SUMMARY

CHAPTER 1 - INTRODUCTION

The definition of the word *Linux* depends on the context in which it is used.

Technically speaking, Linux is the *kernel* of the system, which is the central controller of everything that happens on the computer. People that say their computer "runs Linux" are usually referring to the kernel and suite of tools that come with it (called a *distribution*). If someone says they have Linux experience, it might refer to configuring systems, running web servers, or any number of other services and programs that operate on top of Linux. Over time, Linux administration has evolved to encompass just about every task that a modern business, educational, or government institution might use in their daily operations.

What about *UNIX*? UNIX was originally an operating system developed at AT&T Bell Labs in the 1970s. It has been 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 most if not all of the requirements of the UNIX specification, Linux has not been certified, so Linux really isn't UNIX! It's just… UNIX-like.

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, it interprets instructions given to it by the user, and it handles more common and simple tasks such as displaying text on a monitor. When an application needs to write to disk, it must ask the kernel to complete the write operation.

The kernel also handles the switching of applications. A computer will have one or more CPUs and a finite amount of memory. The kernel takes care of unloading tasks and loading new tasks, and can manage multiple tasks across multiple 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 *preemptive multitasking*. Multitasking means that the computer is doing several tasks at once, and preemptive means that the kernel is deciding when to switch focus between tasks. With the tasks so 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 used in a while to the disk.

When the computer starts up, it loads a small piece of code called a *bootloader*. The bootloader's job is to give you a choice (if configured) of options to load one or more versions of Linux, or even other operating systems, and then to load the kernel of the chosen option 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 adjust 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.

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 the block of a 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.

Role of Open Source

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. The Linux kernel is mostly 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.

In 1991, Linux started as a hobby project by Linus Torvalds. He made the source freely available, allowing others to join in and shape this fledgling operating system. It was not the first system to be developed by a volunteer group, but since it was built from scratch, early adopters could influence the project's direction. People took the source, made changes, and shared them back with the rest of the group, greatly accelerating the pace of development, and ensuring mistakes from other operating systems were not repeated.

The Linux kernel is licensed under the GNU Public License (GPL) which requires you to make changes available. This guarantees that those who use the code will also contribute to the greater good by making those changes available to anyone.

Alongside this, was the *GNU project* (GNU's, not UNIX). While GNU (pronounced "guh-noo") was building their own operating system, they were far more successful 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. However, 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.

Linux Distributions

Take the Linux kernel and the GNU tools, add some more user-facing applications like an email client, word processors and other programs 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.

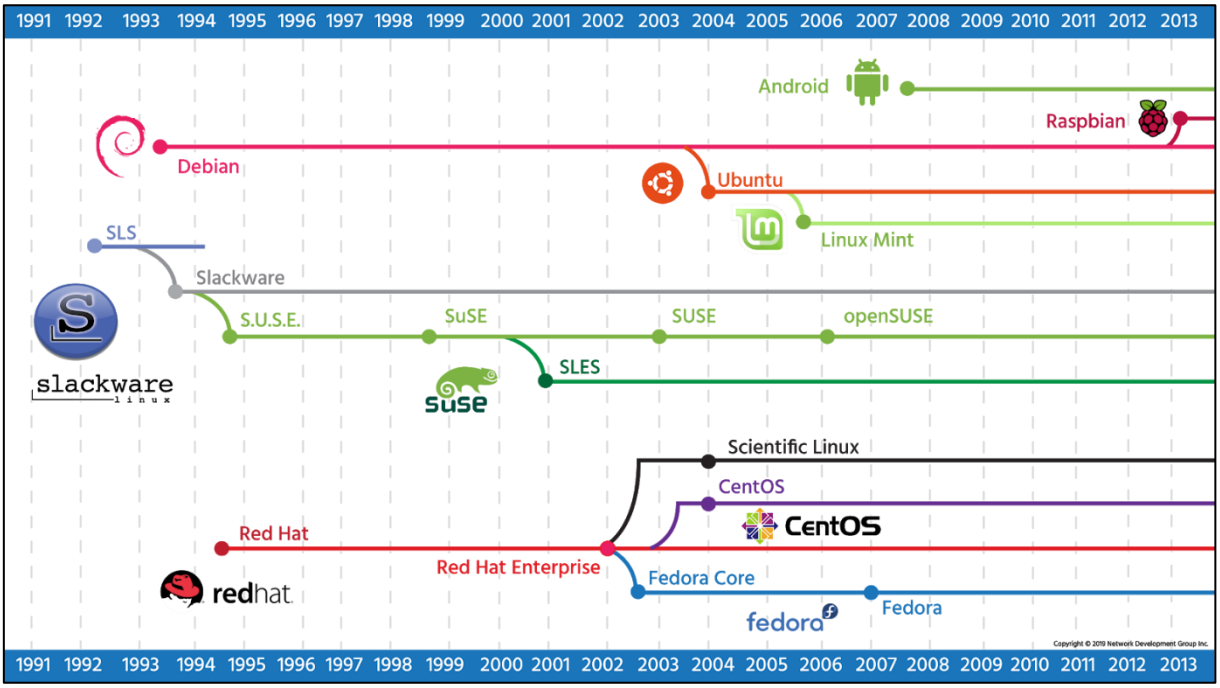
Like UNIX, there are many different flavors of distributions. These days, there are distributions that focus on running servers, desktops, or even industry-specific tools like electronics design or statistical computing. The major players in the market can be traced back to either **Red Hat** or **Debian**. 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) based on the .rpm file format. 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, while 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 makes money by providing support.



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 support for other hardware was built. Eventually, Linux started supporting other chipsets, including hardware that was made to run competitive operating systems!

The types of hardware grew from the humble Intel chip up to supercomputers. Smaller sized Linux-supported chips were eventually 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 adopted 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 developments 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. 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 how to use Linux. The custom software interacts with the user and Linux provides a stable platform.

SHELL

An operating system provides at least one *shell* that allows the user to communicate with the operating system. 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).

Microsoft Windows™ typically uses a GUI shell, primarily using the mouse to indicate what you want to accomplish. 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 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 more flexible and powerful interface.
- **Resources**: A GUI shell typically uses a relatively large number of resources (RAM, CPU, etc.). This is because a great deal of processing power and memory is needed to display graphics. By contrast, 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 just typing 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 execute commands in a GUI shell remotely, 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 usually thousands of CLI programs on a typical Linux OS compared to only a couple hundred programs in a primarily GUI-based OS like Microsoft Windows. More programs mean 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 Command Prompt, also called the DOS shell. Recently, Microsoft realized how important it is to have a powerful command line environment and introduced 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 only to run the CLI without the GUI; this is typically done on servers that don't require a GUI, primarily to free up system resources.

BASH SHELL

Not only does the Linux operating system provide multiple GUI shells, but also multiple CLI shells are 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, a default

Consider This

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. Most of the time the default shell a system offers will be adequate for basic tasks. Occasionally, an administrator will want to change the shell to have access to more advanced features, or simply because they are more familiar with a different shell and the features it offers. On systems that are near capacity, it may be advisable not to change shells as it could require additional resources and slow processing for all users.

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

CLI shell used in modern Linux operating systems, 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.

Users interact with a system by executing *commands* which are interpreted by the shell and transformed into actions by the kernel. These actions may or may not return information to the command line depending on the command issued and its result. For example, when the ls command is typed into the console, it will return the contents of whichever directory the user is currently in.

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

A command can be followed by options that modify how the command is executed, and arguments, that are typically the files to be operated on:

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

Note

Some commands require options and arguments while others, like `ls`, can be used alone.

Commands entered are considered standard input, (stdin) whether they are typed by an operator, entered by a script, or as the result of another command. Text returned to the console can be either standard output (stdout), or standard error (stderr).

This deceptively simple method of communicating with the Linux kernel is the basis for almost every interaction a Linux administrator has with their systems. It can be confusing at first for users who have only experienced GUI interfaces, but ultimately it gives the experienced operator far more power than any graphical interface can.

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.

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. 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.

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 they present a CLI instead. If the system is configured to present a CLI, then the system runs a terminal application automatically after you log in.

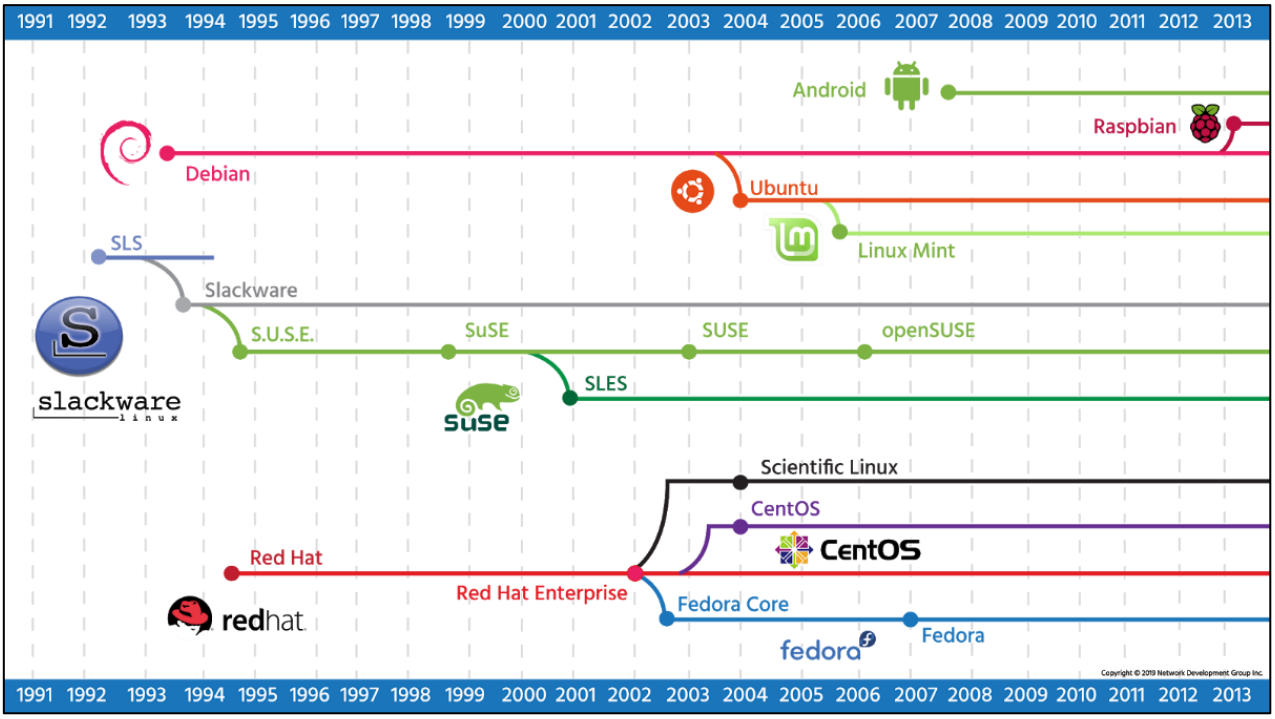
In the early days of computing, terminal devices were large machines that allowed users to provide input through a keyboard and displayed output by printing on paper. Over time, terminals evolved and their size shrank down into something that looked similar to a desktop computer with a video display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

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.

CHAPTER 2 - COMMAND LINE INTERFACE

Most consumer operating systems are designed to shield the user from the “ins and outs” of the CLI. The Linux community is different in that it positively celebrates the CLI for its power, speed, and ability to accomplish a vast array of tasks with a single command line instruction.

When a user first encounters the CLI, they can find it challenging because it requires memorizing a dizzying amount of commands and their options. However, once a user has learned the structure of how commands are used, where the necessary files and directories are located, and how to navigate the hierarchy of a file system, they can be immensely productive. This capability provides more precise control, greater speed, and the ability to automate tasks more easily through scripting.

Furthermore, by learning the CLI, a user can easily be productive almost instantly on ANY flavor or distribution of Linux, reducing the amount of time needed to familiarize themselves with a system because of variations in a GUI.

COMMANDS

What is a command? The simplest answer 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:

- **Internal Commands**: Also called *built-in commands*, these commands are built-in to the shell itself. A good example is the cd (change directory) 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.
- **External Commands**: These commands are stored in files that are searched by the shell. If you type the ls command, then the shell searches through a predetermined list of directories 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.

EXTERNAL COMMANDS

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. 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.

Note

It is possible to view available software packages, binary programs that can be installed directly at the command line. Type the following command into the terminal to view the available source packages (source code that can be modified before it's compiled into binary programs) for the GNU Compiler Collection:

```
sysadmin@localhost:~$ apt-cache search gcc | grep source
gcc-4.8-source - Source of the GNU Compiler Collection
gcc-5-source - Source of the GNU Compiler Collection
gcc-6-source - Source of the GNU Compiler Collection
gcc-7-source - Source of the GNU Compiler Collection
gcc-8-source - Source of the GNU Compiler Collection
gcc-arm-none-eabi-source - GCC cross compiler for ARM Cortex-A/R/M processors (source
```

The `apt-cache` command allows us to display information from the APT database cache. It is commonly used to find information about programs you wish to install and the components required to make them work.

Although there are a tremendous number of free and open source programs available, quite often the binary code you will need as a Linux administrator won't exist for the particular distribution you are running. Since open source licensing gives you access to the code for these programs, one of your tasks will be compiling, and sometimes modifying that code into executable programs that can be installed on the systems you manage. The Free Software Foundation (FSF) distributes the GNU Compiler Collection (GCC) to make this process easier. The GCC provides a compiler system (the special programs used to convert source code into usable binary programs) with front ends for many different programming languages. In fact, the FSF doesn't limit these tools to just Linux. There are versions of the GCC that run on Unix, Windows, MacOS, and many other systems including specific microcontroller environments.

Linux package management will be covered in greater detail later in the course.

Note

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*.

ALIASES

An *alias* can be used to map longer commands to shorter key sequences. When the shell sees an alias being executed, it substitutes the longer sequence before proceeding to interpret commands.

For example, the command ls -l is commonly aliased to l or ll. Because these smaller commands are easier to type, it becomes faster to run the ls -l command line.

To determine what aliases are set on the current shell use the alias command:

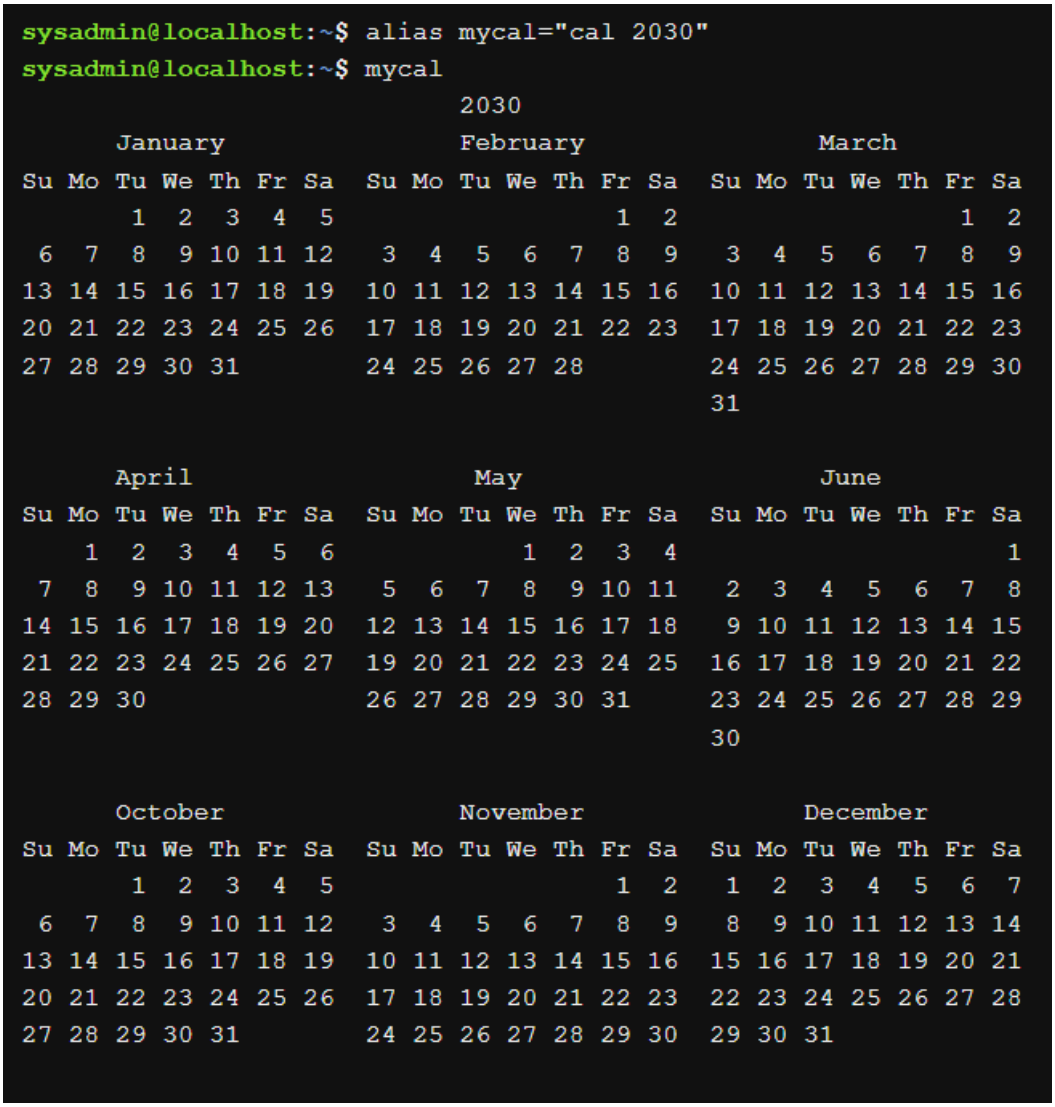
```
sysadmin@localhost:~$ alias
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l='ls -CF'
alias la='ls -A'
alias ll='ls -alF'
alias ls='ls --color=auto'
```

The aliases from the previous examples were created by initialization files. These files are designed to make the process of creating aliases automatic.

