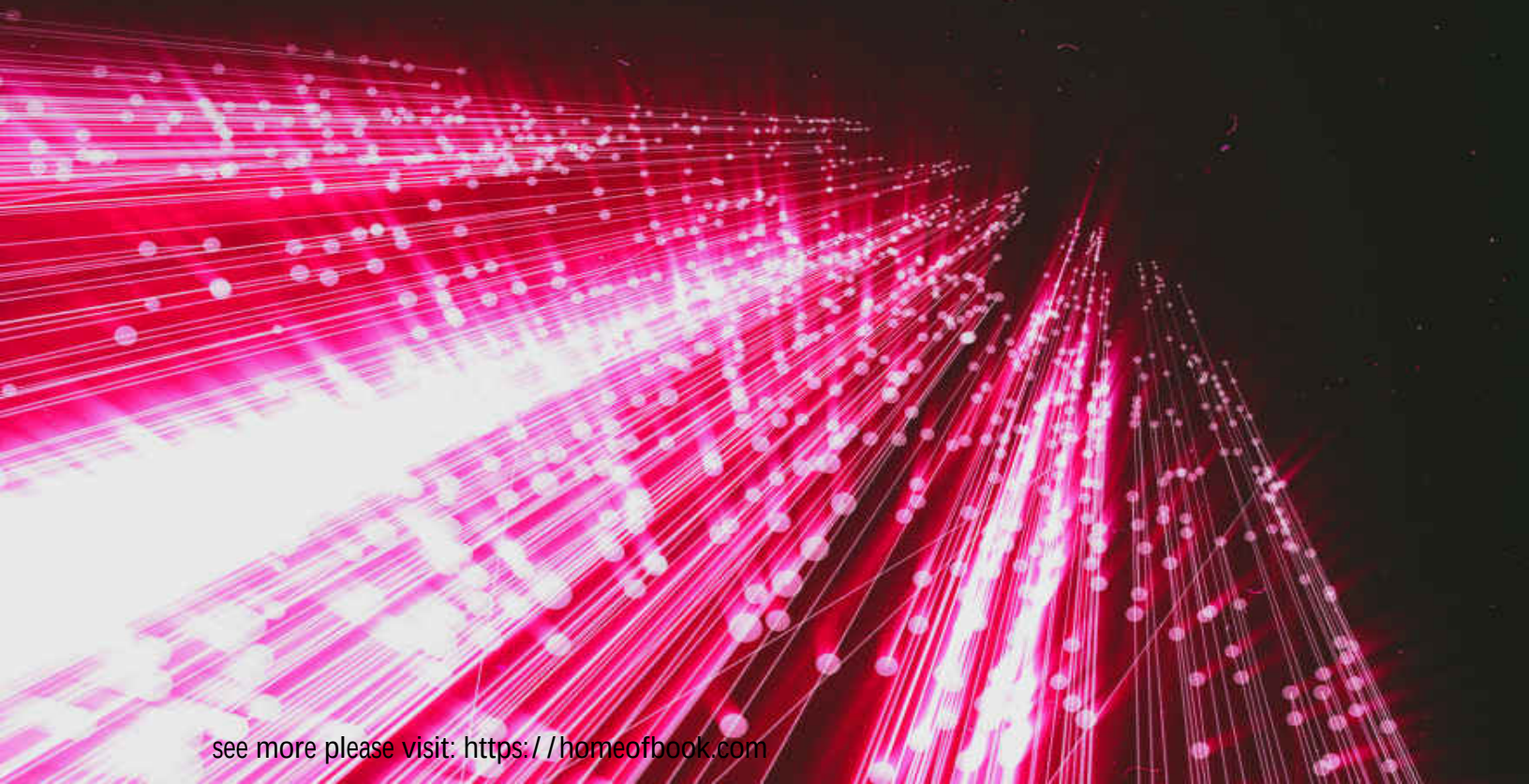
ANDY VICKLER

LINUX

LINUX COMMAND LINES AND SHELL SCRIPTING



Linux

Linux Command Lines and Shell Scripting

Table of Contents

Introduction

Chapter One: What is Linux?

- *The Birth of the Linux Operating System*
- *Linux Distributions*
- *Linux is Open-Source*
- *The Linux Shell*
- *Root*
- *Reasons to Use the Linux Operating System*
- *Installation of the Linux Operating System*

Chapter Two: Working with Linux Commands

- *Let's Start*
- *The First Commands for Your Linux Operating System*
- *Basic Commands*
- *File Navigation in Linux*
  - The ls command
- *Options and Arguments Commands*
- *Finding Out the Type of File*
  - The less command

Chapter Three: Files & Directories Commands

- *Special Characters*
- *Command-Line Editing*
- *Linux Commands for Directories*
  - The cp command
  - Examples of cp command
  - The mv command
  - Examples for mv command
  - The rm command
  - Examples of rm command

Chapter Four: Practical Work with Commands

The type command
The which command
Documentation of Commands
The help command
The man command
The apropos command
The whatis command
The info command
Chapter Five: Redirection Commands & Keyboard Tricks with Linux Commands
Redirection
Keyboard Tricks
Text Modification
Completion Commands
Completion commands
Searching History
History Expansion
Chapter Six: Process Commands
Process States
Using the Top Command to View Processes
Chapter Seven: Working with Linux Editors
The Vim Editor
Data Editing
Copy & Paste
The KDE Editor Family
The KWrite Editor
The Edit Menu of KWrite Editor
The KWrite Editor Tools
The GNOME Editor
Basic Features of the Editor
Preferences
View

Editor Tab
Syntax Highlighting
Plugins
The emacs Editor
Basic Commands of emacs Editor
Editing Data
Popular Linux Commands
Chapter Eight: Linux Shell Scripting
Writing the Shell Script
The Format of Linux Shell Scripts
Displaying Text
The if-then Statement
The if-then-else Statement
Advanced if-then Features
The for Command
Reading a List through a Variable
The while Command
Multiple Test Commands
The until Command
Nesting Loops
Loop Control
More Basic Shell Scripts
Chapter Nine: Linux Shell Scripting for Functions
The Basics
Function Creation
Returning a Value in Linux Functions
Passing Parameters to a Function
Passing Arrays to Functions
Handling User Input
Conclusion

References

Introduction

This book on the Linux command line and shell scripting contains proven steps and strategies for using the keyboard to type commands and writing shell scripts to create your customized programs. Linux is tougher than Windows and Mac operating systems, and it is the securest of the three. When you have learned Linux, you can create your custom software and use it to do common daily tasks. You can customize how you want your computer should work.

Linux is fun to operate, but it is not easy to operate. Unlike Windows that has a graphical interface, Linux has a command line. If you are unfamiliar with what a command line is, you should refer to Dos or Command Prompt in Windows. It is a black screen on which you have to enter commands and complete your daily tasks.

I have written this book in two-parts: the first part deals with commands and the second part deals with shell scripting. Unless you have a good grasp of Linux commands and shell scripting, you cannot expertly operate the system. I have packed the book with the most used and important commands that you'll need. I have also included a few results you can see on the command line after entering a command.

The shell scripting section can be tough. If you have a programming background, it is fun to read. To practice scripting, you will have to work in an editor, but don't worry, you will find a chapter on Linux editors later in the book. You can write your own scripts or copy mine and paste them into the editor to practice. For convenience, I have included the results of each script so that you may compare them. I have also included lots of basic script examples on how to use commands.

Chapter One: What is Linux?

As a Linux student, the first thing you should learn is Linux's origins to understand the concept behind this operating system. Linux is a software layer between the hardware and the software in a computer operating system. It allows you to do productive things and create custom programs on your computer. In simple words, an operating system is a medium between the software and a computer system's hardware. An operating system allows you to store data on your storage devices like hard drives, solid-state drives, and USBs. It manages the transmission of data from one element to another; for example, it oversees data flow from the operating system to printers in your office or home. If you have installed a normal Windows environment such as the Microsoft Windows operating system on your computer system, the Windows operating system runs the hardware. It controls the mouse, keyboard, printers, scanners, and other accessories. You will have to install Microsoft Office, Adobe readers, pdf converters, and other software as per your needs. You will pay for each program and have them installed on your system.

Linux is the same as the Windows operating system regarding the process of controlling the hardware. It is different because it acts as a medium between the instructional code of the software and the physical device. The biggest difference is that of the software you will use in the Linux operating systems. The software will be of a different type as compared to the ones that run on Windows systems. You cannot install and run Microsoft Office or Adobe Photoshop on Linux environment. Linux runs different servers like web virtualization servers, Apache servers, database servers, etc.

However, Linux has several distributions that are made for personal desktop computers. These distributions are similar to macOS and Windows operating systems. They run the same type of programs like word processors, image editors, and games. These Linux distributions appear to be more targeted for the home users searching for a free system alternative.

Linux did not kick-off as an operating system or a challenger to the Windows operating system. In the start, Linux happened to be a kernel that was created by Linus Torvalds. Linus was a student then at the University of Helsinki. The kernel is still useful in the system. In the start, the Linux kernel was used along with the GNU operating system. You can say that the GNU system was incomplete without the kernel.

A kernel is defined as an integral component of Linux. A kernel is considered the central part of operating systems, responsible for all the interfacing of applications and hardware. There are two types of kernels in the market now, namely Unix-like kernels and Windows kernels.

he Birth of the Linux Operating System

Between 1991 and 1994, Linus took a step further to create the Linux operating system. He combined the GNU OS with Linux Kernel. At the start, he wanted to create an operating system that did not come free, but instead, he needed something that he could customize to fit as per his programming needs. Linux appeared to be his pet project at the start. It was like a side hustle. UNIX is different from the Linux operating system.

Linus built the entire Linux system from scratch. He created Linux because he desired to build an open-source operating system for the people to use. At that time, UNIX was not open-source. People had to pay someone to use UNIX. Similarly, Microsoft was also a paid operating system.

Therefore, Linux came up with the idea of an open-source operating system. He worked up the idea with his friends from the Massachusetts Institute of Technology (MIT). Coupled with building an open-source operating system, they needed an easy-to-use and efficient operating system they could customize to suit their programming needs.

Linux Distributions

When Linus was creating the Linux operating system, he stopped working on it for a while. During that period, he made the code for the operating system public. This allowed everyone to take part in the creation of the system. Scientists and computer geeks started working on the concept as well. They changed the operating system as they deemed fit.

Major educational institutions and companies liked the concept of this new operating system because everyone who had the source code could install Linux on his or her computer. This is how people started creating different versions of the Linux operating system. Students from the University of California, Berkeley, tried to start creating a version. People from China and people with different occupations also started creating versions to suit their personal needs.

The availability of the source code to the public facilitated the creation of distros or distributions. Distributions are different versions of Linux that people have been creating over time. Linux has different versions, and its many distributions offer it several capabilities. When you have to decide which Linux distribution you need to use, you have to decide what you want your computer to do with Linux. I will explain it by running an analogy with the Windows operating system. When you install Microsoft Windows operating system on our computer, every distribution is built to do things in a particular manner.

There is a version of Linux known as Trustix. Linux Trustix is labeled as the most secure Linux operating system in the market. It is simply a brick. You set up Linux Trustix, and no one will be able to hack it until you do something immensely stupid. There will be no sneaking in by viruses. It is a secure and solid server. However, you have to decide that you really need a secure server before picking up the source code and installing the system.

If you are looking out for a computer that you can use for some office applications, you may need an Ubuntu Linux desktop version. If you are looking for a super-secure computer, then you may need Linux

Trustix. If you are looking forward to something that comes with enterprise-level support, you may need to use a Linux distribution paired up with a tech support center to help you out. In this case, Red Hat Linux will be the answer to your needs. Again, you must decide what you want your computer to do to determine the most appropriate Linux distribution to install on your computer system.

Linux is Open-Source

Now that you have learned about the origins of Linux and its distributions, it is time to move forward to the concept of open-source licensing, which makes Linux different from other operating systems. Linux has open-source licensing. You might have heard of open-source software at some point in your life. Open source does not mean that your software is free to use. If you treat all open-source software as free, you will be on the verge of jeopardizing your programming career and company as well. This is legally bad; therefore, we must discuss open-source software to clear the air.

Open-source software means that whenever programmers write a code for a software, they give you the code to see how he or she wrote the program in the first place. It does not mean that the program is free to use. There are different ways by which open-source vendors are paid.

The first way is through the Open Source model, where they give off their software free of cost. However, when you require support or training for the software, they will charge you a certain amount. For example, you can download MySQL server for the Linux server. You find it useful and powerful as well. Even though you have learned the MySQL program's different intricacies, there are some aspects of the software you may need to learn or need support with. Therefore, you approach the software developer, ask for training or support with the software. At this point, you have to pay the programmer or developer for his or her development efforts.

The second way by which developers are paid is through a non-commercial open-source license. This is where most people get into

trouble. You have to pay them to use them.

If you want software for home use, there is no problem. Once you use it to connect to a business server, you own a licensing fee to use that software for commercial use. The worst thing is that licensing fees may be over $8,000. It can be that much expensive in some cases. Therefore, it is wise to stay conscious of how you may use the software for non-commercial, personal, or commercial purposes.

The third way by which open-source software programmers are paid is through a paid open-source license. Some of you might ask how software can be on the open-source license if it happens to be a paid software. A paid software is called open-source if the programmer of the software allows you to see their code.

The fourth way by which these programmers earn money is by recurring license fees for the open-source software. This is like most of the open license programs. They will let you download and test their software free of charge. They would let you see its code as well so that you know how the software works. However, if you want to have the software's legal rights, you will have to pay a yearly or monthly fee for that. This is much cheaper than a one-minute licensing fee that is too much expensive.

The Linux Shell

Now I will move on to the Linux shell of the Linux operating system. The shell of any operating system is the screen by which you interact with that system. Take the example of Microsoft Windows. The Windows shell is its graphical user interface where you can see the mouse pointer at work. You use a pointer to navigate the screen and click on different desktop elements such as icons and folders.

The shell is generally of two types, the first being the graphical user interface (GUI) and the second being the line user interface (LUI). The LUI appears to be as DOS prompts. If you ever have the opportunity to work on the Microsoft DOS prompt, you should know that the screen you see and work on is the line user interface (LUI).

It is a black-and-white screen. You see a bunch of commands on the screen to get a specific output from your computer.

Linux is a technical operating system, which is why programmers, engineers, and geeks prefer it. They prefer to use this line user interface because it facilitates them in programming. When you install Linux, you can install the Linux graphical user interface on your system, just like Windows. Here, you can use a mouse to click on things or access a line user interface more suitable for programmers. However, the line user interface on Linux works on a bunch of commands.

You should keep in mind that the line user interface on Linux is more robust than the graphical user interface (GUI). However, when you install Linux with a line user interface for the Linux shell, you see a prompt instead of a mouse. If you do not know what a command prompt is or what you will do with it, you will most likely be stuck. To help you out, I will give functional examples of Linux commands and shell scripts.

Root

The next concept that you must chew down and digest is that the concept of the root. In the Linux operating system, root relates to the top level of anything. When you work on Linux, you will hear more often about the word ‘root.’ A root user refers to a computer's administrator. A root user is the highest-level user that anyone can be on a computer system. If you can log in as a root user, you can use any command and do anything with the computer. A root user refers to the root of the Linux operating system. It is the point where the operating system is installed on your hard drive. If you think about this in terms of the Windows operating system, C:/ is the operating system root because it is where you have installed the Windows operating system.

A root user has several privileges and the highest level of access that any user can reach. The Linux operating system has many folders, and the home folder is packed up with the user’s data like

settings, documents, programs, etc. Therefore, the home directory can be the highest level for a specific user. The most important thing to keep in mind is that when you talk about the root user in the Linux operating system, the root level is the highest level. Root users have complete access to anything in an operating system. Once we move toward typing commands and making the Linux operating system do different tasks, we will realize how important the root user concept is. Where we will be blocked while logged in as a non-root user, we will have access there as a root user.

Reasons to Use the Linux Operating System

The reason you should learn about the Linux operating system is the functionality of the server. The Linux operating system is incredibly rock-solid. Once you have installed the Linux operating system and once you have gone through the quirks and set up the configurations, a Linux operating system will run without overheating and dying in the middle of working. It would run on end. Once you have installed the Linux operating system correctly, it has the power to run for a hundred and fifty days without shutting down.

The Linux operating system is unlike Windows in the sense that you have to reboot it on a weekly basis to avoid certain losses of memory. If you have configured it properly, the Linux operating system would run and do the job with the least concern about the circumstances. There will be little to no operational problems when you install a Linux operating system on your computer.

Installation of the Linux Operating System

In this section, I will walk you through how to install the Linux operating system to try it out and have a feel for the process of using the Linux operating system. I will explain the installation process of the Ubuntu Linux server edition. This is open-source and free to use. Whether you will use Ubuntu for commercial or personal use, there will be no charges. Google and open up the website of Ubuntu and download the ISO file to install the Ubuntu server edition. Burn the file on a disk or load it up on a USB thumb drive. I hope you already

know about the burning processor making bootable thumb drive. I am not delving into that to save time.

