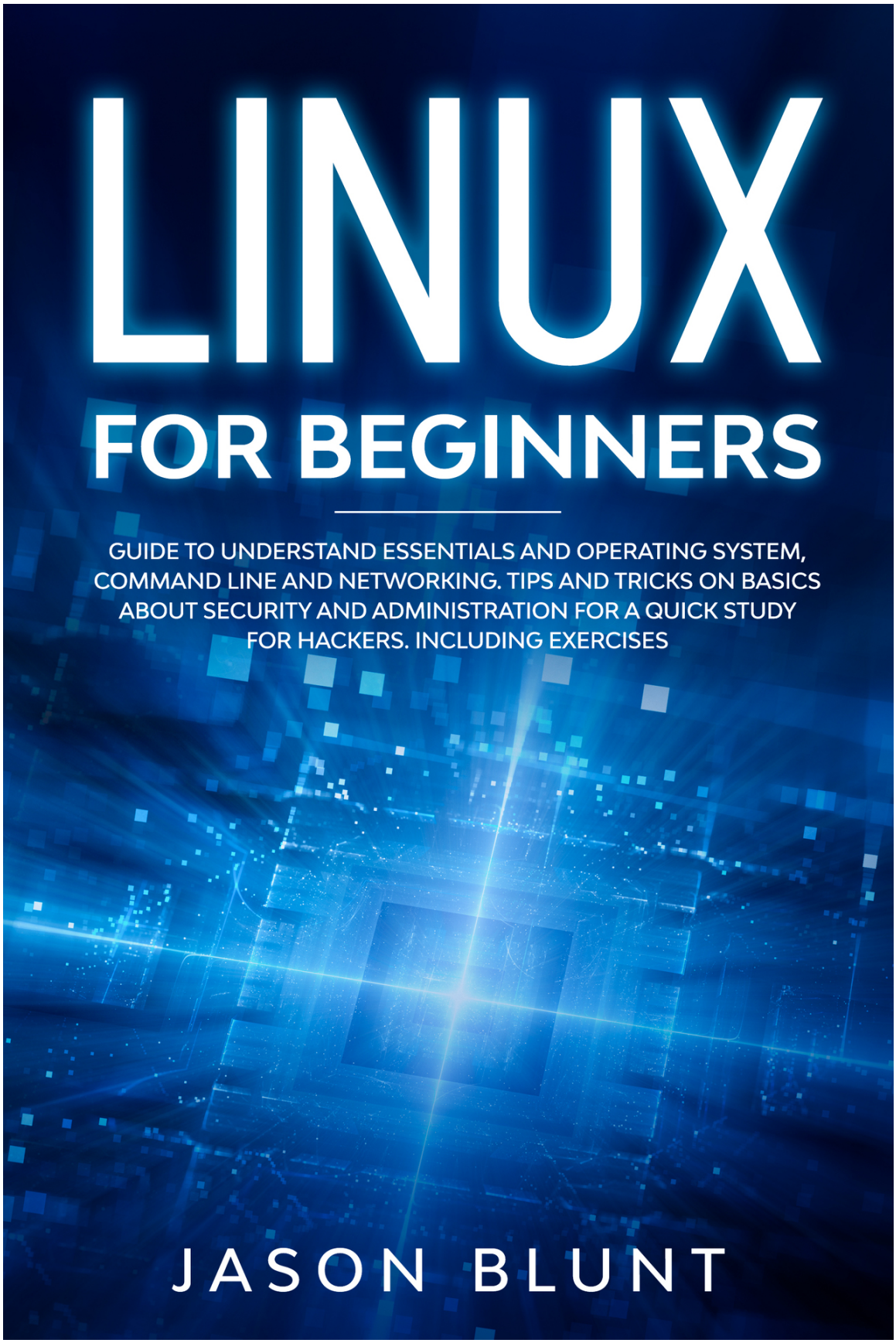
LINUX

FOR BEGINNERS

GUIDE TO UNDERSTAND ESSENTIALS AND OPERATING SYSTEM, COMMAND LINE AND NETWORKING. TIPS AND TRICKS ON BASICS ABOUT SECURITY AND ADMINISTRATION FOR A QUICK STUDY FOR HACKERS. INCLUDING EXERCISES

JASON BLUNT


LINUX
FOR BEGINNERS
GUIDE TO UNDERSTAND ESSENTIALS AND OPERATING SYSTEM, COMMAND LINE AND NETWORKING. TIPS AND TRICKS ON BASICS ABOUT SECURITY AND ADMINISTRATION FOR A QUICK STUDY FOR HACKERS. INCLUDING EXERCISES
JASON BLUNT

LINUX FOR BEGINNERS:

GUIDE TO UNDERSTAND ESSENTIALS AND OPERATING SYSTEM, COMMAND LINE AND NETWORKING. TIPS AND TRICKS ON BASICS ABOUT SECURITY AND ADMINISTRATION FOR A QUICK STUDY FOR HACKERS. INCLUDING EXERCISES.

Contents

Description

Introduction

Chapter 1 What is Linux?

Chapter 2 What is Ubuntu?

Chapter 3 Linux set-up process

Chapter 4 Linux Command Lines

Chapter 5 Introduction to Linux Terminals

Chapter 6 Variables

Example:

Example:

Example:

Chapter 7 Using the Shell

The Bash Shell

The Shell Command

Putting Together Your Shell Commands

Chapter 8 Nested If Statements

If Else

If Elif Else

Chapter 9 Boolean Operators

Case Statements

Chapter 10 Working with Linux Files and Directories

Spaces in file or directory names

Hidden directories and files

Practical work

More on running commands

Man pages

Practical work

Removing a directory

Creating a blank file

Copying a file or directory

Moving a file or directory

Renaming files and directories

Removing files

Removing directories that aren't empty

One last note

Practical work

Chapter 11 Log Analysis

Architecture of System Logs

System logging

Syslog File Review

Syslog files

Log file rotation

Syslog entry analysis

Using the tail command to monitor log files

Using logger to send a syslog message

Reviewing Journal Entries for Systemd

Systemd Journal Preservation

Permanently storing the system journal

Maintaining time accuracy

Setting the local time zone and clock

The Chronyd Service

Chapter 12 Create a bootable stick

Systemd Journal Preservation

Permanently storing the system journal

Maintaining time accuracy

Setting the local time zone and clock

The Chronyd Service

Chapter 13 BONUS!

ping

traceroute

/etc/hosts

DNS

Chapter 14 Programming – A Logical Breakdown

Programs and How They Run

What Makes a Program?

Program Flow - Procedural, Functional, Object-Oriented, and So On

C and C-Style Languages

Objects and Classes

Control Flow (Loops)

Conclusion

Description

Linux is a relatively complex system of different interconnected programs. There are many components to a Linux system. Even though most of them are interchangeable and allow for a great deal of customization, there are some components that a Linux must have in order to function properly. Some of the programs are more important than others, of course, but the core principles always stay the same.

This program is responsible for the proper booting of a Linux system. It selects the optimal parameters for the kernel and the initial RAM disk of Linux, which is also known as ignited. The kernel is the core of the operating system. It starts the initialization process immediately after being launched. The init process can be replaced by any suitable replacement like the system, but the function of the boot loader stays the same. The initial RAM disk gives you temporary memory that loads critical files before the root system is loaded.

If you have an older computer that runs with a basic input/output system, you can find your boot loader in the Master Boot Record which takes up the first 512 bytes on your disk. Newer computers have a Unified Extensible Firmware Interface and store it in a special partition. The partition is called the EFI system partition.

Immediately after your device is powered on, a self-test process is performed immediately. This self-test is called the Power-On Self-Test and if it is successful, the boot loader will be loaded by the BIOS or UEFI.

Despite being very small and simple, boot loaders play a critical role in the process. On most Linux-related forums you will find people that have a problem with their boot loader, and you will also find people that can help you fix the problems you might encounter. To avoid these problems you will have to understand which boot loaders are the best for you and what kind of role they play in the booting process.

There are many different boot loaders that you can install on your Linux systems, but below we will talk about the most popular and best to work with.

This guide will focus on the following:

- What is Linux?

- What is Ubuntu?
- Linux set-up process
- Linux Command Lines
- Variables
- Using the Shell
- Nested If Statements
- Boolean Operators
- Log Analysis
- Create a bootable stick… AND MORE!!!

Introduction

In this chapter, I will provide some brief information on this operating system's architecture, and discuss the different flavors of Linux that a beginner can choose from.

Linux Architecture

Linux architecture can be divided into two spaces. The User Space and the Kernel Space.

- *User Space* – This is where the applications are used. The GNU C library, in the User space, is the interface that connects to the kernel and transitions between User and Kernel space. This uses all the available memory.
- *Kernel Space* – All Kernel services are processed here. The Kernel space is further divided into 3.
    - *System Call Interface* – A User process can access Kernel space through a System Call. When a System Call is performed, arguments are passed from User to Kernel space. This is the layer that implements basic functions.
    - *Kernel Code* – This is the architecture-independent code, and can be seen in all architectures that Linux supports.
    - *Architecture-Dependent Kernel Code* – This is the layer for platform-specific codes.

Linux Distributions

Each Linux distribution consists of a Linux kernel plus utilities and configuration files. Most Linux distributions can be downloaded from their websites.

Let's look at how several of the popular Linux distributions, or flavors, differ from each other based on the following criteria:

Availability

As previously mentioned, Linux is a free software, but companies offering a support contract and proprietary components offer it for a fee. In the table below, Red Hat Enterprise and SUSE Enterprise both offer Linux

commercially, but they also have the free alternatives – Fedora and openSUSE.

Package Format

Linux distributions come in packages. Packages are files grouped into one single file. RPM is the most commonly used.

Release Cycle

This is how often a distribution releases new software. The ones with shorter release cycles provide the latest software in the shortest possible length of time, while those distributions with long release cycles aim to offer the most stable environments possible. A distribution can have it done both ways. Look at Ubuntu who releases both long-term support (LTS) versions (longer cycle) and the latest software through a 6month cycle.

To help you decide on which is the right distribution for you, consider the criteria mentioned above and research the other fields listed below:

- *Desktop environment* : Do your research and find out if the distribution that you're eyeing has a basic look and feel that you like, please check how customizable it is

- *Hardware Compatibility* : Depending on the hardware that you are using, some drivers might not be available yet by the time you install your distro. Check from online resources first to know which ones can be supported out-of-the-box.

- *Community Support* : Find the one with a large online community. The bigger the community is, the easier it will be to find documentation and get support.

Chapter 1 What is Linux?

Linux is a family of free and open-source software operating systems based on the Linux Kernel, an operating system kernel first released on September 17, 1991 by Linus Torvalds. Linux is typically packaged in a Linux distribution (or distro for short).

Distributions include the Linux kernel and supporting system software and libraries, many of which are provided by the GNU Project.

(Source: Wikipedia.org)

The Linux kernel saw the light of day in 1991 in version 0.0.1.

Linus Torvalds, who is still in charge of kernel development today, had no idea what he would initiate with the release of his small hobby project.

Today, the term Linux is usually used as a synonym for the entire system. Strictly speaking, however, only the kernel is called Linux.

Only when combined with applications and programs that follow the GNU guidelines it becomes the system we know and love today.

The first Linux distributions were released in 1994. They had made it their business to develop usable systems as an alternative to Windows and MacOS:

Debian

Suse

Slackware

RedHat

A major step forward was the release of *Open Office* (Version 1) in 2002.

It was a full-featured open source office suite, which was available free of charge.

Of course, the functionality was still limited compared to Microsoft Office, but Open Office became better and better over time.

Libre Office has been developed later as an independent fork of Open Office. This extensive and meanwhile very mature office suite is pre-installed in Ubuntu 18.04.

Meanwhile there are countless variants of Linux distributions.

However, most of them are based on Debian, OpenSuse, Ubuntu, Arch Linux, Gentoo or Red Hat.

Chapter 2 What is Ubuntu?

The Linux operating system *Ubuntu* exists since 2004.

The Ubuntu project was founded by Marc Shuttleworth, a South African multimillionaire, who is also the main sponsor through his company *Canonical* .

In Zulu language, *Ubuntu* means "humanity towards others“.

The focus of the project is to develop an operating system that should be available to as many people as possible all over the world.

Special attention was paid to intuitive usability and tools for universal accessibility.

Ubuntu is open source software that is available free of charge.

Since the release of the first version, the degree of recognition and popularity could be increased steadily.

Ubuntu is based on *Debian* . This means that the package management created by Debian and the software selection has been adopted.

However, over time both distributions moved away from each other.

For example, Ubuntu has often been accused of being too commercial, while Debian continued to follow the philosophy of open source, denying some of the innovations that Ubuntu introduced.

Each year in April and October a new version of Ubuntu is released, which receives support for *nine months* .

It is named after the year and month of release.

Ubuntu 18.10 was released in October (10) of 2018 (18).

Support means that Ubuntu's operating system receives updates in the form of security updates and bug fixes.
However, every two years (even years!) a special version is released which will receive *support* from Ubuntu for a period of *5 years* .

This is called an *LTS version* .

LTS stands for *Long-Term-Support* .

The current version is *Ubuntu 18.04* .

Support for this version will end after 5 years in *April 2023* .

The next LTS version Ubuntu *20.04* will be released in April 2020.

Support for this version will end after 5 years in *April 2025* .

All other versions are supported for 9 months.

Ubuntu 16.04 LTS	supported until	04/2021
Ubuntu 18.04 LTS	supported until	04/2023
Ubuntu 18.10	supported until	07/2019
Ubuntu 19.04	supported until	01/2020
Ubuntu 19.10	supported until	07/2020
Ubuntu 20.04 LTS	supported unitl	04/2025

The advantage of short-term supported systems is that the software they contain is usually more up-to-date than that of the last regular LTS-version.

The LTS-version essentially remains the same for five years.

Updates only include bug fixes and security patches.

This may be too conservative for some users in the long run.

However, the older a software is, the more it is tested and therefore more stable.

Newer versions of applications, such as those included in the short-term supported versions, may be more unstable because the testing period was shorter.

In this quick guide, I describe *Ubuntu 18.04 LTS* .

Much of it is also applicable to Ubuntu 18.10 and Ubuntu 19.04.

For beginners it is recommended to try out the LTS version first.

A Shortterm- version makes sense if the LTS version does not support your hardware optimally.

In addition to the official names that refer to the release date, Ubuntu versions always have an additional fantasy name.

That is 18.04 Bionic Beaver *and* 18.10 Cosmic Cuttlefish.

Chapter 3 Linux set-up process

In this section, we will highlight the process of installing and configuring Debian, showing you every single step along the way.

The types of installations

Debian offers a variety of methods for a proper setup. This includes a graphical and a text-based installation; we will use the former. For installation media the Debian developers offer three variants:

- A CD or DVD for 32 bit and 64 bit
- A network image for 32 bit and 64 bit (a so-called Netinst-ISO)
- A tiny CD for 32 bit or 64 bits

We also have test media available. These include live images for 32 bit and 64 bit and allow you to try Debian before installing it on your computer. During the time of writing this document, version 9.5 is the current stable release of Debian. The setup described here is based on this release and the amd64 architecture.

After downloading the network image from www.debian.org/distrib, no further static images are required to be referenced in the system. Instead, it depends on the internet connection to retrieve the packages to be installed and keep your operating system up to date.

The entire process will take you about an hour, and it allows you to have a lean software selection according to your specific needs. Software packages that you do not use will not be available on your system. They can be added whenever you feel the need for them.

The target system of our installation is an XFCE-based desktop system for a single user with a web browser and a music player. For the web browser we use Mozilla Firefox and for the music player, VLC. Both programs are a permanent component of the Linux distribution. The environment we use for demonstration purposes is a virtual machine based on VirtualBox with 4 GB of RAM and 15 GB of disc space.