New aliases can be created using the following format, where *name* is the name to be given the alias and *command* is the command to be executed when the alias is run.

```
alias name=command
```

For example, the cal 2030 command displays the calendar for the year 2030. 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:



Aliases created this way only persists while the shell is open. Once the shell is closed, the new aliases are lost. Additionally, each shell has its own aliases, so, aliases created in one shell won't be available in a new shell that's opened.

Basic Command Syntax

To execute a command, the first step is to type the name of the command. Click in the terminal on the right. Type ls and hit **Enter**. The result should resemble the example below:

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

> **Note**
>
> By itself, the `ls` command lists files and directories contained in the current working directory. At this point, you shouldn't worry too much about the output of the command, but instead, focus on understanding how to format and execute commands.
>
> *The `ls` command will be covered in greater detail later in the course.*

Many commands can be used by themselves with no further input. Some commands require additional input to run correctly. This additional input comes in two forms: *options* and *arguments*. Commands typically follow a simple pattern of syntax:

```
command [options…] [arguments…]
```

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 the command 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.

Keep in mind that every part of the command is normally case-sensitive, so LS is incorrect and will fail, but ls is correct and will succeed.

Specifying Arguments

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

An *argument* can be used to specify something for the command to act upon. Following a command, any desired arguments are allowed or are required depending on the command. For example, the touch command is used to create empty files or update the timestamp of existing files. It requires at least one argument to specify the file name to act upon.

```
touch FILE...
```

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

The ls command, on the other hand, allows for a path and/or file name to be specified as an argument, but it's not required.

```
ls [FILE]...
```

An example of a scenario where an argument is allowed but not required is the use of the ls command. If the ls command is used without an argument, it will list the contents of the current directory:

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

If the ls command is given the name of a directory as an argument, it will list the contents of that directory. In the following example, the /etc/ppp directory is used as an argument; the resulting output is a list of files contained in the /etc/ppp directory:

```
sysadmin@localhost:~$ ls /etc/ppp
ip-down.d  ip-up.d
```

The ls command also accepts multiple arguments. To list the contents of both the /etc/ppp and /etc/ssh directories, pass them both as arguments:

```
sysadmin@localhost:~$ ls /etc/ppp /etc/ssh
/etc/ppp:
ip-down.d  ip-up.d

/etc/ssh:
moduli                 ssh_host_ecdsa_key         ssh_host_rsa_key
ssh_config             ssh_host_ecdsa_key.pub     ssh_host_rsa_key.pub
ssh_host_dsa_key       ssh_host_ed25519_key       ssh_import_id
ssh_host_dsa_key.pub   ssh_host_ed25519_key.pub   sshd_config
```

Some commands, like the cp command (*copy* file) and the mv command (*move* file), always require at least two arguments: a source file and a destination file.

```
cp SOURCE... DESTINATION
```

In the example below, we will copy the public ssh rsa key called ssh_host_rsa_key.pub which resides in the /etc/ssh directory, to the /home/sysadmin/Documents directory and verify that it is there:

```
sysadmin@localhost:~$ cp /etc/ssh/ssh_host_rsa_key.pub /home/sysadmin/Documents
sysadmin@localhost:~$ ls ~/Documents
School            alpha.txt    linux.txt     profile.txt
Work              animals.txt  longfile.txt  red.txt
adjectives.txt    food.txt     newhome.txt   spelling.txt
alpha-first.txt   hello.sh     numbers.txt   ssh_host_rsa_key.pub
alpha-second.txt  hidden.txt   os.csv        words
alpha-third.txt   letters.txt  people.csv
```

Consider This

An *ssh rsa key* is a file that contains an authentication credential (similar to a password) used to verify the identity of a machine that is being logged into using the `ssh` command. The `ssh` command allows users on a system to connect to another machine across a network, log in, and then perform tasks on the remote machine.

*The `ssh` command is covered in greater detail later in the **NDG Introduction to Linux 2**.*

QUOTING

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 considered safer and should probably be used whenever you have an argument that contains characters that aren't alphanumeric. To understand the importance of quotes, consider the echo command. The echo command displays text to the terminal and is used extensively in shell scripting.

```
echo [STRING]...
```

Consider the following scenario in which you want to list the contents of the current directory using the ls command and then use the echo command to display the string hello world!! on the screen.

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

```
sysadmin@localhost:~$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos
sysadmin@localhost:~$ echo hello world!!
echo hello worldls
hello worldls
```

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 ls command, so ls replaced !! and then the echo command displayed hello worldls to the screen.

You may want to try the double quote " characters to see if they will block the interpretation (or expansion) of the exclamation !! 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 the exclamation !! characters:

```
sysadmin@localhost:~$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos
sysadmin@localhost:~$ echo "hello world!!"
echo "hello worldls"
hello worldls
```

Using double quotes preserves the literal value of all characters that they enclose except metacharacters such as the $ dollar sign character, the ` backquote character, the \ backslash character and the ! exclamation point character. These characters, called *wild cards*, are symbol characters that have special meaning to the shell. Wild card characters are used for *globbing* and are interpreted by the shell itself before it attempts to run any command. Glob characters are useful because they allow you to specify patterns that make it easier to match file names in the command line. For example, the command ls e?? would list all files in that directory that start with an e and have any two characters after it. However, because glob characters are interpreted differently by the shell, they need to be enclosed appropriately to be interpreted literally.

If you enclose text within the ' single quote characters, then all characters have their literal meaning:

```
sysadmin@localhost:~$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos
sysadmin@localhost:~$ echo 'hello world!!'
hello world!!
```

OPTIONS

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

Options can be used with commands to expand or modify the way a command behaves. If it is necessary 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.

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
```

Often the character is chosen to be mnemonic for its purpose, like choosing the letter *l for long* or *r for reverse*. By default, the ls command prints the results in alphabetical order, so adding the -r option prints the results in reverse alphabetical order.

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

In most cases, options can be used in conjunction with other options. They can be given as separate options like -l -r or combined like -lr. The combination of these two options would result in a long listing output in reverse alphabetical order:

```
sysadmin@localhost:~$ ls -l -r
total 0
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
```

Multiple single options can be either given as separate options like -a -l -r or combined like -alr. The output of all of these examples would be the same:

```
ls -l -a -r
ls -rla
ls -a -lr
```

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 -lr -w 40
total 32
drwxr-xr-x 2 sysadmin sysadmin 4096 Feb 22 16:32 Videos
drwxr-xr-x 2 sysadmin sysadmin 4096 Feb 22 16:32 Templates
drwxr-xr-x 2 sysadmin sysadmin 4096 Feb 22 16:32 Public
drwxr-xr-x 2 sysadmin sysadmin 4096 Feb 22 16:32 Pictures
drwxr-xr-x 2 sysadmin sysadmin 4096 Feb 22 16:32 Music
drwxr-xr-x 2 sysadmin sysadmin 4096 Feb 22 16:32 Downloads
drwxr-xr-x 4 sysadmin sysadmin 4096 Feb 22 16:32 Documents
drwxr-xr-x 2 sysadmin sysadmin 4096 Feb 22 16:32 Desktop
```

If you are using multiple options that require arguments, don't combine them. For example, the -T option, which specifies *tab size*, 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
```

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 the --all option, which lists all files, including hidden ones. For example:

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

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
```

A special option exists, the *lone double hyphen* -- option, 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 -- ' option is placed before the file name, indicating that there are no more options, then the file name can successfully be interpreted as an argument:

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

Consider This

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. For example, using the `ls` command without any options to list the path of the `--badname` file above, will result in an error:

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

To remove the `--badname` file from the home directory, use the `rm` *remove* command:

```
sysadmin@localhost:~$ rm -- --badname
```

The `rm` command will be covered in greater detail later in the course.

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 **Berkeley 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
```

Scripts

One exception to the basic command syntax used is the exec command, which takes another command to execute as an argument. 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 is 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.

A wrapper script often uses the following as the last line of the script to execute another program.

```
exec 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 the exec command can be used to cause redirection for one or more statements in a script.

Displaying System Information

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

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

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.

To display additional information about the system, you can use one of the many options available for the uname command. For example, to display *all* the information about the system, use the -a option with the uname command:

```
sysadmin@localhost:~$ uname -a
Linux localhost 4.4.0-72-generic #93~14.04.1-Ubuntu SMP Fri Mar 31 15:05:15 UTC
2017 x86_64 x86_64 x86_64 GNU/Linux
```

To display information about what kernel version the system is running, use the -r option:

```
sysadmin@localhost:~$ uname -r
4.4.0-72-generic
```

The options for the uname command are summarized below:

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
	`--version`	Version information

CURRENT DIRECTORY

One of the simplest commands available is the pwd command, which is an acronym for *print working directory*. 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. When a user logs into a system, they are normally placed in their *home directory* where files they create and control reside. As you navigate around the file system it is often helpful to know what directory you're in.

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

Notice our virtual machines employ a prompt that displays the current working directory, emphasized with the color blue. In the first prompt above, the blue tilde ~ character is equivalent to /home/sysadmin, representing the user's home directory:

```
sysadmin@localhost:~$
```

After changing directories, the new location can also be confirmed in the new prompt by using the pwd command, and is again shown in blue:

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

To get back to the home directory after changing to a new location, use the cd *change directory* command without any arguments:

```
sysadmin@localhost:~/Documents$ cd
sysadmin@localhost:~$
```

The type command displays information about a command type. For example, if you entered type ls at the command prompt, it will return that the ls command is actually an alias for the ls --color=auto command:

```
sysadmin@localhost:~$ type ls
ls is aliased to `ls --color=auto'
```

Using the -a option with the type command will return all locations of the files that contain a command; also called an *executable* file:

```
sysadmin@localhost:~$ type -a ls
ls is aliased to `ls --color=auto'
ls is /bin/ls
```

In the output above, the /bin/ls file path is the file location of the ls command.

This command is helpful for getting information about commands and where they reside on the system. For internal commands, like the pwd command, the type command will identify them as shell builtins:

```
sysadmin@localhost:~$ type pwd
pwd is a shell builtin
```

For external commands like the ip command, the type command will return the location of the command, in this case, the /sbin directory:

```
sysadmin@localhost:~$ type ip
ip is /sbin/ip
```

Consider This

The `/bin` directory contains executable programs needed for booting a system, commonly used commands, and other programs needed for basic system functionality.

The `/sbin` directory also contains executable programs; mainly commands and tools designed for system administration.

If a command does not behave as expected or if a command is not accessible, that should be, it can be beneficial to know where the shell is finding the command.

The which command searches for the location of a command in the system by searching the PATH variable.

```
sysadmin@localhost:~$ which ls
/bin/ls
```

Note

The `PATH` variable contains a list of directories that are used to search for commands entered by the user.

The `PATH` variable will be covered in greater detail later in the course.

Command Completion

A useful tool of the Bash shell is the ability to complete commands and their arguments automatically. Like many command line shells, Bash offers command line completion, where you type a few characters of a command (or its file name argument) and then press the **Tab** key. The Bash shell will complete the command (or its file name 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 use a *pager* to display the output, which will allow you to scroll through the possible matches.

For example, if you just type c and press the **Tab** key twice, the system will provide you with many matches that you can scroll through:

```
c_rehash                cksum
cal                     clear
calendar                clear_console
caller                  cmp
capsh                   codepage
captoinfo               col
case                    colcrt
cat                     colrm
catchsegv               column
catman                  comm
cautious-launcher       command
cd                      compgen
cfdisk                  complete
chage                   compopt
chardet3                compose
chardetect3             continue
chattr                  coproc
chcon                   corelist
chcpu                   cp
chfn                    cpan
chgpasswd               cpan5.26-x86_64-linux-gnu
chgrp                   cpgr
chmem                   cpio
--More--
```

You may not wish to scroll through all these options; in that case, press **Q** to exit the pager. In a situation like this, you should probably continue to type more characters to achieve a more refined match.

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

Getting Help

As previously mentioned, UNIX was the operating system from which the foundation of Linux was built. The developers of UNIX created help documents called *man* pages (short for *manual* page).

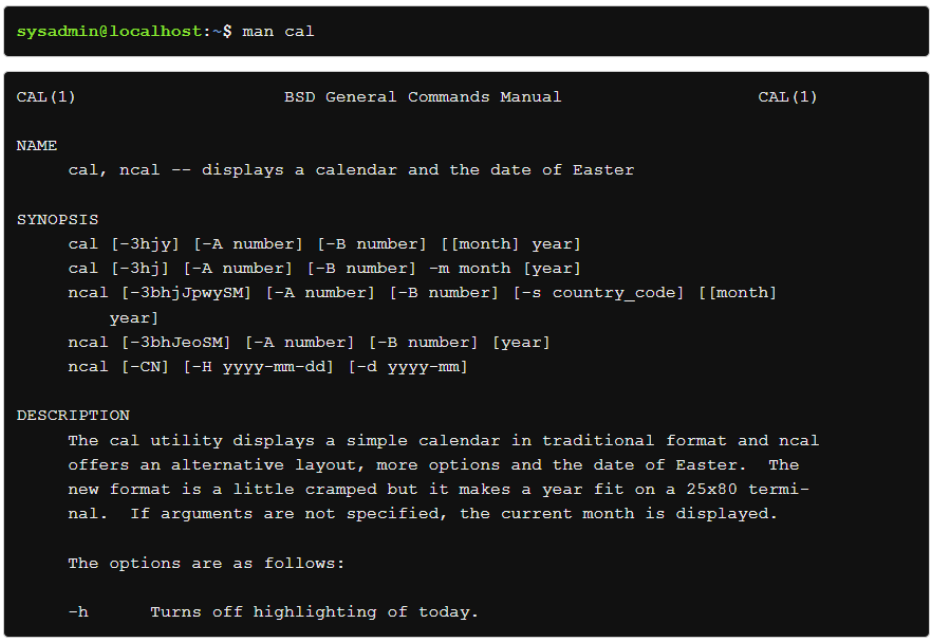
Referring to the man page for a command will provide you with the basic idea behind the purpose of the command, as well as details regarding the options of the command and a description of its features.

Viewing Man Pages

To view a man page for a command, execute the man command in a terminal window.

```
man command
```

For example, the following graphic shows the partial man page for the cal command:



Consider This

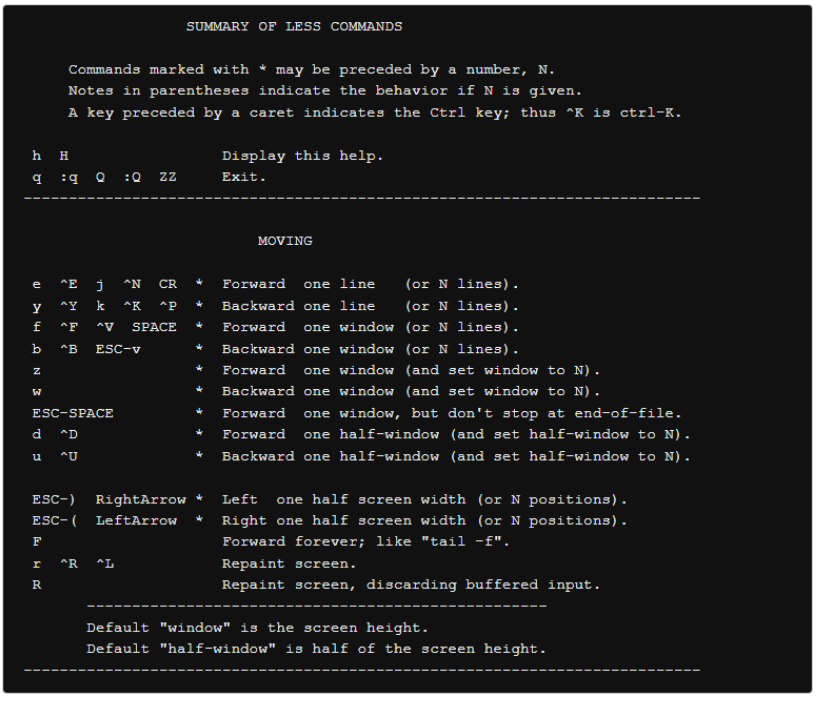
It is possible to print man pages from the command line. If you wanted to send a man page to a default printer from a local machine, you would execute the following command:

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

The command above will not function in our virtual environment.

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. To view the various movement commands that are available, use the **H** key or **Shift+H** while viewing a man page. This will display a help page.



To exit the SUMMARY OF LESS COMMANDS, type **Q**.

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 the beginning of the page
`G`	Go to the end of the page
`h`	Display help
`q`	Quit man page

Sections Within Man Pages

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:

NAME

Provides the name of the command and a very brief description.

```
NAME
       ls - list directory contents
```

SYNOPSIS

A brief summary of the command or function's interface. A summary of how the command line syntax of the program looks.

```
SYNOPSIS
       ls [OPTION]... [FILE]...
```

DESCRIPTION

Provides a more detailed description of the command.

```
DESCRIPTION
       List  information  about  the FILEs (the current directory by default).
       Sort entries alphabetically if none of -cftuvSUX nor --sort  is  speci-
       fied.
```

OPTIONS

Lists the options for the command as well as a description of how they are used. Often this information is found in the DESCRIPTION section and not in a separate OPTIONS section.

```
-a, --all
              do not ignore entries starting with .

       -A, --almost-all
              do not list implied . and ..

       --author
              with -l, print the author of each file

       -b, --escape
              print C-style escapes for nongraphic characters

       --block-size=SIZE
              scale sizes by SIZE before printing them; e.g., '--block-size=M'
              prints sizes in units of 1,048,576 bytes; see SIZE format below
```

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 is found in the DESCRIPTION section and not in a separate FILES section.