Since it is a server edition, there will be no graphical user interface on your screen. You will see a line user interface, which in simple words means a black-and-white screen. There will be a blinking cursor on your screen. You have to write commands to move further with your work, or you will be stuck. Let me answer the question that is popping up in your heads as to why the screen is black and white in the Linux operating system's server edition. The biggest reason is safety. The server version of the Linux operating system has a line user interface because every function is like an attack vector for a black hat hacker. Any program that you install on your computer's graphical interface gives a hacker a loophole to enter your system. As there is no graphical interface or external apps in the Linux operating system, the threat of hacking attacks is minimum.

Although the MAC operating system is secure, hackers have deciphered the art of hacking into the Adobe Flash applet that the Mac operating system uses. Over the years, hackers have learned the art of hacking into the Adobe Flash applet that the Mac operating system has been using. Even though the Mac operating system is solid in terms of security, the Adobe Flash has turned into a security vulnerability. Hackers are now in a position to take over a Mac computer with the help of the Flash software. However, the Linux operating system is free of this vulnerability. Let us move on to the process of installing a Linux operating system on your computer.

The first step for the installation is to download the Ubuntu Linux server operating system. Go to ww.ubuntu.com. On the home page of Ubuntu, you will find different versions of Ubuntu to do several things. You will find a desktop version of Ubuntu. You will also see a netbook version. There will be a cloud version as well, along with a version for tablets and cell phones. You have to locate the server version of Ubuntu and find its link for download.

If you are looking forward to installing Ubuntu on your computer system, you should make sure that the file boots off a DVD and not a

hard disk drive. You can do this by pressing the DELETE or F1 key on the keyboard to enter the motherboard's BIOS settings. From there on, you may set up the point where you may want to boot the system from. Several motherboard manufacturers have different ways to enter the BIOS settings, so you should make sure that you consult the user’s manual of the motherboard on how you should do this.

Once you have downloaded the file and made it ready for booting from a CD or DVD, you may proceed to install it. You will be asked in which language you want to install Ubuntu Linux operating system. Choose the desired language and hit Enter. Linux will be installed on your computer system.

Chapter Two: Working with Linux Commands

When we talk about the Linux command line, we refer to the Linux shell. The Linux shell is a program that accepts keyboard commands and passes them on to your operating system for execution. All Linux distributions have a shell program, namely , bash. The bash is the short form of Bourne Again Shell. This is a reference to the fact that bash is the next level version of sh, which was the original UNIX shell program.

When you are using a graphical user interface, you need another program, namely a terminal emulator for interaction with the Linux shell. If you look through the desktop menus, you will be able to locate one.

Let's Start

It is time to start working on the command line of your Linux operating system. The first step that you need to take is to kick off the terminal emulator. Once you have opened it, you should be able to see the following line.

linuxbox1:~$

This is known as the shell prompt. You will see that whenever the Linux shell is ready to take in the input. It may vary in appearance based on the distribution you are using. If you see a (#) instead of a dollar sign ($), the terminal session will offer you superuser privileges. Either you will be logged in as a root user, or you have selected a terminal emulator that offers superuser privileges. Let us type something on the shell and see what it does for you.

linuxbox1:~$ hkhkhhkhk

/bin/sh: hkhkhhkhk: not found

If you want to check on your command history, you may check it out by pressing the up-arrow key. Whether you have written and entered a hundred commands, you will see all of them on the screen. When you press the down-arrow, the command history will disappear. You

might be thinking that the command line is all about the keyboard. However, that is not the case. You may use a mouse with the terminal emulator.

The First Commands for Your Linux Operating System

It is time to enter your first commands on your Linux system to see how the terminal works. These are the simplest commands that you will be using to do some common tasks. The first command on the line is the calendar command by which you can display a full calendar of the current month.

```
linuxbox1:~$ cal
    January 2021
Su Mo Tu We Th Fr Sa
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24  25 26 27 28 29 30
31
```

You can experiment with the date and time command as well. By entering a simple command, you may display the current time and data on your screen. See the following command.

```
linuxbox1:~$ date
Sun Jan 24 10:35:23 UTC 2021
```

When you are using an operating system, you will need to know the exact space on your system to keep the functioning of the system

smooth and easy.

```
linuxbox1:~$ df
Filesystem      1K-blocks      Used Available Use% Mounted on
/dev/root         5120000   2417728   2702272  47% /
devtmpfs           125912         0    125912   0% /dev
tmpfs              126068         8    126060   0% /run
none               126068         0    126068   0% /dev/shm
```

If you want to see the amount of free memory, you will have to enter the free command.

```
linuxbox1:~$ free
              total        used        free      shared  buff/cache   available
Mem:         252140        5252      245980           8         908      243104
Swap:             0           0           0
```

When you have finished your work and you want to end your terminal session, you simply write exit on the command line and close the terminal session.

Basic Commands

This section of the chapter will walk you through the Unix commands and utilities. The first on the line is the cat command that will display the contents of a file. I will use the system information file and then get back to the shell prompt. This is one of the easiest commands to understand. You can get the output of one or more than one file and read through them.

Note: I will use Fedora distribution to test commands from now onwards.

File Navigation in Linux

Most of you might have been familiar with a graphical user interface. It is very easy to navigate in that system. You see a perfect file tree to see different files on your system. You can open them and navigate through them in seconds. However, that is not the case in Linux. There is no graphical display of files in a Linux operating system. It's better to look at the Linux file system as a maze. The directory in which you are currently present is the current directory. The directory above you is the parent directory. You can display the current working directory with the pwd command.

[gha@localhost ~]$ pwd

/root

The ls command

The ls command will display the contents of one of the directories in your system. The default is the current directory. You can use ls -1 command to display a lengthier version of the file. You can also use the ls –F command to display the information of the type of files.

[gha@localhost ~]$ ls

bench.py hello.c

You can change the current working directory by using the cd command. Just type cd in the shell and the pathname of the directory you want to work in. A pathname is a general route that you take along the branches of a tree to reach the directory you want. You can specify it in two ways. The first way is the absolute pathnames. An absolute pathname starts with the root directory, and it follows the tree branch by branch until the path to file or desired directory stands completed. Take the example of a directory on your system in which most of the system's programs are installed.

[gha@localhost ~]$ cd /usr/downloads

[gha@localhost downloads]$ pwd

/usr/downloads

[gha@localhost downloads]$ ls

This command will list the files in that directory. The ls command is one of the widely used commands in a Linux operating system. With this command's help, you can see the contents of a directory and determine the number of files and directory attributes. As you have seen, you can enter the ls command to see the list of files and subdirectories that are contained inside a current working directory.

[gha@localhost ~]$ ls /usr

bin games include lib lib64 libexec local sbin share src tmp

You can list the home directory of the user and the /usr by using the following command.

[gha@localhost ~]$ ls ~ /usr

/root:

bench.py hello.c

/usr:

bin games include lib lib64 libexec local sbin share src tmp

You can add -1 to the ls command to change the format of the display and reveal more details. By adding -1 to the same ls command, you can change the output to a longer format.

Options and Arguments Commands

Most Linux commands are made up of one character preceded by a dash. However, some are longer, with two dashes followed by a word, such as those in the GNU support project. Some commands even allow two or more shorter options to be tagged to one another. In the example you see below, we will give the ls command two

options – the first (l) will reduce the long output, while the second (t) sorts the results by modification time.

Let us see the result of the –l command.

[gha@localhost ~]$ ls -l

total 8

-rw-r--r-- 1 root root 114 Dec 26 15:19 bench.py

-rw-r--r-- 1 root root 185 Sep 9 2018 hello.c

Now I will add t to –l to prolong the output if it can be prolonged.

[gha@localhost ~]$ ls -l

total 8

-rw-r--r-- 1 root root 114 Dec 26 15:19 bench.py

-rw-r--r-- 1 root root 185 Sep 9 2018 hello.c

You can reverse the order of the output by using the reverse command. See the syntax of the command as under:

[gha@localhost ~]$ ls -lt --reverse

total 8

-rw-r--r-- 1 root root 185 Sep 9 2018 hello.c

-rw-r--r-- 1 root root 114 Dec 26 15:19 bench.py

Let’s take a look at some miscellaneous commands that you can pair up with the ls command.

[gha@localhost ~]$ ls is used to produce a list of all the files with proper names that start with a period. It also produces those files that are usually listed as hidden.

[gha@localhost ~]$ ls --all is the long option for the same command.

[gha@localhost ~]$ ls –d is used to specify a directory and display its contents. It does not display the directory. You can pair up this command with the –l command to explore details about the directory instead of the contents.

[gha@localhost ~]$ ls --directory is used to produce more details about the directory or its contents.

[gha@localhost ~]$ ls –F is used to append a particular indicator character to the tail of listed names, like the forward-slash of the name is already in the directory.

[gha@localhost ~]$ ls --classify is used to append the indicatory character to the tail of a listed name.

[gha@localhost ~]$ ls --human-readable is used to display the contents in long format listings and display the files in human-readable formats rather than in the form of bytes.

[gha@localhost ~]$ ls –S is used to sort out the results by the size of the files.

[gha@localhost ~]$ ls –t is used to sort out the results by the modification time.

Finding Out the Type of File

As we begin to explore the system, one of the most useful things you should learn is what is in the files. To do this, we use the file command to help us work out the file type. Linux doesn't require the filenames to specify the file contents, so, for example, where you would expect a .jpg file to contain an image, Linux doesn't require this. When the Linux File command is invoked, it prints out a description of the file contents.

[gha@localhost ~]$ file image.jpg

There are lots of file types, and, like many other operating systems, Linux treats everything in the system as a file.

The less command

We use the less command for viewing text files. The Linux operating system contains several files with text readable by the human eye, and the less command gives us an easy way to look through them. Text files need to be examined because some of them will be full of system settings. You see these labeled as configuration files, and all of them are stored in a particular format. When you can read the files, you will gain an idea of the way Linux works, and you will find several of the programs used by your system, called scripts, are stored the same way.

Once the less command has been executed, you can scroll through a text file, back or forward. Let's say you are looking at a file defining user accounts for your system. You could use this command:

```
[gha@localhost ~]$ less /etc/passwd
```

Once the command has been given, you can look at the content of a specific file. If it is a long file, use the arrow keys on your keyboard to navigate the contents. Below you can see the other common keyboard shortcuts that help you navigate files with the less command:

- Page Up to scroll back a page
- Page Down to scroll forward a page
- Spacebar to scroll forward a page
- Up Arrow to scroll up a line on the page
- Down Arrow to scroll down the line on the page
- G to go to the end of the text file
- 1G or g to move to the start of the file on the page
- h to display the help screen on the page
- n to search for the next possible occurrence of different characters
- /characters to search forward to the characters' next occurrence on the page

- q to quit the less command

Your Linux file system is similar to that of other UNIX-like systems, and the Linux File System Hierarchy Standard is the published standard that specifies the design. Below are some of the directories you can explore:

- / is the root directory that plays the role of the starting point of everything.
- /bin is the root directory that contains binaries or programs that ought to be present for a system to boot and run.
- /boot is the root directory containing the Linux kernel. This is the boot loader and contains the drivers needed for the system to boot.
- /var is the root directory which is where data that is most likely to change is stored. You will find a number of pool files, databases, user mail in this directory.
- /var/log is the root directory which contains different log files. It also contains certain records of system activities. The files packed inside the directory are very important. You should monitor it from time to time. The most useful one is /var/log/messages. It is possible that on some systems, you will have to log in as a superuser to access the log files.
- / imp is used to store transient files created by a number of programs. Some configurations may cause the directory to be emptied each time you reboot the system.
- /usr is the largest one in your Linux operating system. It may contain the programs and support certain files that are used by regular users.
- /proc is a special directory because it is not a filesystem used for storing files on the hard drive. Instead, it is a virtual filesystem the kernel maintains, and the files are snapshots into the kernel, are readable, and will tell you the kernel views a computer.

- /root is the major home directory if you are operating a root account.
- /tmp is used to store transient or temporary files. Be aware that some configurations may empty this directory whenever your system is rebooted.
- /usr/bin carries some executable programs that are installed by the Linux distribution. It is not uncommon for the directory to hold a number of programs.
- /usr/lib is a shared library for different programs that sit in /usr/bin.
- /usr/sbin is loaded with many system administration programs.
- /usr/local is a tree that contains programs not part of the actual distribution but still needed for system-wide use. The programs compiled from the source code go into /usr/local/bin, and you will find this on a new install of the Linux system. However, until the system administrator adds files to it, it will remain empty.
- /usr/share is packed up with shared data that the programs in /usr/bin use. This directory includes different things such as icons, configuration files, sound files, and screen backgrounds.
- /usr/share/doc contains documentation files that are organized by the package.
- /lib is filled up with shared library files that are used by core system programs. These programs are similar to DLLs in the Windows operating system.
- /mnt is used in older Linux operating systems. The /mnt directory carries certain mount points for removable devices in the system. The condition is that the devices should be manually mounted.
- /lost+found is used in case of a partial recovery from the filesystem's corruption event. This directory stays empty until your system passes through something bad.

- /media is used on modern Linux operating systems. The /media directory contains mount points for the removable media like CD-ROMs, USB drives, and any other device mounted automatically.

Chapter Three: Files & Directories Commands

This chapter will walk you through the commands to navigate through files and directories in the Linux operating system. You can manipulate directories and files. Some of the tasks are easy to do. When you are using commands for navigation, you have more power and flexibility as compared to using a graphical user interface. While the latter option is easy to perform and is suitable to execute simple tasks, you need the command line for pulling off complicated tasks. Copying HTML files from one directory to another is hard in a graphical user interface; however, it is easier to do in a command-line system. Before I move on to exploring the commands that you can use to manipulate files and directories, I will explain some special characters that you can use in Linux.

Special Characters

When you are discussing Linux with others, you may hear about some special characters that Linux users encounter during operations. Each special character has a job to do, knowing which will make things easier for you.

- * is technically called asterisk or star. It is used for a regular expression and as a glob character.
- ‘ is technically called tick or single quote. It is used for literal strings.
- Special character . is technically called dot. It is used to denote the current directory you are working in or the hostname or file delimiter.
- $ is technically called the dollar sign. It is used for denotation of a variable or for representing the end of the line.
- \ is technically called backslash. It is used for macros and literals.
- / is technically called forward slash. It is used as a search command or a directory delimiter.

- ! is technically called bang. It is used for command history and negation.
- | is technically called a pipe. It is used for command pipes.
- _ is technically an underscore or under. It is used for the cheap subtitle for some particular space.
- ` is technically called backtick or backquote. It is used for the substitution of command.
- “ is technically called double quote. It is used for semi-literal strings.
- {} is technically called braces or curly brackets. It is used for ranges or statement blocks.
- ^ is technically called caret. It is used to denote the start of line and negation.
- [] is technically squares or brackets. It is used for ranges.
- ~ is technically called squiggle or tilde. It is used as a directory shortcut and negation.
- # is technically called pound, hash, or sharp. It is used for preprocessor, comments, and substitutions.

Command-Line Editing

As you move around and explore the Linux shell, you will find out that you have the freedom to move in the shell through arrow keys. This comes as a standard on most Linux operating systems. It is a good idea to know about the standard keystrokes on Linux operating systems.

- CTRL-B is used to move the cursor on the screen to the left side.
- CTRL-F is used to move the cursor on the screen to the right side.
- CTRL-E is used to move the cursor on the screen to the end of a line.

- CTRL-A is used to move the cursor on the screen to the start of a line.
- CTRL-P is used to view any previous commands on the screen. It is used to move the cursor upside.
- CTRL-N is used to move the cursor to the downside, and it is also used to view the next command.
- CTRL-K is used to erase everything from the cursor to the end of a line.
- CTRL-W is used to erase the preceding word from the cursor on the screen.
- CTRL-U is used to erase everything from the cursor to the start of a line.
- CTRL-Y is used to paste the erased text that you have already cut off on the screen by the command CTRL-U. It is similar to copy and paste in word processors.

Linux Commands for Directories

The mkdir Linux command is used for the creation of directories. It works as the following:

mkdir directoryname

The cp command

Another command, the cp command, is used to copy directories and files. You can use it in different ways, such as the following:

cp file1 file2

cp files… directoryname

You can use the following options with the cp command.

- The cp command option –a or --archive is used to copy the directories and files and their entire attributes, including permissions and ownerships. Usually, copies take on average default attributes of a user who is performing the copy action.

- The cp command option –u or --update is used when you are copying from one directory to another, and you are copying files that do not exist or newer to the files in the destination directory.
- The cp command option –v or --verbose is used to display the informative messages as you perform the copy.
- The cp command option –i or --interactive is used before overwriting existing files, which prompts the user for confirmation. If you have not specified the option, the cp command will overwrite the existing files, which is not good for your document records.
- The cp command option –r or --recursive is used to copy the contents and files recursively. This particular option is needed when you are copying directories.

Examples of cp command

- The Linux command cp item1 item2 will copy the files of item1 to the files of item2. If item2 exists, it will be overwritten with the contents of item1. If it does not exis, the system will create it.
- The Linux command –i item1 item2 will copy the files of item1 to the files of item2. However, the only exception is that if item2 exists, the user will be prompted before the file is overwritten.
- The Linux command item1 item2 dir1 will copy contents of item1 and item2 into dir1. The directory dir1 must exist before you execute this command.
- The Linux command cp dir1* dir2 uses a wildcard. It will copy all the files of dir1 into dir2. The condition is that dir2 must exist before the execution of the command.
- The Linux command –r dir1 dir2 will copy the directory dir1 to the directory dir2. If dir2 is non-existent, it will be created and will contain the same contents as the directory dir1 will.