Installing Linux Step-by-Step

Boot Menu

In order to begin with the installation of Debian, first, boot the computer (in our case the VirtualBox image) from the ISO image you have downloaded. If you are also using a virtual machine, see your virtual machine vendor website for help on how to enable the ISO image. Next, wait for the boot menu to appear on the screen. The image below shows you the different options that are offered. Using the cursor keys, you can navigate the boot menu, and the Enter key selects an entry.

The different options are:

Graphical install: start the installation process using a graphical installer

Install: starting the installation process using a text-based installer

Advanced options: select further options like Expert mode, Automated install or Rescue mode (see image below for more details)

Help: get further help

Install with speech synthesis: starting the installation process with speech support

From the main boot menu choose the Graphical entry install and press the Enter key to proceed.

Language Selection

Next, choose the language you prefer to be used during the entire installation process. The dialogs and messages are translated accordingly. This selection does not determine the language irrevocably your Linux system will have; you can always choose a different language later.

The image below shows the dialog box. English is already pre-selected, and so you just must click the Continue button on the lower-right corner of the dialog box in order to continue.

Location Selection

Third, select regarding your location (see image below). Based on your language setting made before, the countries are listed in which the chosen language is mainly spoken. This also influences the locale settings like the time zone your computer is in. In order to have a different setting choose the entry titled other from the end of the list and go on from there. Immediately you are over, click the Continue button to proceed with step four.

Keyboard Selection

Fourth, choose your keyboard layout from the list (see image below). For the United States, the pre-selection is American English. If you use a different keyboard layout select the right one from the list. If done, click the Continue button to proceed with step five.

Network Setup

Step five includes loading the installer components from the ISO image, and the detection of the network hardware in order to load the correct network driver. Then, the installer tries to connect to the internet to retrieve an IP address via DHCP from your local network server.

When done, you can set up the hostname of your computer (see image below). Choose a unique name for your machine that consists of a single name and does not exist yet in your local network segment. It is common to use names of fruits, places, musical instruments, composers and characters from movies. In this case, we choose the name debian95 that simply represents the Linux distribution and its number version

When you are done, press the Continue key in order to proceed with step six to add a domain name like yourcompany.com (see image below). In this case, it is not needed. That’s why we leave the entry field empty. Click the Continue button on the lower-right corner to proceed with the installation.

Users and Roles

Our Linux system needs at least two users in order to be appropriately operated. One is an administrative user that has a fixed name root and the other is a regular user that we just give the name of User in this case.

In the next two steps, you set the password for the user root (see image below) and both the full name and account name for the regular user. For simplicity, we use Debian User as the full name and User as the account name. For both users, choose a password that is dissimilar and that you can remember. You will need these passwords later in order to log onto your computer.

Time Zone

Setting the correct time zone is significant for communication with other services, especially in a network. Choose the value from the list as seen in the image below. The entries in the list are based on the location you have

selected before. When done, click the Continue button to define the storage media and the accompanying partitions.

Storage Media and Partitioning

A Linux system can be distributed across several different storage media like hard disks and flash drives. Over and above, a storage media can be separated into multiple disk partitions. In order to do so, the setup program of Debian has the following methods available (see image below):

Guided - use entire disk: follow the steps as provided and use the entire disk space for the Linux installation. This creates partitions with fixed sizes.

Guided - use the entire disk and set-up LVM: follow the steps as provided and use the entire disk space for the Linux installation. This option makes use of Logical Volume Management (LVM) in order to create partitions with sizes that can be changed later.

Guided - use the entire disk and set-up encrypted LVM: follow the steps as provided and use the entire disk space for the Linux installation. This option makes use of Logical Volume Management (LVM) in order to create encrypted partitions with sizes that can be changed later.

Manual: create partitions individually. This is the expert mode and requires more profound knowledge about partitions and file system parameters.

From the list choose the entry Guided-use the entire disk. The values for partition sizes are chosen according to experience, implemented as an algorithm. A manual calculation is not required. Click the Continue button on the lower-right corner to proceed with the installation.

Next, select the disk to partition. As for our case, we have only one disk available (see image below). Later, in this guide, the disk will be referred to as /dev/sda for the 1st SCSI disk.

A disk partition refers to a piece of the storage media that is organized separately and is intended to contain a branch of the Linux file system tree. There is no universal way to do this separation correctly. This guide shows a simple but safe solution that works for a basic system. The menu in the dialog box offers the following options:

All files in one partition: use just a single partition to keep programs and user data

Separate /home partition: store programs and user data in separate partitions

Separate /home, /var, and /tmp partitions: keep user data, variable data and temporary data in separate partitions

Click the Continue button on the lower-right corner to proceed with the installation.

The next step is to confirm the partition scheme. This is calculated automatically based on experience and contains these partitions:

sda1: the first partition of the first SCSI disk is a primary partition with a size of 3 GB, formatted with the ext4 file system, and referred to as the root part of the file system tree (indicated with /)

sda5: the fifth partition of the first SCSI disk is a logical partition with a size of 1.3 GB, formatted with the ext4 file system, and reserved to store variable data of the file system tree (indicated with /var)

sda6: the sixth partition of the first SCSI disk is a logical partition with a size of 3.3 GB, formatted as a swap file system

sda7: the seventh partition of the first SCSI disk is a logical partition with a size of 311 MB, formatted with the ext4 file system, and reserved to store temporary data of the file system tree (indicated with /tmp)

sda8: the eighth partition of the first SCSI disk is a logical partition with a size of 8.8 GB, formatted with the ext4 file system, and reserved to store the user data of the file system tree aka home directories (indicated with /home)

Due to historical reasons, a hard disk can contain four primary partitions only. The fourth one is called an Extended Partition if divided into so-called logical partitions or logical drives. In our case the logical partitions /dev/sda5, /dev/sda6, /dev/sda7 and /dev/sda8 are stored on the primary partition /dev/sda4. The partitions /dev/sda2 and /dev/sda3 are not in use.

From the above list, choose the entry Finish partitioning and write changes to disk. Click the Continue button on the lower-right corner to proceed and to confirm the partition scheme (see below image). Choose yes from the list and click the Continue button to partition the disk. Note that all the data on the selected storage device will be lost and the disk will be empty.

Having divided the storage media into single partitions, the partitions will be formatted with the file system as defined before. In our case the partitions /dev/sda1, /dev/sda5, /dev/sda7 and /dev/sda8 will get an ext4 file system, and the partition /dev/sda6 will get a swap filesystem. As soon as this step is completed, the base system of Debian will be installed next.

Package Management

The ISO image contains the installer and several packages to set up the Linux base system. For example, this includes the Linux kernel being installed below.

The next step is that you must decide whether to use additional installation media or not. In our case, we have just a single installation disk and can consequentially select No from the menu (see image below). Then click the Continue button to proceed. As pointed out earlier, the software for Debian is organized in packages. These packages are provided in multiple software repositories. The repositories are made available via package mirrors that are maintained by universities, private persons, companies and other organizations. These mirrors are in different countries. In the next step, you will have to decide from which country you would like to retrieve your Debian packages. It is recommended to choose a mirror that is geographically located near you to minimize the time that is needed to transport the data from the mirror to your computer via a network. As an initial step, choose your desired country. As a second step choose a preferred mirror from the list (see image below). The list contains universities, internet providers, government services and other organizations. In case your computer network includes a proxy server to communicate with the outside world, enter the according to information here. In such an instance, we do not have that and leave the entry field empty. Click the Confirm button to proceed. As soon as the single parameters are set, the Debian installer connects to the previously selected package mirror and retrieves the package lists from there. A package list contains the packages that are available, including the name, size, and description. Depending on the quality and bandwidth of your network connection this step can take a while. Next, you are asked to take part in Popcon, the Debian package popularity contest (see image below). This information is optional and is used only by the team of developers that is responsible for the Debian packages. Based on Popcon, they figure out

which Debian packages are the ones that are installed most often. The information has a direct influence on the preparation of installation images and which packages to keep or to dismiss for the different architectures.

In our case, we do not participate in Popcon, and leave the selection to No. Click the Confirm button to proceed with the selection of software tasks (tasksel).

Software Selection

Debian offers carefully arranged selections of packages, so-called tasks. The idea behind them is to group packages for specific uses in order to simplify the installation. From the list in the dialog window (see image below) you can choose between different desktop environments, as well as a web server, a print server, an SSH server and the standard system utilities. To have a minimal installation just enable the last two entries from the list. We will install the XFCE desktop environment later.

Having confirmed the software selection, the Debian installer retrieves the needed packages from the package mirror, unpacks and then installs them. The bar below shows the progress. In this case, 140 packages must be downloaded.

Setting up GRUB

In order to start our newly installed Debian system, we must set this information too. This process is called booting the system. The software component that handles this step is named Grand Unified Boot Loader (GRUB or GNU GRUB to be precise). The entry Yes is already pre-selected, and so we can click Continue to proceed. The Debian installer needs to know where to install GRUB. The menu in the image below lists the storage media to be considered. In our case, we choose the second entry from the list (/dev/sda). Then, click Continue to proceed. Now that we are nearly finished with the basic installation, the only thing left to do is just a single dialog box to read.

Finishing the Installation

The following dialog box informs you that the installation is complete. Then click Continue to reboot the newly installed system.

After a few seconds, the text-based GRUB boot menu will appear on the screen. The boot menu includes two options: Debian GNU/Linux and

Advanced options for Debian GNU/Linux. The first menu item is highlighted and pre-selected. The second menu entry allows you to set specific boot options. For a comprehensive list of these options have a look at the GNU GRUB manual at www.gnu.org/software/grub/manual/grub

Keep the first menu item selected and press Enter to proceed and boot the new system. Your Debian system will start and initialize the necessary system services. This step will take a few seconds to be completed. Finally, a black-and-white screen will be visible on the first text terminal named tty1 (see image below) and ask you to log into the system. The login prompt consists of two components: the hostname of your Linux system (debian95) followed by space and the word login.

Debian GNU/Linux 9 debian95 tty1

debian95 login: _

Having logged into the system, you will add further software to be able to use a graphical user interface, based on the XFCE desktop. At this step of the installation process, adding further software can only be done by logging in as the administrative root user.

In order to do so, type in root at the login prompt, press Enter, wait for the text Password: to appear and type in the password set for the root user you defined before. The Linux system will welcome you (see image below) with a login message. The first line of the login message will give you a display on the time of your last login followed by the version of the Linux kernel that is currently running (line two) and the usage advice (lines five to eight).

After the welcome message Debian opens a command-line prompt:

root@debian95 ~#

This command-line prompt consists of four components:

root: the name of the user who logged in

debian95: the hostname of the Linux system

~: the current directory. In this case, it is the home directory of the user currently logged in. If not otherwise set, this is defined as /home/username for regular users and /root for the administrative user. ~ is just an abbreviation of it.

#: the login symbols. The # symbol represents the user root whereas $ is used by regular users without administrative rights

Important: You are now logged in as the administrative root user. Take care which commands you type in and check twice if in doubt. Any mistakes can lead to the necessity to install or repair your Linux system.

Adding a Graphical User Interface

At its current stage, the Debian system is fully active and can be used in production. For a desktop system suitable for a regular user, it still lacks an excellent and easy-to-use graphical user interface. In this step, we will change that and install the XFCE desktop manager.

In order to do so, we will install the following packages:

The aptitude package manager

The xdm display manager

The xfce4 desktop environment

Debian uses the package manager apt to handle the installation, the update and the removal of software packages and the related package lists. Also, apt resolve all the package dependencies and ensures that the relevant software is available on the system. In our case, a total of roughly 450 MB of data/software must be retrieved from the package mirror and installed. As pointed out earlier, this requires a working internet connection to download the needed software packages.

In order to install the three packages, type in the following commands at the command-line prompt. Just type the command after the # symbol:

root@debian95 ~# apt-get install aptitude xdm xfce4

apt will display further information regarding the packages to be installed. This includes the list of depending packages and recommended packages. In the end, you will see a command-line prompt. Type Y or press Enter to install aptitude, xdm and xfce4 as well as the depending packages. For the xdm display manager, two dialog boxes must be confirmed (see images below). Press Enter two times to confirm the use of xdm.

The retrieval of the software packages to be installed takes a while. When finished, the command-line prompt will appear again. In order to activate

the changes regarding the graphical desktop environment, restart the system. Type the command reboot at the command-line prompt as follows:

root@debian95 ~# reboot

Having restarted the Linux system, Debian welcomes us with a graphical boot screen (see image below). Press Enter to start the Linux system with the default settings.

A few seconds later a graphical login screen will be visible (see image below). Log in to the system with the regular user named user as created earlier. Type in user, press Enter and type in the password for the user. Then, press Enter again to confirm and log in.

Next, the XFCE desktop will be visible (see image below). The desktop comes with several default elements: an upper navigation bar, a lower navigation bar and desktop icons. All the mentioned elements are explained in more detail below.

Upper navigation bar: this shows buttons to access the different applications, the four virtual desktop screens, the clock and a button for various user actions such as to lock the screen, change the user, change to standby mode and exit the current session

Lower navigation bar: this bar contains several buttons to hide all the opened windows and show the empty desktop, to open a terminal, the file manager, a web browser, to find an application and to open the file manager directly with your home directory.

Adding Additional Software

Up until now the software available to be used on your Linux system has been somewhat limited. After all that, the next thing to do is to add the following four Debian packages to make your life a bit easier:

firefox-esr: the web browser Mozilla Firefox (Extended Support Release)

gnome-terminal: a terminal emulation maintained by the GNOME project

xscreensaver: a basic screensaver for the X11 system

vlc: the video player Video Lan Client (VLC)

The installation of the three packages will be done using the command-line in a terminal emulator (we will look at terminals in detail later in the guide). Currently, on your system, the X11 terminal emulator xterm is installed. In

order to open xterm, click on the terminal button in the lower navigation bar or select the entry Application > System > Xterm from the context menu.

As step one, open xterm. Next, type in the command su next to the command-line prompt as follows and press Enter:

user@debian95: ~$ su

Password:

You may remember from the previous steps that only an administrative user can install, update or remove software on a Debian system. The su command abbreviates switch user and changes your current role. Used without an additional name, the role changes to the administrative root user. At the password prompt type in the password for the administrative user and press Enter.

As an administrative user, install the packages: firefox-esr, gnome-terminal, xscreenserver and vlc as follows:

root@debian95: ~# apt-get install firefox-esr gnome-terminal xscreensaver vlc

The output is seen below:

After the installation of the four packages, you can switch back to your role as a regular user. Press Ctrl+D to quit the admin part, and press Ctrl+D again to close xterm.

The installation of Firefox has the following effects:

The new software Firefox is available from the application menu

The new command Firefox is available from the command-line

The earth icon from the lower navigation bar links to the Firefox web browser

The installation of the GNOME terminal package has the following effects:

The new software gnome-terminal is available from both the command-line and the application menu

The terminal icon from the lower navigation bar links to the GNOME terminal

The entry Open Terminal Here from the context menu refers to the GNOME terminal

In order to see the changes, select the entry Open Terminal Here from the context menu. The image below shows the terminal window. It comes with a white background and a bigger font that is easier to read.

Exiting Linux

In order to use Linux properly, you also must learn how to exit Linux and to reboot the system. Below you will learn how to quit the XFCE desktop environment, to shut down the Linux system and to reboot the system. We will look at two methods; the first is based on the graphical interface and the second can also be used on non-graphical systems like servers and wireless routers.