AUTHOR

Provides the name of the person who created the man page and (sometimes) how to contact the person.

```
AUTHOR
       Written by Richard M. Stallman and David MacKenzie.
```

REPORTING BUGS

Provides details on how to report problems with the command.

```
REPORTING BUGS
       GNU coreutils online help: <http://www.gnu.org/software/coreutils/>
       Report ls translation bugs to <http://translationproject.org/team/>
```

COPYRIGHT

Provides basic copyright information.

```
COPYRIGHT
       Copyright (C) 2017 Free Software Foundation, Inc.  License GPLv3+:  GNU
       GPL version 3 or later <http://gnu.org/licenses/gpl.html>.
       This  is  free  software:  you  are free to change and redistribute it.
       There is NO WARRANTY, to the extent permitted by law.
```

SEE ALSO

Provides you with an idea of where you can find additional information. This often includes other commands that are related to this command.

```
SEE ALSO
       Full documentation at: <http://www.gnu.org/software/coreutils/ls>
       or available locally via: info '(coreutils) ls invocation'
```

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 cal command:

```
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 [-A number] allows you to specify the number of months to be added to the end of the display.

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

The fourth set of square brackets in the [[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*.

Searching Within a Man Page

In order to search a man page for a term, type the / character followed by the term and hit 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 **Shift+N**.

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 sections of man pages:

1. Executable programs or shell commands
2. System calls (functions provided by the kernel)
3. Library calls (functions within program libraries)
4. Special files (usually found in /dev)
5. File formats and conventions, e.g. /etc/passwd
6. Games
7. Miscellaneous (including macro packages and conventions), e.g. man(7), groff(7)
8. System administration commands (usually only for root)
9. 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, for example "3p".

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:

```
sysadmin@localhost:~$ man cal
```

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

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 following command will display the man page for the passwd command, located in the first section, by default:

```
sysadmin@localhost:~$ man passwd
```

```
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 passwd will look for the passwd man page in section 5 only:

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

NAME
       passwd - the password file
```

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 man page:

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

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 (1ssl)        - compute password hashes
passwd (1)           - change user password
passwd (5)           - the password file
```

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...
```

Warning

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.

CHAPTER 4: FILE GLOBBING

File globbing might sound like an unusual term, but the concept of globbing is probably a familiar one. Like a joker in a deck of cards, which you might be able to use as any card (*a wildcard*), glob characters are special characters that can be used to match file or path names.

For the details of how globbing works, you can view the man page on *glob* in section 7 of the manual pages by executing:

```
sysadmin@localhost:~$ man 7 glob
```

From that manual page:

- *A string is a wildcard pattern if it contains one of the characters '?', '*', or '['. Globbing is the operation that expands a wildcard pattern into the list of path names matching the pattern.*

The shell is what performs the expansion of the wildcards; therefore, the use of globbing is not limited to a particular command. Globbing is especially useful for file management since it can be used to match the path names to files that you want to manage.

ASTERISK * CHARACTER

The *asterisk* wildcard* can be used to match any string, including the empty string.

```
sysadmin@localhost:~$ echo *
Desktop Documents Downloads Music Pictures Public Templates Videos
```

In the screen capture above, the echo command is used to see which files a glob pattern will match. In this case, it displays the name of every file in the user's current directory.

> **Consider This**
> An empty string is a string that has nothing in it. It is commonly referred to as "".

In order to limit the * wildcard, so that it matches fewer file names, it can be combined with other characters. For example, the D* pattern will only match file names that start with a D character:

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

The * character can appear anywhere in the pattern; it can even appear multiple times. It is possible to find a file that not only starts with a D, but also contains the letter n, by using the D*n* pattern:

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

QUESTION MARK ? CHARACTER

The *question mark ? character* in a string will match exactly one character. For example, to see all of the files in the /usr/bin directory that only have one character in the file name, use the /usr/bin/? pattern:

```
sysadmin@localhost:~$ cd /usr/bin
sysadmin@localhost:/usr/bin$ echo /usr/bin/?
/usr/bin/[ /usr/bin/w
```

By adding more question mark ? characters, it is possible to match additional characters. In order to see which files had two or three characters as their file names, execute the following two echo commands:

```
sysadmin@localhost:/usr/bin$ echo ??
du ex hd id nl od pg pr sg tr ul vi wc xz
sysadmin@localhost:/usr/bin$ echo ???
a2p apt awk cal cmp col cut dig env eqn fmt gpg lcf ldd man pdb pic ptx rcp rev
rsh s2p scp see seq ssh sum tac tbl tee tic toe top tty ucf who xxd yes zip
```

The incorporation of other characters to patterns using the question mark ? character, including other wildcard characters, allow for more specific searches. For example, the w?? pattern will match files that started with the letter w, and are *exactly* three characters long:

```
sysadmin@localhost:/usr/bin$ echo w??
who
```

While the w??* pattern will match files that started with the letter w, but are *at least* three characters long:

```
sysadmin@localhost:/usr/bin$ echo w??*
w.procps wall watch wget whatis whereis which who whoami write
```

BRACKETS [ ] CHARACTERS

By using the *square bracket []characters*, a set of characters can be enclosed that will be used to match exactly one character. You can think of the square brackets as being like a question mark ? wildcard, but limited to the characters that are listed inside of the square brackets.

For example, using a pattern like [abcd]?? will match file names that start with an a, b, c, or d character and that are three characters long. Alternatively, letters that are *consecutive* can also be expressed as a range like the [a-d]?? pattern:

```
sysadmin@localhost:/usr/bin$ echo [a-d]??
a2p apt awk cal cmp col cut dig
```

Consider This

When the hyphen `-` character is used with square brackets, it is referred to as a *range*. For example: the `[a-z]` pattern would match any single character that is within the range of characters from `a` to `z`.

This range is defined by the ASCII text table. Simply use a lower value character from the ASCII table as the first character and a higher value character as the second.

To see the numeric value assigned to each character in the ASCII text table, either do a simple search for "ASCII text table" on the internet or view the table with the `ascii` command:

```
sysadmin@localhost:~$ ascii
```

The touch command in the following example is used to create the files that will be matched. Single quotes were used when specifying the file name to create with the touch command because within single quotes, globbing characters lose their special meaning to the shell.

```
sysadmin@localhost:/usr/bin$ cd ~/Documents
sysadmin@localhost:~/Documents$ touch '*' '**' '***' '?' '??' '???' '[][][]'
sysadmin@localhost:~/Documents$ ls
'*' School alpha-third.txt letters.txt people.csv
'**' Work alpha.txt linux.txt profile.txt
'***' '[][][]' animals.txt longfile.txt red.txt
'?' adjectives.txt food.txt newhome.txt spelling.txt
'??' alpha-first.txt hello.sh numbers.txt words
'???' alpha-second.txt hidden.txt os.csv
```

When placed inside the square brackets, the wildcard characters lose their wildcard meaning and only match themselves. So, a pattern like [*]* would match a file name that begins with an * character. A pattern of [?]* would match a file name that begins with a ? character.

```
sysadmin@localhost:~/Documents$ echo [*]*
* ** ***
sysadmin@localhost:~/Documents$ echo [?]*
? ?? ???
```

Even using square brackets inside of the square brackets will match a square bracket, so [[]]* would match a file name that starts with the square brackets:

```
sysadmin@localhost:~/Documents$ echo [[]]*
[][][]
```

Warning

Normally, you want to avoid using special characters like `*`, `?`, `[` and `]` within file names because these characters, when used in a file name on the command line, can end up matching other file names unintentionally.

Just as with the other wildcard characters, the square brackets can appear multiple times in a pattern. For example, to match a file in the /usr/bin directory that contains at least two digits in the file name, use the /etc/*[0-9][0-9]* pattern:

```
sysadmin@localhost:~/Documents$ cd /usr/bin
sysadmin@localhost:/usr/bin$ echo *[0-9][0-9]*
base64 i386 linux32 linux64 perl5.18.2 pstree.x11 sha224sum sha256sum sha384sum sha512sum x86_64
```

The square brackets also support negating the set of characters (also called complementation). If the first character inside of the square brackets is either an exclamation ! character or a caret ^ character, then that first character has a special meaning of *not the following characters*, meaning match a single character that is not within the set of characters that follows it.

In other words, the [!a-v]* pattern would match a file name that does not begin with a letter a through v:

```
sysadmin@localhost:/usr/bin$ echo [!a-v]*
2to3 2to3-2.7 2to3-3.4 [ w w.procps wall watch wc wget whatis whereis which who
whoami write x86_64 xargs xauth xgettext xsubpp xxd xz xzcat xzcmp xzdiff xzegrep xzfgrep xzgrep xzless xzmo
```

Chapter 5: File Manipulation

Everything is considered a file in Linux. Therefore, file management is an important topic. Not only are your documents considered files, but hardware devices such as hard drives and memory are considered files 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 cp command to *copy* files, the mv command to *move* or rename files, and the rm command to *remove* 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.

> **Note**
>
> While *permissions* will be covered in greater detail later in the course, they can have an impact on file management commands.
>
> Permissions determine the level of access that a user has on a file. When a user runs a program and the program accesses a file, then the permissions are checked to determine if the user has the correct access rights to the file.
>
> There are three types of permissions: *read*, *write*, and *execute*. Each has a different meaning depending on whether they are applied to a *file* or a *directory*.
>
> Read:
> - On a *file*, this allows processes to read the contents of the file, meaning the contents can be viewed and copied.
> - On a *directory*, file names in the directory can be listed, but other details are not available.
>
> Write:
> - A *file* can be written to by the process, so changes to a file can be saved. Note that the *write* permission usually requires the *read* permission on the file to work correctly with most applications.
> - On a *directory*, files can be added to or removed from the directory. Note that the *write* permission requires the *execute* permission on the directory to work correctly.
>
> Execute:
> - A *file* can be executed or run as a process.
> - On a *directory*, the user can use the `cd` command to "get into" the directory and use the directory in a pathname to access files and, potentially, subdirectories under this directory.
>
> Typically, there are two places where you should always have the *write* and *execute* permissions on the directory: your home directory (for permanent files) and the `/tmp` directory (for temporary files).
>
> *Permissions will be covered in greater detail later in the course.*

Listing Files

While technically not a file management command, the ability to *list* files using the ls command is 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
```

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

```
ls [OPTION]... [FILE]...
```

Note the use of the period . character in the following example. When used where a directory is expected, it represents the *current directory*.

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

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

Documents:
School            alpha-third.txt  hidden.txt   numbers.txt  spelling.tx
Work              alpha.txt        letters.txt  os.csv       words
adjectives.txt    animals.txt      linux.txt    people.csv
alpha-first.txt   food.txt         longfile.txt profile.txt
alpha-second.txt  hello.sh         newhome.txt  red.txt
```

However, when the period . 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 -a option will display all files, including hidden files:

```
sysadmin@localhost:~$ ls -a
.             .bashrc   .selected_editor  Downloads  Public
..            .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
.cache         Desktop          Music      Templates
```

The -d option to the ls command is 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 4096 Mar  8 02:11 /var/log
```

Listing the contents of a directory using the ls command only requires the *read* permission on the directory.

Command	File Type	Permission
`ls`	Directory	Read

Long Listing

To learn the details about a file, such as the type of file, permissions, ownership, or the timestamp, perform a *long listing*, using the -l option to the ls command. Because it provides a variety of output, the /var/log directory is listed as an example below:

```
sysadmin@localhost:~$ ls -l /var/log/
total 900
-rw-r--r-- 1 root   root  15322 Feb 22 16:32 alternatives.log
drwxr-xr-x 1 root   root   4096 Jul 19  2018 apt
-rw-r----- 1 syslog adm     560 Mar  8 02:17 auth.log
-rw-r--r-- 1 root   root  35330 May 26  2018 bootstrap.log
-rw-rw---- 1 root   utmp      0 May 26  2018 btmp
-rw-r----- 1 syslog adm     293 Mar  8 02:17 cron.log
-rw-r--r-- 1 root   adm   85083 Feb 22 16:32 dmesg
-rw-r--r-- 1 root   root 351960 Jul 19  2018 dpkg.log
-rw-r--r-- 1 root   root  32064 Feb 22 16:32 faillog
drwxr-xr-x 2 root   root   4096 Jul 19  2018 journal
-rw-rw-r-- 1 root   utmp 292584 Mar  8 02:11 lastlog
-rw-r----- 1 syslog adm   14289 Mar  8 02:17 syslog
-rw------- 1 root   root  64128 Feb 22 16:32 tallylog
-rw-rw-r-- 1 root   utmp.   384 Mar  8 02:11 wtmp
```

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

- **File Type**

  ```
  - rw-r--r-- 1 root root 15322 Feb 22 16:32 alternatives.log
  d rwxr-xr-x 1 root root  4096 Jul 19  2018 apt
  ```

  The first character of each line indicates the type of file. The file types are:

Symbol	File Type	Description
`d`	directory	A file used to store other files.
`-`	regular file	Includes readable files, image files, binary files, and compressed files.
`l`	symbolic link	Points to another file.
`s`	socket	Allows for communication between processes.
`p`	pipe	Allows for communication between processes.
`b`	block file	Used to communicate with hardware.
`c`	character file	Used to communicate with hardware.

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

- **Permissions**

```
d rwxr-xr-x 1 root root 4096 Jul 19 2018 journal
```

The next nine characters demonstrate the permissions of the file. Permissions indicate how certain users can access a file.

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

User Owner			Group Owner			Other		
Read	Write	Execute	Read	Write	Execute	Read	Write	Execute
r	w	x	r	-	x	r	-	x

- **Hard Link Count**

```
-rw-r----- 1 syslog adm 560 Mar 8 02:17 auth.log
```

There is only one directory name that links to this file.

- **User Owner**

```
-rw-r----- 1 syslog adm 293 Mar 8 02:17 cron.log
```

The syslog user owns this file. Every time a file is created, the ownership is automatically assigned to the user who created it. This is important because the owner has the *rights* to set permissions on a file.

- **Group Owner**

```
-rw-rw-r-- 1 root utmp 292584 Mar 8 02:11 lastlog
```

This indicates which group owns this file. This is important because any member of this group has a set of permissions on the file. 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 14289 Mar 8 2018 syslog
```

This displays the size of the file in bytes.

- **Timestamp**

```
-rw-rw---- 1 root   utmp      0 May 26  2018 btmp
-rw-r--r-- 1 root   adm   85083 Feb 22 16:32 dmesg
```

This indicates the time that the file's contents were last modified. This time would 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.

- **File Name**

```
-rw-r--r-- 1 root root 35330 May 26 2018 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 path name of the file that is linked is shown.

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

While listing the contents of a directory using the ls command only requires the *read* permission on the directory, viewing file details with the -l option also requires the *execute* permission on the directory.

Command	File Type	Permission
`ls -l`	Directory	Read, Execute

Sorting

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

Option	Function
`-S`	Sorts files by size
`-t`	Sorts 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 32
drwxr-xr-x 2 sysadmin sysadmin 4096 2019-02-22 16:32:34.000000000 +0000 Desktop
drwxr-xr-x 4 sysadmin sysadmin 4096 2019-02-22 16:32:34.000000000 +0000 Documents
drwxr-xr-x 2 sysadmin sysadmin 4096 2019-02-22 16:32:34.000000000 +0000 Downloads
drwxr-xr-x 2 sysadmin sysadmin 4096 2019-02-22 16:32:34.000000000 +0000 Music
drwxr-xr-x 2 sysadmin sysadmin 4096 2019-02-22 16:32:34.000000000 +0000 Pictures
drwxr-xr-x 2 sysadmin sysadmin 4096 2019-02-22 16:32:34.000000000 +0000 Public
drwxr-xr-x 2 sysadmin sysadmin 4096 2019-02-22 16:32:34.000000000 +0000 Templates
drwxr-xr-x 2 sysadmin sysadmin 4096 2019-02-22 16:32:34.000000000 +0000 Videos
```

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. To perform a recursive listing with the ls command, add the -R option.

```
sysadmin@localhost:~$ ls -lR /var/log
/var/log:
total 900

-rw-r--r-- 1 root   root  15322 Feb 22 16:32 alternatives.log
drwxr-xr-x 1 root   root   4096 Jul 19  2018 apt
-rw-r----- 1 syslog adm     560 Mar  8 02:17 auth.log
-rw-r--r-- 1 root   root  35330 May 26  2018 bootstrap.log
-rw-rw---- 1 root   utmp      0 May 26  2018 btmp
-rw-r----- 1 syslog adm     293 Mar  8 02:17 cron.log
-rw-r--r-- 1 root   adm   85083 Feb 22 16:32 dmesg
-rw-r--r-- 1 root   root 351960 Jul 19  2018 dpkg.log
-rw-r--r-- 1 root   root  32064 Feb 22 16:32 faillog
drwxr-xr-x 2 root   root   4096 Jul 19  2018 journal
-rw-rw-r-- 1 root   utmp 292584 Mar  8 02:11 lastlog
-rw-r----- 1 syslog adm   14289 Mar  8 02:17 syslog
-rw------- 1 root   root  64128 Feb 22 16:32 tallylog
-rw-rw-r-- 1 root   utmp.   384 Mar  8 02:11 wtmp

/var/log/apt:
total 144
-rw-r--r-- 1 root root  13232 Jul 19  2018 eipp.log.xz
-rw-r--r-- 1 root root  18509 Jul 19  2018 history.log
-rw-r----- 1 root adm  108711 Jul 19  2018 term.log

/var/log/journal:
total 0
```

Viewing File Types

When viewing a binary file in a CLI, the terminal may become corrupted and unable to display the output of other subsequent commands correctly. To avoid viewing binary files, use the 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, the Documents/newhome.txt file is in ASCII English text format and can be viewed without any problem:

```
sysadmin@localhost:~$ file Documents/newhome.txt
Documents/newhome.txt: ASCII text
```

In the next example, the file /bin/ls is an Executable Link Format (ELF); this file shouldn't be displayed with the cat command. While the cat command is able to output the contents of binary files, the terminal application may not handle displaying the content of binary files correctly.

