around a document, manipulate text, and access the other two modes. To return to command mode at any time, press the **ESC** key.

Once some text has been added into a document, to perform actions like moving the cursor, the **ESC** key needs to be pressed first to return to command mode. This seems like a lot of work, but remember that `vi` works in a terminal environment where a mouse is useless.

Movement commands in `vi` have two aspects, a motion and an optional number prefix, which indicates how many times to repeat that motion. The general format is as follows:

```
[count] motion
```

The following table summarizes the motion keys available:

Motion	Result
`h`	Left one character
`j`	Down one line
`k`	Up one line
`l`	Right one character
`w`	One word forward
`b`	One word back
`^`	Beginning of line
`$`	End of the line

Note: Since the upgrade to `vim` it is also possible to use the arrow keys `←↓↑→` instead of `hjkl` respectively.

These motions can be prefixed with a number to indicate how many times to perform the movement. For example, `5h` would move the cursor five characters to the left,`3w` would move the cursor three words to the right.

To move the cursor to a specific line number, type that line number followed by the `G` character. For example, to get to the fifth line of the file type `5G`. `1G` or `gg` can be used to go to the first line of the file, while a lone `G` will take you to the last line. To find out which line the cursor is currently on, use **CTRL-G**.

9.3 Command Mode Actions

The standard convention for editing content with word processors is to use copy, cut, and paste. The `vi` program has none of these, instead `vi` uses the following three commands:

Standard	Vi	Meaning
cut	`d`	delete
copy	`y`	yank

Standard	Vi	Meaning
paste	`P` | `p`	put

The motions learned from the previous page are used to specify where the action is to take place, always beginning with the present cursor location. Either of the following general formats for action commands is acceptable:

```
action [count] motion
[count] action motion
```

Delete

Delete removes the indicated text from the page and saves it into the buffer, the buffer being the equivalent of the "clipboard" used in Windows or Mac OSX. The following table provides some common usage examples:

Action	Result
`dd`	Delete current line
`3dd`	Delete the next three lines
`dw`	Delete the current word
`d3w`	Delete the next three words
`d4h`	Delete four characters to the left

Change

Change is very similar to delete, the text is removed and saved into the buffer, however the program is switched to insert mode to allow immediate changes to the text. The following table provides some common usage examples:

Action	Result
`cc`	Change current line
`cw`	Change current word

Action	Result
`c3w`	Change the next three words
`c5h`	Change five characters to the left

Yank

Yank places content into the buffer without deleting it. The following table provides some common usage examples:

Action	Result
`yy`	Yank current line
`3yy`	Yank the next three lines
`yw`	Yank the current word
`y$`	Yank to the end of the line

Put

Put places the text saved in the buffer either before or after the cursor position. Notice that these are the only two options, put does not use the motions like the previous action commands.

Action	Result
`p`	Put (paste) after cursor
`P`	Put before cursor

Searching in vi

Another standard function that word processors offer is find. Often, people use **CTRL+F** or look under the edit menu. The `vi` program uses search. Search is more powerful than find because it supports both literal text patterns and regular expressions.

To search forward from the current position of the cursor, use the `/` to start the search, type a search term, and then press the **Enter** key to begin the search. The cursor will move to the first match that is found.

To proceed to the next match using the same pattern, press the `n` key. To go back to a previous match, press the `N` key. If the end or the beginning of the document is reached, it will automatically wrap around to the other side of the document.

To start searching backwards from the cursor position, start by typing `?`, then type the pattern to search for matches and press the **Enter** key.

9.4 Insert Mode

Insert mode is used to add text to the document. There a few ways to enter insert mode from command mode, each differing by where the text insertion will begin. The following table covers the most common:

Input	Purpose
`a`	Enter insert mode right after the cursor
`A`	Enter insert mode at the end of the line
`i`	Enter insert mode right before the cursor
`I`	Enter insert mode at the beginning of the line
`o`	Enter insert mode on a blank line after the cursor
`O`	Enter insert mode on a blank line before the cursor

9.5 Ex Mode

Originally, the `vi` editor was called the `ex` editor. The name `vi` was the abbreviation of the **visual** command in the `ex` editor that switched the editor to "visual" mode.

In the original normal mode, the `ex` editor only allowed users to see and modify one line at a time. In the visual mode, users could see as much of the document that will fit on the screen. Since most users preferred the visual mode to the line editing mode, the `ex` program file was linked to a `vi` file, so that users could start `ex` directly in visual mode when they ran the `vi` link.

Eventually, the actual program file was renamed `vi` and the `ex` editor became a link that pointed the `vi` editor.

When the ex mode of the `vi` editor is being used, it is possible to view or change settings, as well as carry out file-related commands like opening, saving or aborting changes to a file. In order to get to the ex mode, type a `:` character in command mode. The following table lists some common actions performed in ex mode:

Input	Purpose

Input	Purpose
`:w`	Write the current file to the filesystem
`:w filename`	Save a copy of the current file as filename
`:w!`	Force writing to the current file
`:1`	Go to line number 1 or whatever number is given
`:e filename`	Open filename
`:q`	Quit if no changes made to file
`:q!`	Quit without saving changes to file

A quick analysis of the table above reveals if an exclamation mark `!` is added to a command, then attempts to force the operation. For example, imagine you make changes to a file in the `vi` editor and then try to quit with `:q`, only to discover that the "quit" command fails. The `vi` editor doesn't want to quit without saving the changes you made to a file, but you can force it to quit with the ex command `:q!`.