Quitting the XFCE Desktop Environment

The easiest way to quit the XFCE desktop and to log out from your current session is to click on the button in the right corner of the upper navigation bar. The button is labeled with your username, which in our case is Debian User. From the small menu select the last item labeled Log Out (see image below). Alternatively, you can choose either the item labeled Log Out from the Application button in upper-left corner or from the context menu.

A second window opens that contains the five buttons labeled Log Out, Restart, Shutdown, Suspend and Hibernate (see image below). Click on Log Out to quit your session. Subsequently, you will return to the login screen.

Shutting Down the Linux System

The way to shut down the entire Linux system is like exiting XFCE. Click on the button in the right corner of the upper navigation bar. The button is labeled with your username which in our case is Debian User. From the small menu (see image above) select the item labeled Shut Down. From the next dialog window click on the button labeled with Shut Down to stop the Linux system.

From a terminal session you can run the commands halt or shutdown -h now as an administrative user. A regular user is not allowed to issue these commands.

```
# halt
```

Rebooting the Linux System

In order to restart the Linux system, click on the button in the right corner of the upper navigation bar. The button is labeled with your username, which in our case is Debian User. From the small menu select the item labeled Restart (see image above). From the next dialog window click on the button labeled Restart to reboot the Linux system.

From a terminal session, you can run the command reboot or shutdown -r now as an administrative user. A regular user is not allowed to issue these commands.

```
# reboot
```

Subsequently, the system will reboot, and you will return to the boot screen.

Chapter 4 Linux Command Lines

At this juncture, you should have a fair understanding of basic commands, and Linux should be installed in your system. You now have an incredible opportunity ahead of you – a completely blank slate where you can begin to design an operating system. With Lunix, you can easily customize your operating system so that is does exactly what you would like for it to do. To get started, you need to install a selection of reliable and functional applications.

For ease of explanation, it is assumed that you are using Ubuntu. When you are looking to install an application in Linux, the process is quite different than what you would encounter in Windows. With Windows, you normally need to download an installation package sourced at a website, and then you can install the application.

With Linux, this process is not necessary as most of the applications are stored in the distribution's repositories. To find these applications, follow these steps.

Go to System -> Administration -> Synaptic Package Manager

When you get to this point, you need to search for the package that you require. In this example, the package shall be called comp. Next, you should install the package using a command line as follows: -

sudo apt-get install comp

Linux also has another advantage over some popular operating systems. This include the ability to install more than one package at a time, without having to complete a process or more between windows. It all comes down to what information is entered in the command lines. An example of this is as follows: -

sudo apt-get install comp beta-browser

There are even more advantages (other than convenience) to being able to install multiple packages. In Linux, these advantages include updating. Rather than updating each application, one at a time, Linux allows for all the applications to be updated simultaneously through the update manager.

The Linux repository is diverse, and a proper search through it will help you to identify a large variety of apps that you will find useful. Should there

be an application that you need which is not available in the repository, Linux will give you instructions on how you can add separate repositories.

The Command Line

Using Linux allows you to customize your system to fit your needs. For those who are not tech savvy, the distributions settings are a good place to change things until you get what you want. However, you could spend hours fiddling with the available settings and still fail to find setting that is perfect for you.

Luckily, Linux has a solution and that comes in the form of the command line. Even though the command line sounds complex, like something that can only be understood by a tech genius, it is quite simple to discern.

The beauty of adjusting things in your operating system using the command line, so that the sky is the limit and creativity can abound.

To begin, you need to use “The Shell”. This is basically a program which can take in commands from your keyboard and ensure that the operating systems performs these commands. You will also need to start a “Terminal”. A terminal is also a program and it allows you to interact with the shell.

To be a terminal, you should select the terminal option from the menu. In this way, you can gain access to a shell session. In this way you can begin practicing your command prompts.

In your shell session, you should see a shell prompt. Within this shell prompt you will be see your username and the name of the machine that you are using, followed by a dollar sign. It will appear as follows: -

[name@mylinux me] $

If you try to type something under this shell prompt, you will see a message from bash. For example,

[name@mylinux me] $

lmnopqrst

bash: lmnopqrst

command not found

This is an error message where the system lets you know that it is unable to comprehend the information you put in. If you press the up-arrow key, you

will find that you can go back to your previous command, the lmnopqrst one.

If you press the down arrow key, you will find yourself on a blank line.

This is important to note because you can then see how you end up with a command history. A command history will make it easier for you to retrace your steps and make corrections as you learn how to use the command prompt.

Command Lines for System Information

The most basic and perhaps most useful command lines are those that will help you with system information. To start, you can try the following: -

Command for Date

This is a command that will help you to display the date.

root@nagsis: -# date

Thursday May 21 12.31.29 IST 2o15

Command for Calendar

This command will help display the calendar of the current month, or any other month that may be coming up.

root@nagsis: -# cal

Command for uname

This command is for Unix Name, and it will provde detailed information about the name of the machine, its operating system and the Kernel.

root@nagsis: -# uname –a

Linux Nagsis 3.8.0-19-generic #30

Ubuntu SMp Wed May 20 16:36:12 UTC

2013 i686 i686 GNU/Linux

Navigating Linux Using Command Lines

You can use the command lines in the same way that you would use a mouse, to easily navigate through your Linux operating system so that you

can complete the tasks you require. In this section, you will be introduced to the most commonly used commands.

Finding files in Linux is simple, as just as they are arranged in order in familiar Windows programmes, they also follow a heriarchical directory structure. This structure resembles what you would find with a list of folders and is referred to as directories.

The primary directory within a file system is referred to as a root directory. In it, you will be able to source files, and subdirectories which could contain additional sorted files. All files are stored under a single tree, even if there are several storage devices.

pwd

pwd stands for print working directory. These will help you to choose a directory where you can store all your files. Command lines do not give any graphical representation of a filing structure. However, when using a command line interface, you can view all the files within a parent directory, and all the pathways that may exist in a subdirectory.

This is where the pwd comes in. Anytime that you are simply standing in a directory, you are in a working directory. The moment you log onto your Linux operating system, you will arrive in your home directory (which will be your working directory while you are in it). In this directory, you can find all your files. To identify the name of the directory that you are in, you should use the following pwd command.

[name@mylinux me] $pwd

/home/me

You can then begin exploring within the directory by using the ls command. ls stands for list files in the directory. Therefore, to view all the files that are in your working directory, type in the following command and you will see results as illustrated below.

[name@mylinux me] $ls

Desktop bin linuxcmd

GNUstep ndeit.rpm nsmail

cd

cd stands for change directory. This is the command that you need to use when you want to switch from your working directory and view other files. To use this command, yu need to know the pathname for the working directory that you want to view. There are two different types of pathnames for you to discern. There is the absolute pathname and the relative pathname.

The absolute pathname is one that starts at your root directory, and by following a file path, it will easily lead you to your desired directory. Suppose your absolute pathname for a directory is /usr/bin. The directory is known as usr and there is another directory within it using the name bin. If you want to use the cd command to access your absolute pathname, you should type in the following command: -

[name@mylinux me] $cd/user/bin

[name@mylinux me] $pwd

/usr/bin [name@mylinux me] $ls

When you enter this information, you would have succeeded in changing your working directory to /usr/bin.

You can use a relative pathname when you want to change the new working directory which is /usr/bin to the parent directory, which would be /usr. To execute this, you should type in the following prompt: -

[name@mylinux me] $cd ..

[name@mylinux me] $pwd

/usr

Using a relative pathway cuts down on the amount of typing that you must do when using command lines, therefore, it is recommended that you learn as many of these as possible.

When you want to access a file using Linux command prompts, you should take note that they are case sensitive. Unlike other files which you would find on Windows Operating Systems and programs, the files in Linux do not have file extensions. This is great because it gives you the flexibility of labeling the files anything that you like. One thing you need to be careful of are the application programs that you use. There are some that may

automatically create extensions on files, and it is these that you need to be careful and watch out for.

Chapter 5 Introduction to Linux Terminals

Working with the Linux operating system requires you to have knowledge about the terminal and the command line. It is essential to know what these things are, to be at least slightly familiar with their usage and the standard commands available.

What is a Terminal?

A terminal is described as "a program that emulates a video terminal within some other display architecture. Though typically synonymous with a *shell* or text terminal, the term *terminal* covers all remote terminals, including graphical interfaces. A terminal emulator inside a graphical user interface is often called a terminal window".

To be precise, a terminal is simply the outside. Inside a terminal runs a command line interpreter that is called a shell.

Debian Linux supports a long list of terminal software. This includes the Aterm, as well as the GNOME terminal, the Kterm, the Mate Terminal, the Rxvt, the Xterm and the Xvt. These different implementations of terminal software vary in terms of stability, support for character sets, design, colors and fonts, as well as the possibility to apply background images or work with transparency

In this book we will use the GNOME terminal because of its stability, simplicity and adjustability. In order to increase and decrease the size of the content that is displayed inside the terminal window; use the two-key combinations CTRL+ and CTRL-.

6.2 What is a Shell?

Simply speaking, a shell is a sophisticated command-line interpreter. In a loop the shell reads characters, modifies them under certain conditions, and executes the result.

Under certain conditions, between reading from the command-line and the execution of the actual result, the shell must interpret special characters that are part of your input. The table below displays the special characters that are available:

Character	Meaning

$name	substitution of a variable
name=	assignment of a variable
'command'	substitution of a command
$(command)	substitution of a command
< > >> 2> 2>>	redirection of input and output
|	pipelining of commands
" ' \	quoting
* [ ] ?	creation of file names (globbing)
;	separation of commands
&	run the command in the background
|| &&	run the commands under certain conditions
{ } ( )	cramp commands

For interest sake, the shell divides the command-line into single words in order to determine the commands, by means of the following process:

Step	Description	Special Characters
1	read until the command separator and tagging (substitution of commands)	\n & | || && ; $(...) \ "..." '..."
2	tagging (input-/output redirection, assignment of values)	< > >> 2> 2>> name=
3	division into single words	space, tabulator
4	substitution of variables	$name
5	substitution of commands	tagging, see step 1 and 2
6	input-/output redirection	tagging, see step 1 and 2
7	assignment of variables	

		tagging, see step 1 and 2
8	division into words	based on IFS
9	creation of file names	* [-]
10	command execution	

In steps 1 and 2 tagging happens for steps 5 to 7 only, and no further action takes place. Steps 5 to 7 are executed if tagging is completed. Next, the shell removes all the special characters. In step 10 the command-line contains only the arguments that are needed to execute the command. The shell interprets the first word as the name of the command to be executed, and the remaining words are its arguments:

```
command  arg1  arg2  arg3
...
```

6.3 Available Shells

Your Linux system allows the usage of various shells. Each shell is available as a separate software package through the Debian package manager, Aptitude, we installed earlier. The list of shells is quite long, so we list only a selection of the available shells that are the most popular:

- Almquist Shell (ash)
- Bourne Again Shell (bash)
- Debian Almquist Shell (dash)
- Z Shell (zsh)
- C Shell (csh)
- Korn Shell (ksh)
- Tenex C Shell (tcsh)

Unless otherwise stated the examples in this document are based on the Bourne Again Shell (bash). At the time of writing this document this is the default shell on Debian.

Which other shells are allowed in your Linux system, is set up in the configuration file */etc/shells* . Using the *cat* command you can see the list of allowed shells:

```
user@debian95:~$ cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/dash
/bin/bash
/bin/rbash
/usr/bin/screen
user@debian95:~$
```

The name of the shell that is currently in use in your terminal is kept in the shell variable *$0* . The following simple command outputs the shell's name:

```
user@debian95:~$ echo $0
bash
user@debian95:~$
```

In order to change the shell currently in use, have a look at the *chsh* command and the configuration file */etc/passwd* covered in the following section.

Chapter 6 Variables

In Linux, creation of variables is very easy. For example, in order to store a name, John into a variable name, you can do something like what's being shown below:

[root@archserver ~]# name="John"

The double quotation marks tell Linux that you are creating a variable which will hold string typed value: John. If your string contains only one word then you can ignore the quotation marks, but if you are storing a phrase that contains more than one word and whitespaces than you must use the double quotation marks. To see the value inside any variable, you must use the dollar sign ($) before mentioning the name of the variable in the echo command. Like this:

[root@archserver ~]# echo $name

John

If you miss the dollar sign ($), echo will treat the argument passed to it as a string and will print that string for example:

[root@archserver ~]# echo name

name

You should keep in mind that there should not be any white spaces present between the identifier of the variable and its value. An identifier is basically the name or signature of a variable:

[root@archserver ~]# x=5 # This syntax is correct because there aren't any whitespaces

[root@archserver ~]# x = 10 # This syntax is incorrect because whitespaces are

#present between the variable name and its value

If you want to store some value inside a file whilst using the echo command, you could do something like this:

[root@archserver NewFolder]# echo name > new.txt

[root@archserver NewFolder]# cat new.txt

name

In the example above, I am storing a string name into a file that I created. After storing the text in the file, I printed it on the terminal and got exactly what I stored in the text file. In the following set of commands, I am using double >> signs to append new text in the existing file.

[root@archserver NewFolder]# echo "is something that people use to recognize you!" >> new.txt

[root@archserver NewFolder]# cat new.txt

name

is something that people use to recognize you!

You can also create and print two variables with a single line of command each, respectively.

Example:

[root@archserver ~]# x=1; y=2

[root@archserver ~]# echo -e "$x\t$y"

12

[root@archserver ~]# echo -e "$x\n$y"

1

2

The flag *–e* tells Linux that I am going to use an escape character whilst printing the values of my variable. The first *echo* command in the example above contains the *\t* escape character, which means that a *tab of space* should be added whilst printing the values of variables passed. The second echo command also contains an escape character of new line: *\n* . This escape character will print a new line between the values of two variables, as it is shown in the example.

There are other escape sequences present in Linux terminal as well. For example, in order to print a back slash as part of your string value, you must use double back slash in your echo command:

[root@archserver ~]# echo -e "$x\\$y"

1\2

There are other variables present in the Linux too, there variables store some values that come in handy whilst using any distribution of Linux. These predefined variables are often referred to as global variables. For example, $HOME is one of those global variable. The $HOME variable stores the path of our default directory, which in our case is the HOME folder. We can see the path stored in the $HOME folder using the echo command:

[root@archserver ~]# echo $HOME

/root

We can also change the values of these global variables, using the same method that I used to store Value into a newly created variables. For now, I would ask you not to try that, as these kind of things only concern expert Linux users, which you are not right now, but soon you will be. Other global variables are:

1. PATH
2. PS1
3. *TMPDIR*
4. EDITOR
5. DISPLAY

Try echoing there values, but don’t change them, as they will affect the working of your Linux Installation:

[root@archserver ~]# echo $PS1

[\u@\h \W]\$

[root@archserver ~]# echo $EDITOR

[root@archserver ~]# echo $DISPLAY

:1

[root@archserver ~]# echo $TMPDIR

The most important global variable of all is the $PATH variable. The $PATH variable contains the directories / locations of all the programs that you can use from any directory. $PATH is like the environment variables present in the WINDOWS operating system. Both hold the directory paths to the programs. Let's print the $PATH variable. Our outputs might differ so don't worry if you see something different:

Example:

[root@archserver ~]# echo $PATH

Output:

/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/bin/site_perl:/usr/bin/vendor_perl:/usr/bin/core_perl

The output of the example above shows the path where Linux can find files related to *site_perl* , *vendor_perl* or *core_perl* . You can add values to the path variable too. But again, at this stage you shouldn't change any value present in the $PATH variable.

If you want to see where the commands that you use, reside in the directory structure of Linux, you should use the *which* command. It will print out the directory from where Linux is getting the definition of a command passed.