The mv command

The mv Linux command performs movement and renaming of files based on how a user uses it. The original filename does not exist after you have executed the command. The mv command is used in the same way as the cp command.

mv file1 file2

Just like the cp command, the mv command comes with different options that are given as under:

- The mv command option –u or --update is used when you are moving from one directory to another, and you are moving files that do not exist or newer to the files in the destination directory.
- The mv command option –v or --verbose is used to display informative messages as you perform the movement.
- The mv command option –i or --interactive is used before overwriting existing files, which prompt the user for confirmation. If you have not specified the option, the mv command will overwrite the existing files, which is not good for your document records.

Examples for mv command

- the Linux command mv item1 item2 will copy the files of item1 to the files of item2. If item2 exists, it will be overwritten with the contents of item1. If it does not exist, the system will create it. The file item1 will cease to exist in either case.
- the Linux command mv –i item1 item2 will move the files of item1 to the files of item2. However, the only exception is that if item2 exists, the user will be prompted before the file is overwritten.
- the Linux command mv item1 item2 dir1 will move contents of item1 and item2 into dir1. The directory dir1 must exist before you execute this command.

- the Linux command mv dir1* dir2 uses a wildcard. It will move all the files of dir1 into dir2. The condition is that dir2 must exist before the execution of the command.
- the Linux command mv dir1 dir2 will move the directory dir1 to the directory dir2. If dir2 is non-existent, you will have to create it and move the contents of dir1 into dir2.

The rm command

The rm command is used for removing or deleting directories and files like the following:

rm file…

In this command file is the name of one or more than one directory and files. Using this command can be highly treacherous. Once you delete something with the rm command, you will never get it back. Linux assumes that you have removed a file on purpose. It does not give it back to you. Therefore, you should use it carefully. Let us say that you want to delete the HTML files in some directory.

So when you are using wildcards with the rm command, you should test the wildcard with the ls command to see the contents of the file you are going to remove from the system. You can use the arrow key to recall the command and replace it with rm then.

- The mv command option –r or --recursive is used to delete directories recursively. This means that if the directory you are deleting has subdirectories, you can delete them as well. If you want to delete a directory, you should specify this option.
- The mv command option –f or --force is used to ignore the non-existent files. It does not prompt. It tends to override the –interactive option.
- The mv command option –v or --verbose is used to display the informative messages as you perform the deletion.
- The mv command option –i or --interactive is used before deleting the existing files, which prompts the user for

confirmation. If you have not specified the option, the rm command will silently delete the existing files, which is not good for your document records.

Examples of rm command

- The Linux command rm item1 will silently delete files of item1.
- The Linux command rm –i item1 will delete the files of item1. However, the only exception is that the user will be prompted before the file is deleted.
- The Linux command rm item1 dir1 will delete the contents of item1 and dir1. The directory dir1 must exist before you execute this command.
- The Linux command rm –rf item1 dir1 works the same as above, with the exception that if both item1 and dir1 do not exist, the rm command will go on silently.

Chapter Four: Practical Work with Commands

You have gone through a series of commands. Each has some mysterious options and arguments. This chapter will walk you through some more commands to make you more familiar with the Linux command line.

The type command

This is a built-in Linux command that displays a typical type of command that the shell is going to execute. See how you can use it.

[gha@localhost ~]# type type

type is a shell builtin

[gha@localhost ~]# type ls

ls is aliased to `ls --color=auto'

[gha@localhost ~]# type cp

cp is /bin/cp

[gha@localhost ~]# type mv

mv is /bin/mv

[gha@localhost ~]# type rm

rm is /bin/rm

[gha@localhost ~]# type cd

cd is a shell builtin

[gha@localhost ~]# type help

help is a shell builtin

You can see that there are different results of all these different commands.

The which command

Sometimes more than one version of executable programs is installed on a particular system. While this is not common on desktop systems, it is not quite unusual on large servers. You can use the ‘which’ command to know the exact location of an executable.

```
[gha@localhost ~]# which ls
ls=’ls --color=auto’
        /bin/ls
```

The which command works well for executable programs only and not for built-ins or alias that are mostly substitutes for executable programs. When you try to use the which command on shell built-in, you get either no response or a big error message.

```
[gha@localhost ~]# which help
/usr/bin/which: no help in (/usr/local/sbin:/bin:/sbin:/usr/bin:/usr/sbin:/usr/l
local/bin)
```

Documentation of Commands

You can take a step further and fish out the available documentation in the system for each type of command.

The help command

Bash shell has a built-in help facility for shell built-ins. If you want to use it, you can type help and the name of the shell built-in. See the following example.

[gha@localhost ~]$ help cd

cd: cd [-L|[-P [-e]] [-@]] [dir]

Change the shell working directory.

Change current directory to DIR. The default DIR is the HOME Shell variable's value.

The CDPATH variable is used to define the search path leading to the directory where DIR is. CDPATH indicates alternative file names by using a colon (:) to separate them. A null name indicates the current directory, and where a slash (/) precedes a directory name, then CDPATH isn't used.

The details are longer than I have mentioned here. I cut it down to save space. You can type the command on the terminal and see how the help command can help you out when you get stuck as you operate the Linux operating system. When you see square brackets in the description of a command's syntax, you should keep in mind that they indicate only optional items. If you see a vertical bar character in the description, it denotes mutually exclusive items. Some executable programs support the --help option. When you apply the option, it displays a lengthy description of the command's supported syntax and the related options.

[gha@localhost ~]$ mkdir --help

Usage: mkdir [OPTION]... DIRECTORY...

Create the DIRECTORY(ies) if they do not already exist.

Mandatory arguments to long options are mandatory for short option s too.

-m, --mode=MODE set file mode (as in chmod), not a=rwx - umask

The man command

Most executable programs intended for command-line use offer a formal piece of documentation known as a man page or a manual page. A special paging program known as man is loaded up on Linux to view the details. You can type the man command and then the title of the command. The man pages differ in format and contain a title or a synopsis of the command's syntax, a listing, and a description of the purpose of the command. The man page does not include examples, and they are intended as a special reference and not as a tutorial. See the following example.

[gha@localhost ~]$ man ls

LS(1) User Commands LS(1)

NAME

ls - list directory contents

SYNOPSIS

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

DESCRIPTION

List information about the FILEs (the current directory by default).

Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.

On most of the Linux operating systems, man uses less for displaying the manual page so that all familiar less commands work while the page is displayed. The manual that man displays has different sections, and it covers user commands and system administration commands. In addition to that, it also covers file formats, programming interfaces, and much more. See the complete layout of the manual as under:

- The number 1 section of the man command will display user commands.
- The number 2 section of the man command will display programming interfaces for the kernel system calls.
- The number 3 section of the man command will display programming interfaces to the C library.
- The number 4 section of the man command will display special files such as the device nodes and the drivers.
- The number 5 section of the man command will display formats of the files.
- The number 6 section of the man command will display the details of games and entertainment like screensavers.
- The number 7 section of the man command will display miscellaneous items.
- The number 8 section of the man command will display the commands related to system administration.

At times, you may need to look into a specific section of a manual to find out what you are looking for. If you do not specify a section number, you will always get the very first instance of a specific match in section 1.

The apropos command

It is possible to search the list of man pages for some possible matches based on a specific search term. This crude approach is often helpful. The syntax is as under:

```
[gha@localhost ~]$ apropos floppy
```

The whatis command

The whatis command in Linux operating displays the name and a short description of a man page. The syntax is as under:

```
[gha@localhost ~]$ whatis ls
```

The man pages that come with a Linux distribution system are labeled as a reference documentation instead of a tutorial. A number of man pages are tough to read, so tough that a majority of us skip them right away without giving them a second glance. However, the most difficult of the man pages are of bash. They are long, complicated, and almost unreadable. However, their length becomes their strength as they contain plenty of useful information for you to add to your knowledge. When you have gone through one page, it will start making sense to you.

The info command

The GNU project provides info pages as an alternative option to the man pages. A reader program called info displays these pages, and these are hyperlinked in the same way web pages are. For example:

```
[gha@localhost ~]$ info ls
```

Next: dir invocation, Up: Directory listing

10.1 ‘ls’: List directory contents

In the ‘ls’ program, you will find information about files of any type, and that includes directories. As usual, arbitrary mixing of options ad file arguments is accepted.

By default, ‘ls’ will list the contents of non-option command-line arguments that are also directories, but this is not a recursive list, and files starting with ‘.’ are left out.

The info program will read the info files – these are structures that go into individual nodes, each containing one topic. Hyperlinks in the info files help users move between the nodes, and these hyperlinks are identified by a leading asterisk. Typically, activating these hyperlinks requires the user to click them with the cursor and press Enter. The info command provides this information:

- The ? key in the info menu will display the command help.
- The Page up or Backspace keys in the info menu will display the previous page.
- The Page down or Spacebar keys in the info menu will display the succeeding page.
- The n key in the info menu will display the next node in the menu.
- The p key in the info menu will display the previous node in the menu.
- The u key in the info menu will display the parent node of the node that is presently on display.
- The Enter key in the info menu will allow you to follow the hyperlink at the cursor's present location.

- The q key in the info menu will allow you to quit the menu.

Chapter Five: Redirection Commands & Keyboard Tricks with Linux Commands

In this chapter, we will talk about the coolest feature of the command line, which is called I/O redirection. The I/O stands for input/output. With this facility, you may redirect the input and output of commands and connect multiple commands to make a powerful command known as pipelines.

The output of a program is of two types. First, you have the result of the program which contains the data program produces. Secondly, there are status and error messages that instruct you how the program is getting along. If you look at the command ls, you will see that it delivers results and error messages on your screen.

This program sends the output to another file called standard error. Both standard error and output are connected to the screen, and they are saved into a disk file. Many programs take input from a facility known as standard input stdin, which is attached to a keyboard.

I/O redirection allows you to change where the output goes and where the input flows in from. Usually, output goes to the screen, and the input flows in from the keyboard in Linux; I/O has the power to change that.

Redirection

I/O redirection helps you in redefining the direction of the output. To redirect the standard output to a file instead of the screen, you have to use the redirection operator that is followed by the name of the file. It is useful more often to store the output of a certain command inside a file. You may tell the shell to direct the ls command's output to a file namedls-output.txt instead of the screen.

```
[gha@localhost ~]$  ls -1 /usr/bin > ls-pt.txt
```

I have created a long listing of /usr/bin directory and then sent the results to the file ls-pt.txt.

Keyboard Tricks

Unix is an operating system for people who love to type. The command line does not allow a mouse to operate. Linux commands can be exhaustive, therefore, you should learn a bunch of keyboard tricks.

Text Modification

The following list shows how you can modify text in the Linux command line. The terms killing and yanking refer to cutting and pasting, respectively.

- The keyboard trick CTRL-D will help you delete the character that is at the present position of the cursor.
- The keyboard trick CTRL-T will help you transpose the character at the present location of the location with the one that is preceding it.
- The keyboard trick Alt-U will help you convert into uppercase the characters from the cursor's present position to the ending point of the word.
- The keyboard trick Alt-L will help you convert into lowercase the characters from the cursor's present position to the ending point of the word.
- The keyboard trick Alt-T will help you transpose the words at the cursor's present position with the preceding one.
- The keyboard trick CTRL-K will help you kill text from the cursor's present position to the ending point of a line.
- The keyboard trick CTRL-U will help you kill text from the cursor's present position to the starting point of a line.
- The keyboard trick ALT-D will help you kill text from the cursor's present position to the end of a current word.
- The keyboard trick ALT-Backspace will help you kill text from the cursor's present position to the starting point of the current

word. If the cursor is presently at the start of a word, it will also kill the previous word.

- The keyboard trick CTRL-Y will help you yank text from the kill-ring and paste it at the cursor's present position.

Completion Commands

Another way by which the shell will help you is through completion. This occurs when you hit the Tab key while you are typing the command. When you are halfway through a command, you can enter tab to complete the command. The important thing to remember is that you should not hit the Enter key.

Completion commands

- The keyboard trick Alt-$ will help you display the list of all possible completions. You may also do this by pressing the Tab key twice. This is much easier to do.
- The keyboard trick Alt-* will help you insert the possible completions. This is highly useful when you intend to use more than one match.

Searching History

Bash maintains a history of the commands you type in it. The list of commands is kept in the home directory in a file named .bash_history. The history facility is highly useful for cutting down on the amount of typing that you need to do especially when you combine it with command-line editing. Here is the syntax of the history command.

[gha@localhost ~]$ history | less

Bash, by default, stores the last 500 commands that you have entered. Suppose you want to find the commands that you used to list /usr/bin. Here is the way to do that.

[gha@localhost ~]$ history | grep /usr/bin

There is a list of keystrokes that you may use when you are navigating through the history command.

- You can use the keyboard shortcut CTRL-P to move to the latest history entry. This performs the same action as the up arrow.
- You can use the keyboard shortcut CTRL-N to move to the next history entry. This performs the same action as the down arrow.
- You can use the keyboard shortcut ALT-> to move to the bottom of the list of history that is the current command line.
- You can use the keyboard shortcut ALT-< to move to the starting point of the list of history.
- You can use the keyboard shortcut CTRL-R to reverse the incremental search. This option incrementally searches from the current command line up the list of history.
- You can use the keyboard shortcut CTRL-O to execute the present item in the list of history and then move on to the next one. This is handy if you are looking forward to re-execute a sequence of commands in the list.
- You can use the keyboard shortcut ALT-N to forward the search. This is non-incremental.
- You can use the keyboard shortcut ALT-P to reverse the search. This also is non-incremental. Coupled with this keyboard shortcut, you can type the search string and then press ENTER before you have performed the search.

History Expansion

The shell offers a special type of expansion for different items in the history list by using the ! character. We already have seen how a number may follow the exclamation point to insert a particular entry from the list of history. There are many other expansion features.

History expansion mechanism has many available elements, but this subject is too arcane.

- You can type the sequence !! to repeat the latest command. However, it may be easier to press the up arrow and then hit ENTER.
- You can type the sequence !?string to repeat the latest history list item that contains a string.
- You can type the sequence !string to repeat the latest history list item that starts with a string.
- You can type the sequence !number to repeat the latest history list item.

Chapter Six: Process Commands

Today's operating systems are capable of multitasking, which means they can do more than one thing at a time. They switch rapidly between executing programs, and the Linux kernel manages this. Linux uses processes to organize the programs queued for execution in the CPU.

Sometimes, a computer system can become slow, or a specific application might stop working. This chapter is designed to show you the command-line tools you can use to terminate certain processes that don't work properly during the system operations.

When your system wakes, certain activities are activated as processes by the kernel. This is followed by a program called init being launched. This program runs shell scripts known as init scripts, which are used to start the system services. Lots of these services are implemented as daemon programs, working in the background and doing what they have to do without a user interface. You can find the processes command here:

```
[gha@localhost ~]$ ps
PID TTY          TIME CMD
  48 hvc0     00:00:00 sh
  82 hvc0     00:00:00 ps
```

The result lists process that are sh and ps respectively. If we add an option to the command ps, we can detailed result on our screens. See the following example.

```
[gha@localhost ~]$ ps x
  PID  TTY      STAT   TIME COMMAND
    1 ?        S      0:00 /bin/sh /sbin/init
    2 ?        S      0:00 [kthreadd]
    3 ?        I      0:00 [kworker/0:0]
```

```
  4 ?        I<     0:00 [kworker/0:0H]
  5 ?        I      0:00 [kworker/u2:0]
  6 ?        I<     0:00 [mm_percpu_wq]
  7 ?        S      0:00 [ksoftirqd/0]
  8 ?        S      0:00 [kdevtmpfs]
  9 ?        I<     0:00 [netns]
 10 ?        S      0:00 [oom_reaper]
 11 ?        I<     0:00 [writeback]
 12 ?        I<     0:00 [crypto]
 13 ?        I<     0:00 [kblockd]
 14 ?        S      0:00 [kswapd0]
 15 ?        I      0:00 [kworker/0:1]
 33 ?        S      0:00 [khvcd]
 43 ?        Ss     0:00 dhcpcd
 48 hvc 0    Ss     0:00 sh -l
 71 ?        I      0:00 [kworker/u2:1]
102 hvc0     R+     0:00 ps x
```

The x option informs ps that it needs to show all the processes, no matter what terminal controls them. Because the system is running many processes, a long list is produced, and more often than not, it is helpful to pipe the ps output into less and view them that way. Some combinations of options produce many output lines, and it is a good idea to maximize the emulator window in the terminal.

Process States

Here is a rundown of process states.

- The process state R denotes running. This process is either running or is ready to run in the system.
- The process state S denotes Sleeping. The process is in Sleep mode and not running. It is waiting for a certain event to start, such as a keystroke or network packet.
- The process state Z denotes the zombie process. This is a child process that has been terminated, but the parent process has not cleaned it up.
- The process state D denotes Uninterruptable sleep. This process is waiting for the I/O like a disk drive.
- The process state T denotes stopping. This process is instructed to stop.
- The process state N denotes a low-priority process. It's a good process that will obtain the processor time when all other high-priority processes have been serviced.
- The process state < denotes a high priority process. It is possible to give away more importance to a process and allowing it more time on your CPU. A process that has higher priority is less nice because it takes more of the CPU's time.