Consider This

While it may seem impossible, the `vi` editor can save changes to a read-only file. The command `:w!` will try to write to a file, even if it is read-only, by attempting to change the permissions on the file, perform the write to the file and then change the permissions back to what they were originally.

This means that the root user can make changes to almost any file in the `vi` editor, regardless of the permissions on the file. However, ordinary users will only be able to force writing to the files that they own. Using `vi` doesn't change the fact that regular users can't modify the permissions on file that they do not own.

Although the ex mode offers several ways to save and quit, there's also `ZZ` that is available in command mode; this is the equivalent of `:wq`. There are many more overlapping functions between ex mode and command mode. For example, ex mode can be used to navigate to any line in the document by typing `:` followed by the line number, while the `G` can be used in command mode as previously demonstrated.

Chapter 9: The vi Editor

This chapter will cover the following exam objectives:

103.8: Perform basic file editing operations using vi

Weight: 3

Description: Candidates should be able to edit text files using vi. This objective includes vi navigation, basic vi modes, inserting, editing, deleting, copying and finding text.

Key Knowledge Areas:

- Navigate a document using vi
  Section 9.2
- Use basic vi modes
  Section 9.2
- Insert, edit, delete, copy and find text
  Section 9.4

Chapter 9: The vi Editor

/, ?

This is used to search for text while in command mode. the / is used to start searching. Enter a key term and press enter to begin searching the file for the text entered. If the user would like to search backwards in the document, a ? can be used instead of the /.
Section 9.3

ZZ, :w!, :q!, :e!

These keys are used to exit the vi editor from command mode. ZZ is used to save and quit the file. It must be done for each file. :e! is used to restor the original file allow the user to start over. :w! will force the writing of the current file. :q! will exit the editor without saving changes to the current file.
Section 9.5

c, d, p, y, dd, yy

These are used to cut, copy, replace and paste text when in command mode. c is used to change a line from the the current cursor location to the end of the line with whatever the user types. d is used to cut one alphabetic word, where as dd is used to cut an entire line of text. y is used to copy one one alphabetic word, where as yy is used to copy and entire line at a time. If a number preceeds either dd or yy, this will copy that number of lines. For example if 3dd is typed this will cut 3 lines at a time.
Section 9.3

h, j, k, l

These keys are used for basic cursor movement in vi when in command mode. h moves left one character, j moves down one line, k moves up one line, and l moves right one character.
Section 9.2

i, o, a

i, o, and a are used to enter insert mode from command mode. i will allow a user to start inserting text at the current location of the cursor. o will allow a user to start inserting text a line below the current location of the cursor, and a will allow a user to insert text one postion after the current location of the cursor.
Section 9.4

vi

A screen-oriented text editor originally created for Unix operating systems. vi is also known as a modal editor in which the user must switch modes to create, edit, and search text in a file.
Section 9.1 | Section 9.2 | Section 9.3 | Section 9.5

9.2 Linux Essentials Exam Objectives

This chapter will cover the topics for the following Linux Essentials exam objectives:

Topic 3: The Power of the Command Line (weight: 10)

- **3.3: Turning Commands into a Script**
    - Weight: 4
    - Description: Turning repetitive commands into simple scripts.
    - Key Knowledge Areas:
        - Basic text editing
        - Basic shell scripting
    - The following is a partial list of the used files, terms, and utilities:
        - /bin/sh
        - Variables
        - Arguments
        - for loops
        - echo
        - Exit status
    - Things that are nice to know:
        - pico, nano, vi (only basics for creating scripts)
        - Bash
        - if, while, case statements
        - read and test, and [commands

9.3 Shell Scripts in a Nutshell

A *shell script* is a file of executable commands that has been stored in a text file. When the file is run, each command is executed, Shell scripts have access to all the commands of the shell, including logic. A script can therefore test for the presence of a file or look for particular output and change its behavior accordingly. You can build scripts to automate repetitive parts of your work, which frees your time and ensures consistency each time you use the script. For instance, if you run the same five commands every day, you can turn them into a shell script which reduces your work to one command.

A script can be as simple as one command:

```
echo “Hello, World!”
```

The script, `test.sh`, consists of just one line that prints the string “Hello, World!” to the console.

Running a script can be done either by passing it as an argument to your shell or by running it directly:

```
bob:tmp $ sh test.sh
```

```
Hello, World!
bob:tmp $ ./test.sh
-bash: ./test.sh: Permission denied
bob:tmp $ chmod +x ./test.sh
bob:tmp $ ./test.sh
Hello, World
```

In the example above, first, the script is run as an argument to the shell. Next, the script is run directly from the shell. It is rare to have the current directory in the binary search path ($PATH) so the name is prefixed with "./" to indicate it should be run out of the current directory.

The error "Permission denied" means that the script has not been marked as executable. A quick `chmod` later and the script works. `chmod` is used to change the permissions of a file, which will be explained in detail in a later chapter.

There are different shells with their own language syntax so once you get to more complicated scripts you will want to make sure that your script knows which shell it should run under by specifying the path to the interpreter as the first line, prefixed by "#!" as shown:

```
#!/bin/sh
echo "Hello, World!"
```

The two characters, "#!" are traditionally called the hash and the bang respectively, which leads to the shortened form of *"shebang"* when they're used at the head of a script.