Example:

[root@archserver ~]# which ls

/usr/bin/ls

[root@archserver ~]# which pwd

/usr/bin/pwd

[root@archserver ~]# which cd

which: no cd in (/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/bin/site_perl:/usr/bin/vendor_perl:/usr/bin/core_perl)

[root@archserver ~]# which mv

/usr/bin/mv

Chapter 7 Using the Shell

There are times that you will notice that something is not working right while you are in a GUI desktop environment – there are times wherein a program crashes and the entire system refuses to respond to mouse clicks. There are even situations wherein the GUI may not start at all. When you run into trouble, you can still tell your operating system what to do, but you will have to do it using a text screen or the shell, which serves as the command interpreter for Linux.

Since Linux is essentially a Unix program, you will be doing a lot of tasks using the text terminal. Although the desktop will be doing the job of providing you the convenience to access anything through a click of the mouse, there are several occasions wherein you need to access the terminal.

Learning how to enter commands will save you from a lot of trouble when you encounter an X Window system error, which is the program that controls all the menus and the windows that you see in your desktop GUI. To fix this error, or to prevent it from stopping you to access the program or file that you want, you can pull up a terminal and enter a command instead. In the future, you might want to keep a terminal open in your desktop since it can make you order your computer faster than having to point and click.

The Bash Shell

If you have used the MS-DOS OS in the past, then you are familiar with command.com, which serves as the command interpreter for DOS. In Linux, this is called the shell. The default shell in all the different distros, is called the bash.

Bourne-Again Shell, or bash, can run any program that you have stored in your computer as an executable file. It can also run shell scripts, or a text files that are made up of Linux commands. In short, this shell serves as a command interpreter, or a program that interprets anything that you type as a command and performs what this input is supposed to do.

Pulling up the terminal window can be as simple as clicking on a monitor-looking icon on the desktop – clicking on it will lead you to a prompt. If you can’t find that icon, simply search through the Main menu and select the item with has the Terminal or Console label.

Tip: You have the choice to use other shells apart from the bash, just like you have a choice in choosing desktops. You can always change the shell

that you are using for your distro by entering the chsh command on the terminal.

The Shell Command

Every shell command follows this format:

A command line, such as a command that follows the above format, is typically followed by parameters (also known as arguments). When entering a command line, you need to enter a space to separate the main command from the options and to separate one option from another. However, if you want to use an option that contains a space in its syntax, you will need to enclose that option in quotation marks. Look at this example:

The grep command allowed you to find for a text in a file, which is Emmett Dulaney in this case. Once you press enter, you will get the following result:

If you want to read a file, you can use the “more” command. Try entering this command:

You will be getting a result that appears like this:

To see all programs are running on your computer, use the “ps” command. Try entering this command on the terminal:

The options ax (option a lists all running processes, while opion x shows the rest of the proceses) allows you to see all the processes that are available in your system, which looks like this:

```
PID TTY STAT TIME COMMAND
1 ? S 0:01 init [5]
2 ? SN 0:00 [ksoftirqd/0]
3 ? S< 0:00 [events/0]
4 ? S< 0:00 [khelper]
9 ? S< 0:00 [kthread]
22 ? S< 0:00 [kblockd/0]
58 ? S 0:00 [kapmd]
79 ? S 0:00 [pdflush]
80 ? S 0:00 [pdflush]
82 ? S< 0:00 [aio/0]
. . . lines deleted . . .
5325 ? Ss 0:00 /opt/kde3/bin/kdm
5502 ? S 0:12 /usr/X11R6/bin/X -br -nolisten tcp :0 vt7 -auth
    /var/lib/xdm/authdir/authfiles/A:0-p1AOrt
5503 ? S 0:00 -:0
6187 ? Ss 0:00 /sbin/portmap
6358 ? Ss 0:00 /bin/sh /usr/X11R6/bin/kde
6566 ? Ss 0:00 /usr/sbin/cupsd
6577 ? Ssl 0:00 /usr/sbin/nscd
. . . lines deleted . . .
```

The amount of the command-line options and their corresponding formats would depend on the actual command. These options appear like the –X, wherein X represents one character. For esampe, you can opt to use the option –l for the ls command, which will list a directory’s contents. Look at

what happens when you enter the command ls –l in the home directory for a user:

If you enter a command that is too long to be contained on a single line, press the \ (backslash) key and then hit Enter. Afterwards, go on with the rest of the command on the following line. Try typing the following command and hit Enter when you type a line:

This will display all the contents inside the /ets/passwd file.

You can also string together (also known as concatenate) different short commands on one line by separating these commands with the ; (semicolon) symbol. Look at this command:

This command will make you jump to your user's home directory, show the contents of the directory you shanged into, and then display the name of the current directory.

Putting Together Your Shell Commands

If you are aiming to make a more sophisticated command, such as finding out whether you have a file named sbpcd in the /dev directory because you need that file for your CD drive, you can opt to combine different commands to make the entire process shorter. What you can do is that you can enter the ls /dev command to show the contents of the /dev directory and see if it contains the file that you want.

However, you may also get too many entries in the /dev directory when the command returns with the results. However, you can combine the grep command, which you have learned earlier, with the ls command and search for the exact file that you are looking for. Now, type in the following command:

```
ls /dev | grep sbpcd
```

This will show you the directory listing (result of the ls command) while the grep command searches for the string "sbpcd". The pipe (|) serves as the connection between the two separate commands that you use, wherein the first command's output is used as the input for the second one.

Chapter 8 Nested If Statements

While we're still on the topic of indenting, this is one great example showing how it can make life simple for you. Your script may have many 'if' statements as there can be; you can also have an "if" statement within another "statement".

For instance, you may want to analyze the number given on the command line as follows:

Let's break that down:

In the fourth line, it's clear that you only perform the following if the first command line argument is more than 100.

In the eighth line, we see a little variation on the 'if' statement. If you want to check any expression, you may utilize the double brackets.

In the tenth line, both 'if' statements must be true for it to get run.

Tip

Generally, while you can nest as many if statements as you want, you may want to consider reorganizing your logic if you need to nest more than 3 levels deep.

If Else

You'd agree with me that there are times when you would want to perform a particular action if a statement is true, and another set of actions if it is false. You can fit this in with the 'else' mechanism.

Look:

You could now easily read from a file if it is delivered as a command line argument, 'else' read from "stdin" as illustrated below.

Note: Stdin (standard input) is an input stream where data is sent to a program and read by the program. This will be discussed more shortly.

If Elif Else

We also have times when we have a series of conditions that may lead to different paths.

Look:

This may be the case in this example:

You are 18 years or above, so you may go to a party; if you are not, but have an okay from your guardians, you may go but must be back by midnight. Otherwise, you can’t attend the party.

This is how it looks:

You can have as many ‘elif’ branches as you want; the final else is also optional.

Chapter 9 Boolean Operators

Now, what if you only want to do something if multiple conditions are met? Indeed, there are times when you'll want to perform the action if one or more conditions are met. This is where Boolean operators come in (to accommodate these).

They include:

- And (&&)
- Or (||)

As an example, consider that you want to perform an operation if your file is both readable and its size is greater than zero:

Perhaps you now want to do something that is a bit different if the user is either Andy or Bob:

Case Statements

You may also want to take a different path based upon a variable corresponding to series of patterns. You can obviously use a series of 'elif' and 'if' statements but that would soon become unwieldy. Luckily, we can use a 'case' statement to make things cleaner.

Look at the following examples to understand what I'm talking about:

Here's a more basic example:

Now let me break that down for you:

In the fourth line, we see that the line starts the mechanism 'case'.

In the fifth line, 'start' is equal to $1; you then perform the actions that follow. ')' denotes the end of that pattern.

In the seventh line, the end of this set of statements is identified with a double semi-colon (;;). The next case to consider follows this.

In the fourteenth line, you must note that the test for each case is a pattern. '*' denotes any number of any character. It essentially is a catch-all in the instance no other cases match. It is often used even though it is not necessary.

In the last line, we see esac. It is 'case backwards'; it shows you that you are at the end of the case statement. All other statements after this one are

executed normally.

Let’s now look at one slightly complex case where the use of patterns is slightly more.

Activity: make your own decisions

1. Build a Bash script that takes two numbers as command line arguments.
2. Create a script that accepts a file as a command line argument and then analyze it. For instance, you could check whether the file is writable or executable. A message should be printed if true, and another one if false.
3. Now create a script that prints a message according to the day of the week it is. For instance, ‘Happy surf day’ for Saturday, ‘TGIF for Sunday’ and so on.

Chapter 10 Working with Linux Files and Directories

Now that you've got your hands a little dirty, you must be eager to get on to some more coding and really get down to playing about in the system. You will, I promise, but first, there is some more theory that you really need to learn so that, when you do get into the system, you truly understand why it does what it does and how to do more with the commands you have learned. So, let's take a deeper look at the core concepts you need to learn.

Everything in Linux is a file.

This is a very important concept to grasp—everything in Linux, no matter where it's or what it does, is a file. Text files are, obviously, files. Directories are files. Your computer keyboard is a read-only file, the monitor on your computer—even this is a file, albeit a write-to file. Everything. To start with, this won't have any real effect on what you do, but you must always keep it in mind as it will help you to grasp how Linux works with directory and file management.

Linux is known as an "extensionless system."

This one is a little bit harder to grasp, but as you work through this book and learn more, you'll find that it makes sense. File extensions are usually made up of a series of 2, 3, or 4 characters after a dot. You've already seen these with Word (.doc, .docx), .pdf, and so on. This extension is what tells you the type of file you are working with. These are some of the more common ones that you'll come across:

- *File.exe* : a program or a file that's executable
- *File.txt* : a file in plain text
- **File.png, file.jpg, file.gif*: all image files***

In many systems, like Windows, this file extension is very important because the system will use it to work out what kind of file it's working with. In Linux, it's different. The Linux system will ignore that extension and, instead, it will investigate the file itself to see what it's to do. So, for example, you could have a file that's named monkey.gif, a picture of a monkey. You could give the file a new name, such as monkey.txt or even just monkey, and Linux will see it as the image file it really is. Because of this, you might sometimes find it difficult to determine what type a file is,

but there is a really easy way to find out in Linux by using one simple command: file.

file [path]

Now, you are probably asking yourself why the command line argument is a path and not a file. If you think back, remember that every directory or file that we specify on the command line is a path. And, because a directory is also a file, it's correct to say that the path is nothing more than a journey to a system location, and the location is the file.

Linux is case sensitive.

This is one of the most important concepts and is the one that tends to cause the most problems for those new to the Linux program. You already know that systems like Windows are case insensitive in terms of file referencing, but Linux is different. You can have multiple directories or files with identical names but different cases. For example:

ls Documents

FILE2.txt File2.txt file2.TXT

...

file Documents/file2.txt

Documents/file2.txt: ERROR: cannot open 'file2.txt' (No such file or directory)

Each one of these is seen as an individual file by Linux.

Case sensitivity also comes into command line options. For example, when you use the command ls, there are two distinct options: ls and lS, and both do something different. One of the more common errors is to enter an option in lowercase when it should be uppercase, and then wonder why your program doesn't do what it should.

Spaces in file or directory names

While it's perfectly okay to have spaces in your directory or file names, you must exercise some small measure of caution. When we want to separate several items on the command line, we use a space between each one. The spaces are what tell us what the program name is and how to identify the different arguments on the command line. For example, if we wanted to go

into a directory that we called Summer Holiday, this next example would not work:

ls Documents

FILE2.txt File2.txt file2.TXT Summer Holiday

...

cd Summer Holiday

bash: cd: Holiday: No such file or directory

Why not? Well, Summer Holiday is seen by Linux as 2 separate command line arguments, so the command, cd, will go into the first one only. The only way to stop this from happening is to tell the terminal that Summer Holiday is just one argument and there are two ways to do this, both of which are perfectly valid.

Quotes

The first way involves surrounding the whole item with quote marks. You can use singles or doubles (although there is a small difference between the two), but do make sure that you use the same to open as you do to close the quote marks. For example, don't open with a single quote and then close with a double. Anything that goes inside those quotes is then seen as one item, for example:

cd 'Summer Holiday'

pwd

/home/cleopatra/Documents/Summer Holiday

Escape characters

The other way is using an escape character, the backslash (\). The escape character will nullify or escape the special meaning attached to the following character, for example:

cd Holiday\ Photos

pwd

/home/cleopatra/Documents/Summer Holiday

As you can see from this example, the space that separates Summer and Holiday would usually have a special meaning: to separate each work as a separate command line argument. By using the escape character, we removed that meaning and the two words are one argument.

Earlier we talked about something called TAB completion. If you were to use this before you get to a space in the name of a directory, the terminal escapes the spaces automatically.

Hidden directories and files

Linux has a very nice way of specifying whether a directory or a file is hidden or not. If the name starts with a. (a followed by a full stop), it's a hidden directory or file. There are no requirements for special commands or special actions to hide them.

There are several reasons why you may want to hide a directory or a file, and one of those is if they relate to the configuration for a specific user. These can be hidden so they don't interfere with everyday tasks that the user may be doing.

So, if you want to hide a file or a directory, simply create it or rename it with the name beginning with a. In the same way, if you wanted to unhide a directory or file, you would simply rename it, removing the a. from the start of the name. The command that you learned to list directories and files will NOT show those that are hidden. Instead, you can modify the command to add in -a as a command line option, allowing those hidden files and directories to be shown.

ls Documents

FILE2.txt File2.txt file2.TXT

...

ls -a Documents

. .. FILE2.txt File2.txt file2.TXT .hidden .file.txt

...

In this example, you can see that, when the directories and files were listed, the first two were . and then .. and if you need to brush up on those head back to the section on paths.

Summary

Here's a quick summary of what you learned in this section:

- "File" is used to get information about the type of a directory or file.
- "ls –a" lists all files in a directory, including those that are hidden.
- Everything is a file in Linux, including directories.
- File extensions are not important because Linux will investigate the file to see what it is.
- Linux is a case-sensitive language, so watch out for capitalization.

Practical work

Now we need to put it all into practice, so try these:

- Run the command file with several different entries and ensure that you use a combination of absolute and relative paths.
- Next, give a command that will show everything that's in your home directory, including any directories or files that are hidden.

More on running commands

Much of understanding Linux is down to knowing the right command line options to use to change how your commands work, based on what you are doing with them. Many of these will have a long and shorthand version; for example, we saw earlier that, when we want to list everything in a directory, including hidden entries, we use -a or --all. The longhand version is just a version that's easily read by the human eye. You can use either of these because they both have the same outcome. The advantage to longhand is that, when you read back over your code, you'll find it easier to understand what the commands do, while using shorthand allows you to string several commands together easily.

pwd

/home/cleopatra

ls -a

ls --all

ls -alh

So, you can see that the longhand options each start with a pair of dashes (--), while the shorthand options all start with a single dash (-). When you use the single dash, you can invoke several command options by adding all the letters that represent those options after a single dash. To see what the last option is doing, look up the main ls page.

Man pages

These are the commands for looking up main manual pages:

- *man <command>* : looks up the manual page for a command
- *man -k <search term>* : use a keyword to search for all pages that have the keyword in them
- */<term>* : used in a manual page to search for the word "term"
- *n:* after the search is done on a page, choose the next item found
- Remember: man pages are useful so use them as often as you need to. You don't need to try and remember everything when you have these at your disposal.

Practical work

Try the following to get some practice:

Look through the manual page for the ls command. Play around with some of the options you see there and try some of them as combinations. Don't forget to use a variety of relative and absolute paths with the ls command.

Now have a go at some searches in the man pages. Depending on what you search for, you could end up with a few large lists. Look at some of the pages to get an idea of what they are like.