Other characters may follow this process and these indicate characteristics of other processes. The aux command can be used with ps to provide another set of options:

```
[gha@localhost ~]$ ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  1.3   3136  2516 ?        S    14:35   0:00 /bin/sh /sbin/i
root         2  0.0  0.0      0     0 ?        S    14:35   0:00 [kthreadd]
root         3  0.0  0.0      0     0 ?        I    14:35   0:00 [kworker/0:0]
root         4  0.0  0.0      0     0 ?        I<   14:35   0:00 [kworker/0:0H]
```

```
root         5  0.0  0.0      0     0 ?        I    14:35   0:00 [kworker/u2:0]
root         6  0.0  0.0      0     0 ?        I<   14:35   0:00 [mm_percpu_w
q]
root         7  0.1  0.0      0     0 ?        S    14:35   0:02 [ksoftirqd/0]
root         8  0.0  0.0      0     0 ?        S    14:35   0:00 [kdevtmpfs]
root         9  0.0  0.0      0     0 ?        I<   14:35   0:00 [netns]
root        10  0.0  0.0      0     0 ?        S    14:35   0:00 [oom_reaper]
root        11  0.0  0.0      0     0 ?        I<   14:35   0:00 [writeback]
root        12  0.0  0.0      0     0 ?        I<   14:35   0:00 [crypto]
root        13  0.0  0.0      0     0 ?        I<   14:35   0:00 [kblockd]
root        14  0.0  0.0      0     0 ?        S    14:35   0:00 [kswapd0]
root        15  0.0  0.0      0     0 ?        I    14:35   0:00 [kworker/0:1]
root        33  0.0  0.0      0     0 ?        S    14:35   0:00 [khvcd]
root        43  0.0  0.7   1944  1472 ?        Ss   14:35   0:00 dhcpcd
root        48  0.0  1.5   6204  2940 hvc0     Ss   14:35   0:00 sh -l
root        71  0.0  0.0      0     0 ?        I    14:35   0:00 [kworker/u2:1]
root       160  0.0  1.4   8180  2740 hvc0     R+   15:02   0:00 ps aux
```

This command displays the processes that belong to all users of a system. When you use the options without a leading dash, it invokes the command with BSD-style behavior. Here are the details of BSD-Style ps Column Headers.

- The BDS-style ps column header, namely USER alludes to User ID. This denotes the owner of a process.
- The BDS-style ps column header called RSS indicates Resident Set Size. This denotes the amount of physical memory (RAM) used by the process.

- The BDS-style ps column header, namely VSZ alludes to the virtual memory size of the system.
- The BDS-style ps column header, namely %CPU alludes to the CPU usage in percentage.

Using the Top Command to View Processes

The ps command reveals a lot about what your Linux machine is doing. However, it provides you just a snapshot of the state of the machine as the ps command is executed in the system. To see a dynamic view of the machine's activity, I will use the top command.

```
[gha@localhost ~]$ top
top - 15:10:01 up 34 min,  0 users,  load average: 0.05, 0.05, 0.02
Tasks:  20 total,   1 running,  19 sleeping,   0 stopped,   0 zombie
%Cpu(s):  1.9 us,  4.9 sy,  0.0 ni, 93.2 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :    182.0 total,    171.2 free,      4.9 used,      5.8 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.    172.1 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
  180 root      20   0    8424   3084   2584 R   7.4   1.7   0:01.82 top

    1 root      20   0    3136   2516   2148 S   0.0   1.4   0:00.95 init
    2 root      20   0       0      0      0 S   0.0   0.0   0:00.00 kthreadd
    3 root      20   0       0      0      0 I   0.0   0.0   0:00.00 kworker/0+
    4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker/0+
    5 root      20   0       0      0      0 I   0.0   0.0   0:00.00 kworker/u+
    6 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 mm_percpu+
    7 root      20   0       0      0      0 S   0.0   0.0   0:02.45 ksoftirqd+
```

```
  8 root      20   0       0      0      0 S   0.0   0.0   0:00.10 kdevtmpfs
  9 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 netns
 10 root      20   0       0      0      0 S   0.0   0.0   0:00.00 oom_reaper
 11 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 writeback
 12 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 crypto
 13 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kblockd
 14 root      20   0       0      0      0 S   0.0   0.0   0:00.00 kswapd0
 15 root      20   0       0      0      0 I   0.0   0.0   0:00.03 kworker/0+
 33 root      20   0       0      0      0 S   0.0   0.0   0:00.00 khvcd
 43 root      20   0    1944   1472   1196 S   0.0   0.8   0:00.22 dhcpc
d
 48 root      20   0    6204   2940   2380 S   0.0   1.6   0:01.04 sh
 71 root      20   0       0      0      0 I   0.0   0.0   0:00.01 kworker/u+
```

As you enter the top command in the command line, you will see that the figures are continuously updating. They update every three seconds.

Chapter Seven: Working with Linux Editors

Before moving on to Linux shell scripting, we should make ourselves acquainted with the text editors that we will have to use to write shell scripts. This chapter will walk you through the features like searching, copying, cutting, and pasting. The more practice you do to learn an editor, the faster you will learn to write shell scripts in Linux.

The Vim Editor

If you are working in the command line mode, you may need to become familiar with a text editor that will be operating in a Linux console. The vim editor is the original editor that Unix uses. It makes use of the console graphics mode for the emulation of a text-editing window, which allows you to see different lines of the file, move around across the files, and edit, insert or replace a piece of text. The vim editor works well with the data that is in a memory buffer. You have to type vim and the name of the file that you have to edit to open the editor with the desired file.

If the editor is started without a filename being supplied, it opens but with no file. The vim editor detects the session's terminal type and uses full-screen mode so the console window can use the editor area. The initial window will show the file contents and a message line at the bottom of the window. If the contents don't take up the entire screen, a tilde is placed on the lines excluded from the file. The vim editor has two operational modes – normal and insert mode. When you open a file for editing, vim goes into normal mode, and certain keystrokes are interpreted as commands.

In the insert mode, each key typed at the cursor is inserted into a buffer. To get into insert mode, press the i key, and to get out of it and back to normal mode, press the ESC key.

You can use the arrow keys in normal mode to move around the text so long as vim has detected the correct terminal type. The vim command includes those for cursor navigation, as you can see below:

- The command h in vim Linux editor is used to move to the left by one character.
- The command l in vim Linux editor is used to move to the right by one character.
- The command j in vim Linux editor is used to move down the cursor by one line.
- The command k in vim Linux editor is used to move up the cursor by one line.
- The command PageDown or Ctrl-f in vim Linux editor is used to move forward one screen of your data.
- The command PageUp or Ctrl-b in vim Linux editor is used to move forward one screen of your data.
- The command gg in the vim Linux editor is used to shift to the first line in the buffer.
- The command G in vim Linux editor is used to move to the last line in the buffer.
- The command num G in vim Linux editor is used to move to the line number in the buffer.

The vim editor carries a special feature in the normal mode that is known as the command line mode. The command line mode offers an interactive command line where you may enter additional commands to control different types of actions in the vim editor. To get to the command line mode in the vim editor, you have to press the colon key while you are in the normal mode. The cursor will move to the message line. Here you will see a colon popping up on the screen which indicates that you should now enter a command.

When you are in the command line mode, you may enter several commands to save the buffer to file and move out of the editor.

- The command q in vim Linux editor is used to exit the system if you have made no changes to buffer data.

- The command wq in vim Linux editor is used to save buffer data to file and then exit the editor.
- The command q! in vim Linux editor is used to quit the editor and then discard the changes that are made to the data of buffer.
- The command w *filename* in vim Linux editor is used to save your file with a different filename.

Data Editing

When you are in the insert mode, you may insert data in the buffer. Sometimes you have to add or remove some data after entering into the buffer. While you are in normal mode, the vim editor will provide certain commands for editing data in the buffer. You should take extra care when you try to use a PC keyboard Delete or Backspace keys while you are inside the vim editor. The vim editor recognizes the keyboard Delete key as an x command functionality, which will delete a character at the current cursor location.

Copy & Paste

A standard feature of these editors is their ability to make a cut or copy data, and after that, paste it into the document. Cutting and pasting text is easy. When vim removes your data, it will forward it into a separate register. You can then retrieve that data by using the p command. You can use the dd command to remove different lines of text. When you try to copy text, you may find it a bit trickier.

- x in vim Linux editor is used to delete a character at the cursor's current position.
- R *text* in vim Linux editor is used to overwrite the data at the cursor's current position with the text until you hit the Escape key.
- r *char* in vim Linux editor is used to replace any single character at the cursor's current position with char.

- dd in vim Linux editor is used to delete a line at the cursor's current position.
- A in vim Linux editor is used to append data to the end point of the line at the cursor's current position.
- a in vim Linux editor is used to append data next to the cursor's current position.
- J in vim Linux editor is used to delete a line break to the end of the line at the cursor's current position.
- dw in vim Linux editor is used to delete a word at the cursor's current position.
- d$ in vim Linux editor is used to delete the text to the ending point of a line from the cursor's current position.

The KDE Editor Family

If you are using a Linux distribution that uses KDE desktop, you will see a couple of options in relation to text editors. The KDE project will officially support a couple of editors, such as Kate and KWrite. Both are graphical text editors containing a number of advanced features and some extra niceties that are not found in some common standard editors. This section will describe each editor and also the features that you may use in shell scripting.

The KWrite Editor

This is the basic editor for the KDE environment. It offers word-style text editing with support for code syntax editing and highlighting and is used for different types of programming languages. It uses color-coding to distinguish comments, constants, and functions. You should notice that the for loop carries an icon that connects the opening and closing braces. The editing window provides cut-and-paste capabilities.

The editor has many command line parameters you may use to customize how it starts.

- The Kwrite editor command --stdin triggers the editor to read the data from a standard input device in place of a file.
- --column triggers the editor to specify a column number in a file to initiate the process in the window of the editor.
- --encoding triggers the editor to specify the type of character encoding for a file.
- --line triggers the editor to specify the number of a line in a file to initiate in the window of the editor.

The KWrite editor offers a toolbar and a menu bar at the top of the window used for editing, which allows you to select different features and then change the editor's configuration settings. The menu bar will contain the following items.

- File helps you load up, save, export or print different pieces of text from different files.
- Edit aids you in manipulating text in buffer.
- View helps you manage the appearance of the text in the editor of the window.
- Bookmarks helps you handle the pointers to get back to certain locations in the text.
- Tools has some specialized features for manipulation of the text.
- Settings helps you configure the way your editor handles the text.
- Help will give you the information about the editor and its commands.

The Edit Menu of KWrite Editor

Here is a rundown of different commands of the Edit menu of the KWrite editor.

- Undo is used to reverse your latest action in the editor.

- Redo is used to reverse the latest action you undid.
- Cut is used to delete the text that you have selected and then placed in the clipboard.
- Copy is used to copy any piece of text that you have selected to the clipboard.
- Copy as HTML is used to copy the selected text to the clipboard as the HTML code.
- Paste is used to insert the current content of the clipboard at the present position of the cursor on the screen.
- Select All is used to select the entire text in your editor.
- Deselect is used to deselect the piece of text that you have currently selected.
- Block Selection Mode permits you to select a piece of text between the columns instead of all the lines.
- Overwrite Mode is used to toggle between insert mode and overwrite mode by replacing text with newly typed text instead of inserting the new text.
- Find is used to produce the Find Text dialog box, which allows you to customize the text search.
- Find Next is used to repeat the last find operation forward in the buffer.
- Find Previous is used to repeat the last find operation backward in the buffer.
- Replace is used to bring about on the screen the Replace With Dialog box. You can use it to customize a text search.
- Go to Line is used to produce the Goto dialog box. It allows you to enter a certain line number. The cursor jumps to a specified line.

The KWrite Editor Tools

KWrite offers a wide range of tools for users to write and edit their shell scripts. Some of the tools are as under:

- The KWrite editing tool, Read Only Mode will lock the text so that no changes may be made while you are still using the editor.
- Filetype will select the file-type scheme used in a piece of text.
- Spelling will start the spellcheck program at the starting point of a piece of text.
- Highlighting will highlight the text based on the content like program code or the configuration file.
- Indent will increase the indentation in a paragraph by one.
- Indentation, will automate indentation based on the selection you have made in the editor.
- Spelling will initiate the spellcheck program at the beginning point of the text.
- Spellcheck Selection will initiate the spellcheck program on the selected section of a piece of text.
- Unindent will cut down on the indentation of a paragraph by one.
- Clean Indentation will return the paragraph indentation to the original settings.
- Align will bring back the selected lines' current line to the default indentation settings.
- Uncomment eliminates a comment symbol from the present line based on the type of file.
- Lowercase, will set the selected text or the character at the cursor's present position to lower case.
- Uppercase, will set the selected text or the character at the cursor's present position to upper case.

- Capitalize, will set the first letter of the selected text or the word at the cursor's present position to uppercase.
- Join Lines will pair up the selected text or lines at the present position with the next line into a single line.
- Word Wrap Document will enable word wrapping inside a piece of text. If a line moves past the editor window edge, it will continue on the next line.

The GNOME Editor

If you are operating a Linux operating system through the GNOME editor, you will see a graphical text editor that you can use easily and well. It is a basic text editor that has a couple of advanced features for the fun of editing. When you start gedit with multiple files, it will load the files into individual buffers and display each of them as a tabbed window inside the editor's main window. The left frame inside the gedit editor will show the documents that you have been editing.

Basic Features of the Editor

Besides a window for editing, gedit uses a menu bar and a toolbar that allows you to set up the configure and feature settings. The toolbar provides some real quick access to the menu bar items, which are as under:

- The menu item File helps you load up new files, save existing files, and print different pieces of text from different files.
- Edit aids you in manipulating text in the buffer and for setting up the editor preferences.
- View helps you set up features of the editor related to the display. It also works on setting up the text-highlighting mode.
- Search helps you find and replace pieces of text in the editor's active buffer spot.
- Tools is used to access the plugin tools that are installed in gedit.

- Documents helps you manage different files that are open in buffer areas.
- Help will give you information about the editor and its commands. You will see a complete manual to use the editor and work with the commands.

Preferences

The Edit menu offers you the Preferences item, which allows you to customize the operations of the editor. The Preferences dialog box carries tabbed areas that allow you to set up the features of the editor as per your requirements.

View

The View tab offers many options for displaying the text in the editor window.

- The first option you will find in the View tab is named Text Wrapping. It determines how you can handle long pieces of text in your editor. When you enable the option, it wraps up long pieces of text to your editor's next line. The Do Not Split Words Over Two Lines option bars the auto-inserting of hyphens into long words to prevent the same from splitting in between the two lines.
- The second option you will find in the View tab is named Line Numbers. This option displays the numbers of lines in the left margin in the window of the editor.
- The third option you will find in the View tab is Current Line, highlighting a line where the cursor is positioned. This enables you to find out the cursor position easily.
- The fourth option you will find in the View tab is named Right Margin, which enables the right margin. This option also allows you to set up the number of columns in the editor window. The default value of the columns is 80.

- The fifth option you will find in the View tab is named bracket matching. When you enable this option, it will allow you to easily match brackets when you are writing if-then statements, for loops or while loops, and any other code lines that involve brackets.

The line-numbering and matching brackets offer a handy environment to users. This is easy for troubleshooting and is not found in many other text editors.

Editor Tab

The Editor tab offers options to handle indentation and tabs. It also oversees how you save the document after editing.

- The Tab Stops option in the Editor tab allows you to set up the number of spaces that should be skipped while you hit the Tab key. The default value of skipping is 8. You can reset that. The feature also has a checkbox that inserts spaces rather than tab spaces.
- The second option you will find in the Editor tab is named Automatic Indentation. When you enable that, it allows the gedit to automate line indentation in the text for code elements and paragraphs like if-then statements and the loops.
- The third option you will find in the Editor tab is named File Saving, which offers two features to save files. You have the option to create a backup copy of the file, which has been opened in the edit window. You can also set up the option whether you like to save the file at a preselected interval or not.

Font & Colors

The Font & Colors option offers a configuration of two items.

- The Font option in the Font & Colors tab allows you to select Monospace 10 as a default font or select a customized font style and size from a dialog box.

- The Colors option in the Font & Colors tab allows you to select a default color scheme that is used for text, selected text, background, and selection colors. It is used to choose a custom color for different categories in the list.

The default colors match the standard desktop theme of GNOME, which has been selected for the desktop. These colors change to match the desktop theme when you select a different color.

Syntax Highlighting

The Syntax Highlighting tab gives options for the configuration of how gedit highlights different elements in programming mode. The gedit editor has the power to detect what programming language exists in a text file. It automatically set up the appropriate syntax highlighting for that text. Also, it allows you to customize the highlighting feature by selecting your favorite colors to highlight different elements of the code. The elements change based on the programming code type that you have selected. For the shell scripts, you may select the sh highlighting mode, which contains certain color schemes.