Incidentally, the shebang is used for traditional shell scripts and other text-based languages like Perl, Ruby, and Python. Any text file marked as executable will be run under the interpreter specified in the first line as long as the script is run directly, such as the second invocation shown in the second example in this section (that is, ./script). If you pass the script as an argument to an interpreter, such as "sh script", the given shell will be used no matter what's in the shebang line.

Before you can write shell scripts, you must become comfortable using a text editor. Traditional office tools like LibreOffice aren't appropriate for this task as shell scripts are text files; this job calls for a text editor.

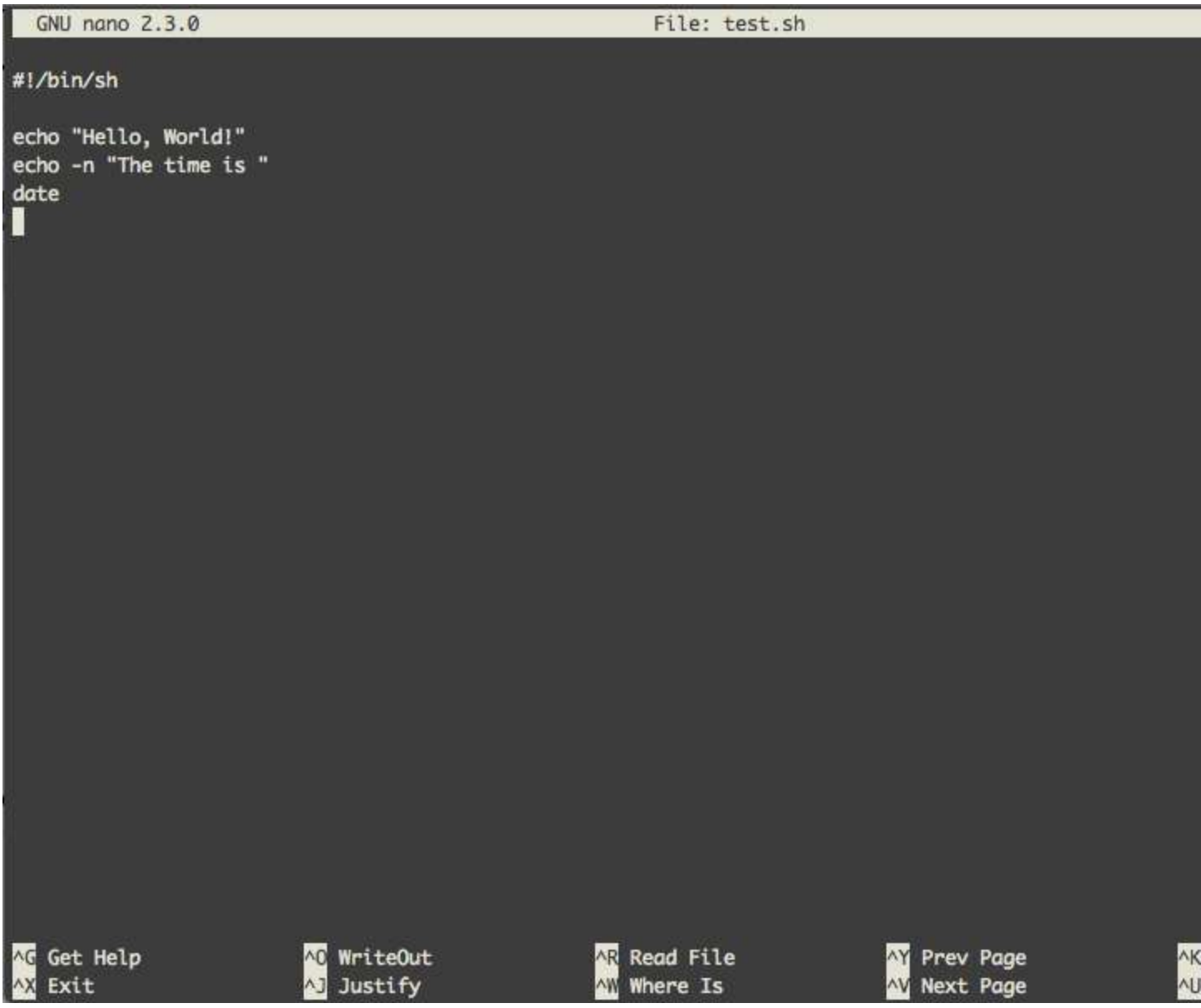
9.4 Editing Shell Scripts

UNIX has many text editors, the merits of one over the other are often hotly debated. Two are specifically mentioned in the LPI Essentials syllabus: The GNU nano editor is a very simple editor well suited to editing small text files. The Visual Editor, vi, or its newer version, VI improved (vim), is a remarkably powerful editor but has a steep learning curve. We'll focus on nano.

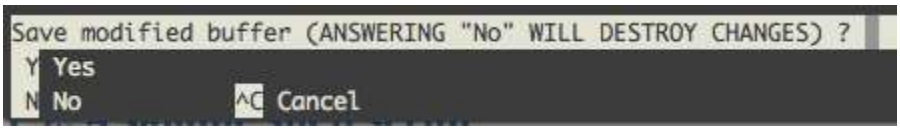
Type `nano test.sh` and you'll see screen similar to this:

Nano has few features to get in your way. You simply type with your keyboard, using the arrow keys to move around and the delete/backspace button to delete text. Along the bottom of the screen you can see some commands available to you, which are context sensitive and change depending on what you're doing. If you're directly on the Linux machine itself, as opposed to connecting over the network, you can also use the mouse to move the cursor and highlight text.