Removing a directory

As you have seen, it's easy to create a new directory and it's just as easy to remove or delete a directory too. There is one thing you should be aware of, though—the Linux command line doesn't contain any UNDO options. This means being very careful about what you are doing because once you delete

a directory, it’s gone forever. To remove a directory, we use a simple command of rmdir, which is shorthand for remove directory.

rmdir [options] <Directory>

There are two more things that you need to note here. First, in a similar way to how mkdir (make directory) supports the options -v and -p, rmdir also supports those options. Second, before you can remove a directory, it must be empty. However, there is a way around this, and we will look at that later.

rmdir linuxpractice/foo/bar

ls linuxpractice/foo

Creating a blank file

There are quite a few commands that involve data manipulation in files that, if you refer to a file that doesn’t exist, will create that file for you. We can use this to create a blank file by using the touch command.

touch [options] <filename>

pwd

/home/cleopatra/linuxpractice

ls

foo

touch example1

ls

example1 foo

We can use the touch command to modify the times for file access and modification. This isn’t something you would need to do unless you are testing a system with a reliance on these access and modification times. Basically, what happens here—and this is something you make good use of—is that, when you touch a file that doesn’t exist, it will be created for you.

There are a lot of things that are not directly done in Linux but, if you learn how certain commands behave and learn how to be creative with them, you can achieve what you want. We talked about how the command line can give you a range of building blocks and that these can be used in any way

that you want, but you can only be effective if you fully understand how they work, not just why.

Right now, all we have is a blank file but soon, we will look at how we can put data into a file and how to extract data as well.

Copying a file or directory

There are a few reasons why you might want to make a copy of a directory or file—perhaps you want to make a change to something, so you would make a copy of the original. That way, if anything goes wrong, you can easily go back to what it was. To do this we use the command cp, which means copy.

cp [options] <source> <destination>

This command has a few options available to it, and we're going to talk about one of them in a minute. First though, check out the cp man page to see what other options are available.

ls

example1 foo

cp example1 sammy

ls

sammy example1 foo

Did you spot that the source and the destination are both paths? What this means is that we can use absolute and relative paths to refer to them. Have a look at these few examples:

cp /home/cleopatra/linuxpractice/example2 example3

cp example2 ../../backups

cp example2 ../../backups/example4

cp /home/cleopatra/linuxpractice/example2 /otherdir/foo/example5

So, when cp is used, the destination may be a path that goes to a directory or to a file. If it goes to a file, as in the first, third and fourth example lines above, a copy of the source will be created, and it will have the filename that was specified in the destination path. If we were going to a directory,

then the file would be copied to the directory and will be named the same as the source.

By default, cp can only copy a file so, if you wanted to copy a directory, you would need to use the option -r, which means recursive. Recursive means looking into a directory, at the files and the directories contained in it. To see the subdirectories, you do the same but from within each directory.

ls

sammy example1 foo

cp foo foo2

cp: omitting directory 'foo'

cp -r foo foo2

ls

sammy example1 foo foo2

In this example, we're copying all the files and the directories from the foo directory to foo2.

Moving a file or directory

Moving a file is done quite simply with the mv command, which means move. It works in much the same way as cp, except that we can use it without -r for moving directories.

mv [options] <source> <destination>

ls

sammy example1 foo foo2

mkdir backups

mv foo2 backups/foo3

mv sammy backups/

ls

backups example1 foo

Let’s look at that in more detail:

Line 3 : a new directory has been created with the name backups.

Line 4 : the directory called foo2 was moved into the directory called backups and was renamed as foo3.

Line 7 : the file called sammy was moved into the backups directory. It retained the same name because we didn’t give a destination name.

Note that, once again, we used paths for source and destination and these are absolute or relative paths.

Renaming files and directories

Just as we did with the touch command, the behavior of mv can also be used creatively to give us a different outcome. Normally, we use mv to move files or directories into a new directory. Part of that move includes renaming that file or directory. If we were to specify that the destination and the source are the same directory but named differently, that would be a creative use of mv for renaming directories or files.

ls

backups example1 foo

mv foo foo3

ls

backups example1 foo3

cd ..

mkdir linuxpractice/testdir

mv linuxpractice/testdir /home/cleopatra/linuxpractice/frieda

ls linuxpractice

backups example1 foo3 frieda

Let’s delve into this one:

Line 3 : the file called foo was given a new name of foo3. Both paths are relative.

Line 6 : the parent directory was moved. We did this so that in the next line, we could show how to run commands on a file or a directory from outside the directory they are contained in.

Line 8 : the directory called testdir was renamed to frieda. We used a relative path to the source and an absolute path to the destination.

Removing files

Like we saw with rmdir, when you remove a file, you cannot undo it, so take care with what you are doing. The command for removing or deleting files is rm, which means remove.

rm [options] <file>

ls

backups example1 foo3 frieda

rm example1

ls

backups foo3 frieda

Removing directories that aren’t empty

The command rm can also be altered by several different options. Again, check out the rm man page to see what can be done, but one of the most useful options is -r. Like cp, it means recursive, and we use it with rm to remove specified directories and everything that’s contained inside them.

ls

backups foo3 frieda

rmdir backups

rmdir: failed to remove 'backups': Directory not empty

rm backups

rm: cannot remove 'backups': Is a directory

rm -r backups

ls

foo3 frieda

Another option you can use with r is i, which means interactive. This will check with you before any file or directory is removed and provide you with an option to change your mind and cancel out the command.

One last note

I have already said this a few times, and I will keep on saying it: when we refer, on the command line, to files or directories, they are paths. They may be absolute paths or relative paths, and this is going to be the case pretty much all the time. I won't remind you of this again, and the examples I use will not show this, so please remember it. Don't forget to experiment with relative and absolute paths in the commands that we use, as each will provide a different output.

Summary

This is some of what you learned in this section:

- *Mkdir* : stands for Make Directory and allows us to create new directories
- *Rmdir* : stands for Remove Directory and allows us to delete directories
- *Touch* : allows us to create blank files
- *Cp* : stands for Copy and allows us to copy directors or files
- *Mv* : stands for Move and allows us to move or rename files or directories
- *Rm* : stands for Remove and allows us to delete files
- There is no option to undo anything you do, so delete and move files and directories with care.
- Command line options: there are plenty of them, so use the man pages of each command to familiarize yourself with what there is and what you can do

Practical work

You now have several different commands that will let you interact with your system, so let's practice with them. Try the following:

- Create a brand new directory within your home directory (this will allow you to experiment).
- Inside the directory create some files and some directories and create more files and directories inside of each of those.

- Now give some of them new names.
- Go to the home directory and copy one file from a subdirectory into the first directory that you created.
- Move the file to another directory.
- Rename a few more files.
- Move one file and rename it at the same time.
- Lastly, look at the directories that are already in the home directory. Most likely you have Downloads, Documents, Images, and Music. Have a think about other directories that can help you to keep things organized and start to set them up.

Chapter 11 Log Analysis

In this chapter, we will learn how to locate logs in the Linux system and interpret them for system administration and troubleshooting purposes. We will describe the basic architecture of syslog in Linux systems and learn to maintain synchronization and accuracy for the time zone configuration such that timestamps in the system logs are correct.

Architecture of System Logs

In this section, we will learn about the architecture of system logs in Red Hat Enterprise Linux 7 system.

System logging

Events that take place as a result of processes running in the system and the kernel of the operating system need to be logged. The logs will help in system audits and to troubleshoot issues that are faced in the system. As a convention, all the logs in Linux based systems are stored at */var/log* directory path.

Red Hat Enterprise Linux 7 has a system built for standard logging by default. This logging system is used by many programs. There are two services *systemd-journald* and *rsyslog, which* handle logging in Red Hat Enterprise Linux 7.

The *systemd-journald* collects and stores logs for a series of process, which are listed below.

- Kernel
- Early stages of the boot process
- Syslog
- Standard output and errors of various daemons when they are in the running state

All these activities are logged in a structural pattern. Therefore, all these events get logged in a centrally managed database. All messages relevant to syslog are also forwarded by *systemd-journald* to *rsyslog* to be processed further.

The messages are then sorted by rsyslog based on facility or type and priority and then writes them to persistent files in */var/log* directory.

Let us go through all the types of logs, which are stored in /var/log based on the system and services.

Log file	*Purpose*
/var/log/messages	*Most of the syslog messages are stored in this file except for messages related to email processing and authentication, cron jobs and debugging related errors*
/var/log/secure	*Errors related to authentication and security are stored in this file*
/var/log/maillog	*Mail server related logs are stored in this file*
/var/log/cron	*Periodically executed tasks known are known as cron. Related logs are stored in this file*
/var/log/boot.log	*Messages that are associated with boot up are stored here*

Syslog File Review

In this section, we will learn how to review system logs, which can help a system admin to troubleshoot system related issues.

Syslog files

The syslog protocol is used by many programs in the system to log their events. The log message is categorized by two things.

- Facility, which is the type of message
- Priority, which is the severity of the message

Let us go through an overview of the priorities one by one.

Code	*Priority*	Severity
0	*emerg*	The state of the system is unusable
1	*alert*	Immediate action needs to be taken
2	*crit*	The condition is critical
3	*err*	The condition is non-critical with errors
4	*warning*	There is a warning condition
5	*notice*	The event is normal but significant
6	*info*	There is an informational event
7	*debug*	The message is debugging-level

The method to handle these log messages is determined by the priority and the type of the message by rsyslog. This is already configured in the file at /etc/rsyslog.conf and by other conf files in /etc/rsyslog.d. As a system admin, you can overwrite this default configuration and customize the way rsyslog file to be able to handle these log messages as per your requirement. A message that has been handled by the rsyslog service can show up in many different log files. You can prevent this by changing the severity field to none so that messages directed to this service will not append to the specified log file.

Log file rotation

There is *logrotate* utility in place in Red Hat Enterprise Linux 7 and other linux variants so that log files do not keep piling up the /var/log file system and exhaust the disk space. The log file gets appended with the date of rotation when it is rotated. For example, and old file named /var/log/message will change to /var/log/messages-20161023 if the file was rotated on October 23, 2016. A new log file is created after the old log file is rotated and it is notified to the relevant service. The old log file is usually discarded after a few days, which is four weeks by default. This is done to free up disk space. There is a cron job in place to rotate the log files. Log files get rotated on a weekly basis, but this may vary based on the size of the log file and could be done faster or slower.

Syslog entry analysis

The system logs, which are logged by the rsyslog program have the oldest log message at the top of the file and the latest message at the end of the file. There is a standard format that is used to

maintain log entries that are logged by rsyslog. Let us go through the format of the /var/log/secure log file.

Feb 12 11:30:45localhostsshd[1432]Failed password for user from 172.25.0.11 port 59344 ssh2

- The first column shows the timestamp for the log entry
- The second column shows the host from, which the log message was generated
- The third column shows the program or process, which logged the event
- The final column shows the message that was sent

Using the tail command to monitor log files

It is a common practice for system admins to reproduce the issue so that error logs for the issue get generated in real time. The *tail -f /path/to/file* command can be used to monitor logs that are generated in real time. The last 10 lines of the log file are displayed with this command while it continues to print new error logs that are generated in real time. For example, if you wanted to look for real time logs of failed login attempts, you can use the following tail command, which will help you see real time logs.

[root@desktop ~]# tail -f /var/log/secure

…

Feb 12 11:30:45localhostsshd[1432]Failed password for user from 172.25.0.11 port 59344 ssh2

Using logger to send a syslog message

You can send messages to the rsyslog service by using the *logger* command. The command useful when you have made some changes to the configuration file of rsyslog and you want to test it. You can execute the following command, which will send a message that gets logged at /var/log/boot.log

[root@desktop ~]# logger -p local7.notice “Log entry created”

Reviewing Journal Entries for Systemd

In this section, we will learn to review the status of the system and troubleshoot problems by analyzing the logs in the systemd journal.

Using journalctl to find events:

The systemd journal uses a structured binary file to log data. Extra information about logged events is included in this data. For syslog events, this contains the severity and priority of the original message.

When you run *journalctl* as the root user, the complete system journal is shown, starting from the oldest log entry in the file.

Messages of priority notice or warning are highlighted in bold by the journalctl command. The higher priority messages are highlighted in red color.

You can use the journalctl command successfully to troubleshoot and audit is to limit the output of the command to only show relevant output.

Let us go through the various methods available to limit output of the journalctl command to show only desired output.

You can display the last 5 entries of the journal by using the following command.

[root@server ~]# journalctl -n 5

You can use the priority criteria to filter out journalctl output to help while troubleshooting issues. You can use the *-p* option with the journalctl command to specify a name or several the priority levels, which shows the entries that are of high level. journalctl know the priority levels such as info, debug, notice, err, warning, crit, emerg, and alert.

You can use the following command to achieve the above mentioned output.

[root@server ~]# journalctl -p err

There is command for journalctl like tail -f, which is *journalctl -f* . This will again list the last 10 lines of journal entry and then keep printing log entries in real time.

[root@server ~]# journalctl -f

You can also use some other filters to filter out journalctl entries as per your requirement. You can pass the following options of *--since* and *--until* to filters out journal entries as per timestamps. You need to then pass the arguments such as *today, yesterday* or an actual timestamp in the format *YYYY-MM-DD hh:mm:ss*

Let us look at a few examples below.

[root@server ~]# journalctl --since today

[root@server ~]# journalctl --since “2015-04-23 20:30:00” --until “2015-05-23 20:30:00”

There are more fields attached to the log entries, which will be visible only if you use the verbose option by using *verbose* with the journalctl command.

[root@server ~]# journalctl -o verbose

This will print out detailed journalctl entries. The following keywords are important for you to know as a system admin.

- _COMM, which is the name of the command
- _EXE show the executable path for the process
- _PID will show the PID of the process
- _UID will show the user associated with the process
- _SYSTEMD_UNIT show the systemd unit, which started the process

You can combine one or more of these options to get an output from the journalctl command as per your requirement. Let us have a look at the example below, which will print journal entries that contain the systemd unit file sshd.service bearing the PID 1183.

[root@server ~]# journalctl _SYSTEMD_UNIT=sshd.service _PID=1183

Systemd Journal Preservation

In this section, we will learn how to make changes to the *systemd-journald* configuration such that the journal is stored on the disk instead of memory.

Permanently storing the system journal

The system journal is kept at */run/log/journal* by default, which means that when the system reboots, the entries are cleared. The journal is a new implementation in Red Hat Enterprise Linux 7.

We can be sure that if we create a directory as */var/log/journal* , the journal entries can be logged there instead. This will give us an advantage that historical data will be available even after a reboot. However, even though we will have a journal that is persistent, we cannot have any data

that can be kept forever. There is a log rotation, which is triggered by a journal on a monthly basis. Also, by default, the journal is not allowed to have a dusk accumulation of more than 10% of the file system it occupies, or even leave less than 15% of the file system free. You can change these values as per your needs in the configuration file at */etc/systemd/journald.conf* and one the process for systemd-journald starts, the new values will come into effect and will be logged.

As discussed previously, the entries of the journal can be made permanent by creating a directory at /var/log/journal

[root@server ~]# mkdir /var/log/journal

You will need to make sure that the owner of the /var/log/journal directory is root and the group owner is systemd-journal, and the directory permission is set to 2755.

[root@server ~]# chown root:systemd-journal /var/log/journal

[root@server ~]# chmod 2755 /var/log/journal

For this to come into effect, you will need to reboot the system or as a root user, send a special signal *USR1* to the systemd-journald process.

[root@server ~]# killall -USR1 systemd-journald

This will make the systemd journal entries permanent even through system reboots, you can now use the command *journalctl -b* to show minimal output as per the latest boot.

[root@server ~]# journalctl -b

If you are investigating an issue related to system crash, you will need to filter out the journal output to show entries only before the system crash happened. That will ideally be the last reboot before the system crash. In such cases, you can combine the *-b* option with a negative number, which will indicate how many reboots to go back to limit the output. For example, to show outputs till the previous boot, you can use *journalctl -b -1* .