Plugins

The Plugins tab offers control over the plugins that are used in gedit. Plugins are individual programs that tend to interface with gedit and provide added functionality. Many plugins are available for gedit. However, not all of them are installed in the system by default. The plugins that are enabled show a checkmark in a checkbox that is next to their names. Some plugins like External Tools plugin provide some additional configuration features after you have selected them. It allows you to set up a shortcut key to start the terminal where gedit displays the output.

- You can use the plugin Change Case to change the case of a piece of selected text.
- You can use the plugin Document Statistics to report the number of words, characters, lines, and non-space characters.

- You can use the plugin Insert Date/Time to insert the current time and date in multiple formats at the cursor's present position.
- You can use the plugin External Tools to provide a particular shell environment in the editor to execute scripts and commands.
- You can use the plugin Indent Lines to provide un-indentation and advanced line indentation.
- You can use the plugin File Browser Pane to provide a file browser to ease off the process of file selection and editing.
- You can use the plugin Tag List to provide a particular list of commonly used strings that you can easily enter into the text.
- You can use the plugin Modelines to provide emacs-style messages to the bottom side of your editor's window.
- You can use the plugin Python Console to provide interactive consoles to the bottom side of the editor window to enter commands with the Python programming language's help.
- You can use the plugin Spell Checker to provide proper dictionary spellchecking for a text file.
- You can use the plugin Snippets to store some often-used pieces of text for retrieval in the text.
- You can use the plugin Sort to sort out an entire file or a piece of selected text.

The emacs Editor

Emacs is a popular editor that existed before Unix did, and developers loved it because it could be ported into Linux. It began life as a console editor, similar to vi, but soon migrated to the graphical environment. It provides access to the original console editor and uses a graphical X windows window for text editing in graphical environments.

When the emacs editor is started from the command line, it determines whether an active X Window session is present and starts in graphical mode. The console mode version uses multiple commands that you need to learn to edit a program. Certain key combinations are required using the CTRL and Meta keys on the keyboard. Typically, the Meta key is mapped to a PC's ALT key, although this may not be the case on all systems. Abbreviations are used – CTRL is abbreviated as C- and Meta as M-.

Basic Commands of emacs Editor

To edit an emacs file from the command line, the following command is required:

$ emacs myfile.c

When the emacs console window is opened, you see a short introduction and a help screen. When a key is hit, a file is loaded in the buffer, and the text is displayed. You see a standard menu bar at the top of the window, but this cannot be used while in console mode, only in graphical mode. Unlike the vim editor, the emacs editor has a single mode, where you move into and out of insert mode to flick between inserting commands and text. When you type printable characters, emacs will insert them at the cursor's current position, and when a command is typed, it is executed. To move the cursor around the buffer, the arrow keys and the PageUp and PageDown keys are required. Other commands needed to move the cursor include:

- You can use the C-p command to move up the previous line in the text.
- You can use the C-n command to move down the next line in the text.
- You can use the C-b command to move back or to the left side by one character.
- You can use the C-f command to move forward or to the right side by one character.

If you are looking forward to making longer jumps, you can use the following commands.

- You can use the M-f command to move to the right side onto the next word.
- You can use the M-b command to move to the left side, back to the previous word.
- You can use the M-a command to move to the start of the present sentence.
- You can use the M-e command to move to the ending point of the present sentence.
- You can use the C-e command to move to the ending point of the present sentence.
- You can use the C-a command to move to the starting point of the present sentence.
- You can use the M-> command to move to the ending line of the text.
- You can use the M-< command to move to the starting line of the text.
- You can use the M-v command to move back by a single screen of the data.
- You can use the C-v command to move forward by a single data screen.

You can save the editor buffer as a file and exit the emacs editor using the following commands:

- You can use the C = x C = s command to save what you have written in the current buffer into a file.
- You can use the C = x C = c command to exit the emacs editor and stop the operations of the editor.
- You can use the C = z command to exit the emacs editor. However, the command does not keep the session running, so

that you may come back to that.

Editing Data

The emacs editor is powerful in terms of inserting and deleting text. Inserting text simply requires that you move your cursor to where you want the text inserted and type the text. The backspace key is used to delete the character immediately before the cursor's position to delete text. The delete key is also used to delete the character at the cursor's position.

The editor also offers options for killing text; the difference between deleting and killing text is that when text is killed, it is placed into a temporary area and can be retrieved at any time, whereas deleted text is permanently removed. Emacs offers the following commands to kill text:

- You can use the M-Backspace command to kill a word before the present position of the cursor.
- You can use the M-k command to kill everything from the cursor's present position to the ending point of the sentence.
- You can use the M-D command to kill a word after the present position of the cursor.
- You can use the C-k command to kill everything from the cursor's present position to the ending point of the line.

You can also kill text on a large scale by moving the cursor to the beginning of the section you want to be killed, pressing C-spacebar or C-@, and placing the cursor at the end of the section. Then press C-w, and all the indicated text is killed.

Popular Linux Commands

There are far too many Linux commands to discuss in any detail but we have covered some of the most common. If you want to know what any Linux command does, simply access their manual page using the syntax below and the name of the command:

$ man command-name

The table below shows you many more commands, with their syntax and a description of what they do:

Command	Syntax	Description
adduser/addgroup	$ sudo adduser $ sudo addgroup	These two commands add users or groups respectively to the system
agetty	$ agetty -L 9600 ttyS1 vt100	A program managing virtual and physical terminals. Invoked by init, it opens a tty port when a connection is made
alias	$ alias home='cd	Creates a shortcut or alias to a specific Linux command on the user's system
apropos	$ apropos adduser	Searches and displays a command or program's man page description
apt	$ sudo apt update	High-level package manager for Ubuntu and Debian systems
apt-get	$ sudo apt-get update	Used to install, remove and upgrade software packages and upgrade the operating system
aptitude	$ sudo aptitude update	Text-based GNU/Linux package

		manager system used for installing or removing packages
arch	$ arch	Displays the machine hardware or architecture name
arp	$ sudo arp-scan --	ARP maps IP network addresses to MAC addresses – this command all live hosts on a specified network
at	$ sudo echo	Schedules tasks to be executed at a future specified time
atq	$ atq	Used for viewing jobs queued in the at command
atrm	$ atrm (job number)	Used for deleting jobs from the at command queue by their job number
awk	$ awk '	Program created for processing text and as a reporting and data extraction tool
basename	$ basename bin/findhosts.sh	Prints a file name from stripped of directories in the absolute path
bc	$ echo 20.05 + 15.00 | bc	Powerful, recuses CLI calculator language – see

		example under syntax
bzip2	$ bzip2 -z filename #Compress $ bzip2 -d filename.bz2 #Decompress	Compresses or decompresses specified files
cal	$ cal	Prints a calendar
cat	$ cat file.txt	View file contents, concatenate files or data and display it
chgrp	$ chgrp (new) (old)	Changes a file's group ownership. New name is provided first with the existing one second
chmod	$ chmod	Changes or updates access permissions for specified files
chown	$ chown	Changes or updates group and user ownership of directories or files
cksum	$ cksum	Displays an input file's byte count and CRC checksum
clear	$ clear	Clears terminal screen
cmp	$ cmp file1 file2	Compares two files byte-by-byte
comm	$ comm file1 file2	Compares sorted

		files on a line-by-line basis
cp	$ cp	Copies directories and files from place to another – locations must be specified
date	$ date	Displays and sets system time and date
df	$ df -h	Displays the disk space usage on a specified file system
diff	$ diff file1 file2	Compares specified files line-by-line and can also find and locate differences between directories
dir	$ dir	Similar to ls command, it lists directory contents
dmidecode	$ sudo dmidecode	Retrieves information about a Linux system's hardware
du	$ du	Shows how files use disk space in a directory and related sub-directories
echo	$ echo	Takes a specified line of text and prints it
eject	$ eject	Ejects DVD, CD

		ROM, and other removeable media
env	$ env	Lists and sets current environment variables
exit	$ exit	Exits the shell
expr	$ expr	Calculates a specified expression
factor	$ factor	Shows a specified number’s prime factors
find	$ find	Searches a directory and sub-directories for specified files using attributes like users, permissions, data, file type size, etc.
free	$ free	Shows the systems memory use. Add -h to the command to show the info in human readable format
grep	$ grep	Searches files for specified patterns and displays those lines with the pattern
groups	$ groups	Displays the groups a specified user belongs to
gzip	$ gzip	Compresses a specified file and

		replaces it with a file with a .gz extension
gunzip	$ gunzip	Expands/restores files compressed using gzip
head	$ (file or stdin) | head	Displays the first 10 lines of the file or stdin specified
history	$ history	Shows commands or retrieves info about commands a user has already executed
hostname	$ hostname	Prints or sets a Linux system hostname
hostnamectl	$ hostnamectl	Takes control of the system hostname in system and modifies or prints the hostname and related settings
hwclock	$ sudo hwclock	Manages the hardware clock and reads or sets it
hwinfo	$ hwinfo	Looks into a system to see what the hardware is
id	$ id	Displays group and user information about current or specified username
ifconfig	$ ifconfig	Configures, views

		and controls the network interfaces for specified Linux systems
ionice	$ ionice	Sets/views process I/O scheduling class and specified process priority
iostat	$ iostat	Shows input/output and CPU stats for partitions and devices and produces reports on how to update configurations for input/output balancing
ip	$ sudo ip	Displays/manages devices, routing, tunnels and policy routing
iptables	$ sudo iptables	Manages incoming/outgoing traffic using configurable table rules
iw	$ iw list	Manages wireless devices and configurations
iwlist	$ iwlist	Shows detailed info from specified wireless interfaces
kill	$ kill	Uses a specified process's PID to kill

		it
killall	$ killall (process name)	Kills a specified process using its name
kmod	$ kmod	Manages kernel modules
last	$ last	Shows the last users logged in
ln	$ ln -s	Uses -s flag to create soft links between files
locate	$ locate (file name)	Finds a file by its specified name
login	$ sudo login	Creates a new system session
ls	$ ls	Lists a directory’s contents and using the -1 flag will create a long listing
ishw	$ sudo ishw	Provides basic details on the machine’s hardware configuration – use superuser privileges to get more detailed info
iscpu	$ iscpu	Displays info about the system architecture
lsof	$ lsof	Shows details of files a process has opened
lsusb	$ lsusb	Shows information

		about system USB buses and any connected devices
man	$ man (command)	Shows specified command or program manual pages
mkdir	$mkdir	Creates one or more directories
more	$ more (file name)	Displays long files one screen at a time
mv	$ mv	Renames directories or files or moves them to specified locations in the structure
nc/netcat	$ nc/netcat	Performs UDP, TCP or UNIX-domain socket operations
netstat	$ netstat	Shows useful info about the networking subsystem
nice	$ nice	Shows or changes nice value of a specified running program. Adjusted niceness must be specified otherwise current niceness is displayed
nmap	$ nmap	Open-source network scanner
nproc	$ nproc	Displays available

		processing unit number for current processes
passwd	$ passwd	Creates or updates passwords for specified accounts and can also change validity period.
pidof	$ pidof	Shows a running command or program's process ID
ping	$ ping	Determines connectivity between network or internet hosts
ps	$ ps	Displays info about active running system processes
pstree	$ pstree	Shows running processes in tree format, rooted at init or PID
pwd	$ pwd	Shows the current or working directory's name
reboot	$ reboot	Used to stop, reboot or turn off a system
rename	$ rename	Can rename multiple files at a time, for example renaming all files with .html extension to .php extension

rm	$ rm (file/directory name)	Removes specified directories or files
rmdir	$ rmdir (directory name)	Deletes or removes empty directories
scp	$ scp	Allows files to be securely copied between network hosts
shutdown	$ shutdown	Schedules a time to stop, power off or reboot the system/machine
sleep	$ sleep	Delays or pauses a command execution for a specified time period
sort	$ sort (file name)	Sorts text lines in a specified file or stdin
split	$ split	Splits specified files into smaller bits
ssh	$ ssh	Accesses and runs commands on specified remote machines using an encrypted and secure communication over insecure networks
stat	$ stat	Shows the status of a specified file or file system
su	$ su	Switches from one user ID to another or

		to root in a login session. If no username is specified, the default is root
sudo	$ sudo	Allows system user to run commands as another user or root
sum	$ sum	Displays block counts and checksum for all specified files
tac	$ tac	Concatenates specified files and displays them in reverse
tail	$ tail	Displays last 10 lines of specified files to standard output
talk	$ talk person (login name) $ talk ‘user@host’	Talks to other network or system users. Login name is used to talk to a person on the same machine and user@host to talk to a user on a different machine
tar	$ tar	Powerful file archiving utility
tree	$ tree	Cross-platform command-line program for

		recursively displaying or listing directory contents in tree format
time	$ time	Runs a program and provides a summary of the system resource used
top	$ top	Shows all Linux system process CPU and memory usage
touch	$ touch	Changes the timestamp of a file or creates files
uname	$ uname	Shows system info, such as version, release date, hostname kernel name, operating system and so on. The -a flag is used to display all information
uptime	$ uptime	Displays length of system running time, how many users are logged on and the load averages
users	$ users	Displays names of current users
vim/vi	$ vim file	Text editor used for editing program and text files

w	$ w	Shows load averages, uptime and info about current users, along with their processes, etc.
wall	$ wall (message)	Sends a specified message to all system users
watch	$ watch	Repeatedly runs a specified program while showing the program output. Can also watch file or directory changes
wc	$ wc (filename)	Displays the word, newline and byte counts for specified files and totals for multiple files
wget	$ wget	Downloads web files non interactively
whatis	$ whatis (command)	Displays short man page descriptions of specified commands
which	$ which	Shows absolute paths for specified files that may be executed in the current environment
who	$ who	Shows info about current logged-in users
whereis	$ whereis	Shows manual,

	(command)	source and binary files for specified commands
yes	$ yes (string)	Repeatedly displays a specified string until killed or terminated
zip	$ zip	Packages and compresses archive files

Chapter Eight: Linux Shell Scripting

The command-line tools are good for solving computing problems, but they do not make you the master artist in the world of Linux. The shell makes you the master of Linux. You should know how the shell works and how you can write different types of scripts in a Linux shell. By learning the command line and Linux shell, you will be able to carry out a number of tasks by itself.

A shell script, in the simplest terms, is a file that contains a series of commands. The shell will read the file and carry out the commands as though they have been entered on the command line. The shell emerges out as distinctive in that it is a robust command-line interface system as well as a scripting language interpreter. Most of the things that you can do on the command line can also be done inside shell scripts. Most of the things that you can do in shell scripts can also be done on the command line in Linux.

We now have covered most of the shell scripting features; however, we have focused on the ones that are more often used on the command line. The shell gives us many features when we are writing programs.

Writing the Shell Script

Shell scripts are just like ordinary text files which is why you need a text editor to write the scripts. The best text editors will provide you syntax highlighting, which allows you to see a color-coded view of different script elements. Syntax highlighting will help you spot different kinds of common errors and use kate or vim to write shell scripts. The system is a bit fussy about blocking old text files to run as a program. All this happens for a good reason. You need to set up permissions of the script file to allow the execution. You should also put the script in a place where you can easily find it because the shell searches the file and executes it.

The Format of Linux Shell Scripts

I will now enter the writing phase and show you how you can write a shell script. The language of the script is simple. See the script as under:

[gha@localhost ~]$ echo 'I am learning Linux'

I am learning Linux

This is shell scripting. Now I will add a comment to the same script.

[gha@localhost ~]$ echo 'I am learning Linux' # I have added a comment to the code.

I am learning Linux

Now you have to make the script executable. See the following:

[gha@localhost ~]$ ls -1 Linux_learning

[gha@localhost ~]$ chmod 755 Linux_learning

The text Linux_learning is the name of the file in which I had saved the script. The chmod command is used to make the script executable in Linux.

Displaying Text

Most of the shell commands tend to produce a specific output, which is displayed on your console monitor where you are running the script. You may need to add text messages to help out the script user know what is going to happen inside the script. You can do this by using the echo command I have talked about in the past section. The echo command displays simple text strings. There are different techniques to write a script with the echo command. I will explore all of them.

[gha@localhost ~]$ echo I am learning Linux shell scripting

I am learning Linux shell scripting

The most remarkable thing is that you do not have to use quotes to enclose the string text, as is the case with other programming languages. However, sometimes it becomes necessary to use quotes. Either you need single or double quotes to display the text strings.

[gha@localhost ~]$ echo ‘I am learning Linux shell scripting.’

I am learning Linux shell scripting

In the next example, using quotes is going to be inevitable.

[gha@localhost ~]$ echo 'Adam says, "I am learning Linux shell scripting."'

Adam says, “I am learning Linux shell scripting."

You can see that when you have to add a text that is in the form of direct speech, you will have to add quotation marks to the text.

The if-then Statement

Many Linux shell scripting programs require some logic flow control between different commands inside the script, which means that the shell tends to execute different commands. In addition, it keeps the ability to execute different other commands that permit the script to loop through the commands based on the result of different other commands. We refer to them as structured commands. They are also known as conditionals.