To get familiar with the editor, start typing out a simple shell script while inside nano:



Note that the bottom-left option is **^X Exit** which means "press control and X to exit". Press Ctrl and X together and the bottom will change:



At this point, you can exit the program without saving by pressing the **N** key, or save first by pressing **Y** to save. The default is to save the file with the current file name. You can press the Enter key to save and exit.

You will be back at the shell prompt after saving. Return to the editor. This time press Ctrl and O together to save your work without exiting the editor. The prompts are largely the same, except that you're back in the editor.

This time use the arrows keys to move your cursor to the line that has “The time is”. Press Control and K twice to cut the last two lines to the copy buffer. Move your cursor to the remaining line and press Control and U once to paste the copy buffer to the current position. This makes the script echo the current time before greeting you and saved you needing to re-type the lines.

Other helpful commands you might need are:

Command	Description
Ctrl + W	search the document
Ctrl + W, then Control + R	search and replace
Ctrl + G	show all the commands possible
Ctrl + Y/V	page up / down
Ctrl + C	show the current position in the file and the file’s size

9.5 Scripting Basics

You got your first taste of scripting earlier in this chapter where we introduced a very basic script that ran a single command. The script started with the shebang line, telling Linux that /bin/bash (which is Bash) is to be used to execute the script.

Other than running commands, there are 3 topics you must become familiar with:

- Variables, which hold temporary information in the script
- Conditionals, which let you do different things based on tests you write
- Loops, which let you do the same thing over and over

9.5.1 Variables

Variables are a key part of any programming language. A very simple use of variables is shown here:

```
#!/bin/bash


ANIMAL="penguin"
echo "My favorite animal is a $ANIMAL"
```

After the shebang line is a directive to assign some text to a variable. The variable name is ANIMAL and the equals sign assigns the string “penguin”. Think of a variable like a box in which you can store things. After executing this line, the box called “ANIMAL” contains the word “penguin”.

It is important that there are no spaces between the name of the variable, the equals sign, and the item to be assigned to the variable. If you have a space there, you will get an odd error such as “command not found”. Capitalizing the name of the variable is not necessary but it is a useful convention to separate variables from commands to be executed.

Next, the script echos a string to the console. The string contains the name of the variable preceded by a dollar sign. When the interpreter sees that dollar sign it recognizes that it will be substituting the contents of the variable, which is called *interpolation*. The output of the script is then “My favorite animal is a penguin”.

So remember this: To assign to a variable, just use the name of the variable. To access the contents of the variable, prefix it with a dollar sign. Here, we show a variable being assigned the contents of another variable!

```
#!/bin/bash


ANIMAL=penguin
SOMETHING=$ANIMAL
echo "My favorite animal is a $SOMETHING"
```

ANIMAL contains the string “penguin” (as there are no spaces, the alternative syntax without using quotes is shown). SOMETHING is then assigned the contents of ANIMAL (because ANIMAL has the dollar sign in front of it).

If you wanted, you could assign an interpolated string to a variable. This is quite common in larger scripts, as you can build up a larger command and execute it!

Another way to assign to a variable is to use the output of another command as the contents of the variable by enclosing the command in back ticks:

```
#!/bin/bash
CURRENT_DIRECTORY=`pwd`
echo "You are in $CURRENT_DIRECTORY"
```

This pattern is often used to process text. You might take text from one variable or an input file and pass it through another command like `sed` or `awk` to extract certain parts and keep the result in a variable.

It is possible to get input from the user of your script and assign it to a variable through the `read` command:

```
#!/bin/bash


echo -n "What is your name? "
read NAME
echo "Hello $NAME!"
```

Read can accept a string right from the keyboard or as part of command redirection like you learned in the last chapter.

There are some special variables in addition to the ones you set. You can pass arguments to your script:

```
#!/bin/bash
echo "Hello $1"
```

A dollar sign followed by a number *N* corresponds to the *Nth* argument passed to the script. If you call the example above with `./test.sh` Linux then the output will be “Hello Linux”. The $0 variable contains the name of the script itself.

After a program runs, be it a binary or a script, it returns an *exit code* which is an integer between 0 and 255. You can test this through the $? variable to see if the previous command completed successfully.

```
bob:tmp $ grep -q root /etc/passwd
bob:tmp $ echo $?
0
bob:tmp $ grep -q slartibartfast /etc/passwd
bob:tmp $ echo $?
1
```

The `grep` command was used to look for a string within a file with the `-q` flag, which means “quiet”. Grep, while running in quiet mode, returns 0 if the string was found and 1 otherwise. This information can be used in a conditional to perform an action based on the output of another command.

Likewise you can set the exit code of your own script with the `exit` command:

```
#!/bin/bash
# Something bad happened!
exit 1
```

The example above shows a comment (#). Anything after the hash mark is ignored which can be used to help the programmer leave notes. The “exit 1” returns exit code 1 to the caller. This even works in the shell, if you run this script from the command line and then type “echo $?” you will see it returns 1.

By convention, an exit code of 0 means “everything is OK”. Exit codes greater than 0 mean some kind of error happened, which is specific to the program. Above you saw that grep uses 1 to mean the string was not found.