```
sysadmin@localhost:~$ file /bin/ls
/bin/ls: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically lin
ked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0, BuildID[sha1]
=9567f9a28e66f4d7ec4baf31cfbf68d0410f0ae6, stripped
```

Note

The `cat` command is used to display the content of text files.

This command will be covered in greater detail later in the course.

Important

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 anymore.

The result will be strange characters being displayed in the terminal. If this occurs, use the `exit` or `logout` commands to leave the terminal you were using and then reconnect or log in again. It may also be possible to execute the `reset` command. If this does not solve the problem, closing the terminal and opening a new one will solve the problem.

Creating and Modifying Files

The touch command performs two functions. It can create an empty file or update the modification timestamp on an existing file.

```
touch [OPTION]... FILE...
```

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

```
sysadmin@localhost:~$ ls -l Documents/red.txt
-rw-r--r-- 1 sysadmin sysadmin 51 Feb 22 16:32 Documents/red.txt
sysadmin@localhost:~$ touch Documents/red.txt
sysadmin@localhost:~$ ls -l Documents/red.txt
-rw-r--r-- 1 sysadmin sysadmin 51 Mar  8 02:59 Documents/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
sysadmin@localhost:~$ ls -l newfile
-rw-rw-r-- 1 sysadmin sysadmin 0 Mar  8 03:00 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 -a option with the touch command.
- The last time the file *attributes* were *modified*. These file attributes, which include permissions and file ownership, are also called the file's *metadata*. To modify this timestamp, use the -c option with the touch command.

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

```
sysadmin@localhost:~$ touch -t 201912251200 newfile
sysadmin@localhost:~$ ls -l newfile
-rw-rw-r-- 1 sysadmin sysadmin 0 Dec 25  2019 newfile
```

In the above command, the date/time syntax for the touch command is CCYYMMDDHHMM (optionally add SS for the seconds). The table below breaks down the date/time structure used in the example above in greater detail:

Syntax	Example	Meaning
`CC`	20	The century
`YY`	19	The year
`MM`	12	The month
`DD`	25	The day
`HH`	12	The hour
`MM`	00	The minutes

Because the example above uses the precise time of 12:00 AM, we don't need to put 00 in the seconds (SS) field.

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 Documents/alpha.txt
  File: Documents/alpha.txt
  Size: 390             Blocks: 8          IO Block: 4096   regular file
Device: 802h/2050d      Inode: 5898795     Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1001/sysadmin)   Gid: ( 1001/sysadmin)
Access: 2019-02-22 16:32:34.000000000 +0000
Modify: 2019-02-22 16:32:34.000000000 +0000
Change: 2019-02-22 16:37:01.149507126 +0000
 Birth: -
```

To create a file with the touch command, you must have the *write* and *execute* permission on the parent directory. If the file already exists, modifying it with the touch command only requires the *write* permission on the file itself.

Copying Files

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; in this way, it is possible to take advantage of the existing layout and content to get started more quickly than from scratch.

The cp command is used to *copy* files. It 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 location where the copies are placed is called the *destination*.

```
cp [OPTION]... SOURCE DESTINATION
```

Recall that the tilde ~ character represents your home directory; it is commonly used as a source or destination. To view the current contents of the home directory, use the ls command:

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

To demonstrate how the cp command works, the /etc/services file can be copied to the home directory using 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. The success of the cp command can be verified using the ls command:

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

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 your home directory:

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

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:

```
cp [OPTION]... SOURCE... DIRECTORY
```

Although they may only represent one argument, a wildcard pattern can be expanded to match multiple path names. The following command copies all files in the Documents directory that start with an n to the home directory; as a result, newhome.txt and numbers.txt are copied.

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

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 the -r or -R *recursive* options.

```
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'
```

Copying a file using the cp command requires the *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 the *write* and *execute* permissions on the directory where you want to put the copied file.

Command	File Type	Permission
`cp`	Source Directory	Execute
`cp`	Source File	Read
`cp`	Destination Directory	Write, Execute

Consider This

The *archive* `-a` option of the `cp` command 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 `cp` command 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. Therefore, it can also be used to copy a directory.

Moving Files and Directories

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

```
mv [OPTION]... SOURCE DESTINATION
```

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 ~/Documents/red.txt file into the home directory and the resulting file name will be red.txt:

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

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.

```
mv [OPTION]... SOURCE... DIRECTORY
```

For example:

```
sysadmin@localhost:~/Documents$ mv animals.txt food.txt alpha.txt /tmp
sysadmin@localhost:~/Documents$ 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:~/Documents$ mv people.csv founders.csv
```

Use the mv command to move a directory from one location to another. Just like moving files, the original directory will be deleted from its previous location. In the following example, the mv command is used to move the Engineering directory from the School directory to the Work directory:

```
sysadmin@localhost:~/Documents$ cd
sysadmin@localhost:~$ ls Documents/School
Art  Engineering  Math
sysadmin@localhost:~$ mv Documents/School/Engineering Documents/Work
sysadmin@localhost:~$ ls Documents/Work
Engineering
```

Moving a file using the mv command requires the *write* and *execute* permissions on both the origin and destination directories.

Deleting Files

The rm command is used to *remove* files and directories.

```
rm [OPTION]... [FILE]...
```

Warning

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:~$ cd Documents
sysadmin@localhost:~/Documents$ rm alpha.txt
sysadmin@localhost:~/Documents$ 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 accidentally 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:~/Documents$ 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
sysadmin@localhost:~/Documents$ cd
sysadmin@localhost:~$
```

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

Deleting a file with the rm command requires the *write* and *execute* permissions on its parent directory. Regular users typically only have this type of permission in their home directory and its subdirectories.

> **Consider This**
>
> 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 `/tmp` directory, then other users of the system will not be able to delete your file.

Creating Directories

The mkdir command allows you to create (make) 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.

```
mkdir [OPTION]... DIRECTORY...
```

Typically, you 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 Ubuntu, the following directories have already been created automatically in the user's home directory:

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

The mkdir command can accept a list of space-separated path names for new directories to create:

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

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 attempted to create a directory named blue inside of the non-existent /home/sysadmin/red directory, 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 path names:

```
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:
```

To create a directory with the mkdir command, you must have the *write* and *execute* permissions on the parent of the proposed directory. For example, to create the /etc/test directory requires the *write* and *execute* permissions in the /etc directory.

Removing Directories

The rmdir command is used to *remove* empty directories:

```
rmdir [OPTION]... DIRECTORY...
```

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

```
sysadmin@localhost:~$ mkdir test
sysadmin@localhost:~$ touch test/file1.txt
sysadmin@localhost:~$ rmdir test
rmdir: failed to remove 'test': Directory not empty
sysadmin@localhost:~$ rm test
rm: cannot remove 'test': Is a directory
sysadmin@localhost:~$ rm -r test
sysadmin@localhost:~$
```

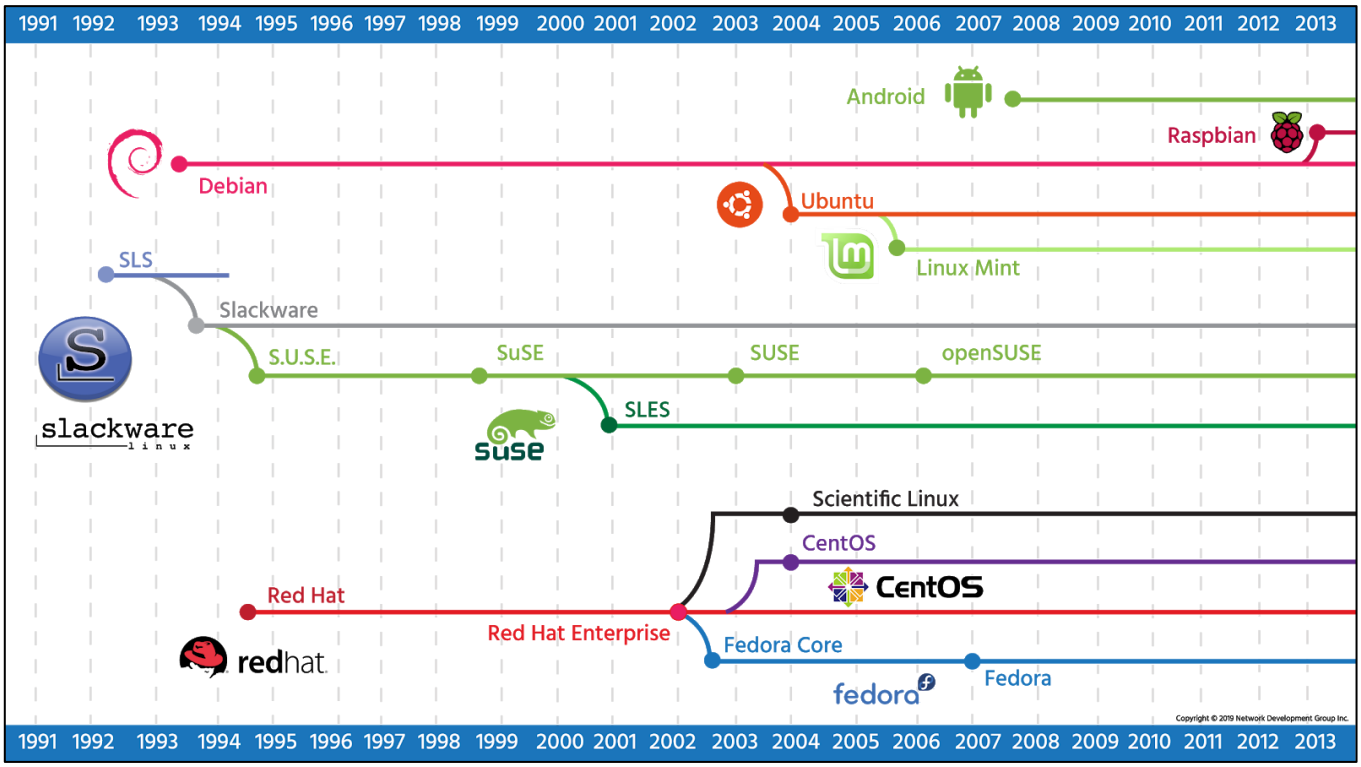
To delete a directory with the rmdir command, you must have the *write* and *execute* permissions on the parent directory. For example, to remove the /etc/test directory requires the *write* and *execute* permissions in the /etc directory.

CHAPTER 6 - FILESYSTEM HIERARCHY STANDARD (FHS)

In Linux, everything is considered a file. Files are used to store data such as text, graphics, and programs. Directories are a type of file used to store other files. This chapter will cover how to find files, which requires knowledge about the Linux directory structure, typically called a filesystem and the filesystem standards supported by the Linux Foundation; the **Filesystem Hierarchy Standard (FHS)**. You will also learn about how to find files and commands within the Linux filesystem from the command line.

FILESYSTEM HIERARCHY STANDARD

The open source licensing of many Linux components makes it possible for anyone to create their own distribution. Most people start with a well-known distribution, make modifications, and release a fork of the original.

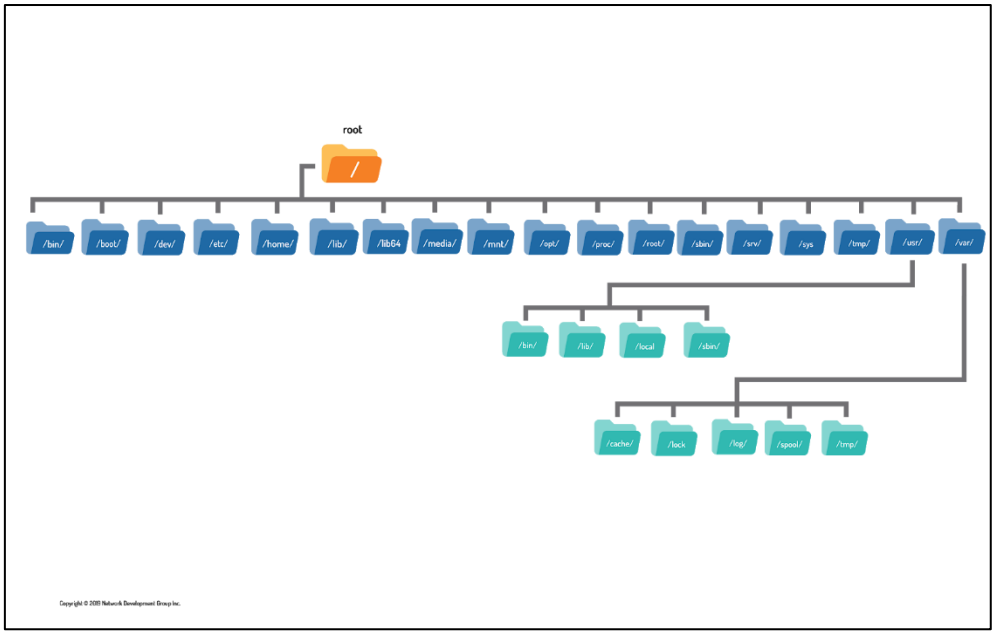


Since there are so many Linux distributions, it would be expected that numerous people would change the names of the files and folders, eventually making the distributions incompatible. This makes a basic agreement necessary concerning the naming and location of important system files and directories.

The **Filesystem Hierarchy Standard (FHS)** is an agreement to standardize the names and locations of directories and their content for use within most Linux filesystems. It helps to know what directories to expect to find, and what files one should expect to find in them. More importantly, it allows programmers to write programs that will be able to work across a wide variety of systems that conform to this standard.

During the development of the first series of this standard from 1994 to 1995, it was known as the Filesystem Standard (FSSTND). When the second series was started in 1997, it was renamed to the Filesystem Hierarchy Standard (FHS). The final 2.3 version of this second series of this FHS standard was published in 2004 at http://refspecs.linuxfoundation.org/fhs.shtml. In 2015, a version of the third series of this standard was published at http://www.linuxbase.org/betaspecs/fhs/fhs.html.

The Linux file structure is best visualized as an upside-down tree, with directories and files branching out from the top-level root / directory. While the actual standard details many more directories than listed below, the image and table highlight some of the most important ones to know.



Directory	Purpose
/	The root of the primary filesystem hierarchy.
/bin	Contains essential user binary executables.
/boot	Contains the kernel and bootloader files.
/dev	Populated with files representing attached devices.
/etc	Configuration files specific to the host.
/home	Common location for user home directories.
/lib	Essential libraries to support /bin and /sbin executables.
/mnt	Mount point for temporarily mounting a filesystem.
/opt	Optional third-party add-on software.
/root	Home directory for the root user.
/sbin	Contains system or administrative binary executables.
/srv	May contain data provided by system services.
/tmp	Location for creating temporary files.
/usr	The root of the secondary filesystem hierarchy.
/usr/bin	Contains the majority of the user commands.
/usr/include	Header files for compiling C-based software.
/usr/lib	Shared libraries to support /usr/bin and /usr/sbin .
/usr/local	The root of the third filesystem hierarchy for local software.
/usr/sbin	Non-vital system or administrative executables.
/usr/share	Location for architecturally-independent data files.
/usr/share/dict	Word lists.
/usr/share/doc	Documentation for software packages.
/usr/share/info	Information pages for software packages.
/usr/share/locale	Locale information.
/usr/share/man	Location for man pages.
/usr/share/nls	Native language support files.

Vendors of Linux distributions have continued to make some changes, even though a new version of the standard has not been published in over ten years. Two notable new additions include the /run directory and the /sys directory. The /run directory is being considered for use in the forthcoming FHS versions to contain volatile data that changes at runtime. Previously, this data was supposed to be stored under the /var/run directory, but due to the unavailability of this directory at boot time, this data can become scattered in other places, such as hidden files in the /dev directory.

The /sys directory in some traditional UNIX systems was used to hold files related to the kernel. In modern Linux systems, the /sys directory is used to mount the sysfs pseudo-filesystem. This filesystem is used to export information about kernel objects and their relationships to each other. The kernel objects are represented by directories, and the files that they contain are named for the attributes of those objects. The contents of the files represent the value for that attribute. Symbolic links are used to represent relationships between objects.

Another notable change that some Linux distributions are making is the conversion of the /bin, /sbin and /lib directories into symbolic links which point to /usr/bin, /usr/sbin and /usr/lib, respectively. All user executables are now in the /usr/bin directory, administrator executables are now in the /usr/sbin directory, and the libraries to support all these executables are now in the /usr/lib directory.

> **Note**
>
> A *symbolic link*, also called a *soft link*, is simply a file that points to another file.
>
> *Links will be covered in greater detail later in the course.*

The merger of the /bin, /sbin and /lib directories into the /usr/bin, /usr/sbin and /usr/lib directories has been somewhat controversial. Many administrators are comfortable with the long-standing subdivisions of these files into different directories.

Because of the way that UNIX booted, the /bin, /sbin and /lib directories had to be part of the root filesystem as they contain critical boot executables. Some developers now argue that the reason for having them split is no longer valid. In early versions of UNIX, the developers had two filesystems of about 1.5 MiB each on two separate disks for the root filesystem and the /usr filesystem. As the root filesystem started to become full, the developers decided to move some of the executable files that were in the /bin and /sbin directories that were not essential to booting the system into the corresponding directories /usr/bin and /usr/sbin (in the separate /usr filesystem).

The FHS standard categorizes each system directory in a couple of ways for security purposes:

Shareable / Unshareable

- Shareable files can be stored on one host and used on others. For instance, /var/www is often used as the root directory of a web server, which shares files with other hosts. Another example is the user home directories.
- Unshareable files should not be shared between hosts. These include process states in the /var/run directory and the /boot directory.

Variable / Static

- Static files generally do not change, including library files and documentation pages. An example is the info pages located at /usr/share/info.
- Variable files normally change during the execution of programs. The /var/run directory contains files that are both variable and unshareable.