Maintaining time accuracy

In this section, we will learn how to make sure that the system time is accurate so that all the event logs that are logged in the log files show accurate timestamps.

Setting the local time zone and clock

If you want to analyze logs across multiple systems, it is important that the clock on all those systems is synchronized. The systems can fetch the correct time from the Internet using the Network Time Protocol NTP. There are publicly available NTP projects on the Internet like the Network pool Project, which will allow a system to fetch the correct time. The other option is to maintain a clock made up of high quality hardware to serve time to all the local systems.

To view the current settings for date and time on a Linux system, you can use the *timedatectl* command. This command will display information such as the current time, the NTP synchronization settings and the time zone.

[root@server ~]# timedatectl

The Red Hat Enterprise Linux 7 maintains a database with known time zones. It can be listed using the following command.

[root@server ~]# timedatectl list-timezones

The names of time zones are based on zoneinfo database that IANA maintains. The naming convention of time zones is based on the ocean or continent. This is followed by the largest city

in that time zone or region. For example, if we look at the Mountain Time in the USA, it is represented as “America/Denver”.

It is critical to select the correct name of the city because sometimes even regions within the same time zone may maintain different settings for daylight savings. For example, the US mountain state of Arizona does not have any implementation of daylight savings and therefore falls under the time zone of “America/Phoenix”.

The *tzselect* command is used to identify zone info time zone names if they are correct or not. The user will get question prompts about their current location and mostly gives the output for the correct time zone. While suggesting the time zone, it will not automatically make any changes to the current time on the system. Once you know, which timezone, you should be using, you can use the following command to display the same.

[root@server ~]# timedatectl set-timezone America/Phoenix

[root@server ~]# timedatectl

If you wish to change the current date and time for your system, you can use the *set-time* option with the timedatectl command. The time and date can be specified in the format “”YYYY-MM-DD hh:mm:ss”. If you just want to set the time, you can omit the date parameters.

[root@server ~]# timedatectl set-time 9:00:00

[root@server ~]# timedatectl

You can use the automatic time synchronization for Network Time Protocol using the *set-ntp* option with the timedatectl command. The argument to be passed along is *true* or *false*, *which* will turn the feature on or off.

[root@server ~]# timedatectl set-ntp true

The Chronyd Service

The local hardware clock of the system is usually inaccurate. The *chronyd* service is used to keep the local clock on track by synchronizing it with the configured Network Time Protocol NTP servers. If the network is not available it synchronizes the local clock to the RTC clock drift that is calculated and recorded in the *driftfile, which* is maintained in the configuration file at */etc/chrony.conf.*

The default behavior of the chronyd service is to use the clocks from the NTP network pool project to synchronize the time and no additional configuration is needed. It is advisable to change the NTP servers if your system happens to be on an isolated network.

There is something known as a *stratum* value, which is reported by an NTP time source. This is what determines the quality of the NTP time source. The stratum value refers to the number of hops required for the system to reach a high performance clock for reference. The source reference clock has a stratum value of 0. An NTP server that is attached to the source clock will have a stratum value of 1, while a system what is trying to synchronize with the NTP server will have a stratum value of 2.

You can use the */etc/chrony.conf* file to configure two types of time sources, *server* and *peer* . The stratum level of the server is one level above the local NTP server. The stratum level of the peer is the same as that of the local NTP server. You can specify one or more servers and peers in the configuration file, one per line.

For example, if your chronyd service synchronizes with the default NTP servers, you can make changes in the configuration file to change the NTP servers as per your need. Every time you change the source in the configuration file, you will need to restart the service for the change to take effect.

[root@server ~]# systemctl restart chronyd

The chronyd service has another service known as *chronyc, which* is a client to the chronyd service. Once you have set up the NTP synchronization, you may want to know if the system clock synchronizes correctly to the NTP server. You can use the *chronyc sources* command or if you want a more detailed output, you use the command *chronyc sources -v* with the verbose option.

[root@server ~]# chronyc sources –v

Chapter 12 Create a bootable stick

Once you have downloaded the *ISO file* , you need to create a *bootable stick* .

This means that you must prepare an USB-stick so that you can start Ubuntu with it.

For this task you need an appropriate program.

There are several applications that you can use for this task.

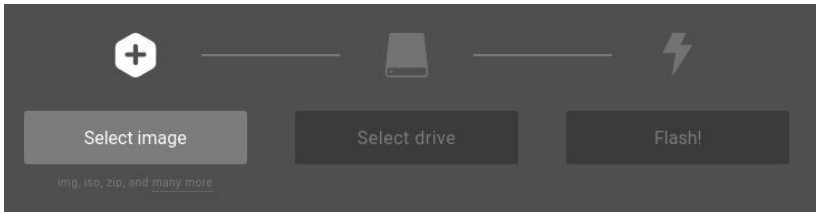
Etcher is a good choice in my opinion.

Etcher is free for Linux, Windows and MacOS.

HTTPS://WWW.BALENA.IO/ETCHER/

Download the appropriate version and then create the required stick.

Etcher is self-explanatory.



Select image (the ISO file for Ubuntu 18.04.)

Select drive (you will see a selection of sticks that you have plugged into a free USB port).

Choose the one you want to use for Ubuntu.

Flash *(Start!)*

Double check if the right stick is selected.

Any content on the stick will be completely erased!

Systemd Journal Preservation

In this section, we will learn how to make changes to the *systemd-journald* configuration such that the journal is stored on the disk instead of memory.

Permanently storing the system journal

The system journal is kept at */run/log/journal* by default, which means that when the system reboots, the entries are cleared. The journal is a new implementation in Red Hat Enterprise Linux 7.

We can be sure that if we create a directory as */var/log/journal* , the journal entries can be logged there instead. This will give us an advantage that historical data will be available even after a reboot. However, even though we will have a journal that is persistent, we cannot have any data that can be kept forever. There is a log rotation, which is triggered by a journal on a monthly basis. Also, by default, the journal is not allowed to have a dusk accumulation of more than 10% of the file system it occupies, or even leave less than 15% of the file system free. You can change these values as per your needs in the configuration file at */etc/systemd/journald.conf* and one the process for systemd-journald starts, the new values will come into effect and will be logged.

As discussed previously, the entries of the journal can be made permanent by creating a directory at /var/log/journal

[root@server ~]# mkdir /var/log/journal

You will need to make sure that the owner of the /var/log/journal directory is root and the group owner is systemd-journal, and the directory permission is set to 2755.

[root@server ~]# chown root:systemd-journal /var/log/journal

[root@server ~]# chmod 2755 /var/log/journal

For this to come into effect, you will need to reboot the system or as a root user, send a special signal *USR1* to the systemd-journald process.

[root@server ~]# killall -USR1 systemd-journald

This will make the systemd journal entries permanent even through system reboots, you can now use the command *journalctl -b* to show minimal output as per the latest boot.

[root@server ~]# journalctl -b

If you are investigating an issue related to system crash, you will need to filter out the journal output to show entries only before the system crash happened. That will ideally be the last reboot before the system crash. In such cases, you can combine the *-b* option with a negative number, which

will indicate how many reboots to go back to limit the output. For example, to show outputs till the previous boot, you can use *journalctl -b -1* .

Maintaining time accuracy

In this section, we will learn how to make sure that the system time is accurate so that all the event logs that are logged in the log files show accurate timestamps.

Setting the local time zone and clock

If you want to analyze logs across multiple systems, it is important that the clock on all those systems is synchronized. The systems can fetch the correct time from the Internet using the Network Time Protocol NTP. There are publicly available NTP projects on the Internet like the Network pool Project, which will allow a system to fetch the correct time. The other option is to maintain a clock made up of high quality hardware to serve time to all the local systems.

To view the current settings for date and time on a Linux system, you can use the *timedatectl* command. This command will display information such as the current time, the NTP synchronization settings and the time zone.

[root@server ~]# timedatectl

The Red Hat Enterprise Linux 7 maintains a database with known time zones. It can be listed using the following command.

[root@server ~]# timedatectl list-timezones

The names of time zones are based on zoneinfo database that IANA maintains. The naming convention of time zones is based on the ocean or continent. This is followed by the largest city in that time zone or region. For example, if we look at the Mountain Time in the USA, it is represented as “America/Denver”.

It is critical to select the correct name of the city because sometimes even regions within the same time zone may maintain different settings for daylight savings. For example, the US mountain state of Arizona does not have any implementation of daylight savings and therefore falls under the time zone of “America/Phoenix”.

The *tzselect* command is used to identify zone info time zone names if they are correct or not. The user will get question prompts about their current location and mostly gives the output for the correct time zone. While

suggesting the time zone, it will not automatically make any changes to the current time on the system. Once you know, which timezone, you should be using, you can use the following command to display the same.

[root@server ~]# timedatectl set-timezone America/Phoenix

[root@server ~]# timedatectl

If you wish to change the current date and time for your system, you can use the *set-time* option with the timedatectl command. The time and date can be specified in the format “”YYYY-MM-DD hh:mm:ss”. If you just want to set the time, you can omit the date parameters.

[root@server ~]# timedatectl set-time 9:00:00

[root@server ~]# timedatectl

You can use the automatic time synchronization for Network Time Protocol using the *set-ntp* option with the timedatectl command. The argument to be passed along is *true* or *false, which* will turn the feature on or off.

[root@server ~]# timedatectl set-ntp true

The Chronyd Service

The local hardware clock of the system is usually inaccurate. The *chronyd* service is used to keep the local clock on track by synchronizing it with the configured Network Time Protocol NTP servers. If the network is not available it synchronizes the local clock to the RTC clock drift that is calculated and recorded in the *driftfile, which* is maintained in the configuration file at */etc/chrony.conf.*

The default behavior of the chronyd service is to use the clocks from the NTP network pool project to synchronize the time and no additional configuration is needed. It is advisable to change the NTP servers if your system happens to be on an isolated network.

There is something known as a *stratum* value, which is reported by an NTP time source. This is what determines the quality of the NTP time source. The stratum value refers to the number of hops required for the system to reach a high performance clock for reference. The source reference clock has a stratum value of 0. An NTP server that is attached to the source clock will have a stratum value of 1, while a system what is trying to synchronize with the NTP server will have a stratum value of 2.

You can use the */etc/chrony.conf* file to configure two types of time sources, *server* and *peer* . The stratum level of the server is one level above the local NTP server. The stratum level of the peer is the same as that of the local NTP server. You can specify one or more servers and peers in the configuration file, one per line.

For example, if your chronyd service synchronizes with the default NTP servers, you can make changes in the configuration file to change the NTP servers as per your need. Every time you change the source in the configuration file, you will need to restart the service for the change to take effect.

[root@server ~]# systemctl restart chronyd

The chronyd service has another service known as *chronyc*, *which* is a client to the chronyd service. Once you have set up the NTP synchronization, you may want to know if the system clock synchronizes correctly to the NTP server. You can use the *chronyc sources* command or if you want a more detailed output, you use the command *chronyc sources -v* with the verbose option.

[root@server ~]# chronyc sources –v

Chapter 13 BONUS!

Sometimes your networking may not work as expected and you will need to troubleshoot the problem by testing the network connectivity.

The following commands will be covered in this section:

ping

traceroute

ping

The *ping* command is a basic troubleshooting command that can be used to verify network connectivity.

The following subsections will be covered in this section:

Loopback

Default Gateway

IP

Hostname

Loopback

If you *ping* the loopback address 127.0.0.1 it verifies that the *networking service* is working. In the example below the ping command is successful. Each line *64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.073 ms* indicates a successful ping. The *ping* command will continue until you press *Ctrl+C* .

In the example below the *ping* to the loopback address is unsuccessful as the *network service* is not running.

Default Gateway

On a network the *default gateway* is the router or device that leads out of the network usually eventually to the Internet. If your system cannot communicate with the *default gateway*, then it cannot access the Internet. The *default gateway* is normally the first useable IP address in a network but that is not a requirement. For example, if the network is 10.0.2.0/24 then the *default gateway* would be 10.0.2.1. In the example below the ping to 10.0.2.1 was successful meaning the system can communicate with the *default gateway* .

If you are unable to ping the *default gateway* it does not mean that there is anything wrong with your system as it could be a problem with the network.

IP

If you can ping the *default gateway* the next step is to ping something past the *default gateway* , preferably something on the Internet. In the example below, a ping to *8.8.8.8* is successful which means that everything is working, and the system can reach the Internet. The ping command used the *-c 3* option which cause the ping command to only do three pings and then stop vice running until pressing *Ctrl+C* .

Hostname

While pinging an IP address on the Internet will show you if your network connectivity is working it does not give you the entire picture. In order to be able to access web pages and other services by using a hostname requires DNS to be configured and working correctly.

traceroute

The *traceroute* command is used to trace the path of the IP packets from your system to another system. In the example below a the *traceroute* command is used to trace packets to *www.somewebsite.com* .

5.4 Name Resolution

Name Resolution is the process of converting hostnames to IP addresses. The way the Internet works today we mostly use hostnames and domains to connect to websites and other resources. Your system, however, needs to know the IP address of where you want to go in order to get there.

/etc/hosts

DNS

/etc/hosts

When the Internet was very young, you had to know the IP address of any site that you wanted to access. Then the *hosts* file was created and shared among the users of the Internet which worked because there were not many people on the Internet. The *hosts* file holds the manual mapping of hostnames to IP address. With the advent of DNS, the *hosts* file was no longer necessary but it still remains in Linux as a fallback *Name Resolution* method.

The following subsections are covered in this section:

Ubuntu

Fedora

Debian

Ubuntu

The example below shows the default *Ubuntu hosts* file.

If you want to add a custom IP to hostname mapping you must edit the */etc/hosts* file. The following is the syntax for entries:

ip_address hostname

Fedora

The example below shows the default *Fedora* hosts file.

If you want to add a custom IP to hostname mapping you must edit the */etc/hosts* file. The following is the syntax for entries:

ip_address hostname

Debian

The example below shows the default *Debian* hosts file.

If you want to add a custom IP to hostname mapping you must edit the */etc/hosts* file. The following is the syntax for entries:

ip_address hostname

DNS

The *Domain Name System (DNS)* is a system for naming systems on the Internet based on resource locators, hostnames, and domains and providing a mechanism to rapidly distribute updates when things change.

When you use a web browser to connect to a website you type if something like www.somewebsite.com which is known as a *Universal Resource Locator (URL)* . Starting from the right and moving left the *URL* starts with *.com* which is a *Top-Level Domain (TDL)* . There are lots of *TDLs* , .com, .net, .org, .mil, and each *TDL* has an organization that controls the *TDL* . The next part of the *URL* is the domain name, *somewebsite* . When

someone registers a domain name, the domain name is registered with the appropriate *TDL* controlling authority which which will in turn advertise DNS information about the domain. The last part of the *URL* is the resource locator *www* which is the resource locator for a web page.

The following commands will be covered in this section:

nslookup

dig

nslookup

The *nslookup* command can be used to query the DNS server for information about domains.

The basic use of the *nslookup* command is from the command line as in the example below.

The output lists the IP address of the DNS server. The line *Non-authoritative answer* means that the answer is not from the main DNS server responsible for the domain but rather another DNS server that knows the answer. The last part of the output is the name you were looking for and the IP address.

If you type the command *nslookup* without putting in a domain name you will enter interactive mode. In interactive mode you will have a > character that means *nslookup* is awaiting your input.

The *server* option in interactive mode will show you which DNS server you are getting name resolution from. You can also use the *server* option to change the DNS server to query. This will not change the server which is used for normal name resolution. In the example below the DNS server is changed to *10.8.4.3* .