9.5.2 Conditionals

Now that you can look at and set variables, it is time to make your script do different functions based on tests, called *branching*. The if statement is the basic operator to implement branching.

A basic if statement looks like this:

```
if somecommand; then
  # do this if somecommand has an exit code of 0
fi
```

The next example will run “somecommand” (actually, everything up to the semicolon) and if the exit code is 0 then the contents up until the closing “fi” will be run. Using what you know about grep, you can now write a script that does different things based on the presence of a string in the password file:

```
#!/bin/bash

if grep -q root /etc/passwd; then
  echo root is in the password file
else
  echo root is missing from the password file
fi
```

From previous examples, you might remember that the exit code of grep is 0 if the string is found. The example above uses this in one line to print a message if root is in the password or a different message if it isn’t. The difference here is that instead of an “fi” to close off the if block, there’s an

“else”. This lets you do one action if the condition is true, and another if the condition is false. The else block must still be closed with the fi keyword.

Other common tasks are to look for the presence of a file or directory and to compare strings and numbers. You might initialize a log file if it doesn’t exist, or compare the number of lines in a file to the last time you ran it. “If” is clearly the command to help here, but what command do you use to make the comparison?

The test command gives you easy access to comparison and file test operators. For example:

Command	Description
test –f /dev/ttyS0	0 if the file exists
test ! –f /dev/ttyS0	0 if the file doesn’t exist
test –d /tmp	0 if the directory exists
test –x `which ls`	substitute the location of ls then test if the user can execute
test 1 –eq 1	0 if numeric comparison succeeds
test ! 1 –eq 1	NOT – 0 if the comparison fails
test 1 –ne 1	Easier, test for numeric inequality
test “a” = “a”	0 if the string comparison succeeds
test “a” != “a”	0 if the strings are different
test 1 –eq 1 –o 2 –eq 2	-o is OR: either can be the same
test 1 –eq 1 –a 2 –eq 2	-a is AND: both must be the same

It is important to note that `test` looks at integer and string comparisons differently. 01 and 1 are the same by numeric comparison, but not by string comparison. You must always be careful to remember what kind of input you expect.

There are many more tests, such as `-gt` for greater than, ways to test if one file is newer than the other, and many more. Consult the test man page for more.

`test` is fairly verbose for a command that gets used so frequently, so there is an alias for it called '[' (left square bracket. If you enclose your conditions in square brackets, it's the same as running `test`. So, these statements are identical.

```
if test –f /tmp/foo; then
if [ -f /tmp/foo]; then
```

While the latter form is most often used, it is important to understand that the square bracket is a command on its own that operates similarly to test except that it requires the closing square bracket.

"if" has a final form, that lets you do multiple comparisons at one time using "elif" (short for else if).

```
#!/bin/bash

if [ "$1" = "hello" ]; then
  echo "hello yourself"
elif [ "$1" = "goodbye" ]; then
  echo "nice to have met you"
  echo "I hope to see you again"
else
  echo "I didn't understand that"
fi
```

The code above compares the first argument passed to the script. If it is hello, the first block is executed. If not, the script checks to see if it is goodbye and echos a different message if so. Otherwise, a third message is sent. Note that the $1 variable is quoted and the string comparison operator is used instead of the numeric version (-eq).

The if/elif/else tests can become quite verbose and complicated. The *case* statement provides a different way of making multiple tests easier.

```
#!/bin/bash

case "$1" in
hello|hi)
```

```
  echo "hello yourself"
  ;;
goodbye)
  echo "nice to have met you"
  echo "I hope to see you again"
  ;;
*)
  echo "I didn't understand that"
esac
```

The case statement starts off with a description of the expression being tested: case *EXPRESSION*in. The expression here is the quoted $1.

Next, each set of tests are executed as a pattern match terminated by a closing parenthesis. The previous example first looks for “hello” or “hi”; multiple options are separated by the vertical bar (|) which is an OR operator in many programming languages. Following that are the commands to be executed if the pattern returns true, which are terminated by two semicolons. The pattern repeats.

The * pattern is the same as an else because it matches anything. The behavior of the case statement is similar to the if/elif/else statement in that processing stops after the first match. If none of the other options matched the * ensures that the last one will match.

With a solid understanding of conditionals you can have your scripts take actions only if necessary.

9.5.3 Loops

Loops allow code to be executed repeatedly. They can be useful in numerous situations, such as when you want to run the same commands over each file in a directory, or repeat some action 100 times. There are two main loops in shell scripts: the for loop and the while loop.

For loops are used when you have a finite collection over which you want to iterate, such as a list of files, or a list of server names:

```
#!/bin/bash


SERVERS="servera serverb serverc"
for S in $SERVERS; do
  echo "Doing something to $S"
done
```

The script first sets a variable containing a space separated list of server names. The for statement then loops over the list of servers, each time it sets the S variable to the current server name. The choice of S was arbitrary, but note that the S has no dollar sign but the $SERVERS does, showing that $SERVERS will be expanded to the list of servers. The list does not have to be a variable. This example shows two more ways to pass a list.

```
#!/bin/bash


for NAME in Sean Jon Isaac David; do
  echo "Hello $NAME"
done


for S in *; do
  echo "Doing something to $S"
done
```

The first loop is functionally the same as the previous example, except that the list is passed to the for loop directly instead of using a variable. Using a variable helps the clarity of the script as someone can easily make changes to the variable rather than looking at a loop.