The table below summarizes the main distinctions between file types:

	Shareable	Unshareable
Static	`/usr`	`/etc`
	`/opt`	`/boot`
Variable	`/var/mail`	`/var/run`
	`/var/spool/news`	`/var/lock`

Finding Files and Commands

Most operating systems allow users to search for specific files in the filesystem. A GUI typically provides a search tool that makes it possible to find files and applications. However, there are a few commands available for searching files and commands from the CLI. Both the locate and find commands are useful for searching for a file within the filesystem. While both commands perform similar tasks, each does so by using a different technique, with its own distinct advantages and disadvantages.

locate Command

Of the two main search commands, the locate command can be described as fast, but always potentially out-of-date. Its speed comes from the fact that the locate command searches a database that contains the location of the files on the filesystem, but that database needs to be updated to be accurate.

```
locate [OPTION]... PATTERN...
```

In its simplest form, the locate command accepts a search string as an argument. For example, to find a file named passwd, use the following command:

```
sysadmin@localhost:~$ locate passwd
/etc/passwd
/etc/passwd-
/etc/pam.d/chpasswd
/etc/pam.d/passwd
/etc/security/opasswd
/usr/bin/gpasswd
/usr/bin/passwd
/usr/lib/tmpfiles.d/passwd.conf
/usr/sbin/chgpasswd
/usr/sbin/chpasswd
/usr/sbin/update-passwd
/usr/share/base-passwd
/usr/share/base-passwd/group.master
/usr/share/base-passwd/passwd.master
/usr/share/doc/base-passwd
/usr/share/doc/passwd
/usr/share/doc/base-passwd/README
/usr/share/doc/base-passwd/changelog.gz
/usr/share/doc/base-passwd/copyright
/usr/share/doc/base-passwd/users-and-groups.html
/usr/share/doc/base-passwd/users-and-groups.txt.gz
/usr/share/doc/passwd/NEWS.Debian.gz
```

A few things to consider when using the locate command:

- The locate command will only return results of files that the current user would normally have access to.
- The locate command will display all files that have the search term anywhere in the file name. For example, the search term passwd would match both /etc/passwd and /etc/thishaspasswdinit.
- Like most things in Linux, the locate command is case sensitive. For example, the search term passwd would not match a file named /etc/PASSWD. To have the locate command not be case sensitive, use the -i option.

> **Note**
>
> The `locate` command accepts the `-r` option to use regular expressions in the search pattern which provides a more powerful way to search for files.
>
> *Regular expressions will be covered in greater detail later in the course.*

The locate command is dependent on a database. This database can be updated manually by an administrator using the updatedb command, though typically this command is run automatically every day through *cron*, a system scheduling service that runs commands on a particular recurring schedule.

When executed, the updatedb command creates a database of all files that it finds on the computer for quick searching. This command can only be executed by a user with administrative access, which can be achieved using the sudo command:

```
sysadmin@localhost:~$ sudo updatedb
[sudo] password for sysadmin:
```

Note

The `sudo` command allows users to execute commands using the privileges of another user. The `sudo` command is the preferred way to run commands with escalated privileges, and the root user is assumed by default.

The `sudo` command circumvents the need to login as root; this is bad practice because all commands are executed with root privileges. This is seldom necessary and increases the risk that potentially dangerous commands may be executed as root, even though root privileges were not required or desired.

Accordingly, some Linux distributions now prevent root logins entirely and require `sudo` to escalate privileges. The system administrator must specifically authorize each user that can use the `sudo` command and must specify which user(s) they can impersonate. For Linux system administrators, `sudo` plays a vital role and is among the most frequently-used commands.

The updatedb command can be told not to search a particular name, path, or filesystem by changing the corresponding line in its configuration file, /etc/updatedb.conf. Below is an example of the default file in its entirety:

```
sysadmin@localhost:~$ cat /etc/updatedb.conf
PRUNE_BIND_MOUNTS="yes"
# PRUNENAMES=".git .bzr .hg .svn"
PRUNEPATHS="/tmp /var/spool /media /var/lib/os-prober /var/lib/ceph /home/.ecryp
tfs /var/lib/schroot"

PRUNEFS="NFS nfs nfs4 rpc_pipefs afs binfmt_misc proc smbfs autofs iso9660 ncpfs
 coda devpts ftpfs devfs devtmpfs fuse.mfs shfs sysfs cifs lustre tmpfs usbfs ud
f fuse.glusterfs fuse.sshfs curlftpfs ceph fuse.ceph fuse.rozofs ecryptfs fusesm
b"
```

The /etc/updatedb.conf file can be edited as the root user. Any name, path, or filesystem that is listed in the file on the appropriate line will not be added to the database. Any line starting with the # symbol will be ignored since it is commented out.

Since the locate command works with a database, it is able to work very quickly even on a system with hundreds of thousands of files. However, if you want to use the locate command to search for a file that was created very recently, it will fail to find the file if the database hasn't been updated since the file creation.

Likewise, the database may contain outdated information about files that might have existed in the very recent past, so the command will report them incorrectly as still existing.

The following example demonstrates the consequences that arise when the database is not updated in real time:

1. A new file called lostfile isn't initially found by the locate command.

```
sysadmin@localhost:~$ touch lostfile
sysadmin@localhost:~$ locate lostfile
```

2. After the database is updated by the updatedb command, the locate command can find the lostfile file.

```
sysadmin@localhost:~$ sudo updatedb
[sudo] password for sysadmin:
sysadmin@localhost:~$ locate lostfile
/home/sysadmin/lostfile
```

3. After the lostfile file has been deleted, the locate command will still report that the file exists.

```
sysadmin@localhost:~$ rm lostfile
sysadmin@localhost:~$ locate lostfile
/home/sysadmin/lostfile
```

find Command

If you want to search for files that are currently in the filesystem, then you should use the find command. The find command is slower than the locate command because it searches directories in real time; however, it doesn't suffer from problems associated with an outdated database.

The find command expects a directory as the first argument. This will be the starting point of the search. The find command will search this directory and all of its subdirectories. If no directory is specified, then the find command will start the search at the current directory.

```
find [OPTIONS]... [starting-point...] [expression]
```

Note that the period . character refers to the current directory in the following example, but the current directory could also be referred to using the ./ notation. The find command uses the -name option to search for files by name, in this case, the Downloads directory.

```
sysadmin@localhost:~$ find . -name Download
sysadmin@localhost:~$ find . -name Downloads
./Downloads
sysadmin@localhost:~$ find . -name Dow*
./Downloads
sysadmin@localhost:~$ find . -name 'Do*'
./Documents
./Downloads
```

The first search yielded no results because the string must match the exact name of the file, not just part of the name. The third command demonstrates that globbing can be used, and the fourth command demonstrates multiple matches (notice that single quotes were added so that the find command will interpret the glob, rather than the shell).

If the search for the Downloads directory was conducted from the root / directory, many errors would arise:

```
sysadmin@localhost:~$ find / -name Downloads
find: '/var/lib/apt/lists/partial': Permission denied
find: '/var/cache/ldconfig': Permission denied
find: '/var/cache/apt/archives/partial': Permission denied
find: '/var/spool/rsyslog': Permission denied
find: '/var/spool/cron/crontabs': Permission denied
find: '/etc/ssl/private': Permission denied
find: '/root': Permission denied
Output Omitted...
```

These errors can be ignored for now, but just be aware that errors like these are typical when either a regular user is attempting to search root-only areas, or when the root user is attempting to search areas that are dedicated to the system's processes.

The find command also offers many options for searching files, unlike the locate command which searches only for files based on file name. The following table illustrates some of the more commonly used criteria:

Example	Meaning
`-iname LOSTFILE`	Case insensitive search by name.
`-mtime -3`	Files modified less than three days ago.
`-mmin -10`	Files modified less than ten minutes ago.
`-size +1M`	Files larger than one megabyte.
`-user joe`	Files owned by the user joe.
`-nouser`	Files not owned by any user.
`-empty`	Files that are empty.
`-type d`	Files that are directory files.
`-maxdepth 1`	Do not use recursion to enter subdirectories; only search the primary directory.

If multiple criteria are specified, then all criteria must match as the find command automatically assumes a logical *AND* condition between criteria, meaning all of the conditions must be met. This could be explicitly stated by using the -a option between criteria. For example, the Downloads directory must also be owned by the sysadmin user in order for the find command to produce a result in the following example:

```
sysadmin@localhost:~$ find . -user sysadmin -a -name Downloads
./Downloads
```

Logical *OR* conditions can be specified between criteria with the -o option, meaning at least one of the conditions must be true. The following output lists files that are either named Downloads or owned by the sysadmin user.

```
sysadmin@localhost:~$ find . -user sysadmin -o -name Downloads
.
./Public
./Downloads
./Pictures
./Videos
./.selected_editor
./.profile
 ./.bash_logout
...
```

Logical groupings of criteria can also be specified using parentheses. Be careful to precede the parentheses with a backslash or to use single quotes around them so that the shell doesn't attempt to interpret them as special characters. Also, as mentioned previously, use quotes around any globs that the find command should interpret instead of the shell, as shown in the following example:

```
sysadmin@localhost:~$ find . -iname 'desk*' -o \( -name Downloads -a -user sysadmin \)
./Desktop
./Downloads
```

In plain text, the find command in the preceding example will return files in the current directory, with the non-case sensitive name starting with desk *OR* files with the exact name Downloads owned by the sysadmin user.

By default, the find command simply prints the names of the files that it finds. To generate output showing additional file details, similar to the ls -l command, you can specify the -ls option:

```
sysadmin@localhost:~$ find . -ls -iname 'desk*' -o \( -name Downloads -a -user sysadmin \)
209950600      4 drwxr-xr-x   1 sysadmin sysadmin     4096 Mar 10 04:08 .
210019331      4 drwxr-xr-x   2 sysadmin sysadmin     4096 Mar  8 19:10 ./Templates
102108986      4 -rw-r--r--   1 sysadmin sysadmin      220 Apr  4  2018 ./.bash_logout
102108988      4 -rw-r--r--   1 sysadmin sysadmin      807 Apr  4  2018 ./.profile
210019332      4 drwxr-xr-x   2 sysadmin sysadmin     4096 Mar  8 19:10 ./Downloads
 ...
```

To make the output exactly like the output of the ls -l command, use the -exec option to execute ls -l on each file found. For example:

```
sysadmin@localhost:~$ find -name 'Documents' -exec ls -l {} \;
total 1100
drwxr-xr-x 5 sysadmin sysadmin   4096 Mar  8 19:10 School
drwxr-xr-x 2 sysadmin sysadmin   4096 Mar  8 19:10 Work
-rw-r--r-- 1 sysadmin sysadmin     39 Mar  8 19:10 adjectives.txt
-rw-r--r-- 1 sysadmin sysadmin     90 Mar  8 19:10 alpha-first.txt
-rw-r--r-- 1 sysadmin sysadmin    106 Mar  8 19:10 alpha-second.txt
-rw-r--r-- 1 sysadmin sysadmin    195 Mar  8 19:10 alpha-third.txt
...
```

The previous example tells the find command to execute the ls -l command for each file found. The pair of curly braces {} is a placeholder for the name of each file found; note that there is no space between them. The \; is an escaped semicolon that is added between each command so that multiple commands may be executed in series.

In order to make the find command confirm the execution of the command for each file found, use the action option -ok instead of the -exec option:

```
sysadmin@localhost:~$ touch one two three
sysadmin@localhost:~$ find . -mmin -2 -ok rm {} \;
< rm ... . > ? n
< rm ... ./one > ? y
< rm ... ./two > ? y
< rm ... ./three > ? y
```

The command above tells the find command to execute the rm command for files that were modified less than two minutes ago. Did you notice how the first file that the rm command tried to remove was the current directory (represented by the period. character)? That is enough reason not to use the -exec option because the rm command would have tried to remove the current directory.

There will be situations where it's useful to find a file or set of files depending on their age, or when they were last modified, such as all files that have been changed since a given time (such as the last backup). The *modification time* -mtime option to the find command provides the ability to give time as criteria for a search.

First, a word about how the find command uses times is relevant here. When looking for files modified within the period of a day, such as 3 days ago, you would use -mtime 3, but if you are looking for files that have changed anytime between 3 days ago and now, you would use -mtime -3. If you are looking to find files older than 3 days, you would use -mtime +3.

Option	Function
`-mtime N`	Files modified *N* *24 hours ago.
`-mtime -N`	Files modified less than *N* *24 hours ago.
`-mtime +N`	Files modified more than *N* *24 hours ago.

To summarize, a number without a + or - means exactly *N* days ago, a number with a - means anytime between *N* days ago and NOW, and a number with a + means *N* days ago or more.

For example, if you backup every seven days, use the following command to backup all files that are 3 days or less old:

```
sysadmin@localhost:~$ find /tmp -mtime -3
/tmp
```

The above command starts in the /tmp directory and finds all files that have been modified in the last three 24-hour periods, or 72 hours.

The find command is less exact by default than you might want it to be. When using numbers of days with the -atime, -ctime or -mtime operators, the days refer to 24-hour blocks. Therefore, you may think that you are searching to find all files that occurred between now and your last backup by referring to that day, but depending on the time of day, you may not have found all of the files.

Since the find command's time operators are effectively tied to 24-hour blocks or days, you can use a reference file to make certain you have found all files that occurred precisely since your last backup.

To find all files that are newer than your last backup, find out the date and time of that backup, and create a reference file with that date/time so you can use it with the find -newer command. For example, if the backup was done at precisely 2300 (11:00 PM) on the 1st of March, 2020, you can create a reference file with that exact date and time with the following command:

```
sysadmin@localhost:~$ touch -t 202003012300 /tmp/reference_file
```

Now that we have a file that mirrors precisely the date and time of our last backup, we can search for all files in the /home directory that might have been created or modified since the last backup date with the following command:

```
sysadmin@localhost:~$ find /home -newer /tmp/reference_file
/home
/home/sysadmin
/home/sysadmin/Templates
/home/sysadmin/Downloads
/home/sysadmin/Music
/home/sysadmin/.selected_editor
/home/sysadmin/Documents
/home/sysadmin/Documents/alpha-third.txt
/home/sysadmin/Documents/spelling.txt
/home/sysadmin/Documents/profile.txt
/home/sysadmin/Documents/numbers.txt
/home/sysadmin/Documents/animals.txt
/home/sysadmin/Documents/hidden.txt
/home/sysadmin/Documents/people.csv
/home/sysadmin/Documents/alpha.txt
...
```

Note

The output will vary, depending on what your system includes for users and what they have been doing, but typically at least the current user's `.bash_history` file will appear as having been changed.

The find command also allows the use of friendly notation to find files that are larger or smaller than a particular size. Trying to find files that are above 1 Kilobyte on a busy system is like searching for the word “the” on Google; so much output is returned it’s essentially useless. When searching for files of a given size, or those that are smaller or larger than a given size, try to restrict the search to the smallest valid set of directories and files possible.

For example, to find all files that are smaller than 1 Kilobyte in the /etc directory, you would use:

```
sysadmin@localhost:~$ find /etc -size -1k
/etc/.pwd.lock
/etc/security/opasswd
/etc/apparmor.d/local/sbin.dhclient
/etc/apparmor.d/local/usr.sbin.rsyslogd
/etc/apparmor.d/local/usr.sbin.named
/etc/apparmor.d/local/usr.bin.man
find: '/etc/ssl/private': Permission denied
/etc/newt/palette.original
```

If the files are smaller than 1 Kilobyte, they will be listed as output. You can also use the -size option to find all files on the system that are 1 Megabyte or more with the command:

```
sysadmin@localhost:~$ find / -size +1M
find: '/proc/tty/driver': Permission denied
find: '/proc/1/task/1/fd': Permission denied
find: '/proc/1/task/1/fdinfo': Permission denied
find: '/proc/1/task/1/ns': Permission denied
find: '/proc/1/fd': Permission denied
find: '/proc/1/map_files': Permission denied
find: '/proc/1/fdinfo': Permission denied
find: '/proc/1/ns': Permission denied
...
```

Consider This

The `-size` option rounds up, to ensure that the `find` command will display the results you want.

For example, if you have three files, one less than 512 Kilobytes in size, one above 512 Kilobytes but less than 1024 Kilobytes in size, and one that is above 1024 Kilobytes in size, the `-size` option will not show anything for the output of `-size -1M`, since it rounds down to 0.

The output of `-size -512K` would show only the file that is 512K or less, and if you run the `-size +512K` command, the output would show both the file that is 512K or more and the file that is 1024K or more. You may wish to experiment with this option so you know what to expect.

The find command can also help you find files by specifying a file type, such as regular file, directory, and more, using the -type option. The table below shows all the different types of files you can find using the -type option:

Symbol	File Type
b	Block (i.e., disks, storage)
c	Character (i.e., keyboards, scanners, mice)
d	Directory (directory files)
p	Named pipes (Allows for communication between processes)
f	Regular file (i.e., scripts, text files, graphics)
s	Sockets (Allows for communication between processes)

To demonstrate, in order to search only for directories under the /home directory, you could use the following command:

```
sysadmin@localhost:~$ find /home/sysadmin -type d
/home/sysadmin
/home/sysadmin/Desktop
/home/sysadmin/Templates
/home/sysadmin/Documents
/home/sysadmin/Documents/Work
/home/sysadmin/Documents/School
/home/sysadmin/Documents/School/Engineering
/home/sysadmin/Documents/School/Art
/home/sysadmin/Documents/School/Math
/home/sysadmin/Music
/home/sysadmin/Public
/home/sysadmin/Pictures
/home/sysadmin/Videos
/home/sysadmin/Downloads
/home/sysadmin/.cache
```

The command above will search the /home/sysadmin directory tree structure and filter out all results except for files that are the specified directory type.