The *set type* option allows you to change the type of DNS record to look for. By default, *nslookup* shows *a* records which are resource records. In the example below, the *type* is set to *ns* which is for nameservers or DNS servers. The example below shows four nameservers for the domain *somewebsite.com* .

In the example below, the *type* is set to *mx* which is for mail exchangers or email servers. The example below shows five email servers for the domain *somewebsite.com* .

In the example below, the type is set to *a* which is for resource records. The example below shows the *a* record for www.somewebsite.com.

In the example below, the type is set to *aaaa* which is for IPv6 resource records. The example below shows the *aaaa* record for www.somewebsite.com.

To exit *nslookup* interactive mode, type *exit* .

dig

The *dig* is another command that can be used to query the DNS for information about domains.

The basic use of the *dig* command is from the command line as in the example below. The output lists the IP address of *www.somewebsite.com* . The flag line *AUTHORITY: 0* means that the answer is non-authoritative.

The *dig* command by default looks for *a* record. The *dig* command can also be used to lookup other types of records by using the *-t* option followed by the type of record, *ns* , *mx* , *a* , or *aaaa* .

route

The *route* command is used to view the network routes that your system knows about. On most systems with only one NIC there is not much to do with routes. The most important route that your system has is a route to the *default gateway* which is the route to to Internet.

If your system has multiple NICs then then will be multiple routes. It is important to make sure that *default gateway* is pointed in the right direction. The syntax to add a default gate is as follows:

route add default gw *ip_address* dev *interface_name*

In the example below the *route* command is used to add *192.168.56.1* as the *default gateway* which can be reached via the *enp2s0* interface.

Chapter 14 Programming – A Logical Breakdown

In this chapter, we're going to be talking about a lot of the concepts which underlie programming. The fact is that though there are many ways to write programs and many different concepts which fall into it, many of the concepts remain rather fluid throughout every single implementation. What I mean by this is that regardless of how you're programming, there are many things which will be common for all programs you'll ever write.

I'm a firm believer in the conceptual: I think that if you're going to learn something, you need to know how it works from a conceptual standpoint and then tie that into rigid thoughts and structures at a later point. We're going to be connecting all these concepts, too! However, for right now, we just need to talk about them as that: *concepts* .

Programs and How They Run

So the first thing that we're going to talk about is programs, what they are, and how they run in accordance with Linux since this book is all about trying to learn to program on Linux specifically. Programs are a relatively simple concept in and of themselves, and they can go by many different names, too: the name "program" here can apply to anything from a script that you write in order to automate certain processes to a desktop application like Word or Google Chrome to a desktop video game like Grand Theft Auto V, as well as many different things which rest in between those.

Programs can run in the foreground, meaning that they are right in front of your face performing a specific function, or they can be a process in the background, running without you ever knowing it. An example of a program that runs as a process in the background would be something along the lines of the clock application which is constantly running on your computer. You never quite think about it, but this is a kernel-level program that is constantly running in the background, keeping track of what happens when. Indeed, a huge number of programs will run in such a way without the user either ever being aware of them, with the user being unbothered after initial execution protocol, or with them being so far in the background that the user must fetch them in order to use them.

As a new programmer, especially working with Linux programming, much of the programming that you're going to be doing is going to be taking

place within the console. This is especially important to you as a Linux user and programmer. Why? Because a lot of the appeal of Linux comes from the incredibly useful and in-depth command line tools which it offers.

Console, command line, and terminal all refer to the same thing; in Unix-like systems, the term most often used is Terminal. All of this refers to essentially the same thing: how one can manually enter commands and execute programs through plaintext rather than through more "modern" methods of program execution. Essentially, the command line offers the purest form of communication between a user and a computer; part of the reason that Unix systems are so widely copied and emulated is that they offered one of the historically best methods of interface between the said user and said computer. This remains true today in the Unix-modeled Linux systems.

Note that not all Linux distributions are equal, of course; some will rely on the Terminal. However, many do offer extremely high levels of possibility through the Terminal, even if their primary purpose is to push users away from this sort of low-level interface. For example, while Elementary OS (a high-level operating system based on an incredibly simple and elegant user experience) doesn't have anywhere near as much emphasis on the command line as Arch Linux does (a Linux distribution which is highly customizable and makes intense usage of the command line), they both offer a plethora of different features accessible only *through* the command line and the usage of tools thereof.

As I said, most programs that you're going to be writing early on as a Linux programmer will at the very least print output through the command line. However, there's a good chance that these programs will also be compiled and executed from the command line as well, if only because this is the "correct" way to do it which will make you the most intimate with your tool, the computer.

What Makes a Program?

When I say "make" here, I literally mean *make* ; how does one communicate with a computer and then create something that the computer can understand?

Well, firstly, they're called programming *languages* for a reason. Imagine it like this: you're fluent in English and French. If you meet somebody on the

street who is a native Spanish speaker, you won't really be able to communicate with them. However, if they speak Spanish and French, then after fumbling through some conversation, you'll both be able to communicate in French, even though your native tongues are different.

This is perhaps the best analogy for what programs are and how they work. Computers natively understand things in very simple and mathematical terms; they parse sequences of ones and zeroes in order to perform highly complex mathematical functions and in the end are just operating upon mathematical values. Humans, on the other hand, while they can speak mathematics, are generally not willing to put in the immense amount of time required to independently program everything about a computer in sequences of ones and zeroes. If every programmer were to do this, even a simple program would take years upon years to write.

Programming languages, therefore, offer a middle ground. While there are long genetic lineages that can form between various languages, all languages at their root serve the same purpose: dissecting instructions which are easily understandable by humans into terms which may be understood by computers.

There are two main ways of achieving this goal, then. The first is to directly translate from the language that the program is written in to the language which the computer natively speaks. This is referred to as *compiling* code. Languages which are based around compiling code to a computer's native language are called *compiled languages* .

The second is to offer a sandbox within which programs can run - a virtual machine or runtime environment in which programs can be executed. This runtime environment is directly separated from the operating system in that the program doesn't run on the actual *operating system* (though the program is certainly capable of performing functions on the operating system, such as writing and reading files), but rather within an abstracted environment from the operating system. This means that the code you write doesn't have to be translated into a language the *computer* can understand - it just must be understandable by the virtual environment which is running the code.

So what are the advantages of one or another? Well, each has their own perks and negatives. Compiled programs are generally going to have faster execution times because they're translated directly into machine code and

then run by the operating system directly. This means that there isn't anything extra for the compiled programs to load besides what is necessary for the program alone. Compiled programs also generally offer a great deal of direct interface with the computer that interpreted programs don't. For example, with a compiled program, since you're speaking directly to the computer, you often will oversee doing things such as allocating or releasing stores in memory.

Meanwhile, interpreted languages carry their own slew of benefits. Firstly, execution time generally doesn't matter too much on modern computing hardware because, well, computers are simply a lot faster than they used to be. Additionally, one of the advantages of compiled languages can also be an advantage of interpreted languages: the fact that compiled languages are so near and dear to the computer can make them a little bit harder to manage, deploy, and quickly prototype. However, interpreted languages offer a high-enough level of abstraction that a programmer can reasonably prototype things much faster. Perhaps the biggest appeal of interpreted languages is that you don't have to compile them in a different manner for every single computer; a program runs in an interpreter, period. This means that if an operating system has an interpreter for that language, programs written in that language will run on that operating system. One of the most popular interpreted languages, Java, actually gained a lot of mainstream appeal on these grounds: the idea of *write once, run anywhere* was extremely appealing at a time when there were numerous different operating systems vying for a place in the turbulent computer market, which was still young enough for there to not be an easy answer for software which needed to run on more than one system.

In this book, we're going to be covering both an interpreted and a compiled language, where C++ is the compiled language and Python is the interpreted language. You'll get an intimate feel for how the two different forms compare with one another. Don't be scared by this proposition, though; as I said earlier in this chapter, programming languages are always a lot alike. There are concepts which underpin every single programming language, which we'll be discussing in this chapter.

Program Flow - Procedural, Functional, Object-Oriented, and So On

This is one of the biggest separators between different programming languages; they generally have a paradigm which they prefer in terms of

how things are to be written and programmed.

Historically, functional or procedural languages were king. What this means is that all the code is written in such a way that it happens in a linear and procedural manner, with very few caveats for reuse. A program has clearly defined start and end points and a pretty narrow path for getting from one point to another. This made sense because computers were simpler and had less memory and hard-drive space; essentially, there was overall less room for the programmer to waste time or space. Everything was made with the purpose of being as simple and straightforward as possible for the computer to parse so that nothing was bigger or more consumptive in terms of power than it needed to be.

However, eventually, this paradigm began to change. Perhaps the most important languages in the transition were Ada and Smalltalk. These languages showed early on the power of object-oriented programming, which would later become the primary mechanism of action for most programs on current computing hardware. C++ would eventually come along as a development upon language C, specifically created to include the potential for object-oriented programming. Later would come Java, as well as the development of a huge number of different scripting languages intended to help programmers get prototypes off the ground quickly, like Python, Perl, and Ruby. All these languages have important concepts at their core. Many modern languages, such as C# and Go, are also using the object-oriented paradigm.

So what is object-oriented programming? Well, this book focuses heavily on it in later chapters, given that you're working with two languages which are stocked with different features to accommodate object-oriented programming. Essentially, object-oriented programming offers a far more modular, reusable, and understandable method of programming as compared to the functional or procedural programming paradigm. The object-oriented ethos differs largely in the fact that it enables the programmer to start breaking things down into major chunks. We'll talk more about this in just a second when we start talking about objects and classes. While this isn't going to come in handy for short Linux scripts using Bash, it's going to invariably be useful as you start working with actual applications in C++ and Python later in the book, so it's important to

cover all of the concepts right now so that when they actually start to be used in a certain way, you're familiar with the underlying ideas.

C and C-Style Languages

This is perhaps one of the most important things to cover, conceptually, in this chapter. To understand different programming languages is to understand an entire genesis. At one part of this genesis lies a language called C. The reason this is important is that C and Unix-based operating systems go together pretty much perfectly.

Allow me to take you back to 1972: Dennis Ritchie and Ken Thompson are developing a language which will act as a natural successor to a language called B, which itself is a natural successor to BCPL. This language is one of the most approachable at that point in time. It offers a lot of features that other languages aren't, it's straightforward enough that anybody can feel comfortable picking it up, and it makes the baseline difficulty of using and operating a computer a little bit lower.

This language, however, would soon become something much bigger. Brian Kernighan and Dennis Ritchie were employees of AT&T's Bell Labs - the same Bell Labs who would be the creator of Unix. Well, after the development of Unix, Ken Thompson decided to program a version of Unix that was written in C, rather than the very specific machine language which it was written in prior.

This meant, firstly, that Unix could now be used on any computer for which C code could be compiled. However, this worked reflexively in giving C code a stable and extremely popular example of how well it could work. This meant that the popularity of each fed off the other as C rapidly gained steam.

What would result is one of the most influential programming languages ever, C. The bulk of Unix programs would be written in C, much of early C support would rest in Unix, and when Unix-like operating systems started to be developed, they would normally follow Unix's lead and be developed in C or a combination of C and C++.

C would end up inspiring a whole host of different languages, like the C++. However, it's not just C++; many languages that you hear about rather often as a novice programmer owe a large part of their legacy to C. Java, Perl, PHP, Python, C#, JavaScript, Objective-C, Go, and Rust are just some of

the many languages which are derived largely from C and C standard procedures.

Now, note that every programming language seeks to solve a lot of the same problems, and they're generally based on algorithmic or mathematical assertion of a given problem - in the end, with the key focus on solving problems posed to computer programmers - and so in a lot of ways, they're relatively similar. But C style languages share many similarities, such as in their type systems (which we'll get to momentarily) or their syntax in general.

This chapter, in covering programming concepts, covers concepts which are central to C style programming languages, but which also apply to many other kinds of languages - though they may just appear in different ways.

Variables and Values

This is the first real programming concept that we're going to tackle in this chapter, and it's one of the most important. The entire purpose of a computer is, well, to compute. Computers are designed to manipulate values; this is their primary purpose, in fact! It makes sense, then, that one of the main functions of programming is to maintain, recall, and manipulate different values.

So what exactly is a value? We've talked before about how computers read things as 1s and 0s. In the end, this isn't *entirely* true, but it also is. Everything that happens on a computer is either a one-and-done calculation, or it's stored somewhere; it may be stored temporarily in the fast-access memory known as RAM, or it may be stored more long-term as solid data on the hard-drive. In either case, the point is that computers are based around storing and changing data, known as values.

Values can be much of anything, in terms of programming. Values can be, for example, anything from whole numbers to decimal numbers to long strings of characters to single characters even to true or false qualities.

Of course, sometimes you're going to need to keep track of a certain value. This is where variables come in handy. Variables offer a way to give a name and a consistent means of reference to a given value in the computer's active memory.

However, there’s still a bit more to it than just this. For example, when you’re working with different values, the computer is ultimately just reading them as vast simplifications of what they *appear* to you as the programmer. The computer, thus, needs a way to delineate between all these different forms, because the binary representation of *one* thing isn’t the binary representation of *another* , even though they may appear to have the same value.

One example of this is comparing strings in a language such as Java. If you compare strings directly, you’re going to be getting incorrect data because strings are actually arrays of ASCII values which they, themselves, hold a binary value. If you compare one value directly against another, then you’re going to run into a mathematical comparison rather than a string literal comparison where the ASCII values are compared against one another. However, if you were to compare the string objects *directly* using an object comparison method, as opposed to a mathematical value comparison, then you could feasibly find that the two strings are much the same.

These different kinds of values are then known as *types* . There are an innumerable amount of types, and different types exist in every programming language. However, most C-style languages have a few basic “guidelines” for the types which are supported by them.

First, you have primitive types, which are types that are natively supported and aren’t objects defined by APIs or other foundational software for a given language. These are things such as *ints* (integers), *floats* (floating point numbers), *chars* (single characters), and sometimes *booleans* (true or false values).

Then, you have object types, which as I said earlier, we’ll talk about more in-depth in a bit. Object types are constructed formations of data which serve as a collection of different traits represented by primitive data types. An incredibly popular and often represented object type is the *string* , which is a sequence of ASCII characters.

Some languages require you to explicitly say what sort of value a given variable will hold. These are called *explicitly typed* languages. Others, though, don’t require you to make this distinction, they just let you declare variables. These are called *implicitly typed* languages.

There is no clear preference between one or another, it really depends on what you're trying to achieve and how complex of a data structure you're working with. However, in the end, both achieve the same thing: a system of establishing names for given values to which everything else may correspond.

Objects and Classes

This is perhaps one of the most daunting topics in the book, but it goes together with the topic of values and variables, so we should likely get it out of the way now. We've talked about how object-oriented programming completely shifted the paradigm in terms of programming, but we haven't talked about why, nor what an object is.

Firstly, think of a car. A car has many different attributes which may make it up. Every car has a make and model; it has a certain number of doors; it has an engine of a certain size; it may have a convertible top and it may not; so on and so forth. These are all variables which may shift up any given definition of a car.

In programming most of the time, though, you are forced to only work with one given variable at a time. That is unless you set up a proper paradigm for working with more than one. Establishing a class allows you a way to group a bunch of different variables together.