The second loop uses a * which is a *file glob*. This gets expanded by the shell to all the files in the current directory.

The other type of loop, a *while* loop, operates on a list of unknown size. Its job is to keep running and on each iteration perform a test to see if it should run another time. You can think of it as “while some condition is true, do stuff.”

```
#!/bin/bash


i=0
while [ $i -lt 10 ]; do
  echo $i
  i=$(( $i + 1))
done
echo “Done counting”
```

The example above shows a while loop that counts from 0 to 9. A counter variable, i, is initialized to 0. Then a while loop is run with the test being “is $i less than 10?” Note that the while loop uses the same notation as an if statement!

Within the while loop the current value of i is echoed and then 1 is added to it through the $((*arithmetic*)) command and assigned back into i. Once i becomes 10 the while statement returns false and processing continues after the loop.

7. Archiving and compressing

7.1 Introduction

In this chapter, we discuss how to manage archive files at the command line.

File archiving is used when one or more files need to be transmitted or stored as efficiently as possible. There are two aspects to this:

- **Archiving** – Combining multiple files into one, which eliminates the overhead in individual files and makes it easier to transmit
- **Compressing** – Making the files smaller by removing redundant information

You can archive multiple files into a single archive and then compress it, or you can compress an individual file. The former is still referred to as archiving, while the latter is just called compression. When you take an archive, decompress it and extract one or more files, you are *un-archiving* it.

Even though disk space is relatively cheap, archiving and compression still has value:

- If you want to make a large number of files available, such as the source code to an application or a collection of documents, it is easier for people to download a compressed archive than it is to download individual files.
- Log files have a habit of filling disks so it is helpful to split them by date and compress older versions.
- When you back up directories, it is easier to keep them all in one archive than it is to version each file.
- Some streaming devices such as tapes perform better if you're sending a stream of data rather than individual files.
- It can often be faster to compress a file before you send it to a tape drive or over a slower network and decompress it on the other end than it would be to send it uncompressed.

7.3 Compressing files

Compressing files makes them smaller by removing duplication from a file and storing it such that the file can be restored. A file with human readable text might have frequently used words replaced by something smaller, or an image with a solid background might represent patches of that color by a code. You generally don't use the compressed version of the file, instead you decompress it before use. The *compression algorithm* is a procedure the computer does to encode the original file, and as a result make it smaller. Computer scientists research these algorithms and come up with better ones that can work faster or make the input file smaller.

When talking about compression, there are two types:

- **Lossless**: No information is removed from the file. Compressing a file and decompressing it leaves something identical to the original.
- **Lossy**: Information might be removed from the file as it is compressed so that uncompressing a file will result in a file that is slightly different than the original. For instance, an image with two subtly different shades of green might be made smaller by treating those two shades as the same. Often, the eye can't pick out the difference anyway.

Generally human eyes and ears don't notice slight imperfections in pictures and audio, especially as they are displayed on a monitor or played over speakers. Lossy compression often benefits media because it results in

smaller file sizes and people can't tell the difference between the original and the version with the changed data. For things that must remain intact, such as documents, logs, and software, you need lossless compression.

Most image formats, such as GIF, PNG, and JPEG, implement some kind of lossy compression. You can generally decide how much quality you want to preserve. A lower quality results in a smaller file, but after decompression you may notice artifacts such as rough edges or discolorations. High quality will look much like the original image, but the file size will be closer to the original.

Compressing an already compressed file will not make it smaller. This is often forgotten when it comes to images, since they are already stored in a compressed format. With lossless compression, this multiple compression is not a problem, but if you compress and decompress a file several times using a lossy algorithm you will eventually have something that is unrecognizable.

Linux provides several tools to compress files, the most common is `gzip`. Here we show a log file before and after compression.

```
bob:tmp $ ls -l access_log*
-rw-r--r-- 1 sean sean 372063 Oct 11 21:24 access_log
bob:tmp $ gzip access_log
bob:tmp $ ls -l access_log*
-rw-r--r-- 1 sean sean 26080 Oct 11 21:24 access_log.gz
```

In the example above, there is a file called "access_log" that is 372,063 bytes. The file is compressed by invoking the `gzip` command with the name of the file as the only argument. After that command completes, the original file is gone and a compressed version with a file extension of .gz is left in its place. The file size is now 26,080 bytes, giving a compression ratio of about 14:1, which is common with log files.

Gzip will give you this information if you ask, by using the `–l` parameter, as shown here:

```
bob:tmp $ gzip -l access_log.gz
    compressed    uncompressed  ratio uncompressed_name
         26080          372063  93.0% access_log
```

Here, you can see that the compression ratio is given as 93%, which is the inverse of the 14:1 ratio, i.e. 13/14. Additionally, when the file is decompressed it will be called access_log.

```
bob:tmp $ gunzip access_log.gz
bob:tmp $ ls -l access_log*
-rw-r--r-- 1 sean sean 372063 Oct 11 21:24 access_log
```

The opposite of the `gzip` command is `gunzip`. Alternatively, `gzip –d` does the same thing (gunzip is just a script that calls gzip with the right parameters). After gunzip does its work you can see that the access_log file is back to its original size.