Consider This

To limit the search to just a single level below the `/home` directory, the `-maxdepth` option to the `find` command can be used. The `-maxdepth` option can be set to almost any numeric value. For example, a depth of `1` will show the top-level directories in the `/home/sysadmin` directory:

```
sysadmin@localhost:~$ find /home/sysadmin -type d -maxdepth 1
```

whereis Command

There are many commands available to the operating system, and it is sometimes useful to know where they are. The whereis command can be used to search your PATH for the binary. It is also capable of searching for the man page, and source files for any given command.

```
whereis [OPTION]... NAME...
```

The echo command can be used to determine which directories are in the PATH:

```
sysadmin@localhost:~$ echo $PATH
/home/sysadmin/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games
```

To find out where the grep command is located, use whereis without any options:

```
sysadmin@localhost:~$ whereis grep
grep: /bin/grep /usr/share/man/man1/grep.1.gz /usr/share/info/grep.info.gz
```

The output of the whereis command returns three directories. The first is where the grep command is located, in the /bin/grep directory. There is also a path given for the man page of the grep command at /usr/share/man/man1/grep.1.gz and another path for the info page at /usr/share/info/grep.info.gz. Without an option, the whereis command provides all three pieces of information.

To view the location of the binary separate from the man page and info page, use the -b and -m options respectively:

```
sysadmin@localhost:~$ whereis -b grep
grep: /bin/grep
sysadmin@localhost:~$ whereis -m grep
grep: /usr/share/man/man1/grep.1.gz /usr/share/info/grep.info.gz
```

The -s option can be used to find source code that has been installed for a given command. In the case of grep, only the binary is currently installed:

```
sysadmin@localhost:~$ whereis -s grep
grep:
```

Notice that no results are returned because the source code for the grep command is not installed.

The -u option can be used in one of two ways. It can identify commands that do not have documentation (i.e. a man page) for a requested attribute. It can also be used to identify commands that have more than one documentation file. For example, to find out which commands in the /bin directory either do not have a man page, or have more than one documentation file, use the following command:

```
sysadmin@localhost:~$ cd /bin
sysadmin@localhost:/bin$ whereis -m -u *
dir: /usr/share/man/man1/dir.1.gz /usr/share/info/dir /usr/share/info/dir.old
grep: /usr/share/man/man1/grep.1.gz /usr/share/info/grep.info.gz
gzip: /usr/share/man/man1/gzip.1.gz /usr/share/info/gzip.info.gz
hostname: /usr/share/man/man7/hostname.7.gz /usr/share/man/man5/hostname.5.gz /u
sr/share/man/man1/hostname.1.gz
ip: /usr/share/man/man7/ip.7.gz /usr/share/man/man8/ip.8.gz
nano: /usr/share/man/man1/nano.1.gz /usr/share/info/nano.info.gz
sed: /usr/share/man/man1/sed.1.gz /usr/share/info/sed.info.gz
sysadmin@localhost:/bin$ cd
sysadmin@localhost:~$
```

In the previous example, the asterisk * character is used to consider every file in the current directory.

The example below identifies a specific command that does not have a man page:

```
sysadmin@localhost:~$ whereis type
```

This means that there is no man page for the type command:

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

If the same example is repeated for the ls command, there will be no output because ls has a man page.

```
sysadmin@localhost:~$ whereis ls
ls: /bin/ls /usr/share/man/man1/ls.1.gz
```

To limit the search to a specific path or paths, capitalized options can be used. The -B option searches for binaries, the -M option for manuals and documentation, and the -S option for sources. The -f option must be used to indicate the end of the path list and the beginning of the search term.

```
whereis [OPTION]... [-BMS directory… -f] name...
```

```
sysadmin@localhost:~$ whereis -B /home/sysadmin/ -f Desktop
Desktop: /home/sysadmin//Desktop
```

This type of searching, however, may be best left to the find command, as it allows for more specific criteria to be used in a search.

which Command

Sometimes the whereis command returns more than one result on a command. In this case, an administrator would want to know which command is being used.

```
which [-a] FILENAME...
```

The bash command is an example that normally yields two results:

```
sysadmin@localhost:~$ whereis -b bash
bash: /bin/bash /etc/bash.bashrc
```

To find out which bash result is the real command, in other words, the result that is used when executing the command, use the which command:

```
sysadmin@localhost:~$ which bash
/bin/bash
```

The -a option can be used with the which command to locate multiple executable files. This would be useful to know if an executable script was inserted maliciously to override an existing command.

```
sysadmin@localhost:~$ which -a ping
/bin/ping
```

By using the which command, an administrator can be fairly certain that the only executable running by the name of the ping command is located in the /bin directory.

type Command

The type command can be used to determine information about various commands.

```
type [OPTION]... NAME...
```

Some commands originate from a specific file:

```
sysadmin@localhost:~$ type which
which is /usr/bin/which
```

This output would be similar to the output of the which command:

```
sysadmin@localhost:~$ which which
/usr/bin/which
```

The type command can also identify commands that are built into the Bash (or other) shell:

```
sysadmin@localhost:~$ type echo
echo is a shell builtin
```

This output is significantly different than the output of the which command:

```
sysadmin@localhost:~$ which echo
/bin/echo
```

Using the -a option, the type command can also reveal the path of another command:

```
sysadmin@localhost:~$ type -a echo
echo is a shell builtin
echo is /bin/echo
```

The type command can also identify aliases to other commands:

```
sysadmin@localhost:~$ type ll
ll is aliased to `ls -alF'
sysadmin@localhost:~$ type ls
ls is aliased to `ls --color=auto'
```

The output of these commands indicates that ll is an alias for ls -alF, and even ls is an alias for ls --color=auto. Again, the output is significantly different than the which command:

```
sysadmin@localhost:~$ which ll
sysadmin@localhost:~$ which ls
/bin/ls
```

The type command supports other options, and can lookup multiple commands simultaneously. To display only a single word describing the echo, ll, and which commands, use the -t option:

```
sysadmin@localhost:~$ type -t echo ll which
builtin
alias
file
```

CHAPTER 7 - TEXT UTILITIES

A core philosophy that started with UNIX, and lives on in Linux, is that commands should read text as input and write text as output. To remain true to this philosophy, a large number of commands have been designed with the primary purpose of processing text files. As you will see, many of these commands act as filters, as they somehow alter the text that they read before they produce their output.

With a single file argument, the cat command will simply output the contents of a file:

```
cat [OPTION]... [FILE]...
```

```
sysadmin@localhost:~$ cd Documents
sysadmin@localhost:~/Documents$
sysadmin@localhost:~/Documents$ cat alpha-first.txt
A is for Animal
B is for Bear
C is for Cat
D is for Dog
E is for Elephant
F is for Flower
sysadmin@localhost:~/Documents$ cat alpha-second.txt
G is for Grapes
H is for Happy
I is for Ink
J is for Juice
K is for Kangaroo
L is for Lol
M is for Monkey
```

However, the cat command gets its name from the word *concatenate*, meaning to merge or append. This command has the ability to accept two or more files as input and then output the concatenation (merging) of those files:

```
sysadmin@localhost:~/Documents$ cat alpha-first.txt alpha-second.txt
A is for Animal
B is for Bear
C is for Cat
D is for Dog
E is for Elephant
F is for Flower
G is for Grapes
H is for Happy
I is for Ink
J is for Juice
K is for Kangaroo
L is for Lol
M is for Monkey
```

While the cat command is often used to quickly view the contents of small files, using it to display the contents of larger files will result in frustration since the data will be displayed much too quickly and will scroll off the screen. To display the file one page at a time, use a *pager*, either the more or the less command.

To view a file with the less command, pass the file name as an argument:

```
less FILE
```

```
sysadmin@localhost:~/Documents$ less words
```

The more and less commands allow users to navigate the document using keystroke commands. For example, when using less as a pager, to advance forward a page, press the **Spacebar**.

The less command might look familiar since it is the pager generally used to view man pages. As you may recall, the less command requires the use of movement commands to navigate a file. When viewing a file with the less command, use the **H** key or **Shift + H** to display a help screen with a summary of the less commands:

```
                SUMMARY OF LESS COMMANDS
      Commands marked with * may be preceded by a number, N.

      Notes in parentheses indicate the behavior if N is given.

  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).
HELP -- Press RETURN for more, or q when done
```

To exit the less command, use the **Q** key.

Splitting Files

While the cat command can concatenate two or more files into one file, the split command can take one file and *split* it into multiple files. One application of the split command would be to take a file that is too large to fit on some kind of removable media and break that file apart so that each piece could be stored on separate media. This way, the pieces of the split files can each fit on a removable medium for transfer to another system.

This can also be a useful way to send larger files across a slow network that has connection issues. Breaking files into smaller pieces would make it easier to transfer the data across a poor network connection. Once the pieces are copied to the other system, the cat command can be used to reassemble the pieces into a single file once again.

The syntax for the split command is:

```
split [OPTION]... [INPUT [PREFIX]]
```

The INPUT argument is typically a file name, but the split command is able to read from standard input as well.

By default, the new files will be named with a prefix of x and an alphabetical suffix of aa, ab, etc.:

```
sysadmin@localhost:~/Documents$ split longfile.txt
sysadmin@localhost:~/Documents$ ls
School            alpha.txt    linux.txt     profile.txt   xac  xai
Work              animals.txt  longfile.txt  red.txt       xad  xaj
adjectives.txt    food.txt     newhome.txt   spelling.txt  xae  xak
alpha-first.txt   hello.sh     numbers.txt   words         xaf
alpha-second.txt  hidden.txt   os.csv        xaa           xag
alpha-third.txt   letters.txt  people.csv    xab           xah
```

As indicated by the command syntax, it is possible to specify a prefix other than x. For example, the following example denotes file. as the prefix:

```
sysadmin@localhost:~/Documents$ split longfile.txt file.
sysadmin@localhost:~/Documents$ ls
School            animals.txt  file.ag   hidden.txt    people.csv    xac  xaj
Work              file.aa      file.ah   letters.txt   profile.txt   xad  xak
adjectives.txt    file.ab      file.ai   linux.txt     red.txt       xae
alpha-first.txt   file.ac      file.aj   longfile.txt  spelling.txt  xaf
alpha-second.txt  file.ad      file.ak   newhome.txt   words         xag
alpha-third.txt   file.ae      food.txt  numbers.txt   xaa           xah
alpha.txt         file.af      hello.sh  os.csv        xab           xai
```

The -d option will allow for the split files to have a numeric suffix instead of a default alphabetic suffix. If -d is added to the previous command, the resulting file names would be:

```
sysadmin@localhost:~/Documents$ split -d longfile.txt file.
sysadmin@localhost:~/Documents$ ls
School            file.00  file.08  file.af   hidden.txt    profile.txt   xae
Work              file.01  file.09  file.ag   letters.txt   red.txt       xaf
adjectives.txt    file.02  file.10  file.ah   linux.txt     spelling.txt  xag
alpha-first.txt   file.03  file.aa  file.ai   longfile.txt  words         xah
alpha-second.txt  file.04  file.ab  file.aj   newhome.txt   xaa           xai
alpha-third.txt   file.05  file.ac  file.ak   numbers.txt   xab           xaj
alpha.txt         file.06  file.ad  food.txt  os.csv        xac           xak
animals.txt       file.07  file.ae  hello.sh  people.csv    xad
```

By default, the split command will break apart a file into 1,000 line chunks. The first 1,000 lines of the original file will go into the first file, the second 1,000 lines will go into the second file, etc. The above file longfile.txt splits into 11 files, meaning it is approximately 11,000 lines long.

The -l option can be used to specify the number of lines to split upon. Alternatively, the -b option may be used to specify the maximum number of bytes to use per file.

Numbering the Output of Files

The nl command will *number the lines* of its output.

```
nl [OPTION]... [FILE]...
```

By default, it only numbers non-empty lines:

```
sysadmin@localhost:~/Documents$ nl newhome.txt
     1  Thanks for purchasing your new home!!

     2  **Warning** it may be haunted.

     3  There are three bathrooms.

     4  **Beware** of the ghost in the bedroom.

     5  The kitchen is open for entertaining.

     6  **Caution** the spirits don't like guests.

     7  Good luck!!!
```

The nl command offers many formatting options for line numbering, including separate styles for headers, footers, sections, and the body of a file. For example, to have the nl command number *every* line in the newhome.txt file, execute the following command:

```
sysadmin@localhost:~/Documents$ nl -ba newhome.txt
     1  Thanks for purchasing your new home!!
     2
     3  **Warning** it may be haunted.
     4
     5  There are three bathrooms.
     6
     7  **Beware** of the ghost in the bedroom.
     8
     9  The kitchen is open for entertaining.
    10
    11  **Caution** the spirits don't like guests.
    12
    13  Good luck!!!
```

In the previous example, the option -ba may cause some confusion as it is really just one option, not two. The -b option is the --body-numbering option, abbreviated in its short version as -b. The a is actually an argument that means *number all lines*. In other words, the command could also be executed:

```
sysadmin@localhost:~/Documents$ nl -b a newhome.txt
```

This is one of the reasons some Linux users prefer the long argument technique. Using a long argument, the command could have been written:

```
sysadmin@localhost:~/Documents$ nl --body-numbering=a newhome.txt
```

Consider This

It is also possible to number lines using the `cat` command with the `-n` option, which numbers *every* line by default.

Displaying the Beginning of a File

The purpose of the head command is to view the *beginning* of a file or output.

```
head [OPTION]... [FILE]...
```

By default, the head command will display the first ten lines of a file's contents. For example, the following command displays the first ten lines of the alpha.txt file:

```
sysadmin@localhost:~/Documents$ head alpha.txt
A is for Apple
B is for Bear
C is for Cat
D is for Dog
E is for Elephant
F is for Flower
G is for Grapes
H is for Happy
I is for Ink
J is for Juice
```

There are several options for the head command that are useful. For instance, there is the ability to use a number as an option to indicate how many lines of output to display. For example, to display the first three lines of the alpha.txt file, execute:

```
sysadmin@localhost:~/Documents$ head -3 alpha.txt
A is for Apple
B is for Bear
C is for Cat
```

There is also the -n option which takes an argument for the *number* of lines to display. So, the following would also display the first three lines of the alpha.txt file:

```
sysadmin@localhost:~/Documents$ head -n3 alpha.txt
A is for Apple
B is for Bear
C is for Cat
```

A negative number can also be used as an argument to the -n option, which tells the head command how many lines to *omit* from the bottom of the file. For example, when executing the head -n -90 command on a file that is 100 lines long, the output would only include the first 10 lines by omitting the last 90.

The alpha.txt file is 26 lines long. The following command omits the last 24 lines, resulting in only the first two lines of the file being displayed:

```
sysadmin@localhost:~/Documents$ head -n -24 alpha.txt
A is for Apple
B is for Bear
```

Displaying the End of a File

The opposite of the head command, the tail command, displays contents from the end of the file and not the beginning.

```
tail [OPTION]... [FILE]...
```

The tail command also displays ten lines by default with no options given:

```
sysadmin@localhost:~/Documents$ tail alpha.txt
Q is for Quark
R is for Rat
S is for Sloth
T is for Turnip
U is for Up
V is for Velvet
W is for Walrus
X is for Xenon
Y is for Yellow
Z is for Zebra
```

You may use either a number or the -n option as an argument to the tail command to specify how many lines you want to output from the end of the file. Therefore, the following two commands are equivalent:

```
sysadmin@localhost:~/Documents$ tail -3 alpha.txt
X is for Xenon
Y is for Yellow
Z is for Zebra
sysadmin@localhost:~/Documents$ tail -n3 alpha.txt
X is for Xenon
Y is for Yellow
Z is for Zebra
```

With the tail command, the -n option has an interesting twist to it, as well. When using the -n option with a number prefixed by a plus sign + character, the number is interpreted as the line number in the file to start displaying content from; it will display from that line to the end of the file. In other words, the -n option used with the +20 argument will display the contents of the file starting at line twenty and continuing until the end of the file.

```
sysadmin@localhost:~/Documents$ tail -n +20 alpha.txt
T is for Turnip
U is for Up
V is for Velvet
W is for Walrus
X is for Xenon
Y is for Yellow
Z is for Zebra
```

Consider This

Another unusual, but useful feature of the `tail` command is its ability to *follow* a file. When the `tail` command is executed with the `-f` option and a file name argument, then it will initially output the number of lines specified. Instead of exiting after displaying the lines, the `tail` command continues to run, following any changes in the file and displaying new content as it is added to the end of the file.

One of the main applications for this follow feature is for administrators to watch log files change while they are troubleshooting problems.

Additionally, the `-F` option is used when following a log file that may have been rotated or archived, and a new empty file of the same name replaces it. The `-F` option allows `tail` to notice that the underlying inode number that matches the log file name has changed, and it will continue to watch the new log file.

Combining File Output

The paste command will merge the lines of one or more files, line by line, separating them with a tab as a *delimiter (separator)* by default.

```
paste [OPTION]... [FILE]...
```

To demonstrate the paste command's functionality, the numbers.txt and letters.txt will be used:

```
sysadmin@localhost:~/Documents$ cat numbers.txt
1
2
3
4
5
sysadmin@localhost:~/Documents$ cat letters.txt
a
b
c
d
e
```

To paste these files together with a tab between the elements, use the paste command:

```
sysadmin@localhost:~/Documents$ paste numbers.txt letters.txt
1       a
2       b
3       c
4       d
5       e
```

It is also possible to use other characters as delimiters. For instance, if the intent is to create a file that a spreadsheet could easily open, a comma would be chosen as the delimiter. The -d option is used to specify the delimiter.

```
sysadmin@localhost:~/Documents$ paste -d , numbers.txt letters.txt
1,a
2,b
3,c
4,d
5,e
```

Similar to the paste command, the join command is able to combine two files. Instead of simply going line by line through the files, the join command matches the values of fields to determine which lines to combine. In other words, it will join files based on a common field between the files.