Structured commands allow you to shift the flow of the operations of a program. Some commands are executed, while others are skipped

when they are caught up in certain conditions. In other programming languages, the object after an if statement is the equation that the system evaluates for True or False grounds. The bash shell if statement does not work that way. Instead, it runs the command that is defined on the if line. If this command's exit status is considered zero, the commands that are listed under the then section tend to execute. If this command's exit status is something else, the then commands are put on hold, and the bash shell jumps over to the next command in the script.

```
$ cat testing5
#!/bin/bash
# I am now testing the if statement for bash shell
if date
then
echo "The structured command has completely worked"
fi
```

```
$bash -f main.sh
Tue Jan 26 05:02:54 UTC 2021
The structured command has completely worked
```

The script has used the date command on the if line. If this command is executed successfully, the echo statement must display the text string. When you are running the script from the command line, you will be getting the same results as above. The shell will execute the date command. Since the exit status was zero, it will also execute the echo statement that is listed in the then section. In the next example, I will test a bad command to see how it works.

```
$ cat testing2
#!/bin/bash
# It is time to run a test on a bad command
if hgjdksllh
then
echo "it is not going to work"
fi
echo "I cannot run the command because you're out of the if statement."
```

```
$bash -f main.sh
I cannot run the command because you're out of the if statement
```

Since the above command was a bad command, it produced an exit status that is at the moment a non-zero. The bash shell will skip the echo statement in the then section.

The if-then-else Statement

The if-then statement provides one option to determine whether the command is successful. If a non-zero exit status code is returned, the bash shell goes to the next script command. In a situation like this, it would best if we had another set of commands we could execute, and that is where the if-then-else statement comes in.

If the if-statement command produces the exit status zero code, the commands in the then part of the statement are executed. If a non-zero exit status code is returned, the commands in the else part of the statement are executed.

```
$ cat testing4
#!/bin/bash
# I am now going to test the else section
testinguser=thisisabadtest
if grep $testinguser /etc/passwd
then
echo The files for user $testinguser are:
ls -a /home/$testinguser/.b*
else
echo "The user name $testinguser cannot be found on the system"
fi
$ ./test4
The user name thisisabadtest cannot be found on the system
$
```

```
$bash -f main.sh
The user name thisisabadtest cannot be found on the system
```

On occasion, you may need to check things in the script code. Rather than separate if-then statements, a different version of the else statement can be used. This is called the elif, and this continues the else statement with a subsequent if-then statement. It provides a different evaluation command, which is much like the first if statement. If a zero exit status code is returned, the commands in the second then statement are executed by the bash shell.

Advanced if-then Features

There is a double parentheses command in Linux that allows you to incorporate some advanced mathematical formulas when you are making comparisons. The test command permits simple operations in comparison. However, the double parentheses command offers more mathematical symbols that a number of programmers from different other languages are using.

The expression term may be any kind of mathematical assignment or expression. Besides the standard operators that the test command uses, additional operators are available for use in the double parentheses command.

You also can use the double bracket command for advanced features to do string comparisons. The double bracket command format uses the standard string comparison that we have used in the test command. Where it differs is by the option of pattern matching. You will be able to define a regular expression that you need to match against the string value.

```
$ cat testing2
#!/bin/bash
# I will now be using pattern matching
if [[ $MyUSER == r* ]]
then
echo "Hello $MyUSER I know you from the days when you used to work in a restaurant in Europe."
else
echo "Sorry, I have never seen you before. I don’t know you."
fi
```

$bash -f main.sh

Sorry, I have never seen you before. I don’t know you.

More often, you may find yourself attempting to evaluate the value of some variables in a bid to find out a specific value inside a set of possible values. In this typical scenario, you may end up writing a lengthy if-then-else statement such as the following:

```
$ cat testing555
#!/bin/bash
# I am now looking for some possible values
if [ $MyUSER = "rich" ]
then
echo "Welcome $MyUSER to the heavens on the earth."
echo "Please enjoy your visit in plush green fields and lush green mountains. I wonder if you want to take a bath in the river beneath the valley."
elif [ $MyUSER = Jason ]
then
echo "Welcome $MyUSER to the lakes of Madora."
echo "Please enjoy your stay in the shining waters of Madora lakes. I hope you enjoy the resorts, the sun, the breeze, the food, and the drinks."
elif [ $MyUSER = Simra ]
then
echo "Welcome $MyUSER to the lakes of Madora."
echo "Please enjoy your stay in the shining waters of Madora lakes. I hope you enjoy the resorts, the sun, the breeze, the food, and the drinks."
elif [ $MyUSER = Linda ]
```

then

echo "Don’t forget to logout when you’re done"

else

echo "Sorry, you’re not allowed here"

fi

The for Command

Iterations through a series of commands is normal practice when it comes to programming. Many a time, you may have to repeat different commands. Often you need to repeat a bunch of commands until the program meets a specified condition like processing files inside a directory or in all the lines inside a text file.

The bash shell offers the for command to develop a loop that iterates through a set of values. Each iteration tends to perform a set of commands by using a single value inside the series. One of the most basic uses of the for command is iteration through a list of values that are defined inside the for command itself.

v=$ cat testing111

#!/bin/bash

This is the basic for command in Linux

for mytest in Alabama Texas Alaska Virginia Arizona Ohio Arkansas South Dakota California North Dakota Colorado New Jersey Florida Michigan Georgia

do

echo I am planning to continue my travel circle across the United States in the summer. The next state I will visit is $mytest

done

$bash -f main.sh

I am planning to continue my travel circle across the United States in the summer. The next state I will visit is Alabama

I am planning to continue my travel circle across the United States in the summer. The next state I will visit is Texas

And so on until the last line:

I am planning to continue my travel circle across the United States in the summer. The next state I will visit is Georgia

Each time the for command iterates through the list of objects, it assigns the mytest variable to the next item in the list. The $mytest variable is used like other script variables in the for command statements. After the final iteration, the $test variable remains valid throughout the shell script's remaining part. It will retain the final iteration value unless you change it.

v=$ cat testing111

#!/bin/bash

This is the basic for command in Linux

for mytest in Alabama Texas Alaska Virginia Arizona Ohio Arkansas South Dakota California North Dakota Colorado New Jersey Florida Michigan Georgia

do

echo I am planning to continue my travel circle across the United States in the summer. The next state I will visit is $mytest

done

echo "The last US state I traveled to was $mytest"

mytest=Connecticut

echo "Now I am going to visit $mytest"

$bash -f main.sh

I am planning to continue my travel circle across the United States in the summer. The next state I will visit is Alabama

I am planning to continue my travel circle across the United States in the summer. The next state I will visit is Texas

And so on, until the last line:

I am planning to continue my travel circle across the United States in the summer. The next state I will visit is Georgia

The last US state I traveled to was Georgia

Now I am going to visit Connecticut

You can see that the $mytest variable retained its original value and permitted us to change it outside the for loop. Let's see another for loop example:

v=$ cat testing111

#!/bin/bash

This is the basic for command in Linux

for mytest in pumpkin potato tomato spinach ginger garlic cauliflower cabbage pepper beet ladyfinger broccoli

do

echo I am planning to make a shift to the cooking schedule as the summer sets in. I have bought lots of vegetables. However, today I plan to cook $mytest.

done

echo "The last vegetable I cooked was $mytest"

mytest=onion

echo "Now I am going to cook $mytest"

$bash -f main.sh

I am planning to make a shift to the cooking schedule as the summer sets in. I have bought lots of vegetables. However, today I plan to cook pumpkin.

I am planning to make a shift to the cooking schedule as the summer sets in. I have bought lots of vegetables. However, today I plan to cook potato.

This continues through each of the vegetabes, ending with:

I am planning to make a shift to the cooking schedule as the summer sets in. I have bought lots of vegetables. However, today I plan to cook broccoli.

The last vegetable I cooked was broccoli

Now I am going to cook onion

When you are dealing with the for command, the biggest problem you may encounter is using multi-word values. The for loop assumes that each value ought to be separated by a space. If you have data values to contain spaces, you may run into another problem. You may see that in the example of US states. The bash editor considered South and Dakota as two separate values.

The for command offers us the quotation marks to separate values that have more than one word. You may use double quotes to separate different values in the for command. See the following example.

v=$ cat testing111

#!/bin/bash

```
# This is the basic for command in Linux

for mytest in Alabama Texas "New York" Alaska Virginia Arizona Ohio Arkansas "South Dakota" California "North Dakota" Colorado "New Jersey" Florida Michigan Georgia "North Carolina" "South Carolina" "New Mexico" "Rhode Island" "West Virginia" "New Hampshire" Vermont

do

echo I am planning to continue my travel circle across the United States in the summer. The next state I will visit is $mytest

done

echo "The last US state I traveled to was $mytest"

mytest=Connecticut

echo "Now I am going to visit $mytest"
```

$bash -f main.sh

```
I am planning to continue my travel circle across the United States in the summer. The next state I will visit is Alabama

I am planning to continue my travel circle across the United States in the summer. The next state I will visit is Texas

Continuing through the States until we get to the last one:

I am planning to continue my travel circle across the United States in the summer. The next state I will visit is Vermont

The last US state I traveled to was Vermont

Now I am going to visit Connecticut
```

Now the for command can distinguish between the single word and multi-word values. Also, the best thing is that when you are using double quotation marks, the shell sheds the quotation marks as a part of the value.

Reading a List through a Variable

Often what happens inside a shell script is that you accumulate lists of values that are stored inside a variable. Then it needs to iterate through a list. You may do this with the help of the for command.

```
$ cat testing111

#!/bin/bash

# This is the basic for command in Linux

list="Alabama Texas New York Alaska Virginia Arizona Ohio Arkansas South Dakota California North Dakota Colorado New Jersey Florida Michigan Georgia North Carolina South Carolina New Mexico Rhode Island West Virginia New Hampshire Vermont"

list=$list" Idaho"


for state in $list

do

echo "I am planning to continue my travel circle across the United States in the summer. The next state I will visit is $state"

done
```

$bash -f main.sh

I am planning to continue my travel circle across the United States in the summer. The next state I will visit is Alabama

I am planning to continue my travel circle across the United States in the summer. The next state I will visit is Texas

Right up to the last state:

I am planning to continue my travel circle across the United States in the summer. The next state I will visit is Idaho

The $list variable contains the list of values that have been used for iterations. You should take note that the code uses another assignment statement for the concatenation of items to the existing list. Even when you forget to add items in the list, you can add them up by the concatenation method later on. I will add five more items to the list in the next example.

```
$ cat testing111
#!/bin/bash
# This is the basic for command in Linux
list="Alabama Texas Alaska Virginia Arizona Ohio Arkansas California Colorado Florida Michigan Georgia Vermont Iowa Nebraska Arkansas Kansas Kentucky Tennessee Utah Maine Missouri Minnesota"
list=$list" Idaho"
list=$list" Oregon"
list=$list" Montana"
list=$list" Delaware"

for state in $list
do
echo "I am planning to continue my travel circle across the United States in the summer. The next state I will visit is $state"
done
```

$bash -f main.sh

I am planning to continue my travel circle across the United States in the summer. The next state I will visit is Alabama

I am planning to continue my travel circle across the United States in the summer. The next state I will visit is Texas

This output continues to list the states, right up to the last one:

I am planning to continue my travel circle across the United States in the summer. The next state I will visit is Delaware

There is a way to generate values for custom usage in a list to use the output of a particular command.

```
$ cat testing111
#!/bin/bash
# I will read values from files
myfile="states"

for state in `cat $myfile`
do
echo "I am planning to continue my travel circle across the United States in the summer. The next state I will visit is $state"
done
$cat states
Alabama
Texas
Alaska
Virginia
Arizona
```

Ohio

Arkansas

California

Colorado

Florida

Michigan

Georgia

Vermont

Iowa

Nebraska

Arkansas

Kansas

Kentucky

Tennessee

Utah

Maine

Missouri

Minnesota

Idaho

Oregon

Montana

Delaware

The while Command

The while command is a cross between the if-then statement and Linux for loop. It allows you to define a command to test a condition and then looping through a set of commands until you reach a zero

exit status. It will analyze the test command at the start of each iteration. Upon reaching the non-zero exit status, the while command stops the execution of the set of commands.

```
$ cat test10
#!/bin/bash
# This is a while command test in Linux
var15=20
while [ $var15 -gt 0 ]
do
echo This is $var15 in the while loop.
var15=$[ $var15 - 1 ]
done
```

```
$bash -f main.sh
This is 20 in the while loop.
This is 19 in the while loop.
This is 18 in the while loop.
The loop continues to the final line:
This is 1 in the while loop.
```

Multiple Test Commands

You can pack up multiple test commands in the while loop.

```
$ cat test1133
#!/bin/bash
# Now I am testing a multicommand while loop in Linux shell
```

```
var15=20
while echo $var15
[ $var15 -ge 0 ]
do
echo "This operation is being conducted inside the while loop."
var15=$[ $var15 - 1 ]
done
```

$bash -f main.sh

```
20
This operation is being conducted inside the while loop.
19
This operation is being conducted inside the while loop.
18
This operation is being conducted inside the while loop.
17
This will continue until it reaches the last number:
This operation is being conducted inside the while loop.
0
This operation is being conducted inside the while loop.
-1
```

The first test command displays the present value of var15 variable. The second command will use the test command to know the value of var15 variable. The echo statement returns a simple message within the loop that indicates that the loop has runs its course.

The until Command

The until command works opposite to the while command. It requires a simple test command is specified, which will produce a non-zero exit status, at which point the bash shell will execute the while loop commands.

However, similar to the while command, the until command statement can also contain multiple test commands, and the exit status of the final command is the determining factor.

```
$ cat test1222
#!/bin/bash
# In this code block I will be using the until command
var15=100
until [ $var15 -eq 0 ]
do
echo You are seeing the operations of the until command. The next digit in the loop is $var15
var15=$[ $var15 - 5 ]
done
```

$bash -f main.sh

You are seeing the operations of the until command. The next digit in the loop is 100

You are seeing the operations of the until command. The next digit in the loop is 95

You are seeing the operations of the until command. The next digit in the loop is 90

And continuing, until it reaches the final digit:

You are seeing the operations of the until command. The next digit in the loop is 5

The example has successfully tested the var15 variable in order to determine when the loop will stop - when the value of the variable drops down to zero, the until command stops right away.

```
$ cat test1222
#!/bin/bash
# In this code block, I will be using the until command
var15=100

until echo $var15
      [ $var15 -eq 0 ]
do
echo You are seeing the operations of the until command. The next digit in the loop is $var15
var15=$[ $var15 - 5 ]
done
```

$bash -f main.sh

100

You are seeing the operations of the until command. The next digit in the loop is 100

95

You are seeing the operations of the until command. The next digit in the loop is 95

90

Again, this will continue to the last digit in the loop:

You are seeing the operations of the until command. The next digit in the loop is 5

0

Nesting Loops

A loop statement may use another command inside of the loop, including other commands of the same loop. The process is known as nested looping. You will have an iteration inside another iteration, which will multiply the number of times a command is being run. See an example to get a grasp of how you can nest loops.

```
$ cat test1444
#!/bin/bash
# The following code block shows nesting for loops
for (( x = 1; x <= 7; x++ ))
do
    echo "You are seeing the operations of the until command. The next digit in the loop is $x:"
   for (( y = 1; y <= 7; y++ ))
   do
      echo " Now you are seeing what is going on inside loop: $y"
   done
done
```

$bash -f main.sh

You are seeing the operations of the until command. The next digit in the loop is 1:

Now you are seeing what is going on inside loop: 1

Now you are seeing what is going on inside loop: 2

And so on, finishing with

Now you are seeing what is going on inside loop: 7

You are seeing the operations of the until command. The next digit in the loop is 2:

Now you are seeing what is going on inside loop: 1

Now you are seeing what is going on inside loop: 2

Again, this will continue until it ends with

Now you are seeing what is going on inside loop: 7

You are seeing the operations of the until command. The next digit in the loop is 3:

Now you are seeing what is going on inside loop: 1

Now you are seeing what is going on inside loop: 2

Ending with:

Now you are seeing what is going on inside loop: 7

You are seeing the operations of the until command. The next digit in the loop is 4:

Now you are seeing what is going on inside loop: 1

Now you are seeing what is going on inside loop: 2

Ending with:

Now you are seeing what is going on inside loop: 7

The entire cycle continues through the loop, until it reaches:

Now you are seeing what is going on inside loop: 7

You can experiment with the numbers in the loop to contract or expand the two loops' sizes. By changing the numbers of the loops, you will be able to change how the loop will behave.

```
$ cat test1444
#!/bin/bash
# The following code block shows nesting for loops
for (( x = 1; x <= 3; x++ ))
do
    echo "You are seeing the operations of the until command. The next digit in the loop is $x:"
   for (( y = 1; y <= 5; y++ ))
   do
      echo " Now you are seeing what is going on inside loop: $y"
   done
done
```

$bash -f main.sh

```
You are seeing the operations of the until command. The next digit in the loop is 1:
Now you are seeing what is going on inside loop: 1
 Repeated to:
 Now you are seeing what is going on inside loop: 5
You are seeing the operations of the until command. The next digit in the loop is 2:
 Now you are seeing what is going on inside loop: 1
 Repeated to:
```

Now you are seeing what is going on inside loop: 5

You are seeing the operations of the until command. The next digit in the loop is 3:

Now you are seeing what is going on inside loop: 1

And finishing on:

Now you are seeing what is going on inside loop: 5

The nested loop iterates through the values for iterations of the outer loop. There is no difference between the do and done commands for two loops. You can create a nested loop by pairing up for loops and while loops.

```
$ cat test15
#!/bin/bash
# This is how we can place a for loop in a while loop
var15=15
while [ $var15 -ge 0 ]
do
     echo "You are seeing the operations of nesting loops. The next digit in the loop is $var15:"
   for (( var12 = 1; $var12 < 3; var12++ ))
   do
      var13=$[ $var15 * $var12 ]
     echo " Now you are seeing what is going on inside loop: $var15 * $var12 = $var13"
   done
   var15=$[ $var15 - 1 ]
```

done

$bash -f main.sh

You are seeing the operations of nesting loops. The next digit in the loop is 15:

Now you are seeing what is going on inside loop: 15 * 1 = 15

Now you are seeing what is going on inside loop: 15 * 2 = 30

This will continue down to the final line:

You are seeing the operations of nesting loops. The next digit in the loop is 0:

Now you are seeing what is going on inside loop: 0 * 1 = 0

Now you are seeing what is going on inside loop: 0 * 2 = 0

In the next example, I will pair up until and while loops to test nesting loops' limits. See the following example.

```
$ cat test16
#!/bin/bash
var15=10
until [ $var15 -eq 0 ]
do
echo "You are seeing the operations of the nested until and while loops: $var15"
var12=1
while [ $var12 -lt 5 ]
do
var13=`echo "scale=4; $var15 / $var12" | bc`
```

```
echo " This is how the inner section of the nested loop functions: $var15 / $var12 = $var13"
var12=$[ $var12 + 1 ]
done
var15=$[ $var15 - 1 ]
done
```

$bash -f main.sh

```
You are seeing the operations of the nested until and while loops: 10
This is how the inner section of the nested loop functions: 10 / 1 = 10.0000
This is how the inner section of the nested loop functions: 10 / 2 = 5.0000
This is how the inner section of the nested loop functions: 10 / 3 = 3.3333
This is how the inner section of the nested loop functions: 10 / 4 = 2.5000
This cycle is repeated, until it reaches the final one of:
You are seeing the operations of the nested until and while loops: 1
This is how the inner section of the nested loop functions: 1 / 1 = 1.0000
This is how the inner section of the nested loop functions: 1 / 2 = .5000
This is how the inner section of the nested loop functions: 1 / 3 = .3333
 This is how the inner section of the nested loop functions: 1 / 4 = .2500
```

Loop Control

You might think that once a loop starts, you will be stuck in that until the loop has runs its course. However, there is a way out. You can add to the code a couple of commands that help you control the inside of the loop.

The break command is the simplest way to move out of a loop that is in progress. You may use the break command to exit while and until loops.

```
$ cat test17
#!/bin/bash
for countries in 2 3 4 5 6 7 8 9 10 11 12 13 14
do
if [ $countries -eq 12 ]
then
break
fi
echo "I am going to visit $countries after Covid-19 is over."
done
echo "I have visited all the countries."
```

$bash -f main.sh

```
I am going to visit 2 countries after Covid-19 is over.
I am going to visit 3 countries after Covid-19 is over.
I am going to visit 4 countries after Covid-19 is over.
I am going to visit 5 countries after Covid-19 is over.
I am going to visit 6 countries after Covid-19 is over.
```

I am going to visit 7 countries after Covid-19 is over.

I am going to visit 8 countries after Covid-19 is over.

I am going to visit 9 countries after Covid-19 is over.

I am going to visit 10 countries after Covid-19 is over.

I am going to visit 11 countries after Covid-19 is over.

I have visited all the countries.

The for loop has run its course until the use of the break keyword. When the if-then condition was satisfied, the bash shell executed the break command, which put a stopper to the for loop. This technique works equally well for until and while loops.

```
$ cat test18
#!/bin/bash
# breaking out of a while loop
countries=1
while [ $countries -lt 15 ]
do
if [ $countries -eq 8 ]
then
break
fi
echo "I am going to visit $countries after Covid-19 is over."
countries=$[ $countries + 1 ]
done
echo "I have visited all the countries."
```

$bash -f main.sh

I am going to visit 1 countries after Covid-19 is over.

I am going to visit 2 countries after Covid-19 is over.

All the way down to:

I am going to visit 7 countries after Covid-19 is over.

I have visited all the countries.

In the next example, I will use the break statement in the for loop.

```
$ cat test1444
#!/bin/bash
# The following code block shows nesting for loops
for (( x = 1; x <= 4; x++ ))
do
    echo "You are seeing the operations of the until command. The next digit in the loop is $x:"
   for (( y = 1; y <= 7; y++ ))
   do
   if [ $y -eq 5 ]
   then
      break
   fi
      echo " Now you are seeing what is going on inside loop: $y"
   done
done
```

$bash -f main.sh

You are seeing the operations of the until command. The next digit in the loop is 1:

Now you are seeing what is going on inside loop: 1

Now you are seeing what is going on inside loop: 2

Now you are seeing what is going on inside loop: 3

Now you are seeing what is going on inside loop: 4

This cycle is repeated for four digits and ends with:

You are seeing the operations of the until command. The next digit in the loop is 4:

Now you are seeing what is going on inside loop: 1

Now you are seeing what is going on inside loop: 2

Now you are seeing what is going on inside loop: 3

Now you are seeing what is going on inside loop: 4

You can either redirect the output of the a loop in the shell script or pipe it out for use. You will be needing a processing command to add to the end of the done command.

Loops are a integral part of many programming languages, including Linux and there are three types of looping commands in the bash shell that can be used in scripts.

The first is the for command, which lets you iterate through a list of values in the command line, contained in a variable, or obtained through file globbing (used for file extraction.).

The second is the while command, offering a method of looping by using the command's condition and ordinary commands. In this way, different variable conditions can be tested and, so long as a zero exit status is returned, the while loop will continue iterating through a set of specified commands.

The third is the until command, offering a way of iterating through commands, but based on a command or condition that produces a non-zero exit status. Using this feature, you can set a condition that must be met before the end of the iteration. Loops can be paired up in shell scripts by producing different loop layers. The bash shell provides the continue and break commands that we use to change the normal loop flow process on multiple loop values.

More Basic Shell Scripts

Here are some more simple examples of bash shell scripts that you can try for yourself:

The Hello World Program

Most new programmers start learning their chosen language using the help world program. It is one of the most simple programs that does nothing more than prints a simple string to the standard output, reading "Hello World." In Linux, editors like nano or vim can be used to create the hello-world.sh file and then copy and paste the code below into it:

```
#!/bin/bash

echo "Hello World"
```

Save the file and exit.

This file needs to be made executable with the command below:

```
$ chmod a+x hello-world.sh
```

And then, you can run the file using either of these commands:

```
$ bash hello-world.sh

$ ./hello-world.sh
```

The result is the string you passed to echo being printed on the screen.

How to Print With echo

The echo command will print information from bash and is not unlike the printf function in C. Indeed, they share many of the same options, including re-direction and escape sequences.

Create a new file and call it echo.sh. Make the file executable using the directions in the first example:

```
#!/bin/bash

echo "Printing text"

echo -n "Print text without newline"

echo -e "\nRemoving \t special \t characters\n"
```

Run this script to see what it will do. There are two options in the script: -n is used for newline, and -e tells echo that you have passed a string containing some special characters and it needs extra functionality.

How to Use Comments

Comments are one of the most useful parts of any script and are required for higher-quality codes. They tell the coder and others reading the code what it does but should not be long and rambling – keep them short and to the point. Common practice has comments places inside any code that contains critical logic. The # symbol is used to comment a line, as you can see in the example below:

```
#!/bin/bash


# Add two values

((sum=25+45))


#Print result

echo $sum
```

The script will give us a result of 70. Have a look at how comments are used before each line, except for the first one. The first line is a

shebang, and it tells the system the interpreter it should use to run the script.

How to Use Multi-Line Comments

Some coders document their scripts using multi-line comments, and you can see how this is done below:

```
#!/bin/bash
: '
This script will calculate
the square of 5.
'

((area=5*5))
echo $area
```

Multi-line comments do not use the #; instead, they are enclosed in the' and ' characters.

How to Use While Loops

The while loop is used when you want to run an instruction several times. Have a look at the script below to see how this works:

```
#!/bin/bash
i=0

while [ $i -le 2 ]
do
echo Number: $i
((i++))
done
```

The while loop takes the form you can see below:

```
while [ condition ]
```

```
do

commands 1

commands n

done
```

You must include the spaces surrounding the brackets – this is not optional; it is mandatory. Missing it out will result in errors.

How to Use the For Loop

Another popular and much user construct is the for loop. This is used for efficient iteration over a set of code, and you can see a simple example here:

```
#!/bin/bash


for (( counter=1; counter<=10; counter++ ))

do

echo -n "$counter "

done


printf "\n"
```

Create a file and name it for.sh, save this code, and make it executable. Then run it, and you should see the numbers from 1 to 10 printed on your screen.

How to Receive User Input

Implementing user interaction in bash shell scripts is critical and requires that you get user input. The example below shows you how to get user input in a shell program:

```
#!/bin/bash


echo -n "Enter anything:"
```

```
read anything


echo "You Entered: $anything."
```

In this script, a variable name follows the read construct, and this is how we get user input. The variable is used to store the user input and the $ symbol used to access it.

How to Use the If Statement

The if statement is the most common of the conditional constructs using in shell scripting, and they take this format:

```
if CONDITION

then

STATEMENTS

fi
```

The statements will be executed if the CONDITION is true. The fi keyword marks the end of the statement, and you can see how it all works in the example below:

```
#!/bin/bash


echo -n "Enter a number: "

read num


if [[ $num -gt 10 ]]

then

echo "Number is greater than 10."

fi
```

You will only get an output from this if the given input number is more than 10. The -gt option indicates "greater than" while -lt, if used, will

indicate "less than." -le is for "less than equal," and -ge is for "greater than equal." You must include the [[]] – they are not optional.

How to Get More Control Using If Else

You can get more control over the logic in the script by combining an else and if construct. Have a look at the example:

```
#!/bin/bash


read n
if [ $n -lt 10 ];
then
echo "It is a one-digit number."
else
echo "It is a two-digit number."
fi
```

The else section must go after the action bit of the if statement before fi, which closes the statement.

How to Use the AND Operator

We can use the AND operator to check whether several conditions have been satisfied. Every part that the operator separates has to be true; otherwise, the AND statement will only return false. Have a look at the script below to see how the operator works:

```
#!/bin/bash


echo -n "Enter Number:"
read num


if [[ ( $num -lt 10 ) && ( $num%2 -eq 0 ) ]]; then
```

```
echo "Even Number"
else
echo "Odd Number"
fi
```

We use the double-ampersand (&&) to denote the AND operator.

How to Use the OR Operator

Another important construct is the OR operator. Using it helps us to implement robust and complex logic in a shell script. Unlike the AND operator, any statement with an OR operator will return true when only one operand is true. False is returned when the operand on each side of the OR operator is false. Here's an example:

```
#!/bin/bash


echo -n "Enter any number:"
read n


if [[ ( $n -eq 15 || $n -eq 45 ) ]]
then
echo "You won"
else
echo "You lost!"
fi
```

This is a simple example showing the OR operator working in a Linux shell script. The user will only be declared a winner when the number 15 or 45 is input.

We use the || sign to denote the OR operator.

How to Use Elif

The elif statement means "else-if" and is used to implement some chain logic. Have a look at this example to see how it works:

```
#!/bin/bash

echo -n "Enter a number: "
read num

if [[ $num -gt 10 ]]
then
echo "Number is greater than 10."
elif [[ $num -eq 10 ]]
then
echo "Number is equal to 10."
else
echo "Number is less than 10."
fi
```

This is a self-explanatory example, so I won't go into too much detail. You can play around and change the variable names or their values to see how they function.

How to Use the Switch Construct

This is another very powerful construct in Linux scripts, and you can use it when you need to use nested conditions, but you don't want to write complicated of-else-elif chains. Here is an example of how it works.

```
#!/bin/bash

echo -n "Enter a number: "
```

```
read num


case $num in
100)
echo "Hundred!!" ;;
200)
echo "Double Hundred!!" ;;
*)
echo "Neither 100 nor 200" ;;
esac
```

The conditions go in between two keywords – case and esac. And we use *) to match all the inputs that are not 100 or 200.

How to Use Command Line Arguments

There are several reasons why you might want to get an argument straight from the command shell, and the next example demonstrates how you do this using bash shell:

```
#!/bin/bash
echo "Total arguments : $#"
echo "First Argument = $1"
echo "Second Argument = $2"
```

Run the script with an extra two parameters following the name. The script is saved as test.sh, and you can see the calling procedure here:

```
$ ./test.sh Hey Howdy
```

$1 accesses the first argument while $2 does the second and so on. We use $# to get the total argument number.

How to Use Names to Get Arguments

The next example demonstrates how you can use their names to get the command line arguments:

```
#!/bin/bash


for arg in "$@"
do
index=$(echo $arg | cut -f1 -d=)
val=$(echo $arg | cut -f2 -d=)
case $index in
X) x=$val;;
Y) y=$val;;
*)
esac
done
((result=x+y))
echo "X+Y=$result"
```

Save and call this script test.sh and call it like this:

```
$ ./test.sh X=44 Y=100
```

You should see a return of X+Y=144. We store the arguments in 'S@' and then use the cut command in Linux to get them.

How to Concatenate Strings

Many modern shell scripts use string processing and one of the most common methods is string concatenation. Bash makes this easy and precise and the example below shows you how it works:

```
#!/bin/bash


string1="Ubuntu"
```

```
string2="Pit"

string=$string1$string2

echo "$string is a fantastic Linux resource for beginners."
```

Running this should give you an output of "UbuntuPit is a great Linux Resource for Beginners.:

How to Slice Strings

Most programming languages provide ready-made functions to help you cut parts of a string. Bash doesn't but the example below shows how to do it with a parameter expansion:

```
#!/bin/bash

Str="Learn the Bash Commands from UbuntuPit"

subStr=${Str:0:20}

echo $subStr
```

Running this script should result in "Learn the Bash Commands" as an output. The parameter expansion is the form of ${VAR_NAME:S:L} with S denoting the start of the slice and L indicating how long the slice should be.

How to Use Cut to Extract Substrings

Linux has a command called cut that you can use in your scripts to cut a part out of the string – this is the substring and the next example demonstrates how this works:

```
#!/bin/bash

Str="Learn the Bash Commands from UbuntuPit"

#subStr=${Str:0:20}


subStr=$(echo $Str| cut -d ' ' -f 1-3)

echo $subStr
```

How to Add Two Values

Arithmetic operations are easy to do in a Linux shell script. You can see from the example below how two numbers are taken as an input and added together:

```
#!/bin/bash

echo -n "Enter the first number:"

read x

echo -n "Enter the second number:"

read y

(( sum=x+y ))

echo "The result of the addition=$sum"
```

You can see from this that it is quite straightforward to add two numbers in the bash shell.

How to Add Multiple Values

If you want to get multiple user inputs in your script and add them, you would use loops. The next example shows how this is done:

```
#!/bin/bash

sum=0

for (( counter=1; counter<5; counter++ ))

do

echo -n "Enter the Number:"

read n

(( sum+=n ))

#echo -n "$counter "

done

printf "\n"

echo "Result is: $sum"
```

If you do not include the (()), you wont get an addition operation. Instead it will be concatenation so make sure you double check your script before executing it.

Bash Functions

All computer programming languages rely on functions and the Linux Shell script is no exception. Functions allow you to create blocks of code that you need to use frequently, rather than writing the same code repeatedly. In the next demonstration, you can see how these functions work in bash scripts:

```
#!/bin/bash

function Add()

{

echo -n "Enter a Number: "

read x

echo -n "Enter another Number: "

read y

echo "Addition is: $(( x+y ))"

}


Add
```

What we did here was added two numbers but, unlike the previous addition example, this time we used a function named Add. Whenever you need to do an addition calculation again, you can simply call the function.

Functions with Return Values

One of the best things about functions is that you can use them to pass data between functions. This is useful in lots of different scenarios, so take a look at the next example to see how it works:

```
#!/bin/bash
```

```
function Greet() {

str="Hello $name, what brings you to UbuntuPit.com?"
echo $str
}

echo "-> what's your name?"
read name

val=$(Greet)
echo -e "-> $val"
```

The output will show data coming from the function called Greet.

How to Use Bash Scripts to Create Directories

One way that developers can increase their productivity is to use shell scripts to run system commands. The next example is a simple one demonstrating how directories are created in a shell script:

```
#!/bin/bash
echo -n "Enter directory name ->"
read newdir
cmd="mkdir $newdir"
eval $cmd
```

Take a close look at this script and you will see that it does nothing more than call a shell command named mkdir and passing the directory name to it. The result should be a new directory created in your filesystem. The execution command can also be passed inside backticks (') as you can see below:

`mkdir $newdir`

How to Create a Directory After Checking For Existence

The program we created above won't work if there is a folder named the same in the current working directory. Instead, we need to check if there is a folder called $dir first and, if there isn't we can create one. Here's how it's done:

```
#!/bin/bash

echo -n "Enter the directory name ->"

read dir

if [ -d "$dir" ]

then

echo "The directory exists"

else

`mkdir $dir`

echo "Directory created"

fi
```

If you want to improve your bash scripting skills, write the above program using eval.

