1.1 Introduction

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

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

1.1.1 Role of the Kernel

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

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

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

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

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

1.1.2 Applications

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

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

1.1.3 Role of Open Source

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

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

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

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

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

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

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

1.1.4 Linux Distributions

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

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

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

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

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

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

1.2 Hardware Platforms

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

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

Eventually, cellular phones and tablets started running Linux. A company, later bought by Google, came up with the Android platform which is a bundle of Linux and the software necessary to run a phone or tablet. This means that the effort to get a phone to market is significantly less. Instead of long development on a new operating system, companies can spend their time innovating on the user facing software. Android is now one of the market leaders in the phone and tablet space.

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

1.3 Shell

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

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

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

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

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

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

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

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

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

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

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

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

1.4 Bash Shell

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

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

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

```
[1] 108
```

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

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

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

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

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

1.5 Accessing the Shell

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

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

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

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

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

1.6 Filesystems

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

1.7 What is a Command?

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

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

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

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

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

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

Consider This

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

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

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

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


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

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

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

1.8 Commands That Are Stored In Files

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

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

Consider This

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

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

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

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

Summary of Key Terms

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

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

Source Code: The original human-readable text code.

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

1.9 Basic Command Syntax

Most commands follow a simple pattern of syntax:

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

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

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

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

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

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

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

1.10 Specifying Options

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Consider This

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

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

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

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

1.11 Specifying Arguments

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

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

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

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

Consider This

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

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

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

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

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

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

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

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

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

1.12 exec Command

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

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

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

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

1.13 uname Command

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

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

Options for the `uname` command are as follows:

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

Short Option	Long Option	Prints
	--version	Version information

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

1.14 pwd Command

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

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

1.15 Command Completion

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

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

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

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

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

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

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

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

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

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

Chapter Objectives

Chapter 1: Using the Shell

This chapter will cover the following exam objectives:

103.1: Work on the command line

Weight: 4

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

Key Knowledge Areas:

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

KEY TERMS

Chapter 1: Using the Shell

bash

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

echo

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

ls

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

pwd

Print the name of the current working directory.
Section 1.14

uname

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