In order to break a line into fields, the join command uses either tabs or spaces between the contents of the files and breaks each line into separate fields.

```
join [OPTION]... FILE1 FILE2
```

To demonstrate this properly, we will use the following two files.

```
sysadmin@localhost:~/Documents$ cat adjectives.txt
1 golden
2 honey
3 fruit
4 grey
5 bald
sysadmin@localhost:~/Documents$ cat animals.txt
1 retriever
2 badger
3 bat
4 wolf
5 eagle
```

If the data that is used to join the files is the first field, then there is no need to specify which join fields to use. However, the field that is used in the joining process must be sorted in both files first, or else errors may occur. Also, missing elements in the key of either file will not appear in the results.

In the following example, the first field is used to merge the two files together. When the files are joined, the number field is used as a key to determine which words to join:

```
sysadmin@localhost:~/Documents$ join adjectives.txt animals.txt
1 golden retriever
2 honey badger
3 fruit bat
4 grey wolf
5 bald eagle
```

The first field in each file containing numbers is only shown once in the output, followed by a space and the second field from the first file, finally by another space and the second field from the second file. For illustration purposes, the same two files are shown being combined with the paste and cat commands:

```
sysadmin@localhost:~/Documents$ paste adjectives.txt animals.txt
1    golden    1    retriever
2    honey     2    badger
3    fruit     3    bat
4    grey      4    wolf
5    bald      5    eagle
sysadmin@localhost:~/Documents$ cat adjectives.txt animals.txt
1    golden
2    honey
3    fruit
4    grey
5    bald
1    retriever
2    badger
3    bat
4    wolf
5    eagle
```

Use the -t option to specify an alternate delimiter. For example, if two spreadsheets were exported into *comma separated value* .csv files, join could use those commas to distinguish one field from another.

The next example illustrates a more advanced use of the join command. Two files, people.csv and os.csv., will be combined by the join command using data that isn't in the first field.

```
sysadmin@localhost:~/Documents$ cat people.csv
Dennis,Richie
Andrew,Tanenbaum
Ken,Thompson
Linus,Torvalds
sysadmin@localhost:~/Documents$ cat os.csv
1970,Unix,Richie
1987,Minix,Tanenbaum
1970,Unix,Thompson
1991,Linux,Torvalds
```

The following join command will join the files according to the field containing last names:

```
join -1 2 -2 3 -t',' people.csv os.csv
```

To specify that the join field for the *first* file is the *second* field, -1 2 is used. The -1 option means "field of the first file" and the 2 argument means "the second field".

To specify that the join field for the *second* file is the *third field*, -2 3 is used. The -2 option means "field of the second file" and the 3 argument means "the third field".

These files also use a delimiter that is neither a tab nor a space, so the delimiter is specified as a comma , character with the -t',' option:

```
sysadmin@localhost:~/Documents$ join -1 2 -2 3 -t',' people.csv os.csv
Richie,Dennis,1970,Unix
Tanenbaum,Andrew,1987,Minix
Thompson,Ken,1970,Unix
Torvalds,Linus,1991,Linux
```

Command Line Pipes

To make effective use of filter commands, command pipelines are often used. The *pipe* | character can be used to send the output of one command to another:

```
COMMAND 1 | COMMAND 2 | COMMAND 3 ...
```

Typically, when a command has output or generates an error, the output is displayed on the screen; however, this does not have to be the case. Instead of being printed to the screen, the output of one command becomes input for the next command. This tool can be powerful, especially when looking for specific data; *piping* is often used to refine the results of an initial command.

The head and tail commands are often used in command pipelines to limit the number of lines when dealing with large files. For example, to number only the last three lines of the alpha.txt file, pipe the output of the tail command to the nl command:

```
sysadmin@localhost:~/Documents$ tail -3 alpha.txt | nl
     1  hello
     2  inkling
     3  jogger
```

Command pipelines can also be used to filter command output. For example, to list just the first five files in the current directory, pipe the output of the ls command to the head command:

```
sysadmin@localhost:~/Documents$ ls | head -5
School
Work
adjectives.txt
alpha-first.txt
alpha-second.txt
```

Extract Fields in a File

The cut command *extracts* fields of information from a text file.

```
cut [OPTION]... [FILE]...
```

Note the output of the following head command for the first line of the /etc/passwd command. There are 7 fields, each separated by the colon: character as a delimiter:

```
sysadmin@localhost:~/Documents$ head -1 /etc/passwd
root:x:0:0:root:/root:/bin/bash
```

root	x	0	0	root	/root	/bin/bash
1	2	3	4	5	6	7

The default field separator is either a space or tab; this can be modified with the *delimiter* -d option. Use the -f option to specify a list of comma separated field numbers to display or a hyphenated range of fields to display.

To extract certain fields from the /etc/passwd file, specify the delimiter using the colon character as an argument to the -d option. If only fields 1,5,6, and 7 were important, they could be extracted using the following command:

```
sysadmin@localhost:~/Documents$ head -1 /etc/passwd | cut -d: -f1,5,6,7
root:root:/root:/bin/bash
```

The same result could be achieved by combining the fields 5,6,7 into the range 5-7:

```
sysadmin@localhost:~/Documents$ head -1 /etc/passwd | cut -d: -f1,5-7
root:root:/root:/bin/bash
```

Given a file that has fields at fixed character positions, the cut command can be used to extract the characters one at a time by using the *character* -c option followed by a range.

For example, in the /var/syslog file; the first fifteen characters specify a timestamp. To extract only those characters from the text, specify 1 through 15:

```
sysadmin@localhost:~/Documents$ tail /var/log/syslog | cut -c1-15
Jul 20 15:17:01
Jul 20 15:59:02
Jul 20 16:17:01
Jul 20 16:41:22
Jul 20 16:59:02
Jul 20 17:17:01
Jul 20 17:59:02
Jul 20 18:17:01
Jul 20 18:59:02
Jul 20 19:17:01
```

Sort File Output

The sort command is used to display a file sorted on a specific field of data.

```
sort [OPTION]... [FILE]...
```

Using sort with no options on people.csv results in the lines being sorted in ASCII order, which is similar to alphabetical order:

```
sysadmin@localhost:~/Documents$ sort people.csv
Andrew,Tanenbaum
Dennis,Richie
Ken,Thompson
Linus,Torvalds
```

This command is able to sort using any field, the same way that a spreadsheet is able to sort by any column. By default, sort breaks up each line of a file into fields using whitespace (tabs or spaces) as delimiters. To specify an alternate delimiter, use the -t option. To specify the fields to sort from first to last, use one or more -k options. In the following example, the os.csv file is sorted based on the second field of data, using the comma , character as a delimiter:

```
sysadmin@localhost:~/Documents$ sort -t',' -k2 os.csv
1991,Linux,Torvalds
1987,Minix,Tanenbaum
1970,Unix,Richie
1970,Unix,Thompson
```

While it is possible to sort the first field of os.csv using the ASCII ordering, a numerical sort makes more sense. In order to have the sort command treat a field numerically, add an n as an argument to the -k option for that key field specification. Be aware in the following example the -k option has two arguments, the 1 indicates the first field, and the n specifies sorting that field numerically:

```
sysadmin@localhost:~/Documents$ sort -t',' -k1n os.csv
1970,Unix,Richie
1970,Unix,Thompson
1987,Minix,Tanenbaum
1991,Linux,Torvalds
```

To *reverse* the sort direction from ascending to descending, add an r argument to the key field specification:

```
sysadmin@localhost:~/Documents$ sort -t',' -k1nr os.csv
1991,Linux,Torvalds
1987,Minix,Tanenbaum
1970,Unix,Richie
1970,Unix,Thompson
```

The sort command can be combined with other commands to accomplish more sophisticated tasks. For example, the output of the cut -f7 -d: /etc/passwd command contains many duplicate values:

```
sysadmin@localhost:~/Documents$ cut -f7 -d: /etc/passwd | head -n4
/bin/bash
/usr/sbin/nologin
/usr/sbin/nologin
/usr/sbin/nologin
```

By piping this output to the sort command and using the *unique* -u option, duplicate lines will be removed from the output:

```
sysadmin@localhost:~/Documents$ cut -f7 -d: /etc/passwd | sort -u
/bin/bash
/bin/sync
/usr/sbin/nologin
```

Remove Duplicate Lines in a File

The uniq command does what the *unique* -u option did with the sort command. Unlike the sort command, which rearranges the rows consecutively before eliminating duplicates, the uniq command will only eliminate duplicates if they are already consecutive lines. If the sort command is already being used to rearrange the duplicates into consecutive order, then why not just use the -u option to eliminate duplicates?

There are a couple of reasons to use the uniq command instead of the sort -u command. First, there may be times when the lines should not be sorted first. The uniq command can simply remove lines that are currently consecutive.

Second, the *count* -c option to the uniq command outputs the number of duplicates that were counted (a feature that the sort command lacks):

```
sysadmin@localhost:~/Documents$ cut -f7 -d: /etc/passwd | sort | uniq -c
    3 /bin/bash
    1 /bin/sync
    23 /usr/sbin/nologin
```

Display File Contents in Various Formats

The od command can output data in several different formats. It produces an *octal dump* by default (hence the name *od*). One use of this command is to display the contents of a file when it contains non-printable characters, such as control characters.

```
od [OPTION]... [FILE]...
```

The following table summarizes the formats that the od command can output:

Single Option	Option with Argument	Meaning
`-a`	`-t a`	Named characters, ignoring high bit
`-b`	`-t o1`	Octal bytes
`-c`	`-t c`	ASCII characters or backslash escapes
`-d`	`-t u2`	Unsigned decimal 2-byte units
`-f`	`-t fF`	Floats
`-i`	`-t dI`	Decimal integers
`-l`	`-t dL`	Decimal longs
`-o`	`-t o2`	Octal 2-byte units
`-s`	`-t d2`	Decimal 2-byte units
`-x`	`-t x2`	Hexadecimal 2-byte units

In order to demonstrate the od command, the hidden.txt file will be used:

```
sysadmin@localhost:~/Documents$ cat hidden.txt
NDG LPIC-1 Curriculum
Is The Best.
```

It seems straight-forward enough, but there are non-printable control characters \r causing cat to return to the beginning of the first line. This is where od comes in handy. Use the -c option to examine ASCII characters:

```
sysadmin@localhost:~/Documents$ od -c hidden.txt
0000000    T   h   i   s      \r   i   s      \r   a      \r   h   i   d
0000020    d   e   n      \r   m   e   s   s   a   g   e      \r   N   G
0000040    D       L   P   I   C   -   1       C   u   r   r   i   c   u
0000060    l   u   m      \n   I   s       T   h   e       B   e   s   t
0000100    .  \n  \n
0000103
```

It appears that there was a hidden message in that file that the cat command missed. The point, however, is not to use od to send secret messages. The od command can be used for file recovery, or to open a file that crashes other programs.

As the name implies, od can display raw file content in octal form. It also supports hexadecimal and decimal output. This can be very useful when analyzing a precompiled binary, such as a virus:

```
sysadmin@localhost:~/Documents$ od -x hidden.txt
0000000 6854 7369 0d20 7369 0d20 2061 680d 6469
0000020 6564 206e 6d0d 7365 6173 6567 0d20 474e
0000040 2044 504c 4349 312d 4320 7275 6972 7563
0000060 756c 206d 490a 2073 6854 2065 6542 7473
0000100 0a2e 000a
0000103
```

Two other options control where the od command will read its data from the file:

Option	Function
`-j`	Specify how many bytes to skip from the input.
`-N`	Specify the total number of bytes the `od` command will read from the file.

See how these options work with the same hidden.txt file from the previous example, skipping thirteen bytes and displaying only fifteen bytes total:

```
sysadmin@localhost:~/Documents$ od -j 13 -N 15 -c hidden.txt
0000015    h   i   d   d   e   n      \r   m   e   s   s   a   g   e
0000034
```

Additional options that are used to affect the output of the od command:

Option	Function
`-w`	Specify the width in bytes to output next to the offset address. By default, the width will be sixteen bytes.

The `-A` option has several ways to specify the offset address to the left of the data as the formatting options covered earlier:

Option	Function
`-An`	Prevents the `od` command from outputting the offset address to the left of the data.
`-Ad`	Outputs file offsets using decimal addresses.
`-Ax`	Outputs file offsets using hexadecimal addresses.

File offsets are non-negative integers that specify a point within a file that is a specific number of bytes from a known position. For example, an offset of 64 from the beginning of a file would denote the 65th bit in the file. Offsets are a way to specify a particular piece of information located within a data file. They can be useful for dealing with files that are open or have been damaged, or in computer forensics for recovering deleted information.

In the following example, the od command displays the file with a width of eight bytes of data, the offset address suppressed, and the bytes of data being displayed as ASCII characters:

```
sysadmin@localhost:~/Documents$ od -j 13 -N 15 -c -w8 -An ./hidden.txt
   h   i   d   d   e   n       \r
   m   e   s   s   a   g   e
```

Translate File Characters

The tr command can be used to *translate* from one set of characters to another.

```
tr [OPTION]... SET1 [SET2]
```

The following example will translate all of the lowercase letters in alpha-first.txt to uppercase letters:

```
sysadmin@localhost:~/Documents$ cat alpha-first.txt | tr 'a-z' 'A-Z'
A IS FOR ANIMAL
B IS FOR BEAR
C IS FOR CAT
D IS FOR DOG
E IS FOR ELEPHANT
F IS FOR FLOWER
```

The characters that the tr command is translating do not have to be in a range; they can simply be listed for each set of characters. Be careful to balance the sets to make sure that they both have the same number of characters, or else strange results may occur.

The following example has the tr command replace the vowel characters with symbols:

```
sysadmin@localhost:~/Documents$ cat alpha-first.txt | tr 'aeiou' '@&1*^'
A 1s f*r An1m@l
B 1s f*r B&@r
C 1s f*r C@t
D 1s f*r D*g
E 1s f*r El&ph@nt
F 1s f*r Fl*w&r
```

Aside from performing a translate function, the tr command can also "squeeze" repeated characters and delete characters. In performing these functions, only one set of characters needs to be specified. For example, to eliminate duplicates, use the -s option to *squeeze* repeats of characters from the first set:

```
sysadmin@localhost:~/Documents$ echo 'aaaaaappleeeeeee' | tr -s 'ae'
apple
```

Using the -d option will delete the characters in the first set:

```
sysadmin@localhost:~/Documents$ cat alpha-first.txt | tr -d 'AEIOUaeiou'
 s fr nml
B s fr Br
C s fr Ct
D s fr Dg
 s fr lphnt
F s fr Flwr
```

Stream Editor Command

The *stream editor* sed command is a non-interactive editor that can be used to modify text.

```
sed [OPTION]... {SCRIPT} [FILE]...
```

To do a simple search and replace operation, use the following *script*, or *expression*. This will search for the pattern between the first two slashes, and if it finds that text, then it replaces it with what is specified between the last two slashes:

```
s/PATTERN/REPLACEMENT/
```

In the alpha-first.txt file, to replace the word Animal with the word Apple, execute the following command:

```
sysadmin@localhost:~/Documents$ sed 's/Animal/Apple/' alpha-first.txt
A is for Apple
B is for Bear
C is for Cat
D is for Dog
E is for Elephant
F is for Flower
```

Unlike most filter commands, the sed command can modify the original file by using the -i option.The -i option allows for an optional argument, which will be an *extension* added to the original file. The result is a new file which is a copy of the original file. For example, if you wanted to backup the original file as alpha-first.txt.original file, execute the following:

```
sysadmin@localhost:~/Documents$ sed -i'.original' 's/Animal/Apple/' alpha-first.txt
sysadmin@localhost:~/Documents$ cat alpha-first.txt
A is for Apple
B is for Bear
C is for Cat
D is for Dog
E is for Elephant
F is for Flower
sysadmin@localhost:~/Documents$ cat alpha-first.txt.original
A is for Animal
B is for Bear
C is for Cat
D is for Dog
E is for Elephant
F is for Flower
```

The original file was modified as a result of the -i option. For the rest of the examples, the -i option will not be used. The output will be sent to the terminal instead, and the original file will not be changed.

The Global Modifier

When performing a search and replace operation, the sed command will only replace the *first* occurrence of the search pattern by default. To replace *all* occurrences of the pattern, add the *global modifier* g after the final slash.

```
s/PATTERN/REPLACEMENT/g
```

For example, consider the food.txt text file:

```
sysadmin@localhost:~/Documents$ cat food.txt
Food is good.
```

The following example will only replace the first occurrence of the oo pattern with the 00 pattern:

```
sysadmin@localhost:~/Documents$ sed 's/oo/00/' food.txt
F00d is good.
```

If you want to replace every occurrence of oo with 00, add the g modifier:

```
sysadmin@localhost:~/Documents$ sed 's/oo/00/g' food.txt
F00d is g00d.
```

The sed command is also able to either insert text before a pattern or after a pattern. For this type of expression, do not use the s character before the first slash; use an *insert* change with the i\ expression or an *append* change with the a\ expression.

```
/PATTERN/i\TEXT\
/PATTERN/a\TEXT\
```

Inserting text with the i\ expression affects the line before the line containing the search term:

```
sysadmin@localhost:~/Documents$ sed '/is/i\Hello' food.txt
Hello
Food is good.
```

Appending text with the a\ expression affects the line after the line containing the search term:

```
sysadmin@localhost:~/Documents$ sed '/is/a\Hello' food.txt
Food is good.
Hello
```

The sed command can also be used to search for a line of text containing a pattern and then delete the lines that match. Place the pattern to search for between two slashes followed by a d to carry out this operation:

```
/PATTERN/d
```

Notice the animals.txt file has five lines, but the following example command removes the lines that contain an a:

```
sysadmin@localhost:~/Documents$ cat animals.txt
1 retriever
2 badger
3 bat
4 wolf
5 eagle
sysadmin@localhost:~/Documents$ sed '/a/d' animals.txt
1 retriever
4 wolf
```

To change an entire line that matches a pattern to something else, use a slash, the pattern to match, another slash, c\, and then the new text.

```
/PATTERN/c\REPLACEMENT
```

For example, to change a line containing 3 to three, use the following command:

```
sysadmin@localhost:~/Documents$ cat numbers.txt
1
2
3
4
5
sysadmin@localhost:~/Documents$ sed '/3/c\three' numbers.txt
1
2
three
4
5
```

Keep in mind the *entire line* will be replaced, not just the matching pattern:

```
sysadmin@localhost:~/Documents$ sed '/3/c\three' animals.txt
1 retriever
2 badger
three
4 wolf
5 eagle
```

The sed command normally parses only one expression to filter text. To use multiple expressions, use an -e option with each expression to modify the file.