Gzip can also act as a filter which means it doesn't read or write anything to disk but instead receives data through an input channel and writes it out to an output channel. You'll learn more about how this works in the next chapter, so the next example just gives you an idea of what you can do by being able to compress a stream.

```
bob:tmp $ mysqldump -A | gzip > database_backup.gz
```

```
bob:tmp $ gzip -l database_backup.gz
      compressed        uncompressed  ratio uncompressed_name
           76866          1028003  92.5% database_backup
```

The mysqldump –A command outputs the contents of the local MySQL databases to the console. The | character (pipe) says "redirect the output of the previous command into the input of the next one". The program to receive the output is gzip, which recognizes that no filenames were given so it should operate in pipe mode. Finally, the > database_backup.gz means "redirect the output of the previous command into a file called database_backup.gz. Inspecting this file with gzip –l shows that the compressed version is 7.5% of the size of the original, with the added benefit that the larger file never had to be written to disk.

There is another pair of commands that operate virtually identically to gzip and gunzip. These are bzip2 and bunzip2. The bzip utilities use a different compression algorithm (called Burrows-Wheeler block sorting, versus Lempel-Ziv coding used by gzip) that can compress files smaller than gzip at the expense of more CPU time. You can recognize these files because they have a .bz or bz2 extension instead of .gz.

7.4 Archiving Files

If you had several files to send to someone, you could compress each one individually. You would have a smaller amount of data in total than if you sent uncompressed files, but you would still have to deal with many files at one time.

Archiving is the solution to this problem. The traditional UNIX utility to archive files is called tar, which is a short form of TApe aRchive. Tar was used to stream many files to a tape for backups or file transfer. Tar takes in several files and creates a single output file that can be split up again into the original files on the other end of the transmission.

Tar has 3 modes you will want to be familiar with:

- **Create**: make a new archive out of a series of files
- **Extract**: pull one or more files out of an archive
- **List**: show the contents of the archive without extracting

Remembering the modes is key to figuring out the command line options necessary to do what you want. In addition to the mode, you will also want to make sure you remember where to specify the name of the archive, as you may be entering multiple file names on a command line.

Here, we show a tar file, also called a tarball, being created from multiple access logs.

```
bob:tmp $ tar -cf access_logs.tar access_log*
bob:tmp $ ls -l access_logs.tar
-rw-rw-r-- 1 sean sean 542720 Oct 12 21:42 access_logs.tar
```

Creating an archive requires two named options. The first, **c**, specifies the mode. The second, **f**, tells tar to expect a file name as the next argument. The first argument in the example above creates an archive called access_logs.tar. The remaining arguments are all taken to be input file names, either as a wildcard, a list of files, or both. In this example, we use the wildcard option to include all files that begin with access_log.

The example above does a long directory listing of the created file. The final size is 542,720 bytes which is slightly larger than the input files. Tarballs can be compressed for easier transport, either by gzipping the archive or by having tar do it with the z flag as follows:

```
bob:tmp $ tar -czf access_logs.tar.gz  access_log*
bob:tmp $ ls -l access_logs.tar.gz
-rw-rw-r-- 1 sean sean 46229 Oct 12 21:50 access_logs.tar.gz
bob:tmp $ gzip -l access_logs.tar.gz
         compressed        uncompressed  ratio uncompressed_name
              46229              542720  91.5% access_logs.tar
```

The example above shows the same command as the prior example, but with the addition of the z parameter. The output is much smaller than the tarball itself, and the resulting file is compatible with gzip. You can see from the last command that the uncompressed file is the same size as it would be if you tarred it in a separate step.

While UNIX doesn't treat file extensions specially, the convention is to use .tar for tar files, and .tar.gz or .tgz for compressed tar files. You can use bzip2 instead of gzip by substituting the letter **j** for **z** and using .tar.bz2, .tbz, or .tbz2 for a file extension (e.g. `tar –cjf file.tbz access_log*`).

Given a tar file, compressed or not, you can see what's in it by using the **t** command:

```
bob:tmp $ tar -tjf access_logs.tbz
logs/
logs/access_log.3
logs/access_log.1
logs/access_log.4
logs/access_log
logs/access_log.2
```

This example uses 3 options:

- t: list files in the archive
- j: decompress with bzip2 before reading
- f: operate on the given filename (access_logs.tbz)

The contents of the compressed archive are then displayed. You can see that a directory was prefixed to the files. Tar will recurse into subdirectories automatically when compressing and will store the path info inside the archive.

Just to show that this file is still nothing special, we will list the contents of the file in two steps using a pipeline.

```
bob:tmp $ bunzip2 -c access_logs.tbz | tar -t
logs/
logs/access_log.3
logs/access_log.1
```

```
logs/access_log.4
logs/access_log
logs/access_log.2
```

The left side of the pipeline is `bunzip –c access_logs.tbz`, which decompresses the file but the (`-c` option) sends the output to the screen. The output is redirected to `tar –t`. If you don't specify a file with `–f` then tar will read from the standard input, which in this case is the uncompressed file.

Finally you can extract the archive with the `–x` flag:

```
bob:tmp $ tar -xjf access_logs.tbz
bob:tmp $ ls -l
total 36
-rw-rw-r-- 1 sean sean 30043 Oct 14 13:27 access_logs.tbz
drwxrwxr-x 2 sean sean  4096 Oct 14 13:26 logs
bob:tmp $ ls -l logs
total 536
-rw-r--r-- 1 sean sean 372063 Oct 11 21:24 access_log
-rw-r--r-- 1 sean sean    362 Oct 12 21:41 access_log.1
-rw-r--r-- 1 sean sean 153813 Oct 12 21:41 access_log.2
-rw-r--r-- 1 sean sean   1136 Oct 12 21:41 access_log.3
-rw-r--r-- 1 sean sean    784 Oct 12 21:41 access_log.4
```

The example above uses the similar pattern as before, specifying the operation (eXtract), the compression (the j flag, meaning bzip2), and a file name (-f access_logs.tbz). The original file is untouched and the new **logs** directory is created. Inside the directory are the files.