How to Read Files

Bash scripts offer an effective way of reading files. The example below demonstrates how to use shell scripts to read a file. First create a new file and name it editors.txt, make it executable and add the following:

1. Vim
2. Emacs
3. ed
4. nano
5. Code

This code will result in the five lines being printed on your screen.

```
#!/bin/bash

file='editors.txt'

while read line; do

echo $line

done < $file
```

How to Delete Files

In the next program, we will look at how files are deleted within a shell script. First, the program asks the user for input in the form of the filename. If it exists, the file will be deleted. In this example, the rm command is used for the deletion:

```
#!/bin/bash

echo -n "Enter the filename ->"

read name

rm -i $name
```

Where it asks for the filename, type in editors.txt and, when you are asked to confirm it, press the Y key. The file should be deleted.

How to Append to Files

The next shell example shows you how to use bash scripts to append data to one of your filesystem files. This will add an extra line into the editors.txt file:

```
#!/bin/bash

echo "Before you append the file"

cat editors.txt

echo "6. NotePad++" >> editors.txt

echo "After you append the file"

cat editors.txt
```

By now, you should have noticed that standard terminal commands are being used from the bash scripts.

How to Test For File Existence

In the next example, we can see how to use bash programs to find out if a file exists:

```
#!/bin/bash

filename=$1

if [ -f "$filename" ]; then

echo "File exists"

else

echo "File does not exist"

fi
```

The argument is the filename and it is passed directly from the command-line.

How to Send Mail from Shell Scripts

Sending emails from a bash script is quite simple to do. In the example below, you can see one way to do this:

```
#!/bin/bash

recipient=" admin@example.com"

subject=" Greetings"

message=" Welcome to UbuntuPit"

`mail -s $subject $recipient <<< $message`
```

This will send an email to the specified recipient with the specified subject line and message.

How to Parse the Date and Time

In the next script example, you can see how to use scripts to handle time and date. We use the data command to get the required information and the script will do the necessary parsing:

```
#!/bin/bash
year=`date +%Y`
month=`date +%m`
day=`date +%d`
hour=`date +%H`
minute=`date +%M`
second=`date +%S`
echo `date`
echo "Current Date is: $day-$month-$year"
echo "Current Time is: $hour:$minute:$second"
```

Save and run the program to see how it all works and also run the date command form your own terminal.

Run this program to see how it works. Also, try running the date command from your terminal.

How to Use the Sleep Command

You can use the sleep command to make your shell script pause in between each instruction. This can be useful for many things, including when you want to perform system-level jobs. In the next example, the sleep command can be seen working in a shell script:

```
#!/bin/bash
echo "How long should you wait?"
read time
sleep $time
echo "Waited for $time seconds!"
```

The program stops the execution of the final instruction for the specified time – that time is provided by the user.

How to Use the Wait Command

We can use the wait command to pause bash script system processes. The example below demonstrates in detail how this all works in a bash script:

```
#!/bin/bash

echo "Testing the wait command"

sleep 5 &

pid=$!

kill $pid

wait $pid

echo $pid was terminated.
```

Save and run the program and see for yourself how it all works.

How to Display the Last Updated File

On occasion, you may need to locate the last file that was updated – this may be needed for specific operations. Below you can see a simple program that shows you how to use the awk command in bash to do this. It will list one of two things – the last file updates or the last file created in the current working directory.

```
#!/bin/bash


ls -lrt | grep ^- | awk 'END{print $NF}'
```

Copy this code into a file and run it to see how it all works.

How to Add Batch Extensions

In the program below, we are applying a custom extension to every file inside a specified directory. For the purposes of this demonstration, create a brand new directory and put a few files in it. In my folder, I have five files. Each is called test followed by a number between 0 and 4. For the sake of this demonstration the script will add .UP to the end of each file – you can add whatever

extension you want, just make sure you change the script accordingly.

```
#!/bin/bash

dir=$1

for file in `ls $1/*`

do

mv $file $file.UP

done
```

One important note – you should not try running this script from a regular directory, only from a test one. You must also provide a command-line argument containing the directory name for the files – the period (.) should be used for the current working directory.

How to Print the Number of Files or Directories

The script below will find how many folders or files are in a specified directory, using the find command provided in Linux to do it. The name of the directory where you are looking for the files should be passed from the command-line.

```
#!/bin/bash


if [ -d "$@" ]; then

echo "Files found: $(find "$@" -type f | wc -l)"

echo "Folders found: $(find "$@" -type d | wc -l)"

else

echo "[ERROR] Please retry with another folder."

exit 1

fi
```

If the directory you specify is not available you will be asked to try again with a different one. The same message will appear if you do

not have permission to access the specified directory.

How to Clean Log Files

In the next demonstration, you can see how shell scripts can be used easily in real life. We are going to delete every log file stored in the directory called /var/log. If you want to clean up different logs, simply change the variable holding the directory:

```
#!/bin/bash

LOG_DIR=/var/log

cd $LOG_DIR


cat /dev/null > messages

cat /dev/null > wtmp

echo "Logs cleaned up."
```

Important – this shell script must be run as root.

Using Bash to Back Up Your Script

Shell scripts provide a strong, effective way of backing up directories and files. In the next example, you can see how to back up every file and directory that was modified in the preceding 24 hours. The Linux find command is used for this.

```
#!/bin/bash


BACKUPFILE=backup-$(date +%m-%d-%Y)

archive=${1:-$BACKUPFILE}


find . -mtime -1 -type f -print0 | xargs -0 tar rvf "$archive.tar"

echo "Directory $PWD backed up in archive file \"$archive.tar.gz\"."

exit 0
```

The file and directory names will be printed once the backup is successfully completed.

How To See If a User is Root

In the next example, you can see how a bash script is used to see if a user is root:

```
#!/bin/bash

ROOT_UID=0


if [ "$UID" -eq "$ROOT_UID" ]

then

echo "You are root."

else

echo "You are not root"

fi

exit 0
```

The output will depend entirely on which user is running the script. Root users are matched by the $UID.

How to Remove Duplicate Lines From a File

It can take a lot of time to process files and it can reduce admin productivity severely. One of the most time-consuming tasks is trying to find duplicate lines in your files but a short, simple shell script can save the day:

```
#! /bin/sh


echo -n "Enter the Filename-> "

read filename

if [ -f "$filename" ]; then
```

```
sort $filename | uniq | tee sorted.txt

else

echo "No $filename in $pwd...try again"

fi

exit 0
```

This script will look through every line of the specified file and will remove any duplicate. Those lines are then put into a new file, leaving the original file as it is.

How to do System Maintenance

If you want to make life a little easier for yourself, you can use a small shell script to upgrade your system, rather than having to do it all manually. Here's an example of how this is done:

```
#!/bin/bash


echo -e "\n$(date "+%d-%m-%Y --- %T") --- Starting work\n"


apt-get update

apt-get -y upgrade


apt-get -y autoremove

apt-get autoclean


echo -e "\n$(date "+%T") \t Script Terminated"
```

This script will also remove old packages that you do not need anymore. Make sure to run it using sudo – if you don't, it won't work.

Shell scripting in Linux isn’t as hard as you might think it is, and your scripts can be as diverse as you want. There is no limit in what they can or cannot do and, if you are new to this, new to using Linux and

the bash script, take the time to learn these fundamental scripts – they will form the basis of just about everything you want to do. Once you have learned them, take the time to play around with them, change things, learn how they work, and understand exactly what they can do.

I've tried to give you a decent insight into what you need for Linux shell scripting today. The subject goes far deeper than this, though, so I've kept things basic and simple to help you learn. While it isn't too technical, it is a good starting point.

Chapter Nine: Linux Shell Scripting for Functions

When you write your scripts, there is every chance that you will use the same code in several locations. If it is a small code, rewriting it isn't such a big deal but, where you have large code that needs to be used in several places, it gets tedious. The bash shell provides the solution – encapsulation. In this way, the code is encapsulated in a function, and this can be used wherever it is needed throughout the code.

The Basics

When you start to write more complex scripts, there is every chance that parts of your code performing specific tasks will need to be reused. That could be something as simple as displaying a message multiple times to get answers from users. Or it could be more complex, perhaps complicated calculations that need to be used repeatedly in the script. No matter what it is, having to write the code over and again starts to get tiring, but luckily, the bash shell makes it easy. We simply write the code once and wrap it in a function. Name it according to the contents and, when you need that piece of code anywhere in the script, you can simply call the function name.

Function Creation

Functions can be created in two formats. The first is using the keyword function with the function name assigned to it. The function's name attribute defines the function, but the name should not be one of the reserved keywords; it should be unique and relate to the function contents. The commands are made up of several bash shell commands and when the function is called, the commands are executed in the order they were written in the function.

The second format follows the same function creation pattern used in other programming languages.

```
$ cat test15567
#!/bin/bash
function Myfunction1 {
echo "You are looking at the example of a Linux shell function."
}
count=1
while [ $count -le 10 ]
do
Myfunction1
count=$[ $count + 1 ]
done
echo "You have reached the end of the loop."
Myfunction1
echo "This is the finishing point of your Linux shell script. Where do you want to go now?"
```

$bash -f main.sh

You are looking at the example of a Linux shell function.

You are looking at the example of a Linux shell function.

Repeated eight more times, until the final line of:

You have reached the end of the loop.

You are looking at the example of a Linux shell function.

This is the finishing point of your Linux shell script. Where do you want to go now?

When the name, Myfunction1, is referred to, the bash shell returns its definition and the commands defined in it are executed. The

definition does not need to be first in the shell script but if you try using a function before the definition, an error message is thrown:

```
$ cat test15567
#!/bin/bash
count=1
echo "You are seeing this line that comes before the function definition."

function Myfunction1 {
echo "You are looking at the example of a Linux shell function."
}

while [ $count -le 10 ]
do
Myfunction1
count=$[ $count + 1 ]
done
echo "You have reached the end of the loop."
function Myfunction2
echo "This is a definition of another function"
```

$bash -f main.sh

```
You are seeing this line that comes before the function definition.
You are looking at the example of a Linux shell function.
Repeated nine more times, until the final line of:
You have reached the end of the loop.
main.sh: line 1: $: command not found
```

main.sh: line 17: syntax error near unexpected token `echo'

main.sh: line 17: `echo "This is a definition of another function"'

Food for thought

When we defined the first function, the shell was defined following a series of commands. When we used Myfunction1 in the script, it was immediately located by the shell. However, our script attempted to use Myfunction2 before it had been defined, throwing an error message.

The point here is that a function must be defined before it can be used in a script and you need to be careful about function names. These must be unique, and they must not be keywords. Where a function is redefined, the new definition overrides the previous one. What is important here is that we didn't get an error message on the screen but something has definitely gone wrong; finding out what it is will be hard and these are all things you need to keep in mind when you start writing Linux scripts.

```
$ cat test15567
#!/bin/bash
count=1
function Myfunction1 {
echo "You are looking at the example of a Linux shell function."
}

Myfunction1

function Myfunction1 {
echo "This is a definition of another function"
}
```

Myfunction1

echo "This Linux script ends here."

$bash -f main.sh

You are looking at the example of a Linux shell function.

This is a definition of another function

This Linux script ends here.

Returning a Value in Linux Functions

The bash shell tends to treat functions such as mini-scripts that have an exit status. You can generate exit status for functions in three different ways. The exit status by default is something that is returned by the final command in a function. After the execution of the function, you may use the standard $? variable to determine its exit status.

Passing Parameters to a Function

The bash shell treats functions like mini-scripts, which means that you may pass parameters to a function like a regular script. Functions may use a standard parameter environment variables to represent the parameters that are passed to a function on the command line.

$ cat test6

#!/bin/bash

In this example, I will be passing parameters to a function

function addemdigits {

if [$# -eq 0] || [$# -gt 2]

then

echo -1

elif [$# -eq 1]

```
then
echo $[ $1 + $1 ]
else
echo $[ $1 + $2 ]
fi
}
echo -n "I am now Adding 30 and 85: "
valuedigits=`addemdigits 30 85`
echo $valuedigits
echo -n "Shall we try adding a single number: "
valuedigits=`addemdigits 10`
echo $valuedigits
echo -n "I am not trying to add any numbers now: "
valuedigits=`addemdigits`
echo $valuedigits
echo -n "Finally, I am trying to add three numbers: "
valuedigits=`addemdigits 40 15 90`
echo $valuedigits
```

$bash -f main.sh

```
I am now Adding 30 and 85: 115
Shall we try adding a single number: 20
I am not trying to add any numbers now: -1
Finally, I am trying to add three numbers: -1
```

The addemdigits function checks the number of parameters that are passed to the function. If there are no parameters or more than two

parameters, the value is returned as -1. If there is a single parameter, it adds that to itself for the result. If there are two parameters, it adds all of them for the result. Since the function uses a special parameter environment variable for its parameter values, it cannot access the script parameter values from the command line of the script.

Passing Arrays to Functions

Passing arrays into functions is a precise art but, to start with, it may be confusing. If an array variable is passed as a single variable, it will not work:

```
$ cat badtest3
#!/bin/bash
function func555 {
echo "The parameters being used in the function are: $@"
myarray=$1
echo "The array that is received is as follows: ${myarray[*]}"
}
thisarray=(1 2 3 4 5 6 7 8 9 10)
echo "This is the original array : ${thisarray[*]}"
func555 $thisarray
```

```
$bash -f main.sh
This is the original array : 1 2 3 4 5 6 7 8 9 10
The parameters being used in the function are: 1
The array that is received is as follows: 1
```

If you try to use the array variable as a function parameter, the function takes the first value in the array variable. Fixing this requires that the array variable is broken down into its individual values, which are then used as function parameters. You can reassemble

those parameters in the function as a new array variable. Here's an example:

```
$ cat test10
#!/bin/bash
function testingfunction {
local myarray
myarray=(`echo "$@"`)
echo "This the new value of the array: ${myarray[*]}"
}
thisarray=(1 2 3 4 5 6 7 8 9 10)
echo "This the original value of the array ${thisarray[*]}"
testingfunction ${thisarray[*]}
```

```
$bash -f main.sh
This the original value of the array 1 2 3 4 5 6 7 8 9 10
This the new value of the array: 1 2 3 4 5 6 7 8 9 10
```

The variable named $thisarray contains the individual array values, placing them on the function's command line. The function then builds the array variable using command line parameters and, once you are in the function, you can use the array as you would any other array.

```
$ cat test11
#!/bin/bash
function addingthearray {
local sum=0
local thisarray
thisarray=(`echo "$@"`)
```

```
for values in ${thisarray[*]}
do
sum=$[ $sum + $values ]
done
echo $sum
}
iarray=(1 2 3 4 5 6 7 8 9 10 11 12 13)
echo "This is the original value of the array: ${iarray[*]}"
arg1=`echo ${iarray[*]}`
theresult=`addingthearray $arg1`
echo "This is the final result : $theresult"
```

$bash -f main.sh

This is the original value of the array: 1 2 3 4 5 6 7 8 9 10 11 12 13

This is the final result: 91

Handling User Input

You have learned how to write shell scripts that interact with data, files, and variables on a Linux operating system. However, sometimes you need to write a script that must interact with a person who is running the script. The shell gives us several methods to retrieve data from people, including some command line parameters, command-line options, and reading input from the keyboard. The chapter will walk you through the process of incorporating different methods in the bash shell scripts to get data from the person who is running the script.

Variables called positional parameters are assigned to the command line parameters by the bash shell, including the name of the program executed by the shell. We use standard numbers for the positional parameter variables - $0 is the program name, $1 is the first parameter, $2 the second one, and so on up to $9.

Conclusion

Now that you have reached the end of the book, you should have a good grasp of Linux command line and shell scripting. Unlike other operating systems, the Linux operating system is complicated. You don’t get to see a graphical interface where you can work with a mouse. It is the “all-keyboard” thing that makes Linux hard to learn. However, it is not that hard with this book in your pocket. You can use this book as a reference guide whenever you operate the Linux command line or jump to the editor to write shell scripts. Being consistent and regularly practicing with codes will help you learn the commands and codes faster than just reading the scripts.

Keep this book by your side and you’ll always have keyboard shortcuts and commands at your fingertips. Now, let’s get started with Linux!

References

22 best Linux text editors for coding {2020 Reviews} . (2020, July 12). Knowledge Base by phoenixNAP. https://phoenixnap.com/kb/best-linux-text-editors-for-coding

Bash while loop examples . (2020, November 5). nixCraft. https://www.cyberciti.biz/faq/bash-while-loop/

Blum, R. (n.d.). *Linux Command Line and Shell Scripting* . https://inf.ocs.ku.ac.th/Download/Wiley.Linux.Command.Line.and.Shell.Scripting.Bible.May.2008.pdf

Broida, R. (2017, February 9). *How to install Linux* . CNET. https://www.cnet.com/how-to/how-to-install-linux/

Ward, B. (n.d.). *How Linux Works* . index-of.es/. https://index-of.es/Varios-2/How%20Linux%20Works%20What%20Every%20Superuser%20Should%20Know.pdf