Notice that the order in which those expressions are placed makes a difference. In the first attempt, both expressions are effective, but in the second attempt, only the first is effective. After the first expression deletes all lines containing an a, the second expression doesn't have anything to do because the line that contained 3 was removed:

```
sysadmin@localhost:~/Documents$ sed -e '/3/c\three' -e '/a/d' animals.txt
1 retriever
three
4 wolf
sysadmin@localhost:~/Documents$ sed -e '/a/d' -e '/3/c\three' animals.txt
1 retriever
4 wolf
```

Note

The use of the term *pattern* in this section refers to one of the most powerful aspects of the `sed` command, its compatibility with *regular expressions*. In each example so far, the `sed` command has been using *literal* expressions involving alphanumeric characters like `t` or `o` or strings like `three`. Regular expressions can be used to create patterns for use with the `sed` command and many other, far more powerful commands.

Regular expressions will be covered in greater detail later in the course.

Counting Lines in a File

The wc command can be used to analyze a text file. By default, wc counts the number of lines, words, and byte counts in a passage and outputs this information in the following format:

```
lines words bytes filename
```

Recall the alpha-first.txt file:

```
sysadmin@localhost:~/Documents$ cat alpha-first.txt
A is for Apple
B is for Bear
C is for Cat
D is for Dog
E is for Elephant
F is for Flower
```

Using the wc command on the same file reveals the following:

```
sysadmin@localhost:~/Documents$ wc alpha-first.txt
 6 24 90 alpha-first.txt
```

The wc command reports that there are 6 lines, containing 24 words, totaling 90 bytes in length. The total number of bytes includes spaces, punctuation, and carriage returns.

To view each of these numbers separately, options can be used:

```
sysadmin@localhost:~/Documents$ wc alpha-first.txt -l
6 alpha-first.txt
sysadmin@localhost:~/Documents$ wc alpha-first.txt -w
24 alpha-first.txt
sysadmin@localhost:~/Documents$ wc alpha-first.txt -m
90 alpha-first.txt
```

In the example, the -l option was used to count the number of *lines*. In the second example, the -w option was used to count the number of *words*. Finally, the -m option indicates the number of *characters*.

One might imagine that the -c option would count the number of characters, but -c is used to count the number of bytes. The number of bytes can vary if there are non-printable characters in the file. In this case, they are the same:

```
sysadmin@localhost:~/Documents$ wc alpha-first.txt -c
90 alpha-first.txt
```

One last option, -L returns the maximum line length in the file:

```
sysadmin@localhost:~/Documents$ wc alpha-first.txt -L
17 alpha-first.txt
```

If the word count of all the files in a particular folder is desired, the asterisk wildcard * can be used:

```
sysadmin@localhost:~/Documents$ wc *
  wc: School: Is a directory
      0       0       0 School
wc: Work: Is a directory
      0       0       0 Work
      5      10      39 adjectives.txt
      6      24      90 alpha-first.txt
      6      24      90 alpha-first.txt.original
      7      28     106 alpha-second.txt
     13      52     195 alpha-third.txt
     26     104     390 alpha.txt
      5      10      42 animals.txt
      1       3      14 food.txt
      7      14     647 hello.sh
      3      11      67 hidden.txt
      5       5      10 letters.txt
      8      32     187 linux.txt
  10240   10236   66540 longfile.txt
     13      36     236 newhome.txt
      5       5      10 numbers.txt
      4       4      77 os.csv
      4       4      59 people.csv
      6      21     110 profile.txt
     11      11      51 red.txt
      2      26     130 spelling.txt
 102305  102305  971578 words
 112682  112965 1040667 total
```

The output above reveals that longfile.txt really is the longest file in the folder.

CHAPTER 10 – STANDARD TEXT STREAMS AND REDIRECTION

One of the key points in the UNIX philosophy was that all CLI commands should accept text as input and produce text as output. As this concept was applied to the development of UNIX (and later Linux), commands were developed to accept text as input, perform some kind of operation on the text and then produce text as output. Commands that read in text as input, alter that text in some way, and then produce text as output are sometimes known as filters.

In order to be able to apply filter commands and work with text streams, it is helpful to understand a few forms of *redirection* that can be used with most commands: pipelines, standard output redirection, error output redirection, and input redirection.

STANDARD OUTPUT

When a command executes without any errors, the output that is produced is known as *standard out*, also called stdout or STDOUT. By default, this output will be sent to the terminal where the command is being executed.

It is possible to redirect standard out from a command so it will go to a file instead of the terminal. Standard output redirection is achieved by following a command with the greater-than > character and a destination file. For example, the ls ~ command will list the files in the home directory. To save a list of the files in the home directory, you must direct the output into a text file. To create the /tmp/home.txt file:

```
sysadmin@localhost:~$ ls ~ > /tmp/home.txt
```

After which, the home.txt file will resemble:

```
sysadmin@localhost:~$ cat /tmp/home.txt
Desktop
Documents
Downloads
Music
Pictures
Public
Templates
Videos
```

Redirecting output using a single greater-than > character will create a new file, or *overwrite* the contents of an existing file with the same name. Redirecting standard output with two greater-than >> characters will also create a new file if it does not exist. The difference is that when using the >> characters, the output of the command will be *appended* to the end of a file if it does already exist. For example, to append to the file that was created by the previous command, execute the following:

```
sysadmin@localhost:~$ date >> /tmp/home.txt
```

Which will update the home.txt file to:

```
sysadmin@localhost:~$ cat /tmp/home.txt
Desktop
Documents
Downloads
Music
Pictures
Templates
Videos
Thu Oct  2 17:36:02 UTC 2020
```

There is a number associated with the standard output *file descriptor* (the > character): the number 1 (one). However, since standard output is the most commonly redirected command output, the number can be omitted. Technically, the commands should be executed as shown below:

```
sysadmin@localhost:~$ ls 1> /tmp/ls.txt
sysadmin@localhost:~$ date 1>> /tmp/ls.txt
```

`COMMAND > FILE`	Create or overwrite `FILE` with the standard output of `COMMAND`
`COMMAND 1> FILE`	
`COMMAND >> FILE`	Create or append to `FILE` with the standard output of `COMMAND`
`COMMAND 1>> FILE`	

Standard Error

When a command encounters an error, it will produce output that is known as *standard error*, also called stderr or STDERR. Like standard out, the standard error output is normally sent to the same terminal where the command is currently being executed. The number associated with the standard error file descriptor is 2 (two).

If you tried to execute the ls /junk command, then the command would produce standard error messages because the /junk directory does not exist.

```
sysadmin@localhost:~$ ls /junk
ls: cannot access /junk: No such file or directory
```

Because this output goes to standard error, the greater-than > character alone will not successfully redirect it, and the output of the command will still be sent to the terminal:

```
sysadmin@localhost:~$ ls /junk > output
ls: cannot access /junk: No such file or directory
```

To redirect these error messages, you must use the correct file descriptor, which for standard error is the number 2. Execute the following, and the error will be redirected into the /tmp/ls.err file:

```
sysadmin@localhost:~$ ls /junk 2> /tmp/ls.err
```

Just like standard output, the use of a single > character for redirection will either create the file if it doesn't exist or *clobber* (overwrite) an existing file's contents. To prevent clobbering an existing file when redirecting standard error, use the double >> characters after the number 2 to append instead:

```
sysadmin@localhost:~$ ls /junk 2>> /tmp/ls.err
```

The /tmp/ls.err file now contains the redirected error messages:

```
sysadmin@localhost:~$ cat /tmp/ls.err
ls: cannot access '/junk': No such file or directory
ls: cannot access '/junk': No such file or directory
```

`COMMAND 2> FILE`	Create or overwrite `FILE` with the standard error of `COMMAND`
`COMMAND 2>> FILE`	Create or append to `FILE` with the standard error of `COMMAND`

Some commands will produce both stdout and stderr output:

```
sysadmin@localhost:~$ find /etc -name passwd
/etc/pam.d/passwd
/etc/passwd
find: '/etc/ssl/private': Permission denied
```

These two different outputs can be redirected into two separate files by using the following syntax:

```
sysadmin@localhost:~$ find /etc -name passwd > /tmp/output.txt 2> /tmp/error.txt
```

The cat command can be used to verify the success of the redirection above:

```
sysadmin@localhost:~$ cat /tmp/output.txt
/etc/pam.d/passwd
/etc/passwd
sysadmin@localhost:~$ cat /tmp/error.txt
find: '/etc/ssl/private': Permission denied
```

Sometimes it isn't useful to have the error messages displayed in the terminal or stored in a file. To discard these error messages, use the /dev/null file.

The /dev/null file is like a trash can, where anything sent to it disappears from the system; it's sometimes called the *bit bucket* or *black hole*. Any type of output can be redirected to the /dev/null file; most commonly users will redirect standard error to this file, rather than standard output.

The syntax to use the /dev/null file is the same as it is for redirecting to a regular file:

```
sysadmin@localhost:~$ find /etc -name passw 2> /dev/null
/etc/pam.d/passwd
/etc/passwd
```

What if you wanted all output (standard error and standard out) sent to one file? There are two techniques to redirect both standard error and standard out:

```
sysadmin@localhost:~$ ls > /tmp/ls.all 2>&1
sysadmin@localhost:~$ ls &> /tmp/ls.all
```

Both of the preceding command lines will create a file called /tmp/ls.all that contains all standard out and standard error. The first command redirects stdout to /tmp/ls.all and the 2>&1 expression means "send stderr wherever stdout is going". In the second example, the &> expression means "redirect all output".

A similar technique can be used to append all output to a single file:

```
sysadmin@localhost:~$ ls /etc/au* >> /tmp/ls.all 2>&1
sysadmin@localhost:~$ ls /etc/au* &>> /tmp/ls.all
```

`COMMAND &> FILE` `COMMAND > FILE 2>&1`	Create or overwrite `FILE` with all output (`stdout` , `stderr`) of `COMMAND`
`COMMAND &>> FILE` `COMMAND >> FILE 2>&1`	Create or append to `FILE` with all output (`stdout` , `stderr`) of `COMMAND`

Standard Input

Standard in, also called stdin or STDIN, normally comes from the keyboard with input provided by the user who runs the command. Although most commands are able to read input from files, there are some that expect the user to enter it using the keyboard. One common way that text files are used as standard input for commands is by creating script files. Scripts are plain text files which are interpreted by the shell when given the proper permissions and prefaced with #!/bin/sh on the first line, which tells the shell to interpret the script as standard input:

```
 GNU nano 2.9.3                examplescriptfile.sh

#!/bin/sh
echo HelloWorld
```

When the script file is invoked at the prompt using the ./ syntax, the shell will run all commands in the script file and return the result to the terminal window, or wherever the output is specified to be sent to:

```
sysadmin@localhost:~$ ./examplescriptfile.sh
HelloWorld
```

One example of a command, that isn't normally executed properly from a text file, is the cd command. When the following script is run in the terminal, it is run as a *child process*, however, in order to successfully change to another directory, the cd command must be run as a *parent process*. Consider the example testcd.sh script file below, which specifies to use the cd command to change to the School directory then use the echo command to print the string HelloWorld:

```
GNU nano 2.9.3                    testcd.sh

#!/bin/sh
cd /home/sysadmin/Documents/School
echo HelloWorld




                          [ Read 3 lines ]
^G Get Help  ^O Write Out ^W Where Is  ^K Cut Text  ^J Justify   ^C Cur Pos
^X Exit      ^R Read File ^\ Replace   ^U Uncut Text^T To Linter ^_ Go To Line
```

When we try to run the testcd.sh script the following occurs:

```
sysadmin@localhost:~$ ./testcd.sh
HelloWorld
```

The script executes the echo command and prints HelloWorld, but the cd command cannot run as a parent process, so we are still in the directory where we started.

Therefore, even though the standard input doesn't return an error, it won't create the desired outcome of changing the directory to /home/sysadmin/Documents/School. However, when the standard input is typed by a user, and the desired directory is provided as an argument to the cd command at the command line, it functions normally:

```
sysadmin@localhost:~$ cd /home/sysadmin/Documents/School
sysadmin@localhost:~/Documents/School$
```

In some cases, it is useful to redirect standard input, so it comes from a file instead of the keyboard. A good example of when input redirection is desirable involves the tr command. The tr command *translates* characters by reading data from standard input; translating one set of characters to another set of characters and then writing the changed text to standard output.

For example, the following tr command would take input from a user (via the keyboard) to perform a translation of all lowercase characters to uppercase characters. Execute the following command, type some text, and press **Enter** to see the translation:

```
sysadmin@localhost:~$ tr 'a-z' 'A-Z'
hello
HELLO
```

The tr command doesn't stop reading from standard input unless it's terminated or receives an "End of Transmission" character. This can be accomplished by typing **Ctrl**+**D**.

The tr command won't accept a file name as an argument on the command line. To perform a translation using a file as input, utilize input redirection. To use input redirection, type the command with its options and arguments followed by the less-than < character and a path to a file to use for input. For example:

```
sysadmin@localhost:~$ cat Documents/animals.txt
1 retriever
2 badger
3 bat
4 wolf
5 eagle
sysadmin@localhost:~$ tr 'a-z' 'A-Z' < Documents/animals.txt
1 RETRIEVER
2 BADGER
3 BAT
4 WOLF
5 EAGLE
```

The output shows the animals.txt file "translated" into all uppercase characters.

`COMMAND < FILE`	Use `FILE` as standard input to `COMMAND`

Important

Do not attempt to use the same file for input and output redirection, as the results are probably not desirable (you end up losing all data). Instead, capture the output and place it into another file; use a different file name as shown below:

```
sysadmin@localhost:~$ tr 'a-z' 'A-Z' < Documents/animals.txt > animals.new
```

Command Pipelines

Command pipelines are often used to make effective use of filter commands. In a command pipeline, the output of one command is sent to another command as input. In Linux and most operating systems, the vertical bar or pipe | character is used between two commands to represent a command pipeline.

For example, imagine that the output of the history command is very large. To send this output to the less command, which displays one "page" of data at a time, the following command pipeline can be used:

```
sysadmin@localhost:~$ history | less
```

> **Note**
>
> To exit the `less` pager command, press the **Q** key.

Even better, take the output of the history command and filter the output by using the grep command. In the following example, the text output by the history command is redirected to the grep command as input. The grep command matches the ls strings and sends its output to standard out:

```
sysadmin@localhost:~$ history | grep "ls"
   1  ls ~ > /tmp/home.txt
   5  ls 1> /tmp/ls.txt
   6  ls 1> /tmp/ls.txt
   7  date 1>> /tmp/ls.txt
   8  ls /junk
   9  ls /junk > output
  10  ls /junk 2> /tmp/ls.err
  11  ls /junk 2>> /tmp/ls.err
  14  ls > /tmp/ls.all 2>&1
  15  ls &> /tmp/ls.all
  16  ls /etc/au* >> /tmp/ls.all 2>&1
  17  ls /etc/au* &>> /tmp.ls.all
  20  history | grep "ls"
```

Command pipelines become really powerful when three or more commands are combined. For example, view the contents of the os.csv file in the Documents directory:

```
sysadmin@localhost:~$ cat Documents/os.csv
1970,Unix,Richie
1987,Minix,Tanenbaum
1970,Unix,Thompson
1991,Linux,Torvalds
```

The following command line will extract some fields from the os.csv file with the cut command, then sort these lines with the sort command, and finally eliminate duplicate lines with the uniq command:

```
sysadmin@localhost:~$ cut -f1 -d',' Documents/os.csv | sort -n | uniq
1970
1987
1991
```

tee Command

A server administrator works like a plumber, using pipes, and the occasional tee command. The tee command splits the output of a command into two streams: one directed to standard output, which displays in the terminal, and the other into a file.

The tee command can be very useful to create a log of a command or a script. For instance, to record the runtime of a process, start with the date command and make a copy of the output into the timer.txt file:

```
sysadmin@localhost:~$ date | tee timer.txt
Fri Nov  7 02:21:24 UTC 2020
```

The timer.txt file now contains a copy of the date, the same output displayed in the preceding example:

```
sysadmin@localhost:~$ cat timer.txt
Fri Nov  7 02:21:24 UTC 2020
```

Run the process that needs to be timed. The sleep command is being substituted for a timed process; it pauses for a given number of seconds:

```
sysadmin@localhost:~$ sleep 15
```

Then run the date command again. This time, append the time to the end of the timer.txt file by using the -a option:

```
sysadmin@localhost:~$ date | tee -a timer.txt
Fri Nov  7 02:28:43 UTC 2020
```

Both times will then be recorded in the file.

```
sysadmin@localhost:~$ cat timer.txt
Fri Nov  7 02:21:24 UTC 2020
Fri Nov  7 02:28:43 UTC 2020
```

To run all of the above commands as a single command, use a semicolon ; character as a separator:

```
sysadmin@localhost:~$ date | tee timer.txt; sleep 15; date | tee -a timer.txt
Fri Nov  7 02:35:47 UTC 2020
Fri Nov  7 02:36:02 UTC 2020
```

The command above will display and record the first output of the date command, pause for 15 seconds, then display and record the output of the second date command. The timer.txt file now contains a permanent log of the runtime.

```
sysadmin@localhost:~$ cat timer.txt
Fri Nov  7 02:35:47 UTC 2020
Fri Nov  7 02:36:02 UTC 2020
```