Add the `–v` flag and you will get verbose output of the files processed. This is helpful so you can see what's happening:

```
bob:tmp $ tar -xjvf access_logs.tbz
logs/
logs/access_log.3
logs/access_log.1
logs/access_log.4
logs/access_log
logs/access_log.2
```

It is important to keep the `–f` flag at the end, as tar assumes whatever follows it is a filename. In the next example, the `f` and `v` flags were transposed, leading to tar interpreting the command as an operation on a file called "v" (the relevant message is in italics.)

```
bob:tmp $ tar -xjfv access_logs.tbz
```

```
tar (child): v: Cannot open: No such file or directory
tar (child): Error is not recoverable: exiting now
tar: Child returned status 2
tar: Error is not recoverable: exiting now
```

If you only want some files out of the archive you can add their names to the end of the command, but by default they must match the name in the archive exactly or use a pattern:

```
bob:tmp $ tar -xjvf access_logs.tbz logs/access_log
logs/access_log
```

The example above shows the same archive as before, but extracting only the "logs/access_log" file. The output of the command (as verbose mode was requested with the "v" flag) shows only the one file has been extracted.

Tar has many more features, such as the ability to use patterns when extracting files, excluding certain files, or outputting the extracted files to the screen instead of disk. The documentation for tar has in depth information.

7.5 ZIP files

The de facto archiving utility in the Microsoft world is the ZIP file. It is not as prevalent in Linux but is well supported by the `zip` and `unzip` commands. With tar and gzip/gunzip the same commands and options can be used to do the creation and extraction, but this is not the case with zip. The same option has different meanings for the two different commands.

The default mode of `zip` is to add files to an archive and compress it.

```
bob:tmp $ zip logs.zip logs/*
  adding: logs/access_log (deflated 93%)
  adding: logs/access_log.1 (deflated 62%)
  adding: logs/access_log.2 (deflated 88%)
  adding: logs/access_log.3 (deflated 73%)
  adding: logs/access_log.4 (deflated 72%)
```

The first argument in the example above is the name of the archive to be operated on, in this case it is **logs.zip**. After that, is a list of files to be added. The output shows the files and the compression ratio. It should be noted that tar requires the `–f` option to indicate a filename is being passed, while zip and unzip require a filename and therefore don't need you to explicitly say a filename is being passed.

Zip will not recurse into subdirectories by default, which is different behavior than tar. That is, merely adding "logs" instead of "logs/*" will only add the empty directory and not the files under it. If you want tar like behavior, you must use the `–r` command to indicate recursion is to be used:

```
bob:tmp $ zip -r logs.zip logs
  adding: logs/ (stored 0%)
  adding: logs/access_log.3 (deflated 73%)
  adding: logs/access_log.1 (deflated 62%)
```

```
 adding: logs/access_log.4 (deflated 72%)
 adding: logs/access_log (deflated 93%)
 adding: logs/access_log.2 (deflated 88%)
```

In the example above, all files under the logs directory are added because it uses the `–r` option. The first line of output indicates that a directory was added to the archive, but otherwise the output is similar to the previous example.

Listing files in the zip is done by the `unzip` command and the `–l` option (list):

```
bob:tmp $ unzip -l logs.zip
Archive:  logs.zip
  Length     Date    Time    Name
---------  ---------- -----   ----
        0  10-14-2013 14:07   logs/
     1136  10-14-2013 14:07   logs/access_log.3
      362  10-14-2013 14:07   logs/access_log.1
      784  10-14-2013 14:07   logs/access_log.4
    90703  10-14-2013 14:07   logs/access_log
   153813  10-14-2013 14:07   logs/access_log.2
---------                     -------
   246798                     6 files
```

Extracting the files is just like creating the archive, as the default operation is to extract:

```
bob:tmp $ unzip logs.zip
Archive:  logs.zip
   creating: logs/
  inflating: logs/access_log.3
  inflating: logs/access_log.1
  inflating: logs/access_log.4
  inflating: logs/access_log
  inflating: logs/access_log.2
```

Here, we extract all the files in the archive to the current directory. Just like tar, you can pass filenames on the command line:

```
bob:tmp $ unzip logs.zip access_log
Archive:  logs.zip
caution: filename not matched:  access_log
bob:tmp $ unzip logs.zip logs/access_log
```

```
Archive:  logs.zip
  inflating: logs/access_log
bob:tmp $ unzip logs.zip logs/access_log.*
Archive:  logs.zip
  inflating: logs/access_log.3
  inflating: logs/access_log.1
  inflating: logs/access_log.4
  inflating: logs/access_log.2
```

The example above shows three different attempts to extract a file. First, just the name of the file is passed without the directory component. Like tar, the file is not matched.

The second attempt passes the directory component along with the filename, which extracts just that file.

The third version uses a wildcard, which extracts the 4 files matching the pattern, just like tar.

The zip and unzip man pages describe the other things you can do with these tools, such as replace files within the archive, use different compression levels, and even use encryption.