That is, instead of making a bunch of different unique variables to refer to traits, you can just dump all the variables together. This means that you can apply all the same variables to different vehicles. Take this pseudocode for example. The traditional way to set up important information regarding a car would be like so:

car1make = "Ford"

car1model = "Taurus"

car1color = "gold"

car1doors = 4

car1year = 1998

car2make = "Pontiac"

car2model = "Grand Am"

car2color = "silver"

car2doors = 4

car2year = 2004

As you can see, this is very unwieldy. Additionally, you may have a hard time keeping up with all the different cars. Fortunately, there's a much better way to handle this: establish a class of common traits that you can then use in order to normalize the establishment of cars:

class car {

make

model

color

doors

year

}

Then from here, you can create *objects* . Objects are just instances of a given class.

car one = new car("Ford",'Taurus", "gold", 4, 1998)

car two = new car("Pontiac", "Grand Am", "silver", 4, 2004)

Then, you have all the cars' reference qualities stacked on top of one another, available for instant recall:

print one.make

This would access the object *one* of the class *car* and then print out the attribute *make* , meaning it would print out "Ford" to the console.

There are many other concepts which make up object-oriented programming. In fact, there are four central ideas which drive it: abstraction, encapsulation, inheritance, and polymorphism.

Abstraction is the idea that the programmer should be as far away from the menial tasks of programming as possible, and that ideas shouldn't require much boilerplate. Instead, programming should be a sequence of ideas taking place, and the programmer should be performing an action using the synthesis of these different ideas. Therefore, abstraction is the idea that programming really doesn't need to revolve around the code itself but

rather the natural flow of the code and the efficiency of the code in reaching its end goal.

Encapsulation is the idea that everything should be neatly bubbled together and that a program should be self-contained. Things should be able to be combined in order to form new ideas whenever such a thing is necessary, and the program itself should be an extension of those new ideas which are formed. Therein, object-oriented programming should also offer some level of extensibility - meaning that the concepts which are developed during object-oriented programming should be reusable either on the scale of a single program or on the scale of multiple different programs.

Inheritance is the idea that classes should be able to derive from one another in a natural family tree. A good way to think about this is the biological tree. The biological tree is organized according to common traits into different biological families at the macrobiological level. As these break down and become more and more specified, the same common traits will exist from a parent family designation to a child family designation, but the child designation will have more specific features about it. For example, both birds and mammals have vertebrae and are therefore classified to the phylum *chordata* . However, birds and mammals have many differences, as well, with mammals giving live birth and birds hatching eggs, for example. These specifications come into play in the child designations from parent designation *chordata* . However, because something can be further specified doesn’t mean that earlier specifications go *away* ; giving something an exacting specification doesn’t undo broader specifications, and classifying a bird beyond *chordata* doesn’t take away its vertebrae. They’re just ways of going beyond broader classifications into more exacting classifications.

For instance, we could have a class called *wheeledTransport* , where the only attribute was *numberOfWheels* . This is a very broad classification; cars, bikes, and skateboards could all fit under this classification. We can, therefore, extend the classification to include class definitions for all three with further definitions of what each is. However, we could still access the parent attribute *numberOfWheels* , as it would be inherited from the parent class *wheeledTransport* .

Polymorphism is the final primary definition of object-oriented programming. Polymorphism is the idea that something may have multiple

definitions with the ideal definition being chosen upon usage. This doesn't exactly make sense right now, but in object-oriented programming, you can do something called *argument overloading* , and this is a great example of polymorphism. Argument overloading is when you have two different function declarations which both take different arguments but have the same function name. For example:

```
bark() {
...
}
bark(numberOfTimes) {
...
}
```

While these both have the function name *bark*, the second takes the argument *numberOfTimes* , which could then be used in the function. The first takes no argument. Which one is used later in the code would depend on whether or not the user gave an argument to the function call or not. We'll be discussing functions and methods here in just a second, so no worries.

Understanding classes and objects is one of the primary cornerstones to being an able programmer not just on Linux but in general. This is one of the key points of programming in modern usage, and you need to have a proper understanding of it in order to do anything else.

Data Sets

Another foundational part of programming is understanding *data sets* . Data sets are pretty much exactly what they sound like. There are going to be many times in programming where you need to group data together. An immediate example would be some sort of running record, such as grades. The conventional way to do this would be like so:

```
student1Grade1 = 99
student1Grade2 = 94
student1Grade3 = 92
student2Grade1 = 81
```

student2Grade2 = 90

student2Grade3 = 89

student3Grade1 = 91

student3Grade2 = 93

…

As you can see, this is rather unwieldy, plus it would get complicated to keep up with all this information. Moreover, as we'll talk about in a second, when you need to assign a new value to something, you'll often find yourself writing inconveniently long statements.

So, how can one approach this? A much better way is to link the data together through establishing a data set. There are three primary different types of data sets: *arrays* , *vectors* , and *dictionaries* . *Vectors* , *lists* , and *sets* are pretty much three interchangeable terms with only minute differences between them; Python, for example, foregoes having arrays for the most part and instead uses lists, which are functionally like vectors.

Arrays have established initial sizes. This means that memory is allocated for every member of the list at the program's runtime. The array doesn't have to be immediately filled, but the space remains open for every element in the array to be filled; if the array is overfilled, it's called an array overflow, and these are pretty much concretely not permitted, and you'll run into a compile error (which is when your program refuses to compile or run due to an issue as such).

Vectors, lists, and sets don't have predetermined sizes. They scale automatically in accordance with the size of the data set. They will never be bigger nor smaller than they need to be. This can be both a good thing and a bad thing. Arrays are more memory effective because they have all of the data allocated ahead of time; however, if this isn't a concern for you - as it normally won't be on modern hardware - vectors, lists, and sets serve a far better way to access data within a data set.

Dictionaries are much like vectors, lists, and sets in that they scale automatically. However, dictionaries set up an essential relationship between data pieces, identifying one piece of data as a *key* which links to another *value* . Consider it like a real dictionary; if the definition is the *value* , then you can find the *value* of a given word by using the word *itself*

as the *key* . This means that you can search by the *key* for the given value that you're needing. This is the essential idea lying beneath dictionaries.

Arithmetic and Assignment Operations

Control Flow (Conditionals)

This is one of the first and foremost parts of programming that you need to understand as a new programmer. The truth is that a lot of the programs that you make will be useless without some way to determine how to make decisions. Decision making is the single largest crux of almost any program. Even being able to test basic conditions for an otherwise straightforward program - like whether a script was able to connect to a download server, for instance - is essential to having a well-running and bug-free program.

We talked a little bit about how things can be true or false in programming. This is where that information really starts to matter. Being able to tell whether something is true or false can have a massive impact on the running of a program.

But what does true or false even mean in computing terms? Computers can't understand this simple relation. Computers don't have logical and entropic brains like we do. This means that they can't understand in a logical sense whether something is true or false. However, they also *can* . How is this?

Computers don't view things in terms of abstract thought; rather, they view them as a sequence of things which are on and then off, yes or no, 0 or 1. It's these binary processes that allow a computer to render an understanding of logical processes. These are represented in lower level engineering as things like AND, OR, XAND, and XOR gates, but at this level, we just call these boolean processes.

Being able to ascertain whether something is true or not and then decide is essential to any good program. Computers have one very simple way of doing this: conditional statements.

Conditional statements are essentially comparisons of one value to another (or perhaps multiple) in order to determine the veracity of a given statement.

Let’s say that I was to say “7 is less than 3”. Well, you would instantly object - no, it’s not! And you’d be right to do so because I’d have sounded like an idiot. So where does one go from here? What happens to let you say “no, that’s not right”?

Subconsciously, when I say something like that, you take the two numbers that I said - 7 and 3 - and then you compare them to one another. You realize that 7 is bigger than 3, and you perhaps even visualize them on a subconscious sort of number line. Without realizing it, you compare the two values in much the same way that a computer would.

However, the computer doesn’t have the intuitive knowledge to compare these sorts of numbers. Intuition is something uniquely human, and many people posit that intuition can’t be programmed because it takes a combination of randomly generated thought alongside an extremely extensive knowledge of past events. Because it lacks this sense of intuition, you need to tell it explicitly what it needs to compare. It’ll then tell you whether the statement is true or not.

No matter what you’re compared, whether it’s 3 to 7, or one string to another, you’re always doing so for a result: the computer telling you whether the comparison is true or false. In this way, the comparison can stand as a value on its own. This is one of the densest and most difficult concepts to understand. Comparisons can hold value in and of themselves, because they can be either true or false, with false being 0 and true being 1.

This means that the statement *3 is less than 7* is true. But this also means that the evaluation of this statement as true can be considered a value itself - a value of *true* in a true/false dichotomy. This also means that since boolean variables can hold true or false information, they can also hold the truth or falsehood of a comparison:

myComparison = 3 < 7

The above variable would be equal to “true” or 1, depending upon what language you’re using.

The key point here, though, isn’t that this value can be stored - that’s only a piece of information that will come in handy for you later. The key point is that value of truth or falsehood may be assessed through (potentially a series of) comparisons of values. Things can be assessed as true or false by comparing two different things. Things can also be assessed as true or false

by their very nature - for instance, if you had a boolean variable called isRunning, and you set it to true every time that you started the program, and set it to false when you wanted to end, you could check to see if isRunning was true by comparing the veracity of isRunning to the veracity of “truth”.

In other words, things can be true or false. You can use this important piece of information in order to make program-wide decisions that will change the course of the program. How you do this is the part that’s cool: if statements.

If statements are one of the important foundations of programming. It allows you to evaluate something and then see whether the evaluation is true. If it’s true, it lets you do something. If it’s not true, it skips the code within entirely. This is something you’re going to see in both languages that we work within this book, so it’s important to understand it.

The basic syntax for an if statement is like so:

```
if (condition) {

code

}
```

This applies across several languages. C, C++, Java, and PHP all immediately spring to mind. You can build upon if statements by using what’s called an else statement. This enforces an interesting duality where there are two different kinds of conditionals. I like to call these *passive* and *active* conditionals.

Passive conditionals are conditions which harbor only on an if statement. All that happens is the code within the if statement is evaluated. If it turns out to be true, then the stuff within the if statement will be executed. If it doesn’t turn out to be true, however, the stuff within the if statement will be skipped over and it’s like the conditional didn’t even exist. There is no force clause which makes certain that you’ll be obeying a certain paradigm. It’s just simple on and off - if the condition doesn’t match, the entire statement is ignored.

Else statements, however, give you *active conditionals* . Active conditionals are conditions which *do* have a force clause. What happens is the first if statement is evaluated. If it turns out to be true, then the if statement will be

executed. However, if it's not true, then the else statement will be executed - no matter *what* . There is no passivity here; if the if statement doesn't execute, the other branch of the conditional will always be carried out. The chunk of code will *not* be skipped over as it would otherwise.

This means that you basically have two different ways in which you can assess situational conditions: you can either give them a treatment wherein they may or may not be activated, or you can set up a situation in which some code will *certainly* be activated.

There's one more element of nuance to this whole equation, though. Let's say, for example, that you wanted to add in a condition for more than one possible outcome. That's to say that you wanted to check more than just one condition. What would you do in this situation? With things as they stand, you don't really have any sort of options in this scenario. Thankfully, however, you aren't restricted to having no options. There's one more kind of conditional: the *else if* statement. This goes between your if and else statements. What happens is that the first if statement is evaluated; if it's not true, the condition of the *else if* statement is evaluated; then, if *that* isn't true, it finally proceeds to the else statement. You can stack as many else if statements as you'd like.

Control Flow (Loops)

The other major aspect of control flow occurs in *loops* . So what are loops? Loops are basically a way of reiterating over a set of code a certain number of times until a certain end is achieved. You never really think about it, but you use loop logic all the time.

Consider counting out loud from one to ten. This may not seem like a loop to you at all; after all, you're just saying different numbers. But, you're undergoing loop logic every time you count!

Think of it this way. If you count from one to ten out loud, then what you're doing is thinking of a number *n* , then you're using your mouth to say number *n* , then you're increasing *n* by 1 in your head. This happens over and over until you reach the number that you're counting to.

Loops in programming occur along similar logic. There are two primary types of loops, but they function differently across C++ and Python, so we're just going to discuss those in-depth in their specific chapters. In the meantime, just focus on the idea that loops are a huge part of control flow.

There are many times in programming where you're going to need to just repeat something over and over. Loops provide the perfect opportunity to fulfill that need.

Functions and Methods

Functions are the last major part of programming logic that we're going to be discussing at this point in the book because it's the last major pillar of early conceptual programming that we need to tackle. Don't get me wrong, there are many more much deeper topics that we could go over in a conceptual way, but this is where the scope of the book starts to be a bit more defined.

You might remember working with functions in your high school or college math classes. Functions were always math operations where you took a certain variable then performed a mathematical operation on it in order to get another value. For example, *f(x)* involved taking *x* and then performing some mathematical operation in order to get *f(x)* , which was functionally a *y* value.

Functions and methods, for the record, are functionally the same thing. The preferred term changes depending upon the language which is being used. In older languages or procedural languages like C, as well as era derivatives like C++, function is the preferred term. Meanwhile, in object-oriented languages like Java or Python, method is the preferred term.

Anyhow, methods and functions simply take this idea of mathematical operations upon a given value and then gives you a way by which you can fulfill that specific need. For example, you could create a function called *calculateSquareFootage* which took the arguments *length* and *width* and then return a value called *area* , which would be the overall value of the function *calculateSquareFootage* . *area* could be the value of *length* multiplied by *width.*

Defining a function like this can be referred to as a *function definition* . This is an important term to know, so keep it in mind. Anyhow, moving forward. Once you've defined a function, you'll probably want to use it later. This is known as *calling a function* or *calling a method* .

You call a function by saying the name of it plus the values you want to send. For example, calling *calculateSquareFootage(3, 4)* would return the

value of 12. We could then save this value to a variable or print it out directly.

That’s the last important thing about functions: return values! Almost all functions will give a value back to you, that’s one of their primary purposes. Not all of them must, of course - void functions have no such requirement to give a value back. Every other function, though, must give a value back which corresponds to the type of the function, if using a language where you must type a function when declaring it. The returning of a value signifies the end of a function.

Again, not all functions must return values - void functions do not have to do so. Additionally, many scripting languages - such as Python - avoid function typing at all, meaning that when the function has no more code in it, the function has run its course and will be terminated, going back to the main code of the program.

Basically, functions have two primary purposes.

The first primary purpose is that they allow you to normalize certain interactions and calculations within your program, making it so that you don’t have to type the same thing over and over. This in and of itself is somewhat modular, which is a very nice feature.

The second primary purpose is that they allow you to clean up your code. By dividing a huge main code block into smaller functions, you make your code much more readable and, much of the time, you also make it easier to understand.

Classes can have their own defined functions, and they can also have multiple functions with the same name but different arguments (following the rules of polymorphism). This also lends itself to something else cool. When you’re using an object which is ultimately derived from a created class, like a vector or a string, they’ll generally have built-in methods which you can use to manipulate the object. This really comes in handy!

Conclusion

Over the course of this chapter, we've broken down some of the toughest programming concepts and tried to make them as easy to digest as is even possible. Understanding programming intimately can be a very hard thing. However, it's important if you want to be a programmer that you understand the underlying concepts. The cool thing about these concepts you've learned is that they apply not just to Linux systems but to any code you'll ever write. The fact is that there's not a whole lot special about programming on Linux. There isn't much that will divide code written in C++ for Windows and C++ for Linux, and there's *nothing* to divide code written in Python, Java, or PHP between the two platforms. However, there are key concepts which permeate all these extremely popular languages that in order to understand how to program, you must understand them.

Now that we have learnt what Linux system is and how to get started on the Linux manipulations by learning several basic commands. These commands help us to navigate the system and, thus, I would recommend that you perform as many practice tasks as you can so that you understand what commands does what and the output of running each of those commands. In our next sessions, we will learn more on Grep and regular expressions, piping and redirection, process management and finally bash scripting which will form part of the intermediate level of Linux system. This will prepare us to be comfortable enough to proceed on to the advanced level of Linux system and how to use the commands that we have learnt in writing good GNU/Linux software, learning about threads, processes as well as inter-process communication among others.